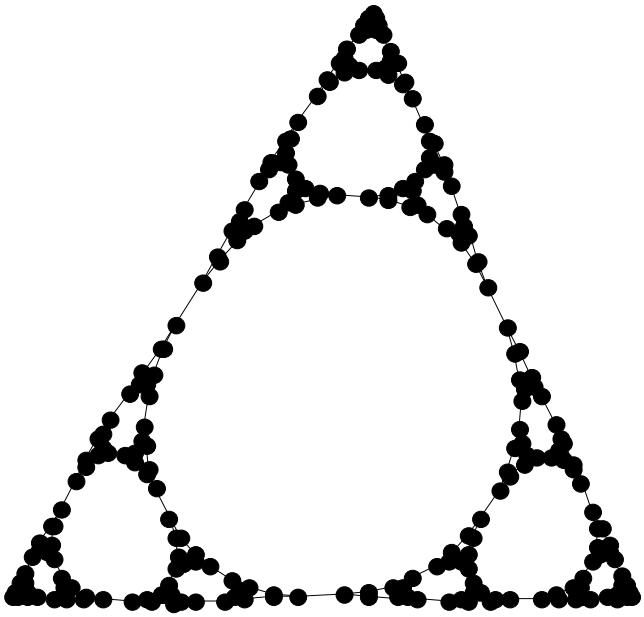
МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
БУРЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Т.В. Бурзалова

# Приемы решения задач по дискретной математике с использованием компьютерной системы «Mathematica»



Улан-Удэ
Издательство Бурятского госуниверситета

УДК 681.323:51
ББК 22.1с + 32ю973ю26 – 018.2
Л68

Утверждено к печати
Редакционно-издательским советом
Бурятского государственного университета

Р е ц е н з е н т ы
А.И. Артюнин
доктор технических наук, профессор
М.Н. Очиров
доктор педагогических наук, профессор

**Бурзалова Т.В.**

Учебно- методический комплекс по решению задач дискретной математики с использованием компьютерной системы «Mathematica». –Улан-Удэ: Издательство Бурятского госуниверситета, 2002.- 300 с.

Компьютерная система «Mathematica» широко используется в образовательных и исследовательских учреждениях многих стран мира как современный инструмент автоматизации математических вычислений. Применение системы помогает учащимся лучше осмыслить теоретические и прикладные вопросы дискретной математики. Прекрасная графика обеспечивает возможность визуализации вычислений, поддерживает оценку влияния различных параметров, представление результатов анализа в современном виде.

Для студентов и преподавателей вузов, учащихся и учителей школ, а также для круга читателей, имеющих дело с математикой в своей работе и учебе и владеющих компьютером на уровне пользователя.

Burzalova T.V.
Solving of problems of Discrete Mathematics with «Mathematica». –Ulan-Ude: Buryat State University Publishing House, 2006.- 300 p.

The computer algebra system «Mathematica» is the most comprehensive software available for educational and research of discrete mathematics, particularly combinatorics and graph theory.
The book is recommended for specialists in

The computer system "Mathematica" is broadly used in educational and exploratory institutions of the many countries of the world as modern instrument to automations of the mathematical calculations. Using the system helps учащимся better comprehend theoretical and applied questions discrete mathematics. The beautiful graphics provides the possibility to visualizations of the calculations, supports the estimation of the influence different parameter, presentation result analysis in modern type.

For students and teachers high school, and teachers of the schools, as well as for readership, dealling with mathematics in its functioning and training and having computer at a rate of user.

# Введение

Без преувеличения можно утверждать, что компьютер в образовании еще сыграет такую же роль в обучении будущих специалистов, какую сыграло в свое время применение книгопечатания в образовании. С приходом в наш мир компьютеров открываются новые методические возможности, которые нельзя заменить каким-либо иным средством и которые должны быть приоритетными в разрабатываемых образцах новых информационных технологий. В последнее время большую популярность в мире приобрели различные системы компьютерной математики, среди которых выделяется система “Mathematica”. Она создана фирмой Wolfram Research, Inc. во главе с ее президентом и главным разработчиком программ профессором Стивеном Вольфрамом и является несомненным мировым лидером среди программ символьной математики для персональных компьютеров. “Mathematica”чрезвычайно популярна за рубежом, число только легальных пользователей системы к настоящему времени уже превысило 1 миллион. Она используется более чем в 50 ведущих университетах мира, в отделениях Госдепартамента США, во многих научных центрах и других учреждениях и организациях. Помимо научных работников, инженеров и педагогов она получила признание и у специалистов художественного и гуманитарного профиля. Система “Mathematica” обладает богатейшей библиотекой встроенных функций, а графические возможности этой системы просто завораживают. Кроме того, “Mathematica”– типичная система программирования с проблемно-ориентированным языком программирования современного сверхвысокого уровня, позволяющая сочетать традиционный процедурный стиль программирования с более емкими и естественными функциональным стилем и стилем правил преобразований. Система интерактивная (то есть работает в режиме постоянного диалога с пользователем), она гибка и универсальна, поэтому может быть использована всеми желающими, как школьниками, так и профессиональными математиками и другими специалистами, практикующими математические методы в своей работе. Для решения математических задач система содержит готовые функции почти для любого специалиста-математика. Однако с помощью пакетов расширения можно использовать возможности системы под запросы любого ее пользователя. И здесь открывается необъятный простор для творчески мыслящих педагогов. Отметим, что, родившись как программа для профессионалов, система “Mathematica” в последние годы упорно позиционируется фирмой Wolfram как система , перспективная не только для высшего, но и для школьного образования. Работать с системой “Mathematica” просто, приятно и поучительно. Благодаря этому освоение системы “Mathematica” воспринимается учащимися с большим интересом, что служит побудительным мотивом к её внедрению в систему образования, причем не только высшего, но и среднего, и даже начального (последнему фирма Wolfram в настоящее время уделяет большое внимание).

В компьютерной системе Mathematica вызывает особенный интерес стандартный пакет расширения DiscreteMath`Combinatorica.

Объяснять алгоритмы дискретной математики, имея только мел, тряпку и доску, чрезвычайно сложно, и неоценимую помощь преподавателю здесь окажет компьютер и стандартный пакет расширения компьютерной системы “Mathematica” – DiscreteMath`Combinatorica. Владение методами дискретной математики является в настоящее время необходимой составной частью образования специалистов в области прикладной математики, экономики, статистической и теоретической физики, теории информации, социологии, математической лингвистики и т.д.

Стандартный пакет расширения <<DiscreteMath`Combinatorica` обладает богатейшим набором встроенных функций теории графов и комбинаторики. Он содержит функции для конструирования графов и других комбинаторных объектов, вычисляет их инварианты, и, наконец, позволяет создать их графическую визуализацию.

Приведем список этих функций:

l={AcyclicQ, AddEdge, AddEdges, AddVertex, AddVertices,Algorithm, AllPairsShortestPath, AlternatingGroup, AlternatingGroupIndex, AlternatingPaths, AnimateGraph, AntiSymmetricQ, Approximate, ApproximateVertexCover, ArticulationVertices, Automorphisms, Backtrack, BiconnectedComponents, BiconnectedQ, BinarySearch, BinarySubsets, BellB, BellmanFord, BipartiteMatching, BipartiteMatchingAndCover, BipartiteQ, BooleanAlgebra, Box,

BreadthFirstTraversal, Brelaz, BrelazColoring, Bridges, ButterflyGraph, ToCanonicalSetPartition, CageGraph, CartesianProduct, Center, ChangeEdges, ChangeVertices, ChromaticNumber, ChromaticPolynomial, ChvatalGraph, CirculantGraph, CircularEmbedding, CircularVertices, CliqueQ, CoarserSetPartitionQ, CodeToLabeledTree, Cofactor, CompleteBin aryTree, CompleteKaryTree, CompleteKPartiteGraph, CompleteGraph, CompleteQ, Compositions, ConnectedComponents, ConnectedQ, ConstructTableau, Contract, CostOfPath, CoxeterGraph, CubeConnectedCycle, CubicalGraph, Cut, Cycle, Cycles,CycleIndex, CycleStructure, Cyclic, CyclicGroup, CyclicGroupIndex, DeBruijnGraph, DeBruijnSequence, Degrees, DegreesOf2Neighborhood, DegreeSequence, DeleteCycle, DeleteEdge, DeleteEdges, DeleteFromTableau, DeleteVertex, DeleteVertices, DepthFirstTraversal, DerangementQ, Derangements, Diameter, Dihedral, DihedralGroup, DihedralGroupIndex, Dijkstra, DilateVertices, Directed, Distances, DistinctPermutations, Distribution, DodecahedralGraph, DominatingIntegerPartitionQ, DominationLattice, DurfeeSquare,Eccentriity,Edge,EdgeChromaticNumber,EdgColor,EdgeColoring,EdgeConnectivity,EdgeDirection, EdgeLabel, EdgeLabelColor, EdgeLabelPosition, Edges, EdgeStyle, EdgeWeight, Element, EmptyGraph, EmptyQ, EncroachingListSet, EquivalenceClasses, EquivalenceRelationQ, Equivalences, Euclidean, Eulerian, EulerianCycle, EulerianQ, ExactRandomGraph, ExpandGraph, ExtractCycles, FerrersDiagram, FindCycle, FindSet, FiniteGraphs, FirstLexicorpaϕicTableau, FolkmanGraph, FranklinGraph, FruchtGraph, FromAdjacencyLists, FromAdjacencyMatrix, FromCycles, FromInversionVector, FromOrderedPairs, FromUnorderedPairs, FunctionalGraph, GeneralizedPetersenGraph, GetEdgeLabels, GetEdgeWeights,GetVertexLabels,GetVertexWeights,Girth,GraphCenter,GraphComplement, GraphDifference,GraphicQ,GraphIntersection, GraphJoin, GraphOptions, GraphPolynomial, GraphPower, GraphProduct, GraphSum, GraphUnion, GrayCode, GrayCodeSubsets, GrayCodeKSubsets, GrayGraph, Greedy, GreedyVertexCover, GridGraph, GrotztschGraph,HamiltonianCycle, HamiltonianPath, HamiltonianQ, Harary, HasseDiagram, Heapify, HeapSort,HeawoodGraph, HerschelGraph, HideCycles, HighlightedEdgeColors, HighlightedEdgeStyle,HighlightedVertexColors, HighlightedVertexStyle, Highlight, Hypercube, IcosahedralGraph, IdenticalQ, IdentityPermutation, IncidenceMatrix, InDegree, IndependentSetQ, Index,InduceSubGraph, InitializeUnionFind, InsertIntoTableau, IntervalGraph, Invariants, InversePermutation, InversionPoset, Inversions, InvolutionQ, Involutions, IsomorphicQ, Isomorphism, IsomorphismQ, Josephus, KnightsTourGraph, KSetPartitions, KSubsetGroup, KSubsetGroupIndex, KSubsets, LNorm, LabeledTreeToCode, Large, LastLexicorpaϕicTableau, LexicorpaϕicPermutations, LexicorpaϕicSubsets, LeviGraph, LineGraph, ListGraphs, ListNecklaces, LongestIncreasingSubsequence, LoopPosition, LowerLeft, LowerRight, M, MakeDirected, MakeGraph, MakeSimple, MakeUndirected, MaximalMatching, MaximumAntichain, MaximumClique, MaximumIndependentSet, MaximumSpanningTree, McGeeGraph, MeredithGraph, MinimumChainPartition, MinimumChangePermutations, MinimumSpanningTree, MinimumVertexColoring, MinimumVertexCover, MultipleEdgesQ, MultiplicationTable, MycielskiGraph, NecklacePolynomial, Neighborhood, NetworkFlow, NetworkFlowEdges, NextBinarySubset, NextComposition, NextGrayCodeSubset, NextKSubset,NextLexicorpaϕicSubset, NextPartition, NextPermutation, NextSubset, NextTableau, NoMultipleEdges, NonLineGraphs, NoPerfectMatchingGraph, Normal, NormalDashed, NormalizeVertices, NoSelfLoops, NthPair, NthPermutation, NthSubset, NumberOfCompositions, NumberOfDerangements, NumberOfDirectedGraphs, NumberOfGraphs, NumberOfInvolutions, NumberOf2Paths, NumberOfKPaths, NumberOfNecklaces, NumberOfPartitions, NumberOfPermutationsByCycles, NumberOfPermutationsByInversions, NumberOfPermutationsByType, NumberOfSpanningTrees, NumberOfTableaux, OctahedralGraph, OddGraph, One, Optimum, OrbitInventory, OrbitRepresentatives, Orbits, Ordered, OrientGraph, OutDegree, PairGroup, PairGroupIndex, Parent, ParentsToPaths, PartialOrderQ, PartitionLattice, PartitionQ, Partitions, Path, PerfectQ, PermutationGraph, PermutationGroupQ, PermutationQ, PermutationToTableaux, PermutationType, PermutationWithCycle, Permute, PermuteSubgraph, PetersenGraph, PlanarQ, PlotRange, Polya, PseudorpaϕQ, RadialEmbedding, Radius, RandomComposition, RandomGraph, RandomHeap, RandomInteger, RandomKSetPartition, RandomKSubset, RandomPartition, RandomPermutation, RandomRGF, RandomSetPartition, RandomSubset, RandomTableau, RandomTree, RandomVertices,

RankBinarySubset, RankedEmbedding, RankGraph, RankGrayCodeSubset, RankKSetPartition, RankKSubset, RankPermutation, RankRGF, RankSetPartition, RankSubset, ReadGraph, RealizeDegreeSequence, ReflexiveQ, RegularGraph, RegularQ, RemoveMultipleEdges, RemoveSelfLoops, ResidualFlowGraph, RevealCycles, RGFQ, RGFs, RGFToSetPartition, RobertsonGraph, RootedEmbedding, RotateVertices, Runs, SamenessRelation, SelectionSort, SelfComplementaryQ, SelfLoopsQ, SetEdgeWeights, SetGraphOptions, SetPartitions, SetPartitionListViaRGF, SetPartitionQ, SetPartitionToRGF, SetEdgeLabels, SetVertexLabels, SetVertexWeights, ShakeGraph, ShortestPath, ShortestPathSpanningTree, ShowLabeledGraph, ShowGraph, ShowGraphArray, ShuffleExchangeGraph, SignaturePermutation, Simple, SimpleQ, Small, SmallestCyclicGroupGraph, Spectrum, SpringEmbedding, StableMarriage, Star, StirlingFirst, StirlingSecond, Strings, Strong, StronglyConnectedComponents, Subsets, SymmetricGroup, SymmetricGroupIndex, SymmetricQ, TableauClasses, TableauQ, Tableaux, TableauxToPermutation, TetrahedralGraph, Thick, ThickDashed, Thin, ThinDashed, ThomassenGraph, ToAdjacencyLists, ToAdjacencyMatrix, ToCycles, ToInversionVector, ToOrderedPairs, TopologicalSort, ToUnorderedPairs, TransitiveClosure, TransitiveQ, TransitiveReduction, TranslateVertices, TransposePartition, TransposeTableau, TravelingSalesmanBounds, TravelingSalesman, TreeIsomorphismQ, TreeQ, TreeToCertеслиicate, TriangleInequalityQ, Turan, TutteGraph, TwoColoring, Type, Undirected, UndirectedQ, UnionSet, Uniquely3ColorableGraph, UnitransitiveGraph, UnrankBinarySubset, UnrankGrayCodeSubset, UnrankKSetPartition, UnrankKSubset, UnrankPermutation, UnrankRGF, UnrankSetPartition, UnrankSubset, UnweightedQ, UpperLeft, UpperRight, V, VertexColor, VertexColoring, VertexConnectivity, VertexConnectivityGraph, VertexCover, VertexCoverQ, VertexLabel, VertexLabelColor, VertexNumber, VertexNumberColor, VertexStyle, VertexWeight, Vertices, WaltherGraph, Weak, WeaklyConnectedComponents, WeightingFunction, WeightRange, Wheel, WriteGraph, Zoom}.
Длина этого списка:

```
In[2]:= Length[l]

Out[2]= 464
```

Приведенный список показывает, какое огромное количество специальных встроенных средств предлагает пакет DiscreteMath`Combinatorica. Для того чтобы правильно и эффективно их применять, необходимо приложить определенные усилия, чтобы использовать при решении сложных задач дискретной математики. И здесь большую помощь призван оказать данный учебно-методический комплекс. Применение пакета при изучении дискретной математики избавляет от массы рутинных вычислений, которых очень много в алгоритмах дискретной математики (например, в задачах прямого перебора) и высвобождает время для обдумывания алгоритмов решения задач, более обоснованной постановки их решения, многовариантного подхода и представления результатов в наиболее наглядной форме. Кроме того, этот пакет позволяет получать многие промежуточные вычисления. Именно в дискретной математике решение задач, даже элементарных, довольно трудоемко и требует зачастую сотни однообразных вычислений, выполнять которые крайне тягостно даже при применении калькуляторов. “Mathematica” делает это за считанные секунды, а то и за доли секунды. Высвободившееся время можно использовать для более глубокого изучения математической сущности решаемых задач и их решения различными методами. Таким образом, применение пакета DiscreteMath`Combinatorica не только не лишает учащихся серьезных математических навыков, но, напротив, помогает значительно их расширить и углубить. Особенно важен методический аспект применения этого пакета при обучении дискретной математики. Пакет позволяет анимировать, показывать в динамике работу эффективных алгоритмов дискретной математики, что как показывает опыт значительно облегчает понимание достаточно сложных алгоритмов.

В предлагаемом пособии приводится описание функций пакета, разобраны типовые примеры и приведены примеры решения некоторых задач дискретной математики с использованием пакета DiscreteMath`Combinatorica. Основное внимание в книге уделено методике решения задач дискретной математики с применением средств компьютерной системы “Mathematica”.

В первой главе даются необходимые сведения из системы “Mathematica”. Вторая, третья и четвертая главы посвящены методике использования пакета DiscreteMath`Combinatorica для изучения перестановок, разбиений, композиций, разбиений подмножеств, табло Янга. В пятой и

шестой главах изучаются основные понятия теории графов, а шестая глава посвящена алгоритмам на графах. В восьмой главе дается методика решения задач в пакете DiscreteMath`Combinatorica.

Изучение дискретной математики с помощью DiscreteMath`Combinatorica в вузе так же необходимо, как изучение, например, текстового редактора Word. Наиболее удобной формой для этого являются спецкурсы, хотя можно предусмотреть и обязательный курс в новых учебных программах. Примерное планирование материала такого спецкурса приведено ниже. Для его проведения необходимо иметь класс, оснащенный современными персональными компьютерами. Минимальные требования к компьютерам:

- IBM/Intel-совместимый компьютер с процессором класса не ниже 80386;
- операционная система Windows 2000 или XP;
- объем ОЗУ не менее 16 Мбайт;
- устройство чтения CD-ROM;
- видеосистема класса SVGA;
- звуковая карта класса Sound Blaster для работы со звуком;
- все файлы системы “Mathematica” занимают 156 Мбайт. Из них лишь 40Мбайт приходится на минимальный вариант установки – файлы ядра (Kernel), интерфейсного процессора (Front end), библиотеки (Mathlink Libraries) и шрифты математических символов (Fonts). Дополнительно могут устанавливаться стандартные пакеты расширения (Standard Add-on Packages) и необходимые для связи с ними файлы инструментария (MathLink Developer’s Kit) – 12 Мбайт. Намного больше занимает онлайновая документация справочной системы – The Mathematica Book, Reference Guide, Standart Add Package и Additional Documentation. Для ее установки нужно еще 104 Мбайт.

Большую помощь в усвоении образного графического материала может оказать применение таких инструментальных средств как интерактивная доска, мультивидеопроекторы.

Автор благодарит профессора О.В. Мантурова за оказанную методическую и научную помощь в подготовке этого издания, а также профессоров М.Н. Очирова и А.И.Артюнина, взявших на себя нелегкий труд по рецензированию книги.

**Цели и задачи спецкурса «Решение задач дискретной математики в системе “Mathematica”» и его место в учебном процессе**

1.1 **Цель преподавания**.

Цель спецкурса – формирование прочной теоретической и вычислительной базы, необходимой будущему специалисту в его профессиональной деятельности.

1.2 Задачи изучения дисциплины.

1. Закрепление знаний основных разделов дискретной математики с применением функций пакета DiscreteMath`Combinatorica .
2. Формирование представления об основных дискретных структурах и функциональных системах.
3. Овладение основными методами дискретной математики с использованием пакета DiscreteMath`Combinatorica .
4. Изучение базовых разделов дискретной математики, связанных с программированием.

## Примерное планирование учебного материала

**(в одном семестре 3 часа лабораторных занятий в неделю, во втором 2 часа )**

| Номера уроков | Содержание учебного материала |
|---|---|
| | **Перестановки (9ч)** |
| 1 – 3 | Порождение перестановок. Лексикографически упорядоченные перестановки. Ранжирование перестановок. Случайные перестановки. Перестановки, перечисленные в минимально измененном порядке. |
| 4-5 | Инверсии и вектор инверсий. Подсчет инверсий. Индекс перестановки. Пробеги перестановок. |
| 6-7 | Циклы перестановок. Четные и нечетные перестановки, сигнатура перестановки. Подсчет циклов перестановок. |
| 8 | Инволюции. Перестановки без неподвижных точек. |
| 9 | Самостоятельная работа №1 |
| | **Комбинации (6ч.)** |
| 10 | Бинарное представление подмножеств. |
| 11-12 | Коды Грэя. Лексикографически упорядоченные подмножества. |
| 13-14 | Порождение k- подмножеств. Строки. |
| 15 | Самостоятельная работа №2. |
| | **Теория Пойа (11ч.)** |
| 16-17 | Группы перестановок. Действие групп. Классы эквивалентности и орбиты. |

| | |
|---|---|
| 18-20 | Цикловой индекс групп перестановок. |
| 21-24 | Применение теоремы Пойа. |
| 25-26 | Самостоятельная работа №3 |
| | **Перестановки, композиции и табло Янга (5ч.)** |
| 27-28 | Разбиение целых чисел. Порождение разбиений целых чисел. Диаграммы Феррерса. Случайные разбиения. |
| 29-30 | Порождение композиций. Случайные композиции. |
| 31 | Самостоятельная работа №4. |
| | **Разбиения множеств (5ч.)** |
| 32-33 | Порождение разбиений множеств. Числа Стирлинга и число Белла. Случайные разбиения множеств. Разбиения множеств и ограниченные возрастающие функции. |
| 34-35 | Порождение табло Янга данной формы. Случайные табло Янга. Операции удаления и вставки. Перестановки и пары табло. |
| 36 | Самостоятельная работа №5. |
| | **Конструирование графов (4ч.)** |
| 37 | Матрицы смежности. Матрицы инцидентности. Списки смежности. Списки ребер. |
| 38-39 | Изображение графов. Функция ShowGraph. Опции вершин и ребер. Наследование опций. Выделение элементов графа и анимация. |
| 40 | Самостоятельная работа №6. |
| | **Основные вложения графов (4ч.)** |
| 41-42 | Циркулярное вложение. Ранжированное вложение. Радиальное вложение. Корневое вложение. |
| 43-44 | Улучшение вложений. Параллельный перенос, сжатие и растяжение изображения графов. Вращение изображений графов Функции ShakeGraph и SpringEmbedding. |
| 45 | Модификация графов: добавление и удаления вершин и ребер. Изменения графов. Установка опций. |
| 46 | Самостоятельная работа №7. |

| | |
|---|---|
| | **Операции над графами (4ч.)** |
| 47-50 | Стягивание вершин. Порождение и перестановка подграфов. Операция объединения и пересечения. Сумма и разность графов. Операция соединения. Произведение графов. Реберный граф. |
| 51 | Самостоятельная работа №8 |
| | **Специальные виды графов** |
| 52 | Полные графы. Графы – циркулянты. Полные k-дольные графы. Циклы, звезды, колеса и решетчатые графы, граф бабочка. |
| 53-54 | Деревья. Помеченные деревья. Полные деревья. |
| 55-56 | Случайные графы. Конструирование случайных графов. Реализация последовательности степеней. |
| 57-58 | Графы отношений. Функциональные графы. |
| 59 | Самостоятельная работа №9 |
| | **Свойства графов** |
| 60-61 | Обходы графа. Поиск в ширину. Поиск в глубину. |
| 62-63 | Связность. Связные компоненты. Сильная и слабая связность. Ориентированные графы. Двусвязные компоненты. Связность. Графы Харари. |
| 64-65 | Циклы в графах. Ациклические графы. Обхват графа. Эйлеровы циклы. Гамильтоновы циклы и пути. Задача коммивояжера. |
| 66-67 | Раскраска графов. Двудольные графы. Хроматический полином. Окраска вершин. Раскраска ребер. |
| 68-69 | Максимальная клика. Минимальные вершинные покрытия. Максимальные независимые множества. |
| 70-71 | Самостоятельная работа №10 |
| | **Алгоритмическая теория графов** |
| 72-75 | Кратчайшие пути. Одноисточниковые кратчайшие пути. Все пары кратчайших путей. Число путей. Приложения. |
| 76-78 | Минимальные остовные деревья. Алгоритм Краскала. Подсчет остовных деревьев. Сетевые потоки. |
| 79-80 | Паросочетания. Максимальные паросочетания. Двудольные паросочетания. Взвешенное двудольное паросочетание и вершинное покрытие. Задача устойчивого брака. |

| | |
|---|---|
| 82-83 | Частичные порядки. Топологическая сортировка. Транзитивное покрытие и транзитивная редукция. Диаграммы Хассе. |
| 84-85 | Изоморфизмы графов. Нахождение изоморфизмов. Изоморфизм деревьев. Самодополняемые графы. |
| 86-87 | Планарные графы. Тестирование планарности. Двойственные графы |
| 88-90 | Самостоятельная работа №11 |

# Глава 1
# НЕОБХОДИМЫЕ СВЕДЕНИЯ ИЗ СИСТЕМЫ “MATHEMATICA”

Наиболее полное описание системы Mathematica дано в монографии [13].
В современных версиях Mathematica имеется диалоговая подсказка. Оперативную помощь о назначении какой-либо функции или объекте в ходе работы с системой можно получить, используя следующие обращения:
? Name или ? Names –справка по заданному объекту Name;
?? Name– расширенная справка по заданному объекту Name:
?Abc* - перечень всех определений, начинающихся с символов Abc;
Options[name] – получение информации об опциях объекта name;
Стандартные пакеты расширения, как, например, Combinatorica должен быть загружен в Mathematica, после чего новые функции из пакета не отличаются от встроенных функций Mathematica.
Эта команда загружает пакет DiscreteMath`Combinatorica:

```
In[1]:= <<DiscreteMath`Combinatorica`
```

Это справка о функции:

```
In[2]:= ?Bridges
```

А теперь расширенная справка:

```
In[3]:= ??PermutationQ

      PermutationQ[p] yields True if p is a list
         representing a permutation and False otherwise. More...

      Attributes[PermutationQ] = {Protected}

      PermutationQ[DiscreteMath`Combinatorica`Private`e_List] :=
       Sort[DiscreteMath`Combinatorica`Private`e] ===
        Range[Length[DiscreteMath`Combinatorica`Private`e]]
```

Вызовем все функции, начинающиеся с букв Ad:

```
In[4]:= ?Ad*

System`
AddOnHelpPath  AdjustmentBox
AddTo          AdjustmentBoxOptions

DiscreteMath`Combinatorica`
AddEdge  AddEdges  AddVertex  AddVertices
```

## 1.1 Работа со списками

Здесь мы рассмотрим списки – один из наиболее фундаментальных способов структурирования данных.

### *1.1.1. Конструирование списков*

Список есть выражение “Mathematica”имеющее заголовок List. Его элементами могут быть любые выражения “Mathematica”, в том числе списки с любыми уровнями вложенности, графические объекты. Список задается следующим образом:
List[x1, x2, . . . , xn] или {x1, x2, . . . , xn},
Пример списка:
{10, Sin[x], List [a, b, c, d], Exp}
Матрица может быть задана как список:

```
In[2]:= {{3, s, 9, 15}, {a, r, 4, 9}, {0, 6, 1, 12}, {0, 6, 0, 1}} //
        MatrixForm

Out[2]//MatrixForm=
        ⎛3  s  9  15⎞
        ⎜a  r  4   9⎟
        ⎜0  6  1  12⎟
        ⎝0  6  0   1⎠
```

Список можно вывести также в виде столбца и таблицы:

```
In[3]:= l = {{2, s, 4, 5}, {a, b, c, d}, {78, 6, 9, 4}, {0, 0, 0, 0}} // ColumnForm
Out[3]= {2, s, 4, 5}
        {a, b, c, d}
        {78, 6, 9, 4}
        {0, 0, 0, 0}
```

Существует четыре функции, порождающих списки: List, Range, Table и Array. Перечислим их форматы кодирования.

| | |
|---|---|
| List[x1, x2, . . . , xn ] | список {x1, x2, . . . ,xn } |
| Range [n] | список {1, 2,. . .,n} |
| Range [m, n] | список {m, m+1,….,n} |
| Table[f, {n}] | генерирует список из n одинаковых элементов f |
| Table[f, {i, n}] | генерирует список n элементов – значений выражения $f$(i), где i = 1,…,n |
| Table[f, {i, m, n}] | генерирует список n-m+1 элементов – значений выражения $f$(i), где i = m, m+1,…,n |
| Table[f, {i, m, n, di}] | то же, с шагом di |
| Table[f, {i, m, n},{j, p, q}, ...] | порождает вложенный список элементов f(i,j, . . .) (i — внешний индекс) |
| Array [a, n] | генерирует список длины n с элементами a[i] для i = 1,…, n |
| Array [a, {n1, $n_2$, ... }] | генерирует вложенный список элементов a[i1,i2, . . ] (где i1 = 1,..,n2, i2 = 1,…,n2, . . .)) общее количество которых n1× n2 × . . . |
| Array [a, iterators, m] | порождает список, в котором индексы аргумента iterators, имеющего вид n или {n1,n2, . . .}, изменяются со значения m (в предыдущих двух видах функции Array равного по умолчанию 1) |
| Array [a, iterators, m, h] | выражение, отличающееся от предыдущего тем, что заголовок List заменяется на заголовок h |

### *1.1.2. Функции выявления структуры списков*

Списки относятся к данным сложной структуры. Поэтому при работе с ними возникает необходимость контроля за структурой, иначе применение списков может привести к грубым ошибкам.

Следующие 10 функций служат для выявления структуры списка (L) или произвольного допустимого выражения (expr).Функции с буквой Q в конце имени являются тестирующими и возвращают логические значения True или False. Остальные функции возвращают численные значения соответствующего параметра списка

| | |
|---|---|
| Length[expr] | длина выражения (количество его элементов) |
| Dimensions [expr] | даёт список размерностей выражения |
| VectorQ[expr] | возвращает True, если expr является вектором (невложенным списком) |
| MatrixQ[expr] | возвращает True, если expr является матрицей |
| MatrixQ[expr, crit] | возвращает True, если expr является матрицей, элементы которой удовлетворяют критерию crit |
| TensorRank[expr] | находит ранг (валентность) тензора, если выражение expr является таковым |
| TensorRank [L] | эквивалентно Length[Dimensions[L]] |
| Depth[expr] | (максимальный уровень вложенности выражения) + 1 |
| Count [L, pattern] | возвращает число элементов списка, соответствующего шаблону pattern |
| MemberQ[L, form] | возвращает True, если в списке есть элемент вида form, False — в противном случае |
| FreeQ[expr, form] | возвращает True, если в выражении нет подвыражения, соответствующего form и False — в противном случае |
| Position[ expr, pattern] | даёт список позиций объектов выражения, соответствующих шаблону pattern |

Следующие простые примеры поясняют смысл перечисленных функций (некоторые из них, ввиду очевидности, оставляем без комментариев).

Найдем длину следующего списка:

```
In[2]:= Length[{{a, b}, 2, x, 4, f}]

Out[2]= 5
```

Найдем глубину этого же списка:

```
In[3]:= Depth[{{a, b}, 2, x, 4, f}]

Out[3]= 3
```

Подсчитаем количество единиц в списке:

```
In[4]:= Count[{1, 2, 3, 2, 4, 51, 4, 5, 1, 1, 4, 8, 9, 1, 1, 1}, 1]

Out[4]= 6
```

Проверим, является ли символ r элементом списка:

```
In[5]:= MemberQ[{{{a, b}, 2, 4, 6}, r, 5}, r]

Out[5]= True
```

Тот же результат получим другим образом:

```
In[6]:= FreeQ[{{{a, b}, 2, 4, 6}, r, 5}, r]

Out[6]= False
```

Найдем позицию элемента r в предыдущем списке:

```
In[7]:= Position[{{{a, b}, 2, 4, 6}, r, 5}, r]

Out[7]= {{2}}
```

### *1.1.3. Выделение элементов списков*

Часто возникает необходимость оперировать частями списка (выражения). Перечислим функции, предназначенные для получения частей списка. Как и в предыдущем пункте, применим обозначения: L — список, expr – произвольное допустимое выражение (в том числе и список).

| | |
|---|---|
| First[expr] | первый элемент выражения |
| Last [expr] | последний элемент выражения |
| Part[expr, n], или expr [[n]] | n-й элемент выражения |
| expr [[-n]] | n-й элемент выражения, считая с его конца |
| Part[expr, m, n, ...], или expr[[m, n, ... ]] | элемент, находящийся на позиции [m, n, ...] |
| Expr[[{m, n, ...}]] | подвыражение, состоящее из элементов с номерами m, n, ... |
| Take[L, n] | даёт первые n элементов списка |
| Take[L, -n] | даёт последние n элементов списка |
| Take[L, {m, n}] | даёт фрагмент списка, содержащий элементы с m-го по n-й (включительно) |
| Rest[expr] | даёт выражение без его первого элемента |
| Drop[L, n] | даёт список без его n первых элементов |
| Drop[L. -n] | даёт список без его n последних элементов |
| Drop[L, {m, n}] | возвращает список без его фрагмента с m-го элемента по n-й (включительно) |
| Extract [expr, {m, n, …}] | извлекает подвыражение с позицией [m, n, ...] (для вложенного списка — элемент a[m, n,...]) |
| Extract [expr,{Ll,L2,…}] | извлекает список подвыражений выражения |
| Select[L,crit] | выбирает все элементы списка L, для которых выполняется crit |
| Select[L, crit, n] | отбирает первые n элементов списка L, удовлетворяющих критерию crit |
| Cases[L, pattern] | список элементов $a_i$ списка L, сравнимых с шаблоном pattern |

Введем список l и применим к нему функции из вышеприведенной таблицы:

```
In[2]:= l = {{a, b}, {{3, 4}, d}, 2, {t, 1, 2}, {x^2 + x, m, n}, 5};
        {First[l], Last[l], l[[3]], l[[-2]], l[[1, 2]], l[[{2, 4}]],
          Take[l, 3], Take[l, -3], Take[l, {2, 3}], Rest[l], Drop[l, 2],
          Drop[l, -2], Drop[l, {2, 4}], Extract[l, {1, 2}], Extract[l, {2, 1, 2}],
          Select[l, IntegerQ], Select[l, IntegerQ, 6], Cases[l, _Integer]} //
         ColumnForm
```

```
Out[3]= {a, b}
        5
        2
        {x + x^2, m, n}
        b
        {{{3, 4}, d}, {t, 1, 2}}
        {{a, b}, {{3, 4}, d}, 2}
        {{t, 1, 2}, {x + x^2, m, n}, 5}
        {{{3, 4}, d}, 2}
        {{{3, 4}, d}, 2, {t, 1, 2}, {x + x^2, m, n}, 5}
        {2, {t, 1, 2}, {x + x^2, m, n}, 5}
        {{a, b}, {{3, 4}, d}, 2, {t, 1, 2}}
        {{a, b}, {x + x^2, m, n}, 5}
        b
        4
        {2, 5}
        {2, 5}
        {2, 5}
```

### *1.1.4. Преобразования списков*

Естественными преобразованиями списков являются прибавление элементов к списку и удаление элементов из списка. В следующей таблице, как и в предыдущей, expr будет означать любое допустимое выражение (в частности, список).

| | |
|---|---|
| Append[expr, a] | дописывание элемента a к выражению expr в его конец |
| Prepend [expr, a] | дописывание элемента a к выражению в его начало |
| Insert[L, a, n] | вставить элемент a в список L на n-е место (т. е. перед n-м элементом L) |
| Insert[expr, a, {m, n, ...}] | вставить элемент a в выражение expr на позицию [m, n, ...] |
| Insert[expr, a, {{m, n, …}, {p, q, …}, …}] | вставить элемент a в выражение на позиции [m, n, ...], [p,q, ...],.. |
| Delete[expr, n] | удалить n-й элемент выражения (при отрицательном n позиция удаляемого элемента отсчитывается с конца) |
| Delete[expr, (m, n, ... }] | удалить элемент выражения, стоящий на позиции [m, n, ...] |
| Delete [expr, {{m, n, . .. },{p, q, ...}, … }] | удалить элементы выражения, занимающие позиции [m, n ...], [p, q, ...], … |
| DeleteCases[expr , pattern] | удалить все элементы выражения expr, сравнимые с шаблоном pattern |
| ReplacePart[expr, a, n] | заменить n-й элемент выражения expr элементом a |
| ReplacePart[expr, a, {m, n, ...}] | заменить элемент с позицией [m, n, ...] элементом a |
| ReplacePart[expr, a, {m, n, .. . },{p, q, .. . }, ... }] | заменить элементы на позициях [m, n, ... ], [p, q, ... ], ... элементом a |

Примеры применения этих функций:

```
In[2]:= {Append[l = {x, y, z, m, {1, 2}}, a], Prepend[l, a]} // ColumnForm

Out[2]= {x, y, z, m, {1, 2}, a}
        {a, x, y, z, m, {1, 2}}

In[3]:= Insert[l, a, 3]

Out[3]= {x, y, a, z, m, {1, 2}}
```

Элемент a вставлен на третье место списка; при этом все последующие элементы списка приобретают номера, на единицу большие, чем они имели до вставки элемента a. Ни один элемент не удаляется.

```
In[4]:= Insert[l, a, -3]

Out[4]= {x, y, z, a, m, {1, 2}}
```

При отрицательном n элемент вставляется на n-е место, считая с конца (в данном примере - на третье).

```
In[5]:= Insert[{x, y, z, m, {1, 2}}, a, {5, 1}]

Out[5]= {x, y, z, m, {a, 1, 2}}
```

Элемент a вставлен в список на позицию [5, 1].

```
In[6]:= Delete[{{x, {5, 4, 7}, y, z}, b, {c, {d, e, f}}}, 3]

Out[6]= {{x, {5, 4, 7}, y, z}, b}
```

Удалён третий элемент списка.

```
In[7]:= Delete[{{x, {7, 6, 5}, y, z}, b, {c, {d, e, f}}}, {1, 2}]

Out[7]= {{x, y, z}, b, {c, {d, e, f}}}
```

Удалён элемент {7, 6, 5}, стоявший в списке на позиции [1, 2] (второй элемент в списке, являющемся первым элементом данного списка).

```
In[8]:= DeleteCases[{b, {1, 2, a}, d}, a]

Out[8]= {b, {1, 2, a}, d}
```

Несмотря на команду удалить из списка элемент a, список остался без изменения. Дело в том, что команда относится только к элементам первого уровня, а они характеризуются так: список, символ, список. Чтобы отнести команду к элементам всех уровней с первого по второй, нужно третьим аргументом функции задать 2 (номер последнего уровня). Получится следующее:

```
In[9]:= DeleteCases[{b, {1, 2, a}, d}, a, 2]

Out[9]= {b, {1, 2}, d}
```

Удалим все элементы (первого уровня), являющиеся списками.

```
In[10]:= DeleteCases[{{x, y, z}, {{m, n}, 7}, {{5}}, a, s, de, {3, 4}},
          _List]

Out[10]= {a, s, de}
```

Удалим третий элемент списка, на его место вставлен новый элемент a (сравните с функцией Insert, при действии которой удаление не производится).

```
In[11]:= ReplacePart[{x, y, {m, n}, z}, a, 3]

Out[11]= {x, y, a, z}
```

Заменим элементом {7,8,9} элемент c, стоявший на позиции [2, 3, 1.

```
In[12]:= ReplacePart[{{a, 1, 2}, {3, 4, {c, d}}, x, 6}, {7, 8, 9}, {2, 3, 1}]

Out[12]= {{a, 1, 2}, {3, 4, {{7, 8, 9}, d}}, x, 6}
```

Функция Sort[L] сортирует список, т. е. располагает его элементы в каноническом (с точки зрения Mathematica) порядке: числа располагаются первыми и по возрастанию, далее идут буквы в алфавитном порядке, затем списки в порядке возрастания их сложности, затем -- функции (в алфавитном порядке). Другой формат кодирования для этой функции: Sort[L, p]. В этом случае список подвергается сортировке согласно заданной функции p. устанавливающей порядок.

```
In[13]:= Sort[{u, x, 6, 13, 1, Cos[y], a, Pi, E, Log[x]}]

Out[13]= {1, 6, 13, a, e, π, u, x, Cos[y], Log[x]}
```

Упорядочим список, элементами которого являются числа, в порядке убывания.

```
In[14]:= Sort[{5, 1, 3.4, 69, 17}, Less]

Out[14]= {1, 3.4, 5, 17, 69}
```

Две функции комбинируют списки: Join и Union.

| | |
|---|---|
| Join[$L_1$ $L_2$, ...] | конкатенация (сцепление) списков $L_1$ $L_2$, , ... вместе в один список |
| Union[$L_1$ $L_2$, ...] | объединяет списки, вычёркивая повторяющиеся элементы, и сортирует результат |

Это пример конкатенации двух списков,

```
In[15]:= Join[{w, e, r, t, x}, {1, 2, 3}]

Out[15]= {w, e, r, t, x, 1, 2, 3}
```

а это пример объединения.

```
In[16]:= Union[{s, r, e, 2, 3, 4}, {2, 3, 7, s}]

Out[16]= {2, 3, 4, 7, e, r, s}
```

Функции пересечения и дополнения – аналоги теоретико-множественных операций над списками:

| | |
|---|---|
| Intersection[$L_1$ $L_2$, ...] | отсортированное теоретико-множественное пересечение списков |
| Complement[L, $L_1$ $L_2$, ...] | отсортированный список элементов списка L, не входящих в списки $L_1$ $L_2$, ... |

```
In[17]:= Intersection[{1, 2, s, d, r, t}, {3, 4, 5, 2, r, t}]

Out[17]= {2, r, t}

In[18]:= Complement[{1, 2, 3, 4, s, d, f}, {3, 4, d, f, a}]

Out[18]= {1, 2, s}
```

Следующие пять функций можно отнести к одной группе — функций переранжировки списков (выражений):

| | |
|---|---|
| Sort[L] | сортирует элементы в стандартном порядке |
| Union [L] | сортирует элементы, удаляя повторяющиеся |
| Reverse [expr] | меняет порядок выражения на обратный |
| RotateLeft[ expr, n] | переставляет первый элемент выражения на его конец (циклирует слева на одну позицию) |
| RotateLeft[expr,{m, n, ... }] | циклирует элементы последовательно расположенных уровней соответственно на. m, n, .. . позиций (слева) |
| RotateRight[expr, n] | циклирует элементы на n позиций справа |
| RotateRight [ expr] | циклирует элементы на одну позицию справа |
| RotateRight[expr,{m, n, ... }] | циклирует элементы последовательно расположенных уровней на m, n, .. .позиций справа |

Приведем примеры применения этих функций:

```
In[19]:= l = {a, b, c, d, e, f, a};
         {Sort[l], Union[l], Reverse[l]} // ColumnForm
Out[20]= {a, a, b, c, d, e, f}
         {a, b, c, d, e, f}
         {a, f, e, d, c, b, a}

In[21]:= {RotateLeft[l, 2], RotateRight[l, 2]} // ColumnForm

Out[21]= {c, d, e, f, a, a, b}
         {f, a, a, b, c, d, e}

In[22]:= l = {{a, b, c}, {1, 3, 5}, {x, y, z, t}, {0, 4, v}};
         {RotateLeft[l, {3, 4}], RotateRight[l, {3, 4}]} // ColumnForm

Out[22]= {{4, v, 0}, {b, c, a}, {3, 5, 1}, {x, y, z, t}}
         {{5, 1, 3}, {x, y, z, t}, {v, 0, 4}, {c, a, b}}
```

В заключении этого пункта рассмотрим функции изменения структуры списков.

| | |
|---|---|
| Partition[L, n] | разбивает список на непересекающиеся части длины n, начиная с первого элемента (если получается остаток длины меньше n, то он отбрасывается) |
| Partition[L, n, d] | разбивает список на части длины n с отступом d (т. е. второй подсписок начинается с (d+ 1)-го элемента L и т. д.); при d < n подсписки перекрываются |
| Partition[L, {n1, n2, ... }, {di,$d_2$, ...}] | разбивает списки 1-го, 2-го, ... уровней на подсписки длин n1, п2, ... с отступами d1, d2, ... соответственно |
| Flatten [L] | приводит вложенный список к одноуровневому |
| Flatten[L, n] | отменяет вложенность списка первых п уровней |
| Flatten[L, n, h] | отменяет заголовок h у подвыражений списка на первых п уровнях |
| Transpose[L] | транспонирует первый и второй уровни списка |
| Transpose[L, {$n_1$, $n_2$, ... }] | транспонирует список, придавая уровню с номером $n_i$ в новом списке значение уровня с номером i в исходном списке |
| Split [L] | разбивает список на подсписки, содержащие серии подряд идущих одинаковых элементов |

Разобьем список на подсписки длины 2

```
In[23]:= Partition[{a, b, c, d, e, f, g, h}, 2]
Out[23]= {{a, b}, {c, d}, {e, f}, {g, h}}
```

Разобьем этот же список на подсписки длины 3. Так как длина списка 8 не делится нацело на 3, остаток удаляется из списков.

```
In[24]:= Partition[{a, b, c, d, e, f, g, h}, 3]
Out[24]= {{a, b, c}, {d, e, f}}
```

Тот же список разбит на части длины 3 с отступом 2, т. е. первый элемент второго подсписка имел номер 3 в исходном списке, первый элемент третьего подсписка имел номер 5 в исходном списке. Первым элементом (i + 1)-го подсписка всегда будет элемент под номером d-i + 1 исходного списка.

```
In[25]:= Partition[{a, b, c, d, e, f, g, h}, 3, 2]
Out[25]= {{a, b, c}, {c, d, e}, {e, f, g}}
```

Теперь разобьем этот список с отступом 1:

```
In[26]:= Partition[{a, b, c, d, e, f, g, h}, 3, 1]
Out[26]= {{a, b, c}, {b, c, d}, {c, d, e}, {d, e, f}, {e, f, g}, {f, g, h}}
```

Рассмотрим матрицу размера 5×5:

```
In[27]:= (l = Partition[Range[25], 5]) // MatrixForm
```

Out[27]//MatrixForm=

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 & 25 \end{pmatrix}$$

Разобьем эту матрицу на блоки размером 2×2.Последний столбец и последняя строка удалены из блоков:

```
In[28]:= Partition[l, {2, 2}] // MatrixForm
```

Out[28]//MatrixForm=

$$\begin{pmatrix} \begin{pmatrix} 1 & 2 \\ 6 & 7 \end{pmatrix} & \begin{pmatrix} 3 & 4 \\ 8 & 9 \end{pmatrix} \\ \begin{pmatrix} 11 & 12 \\ 16 & 17 \end{pmatrix} & \begin{pmatrix} 13 & 14 \\ 18 & 19 \end{pmatrix} \end{pmatrix}$$

Сделаем отступ 1 в первом уровне (в строке) и отступ 2 во втором уровне (в столбце):

```
In[29]:= (l1 = Partition[l, {2, 2}, {1, 2}]) // MatrixForm
```

Out[29]//MatrixForm=

$$\begin{pmatrix} \begin{pmatrix} 1 & 2 \\ 6 & 7 \end{pmatrix} & \begin{pmatrix} 3 & 4 \\ 8 & 9 \end{pmatrix} \\ \begin{pmatrix} 6 & 7 \\ 11 & 12 \end{pmatrix} & \begin{pmatrix} 8 & 9 \\ 13 & 14 \end{pmatrix} \\ \begin{pmatrix} 11 & 12 \\ 16 & 17 \end{pmatrix} & \begin{pmatrix} 13 & 14 \\ 18 & 19 \end{pmatrix} \\ \begin{pmatrix} 16 & 17 \\ 21 & 22 \end{pmatrix} & \begin{pmatrix} 18 & 19 \\ 23 & 24 \end{pmatrix} \end{pmatrix}$$

```
In[30]:= l1
```

```
Out[30]= {{{{1, 2}, {6, 7}}, {{3, 4}, {8, 9}}},
          {{{6, 7}, {11, 12}}, {{8, 9}, {13, 14}}},
          {{{11, 12}, {16, 17}}, {{13, 14}, {18, 19}}},
          {{{16, 17}, {21, 22}}, {{18, 19}, {23, 24}}}}
```

Теперь с помощью функции Flatten список l1 превратим в линейный (одноуровневый).

```
In[31]:= Flatten[l1]

Out[31]= {1, 2, 6, 7, 3, 4, 8, 9, 6, 7, 11, 12, 8, 9, 13, 14, 11,
          12, 16, 17, 13, 14, 18, 19, 16, 17, 21, 22, 18, 19, 23, 24}
```

Теперь отменим вложенность второго уровня:

```
In[32]:= Flatten[l1, 2]

Out[32]= {{1, 2}, {6, 7}, {3, 4}, {8, 9}, {6, 7}, {11, 12},
          {8, 9}, {13, 14}, {11, 12}, {16, 17}, {13, 14},
          {18, 19}, {16, 17}, {21, 22}, {18, 19}, {23, 24}}
```

В этом списке отменим заголовок h у элементов второго уровня

```
In[33]:= Flatten[h[h[a, b], h[h[h[e]]]], 2]

Out[33]= h[a, b, h[e]]
```

Разобьем последовательность на соприкасающиеся подпоследовательности одинаковых элементов

```
In[34]:= Split[{1, 1, 1, 3, 3, 6, 6, 6, 2, 2, 2, 2, 3, 3, 1, 4, 4}]

Out[34]= {{1, 1, 1}, {3, 3}, {6, 6, 6}, {2, 2, 2, 2}, {3, 3}, {1}, {4, 4}}
```

### *1.1.5. Взятие функций от списка*

Если f — математическая функция или операция, то её действие на список L во входной строке можно оформить тремя способами:
f[L] , или f@L , или L // f .
Встроенные математические функции и операции обладают свойством дистрибутивности относительно списков, т. е. они действуют на каждый элемент списка (списков).

```
In[2]:= {Sin[{1, x, y}], Sin@{1, x, y}, {1, x, y} // Sin} // ColumnForm
Out[2]= {Sin[1], Sin[x], Sin[y]}
        {Sin[1], Sin[x], Sin[y]}
        {Sin[1], Sin[x], Sin[y]}

In[3]:= Plus[{1, 2, 3}, {6, 7, 8}]

Out[3]= {7, 9, 11}
```

Как видим, операция сложения двух списков одинаковой длины также дистрибутивна. Результатом стал список сумм соответствующих элементов данных списков.
Применяя к списку произвольную функцию, мы уже не получим такой результат, поскольку произвольные функции не обладают дистрибутивностью относительно списков:

```
In[4]:= f@{3, 5, a}
Out[4]= f[{3, 5, a}]
```

Для придания произвольной функции или операции свойства дистрибутивности служит функция Mathematica - Map. Кодирование этой функции:

| | |
|---|---|
| Map[f, expr], или f/@expr | применяет f к каждому элементу первого уровня expr |
| Map[f, expr, levelspec] | применяет f к каждому элементу уровня levelspec expr |

```
In[5]:= {Map[f, {x^6 + 2 y + z}], Map[f, {x, y, z}]}

Out[5]= {{f[x^6 + 2 y + z]}, {f[x], f[y], f[z]}}
```

Произвольная функция f подействовала дистрибутивно на выражение x^6+2 y+z и на список {x, y, z}.
Если expr (L) является многоуровневым выражением (вложенным списком), то заголовок f можно применить к элементам отдельных уровней:

| | |
|---|---|
| MapAt[f, expr, n] | применяет f к элементам в позиции n в выражении expr |
| MapAt[f, expr,{ i, j, … }] | применяет f к частям expr в позиции { i, j, … } |
| MapAt[f, expr, {{$i_1$, $j_1$, … }, {$i_2$, $j_2$, … }, … }] | применяет f к частям expr в позициях {{$i_1$, $j_1$, … }, {$i_2$, $j_2$, … }, … |

Применим функцию f ко второму и четвертому элементам списка:

```
In[6]:= MapAt[f, {a, b, c, d}, {{2}, {4}}]

Out[6]= {a, f[b], c, f[d]}
```

Теперь применим f к элементам , стоящим на позициях {3,2,1} и {3,1,2}^

```
In[7]:= MapAt[f, {a, {n, m, c}, {{d, e}, {p, q}}}, {{3, 2, 1}, {3, 1, 2}}]

Out[7]= {a, {n, m, c}, {{d, f[e]}, {f[p], q}}}
```

Дистрибутивна ли функция, можно узнать, вычислив выражение Attributes[f]. Если Listable является атрибутом функции, то она дистрибутивна. Пример:

```
In[8]:= Attributes[Tan]

Out[8]= {Listable, NumericFunction, Protected}
```

Функция Tan дистрибутивна.
Если дистрибутивную операцию применить к спискам одинаковой длины, то, как мы видели, результатом будет применение этой операции к элементам списков с одинаковыми номерами. Если же вместо одного из списков взять символ, то он копируется столько раз, какова длина аргументов-списков (как если бы вместо этого символа был задан список с одинаковыми элементами, той же длины, как другие аргументы-списки),

```
In[9]:= Power[{a, y, z, 4}, {4, 3, 2, 5}]

Out[9]= {a^4, y^3, z^2, 1024}
```

Применим степенную функцию как кубическую функцию с с разными аргументами-основаниями и как разные степенные функции с показателями 2, 4, 6, 8 от одного аргумента x.

```
In[10]:= {Power[{a, y, z, 4}, 3], Power[x, {4, 3, 2, 5}]}

Out[10]= {{a^3, y^3, z^3, 64}, {x^4, x^3, x^2, x^5}}
```

Встроенная функция Mathematica Apply[f, expr] (или, что то же, f@@expr) производит замену заголовка выражения expr на заголовок f.

| | |
|---|---|
| Apply[f,{a,b,c,d}] | применяет f к списку, возвращая f[a,b,c,d] |
| Apply[f,expr] или f@@expr | применяет f к верхнему уровню expr |
| Apply[f,expr,{1}] или f@@@expr | применяет f к первому уровню expr |
| Apply[f,expr,lev] | применяет f к уровню lev expr |

Заменим списком сумму a+b+c

```
In[11]:= Apply[List, a + b + c]

Out[11]= {a, b, c}
```

Применим f к разным уровням:

```
In[12]:= m = {{a, b, c}, {b, c, d}};
         {Apply[f, m], Apply[f, m, {1}], Apply[f, m, {0, 1}]} //
          ColumnForm

Out[13]= f[{a, b, c}, {b, c, d}]
         {f[a, b, c], f[b, c, d]}
         f[f[a, b, c], f[b, c, d]]
```

Функция Thread[f[args]] "продевает" (как нить) заголовок *f* сквозь все подвыражения (в том числе; списки), находящиеся среди аргументов args. Эта функция имеет и другие форматы кодирования.

| | |
|---|---|
| Thread[f[args]] | "продевает" заголовок f сквозь все подвыражения (в том числе; списки), находящиеся среди аргументов args. |
| Thread[f[args], h] | "продевает" заголовок f сквозь все подвыражения с заголовком h, находящиеся среди аргументов args. |
| Thread[f[arg], h, n] | "продевает" заголовок f сквозь все подвыражения с заголовком h, находящиеся среди первых n аргументов args. |
| Thread[f[arg], h, -n] | то же самое, но только для последних n аргументов |
| Thread[f[arg], h, {m, n}] | то же самое, но только для аргументов от m до n |

Здесь h применяется ко всем элементам с заголовком Plus.

```
In[15]:= Thread[h[a + b, c d], Plus]

Out[15]= h[a, c d] + h[b, c d]
```

Для сравнения применим h ко всем элементам с заголовком Times

```
In[16]:= Thread[h[a + b, c d], Times]

Out[16]= h[a + b, c] h[a + b, d]
```

Здесь h применяется к последним двум аргументам

```
In[17]:= Thread[h[a + b, c, d + e], Plus, -2]

Out[17]= h[a + b, c, d] + h[a + b, c, e]
```

Теперь применим h к элементам от второго до четвертого:

```
In[18]:= Thread[h[a+b, c+d, e+f, g+h], Plus, {2, 4}]

Out[18]= h[a+b, c, e, g]+h[a+b, d, f, h]
```

## 1.2. Основы программирования

Mathematica является диалоговым языком сверхвысокого уровня.
Можно выделить три основных подхода к программированию: процедурное программирование (задаёт пошаговые алгоритмы), функциональное программирование (определяет набор функций, которые следует применять) и программирование, основанное на правилах преобразований (задаёт математические соотношения).

### *1.2.1 Функциональное программирование*

Мощь системы Mathematica как средства программирования решения математических задач обусловлена необычно большим набором функций, среди которых немало таких, которые реализуют достаточно сложные алгоритмы. Основная идея функционального программирования - составлять программу из функций, каждая из которых использует результаты предыдущих, т. е. сначала строится функция от аргументов, затем — функция от этой функции и т. д. Важно так писать программу, чтобы структура композиции функций была ясной. Функции, не содержащиеся в ядре системы, а задаваемые пользователем, называются внешними.
Для обозначения переменной можно использовать так называемые именованные шаблоны (Patterns). Они служат для задания выражений различных классов и придания переменным особых свойств. Это очень гибкое и мощное средство обобщенного представления математических выражений.
Признаком образца являются знаки подчеркивания «_» (от одного до трех). Наиболее распространенное применение образцов – указание на локальный характер переменных при задании внешней функции. Например, функция

```
In[1]:= f[x_, y_] := x+y* Sin[z];
```

в списке параметров содержит два шаблона x_ и y_. В правой части этого выражения переменные x,y, связанные с образцами x_ ,y_ становятся локальными переменными, тогда как переменная z будет глобальной переменной. В качестве x,y могут быть и списки:

```
In[2]:= f[{2, 1, 3}, {4, 5, 6}]

Out[2]= {2+4 Sin[z], 1+5 Sin[z], 3+6 Sin[z]}
```

В шаблоне можно указывать тип его данных:
x_ Integer –целочисленный шаблон;
x_Real –шаблон с действительным значением;
x_Complex- шаблон с комплексным значением;
x_List- шаблон с заголовком List
x_h - шаблон с заголовком h.
В системе Mathematica используются следующие типы шаблонов

| | |
|---|---|
| _ | любое выражение |
| x_ | любое выражение, представленное именем x |
| x:pattern | шаблон, представленный именем x |
| pattern ? test | возвращает True, когда test применен к значению шаблона |
| _h | любое выражение с заголовком h |

| | |
|---|---|
| x_h | любое выражение с заголовком h, представленное именем x |
| __ | любая последовательность с одним или более выражений |
| ___ | любая последовательность с нулем или более выражений |
| x__ или x___ | последовательности выражений, представленные именем x |
| __h или h___ | последовательности выражений, каждое с заголовком h |
| x__h или x__h | последовательности выражений с заголовком h, представленные именем x |
| x_:v | выражение с определенным значением v |
| x_h:v | выражение с заголовком h и определенным значением v |
| x_. | выражение с глобально заданным значением по умолчанию |
| Optional[x_h] | выражение с заголовком h и с глобально заданным значением по умолчанию |
| Pattern .. | шаблон, повторяемый один или более раз |
| Pattern … | шаблон, повторяемый ноль или более раз |

Введем внешнюю функцию для возведения x в степень n:

```
In[3]:= power[x_, n_] := x^n;
```

Можно задать внешнюю функцию, содержащую несколько выражений, заключив их в круглые скобки:

```
In[4]:= f[x_] := (s = (3 + 2 x) ^3 + Power[x, 3]; s = Expand[s])
In[5]:= f[a]
Out[5]= 27 + 54 a + 36 a^2 + 9 a^3
```

После своего задания внешние функции могут использоваться по тем же правилам, что и встроенные функции.
Если внешняя функция, возникающая в определённом месте программы, больше нигде, кроме этого места, не используется, то вы не обязаны присваивать ей определённое имя. Функция, использующаяся только в момент её создания и не являющаяся именованной, называется чистой функцией.
Есть несколько эквивалентных способов образовать чистую функцию. Идея во всех случаях одна: конструировать объект, который, будучи снабжён подходящими аргументами, вычислит конкретную функцию. Для построения чистой функции используется встроенная функция Function (или в сокращённой постфиксной форме &)

| | |
|---|---|
| Function[x, body] | чистая функция, в которой x заменяется на любой аргумент, который вы дадите (т. е. x — локальная переменная); body — так называемое тело функции, задаваемое выражением Mathematica |
| Function[{x1., x2, ...}, body] | чистая функция от нескольких аргументов |
| Fxmction[body], или body & | чистая функция, аргументы которой определяются как # или #1, #2, #3, .... |

Применение чистой функции к аргументу осуществляется как обычно: если fun — чистая функция, то fun [a] вычисляет её значение от a.

```
In[6]:= g[x_, y_] := #1^#2 &[x, y];
        {Function[{x, y}, x^y][3, 4], g[3, 4]}
Out[6]= {81, 81}
```

При функциональном программировании часто используется суперпозиция функций.Для ее реализации используются следующие функции

| | |
|---|---|
| Nest[expr,x,n] | n раз применяет выражение (функцию) expr к аргументу x |
| NestList[f,x,n] | возвращает список результатов (n+1) –кратного применения f к аргументу x |
| FoldList[f, x, {a, b, … }] | возвращает {x, f[x, a], f[f[x, a], b], … } |
| Fold[f,x,list] | дает последний элемент FoldList[f, x,list] |
| ComposeList[{$f_1$,$f_2$,…},x] | генерирует список в форме { {x, $f_1$[x], $f_2$[$f_1$ [x]], … }. } |

Покажем, как можно создать цепную дробь с помощью функции Nest:

```
In[7]:= Nest[Function[t, 1 / (1 + t)], y, 4]
```

$$\text{Out[7]}= \cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+y}}}}$$

Вычислим последовательное выполнение функции Sin:

```
In[8]:= NestList[Sin, y, 7]

Out[8]= {y, Sin[y], Sin[Sin[y]], Sin[Sin[Sin[y]]], Sin[Sin[Sin[Sin[y]]]],
         Sin[Sin[Sin[Sin[Sin[y]]]]], Sin[Sin[Sin[Sin[Sin[Sin[y]]]]]],
         Sin[Sin[Sin[Sin[Sin[Sin[Sin[y]]]]]]]}

In[9]:= FoldList[h, x, {1, 2, 3}]

Out[9]= {x, h[x, 1], h[h[x, 1], 2], h[h[h[x, 1], 2], 3]}

In[10]:= ComposeList[{Sin, Cos, Tan}, a]

Out[10]= {a, Sin[a], Cos[Sin[a]], Tan[Cos[Sin[a]]]}
```

В функциональном программировании вместо циклов могут использоваться следующие функции

| | |
|---|---|
| FixedPoint[f,expr] | вычисляет expr и применяет к нему f, пока результат не перестанет изменяться |
| FixedPoint[f,expr, SameTest–>comp] | вычисляет expr и применяет к нему f, пока два последующих результата не дадут True в тесте SameTest |
| Catch[expr] | вычисляет expr пока не встретится Throw[v] |
| Catch[expr,form] | вычисляет expr пока не встретится Throw[value,tag], затем возвращает value |
| Catch[expr,form,f] | возвращает f[value,tag] вместо value |

```
In[11]:= FixedPoint[Function[t, Print[t]; Floor[t/5]], 18]

         18

         3

         0

Out[11]= 0
```

```
In[12]:= Catch[NestList[1/(# + 1) &, -3, 6]]

Out[12]= {-3, -1/2, 2, 1/3, 3/4, 4/7, 7/11}
```

### *1.2.2 Рекурсии*

Важное место в решении многих математических задач занимает реализация рекурсивных алгоритмов. Рекурсия может быть определена таким образом: задача разбивается на много эквивалентных шагов и в процессе выполнения каждого шага программа обращается к самой себе для выполнения последующих шагов.
Основной мотив использования рекурсии состоит в том, что для неё не нужно определять никаких промежуточных переменных, которые управляют потоком вычислений. Классический пример реализации рекурсивного алгоритма – вычисление факториала путем задания функции

```
In[1]:= f[n_] := n f[n - 1]; f[0] = 1; f[1] = 1
In[2]:= f[20]

Out[2]= 2432902008176640000
```

В качестве примера рассмотрим еще одну классическую рекурсивную программу для вычисления целочисленной функции Фибоначчи:

```
In[3]:= Fb[n_] := Fb[n - 1] + Fb[n - 2]; Fb[0] = 1; Fb[1] = 1;
        Fb[30]

Out[4]= 1346269
```

Недостаток программы, приведённой в предыдущем примере, в том, что время для вычисления значения функции Фибоначчи растет экспоненциально относительно n. Этого можно избежать, если каждый раз запоминать уже вычисленные значения.

```
In[5]:= Fib[n_Integer] := Fib[n] = Fib[n - 1] + Fib[n - 2];
        Fib[0] = Fib[1] = 1;
```

Сравним время вычисления:

```
In[71]:= {Timing[Fb[80]], Timing[Fib[80]]}

Out[71]= {{8.75 Second, 37889062373143906}, {0. Second, 37889062373143906}}
```

### *1.2.3. Правила преобразований*

Мы уже рассматривали функцию Replace, применение которой по сути эквивалентно применению правила преобразования x–>value и заменяет символы на value
Однако использование правил преобразований в Mathematica является более общим. Правила преобразований можно применять не только к символам, но и к любым выражениям Mathematica
.
Применение правила преобразования заменяет x–>5

```
In[2]:= Sih[x] + Ln[y] + z^5 /. z → 5

Out[2]= 3125 + Ln[y] + Sih[x]
```

Правило преобразований можно также применить к f[x]. Это правило не затрагивает f[y] и f[x] .

```
In[4]:= f[x] + f[y] + f[3 z] /. f[x] -> a

Out[4]= a + f[y] + f[3 z]
```

f[t_] это шаблон, который заменяет любой аргумент в f.

```
In[7]:= 1 + f[x] + f[y] /. f[t_] -> t^2

Out[7]= 1 + x^2 + y^2
```

Вообще наиболее мощный аспект правил преобразований в Mathematica состоит в том, что правила преобразований можно применять не только к символьным выражениям, но и к шаблонам. Таким образом, правило преобразования для f[t_] указывает, что должна быть преобразована функция f с любым аргументом. Когда мы формируем функцию, например, f[t_] := t^2, мы указываем Mathematica применять правило преобразования f[t_] -> t^2 везде, где возможно.
Мы можем применять правило преобразования для выражения в любой форме

```
In[9]:= f[a b] + f[c d] /. f[x_ y_] -> f[x] + f[y]

Out[9]= f[a] + f[b] + f[c] + f[d]
```

Применим правило преобразования для x^p_.

```
In[10]:= 1 + x^2 + x^4 /. x^p_ -> f[p]

Out[10]= 1 + f[2] + f[4]
```

Основную идею программирования, основанного на правилах преобразований, можно сформулировать следующим образом: всякий раз, когда Mathematica встречает выражение, которое подходит под некоторый образец, она должна преобразовать его надлежащим образом. Внутренний код Mathematica для применения правил преобразований очень сильно оптимизирован, и в большинстве случаев программа, основанная на правилах преобразований, будет работать значительно быстрее, чем соответствующая процедурная программа, использующая условные операторы.

Два типа глобальных присвоений (подстановок)

| | |
|---|---|
| expr1 = expr2 (непосредственное присвоение) | выражение expr2 вычисляется в момент присвоения, и значение этого вычисления закрепляется за expr1 |
| expr1 := expr2 (отложенное присвоение) | expr2 вычисляется заново каждый раз, когда запрашивается значение expt1 (это удобно, когда некоторые параметры выражения expr1 могут быть изменены в течение сеанса.) |
| $expr_1$–> expr2 (непосредственная локальная подстановка) | expr2 вычисляется в момент задания правила, но само правило применяется локально, т. е. только внутри выражения F при помощи команды ReplaceAll: ReplaceAll[F, expr1 —> expr2] (или F/. expr1 — > expr2 ) |
| $expr_1$ :> expr2 (отложенная локальная подстановка) | expr2 вычисляется в момент применения правила, и это правило применяется локально с помощью той же команды ReplaceAll |

Правило локальной подстановки можно именовать, а применяя его, использовать это имя:

rule = expr1 —> $expr_2$ или rule = expr1 :> expr2 ReplaceAll[F, rule] (или F/. rule )
Действие присвоения можно ограничивать условиями с помощью функции Condition (краткая форма /; ), как в следующем примере:
abs[x_] := x /; x>=0
abs[x_] := -x /; x<0

### *1.2.4. Процедурное программирование*

В основе процедурного программирования лежит понятие процедуры и типовых средств управления циклов, условных и безусловных выражений и т.д. Процедурный подход – самый распространенный в программировании и Mathematica обеспечивает его полную поддержку. Однако программирование системы Mathematica и в этом случае остается функциональным, поскольку элементы процедурного программирования существуют в конечном счете в виде функций.

Процедура – это последовательность выражений Mathematica , заданных подряд (в одной строке) и разделенных точкой с запятой.

```
In[2]:= t = (2 x + 3) ^3; t = Expand[t]; t - 27

Out[2]= 54 x + 36 x^2 + 8 x^3
```

Здесь в теле процедуры символ t используется как вспомогательная переменная.
При задании процедур иногда приходится использовать «временные» переменные для хранения значений промежуточных результатов. Встроенная конструкция Block предназначена для того, чтобы «локализовать» переменные, т.е. делает их независимыми, даже если они обозначены одинаково.

| | |
|---|---|
| Block[{x,y,…},proc] | объявляет, что переменные списка {x,y,..},будут локальными в процедуре proc |
| Block[{$x=x_0$,$y=y_0$,…},proc] | задает локальные переменные в процедуре с присвоением им начальных значений |

```
In[2]:= g[x_] := Block[{a}, a = (2 x + 3) ^3; a = Expand[a]];
        {g[m + n], a, g[6]} // ColumnForm

Out[2]= 27 + 54 m + 36 m^2 + 8 m^3 + 54 n + 72 m n + 24 m^2 n + 36 n^2 + 24 m n^2 + 8 n^3
        a
        3375
```

В силу локальности переменной a внутри блока ее значение, которое она имела во время вычисления g , исчезло, когда вычисление закончилось.
Часто возникает необходимость в осуществлении циклических алгоритмов:

| | |
|---|---|
| Do[expr,{n}] | вычисляет expr n раз |
| Do[expr,{i,$i_{max}$}] | вычисляет expr с переменной i, принимающей с шагом 1 значения от1 до $i_{max}$ |
| Do[expr,{i, $i_{min}$, $i_{max}$}] | вычисляет expr с переменной i, принимающей с шагом 1 значения от $i_{min}$ до $i_{max}$ |
| Do[expr,{i, $i_{min}$, $i_{max}$, di}] | то же самое с шагом di |
| Do[expr, {i, $i_{min}$, $i_{max}$},{j, $j_{min}$, $j_{max}$}] | организует ряд вложенных циклов |
| While[test,expr] | вычисляет expr повторно до тех пор, пока test принимает значение True |
| For [start,test,step,expr] | вычисляет start, потом, повторяясь, вычисляет step и expr, пока test не примет значение False |

Приведем функцию умножения перестановок y и x с помощью цикла Do:

```
In[3]:= Cmps[x_, y_, n_] :=
          (t = Table[0, {n}]; Do[t[[i]] = x[[y[[i]]]], {i, 1, n}]; t)
In[4]:= Cmps[{1, 2, 3, 5, 4}, {5, 4, 2, 3, 1}, 5]

Out[4]= {4, 5, 2, 3, 1}
```

А это пример условного цикла:

```
In[5]:= For[i = 1; t = x, i < 4, i = i + 1, t = i t; Print[i + Sqrt[t]]]
```

$1 + \sqrt{x}$

$2 + \sqrt{2}\ \sqrt{x}$

$3 + \sqrt{6}\ \sqrt{x}$

Переменные цикла в операторе For не локализованы, поэтому выполнение предыдущей программы имеет побочный эффект: переменным i и t присвоены значения 3 и 5 соответственно. Mathematica обеспечена функциинй Module, позволяющей локализовать «временные» переменные, во избежание возможного конфликта символов.

| | |
|---|---|
| Module[{x,y,…}, expr] | локализует переменные x,y,…при выполнении exp |
| Module[{$x=x_0$,$y=y_0$,…}, expr] | То же, с присвоением начальных значений локальным переменным |

Присвоим переменной t глобальное значение 15.

```
In[6]:= t = 15

Out[6]= 15
```

Локализуем переменную t:

```
In[7]:= Module[{t}, t = 8; t]

Out[7]= 8
```

Вычислим t:

```
In[8]:= t

Out[8]= 15
```

Модули могут быть использованы в более общем случае, например, если нужно установить «временные» переменные внутри конструируемой функции:

```
In[9]:= f[v_] := Module[{t}, t = (1 + v) ^2; t = Expand[t]]

In[10]:= f[a]

Out[10]= 1 + 2 a + a^2

In[11]:= t

Out[11]= 15
```

При построении процедур часто бывает нужно, чтобы определенный набор операций выполнялся только при соблюдении некоторых условий. Приведем список условных операторов.

| | |
|---|---|
| If[test, t, f] | выполняет t, если test дает True, и f, если test дает False |
| If[test, t, f ,u] | полная форма If, включая выполнение u, если test не дает ни True , ни False |
| Which[$test_1$, $value_1$, $test_2$, $value_{2,...}$] | вычисляет каждый $test_i$ по очереди и возвращает в качестве результата $value_i$ , где i – номер первого теста, принимающего значение True |
| Switch[expr, $form_1$, $value_1$,…] | сравнивает выражение expr с $form_1$, $form_{2,,...}$и выдает значение $value_i$ , где i- номер первой из $form_i$, которой походит expr |
| Switch[expr, $form_1$, $value_1$,…,def] | использует def как значение по умолчанию |

Вывод этой функции зависит от типа вводимых аргументов:

```
In[12]:= f[x_] := Switch[x, _Integer, FactorInteger[x], _Rational, N[x],
           _Real, Rationalize[x], _Complex, Im[x], _Symbol, 1 + x^2]

In[13]:= Map[f, {4, 2.3, 1/3, π, 5 + 5 i}]

Out[13]= {{{2, 2}}, 23/10, 0.333333, 1 + π^2, {{5, 2}}}
```

Отметим еще две полезные функции, с помощью которых любое выражение можно преобразовать в предикат, принимающий значения True или False:

| | |
|---|---|
| TrueQ[expr] | возвращает True, если expr True и False в противном случае |
| SameQ[$expr_1$, $expr_2$] | возвращает True, если $expr_1$ тождественно равно $expr_2$ |

# Глава 2. Перестановки и комбинации

Перестановки и подмножества – основные комбинаторные объекты. Стандартный пакет расширения DiscreteMath`Combinatorica` обеспечивает нас встроенными функциями для конструирования объектов как определенных, так и случайных, упорядочивая и не упорядочивая их, вычисляет их инварианты. Приведем примеры некоторых из этих функций в действии.

## ■2.1 Конструирование перестановок

### *2.1.1 Порождение перестановок*

Пусть Q – конечное множество из n элементов. Поскольку природа его элементов для нас несущественна, удобно считать, что Q={1,2,....n}. Группа S(Q) всех взаимно-однозначных отображений Q->Q называется **симметрической группой** степени n и обозначается S(n).Элементы этой группы называются **перестановками**.

Встроенная функция Permutations[l] конструирует список всех возможных перестановок элементов списка l.

```
In[2]:= Permutations[{3, m, 1, 4}]
Out[2]= {{3, m, 1, 4}, {3, m, 4, 1}, {3, 1, m, 4}, {3, 1, 4, m},
         {3, 4, m, 1}, {3, 4, 1, m}, {m, 3, 1, 4}, {m, 3, 4, 1},
         {m, 1, 3, 4}, {m, 1, 4, 3}, {m, 4, 3, 1}, {m, 4, 1, 3},
         {1, 3, m, 4}, {1, 3, 4, m}, {1, m, 3, 4}, {1, m, 4, 3},
         {1, 4, 3, m}, {1, 4, m, 3}, {4, 3, m, 1}, {4, 3, 1, m},
         {4, m, 3, 1}, {4, m, 1, 3}, {4, 1, 3, m}, {4, 1, m, 3}}
```

Тот же результат можно получить, если применить функцию DistinctPermutations[l], которая возвращает все перестановки множества l, где элементы l – суть подмножества.

```
In[3]:= DistinctPermutations[{3, m, 1, 4}]
Out[3]= {{1, 3, 4, m}, {1, 3, m, 4}, {1, 4, 3, m}, {1, 4, m, 3},
         {1, m, 3, 4}, {1, m, 4, 3}, {3, 1, 4, m}, {3, 1, m, 4},
         {3, 4, 1, m}, {3, 4, m, 1}, {3, m, 1, 4}, {3, m, 4, 1},
         {4, 1, 3, m}, {4, 1, m, 3}, {4, 3, 1, m}, {4, 3, m, 1},
         {4, m, 1, 3}, {4, m, 3, 1}, {m, 1, 3, 4}, {m, 1, 4, 3},
         {m, 3, 1, 4}, {m, 3, 4, 1}, {m, 4, 1, 3}, {m, 4, 3, 1}}
```

Рассмотрим всевозможные перестановки заданных множеств:

```
In[4]:= DistinctPermutations[{{3, 2}, {1, 4}, {a, b}}]
Out[4]= {{{1, 4}, {3, 2}, {a, b}}, {{1, 4}, {a, b}, {3, 2}},
         {{3, 2}, {1, 4}, {a, b}}, {{3, 2}, {a, b}, {1, 4}},
         {{a, b}, {1, 4}, {3, 2}}, {{a, b}, {3, 2}, {1, 4}}}
```

Проверить, является ли данная множество перестановкой? можно с помощью теста PermutationQ[p].

```
In[5]:= {PermutationQ[{2, 3, 1}], PermutationQ[{2, 3, 1, y}]}
Out[5]= {True, False}
```

LexicographicPermutations[l] конструирует все перестановки списка l в лексикографическом порядке

```
In[6]:= LexicographicPermutations[{3, 2, 1, 4}]
Out[6]= {{3, 2, 1, 4}, {3, 2, 4, 1}, {3, 1, 2, 4}, {3, 1, 4, 2},
         {3, 4, 2, 1}, {3, 4, 1, 2}, {2, 3, 1, 4}, {2, 3, 4, 1},
         {2, 1, 3, 4}, {2, 1, 4, 3}, {2, 4, 3, 1}, {2, 4, 1, 3},
         {1, 3, 2, 4}, {1, 3, 4, 2}, {1, 2, 3, 4}, {1, 2, 4, 3},
         {1, 4, 3, 2}, {1, 4, 2, 3}, {4, 3, 2, 1}, {4, 3, 1, 2},
         {4, 2, 3, 1}, {4, 2, 1, 3}, {4, 1, 3, 2}, {4, 1, 2, 3}}
```

Функция NextPermutation[p] возвращает перестановку, следующую за перестановкой p размера n в LexicographicPermutations[n]

```
In[7]:= NextPermutation[{3, 4, 1, 2}]
Out[7]= {3, 4, 2, 1}
```

IdentityPermutation[n] возвращает тождественную перестановку размера n.

```
In[8]:= IdentityPermutation[10]
Out[8]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

RandomPermutation[n] порождает случайную перестановку первых n натуральных чисел.

```
In[9]:= RandomPermutation[10]
Out[9]= {2, 5, 10, 6, 8, 4, 9, 1, 7, 3}
```

Подсчитаем, сколько раз встречается тождественная перестановка среди двухсот случайных перестановок:

```
In[10]:= Count[s = Table[RandomPermutation[3], {200}],
          IdentityPermutation[3]]
Out[10]= 31
```

Подсчитаем частоту появления каждой перестановки из множества Permutations[3] в множестве s.

```
In[11]:= Distribution[s, Permutations[3]]
Out[11]= {31, 30, 34, 30, 42, 33}
```

Функция Permute[l, p] переставляет элементы списка l в соответствии с перестановкой p.

```
In[12]:= Permute[{a, b, c, d}, Permutations[4]]
Out[12]= {{a, b, c, d}, {a, b, d, c}, {a, c, b, d}, {a, c, d, b},
          {a, d, b, c}, {a, d, c, b}, {b, a, c, d}, {b, a, d, c},
          {b, c, a, d}, {b, c, d, a}, {b, d, a, c}, {b, d, c, a},
          {c, a, b, d}, {c, a, d, b}, {c, b, a, d}, {c, b, d, a},
          {c, d, a, b}, {c, d, b, a}, {d, a, b, c}, {d, a, c, b},
          {d, b, a, c}, {d, b, c, a}, {d, c, a, b}, {d, c, b, a}}
```

Переставить элементы перестановки можно, также используя функцию SelectionSort.
SelectionSort[l, f] упорядочивает элементы списка l согласно упорядочивающей функции f.

```
In[13]:= SelectionSort[{4, 3, 2, 5, 6, 1}, (#1^2 < #2^2 - 4) &]
Out[13]= {2, 1, 3, 4, 5, 6}
```

Функция Permute также используется для умножения перестановок.

```
In[14]:= Permute[{1, 3, 2, 4, 5}, {5, 4, 3, 2, 1}]
Out[14]= {5, 4, 2, 3, 1}
```

Введем операцию умножения другим образом:

```
In[15]:= Cmps[x_, y_, n_] :=
          (t = Table[0, {n}]; Do[t[[i]] = x[[y[[i]]]], {i, 1, n}]; t)
```

Перемножим две случайные перестановки размера 200:

```
In[16]:= SameQ[Cmps[a = RandomPermutation[200],
           b = RandomPermutation[200], 200], Permute[a, b]]
Out[16]= True
```

Умножение любой перестановки на тождественную перестановку есть тождественный оператор:

```
In[17]:= Permute[p = RandomPermutation[200],
           IdentityPermutation[200]] == p
Out[17]= True
```

Относительно операции умножения множество перестановок данного размера n образует группу, поэтому для любой перестановки существует обратная перестановка относительно произведения перестановок.
Функция InversePermutation[p] возвращает обратную перестановку для перестановки p относительно операции умножения перестановок.

```
In[18]:= a = RandomPermutation[100]; b = InversePermutation[a];
          SameQ[Permute[a, b], IdentityPermutation[100]]
Out[18]= True
```

Произведение перестановок некоммутативно, но ассоциативно

```
In[19]:= a = RandomPermutation[1000]; b = RandomPermutation[1000];
         c = RandomPermutation[1000];
         {Permute[a, b] == Permute[b, a],
          Permute[a, Permute[b, c]] == Permute[Permute[a, b], c]}
Out[19]= {False, True}
```

| | |
|---|---|
| Permutations[l] | конструирует список всех возможных перестановок элементов списка l |
| LexicographicPermutations[l] | конструирует все перестановки списка l в лексикографическом порядке |
| DistinctPermutations[l] | возвращает все перестановки множества l, где элементы l-суть подмножества |
| PermutationQ[p] | возвращаетTrue, если список p является перестановкой |
| NextPermutation[p] | возвращает перестановку, следующую за перестановкой p размера n в LexicographicPermutations[n] |
| IdentityPermutation[n] | возвращает тождественную перестановку размера n |
| RandomPermutation[n] | порождает случайную перестановку первых n натуральных |

| | |
|---|---|
| | чисел |
| Permute[l, p] | переставляет элементы списка l в соответствии с перестановкой p. Если l-перестановка, то Permute[l, p]- произведение перестановок |
| SelectionSort[l, f] | упорядочивает элементы списка l, согласно упорядочивающей функции f |
| InversePermutation[p] | возвращает обратную перестановку для перестановки p относительно операции умножения перестановок |

### *2.1.1. Ранг перестановки*

**Рангом** перестановки p размера n называется количество предшествующих p перестановок в упорядоченном множестве LexicographicPermutations[n].

Функция RankPermutation[p] возвращает ранг перестановки p. Функция UnrankPermutation[r, n] возвращает r-ю перестановку в лексикографически упорядоченном множестве перестановок размера n.

Оказывается, перестановке {5,4,2,1,3} предшествуют 116 из 120 перестановок размера 5.

```
In[2]:= RankPermutation[{5, 4, 2, 1, 3}]
Out[2]= 116
```

Покажем, что функция Permutations[n] возвращает перестановки в лексикографическом порядке:

```
In[3]:= Map[RankPermutation, Permutations[5]]
Out[3]= {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
         17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
         31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44,
         45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58,
         59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71,
         72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
         85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97,
         98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108,
         109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119}
```

Получим множество всех перестановок, упорядоченных в лексикографическом порядке с помощью UnrankPermutation, примененной n! раз

```
In[4]:= Table[UnrankPermutation[i, Range[4]], {i, 0, 23}]
Out[4]= {{1, 2, 3, 4}, {1, 2, 4, 3}, {1, 3, 2, 4}, {1, 3, 4, 2},
         {1, 4, 2, 3}, {1, 4, 3, 2}, {2, 1, 3, 4}, {2, 1, 4, 3},
         {2, 3, 1, 4}, {2, 3, 4, 1}, {2, 4, 1, 3}, {2, 4, 3, 1},
         {3, 1, 2, 4}, {3, 1, 4, 2}, {3, 2, 1, 4}, {3, 2, 4, 1},
         {3, 4, 1, 2}, {3, 4, 2, 1}, {4, 1, 2, 3}, {4, 1, 3, 2},
         {4, 2, 1, 3}, {4, 2, 3, 1}, {4, 3, 1, 2}, {4, 3, 2, 1}}
```

| | |
|---|---|
| RankPermutation[p] | возвращает ранг перестановки p |
| UnrankPermutation[r, n] | возвращает r-ю перестановку в лексикографически упорядоченном множестве перестановок размера n |

## ■ 2.2. Инверсии

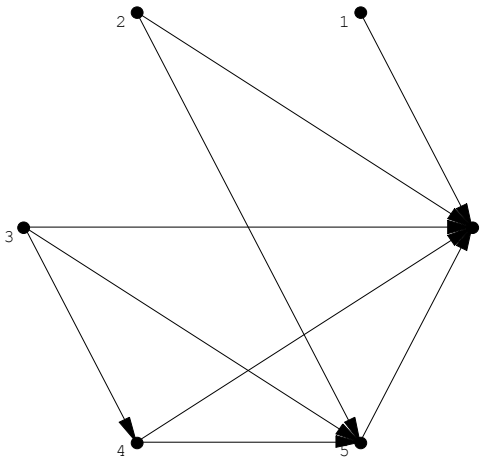
### *2.2.1. Вектор инверсий*

Пусть Q={1,2,...n}, Q×Q – декартов квадрат. Пара (i,j) называется **инверсией** перестановки pϵS(n), если i<j, но p(i)>p(j).
Для любой n-перестановки p следующим образом определяется вектор инверсий v: для любого i, 1≤i≤n-1, i-й элемент вектора v есть число элементов p, больших, чем i, и лежащих левее i. Очевидно, что вектор инверсий содержит ровно n-1 элементов. Известно, что нет двух различных перестановок, имеющих один и тот же вектор инверсий. Следовательно, перестановку однозначно можно задать непосредственно вектором инверсий.
Функция ToInversionVector[p] выдает вектор инверсий перестановки p, а FromInversionVector[v] однозначно восстанавливает перестановку с данным вектором инверсий v по вектору v.

```
In[2]:= {p = RandomPermutation[10], ToInversionVector[p],
          FromInversionVector[ToInversionVector[p]]} //
        ColumnForm
Out[2]= {5, 2, 1, 3, 9, 7, 4, 6, 8, 10}
        {2, 1, 1, 3, 0, 2, 1, 1, 0}
        {5, 2, 1, 3, 9, 7, 4, 6, 8, 10}
```

Другой метод выделения инверсий в перестановке использует граф перестановки. Граф перестановки p – это граф, множество вершин которого есть множество {1,2,...,n}, а вершины {i,j} соединены ребром тогда и только тогда, когда {i,j} – инверсия перестановки p. Такие графы называются графами Пойа перестановки.

```
In[3]:= ShowLabeledGraph[g = PermutationGraph[{6, 1, 5, 2, 4, 3}],
          EdgeDirection → True]
```




В вышеприведенном графе шестая вершина соединена со всеми остальными вершинами, значит, 6 образует пять инверсий и т.д., а число инверсий перестановки равно числу ребер графа перестановки

```
In[4]:= {M[g], Inversions[{6, 1, 5, 2, 4, 3}]}
Out[4]= {9, 9}
```

Функция Inversions[p] подсчитывает число инверсий перестановки p.
Число инверсий в перестановке равно числу инверсий в обратной перестановке:

```
In[5]:= p = RandomPermutation[100];
        {Inversions[p], Inversions[InversePermutation[p]]}
Out[5]= {2628, 2628}
```

Число инверсий в перестановке размера n находится в пределах от 0 до $C_n^2$, причем наибольшее число инверсий в перестановке, полученной перечислением тождественной перестановки в обратном порядке:

```
In[6]:= {Inversions[Reverse[Range[200]]], Binomial[200, 2]}
Out[6]= {19900, 19900}
```

Перестановка называется **четной,** если число инверсий в ней четное число, и **нечетной** в противном случае. Число $(-1)^k$, где k – число инверсий перестановки называется **сигнатурой** перестановки.
Функция SignaturePermutation[p] возвращает сигнатуру перестановки.
Рассмотрим сигнатуру всех двадцати четырех перестановок размера 4.

```
In[7]:= l = Map[SignaturePermutation, Permutations[4]]
Out[7]= {1, -1, -1, 1, 1, -1, -1, 1, 1, -1,
          -1, 1, 1, -1, -1, 1, 1, -1, -1, 1, 1, -1, -1, 1}
```

Количество четных перестановок в Permutations[4] должно быть равно числу нечетных:

```
In[8]:= Count[l, 1] == Count[l, -1]
Out[8]= True
```

NumberOfPermutationsByInversions[n, k] возвращает число перестановок длины n с ровно k инверсиями.
NumberOfPermutationsByInversions[n] возвращает таблицу числа перестановок длины n с k инверсиями для всех k.
Подсчитаем число инверсий в каждой из 24-х перестановок размера 4:

```
In[9]:= s = Map[Inversions, Permutations[4]]
Out[9]= {0, 1, 1, 2, 2, 3, 1, 2, 2, 3, 3, 4, 2, 3, 3, 4, 4, 5, 3, 4, 4, 5, 5, 6}
```

Теперь подсчитаем в списке s количество перестановок с i инверсиями (i=1,…,6)

```
In[10]:= Table[Count[s, i], {i, 0, 6}]
Out[10]= {1, 3, 5, 6, 5, 3, 1}
```

С другой стороны подсчитаем количество перестановок с i (i=1,…,6) инверсиями с помощью встроенных функций:

```
In[11]:= NumberOfPermutationsByInversions[4]
Out[11]= {1, 3, 5, 6, 5, 3, 1}
```

Подсчитаем число перестановок размера 100 с 45 инверсиями:

```
In[12]:= NumberOfPermutationsByInversions[100, 45]
Out[12]= 295361905824809968689323674648550318 70
```

Подсчитаем, сколько времени тратит Mathematica на эту операцию:

```
In[13]:= Timing[NumberOfPermutationsByInversions[100, 45] ;]
Out[13]= {87.094 Second, Null}
```

| | |
|---|---|
| ToInversionVector[p] | возвращает вектор инверсий перестановки p |
| FromInversionVector[v] | однозначно восстанавливает перестановку с данным вектором инверсий v по вектору v |
| PermutationGraph[p] | конструирует граф Пойа перестановки p |
| Inversions[p] | подсчитывает число инверсий перестановки p |
| SignaturePermutation[p] | возвращает сигнатуру перестанови |
| NumberOfPermutationsByInversions[n, k] | возвращает число перестановок длины n с ровно к инверсиями |
| NumberOfPermutationsByInversions[n] | возвращает таблицу числа перестановок длины n с к инверсиями для всех к |

### *2.2.2. Индекс перестановки*

**Индексом** перестановки p называется сумма всех j, таких, что p[j]>p[j+1].
Функция Index[p] возвращает индекс перестановки p.
Рассмотрим перестановки размера 4 с индексом 3

```
In[14]:= a = Select[Permutations[5], (Index[#] == 4) &]
Out[14]= {{1, 2, 3, 5, 4}, {1, 2, 4, 5, 3}, {1, 3, 4, 5, 2}, {2, 1, 4, 3, 5}, {2, 1, 5, 3, 4},
          {2, 3, 4, 5, 1}, {3, 1, 4, 2, 5}, {3, 1, 5, 2, 4}, {3, 2, 4, 1, 5}, {3, 2, 5, 1, 4},
          {4, 1, 3, 2, 5}, {4, 1, 5, 2, 3}, {4, 2, 3, 1, 5}, {4, 2, 5, 1, 3}, {4, 3, 5, 1, 2},
          {5, 1, 3, 2, 4}, {5, 1, 4, 2, 3}, {5, 2, 3, 1, 4}, {5, 2, 4, 1, 3}, {5, 3, 4, 1, 2}}
```

Убедимся, что действительно индекс каждой перестановки в списке a равен 4.

```
In[15]:= Map[Index, a]
Out[15]= {4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4}
```

| | |
|---|---|
| Index[p] | возвращает индекс перестановки p |

### *2.2.3. Разбиения перестановок и эйлеровы числа*

Читая любую перестановку слева направо, можно разбить перестановку на множество возрастающих соприкасающихся последовательностей элементов перестановки, которые называются **пробегами.** Функция Runs[p] возвращает пробеги перестановки p.

```
In[2]:= Runs[{2, 3, 1, 6, 4, 8, 7, 5}]
Out[2]= {{2, 3}, {1, 6}, {4, 8}, {7}, {5}}
```

Очевидно, что сумма пробегов в перестановке из n символов и числа пробегов в перестановке, перечисленной в обратном порядке, равна n+1.

```
In[3]:= Table[Length[Runs[a = RandomPermutation[n]]] +
          Length[Runs[Reverse[a]]], {n, 100, 150}]
```

```
Out[3]= {101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112,
         113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125,
         126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138,
         139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151}
```

Функция Eulerian[n, k] выдает число перестановок длины n с к+1 пробегами.
Рассмотрим список перестановок из 15 символов с i возрастающими соприкасающимися подпоследовательностями ($1 \le i \le 14$). Симметрия этого списка указывает, что Eulerian[n,k]=Eulerian[n,n-1-k].

```
In[4]:= Table[{Eulerian[15, i], Eulerian[15, 14 - i]}, {i, 14}]
Out[4]= {{32752, 32752}, {13824739, 13824739}, {848090912, 848090912},
         {15041229521, 15041229521}, {102776998928, 102776998928},
         {311387598411, 311387598411}, {447538817472, 447538817472},
         {311387598411, 311387598411}, {102776998928, 102776998928},
         {15041229521, 15041229521}, {848090912, 848090912},
         {13824739, 13824739}, {32752, 32752}, {1, 1}}
```

Если перестановка имеет k+1 пробегов, то перестановка, перечисленная в обратном порядке, имеет n-k пробегов.

```
In[5]:= {Length[Runs[p = RandomPermutation[100]]], Length[Runs[Reverse[p]]]}
Out[5]= {51, 50}
```

LongestIncreasingSubsequence[p] находит наибольшую возрастающую подпоследовательность в перестановке p.
Выделим случайную перестановку размера 20, наибольшую возрастающую подпоследовательность в этой перестановке, и позиции элементов этой подпоследовательности в исходной перестановке.

```
In[6]:= {p = RandomPermutation[20], l = LongestIncreasingSubsequence[p],
          Flatten[Table[Position[p, l[[i]]], {i, 1, Length[l]}]]} //
         ColumnForm
Out[6]= {4, 15, 7, 12, 18, 16, 17, 2, 20, 1, 10, 9, 13, 11, 3, 19, 8, 14, 5, 6}
        {4, 7, 12, 16, 17, 19}
        {1, 3, 4, 6, 7, 16}
```

Определить, является ли данный элемент элементом списка и найти позицию этого элемента в списке, можно также с помощью функции BinarySearch.
Функция BinarySearch[l,k] выдает позицию списка l, содержащую к, если k присутствует в l. Если k отсутствует в списке l, функция возвращает (p+1/2), где k находится в позиции между p и p+1.
BinarySearch[l,k,f] выдает позицию k в списке, полученном из l применением f к каждому элементу в l.
Рассмотрим случайную перестановку размера 100, выделим в ней наибольшую возрастающую подпоследовательность l и выясним, являются ли 3 и 50 элементами l.

```
In[8]:= p = RandomPermutation[100];
        {l = LongestIncreasingSubsequence[p], BinarySearch[l, 3],
         BinarySearch[l, 50]}
```

$$\text{Out[8]= } \left\{\{1, 11, 14, 25, 32, 33, 37, 48, 50, 60, 67, 79, 84, 93\}, \frac{3}{2}, 9\right\}$$

Получили, что список l не содержит 3, а число 50 содержится в l на девятой позиции .
Теперь покажем, что 25 стоит на четвертой позиции списка, элементами которого являются корни квадратные из элементов l.

```
In[9]:= BinarySearch[l, 5, Sqrt]
Out[9]= 4
```

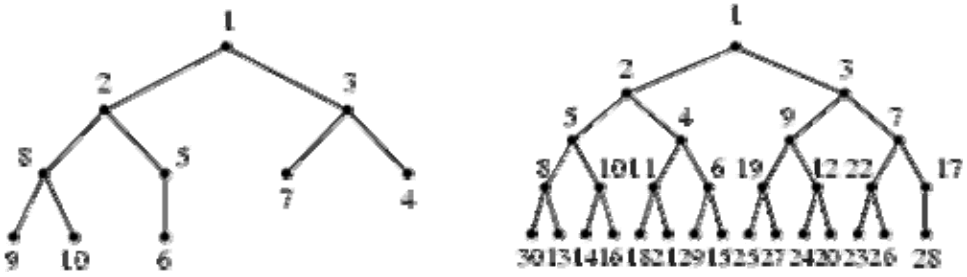
Функция Element[x, dom] проверяет, является ли x элементом области dom.
Element[{x1, x2, ... }, dom] проверяет, являются ли x1,x2,… элементами области dom.
Element[patt, dom] проверяет, является ли выражение, удовлетворяющее шаблону patt элементом области dom.

```
In[10]:= Element[{6, 3.5}, Integers]
Out[10]= False
```

Функция EncroachingListSet [p] возвращает упорядоченное множество упорядоченных подмножеств перестановки p, причем каждый последующий список должен быть вложен в предыдущий. Рассмотрим все такие списки на 5- перестановках. Их ровно 9, и в каждом из них последующий список находится в пределах предыдущего:

```
In[11]:= Union[Map[EncroachingListSet, Permutations[5]]]
Out[11]= {{{1, 2, 3, 4, 5}}, {{1, 5}, {2, 3, 4}}, {{1, 2, 5}, {3, 4}},
          {{1, 3, 5}, {2, 4}}, {{1, 4, 5}, {2, 3}}, {{1, 2, 3, 5}, {4}},
          {{1, 2, 4, 5}, {3}}, {{1, 3, 4, 5}, {2}}, {{1, 5}, {2, 4}, {3}}}
```

Последовательность $\{a_n\}_{n=1}^{N}$ называется heap, если оно удовлетворяет условию $a_{[j/2]} \le a_j$ для $2 \le j \le N$, где [x]- целая часть x. Heap можно представить как помеченное бинарное дерево, в котором метка i-ой вершины меньше, чем метка ее потомка. На нижнем рисунке слева задан heap {1,2,3,8,5,7,4,9,10,6}, а справа показан heap на тридцати элементах.




Heap может быть сконструирован из перестановки p с помощью функции Heapify.
Heapify[p] конструирует heap из перестановки p.
Рассмотрим все heaps, которые можно сконструировать из перестановок размера i, где i=1,2,3,4:

```
In[12]:= Table[Union[Map[Heapify, Permutations[i]]], {i, 4}] // ColumnForm
Out[12]= {{1}}
         {{1, 2}}
         {{1, 2, 3}, {1, 3, 2}}
         {{1, 2, 3, 4}, {1, 2, 4, 3}, {1, 3, 2, 4}}
```

Функция RandomHeap[n] конструирует случайный heap на n элементах.
Рассмотрим случайные heaps на n элементах, n=1,2,….,10:

```
In[13]:= Table[RandomHeap[n], {n, 1, 10}]
```

```
Out[13]= {{1}, {1, 2}, {1, 3, 2}, {1, 2, 3, 4}, {1, 2, 4, 3, 5},
          {1, 2, 3, 5, 4, 6}, {1, 2, 3, 5, 6, 4, 7}, {1, 3, 2, 4, 8, 5, 7, 6},
          {1, 3, 2, 6, 5, 8, 4, 7, 9}, {1, 3, 2, 6, 4, 10, 5, 9, 7, 8}}
```

Функция HeapSort[l] производит heap сортировку списка l.

```
In[14]:= Map[HeapSort, Table[RandomPermutation[n], {n, 1, 10}]] //
         ColumnForm
Out[14]= {1}
         {1, 2}
         {1, 2, 3}
         {1, 2, 3, 4}
         {1, 2, 3, 4, 5}
         {1, 2, 3, 4, 5, 6}
         {1, 2, 3, 4, 5, 6, 7}
         {1, 2, 3, 4, 5, 6, 7, 8}
         {1, 2, 3, 4, 5, 6, 7, 8, 9}
         {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

Введем функцию, которая тестирует перестановку p, является ли p heap:

```
In[15]:= HeapQ[a_] :=
         And @@ Table[a[[Floor[i / 2]]] < a[[i]], {i, 2, Length[a]}] ;

In[16]:= {HeapQ[{1, 2, 3, 8, 5, 7, 4, 9, 10, 6}],
          HeapQ[{6, 2, 7, 9, 5, 3, 4, 8, 10, 1}]}
Out[16]= {True, False}
```

Подсчитаем число heaps на n элементах, n=1,2,…,10

```
In[17]:= Table[Count[Map[HeapQ, Permutations[i]], True], {i, 1, 10}]
Out[17]= {1, 1, 2, 3, 8, 20, 80, 210, 896, 3360}
```

| | |
|---|---|
| Runs[p] | разбивает перестановку p на соприкасающиеся возрастающие подпоследовательности |
| Eulerian[n, k] | выдает число перестановок длины n с к+1 пробегами |
| LongestIncreasingSubsequence[p] | находит наибольшую возрастающую дискретную подпоследовательность в перестановке p |
| BinarySearch[l,k] | возвращает позицию k в списке l, если k присутствует в l. Если k отсутствует в списке l, функция возвращает (p+1/2), где k лежит в позиции между p и p+1. |
| BinarySearch[l,k,f] | возвращает позицию k в списке, полученном из l применением f к каждому элементу в l. |
| Element[x, dom] | проверяет, является ли x элементом области dom |
| Element[{x1, x2, ... }, dom] | проверяет, являются ли x1,x2,… элементами области dom |
| Element[patt, dom] | проверяет, является ли выражение, удовлетворяющее шаблону patt элементом области dom |
| EncroachingListSet [p] | возвращает упорядоченное множество упорядоченных подмножеств перестановки p, причем каждый последующий список должен быть вложен в предыдущий |
| Heapify[p] | конструирует heap из перестановки p. Heap можно представить как помеченное бинарное дерево, в котором метка i-й вершины меньше, чем метка ее потомка. |

| | |
|---|---|
| RandomHeap[n] | конструирует случайный heap на n элементах |
| HeapSort[l] | производит heap сортировку списка l |

## ■ 2.3. Циклическая структура перестановок

### *2.3.1 Разложение перестановки на циклы*

**Циклом** называется такая перестановка, что при повторении ее достаточное число раз, всякий из действительно перемещаемых ею символов может быть переведен в любой другой из этих символов. Два цикла называются **независимыми,** если они не имеют общих действительно перемещаемых символов. Известно, что всякая перестановка может быть единственным образом разложена в произведение попарно независимых циклов.
Функция ToCycles[p] возвращает список циклов, на которые раскладывается перестановка p.

```
In[2]:= ToCycles[{3,5,1,2,4}]

Out[2]= {{3, 1}, {5, 4, 2}}
```

Тождественная перестановка длины n состоит из n циклов единичной длины (или неподвижных точек).

```
In[3]:= ToCycles[IdentityPermutation[20]]

Out[3]= {{1}, {2}, {3}, {4}, {5}, {6}, {7}, {8}, {9}, {10},
         {11}, {12}, {13}, {14}, {15}, {16}, {17}, {18}, {19}, {20}}
```

Функция FromCycles[{c1, c2, ...}] выдает перестановку, которая имеет заданную циклическую структуру {c1, c2, ...}.

```
In[4]:= FromCycles[{{1, 2}, {4, 3, 5}}]
        {2, 1, 5, 3, 4}

Out[4]= {2, 1, 5, 3, 4}
```

Последовательное применение функций ToCycles и FromCycles есть тождественный оператор:

```
In[5]:= SameQ[p = RandomPermutation[100], FromCycles[ToCycles[p]]]
Out[5]= True
```

Однако, переходя от циклической структуры перестановки к самой перестановке и обратно, мы можем получить другую запись перестановки:

```
In[6]:= ToCycles[FromCycles[{{1, 2}, {4, 3, 5}}]]
Out[6]= {{2, 1}, {5, 4, 3}}
```

Скобки в циклическом представлении перестановок оказываются необязательными, т.к. циклы могут быть однозначно представлены и без них. Чтобы убрать скобки, переставим элементы каждого цикла так, чтобы минимальный элемент был первым и затем расставим циклы в порядке убывания относительно минимального (т.е. первого) элемента цикла. Такая запись задает нам **каноническую** циклическую структуру представления перестановки.
Функция HideCycles [c] берет циклическую структуру с перестановки, конвертирует ее в каноническую форму и отбрасывает скобки, тем самым превращая ее в перестановку.
Уберем скобки в циклическом разложении всех двадцати четырех перестановок размера 4.

```
In[7]:= a = Map[HideCycles, Map[ToCycles, Permutations[4]]]

Out[7]= {{4, 3, 2, 1}, {3, 4, 2, 1}, {4, 2, 3, 1}, {2, 3, 4, 1}, {2, 4, 3, 1}, {3, 2, 4, 1},
         {4, 3, 1, 2}, {3, 4, 1, 2}, {4, 1, 2, 3}, {1, 2, 3, 4}, {1, 2, 4, 3}, {3, 1, 2, 4},
         {4, 1, 3, 2}, {1, 3, 4, 2}, {4, 2, 1, 3}, {2, 1, 3, 4}, {2, 4, 1, 3}, {1, 3, 2, 4},
         {1, 4, 3, 2}, {3, 1, 4, 2}, {2, 1, 4, 3}, {3, 2, 1, 4}, {1, 4, 2, 3}, {2, 3, 1, 4}}
```

Теперь, применив к списку a RevealCycles[p], получим каноническую циклическую структуру перестановок списка Permutations[4]:

```
In[8]:= Map[RevealCycles, a]
Out[8]= {{{4}, {3}, {2}, {1}}, {{3, 4}, {2}, {1}}, {{4}, {2, 3}, {1}}, {{2, 3, 4}, {1}},
         {{2, 4, 3}, {1}}, {{3}, {2, 4}, {1}}, {{4}, {3}, {1, 2}}, {{3, 4}, {1, 2}},
         {{4}, {1, 2, 3}}, {{1, 2, 3, 4}}, {{1, 2, 4, 3}}, {{3}, {1, 2, 4}},
         {{4}, {1, 3, 2}}, {{1, 3, 4, 2}}, {{4}, {2}, {1, 3}}, {{2}, {1, 3, 4}},
         {{2, 4}, {1, 3}}, {{1, 3, 2, 4}}, {{1, 4, 3, 2}}, {{3}, {1, 4, 2}},
         {{2}, {1, 4, 3}}, {{3}, {2}, {1, 4}}, {{1, 4, 2, 3}}, {{2, 3}, {1, 4}}}
```

Каждая строка в нижеприведенной матрице есть перестановка размера 3, циклическое разложение этой перестановки, ее каноническое циклическое разложение без скобок и со скобками.

```
In[9]:= l =
          Map[{#, ToCycles[#], HideCycles[ToCycles[#]],
              RevealCycles[HideCycles[ToCycles[#]]]} &, Permutations[3]] // MatrixForm
Out[9]//MatrixForm=
        {1, 2, 3}  {{1}, {2}, {3}}  {3, 2, 1}  {{3}, {2}, {1}}
        {1, 3, 2}   {{1}, {3, 2}}   {2, 3, 1}   {{2, 3}, {1}}
        {2, 1, 3}   {{2, 1}, {3}}   {3, 1, 2}   {{3}, {1, 2}}
        {2, 3, 1}    {{2, 3, 1}}    {1, 2, 3}    {{1, 2, 3}}
        {3, 1, 2}    {{3, 2, 1}}    {1, 3, 2}    {{1, 3, 2}}
        {3, 2, 1}   {{3, 1}, {2}}   {2, 1, 3}   {{2}, {1, 3}}
```

| | |
|---|---|
| ToCycles[p] | возвращает циклическую структуру перестановки p как список циклов |
| FromCycles[{c1, c2, ...}] | выдает перестановку, которая имеет заданную циклическую структуру {c1, c2, ...} |
| HideCycles [c] | берет циклическую структуру с перестановки, конвертирует ее в каноническую форму и отбрасывает скобки |
| RevealCycles[p] | возвращает каноническую структуру перестановки |

### *2.3.2. Тип перестановки*

**Типом** перестановки называется последовательность чисел {k1,...kn}, где ki-число циклов длины i.

Очевидно, что сумма всех ki равна количеству циклов перестановки, а сумма произведений i×ki есть размер перестановки.

Функция PermutationType[p] возвращает тип перестановки p.

Тождественная перестановка размера n состоит из n единичных циклов:

```
In[2]:= PermutationType[IdentityPermutation[10]]

Out[2]= {10, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

Перестановка и обратная ей имеют одинаковый тип:

```
In[3]:= p = RandomPermutation[100]; q = InversePermutation[p];
        SameQ[PermutationType[p], PermutationType[q]]
Out[4]= True
```

Функция NumberOfPermutationsByType[l] возвращает число перестановок типа l.
Существует ровно (n-1)! перестановок размера p, раскладываемых в один цикл:

```
In[5]:= NumberOfPermutationsByType[{0, 0, 0, 0, 0, 0, 0, 1}]
Out[5]= 5040
```

Существует ровно n(n-2)! перестановок размера p с одним циклом длины (n-1) и одним циклом длины 1.

```
In[6]:= n = 10; {NumberOfPermutationsByType[{1, 0, 0, 0, 0, 0, 0, 0, 1, 0}], n (n - 2) !}
Out[6]= {403200, 403200}
```

Функция NumberOfPermutationsByCycles[n, m] считает число перестановок длины n с ровно m циклами.
Существует ровно 50 перестановок с двумя циклами среди 120 перестановок размера 5.

```
In[7]:= {Length[Select[Map[ToCycles, Permutations[5]], (Length[#] == 2) &]],
         NumberOfPermutationsByCycles[5, 2]}
Out[7]= {50, 50}
```

PermutationWithCycle[n, {i, j, ...}] возвращает n - перестановку, в которой {i, j,……} цикл, а все другие элементы – циклы длины 1.

```
In[8]:= PermutationWithCycle[10, {7, 5, 3, 8, 2}]
Out[8]= {1, 7, 8, 4, 3, 6, 5, 2, 9, 10}

In[9]:= ToCycles[%]
Out[9]= {{1}, {7, 5, 3, 8, 2}, {4}, {6}, {9}, {10}}
```

Циклическую структуру перестановки можно увидеть, применив функцию CycleStructure.
CycleStructure[p, x] возвращает моном от переменных x[1], x[2], ..., x[k], где k – длина перестановки, отображающей циклическую структуру перестановки.

```
In[10]:= {ToCycles[p = {2, 3, 1, 4, 5}], CycleStructure[p, x]}
Out[10]= {{{2, 3, 1}, {4}, {5}}, x[1]^2 x[3]}
```

Моном $x[1]^2x[3]$ показывает, что перестановка имеет два цикла длины один и один цикл длины три.
Рассмотрим циклическую структуру каждой из шести перестановок размера 3:

```
In[11]:= Map[{#, ToCycles[#], CycleStructure[#, x]} &, Permutations[3]] // ColumnForm
Out[11]= {{1, 2, 3}, {{1}, {2}, {3}}, x[1]^3}
         {{1, 3, 2}, {{1}, {3, 2}}, x[1] x[2]}
         {{2, 1, 3}, {{2, 1}, {3}}, x[1] x[2]}
         {{2, 3, 1}, {{2, 3, 1}}, x[3]}
         {{3, 1, 2}, {{3, 2, 1}}, x[3]}
         {{3, 2, 1}, {{3, 1}, {2}}, x[1] x[2]}
```

Подсчитаем, сколько перестановок среди 5 - перестановок раскладываются в произведение циклов длины три и цикла длины два:

```
In[12]:= Count[Map[CycleStructure[#, x] &, Permutations[5]], x[2] x[3]]
Out[12]= 20
```

Цикл длины 2 называется **транспозицией** перестановки.
Функция MinimumChangePermutations[l] конструирует все перестановки списка l, так, что смежные перестановки отличаются только одной транспозицией.

```
In[13]:= MinimumChangePermutations[{x, y, z, r}]
Out[13]= {{x, y, z, r}, {y, x, z, r}, {z, x, y, r}, {x, z, y, r}, {y, z, x, r}, {z, y, x, r},
          {r, y, x, z}, {y, r, x, z}, {x, r, y, z}, {r, x, y, z}, {y, x, r, z}, {x, y, r, z},
          {x, z, r, y}, {z, x, r, y}, {r, x, z, y}, {x, r, z, y}, {z, r, x, y}, {r, z, x, y},
          {r, z, y, x}, {z, r, y, x}, {y, r, z, x}, {r, y, z, x}, {z, y, r, x}, {y, z, r, x}}
```

| | |
|---|---|
| PermutationType[p] | возвращает тип перестановки p |
| NumberOfPermutationsByType[l] | возвращает число перестановок типа l |
| NumberOfPermutationsByCycles[n, m] | считает число перестановок длины n с ровно m циклами |
| PermutationWithCycle[n, {i, j, ...}] | возвращает n- перестановку, в которой {i, j,……} цикл, а все другие элементы - циклы длины 1 |
| CycleStructure[p, x] | возвращает моном от переменных x[1], x[2], ..., x[k], где k – длина перестановки, отображающей циклическую структуру перестановки |
| MinimumChangePermutations[l] | конструирует все перестановки списка l, так, что смежные перестановки отличаются только одной транспозицией |

## ■ 2.4. Специальные классы перестановок

### *2.4.1. Инволюции*

Перестановка p называется **инволюцией**, если она является обратной самой себе.
Функция InvolutionQ[p] возвращает True, если перестановка p является инволюцией.
Выделим все инволюции среди всех перестановок размера 5:

```
In[2]:= inv = Select[Permutations[5], InvolutionQ]
Out[2]= {{1, 2, 3, 4, 5}, {1, 2, 3, 5, 4}, {1, 2, 4, 3, 5}, {1, 2, 5, 4, 3},
         {1, 3, 2, 4, 5}, {1, 3, 2, 5, 4}, {1, 4, 3, 2, 5}, {1, 4, 5, 2, 3},
         {1, 5, 3, 4, 2}, {1, 5, 4, 3, 2}, {2, 1, 3, 4, 5}, {2, 1, 3, 5, 4},
         {2, 1, 4, 3, 5}, {2, 1, 5, 4, 3}, {3, 2, 1, 4, 5}, {3, 2, 1, 5, 4},
         {3, 4, 1, 2, 5}, {3, 5, 1, 4, 2}, {4, 2, 3, 1, 5}, {4, 2, 5, 1, 3}, {4, 3, 2, 1, 5},
         {4, 5, 3, 1, 2}, {5, 2, 3, 4, 1}, {5, 2, 4, 3, 1}, {5, 3, 2, 4, 1}, {5, 4, 3, 2, 1}}
```

Их ровно 26:

```
In[3]:= Length[%]
Out[3]= 26
```

Если перестановка является инволюцией, то произведение ее на саму себя есть тождественная перестановка:

```
In[4]:= Map[(Permute[#, #] == IdentityPermutation[5]) &, inv]
Out[4]= {True, True, True, True, True, True, True, True, True, True, True, True,
         True, True, True, True, True, True, True, True, True, True, True, True}
```

Если перестановка является инволюцией, то она раскладывается в произведение циклов, максимальная длина которых равна двум.

```
In[5]:= Map[Max[Length[#]] &, Flatten[Map[ToCycles, inv], 1]]
Out[5]= {1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 2, 1, 1, 1, 2, 1, 1, 2, 1, 1, 1, 2, 2, 1, 2, 1, 1, 1, 2, 2,
         1, 2, 1, 1, 1, 2, 2, 2, 1, 1, 1, 2, 1, 2, 2, 2, 1, 2, 2, 1, 2, 1, 1, 1, 2, 1, 2, 2, 2, 1,
         2, 2, 1, 2, 1, 1, 1, 2, 1, 2, 2, 2, 1, 2, 2, 1, 2, 1, 1, 1, 2, 1, 2, 2, 2, 1, 2, 2, 1}
```

Involutions[l] выдает список инволюций элементов из списка l.
Involutions[l, Cycles] выдает инволюции в их циклическом представлении.
Involution[n] выдает инволюцию размера n.
Involutions[n, Cycles]выдает инволюции размера n в их циклическом представлении.
Таким образом, все инволюции среди перестановок размера 5 можно вычислить следующим образом:

```
In[6]:= Involutions[5]

Out[6]= {{2, 1, 4, 3, 5}, {2, 1, 5, 4, 3}, {2, 1, 3, 5, 4}, {2, 1, 3, 4, 5},
         {3, 4, 1, 2, 5}, {3, 5, 1, 4, 2}, {3, 2, 1, 5, 4}, {3, 2, 1, 4, 5},
         {4, 3, 2, 1, 5}, {4, 5, 3, 1, 2}, {4, 2, 5, 1, 3}, {4, 2, 3, 1, 5},
         {5, 3, 2, 4, 1}, {5, 4, 3, 2, 1}, {5, 2, 4, 3, 1}, {5, 2, 3, 4, 1},
         {1, 3, 2, 5, 4}, {1, 3, 2, 4, 5}, {1, 4, 5, 2, 3}, {1, 4, 3, 2, 5}, {1, 5, 4, 3, 2},
         {1, 5, 3, 4, 2}, {1, 2, 4, 3, 5}, {1, 2, 5, 4, 3}, {1, 2, 3, 5, 4}, {1, 2, 3, 4, 5}}
```

Теперь рассмотрим разложение этих перестановок в циклы:

```
In[7]:= Involutions[5, Cycles]
Out[7]= {{{5}, {3, 4}, {1, 2}}, {{4}, {3, 5}, {1, 2}},
         {{4, 5}, {3}, {1, 2}}, {{5}, {4}, {3}, {1, 2}}, {{5}, {2, 4}, {1, 3}},
         {{4}, {2, 5}, {1, 3}}, {{4, 5}, {2}, {1, 3}}, {{5}, {4}, {2}, {1, 3}},
         {{5}, {2, 3}, {1, 4}}, {{3}, {2, 5}, {1, 4}}, {{3, 5}, {2}, {1, 4}},
         {{5}, {3}, {2}, {1, 4}}, {{4}, {2, 3}, {1, 5}}, {{3}, {2, 4}, {1, 5}},
         {{3, 4}, {2}, {1, 5}}, {{4}, {3}, {2}, {1, 5}}, {{4, 5}, {2, 3}, {1}},
         {{5}, {4}, {2, 3}, {1}}, {{3, 5}, {2, 4}, {1}}, {{5}, {3}, {2, 4}, {1}},
         {{3, 4}, {2, 5}, {1}}, {{4}, {3}, {2, 5}, {1}}, {{5}, {3, 4}, {2}, {1}},
         {{4}, {3, 5}, {2}, {1}}, {{4, 5}, {3}, {2}, {1}}, {{5}, {4}, {3}, {2}, {1}}}
```

NumberOfInvolutions[n] считает число инволюций из n элементов.

```
In[8]:= {Length[Involutions[5]], NumberOfInvolutions[5]}
Out[8]= {26, 26}
```

| | |
|---|---|
| InvolutionQ[p] | возвращает True, если перестановка p является инволюцией |
| Involutions[l] | возвращает список инволюций элементов из списка l |
| Involutions[l, Cycles] | возвращает список инволюций элементов из списка l в их циклическом представлении |

| | |
|---|---|
| Involution[n] | возвращает инволюцию размера n |
| Involutions[n, Cycles] | возвращает инволюции размера n в их циклическом представлении |
| NumberOfInvolutions[n] | считает число инволюций из n элементов |

### *2.4.2 Перестановки без неподвижных точек*

Перестановка называется **перестановкой без неподвижных точек,** если она не содержит циклов длины 1.

DerangementQ[p] тестирует, будет ли p перестановкой без неподвижных точек.

```
In[2]:= Table[{p = RandomPermutation[i], DerangementQ[p]}, {i, 1, 10}] // ColumnForm
Out[2]= {{1}, False}
        {{1, 2}, False}
        {{3, 1, 2}, True}
        {{3, 1, 4, 2}, True}
        {{1, 5, 4, 2, 3}, False}
        {{5, 1, 4, 2, 6, 3}, True}
        {{7, 1, 3, 2, 5, 4, 6}, False}
        {{3, 8, 6, 5, 2, 1, 7, 4}, False}
        {{8, 9, 3, 5, 6, 1, 4, 7, 2}, False}
        {{6, 5, 4, 7, 10, 3, 8, 2, 9, 1}, False}
```

NumberOfDerangements[n] подсчитывает количество перестановок из n элементов, не содержащих неподвижных точек.

```
In[3]:= {Length[Select[Permutations[5], DerangementQ]], NumberOfDerangements[5]}
Out[3]= {44, 44}
```

Подсчитаем число перестановок без неподвижных точек среди перестановок размера i (i=1,…,10)

```
In[4]:= Table[NumberOfDerangements[i], {i, 1, 10}]
Out[4]= {0, 1, 2, 9, 44, 265, 1854, 14833, 133496, 1334961}
```

Derangements[p] конструирует все перестановки без неподвижных точек.
Рассмотрим все перестановки без неподвижных точек из четырех элементов и их циклическое разложение.

```
In[5]:= Table[{Derangements[4][[i]], Map[ToCycles, Derangements[4]][[i]]}, {i, 1, 9}] //
        ColumnForm
Out[5]= {{2, 1, 4, 3}, {{2, 1}, {4, 3}}}
        {{2, 3, 4, 1}, {{2, 3, 4, 1}}}
        {{2, 4, 1, 3}, {{2, 4, 3, 1}}}
        {{3, 1, 4, 2}, {{3, 4, 2, 1}}}
        {{3, 4, 1, 2}, {{3, 1}, {4, 2}}}
        {{3, 4, 2, 1}, {{3, 2, 4, 1}}}
        {{4, 1, 2, 3}, {{4, 3, 2, 1}}}
        {{4, 3, 1, 2}, {{4, 2, 3, 1}}}
        {{4, 3, 2, 1}, {{4, 1}, {3, 2}}}
```

| | |
|---|---|
| DerangementQ[p] | тестирует, будет ли p перестановкой без неподвижных точек |

| | |
|---|---|
| NumberOfDerangements[n] | подсчитывает количество перестановок из n элементов, не содержащих неподвижных точек |
| Derangements[p] | конструирует все перестановки без неподвижных точек |

## ■ 2.5. Комбинации

### *2.5.1. Множество всех подмножеств*

Любое n- элементное множество содержит $2^n$ подмножеств.
Subsets[l] возвращает все подмножества множества l.

```
In[2]:= Subsets[{m, n, r, t}]
Out[2]= {{}, {t}, {r, t}, {r}, {n, r}, {n, r, t}, {n, t}, {n}, {m, n},
        {m, n, t}, {m, n, r, t}, {m, n, r}, {m, r}, {m, r, t}, {m, t}, {m}}
```

NthSubset[n, l] выдает n-е подмножество списка l, заданного в каноническом порядке (при этом пустое множество имеет номер 0)

```
In[3]:= NthSubset[9, {m, n, r, t}]
Out[3]= {m, n, t}
```

С помощью NthSubset можно перечислить все подмножества в каноническом порядке:

```
In[4]:= Table[NthSubset[i, {m, n, r, t}], {i, 0, 15}]
Out[4]= {{}, {t}, {r, t}, {r}, {n, r}, {n, r, t}, {n, t}, {n}, {m, n},
        {m, n, t}, {m, n, r, t}, {m, n, r}, {m, r}, {m, r, t}, {m, t}, {m}}
```

NextSubset[l,s] конструирует подмножество множества l, следующее за s в каноническом порядке.
Теперь перечислим все подмножества данного множества с помощью NextSubset:

```
In[5]:= Map[(NextSubset[{m, n, r, t}, #]) &, Subsets[{m, n, r, t}]]
Out[5]= {{t}, {r}, {n}, {r, t}, {n, r, t}, {m}, {n, r}, {n, t}, {m, n, t},
        {m, n, r}, {}, {m, n, r, t}, {m, r, t}, {m, n}, {m, r}, {m, t}}
```

RandomSubset[l] создает случайное подмножество множества l.
Рассмотрим случайные подмножества, взятые по одному из множеств {1,2,3,…,n}, где n возрастает от 1 до 20:

```
In[6]:= Table[RandomSubset[Range[i]], {i, 1, 20}]
Out[6]= {{1}, {2}, {2}, {2, 4}, {1}, {2, 3, 4}, {2, 4, 5, 6, 7}, {2, 5, 6},
        {1, 3, 6, 7, 8, 9}, {3, 5, 6, 7, 8, 9}, {3, 5, 7, 9, 11}, {4, 5}, {2, 3, 5, 8, 9, 13},
        {2, 6, 7, 11, 12, 14}, {1, 2, 3, 4, 6, 11}, {2, 4, 7, 8, 9, 10, 14, 15},
        {1, 2, 3, 6, 7, 8, 10, 11, 12, 14, 16}, {1, 2, 3, 6, 8, 9, 10, 11, 12, 13, 14, 16, 18},
        {3, 8, 9, 13, 14, 15, 16, 17, 18}, {1, 2, 6, 7, 8, 10, 14, 16, 17, 19, 20}}
```

RankSubset[l, s] возвращает **ранг** подмножества – число подмножеств, предшествующих подмножеству s в множестве всех подмножеств множества l, перечисленных в каноническом порядке.
UnrankSubset[n, l] возвращает n-е подмножество в множестве всех подмножеств множества l, перечисленных в каноническом порядке.

Перечислим в каноническом порядке все подмножества множества {b,a,c,d}, применяя UnrankSubset.

```
In[7]:= Table[UnrankSubset[i, {b, a, c, d}], {i, 0, 15}]
Out[7]= {{}, {d}, {c, d}, {c}, {a, c}, {a, c, d}, {a, d}, {a}, {b, a},
        {b, a, d}, {b, a, c, d}, {b, a, c}, {b, c}, {b, c, d}, {b, d}, {b}}
```

Ранг подмножества {a,c,d} в {b,a,c,d} равен 5.

```
In[8]:= RankSubset[{b, a, c, d}, {a, c, d}]
Out[8]= 5
```

Перечислить все подмножества множества в лексикографическом порядке можно, применив функцию LexicographicSubsets.
LexicographicSubsets[l] возвращает все подмножества множества l в лексикографическом порядке. LexicographicSubsets[n] возвращает все подмножества множества {1,2,3,…,n} в лексикографическом порядке.

```
In[9]:= LexicographicSubsets[{b, a, c, d}]
Out[9]= {{}, {b}, {b, a}, {b, a, c}, {b, a, c, d}, {b, a, d}, {b, c},
        {b, c, d}, {b, d}, {a}, {a, c}, {a, c, d}, {a, d}, {c}, {c, d}, {d}}
```

NextLexicographicSubset[l, s] выдает следующее за подмножеством s подмножество множества l, в множестве всех подмножеств множества l, перечисленного в лексикографическом порядке.

```
In[10]:= NextLexicographicSubset[{b, a, c, d}, {a, c, d}]
Out[10]= {a, d}
```

Биекция между строками, состоящими из нулей и единиц длины n (бинарными строками) и множеством n целых чисел $\{0,1,\ldots,2^n-1\}$, задается с помощью соответствия:

$$b_{n-1}b_{n-2}\ldots b_1b_0 \Leftrightarrow \sum_{i=0}^{n-1} 2^i b_i$$

BinarySubsets[l] выдает все подмножества множества l, упорядоченные согласно бинарной строке, определяющей каждое подмножество.
Для каждого положительного целого n функция BinarySubsets[n] возвращает все подмножества множества {1, 2,.., n}, упорядоченные согласно бинарной строке, определяющей каждое подмножество.

```
In[11]:= a = BinarySubsets[{m, n, r, t}]
Out[11]= {{}, {t}, {r}, {r, t}, {n}, {n, t}, {n, r}, {n, r, t}, {m},
         {m, t}, {m, r}, {m, r, t}, {m, n}, {m, n, t}, {m, n, r}, {m, n, r, t}}
```

Функция NextBinarySubset[l, s] конструирует подмножество множества l, которое следует за подмножеством s в порядке, полученном интерпретацией подмножеств как бинарной строки представления целых чисел:
Выделим в списке a подмножество, следующее за подмножеством {m,r}:

```
In[12]:= NextBinarySubset[{m, n, r, t}, {m, r}]
Out[12]= {m, r, t}
```

С помощью NextBinarySubset можно упорядочить все подмножества данного множества в порядке, полученном интерпретацией подмножеств как бинарной строки представления целых чисел:

```
In[13]:= b = NestList[NextBinarySubset[{m, n, r, t}, #] &, {}, 15]
Out[13]= {{}, {t}, {r}, {r, t}, {n}, {n, t}, {n, r}, {n, r, t}, {m},
         {m, t}, {m, r}, {m, r, t}, {m, n}, {m, n, t}, {m, n, r}, {m, n, r, t}}

In[14]:= SameQ[a, b]
Out[14]= True
```

RankBinarySubset[l, s] возвращает ранг подмножества s множества l в порядке следования подмножеств l, полученных интерпретацией этих подмножеств как бинарной строки представления целых чисел, где ранг подмножества – это количество подмножеств, предшествующих этому подмножеству в данном порядке.

```
In[15]:= RankBinarySubset[{m, n, r, t}, {m, r}]
Out[15]= 10
```

UnrankBinarySubset[n, l] возвращает n-е подмножество списка l, где l упорядочено в порядке, полученном интерпретацией подмножеств как бинарной строки представления целых чисел:

```
In[16]:= UnrankBinarySubset[10, {m, n, r, t}]
Out[16]= {m, r}
```

| | |
|---|---|
| Subsets[l] | выдает все подмножества множества l |
| NthSubset[n, l] | выдает n-е подмножество списка l, заданного в каноническом порядке |
| NextSubset[l,s] | конструирует подмножество множества l, следующее за s в каноническом порядке |
| RandomSubset[l] | конструирует случайное подмножество множества l |
| RankSubset[l, s] | возвращает число подмножеств, предшествующих подмножеству s в множестве всех подмножеств множества l, заданного в каноническом порядке |
| UnrankSubset[n, l] | возвращает n-е подмножество множества l, заданное в каноническом порядке |
| LexicographicSubsets[l] | возвращает все подмножества множества l в лексикографическом порядке |
| LexicographicSubsets[n] | возвращает все подмножества множества{1,2,3,…,n} в лексикографическом порядке |
| BinarySubsets[l] | возвращает все подмножества множества l, упорядоченные согласно бинарной строке, определяющей каждое подмножество |
| NextBinarySubset[l, s] | конструирует подмножество множества l, которое следует за подмножеством s в порядке, полученном интерпретацией подмножеств как бинарной строки представления целых чисел |
| RankBinarySubset[l, s] | выдает ранг подмножества s множества l в порядке следования подмножеств l, полученных интерпретацией этих подмножеств как бинарной строки представления целых чисел. |
| UnrankBinarySubset[n, l] | возвращает n-е подмножество списка l, где l упорядочено в порядке, полученном интерпретацией подмножеств как бинарной строки представления целых чисел |

### *2.5.2. Код Грэя*

В предыдущей части мы перечисляли перестановки в минимально измененном порядке с помощью функции MinimumChangePermutation. Аналогичная задача для подмножеств: перечислить все подмножества данного множества так, чтобы смежные подмножества отличались друг от друга удалением или вставкой ровно одного элемента. Такое перечисление подмножеств задает функция GrayCodeSubsets[l].
GrayCodeSubsets[l] перечисляет все подмножества множества l в таком порядке, что смежные подмножества отличались друг от друга удалением или вставкой ровно одного элемента.
Перечислим элементы множества {m,n,r,t} в «минимально измененном порядке»:

```
In[2]:= GrayCodeSubsets[{m, n, r, t}]
Out[2]= {{}, {t}, {r, t}, {r}, {n, r}, {n, r, t}, {n, t}, {n}, {m, n},
         {m, n, t}, {m, n, r, t}, {m, n, r}, {m, r}, {m, r, t}, {m, t}, {m}}
```

Функция NextGrayCodeSubset[l, s] конструирует следующее за s подмножество в множества l, перечисленного в порядке Грэйкода.
RankGrayCodeSubset[l, s] возвращает ранг подмножества s множества l, перечисленного в порядке Грэйкода.
UnrankGrayCodeSubset[n, l] возвращает n -е подмножество списка l , перечисленного в порядке Грэйкода.
Последовательное применение функции UnrankGrayCodeSubset к данному множеству позволяет упорядочить элементы этого множества в порядке Грэйкода:

```
In[3]:= s = Table[UnrankGrayCodeSubset[i, {m, n, r, t}], {i, 0, 15}]
Out[3]= {{}, {t}, {r, t}, {r}, {n, r}, {n, r, t}, {n, t}, {n}, {m, n},
         {m, n, t}, {m, n, r, t}, {m, n, r}, {m, r}, {m, r, t}, {m, t}, {m}}
```

Вычислим теперь ранги подмножеств в вышеприведенном Грэйкоде:

```
In[4]:= Map[RankGrayCodeSubset[{m, n, r, t}, #] &, s]
Out[4]= {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}
```

| | |
|---|---|
| GrayCodeSubsets[l] | перечисляет все подмножества множества l в таком порядке, что смежные подмножества отличались друг от друга удалением или вставкой ровно одного элемента |
| NextGrayCodeSubset[l, s] | конструирует следующее за s подмножество в Грэйкоде множества l |
| UnrankGrayCodeSubset[n, l] | возвращает n -е подмножество списка l , перечисленное в порядке Грэйкода |

### *2.5.3. К-подмножества*

**К- подмножеством** множества называется такое его подмножество, которое содержит ровно k элементов. В каждом n- элементном множестве содержится ровно $C_n^k$ k- подмножеств.
KSubsets[l,k] возвращает все k- подмножества множества l в лексикографическом порядке.

```
In[2]:= KSubsets[{a, b, c, d, e, f}, 3]
Out[2]= {{a, b, c}, {a, b, d}, {a, b, e}, {a, b, f}, {a, c, d}, {a, c, e},
         {a, c, f}, {a, d, e}, {a, d, f}, {a, e, f}, {b, c, d}, {b, c, e}, {b, c, f},
         {b, d, e}, {b, d, f}, {b, e, f}, {c, d, e}, {c, d, f}, {c, e, f}, {d, e, f}}
```

Количество r-элементных подмножеств в десятиэлементном подмножестве равно $C_{10}^r$

```
In[3]:= Table[{Length[KSubsets[Range[10], r]], Binomial[10, r]}, {r, 0, 5}]
Out[3]= {{1, 1}, {10, 10}, {45, 45}, {120, 120}, {210, 210}, {252, 252}}
```

А теперь подсчитаем количество трехэлементных подмножеств в множестве {1,2,…,r}, где r изменяется от 3 до 10.

```
In[4]:= Table[{Length[KSubsets[Range[r], 3]], Binomial[r, 3]}, {r, 3, 10}]
Out[4]= {{1, 1}, {4, 4}, {10, 10}, {20, 20}, {35, 35}, {56, 56}, {84, 84}, {120, 120}}
```

NextKSubset[l, s] возвращает к-подмножество списка l, следующее за к-подмножеством s в лексикографическом порядке.
Перечислим в лексикографическом порядке двадцать подмножеств множества {a,b,c,d,e,f,g} начиная с подмножества {a,b,c,d,e}.

```
In[5]:= NestList[NextLexicographicSubset[{a, b, c, d, e, f, g}, #] &, {a, b, c, d, e},
         20]
Out[5]= {{a, b, c, d, e}, {a, b, c, d, e, f}, {a, b, c, d, e, f, g}, {a, b, c, d, e, g},
          {a, b, c, d, f}, {a, b, c, d, f, g}, {a, b, c, d, g}, {a, b, c, e},
          {a, b, c, e, f}, {a, b, c, e, f, g}, {a, b, c, e, g}, {a, b, c, f},
          {a, b, c, f, g}, {a, b, c, g}, {a, b, d}, {a, b, d, e}, {a, b, d, e, f},
          {a, b, d, e, f, g}, {a, b, d, e, g}, {a, b, d, f}, {a, b, d, f, g}}
```

RandomKSubset[l, k] возвращает случайное к-подмножество множества l.
Перечислим случайные k-подмножества в десятиэлементном множестве, где k меняется от 1 до 10.

```
In[6]:= Table[RandomKSubset[10, i], {i, 1, 10}] // ColumnForm
Out[6]= {5}
        {8, 10}
        {2, 5, 8}
        {2, 3, 7, 8}
        {3, 5, 6, 7, 9}
        {2, 3, 5, 6, 9, 10}
        {3, 4, 5, 6, 7, 8, 9}
        {1, 2, 4, 5, 7, 8, 9, 10}
        {1, 2, 3, 4, 6, 7, 8, 9, 10}
        {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

Распределение 4-подмножеств является достаточно равномерным:

```
In[7]:= Distribution[Table[RandomKSubset[6, 4], {300}]]
Out[7]= {15, 13, 16, 20, 15, 21, 30, 21, 16, 30, 17, 31, 20, 18, 17}
```

RankKSubset[s, l] возвращает ранг k-подмножества s – число предшествуюших k-подмножеству s в списке k-подмножеств множества l, упорядоченных в лексикографическом порядке.
UnrankKSubset[m, k, l] возвращает m-е k-подмножество списка l, перечисленного в лексикографическом порядке.
Перечислим в лексикографическом порядке все 3-подмножества в множестве {a,b,c,d,e,f}, применяя UnrankKSubset

```
In[8]:= Map[UnrankKSubset[#, 3, {a, b, c, d, e, f}] &, Range[0, 19]]
Out[8]= {{a, b, c}, {a, b, d}, {a, b, e}, {a, b, f}, {a, c, d}, {a, c, e},
          {a, c, f}, {a, d, e}, {a, d, f}, {a, e, f}, {b, c, d}, {b, c, e}, {b, c, f},
          {b, d, e}, {b, d, f}, {b, e, f}, {c, d, e}, {c, d, f}, {c, e, f}, {d, e, f}}
```

Выделим 1000-е подмножество в списке всех 10-подмножеств первых тридцати натуральных чисел, которые перечислены в лексикографическом порядке.

```
In[9]:= UnrankKSubset[999, 10, Range[30]]
Out[9]= {1, 2, 3, 4, 5, 6, 7, 13, 16, 30}
```

Ранг этого подмножества должен быть равен 999.

```
In[10]:= RankKSubset[{1, 2, 3, 4, 5, 6, 7, 13, 16, 30}, Range[30]]
Out[10]= 999
```

K-подмножества данного множества можно перечислить в «минимально измененном порядке» с помощью GrayCodeKSubsets.
GrayCodeKSubsets[l, k] возвращает все k-подмножества множества l в порядке Грэй-кода, то есть каждое подмножество отличается от смежного подмножества удалением и включением ровно одного элемента.

```
In[11]:= GrayCodeKSubsets[{a, b, c, d, e, f}, 3]
Out[11]= {{a, b, c}, {a, c, d}, {b, c, d}, {a, b, d}, {a, d, e}, {b, d, e},
          {c, d, e}, {a, c, e}, {b, c, e}, {a, b, e}, {a, e, f}, {b, e, f}, {c, e, f},
          {d, e, f}, {a, d, f}, {b, d, f}, {c, d, f}, {a, c, f}, {b, c, f}, {a, b, f}}
```

**Строкой** длины k на алфавите M называется упорядоченное множество k не обязательно различных символов алфавита M.
Функция Strings[l, n] конструирует все возможные строки длины n из символов списка l.

```
In[12]:= Strings[{a, b, c, d}, 3]
Out[12]= {{a, a, a}, {a, a, b}, {a, a, c}, {a, a, d}, {a, b, a}, {a, b, b}, {a, b, c}, {a, b, d},
          {a, c, a}, {a, c, b}, {a, c, c}, {a, c, d}, {a, d, a}, {a, d, b}, {a, d, c},
          {a, d, d}, {b, a, a}, {b, a, b}, {b, a, c}, {b, a, d}, {b, b, a}, {b, b, b},
          {b, b, c}, {b, b, d}, {b, c, a}, {b, c, b}, {b, c, c}, {b, c, d}, {b, d, a},
          {b, d, b}, {b, d, c}, {b, d, d}, {c, a, a}, {c, a, b}, {c, a, c}, {c, a, d},
          {c, b, a}, {c, b, b}, {c, b, c}, {c, b, d}, {c, c, a}, {c, c, b}, {c, c, c},
          {c, c, d}, {c, d, a}, {c, d, b}, {c, d, c}, {c, d, d}, {d, a, a}, {d, a, b},
          {d, a, c}, {d, a, d}, {d, b, a}, {d, b, b}, {d, b, c}, {d, b, d}, {d, c, a},
          {d, c, b}, {d, c, c}, {d, c, d}, {d, d, a}, {d, d, b}, {d, d, c}, {d, d, d}}
```

Количество строк длины n из алфавита длины m равно $m^n$.

```
In[13]:= Table[{Length[Strings[Range[5], n]], 5^n}, {n, 1, 6}] // ColumnForm
Out[13]= {5, 5}
         {25, 25}
         {125, 125}
         {625, 625}
         {3125, 3125}
         {15625, 15625}
```

| | |
|---|---|
| KSubsets[l,k] | возвращает все k- подмножества множества l в лексикографическом порядке |
| NextKSubset[l, s] | возвращает k-подмножества списка l, следующее за k-подмножеством s в лексикографическом порядке |
| RandomKSubset[l, k] | возвращает случайное k-подмножество множества l |
| RankKSubset[s, l] | возвращает число предшествуюших k-подмножеству s в списке |

| | |
|---|---|
| | k-подмножеств множества l, упорядоченных в лексикографическом порядке |
| UnrankKSubset[m, k, l] | возвращает m-e k-подмножество списка l, перечисленного в в лексикографическом порядке |
| GrayCodeKSubsets[l, k] | возвращает все k-подмножества множества l в порядке Грэй-кода, то есть каждое подмножество отличается от смежного подмножества удалением и включением ровно одного элемента |
| Strings[l, n] | конструирует все возможные строки длины n из символов списка l |

# Глава 3. Группы перестановок

## ■ 3.1. Группы и бинарные отношения

**Группой** называется непустое множество S с бинарной операцией ■, удовлетворяющее следующим условиям:

1. Замкнутость относительно операции: Для любых a, b∈S, a■b∈S
2. Ассоциативность: Для любых a, b, c∈S, (a■b)■c=a■(b■c)
3. Существование единичного элемента: Существует элемент e∈S, называемый единицей, такой, что для всех a∈S, a■e=a.
4. Существование обратного элемента: Для любого a∈S, существует элемент в S, обозначаемый $a^{-1}$ и называемый обратным к a, такой, что a■ $a^{-1}$=e.

Число элементов в группе называется ее **порядком**; конечная группа имеет конечный порядок. Нас в основном будут интересовать группы S перестановок, то есть S некоторое множество перестановок с бинарной операцией ■ - операцией умножения перестановок, при условии, что выполняются условия 1- 4.

**Бинарным отношением** R на множестве S называется подмножество декартова произведения этих множеств S×S. Другими словами, R есть некоторое подмножество упорядоченных пар элементов из S.

R называется **рефлексивным**, если для каждого i отношение R содержит пару {i,i}.

R называется **симметричным**, если для любых i,j∈S, если из условия, что {i,i}∈R, следует, что {j,i}}∈R.

R называется **транзитивным**, если для любых i,j,k∈S, из условия, что {i,i}∈R и {j,k}∈R следует, что {i,k}}∈R.

Бинарное отношение R называется **отношением эквивалентности**, если оно рефлексивно, симметрично и транзитивно.

Отношение эквивалентности разбивает элементы множества S на классы эквивалентности: каждая пара одного класса находится в отношении, и никакая пара из разных классов не находится в отношении R.

Combinatorica обеспечена булевыми функциями ReflexiveQ, SymmetricQ, TransitiveQ, которые тестируют, является ли бинарное отношение рефлексивным, симметричным и транзитивным соответственно .

Функция EquivalenceRelationQ тестирует, является ли бинарное отношение отношением эквивалентности.

Эти функции принимают отношение R:

1) в виде квадратной матрицы, чьи ненулевые элементы означают пары, находящиеся в отношении

2) в виде графа, чьи ребра соединяют пары, находящиеся в отношении.

Функция EquivalenceClasses[r] определяет классы эквивалентности отношения, заданного матрицей r.

Функция SamenessRelation[l] конструирует бинарное отношение из списка l перестановок, которое является отношением эквивалентности, если l- группа перестановок.

```
In[2]:= l = {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}};
        {EquivalenceRelationQ[l], ReflexiveQ[l], SymmetricQ[l], TransitiveQ[l]}
Out[2]= {True, True, True, True}

In[3]:= EquivalenceClasses[l]
Out[3]= {{1}, {2}, {3}}
```

| | |
|---|---|
| ReflexiveQ[g] | возвращает True, если матрица смежности графа g представляет рефлексивное бинарное отношение |
| SymmetricQ[g] | возвращает True, если матрица смежности графа g представля- |

| | |
|---|---|
| | ет симметричное бинарное отношение |
| TransitiveQ[g] | возвращает True, если матрица смежности графа g представляет транзитивное бинарное отношение |
| EquivalenceRelationQ[r] | возвращает True, если матрица r определяет отношение эквивалентности |
| EquivalenceRelationQ[g] | возвращает True, если матрица смежности графа g определяет отношение эквивалентности |
| EquivalenceClasses[r] | определяет классы эквивалентности, заданные матрицей r |
| SamenessRelation[l] | конструирует бинарное отношение из списка l перестановок, которое является отношением эквивалентности, если l- группа перестановок |

## ■ 3.2. Основные группы перестановок

### 3.2.1 Симметрическая, циклическая, диэдральная, знакопеременная группы

Определим основные группы перестановок. Множество всех n-перестановок образует группу относительно операции умножения перестановок и называется **симметрической группой** и обозначается $S_n$. **Циклической группой** $C_n$ называется множество всех n-перестановок, получающихся циклическим сдвигом на множестве {1,2,3,…,n}. **Диэдральная группа** $D_{2n}$ состоит из перестановок группы $C_n$ и всех перестановок этой группы, перечисленных в обратном порядке. **Альтернативная (знакопеременная) группа** $A_n$ состоит из всех четных n-перестановок
Функция SymmetricGroup[n] возвращает симметрическую группу из n символов.
PermutationGroupQ[l] возвращает True, если список перестановок l образует группу перестановок.
Рассмотрим симметрическую группу на четырех элементах:

```
In[2]:= SymmetricGroup[4]
Out[2]= {{1, 2, 3, 4}, {1, 2, 4, 3}, {1, 3, 2, 4}, {1, 3, 4, 2}, {1, 4, 2, 3}, {1, 4, 3, 2},
         {2, 1, 3, 4}, {2, 1, 4, 3}, {2, 3, 1, 4}, {2, 3, 4, 1}, {2, 4, 1, 3}, {2, 4, 3, 1},
         {3, 1, 2, 4}, {3, 1, 4, 2}, {3, 2, 1, 4}, {3, 2, 4, 1}, {3, 4, 1, 2}, {3, 4, 2, 1},
         {4, 1, 2, 3}, {4, 1, 3, 2}, {4, 2, 1, 3}, {4, 2, 3, 1}, {4, 3, 1, 2}, {4, 3, 2, 1}}

In[3]:= SameQ[Permutations[4], SymmetricGroup[4]]
Out[3]= True
```

Проверим, что это множество является группой перестановок:

```
In[4]:= PermutationGroupQ[Permutations[4]]
Out[4]= True
```

CyclicGroup[n] возвращает циклическую группу перестановок из n символов.

```
In[5]:= a = CyclicGroup[10]
Out[5]= {{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, {10, 1, 2, 3, 4, 5, 6, 7, 8, 9},
         {9, 10, 1, 2, 3, 4, 5, 6, 7, 8}, {8, 9, 10, 1, 2, 3, 4, 5, 6, 7},
         {7, 8, 9, 10, 1, 2, 3, 4, 5, 6}, {6, 7, 8, 9, 10, 1, 2, 3, 4, 5},
         {5, 6, 7, 8, 9, 10, 1, 2, 3, 4}, {4, 5, 6, 7, 8, 9, 10, 1, 2, 3},
         {3, 4, 5, 6, 7, 8, 9, 10, 1, 2}, {2, 3, 4, 5, 6, 7, 8, 9, 10, 1}}
```

Проверим, что множество перестановок a образует группу перестановок:

```
In[6]:= PermutationGroupQ[a]
```

```
Out[6]= True
```

Рассмотрим циклическую структуру перестановок из множества a:

```
In[7]:= Union[Map[CycleStructure[#, x] &, a]]
Out[7]= {x[1]^10, x[2]^5, x[5]^2, x[10]}
```

Функция DihedralGroup[n] возвращает диэдральную группу из n символов. Заметим, что порядок этой группы - 2 n.

```
In[8]:= DihedralGroup[5]
Out[8]= {{1, 2, 3, 4, 5}, {5, 1, 2, 3, 4}, {4, 5, 1, 2, 3}, {3, 4, 5, 1, 2}, {2, 3, 4, 5, 1},
         {5, 4, 3, 2, 1}, {4, 3, 2, 1, 5}, {3, 2, 1, 5, 4}, {2, 1, 5, 4, 3}, {1, 5, 4, 3, 2}}
```

Построим это множество по определению:

```
In[9]:= d = Union[b = CyclicGroup[5], Map[Reverse, b]]
Out[9]= {{1, 2, 3, 4, 5}, {1, 5, 4, 3, 2}, {2, 1, 5, 4, 3}, {2, 3, 4, 5, 1}, {3, 2, 1, 5, 4},
         {3, 4, 5, 1, 2}, {4, 3, 2, 1, 5}, {4, 5, 1, 2, 3}, {5, 1, 2, 3, 4}, {5, 4, 3, 2, 1}}
```

Покажем, что d есть DihedralGroup[5]:

```
In[10]:= Complement[d, DihedralGroup[5]]
Out[10]= {}
```

AlternatingGroup[n] порождает множество четных перестановок из n символов, знакопеременную группу из n символов.
AlternatingGroup[l] порождает множество четных перестановок множества l.
Множество четных перестановок действительно образует группу:

```
In[11]:= PermutationGroupQ[AlternatingGroup[5]]
Out[11]= True
```

Произведение четных перестановок есть четная перестановка. Проверим это на случайных перестановках, чтобы подтвердить замкнутость относительно умножения альтернативной группы перестановок.

```
In[12]:= p = RandomPermutation[200]; q = RandomPermutation[200];
         SameQ[SignaturePermutation[p] SignaturePermutation[q],
          SignaturePermutation[Permute[p, q]]]
Out[13]= True
```

| | |
|---|---|
| SymmetricGroup[n] | возвращает симметрическую группу из n символов |
| CyclicGroup[n] | возвращает циклическую группу перестановок из n символов |
| DihedralGroup[n] | возвращает диэдральную группу из n символов |
| AlternatingGroup[n] | порождает множество четных перестановок из n символов, альтернативную группу из n символов |
| AlternatingGroup[l] | порождает множество четных перестановок множества l |
| PermutationGroupQ[l] | возвращаетет True, если список перестановок l образует группу перестановок |

### *3.2.2. Таблица умножения*

Удобным способом проверки, является ли множество некоторых элементов G с бинарной операцией ■ группой , построить таблицу умножения G относительно ■.
Это двумерная таблица размера |G|×|G|, (i,j)-й элемент которой равен k тогда и только тогда, когда i-й элемент G, умноженный на j-й элемент G равен k-му элементу в G.
Функция MultiplicationTable[l, f] конструирует полную таблицу умножения, определенную бинарным отношением функции f на элементах списка l.
Составим таблицу умножения циклической группы на 6 элементах;

```
In[2]:= MultiplicationTable[CyclicGroup[6], Permute] // TableForm
```

```
Out[2]//TableForm=
1  2  3  4  5  6
2  3  4  5  6  1
3  4  5  6  1  2
4  5  6  1  2  3
5  6  1  2  3  4
6  1  2  3  4  5
```

Покажем, что группы SymmetricGroup[4] и Permutations[4] изоморфны:

```
In[3]:= SameQ[MultiplicationTable[SymmetricGroup[4], Permute],
          MultiplicationTable[Permutations[4], Permute]]

Out[3]= True
```

Рассмотрим таблицу умножения знакопеременной группы шестого порядка. Это множество, естественно, является группой

```
In[2]:= PermutationGroupQ[s = MultiplicationTable[AlternatingGroup[6], Permute]]

Out[2]= True
```

Проверим замкнутость относительно операции:

```
In[3]:= {Count[Map[Signature, s], 1], Length[s]}

Out[3]= {360, 360}
```

| | |
|---|---|
| MultiplicationTable[l, f] | конструирует полную таблицу умножения, определенную бинарным отношением функции f на элементах списка l |

## ■3.3. Теорема Пойа

### *3.3.1. Действие группы*

В основе теории перечисления Пойа лежит понятие действия группы на множестве. Рассмотрим пример.
Пусть S множество всех 2-подмножеств в {1,2,3,4}, p-произвольная перестановка первых четырех натуральных чисел. Для любого 2-подмножества $s=\{x_1,x_2\}$ p-можно представить как отображение этого множества в $\{p(x_1), p(x_2)\}$.
Заметим, что p(s) также 2- подмножество и поэтому является элементом S. Таким образом, p может быть представлено как отображение S на себя, или, более точно, легко видеть, что p:S->S является биекцией. Занумеруем элементы S в некотором порядке от 1 до 6, p может быть пред-

ставлена как индуцированная перестановка 6 порядка, переставляющая элементы S. Обозначим ее как p(S).
Возьмем перестановку p={3,1,4,2}. Эта перестановка 1->3, 2->1, 3->4, 5->2.

```
In[2]:= p = {3, 1, 4, 2}; {S = KSubsets[Range[4], 2], p = Map[p[[#]] &, S]} //
        ColumnForm
Out[2]= {{1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4}, {3, 4}}
        {{3, 1}, {3, 4}, {3, 2}, {1, 4}, {1, 2}, {4, 2}}
```

Здесь верхняя строка – множество S всех 2-подмножеств множества {1,2,3,4}, а нижняя строка их образы при действии перестановки p. При этом порядок элементов в множестве не учитывается, то есть множества {1,3}и {3,1} идентичны.
Пусть теперь G - группа перестановок. Очевидно, что H={p(S)/pϵG} также является группой.
Более того для любых $p_1,p_2 \in G$ $p_1(S)\times p_2(S)= ( p_1\times p_2) (S)$, то есть группы G,H гомоморфны.
В общем группа G с бинарной операцией ⋄ гомоморфна группе H с бинарной операцией +
если существует отображение f:G->H, такое, что f(a⋄b)=f(a)+f(b)для любых a, bϵG. В этом случае отображение f называется гомоморфизмом. Легко проверить, что h сохраняет единицу и обратный элемент. Другими словами, h отображает единицу G в единицу H, и h отображает $p^{-1}$ в $(h(p))^{-1}$

Изоморфизм групп – это гомоморфизм групп с дополнительным условием: отображение h должно быть биекцией. Говорят, что группа G действует на множестве s, если существует гомоморфизм из G в группу H перестановок S.
Функция KSubsetGroup[pg, s] возвращает группу, порожденную действием группы перестановок pg на множестве s всех k-подмножеств n-элементного множества. Допускается опция Type, которая может принимать значения Ordered или Unordered.

Рассмотрим группу перестановок, порожденную действием группы $D_4$ на множестве всех 2-подмножеств множества {1,2,3,4}. Поскольку существует шесть 2-подмножеств, она является группой из 6 перестановок.

```
In[3]:= t = KSubsetGroup[G = DihedralGroup[4], KSubsets[Range[4], 2]]
Out[3]= {{1, 2, 3, 4, 5, 6}, {3, 5, 6, 1, 2, 4}, {6, 2, 4, 3, 5, 1}, {4, 5, 1, 6, 2, 3},
        {6, 5, 3, 4, 2, 1}, {4, 2, 6, 1, 5, 3}, {1, 5, 4, 3, 2, 6}, {3, 2, 1, 6, 5, 4}}
```

Таблица умножения показывает, что это множество действительно является группой.

```
In[4]:= MultiplicationTable[t, Permute] // TableForm
```

Out[4]//TableForm=

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 2 | 3 | 4 | 1 | 6 | 7 | 8 | 5 |
| 3 | 4 | 1 | 2 | 7 | 8 | 5 | 6 |
| 4 | 1 | 2 | 3 | 8 | 5 | 6 | 7 |
| 5 | 8 | 7 | 6 | 1 | 4 | 3 | 2 |
| 6 | 5 | 8 | 7 | 2 | 1 | 4 | 3 |
| 7 | 6 | 5 | 8 | 3 | 2 | 1 | 4 |
| 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |

Сформируем теперь множество S упорядоченных 2-подмножеств четырехэлементного множества:

```
In[5]:= S = Select[Flatten[Table[{i, j}, {i, 4}, {j, 1, 4}], 1], #[[1]] ≠ #[[2]] &]
Out[5]= {{1, 2}, {1, 3}, {1, 4}, {2, 1}, {2, 3},
        {2, 4}, {3, 1}, {3, 2}, {3, 4}, {4, 1}, {4, 2}, {4, 3}}
```

Рассмотрим группу перестановок, порожденную действием диэдральной группы на упорядоченных 2-подмножествах.

```
In[6]:= q = KSubsetGroup[DihedralGroup[4], S, Ordered]
Out[6]= {{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}, {10, 11, 12, 3, 1, 2, 6, 4, 5, 9, 7, 8},
         {9, 7, 8, 12, 10, 11, 2, 3, 1, 5, 6, 4}, {5, 6, 4, 8, 9, 7, 11, 12, 10, 1, 2, 3},
         {12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1}, {8, 7, 9, 5, 4, 6, 2, 1, 3, 12, 11, 10},
         {4, 6, 5, 1, 3, 2, 11, 10, 12, 8, 7, 9}, {3, 2, 1, 10, 12, 11, 7, 9, 8, 4, 6, 5}}
```

Функция PairGroup[g] возвращает группу KSubsetGroup при k=2. PairGroup[g] иногда называют парной группой.

```
In[7]:= SameQ[PairGroup[Permutations[7]],
          KSubsetGroup[Permutations[7], KSubsets[Range[7], 2]]]
Out[7]= True
```

Пусть G – группа, действующая на множестве S. По определению действия группы существует гомоморфизм f из G в группу H перестановок множества S. Для любого pєG пусть $p_f$-краткая форма записи f(p). Определим бинарное отношение на S как i≅ j тогда и только тогда, когда $p_f(i)=j$ для некоторого pєG. Орбиты группы G, действующей на S, являются классами эквивалентности по отношению≅ .
Лемма Коши-Фробениуса, часто называемая леммой Бернсайда, утверждает, что число орбит действия группы G на множестве S равно $|G|^{-1} \sum_{p\in G} l(p)$, где l(p)- число неподвижных точек в $p_f$.
Функция Orbits[pg, x] возвращает орбиты группы, порожденной действием группы pg на множестве x. Группа pg - группа перестановок на множестве первых n натуральных чисел, а x - множество функций, определенных на множестве первых n натуральных чисел. Если функция f не объявлена, по умолчанию применяется функция Permute.
Перечислить представителей орбит действия группы, порожденной действием группы на множестве, можно, применив функцию OrbitRepresentatives.
Функция OrbitRepresentatives[pg, x] возвращает представителя из каждой орбиты группы, порожденной действием группы pg на множестве x. Группа pg – группа перестановок на множестве первых n натуральных чисел, а x – множество функций, определенных на множестве первых n натуральных чисел. Если функция f не объявлена, по умолчанию применяется функция Permute.
Рассмотрим классическую задачу перечисления ожерелий. Задача перечисления ожерелий состоит в том, чтобы перечислить общее число различных ожерелий, которые могут быть собраны из n бусин с цветов, предполагая, что мы обеспечены бесконечным множеством бусин каждого цвета. Пара ожерелий считается различными, если одно не может быть получено из другого вращением, отражением или любой комбинацией этих операций.
Пусть нам нужно составить ожерелья из шести бусин двух различных цветов R, G.
Составим строки длины шесть из двух символов R,B:

```
In[8]:= x = Strings[{R, B}, 6]; Length[x]
Out[8]= 64
```

Найдем множество различных шестибусинных ожерелий, составленных из бусин двух цветов:

```
In[9]:= or = OrbitRepresentatives[DihedralGroup[6], x]
Out[9]= {{R, R, R, R, R, R}, {B, B, B, B, B, B}, {B, B, B, B, B, R},
         {B, B, B, B, R, R}, {B, B, B, R, B, R}, {B, B, B, R, R, R},
         {B, B, R, B, B, R}, {B, B, R, B, R, R}, {B, B, R, R, R, R},
         {B, R, B, R, B, R}, {B, R, B, R, R, R}, {B, R, R, B, R, R}, {B, R, R, R, R, R}}
```

Число нужных различных ожерелий равно числу элементов в списке or:

```
In[10]:= Length[or]
Out[10]= 13
```

Подсчитаем число различных ожерелий из шести бус двух цветов, если считать различными такие ожерелья, что одно не может быть получено из другого вращением.

```
In[11]:= or1 = OrbitRepresentatives[CyclicGroup[6], x]
Out[11]= {{R, R, R, R, R, R}, {B, B, B, B, B, B}, {B, B, B, B, B, R},
          {B, B, B, B, R, R}, {B, B, B, R, B, R}, {B, B, B, R, R, R},
          {B, B, R, B, B, R}, {B, B, R, B, R, R}, {B, B, R, R, B, R}, {B, B, R, R, R, R},
          {B, R, B, R, B, R}, {B, R, B, R, R, R}, {B, R, R, B, R, R}, {B, R, R, R, R, R}}

In[12]:= Length[or1]
Out[12]= 14
```

Вычислим длины всех орбит:

```
In[13]:= t = Orbits[DihedralGroup[6], x]; Table[Length[t[[i]]], {i, 1, 13}]
Out[13]= {12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12}
```

Эту задачу можно решить, применяя функции NumberOfNecklaces , NecklacePolynomial или ListNecklaces.
ListNecklaces[n, c, Cyclic] – возвращает все различные ожерелья, окрашенные в цвета из списка c. Список c состоит из n необязательно различных цветов. Два ожерелья считаются одинаковыми, если одно может быть получено из другого вращением.
ListNecklaces[n, c, Dihedral] работает аналогично, но два ожерелья считаются одинаковыми, если одно может быть получено из другого вращением или отражением.
NecklacePolynomial[n, c, Cyclic] возвращает полином от количества цветов c, коэффициенты которого суть число способов окраски ожерелий из n бусин в цвета из списка c, причем два ожерелья считаются одинаковыми, если одно получается из другого вращением. NecklacePolynomial[n, c, Dihedral] отличается только тем, что два ожерелья являются одинаковыми, если одно может быть получено из другого вращением или отражением или их композицией.
NumberOfNecklaces[n, nc, Cyclic] возвращает число различных способов раскраски ожерелий из n бусин в цвета из списка nc, причем два ожерелья являются одинаковыми, если одно может быть получено из другого вращением.
NumberOfNecklaces[n, nc, Dihedral] отличается только тем, что два ожерелья являются одинаковыми, если одно может быть получено из другого вращением или отражением или их композицией.

```
In[14]:= {NumberOfNecklaces[6, 2, Dihedral], NumberOfNecklaces[6, 2, Cyclic]}
Out[14]= {13, 14}
```

Рассмотрим полином NecklacePolynomial от c:

```
In[15]:= NecklacePolynomial[6, c, Cyclic]
Out[15]= c/3 + c^2/3 + c^3/6 + c^6/6
```

Подставляя c=2 , найдем число ожерелий из шести бусин двух цветов:

```
In[16]:= NecklacePolynomial[6, c, Cyclic] /. c → 2
Out[16]= 14
```

Вычислим полиномы ожерелий для ожерелий из шести бусин относительно действия диэдральгой и циклической групп:

In[17]:= **{NecklacePolynomial[6, c, Dihedral], NecklacePolynomial[6, c, Cyclic]}**

Out[17]= $\left\{\frac{c}{6}+\frac{c^2}{6}+\frac{c^3}{3}+\frac{c^4}{4}+\frac{c^6}{12},\ \frac{c}{3}+\frac{c^2}{3}+\frac{c^3}{6}+\frac{c^6}{6}\right\}$

| | |
|---|---|
| KSubsetGroup[pg, s] | возвращает группу, порожденную действием группы перестановок pg на множестве s всех k-подмножеств n-элементного множества. Допускается опция Type , которая может принимать значения Ordered или Unordered |
| PairGroup[g] | возвращает группу KSubsetGroup при k=2 |
| Orbits[pg, x] | возвращает орбиты группы, порожденной действием группы pg на множестве x. Группа pg - группа перестановок на множестве первых n натуральных чисел, а x - множество функций, определенных на множестве первых n натуральных чисел. Если функция f не объявлена, по умолчанию применяется функция Permute |
| OrbitRepresentatives[pg, x] | возвращает представителя из каждой орбиты группы, порожденной действием группы pg на множестве x. Группа pg - группа перестановок на множестве первых n натуральных чисел, а x - множество функций, определенных на множестве первых n натуральных чисел. Если функция f не объявлена, по умолчанию применяется функция Permute |
| ListNecklaces[n, c, Cyclic] | возвращает все различные ожерелья, окрашенные в цвета из списка c. Список c состоит из n необязательно различных цветов. Два ожерелья считаются одинаковыми, если одно может быть получено из другого вращением |
| ListNecklaces[n, c, Dihedral] | работает аналогично, но два ожерелья считаются одинаковыми, если одно может быть получено из другого вращением или отражением |
| NecklacePolynomial[n, c, Cyclic] | возвращает полином от количества цветов c, коэффициенты которого суть число способов окраски ожерелий из n бусин в цвета из списка c, причем два ожерелья считаются одинаковыми, если одно получается из другого вращением |
| NecklacePolynomial[n, c, Dihedral] | Отличается от предыдущей функции только тем, что два ожерелья являются одинаковыми, если одно может быть получено из другого вращением или отражением или их композицией |
| NumberOfNecklaces[n, nc, Cyclic] | возвращает число различных способов раскраски ожерелий из n бусин в цвета из списка nc, причем два ожерелья являются одинаковыми, если одно может быть получено из другого вращением |
| NumberOfNecklaces[n, nc, Dihedral] | отличается от предыдущей функции только тем, что два ожерелья являются одинаковыми, если одно может быть получено из другого вращением или отражением или их композицией |

### *3.3.2. Цикловой индекс группы перестановок*

Цикловым индексом группы перестановок называется полином $Z(G; x_1,x_2,\ldots,x_n)$, определяемый следующим образом:

$Z(G; x_1,x_2,\ldots,x_n)=|G|^{-1}\sum_{\pi\in G} x_1^{\lambda_1(\pi)}x_2^{\lambda_2(\pi)}\ldots\ldots x_n^{\lambda_n(\pi)}$, где $\lambda=(\lambda_1(\pi),\lambda_2(\pi),\ldots,\lambda_n(\pi))$ – тип перестановки $\pi\epsilon G$. Здесь $x_1,x_2,\ldots,x_n$ – произвольные формальные переменные, а $Z(G; x_1,x_2,\ldots,x_n)$ – полином от этих переменных.

CycleIndex[pg, x] возвращает полином от x[1], x[2], ..., x[index[g]], который является цикловым индексом группы перестановок pg. Здесь index[pg] -длина каждой перестановки в группе pg.

```
In[2]:= CycleIndex[Permutations[5], x]
```

$$\text{Out[2]}= \frac{x[1]^5}{120} + \frac{1}{12}x[1]^3 x[2] + \frac{1}{8}x[1]x[2]^2 + \frac{1}{6}x[1]^2 x[3] + \frac{1}{6}x[2]x[3] + \frac{1}{4}x[1]x[4] + \frac{x[5]}{5}$$

SymmetricGroupIndex[n, x] возвращает цикловой индекс симметрической группы от n переменных.

```
In[3]:= SymmetricGroupIndex[5, x]
```

$$\text{Out[3]}= \frac{x[1]^5}{120} + \frac{1}{12}x[1]^3 x[2] + \frac{1}{8}x[1]x[2]^2 + \frac{1}{6}x[1]^2 x[3] + \frac{1}{6}x[2]x[3] + \frac{1}{4}x[1]x[4] + \frac{x[5]}{5}$$

Видим, что CycleIndex[Permutations[5], x] и SymmetricGroupIndex[5, x] дают один и тот же полином, но SymmetricGroupIndex тратит гораздо меньшее время:

```
In[4]:= g = SymmetricGroup[5];
        {Timing[CycleIndex[g, x];], Timing[SymmetricGroupIndex[g, x]];}
Out[4]= {{0.016 Second, Null}, Null}
```

CyclicGroupIndex[n, x] возвращает цикловой индекс циклической группы от n переменных.

```
In[5]:= CyclicGroupIndex[100, x]
```

$$\text{Out[5]}= \frac{x[1]^{100}}{100} + \frac{x[2]^{50}}{100} + \frac{x[4]^{25}}{50} + \frac{x[5]^{20}}{25} + \frac{x[10]^{10}}{25} + \frac{2x[20]^5}{25} + \frac{x[25]^4}{5} + \frac{x[50]^2}{5} + \frac{2x[100]}{5}$$

DihedralGroupIndex[n, x] возвращает цикловой индекс диэдральной группы из n символов, выраженный как полином от x[1], x[2], ..., x[n].

```
In[6]:= DihedralGroupIndex[200, x]
```

$$\text{Out[6]}= \frac{x[1]^{200}}{400} + \frac{1}{4}x[1]^2 x[2]^{99} + \frac{101\,x[2]^{100}}{400} + \frac{x[4]^{50}}{200} + \frac{x[5]^{40}}{100} + \frac{x[8]^{25}}{100} + \frac{x[10]^{20}}{100} + \frac{x[20]^{10}}{50} + \frac{x[25]^8}{20} + \frac{x[40]^5}{25} + \frac{x[50]^4}{20} + \frac{x[100]^2}{10} + \frac{x[200]}{5}$$

Сравним время, затраченное на вычисление циклового индекса с помощью DihedralGroupIndex и CycleIndex:

```
In[7]:= {Timing[DihedralGroupIndex[200, x]]; Timing[CycleIndex[DihedralGroup[200], x];]}
Out[7]= {{1.344 Second, Null}}
```

AlternatingGroupIndex[n, x] возвращает цикловой индекс знакопеременной группы перестановок размера n.

```
In[8]:= AlternatingGroupIndex[6, x]
```

$$\text{Out[8]}= \frac{x[1]^6}{360} + \frac{1}{8}x[1]^2 x[2]^2 + \frac{1}{9}x[1]^3 x[3] + \frac{x[3]^2}{9} + \frac{1}{4}x[2]x[4] + \frac{2}{5}x[1]x[5]$$

KSubsetGroupIndex[g, s, x] возвращает цикловой индекс группы, порожденной группой перестановок, действующей на множестве s k-подмножеств как полином от x. Функция допускает опцию Type, которая может принимать значения Ordered или Unordered

```
In[9]:= KSubsetGroupIndex[CyclicGroup[4], KSubsets[Range[4], 2], x]
```

$$\text{Out[9]}= \frac{x[1]^6}{4} + \frac{1}{4}x[1]^2 x[2]^2 + \frac{1}{2}x[2]x[4]$$

PairGroupIndex[g, x] возвращает цикловой индекс парной группы.
PairGroupIndex[ci, x] берет цикловой индекс ci группы g с формальными переменными x[1], x[2], ..., и возвращает цикловой индекс парной группы , порожденной g.
PairGroupIndex[g, x, Ordered] возвращает цикловой индекс упорядоченной парной группы. PairGroupIndex[ci, x, Ordered] берет цикловой индекс ci группы g с формальными переменными x[1], x[2], ..., и возвращает цикловой индекс упорядоченной парной группы.
Рассмотрим множество s упорядоченных пар первых n натуральных чисел группу q, действующую на s, и вычислим цикловой индекс двумя способами:

```
In[10]:= s = Select[Flatten[Table[{i, j}, {i, 4}, {j, 4}], 1], #[[1]] ≠ #[[2]] &];
         q = KSubsetGroup[CyclicGroup[4], s, Ordered];
         SameQ[PairGroupIndex[CyclicGroup[4], x, Ordered], CycleIndex[q, x]]
Out[11]= True
```

Теперь вычислим цикловой индекс двумя способами. Вначале вычислим парную группу $S_{6,}$ а потом ее цикловой индекс. Затем вычислим цикловой индекс $S_6$, и используем его в PairGroupIndex для вычисления циклового индекса парной группы. Последнее вычисление производится быстрее.

```
In[12]:= {Timing[CycleIndex[PairGroup[SymmetricGroup[6]], x];],
          Timing[PairGroupIndex[CycleIndex[SymmetricGroup[6], x], x];]}
Out[12]= {{0.485 Second, Null}, {0.14 Second, Null}}
```

| | |
|---|---|
| CycleIndex[pg, x] | возвращает полином от x[1], x[2], ..., x[index[g]], который является цикловым индексом группы перестановок pg. Здесь index[pg] -длина каждой перестановки в pg |
| SymmetricGroupIndex[n, x] | возвращает цикловой индекс симметрической группы от n переменных |
| CyclicGroupIndex[n, x] | возвращает цикловой индекс циклической группы от n переменных |
| DihedralGroupIndex[n, x] | возвращает цикловой индекс диэдральной группы из n символов, выраженный как полином от x[1], x[2], ..., x[n] |
| AlternatingGroupIndex[n, x] | возвращает цикловой индекс знакопеременной группы перестановок размера n |
| KSubsetGroupIndex[g, s, x] | возвращает цикловой индекс группы, порожденной группой перестановок, действующей на множестве s k-подмножеств как полином от x. Функция допускает опцию Type, которая может принимать значения Ordered или Unordered |
| PairGroupIndex[g, x] | возвращает цикловой индекс парной группы |
| PairGroupIndex[ci, x] | берет цикловой индекс ci группы g с формальными переменными x[1], x[2], ..., и возвращает цикловой индекс парной группы , порожденной g |
| PairGroupIndex[g, x, Ordered] | PairGroupIndex[g, x, Ordered] возвращает цикловой индекс упорядоченной парной группы |
| PairGroupIndex[ci, x, Ordered] | берет цикловой индекс ci группы g с формальными переменными x[1], x[2], ..., и возвращает цикловой индекс упорядоченной парной группы |

### *3.3.3. Применение теоремы Пойа*

Многие комбинаторные задачи решаются с помощью теоремы Пойа. Пусть G - группа всех перестановок размера n и пусть Ф-множество всех функций ф :{ 1,2,....,n}->R, где R – r- элемент-

ное конечное множество. Заметим, что число функций из Φ равно $r^n$. Группа G действует на множестве Φ. Каждому rϵR сопоставим вес w(r). Вес w(r) может быть как числом, так и символом. Каждой функции фϵΦ сопоставим вес

$$w(\phi) = \prod_i w(\phi(i)),$$

где произведение берется по всем iϵ{1,2,...,n}. Тогда все функции в орбите имеют один и тот же вес. Это позволяет определить вес орбиты C, – w(C) как w(ф) для любой фϵC. Теорема Пойа утверждает, что сумма весов орбит может быть вычислена простой заменой x[i] в цикловом индексе группы G элементами $\sum_{r\in R} w(r)^i$ .

OrbitInventory[ci, x, weights] берет цикловой индекс в переменных x[1], x[2]..... и список weights весов $w_1, w_2$,..... и вычисляет полином, полученный подстановкой в цикловой индекс суммы ($w_1^i+w_2^i$+.....) вместо x[i].

OrbitInventory[ci, x, w] возвращает значение циклового индекса ci , в котором формальные переменные x[1], x[2]... заменены на w .

Если мы присваиваем единичный вес каждому элементу из R, то каждая функция получает единичный вес. Это означает, что каждая орбита получает единичный вес, и поэтому сумма весов в орбите – просто число орбит. Рассмотрим число ожерелий из шести бус из бусин двух цветов:

```
In[2]:= {OrbitInventory[DihedralGroupIndex[6, x], x, 2],
         OrbitInventory[CyclicGroupIndex[6, x], x, 2]}
Out[2]= {13, 14}
```

Тот же результат можно получить, используя функцию Polya:

```
In[3]:= {Polya[DihedralGroup[6], 2], Polya[CyclicGroup[6], 2]}
Out[3]= {13, 14}
```

Рассмотрим группу $D_{12}$, действующую на Φ, множестве функций $\phi$:{1,2,...,6}->{p,q}, где p,q,- цвета бусин, и каждое $\phi$ представляет ожерелье. Предположим, что w(p)=p, w(q)=q. Вычислим сумму весов орбит действия группы $D_{12}$ на Φ. Вначале мы вычислим орбиты $D_{12}$, действующие на Φ, с помощью функции Orbits. Затем вычислим веса орбит и, наконец, суммируем их. По теореме Пойа, полученный полином от p,q идентичен полиному, полученному заменой x[i] в цикловом индексе $D_{12}$ на ($p^i+q^i$):

```
In[4]:= Apply[Plus, Map[Apply[Times, #[[1]]] &, Orbits[DihedralGroup[6], Strings[{p, q}, 6]]]]
Out[4]= p^6 + p^5 q + 3 p^4 q^2 + 3 p^3 q^3 + 3 p^2 q^4 + p q^5 + q^6
```

Теперь подсчитаем то же самое с помощью встроенной функции:

```
In[5]:= OrbitInventory[DihedralGroupIndex[6, x], x, {p, q}]
Out[5]= p^6 + p^5 q + 3 p^4 q^2 + 3 p^3 q^3 + 3 p^2 q^4 + p q^5 + q^6
```

| | |
|---|---|
| OrbitInventory[ci, x, weights ] | берет цикловой индекс в переменных x[1], x[2]..... и список weights весов $w_1, w_2$,.....и вычисляет полином, полученный подстановкой в цикловой индекс суммы ($w_1^i+w_2^i$+.....) вместо x[i] |
| OrbitInventory[ci, x, w] | возвращает значение циклового индекса ci , в котором формальные переменные x[1], x[2]..... заменены на w |
| Polya[g,m] | возвращает полином степени m (m цветов), структуры, определяемой группой перестановок g. Лучше использовать OrbitInventory. |

# Глава 4. Разбиения, композиции и табло Янга

## ■ 4.1. Разбиения

### *4.1. Порождение разбиений. Число разбиений.*

**Разбиением** целого положительного числа n называется множество из к целых строго положительных чисел, чья сумма равна n.
Встроеннная функция PartitionQ[p] возвращает True, если p-целое разбиение.
PartitionQ[n,p]- возвращает True, если p-целое разбиение n.

```
In[2]:= {PartitionQ[{2, 2, 2}], PartitionQ[{2, 2, 2.2}], PartitionQ[4, {2, 2, 2}]}
Out[2]= {True, False, False}
```

Функция Partitions[n] конструирует все разбиения целого числа n в обратном лексикографическом порядке. Partitions[n, k] конструирует все разбиения целого числа n с элементами разбиения, не превосходящими k , в обратном лексикографическом порядке.
Приведем все разбиения числа 8. Заметим, что они следуют в обратном лексикографическом порядке.

```
In[3]:= Partitions[8]
Out[3]= {{8}, {7, 1}, {6, 2}, {6, 1, 1}, {5, 3}, {5, 2, 1}, {5, 1, 1, 1}, {4, 4},
         {4, 3, 1}, {4, 2, 2}, {4, 2, 1, 1}, {4, 1, 1, 1, 1}, {3, 3, 2}, {3, 3, 1, 1},
         {3, 2, 2, 1}, {3, 2, 1, 1, 1}, {3, 1, 1, 1, 1, 1}, {2, 2, 2, 2}, {2, 2, 2, 1, 1},
         {2, 2, 1, 1, 1, 1}, {2, 1, 1, 1, 1, 1, 1}, {1, 1, 1, 1, 1, 1, 1, 1}}
```

Теперь выведем все разбиения числа 8, каждый элемент которых не превосходит 5.

```
In[4]:= Partitions[8, 5]
Out[4]= {{5, 3}, {5, 2, 1}, {5, 1, 1, 1}, {4, 4}, {4, 3, 1}, {4, 2, 2}, {4, 2, 1, 1}, {4, 1, 1, 1, 1},
         {3, 3, 2}, {3, 3, 1, 1}, {3, 2, 2, 1}, {3, 2, 1, 1, 1}, {3, 1, 1, 1, 1, 1}, {2, 2, 2, 2},
         {2, 2, 2, 1, 1}, {2, 2, 1, 1, 1, 1}, {2, 1, 1, 1, 1, 1, 1}, {1, 1, 1, 1, 1, 1, 1, 1}}
```

Функция NumberOfPartitions [n] возвращает число целых разбиений целого числа n, это число обозначается $P_{n,k.}$
Хотя число разбиений растет экспоненциально, этот рост гораздо медленнее, чем рост числа перестановок, поэтому Mathematica очень быстро считает число разбиений для больших значений n.

```
In[5]:= {Length[Partitions[20]], NumberOfPartitions[20]}

Out[5]= {627, 627}
```

Составим таблицу значений для $P_{n,k}$ для n=1, 2, 3, 4,…,10 и $k \le n$.

```
In[6]:= Table[NumberOfPartitions[i, j], {i, 10}, {j, i}] // ColumnForm
```

```
Out[6]= {1}
        {1, 2}
        {1, 2, 3}
        {1, 3, 4, 5}
        {1, 3, 5, 6, 7}
        {1, 4, 7, 9, 10, 11}
        {1, 4, 8, 11, 13, 14, 15}
        {1, 5, 10, 15, 18, 20, 21, 22}
        {1, 5, 12, 18, 23, 26, 28, 29, 30}
        {1, 6, 14, 23, 30, 35, 38, 40, 41, 42}
```

Combinatorica вычисляет число разбиений для очень больших чисел за считанные секунды:

```
In[7]:= Timing[NumberOfPartitions[2000]]
Out[7]= {0.797 Second, 472081917561941388860143240679995951220034416 6}
```

Функция NextPartition[p] возвращает разбиение, следующее за p в множестве разбиений, перечисленных в обратном лексикографическом порядке.

```
In[8]:= NextPartition[{4, 2, 2}]
Out[8]= {4, 2, 1, 1}
```

Сформируем множество всех разбиений числа 8, перечисленных в обратном лексикографическом порядке.

```
In[9]:= t = Table[1, {8}]; s = Table[t = NextPartition[t], {NumberOfPartitions[8]}]
Out[9]= {{8}, {7, 1}, {6, 2}, {6, 1, 1}, {5, 3}, {5, 2, 1}, {5, 1, 1, 1}, {4, 4},
         {4, 3, 1}, {4, 2, 2}, {4, 2, 1, 1}, {4, 1, 1, 1, 1}, {3, 3, 2}, {3, 3, 1, 1},
         {3, 2, 2, 1}, {3, 2, 1, 1, 1}, {3, 1, 1, 1, 1, 1}, {2, 2, 2, 2}, {2, 2, 2, 1, 1},
         {2, 2, 1, 1, 1, 1}, {2, 1, 1, 1, 1, 1, 1}, {1, 1, 1, 1, 1, 1, 1, 1}}

In[10]:= SameQ[s, Partitions[8]]
Out[10]= True
```

RandomPartition[n] возвращает случайное разбиение целого положительного числа n.

```
In[11]:= RandomPartition[100]
Out[11]= {35, 12, 11, 8, 4, 4, 4, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
```

TransposePartition[p] берет разбиение p натурального n, состоящее из k частей и отражает его относительно центрального элемента множества разбиений Partition[n], создавая разбиение с максимальной частью k.

```
In[12]:= {Partitions[5], Map[TransposePartition, Partitions[5]]} // ColumnForm
Out[12]= {{5}, {4, 1}, {3, 2}, {3, 1, 1}, {2, 2, 1}, {2, 1, 1, 1}, {1, 1, 1, 1, 1}}
         {{1, 1, 1, 1, 1}, {2, 1, 1, 1}, {2, 2, 1}, {3, 1, 1}, {3, 2}, {4, 1}, {5}}
```

Покажем, что списки Partitions[5] и список транспонированных разбиений симметричны относительно центрального элемента списков:

```
In[13]:= Map[SameQ,
          Table[{Partitions[5][[i]],
            Map[Transposition, Partitions[5]][[NumberOfPartitions[5] + 1 - i]]},
           {i, 1, NumberOfPartitions[5]}]]
Out[13]= {True, True, True, True, True, True, True}
```

Рассмотрим целые разбиения $\lambda=(\lambda_1, \lambda_2,\ldots, \lambda_k)$ и $\mu=(\mu_1, \mu_2,\ldots,\mu_j)$ некоторого положительного целого числа n. Будем говорить, что разбиение $\lambda$ **доминирует** над разбиением $\mu$, если $\lambda_1+\lambda_2+\ldots+\lambda_t \geq \mu_1+\mu_2+\ldots+\mu_t$ для всех $t\geq 1$.
DominatingIntegerPartitionQ[a, b] возвращает True, если целое разбиение a доминирует над целым разбиением b.
Выберем все разбиения числа 7, которые доминируют над разбиением {2,2,2,1}:

```
In[14]:= Select[Partitions[7], DominatingIntegerPartitionQ[#, {2, 2, 2, 1}] &]
Out[14]= {{7}, {6, 1}, {5, 2}, {5, 1, 1}, {4, 3}, {4, 2, 1},
          {4, 1, 1, 1}, {3, 3, 1}, {3, 2, 2}, {3, 2, 1, 1}, {2, 2, 2, 1}}
```

Разбиение числа n связано с типом n-перестановки. Существует ровно столько типов n-перестановок, сколько существует разбиений числа n.
Рассмотрим разбиения числа 4.

```
In[15]:= p = Partitions[5]
Out[15]= {{5}, {4, 1}, {3, 2}, {3, 1, 1}, {2, 2, 1}, {2, 1, 1, 1}, {1, 1, 1, 1, 1}}
```

Чтобы построить список всех типов перестановок из восьми элементов, конвертируем каждое разбиение в тип перестановки.

```
In[16]:= t1 = Table[t = {0, 0, 0, 0, 0}; Scan[t[[#]]++ &, p[[i]]]; t, {i, Length[p]}]
Out[16]= {{0, 0, 0, 0, 1}, {1, 0, 0, 1, 0}, {0, 1, 1, 0, 0},
          {2, 0, 1, 0, 0}, {1, 2, 0, 0, 0}, {3, 1, 0, 0, 0}, {5, 0, 0, 0, 0}}
```

Подсчитаем число перестановок каждого типа:

```
In[17]:= Map[{#, NumberOfPermutationsByType[#]} &, t1] // ColumnForm
Out[17]= {{0, 0, 0, 0, 1}, 24}
         {{1, 0, 0, 1, 0}, 30}
         {{0, 1, 1, 0, 0}, 20}
         {{2, 0, 1, 0, 0}, 20}
         {{1, 2, 0, 0, 0}, 15}
         {{3, 1, 0, 0, 0}, 10}
         {{5, 0, 0, 0, 0}, 1}
```

| | |
|---|---|
| PartitionQ[p] | возвращает True, если p-целое разбиение |
| PartitionQ[n,p]- | возвращает True, если p-целое разбиение n |
| Partitions[n] | конструирует все разбиения целого числа n в обратном лексикографическом порядке |
| Partitions[n, k] | конструирует все разбиения целого числа n с элементами разбиения, не превосходящими k , в обратном лексикографическом порядке |
| NumberOfPartitions [n] | считает число целых разбиений целого числа n, это число обозначается $P_{n,k}$ |
| NextPartition[p] | возвращает разбиение, следующее за p в множестве разбиений, перечисленных в обратном лексикографическом порядке |
| RandomPartition[n] | возвращает случайное разбиение целого положительного |

| | |
|---|---|
| | числа n |
| TransposePartition[p] | берет разбиение p натурального n, состоящее из k частей и отражает его относительно центрального элемента множества разбиений Partition[n] , создавая разбиение с максимальной частью k |
| DominatingIntegerPartitionQ[a, b] | возвращает True, если целое разбиение a доминирует над целым разбиением b |

### *4.2. Диаграммы Феррерса*

Диаграммы Феррерса представляет разбиения натурального n как шаблон из точек. Каждому $k_i$ из разбиения $\{k_1,k_2,\ldots,k_s\}$ соответствуют ровно $k_i$ точек, расположенных в одной строке. FerrersDiagram[p] изображает диаграмму Феррерса разбиения p.

```
In[2]:= p = RandomPartition[30]
Out[2]= {9, 6, 4, 3, 1, 1, 1, 1, 1, 1, 1}
```

Построим диаграмму Феррерса разбиения p. Первая строка должна содержать 9 точек, вторая – 6 точек, третья -4 точки и т.д.:

```
In[3]:= FerrersDiagram[p]
```

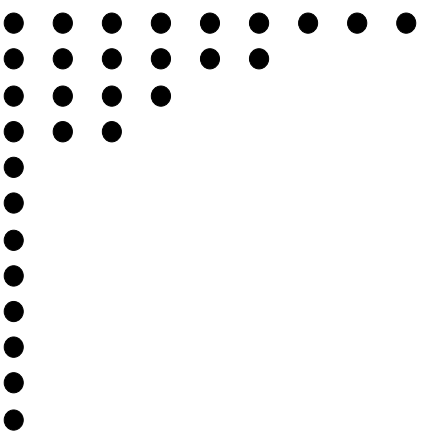
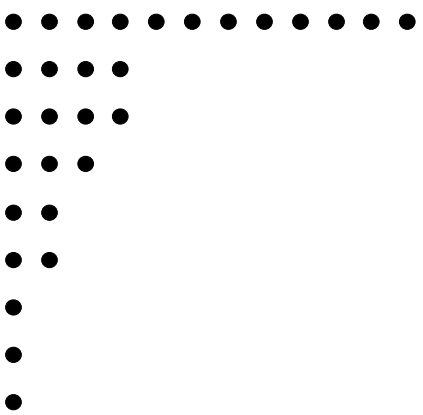


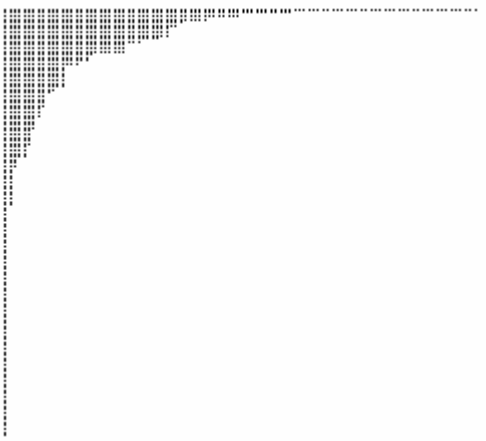
Диаграмма Феррерса показывает, что число разбиений натурального числа n с максимальной частью k равно числу разбиений целого числа n с k частями. Транспонируем разбиение p и затем построим его диаграмму Феррерса:

```
In[4]:= FerrersDiagram[TransposePartition[p]]
```



Combinatorica строит диаграмму Феррерса даже для разбиений достаточно больших n:

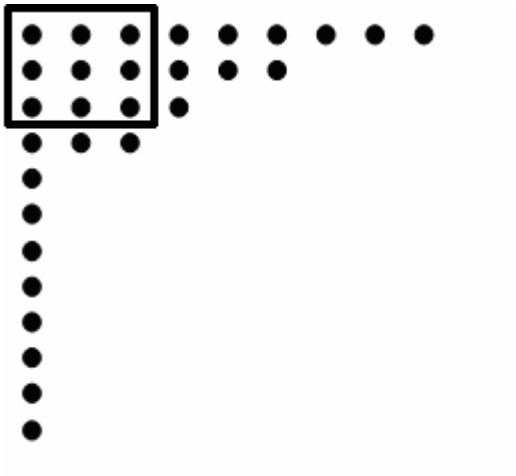
```
In[5]:= FerrersDiagram[RandomPartition[1000]]
```



Функция DurfeeSquare[p] возвращает размер наибольшего квадрата, содержащегося в диаграмме Феррерса разбиения p.

```
In[6]:= DurfeeSquare[p]
Out[6]= 3
```

Действительно, сторона наибольшего квадрата, содержащего точки диаграммы Феррерса разбиения {9, 6, 4, 3, 1, 1, 1, 1, 1, 1, 1, 1} равен 3. Это квадрат 3×3 в левом верхнем углу диаграммы.



| | |
|---|---|
| FerrersDiagram[p] | изображает диаграмму Феррерса разбиения p |
| DurfeeSquare[p] | возвращает размер наибольшего квадрата, содержащегося в диаграмме Феррерса разбиения p |

## ■4. 2. Композиции

### *4.2.1. Порождение композиций. Случайные композиции*

Композиция целого положительного числа n - это множество из к целых **неотрицательных** чисел, чья сумма равна n.
Отличие композиций от разбиений в том, что у композиций могут быть нулевые элементы.
Compositions[n, k] возвращает список всех композиций целого n на k частей.

```
In[2]:= Compositions[5, 3]
Out[2]= {{0, 0, 5}, {0, 1, 4}, {0, 2, 3}, {0, 3, 2}, {0, 4, 1}, {0, 5, 0}, {1, 0, 4},
         {1, 1, 3}, {1, 2, 2}, {1, 3, 1}, {1, 4, 0}, {2, 0, 3}, {2, 1, 2}, {2, 2, 1},
         {2, 3, 0}, {3, 0, 2}, {3, 1, 1}, {3, 2, 0}, {4, 0, 1}, {4, 1, 0}, {5, 0, 0}}
```

NumberOfCompositions[n, k] считает число различных композиций натурального n на k частей. Вычислим таблицу значений для композиций натурального n на k частей при n=1,2,….,10 и k≤n. Элементы на главной диагонали есть общее число композиций n на n частей.

```
In[3]:= Table[NumberOfCompositions[i, j], {i, 1, 10}, {j, i}] // ColumnForm
Out[3]= {1}
        {1, 3}
        {1, 4, 10}
        {1, 5, 15, 35}
        {1, 6, 21, 56, 126}
        {1, 7, 28, 84, 210, 462}
        {1, 8, 36, 120, 330, 792, 1716}
        {1, 9, 45, 165, 495, 1287, 3003, 6435}
        {1, 10, 55, 220, 715, 2002, 5005, 11440, 24310}
        {1, 11, 66, 286, 1001, 3003, 8008, 19448, 43758, 92378}
```

Функция RandomComposition[n, k] формирует случайную композицию натурального n на k частей.

```
In[4]:= RandomComposition[200, 50]
Out[4]= {0, 10, 2, 7, 4, 0, 3, 10, 10, 1, 0, 0, 1, 0, 1, 3, 1, 3, 4, 8, 5, 2, 17, 0,
         1, 9, 5, 13, 1, 2, 2, 2, 3, 0, 1, 6, 3, 6, 24, 0, 1, 6, 3, 2, 5, 5, 2, 4, 0, 2}
```

NextComposition[l] возвращает композицию натурального n, следующую за композицией l в множестве Compositions[n, k] каноническом порядке.
Перечислим все композиции числа 6 на 3 части с помощью NextComposition:

```
In[5]:= NestList[NextComposition, {0, 0, 6}, 28]
Out[5]= {{0, 0, 6}, {0, 1, 5}, {0, 2, 4}, {0, 3, 3}, {0, 4, 2}, {0, 5, 1}, {0, 6, 0}, {1, 0, 5},
         {1, 1, 4}, {1, 2, 3}, {1, 3, 2}, {1, 4, 1}, {1, 5, 0}, {2, 0, 4}, {2, 1, 3},
         {2, 2, 2}, {2, 3, 1}, {2, 4, 0}, {3, 0, 3}, {3, 1, 2}, {3, 2, 1}, {3, 3, 0},
         {4, 0, 2}, {4, 1, 1}, {4, 2, 0}, {5, 0, 1}, {5, 1, 0}, {6, 0, 0}, {0, 0, 6}}
```

| | |
|---|---|
| Compositions[n, k] | возвращает список всех композиций целого n на k частей |
| NumberOfCompositions[n, k] | считает число различных композиций натурального n на k частей |
| RandomComposition[n, k] | формирует случайную композицию натурального n на k частей |
| NextComposition[l] | конструирует композицию натурального n , следующую за композицией l в каноническом порядке |

## ■4.3. Разбиение множества

### *4.3.1. Порождение разбиений множества*

Разбиения множества есть разбиение элементов множества на подмножества. Порядок, в котором перечисляются подмножества, и порядок элементов в этих подмножествах не имеет значения. Элемент разбиения множества обычно называют частью или блоком.

SetPartitions[set] возвращает список разбиений множества set.
SetPartitions[n] возвращает список разбиений множества {1,2,…,n}.

```
In[2]:= SetPartitions[{a, b, c, d}]
Out[2]= {{{a, b, c, d}}, {{a}, {b, c, d}}, {{a, b}, {c, d}}, {{a, c, d}, {b}},
        {{a, b, c}, {d}}, {{a, d}, {b, c}}, {{a, b, d}, {c}}, {{a, c}, {b, d}},
        {{a}, {b}, {c, d}}, {{a}, {b, c}, {d}}, {{a}, {b, d}, {c}}, {{a, b}, {c}, {d}},
        {{a, c}, {b}, {d}}, {{a, d}, {b}, {c}}, {{a}, {b}, {c}, {d}}}
```

SetPartitionQ[sp, s] определяет, является ли список sp разбиением множества s.
SetPartitionQ[sp] определяет, является ли sp множеством непересекающихся подмножеств.
Множество {{a,c},{d,e},{f},{b}} является разбиением множества {a,b,c,d,e,f}, но не является разбиением {a,b,c,d}:

```
In[3]:= {SetPartitionQ[{{a, c}, {d, e}, {f}, {b}}, {a, b, c, d, e, f}],
         SetPartitionQ[{{a, c}, {d, e}, {f}, {b}}, {a, b, c, d}]}
Out[3]= {True, False}
```

Список {{a,c},{d,c},{f},{b}} не является разбиением множества, так как первое и второе подмножества пересекаются.

```
In[4]:= SetPartitionQ[{{a, c}, {d, c}, {f}, {b}}, {a, b, c, d, e, f}]
 Out[4]= False
```

Проверим, пересекаются ли множества {1,2,3,4,5}и {a,b,c}:

```
In[5]:= SetPartitionQ[{{a, b, c}, {1, 2, 3, 4, s, d}}]
Out[5]= True
```

ToCanonicalSetPartition[sp, set] переупорядочивает список sp в каноническом порядке относительно списка set. В каноническом порядке означает, что элементы каждого подмножества в разбиении упорядочены в том порядке, в котором они находятся в set, а подмножества упорядочиваются в порядке их первых элементов.
ToCanonicalSetPartition[sp] переупорядочивает sp в каноническом порядке, предполагая, что Mathematica знает порядок множества, для которого sp является разбиением.

```
In[6]:= ToCanonicalSetPartition[{{a, d}, {b}, {c}}, {b, c, a, d}]
Out[6]= {{b}, {c}, {a, d}}
```

Переупорядочим список SetPartitions[{a,b,c,d}] в порядке{b,c,a,d}, то есть считем, что b<c<a<d:

```
In[7]:= Map[ToCanonicalSetPartition[#, {b, c, a, d}] &, SetPartitions[{a, b, c, d}]]
Out[7]= {{{b, c, a, d}}, {{b, c, d}, {a}}, {{b, a}, {c, d}}, {{b}, {c, a, d}},
        {{b, c, a}, {d}}, {{b, c}, {a, d}}, {{b, a, d}, {c}}, {{b, d}, {c, a}},
        {{b}, {c, d}, {a}}, {{b, c}, {a}, {d}}, {{b, d}, {c}, {a}}, {{b, a}, {c}, {d}},
        {{b}, {c, a}, {d}}, {{b}, {c}, {a, d}}, {{b}, {c}, {a}, {d}}}
```

| | |
|---|---|
| SetPartitions[set] | возвращает список разбиений множества set |
| SetPartitions[n] | возвращает список разбиений множества {1,2,…,n} |
| SetPartitionQ[sp, s] | определяет, является ли список sp разбиением множества s |
| SetPartitionQ[sp] | определяет, является ли sp множеством непересекающихся подмножеств |
| ToCanonicalSetPartition[sp, | переупорядочивает список sp в каноническом порядке относи- |

| | |
|---|---|
| set] | тельно списка set. В каноническом порядке элементы каждого подмножества в разбиении упорядочены в том порядке, в котором они находятся в set, а подмножества упорядочиваются в порядке их первых элементов |
| ToCanonicalSetPartition[sp] | переупорядочивает sp в каноническом порядке, предполагая, что Mathematica знает порядок множества, для которого sp является разбиением |

### *4.3.2. Разбиение множества на k-подмножества*

KSetPartitions[set, k] возвращает разбиение множества set на k частей.
KSetPartitions[n, k] возвращает разбиение множества {1,2,….,n} на k частей.
Рассмотрим разбиение множества {a,b,c,d,e} на два множества.

```
In[2]:= KSetPartitions[{a, b, c, d, e}, 2]
Out[2]= {{{a}, {b, c, d, e}}, {{a, b}, {c, d, e}}, {{a, c, d, e}, {b}},
         {{a, b, c}, {d, e}}, {{a, d, e}, {b, c}}, {{a, b, d, e}, {c}}, {{a, c}, {b, d, e}},
         {{a, b, c, d}, {e}}, {{a, e}, {b, c, d}}, {{a, b, e}, {c, d}}, {{a, c, d}, {b, e}},
         {{a, b, c, e}, {d}}, {{a, d}, {b, c, e}}, {{a, b, d}, {c, e}}, {{a, c, e}, {b, d}}}
```

Число разбиений n- элементного множества на два подмножества равно $2^{n-1}$-1:

```
In[3]:= {Map[Length, Table[KSetPartitions[i, 2], {i, 1, 10}]],
          Table[(2^(n - 1)) - 1, {n, 1, 10}]} // ColumnForm
Out[3]= {0, 1, 3, 7, 15, 31, 63, 127, 255, 511}
        {0, 1, 3, 7, 15, 31, 63, 127, 255, 511}
```

Число разбиений множества {1,2,3,...,n} на к блоков называется числом Стирлинга второго рода.
Вычислим число разбиений n-элементного множества на k блоков при n=1,…,10:

```
In[4]:= Table[StirlingSecond[n, k], {n, 10}, {k, 1, n}] // ColumnForm
Out[4]= {1}
        {1, 1}
        {1, 3, 1}
        {1, 7, 6, 1}
        {1, 15, 25, 10, 1}
        {1, 31, 90, 65, 15, 1}
        {1, 63, 301, 350, 140, 21, 1}
        {1, 127, 966, 1701, 1050, 266, 28, 1}
        {1, 255, 3025, 7770, 6951, 2646, 462, 36, 1}
        {1, 511, 9330, 34105, 42525, 22827, 5880, 750, 45, 1}
```

Combinatorica позволяет считать число разбиений очень больших множеств:

```
In[5]:= StirlingSecond[200, 100]

Out[5]= 22839435964738549264941860239810502575992576012385773346189261281146257736498595608736896568075780615586778324044855917353941596234632122679102898580700882344102094584094303303456156354505852418666317069775795452784560323333501899075 56
```

Чтобы вычислить число Стирлинга второго рода можно воспользоваться встроенной функцией StirlingS2.

```
In[6]:= StirlingS2[200, 100]

Out[6]= 228394359647385492649418602398105025759925760123857733461892612811462577364985956087×
        368965680757806155867783240448559173539415962346321226791028985807008823441020945 84×
        094303303456156354505852418666317069775795452784560323333501899075 56
```

Общее число разбиений множества {1,2,...n} называется n-м числом Белла и обозначается $B_n$.
Функция BellB[n] возвращает n-е число Белла:
Вычисли число разбиений 200- элементного множества:

```
In[7]:= BellB[500]

Out[7]= 160607260103999145374373286046550778629192454664500124922145864703660903169238874226×
        453306837738154752608395670137495550103762064426599148582399756042391947225367315 42×
        877114522434826259434253245235878076832220161634826021276374628321106317547158830 59×
        004999887672474956910305625687386159330549479316789123608152541634373822059048622 68×
        519694464560733867201285632186417439144562222755925311694024672179283672281903035 51×
        224065103356932350609229342605167255200156770306847016242592054728359001944534021 89×
        046985409254748392090704758491594261624209127147911854676983943733984734941560341 13×
        380195089364116722935354329869916808816319793326636136117174956780625221057798069 55×
        696728013492932766714566494075171880028357031076491191614894597598751539382954829 60×
        189635009623545528574680095812422777380790576825931823862858389709617386930741651 34×
        5394229457772
```

Вычислим это же число суммированием чисел Стирлинга второго рода:

```
In[8]:= Sum[StirlingSecond[500, k], {k, 500}]

Out[8]= 160607260103999145374373286046550778629192454664500124922145864703660903169238874226×
        453306837738154752608395670137495550103762064426599148582399756042391947225367315 42×
        877114522434826259434253245235878076832220161634826021276374628321106317547158830 59×
        004999887672474956910305625687386159330549479316789123608152541634373822059048622 68×
        519694464560733867201285632186417439144562222755925311694024672179283672281903035 51×
        224065103356932350609229342605167255200156770306847016242592054728359001944534021 89×
        046985409254748392090704758491594261624209127147911854676983943733984734941560341 13×
        380195089364116722935354329869916808816319793326636136117174956780625221057798069 55×
        696728013492932766714566494075171880028357031076491191614894597598751539382954829 60×
        189635009623545528574680095812422777380790576825931823862858389709617386930741651 34×
        5394229457772
```

Сравним время вычисления этого числа:

```
In[9]:= {Timing[BellB[500];], Timing[Sum[StirlingSecond[500, k], {k, 300}];]}

Out[9]= {{0. Second, Null}, {1.312 Second, Null}}
```

Функция RandomSetPartition[set] возвращает случайное разбиение множества set.
RandomSetPartition[n] возвращает случайное разбиение первых n натуральных чисел.
RandomKSetPartition[set, k] возвращает случайное разбиение множества set на k блоков.
RandomKSetPartition[n, k] возвращает случайное разбиение первых n натуральных чисел на k блоков.
Рассмотрим случайно выбранное разбиение множества {1,2,….,10} и случайно выбранное разбиение множества {1,2,…,10} с пятью блоками:

```
In[10]:= {RandomSetPartition[10], RandomKSetPartition[10, 5]} // ColumnForm
Out[10]= {{1, 2, 3, 4, 7, 8, 9}, {5, 6, 10}}
         {{1, 7, 8, 10}, {2, 4}, {3, 5}, {6}, {9}}
```

Покажем распределение числа блоков в 1000 случайно выбранных разбиений множества {1,2,...10}. Распределение вначале возрастает, а затем убывает.

```
In[11]:= Distribution[Map[Length, Table[RandomSetPartition[10], {1000}]]]
Out[11]= {7, 80, 301, 380, 180, 45, 7}
```

RankSetPartition[sp, s] возвращает ранг разбиения sp в списке всех разбиений множества s.
RankSetPartition[sp] возвращает ранг разбиения sp в списке всех разбиений множества являющегося объединением элементов sp.
RankKSetPartition[sp, s] возвращает ранг разбиения sp в списке всех k-блоков разбиения множества s.
RankKSetPartition[sp] возвращает ранг разбиения sp в списке всех разбиений множества, являющегося объединением элементов sp на k-блоки

```
In[12]:= RankSetPartition[{{a, d}, {b}, {c}}]
Out[12]= 13
```

С помощью RankSetPartition вычислим ранги разбиений в множестве SetPartitions[{a,b,c,d,e}]:

```
In[13]:= Map[RankSetPartition, SetPartitions[{a, b, c, d, e}]]
Out[13]= {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
          18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34,
          35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51}
```

Ранг sp в списке семиэлементных разбиений больше, чем ее ранг в списке таких разбиений с ровно 4 блоками

```
In[14]:= sp = {{a, f}, {b}, {g, c, d}, {e, h}};
         {RankKSetPartition[sp], RankSetPartition[sp]}
Out[14]= {652, 1746}
```

Разница между двумя рангами в вышеприведенном примере равна точно числу разбиений множества {a,b,c,d,e,f,g,h} с одним, двумя или тремя блоками.

```
In[15]:= {RankSetPartition[sp] - RankKSetPartition[sp], Sum[StirlingSecond[8, i], {i, 1, 3}]}
Out[15]= {1094, 1094}
```

Функции UnrankKSetPartition и UnrankSetPartition являются обратными к RankKSetPartition и RankSetPartition соответственно.
UnrankKSetPartition [r, s, k] находит k-блок разбиения ранга r множества s.
UnrankKSetPartition [r, n, k] находит k-блок разбиения ранга r множества {1, 2,..., n}.
UnrankSetPartition[r, set] находит разбиение ранга r множества set.
UnrankSetPartition[r, n] находит разбиение ранга r множества {1, 2,..., n}.
С помощью UnrankSetPartition получим множество всех разбиений множества {a,b,c,d}

```
In[16]:= t = Table[UnrankSetPartition[i, {a, b, c, d}], {i, BellB[4] - 1}]
```

```
Out[16]= {{{a}, {b, c, d}}, {{a, b}, {c, d}}, {{a, c, d}, {b}},
         {{a, b, c}, {d}}, {{a, d}, {b, c}}, {{a, b, d}, {c}}, {{a, c}, {b, d}},
         {{a}, {b}, {c, d}}, {{a}, {b, c}, {d}}, {{a}, {b, d}, {c}}, {{a, b}, {c}, {d}},
         {{a, c}, {b}, {d}}, {{a, d}, {b}, {c}}, {{a}, {b}, {c}, {d}}}
```

Функции UnrankSetPartition и RankSetPartition взаимно обратные:

```
In[17]:= Map[RankSetPartition, t]
Out[17]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}
```

Выделим 501-е разбиение в множестве разбиений на 5 блоков множества первых двадцати натуральных чисел:

```
In[18]:= UnrankKSetPartition[500, Range[20], 5]
Out[18]= {{1}, {2}, {3}, {4, 7, 8, 10, 12}, {5, 6, 9, 11, 13, 14, 15, 16, 17, 18, 19, 20}}
```

Функция CoarserSetPartitionQ[a, b] возвращает True, если разбиение множества b грубее, чем разбиение множества a, т.е. каждый блок a, содержится в некотором блоке в b.

```
In[19]:= CoarserSetPartitionQ[{{a}, {b}, {c}}, {{a}, {b, c}}]
Out[19]= True
```

| | |
|---|---|
| KSetPartitions[set, k] | возвращает разбиение множества set на k частей |
| KSetPartitions[n, k] | возвращает разбиение множества {1,2,….,n} на k частей |
| StirlingSecond[n, k] | возвращает число Стирлинга второго рода |
| StirlingS2[n, m] | возвращает число Стирлинга второго рода |
| StirlingFirst[n, k] | возвращает число Стирлинга первого рода |
| StirlingS1[n, m] | возвращает число Стирлинга первого рода |
| BellB[n] | возвращает n-е число Белла |
| RandomSetPartition[set] | возвращает случайное разбиение множества set |
| RandomKSetPartition[set, k] | возвращает случайное разбиение множества set на k блоков |
| RandomKSetPartition[n, k] | возвращает случайное разбиение первых n натуральных чисел на k блоков |
| RankSetPartition[sp, s] | возвращает ранг разбиения sp в списке всех разбиений множества s |
| RankSetPartition[sp] | возвращает ранг разбиения sp в списке всех разбиений множества являющегося объединением элементов sp |
| RankKSetPartition[sp, s] | возвращает ранг разбиения sp в списке всех k-блоков разбиения множества s |
| RankKSetPartition[sp] | возвращает ранг разбиения sp в списке всех разбиений множества являющегося объединением элементов sp на k-блоки |
| UnrankKSetPartition [r, s, k] | находит k-блок разбиения ранга r множества s |
| UnrankKSetPartition [r, n, k] | находит k-блок разбиения ранга r множества {1, 2,..., n} |
| UnrankSetPartition[r, set] | находит разбиение ранга r множества set |
| UnrankSetPartition[r, n] | находит разбиение ранга r множества {1, 2,..., n} |
| CoarserSetPartitionQ[a, b] | возвращает True, если разбиение множества b грубее, чем разбиение множества a, т.е. каждый блок a, содержится в некотором блоке в b |

### *4.2.4. Разбиения множеств и ограниченные возрастающие функции*

Существует биекция между разбиениями множества и некоторыми структурами, называемыми ограниченными возрастающими функциями (Restricted Growth Function, сокращенно RGF).
Функция, действующая из множества {1,2,....n} в множество {1,2,....n}, называется ограниченной возрастающей функцией, если она удовлетворяет условиям:

1. f(1)=1
2. $f(i) \le \max_{1 \le j < i} \{f(j)\} + 1$ для 1<i≤ n

Последнее условие «ограничивает» рост функции требованием, что f(i) по крайней мере на 1 больше, чем максимум предыдущих значений функции. RGF. f задается как последовательность (f(1),f(2),….,f(n)).
Функция RGFs[n] перечисляет все ограниченные возрастающие функции на первых n натуральных числах в лексикографическом порядке. Чтобы увидеть биекцию, возьмем разбиение S множества {1,2,...n} в канонической форме. Тогда положим f(i)=j, если элемент i принадлежит блоку j в S.
Рассмотрим все RGF на множестве {1,2,3,4}, перечисленные в лексикографическом порядке:

```
In[2]:= RGFs[4]
Out[2]= {{1, 1, 1, 1}, {1, 1, 1, 2}, {1, 1, 2, 1}, {1, 1, 2, 2}, {1, 1, 2, 3},
         {1, 2, 1, 1}, {1, 2, 1, 2}, {1, 2, 1, 3}, {1, 2, 2, 1}, {1, 2, 2, 2},
         {1, 2, 2, 3}, {1, 2, 3, 1}, {1, 2, 3, 2}, {1, 2, 3, 3}, {1, 2, 3, 4}}
```

Функция RGFQ[l] возвращает True, если l является RGF, и False в противном случае.

```
In[3]:= {RGFQ[{1, 1, 2}], RGFQ[{1, 3, 2}]}
Out[3]= {True, False}
```

Функция RGFToSetPartition[rgf, set] конвертирует RGF в соответствующее разбиение множества set. Если второй аргумент set с опциями отсутствует, то RGF конвертируется в множество разбиений множества {1, 2, ..., Length[rgf]}.
Рассмотрим разбиения, соответствующие RGF {1,1,1,2,3} и RGF {1,2,2} множества {a,b,c} в канонической форме.

```
In[4]:= {RGFToSetPartition[{1, 1, 1, 2, 3}], RGFToSetPartition[{1, 2, 2}, {a, b, c}]}
Out[4]= {{{1, 2, 3}, {4}, {5}}, {{a}, {b, c}}}
```

Функция SetPartitionToRGF[sp, set] конвертирует разбиение множества sp в соответствующую RGF. Если второй аргумент set отсутствует, то предполагается, что Mathematica знает порядок множества set, для которого sp является разбиением.

Рассмотрим RGF, соответствующие вышеприведенным разбиениям :

```
In[5]:= {ToCanonicalSetPartition[{{1, 2, 3}, {4}, {5}}],
         ToCanonicalSetPartition[{{a}, {b, c}}, {a, b, c}]}
Out[5]= {{{1, 2, 3}, {4}, {5}}, {{a}, {b, c}}}
```

RandomRGF[n] возвращает случайную (RGF), определенную на множестве первых n натуральных чисел.
RandomRGF[n, k] возвращает случайную (RGF), определенную на множестве первых n натуральных чисел, имеющую максимальный элемент равный к.

```
In[6]:= {f = RandomRGF[10], RGFToSetPartition[f]} // ColumnForm
```

```
Out[6]= {1, 2, 3, 4, 2, 1, 1, 4, 4, 2}
        {{1, 6, 7}, {2, 5, 10}, {3}, {4, 8, 9}}
```

Число блоков в разбиении равно максимальному значению, которое принимает соответствующая RGF:

```
In[7]:= {Length[p = RandomSetPartition[100]], Max[SetPartitionToRGF[p]]}
Out[7]= {25, 25}
```

Функция RankRGF[f] возвращает ранг RGF f в лексикографическом порядке среди всех RGFs.
UnrankRGF[r, n] возвращает RGF, определенные на первых n натуральных числах, чей ранг равен r.

```
In[8]:= Table[UnrankRGF[i, 4], {i, 0, 14}]
Out[8]= {{1, 1, 1, 1}, {1, 1, 1, 2}, {1, 1, 2, 1}, {1, 1, 2, 2}, {1, 1, 2, 3},
         {1, 2, 1, 1}, {1, 2, 1, 2}, {1, 2, 1, 3}, {1, 2, 2, 1}, {1, 2, 2, 2},
         {1, 2, 2, 3}, {1, 2, 3, 1}, {1, 2, 3, 2}, {1, 2, 3, 3}, {1, 2, 3, 4}}
```

Сформируем RGFs[4] с помощью RankRGF:

```
In[9]:= Map[RankRGF, RGFs[4]]
Out[9]= {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}
```

SetPartitionListViaRGF[n] перечисляет все разбиения на множестве первых n натуральных чисел, вначале перечисляя все RGFs на них, а затем отображает RGFs на соответствующие разбиения множества.
SetPartitionListViaRGF[n, k] перечисляет все RGFs на множестве первых n натуральных чисел, чей максимальный элемент -к, а затем отображает эти RGFs в соответствующее множество разбиений , каждое из которых содержит ровно к блоков.

```
In[10]:= SetPartitionListViaRGF[4]
Out[10]= {{{1, 2, 3, 4}}, {{1, 2, 3}, {4}}, {{1, 2, 4}, {3}}, {{1, 2}, {3, 4}},
          {{1, 2}, {3}, {4}}, {{1, 3, 4}, {2}}, {{1, 3}, {2, 4}}, {{1, 3}, {2}, {4}},
          {{1, 4}, {2, 3}}, {{1}, {2, 3, 4}}, {{1}, {2, 3}, {4}}, {{1, 4}, {2}, {3}},
          {{1}, {2, 4}, {3}}, {{1}, {2}, {3, 4}}, {{1}, {2}, {3}, {4}}}
```

Этот же список можно вычислить по определению:

```
In[11]:= SameQ[Map[RGFToSetPartition, RGFs[5]], SetPartitionListViaRGF[5]]
Out[11]= True
```

| | |
|---|---|
| RGFs[n] | перечисляет все ограниченные возрастающие функции на первых n натуральных числах в лексикографическом порядке |
| RGFQ[l] | возвращает True, если l является RGF, и False в противном случае |
| RGFToSetPartition[rgf, set] | конвертирует RGF в соответствующее разбиение множества set. Если второй аргумент set с опциями отсутствует, то RGF конвертируется в множество разбиений множества {1, 2, ..., Length[rgf]} |
| SetPartitionToRGF[sp, set] | конвертирует разбиение множества sp в соответствующую RGF. Если второй аргумент set отсутствует, то предполагается, что Mathematica знает порядок множества set, для которого sp является разбиением |
| RandomRGF[n] | возвращает случайную (RGF), определенную на множестве пер- |

| | |
|---|---|
| | вых n натуральных чисел |
| RandomRGF[n, k] | возвращает случайную (RGF), определенную на множестве первых n натуральных чисел, имеющую максимальный элемент равный к |
| RankRGF[f] | возвращает ранг RGF f в лексикографическом порядке среди всех RGFs |
| UnrankRGF[r, n] | возвращает RGF, определенные на первых n натуральных числах, чей ранг равен r |
| SetPartitionListViaRGF[n] | перечисляет все разбиения на множестве первых n натуральных чисел, вначале перечисляя все RGFs на них, а затем отображает RGFs на соответствующие разбиения множества |
| SetPartitionListViaRGF[n, k] | перечисляет все RGFs на множестве первых n натуральных чисел, чей максимальный элемент -к, а затем отображает эти RGFs в соответствующее множество разбиений , каждое из которых содержит ровно к блоков |

## ■4.4. Табло Янга

### *4.4.1. Порождение табло Янга*

Рассмотрим $n_1+n_2+\ldots+n_m$ различных целых чисел ($n_1 \geq n_2 \geq \ldots \geq n_m > 0$), расположенных в m строках по $n_i$ элементов в i-ой строке, причем элементы в каждой строке и столбце расположены в возрастающем порядке. Такая таблица называется табло Янга формы $(n_1,n_2,\ldots,n_m)$.
Функция Tableaux[p] конструирует все табло заданной формы, где форма задается разбиением числа p.

```
In[2]:= Map[Tableaux, Partitions[4]] // ColumnForm
Out[2]= {{{1, 2, 3, 4}}}
        {{{1, 3, 4}, {2}}, {{1, 2, 4}, {3}}, {{1, 2, 3}, {4}}}
        {{{1, 3}, {2, 4}}, {{1, 2}, {3, 4}}}
        {{{1, 4}, {2}, {3}}, {{1, 3}, {2}, {4}}, {{1, 2}, {3}, {4}}}
        {{{1}, {2}, {3}, {4}}}
```

Выделим все 16 табло формы {3,2,1}:

```
In[3]:= Map[ColumnForm, Tableaux[{3, 2, 1}]]
Out[3]= {{1, 4, 6}, {1, 3, 6}, {1, 2, 6}, {1, 3, 6},
         {2, 5}     {2, 5}     {3, 5}     {2, 4}
         {3}        {4}        {4}        {5}

        {1, 2, 6}, {1, 4, 5}, {1, 3, 5}, {1, 2, 5}, {1, 3, 4}, {1, 2, 4},
        {3, 4}     {2, 6}     {2, 6}     {3, 6}     {2, 6}     {3, 6}
        {5}        {3}        {4}        {4}        {5}        {5}

        {1, 2, 3}, {1, 3, 5}, {1, 2, 5}, {1, 3, 4}, {1, 2, 4}, {1, 2, 3}}
        {4, 6}     {2, 4}     {3, 4}     {2, 5}     {3, 5}     {4, 5}
        {5}        {6}        {6}        {6}        {6}        {6}
```

Функция TableauQ[t] возвращает True, если t – табло Янга и False в противном случае.
Рассмотрим t – разбиение первых семи натуральных чисел:

```
In[4]:= t = {{1, 2, 4}, {3, 7}, {5}, {6}} // TableForm
```

```
Out[4]//TableForm=
        1       2       4
        3       7
        5
        6
```

Этот тест подтверждает, что это табло.

```
In[5]:= t = {{1, 2, 4}, {3, 7}, {5}, {6}}; TableauQ[t]
Out[5]= True
```

А это не табло Янга, так как во второй строке элементы расположены не в возрастающем порядке.

```
In[6]:= TableauQ[{{1, 2, 5, 9, 10}, {5, 4, 7, 12}, {4, 8, 13}, {14}}]
Out[6]= False
```

Функция FirstLexicographicTableau[p] конструирует первое табло Янга формы разбиения p.
Первое лексикографическое табло состоит из столбцов соприкасающихся натуральных чисел:

```
In[7]:= FirstLexicographicTableau[{4, 4, 3, 3, 2}] // TableForm
Out[7]//TableForm=
        1       6       11      15
        2       7       12      16
        3       8       13
        4       9       14
        5       10
```

Функция LastLexicographicTableau[p] конструирует последнее табло Янга формы данного разбиения целого числа p.
Последнее лексикографическое табло состоит из строк соприкасающихся целых чисел.

```
In[8]:= LastLexicographicTableau[{4, 4, 3, 3, 2}] // TableForm
Out[8]//TableForm=
        1       2       3       4
        5       6       7       8
        9       10      11
        12      13      14
        15      16
```

Функция NextTableau[t] возвращает табло, следующее за t в лексикографическом порядке.

```
In[9]:= NextTableau[{{1, 2, 6}, {3, 4, 7}, {5}}]
Out[9]= {{1, 4, 5}, {2, 6, 7}, {3}}
```

| | |
|---|---|
| Tableaux[p] | конструирует все табло заданной формы, где форма задается разбиением числа p |
| TableauQ[t] | возвращает True, если t – табло Янга и False в противном случае |
| FirstLexicographicTableau[p] | конструирует первое табло Янга формы разбиения p в Tableaux[p] |
| LastLexicographicTableau[p] | конструирует последнее табло Янга формы разбиения целого числа p в Tableaux[p] |
| NextTableau[t] | возвращает табло, следующее за t в лексикографическом порядке |

### *4.4.2. Вставка элемента в табло и удаление элемента из табло*

Предположим, мы имеем табло Янга T и элемент x, не содержащийся в T. Нам нужно вставить элемент x и получить новое табло. Попытаемся вставить x в первую строку T. Если элемент x больше, чем последний элемент первой строки, то x может быть помещен за ним, получая результатом новое табло. Если нет, то ищем в строке наименьший элемент y, который больше, чем x. Элемент x занимает место y в этой строке и перебрасывает y на следующую строку, где процедура вставки начинает заново - с вставки элемента y. Этот алгоритм называется **«прыгающим алгоритмом».**
InsertIntoTableau[e, t] вставляет целое число e в табло Янга t, используя «прыгающий алгоритм».
InsertIntoTableau[e, t, All] вставляет элемент e в табло Янга t и возвращает новое табло Янга и номер строки, размер которой увеличился за счет вставки.

Рассмотрим табло, не содержащее 3:

```
In[2]:= TableForm[t = {{1, 2, 7, 8}, {4, 5}, {6, 9}}]

Out[2]//TableForm=
        1      2      7      8
        4      5
        6      9

In[3]:= TableauQ[t]
Out[3]= True
```

Вставка 3 в табло отбрасывает 7 из первой строки и, таким образом, прибавляет элемент во вторую строку.

```
In[4]:= TableForm[InsertIntoTableau[3, t]]

Out[4]//TableForm=
        1      2      3      8
        4      5      7
        6      9
```

Теперь вставим 3 в табло. Вывод показывает номер строки, размер которой увеличился за счет вставки. Это вторая строка.

```
In[5]:= InsertIntoTableau[3, {{1, 2, 7}, {3, 4}, {6}}, All]
Out[5]= {{{1, 2, 3}, {3, 4, 7}, {6}}, 2}
```

DeleteFromTableau[t, r] удаляет последний элемент в r-ой строке табло t.
Удалим последний элемент из второй строки:

```
In[6]:= TableForm[DeleteFromTableau[t, 2]]

Out[6]//TableForm=
        1      5      7      8
        4
        6      9
```

Пусть дана перестановка p. Combinatorica позволяет строить табло Янга, ассоциированное с перестановкой p, используя алгоритм вставки, начиная с пустого табло, и вставляя каждый элемент перестановки в табло в порядке p(1),p(2),....p(n). Каждая перестановка порождает табло, не обязательно единственное.

Функция ConstructTableau[p] для каждого элемента перестановки p несколько раз повторяет «прыгающий алгоритм», и возвращает результат – табло Янга, ассоциированное с перестановкой p.

```
In[7]:= Map[ConstructTableau, Permutations[4]]
Out[7]= {{{1, 2, 3, 4}}, {{1, 2, 3}, {4}}, {{1, 2, 4}, {3}}, {{1, 2, 4}, {3}},
        {{1, 2, 3}, {4}}, {{1, 2}, {3}, {4}}, {{1, 3, 4}, {2}}, {{1, 3}, {2, 4}},
        {{1, 3, 4}, {2}}, {{1, 3, 4}, {2}}, {{1, 3}, {2, 4}}, {{1, 3}, {2}, {4}},
        {{1, 2, 4}, {3}}, {{1, 2}, {3, 4}}, {{1, 4}, {2}, {3}}, {{1, 4}, {2}, {3}},
        {{1, 2}, {3, 4}}, {{1, 4}, {2}, {3}}, {{1, 2, 3}, {4}}, {{1, 2}, {3}, {4}},
        {{1, 3}, {2}, {4}}, {{1, 3}, {2}, {4}}, {{1, 2}, {3}, {4}}, {{1}, {2}, {3}, {4}}}
```

Как уже упоминалось, построенное из данной перестановки табло не обязательно единственное:

```
In[8]:= {ConstructTableau[{6, 4, 9, 7, 8, 1, 2}], ConstructTableau[{6, 4, 9, 5, 7, 1, 2, 8}]}
Out[8]= {{{1, 2, 8}, {4, 7}, {6, 9}}, {{1, 2, 7, 8}, {4, 5}, {6, 9}}}
```

Между перестановками и упорядоченными парами табло (P,Q) существует биективное соответствие. Из любой данной n-перестановки p можно построить пару табло (P,Q) идентичной формы, содержащие элементы от 1 до n.
Рассмотрим случайную перестановку:

```
In[9]:= p = RandomPermutation[10]
Out[9]= {7, 1, 9, 5, 10, 3, 8, 2, 6, 4}
```

Сконструируем соответствующую пару табло. Эти два табло одной и той же формы с содержат элементы от 1 до n.

```
In[39]:= ({P, Q} = PermutationToTableaux[p]) // ColumnForm

Out[39]= {{1, 2, 6, 9, 10}, {3, 5, 7}, {4}, {8}}
         {{1, 2, 3, 4, 6}, {5, 8, 10}, {7}, {9}}
```

Функция TableauxToPermutation[t1, t2] возвращает перестановку, ассоциированную с парой табло t1, t2, которые имеют одну и ту же форму. Для t1, t2 такая перестановка единственная.
Из полученной пары табло P,Q получим исходную перестановку:

```
In[10]:= ({P, Q} = PermutationToTableaux[p]) // ColumnForm
Out[10]= {{1, 2, 4}, {3, 6, 10}, {5, 8}, {7, 9}}
         {{1, 3, 5}, {2, 4, 7}, {6, 9}, {8, 10}}
```

Важная черта этой биекции состоит в том, что перестановка p соответствует паре табло (P,Q) тогда и только тогда, когда перестановка $p^{-1}$ соответствует паре (Q,P ).
Рассмотрим перестановку, обратную p, и соответствующую ей пару табло. Если сравнить их с P,Q то эти пары являются отражениями друг друга.

```
In[11]:= q = InversePermutation[p]; PermutationToTableaux[q]
Out[11]= {{{1, 3, 5}, {2, 4, 7}, {6, 9}, {8, 10}}, {{1, 2, 4}, {3, 6, 10}, {5, 8}, {7, 9}}}
```

Если перестановка является инволюцией то есть $p=p^{-1}$, то пары (P,Q) и (Q,P) совпадают.
Рассмотрим инволюцию p:

```
In[12]:= p = FromCycles[{{1, 3}, {5}, {2, 6}, {4, 7}, {8}}]
Out[12]= {3, 6, 1, 7, 5, 2, 4, 8}
```

Действительно, соответствующая пара табло одна и та же.

```
In[13]:= PermutationToTableaux[p] // ColumnForm
Out[13]= {{1, 2, 4, 8}, {3, 5, 7}, {6}}
         {{1, 2, 4, 8}, {3, 5, 7}, {6}}
```

Функция RandomTableau[p] конструирует случайное табло Янга формы p.

```
In[14]:= TableForm[RandomTableau[{6, 5, 5, 4, 3, 2, 2, 1}]]
Out[14]//TableForm=
1    2    7    12   14   18
3    5    8    21   22
4    6    13   25   26
9    11   20   27
10   17   24
15   23
16   28
19
```

Функция NumberOfTableaux[p] возвращает число табло Янга данной формы p.
Подсчитаем число всех табло формы, заданной Partitions[10].

```
In[15]:= Map[NumberOfTableaux, Partitions[10]]
Out[15]= {1, 9, 35, 36, 75, 160, 84, 90, 315, 225, 350, 126, 42, 288,
          450, 567, 525, 448, 126, 252, 300, 210, 768, 525, 300, 567, 350,
          84, 210, 252, 450, 225, 288, 315, 160, 36, 42, 90, 75, 35, 9, 1}
```

Функция TransposeTableau[t] отражает табло относительно главной диагонали (строки табло становятся столбцами).

```
In[16]:= TransposeTableau[{{1, 2, 7, 8}, {4, 5}, {6, 9}}]
Out[16]= {{1, 4, 6}, {2, 5, 9}, {7}, {8}}
```

Табло Янга конструируется с помощью алгоритма вставки, первый элемент вначале вставляется в некоторую позицию первой строки, с которой он в дальнейшем может быть вытолкнут. Элементы, которые находились в i-м столбце, называются принадлежащими i-му классу табло.
TableauClasses[p] разбивает элементы перестановки p на классы в соответствии их первоначальным столбцам во время конструирования табло Янга.

```
In[17]:= TableauClasses[{6, 4, 9, 5, 7, 1, 2, 8, 3}]
Out[17]= {{1, 4, 6}, {2, 5, 9}, {3, 7}, {8}}
```

| | |
|---|---|
| InsertIntoTableau[e, t] | вставляет целое число e в табло Янга t, используя «прыгающий алгоритм» |
| InsertIntoTableau[e, t, All] | вставляет элемент e в табло Янга t и возвращает новое табло Янга и номер строки, размер которой увеличился за счет вставки |
| DeleteFromTableau[t, r] | удаляет последний элемент в r-й строке табло t |
| ConstructTableau[p] | для каждого элемента перестановки p повторяет несколько раз «прыгающий алгоритм», и возвращает результат – табло Янга, ассоциированное с перестановкой p |
| TableauxToPermutation[t1, t2] | TableauxToPermutation[t1, t2] возвращает единственную перестановку, ассоциированную с парой табло t1, t2, которые имеют одну и ту же форму |
| PermutationToTableaux[p] | возвращает пару табло, которая может быть сконструирована из перестановки p. |

| | |
|---|---|
| RandomTableau[p] | конструирует случайное табло Янга формы p |
| NumberOfTableaux[p] | возвращает число табло Янга данной формы p |
| TableauClasses[p] | разбивает элементы перестановки p на классы в соответствии их первоначальным столбцам во время конструирования табло Янга. |

# Глава 5. Построение графов

## ■ 5.1. Основные понятия и определения

Граф g=(E,V) состоит из двух множеств: конечного множества элементов, называемых **вершинами**, и конечного множества элементов, называемых **ребрами**. Каждое ребро определяется парой вершин. Если ребра графа определяются упорядоченными парами вершин, то граф называется **ориентированным** графом. В противном случае g называется неориентированным графом.

Для обозначения вершин графа будем использовать символы $v_1$, $v_2$, $v_3$,... а для обозначения ребер $e_1$, $e_2$, $e_3$,.... Заметим, что во множестве E допускается более чем одно ребро с одинаковыми концевыми вершинами. Все ребра с одинаковыми концевыми вершинами называются **параллельными** или кратными ребрами. Кроме того, концевые вершины ребра не обязательно различны. Если $e_i=(v_i, v_i)$, то ребро $e_i$ называется **петлей.** Граф называется **простым**, если он не содержит петель и параллельных ребер. Если граф имеет петли или параллельные ребра, то он называется **псевдографом.** Граф g называется графом порядка n, если множество его вершин состоит из n элементов. Граф, не имеющий ребер, называется **пустым**. Граф, не имеющий вершин (и, следовательно, ребер), называется **нуль-графом.**

Графически граф может быть представлен диаграммой, в которой вершина изображена точкой или кружком, а ребро - отрезком линии, соединяющей точки или кружки, соответствующие концевым вершинам ребра.

Сформируем множество вершин v и множество ребер e графа g:

```
In[2]:= v = {{-0.5, -0.5}, {0, -1}, {0, 0}, {1, 0}, {2, 0}, {2, -1}};
        e = {{2, 3}, {3, 4}, {5, 5}, {5, 6}, {2, 3}};
```

Зададим неориентированный граф g из пар списка e:

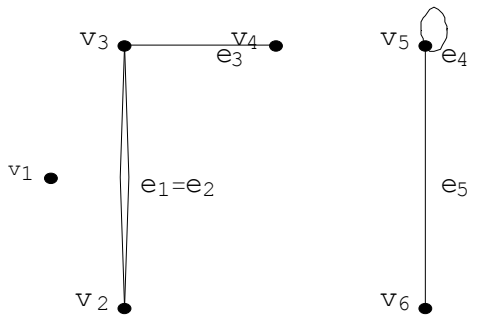
```
In[4]:= g = FromUnorderedPairs[e, v]
Out[4]= -Graph:<5, 6, Undirected>-
```

Ячейка вывода показывает, что g – неориентированный граф с пятью ребрами и шестью вершинами.
Пометим вершины графа множеством меток s, а ребра - метками l:

```
In[5]:= s = {"v1", "v2", "v3", "v4", "v5", "v6"};
        l = {"e1=e2", "", "e3", "e4", "e5"};
```

С помощью функции ShowGraph[g] (см. следующий пункт) изобразим граф g:

```
In[7]:= ShowGraph[g, VertexLabel → s, EdgeLabel → l,
          EdgeLabelPosition → LowerRight, VertexLabelPosition → UpperLeft,
          TextStyle → {FontSize → 13}, PlotRange → 0.25]
```




В этом графе $e_1$, $e_2$ - параллельные ребра, $e_4$ - петля.
Combinatorica обеспечена функциями – тестами MultipleEdgeQ, SelfLoopsQ, которые проверяют граф на наличие параллельных ребер и петель соответственно.Функции SimpleQ, EmptyQ, PseudographQ проверяют, является ли граф простым, пустым или псевдографом соответственно.

| | |
|---|---|
| MultipleEdgesQ[g] | возвращает True, если g имеет параллельные ребра, и False в противном случае |
| SelfLoopsQ[g] | возвращает True, если g имеет петли, и False в противном случае |
| SimpleQ[g] | возвращает True, если g – простой граф, и False в противном случае |
| EmptyQ[g] | возвращает True, если g – пустой граф, и False в противном случае |
| UndirectedQ[g] | возвращает True, если g – неориентированный граф, и False в противном случае |

Протестируем построенный граф g:

```
In[8]:= {MultipleEdgesQ[g], SelfLoopsQ[g], SimpleQ[g], EmptyQ[g],
         PseudographQ[g], UndirectedQ[g]}
Out[8]= {True, True, False, False, True, True}
```

Говорят, что ребро **инцидентно** своим концевым вершинам, а две вершины **смежны**, если они являются концевыми вершинами некоторого ребра. Если два ребра имеют общую концевую вершину, они называются **смежными**. Например, в графе g ребро $e_3$ инцидентно вершинам $v_3$, $v_4$; вершины $v_5$, $v_6$ – смежные вершины; ребра $e_1$, $e_3$ – смежные ребра.

Число инцидентных вершине ребер называется **степенью** вершины. Вершина степени 1 называется **висячей (концевой)** вершиной. Вершина степени 0 называется **изолированной.** По определению петля при вершине добавляет 2 в степень соответствующей вершины.
Встроенные функции Degrees[g], DegreeSequences[g], DegreesOf2Neighborhood[g,v] возвращают степени вершин:

| | |
|---|---|
| Degrees[g] | возвращает степени вершин 1,2,3... в этом же порядке |
| DegreeSequences[g] | возвращает упорядоченную в невозрастающем порядке последовательность вершин графа g |
| DegreesOf2Neighborhood[g,v] | возвращает упорядоченную в неубывающем порядке последовательность степеней вершин, находящихся в пределах двух ребер от вершины v |

Рассмотрим последовательность степеней вершин построенного графа g:

```
In[9]:= {Degrees[g], DegreeSequence[g]} // ColumnForm
Out[9]= {0, 2, 3, 1, 2, 1}
        {3, 2, 2, 1, 1, 0}
```

Теперь рассмотрим степени вершин, находящихся на расстоянии двух ребер от вершины i (i=1,2,…,6):

```
In[10]:= Table[DegreesOf2Neighborhood[g, i], {i, 1, 6}]
Out[10]= {{0}, {1, 2, 3}, {1, 2, 3}, {1, 2, 3}, {1, 2}, {1, 2}}
```

Заметим, что $v_1$ – изолированная вершина, $v_6$, $v_4$ – висячие вершины, $e_3$ - висячее ребро.
Для степеней вершин справедливы следующие свойства:

**Теорема 1.1** Сумма степеней вершин графа g равна 2m, где m-число ребер графа g.

**Теорема 2.2** Число вершин нечетной степени в любом графе четно.

## ■5.2. Изображение графов

### *5.2.1. Функции ShowGraph, ShowLabeledGraph*

ShowGraph[g], ShowLabeledGraph[g] – функции Combinatorica, которые изображают граф.
Первый аргумент этих функций – графический объект, а оставшиеся аргументы – опции.

| | |
|---|---|
| ShowGraph[g] | изображает граф g |
| ShowGraph[g, options] | модифицирует изображение графа g с помощью опций options |
| ShowLabeledGraph[g] | изображает помеченный граф g с метками 1,2,.. |

Рассмотрим опции этих функций:

```
In[7]:= Options[ShowGraph]
Out[7]= {VertexColor → RGBColor[0., 0., 0.], VertexStyle → Disk[Normal],
         VertexNumber → False, VertexNumberColor → RGBColor[0., 0., 0.],
         VertexNumberPosition → LowerLeft, VertexLabel → False,
         VertexLabelColor → RGBColor[0., 0., 0.],
         VertexLabelPosition → UpperRight, PlotRange → Normal, AspectRatio → 1,
         EdgeColor → RGBColor[0., 0., 0.], EdgeStyle → Normal, EdgeLabel → False,
         EdgeLabelColor → RGBColor[0., 0., 0.], EdgeLabelPosition → LowerLeft,
         LoopPosition → UpperRight, EdgeDirection → False}
```

Здесь мы вывели семнадцать опций, принимаемых функциями, изображающими граф. Приведем опции вершин:

| Название опции | Значение по умолчанию | Определение |
|---|---|---|
| VertexColor | Black | устанавливает цвет вершин |
| VertexStyle | Disk[Normal] | устанавливает размер и форму вершин |

| | | |
|---|---|---|
| VertexNumber | False | скрывает или открывает номера вершин |
| VertexNumberColor | Black | устанавливает цвет номеров вершин |
| VertexNumberPosition | LowerLeft | устанавливает позиции номеров вершин |
| VertexLabel | False | скрывает или присваивает метки вершин |
| VertexLabelColor | Black | устанавливает цвет меток |
| VertexLabelPosition | UpperRight | устанавливает позиции меток вершин |

Следующая таблица дает возможные значения для каждой опции вершин для функций, изображающих графы:

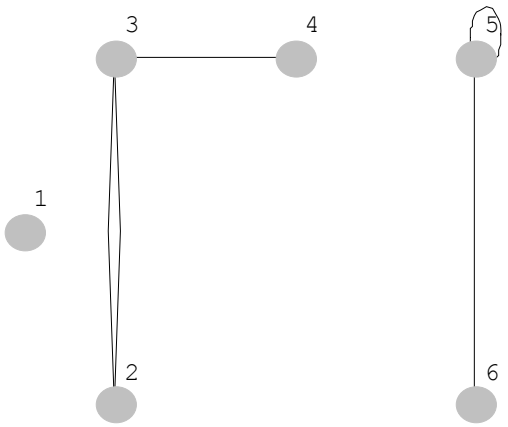
| Название опции | Возможные значения |
|---|---|
| VertexColor | любой цвет, определенный в пакете Graphics`Color` |
| VertexStyle | X[Y], где X может быть Disk или Box, а Y может быть Small, Normal, Large или неотрицательное вещественное число |
| VertexNumber | True или False |
| VertexNumberColor | любой цвет, определенный в пакете Graphics`Color` |
| VertexNumberPosition | Center, LowerLeft, UpperRight, LowerRight, UpperLeft, или отступ, заданный упорядоченной парой вещественных чисел [x,y] |
| VertexLabel | False или список меток, где метки присваиваются вершинам по порядку |
| VertexLabelColor | любой цвет, определенный в пакете Graphics`Color` |
| VertexLabelPosition | Center, LowerLeft, UpperRight, LowerRight, UpperLeft или упорядоченная пара вещественных чисел [x, y] |

Переменная AllColor в пакете расширения Graphics`Color ` содержит имена всех 192 цветов, поддержанных в Mathematica. Однако многие из них не устанавливаются на обычном экране. Всякий раз, когда требуется окрасить объекты, такие как вершины и ребра, Combinatorica использует следующее подмножество цветов: Black, Red, Blue, Green, Yellow, Purple, Brown, Orange, Olive, Pink, DeepPink, DarkGreen, Maroon, Navy.

Кроме перечисленных опций ShowGraph наследует все опции, которые могут быть использованы во встроенной функции Mathematica - Plot. Эти опции могут быть использованы для изменения цвета фона, изображения надписи заголовка, изменения масштаба изображения в некоторой области и т.д. Различные стили и соответствующие опции для изображения стрелок наследуются из пакета расширения Graphics`Arrow`.

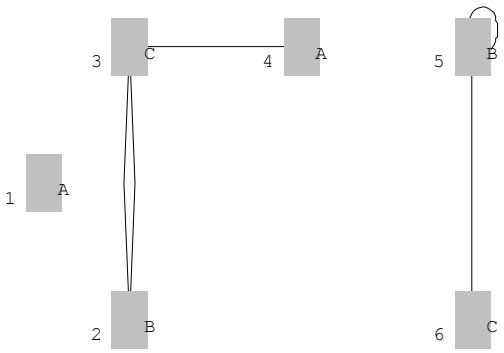
Изобразим вершины построенного графа g в виде квадратиков серого цвета, сдвигая номера вершин на {0.05, 0.05}. Для того чтобы номера вершин поместились на изображаемом рисунке, PlotRange увеличен на 15%. PlotRange и TextStyle – опции встроенной функции Mathematica – Plot.

```
In[8]:= ShowGraph[g, VertexNumber → True, VertexNumberPosition → {0.05, 0.05},
          VertexStyle → Disk[Large], VertexColor → Gray,
          TextStyle → {FontSize → 13}, PlotRange → 0.15]
```




Чтобы установить и изобразить метки вершин, используется опция VertexLabel. Если вершин больше, чем меток, как в нижеприведенном примере, метки используются циклически. По умолчанию позиция меток - UpperRight, но чтобы установить их ниже и справа от вершин, здесь используется опция VertexLabelPosition.

```
In[11]:= ShowGraph[g, VertexNumber → True, VertexNumberPosition → {-0.04, -0.04},
          VertexStyle → Box[Large], VertexColor → Gray,
          VertexLabel → {"A", "B", "C"}, VertexLabelPosition → LowerRight,
          PlotRange → 0.1, TextStyle → {FontSize → 13}]
```




Следующая таблица описывает опции, принимаемые ShowGraph, которые применяются к изображаемым ребрам, и атрибуты, относящиеся к ребрам.

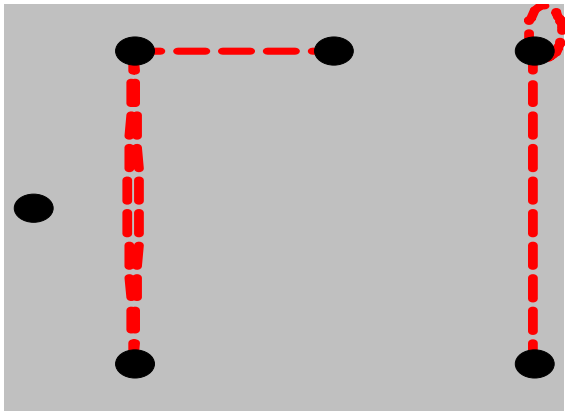
| Название опции | Значение по умолчанию | определение |
|---|---|---|
| EdgeColor | Black | устанавливает цвет ребер |
| EdgeStyle | Normal | устанавливает стиль ребер |
| EdgeLabel | False | скрывает или устанавливает метки ребер |
| EdgeLabePosition | LowerRight | устанавливает позиции меток ребер |
| EdgeLabelColor | Black | устанавливает цвет меток ребер |
| LoopPosition | Upper Right | устанавливает позиции петель |
| EdgeDirection | False | изображает ребра как ориентированные, так и неориентированные |

Приведем возможные значения для опций ребер

| Название опции | Возможные значения |
|---|---|
| EdgeColor | любой цвет, определенный в пакете Graphics`Color` |
| EdgeStyle | Thick, Normal, Thin, ThickDashed, NormalDashed или ThinDashed |
| EdgeLabel | False или список меток, где метки присваиваются ребрам по порядку |
| EdgeLabelColor | любой цвет, определенный в пакете Graphics`Color` |
| EdgeLabelPosition | Center, LowerLeft, UpperRight, LowerRight, UpperLeft, или отступ, заданный упорядоченной парой вещественных чисел [x,y] |
| LoopPosition | UpperRight, UpperLeft, LowerLeft, LowerRight |
| EdgeDirection | True или False |

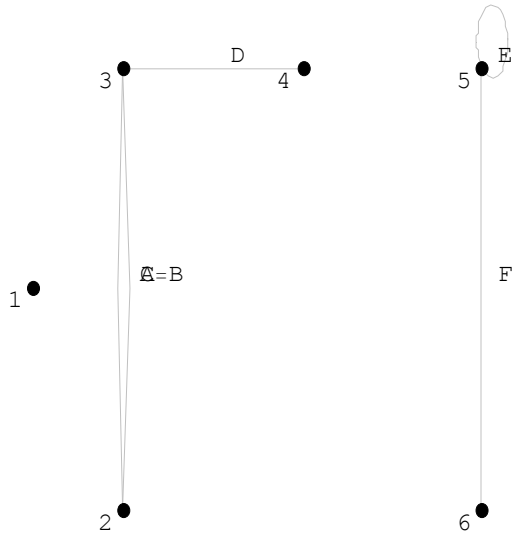
Изобразим ребра графа g жирным пунктиром и, чтобы вершины не скрылись за ними, увеличим размер вершин:

```
In[12]:= ShowGraph[g, VertexStyle → Disk[Large], EdgeColor → Red,
          EdgeStyle → ThickDashed, Background → Gray]
```



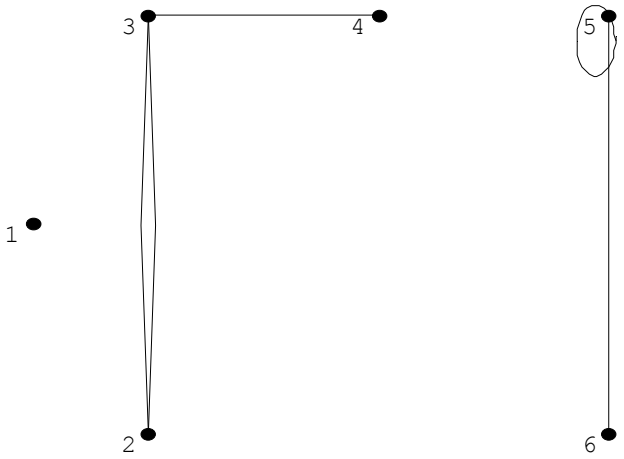
Метки ребер устанавливаются опцией EdgeLabels. Расположим их в центре соответствующих ребер, используя опцию EdgeLabelPosition. Чтобы выделить метки ребер, ребра окрашены в серый цвет. Как и в случае меток вершин, метки ребер присваиваются циклически в порядке следования ребер графа. В этом примере на параллельных ребрах метки совмещены. С помощью SetGraphOptions мы сможем избежать этого.

```
In[13]:= ShowGraph[g, VertexNumber → True,
          EdgeLabel → {"A=B", "C", "D", "E", "F"}, EdgeColor → Gray,
          EdgeLabelPosition → UpperRight, PlotRange → 0.1]
```



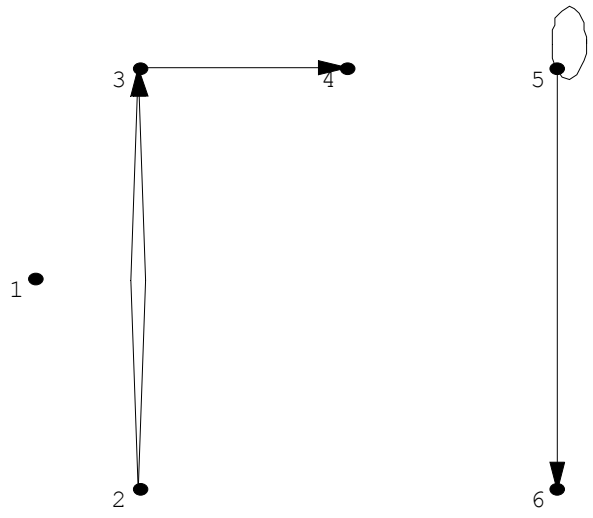

ShowGraph корректно изображает петли и параллельные ребра. Расположение петель, относящихся к вершинам, могут контролироваться опцией LoopPosition. Расположим петлю в g слева и ниже от значения по умолчанию.

In[14]:= **ShowLabeledGraph[g, LoopPosition → LowerLeft]**



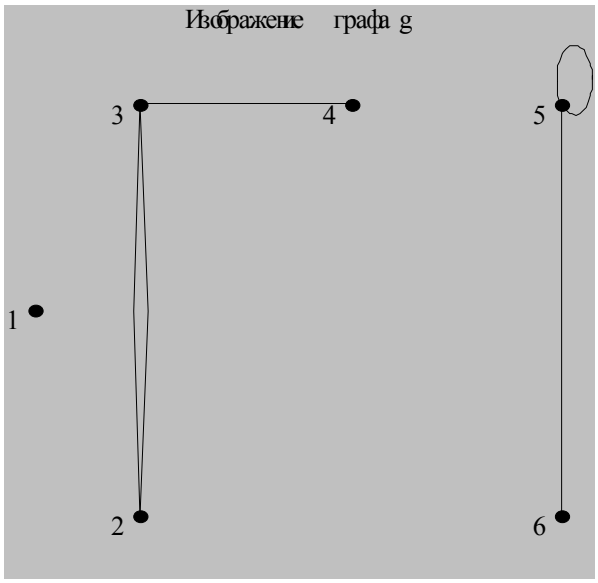

Ориентируем теперь ребра графа g с помощью опции EdgeDirection->True. Каждое ребро (i,j) трактуется как ориентированное ребро из i в j. Граф g неориентирован, но так как ребра неориентированного графа сохраняются как пары натуральных чисел, в которой первое число меньше, чем второе, мы получим изображение графа, в котором все ребра ориентированы от вершины с меньшим номером к вершине с большим номером.

In[15]:= **ShowLabeledGraph[g, EdgeDirection → True]**



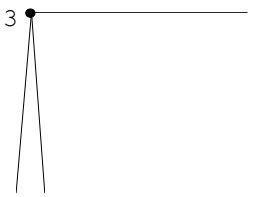

Как мы уже упоминали, ShowGraph, ShowLabeledGraph, ShowGraphArray наследуют опции из встроенной функции Plot. Продемонстрируем действие нескольких опций, которые наследуются из Plot, в частности PlotLabel, которая устанавливает общее название изображения графа, и опции TextStyle, которая используется, чтобы выделить текст.

```
In[16]:= ShowLabeledGraph[g, Background→ Gray,
          PlotLabel -> "Изображение графа g", TextStyle→ {FontFamily→ Times},
          FontSize→ 14]
```




Одна из опций, которые ShowGraph наследует из Plot – PlotRange. Однако в ShowGraph PlotRange может использоваться новыми способами. Установка PlotRange на действительное число $\alpha$ увеличивает PlotRange в $(1+\alpha)$ раз его первоначальный размер. Это может быть использовано для того, чтобы растянуть или сжать область изображения графа. Другой отличительной чертой применения PlotRange в Combinatorica – возможность рассмотреть фрагмент окрестности нужной вершины, устанавливая PlotRange->Zoom. Например, рассмотрим окрестность третьей вершины.

```
In[17]:= ShowGraph[g, PlotRange→ Zoom[{3}], VertexNumber→ True,
          TextStyle→ FontSize→ 14]
```




ShowGraph также наследует опции из пакета Graphics`Arrow`, которые позволяют контролировать форму и размер стрелок. Рассмотрим эти опции:

```
In[18]:= Options[Arrow]
Out[18]= {HeadScaling → Automatic, HeadLength → Automatic, HeadCenter → 1,
          HeadWidth → 0.5, HeadShape → Automatic, ZeroShape → Automatic}
```

### *5.2.2.Функция ShowGraphArray*

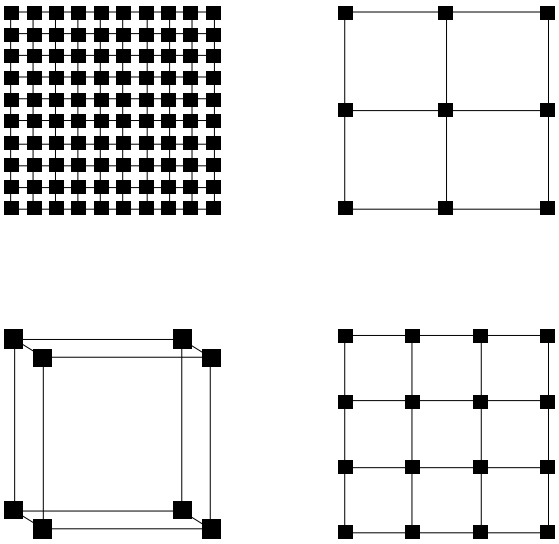
Отметим еще одну функцию Combinatorica – ShowGraphArray, которая позволяет показ нескольких графов в одном окне.

| | |
|---|---|
| ShowGraphArray[{g1, g2, ...}] | изображает в одной строке графы g1, g2, .. |
| ShowGraphArray[{ {g1, ...}, {g2, ...}, ...}] | изображает в двумерной таблице графы g1, g2, .. |

ShowGraphArray принимает все опции ShowGraph и еще одну дополнительную опцию GraphicsSpacing -> d, которая задает расстояние между изображаемыми графами.

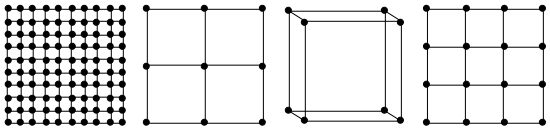
Изобразим четыре графа в двумерной таблице. Опция VertexStyle->Box[Large] затрагивает все четыре графа. По умолчанию пространство между картинками соответствует установке GraphicsSpacing-> 0.1. Введем большее разделение чем обычно, увеличив GraphicsSpacing на 0.5.

```
In[2]:= ShowGraphArray[
          gt = {{GridGraph[10, 10], GridGraph[3, 3]},
            {GridGraph[2, 2, 2], GridGraph[4, 4]}}, VertexStyle → Box[Large],
          GraphicsSpacing → 0.5]
```



Расположим эти графы в ряд:

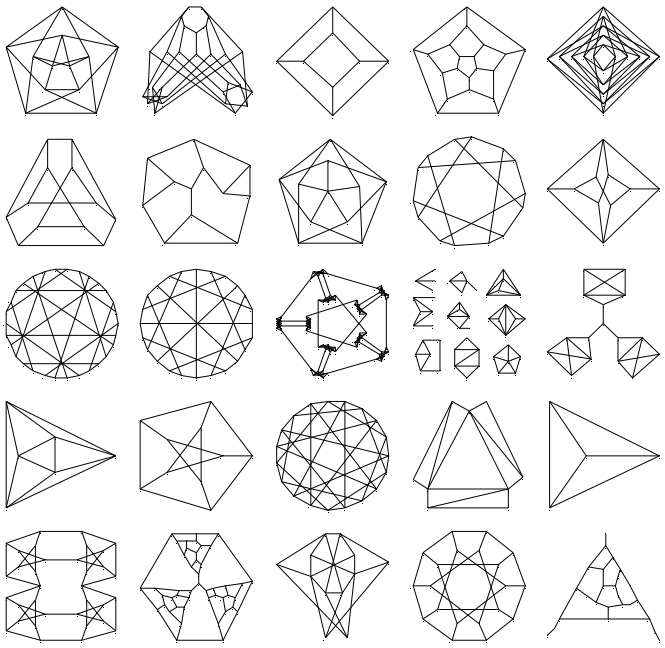
```
In[3]:= ShowGraphArray[Flatten[gt, 1], VertexStyle → Disk[Large]]
```



```
Out[3]= - GraphicsArray -
```

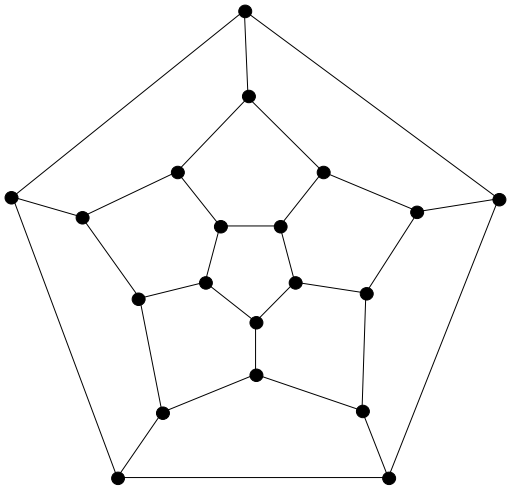
Некоторые из встроенных функций Combinatorica не зависят от параметров и строят отдельный граф. Функция FiniteGraphs собирает их вместе в одном списке. В этом списке собраны двадцать пять непараметризованных графов, играющих важную роль в теории графов.

```
In[4]:= ShowGraphArray[Partition[FiniteGraphs, 5, 5]]
```



Рассмотрим, например, четвертый элемент этого списка, это известный граф додекаэдра:

```
In[5]:= ShowGraph[FiniteGraphs[[4]]]
```



### *5.2.3.Выделение элементов графа подсветкой и анимация графов*

Еще одна функция, улучшающая изображение – Highlight[g,p], которая в графе g выделяет подсветкой некоторые элементы p. Второй элемент p имеет вид {s1, s2,...}, где si – непересекающиеся подмножества вершин и ребер g. Highlight принимает опции HighlightedVertexStyle, HighlightedEdgeStyle, HighlightedVertexColors и HighlightedEdgeColors

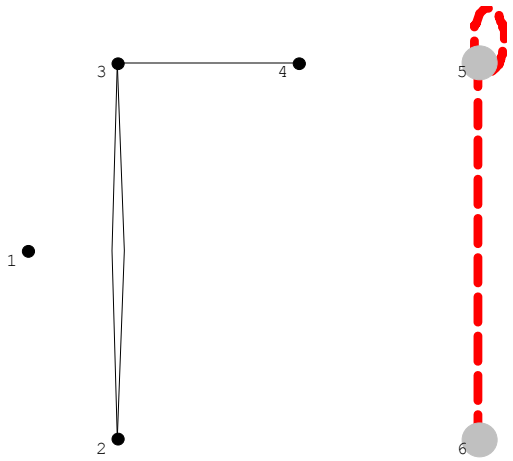
| название опции | значение по умолчанию | определение |
|---|---|---|
| HighlightedVertexStyle | Disk[Large] | устанавливает стиль выделяемой вершины |
| HighlightedEdgeStyle | Thick | устанавливает стиль выделяемого ребра |
| HighlightedVertexColors | Black | устанавливает цвет выделяемой вершины |
| HighlightedEdgeColors | Black | устанавливает цвет выделяемого ребра |

Приведем возможные значения для опций ребер

| Название опции | Возможные значения |
|---|---|
| HighlightedVertexStyle | X[Y], где X может быть Disk или Box, а Y может быть Small, Normal, Large, или неотрицательным вещественным числом |
| HighlightedEdgeStyle | Thick, Normal,Thin,ThickDashed, NormalDashed или ThinDashed |
| HighlightedVertexColors | любой цвет, определенный в пакете Graphics`Color` |
| HighlightedEdgeColors | любой цвет, определенный в пакете Graphics`Color` |

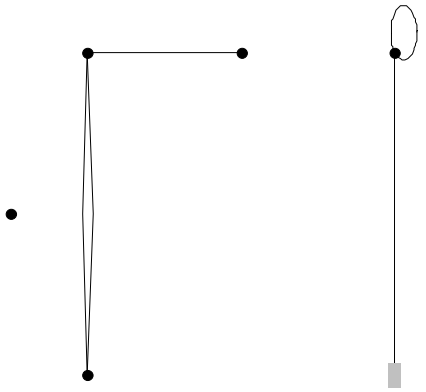
Выделим с помощью Highlight петлю и ребро {5,6}, причем вершины выделяются большими дисками серого цвета, а ребра выделяются красным жирным пунктиром:

```
In[2]:= ShowLabeledGraph[Highlight[g, {{6, 5}, {{5, 5}, {5, 6}}},
          HighlightedEdgeStyle → ThickDashed, HighlightedEdgeColors → {Red},
          HighlightedVertexColors → {Gray}, HighlightedVertexStyle → Disk[Large]]]
```




Функция AnimateGraph[g,l] дает возможность оживлять (анимировать) графы. Второй аргумент этой функции - это список, содержащий вершины и ребра. Элементы этого списка подсвечиваются последовательно. Функция принимает третий аргумент – опцию, которая может принимать значения All или One. Предыдущие подсвеченные элементы продолжают подсвечиваться по умолчанию в случае All. В противном случае подсвеченные ранее элементы не поддерживаются. AnimateGraph принимает все опции функции Highlight.

```
In[3]:= AnimateGraph[g, {6, {6, 5}, 5, {5, 5}}, HighlightedVertexStyle → Box[Normal],
          HighlightedVertexColors → {Gray}]
```



Кликнув по рисунку дважды, вы увидите анимацию.

## ■5.3. Первые задачи теории графов

### *5.3.1. Подграфы*

Рассмотрим граф g=(E,V). Граф $g_1=(E_1,V_1)$ называется подграфом g, если $E_1$ и $V_1$ являются соответственно такими подмножествами E и V, что ребро $(v_i,v_j)$ содержится в $E_1$ только в том случае, если $v_i$ и $v_j$ содержатся в $V_1$. Граф $g_1$ называется собственным подграфом графа g, если $E_1$ – собственное подмножество E или $V_1$ - собственное подмножество V. Если все вершины графа g присутствуют в подграфе $g_1$ графа g, тогда $g_1$ называется остовным подграфом графа g. Некоторые из вершин подграфа могут быть изолированными

Если подграф $g_1=(E_1,V_1)$ графа g не содержит изолированных вершин, тогда, по определению подграфа, каждая вершина $V_1$ является концевой вершиной некоторого ребра из E. Таким образом, в этом случае E однозначно определяет $V_1$ и, следовательно, подграф

$g_1$. Подграф $g_1$ называется **порожденным подграфом** графа g на множестве ребер $E_1$(или просто реберно-порожденным подграфом графа g) и обозначается $< E_1>$.

Заметим, что множество вершин $V_1$ графа $<E_1>$ является наименьшим подмножеством V, содержащим все концевые вершины ребер в $E_1$.

Определим теперь вершинно-порожденный подграф. Пусть $V_1$ – подмножество множества вершин V графа g=(E,V). Тогда подграф $g_1=(E_1,V_1)$ называется порожденным подграфом графа g на множестве вершин $V_1$ (или просто вершинно-порожденным подграфом графа g), если $E_1$ является таким подмножеством E, что ребро $(v_i,v_j)\epsilon E$ входит в $E_1$ тогда и только тогда, когда $v_i$ и $v_j$ входят в $V_1$. Другими словами, если $v_i$ и $v_j$ принадлежат $V_1$, то каждое ребро E, имеющее в качестве концевых вершин $v_i$ и $v_j$ , должно входить в $E_1$. Заметим, что в этом случае $V_1$ полностью определяет $E_1$ и, таким образом, подграф $g_1$.Следовательно, вершинно-порожденный подграф можно обозначить как $<V_1>$. Множество ребер $E_1$ вершинно-порожденного подграфа на множестве $V_1$ является таким наибольшим подмножеством E, что концевые вершины всех его ребер принадлежат $V_1$.

Подграф $g_1$ графа g называется максимальным подграфом по отношению к некоторому свойству P, если $g_1$ обладает свойством P и $g_1$ не является собственным подграфом никакого другого подграфа графа g, обладающего свойством P.

Подграф $g_1$ графа g называется минимальным подграфом по отношению к некоторому свойству P, если $g_1$ обладает свойством P и никакой подграф графа g, обладающий свойством P, не является собственным подграфом графа $g_1$.

Максимальное и минимальное подмножества некоторого множества по отношению к свойству определяются аналогично.

Встроенная функция Combinatorica InduceSubgraph и PermuteSubgraph конструируют вершинно-порожденные подграфы.

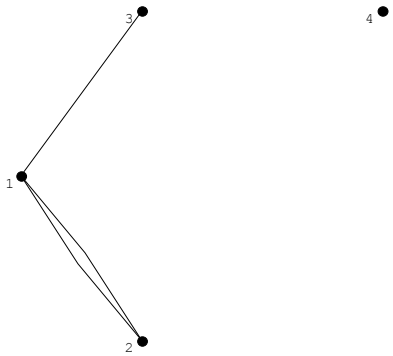
| | |
|---|---|
| InduceSubgraph [g, s] | конструирует подграф, порожденный списком вершин s |
| PermuteSubgraph[g, p] | переставляет вершины подграфа графа g в соответствии с перестановкой p. |

Рассмотрим некоторые подграфы графа g из предыдущего пункта:

```
In[2]:= ShowGraphArray[Map[InduceSubgraph[g, #] &, {{1, 2, 3}, {5, 6}, {2, 3, 5, 6}}]]
```

Рассмотрим другое изображение подграфа, порожденного вершинами {1, 2, 3, 4}. Для этих вершин используется то же расположение, но вершины перемещены в соответствии с указанной перестановкой. PermuteSubgraph позволяет получить новое вложение графов.

```
In[3]:= ShowLabeledGraph[PermuteSubgraph[g, {3, 2, 4, 1}]]
```




### *5.3.2 Маршруты, цепи, пути и циклы*

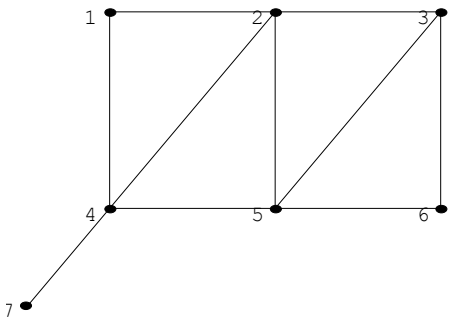
Маршрут в графе g=(E,V) представляет собой конечную чередующуюся последовательность вершин и ребер $v_0$, $e_1$, $v_1$, $e_2$,….,$e_k$, $v_k$ , начинающуюся и заканчивающуюся на вершинах, причем $v_i$ и $v_{i-1}$ являются концевыми вершинами ребра $e_i$, $1 \le i \le k$. Мы будем рассматривать маршрут как конечную последовательность таких вершин ребер $v_0,v_1,\ldots,v_k$ , что ($v_{i-1}$, $v_i$), $1 \le i \le k$, - ребро графа g. Заметим, что ребра и вершины в маршруте могут появляться более одного раза. Маршрут называется **открытым**, если его концевые вершины различны, в противном случае он называется замкнутым. Маршрут называется **цепью**, если все его ребра различны. Цепь называется **открытой**, если концевые вершины различны, в противном случае она называется **замкнутой.** Замкнутая цепь называется **циклом,** если различны все ее вершины, за исключением концевых вершин.

Пример. Сформируем ребра и вершины графа:

```
In[2]:= u = {{-1, 0}, {0, 0}, {1, 0}, {-1, -1}, {0, -1}, {1, -1}, {-1.5, -1.5}};
        l = {{1, 2}, {1, 4}, {2, 4}, {2, 5}, {2, 3}, {3, 5}, {3, 6}, {4, 7},
          {4, 5}, {5, 6}};
```

Сконструируем граф из пар списка l:

```
In[4]:= h = FromUnorderedPairs[l, u]
Out[4]= -Graph:<10, 7, Undirected>-

In[5]:= ShowGraph[h]
```




В графе h последовательность {7, 4, 5, 2, 1, 4, 5, 2, 3, 6}-открытый маршрут, {1, 2, 4, 5, 3, 2, 5, 4, 1}- замкнутый маршрут, {1, 4, 2, 5, 3, 6, 5, 4}- открытая цепь, {7, 4, 5, 2, 3} –путь, а {4, 2, 3, 6, 5, 4} – цикл.

Ребро графа называется **циклическим**, если в графе g существует цикл, содержащий это ребро. В графе h все ребра, за исключением ребра {4,7}, циклические. Число ребер в пути называется **длиной пути**. Аналогично определяется длина цикла.

В Combinatorica функции ExtractCycles, FindCycle возвращают циклы:

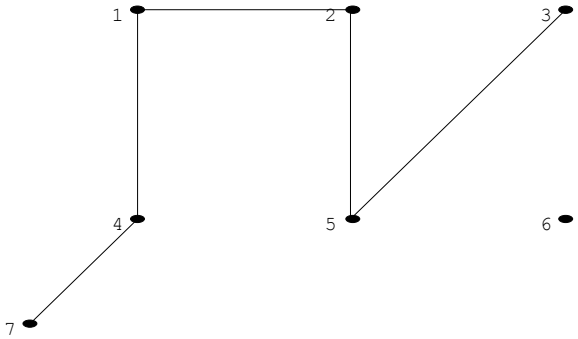
| | |
|---|---|
| FindCycle[g] | возвращает список вершин графа g, которые образуют цикл |
| ExtractCycles[g] | находит максимальный список реберно непересекающихся циклов |
| DeleteCycle[g, c] | удаляет простой цикл c из графа g. Цикл c задается последовательностью вершин, в которой первая и последняя вершины совпадают. При этом граф g может быть как ориентированным, так и неориентированным. |

Найдем цикл графа h

```
In[6]:= FindCycle[h]
Out[6]= {5, 2, 3, 5}
```

Найдем максимальный список реберно – непересекающихся циклов графа h:

```
In[7]:= ExtractCycles[h] // ColumnForm
Out[7]= {4, 1, 2, 4}
        {5, 2, 3, 5}

In[8]:= ShowLabeledGraph[DeleteCycle[h , {4, 2, 3, 6, 5, 4}]]
```




Укажем следующие свойства путей и циклов.

1.Степень каждой неконцевой вершины пути равна 2, концевые вершины имеют степень 1.

2.Каждая вершина цикла имеет степень 2 или другую четную степень.

3.Число вершин в пути на единицу больше числа ребер, тогда как в цикле число ребер равно числу вершин.

### *5.3.3. Связность и компоненты графа*

Важным понятием теории графов является связность. Две вершины $v_i$ и $v_j$ называются связанными в графе g, если в нем существует путь с концевыми вершинами $v_i$ и $v_j$. Вершина связана сама с собой.

Граф g называется **связным,** если в нем существует путь между каждой парой вершин.

Функция ConnectedQ тестирует, является ли граф g связным.

| | |
|---|---|
| ConnectedQ | Возвращает True, если g– связный граф, и False в противном случае |

Протестируем на связность графы g, h, построенные в п. 5.2.1 и 5.3.2.

```
In[9]:= Map[ConnectedQ, {g, h}]
Out[9]= {False, True}
```

Рассмотрим несвязный граф g=(E,V). Тогда множество вершин V можно разбить на такие подмножества $V_1,V_2,\ldots,V_p$, что вершинно-порожденные подграфы $<V_i>$, i=1,2,…,p связны, и никакая вершина подмножества $V_i$ не связана ни с какой вершиной подмножества $V_j$, j□i. Подграфы $<V_i>$, i=1,2,…,p, называются компонентами графа Легко видеть, что компонентой графа g является максимально связный подграф графа g.

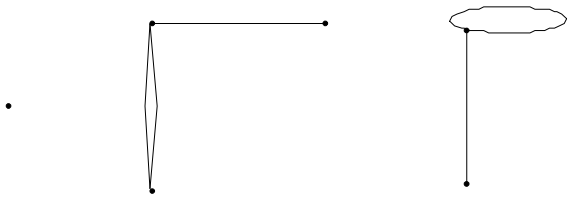
| | |
|---|---|
| ConnectedComponents[g] | Возвращает связные компоненты графа g |

Рассмотрим связные компоненты графа g.

```
In[10]:= ConnectedComponents[g]
Out[10]= {{1}, {2, 3, 4}, {5, 6}}
```

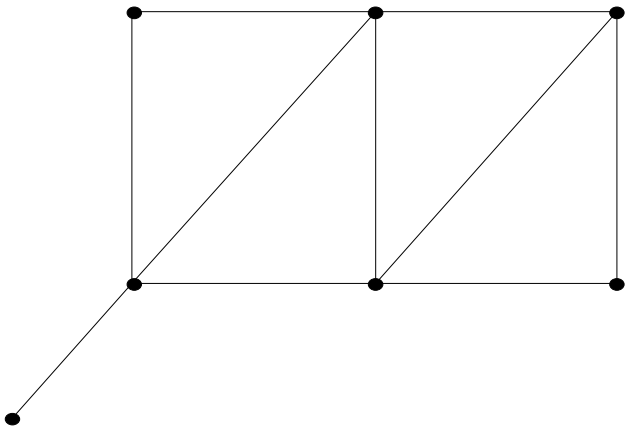
Отметим, что изолированную вершину также следует рассматривать как компоненту, поскольку по определению вершина связана сама с собой.

```
In[11]:= ShowGraphArray[Map[InduceSubgraph[g, #] &, ConnectedComponents[g]]]
```



Если граф g связен, то он имеет только одну компоненту, которая является графом g.

```
In[12]:= ShowGraphArray[Map[InduceSubgraph[h, #] &, ConnectedComponents[h]]]
```



Теперь рассмотрим некоторые свойства связных графов.

**Теорема.** В связном графе любые два пути максимальной длины имеют общие вершины.

**Теорема.** Если граф g=(E,V) связен, то граф $g_1$=(E-e,V), получающийся после удаления циклического ребра e, тоже связен.

### *5.3.4. Изоморфизм графов*

Рассмотрим графы $g_1=(E_1,V_1)$ и $g_2=(E_2,V_2)$. Пусть f: $V_1$->$V_2$ – биекция, причем u, v смежны в $V_1$, тогда и только тогда, когда вершины f(u), f(v) смежны в $V_2$. Тогда графы $g_1$ и $g_2$ называются **изоморфными,** а f – называется **изоморфизмом** графов $g_1$ и $g_2$.

Очевидно, что отношение изоморфизма является отношением эквивалентности, т.е. обладает свойством симметричности, рефлексивности и транзитивности. Поэтому множество всех графов распадается на непересекающиеся классы изоморфных графов. Изоморфные графы имеют одинаковые комбинаторные свойства и с точки зрения теории графов, неразличимы.

Из определения изоморфизма следует, что изоморфные графы имеют одинаковое количество элементов. Вершины, соответствующие при изоморфизме, имеют одинаковую степень. При изоморфизме петли переходят в петли, а параллельные ребра – в параллельные ребра и т.д. В частности, изоморфизм графов сохраняет смежность вершин и смежность ребер.

Важно заметить, что не всякое взаимно однозначное соответствие, сохраняющее смежность, является изоморфизмом графов.

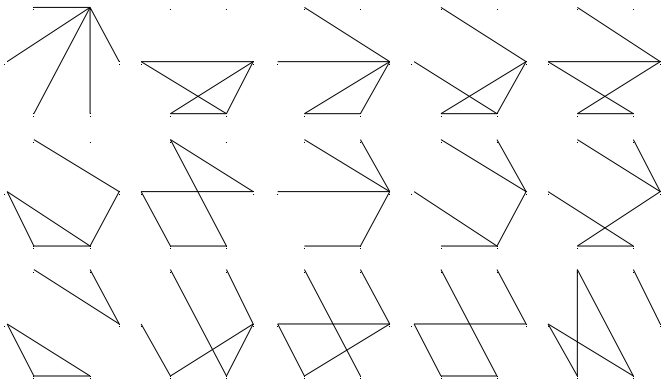
Функция IsomorphicQ [g,h] возвращает True, если g, h – изоморфные графы, и False в противном случае.

Графы, построенные в п. 5.2.1 и 5.3.2, очевидно, неизоморфны:

```
In[13]:= IsomorphicQ[g, h]
Out[13]= False
```

Функция ListGraphs[n, m] возвращает все неизоморфные неориентированные графы с n вершинами и m ребрами. ListGraphs[n] возвращает все неизоморфные неориентированные графы с n вершинами. Добавляя опцию Directed, можно получить ориентированные графы.

```
In[14]:= ShowGraphArray[Partition[ListGraphs[6, 5], 5]]
```



Combinatorica содержит очень интересную функцию GraphPolynomial, которая считает количество неизоморфных графов. GraphPolynomial[n, x] возвращает полином от x, в котором коэффициент перед $x^m$ – число неизоморфных графов с n вершинами и m ребрами. GraphPolynomial[n, x, Directed] возвращает полином от x, где коэффициент перед $x^m$ – число неизоморфных ориентированных графов с n вершинами и m ребрами.

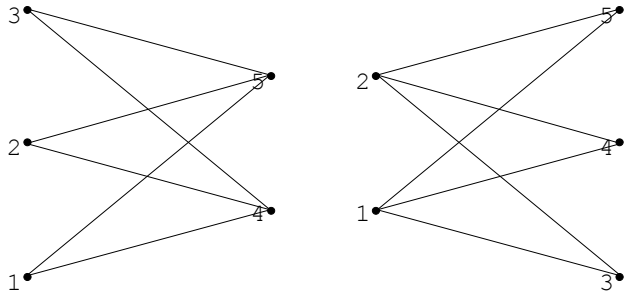
```
In[15]:= GraphPolynomial[6, x]
Out[15]= 1 + x + 2 x^2 + 5 x^3 + 9 x^4 + 15 x^5 + 21 x^6 + 24 x^7 +
         24 x^8 + 21 x^9 + 15 x^10 + 9 x^11 + 5 x^12 + 2 x^13 + x^14 + x^15

In[16]:= {Coefficient[GraphPolynomial[6, x], x^5], Length[ListGraphs[6, 5]]}
Out[16]= {15, 15}
```

Число неизоморфных неориентированных графов на n вершинах также считает функция NumberOfGraphs[n]. NumberOfGraphs[n,m] возвращает число неизоморфных неориентированных графов с n вершинами и m ребрами. NumberOfDirectedGraphs[n] возвращает число неизоморфных ориентированных n-вершинных графов.
NumberOfDirectedGraphs[n, m] возвращает число неизоморфных ориентированных n-вершинных графов с m ребрами.

```
In[17]:= {Coefficient[GraphPolynomial[6, x], x^5], Length[ListGraphs[6, 5]],
          NumberOfGraphs[6, 5]}
Out[17]= {15, 15, 15}
```

Графы называются **идентичными**, если они имеют идентичный список ребер, хотя ассоциированная с ними графическая информация не обязательно одна и та же. Функция IdenticalQ[g,h] тестирует, являются ли g,h идентичными графами.
Рассмотрим следующие графы:

```
In[18]:= ShowGraphArray[{p = CompleteGraph[3, 2], q = CompleteGraph[2, 3]},
          VertexNumber → True]
```




Эти графы изоморфны, но не идентичны:

```
In[19]:= {IsomorphicQ[p, q], IdenticalQ[p, q]}
Out[19]= {True, False}
```

| | |
|---|---|
| ListGraphs[n, m] | возвращает все неизоморфные неориентированные графы с n вершинами и m ребрами. |
| ListGraphs[n, m, Directed] | возвращает все неизоморфные ориентированные графы с n вершинами и m ребрам |
| ListGraphs[n] | возвращает все неизоморфные неориентированные графы с n вершинами |
| ListGraphs[n, Directed] | возвращает все неизоморфные ориентированные графы с n вершинами |
| GraphPolynomial[n, x, Directed] | возвращает полином от x, в котором коэффициент перед $x^m$ – число неизоморфных графов с n вершинами и m ребрами |
| GraphPolynomial[n, x, Directed] | возвращает полином от x, где коэффициент перед $x^m$ – число неизоморфных ориентированных графов с n вершинами и m ребрами |
| NumberOfGraphs[n] | возвращает число неизоморфных неориентированных графов на n вершинах |
| NumberOfGraphs[n, m] | возвращает число неизоморфных неориентированных графов с n вершинами и m ребрами. |
| NumberOfDirectedGraphs[n] | возвращает число неизоморфных ориентированных n-вершиных графов. |
| NumberOfDirectedGraphs[n, m] | возвращает число неизоморфных ориентированных n- |

| | |
|---|---|
| | вершиных графов с m ребрами. |
| IsomorphicQ [g,h] | возвращает True, если g,h – изоморфные графы, и False в противном случае |
| IdenticalQ[g,h] | возвращает True, если g,h – идентичные графы, и False в противном случае |

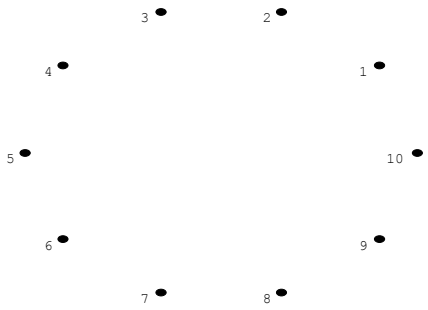
## ■5.4. Специальные графы

В этом разделе мы рассмотрим специальные классы графов, часто встречающиеся в теории графов.

### *5.4.1 Пустые графы, полные графы, регулярные графы*

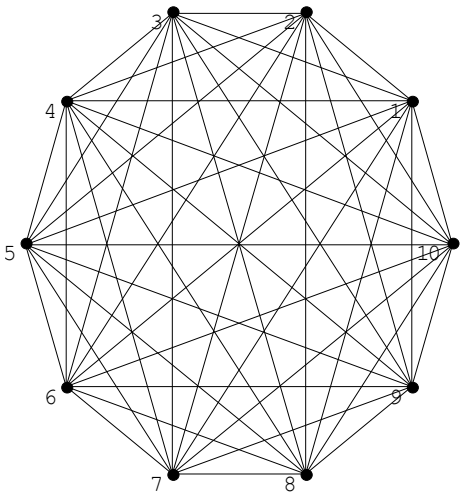
Пустой граф на n вершинах конструируется функцией EmptyGraph[n]

```
In[20]:= ShowLabeledGraph[EmptyGraph[10]]
```




Полный граф g – простой граф, в котором каждая пара вершин смежна. Если полный граф g имеет n вершин, то он обозначается $K_n$. Легко видеть, что $K_n$ имеет n(n-1)/2 ребер.
Функция CompleteGraph[n] возвращает полный граф на n вершинах. Эта функция допускает опцию Type, принимающую значения Directed или Undirected. По умолчанию Type –> Undirected.

```
In[21]:= ShowLabeledGraph[CompleteGraph[10]]
```




Проверить, является ли граф g – полным, можно с помощью CompleteQ. Проверим, являются ли графы h, g из п. 5.2.1 и 5.3.2 полными.

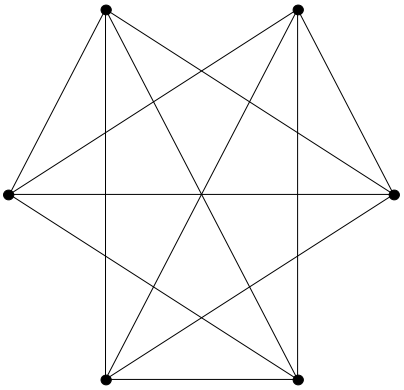
```
In[22]:= {CompleteQ[h], CompleteQ[g]}
Out[22]= {False, False}
```

Граф g называется **регулярным**, если в нем все вершины имеют равную степень. Если граф g регулярен и степень всех вершин равна r, то g называют r-регулярным графом. Отметим, что $K_n$ является (n-1) – регулярным графом.
Покажем, что полный граф – регулярный:

```
In[23]:= Degrees[CompleteGraph[15]]
Out[23]= {14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14}
```

RegularQ[g] возвращает True, если граф g – регулярный.
Функция RegularGraph[k, n] конструирует полуслучайный k-регулярный граф, если таковой существует.

```
In[24]:= ShowGraph[RegularGraph[4, 6]]
```

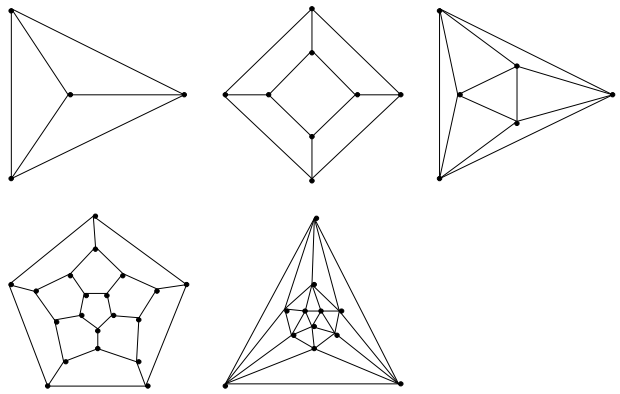


Протестируем на регулярность графы g, h, построенные в п. 5.2.1 и 5.3.2:

```
In[25]:= {RegularQ[g], RegularQ[h]}
Out[25]= {False, False}
```

Примерами регулярных графов являются графы, составляемые ребрами и вершинами пяти правильных многогранников, или платоновых тел: тетраэдра, куба, октаэдра, додекаэдра и икосаэдра. Функции TetrahedralGraph, CubicalGraph, OctahedralGraph, DodecahedralGraph, IcosahedralGraph конструируют графы платоновых тел.

```
In[26]:= ShowGraphArray[{{TetrahedralGraph, CubicalGraph, OctahedralGraph},
          {DodecahedralGraph, IcosahedralGraph}}]
```



| | |
|---|---|
| EmptyGraph[n] | возвращает пустой граф на n вершинах. Допустима опция Type, которая принимает значения Directed or Undirected. По умолчанию Type -> Undirected. |
| CompleteGraph[n] | возвращает полный граф порядка n. Допустима опция Type, которая принимает значения Directed or Undirected. По умолчанию Type -> Undirected. |
| CompleteQ[g] | возвращает True, если граф g - полный |
| RegularGraph[k, n] | возвращает полуслучайный – регулярный граф на n вершинах, если таковой существует. |
| TetrahedralGraph | возвращает граф тетраэдра |
| CubicalGraph | возвращает граф куба |
| OctahedralGraph | возвращает граф октаэдра |
| DodecahedralGraph | возвращает граф додекаэдра |
| IcosahedralGraph | возвращает граф икосаэдра |

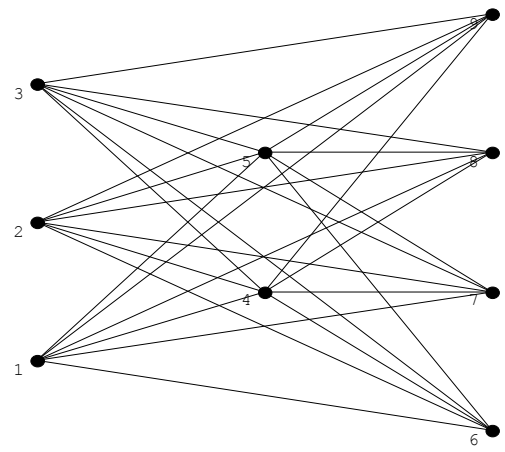
### *5.4.2. K - дольные графы*

Граф g=(E,V) называется двудольным графом, если множество его вершин V можно разбить на два таких подмножества $V_1$ и $V_2$, что каждое ребро принадлежащее E, имеет одну концевую вершину в подмножестве $V_1$, а другую - в $V_2$ ; ($V_1$ , $V_2$) называют двудольным разбиением графа g. Если в простом двудольном графе g с разбиением ($V_1$ , $V_2$) для каждой вершины $v_i$ в $V_1$ и $v_j$ в $V_2$ существует ребро ($v_i$.$v_j$), то g называют полным двудольным графом и обозначают через $K_{m,n}$, если $V_1$ содержит m вершин, а $V_2$ - n вершин.
Функция BipartiteQ [g] проверяет, является ли граф g двудольным.
Протестируем на двудольность те же графы g, h:

```
In[27]:= {BipartiteQ[g], BipartiteQ[h]}
Out[27]= {False, False}
```

Граф g=(E,V) называется **k-дольным**, если V можно разбить на k таких подмножеств $V_1,V_2,\ldots,V_k$, что каждое ребро графа g имеет одну концевую вершину в некотором подмножестве $V_i$, а другую - в некотором подмножестве $V_j$ , i ≤ j. Полный k-дольный граф g – простой k-дольный граф с разбиением множества вершин $\{V_1,V_2,\ldots,V_k\}$, обладающей таким свойством, что для каждой вершины $v_i$ в $V_r$ и $v_j$ в $V_s$, r≠s, 1≤r,s≤k, ($v_i$,$v_j$) является ребром графа g.
Функция CompleteKPartiteGraph конструирует полный k-дольный граф.

```
In[28]:= ShowLabeledGraph[CompleteKPartiteGraph[3, 2, 4]]
```

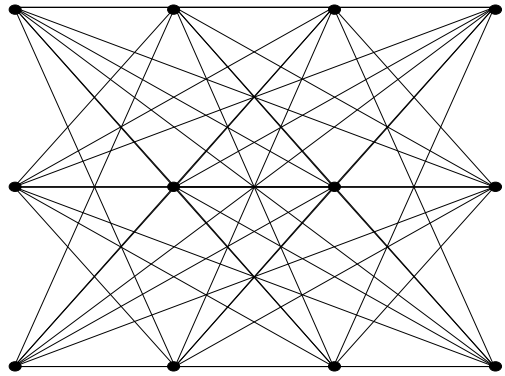



Полный граф $K_n$ может быть определен как $K_{1,1,\ldots,1}$

```
In[29]:= CompleteQ[CompleteKPartiteGraph[1, 1, 1, 1, 1, 1]]
Out[29]= True
```

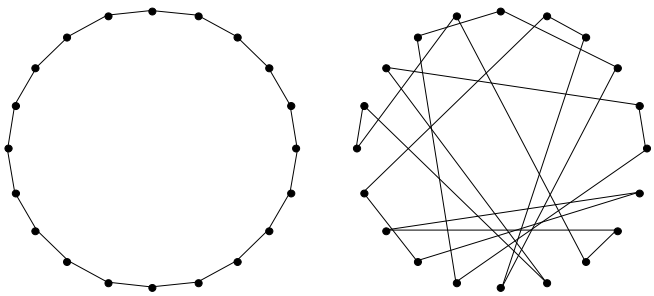
Специальный случай полного к-дольного графа – это граф Турана, который часто встречается в экстремальной теории графов. Экстремальный граф Ex(n,G) - это наибольший граф порядка n, который не содержит G как подграф.
Граф Turan[n, p] экстремальный граф на n вершинах, который не содержит $K_p$ как подграф.
В этом графе как можно более равномерно сбалансирован размер каждого этажа и максимально число ребер между каждой парой этажей.

```
In[30]:= ShowGraph[Turan[12, 5]]
```



Цикл $C_n$ – связный 2-регулярный граф порядка n. Частные случаи цикла – треугольник $C_3=K_3$ и квадрат $C_4$.
Рассмотрим цикл на 20 вершинах и этот же цикл, со случайно переставленными вершинами.

```
In[31]:= ShowGraphArray[
          {g = Cycle[20], h = PermuteSubgraph[Cycle[20], RandomPermutation[20]]}]
```

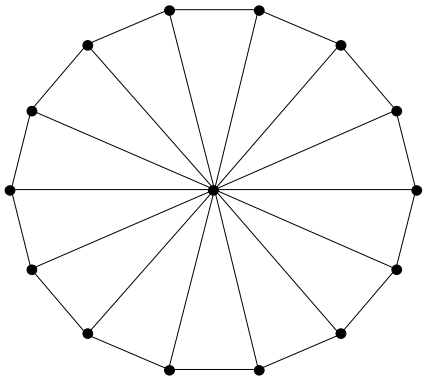


Звезда – связный граф на n вершинах, с одной вершиной степени n-1, и остальными – степени 1. Звезда с n вершинами есть двудольный граф $K_{1,n-1}$.

```
In[32]:= BipartiteQ[Star[100]]
Out[32]= True
```

Колесо- это граф, который получится, если соединить ребрами одну изолированную вершину $K_1$ с каждой вершиной цикла $C_{n-1}$ Результирующие ребра, формирующие звезду, называются спицами колеса.

```
In[33]:= ShowGraph[Wheel[15]]
```

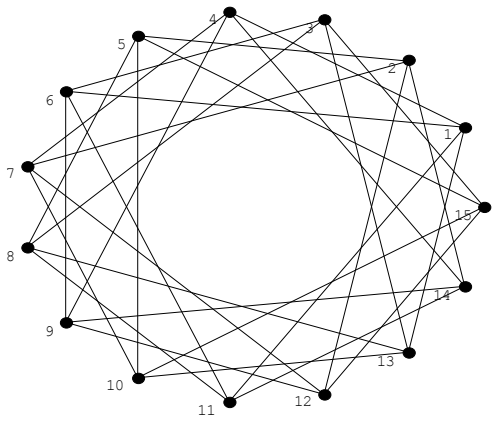


| | |
|---|---|
| BipartiteQ[g] | возвращает True, если g –двудольный граф |
| CompleteKPartiteGraph [a, b, c, ...] | конструирует полный k-дольный граф формы [a, b, c, ...]. Допустима опция Type, которая принимает значения Directed или Undirected. По умолчанию Type –> Undirected. |
| Turan[n,p] | возвращает граф Турана, наибольший граф на n вершинах, не содержащий полного подграфа на p вершинах |
| Cycle[n] | возвращает цикл на n вершинах, 2-регулярный связный граф. Допустима опция Type, которая принимает значения Directed or Undirected. По умолчанию Type –> Undirected. |
| Star[n] | возвращает звезду на n вершинах. Допустима опция Type, которая принимает значения Directed or Undirected. По умолчанию Type –> Undirected. |
| Wheel[n] | возвращает колесо на n вершинах. Допустима опция Type, которая принимает значения Directed или Undirected. По умолчанию Type –> Undirected. |

### *5.4.3 Графы циркулянты и решетчатые графы*

CirculantGraph[n, l] конструирует граф-циркулянт на n вершинах, у которого i-я вершина смежна (i+j) – й и (i-j)–й вершинам для каждого j из списка l. Таким образом, графы-циркулянты включают полные графы и циклы как частные случаи.
Построим циркулянт - граф на 21 вершине, где каждая вершина связана с другой через 3 и через 7 по каждую сторону. Связь через 3 вершины создает 3 цикла длины 7, и связи через 7 вершин создает 7 циклов длины 3 каждый.

```
In[2]:= ShowLabeledGraph[g = CirculantGraph[21, {3, 7}]]
```

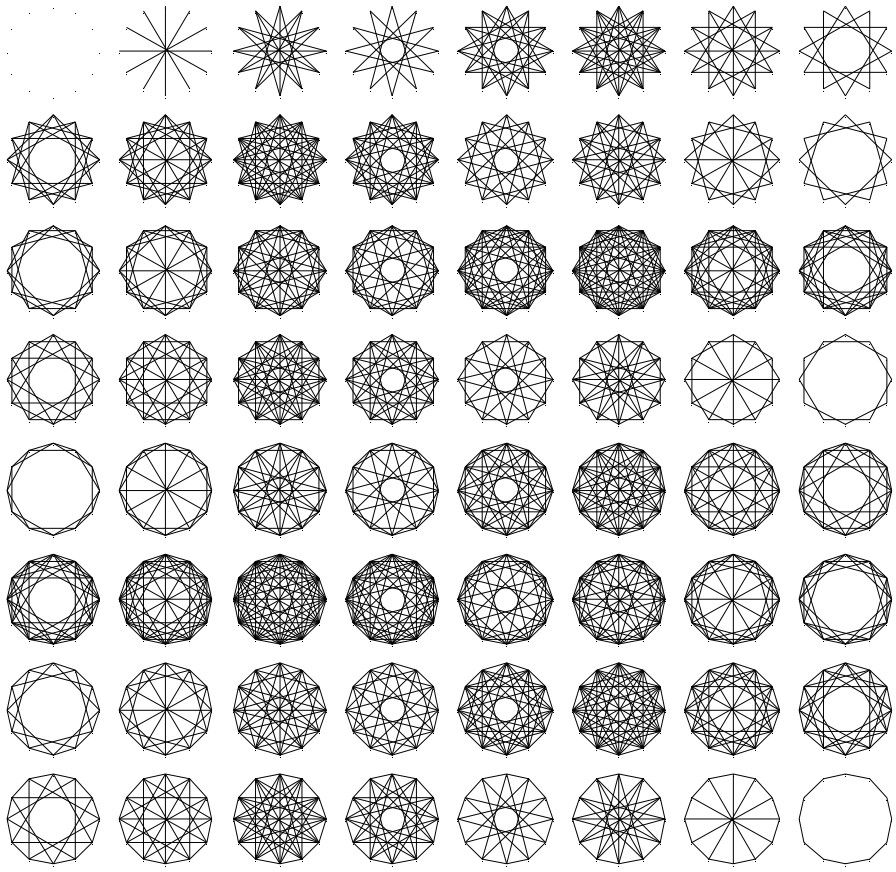


Конструирование 21-вершинного циркулянта связыванием вершин через 7 ведет к 7 связным компонентам, каждый из которых есть 3-цикл.

```
In[3]:= ConnectedComponents[CirculantGraph[21, {7}]]
Out[3]= {{1, 8, 15}, {2, 9, 16}, {3, 10, 17},
        {4, 11, 18}, {5, 12, 19}, {6, 13, 20}, {7, 14, 21}}
```

Графы-циркулянты обладают высокой степенью симметричности. Представим все $2^6$ графов-циркулянтов на 12 вершинах, построенных из списка всех 6-элементных подмножеств

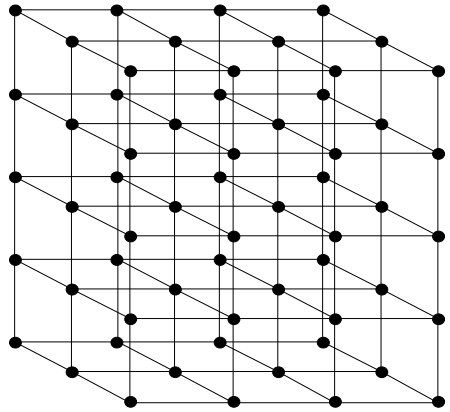
```
In[4]:= ShowGraphArray[Partition[Map[(CirculantGraph[12, #]) &, Subsets[6]], 8]]
```



Решетчатым графом (GridGraph) называется граф порядка m×n, причем две вершины соответствуют упорядоченной паре (i,j), где i,j - целые, $1 \le i \le m$, и $1 \le j \le n$. Каждое ребро связывает пару (i,j) и $(i_1,j_1)$, где $Abs(i-i_1) + Abs(j-j_1) = 1$. Любой, кто имел дело с разграфленным куском бумаги, знаком с решетчатыми графами.

Решетчатые графы естественным образом могут быть обобщены на более высокие размерности. Combinatorica обеспечена функцией GridGraph, которая берет как ввод два или три положительных целых числа и возвращает двумерный или трехмерный решетчатый граф.

```
In[5]:= ShowGraph[GridGraph[3, 4, 5]]
```



| | |
|---|---|
| CirculantGraph[n, l] | конструирует граф-циркулянт на n вершинах, у которого i-я вершина смежна (i+j) –й и (i-j)–й вершинам для каждого j из списка l. |
| GridGraph[n, m] | возвращает m× n – решетчатый граф |
| GridGraph[p, q, r] | возвращает p×q×r – решетчатый граф |

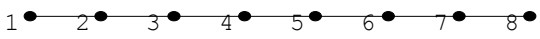
### *5.4.4 Деревья*

Граф называется **ациклическим**, если он не содержит циклов. **Деревом** называется связный ациклический граф. Чтобы проверить является ли данный граф ациклическим или деревом, применяются функции AcyclicQ и TreeQ:

```
In[2]:= {AcyclicQ[Star[20]], TreeQ[Star[20]]}
Out[2]= {True, True}
```

Самое простое дерево – это путь. Функция Path[n] конструирует дерево, состоящее из одного пути на n вершинах, и допускает опцию Type, которая может принимать значения Directed или Undirected. По умолчанию Type –> Undirected

```
In[3]:= ShowLabeledGraph[Path[8]]
```




Хотя определение дерева как связного ациклического графа просто для понимания, существует еще несколько других определений дерева, которые рассматриваются в следующей теореме:
Теорема. Для графа g, имеющего n вершин и m ребер, следующие утверждения эквивалентны:

1. g является деревом;
2. существует только один путь между любыми двумя вершинами в g;
3. g является связным и m=n-1;
4. g – ациклический граф, и m=n-1;

5. g- ациклический граф, и при соединении ребром произвольных двух несмежных его вершин получается граф, имеющий точно один цикл;

Отметим следующие свойства деревьев.

**Теорема** Подграф $g_1$ графа g на n вершинах является остовом g тогда и только тогда, когда $g_1$ является ациклическим и имеет n-1 ребер.

**Теорема** Граф g является связным тогда и только тогда, когда он имеет остов.

**Теорема** Подграф $g_1$ связного графа g является подграфом некоторого остова g тогда и только тогда, когда $g_1$ – ациклический.

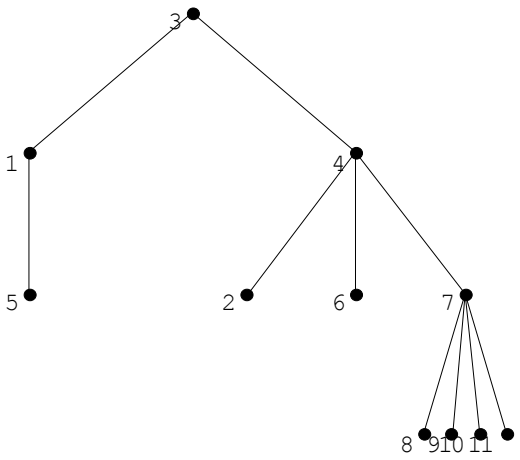
**Теорема.** В нетривиальном дереве существует, по крайней мере, две висячие вершины.

Рассмотрим произвольное дерево с n заданными вершинами, пронумерованными в произвольном порядке. Те же вершины можно соединить попарно десятью ребрами другим образом, чтобы получилось какое-то новое дерево.

Спрашивается: сколько существует таких разных деревьев? На этот вопрос отвечает теорема А.Кэли: деревьев с n пронумерованными вершинами существует ровно $n^{n-2}$. Немецкий математик Прюфер указал алгоритм, следуя которому каждому дереву можно поставить во взаимно однозначное соответствие такую последовательность длины n-2, элементы которой - первые n натуральных чисел. Вот этот алгоритм.

Рассмотрим, например, граф g на нижеприведенном рисунке.

```
In[4]:= ShowLabeledGraph[
          g1 = RootedEmbedding[
            FromUnorderedPairs[{{1, 5}, {3, 1}, {3, 4}, {2, 4}, {6, 4}, {7, 4},
              {8, 7}, {9, 7}, {10, 7}, {11, 7}}], 3]]
```




Выберем у него висячую вершину с наименьшим номером. Это вершина 2. Удалим ее вместе с принадлежащим ей ребром. Запишем 4 (4 - номер вершины полученного дерева, ближайшей к удаленной). Переходим к следующему шагу алгоритма. Вновь выбираем висячую вершину с наименьшим номером. Это вершина 5. Удаляем ее с вместе с ребром. Записываем (4,1). Повторяем процедуру до тех пор, пока не останутся две висячие вершины, связанные между собой ребром, получаем для данного дерева определенную единственным образом последовательность (4, 1, 3, 4, 4, 7, 7, 7, 7) длины n-2.

В Combinatorica функция LabeledTreeToCode[g] –возвращает код Прюфера графа g.

```
In[5]:= LabeledTreeToCode[g1]
Out[5]= {4, 1, 3, 4, 4, 7, 7, 7, 7}
```

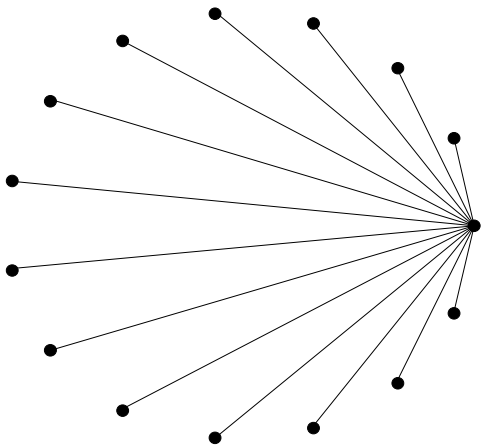
Обратно, следуя методу Прюфера, будем соединять соответствующие вершины ребрами и получать деревья. Вначале найдем наименьшее число натурального ряда, которое не встречается в

последовательности (1, 4, 1, 4, 6, 6, 6, 6, 6). Это 2. Соединяем ребром вершину 2 с вершиной 1, которая в последовательности записана первой. Следующее число натурального ряда, не входящее в оставшиеся элементы последовательности, 3. Поэтому вершину 3 соединяем с вершиной 4 (следующее число в последовательности).

Таким образом, в конце концов, получим единственно возможное дерево. Таким образом, мы убеждаемся, что деревьев с пронумерованными вершинами, столько же, сколько последовательностей рассматриваемого вида, то есть $n^{n-2}$.

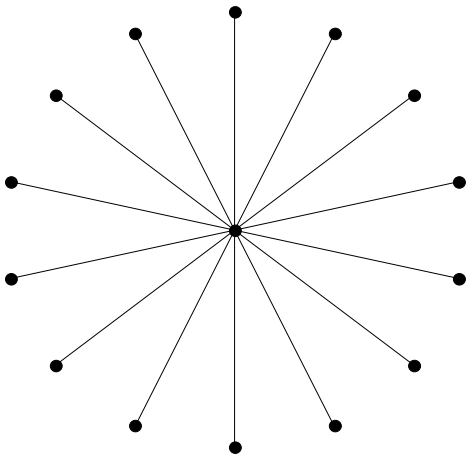
Функция CodeToLabeledTree[l] - конструирует однозначно определенное помеченное дерево на n вершинах из кода Прюфера l, где l строка из n-2 натуральных чисел, не превосходящих n. Звезда на n вершинах содержит (n-1) вершину степени 1, все инцидентные одному центру. Очевидно, что код Прюфера звезды n вершинах - это строка из (n-2 ) чисел n.

```
In[6]:= ShowGraph[
          p = CodeToLabeledTree[{15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15}]]
```



Это не совсем привычное изображение звезды, расположим вершины привычным образом с помощью функции RadialEmbedding[g], которая будет рассмотрена в п.5.6.2

```
In[7]:= ShowGraph[RadialEmbedding[p]]
```



Так как существует биекция между деревьями и кодами, композиция двух функций дает тождественный оператор.

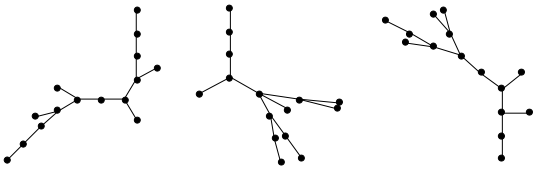
```
In[8]:= LabeledTreeToCode[CodeToLabeledTree[{3, 4, 2, 5, 6}]]
```

```
Out[8]= {3, 4, 2, 5, 6}
```

Биекция между кодами Прюфера и помеченными деревьями дает алгоритм для выбора случайного помеченного дерева. Просто порождается случайный код Прюфера и конвертируется в помеченное дерево. Это делает функция RandomTree[n], которая конструирует случайное дерево на n вершинах.

Рассмотрим три случайных дерева на пятнадцати вершинах:

```
In[9]:= ShowGraphArray[Table[RandomTree[15], {3}], VertexStyle → Disk[0.05]]
```



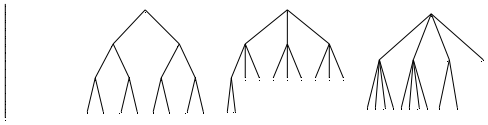
```
Out[9]= - GraphicsArray -
```

Если в дереве выбрана некоторая вершина v, то назовем v **корнем** дерева, а само дерево – корневым деревом. В корневом дереве расстояние между вершиной v и корнем называется **глубиной** вершины v. Максимальная глубина вершины в корневом дереве есть вес дерева.

K-арное дерево – это корневое дерево, в котором каждая вершина имеет по крайней мере k детей. Полное k-арное дерево – это k-арное дерево, в котором все вершины глубины h-1 или меньше имеют ровно k детей, где h- вес дерева. С этим определением, для фиксированных n и k могут существовать несколько различных n-вершинных полных k-арных деревьев, зависящих от того, как распределены висячие вершины дерева. Функция CompleteKaryTree берет как ввод числа n и k и производит k-арное полное дерево с n вершинами, в которых висячие вершины в последнем уровне все находятся слева.

Выделим полные k-арные деревья с 20 вершинами для $1 \le k \le 4$.

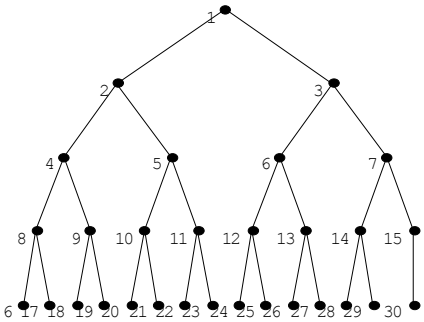
```
In[10]:= ShowGraphArray[Table[CompleteKaryTree[15, i], {i, 4}]]
```



```
Out[10]= - GraphicsArray -
```

Полное бинарное дерево – специальный случай полных k-арных деревьев для k=2. CompleteBinaryTree[n] производит полные бинарные деревья на n вершинах.

```
In[11]:= ShowLabeledGraph[CompleteBinaryTree[30]]
```



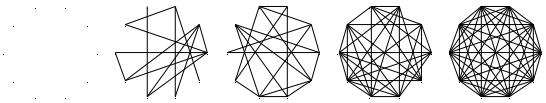

| | |
|---|---|
| AcyclicQ[g] | возвращает True, если граф g - ациклический |
| TreeQ[g] | возвращает True, если граф g -дерево |

| | |
|---|---|
| Path[n] | возвращает путь на n вершинах. Допустима опция Type, которая принимает значения Directed or Undirected. По умолчанию Type -> Undirected. |
| LabeledTreeToCode[g] | возвращает код Прюфера графа g. |
| CodeToLabeledTree[l] | конструирует однозначно определенное помеченное дерево на n вершинах из кода Прюфера l, где l строка из n-2 натуральных чисел, не превосходящих n. |
| RandomTree[n] | конструирует случайное дерево на n вершинах. |
| CompleteKaryTree [n, k] | возвращает полное k-арное дерево на n вершинах |
| CompleteBinaryTree[n] | возвращает полное бинарное дерево на n вершинах |

### *5.4.5 Случайные графы*

Очень полезной функцией для изучения теории графов является функция RandomGraph[n,p], которая конструирует случайный помеченный граф на n вершинах, где вершины случайно соединены ребрами с заданной вероятностью p. Допускается опция Type, которая может принимать значения Directed или Undirected, по умолчанию значение – Undirected.
Построим 10-вершинные случайные графы с p, возрастающими от 0 до 1. Мы начинаем с пустого графа и заканчиваем полным графом.

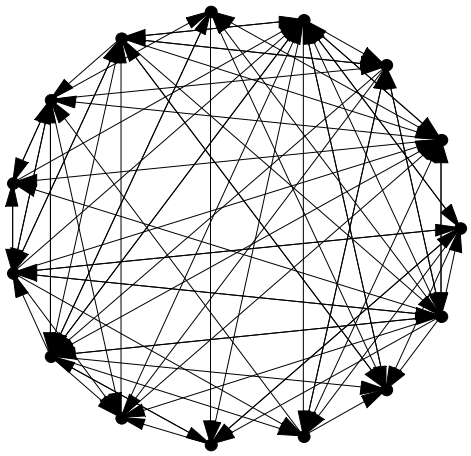
```
In[2]:= ShowGraphArray[Table[RandomGraph[10, p], {p, 0, 1, 0.25}]]
```



```
Out[2]= - GraphicsArray -
```

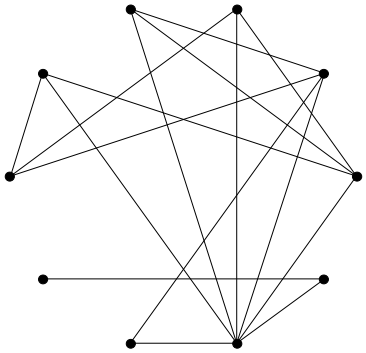
Построим ориентированный случайный граф на 10 вершинах:

```
In[3]:= ShowGraph[RandomGraph[15, 0.4, Type → Directed]]
```



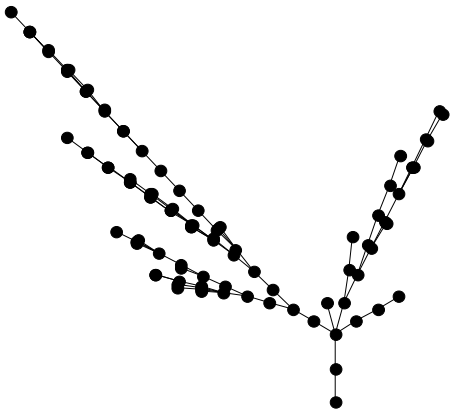
Случайные графы с ровно e ребрами и n вершинами строятся функцией ExactRandomGraph[n,e]

```
In[4]:= ShowGraph[ExactRandomGraph[10, 16]]
```



Функция RandomTree[n] конструирует случайное помеченное дерево на n вершинах.

```
In[5]:= ShowGraph[RandomTree[100]]
```



| | |
|---|---|
| RandomGraph[n, p] | конструирует случайный помеченный граф на n вершинах, где вершины случайно соединены ребрами с заданной вероятностью p. Допустима опция Type, которая принимает значения Directed or Undirected. По умолчанию Type -> Undirected. |
| RandomTree[n] | конструирует случайное помеченное дерево на n вершинах |
| ExactRandomGraph[n,e] | конструирует случайный помеченный граф на n вершинах с ровно e ребрами |

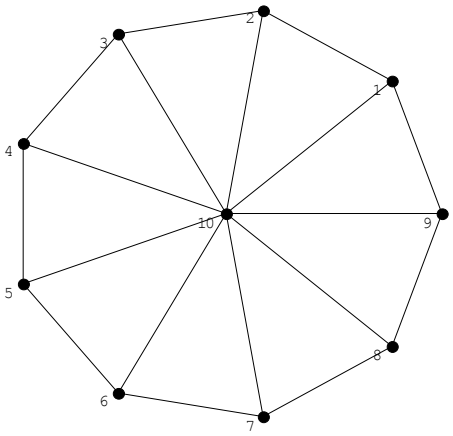
## ■5.5. Конструирование графов

В этой части покажем непосредственное конструирование графов.

### *5.5.1. Установка опций графа*

Построим граф колеса на десяти вершинах:

```
In[2]:= ShowLabeledGraph[Wheel[10]]
```




Внутреннее представление графа на изображении скрыто от пользователя. Вместо этого мы обеспечены коротким описанием графа, дающим число ребер, число вершин, и указанием, является ли граф ориентированным или нет.

```
In[3]:= g = Wheel[10]
Out[3]= -Graph:<18, 10, Undirected>-
```

Заголовок g есть Graph. Вызов заголовка объекта полезен при поиске ошибок.

```
In[4]:= Head[g]
Out[4]= Graph
```

Первый элемент g - это список ребер:

```
In[5]:= g[[1]]
Out[5]= {{{1, 10}}, {{2, 10}}, {{3, 10}}, {{4, 10}}, {{5, 10}},
        {{6, 10}}, {{7, 10}}, {{8, 10}}, {{9, 10}}, {{1, 2}}, {{2, 3}},
        {{3, 4}}, {{4, 5}}, {{5, 6}}, {{6, 7}}, {{7, 8}}, {{8, 9}}, {{1, 9}}}
```

Более корректный способ для пользователя получить эту информацию - использование функции Edges.

```
In[6]:= Edges[g]
Out[6]= {{1, 10}, {2, 10}, {3, 10}, {4, 10}, {5, 10}, {6, 10}, {7, 10}, {8, 10}, {9, 10},
         {1, 2}, {2, 3}, {3, 4}, {4, 5}, {5, 6}, {6, 7}, {7, 8}, {8, 9}, {1, 9}}
```

Количество ребер графа можно узнать, применив функцию M[g], которая возвращает количество ребер.

```
In[7]:= M[g]
Out[7]= 18
```

Второй элемент в g есть список вершин.

```
In[8]:= g[[2]]
Out[8]= {{{0.766044, 0.642788}}, {{0.173648, 0.984808}}, {{-0.5, 0.866025}},
         {{-0.939693, 0.34202}}, {{-0.939693, -0.34202}}, {{-0.5, -0.866025}},
         {{0.173648, -0.984808}}, {{0.766044, -0.642788}}, {{1., 0}}, {{0, 0}}}
```

Получим эту информацию, используя функцию Vertices[g]:

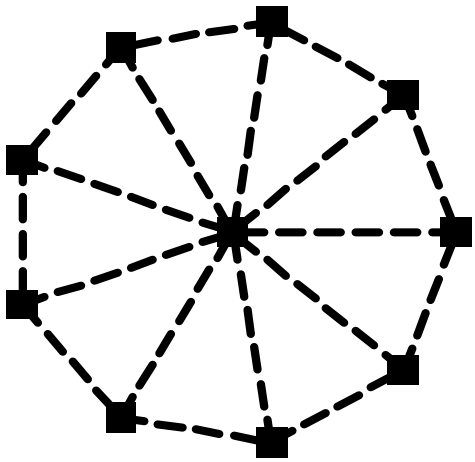
```
In[9]:= Vertices[g]
Out[9]= {{0.766044, 0.642788}, {0.173648, 0.984808}, {-0.5, 0.866025},
         {-0.939693, 0.34202}, {-0.939693, -0.34202}, {-0.5, -0.866025},
         {0.173648, -0.984808}, {0.766044, -0.642788}, {1., 0}, {0, 0}}
```

Кроме того, количество вершин можно узнать, вызвав функцию V[g], которая возвращает число вершин:

```
In[10]:= V[Star[10]]
Out[10]= 10
```

Для того чтобы изменить изображение графа используется функция SetGraphOptions. Т.к. эти опции затрагивают все вершины и ребра графа, мы считаем их как "глобальные" опции. Позже мы покажем, как установить опции так, чтобы затрагивать только некоторое подмножество вершин и ребер.

```
In[11]:= ShowGraph[
           h = SetGraphOptions[g, VertexStyle → Box[Large], EdgeStyle → ThickDashed]]
```



SetGraphOptions дополняет структуру графа информацией о том, как мы хотим изобразить граф. Эта информация сохраняется как опции графа, которые следуют за списком вершин. Таким образом, третий и четвертый элементы h - опции, которые специфицируют стиль рисования вершин и ребер соответственно.

```
In[12]:= {h[[3]], h[[4]]}
Out[12]= {VertexStyle → Box[Large], EdgeStyle → ThickDashed}
```

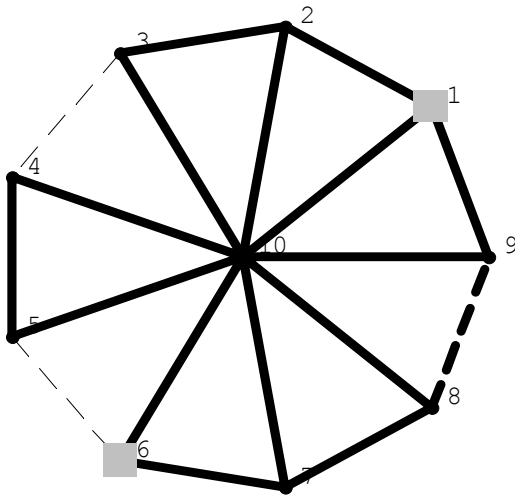
Корректный способ для пользователя получить эту информацию - использование функции GraphOptions:

```
In[13]:= GraphOptions[h]
Out[13]= {VertexStyle → Box[Large], EdgeStyle → ThickDashed}
```

Если мы хотим изменить стиль изображения отдельных вершин или ребер, то есть сделать опции локальными, нужно выделить в фигурных скобках эти вершины и желаемые опции. Продемонстрируем применение локальных опций на примере. Изобразим граф колеса с 10 вершинами, в котором вершины 1 и 6 изображены боксами серого цвета, а ребра, связывающие вершины

{3,4} и {5,6} изображены тонким пунктиром. Этот пример показывает, как стили становятся локальными, затрагивая только указанное множество ребер или вершин.

```
In[14]:= ShowLabeledGraph[
           t = SetGraphOptions[Wheel[10],
             {{1, 6, VertexStyle → Box[Large], VertexColor → Gray},
              {{3, 4}, {5, 6}, EdgeStyle → ThinDashed}, {{8, 9}, EdgeStyle → ThickDashed}},
             EdgeStyle → Thick, VertexNumberPosition → UpperRight, TextStyle → {FontSize → 14}]]
```




Факт, что мы хотим нарисовать вершины 1 и 6 как большие квадраты, сохраняется в списках первой и пятой вершин. Информация сохраняется локально, смещая глобальную информацию.

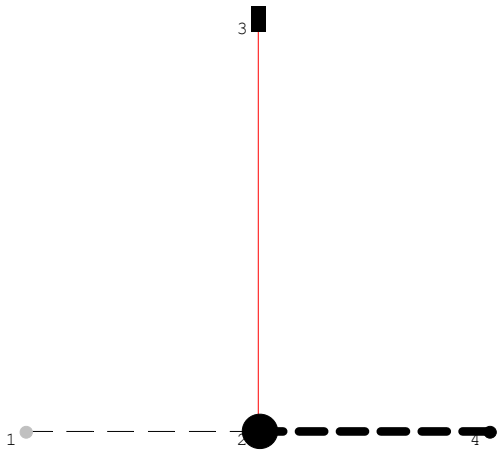
```
In[15]:= Vertices[h, All]
Out[15]= {{{0.766044, 0.642788}}, {{0.173648, 0.984808}}, {{-0.5, 0.866025}},
          {{-0.939693, 0.34202}}, {{-0.939693, -0.34202}}, {{-0.5, -0.866025}},
          {{0.173648, -0.984808}}, {{0.766044, -0.642788}}, {{1., 0}}, {{0, 0}}}
```

Таким образом, графы состоят из последовательности объектов, следующих за заголовком Graph. Первый элемент последовательности - список ребер, второй- список вершин и любые следующие опции - глобальные опции. Все локальные опции, которые затрагивают специфические вершины или ребра, сохраняются в списке, соответствующем части вершин или ребер.

Задать граф g можно с помощью Graph[e,v,opt]:

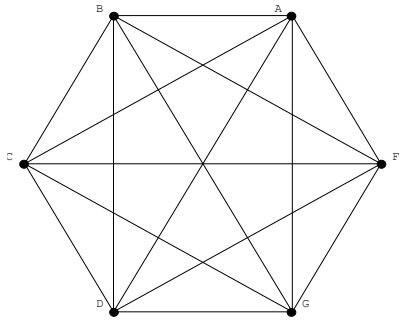
g=Graph[e,v,opt] – представляет собой граф, где e – список ребер, аннотированный графическими опциями, v – список вершин с соответствующими графическими опциями, а opt – множество глобальных опций графа. Список e имеет вид: {{{i1, j1}, opts1}, {{i2, j2}, opts2},...}, где {i1, j1}, {i2, j2},...- ребра графа, а opts1, opts2,...локальные опции этих ребер. Список v имеет вид {{{x1, y1}, opts1}, {{x2, y2}, opts2},...}, где {x1, y1}, {x2, y2},...и a opts1, opts2,...—координаты и опции первой, второй и т.д. вершин соответственно. Допустимые опции ребер (как глобальные, так и локальные) - EdgeWeight, EdgeColor, EdgeStyle, EdgeLabel, EdgeLabelColor, и EdgeLabelPosition. Допустимые (глобальные или локальные) опции вершин - VertexWeight, VertexColor, VertexStyle, VertexNumber, VertexNumberColor, VertexNumberPosition, VertexLabel, VertexLabelColor и VertexLabelPosition. Как глобальные опции могут применяться LoopPosition и EdgeDirection. Если глобальная опция и локальная опция различны, то локальная опция доминирует. Кроме того, наследуются все опции встроенной функции Plot и графического примитива Arrow. Сформируем граф p:

```
In[16]:= p = Graph[{{{1, 2}, EdgeStyle → ThinDashed}, {{2, 3}, EdgeColor → Red},
           {{2, 4}, EdgeStyle - ThickDashed}},
         {{{-1, 0}, VertexColor → Gray}, {{0, 0}, VertexStyle → Disk[Large]},
           {{0, 1}, VertexStyle → Box[0.03]}, {{1, 0}}}]
Out[16]= -Graph:<3, 4, Undirected>-

In[17]:= ShowLabeledGraph[p]
```

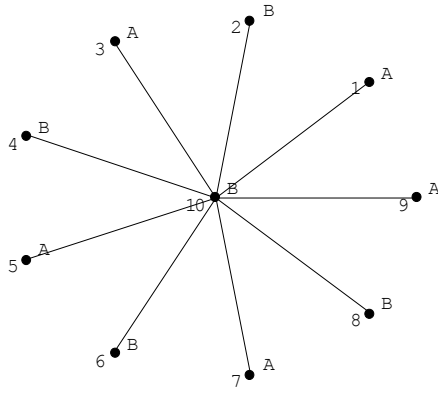



Присвоить метки вершинам графа можно, применив SetVertexLabels.
SetVertexLabels[g, l] присваивает метки из списка l вершинам графа g. Если меток в l больше, чем вершин, то лишние метки игнорируются, а если меток меньше, чем вершин, то они присваиваются циклически.

```
In[18]:= l := {A, B, C, D, G, F, K, L, M};
         ShowGraph[SetVertexLabels[CompleteGraph[6], l],
          {1, 2, 3, 4, VertexLabelPosition → UpperLeft}]
```




```
In[19]:= ShowLabeledGraph[g = SetVertexLabels[Star[10], {"A", "B"}]]
```



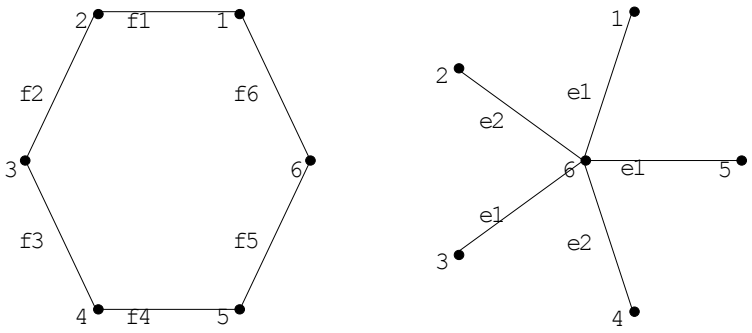

GetVertexLabels[g] возвращает список меток вершин графа. GetVertexLabels[g, vs]- возвращает список меток вершин vs.

```
In[20]:= {GetVertexLabels[g, {3, 2, 5}], GetVertexLabels[g]}
Out[20]= {{A, B, A}, {A, B, A, B, A, B, A, B, A, B}}
```

Метки ребрам можно присвоить при помощи SetEdgeLabels.

SetEdgeLabels[g, l] присваивает метки ребрам графа g. Если длина списка длиннее, чем количество ребер, то лишние метки игнорируются. Если же длина списка короче, то метки присваиваются циклически. Метка ребра располагается в середине ребра.

```
In[21]:= l1 := {f1, f2, f3, f4, f5, f6, f7, f8, f9, f10, f11, f12, f13, f14, f15};
         l2 = {e1, e2};
         ShowGraphArray[{h = SetEdgeLabels[Cycle[6], l1], p = SetEdgeLabels[Star[6], l2]},
          VertexNumber → True, PlotRange → 0.25]
```




GetEdgeLabels[g] возвращает метки ребер графа g. GetEdgeLabels[g, es] возвращает метки ребер из списка es графа g.

```
In[22]:= {GetEdgeLabels[h], GetEdgeLabels[p]} // ColumnForm
Out[22]= {f1, f2, f3, f4, f5, f6}
         {e1, e2, e1, e2, e1}
```

Располагать метки вершин и ребер можно с помощью опций EdgeLabelPosition и VertexLabelPosition, которые можно установить как в теле графа, так и в ShowGraph.

Во многих прикладных задачах теории графов используются веса ребер и вершин. Функция SetEdgeWeights[g] присваивает вещественные случайные веса из отрезка [0,1] ребрам графа g. SetEdgeWeights принимает опции WeightingFunction и WeightRange. WeightingFunction может принимать значения Random, RandomInteger, Euclidean, или LNorm[n] для неотрицательных n, или любую чистую функцию от двух аргументов, причем каждый аргумент имеет вид {Integer,

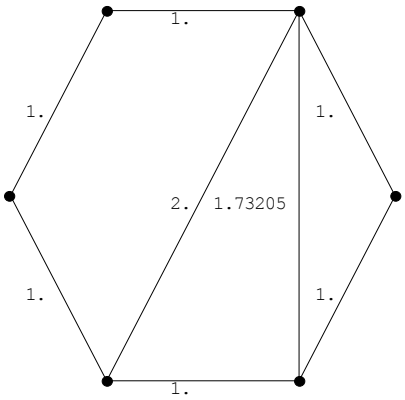
{Number, Number}}. WeightRange может быть отрезком действительных или целых чисел. По умолчанию значение для WeightingFunction устанавливается как Random, а для WeightRange – отрезок [0, 1]. SetEdgeWeights[g, e] присваивает ребрам графа g веса из списка e. SetEdgeWeights[g, e, w] присваивает веса из списка w ребрам графа из списка e.

Присвоим каждому ребру графа вес, равный евклидову расстоянию между его концевыми точками.

```
In[23]:= g = SetEdgeWeights[AddEdges[Cycle[6], {{1, 4}, {1, 5}}], WeightingFunction → Euclidean]
Out[23]= -Graph:<8, 6, Undirected>-
```

Представим веса ребер как метки ребер графа. Эти веса ребер подтверждают, что n вершин цикла равномерно расположены на окружности единичного радиуса.

```
In[24]:= ShowGraph[SetEdgeLabels[g, GetEdgeWeights[g]], TextStyle → {FontSize → 12}]
```




Теперь присвоим ребрам веса – целые числа из отрезка [0,10].

```
In[25]:= GetEdgeWeights[SetEdgeWeights[Star[10], WeightingFunction → RandomInteger,
           WeightRange → {0, 10}]]
Out[25]= {4, 10, 7, 4, 0, 6, 4, 4, 0}
```

Теперь присвоим ребрам конкретные веса из данного списка, по одному на каждое ребро.

```
In[26]:= g = SetEdgeWeights[Star[10], {6, 7, 32, 5, 6, 89, 2, 5, 1}];
         GetEdgeWeights[g]
Out[26]= {6, 7, 32, 5, 6, 89, 2, 5, 1}
```

Веса присваиваются ребрам в порядке, в котором они перечислены. Чтобы обнаружить этот порядок воспользуемся функцией Edges .

```
In[27]:= Edges[g, All]
Out[27]= {{{1, 10}, EdgeWeight → 6}, {{2, 10}, EdgeWeight → 7}, {{3, 10}, EdgeWeight → 32},
          {{4, 10}, EdgeWeight → 5}, {{5, 10}, EdgeWeight → 6}, {{6, 10}, EdgeWeight → 89},
          {{7, 10}, EdgeWeight → 2}, {{8, 10}, EdgeWeight → 5}, {{9, 10}, EdgeWeight → 1}}
```

Чтобы проверить, является ли граф взвешенным, можно проиенить тест UnweightedQ. UnweightedQ[g] – возвращает True, если все ребра имеют вес 1, (вес 1 присваивается графу по умолчанию, т.е. граф невзвешенный), и False в противном случае.

```
In[28]:= UnweightedQ[g]
Out[28]= False
```

Веса можно также присвоить вершинам графа. SetVertexWeights[g] присваивает вещественные случайные веса из отрезка [0,1] вершинам графа g. SetVertexWeights принимает опции WeightingFunction и WeightRange. WeightingFunction может принимать значения Random, RandomInteger или любую чистую функцию от двух аргументов, причем первый аргумент целое число, а второй аргумент имеет вид – {Number, Number}. WeightRange может быть отрезком действительных или целых чисел. По умолчанию значение для WeightingFunction устанавливается как Random, а для WeightRange – отрезок [0, 1]. SetVertexWeights[g, w] присваивает вершинам графа g веса из списка w. SetVertexWeights[g, vs, w] присваивает веса из списка w вершинам графа из списка vs. GetVertexWeights[g] возвращает список весов вершин графа g, а GetVertexWeights[g, vs] – список весов вершин vs.

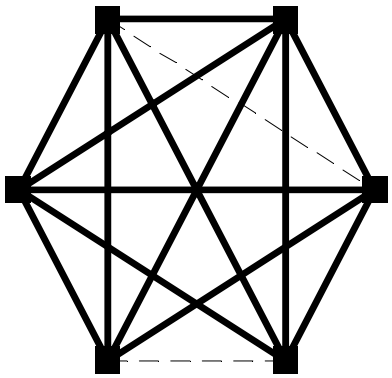
```
In[29]:= GetVertexWeights[g = SetVertexWeights[Star[5], {-5, -4, 3, 6, 9}]]
Out[29]= {-5, -4, 3, 6, 9}
```

Выделим координаты вершин графа и соответствующие им веса.

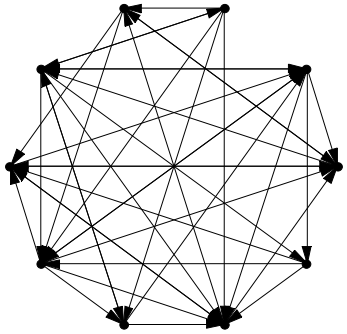
```
In[30]:= Vertices[g, All]
Out[30]= {{{0, 1.}, VertexWeight → -5}, {{-1., 0}, VertexWeight → -4},
          {{0, -1.}, VertexWeight → 3}, {{1., 0}, VertexWeight → 6}, {{0, 0}, VertexWeight → 9}}
```

Что произойдет, если опции в теле графа отличаются от опций, которые задает функция ShowGraph? Установим EdgeStyle->ThinDashed, затем в ShowGraph установим EdgeStyle->Thick. Опции, устанавливаемые в теле графа, имеют большее предпочтение. В результате, ребра изображены тонким пунктиром, а не толстыми линиями. Следующее правило предпочтения снимает неоднозначность использования опций. Опции, которые устанавливаются вместе с графом для отдельных вершин или ребер - "локальные опции" - имеют наибольшее предпочтение. Опции, которые устанавливаются в теле графа для всех вершин или для всех ребер - "глобальные "опции имеют меньшее предпочтение. Опции вне графа, такие, как в ShowGraph, имеют наименьшее предпочтение.
Изобразим ребра {4,5} и {2,6} тонким пунктиром, причем опцию EdgeStyle –>ThickDashed установим в теле графа. Опция EdgeStyle –>Thick установим глобально, поэтому она затрагивает все ребра кроме{4,5} и {2,6} . Попытка в ShowGraph изобразить все ребра графа "нормально" не имеет эффекта, т.к. эта опция имеет меньшее предпочтение.

```
In[31]:= g = SetGraphOptions[CompleteGraph[6], {{4, 5}, {2, 6}, EdgeStyle → ThinDashed},
            EdgeStyle → Thick];
          ShowGraph[g, VertexStyle → Box[Large], EdgeStyle → Normal,
           VertexNumberPosition → UpperRight]
```



```
In[33]:= ShowGraph[g = RandomGraph[10, 0.5, Type → Directed], EdgeDirection → False]
```



| | |
|---|---|
| Edges[g] | возвращает ребра графа g |
| Edges[g, All] | возвращает ребра графа g вместе с графическими опциями, ассоциированнымb с каждым ребром |
| Edges[g, EdgeWeight] | возвращает ребра графа g вместе с весом каждого ребра |
| Vertices[g] | возвращает координаты всех вершин графа g на плоскости |
| Vertices[g, All] | возвращает координаты всех вершин графа g вместе с графическими опциями каждой вершины |
| M[g] | возвращает число ребер графа g |
| V[g] | возвращает число вершин графа g |
| SetGraphOptions[g, opts] | возвращает граф g с опциями opts |
| SetGraphOptions[g, {v1, v2, ..., vopts}, gopts] | возвращает граф g с опциями vopts для вершин v1, v2, ..., и с опциями gopts для графа g |
| SetGraphOptions[g, {e1, e2,..., eopts}, gopts] | возвращает граф g с опциями eopts для ребер e1,e2 ..., и с опциями gopts для графа g |
| SetGraphOptions[g, {{elements1, opts1}, {elements2, opts2},...}, opts] | возвращает граф g с опциями opts1 для элементов последовательности elements1 и т.д. elements могут быть последовательностью ребер или вершин. Тег One or All может быть использован как аргумент перед любой опцией. По умолчанию принимает значение All и полезна в случае, когда граф имеет кратные ребра |
| GraphOptions[g] | возвращает опцииграфа g |
| GraphOptions[g, v] | возвращает опции графа g, ассоциированные с вершиной v |
| GraphOptions[g, {u, v}] | возвращает опции графа g, ассоциированные с ребром {u, v} |
| Graph[e,v,opt] | возвращает граф, где e – список ребер, аннотированный графическими опциями, v – список вершин с соответствующими графическими опциями, а opt – множество глобальных опций графа |
| SetVertexLabels[g, l] | присваивает метки вершинам графа g. Если длина списка длиннее, чем количество ребер, то лишние метки игнорируются. Если же длина списка короче, то метки присваиваются циклически |
| SetEdgeLabels[g, l] | присваивает метки ребрам графа g. Если длина списка длиннее, чем количество ребер, то лишние метки игнорируются. Если же длина списка короче, то метки присваиваются циклически |
| GetVertexLabels[g] | возвращает список меток вершин графа |
| GetEdgeLabels[g, es] | возвращает метки ребер из списка es графа g |
| SetEdgeWeights[g] | присваивает вещественные случайные веса из отрезка [0,1] ребрам графа g. SetEdgeWeights принимает опции WeightingFunction и WeightRange. WeightingFunction может принимать значения Random, RandomInteger, Euclidean, или LNorm[n] для неотрицательных n, или любую чистую функцию от двух аргументов, причем каждый аргумент имеет вид {Integer, {Number, Number}}. WeightRange может быть отрезком действительных или целых чисел. По умолчанию значение для WeightingFunction устанавливается как Random, а для |

| | |
|---|---|
| | WeightRange – отрезок [0, 1]. SetEdgeLabels[g, l] SetEdgeLabels[g, l] |
| GetEdgeWeights[g] | возвращает список весов ребер графа g |
| GetEdgeWeights[g, es] | возвращает список весов ребер из списка es графа g |
| UnweightedQ[g] | возвращает True, если все ребра имеют вес 1, (вес 1 присваивается графу по умолчанию, т.е. граф невзвешенный), и False в противном случае |

### *5.5.2. Построение графа с помощью списка ребер*

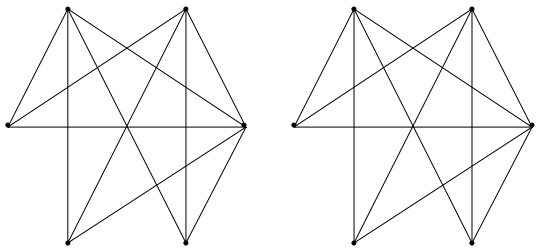
Граф можно задать также списком ребер, где каждая вершина графа представлена как упорядоченная или неупорядоченная пара вершин. Combinatorica обеспечивает функциями FromUnorderedPairs и FromOrderedPairs, которые трансформируют неупорядоченные и упорядоченные пары натуральных чисел соответственно в ориентированные или неориентированные графы.

Построим, например, с помощью упорядоченных пар ребер граф Пойа перестановки l={6, 3, 4, 5, 1, 2}. Выделим упорядоченные пары элементов перестановки l с помощью функции f:

```
In[2]:= l = {6, 3, 4, 5, 1, 2};
In[3]:= t = {};
        f[l_] := Do[If[l[[i]] > l[[i + r]], k = {l[[i + r]], l[[i]]}; t = Append[t, k], x = 0],
          {i, 1, 6}, {r, 1, 6 - i}];
```

Как уже упоминалось, построить этот граф можно с помощью встроенной функции PermutationGraph[l], которая возвращает граф Пойа перестановки l.

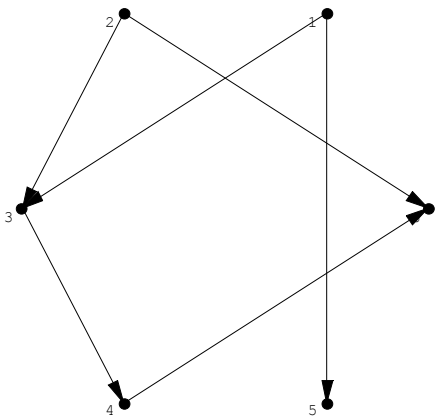
```
In[5]:= ShowGraphArray[{PermutationGraph[l], FromUnorderedPairs[t]}]
```



```
Out[5]= - GraphicsArray -
```

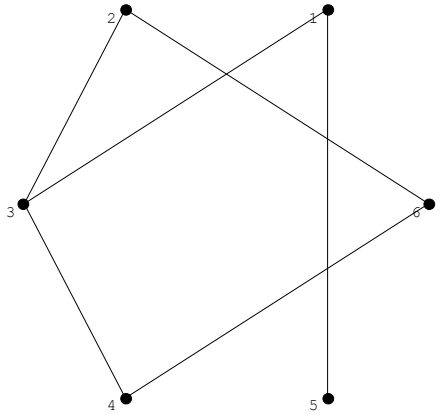
Рассмотрим ориентированный граф.

```
In[6]:= ShowLabeledGraph[g = FromOrderedPairs[{{1, 3}, {2, 3}, {4, 6}, {1, 5}, {2, 6}, {3, 4}}],
          EdgeDirection → False]
```



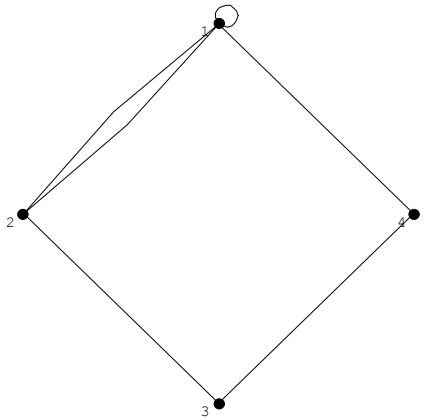
Таким образом, попытка убрать направление в ShowGraph не увенчалась успехом. Если мы не хотим видеть стрелки, то можно убрать их с помощью MakeUndirected

```
In[7]:= ShowLabeledGraph[MakeUndirected[g]]
```




Обратными функциями к функциям FromUnorderedPairs и FromOrderedPairs являются соответственно ToOrderedPairs и ToUnorderedPairs:

```
In[8]:= l = {{1, 2}, {1, 4}, {2, 3}, {3, 4}, {1, 1}, {1, 2}};
        t = FromUnorderedPairs[l]; ShowLabeledGraph[t]
        Null
```




Устанавливая опцию Type, мы получим только те пары, которые соответствуют лежащему в основе простому графу.

```
In[11]:= {ToUnorderedPairs[g, Type → Simple], ToOrderedPairs[t]} // ColumnForm
Out[11]= {{1, 3}, {1, 5}, {2, 3}, {2, 6}, {3, 4}, {4, 6}}
         {{2, 1}, {4, 1}, {3, 2}, {4, 3}, {2, 1}, {1, 2}, {1, 4}, {2, 3}, {3, 4}, {1, 1}, {1, 2}}
```

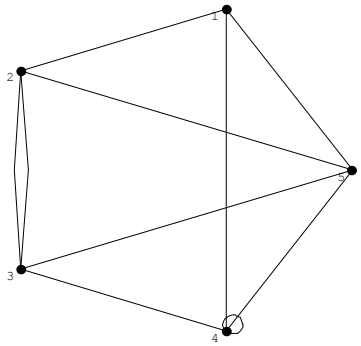
| | |
|---|---|
| FromUnorderedPairs[l] | конструирует список ребер графа из списка неупорядоченных пар l, используя циркулярное вложение. Допустима опция Type, которая принимает значения Directed или Undirected. По умолчанию Type -> Undirected |
| FromUnorderedPairs[l, v] | то же самое, но использует v как список вершин |
| FromOrderedPairs[l] | конструирует список ребер графа из списка упорядоченных пар l, используя циркулярное вложение. Допустима опция Type, которая принимает значения Directed or Undirected. По умолчанию Type -> Directed |
| FromOrderedPairs[l, v] | то же самое, но использует v как список вершин |
| ToUnorderedPairs[g] | конструирует список неупорядоченных пар, представляющих ребра графа g. Допустима опция Type, которая принимает зна- |

| | |
|---|---|
| | чения All или Simple . По умолчани Type -> All. Type -> Simple производит простой граф |
| ToOrderedPairs[g] | конструирует список упорядоченных пар, представляющих ребра графа g.Если g неориентированный граф, то каждое ребро интерпретируется как две упорядоченные пары Допустима опция Type, которая принимает значения All или Simple. По умолчани Type -> All. Type -> Simple игнорирует параллельные ребра и петли |
| MakeUndirected[g] | Возвращает лежащий в основе неориентированный граф данного ориентированного графа g |

### *5.5.3. Построение графов с помощью списков смежности*

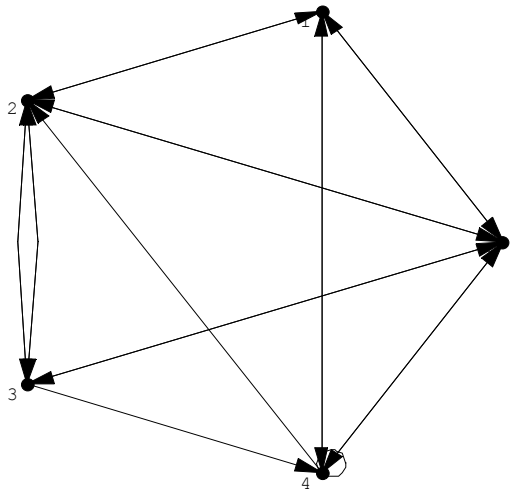
Граф можно также задавать списками смежности. Первый подсписок списка смежности графа на n вершинах перечисляет номера вершин, смежных первой вершине, второй – номера вершин, смежных второй вершине и т.д. Как правило, списки смежности для различных вершин отличаются по размеру, и представление графа списком смежности – намного более компактное представление графа, чем представление графа матрицей смежности. Функция FromAdjacencyLists конструирует граф из списков смежности.

```
In[2]:= ShowLabeledGraph[
          g = FromAdjacencyLists[
            {{2, 4, 5}, {1, 3, 3, 5}, {2, 2, 4, 5}, {1, 2, 4, 5},
             {1, 2, 3, 4}}]]
```




FromAdjacencyLists принимает опцию Type, чтобы специфицировать, будет ли конструируемый граф ориентированным или нет. По умолчанию принимается опция Undirected.

```
In[3]:= ShowLabeledGraph[
          k = FromAdjacencyLists[
            {{2, 4, 5}, {1, 3, 3, 5}, {2, 2, 4, 5}, {1, 2, 4, 5},
             {1, 2, 3, 4}}, Type → Directed]]
```




Функция ToAdjacencyLists обратна FromAdjacencyLists, она берет граф и возвращает список смежности.

```
In[4]:= ToAdjacencyLists[k] // ColumnForm
Out[4]= {2, 4, 5}
        {1, 3, 3, 5}
        {2, 2, 4, 5}
        {1, 2, 4, 5}
        {1, 2, 3, 4}
```

ToAdjacencyLists допускает опцию Type, принимающую значения All или Simple. Type->Simple игнорирует петли и параллельные ребра

```
In[5]:= ToAdjacencyLists[k, Type → Simple] // ColumnForm
Out[5]= {2, 4, 5}
        {1, 3, 5}
        {2, 4, 5}
        {1, 2, 5}
        {1, 2, 3, 4}
```

Если мы хотим присвоить веса ребрам, то следует применить тэг EdgeWeight.

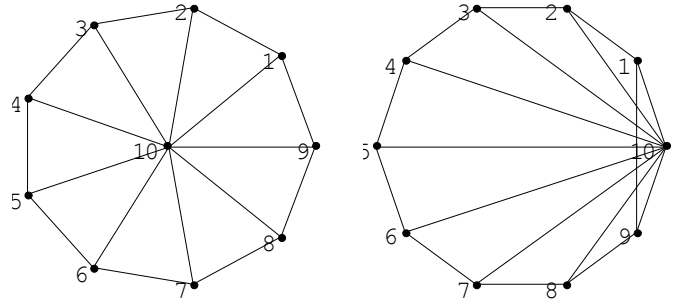
ToAdjacencyLists[g, EdgeWeight] возвращает список смежности вместе с весами ребер. SetEdgeWeights по умолчанию присваивает ребрам графа случайные веса из отрезка [0,1].

Более интересное множество весов – множество евклидовых расстояний между вершинами графа. Для звезды они одинаковы, т.к. все вершины равноудалены от центра.

```
In[6]:= ToAdjacencyLists[SetEdgeWeights[Star[5],
          WeightingFunction -> Euclidean], EdgeWeight] //
        ColumnForm
Out[6]= {{5, 1.}}
        {{5, 1.}}
        {{5, 1.}}
        {{5, 1.}}
        {{1, 1.}, {2, 1.}, {3, 1.}, {4, 1.}}
```

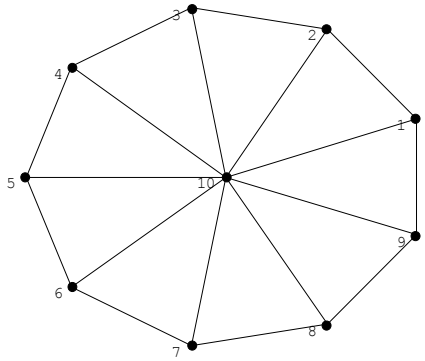
Функция FromAdjacencyLists и ToAdjacencyLists взаимнообратны, поэтому их суперпозиция есть тождественный оператор. Однако, информация о первоначальном вложении теряется и заменяется по умолчанию на циркулярное вложение.

```
In[7]:= g = Wheel[10];
        h = FromAdjacencyLists  [ToAdjacencyLists[g]];
        ShowGraphArray[{g, h}, VertexNumber → True]
```




Изобразим граф колеса в привычном для нас виде:

```
In[9]:= ShowLabeledGraph[RadialEmbedding[h]]
```




| | |
|---|---|
| FromAdjacencyLists[l] | конструирует список ребер графа g из списка смежности l, используя циркулярное вложение. Допустима опция Type, которая принимает значения Directed or Undirected. По умолчанию Type -> Undirected |
| FromAdjacencyLists[l, v] | использует v как вложение результирующего графа |
| ToAdjacencyLists[g] | конструирует список смежности графа g. Допустима опция Type, которая принимает значения All или Simple. По умолчанию Type -> All. Type -> Simple удаляет параллельные ребра и петли |
| ToAdjacencyLists[g, EdgeWeight] | возвращает список смежности вместе с весами ребер |

### *5.5.4. Построение графов с помощью матрицы смежности. Матрица инцидентности*.

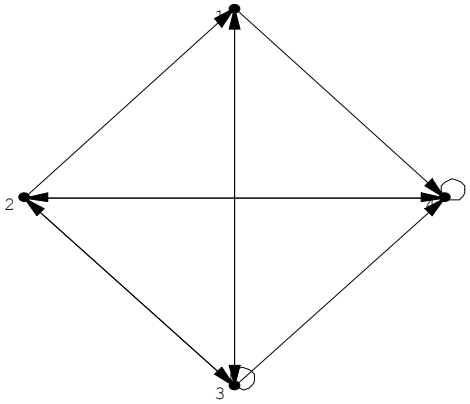
Пусть g=(E,V) ориентированный граф без параллельных ребер, в котором список вершин $V=\{v_1,v_2,\ldots,v_n\}$. Матрицей смежности $M=[m_{ij}]$ графа g называется квадратная матрица порядка n, элементы которой $m_{ij}$ равны единице, если вершины $v_i$, $v_j$ соединены ребром, и нулю в противном случае. Отметим следующее очень интересное свойство матрицы смежности:

**Теорема.** (i,j)-элемент матрицы $M^r$ равен числу ориентированных маршрутов длины r из вершины $v_i$ в вершину $v_j$.

Функция FromAdjacencyMatrix[m] конструирует граф из соответствующей матрицы смежности. FromAdjacencyMatrix[m, v] использует v как вложение графа. Функция принимает опцию

Type, которая принимает значения Directed или Undirected. По умолчанию Type -> Undirected. FromAdjacencyMatrix[m, EdgeWeight] интерпретирует элементы матрицы m как веса ребер.

```
In[2]:= ShowLabeledGraph[
          m = FromAdjacencyMatrix[{{0, 0, 1, 1}, {1, 0, 1, 1}, {1, 1, 1, 1}, {0, 1, 0, 1}},
            Type → Directed]]
```




Функция ToAdjacencyMatrix[g] конструирует матрицу смежности графа g. Эта функция допускает опцию Type, принимающую значения All или Simple. По умолчанию Type -> All, а установка Type –> Simple удаляет петли и параллельные, если таковые существуют. ToAdjacencyMatrix[g, EdgeWeight] – возвращают веса ребер как элементы матрицы смежности, присваивая вес ∞ отсутствующим ребрам.
В матрице смежности полного графа все элементы кроме главной диагонали равны 1.

```
In[3]:= ToAdjacencyMatrix[CompleteGraph[4]] // MatrixForm
Out[3]//MatrixForm=
```

$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

ToAdjacencyMatrix вместе с EdgeWeight конструирует матрицу с весами ребер.
В этом примере ∞ на пересечении первой строки и третьего столбца указывает на отсутствие ребра между первой и третьей вершиной, тогда как число на пересечении первой строки и второго столбца показывает, что существует ребро (1, 2) веса 19.

```
In[4]:= g = SetEdgeWeights[Star[5], WeightingFunction → RandomInteger, WeightRange → {10, 20}];
        ToAdjacencyMatrix[g, EdgeWeight] // MatrixForm
Out[5]//MatrixForm=
```

$$\begin{pmatrix} \infty & \infty & \infty & \infty & 16 \\ \infty & \infty & \infty & \infty & 10 \\ \infty & \infty & \infty & \infty & 18 \\ \infty & \infty & \infty & \infty & 10 \\ 16 & 10 & 18 & 10 & \infty \end{pmatrix}$$

Обозначим через l – матрицу смежности графа m

```
In[7]:= l = ToAdjacencyMatrix[m]

Out[7]= {{0, 0, 1, 1}, {1, 0, 1, 1}, {1, 1, 1, 1}, {0, 1, 0, 1}}
```

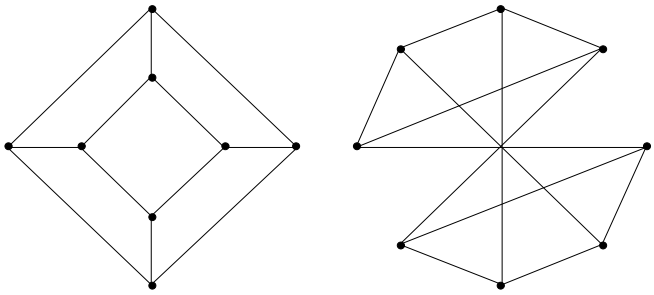
Рассмотрим, например, третью степень матрицы l:

```
In[8]:= l.l.l // MatrixForm
```

```
Out[8]//MatrixForm=
( 3  3  4   6 )
( 4  5  5   8 )
( 5  7  7  11 )
( 2  3  3   5 )
```

Элемент (1,2) этой матрицы определяет число ориентированных маршрутов длины три из первой вершины во вторую. Этими маршрутами являются {{1,4},{4,4},{4,2}}, {{1,3},{3,3},{3,2}},{{1,3},{3,4},{4,2}}.
Функция FromAdjacencyMatrix является обратной к ToAdjacencyMatrix. Она берет матрицу с неотрицательными целыми числами, интерпретируя ее как матрицу смежности графа, и реконструирует соответствующий граф.
Рассмотрим матрицу смежности графа куба и применим FromAdjacencyMatrix. Результатом является граф куба, но изображенный непривычным образом.

```
In[9]:= g = CubicalGraph;
        ShowGraphArray[{g, h = FromAdjacencyMatrix [ToAdjacencyMatrix [g]]}]
```



На первый взгляд кажется, что в графе h девять вершин, но на самом деле “вершина” в центре является иллюзией, это просто пересечение некоторых ребер графа. Рассмотрим последовательность степеней графа h.

```
In[10]:= DegreeSequence[h]
Out[10]= {3, 3, 3, 3, 3, 3, 3, 3}
```

Чтобы разрешить эти сомнения, мы протестируем, являются ли два вышеприведенных графа изоморфными.

```
In[11]:= IsomorphicQ[g, h]
 Out[11]= True
```

Матрицей инцидентности графа с n вершинами и m ребрами называется n□m матрица из нулей и единиц с элементами [v,e]=1 тогда и только тогда, когда вершина v инцидентна ребру e. Для ориентированных графов [v,e]=1, если ребро e выходит из v. Матрица инцидентности была впервые определена Кирхгоффом в контексте задачи об остовных деревьях.
Из определения следует, что всякий столбец матрицы инцидентности содержит две единицы. В матрице инцидентности дерева существует, по крайней мере, две строки, содержащие ровно одну единицу. Убедимся в этом на примере.

```
In[12]:= IncidenceMatrix[CompleteKaryTree[6, 3]] // MatrixForm
```

Out[12]//MatrixForm=

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Матрица смежности полного графа на n вершинах содержит в каждой строке ровно n единиц:

In[13]:= **IncidenceMatrix[CompleteGraph[6]] // MatrixForm**

Out[13]//MatrixForm=

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix}$$

| | |
|---|---|
| FromAdjacencyMatrix[m] | конструирует граф из данной матрицы смежности m, используя циркулярное вложение. Допустима опция Type, которая принимает значения Directed or Undirected. По умолчанию Type -> Undirected |
| FromAdjacencyMatrix[m, v] | использует список v как вложение графа g |
| FromAdjacencyMatrix[m, EdgeWeight] | интерпретирует m как веса ребер, присваивая вес ∞ отсутствующим ребрам и конструирует граф, используя циркулярное вложение. Допустима опция Type вместе с EdgeWeight |
| FromAdjacencyMatrix[m, v, EdgeWeight] | использует список v как вложение графа g |
| ToAdjacencyMatrix[g] | конструирует матрицу смежности графа g. Допустима опция Type, принимающая значения All или Simple. По умолчанию Type -> All, Type -> Simple игнорирует параллельные ребра и петли |
| ToAdjacencyMatrix[g, EdgeWeight] | возвращает веса ребер как элементы матрицы смежности, присваивая вес ∞ отсутствующим ребрам |
| IncidenceMatrix[g] | возвращает матрицу инцидентности графа g |

### *5.5.5 Реализация последовательности степеней*

Последовательность степеней неориентированного графа – число ребер, инцидентных каждой вершине. Вообще самая элементарная теорема теории графов состоит в том, что сумма последовательности степеней должна быть четной, или, что эквивалентно, в любом простом конечном графе количество нечетных вершин – четно. Для удобства последовательности степеней упорядочены в невозрастающем порядке. Степень вершины иногда называется валентностью, а минимальная и максимальная степени графа G обозначаются $\delta(G)$ и $\Delta(G)$ соответственно. Последовательность степеней называется графической, если существует простой граф с той же последовательностью степеней. Известно, что последовательность степеней является графической тогда и только тогда, когда последовательность удовлетворяет следующему условию для каждого целого r<n

$$\sum_{i=1}^{r} d_i \le r(r-1) + \sum_{i=r+1}^{n} \min(r, d_i).$$

Функция GraphicQ[s] возвращает True, если последовательность s целых положительных чисел является графической.
По определению последовательность степеней любого графа является графической:

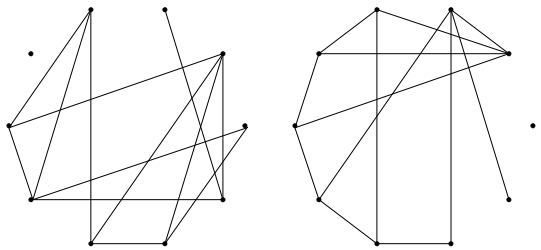
```
In[3]:= GraphicQ[DegreeSequence[RandomGraph[100, 0.8]]]
Out[3]= True
```

Функция RealizeDegreeSequence[s] конструирует полуслучайный граф с последовательностью степеней s.
Рассмотрим последовательность степеней:

```
In[4]:= d = DegreeSequence[g = RandomGraph[10, 0.5]]
Out[4]= {4, 4, 3, 3, 3, 3, 3, 2, 1, 0}
```

Построим граф с последовательностью степеней d. Граф g и сконструированный граф могут быть неизоморфны.

```
In[5]:= ShowGraphArray[{g, h = RealizeDegreeSequence[d]}]
```



```
Out[5]= - GraphicsArray -

In[6]:= IsomorphicQ[g, h]
Out[6]= False
```

Большинство из полуслучайных графов, сконструированных с помощью RealizeDegreeSequence, могут быть не изоморфны g, указывая, что порожденные графы неизоморфны, даже если они имеют одни и те же последовательности степеней.

```
In[7]:= Count[Map[IsomorphicQ[g, #] &, Table[RealizeDegreeSequence[d], {500}]],
          False]
Out[7]= 498
```

С помощью RealizeDegreeSequence можно сконструировать даже несвязные графы.

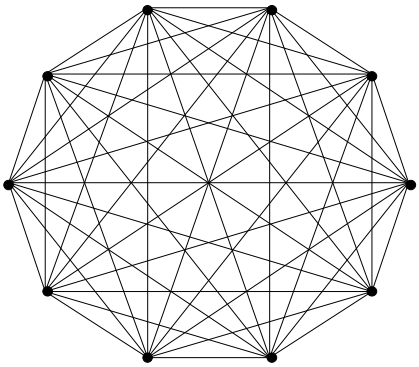
```
In[8]:= ConnectedComponents[RealizeDegreeSequence[{2, 2, 2, 1, 1, 1, 1}]]
Out[8]= {{1, 2, 5, 6}, {3, 4, 7}}
```

| | |
|---|---|
| GraphicQ[s] | возвращаетTrue, если последовательность целых чисел s является графической и False в противном случае |
| RealizeDegreeSequence[s] | конструирует полуслучайный граф с последовательностью степеней s |

### *5.5.6 Построение графов бинарных отношений*

Бинарным отношением R на множестве объектов S называется подмножество декартова произведения S×S. Таким образом, каждый элемент в R есть упорядоченная пара элементов из S, и поэтому R может быть представлен ориентированным графом.
Функция MakeGraph[v, f] конструирует граф, чьи вершины состоят из элементов множества v, и две вершины x и y соединены ребром, если бинарное отношение, определяемое булевой функцией f для этих вершин, принимает значение True. MakeGraph допускает две опции Type и VertexLabel. По умолчанию они принимают значения Directed и False соответственно.
Рассмотрим бинарное отношение «x не равно y» Это отношение симметрично, поэтому ребра ориентированы в двух направлениях. Сделаем граф неориентированным, установив опцию Type->Undirected

```
In[2]:= g = MakeGraph[Range[10], (#1 ≠ #2) &, Type → Undirected]; ShowGraph[g]
```



Очевидно, что граф является полным:

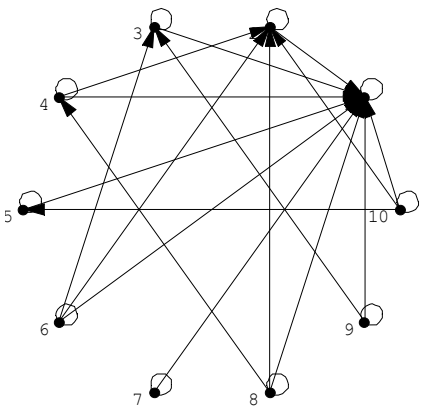
```
In[3]:= CompleteQ[g]
Out[3]= True
```

Покажем, что это бинарное отношение является антирефлексивным (граф не имеет петель), симметричным.

```
In[4]:= {ReflexiveQ[g], SymmetricQ[g]}
Out[4]= {False, True}
```

Рассмотрим отношение делимости на множестве натуральных чисел. Это отношение рефлексивно, антисимметрично и транзитивно, следовательно, изображающий это отношение граф имеет петли в каждой вершине и является ориентированным. Так как 1 является делителем любого натурального числа, то первая вершина смежна всем вершинам графа.

```
In[5]:= ShowGraph[p = MakeGraph[Range[10], (Mod[#1, #2] == 0) &], VertexNumber → True,
          TextStyle → {FontSize → 12}];
```



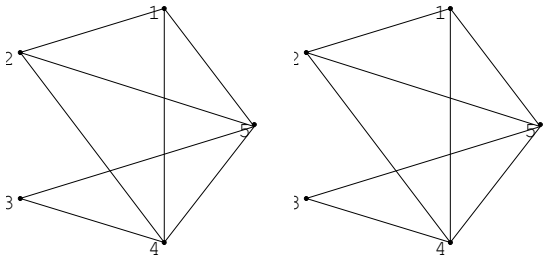
```
In[6]:= {ReflexiveQ[p], AntiSymmetricQ[p], TransitiveQ[p]}

Out[6]= {True, True, True}
```

Функция OddGraph[n] конструирует граф, чьи вершины – (n-1) - подмножества (2n-1) – элементного множества, а ребром связана пара вершин, которые соответствуют непересекающимся подмножествам.
Покажем, что $O_2$ есть $K_3$, а $O_3$ – граф Петерсена:

```
In[8]:= {IsomorphicQ[PetersenGraph,
          MakeGraph[KSubsets[Range[5], 2], (Intersection[#1, #2] == {}) &, Type → Undirected]],
         CompleteQ[MakeGraph[KSubsets[Range[3], 1], (Intersection[#1, #2] == {}) &,
           Type → Undirected]]}
Out[8]= {True, True}
```

Сконструируем с помощью MakeGraph так называемый интервальный граф. IntervalGraph[l] определяется множеством интервалов на прямой, где каждый интервал представляет вершину, а ребро соединяет два интервала тогда и только тогда, когда интервалы пересекаются.

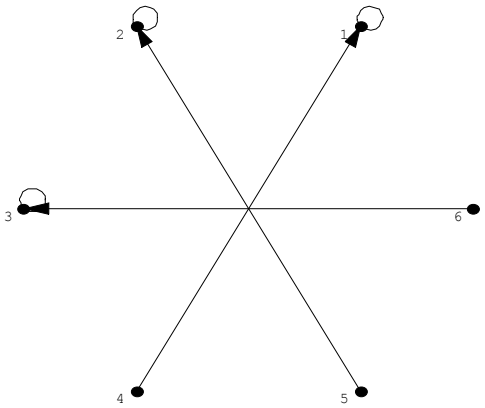
```
In[9]:= v = {{1, 2}, {1.5`, 3}, {4, 6}, {1, 8}, {0.5`, 5}}; s = Map[Interval, v];
        p = MakeGraph[s, (IntervalIntersection[#1, #2] =!= Interval[] && #1 =!= #2) &,
          Type → Undirected];
        ShowGraphArray[{IntervalGraph[v], p}, VertexNumber → True]
```




```
Out[10]= - GraphicsArray -
```

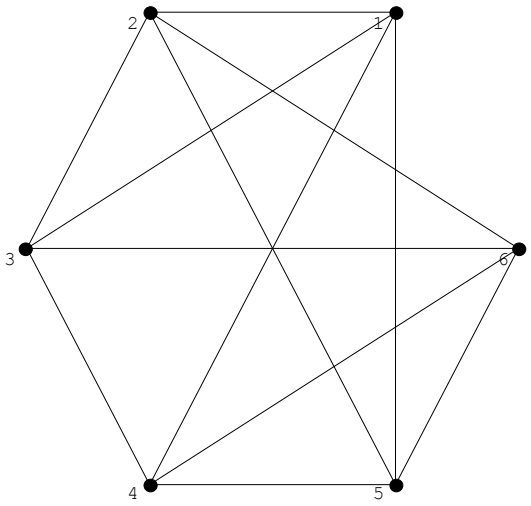
Графы можно также конструировать с помощи функции FunctionalGraph.
FunctionalGraph[f, v] берет множество v и функцию f, действующую из v в v и конструирует ориентированный граф с вершинами из множества v и ребрами (x, f(x)) для каждого x из v.
FunctionalGraph[f, v], где f – список функций, конструирует граф с вершинами из множества v и ребрами (x, fi(x)) для каждой fi в f. Допускается опция Type, принимающая значения Directed или Undirected. По умолчанию Type -> Directed. FunctionalGraph[f, n] берет натуральное n и функцию, действующую из множества {0,1,..., n-1} на себя и конструирует ориентированный граф с вершинами из множества {0, 1,..., n-1} и ребрами {x, f(x)} для каждой вершины x.
Построим граф на шести вершинах {0, 1, 2, 3, 4, 5, 6}, причем ребром соединена пара {x, Mod[x,3]]:

```
In[11]:= ShowLabeledGraph[FunctionalGraph[Mod[#1, 3] &, {0, 1, 2, 3, 4, 5}]]
```




Построим граф на множестве 6-перестановок. Ребра этого графа $(x,f_i(x))$, где $f_1(x)$ –умножение перестановки x на перестановку {2,1,3}, $f_2(x)$ - умножение перестановки x на перестановку {3,1,2}, $f_3(x)$ – на перестановку {3,1,2}.

```
In[12]:= ShowLabeledGraph[
           FunctionalGraph[{Permute[#, {2, 1, 3}] &, Permute[#, {1, 3, 2}] &,
             Permute[#, {3, 1, 2}] &}, Permutations[3], Type → Undirected]]
```



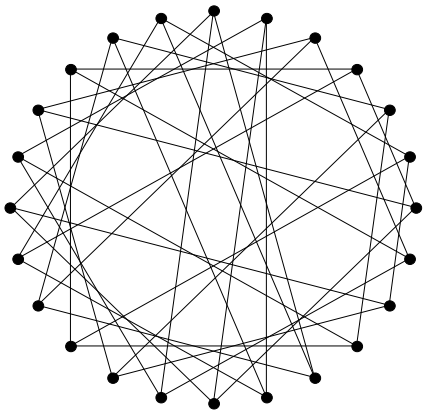

Функциональный граф предоставляет удобный способ конструирования графов Кэли. Граф Кэли определяется группой G и системой образующих S для G. Вершины графа - элементы G, ребро из группового элемента g к групповому элементу h существует тогда и только тогда, когда существует s в S, такой, что g×s=h.
Рассмотрим систему образующих для симметрической группы порядка 4.

```
In[13]:= s = Table[p = Range[4]; {p[[1]], p[[i]]} = {p[[i]], p[[1]]}; p, {i, 2, 4}]
Out[13]= {{2, 1, 3, 4}, {3, 2, 1, 4}, {4, 2, 3, 1}}
```

Придадим этим образующим «функциональную форму»:

```
In[14]:= l = Map[Function[x, Permute[x, #]] &, s]
Out[14]= {Function[x, Permute[x, {2, 1, 3, 4}]],
          Function[x, Permute[x, {3, 2, 1, 4}]], Function[x, Permute[x, {4, 2, 3, 1}]]}
```

```
In[15]:= g = FunctionalGraph[l, Permutations[4], Type → Undirected]; ShowGraph[g]
```



Один и тот же граф можно сконструировать как с помощью MakeGraph, так и с помощью FunctionalGraph, но иногда одним способом граф строится быстрее, чем другим.

```
In[16]:= n = 500;
         Timing[h = MakeGraph[Range[0, n - 1], (Mod[#1 + 1, n] == #2) || (Mod[#1 + 2, n] == #2) &]]
Out[16]= {0.656 Second, -Graph:<1000, 500, Directed>-}

In[17]:= Timing[g = FunctionalGraph[{Mod[# + 1, n] &, Mod[# + 2, n] &}, Range[0, n - 1]]]
Out[17]= {0.094 Second, -Graph:<1000, 500, Directed>-}
```

Два графа, построенные выше, действительно идентичны:

```
In[18]:= IdenticalQ[g, h]
Out[18]= True
```

| | |
|---|---|
| ReflexiveQ[g] | возвращает True, если матрица смежности графа g определяет рефлексивное бинарное отношение |
| SymmetricQ[r] | проверяет, является ли квадратная матрица r симметрической |
| SymmetricQ[g] | возвращает True, если граф g определяет симметричное бинарное отношение |
| TransitiveQ[g] | возвращает True, если граф g определяет транзитивное бинарное отношение |
| AntiSymmetricQ[ g] | возвращает True, если матрица смежности графа g определяет антисимметричное бинарное отношение |
| MakeGraph[v, f] | конструирует граф, чьи вершины соответствуют списку v, и пара вершин x и y связаны ребром, если булева функция f(x,y) принимает значение True. Применимы опции Type и VertexLabel. Type может принимать значения Directed или Undirected. По умолчанию Type-> Directed. VertexLabel может принимать значения True или False, со значением False по умолчанию. VertexLabel -> True присваивает метки списка v вершинам графа |
| FunctionalGraph[ f, v] | берет множество v и функцию f, действующую из v в v и конструирует ориентированный граф с вершинами из множества v и ребрами (x, f(x)) для каждого x из v |
| FunctionalGraph[ f, v] | f – список функций, конструирует граф с вершинами из множества v и ребрами (x, fi(x)) для каждой fi в f. Допускается опция Type, принимающая значения Directed или Undirected. По умолчанию Type -> Directe |
| FunctionalGraph[ f, n] | берет натуральное n и функцию, действующую из множества {0,1,..., n-1} на себя и конструирует ориентированный граф с |

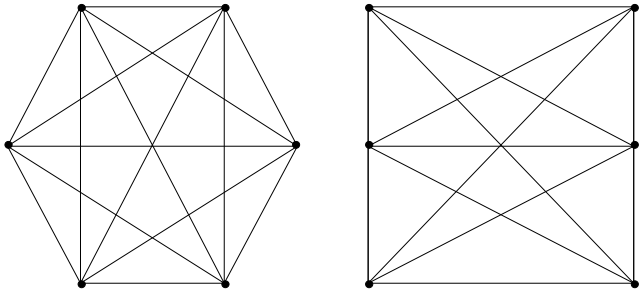
| | |
|---|---|
| | вершинами из множества {0, 1,..., n-1} и ребрами {x, f(x)} для каждой вершины x |
| IntervalGraph[l] | конструирует интервальный граф, определенный списком интервалов l |
| OddGraph[n] | конструирует граф, чьи вершины – (n-1) – подмножества (2n-1) – элементного множества, а ребром связана пара вершин, которые соответствуют непересекающимся подмножествам |

## ■5.6. Модификация графов

### *5.6.1. Прибавления, удаления вершин и ребер графа и изменения множества вершин и ребер*

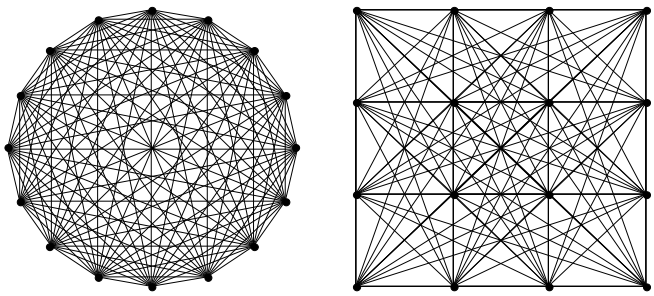
ChangeVertices и ChangeEdges – полезные функции, которые позволяют нам изменять множество вершин или ребер графа. Первый аргумент в ChangeVertices граф, а второй-- список вершин. Функция ChangeVertices[g, v] заменяет вершины графа g на вершины из списка v. Каждая вершина может быть специфицирована своими координатами {x,y} или {{x,y},options}, где options – графическая информация об этих вершинах. ChangeEdges обладает теми же возможностями.
Заменим вершины полного графа на шести вершинах на вершины решетчатого графа на шести вершинах.

```
In[2]:= ShowGraphArray[{g = CompleteGraph[6], ChangeVertices[g, Vertices[GridGraph[2, 3]]]}];
```



Таким образом, применение функции ChangeVertices[g] – удобный способ изменения вложения графа.
Теперь заменим ребра 4x4 решетчатого графа на ребра полного графа с 16 вершинами.

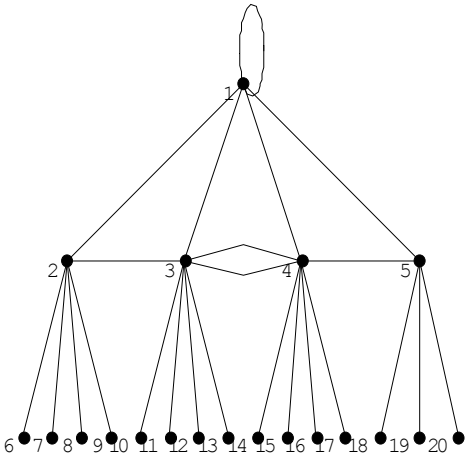
```
In[3]:= ShowGraphArray[
        {CompleteGraph[16], ChangeEdges[GridGraph[4, 4], Edges[CompleteGraph[16]]]}]
```



Теперь рассмотрим очень важную функцию AddEdges , которая позволяет прибавлять одно ребро, несколько ребер или несколько ребер вместе с информацией об их изображении.

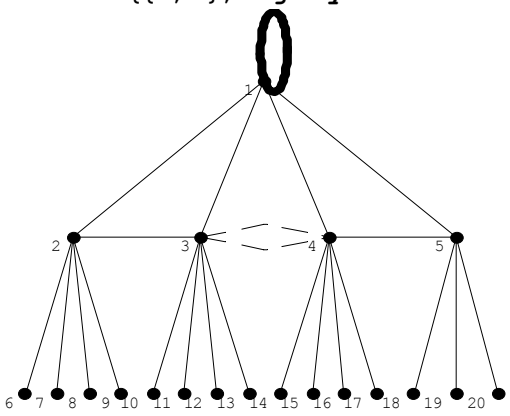
Прибавим к графу полного дерева несколько ребер, при этом корректно изображаются петли и параллельные ребра:

```
In[4]:= ShowLabeledGraph[AddEdges[CompleteKaryTree[20, 4],
          {{1, 1}, {3, 4}, {3, 2}, {3, 4}, {5, 4}}]]
```



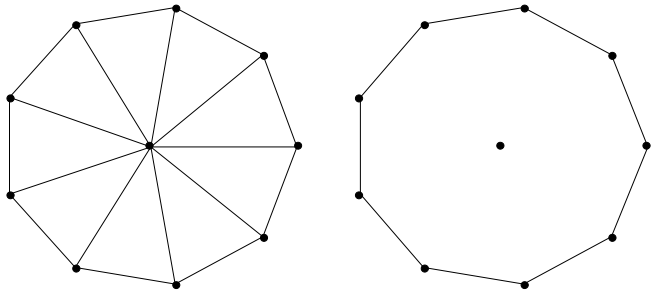

Выделим параллельные ребра {3,4} пунктиром, а петлю толстой линией:

```
In[5]:= ShowLabeledGraph[AddEdges[CompleteKaryTree[20, 4],
          {{{1, 1}, EdgeStyle → Thick}, {{3, 4}, EdgeStyle → NormalDashed},
           {{3, 4}, EdgeStyle → NormalDashed}, {{3, 2}}, {{5, 4}}}]]
```



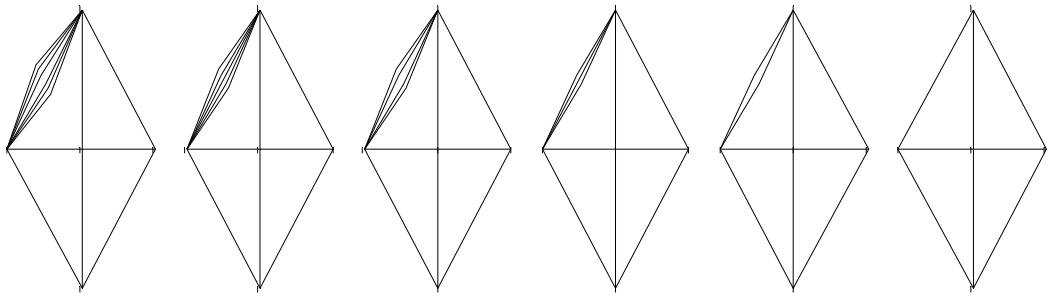

Функция DeleteEdges [g, edgeList] удаляет ребра списка edgeList из графа g. Если g неориентированный граф, то ребра из edgeList трактуются как неориентированные, в противном случае они считаются ориентированными. Если ребра параллельные, то удаляется только одно ребро. Если нужно удалить все ребра, то нужно применить тэг All. При удалении одного ребра список удаляемых ребер имеет вид {s, t} и {{s1, t1}, {s2, t2}, ...} – в противном случае. Кроме того, если удаляется одно ребро, то можно применить функцию DeleteEdge[g, e].
Удалим из десятивершинного колеса спицы:

```
In[6]:= ShowGraphArray[{g = Wheel[10], DeleteEdges[g, Edges[Star[10]]]}]
```

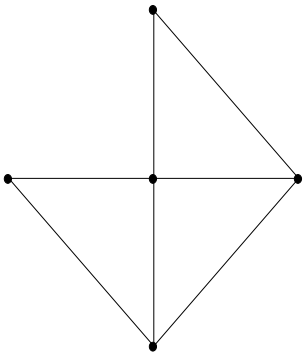


Рассмотрим граф с пятью параллельными ребрами. Использование DeleteEdges без тег All удаляет только одно из пяти ребер.

```
In[7]:= g = AddEdges[Wheel[5], {{1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}}];
        ShowGraphArray[NestList[(DeleteEdges[#, {{1, 2}}]) &, g, 5]]
```

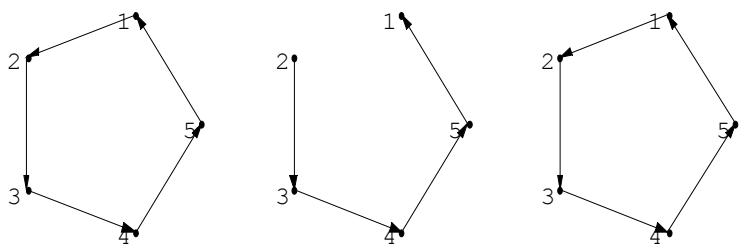


```
In[9]:= ShowGraph[DeleteEdges[g, {1, 2}, All]]
```



Рассмотрим ориентированное колесо. Удаление ориентированного ребра {1,2}, который содержится в g, удаляет его из графа, а удаление ребра {2,1} не изменяет граф.

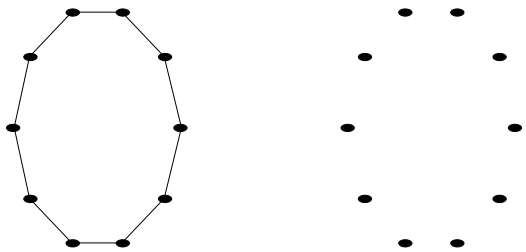
```
In[10]:= g = Cycle[5, Type → Directed]; h1 = DeleteEdges[g, {1, 2}]; h2 = DeleteEdges[g, {2, 1}];
         ShowGraphArray[{g, h1, h2}, VertexNumber → True, PlotRange → 0.25]
```



AddVertices[g,n] прибавляет n несвязанных вершин к графу g. AddVertices[g, vList] прибавляет к графу g вершины из списка vList.

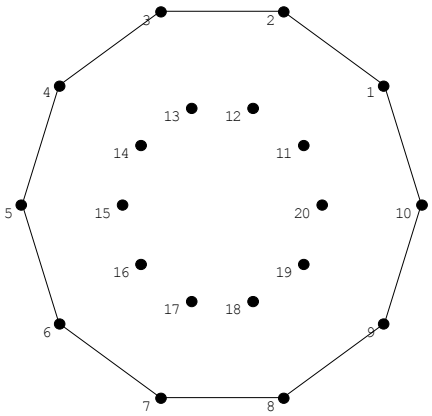
Прибавим к циклу десять вершин. Так как расположение вершин не указывается, то прибавленные точки расположены справа от цикла.

```
In[11]:= ShowGraph[AddVertices[Cycle[10], 10]]
```



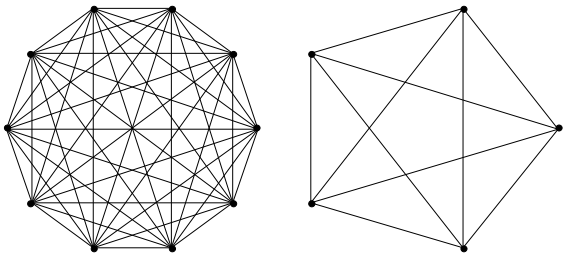
Теперь прибавим к циклу, располагая их внутри цикла.

```
In[12]:= g = Cycle[10]; ShowLabeledGraph[AddVertices[g, 0.5 * Vertices[g]]]
```




DeleteVertices[g, vList] удаляет вершины списка vList из графа g. DeleteVertex[g, v] удаляет одну вершину с номером v из графа g.
Вместе с вершинами удаляются инцидентные им ребра.

```
In[13]:= ShowGraphArray[{CompleteGraph[10], DeleteVertices[CompleteGraph[10], {1, 3, 5, 7, 9}]}]
```

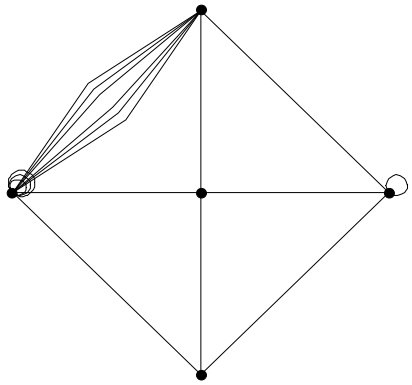


Граф, полученный при удалении множества вершин v, можно получить, порождая граф из дополнения v.

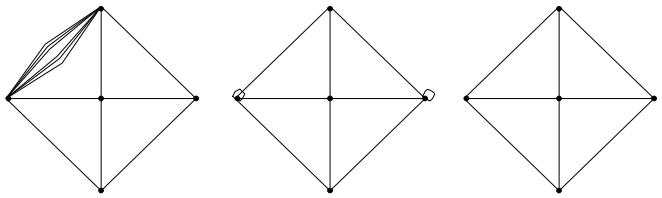
```
In[14]:= IsomorphicQ[DeleteVertices[g = RandomGraph[10, 0.7], s = RandomSubset[10]],
          InduceSubgraph[g, Complement[Range[10], s]]]
Out[14]= True
```

Если нам необходимо удалить петли, параллельные ребра или сделать граф простым, то можно воспользоваться функциями RemoveSelfLoops, RemoveMultipleEdges или MakeSimple соответственно.
RemoveSelfLoops[g] – удаляет все петли в графе g. RemoveMultipleEdges[g] – удаляет все параллельные ребра в графе g. MakeSimple[g] возвращает простой граф, удаляя из g все петли и все параллельные ребра.

```
In[15]:= ShowGraph[
          g = AddEdges[Wheel[5], {{1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {2, 2}, {2, 2},
             {2, 2}, {4, 4}}]];
```



```
In[16]:= u = RemoveSelfLoops[g]; v = RemoveMultipleEdges[g]; w = MakeSimple[g];
         ShowGraphArray[{u, v, w}]
```



Протестируем графы u,v,w:

```
In[18]:= {Map[SelfLoopsQ, {u, v, w}], Map[MultipleEdgesQ, {u, v, w}], Map[SimpleQ, {u, v, w}]} //
         ColumnForm
Out[18]= {False, True, False}
         {True, False, False}
         {False, False, True}
```

Кроме того, удалить петли и параллельные ребра можно, установив в функциях, которые модифицируют граф, опцию Type, которая может принимать значения Simple, NoMultipleEdges, NoSelfLoops.

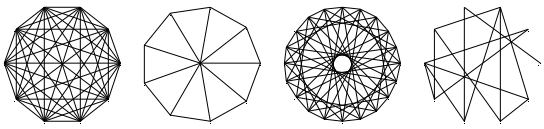
| | |
|---|---|
| ChangeVertices[g, v] | заменяет вершины графа g на вершины из списка v. Каждая вершина может быть специфицирована своими координатами {x,y} или {{x,y},options}, где options – графическая информация об этих вершинах |
| ChangeEdges[g, e] | заменяет ребра графа g ребрами из списка e. Список e может иметь форму {{s1, t1}, {s2, t2}, ...} или { {{s1, t1}, gr1}, {{s2, t2}, gr2}, ...}, где {s1, t1}, {s2, t2}, ...- концевые точки ребер, а gr1, gr2, ... графическая информация об этих ребрах |

| | |
|---|---|
| AddEdges[g, edgeList] | возвращает граф g с прибавленными новыми ребрами из списка edgeList, который имеет вид {a, b}, если прибавляется одно ребро, или вид {{a, b}, {c, d}, ...},если прибавляются несколько ребер. Список edgeList может также иметь вид {{{a, b}, x}, {{c, d}, y}, ...}, где x,y – информация об изображении прибавленных ребер |
| AddEdge[g, e] | прибавляет одно ребро e к графу g, где e имеет вид {a, b} или {{a, b}, options} |
| DeleteEdges [g, edgeList] | удаляет ребра списка edgeList из графа g. Если g неориентированный граф, то ребра из edgeList трактуются как неориентированные, в противном случае они считаются ориентированными. Если ребра параллельные, то удаляется только одно ребро. Если нужно удалить все ребра, то нужно применить тег All. При удалении одного ребра список удаляемых ребер имеет вид {s, t} и {{s1, t1}, {s2, t2}, ...} в противном случае |
| DeleteEdge[g, e] | удаляет ребро e из графа g |
| AddVertices[g,n] | прибавляет n несвязанных вершин к графу g |
| AddVertices[g, vList] | прибавляет к графу g вершины из списка vList. Список vList может иметь вид {x, y} или{{x1, y1}, {x2, y2}...} или {{{x1, y1}, g1}, {{x2, y2}, g2},...}, где {x, y}, {x1, y1} и {x2, y2} – координаты точек, а g1,g2 – графическая информация об этих вершинах. Если не указывать месторасположение вершин, то создается пустой граф и возвращается их объединение |
| AddVertex[g] | прибавляет одну несвязанную вершину к графу g |
| AddVertex[g, v] | прибавляет одну несвязанную вершину с координатами v графу g |
| DeleteVertices[g, vList] | удаляет вершины списка vList из графа g. Список vList имеет вид i,j,…..}, где i, j,….- номера вершин |
| DeleteVertex[g, v] | удаляет одну вершину с номером v из графа g |
| RemoveSelfLoops | возвращает граф, полученный удалением петель графа g |
| RemoveMultipleEdges[g] | удаляет все параллельные ребра в графе g |
| MakeSimple[g] | возвращает простой граф g |

### *5.6.2. Основные вложения графов*

В этой части рассмотрим основные вложения графов в Combinatorica. Эти вложения позволяют строить достаточно привлекательные изображения графов.
Циркулярное вложение – это равномерное расположение вершин графа на окружности единичного радиуса. Циркулярное вложение графа g конструируется с помощью функции CircularEmbedding[g].
Важное свойство циркулярного вложения: никакие три вершины не лежат на одной прямой, поэтому каждое ребро представлено однозначно. Циркулярное вложение естественно для графов-циркулянтов, полных графов и циклов.

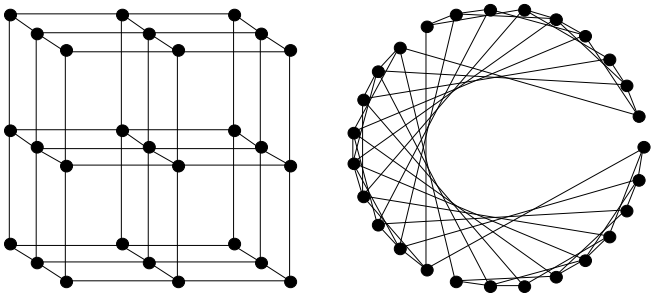
```
In[2]:= ShowGraphArray[{CompleteGraph[10], Wheel[10], CirculantGraph[20, {1, 5, 9}],
          RandomGraph[10, 0.3]}]
```



```
Out[2]= - GraphicsArray -
```

Заменим стандартное вложение 3×3×3 – решетчатого графа на циркулярное вложение. Благодаря симметрии графа циркулярное вложение остается достаточно привлекательным.

```
In[3]:= ShowGraphArray[{g = GridGraph[3, 3, 3], ChangeVertices[g, CircularEmbedding[27]]},
        VertexStyle → Disk[0.04]]
```



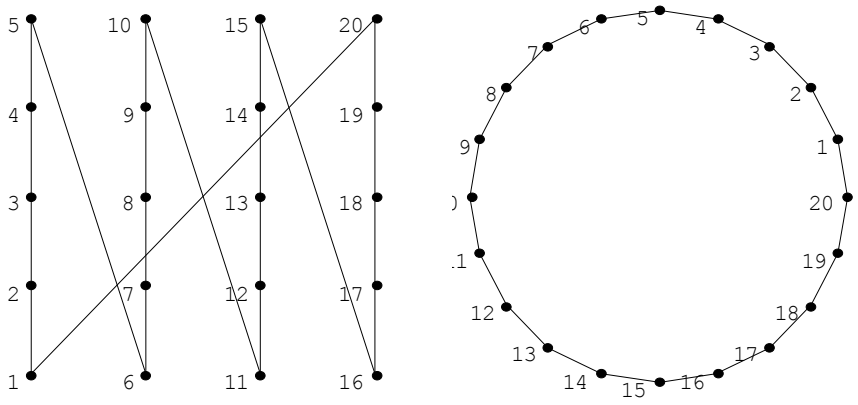
По умолчанию вложение случайных графов – циркулярное, т.к. оно однозначно представляет каждое ребро. Рассмотрим пять полуслучайных 3-регулярных графа.

```
In[4]:= ShowGraphArray[Table[RegularGraph[3, 12], {5}]]
```

```
Out[4]= - GraphicsArray -
```

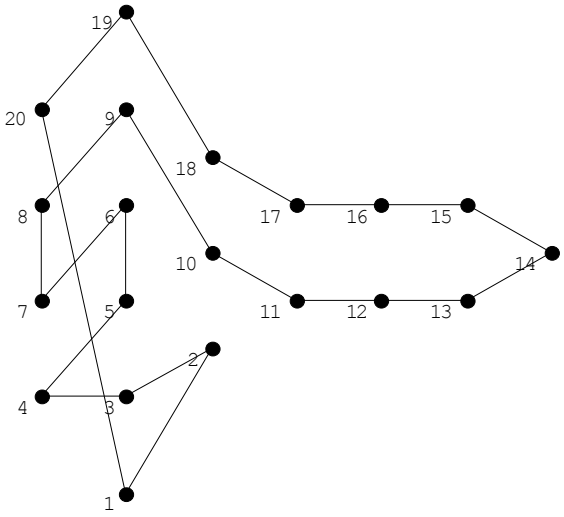
При конструировании графов Combinatorica использует именно циркулярное вложение.
При ранжированном вложении вершины равномерно располагаются на вертикальных прямых.
RankedEmbedding[l] берет как ввод разбиение l списка вершин {1,2,….,n} и возвращает такое вложение вершин, при котором вершины каждого блока появляются на вертикальной прямой: вершины первого блока на левой вертикальной линии, вершины второго блока на следующей линии и т.д.
RankedEmbedding[g, l] берет граф g и разбиение вершин графа l и возвращает граф g с вершинами, вложенными согласно RankedEmbedding[l].
RankedEmbedding[g, s] берет граф g и подмножество s вершин графа и возвращает ранжированное вложение графа g, в котором вершины множества s - в первом блоке, вершины, находящиеся на расстоянии 1 от любой вершины блока 1, - во втором блоке и т.д.
Покажем, например, ранжированное вложение цикла на десяти вершинах:

```
In[5]:= ShowGraphArray[{RankedEmbedding[Cycle[20], Partition[Range[20], 5]], Cycle[20]},
        VertexNumber → True]
```



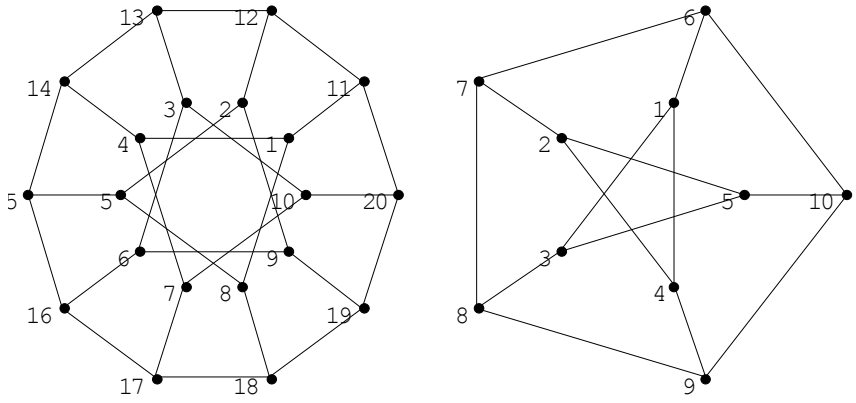
А теперь расположим вершины {7, 8, 4, 20} на первой вертикали. При этом вершины, отстоящие на расстоянии 1 от любой вершины из {7,8,4,20}, будут находиться на второй вертикали и т.д.

```
In[6]:= ShowLabeledGraph[RankedEmbedding[Cycle[20], {7, 8, 4, 20}]]
```




Рассмотрим обобщенный граф Петерсена, который вызывается с помощью функции GeneralizedPetersenGraph[n, k], где n, k – целые числа, такие, что n > 1 и k > 0. Вершины этого графа состоят из двух подмножеств $\{u_1,u_2,\ldots u_n\}$ и $\{v_1,v_2,\ldots,v_n\}$, а ребрами соединены следующие пары вершин: $\{u_i,u_{i+1}\},\{v_i,v_{i+k}\}$ и $\{u_i,v_i\}$. При n = 5 и k = 2 – это обычный граф Петерсена.

```
In[7]:= ShowGraphArray[{g = GeneralizedPetersenGraph[10, 3], PetersenGraph},
        VertexNumber → True]
```



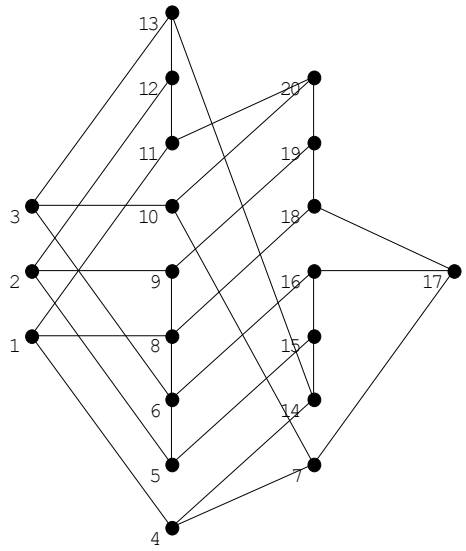

Функция RankGraph[g, l] разбивает вершины графа g на классы, элементы каждого класса находятся на одном и том же расстоянии от вершин списка l, причем каждая вершина в разбиении представлена соответствующим расстоянием.
Рассмотрим разбиение обобщенного графа Петерсена, базирующееся на расстояниях от подмножества вершин {1,2,3}.

```
In[8]:= l = RankGraph[g, {1, 2, 3}]
Out[8]= {1, 1, 1, 2, 2, 2, 3, 2, 2, 2, 2, 2, 2, 3, 3, 3, 4, 3, 3, 3}
```

Теперь разобьем множество вершин на соответствующие группы. Это дает нам вложение, в котором вершины 1, 2, 3 появляются на левой вертикальной линии, а другие вершины в графе по-

являются на различных вертикальных прямых, в зависимости от того, как далеки они от первых трех вершин.

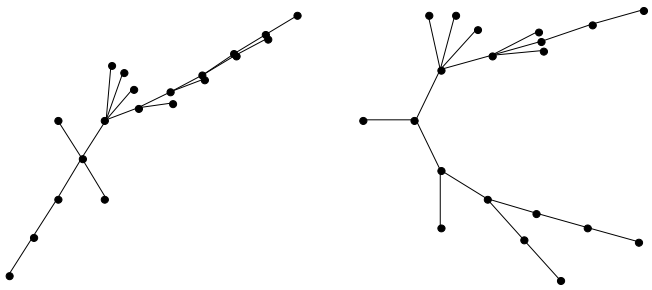
```
In[9]:= l = Table[Flatten[Position[l, i]], {i, Max[l]}]
Out[9]= {{1, 2, 3}, {4, 5, 6, 8, 9, 10, 11, 12, 13}, {7, 14, 15, 16, 18, 19, 20}, {17}}
```




RadialEmbedding[g, v] – конструирует радиальное вложение графа g, при котором вершины располагаются на концентрических окружностях с центром в точке v, а радиусы окружностей есть расстояния до точки v. RadialEmbedding[g] конструирует радиальное вложение, где v –центр графа g.

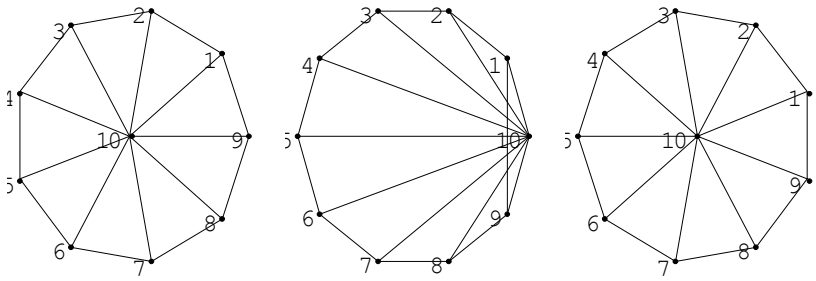
Функция RandomTree использует радиальное вложение с произвольной вершиной v. Рассмотрим радиальное вложение 20-вершинного дерева, центрированного в произвольной вершине и в центре дерева.

```
In[10]:= ShowGraphArray[{g = RandomTree[20], RadialEmbedding[g, First[GraphCenter[g]]]}]
```



Функция FromAdjacencyLists и ToAdjacencyLists взаимнообратны, поэтому их суперпозиция есть тождественный оператор. Однако информация о первоначальном вложении теряется и заменяется по умолчанию на циркулярное вложение. Применяя радиальное вложение, получим исходный граф.

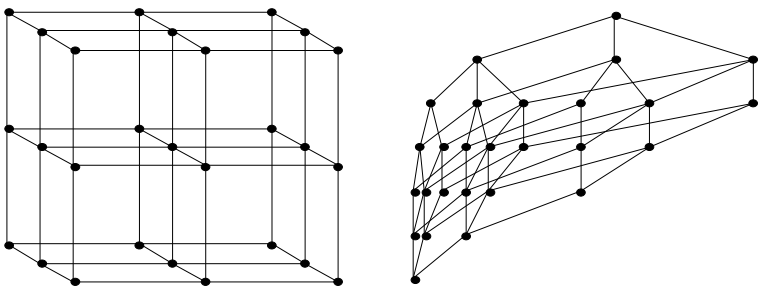
```
In[11]:= g = Wheel[10]; h = FromAdjacencyLists [ToAdjacencyLists[g]];
         ShowGraphArray[{g, h, RadialEmbedding[h, 10]}, VertexNumber → True]
```




RootedEmbedding[g, v] конструирует корневое вложение графа g с корнем в вершине v. RootedEmbedding[g] – возвращает корневое вложение графа g с корнем в центре графа.
Корневое вложение полезно применять для графов, которые имеют иерархию. Одна вершина выбрана как специальная вершина – корень, в то время как оставшиеся вершины ранжируются в зависимости от их расстояний от корня. Вершины распределены на параллельных прямых с вершинами одного ранга, расположенных на одной линии.

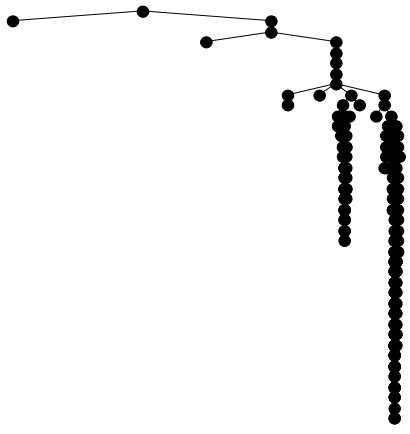
Рассмотрим корневое вложение решетчатого 2x2x2 графа:

```
In[13]:= ShowGraphArray[{GridGraph[3, 3, 3], RootedEmbedding[GridGraph[3, 3, 3], 1]}]
```



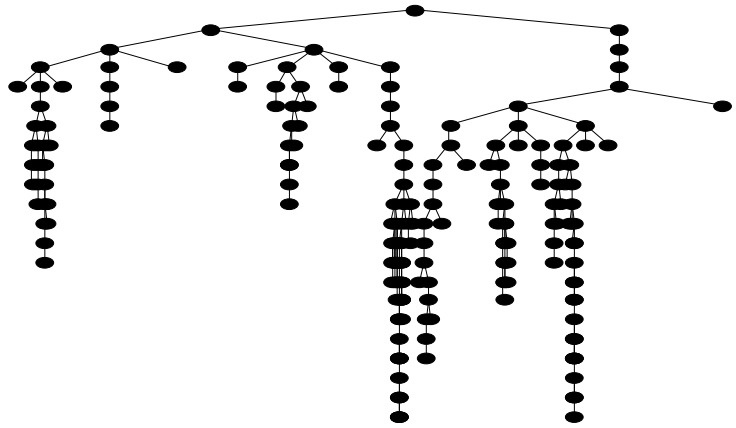
Рассмотрим корневое вложение 200-вершинного дерева с корнем в произвольной вершине. Видим, что поддеревья очень перегружены.

```
In[14]:= ShowGraph[RootedEmbedding[t = RandomTree[200], 1]]
```



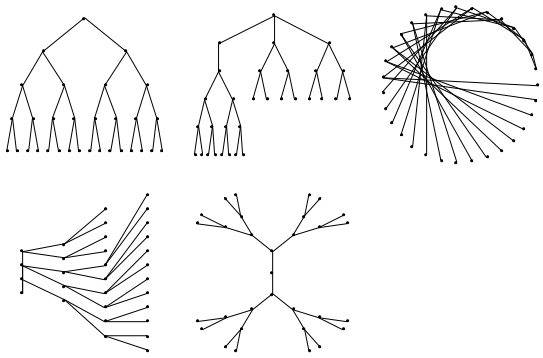
Теперь возьмем корень в центре графа и изображение его становится более привлекательным

```
In[15]:= c = First[GraphCenter[t]]; ShowGraph[RootedEmbedding[t, c]]
```



Рассмотрим теперь различные вложения одного и того же графа:

```
In[16]:= g = CompleteBinaryTree[31];
         ShowGraphArray[{{g, RootedEmbedding[g, 2], CircularEmbedding[g]},
           {RankedEmbedding[g, {1, 2, 3, 4}], RadialEmbedding[g]}}, GraphSpacing → 0.5]
```



```
Out[16]= - GraphicsArray -
```

| | |
|---|---|
| CircularEmbedding[n] | конструирует список точек, равномерно распределенных на окружности |
| CircularEmbedding[g] | равномерно распределяет вершины графа g на окружности |
| RankedEmbedding[l] | берет как ввод разбиение l списка вершин {1,2,….,n} и возвращает такое вложение вершин, при котором вершины каждого блока появляются на вертикальной прямой: вершины первого блока на левой вертикальной линии, вершины второго блока на следующей линии и т. д. |
| RankedEmbedding[g, l] | берет граф g и разбиение вершин графа l и возвращает граф g с вершинами, вложенными согласно RankedEmbedding[l] |
| RankedEmbedding[g, s] | берет граф g и подмножество s вершин графа и возвращает ранжированное вложение графа g, в котором вершины множества s - в первом блоке, вершины, находящиеся на расстоянии 1 от любой вершины блока 1, - во втором блоке и т.д. |
| RadialEmbedding[g, v] | конструирует радиальное вложение графа g, при котором вершины располагаются на концентрических окружностях с центром в точке v, а радиусы окружностей есть расстояния до точки v |
| RadialEmbedding[g] | конструирует радиальное вложение, где v –центр графа g |
| RootedEmbedding[g, v] | конструирует корневое вложение графа g с корнем в вершине v |
| RootedEmbedding[g] | возвращает корневое вложение графа g с корнем в центре графа |
| RankGraph[g, l] | разбивает вершины графа g на классы, элементы каждого класса |

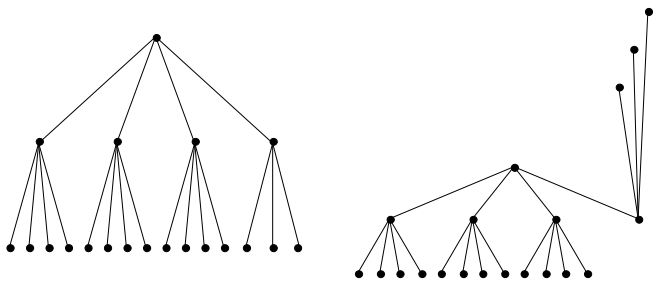
| | |
|---|---|
| | находятся на одном и том же расстоянии от вершин списка l, причем каждая вершина в разбиении представлена соответствующим расстоянием |
| GeneralizedPetersenGraph[n, k] | возвращает обобщенный граф Петерсена, где n, k – целые числа, такие, что n > 1 и k > 0. Вершины этого графа состоят из двух подмножеств $\{u_1,u_2,\ldots u_n\}$ и $\{v_1,v_2,\ldots,v_n\}$, а ребрами соединены следующие пары вершин: $\{u_i,u_{i+1}\},\{v_i,v_{i+k}\}$ и $\{u_i,v_i\}$.При n = 5 и k = 2 – это обычный граф Петерсена |

### *5.6.3. Улучшение вложения графов*

Для изображения произвольных графов часто возникает необходимость улучшения качества вложения или модификации вышеперечисленных стандартных вложений по различным критериям.

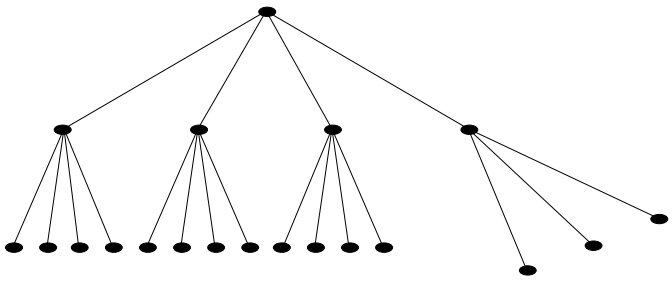
Функция RotateVertices[v, theta] поворачивает каждую вершину из списка v на theta радиан относительно начала координат. RotateVertices[g, theta] поворачивает граф g на theta радиан относительно начала координат.

```
In[2]:= ShowGraphArray[{g = CompleteKaryTree[20, 4], h = RotateVertices[g, {18, 19, 20}, π / 3]}]
```



Функция DilateVertices[v, d] умножает каждую координату каждой вершины из списка v на множитель d. DilateVertices[g, d] умножает каждую координату каждой вершины графа на множитель d. DilateVertices просто растягивает или сжимает либо отдельные вершины, либо все вложение.

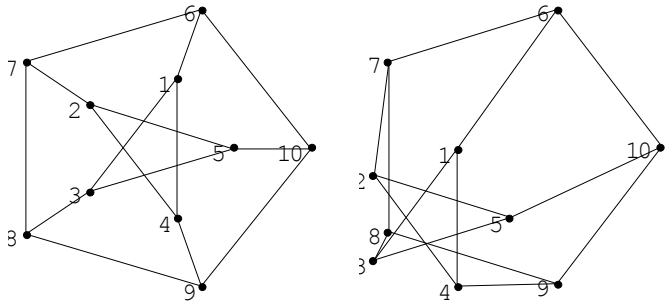
```
In[3]:= ShowGraphArray[{g = CompleteKaryTree[20, 4], DilateVertices[h, 4]}]
```



TranslateVertices[ v, {x, y}] производит параллельный перенос на вектор {x, y} вершин из списка v.

TranslateVertices[g, {x, y}] осуществляет параллельный перенос графа g на вектор {x, y}.

```
In[4]:= ShowGraphArray[{g = PetersenGraph, TranslateVertices[g, {6, 7, 8, 9, 10}, {1, 1}]},
        VertexNumber → True]
```

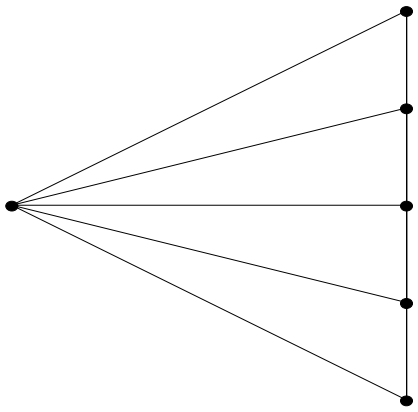



Графическая информация, ассоциированная с вершинами, остается инвариантной относительно поворотов, параллельных переносов и растяжений графа. Это верно также для функций усовершенствования вложений графов – ShakeGraph и SpringEmbedding.

ShakeGraph[g, d] совершает случайные перемещения вершин графа g, передвигая каждую вершину по крайней мере на расстояние d от ее первоначальной позиции. Случайные перемещения вершин помогают так расположить вершины, что никакие три вершины не лежат на одной прямой, и, таким образом, все ребра будут однозначно представлены.

Рассмотрим ранжированное вложение полного графа на 6 вершинах:

```
In[5]:= ShowGraph[g = RankedEmbedding[CompleteGraph[6], {1}]]
```

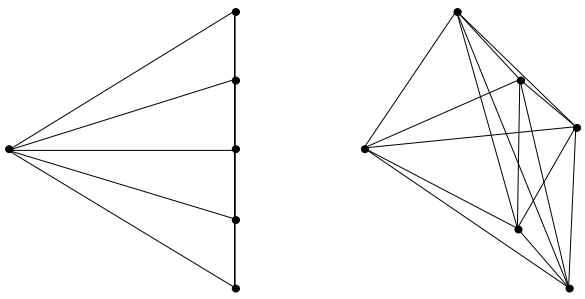


Вычислим число ребер графа g:

```
In[6]:= M[g]

Out[6]= 15
```

На рисунке некоторые тройки вершин находятся на одной прямой, поэтому некоторые ребра представлены неоднозначно. Применим функцию ShakeGraph[g, 0.6]:

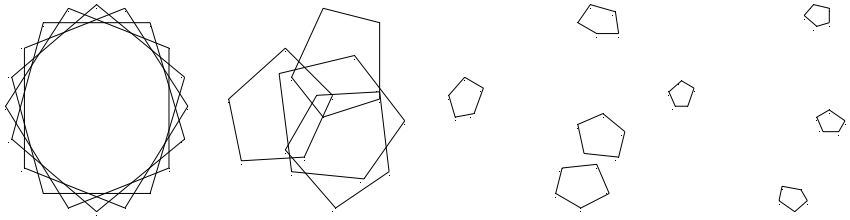
```
In[7]:= ShowGraphArray[{g, ShakeGraph[g, 0.6]}]
```



Очень интересным улучшением вложения является так называемая «прыжковая» модель. SpringEmbedding[g] изменяет вложение графа g, моделируя систему прыжков. SpringEmbedding

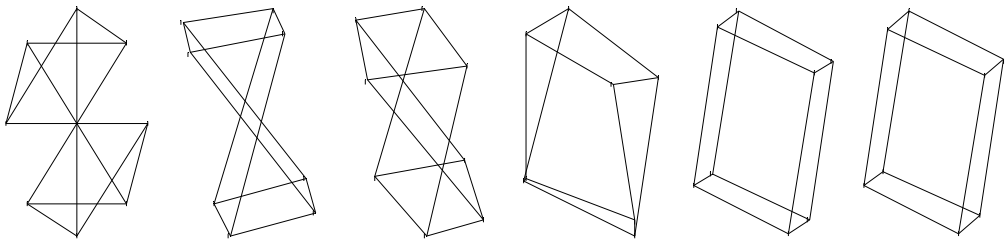
располагает вершины графа на плоскости так, чтобы привести в равновесие систему связанных по ребрам законом Гука отрицательных электрических зарядов, расположенных в вершинах.
SpringEmbedding заставляет прыгать вершины, причем смежные вершины притягиваются, а несмежные отталкиваются. Практически граф рассматривается как система тел, с силами, действующими на них. Алгоритм ищет конфигурацию тел с локальным минимумом энергии. Он заставляет ребра быть короткими насколько возможно, а вершины расположиться обособленно в максимально возможной степени.
Связные компоненты графа – это максимальные подграфы, которые являются связными. Несмежные вершины отталкиваются друг от друга, так, что связные компоненты имеют тенденцию расходиться одна от другой. Рассмотрим CirculantGraph[20,4]. С каждым применением SpringEmbedding 4 связные компоненты расходятся дальше, позволяя нам ясно их разглядеть.

```
In[8]:= ShowGraphArray[{NestList[SpringEmbedding, CirculantGraph[20, 4], 3]}]
```



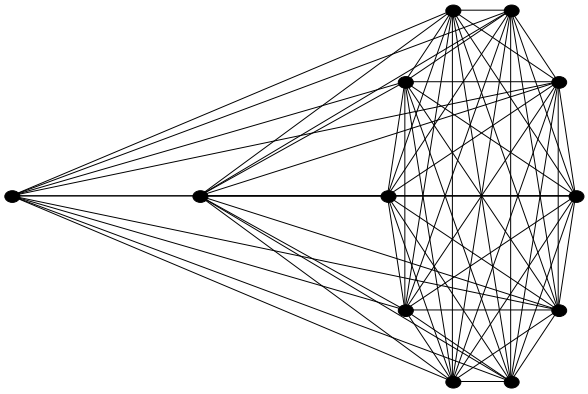
Рассмотрим циркулярное вложение 2×2×2 - решетчатого графа. Из циркулярного вложения не видно решетчатой структуры графа. Но после нескольких применений SpringEmbedding структура графа становится ясной!

```
In[10]:= ShowGraphArray[gt = NestList[SpringEmbedding, h, 5]]
```



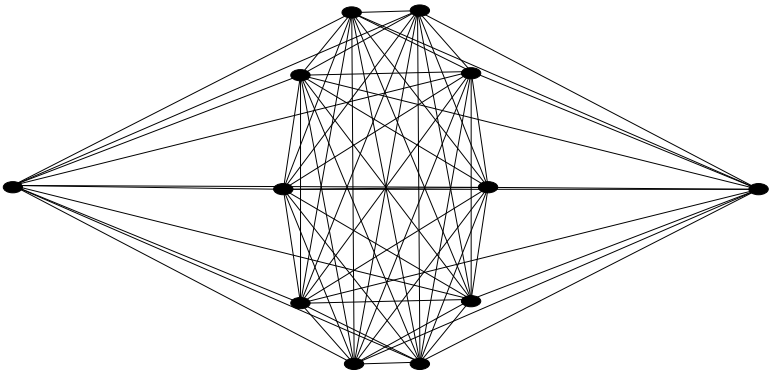
SpringEmbeddings хорошо работает на редких графах. Свяжем каждую вершину полного десятивершинного графа с каждой из двух несвязанных вершин с помощью GraphJoin. GraphJoin[g1, g2, ...] – возвращает граф, полученный прибавлением всех возможных ребер между различными графами в объединении графов g1,g2.

```
In[11]:= ShowGraph[GraphJoin[EmptyGraph[2], CompleteGraph[10]]]
```



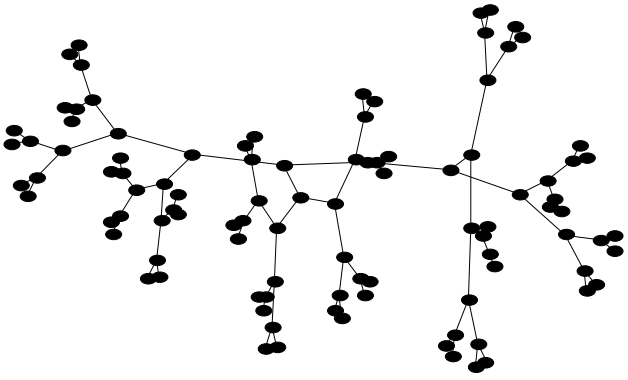
Применим SpringEmbedding к полученному графу. Так как достигается конфигурация минимальной энергии, эти две вершины пустого графа оказываются конечными точками по разные стороны от полного графа.

```
In[12]:= ShowGraph[SpringEmbedding[GraphJoin[EmptyGraph[2], CompleteGraph[10]]]]
```



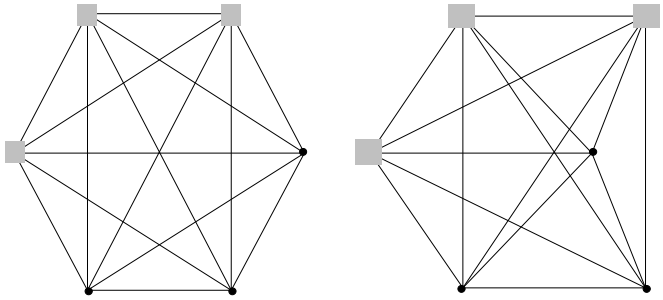
Опции, контролирующие число итераций и размер шага вершин, улучшают алгоритм.

```
In[13]:= ShowGraph[SpringEmbedding[CompleteBinaryTree[100], 200, 0.05]]
```



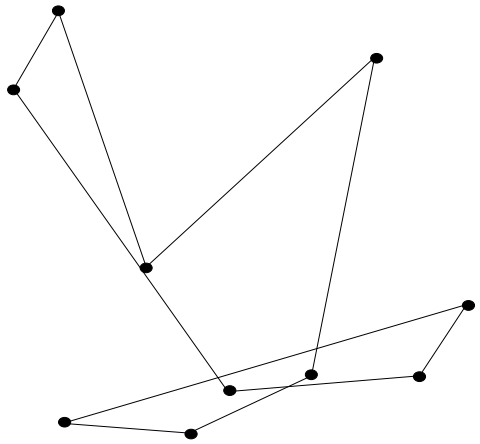
Графическая информация, ассоциированная с вершинами, остается инвариантной относительно поворотов, параллельных переносов и растяжений графа. Это также верно для функций совершенствования вложений графов ShakeGraph и SpringEmbedding.

```
In[14]:= ShowGraphArray[
          {g = SetGraphOptions[CompleteGraph[6],
             {{1, 2, 3, VertexStyle → Box[Large], VertexColor → Gray}}],
           TranslateVertices[g, Range[5], {0.8, 0}]}]
```



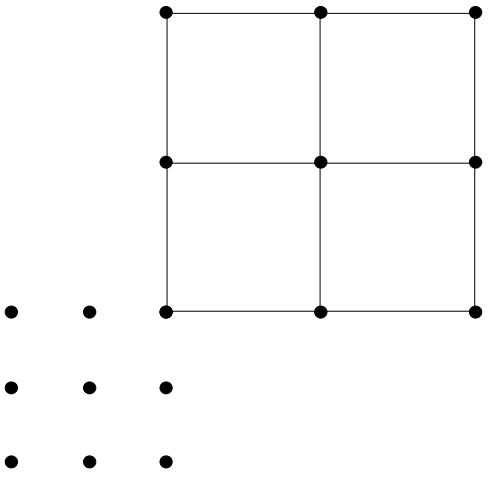
Если мы хотим получить случайное вложение графа g, то нужно применить функцию RandomVertices[g], которая возвращает случайное вложение графа g.

```
In[15]:= ShowGraph[RandomVertices[Cycle[10]]]
```



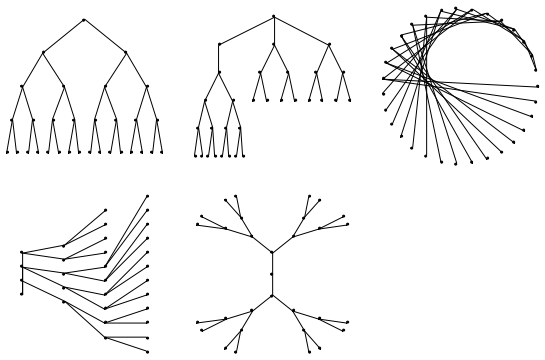
Встроенная функция NormalizeVertices[v] возвращает список вершин с тем же вложением, что и v, но координаты всех вершин нормированы

```
In[16]:= ShowGraph[AddVertices[ GridGraph[3, 3],
          NormalizeVertices[Vertices[GridGraph[3, 3]]]]]
```



Рассмотрим различные измененные вложения одного и того же графа:

```
In[17]:= g = CompleteBinaryTree[31];
         ShowGraphArray[{{g, RootedEmbedding[g, 2], CircularEmbedding[g]},
           {RankedEmbedding[g, {1, 2, 3, 4}], RadialEmbedding[g]}}, GraphSpacing→ 0.5]
```



```
Out[17]= - GraphicsArray -
```

| | |
|---|---|
| RotateVertices[v, theta] | поворачивает каждую вершину из списка v на theta радиан относительно начала координат |
| RotateVertices[g, theta] | поворачивает граф g на theta радиан относительно начала координат |
| DilateVertices[g, v, d] | умножает каждую координату каждой вершины из списка v на множитель d |
| DilateVertices[g, d] | умножает каждую координату каждой вершины графа на множитель d |
| TranslateVertices[ v, {x, y}] | производит параллельный перенос на вектор {x, y} вершин из списка v |
| TranslateVertices[g, {x, y}] | производит параллельный перенос на вектор {x, y} вершин графа g |
| ShakeGraph[g, d] | совершает случайные перемещения вершин графа g, передвигая каждую вершину по крайней мере на расстояние d от ее первоначальной позиции |
| SpringEmbedding[g] | изменяет вложение графа g, моделируя систему прыжков |
| SpringEmbedding[g, step, increment] | используется для усовершенствования алгоритма. Значение step указывает, сколько итераций пробегает алгоритм, а increment указывает расстояние, на которое передвигаются вершины при каждом шаге. По умолчанию значения step и increment равны 10 и 0.15 соответственно |
| RandomVertices[g] | возвращает случайное вложение графа g |
| NormalizeVertices[v] | возвращает список вершин с тем же вложением, что и v, но координаты всех вершин нормированы |

## ■5.7. Операции над графами

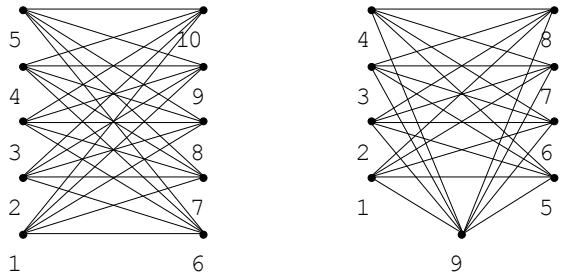
В этом разделе мы рассмотрим некоторые операции над графами.

### *5.7.1. Стягивание вершин*

Под стягиванием мы подразумеваем операцию удаления ребра e и отождествление его концевых вершин. Операция стягивания выполняется с помощью функции Contract. Contract[g, {x, y}] – возвращает граф, полученный стягиванием пары вершин {x, y} графа g.

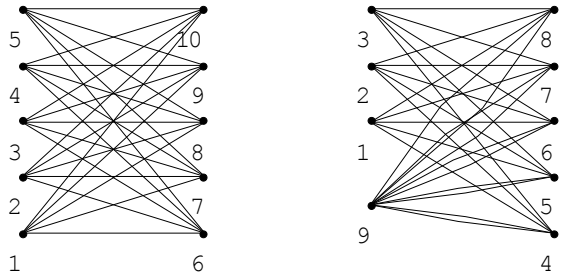
В этом примере стягивание ребра (1,6) уменьшает число ребер и вершин на 1, потому что стягиваемое ребро исчезает. Вершины перенумеровываются: новая вершина занумерована 9, а номера всех других вершины снижаются на 1 по порядку. Contract наследует вложение исходного графа. Новая вершина расположена посередине двух стянутых вершин и не изменяет положения всех других вершин.

```
In[3]:= ShowGraphArray[{g=CompleteGraph[5,5],Contract[g,{1,6}]},VertexN
        umber->True,VertexNumberPosition->{0,-0.1},PlotRange->0.25]
```



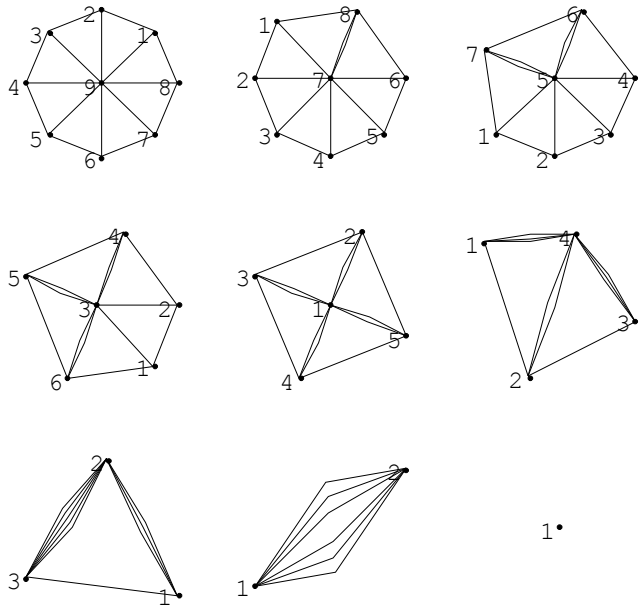

Если стягиваемые вершины смежны одной и той же вершине, то Contract создаёт параллельные ребра. Стянем в вышеприведенном графе первую и вторую вершины:

```
In[4]:= g = CompleteGraph[5, 5]; h = Contract[g, {1, 2}];
        ShowGraphArray[{g, h}, VertexNumber → True,
         VertexNumberPosition → {0, -0.1}, PlotRange → 0.25]
```




Стягивание вершин будет создавать параллельные ребра всякий раз, когда стягиваемые вершины имеют общих соседей. Таким образом, операция стягивания может уменьшить число вершин без уменьшения числа ребер. Последовательно стянем первую и вторую вершины 6- вершинного колеса

```
In[5]:= g=Wheel[9];ShowGraphArray[Partition[NestList[Contract[#,{1,2}]&
        ,g,8],3],VertexNumber->True,PlotRange→0.25]
```

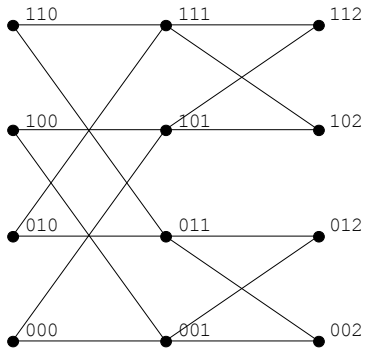


Combinatorica содержит функцию ButterflyGraph[n], очень интересный граф бабочки:
r-мерный граф бабочки имеет $(r+1)2^r$ вершин и $(r+1)2^{r+1}$ ребер. Вершины соответствуют парам (w,i), где i – уровень вершины ($0 \le i \le r$), а w – r-битовое двоичное число, которое обозначает строку вершины. Связываются ребром вершины (w,i) и ($w^*$,i+1) тогда и только тогда, когда либо

1. w и $w^*$ идентичны;
2. w и $w^*$отличаются в ровно (i+1) -м бите.

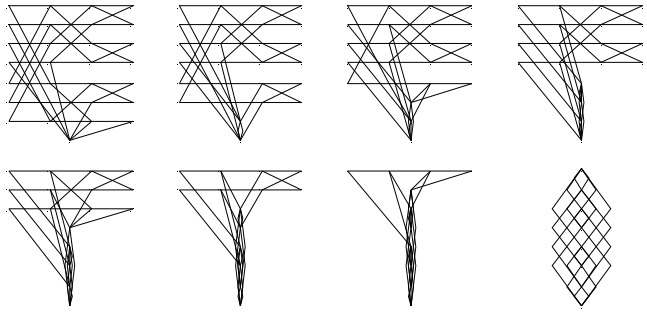
Ребра, которые связывают вершины с одинаковыми бинарными строками, называются прямыми ребрами. Оставшиеся ребра называются пересекающимися ребрами, они связывают вершины, которые отличаются в i+1-м бите и находятся в уровнях i и i+1.
Функция ButterflyGraph[n] возвращает n-мерный граф бабочки. Допускается опция VertexLabel, которая принимает значения False и True. Если установить VertexLabel-> True, то вершины помечены строками (w,i).
Приведем пример двумерного графа бабочки. Он состоит из $(2+1)2^2=12$ вершин, расположенных в трех вертикальных уровнях и $(2+1)2^{2+1}=24$:

```
In[6]:= ShowGraph[g=ButterflyGraph[2,VertexLabel->True],TextStyl
        e->{FontSize->12},PlotRange->0.2]
```




Последовательно будем стягивать вершины, находящихся в одной строке: Самая нижняя строка стянута первой, потом строка выше и т.д. Получается очень красивая картинка:

```
In[7]:= g=Contract[ButterflyGraph[3],{1,2,3,4}];ShowGraphArray[P
        artition[l=NestList[Contract[#,{1,2,3,4}]&,g,7],4]]
```



| | |
|---|---|
| Contract[g, {x, y}] | возвращает граф, полученный стягиванием пары вершин {x, y} графа g |
| ButterflyGraph[n] | возвращает n-мерный граф бабочки: ориентированный граф, чьи вершины - пары (w,i),где w-бинарная строка длины n, i - целое от 0 до n, а ребра соединяют вершины (w,i) и (w,j) и (w1, i+1),если w и w1 отличаются ровно в одном бите (разряде) |

### *5.7.2. Объединение и пересечение графов*

Графы являются множеством вершин и ребер и естественными операциями на множествах являются операции объединения и пересечения.

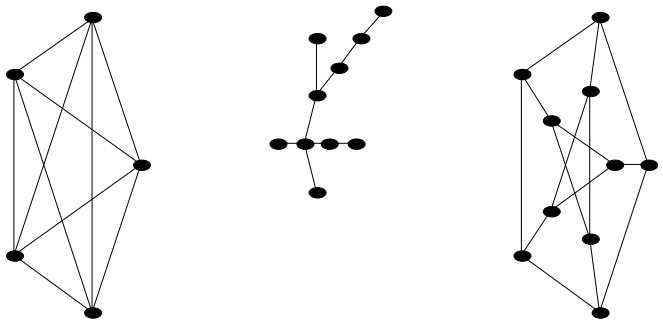
Рассмотрим графы $g_1=(E_1,V_1)$ и $g_2=(E_2,V_2)$;

**Объединением** графов $g_1=(E_1,V_1)$ и $g_2=(E_2,V_2)$ называется такой граф $g_1 \cup g_2=(E_1 \cup E_2, V_1 \cup V_2)$, что множество его вершин является объединением $V_1$ и$V_2$, а множество ребер – объединением $E_1 \cup E_2$.

Функция GraphUnion[g1, g2, ...] конструирует объединение данных графов, при этом вначале нормализуются вложения данных графов, обеспечивая одинаковый размер их изображений. Заметим, что каждый последующий граф появляется справа от предыдущего. GraphUnion[n, g] возвращает n копий данного графа g.

Покажем результат объединения полного графа, случайного дерева и графа Петерсена. Компоненты объединения расположены в порядке слева направо:

```
In[2]:= ShowGraph[g = GraphUnion[CompleteGraph[5], RandomTree[10], PetersenGraph]]
```



Объединение графов всегда несвязно. Проверим это для графа g:

```
In[3]:= ConnectedQ[g]

Out[3]= False
```

Рассмотрим связные компоненты графа g:

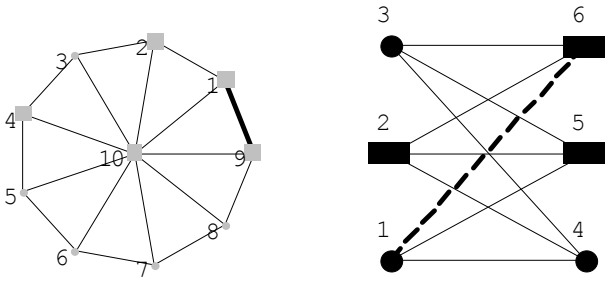
```
In[4]:= ConnectedComponents[g]
Out[4]= {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10, 11, 12, 13, 14, 15},
        {16, 17, 18, 19, 20, 21, 22, 23, 24, 25}}
```

Подсчитаем число вершин, ребер и компонент связности. Граф объединения состоит из 25 вершин, 34 ребер и имеет три связные компоненты.

```
In[5]:= {V[g], M[g], Length[ConnectedComponents[g]]}
 Out[5]= {25, 34, 3}
```

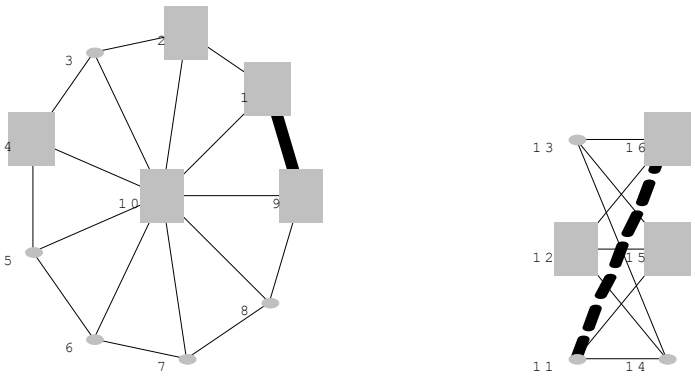
GraphUnion наследует глобальные опции из первого графа, при этом сохраняются все его локальные опции. Локальные опции сохраняются для обоих графов.

```
In[6]:= g = SetGraphOptions[CompleteGraph[3, 3],
          {{2, 5, 6, VertexStyle → Box[Large]}, {{1, 6}, EdgeStyle → ThickDashed}},
          VertexStyle → Disk[Large], VertexNumberPosition → {0, 0.1}];
        h = SetGraphOptions[Wheel[10],
          {{1, 2, 4, 9, 10, VertexStyle → Box[Large]}, {{1, 9}, EdgeStyle → Thick}},
          VertexColor → Gray];
        ShowGraphArray[{h, g}, VertexNumber → True, PlotRange → 0.25]
```




Применим функцию GraphUnion[h,g]:

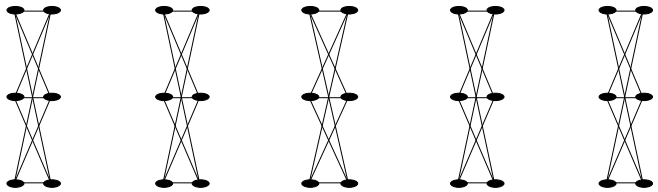
```
In[9]:= ShowLabeledGraph[GraphUnion[h, g]]
```




Первый граф в новом графе, граф h имел глобальную опцию VertexColor->Gray, поэтому все вершины в объединении – серого цвета. Кроме того, сохранены все локальные опции обоих графов. Глобальной опцией второго графа g, VertexNumberPosition->{0,0.1}, функция пренебрегает.

Для любого натурального n GraphUnion[n, g] позволяет строить n копий графа g.

```
In[10]:= ShowGraph[GraphUnion[5, CompleteGraph[3, 3]]]
```



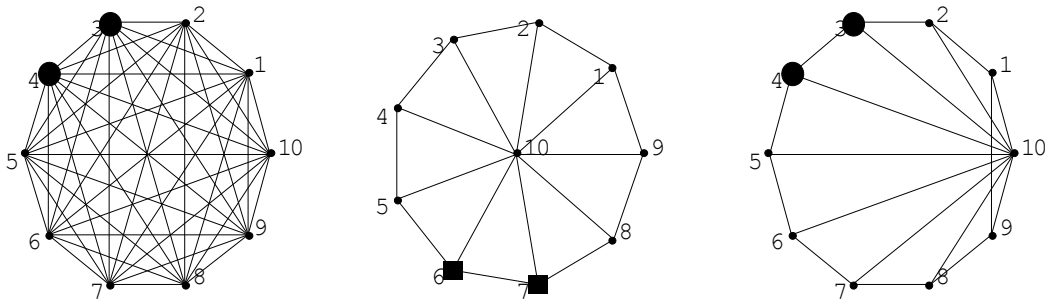
GraphUnion требует, чтобы вводимые графы были все одновременно ориентированы или неориентированы. Смешанный ввод не выдает значений.

Пересечением графов $g_1=(E_1,V_1)$ и $g_2=(E_2,V_2)$ с одинаковым числом вершин называется граф $g_3= g_1 \cap g_2=(E_1 \cap E_2, V_1 \cap V_2)$. Таким образом, множество вершин графа $g_3$ состоит только из вершин, присутствующих одновременно в $g_1$ и $g_2$, а множество ребер $g_3$ состоит только из ребер, присутствующих одновременно в $g_1$ и $g_2$.

Функция GraphIntersection[g1, g2, ...] возвращает граф – пересечение графов g1, g2, .. Подчеркнем, что графы g1, g2,… должны иметь одинаковое количество вершин.

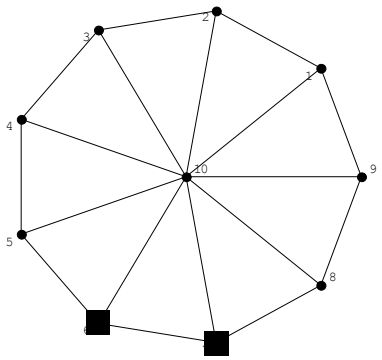
Рассмотрим пересечение десятивершинных полного графов и колеса. Очевидно, что их пересечением является колесо, но это колесо имеет непривычное вложение. Опции в пересечении графов работают точно так же, как и в случае объединения.

```
In[11]:= g = SetGraphOptions[CompleteGraph[10],
            {{1, 2, 8, 9, 10, VertexNumberPosition → UpperRight},
             {3, 4, VertexStyle → Disk[Large]}}, EdgeStyle → Thin];
          h = SetGraphOptions[Wheel[10],
            {{8, 9, 10, VertexNumberPosition → UpperRight}, {6, 7, VertexStyle → Box[Large]}}];
          ShowGraphArray[{g, h, GraphIntersection[g, h]}, VertexNumber → True,
           PlotRange → 0.25]
```



Рассмотрим GraphIntersection[h,g].

```
In[13]:= ShowLabeledGraph[GraphIntersection[h, g]]
```

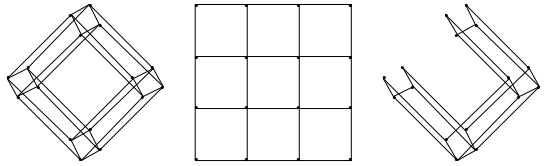


Таким образом, пересечением любого графа g с n вершинами с $K_n$ есть граф g. Действительно,

```
In[14]:= IdenticalQ[GraphIntersection[g = RandomGraph[10, 0.5], CompleteGraph[10]],
          g]

Out[14]= True
```

Что является пересечением гиперкуба и решетчатого графа одного порядка?

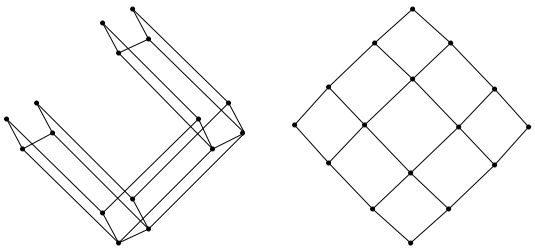
```
In[15]:= ShowGraphArray[{g = Hypercube[4], h = GridGraph[4, 4], GraphIntersection[g, h]}]
```



```
Out[15]= - GraphicsArray -
```

Применим к пересечению графов g, h функцию SpringEmbedding. Оказывается, решетчатый граф есть подграф гиперкуба.

```
In[16]:= ShowGraphArray[{gh = GraphIntersection[Hypercube[4], GridGraph[4, 4]],
          SpringEmbedding[gh, 200]}]
```

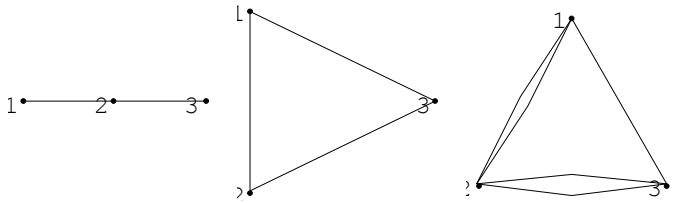


| | |
|---|---|
| GraphUnion[g1, g2, ...] | конструирует объединение данных графов, при этом вначале нормализуются вложения данных графов, обеспечивая одинаковый размер их изображений |
| GraphUnion[n, g] | возвращает n копий данного графа g |
| GraphIntersection[g1, g2, ...] | возвращает граф – пересечение графов g1, g2,… |

### *5.7.3. Сумма и разность графов*

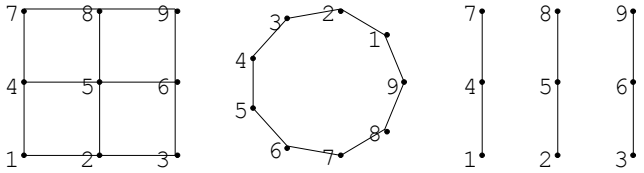
Поскольку графы могут быть представлены матрицами смежности, то они могут складываться, вычитаться и умножаться естественным образом (как матрицы), при условии, что они имеют одинаковое число вершин.
GraphSum[g1, g2, ...] – конструирует граф, список ребер которого есть конкатенация списков ребер графов g1, g2, … GraphSum[g,h] берет ребра второго графа и прибавляет их к первому графу.

```
In[2]:= ShowGraphArray[{g = Path[3], h = Cycle[3], RootedEmbedding[GraphSum[g, h], 1]},
         VertexNumber → True]
```



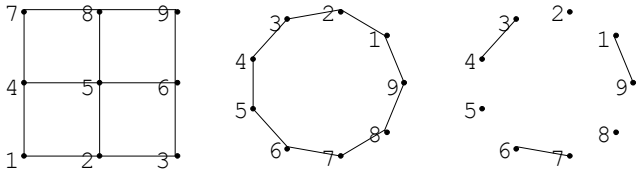
GraphDifference[g, h] конструирует граф, полученный вычитанием ребер графа h из ребер графа g.

```
In[3]:= ShowGraphArray[{g = GridGraph[3, 3], h = Cycle[9], GraphDifference[g, h]},
          VertexNumber → True, PlotRange → 0.25]
```



Теперь рассмотрим разность между h и g:

```
In[4]:= ShowGraphArray[{g = GridGraph[3, 3], h = Cycle[9], GraphDifference[h, g]},
          VertexNumber → True, PlotRange → 0.25]
```

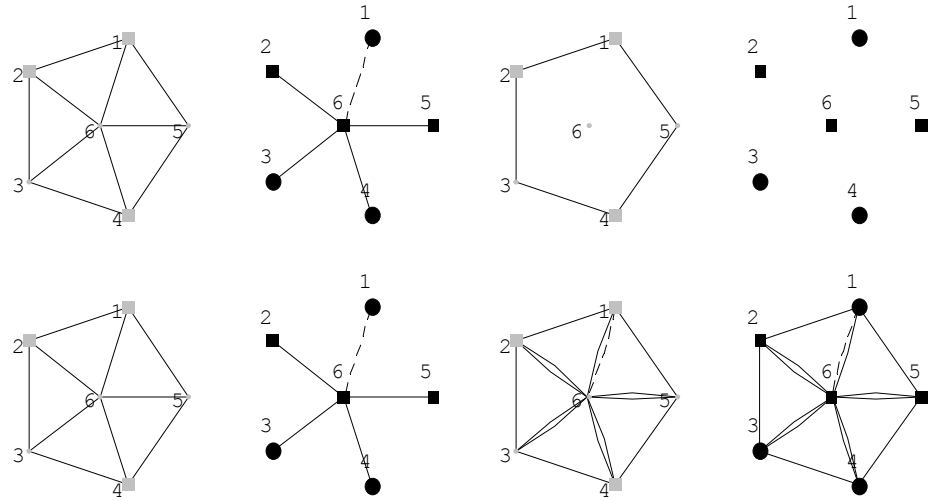


Разность графа и самого себя есть пустой граф:

```
In[5]:= EmptyQ[GraphDifference[Cycle[10], Cycle[10]]]

Out[5]= True
```

Все глобальные опции приходят из первого графа – слагаемого, каждое ребро в сумме принадлежит одному из введенных графов и сохраняет его локальные опции. Каждая вершина наследует свои опции из первого графа. GraphDifference[g, h] сохраняет все опции первого графа.

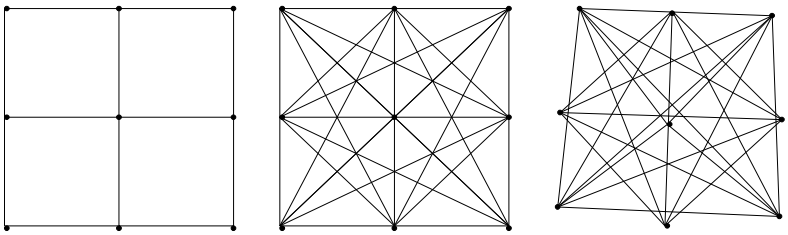
```
In[6]:= g = SetGraphOptions[Star[6],
            {{2, 5, 6, VertexStyle → Box[Large]}, {{1, 6}, EdgeStyle → ThinDashed}},
            VertexStyle → Disk[Large], VertexNumberPosition → {0, 0.1}];
          h = SetGraphOptions[Wheel[6],
            {{1, 2, 4, 9, 10, VertexStyle → Box[Large]}, {{1, 9}, EdgeStyle → Thick}},
            VertexColor → Gray];
          ShowGraphArray[{{h, g, GraphDifference[h, g], GraphDifference[g, h]},
            {h, g, GraphSum[h, g], GraphSum[g, h]}}, VertexNumber → True, PlotRange → 0.25]
```



Пусть g – граф. Граф $g_1$ называется **дополнением** графа g, если их множества вершин совпадают, а любые две несовпадающие вершины смежны в $g_1$ тогда и только тогда, когда они не смежны в g.

Функция GraphComplement[g] возвращает дополнение графа g. Рассмотрим дополнение 3×3 – решетчатого графа. На первый взгляд, кажется, что ребра исходного графа присутствуют в его дополнении, но SpringEmbedding ясно показывает его структуру.

```
In[9]:= ShowGraphArray[{p = GridGraph[3, 3], GraphComplement[p],
          GraphComplement[SpringEmbedding[p, 4]]}]
```



Дополнение графа g на n вершинах есть разность между полным графом на n вершинах и графом g.

```
In[10]:= CompleteQ[GraphSum[Wheel[10], GraphComplement[Wheel[10]]]]
Out[10]= True
```

Граф называется **самодополняемым,** если он изоморфен своему дополнению. SelfComplementaryQ[g] возвращает True, если граф является самодополняемым.

Наименьшие нетривиальные самодополняемые графы – это дорожка на четырех вершинах и цикл на пяти.

```
In[11]:= SelfComplementaryQ[Cycle[5]] && SelfComplementaryQ[Path[4]]
Out[11]= True
```

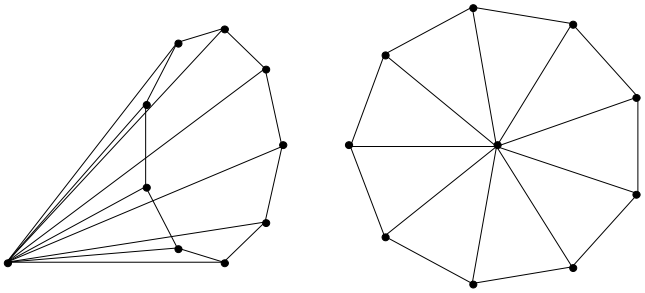
| | |
|---|---|
| GraphSum[g1, g2, ...] | конструирует граф, список ребер которого есть конкатенация списков ребер графов g1, g2, … |
| GraphSum[g,h] | берет ребра второго графа и прибавляет их к первому графу |
| GraphDifference[g, h] | конструирует граф, полученный вычитанием ребер графа h из ребер графа g |
| GraphComplement[g] | возвращает дополнение графа g |

| | |
|---|---|
| SelfComplementaryQ[g] | возвращает True, если граф является самодополняемым |

### *5.7.4. Соединение графов*

GraphJoin[g1, g2, ...] конструирует соединение графов g1, g2, ...., причем возвращается граф, полученный прибавлением всех возможных ребер между различными графами. Декартово произведение A×B - множество пар элементов, так что один элемент принадлежит A, а второй B. Ребра, которые мы прибавляем к объединению двух графов, есть точно декартово произведение двух множеств вершин. Combinatorica содержит функцию CartesianProduct[l1, l2], которая возвращает декартово произведение множеств l1 и l2.
Wheel[n] есть результат Join изолированной вершины $K_1$ и цикла на n-1 вершинах.

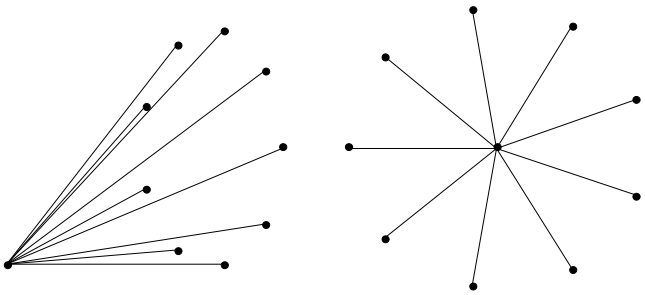
```
In[2]:= ShowGraphArray[{q = GraphJoin[EmptyGraph[1], Cycle[9]], RadialEmbedding[q]}]
```



```
In[3]:= IsomorphicQ[GraphJoin[EmptyGraph[1], Cycle[9]], Wheel[10]]
Out[3]= True
```

Звезда на n вершинах есть Join изолированной вершины $K_1$ и и пустого графа на n-1 вершинах.

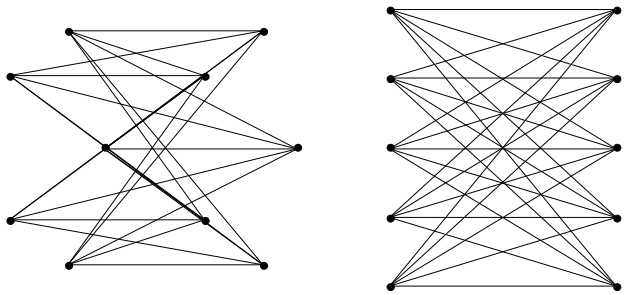
```
In[4]:= ShowGraphArray[{p = GraphJoin[EmptyGraph[1], EmptyGraph[9]], RadialEmbedding[q]}]
```



```
In[5]:= IsomorphicQ[GraphJoin[EmptyGraph[1], EmptyGraph[9]], Star[10]]
Out[5]= True
```

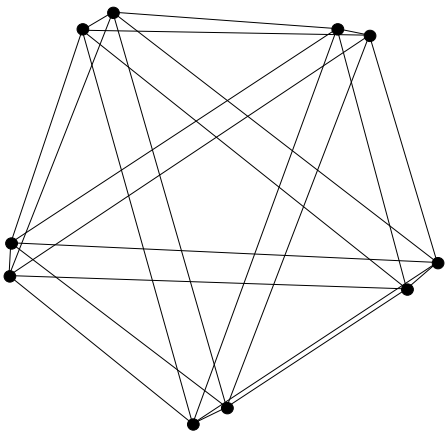
Двудольный граф на 2n вершинах есть соединение двух пустых графов на n вершинах:

```
In[6]:= ShowGraphArray[{q = GraphJoin[EmptyGraph[5], EmptyGraph[5]],
        RankedEmbedding[q, Range[5]]}]
```



Посмотрим, как выглядит SpringEmbedding левого графа на вышеприведенном рисунке:

```
In[7]:= ShowGraph[SpringEmbedding[GraphJoin[EmptyGraph[5], EmptyGraph[5]]]]
```



GraphJoin наследует опции так же, как и предыдущие операции над графами.

| | |
|---|---|
| GraphJoin[g1, g2, ...] | конструирует соединение графов g1, g2, ...., причем возвращается граф, полученный прибавлением всех возможных ребер между графами g1, g2, ... |
| CartesianProduct[l1, l2] | возвращает декартово произведение множеств l1 и l2 |

GraphJoin[g1, g2, ...]-конструирует соединение графов g1, g2, ...., причем возвращается граф, полученный прибавлением всех возможных ребер между различными графами.

### *5.7.5. Произведение графов*

Пусть даны два графа $g_1=(E_1,V_1)$ и $g_2=(E_2,V_2)$. Произведением графов $g_1$ и $g_2$ называется граф $g=g_1 \times g_2$, вершины которого – декартово произведение множеств вершин графов $g_1$ и $g_2$, а вершины $(u_1, v_1)$ и $(u_2,v_2)$ смежны в g тогда и только тогда, когда
1.$u_1= u_2$, а $v_1, v_2$ - смежны в $g_2$;
2.$v_1=v_2$, а .$u_1, u_2$ – смежны в $g_1$.
Функция GraphProduct[g1, g2, ...] конструирует произведение графов g1, g2, ... Вложение произведения графов формируется сжатием первого графа и параллельным переносом его на позиции каждой вершины второго графа. Изменение порядка произведения двух графов изменяет вложение произведения, но результирующие графы изоморфны.

Рассмотрим, например, произведение пути на трех вершинах и полного пятивершинного графа.

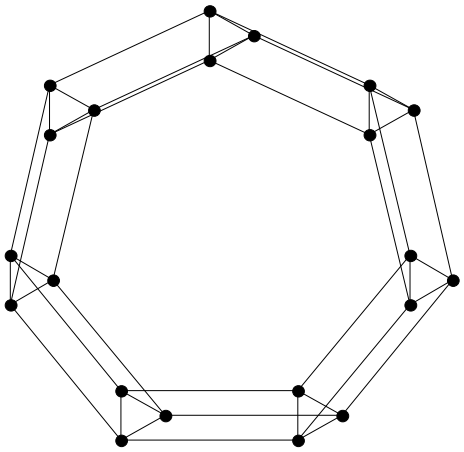
```
In[2]:= g = GraphProduct[Path[3], CompleteGraph[5]]; h = GraphProduct[CompleteGraph[5], Path[3]];
        ShowGraphArray[{g, h}]
In[3]:= IsomorphicQ[g, h]

Out[3]= True
```

Произведение любого графа g на $K_1$ есть граф g.

```
In[4]:= IdenticalQ[GraphProduct[CompleteGraph[1], CompleteGraph[5]], CompleteGraph[5]]
Out[4]= True
```

Рассмотрим произведение 3-цикла и 7-цикла:

```
In[5]:= ShowGraph[h = GraphProduct[Cycle[3], Cycle[7]]]
```

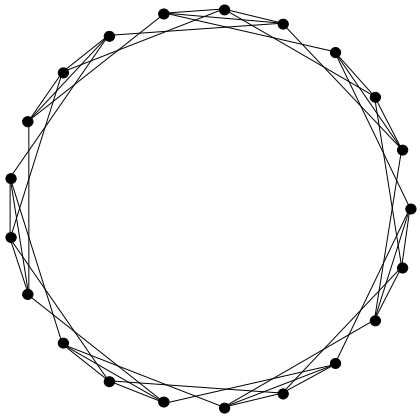


На самом деле это произведение является графом-циркулянтом на 21 вершине, в котором каждая вершина связана через 3 и через 7 по каждую сторону. Действительно,

```
In[6]:= IsomorphicQ[h, CirculantGraph[21, {3, 7}]]
Out[6]= True
```

Рассмотрим циркулярное вложение графа h:

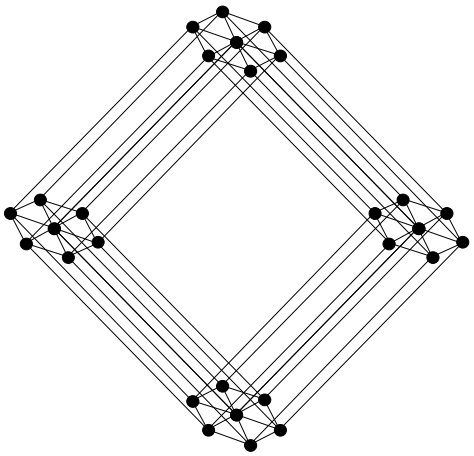
```
In[7]:= ShowGraph[p = CircularEmbedding[h]]
```



С помощью операции произведения вводится важный класс графов – n- мерные гиперкубы
n-мерный гиперкуб $Q_n$ определяется рекуррентно: $Q_1=K_2$, $Q_2=K_2\times Q_1$, $Q_3= K_2\times Q_2$,….,
$Q_n =K_2 \times Q_{n-1}$. Таким образом, n-мерный гиперкуб $Q_n$ есть произведение (n-1)-мерного куба и отрезка $K_2$.
Очевидно, что $Q_n$ – граф порядка $2^n$, вершины которого можно представить бинарными (0,1) строками длины n таким образом, что две вершины смежны тогда и только тогда, когда соответствующие строки отличаются только одной координатой.
Функция Hypercube[n] строит n-мерный гиперкуб.

Построим гиперкуб $K_5$.

```
In[8]:= ShowGraph[Hypercube[5]]
```



Так как каждая вершина n-мерного гиперкуба инцидентна n ребрам, то число его ребер равно $n2^{n-1}$.

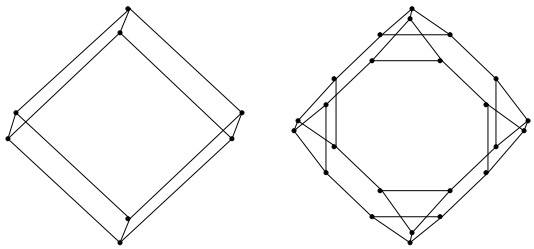
```
In[9]:= n = 10; {{n 2^(n - 1), 2^n}, {M[Hypercube[n]], V[Hypercube[n]]}} // ColumnForm
Out[9]= {5120, 1024}
        {5120, 1024}
```

Все гиперкубы двудольны и регулярны

```
In[10]:= {BipartiteQ[Hypercube[10]], RegularQ[Hypercube[10]]}
Out[10]= {True, True}
```

Функция CubeConnectedCycle[d] возвращает граф, полученный заменой каждой вершины в d-мерном гиперкубе на цикл длины d. Этот граф обладает многими свойствами гиперкуба, но имеет еще одно дополнительное свойство, что при d>1 каждая вершина имеет степень 3.

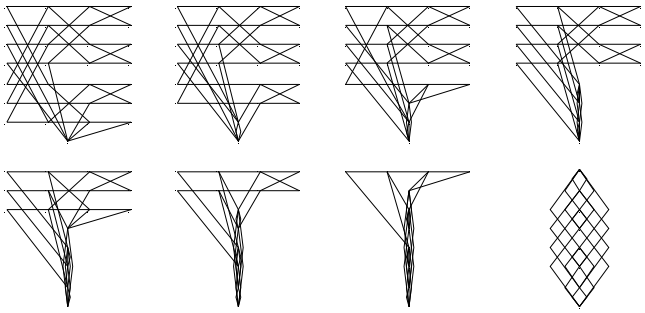
```
In[11]:= ShowGraphArray[{Hypercube[3], CubeConnectedCycle[3]}]
```



```
Out[11]= - GraphicsArray -
```

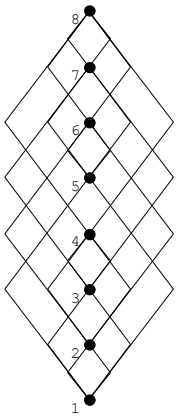
Рассмотрим теперь трехмерный граф бабочки с последовательно стянутыми вершинами, находящихся на одной строке:

```
In[12]:= g=Contract[ButterflyGraph[3],{1,2,3,4}];ShowGraphArray[P
         artition[l=NestList[Contract[#,{1,2,3,4}]&,g,7],4]]
```



Выделим теперь конечный восьмивершинный граф:

```
In[13]:= ShowLabeledGraph[r = l[[8]]]
```

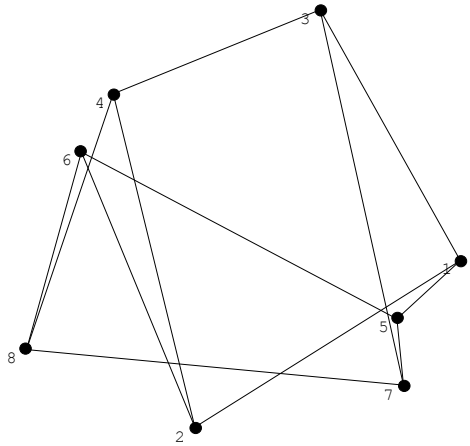



Этот граф содержит 8 вершин и 24 ребра, среди которых есть параллельные

```
In[14]:= r
Out[14]= -Graph:<24, 8, Undirected>-
 In[15]:= Edges[r]
 Out[15]= {{1, 5}, {1, 3}, {1, 5}, {1, 2}, {1, 3}, {1, 2},
          {2, 6}, {2, 4}, {2, 6}, {2, 4}, {3, 7}, {3, 7},
          {3, 4}, {3, 4}, {4, 8}, {4, 8}, {5, 7}, {5, 6},
          {5, 7}, {5, 6}, {6, 8}, {6, 8}, {7, 8}, {7, 8}}
```

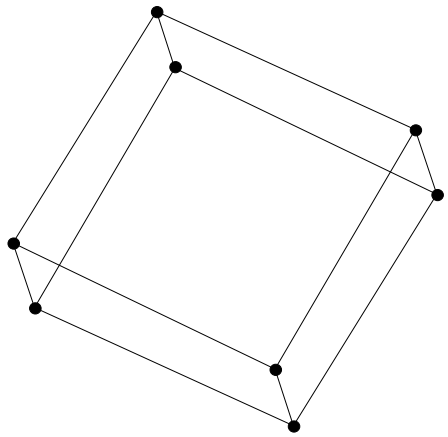
Изменим случайным образом вложение графа r, предварительно удалив параллельные ребра:

```
In[16]:= ShowLabeledGraph[p = RandomVertices[MakeSimple[r]]]
```




Применим к этому графу SpringEmbedding:

```
In[17]:= ShowGraph[SpringEmbedding[p, 200]]
```



Оказывается, стянув все вершины одного уровня в одну вершину в r-мерном графе бабочки и удалив кратные ребра, получим r- мерный гиперкуб!

| | |
|---|---|
| GraphProduct[g1, g2, ...] | конструирует произведение графов g1, g2, ... |
| Hypercube[n] | строит n-мерный гиперкуб |
| CubeConnectedCycle[d] | возвращает граф, полученный заменой каждой вершины в d-мерном гиперкубе на цикл длины d |

### *5.7.6. Реберный граф*

**Реберным графом** L(g) графа g называется граф L(g), такой, что:

1. V(L(g))=E(g)
2. Вершины $e_1$, $e_2$ смежны в L(g) , если ребра $e_1$, $e_2$ смежны в g.

Функция LineGraph[g] конструирует реберный граф графа g.
Очевидно, что если g, h – изоморфные графы, то L(g), L(h) изоморфны. В то же время справедливы соотношения $L(K_3) \cong L(K_{1,3})=K_3$.

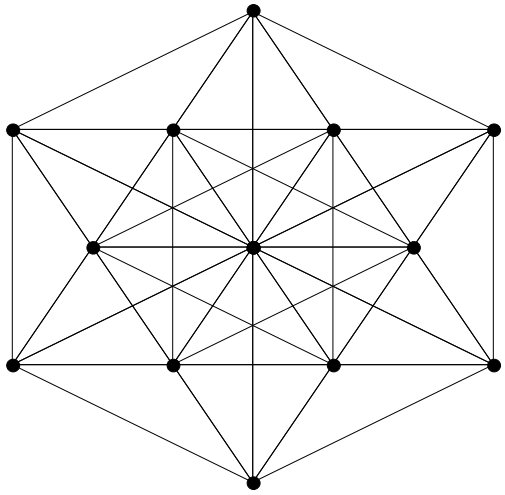
```
In[2]:= {IsomorphicQ[LineGraph[CompleteGraph[3]], LineGraph[CompleteGraph[1, 3]]],
         IsomorphicQ[LineGraph[CompleteGraph[1, 3]], CompleteGraph[3]]}
Out[2]= {True, True}
```

Реберный граф графа с n вершинами и m ребрами содержит m вершин

$0.5 \sum_{i=1}^{n} d_i^2 - m$ ребер.

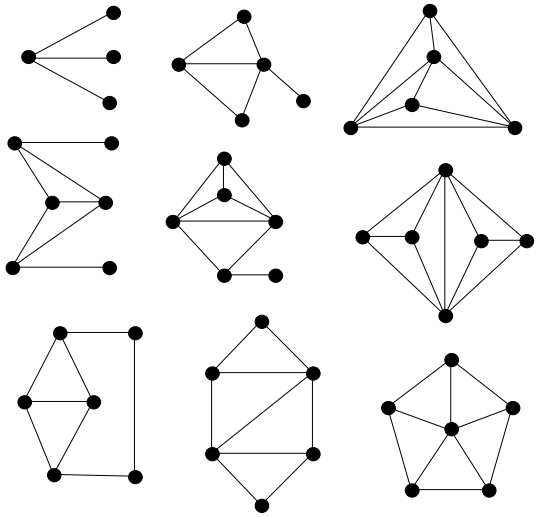
Рассмотрим реберный граф полного графа $K_6$

```
In[3]:= ShowGraph[p = LineGraph[CompleteGraph[6]]]
```



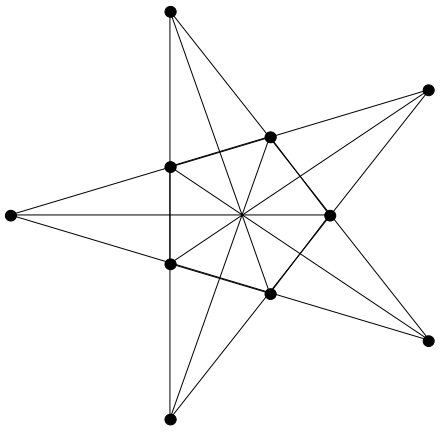
Известно, что граф является реберным тогда и только тогда, когда он не содержит любой из нижеприведенных графов как подграфы. Функция NonLineGraphs возвращают эти 9 графов.

```
In[4]:= ShowGraph[NonLineGraphs]
```



Уитни доказал, что $K_3$, $K_{1,3}$ – единственная пара несовпадающих связных графов, имеющих один и тот же реберный граф, то есть, за исключением $K_3$, $K_{1,3}$, любые два связных графа с изоморфными реберными графами являются изоморфными.
Дополнение реберного графа полного пятивершинного графа есть граф Петерсена. Но мы получим незнакомое вложение графа Петерсена.

```
In[5]:= ShowLabeledGraph[h = GraphComplement[LineGraph[CompleteGraph[5]]]]
```

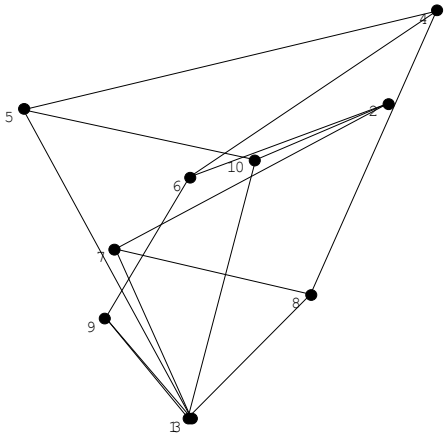


На первый взгляд, кажется, что этот граф имеет одиннадцать вершин, но на самом деле «вершина в центре» - это просто пересечение пяти ребер.

```
In[6]:= IsomorphicQ[h, PetersenGraph]
Out[6]= True
```

Наконец, рассмотрим случайное вложение графа h:

```
In[7]:= ShowLabeledGraph[RandomVertices[h]]
```




```
In[8]:= V[h]
Out[8]= 10
```

| | |
|---|---|
| LineGraph[g] | конструирует реберный граф графа g |
| NonLineGraphs | возвращает 9 графов. Граф является реберным графом какого-либо графа тогда и только тогда, когда он не содержит любой из этих графов как подграфы |

# Глава 6. Анализ графов

## ■ 6.1. Последовательные обходы вершин графа

Очень многие задачи теории графов сводятся к задаче последовательного обхода по всем вершинам графа или, что то же самое, нумерации всех вершин графа. Многие алгоритмы на графах базируются на известных методах последовательного обхода всех вершин: поиске в ширину и поиске в глубину.

### *6.1.1. Поиск в ширину*

Поиск в ширину (ПВШ) следующим образом приписывает вершинам графа номера 0, 1, 2, 3,…
Начиная с произвольной вершины, приписываем ей номер 0. Множество всех вершин графа g, смежных с некоторой вершиной v, назовем окружением вершины v. Каждой вершине из окружения вершины 0 приписываем номер 1. Теперь рассмотрим поочередно окружение всех вершин с номером 1 и каждой из входящих в это окружение вершин, еще не занумерованных, приписываем номер 2. Рассмотрим окружение всех вершин с номером 2 и т.д., пока возможно. Если исходный граф g связен, то поиск в ширину занумерует все вершины графа.
Функция BreadthFirstTraversal[g, v] производит поиск в ширину графа g, начиная с вершины v, и возвращает номера вершин прохода поиска в ширину.
BreadthFirstTraversal[g, v, Edge] возвращает ребра графа, по которым производится проход при поиске в ширину
BreadthFirstTraversal[g, v, Tree] производит дерево прохода поиска в ширину.
BreadthFirstTraversal[g, v, Level] возвращает номер уровня, на котором находятся вершины.
Рассмотрим результат поиска в ширину решетчатого 3×3×3- решетчатого графа

```
In[2]:= BreadthFirstTraversal[GridGraph[3, 3, 3], 1]
Out[2]= {1, 2, 5, 3, 6, 11, 8, 13, 18, 4, 7, 12, 9,
       14, 19, 16, 21, 24, 10, 15, 20, 17, 22, 25, 23, 26, 27}
```

Явный порядок вершин задается обратной перестановкой

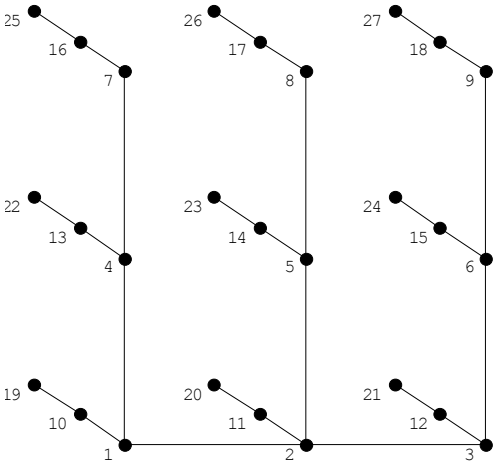
```
In[3]:= InversePermutation[%]
Out[3]= {1, 2, 4, 10, 3, 5, 11, 7, 13, 19, 6, 12, 8,
        14, 20, 16, 22, 9, 15, 21, 17, 23, 25, 18, 24, 26, 27}
```

Использование тег Edge позволяет увидеть ребра дерева, которые связывают вершины с их родителями.

```
In[4]:= BreadthFirstTraversal[GridGraph[3, 3, 3], 1, Edge]
Out[4]= {{1, 2}, {1, 4}, {1, 10}, {2, 3}, {2, 5}, {2, 11}, {4, 7}, {4, 13}, {10, 19},
        {3, 6}, {3, 12}, {5, 8}, {5, 14}, {11, 20}, {7, 16}, {13, 22}, {6, 9}, {6, 15},
        {12, 21}, {8, 17}, {14, 23}, {16, 25}, {9, 18}, {15, 24}, {17, 26}, {18, 27}}
```

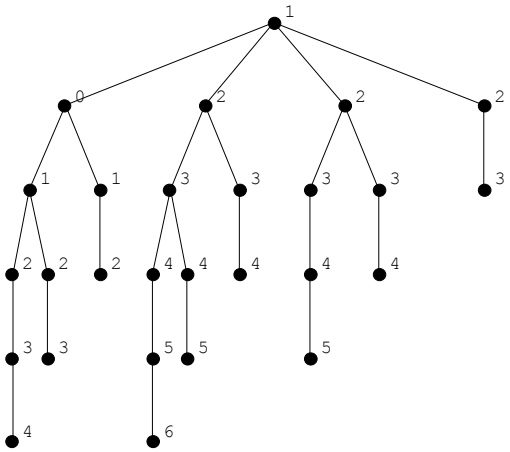
Если мы начинаем со связного графа, производим поиск в ширину и удаляем все ребра из графа, которые не являются ребрами дерева, то получим дерево поиска в ширину.

```
In[5]:= ShowLabeledGraph[h = BreadthFirstTraversal[g = GridGraph[3, 3, 3], 1, Tree]]
```



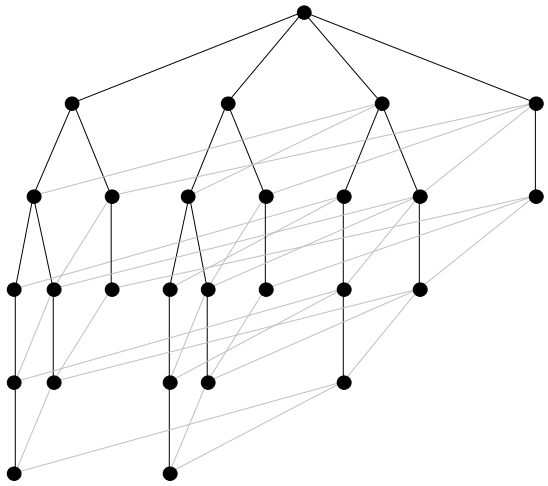

Если мы берем ПВШ-дерево как корневое в источнике поиска, тогда расстояние от корня r к любой вершине v вдоль единственного простого пути в дереве равно расстоянию между ними в начальном графе. Таким образом, ПВШ -дерево обеспечивает удобное представление всех кратчайших путей из корня. Использование тег Level в BreadthFirstTraversal возвращает эти расстояния.

```
In[6]:= nh = RootedEmbedding[BreadthFirstTraversal[g, 1, Tree]];
        ShowGraph[SetVertexLabels[nh, BreadthFirstTraversal[g, 1, Level]]]
```



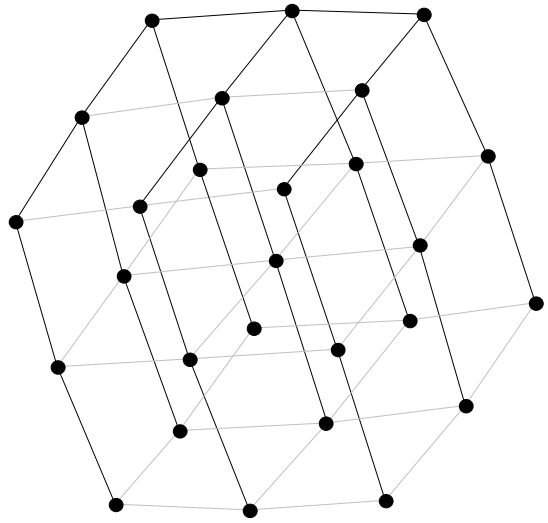

Прибавим ребра недерева графа к дереву поиска в ширину и получим незнакомое вложение 3×3×3-решетчатого графа. Заметим, что все ребра связывают вершину на глубине d с вершиной на глубине d+1 для некоторого d. Это означает, что граф не имеет циклов нечетной длины, и это, в свою очередь, означает, что граф двудольный.

```
In[7]:= ne = Complement[Edges[g], Edges[h]];
        ShowGraph[f = SetGraphOptions[AddEdges[nh, ne], Append[ne, EdgeColor → Gray]]]
```



Изменим вложение графа f:

```
In[8]:= ShowGraph[SpringEmbedding[f]]
```

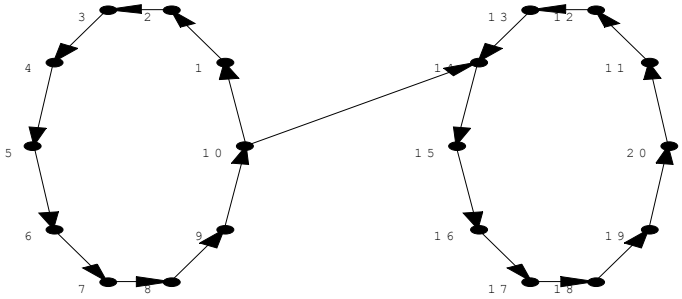


Поиск в ширину исследует только одну связную компоненту. Неисследованным вершинам в выводе соответствуют нули.

```
In[9]:= BreadthFirstTraversal[GraphUnion[2, Cycle[10]], 1]
Out[9]= {1, 2, 4, 6, 8, 10, 9, 7, 5, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

Построим  ориентированный граф, начиная с объединения двух ориентированных циклов и прибавляя ребро от первого цикла ко второму

```
In[10]:= ShowLabeledGraph[g = AddEdge[GraphUnion[2, Cycle[10, Type → Directed]], {10, 14}]]
```



Если поиск в ширину начинается из вершины левого цикла, мы пройдем все вершины графа.

```
In[11]:= BreadthFirstTraversal[g, 1]
Out[11]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 18, 19, 20, 11, 12, 13, 14, 15, 16, 17}
```

Но если поиск начинается в правом цикле, то вершины слева не могут быть пройдены

```
In[12]:= BreadthFirstTraversal[g, 15]
Out[12]= {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7, 8, 9, 10, 1, 2, 3, 4, 5, 6}
```

| | |
|---|---|
| BreadthFirstTraversal[g, v] в ширину | производит поиск в ширину графа g, начиная с вершины v, и возвращает номера вершин прохода поиска |
| BreadthFirstTraversal[g, v, Edge] | возвращает ребра графа, которые проходятся при поиске в ширину |
| BreadthFirstTraversal[g, v, Tree] | производит дерево прохода поиска в ширину |
| BreadthFirstTraversal[g, v, Level] | возвращает номер уровня, на котором находятся вершины |

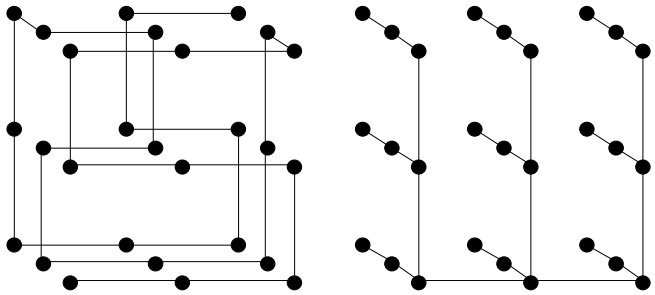
### *6.1.2. Поиск глубину*

Рассмотрим в начале поиск в глубину (ПВГ) в неориентированном графе. Мы предполагаем, что рассматриваемые графы связны. Если граф несвязен, то поиск в глубину выполняется отдельно в каждой компоненте графа. Мы также будем предполагать, что в графе нет петель. Поиск в глубину в неориентированном графе выполняется следующим образом:
В графе g выбираем произвольную вершину v и начинаем из нее поиск. Стартовая вершина v, называемая корнем ПВГ, после этого считается пройденной. Затем выбираем ребро (v,w), инцидентное вершине v, и проходим его, чтобы попасть в вершину w. Ориентируем при этом ребро из v в w. Ребро (v,w) после этих действий считается просмотренным и называется ребром дерева. Вершина v называется отцом вершины w и обозначается как Father(w).
В общем случае, когда мы находимся в какой-либо вершине x, возникают две возможности:
1. Если все ребра, инцидентные x, уже просмотрены, то мы возвращаемся к отцу x и продолжаем поиск из Father(x). Вершина x с этого момента называется полностью сканированной.
2.Если существуют непросмотренные ребра, инцидентные x, то мы выбираем одно из таких ребер (x,y) и ориентируем его из x в y. Ребро (x,y) с этого момента считается просмотренным. Необходимо рассмотреть два случая:
Случай 1. Если y ранее не была пройдена, то мы проходим ребро (x,y), вершину y и продолжаем поиск из вершины y. В этом случае ребро (x,y) называется ребром дерева и x= Father(y).
Случай 2. Если y ранее была пройдена, то мы продолжаем поиск другого непросмотренного ребра, инцидентного x. В этом случае ребро (x,y) называется обратным ребром. Во время поиска в глубину, когда вершину x проходят в первый раз, ей сопоставляется такое целое число DFN(x), что DFN(x) равно i, если x является i-й по порядку прохождения вершиной. DFN(x) называется глубиной x. Ясно, что глубина указывает порядок, в котором проходят вершины при поиске в глубину. ПВГ завершается, когда мы возвращаемся в корень и все вершины графа пройдены. Из

описания видно, что поиск в глубину разбивает ребра графа g на ребра дерева и обратные ребра. Легко показать, что ребра дерева образуют остов графа g. ПВГ вводит ориентацию на ребра графа g. Получаемый в результате ориентированный граф мы будем обозначать $g_1$. Ребра дерева с ориентацией, налагаемой ПВГ, будут образовывать ориентированный остов $g_1$. Этот ориентированный остов называется деревом ПВГ. Отметим, что способ прохождения графа не единственен, так как ребра, инцидентные вершине, могут выбираться для рассмотрения в произвольном порядке.

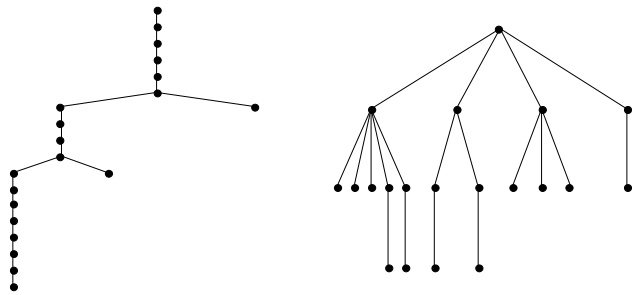
DepthFirstTraversal[g, v] производит поиск в глубину графа g, стартуя с вершины v, и выдает список вершин в том порядке, в котором они встречаются при поиске.
DepthFirstTraversal[g, v, Edge] возвращает ребра графа, по которым совершается поиск в глубину в том порядке, в котором они встречаются при поиске.
DepthFirstTraversal[g, v, Tree] возвращает дерево поиска в глубину графа g.
Рассмотрим поиск в глубину и поиск в ширину вышеприведенного 3×3×3- решетчатого графа. Поиск в глубину дает простой путь на 27 вершинах, а поиск в ширину производит более «низкое» дерево, каждая из вершин которого находится на расстоянии по крайней мере шести ребер от стартовой вершины.

```
In[2]:= g = GridGraph[3, 3, 3];
        ShowGraphArray[{DepthFirstTraversal[g, 1, Tree], BreadthFirstTraversal[g, 1, Tree]},
         VertexStyle → Disk[0.05]]
```



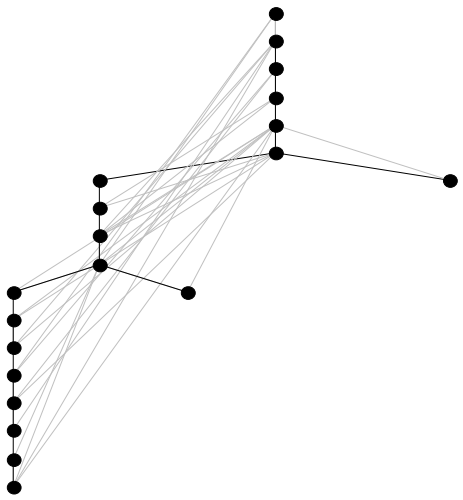
Рассмотрим еще один пример, показывающий разницу между поиском в ширину и поиском в глубину на случайном 20-вершинном графе:

```
In[3]:= g = RandomGraph[20, 0.3];
        ShowGraphArray[{h = RootedEmbedding[DepthFirstTraversal[g, 1, Tree], 1],
          RootedEmbedding[BreadthFirstTraversal[g, 1, Tree], 1]}]
```



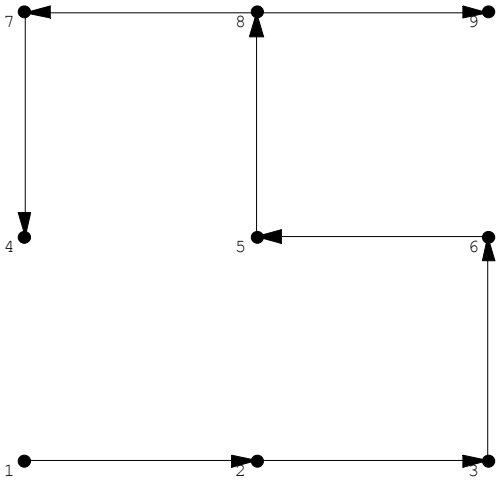
Добавим обратные ребра к ПВГ-дереву и окрасим их в серый цвет. Каждое обратное ребро (v,u) превращает цикл с путем из u в v в ПВГ-дерево. Ясно, что поиск в глубину лежит в основе алгоритмов, которые ищут циклы, двусвязные компоненты и сильно связные компоненты.

```
In[4]:= ne = Complement[Edges[g], Edges[h]];
        ShowGraph[SetGraphOptions[AddEdges[h, ne], Append[ne, EdgeColor → Gray]]]
```



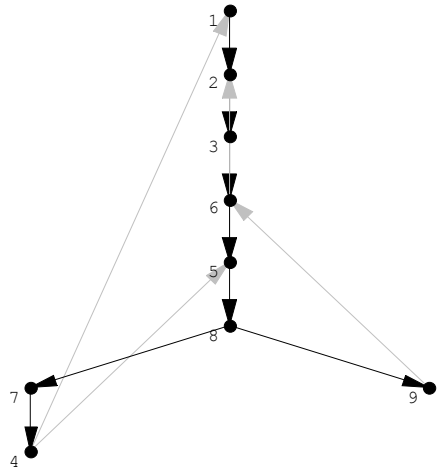
Поиск в глубину в ориентированном графе в основном совпадает с поиском в неориентированном графе. Главное отличие в этом случае заключается в том, что ребра графа проходятся только в соответствии с ориентацией. Как следствие этого ограничения ребра в ориентированном графе g разбиваются поиском в глубину в g на четыре категории (а не на две, как в случае неориентированного графа). Непросмотренное ребро (v,w), встречающееся, когда мы находимся в вершине v, можно классифицировать следующим образом:
Случай 1. Вершина w еще не пройдена. В этом случае (v,w)-ребро дерева.
Случай 2. Вершина w уже была пройдена.
А.Если w – потомок v в лесе ПВГ, то ребро (v,w) называется прямым ребром.
Б. Если w-предок v в лесе ПВГ, то ребро (v,w) называется обратным ребром.
В.Если v и u несоотносимы в лесе ПВГ и DFN(w)<DFN(v), то ребро (v,w) является пересекающим ребром.
Рассмотрим ПВГ ориентированного решетчатого 3×3-графа:

```
In[5]:= ShowLabeledGraph[DepthFirstTraversal[g = OrientGraph[GridGraph[3, 3]], 1, Tree]]
```




Покажем корневое вложение ПВГ этого графа. Ребра, которые связывают вершину с ее потомком, например, (1,2) или (8,7), – прямые ребра. Серые ребра, которые связывают вершину с родителем, например (3,2), являются обратными ребрами. Оставшиеся ребра, пересекающие дерево, например, (4,1) – пересекающие ребра.

```
In[6]:= h = RootedEmbedding[DepthFirstTraversal[g, 1, Tree], 1];
        ne = Select[Complement[Edges[g], Edges[h]], #[[1]] ≠ #[[2]] &];
        ShowGraph[SetGraphOptions[AddEdges[h, ne], Append[ne, EdgeColor → Gray]],
         VertexNumber → True]
```

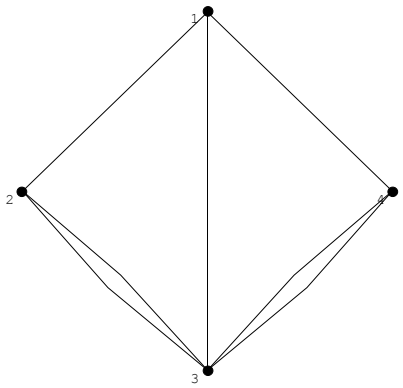



| | |
|---|---|
| DepthFirstTraversal[g, v] | производит поиск в глубину графа g, стартуя с вершины v, и выдает список вершин в том порядке, в котором они встречаются при поиске |
| DepthFirstTraversal[g, v, Edge] | возвращает ребра графа, по которым совершается поиск в глубину в том порядке, в котором они встречаются при поиске |
| DepthFirstTraversal[g, v, Tree] | возвращает дерево поиска в глубину графа g |

## ■6.2. Эйлеровы и гамильтоновы графы

Многие открытия теории графов были использованы для решения практических проблем – задач, головоломок, игр и т.д. К одной из этих задач относится знаменитая задача о кенигсбергских мостах. Город Кенигсберг расположен на берегах и двух островах реки Преголи. Острова между собой и с берегами были связаны семью мостами. Жители города любили размышлять над проблемой: можно ли, выйдя из дома, вернуться обратно, пройдя по каждому мосту ровно один раз?
Мы можем изобразить мосты Кенигсберга, используя Combinatorica

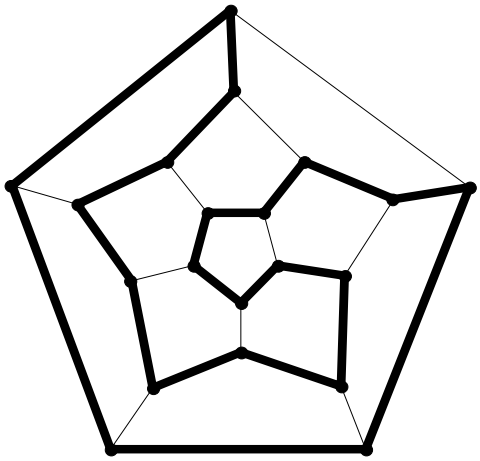
```
In[2]:= ShowLabeledGraph[
          k = FromAdjacencyMatrix[{{0, 1, 1, 1}, {1, 0, 2, 0}, {1, 2, 0, 2}, {1, 0, 2, 0}}]]
```




В этом графе вершины 2 и 4 соответствуют правому и левому берегам реки, вершины 1 и 3 – островам, ребра графа – мостам. На языке графов, следовательно, задача формулируется следующим образом: существует ли в графе простой цикл, содержащий все ребра графа (эйлеров цикл).
Л. Эйлер в 1736 году сформулировал и доказал необходимое и достаточное условие того, чтобы в неориентированном связном графе существовал эйлеров цикл: необходимо и достаточно, что-

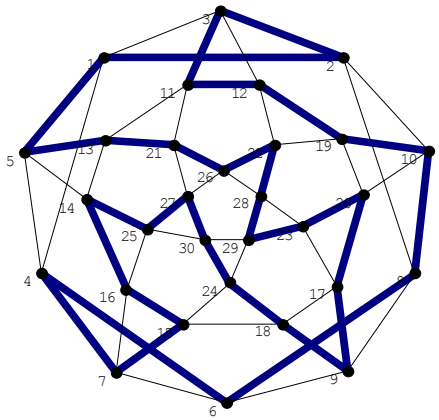
бы все вершины графа имели четную степень. Теперь совершенно очевидно, что в графе, моделирующем задачу о мостах Кенигсберга, эйлерова цикла найти нельзя.
В 1859 году другой известный математик У.Гамильтон придумал игру, в котором требуется обойти замкнутый контур всех ребер додекаэдра, минуя каждую вершину лишь один раз. В теории графов эта игра эквивалентна определению цикла, содержащего все двадцать вершин. Все графы, в которых существует подобный цикл, называются гамильтоновыми.
Выделим гамильтонов цикл в графе додекаэдра:

```
In[3]:= ShowGraph[Highlight[d = DodecahedralGraph, {Partition[HamiltonianCycle[d], 2, 1]}]]
```



Реберный граф любого гамильтонова графа сам является гамильтоновым. Выделим гамильтонов цикл в реберном графе графа додекаэдра:

```
In[4]:= d = Partition[HamiltonianCycle[g = LineGraph[DodecahedralGraph]], 2, 1];
        ShowLabeledGraph[Highlight[g, {d}, HighlightedEdgeColors → {Navy}],
         VertexNumber → True]
```




### *6.2.1. Эйлеровы графы*

**Эйлеровым циклом** в графе g называется цикл, содержащий все ребра графа g. Граф, содержащий эйлеров цикл, называется **эйлеровым графом**. Теперь сформулируем теорему, дающую простую и часто используемую характеризацию эйлеровых графов:

**Теорема 1.** Для связного графа g следующие утверждения эквивалентны:
1. g – эйлеров граф.
2. Степень каждой вершины в графе g четная.

3. g является объединением нескольких реберно-непересекающихся циклов.

Граф k кенигсбергских мостов, изображенный выше, не является эйлеровым, т.к. степени каждой вершины нечетные. Получить дополнительное подтверждение этому можно с помощью функции – теста EulerianQ. Функция EulerianQ[g] возвращает True, если g является эйлеровым и False в противном случае.

```
In[3]:= EulerianQ[k]
Out[3]= False
```

Полный двудольный граф $K_{4,4}$ является эйлеровым

```
In[4]:= EulerianQ[q = CompleteKPartiteGraph[4, 4]]
Out[4]= True
```

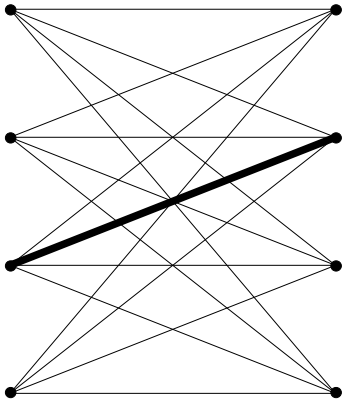
Функция EulerianCycle[g] находит эйлеров цикл графа g, если таковой существует.

Выделим эйлеров цикл в графе q:

```
In[5]:= EulerianCycle[q]
Out[5]= {7, 2, 8, 1, 5, 4, 6, 3, 7, 4, 8, 3, 5, 2, 6, 1, 7}
```

Чтобы увидеть проход эйлерова цикла, анимируем эйлеров цикл. Кликнув дважды на нижней картинке можно увидеть, как жирная ломаная проходит весь цикл.

```
In[6]:= AnimateGraph[q, Partition[EulerianCycle[q], 2, 1]]
```



Полный граф на девятнадцати вершинах является эйлеровым

```
In[7]:= EulerianQ[CompleteGraph[19]]
Out[7]= True
```

Эйлеров цикл проходит каждое ребро ровно один раз:

```
In[8]:= EmptyQ[DeleteCycle[CompleteGraph[19], EulerianCycle[CompleteGraph[19]]]]
Out[8]= True
```

Ориентированным эйлеровым циклом ориентированного графа g называется ориентированный цикл, содержащий все дуги g. В следующей теореме даются простые характеризации ориентированных эйлеровых графов.

**Терема 2.** Для связного ориентированного графа следующие утверждения равносильны:

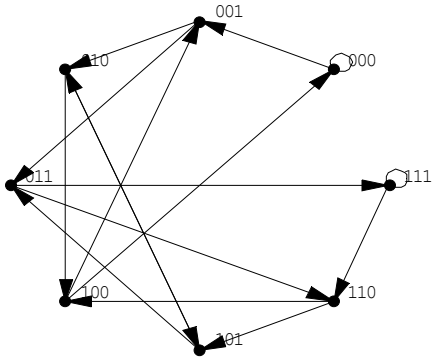
1. g – ориентированный эйлеров граф;
2. для любой вершины v графа g справедливо равенство $d^-(v)=d^+(v)$
3. g –объединение нескольких реберно-непересекающихся контуров.

Рассмотрим одно интересное применение этой теоремы.
Пусть S={0,1,2,……,s-1} – алфавит, состоящий из s букв. Очевидно, что, используя буквы этого алфавита S, можно построить $s^n$ различных слов длины n. Последовательность де Брёйна – это циклическая последовательность $a_0,a_1,a_2,….,a_{L-1}$ длины $L=s^n$, у которой для любого слова w длины n существует такое единственное i, что $w= a_ia_{i+1}…..a_{i+n-1}$, где значения индексов вычисляются по mod L.
Последовательности де Брёйна находят применение в теории кодирования и при исследовании сетей связи. Оказывается, для любых $s\geq2$ и целых n существует последовательность де Брёйна. Встроенная функция DeBruijnSequence[a, n] возвращает последовательность де Брёйна, где a - алфавит.

```
In[9]:= s = DeBruijnSequence[{0, 1, 2}, 3]
Out[9]= {2, 0, 2, 1, 0, 1, 0, 0, 2, 0, 0, 0, 1, 2, 0, 1, 1, 1, 2, 2, 2, 1, 2, 1, 1, 0, 2}
```

Для любых $s\geq2$ и n существует такой ориентированный эйлеров граф $g_{s,n,}$, что всякая ориентированеная эйлерова цепь в этом графе соответствует последовательности де Брёйна длины $s^n$ над алфавитом S={0,1,2,……,s-1}. Этот граф называется графом де Брёйна и строится следующим образом:
1. Последовательность вершин V образуется всеми $s^{n-1}$ словами длины n-1 над алфавитом S.
2. Множество ребер E – множество всех $s^n$ слов длины n над S.
3. Дуга $b_1b_2…..b_n$ имеет начальную вершину $b_1b_2…..b_{n-1}$ и конечную вершину $b_2…..b_n$. Заметим, что каждая вершина v справедливо равенство $d^-(v)=d^+(v)$.
Функция DeBruijnGraph[m, n] конструирует n - мерный граф де Брёйна с m символами, m>0, n>1. DeBruijnGraph[alph, n] конструирует n-мерный граф де Брёйна с символами из алфавита alph. В последней форме функция допускает опцию VertexLabel, которая по умолчанию принимает значения False.

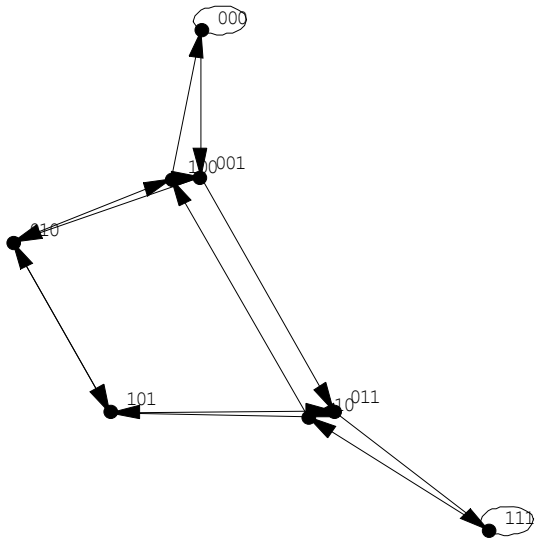
```
In[10]:= ShowGraph[a = DeBruijnGraph[{0, 1}, 3, VertexLabel → True], PlotRange → 0.25]
```




```
In[11]:= EulerianCycle[a]
Out[11]= {4, 7, 5, 2, 3, 6, 4, 8, 8, 7, 6, 3, 5, 1, 1, 2, 4}
```

Применим к графу a SpringEmbedding, что выявит симметрическую структуру графа

```
In[12]:= ShowGraph[SpringEmbedding[a]]
```




Графы де Брёйна всегда эйлеровы

```
In[13]:= EulerianQ[DeBruijnGraph[{2, 3, 4, 5, 6}, 5]]
Out[13]= True
```

| | |
|---|---|
| EulerianQ[g] | возвращает True, если g является эйлеровым и False в противном случае |
| EulerianCycle[g] | возвращает эйлеров цикл графа g, если таковой существует |
| DeBruijnSequence[a, n] | возвращает последовательность де Брёйна, где a-алфавит |
| DeBruijnGraph[m, n] | конструирует n-мерный граф де Брёйна с m символами, m>0, n>1 |
| DeBruijnGraph[alph, n] | конструирует n-мерный граф де Брёйна с символами из алфавита alph. Функция допускает опцию VertexLabel, которая по умолчанию принимает значения False |

### 6.2.2 Гамильтоновы графы

**Гамильтоновым циклом** в графе g называется цикл, содержащий все вершины графа g. Гамильтонов путь в графе g – это путь, содержащий все вершины графа g. Граф называется **гамильтоновым**, если он имеет гамильтонов цикл. Задача вычисления гамильтоновых циклов или гамильтоновых путей фундаментально отличается от задачи вычисления эйлеровых циклов, потому что тестирование, будет ли граф гамильтоновым, – NP полная задача.
Функция HamiltonianQ[g] возвращает True, если g – гамильтонов граф.
Функция HamiltonianCycle[g] – находит гамильтонов цикл в графе g, если таковой существует. HamiltonianCycle[g, All] – возвращает все гамильтоновы циклы графа g.

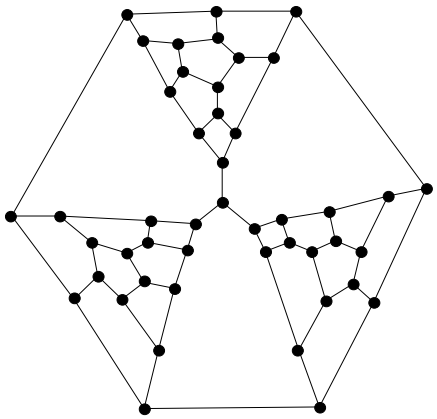
```
In[2]:= HamiltonianCycle[CompleteKPartiteGraph[3, 3], All]
Out[2]= {{1, 4, 2, 5, 3, 6, 1}, {1, 4, 2, 6, 3, 5, 1}, {1, 4, 3, 5, 2, 6, 1}, {1, 4, 3, 6, 2, 5, 1},
        {1, 5, 2, 4, 3, 6, 1}, {1, 5, 2, 6, 3, 4, 1}, {1, 5, 3, 4, 2, 6, 1}, {1, 5, 3, 6, 2, 4, 1},
        {1, 6, 2, 4, 3, 5, 1}, {1, 6, 2, 5, 3, 4, 1}, {1, 6, 3, 4, 2, 5, 1}, {1, 6, 3, 5, 2, 4, 1}}
```

Все гамильтоновы графы двусвязные, но не наоборот.

```
In[3]:= HamiltonianQ[CompleteKPartiteGraph[5, 4]]
Out[3]= False
```

Граф Татта планарный и однородный степени 3, но не является гамильтоновым

```
In[4]:= ShowGraph[TutteGraph]
```



Даже хотя граф Татта достаточно большой (46 вершин и 69 ребер), наша функция способна показать, что он не гамильтонов

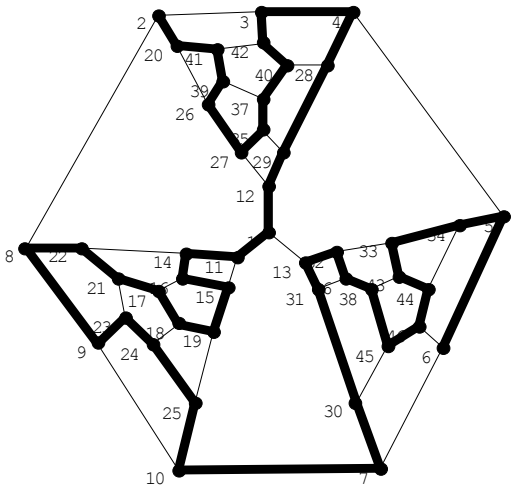
```
In[5]:= HamiltonianQ[TutteGraph]
Out[5]= False
```

Граф Татта не гамильтонов, но он имеет гамильтонов путь:

```
In[6]:= HamiltonianPath[TutteGraph]
Out[6]= {2, 20, 41, 39, 26, 27, 35, 37, 40, 42, 3, 4, 28, 29, 12, 1, 11, 14, 16, 15, 19, 18, 17,
         21, 22, 8, 9, 23, 24, 25, 10, 7, 30, 31, 13, 32, 36, 38, 45, 46, 44, 43, 33, 34, 5, 6}
```

Выделим этот гамильтонов путь:

```
In[7]:= ShowLabeledGraph[
          Highlight[TutteGraph,
           {Partition[{2, 20, 41, 39, 26, 27, 35, 37, 40, 42, 3, 4, 28, 29, 12, 1, 11,
              14, 16, 15, 19, 18, 17, 21, 22, 8, 9, 23, 24, 25, 10, 7, 30, 31, 13, 32,
              36, 38, 45, 46, 44, 43, 33, 34, 5, 6}, 2, 1]}]]
```

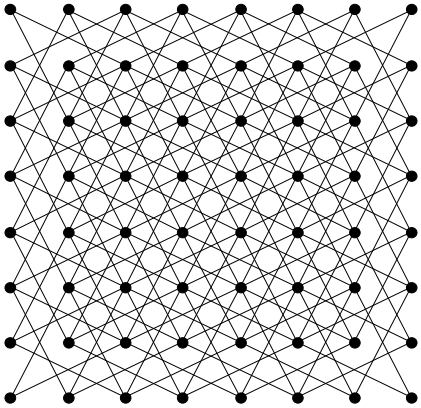



Время вычисления:

```
In[8]:= {Timing[HamiltonianPath[TutteGraph];], Timing[HamiltonianQ[TutteGraph];]}
Out[8]= {{1250.81 Second, Null}, {16.984 Second, Null}}
```

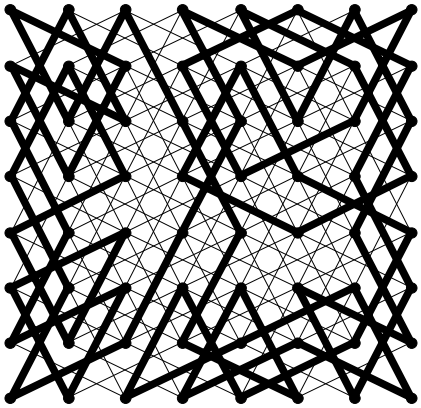
В применениях графов к шахматным играм вершины графа соответствуют различным позициям. Таким образом, существование гамильтонова цикла равносильно существованию циклической последовательности ходов, содержащей каждую позицию по одному разу. Примером является известная задача о шахматном коне: можно ли, начиная из произвольного поля на доске, ходить конем в такой последовательности, чтобы пройти каждое из шестидесяти четырех полей и вернуться в исходное? Функция KnightsTourGraph[m, n] конструирует граф с m×n вершинами, причем каждая из вершин представляет вершину m×n шахматной доски, а каждое ребро соответствует ходу коня из одного поля в другое.

```
In[9]:= ShowGraph[KnightsTourGraph[8, 8]]
```



Чтобы решить задачу о шахматном коне, достаточно выделить гамильтонов цикл в этом графе.

```
In[10]:= ShowGraph[Highlight[k = KnightsTourGraph[8, 8],
           {Partition[HamiltonianCycle[k], 2, 1]}]]
```

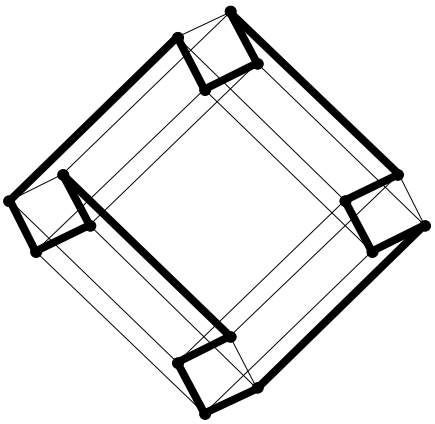


Время вычисления этого гамильтонова цикла:

```
In[11]:= Timing[HamiltonianCycle[KnightsTourGraph[8, 8]]]
Out[11]= {6.562 Second, {1, 11, 5, 15, 21, 4, 10, 20, 3, 9, 19, 2, 12, 6, 16, 22, 7, 13, 23, 8, 14,
           24, 30, 36, 26, 41, 35, 25, 42, 57, 51, 34, 17, 27, 33, 50, 60, 45, 39, 29, 44, 61, 55,
           40, 46, 56, 62, 52, 58, 43, 49, 59, 53, 63, 48, 31, 37, 54, 64, 47, 32, 38, 28, 18, 1}}
```

Четырехмерный гиперкуб является гамильтоновым

```
In[12]:= ShowGraph[Highlight[l = Hypercube[4], {Partition[HamiltonianCycle[l], 2, 1]}]]
```

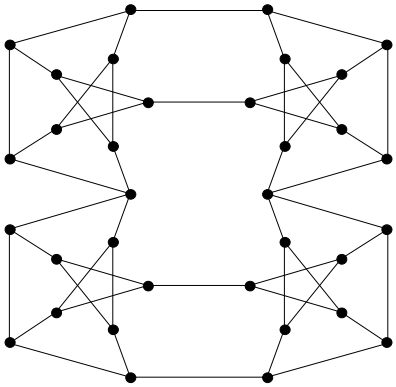


Четырехмерный гиперкуб с удаленной одной вершиной не является гамильтоновым, но он содержит гамильтонов путь

```
In[13]:= {g = DeleteVertex[Hypercube[4], 1]; HamiltonianQ[g], HamiltonianPath[g]}
Out[13]= {False, {1, 2, 6, 5, 4, 7, 11, 8, 9, 10, 14, 13, 12, 15, 3}}
```

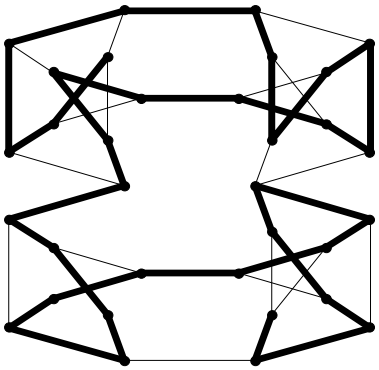
Граф g называется **гипоследным,** если он не имеет гамильтоновых путей, но чей подграф {g –v} имеет гамильтонов путь для каждой вершины v. Граф Томассена - пример гипоследного графа.

```
In[14]:= ShowGraph[ThomassenGraph]
```



Покажем, что граф Томассена не имеет гамильтоновых путей, но, например, граф, полученный удалением одной вершины, имеет гамильтонов путь.

```
In[15]:= ShowGraph[Highlight[g = DeleteVertex[ThomassenGraph, 1],
           {Partition[HamiltonianPath[g], 2, 1]}]]
```

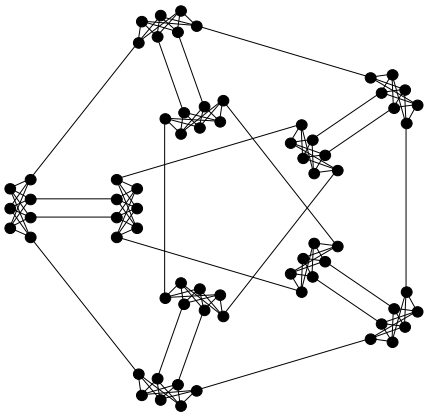


Рассмотрим гамильтонов путь графа g и время его вычисления:

```
In[16]:= Timing[HamiltonianPath[g]]
Out[16]= {437.688 Second, {5, 7, 26, 25, 24, 33, 18, 15, 17, 32, 31, 16, 19,
          9, 6, 8, 20, 21, 1, 3, 23, 22, 2, 4, 14, 12, 29, 30, 13, 11, 28, 27, 10}}
```

Граф Мередит – пример 4-регулярного четырехсвязного графа, который не является гамильтоновым

```
In[17]:= DegreeSequence[MeredithGraph]
Out[17]= {4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
          4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
          4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4}

In[18]:= g = DeleteEdges[MeredithGraph, {{1, 5}, {1, 6}, {1, 7}, {1, 18}}];
          ConnectedQ[g]
Out[18]= False

In[19]:= ShowGraph[MeredithGraph]
```



| | |
|---|---|
| HamiltonianQ[g] | возвращает True, если g – гамильтонов граф |
| HamiltonianCycle[g] | возвращае гамильтонов цикл в графе g, если таковой существует |
| HamiltonianCycle[g, All] | возвращает все гамильтоновы циклы графа g |
| TutteGraph | возвращает граф Тата, первый известный пример 3-связного, 3-регулярного планарного графа, который не является гамильтоновым |
| KnightsTourGraph[m, n] | конструирует граф с m×n вершинами, причем каждая из вершин представляет вершину m×n шахматной доски, а каждое ребро соответствует ходу коня из одного поля в другое |
| ThomassenGraph | возвращает гипоследный граф, который не имеет гамильтоновых путей, но чей подграф g-v для каждой вершины v имеет гамильтонов путь |
| MereditGraph | возвращает граф Мередита, 4-регулярный четырехсвязный граф, который не является гамильтоновым |

### *6.2.3. Задача коммивояжера*

К гамильтоновым путям сводится так называемая задача коммивояжера. Дано множество городов, связанных дорогами, найти кратчайший путь, который посещает каждый город ровно один раз.

Функция TravelingSalesman[g] находит оптимальный тур коммивояжера в неориентированном графе g.
TravelingSalesmanBounds[g] выдает наибольшую и наименьшую границы тура коммивояжера в графе g.
CostOfPath[g, p] суммирует веса ребер пути p в графе g.

Вес гамильтонова цикла в невзвешенном графе равен числу вершин в нем:

```
In[2]:= CostOfPath[g = DodecahedralGraph, HamiltonianCycle[g]]
Out[2]= 20
```

Присвоим ребрам веса – случайные числа из отрезка [1,10]

```
In[3]:= g = SetEdgeWeights[CompleteGraph[6], WeightingFunction → RandomInteger,
          WeightRange → {1, 10}];
```

Выделим веса полного графа в матрице смежности. Неравномерные веса указывают, что различные гамильтоновы циклы возвращают различные цены туров. Так как граф неориентированный, то матрица смежности симметрическая.

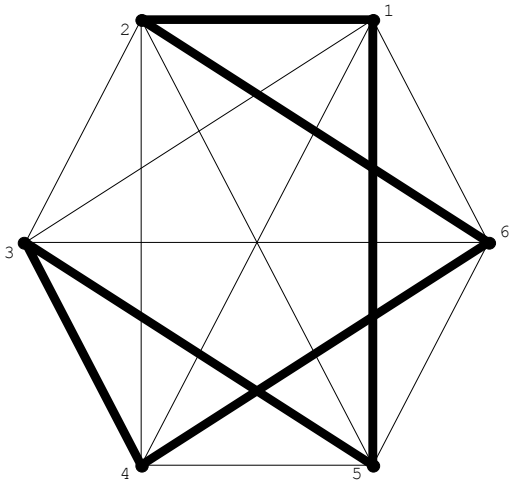
```
In[4]:= ToAdjacencyMatrix[g, EdgeWeight] // ColumnForm
Out[4]= {∞, 1, 10, 7, 6, 9}
        {1, ∞, 9, 3, 5, 2}
        {10, 9, ∞, 5, 3, 10}
        {7, 3, 5, ∞, 6, 4}
        {6, 5, 3, 6, ∞, 3}
        {9, 2, 10, 4, 3, ∞}
```

Так как g – полный граф, он гамильтонов и, таким образом, содержит небесконечное решение.

```
In[5]:= tur = Partition[s = TravelingSalesman[g], 2, 1]
Out[5]= {{1, 2}, {2, 6}, {6, 4}, {4, 3}, {3, 5}, {5, 1}}
```

Выделим оптимальный тур с помощью Highlight:

```
In[6]:= h = SetGraphOptions[g, {{1, 6, VertexNumberPosition → UpperRight}}];
        ShowLabeledGraph[Highlight[h, {tur}]]
```




Цена оптимального тура находится в следующих границах:

```
In[7]:= TravelingSalesmanBounds[h]
Out[7]= {13, 24}
```

Вычислим цену этого тура:

```
In[8]:= CostOfPath[h, s]
Out[8]= 21
```

Вычислим цену цикла {1, 2, 3, 4, 5, 6, 1}, которая должна быть больше 20:

```
In[9]:= CostOfPath[h, {1, 2, 3, 4, 5, 6, 1}]
Out[9]= 33
```

| | |
|---|---|
| TravelingSalesman[g] | находит оптимальный тур коммивояжера в графе g |
| TravelingSalesmanBounds[g] | выдает наибольшую и наименьшую границы тура коммивояжера в графе g |
| CostOfPath[g, p] | суммирует веса ребер пути p в графе g |

## ■6.3. Связность графов

В п. 5.3 мы определили, что граф называется связным, если между двумя произвольными вершинами существует путь. Предположим, что граф g связный. Тогда нас будет интересовать, «как хорошо» он связан. Другими словами, нам бы хотелось знать минимальное число вершин или ребер, удаление которых превращало бы граф g в несвязный. Это приводит нас к понятиям «вершинная связность» и «реберная связность».

**Связностью** $\chi$ (g) называется минимальное число вершин, удаление которых из g приводит к несвязному или тривиальному графу. Тривиальным называется граф, имеющий только одну вершину. Величина $\chi$(g) называется также вершинной связностью, чтобы отличить ее от реберной связности.

**Разделяющее множество** графа g – это множество вершин, удаление которого из графа g приводит к несвязному или тривиальному графу.

Рассмотрим граф на n вершинах. Ясно, что $\chi$(g)=n-1, если граф g полный. Если же он не полный, то имеет хотя бы две несмежные вершины $v_1$ и $v_2$. Удаление из графа g оставшихся n-2 вершин приведет к графу, в котором вершины $v_1$ и $v_2$ не соединены путем. Таким образом, если g не полный, то $\chi(g) \le n-2$.

Функция VertexConnectivity[g] возвращает вершинную связность графа g. VertexConnectivity[g, Cut] возвращает минимальное разделяющее множество графа g. Рассмотрим вершинную связность десятивершинной звезды, полного двудольного графа и полного десятивершинного графа:

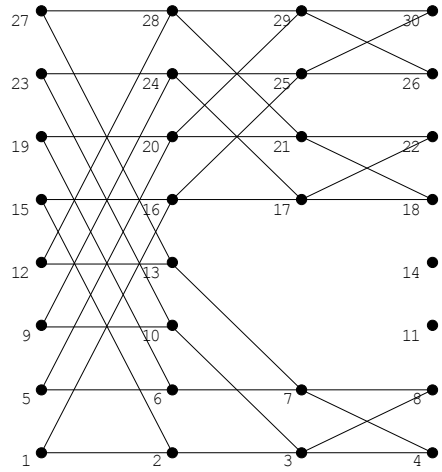
```
In[2]:= {{VertexConnectivity[Star[10]], VertexConnectivity[Star[10], Cut]},
         {VertexConnectivity[CompleteGraph[5, 6]], VertexConnectivity[CompleteGraph[5, 6], Cut]},
         {VertexConnectivity[CompleteGraph[10]], VertexConnectivity[CompleteGraph[10], Cut]}} //
        ColumnForm
Out[2]= {1, {10}}
        {5, {1, 2, 3, 4, 5}}
        {9, {1, 2, 3, 4, 5, 6, 7, 8, 9}}
```

Вычислим вершинную связность графа бабочки:

```
In[3]:= VertexConnectivity[g = ButterflyGraph[3]]
Out[3]= 2
```

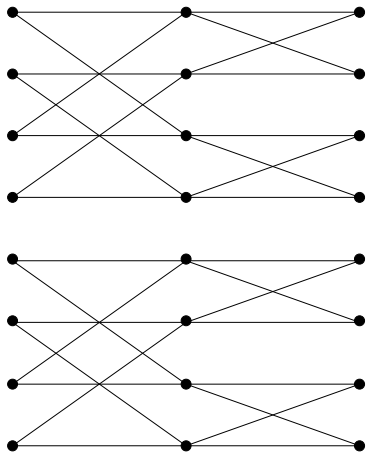
Удалим, например, 11-ю и 15-ю вершины:

```
In[4]:= ShowLabeledGraph[DeleteVertices[ButterflyGraph[3], {11, 15}]]
```



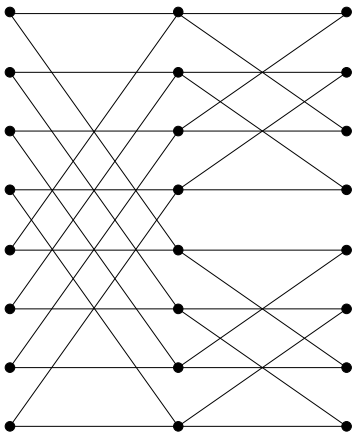

Граф бабочки имеет очень красивую рекурсивную структуру. Удалим теперь нулевой уровень вершин в r –мерном графе бабочки. Получим два r-1- мерных графа бабочки

```
In[5]:= ShowGraph[DeleteVertices[g = ButterflyGraph[3], Table[i, {i, 1, V[g], 4}]]]
```



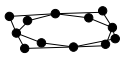
А теперь удалим уровень r вершин

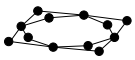
```
In[6]:= ShowGraph[q = DeleteVertices[g, Table[i, {i, 4, V[g], 4}]]]
```



Применим SpringEmbedding к графу q:

```
In[7]:= ShowGraph[SpringEmbedding[q, 200]]
```





Оказывается, граф q состоит из двух одинаковых связных компонент, каждый из которых есть r-1-мерный граф бабочки!
Действительно, граф, полученный удалением уровня r вершин из r-мерного графа бабочки, содержит две связные компоненты равного размера

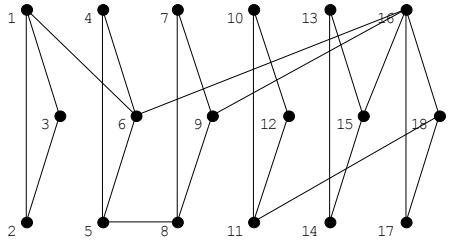
```
In[8]:= w = ConnectedComponents[q]
Out[8]= {{1, 2, 3, 7, 8, 9, 13, 14, 15, 19, 20, 21},
         {4, 5, 6, 10, 11, 12, 16, 17, 18, 22, 23, 24}}
```

Вершинная связность полного двудольного графа есть число вершин в наименьшей доле, так как чтобы сделать граф несвязным, все они должны быть удалены.

```
In[9]:= VertexConnectivity[CompleteGraph[9, 14]]
Out[9]= 9
```

Вершина $v_i$ графа g называется **точкой сочленения** графа g, если граф $(g\text{-}v_i)$ состоит из большего числа компонент, чем g. Любой граф, не имеющий точек сочленения, называется **двусвязным**. Двусвязность – весьма важное свойство по многим причинам. Например, если рассматривать сети связи, то двусвязные сети более терпимы к ошибкам, так как удаление одной вершины не нарушает связь. Двусвязные компоненты графа являются максимальными индуцированными подграфами, которые являются двусвязными. Легко видеть, что любая вершина, которая принадлежит более чем к одной двусвязной компоненте, является точкой сочленения.
Функция ArticulationVertices[g] возвращает список всех точек сочленения графа g.
BiconnectedQ[g] возвращает True, если g-двусвязный.
BiconnectedComponents[g] возвращает список двусвязных компонент графа g.
Рассмотрим шесть случайно соединенных шестью ребрами 3-цикла:

```
In[10]:= ShowLabeledGraph[
           g = AddEdges[GraphUnion[6, Cycle[3]],
             Table[RandomKSubset[Range[18], 2], {6}]]]
```




Проверим, является ли граф двусвязным, и выделим множество точек сочленения:

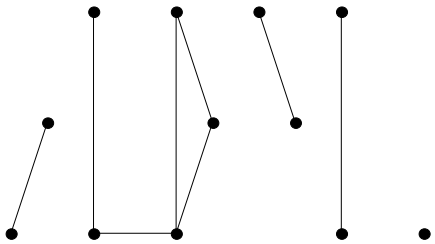
```
In[11]:= {BiconnectedQ[g], ArticulationVertices[g]}
Out[11]= {False, {1, 6, 11, 15, 16, 18}}
```

Выделим двусвязные компоненты графа:

```
In[12]:= BiconnectedComponents[g]
Out[12]= {{1, 2, 3}, {13, 14, 15}, {15, 16}, {10, 11, 12},
          {11, 18}, {16, 17, 18}, {4, 5, 6, 7, 8, 9, 16}, {1, 6}}
```

Удалим точки сочленения из графа:

```
In[13]:= ShowGraph[DeleteVertices[g, ArticulationVertices[g]]]
```



Граф называется k-связным, если $\chi(g) \geq k$. Таким образом, k-связный граф не содержит разделяющих множеств S мощности $|S| \leq k-1$. Если граф связный, его связность больше или равна 1. Следовательно, связные графы 1-связны. Если связный граф не содержит точек сочленения, то его вершинная связность больше 1. Поэтому связные графы без точек сочленения 2-связны. Следующая теорема дает оценку связности графа.

**Теорема.** Для простого связного графа g, имеющего m ребер и n вершин,
$\chi(g) \leq [2m/n]$ ([x]-целая часть числа x).

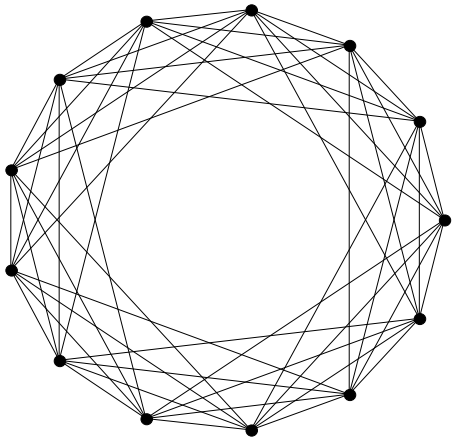
Харари доказал, что равенство в этом выражении достигается с помощью специальной процедуры построения k-связного графа $H_{k,n}$, который содержит точно [kn/2] ребер. Функция Harary[k, n] конструирует минимальный k-связный граф на n вершинах.

Покажем, что цикл - минимальный двусвязный граф.

```
In[14]:= IdenticalQ[Cycle[10], Harary[2, 10]]
Out[14]= True
```

Когда n или k четны, граф Харари есть циркулянт - граф, следовательно, он регулярный.

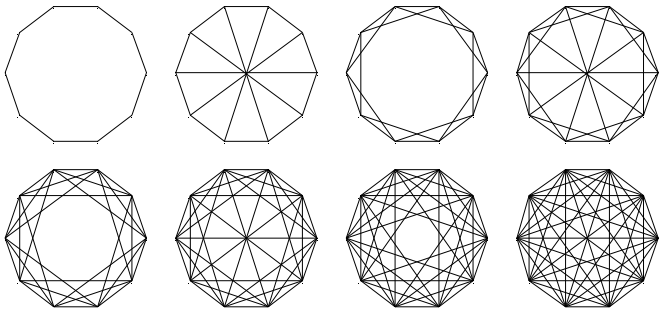
```
In[9]:= ShowGraph[p = Harary[8, 13]]
```



```
In[16]:= RegularQ[p]
Out[16]= True
```

Приведем таблицу 10-вершинных графов Харари. Если связность возрастает, число ребер в графе возрастает.

```
In[17]:= ShowGraphArray[Partition[Table[Harary[k, 10], {k, 2, 9}], 4]]
```



Вычислим вершинную связность этих графов:

```
In[18]:= Table[VertexConnectivity[Harary[k, 10]], {k, 2, 9}]
Out[18]= {2, 3, 4, 5, 6, 7, 8, 9}
```

| | |
|---|---|
| VertexConnectivity[g] | возвращает вершинную связность графа g |
| VertexConnectivity[g, Cut] | возвращает минимальное разделяющее множество графа g |
| ArticulationVertices[g] | возвращает список всех точек сочленения графа g |
| BiconnectedQ[g] | возвращает True, если g-двусвязный граф и False в противном случае |
| BiconnectedComponents[g] | возвращает список двусвязных компонент графа g |
| Harary[k, n] | конструирует минимальный k-связный граф на n вершинах |

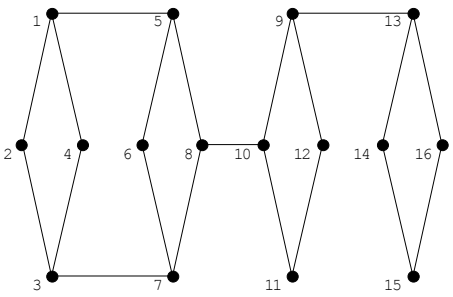
### *6.3.2. Реберная связность*

Реберная связность $\chi_1$(g) графа g – это минимальное число ребер, удаление которых из графа приводит к несвязному или тривиальному графу. Мостом графа g называется такое ребро e, что (g-e) имеет больше компонент, чем g.
EdgeConnectivity[g] возвращает реберную связность графа g.
EdgeConnectivity[g, Cut] возвращает минимальное множество ребер, удаление которых из графа приводит к несвязному или тривиальному графу.
Bridges[g] возвращает список мостов графа g.
Применим эти функции для десятивершинной звезды, полного двудольного графа и полного графа.

```
In[2]:= {{EdgeConnectivity[Star[10]], EdgeConnectivity[Star[10], Cut]},
         {EdgeConnectivity[CompleteGraph[5, 6]],
          EdgeConnectivity[CompleteGraph[5, 6], Cut]},
         {EdgeConnectivity[CompleteGraph[10]], EdgeConnectivity[CompleteGraph[10], Cut]}} //
        ColumnForm

Out[2]= {1, {{1, 10}}}
        {5, {{1, 6}, {2, 6}, {3, 6}, {4, 6}, {5, 6}}}
        {9, {{1, 2}, {1, 3}, {1, 4}, {1, 5}, {1, 6}, {1, 7}, {1, 8}, {1, 9}, {1, 10}}}
```

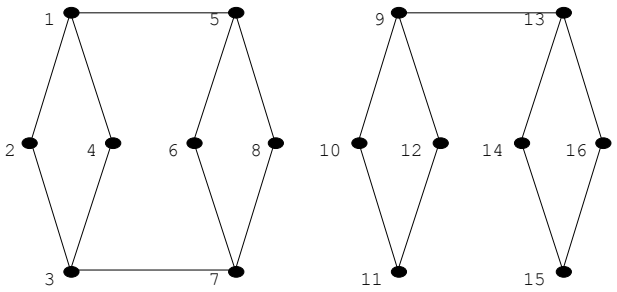
Построим граф u как объединение четырех 4-циклов, с несколькими добавленными ребрами

```
In[3]:= ShowLabeledGraph[
         u = AddEdges[GraphUnion[4, Cycle[4]], {{3, 7}, {1, 5}, {8, 10}, {9, 13}}]]
```




Теперь удалим ребра, содержащиеся в EdgeConnectivity[u,Cut]

```
In[4]:= ShowLabeledGraph[DeleteEdges[u, EdgeConnectivity[u, Cut]]]
```




Подсчитаем число мостов графа u:

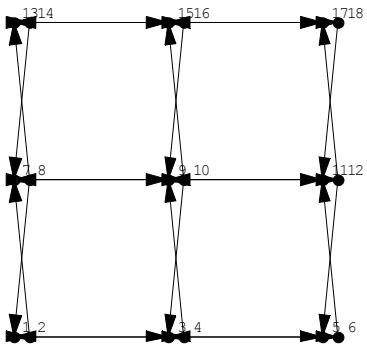
```
In[5]:= Bridges[u]
Out[5]= {{9, 13}, {8, 10}}
```

Граф g называется **k-реберно-связным**, если $\chi_1(g) \geq k$.Таким образом, чтобы сделать несвязным k-реберно-связный граф, необходимо удалить хотя бы k ребер.
Теорема Менгера помогает соотнести связность графа с числом вершинно-непересекающихся путей.
**Теорема** (Менгера). Минимальное число вершин, удаление которых из графа разделяет две несмежные вершины s и t, равно максимальному числу вершинно-непересекающихся s-t путей графа.

**Теорема.** Чтобы простой граф g=(E,V) был k-связным, необходимо и достаточно, чтобы между любыми вершинами s и t в графе g проходило k вершинно-непересекающихся s-t путей графа.

Этот результат получен Уитни. Поскольку он является попросту вариацией теоремы Менгера, ее также называют теоремой Менгера.
Функция VertexConnectivityGraph[g] возвращает ориентированный граф, чьи ребра соответствуют вершинам графа g, а реберно-непересекающиеся пути соответствуют вершинно-непересекающимся путям в g.

```
In[6]:= g = GridGraph[3, 3]; ShowGraph[h = VertexConnectivityGraph[g], VertexNumber → True,
          VertexNumberPosition → UpperRight]
```




| | |
|---|---|
| EdgeConnectivity[g] | возвращает реберную связность графа g |
| EdgeConnectivity[g, Cut] | возвращает минимальное множество ребер, удаление которых из графа приводит к несвязному или тривиальному графу |
| Bridges[g] | возвращает список мостов графа g |

| | |
|---|---|
| VertexConnectivityGraph[g] | возвращает ориентированный граф, чьи ребра соответствуют вершинам графа g, а реберно-непересекающиеся пути соответствуют вершинно-непересекающимся путям в g |

### *6.3.3. Связность ориентированных графов*

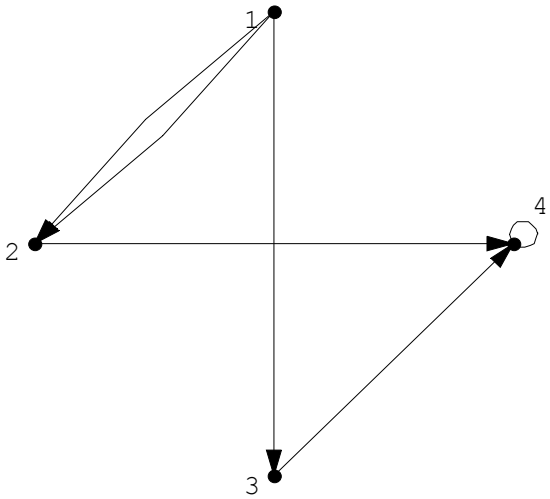
Начнем с введения нескольких основных определений и понятий, относящихся к ориентированным графам.

**Ориентированный граф** g=(E,V) состоит из двух множеств: конечного множества V, элементы которого называются вершинами, и конечного множества E, элементы которого называются ребрами или дугами. Каждое ребро связано упорядоченной парой вершин.

Для обозначения вершин используются символы $v_1,v_2$, …, а для обозначения ребер – символы $e_1,e_2$…. Если $e_i=(v_i, v_j)$, то $v_i$, $v_j$ называются **концевыми** вершинами $e_i$, при этом $v_i$ – начальная вершина, а $v_j$ – конечная вершина $e_i$. Все ребра, имеющие одну пару начальных и конечных вершин, называются **параллельными** или **кратными**. Ребро называется **петлей**, если инцидентная ему вершина $v_i$ является одновременно начальной и конечной ее вершиной.

В графическом представлении ориентированного графа вершины изображаются точками, а ребра (дуги) – отрезками линий, соединяющими точки, представляющие их концевые вершины. Кроме того, ребрам присваивается ориентация, показываемая стрелкой, направленной от начальной вершины к конечной.

```
In[2]:= ShowLabeledGraph[
          p = SetGraphOptions[FromOrderedPairs[{{1, 2}, {1, 2}, {1, 3}, {3, 4}, {4, 4}, {2, 4}}],
            {4, VertexNumberPosition → {0.07, 0.07}}], PlotRange → 0.025,
          TextStyle → {FontSize → 14}]
```




На рисунке {1,2} и {1,2} параллельные ребра, {4,4}- петля.

Говорят, что ребро **инцидентно** своим концевым вершинам. Вершины называются **смежными**, если, если они являются концевыми для одного ребра. Если ребра имеют общую концевую вершину, то они называются **смежными**.

Ребро называется **исходящим** из своей начальной вершины и **заходящим** в свою конечную вершину. Вершина называется **изолированной**, если она не имеет инцидентных ребер.

**Степенью** $d(v_j)$ вершины $v_j$ называется число инцидентных ей ребер. **Полустепенью захода** $^{-}d^{-}(v_j)$ вершины $v_j$ называется число заходящих в $v_j$ ребер, а **полустепенью исхода** $d^{+}(v_j)$ – число исходящих из $v_j$ ребер.

Функция InDegree[g, n] возвращает полустепень захода вершины n в ориентированном графе g. InDegree[g] возвращает последовательность полустепеней захода вершин в ориентированном графе g. OutDegree[g, n] возвращает полустепень исхода вершины n в ориентированном графе g.

OutDegree[g] возвращает последовательность полустепеней исхода вершин в ориентированном графе g.

```
In[3]:= {InDegree[p, 3], OutDegree[p, 3]}
Out[3]= {1, 1}
```

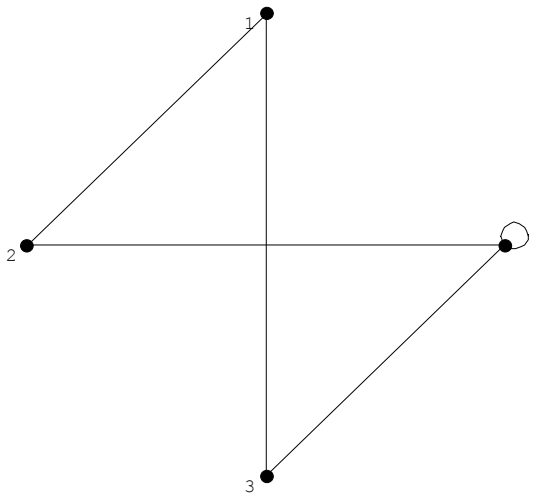
Заметим, что петля увеличивает полустепени как захода, так и исхода этой вершины.
Подграфы и порожденные подграфы ориентированного графа определяются так же, как и в случае неориентированных графов. Неориентированный граф, получающийся в результате снятия ориентации с ребер ориентированного графа g, называется **лежащим в основе неориентированным графом g.**

**Ориентированным маршрутом** ориентированного графа g=(E,V) называется такая конечная последовательность вершин $v_0,v_1,\ldots,v_k$, что $(v_{i-1},v_i)$, $1\le i\le k$, является дугой графа. Такой маршрут обычно называется **ориентированным $v_0$-$v_k$ – маршрутом**, причем $v_0$ – начальная вершина, $v_k$ – конечная вершина маршрута, а все другие вершины – внутренние. Начальная и конечная вершины ориентированного маршрута называются его концевыми вершинами. Отметим, что ребра, а следовательно, и вершины могут появляться в ориентированном маршруте более одного раза. Ориентированный маршрут называется **открытым**, если его концевые вершины различны, в противном случае – замкнутым. Ориентированный маршрут называется **ориентированной цепью**, если все его дуги различны. Ориентированная цепь называется **открытой,** если все ее концевые вершины различны, в противном случае – **замкнутой**. Открытая ориентированная цепь называется **ориентированным путем**, если различны все ее вершины.

Замкнутая ориентированная цепь называется ориентированным циклом, если все ее вершины, за исключением концевых вершин, различны.

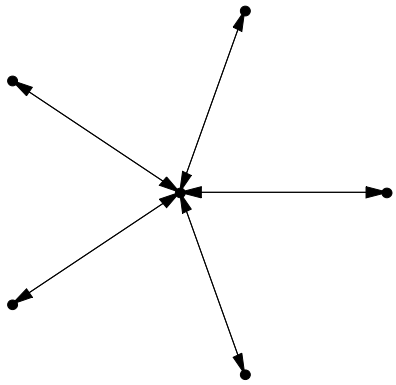
Встроенная функция MakeUndirected[g] выдает лежащий в основе неориентированный граф данного ориентированного графа g.

```
In[4]:= ShowLabeledGraph[MakeUndirected[p]]
```



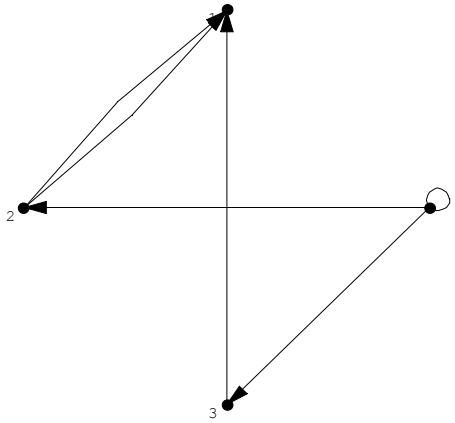

Функция MakeDirected[g] конструирует ориентированный граф из данного неориентированного графа g, придавая ребрам два противоположных направления. Локальные опции ребер не наследуются из графа g. MakeDirected[g, All] обеспечивает наследование обоими противоположно направленными ребрами локальных опций соответствующих ребер графа g.

```
In[5]:= ShowGraph[MakeDirected[Star[6]]]
```



С помощью функции ReverseEdges[g] можно поменять направления ребер на противоположные.

```
In[6]:= ShowLabeledGraph[ReverseEdges[p]]
```




Кроме того, как уже упоминалось, ориентировать граф можно, установив опцию Type->Directed при конструировании графов с помощью функций FromAdjacencyLists, FromAdjacencyMatrix, FromOrderedPairs, FromUnorderedPairs, FunctionalGraph, Graph, MakeGraph или при конструировании специальных графов.

Ориентированный граф называется связным, если связным является лежащий в его основе неориентированный граф. Подграф ориентированного графа g называется компонентой графа g, если он является компонентой лежащего в его основе неориентированного графа.

Вершины $v_i$ и $v_j$ ориентированного графа g называются **сильно связными**, если в g существуют ориентированные пути из $v_i$ в $v_j$ и обратно. Если $v_i$ сильно связна с $v_j$, то, очевидно, и $v_j$ сильно связна с $v_i$. Всякая вершина сильно связна сама с собой.

Если вершина $v_i$ сильно связна с вершиной $v_j$, а $v_j$ с $v_k$, то, как легко видеть, вершина $v_i$ сильно связна с вершиной $v_k$.Следовательно, в этом случае просто говорят, что вершины $v_j$, $v_j$, $v_k$ сильно связны.

Ориентированный граф называется сильно связным, если сильно связны все его вершины. Максимальный сильно связный подграф ориентированного графа g называется сильно связной компонентой графа g. Если граф не сильно связен, но лежащий в основе неориентированный граф связен, то граф называется слабо связным.

Функция ConnectedQ[g] возвращает True, если неориентированный граф g связен, если граф g ориентированный, то функция выдает True, если лежащий в основе неориентированный граф связен. ConnectedQ[g, Strong] и ConnectedQ[g, Weak] возвращает True, если ориентированный граф g строго или слабо связный соответственно.

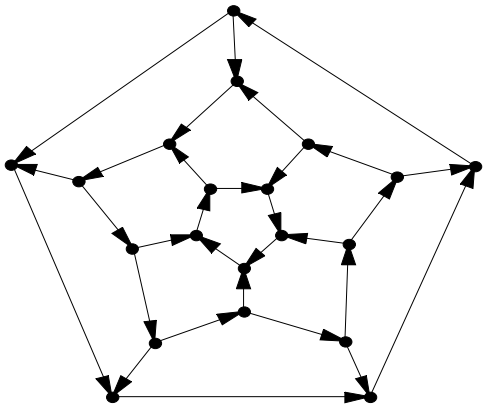
```
In[7]:= {ConnectedQ[p, Strong], ConnectedQ[p, Weak], ConnectedQ[p]}
Out[7]= {False, True, True}
```

Функция StronglyConnectedComponents[g] возвращает сильно связные компоненты ориентированного графа g как списки вершин. Аналогично WeaklyConnectedComponents[g] выдает в виде списков вершин слабо связные компоненты ориентированного графа g.

```
In[8]:= {StronglyConnectedComponents[
           h = GraphJoin[Cycle[3, Type → Directed], Cycle[5, Type → Directed]]],
         WeaklyConnectedComponents[h]} // ColumnForm
Out[8]= {{1, 2, 3}, {4, 5, 6, 7, 8}}
        {{1, 2, 3, 4, 5, 6, 7, 8}}
```

Задать направления ребрам графа можно, вызвав функцию OrientGraph.
OrientGraph[g] присваивает ровно одно направление каждому ребру неориентированного, не имеющего мостов графа g, так, что полученный граф строго связан.
Рассмотрим ориентированный граф додекаэдра. Нас будет интересовать, будет ли полученный с помощью OrientGraph, ориентированный граф додекаэдра строго связным.

```
In[9]:= ShowGraph[q = OrientGraph[DodecahedralGraph]]
```



```
In[10]:= {ConnectedQ[q, Strong], StronglyConnectedComponents[q]}
Out[10]= {True, {{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}}}
```

| | |
|---|---|
| InDegree[g, n] | возвращает полустепень захода вершины n в ориентированном графе g |
| InDegree[g] | возвращает последовательность полустепеней захода вершин в ориентированном графе g |
| OutDegree[g, n] | возвращает полустепень исхода вершины n в ориентированном графе g |
| OutDegree[g] | возвращает последовательность полустепеней исхода вершин в ориентированном графе g |
| MakeUndirected[g] | возвращает лежащий в основе неориентированный граф данного ориентированного графа g |
| MakeDirected[g] | конструирует ориентированный граф из данного неориентированного графа g, придавая ребрам два противоположных направления |
| MakeDirected[g, All] | обеспечивает наследование обоими противоположно направленными ребрами локальных опций соответствующих ребер графа g |
| ReverseEdges[g] | Возвращает граф, ребра которого имеют направление, противоположное направлению соответствующих ребер ориентированного графа g |

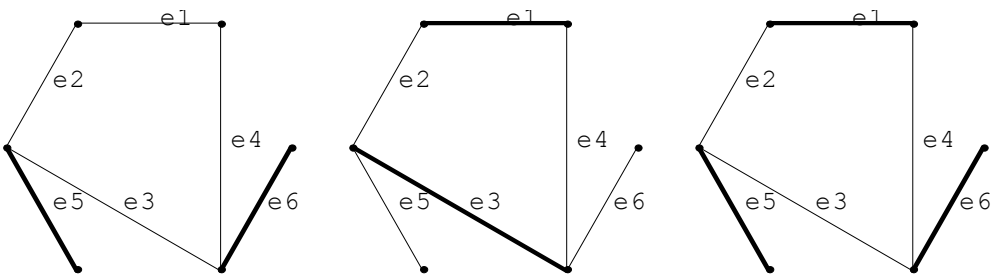
| | |
|---|---|
| ConnectedQ[g] | возвращает True, если неориентированный граф g связен, если граф g ориентированный, то функция выдает True, если лежащий в основе неориентированный граф связен |
| ConnectedQ[g, Strong] | возвращает True, если ориентированный граф g строго связен |
| ConnectedQ[g, Weak] | возвращает True, если ориентированный граф g слабо связен |
| StronglyConnectedComponents[g] | возвращает сильно связные компоненты ориентированного графа g как списки вершин |
| WeaklyConnectedComponents[g] | выдает в виде списков вершин слабо связные компоненты ориентированного графа g |
| OrientGraph[g] | присваивает ровно одно направление каждому ребру неориентированного, не имеющего мостов графа g, так, что полученный граф строго связан |

## ■6.4. Паросочетания

### *6.4.1. Совершенные, максимальные и наибольшие паросочетания*

**Паросочетанием** в графе G называется подмножество ребер в G, никакие два из которых не имеют общей вершины. Ясно, что каждое паросочетание состоит по крайней мере из n/2 ребер.
Паросочетание, содержащее ровно n/2 ребер, называется **совершенным.**
Не все графы имеют совершенные паросочетания, но каждый граф имеет наибольшее паросочетание (maximum).
Паросочетание называется **максимальным**, если оно не содержится в паросочетании с большим числом ребер.
Максимальные паросочетания легко вычислить, если выбирать ребро, непересекающееся с уже выбранными, до тех пор, пока это возможно.
Паросочетание с наибольшим числом ребер называется **наибольшим паросочетанием**.
Вершина называется **насыщенной** в паросочетании M, если она концевая вершина ребра из M.
Ясно, что совершенное паросочетание насыщает все вершины графа g.
Максимальное паросочетание не обязательно наибольшее, граф может иметь максимальные паросочетания различных размеров.

```
In[2]:= p = FromUnorderedPairs[{{1, 2}, {2, 3}, {3, 4}, {3, 5}, {5, 1}, {5, 6}}];
        ShowGraphArray[{Highlight[p, s = {{{3, 4}, {5, 6}}}], Highlight[p, t = {{{1, 2}, {3, 5}}}],
          Highlight[p, {MaximalMatching[p]}]}, TextStyle → {FontSize → 11},
         EdgeLabel → {e1, e4, e2, e5, e3, e6}, EdgeLabelPosition → UpperRight]
```



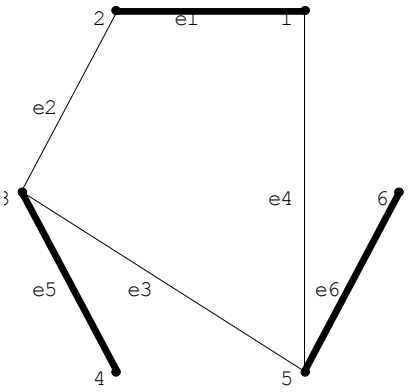

На первом рисунке темным цветом выделены ребра {e5, e6}, которые образуют паросочетание, но оно не является ни максимальным, ни наибольшим; на втором в этом же графе пара ребер {e1, e3} образует максимальное, но не наибольшее паросочетание; на третьем рисунке выделенные

ребра {e1, e5, e6} – максимальное, наибольшее и совершенное паросочетание. На всех трех рисунках, вершина, из которой выходят ребра e3, e4, e6 - насыщенная.
Встроенная функция MaximalMatching[g] возвращает список ребер максимального паросочетания графа g.
Рассмотрим максимальное паросочетание графа p:

```
In[3]:= MaximalMatching[p]
Out[3]= {{1, 2}, {3, 4}, {5, 6}}
```

Выделим это паросочетание:

```
In[4]:= ShowLabeledGraph[Highlight[p, {MaximalMatching[p]}], EdgeLabel → {e1, e4, e2, e5, e3, e6},
         TextStyle → {FontSize → 12}]
```

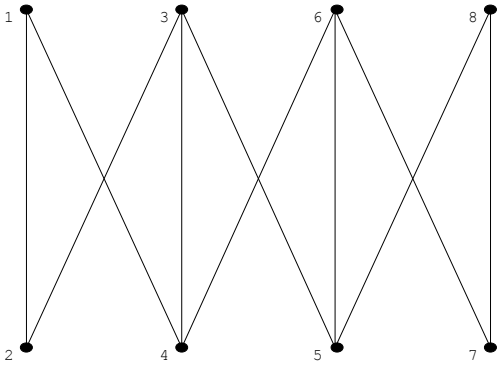



Пусть дан граф g=(E,V) и его паросочетание M. **Чередующаяся цепь** графа g - это цепь, ребра которой входят поочередно в M и (E - M). Те ребра, которые принадлежат M, будем называть темными ребрами, а те, что принадлежат (E - M) - светлыми ребрами.
Например, {e1, e2, e5} – чередующаяся цепь графа p по отношению к паросочетанию {e1, e5, e6}, e1, e5 – темные ребра, e2 – светлое ребро.
Пусть M - паросочетание графа g, а P - чередующаяся цепь между произвольными двумя не насыщенными в M вершинами. Тогда M ⊕ P является паросочетанием с числом ребер на 1 больше, чем в M. Путь P называется **добавляющим** путем по отношению к M.
В следующей теореме формулируется характеризация Бержа максимальных паросочетаний.

**Теорема (Берж).** Паросочетание максимально тогда и только тогда, когда между любыми не насыщенными в нем вершинами не существует чередующийся цепи.

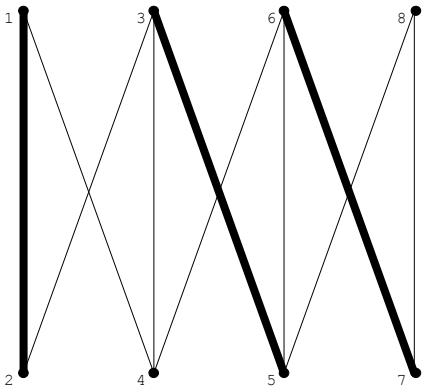
Рассмотрим граф:

```
In[5]:= v11 = {{-1, 1}, {-1, -1}, {0, 1}, {0, -1}, {1, -1}, {1, 1}, {2, -1}, {2, 1}};
        ShowLabeledGraph[
         g = FromAdjacencyLists[{{2, 4}, {3}, {4, 5}, {3, 6}, {6, 8}, {7}, {8}, {7}}, v11]]
```



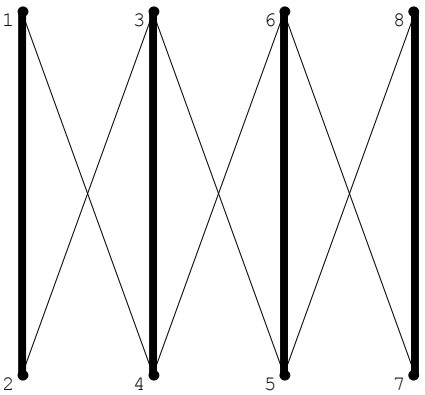

В нем паросочетаниями являются, например, {e1}={{1,2}}, {e2}={{2,3}},{e2,e6}={{2,3},{6,7}}, {e1,e3,e6}={{1,2},{3,4},{6,7}},
{e1,e3,e5,e7}={{1,2},{3,4},{5,6},{7,8}}
Выделим паросочетание s={{2,1},{3,5},{6,7}}

```
In[7]:= s = {{2, 1}, {3, 5}, {6, 7}}; ShowGraph[Highlight[g, {s}], VertexNumber → True]
```




Ясно, что это паросочетание максимально, но несовершенно. Теперь рассмотрим последовательность вершин 4, 3, 5, 8, 7, проходящую через вершины, изменяя каждое светлое ребро на темное ребро. В результате два темных ребра заменяются тремя светлыми ребрами, все темные ребра образуют совершенное паросочетание.

```
In[8]:= ShowGraph[Highlight[g, {MaximalMatching[g]}], VertexNumber → True]
```

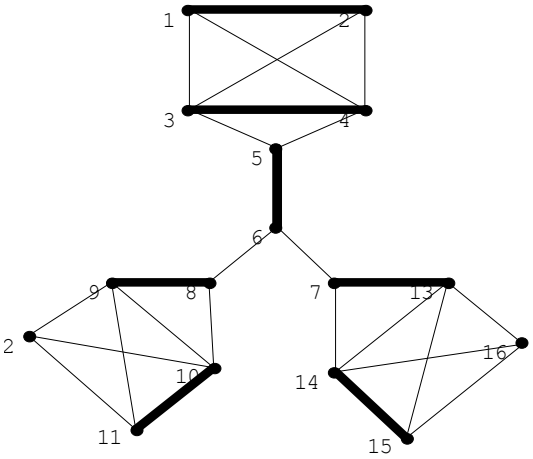



Теорема Татта дает условие существования совершенного паросочетания графа.

**Теорема (Татта).** Граф g={V,E} имеет совершенное паросочетание тогда и только тогда, когда $p_0(S) \le |S|$ для любого $S \subset V$. Здесь $p_0(S)$ – число компонент графа (g - S), содержащих нечетное число вершин.

В Combinatorica функция NoPerfectMatchingGraph возвращает связный граф на 16 вершинах, который не содержит совершенного паросочетания

```
In[9]:= ShowLabeledGraph[Highlight[p = NoPerfectMatchingGraph, {MaximalMatching[p]}]]
```




Максимальное паросочетание, выделенное здесь, является наибольшим. Любое большее паросочетание будет совершенным, но таковое не существует, это следует из того, что условие теоремы Татта не выполняется для S={6}. Действительно, при удалении шестой вершины получим три компоненты из 5 вершин.

```
In[10]:= Map[Length, ConnectedComponents[DeleteVertices[p, {6}]]]
Out[10]= {5, 5, 5}
```

| | |
|---|---|
| MaximalMatching[g] | возвращает список ребер максимального паросочетания графа g |
| NoPerfectMatchingGraph | возвращает связный граф на 16 вершинах, который не содержит совершенного паросочетания |

### *6.4.2. Паросочетания в двудольных графах*

Изучение паросочетаний в графах возникает из задачи о свадьбах. Имеется конечное множество юношей, у каждого из которых есть несколько подруг. При каких условиях можно поженить юношей так, чтобы каждый женился на одной из своих подруг? (При этом девушка выходит замуж только за одного юношу).

Будем говорить, что множество X паросочетается с Y в двудольном графе g= (E, X, Y) , если существует такое паросочетание M, что каждая вершина X насыщена в M. Паросочетание M в этом случае называется полным паросочетанием X с Y. Задача о свадьбах в теоретико-графовых терминах формулируется следующим образом: существует ли полное паросочетание в двудольном графе g=(E, X, Y)?

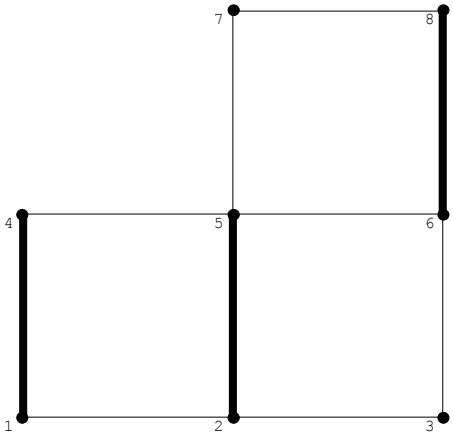
Ответ на этот вопрос дает теорема Холла:

**Теорема (Холл).** В двудольном графе G=(E,X,Y) существует полное паросочетание X с Y тогда и только тогда, когда для любого $S \subseteq X$ справедливо неравенство $|S| \le \Gamma(S)|$, где $\Gamma(S)$ - множество вершин, смежных с S.

Таким образом, для решения задачи о свадьбах необходимо и достаточно, чтобы любое подмножество из k юношей имело вместе не менее k подруг.

**Пример**. Паросочетание, выделенное в этом графе, является максимальным, но не наибольшим. Последовательность {7, 5, 2, 3} является добавляющим путем, который может улучшить паросочетание.

```
In[2]:= g = DeleteVertices[GridGraph[3, 3], {7}];
        m = {{1, 4}, {2, 5}, {6, 8}};
        ShowGraph[Highlight[g, {m}], VertexNumber → True]
```



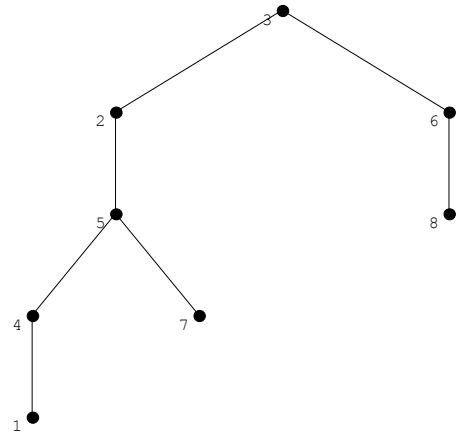

Функция AlternatingPaths[g, start, ME] возвращает добавляющие пути в графе g по отношению к паросочетанию ME, начинающиеся в вершинах списка start.
Пути возвращаются в форме леса с корневыми вершинами в start.
Вычислим дерево альтернативных путей с корнем в вершине 3. Дерево представлено родительскими точками - так, родитель вершины 1 - вершина 4, родитель вершины 2 - 3 и т.д. Путь {7, 5 ,2 ,3} представлен, т.к. 7 указывает на 5, 5 – на 2 и 2 – на 3. Это добавляющий путь, потому что две концевые точки ненасыщенны.

```
In[4]:= d = AlternatingPaths[g, {3}, m]
Out[4]= {4, 3, 3, 5, 2, 3, 5, 6}
```

Функция ParentsToPaths[l,i,j] берет список родителей l и возвращает путь из i в j. ParentsToPaths[l, i] возвращает пути из I ко всем вершинам.

```
In[5]:= ParentsToPaths[d, 3, 7]
Out[5]= {3, 2, 5, 7}
```

Рассмотрим полученное дерево d:



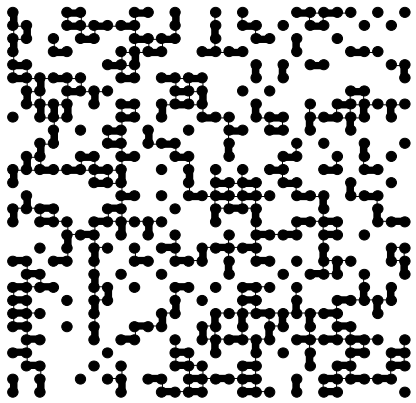

Это дерево показывает возможные добавляющие пути из вершины 3:
{3,2,5,4,1}, {3,2,5,7}, {3,6,8}

Функция BipartiteMatching[g] выдает наибольшее паросочетание в двудольном графе g. Если граф взвешенный, то функция возвращает паросочетание с наибольшим общим весом.
Решетчатый граф – двудольный и поэтому двудольным являются любой порожденный подграф. Рассмотрим наибольшее паросочетание в случайно порожденном 30×30-решетчатом графе:

```
In[6]:= g = InduceSubgraph[GridGraph[30, 30], RandomSubset[900]]; m = BipartiteMatching[g];
        p = FromUnorderedPairs[m];
        {BipartiteQ[GridGraph[30, 30]], BipartiteQ[g], BipartiteQ[p]}
Out[6]= {True, True, True}
```

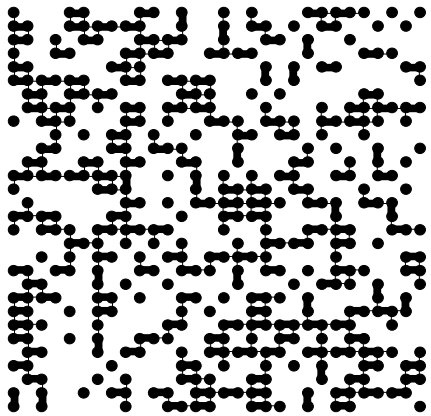
Выделим наибольшее паросочетание m. По отношению к m существует много ненасыщенных вершин, но точно нет добавляющих путей.

```
In[7]:= ShowGraph[Highlight[g, {m}]];
```



Теперь рассмотрим максимальное паросочетание того же графа. Оно не обязательно наибольшее, то есть могут быть добавляющие пути по отношению к m.

```
In[8]:= mm = MaximalMatching[g]; ShowGraph[Highlight[g, {mm}]]
```



По определению, не существовует паросочетания, имеющего больше ребер, чем наибольшее паросочетание.

```
In[9]:= {Length[mm], Length[m]}
Out[9]= {169, 185}
```

| | |
|---|---|
| AlternatingPaths[g, start, ME] | возвращает добавляющие пути в графе g по отношению к паросочетанию ME, начинающиеся в вершинах списка start |

| | |
|---|---|
| ParentsToPaths[l,i,j] | берет список родителей l и возвращает путь из i в j |
| ParentsToPaths[l, i] | возвращает пути из I ко всем вершинам |
| BipartiteMatching[g] | выдает наибольшее паросочетание в двудольном графе g. Если граф взвешенный, то функция возвращает паросочетание с наибольшим общим весом |

### *6.4.3. Задача устойчивого марьяжа*

Не все задачи паросочетаний естественно описываются в терминах теории графов. Наиболее интересный пример тому – задача устойчивых свадеб.
Дано множество из n юношей и n девушек. Как и в реальном мире, каждый юноша знает каждую девушку и ранжирует девушек от 1 до n (первая девушка ему нравится более всех, из всех оставшихся более всех - вторая и т.д.). Каждая девушка также ранжирует юношей. Рассмотрим пары $(m_1, w_1)$ и $(m_2, w_2)$. Если $m_1$ более предпочитает $w_2$, чем $w_1$, а $w_2$ предпочитает $m_1$ юноше $m_2$, то брак между этими парами называется неустойчивым. Цель задачи устойчивого марьяжа - найти способ сочетать девушек и юношей так, чтобы среди полученных пар не было таких, брак между которыми неустойчив. Очевидно, устойчивость - желаемое свойство, но всегда ли это достижимо? Известно, что для любого множества функций предпочтения устойчивый марьяж существует.
Функции предпочтения каждого юноши задается перестановкой из первых n натуральных чисел (т.е. номер самой желаемой девушки на первом месте и т.д.). Аналогично задаются функции предпочтения девушек.
Функция StableMarriage[mpref, fpref] возвращает мужской устойчивый марьяж, где mpref, fpref – функции предпочтения юношей и девушек соответственно.
Сформируем функции предпочтения юношей и девушек:

```
In[2]:= {boys = Table[RandomPermutation[7], {7}], girls = Table[RandomPermutation[7], {7}]} //
        TableForm
```

```
Out[2]//TableForm=
2  1  2  6  5  6  4
1  6  4  5  3  5  5
3  7  5  7  1  7  6
5  5  3  4  4  3  3
7  3  1  2  6  2  1
6  2  6  3  7  1  2
4  4  7  1  2  4  7
7  7  6  4  4  7  2
5  1  1  7  7  3  1
1  3  7  6  2  6  6
2  5  4  2  5  4  5
6  4  2  3  3  5  7
3  2  3  1  1  2  4
4  6  5  5  6  1  3
```

```
In[3]:= StableMarriage[boys, girls]
Out[3]= {2, 7, 3, 5, 1, 6, 4}
```

| | |
|---|---|
| StableMarriage[mpref, fpref] | возвращает мужской устойчивый марьяж, где mpref, fpref – функции предпочтения юношей и девушек соответственно |

# ■6.5. Раскраска графов

## *6.5.1. Независимые множества*

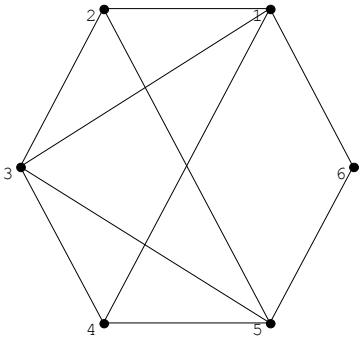
Рассмотрим граф g=(E,V). Подмножество S⊆V называется **независимым множеством** графа g, если никакие две вершины в S не смежны в графе g. Другими словами, если подмножество вершин графа g является независимым в g, то соответствующий вершинно-порожденный подграф является пустым.

Независимое множество S графа g максимально, если граф g не содержит такого независимого множества $S_1$, что $|S_1|>|S|$. Число вершин в максимальном независимом множестве графа g называется **числом независимости** графа g и обозначается $\alpha(g)$.

Функция IndependentSetQ[g,s] возвращает True, если вершины списка s определяют независимое множество графа g.

Рассмотрим граф g:

```
In[2]:= g = AddEdges[Cycle[6], {{1, 4}, {1, 3}, {3, 5}, {2, 5}}]; ShowLabeledGraph[g]
```




Множества {2,6}, {4,6}, {1,5}, {2,4,6} являются независимыми множествами.

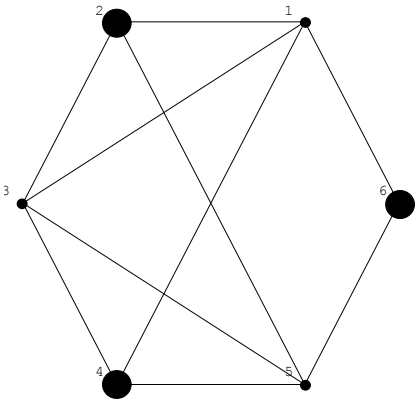
```
In[3]:= Map[IndependentSetQ[g, #] &, {{2, 6}, {4, 6}, {1, 5}, {2, 4, 6}}]
Out[3]= {True, True, True, True}
```

В вышеприведенном графе {2,6}, {4,6} – не наибольшие независимые множества; {1,5} – наибольшее, но не максимальное; {2,4,6} – максимальное.

Функция MaximumIndependentSet[g] находит максимальное независимое множество графа g.

В нижеприведенном графе выделено максимальное независимое множество:

```
In[4]:= ShowLabeledGraph[Highlight[g, MaximumIndependentSet[g]],
        VertexNumberPosition → {-0.03, 0.03}]
```



| | |
|---|---|
| IndependentSetQ[g,s] | возвращает True, если вершины списка s определяют независимое множество графа g |
| MaximumIndependentSet[g] | находит максимальное независимое множество графа g |

### *6.5.2. Вершинные покрытия*

Подмножество К множества вершин V называется **вершинным покрытием**, если любое ребро графа g имеет хотя бы одну концевую вершину в подмножестве К. Если рассматривать вершину как покрывающую все инцидентные им ребра, тогда вершинное покрытие графа g есть подмножество V, которое покрывает все ребра графа g.
Вершинное покрытие К графа g минимально, если граф g не имеет такого вершинного покрытия $K_1$, что $|K_1| < |K|$. Число вершин в минимальном вершинном покрытии графа g называется числом вершинного покрытия графа g и обозначается $\beta(g)$.
Тест VertexCoverQ[g, c] возвращает True, если вершины списка c определяют вершинное покрытие графа.
Например, в вышеприведенном графе {1,3,4,5}, {1,2,3,5}, {2, 3,4,6} и {1,3,5} являются вершинными покрытиями.

```
In[2]:= g = AddEdges[Cycle[6], {{1, 4}, {1, 3}, {3, 5}, {2, 5}}];
        Map[VertexCoverQ[g, #] &, {{1, 3, 4, 5}, {1, 2, 3, 5}, {2, 3, 4, 6}, {1, 3, 5}}]
Out[2]= {True, True, True, True}
```

Из них {1, 3, 4, 5}, {1, 2, 3, 5} – не наименьшие; {2, 3, 4, 6} – наименьшее, но не минимальное; {1, 3, 5} – минимальное.

Функция VertexCover[g] возвращает вершинное покрытие графа g. При вычислении вершинного покрытия могут использоваться различные алгоритмы, в зависимости от установки опции Algorithm, которая может принимать значения Greedy, Approximate или Optimum. По умолчанию Algorithm -> Approximate.
Задача поиска минимального вершинного покрытия NP–полная, но она допускает аппроксимационный алгоритм.
ApproximateVertexCover[g] производит вершинное покрытие графа g.
Размер этого вершинного покрытия может быть улучшен, используя жадный алгоритм. Возможно, что жадная эвристика не улучшит вершинного покрытия, но она не может его ухудшить.
GreedyVertexCover[g] возвращает вершинное покрытие графа g, построенное с помощью жадного алгоритма.
Как и аппромаксимационный алгоритм, жадный алгоритм не гарантирует выполнения задачи поиска вершинного покрытия, но иногда улучшает решение

```
In[3]:= g = MeredithGraph; Map[Length, {GreedyVertexCover[g], ApproximateVertexCover[g]}]
Out[3]= {39, 50}
```

… а иногда нет

```
In[4]:= g = GridGraph[10, 10, 10];
        {Length[GreedyVertexCover[g]], Length[ApproximateVertexCover[g]]}
Out[4]= {500, 500}
```

Жадный алгоритм за разумное время работает даже на стовершинных графах

```
In[5]:= Timing[GreedyVertexCover[RandomGraph[100, 0.5]]]
```

```
Out[5]= {5.297 Second, {18, 28, 94, 66, 88, 41, 46, 29, 75, 100, 7, 6, 45, 56, 93, 42, 86, 39, 32, 83,
         69, 51, 47, 43, 78, 97, 13, 2, 20, 76, 10, 48, 44, 63, 9, 95, 5, 27, 68, 60, 50, 99, 36, 70,
         25, 55, 92, 74, 19, 33, 85, 80, 98, 14, 96, 4, 15, 21, 17, 59, 58, 72, 40, 77, 30, 65, 53, 3,
         89, 12, 49, 23, 52, 84, 79, 24, 54, 57, 62, 87, 71, 81, 37, 16, 31, 91, 64, 73, 82, 22, 35}}
```

Жадный алгоритм также лучше работает на случайных деревьях:

```
In[6]:= Map[(Length[GreedyVertexCover[#]] - Length[ApproximateVertexCover[#]]) &,
          Table[RandomTree[30], {20}]]
Out[6]= {0, 2, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 2, 0, 0, 1, 0, 1, 1, 2}
```

но на плотных графах лучше работает аппромаксимационный алгоритм

```
In[7]:= Map[(Length[GreedyVertexCover[#]] - Length[ApproximateVertexCover[#]]) &,
          Table[RandomGraph[30, 0.5], {20}]]
Out[7]= {-1, -2, -2, 0, -1, 0, 0, -3, -3, 0, 0, 0, 0, -4, 0, 0, -1, 0, 0, 0}
```

MinimumVertexCover[g] находит минимальное вершинное покрытие графа g.
Любое подмножество n-1 вершин есть минимальное вершинное покрытие $K_n$.

```
In[8]:= MinimumVertexCover[CompleteGraph[20]]
Out[8]= {2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}
```

```
In[9]:= {Length[MinimumVertexCover[RandomGraph[100, 0.5]]],
         Length[VertexCover[RandomGraph[100, 0.5]]],
         Length[GreedyVertexCover[RandomGraph[100, 0.5]]],
         Length[ApproximateVertexCover[RandomGraph[100, 0.5]]]}
Out[9]= {91, 93, 92, 92}
```

Независимые множества и вершинные покрытия – тесно связанные понятия, как показывается в следующей теореме

**Теорема.** Рассмотрим граф g=(E,V). Подмножество S⊆V является независимым множеством графа g тогда и только тогда, когда его дополнение является вершинным покрытием.

**Следствие.** Для простого графа на n вершинах $\alpha+\beta$=n.

Рассмотрим произвольное максимальное паросочетание $M^*$ и произвольное минимальное вершинное покрытие $K^*$ графа g. Поскольку для покрытия ребер $M^*$ требуется по крайней мере $|M^*|$ вершин, вершинное покрытие должно содержать хотя бы $|M^*|$ вершин. Поэтому $|M^*|\le|K^*|$. В общем случае равенство в этом выражении не всегда имеет место. Но $|M^*|=|K^*|$, если g –двудольный граф.

**Теорема.** Число ребер в максимальном паросочетании двудольного графа равно числу вершин минимального вершинного покрытия.

```
In[11]:= {Length[MinimumVertexCover[Hypercube[10]]], Length[MaximalMatching[Hypercube[10]]]}
Out[11]= {512, 512}
```

Для любого вершинного покрытия (u,v) и любого совершенного паросочетания M c(u,v)≥w(M). Более того, c(u,v)=w(M) тогда и только тогда, когда ребро (i,j) в M удовлетворяет $u_{i+}\nu j=w_{i,j}$.
В этом случае M – максимум паросочетания и (u,v) – минимум вершинного покрытия.

Задача взвешенного двудольного паросочетания состоит в том, чтобы найти паросочетание в реберно-взвешенном двудольном графе, чей общий вес наибольший. Существует замечательная

двойственность между задачей взвешенного паросочетания и задачей взвешенного вершинного покрытия, которая может быть использована при решении обеих задач двудольных графов. Задача взвешенного вершинного покрытия состоит в том чтобы найти цены $u_i$, $(1 \le i \le n)$ и $v_j$ $(1 \le j \le n)$, такие, что для любых i,j, $u_i + v_j \ge w_{ij}$ и сумма цен минимальна. Пусть $u=(u_1,u_2,\ldots..,u_n)$ и $v=(v_1,v_2,\ldots v_n)$, (u,v) – покрытие, $c(u,v)=\sum_i u_I + \sum_j v_j$ – цена покрытия, которую нужно минимизировать.

Без ограничения общности можно считать исходный граф полным, взвешенным двудольным графом $K_{n,n}$, поскольку если граф не полный, можно добавить столько вершин, чтобы получить две доли равного размера и присвоить нулевые веса отсутствующим ребрам, что не изменит общий вес. Вершины каждой доли занумерованы от 1 до n, а $w_{i,j}$ - вес ребра, соединяющего i-ю вершину с j-й.

Пусть M – совершенное паросочетание графа $K_{n,n}$, а w(M) – его вес. Теорема двойственности, выражающая связь между задачей паросочетания наибольшего веса и задачей вершинного покрытия наименьшего веса, формулируется следующим образом:

**Теорема.** Для любого вершинного покрытия (u,v) и для любого совершенного паросочетания M $c(u,v) \ge w(M)$. Более того, c(u,v)=w(M) тогда и только тогда, когда ребро (i, j) в M удовлетворяет условию, $u_i + v_j = w_{ij}$. В этом случае M – наибольшее паросочетание и (u,v) – наименьшее вершинное покрытие.

Встроенная функция BipartiteMatchingAndCover [g] берет двудольный граф g и возвращает паросочетание наибольшего веса вместе с двойственным вершинным покрытием. Если граф невзвешенный, предполагается, что все веса ребер равны 1.

Присвоим случайные целые веса из отрезка [1,10] графу $K_{10,10}$:

```
In[12]:= g = SetEdgeWeights[CompleteGraph[10, 10], WeightingFunction → RandomInteger,
            WeightRange → {1, 10}];
```

Рассмотрим паросочетание с наибольшим весом и двойственное вершинное покрытие:

```
In[13]:= {m, e} = BipartiteMatchingAndCover[g]

Out[13]= {{{1, 19}, {2, 13}, {3, 12}, {4, 14}, {5, 15}, {6, 11}, {7, 20}, {8, 17}, {9, 18}, {10, 16}},
          {9, 9, 9, 5, 9, 7, 10, 9, 10, 8, 1, 0, 1, 1, 1, 2, 1, 0, 0, 0}}
```

Просуммируем элементы m и e

Равенство этих двух величин есть доказательство того, что паросочетание и покрытие вычисленные выше, являются оптимальными

```
In[14]:= {Apply[Plus, GetEdgeWeights[g, m]], Apply[Plus, e]}
Out[14]= {92, 92}
```

Рассмотрим задачу невзвешенного вершинного покрытия в двудольном графе g. Присвоим каждому ребру в g вес 1, а отсутствующим ребрам – нулевой вес. В вершинном покрытии цены вершин – единицы и нули. Из двойственности между паросочетанием и вершинным покрытием следует, что размер паросочетания M с наибольшим весом равен размеру вершинного покрытия C с наименьшим весом, так, что каждому ребру в M ровно одна концевая точка лежит в C.

Рассмотрим вершинно-порожденный подграф g 10×10-ешетчатого графа и {m,e}=BipartiteMatchingAndCover[g]:

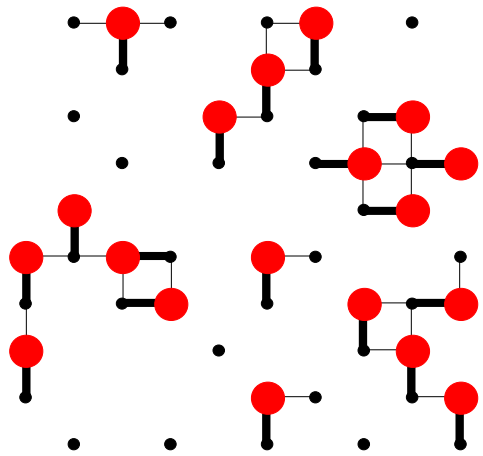
```
In[15]:= g = InduceSubgraph[GridGraph[10, 10], RandomSubset[100]];
         {m, e} = BipartiteMatchingAndCover[g]
```

```
Out[15]= {{{3, 7}, {5, 10}, {6, 11}, {9, 14}, {13, 19},
          {15, 22}, {16, 17}, {18, 26}, {20, 21}, {23, 29}, {24, 25}, {30, 31},
          {33, 39}, {34, 35}, {36, 37}, {40, 44}, {41, 42}, {43, 47}, {45, 50}},
         {0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1,
          0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0}}
```

Выделим веса вершинного покрытия:

```
In[16]:= f = Position[e, 1]
Out[16]= {{7}, {10}, {11}, {14}, {17}, {19}, {21}, {22}, {24},
          {26}, {29}, {31}, {35}, {37}, {39}, {42}, {44}, {47}, {50}}

In[17]:= d = Flatten[f]
Out[17]= {7, 10, 11, 14, 17, 19, 21, 22, 24, 26, 29, 31, 35, 37, 39, 42, 44, 47, 50}

In[18]:= {Length[m], Length[d]}
Out[18]= {19, 19}
```

Выделим элементы (m,e):



| | |
|---|---|
| VertexCoverQ[g, c] | возвращает True, если вершины списка c определяют вершинное покрытие графа |
| VertexCover[g] | возвращает вершинное покрытие графа g. При вычислении вершинного покрытия могут использоваться различные алгоритмы, в зависимости от установки опции Algorithm, которая может принимать значения Greedy, Approximate или Optimum. По умолчанию Algorithm -> Approximate |
| ApproximateVertexCover[g] | производит вершинное покрытие графа g |
| GreedyVertexCover[g] | возвращает вершинное покрытие графа g, построенное с помощью жадного алгоритма |
| MinimumVertexCover[g] | находит минимальное вершинное покрытие графа g |
| BipartiteMatchingAndCover [g] | берет двудольный граф g и возвращает паросочетание наибольшего веса вместе с двойственным вершинным покрытием. Если граф невзвешенный, предполагается, что все веса ребер равны 1 |

### *6.5.3. Вершинная раскраска и хроматическое число*

Вершинная k-раскраска графа – это присвоение его вершинам k различных цветов. Вершинная раскраска **правильная**, если никакие две смежные вершины в ней не получают одного цвета.
**Хроматическим числом** $\chi$ (g) графа g называется минимальное число k, для которого g вершинно k-раскрашиваемый. Граф g называется **k-хроматическим**, если $\chi$ (g)=k. Далее, вместо «правильная k-вершинная раскраска» будем говорить просто «k-раскраска». Аналогично, «вершинно k-раскрашиваемый» будем заменять на «k-раскрашиваемый». Не ограничивая общности, мы можем допустить, что графы, рассматриваемые в этом разделе, простые. Заметим, что k-раскраска графа g=(E,V) порождает разбиение $(V_1,\ldots..,V_k)$ множества V, где каждое $V_i$ – подмножество вершин, которым присвоен цвет i, и поэтому является независимым множеством. Аналогично каждое разбиение множества V на k независимых множеств соответствует k-раскраске графа g.
Приведем некоторые известные результаты для вершинной раскраски.
**Теорема.** Простой граф g является (Δ+1) раскрашиваемым, где Δ-максимальная степень вершины графа.
**Теорема (Брукс).** Пусть g –связный простой граф. Если он не цикл нечетной длины и не полный граф, то $\chi(g)\le\Delta$.
Функция ChromaticNumber[g] возвращает хроматическое число графа g.
Очевидно, что $\chi=\Delta+1$ для полных графов и циклов нечетной длины. Вычислим Δ и $\chi$ для CompleteGraph[i] и Cycle[i] для i=3,..,20:

```
In[2]:= Table[{{i, First[DegreeSequence[CompleteGraph[i]]], ChromaticNumber[CompleteGraph[i]]},
          {i, First[DegreeSequence[Cycle[2 i + 1]]], ChromaticNumber[Cycle[2 i + 1]]}}, {i, 3, 20}] //
        ColumnForm
Out[2]= {{3, 2, 3}, {3, 2, 3}}
        {{4, 3, 4}, {4, 2, 3}}
        {{5, 4, 5}, {5, 2, 3}}
        {{6, 5, 6}, {6, 2, 3}}
        {{7, 6, 7}, {7, 2, 3}}
        {{8, 7, 8}, {8, 2, 3}}
        {{9, 8, 9}, {9, 2, 3}}
        {{10, 9, 10}, {10, 2, 3}}
        {{11, 10, 11}, {11, 2, 3}}
        {{12, 11, 12}, {12, 2, 3}}
        {{13, 12, 13}, {13, 2, 3}}
        {{14, 13, 14}, {14, 2, 3}}
        {{15, 14, 15}, {15, 2, 3}}
        {{16, 15, 16}, {16, 2, 3}}
        {{17, 16, 17}, {17, 2, 3}}
        {{18, 17, 18}, {18, 2, 3}}
        {{19, 18, 19}, {19, 2, 3}}
        {{20, 19, 20}, {20, 2, 3}}
```

Теорема Брукса утверждает, что хроматическое число графа есть по крайней мере максимальная степень Δ вершины, если граф неполный или не нечетный цикл. Вычислим хроматические числа полуслучайных графов, сконструированных с помощью RealizeDegreeSequence

```
In[3]:= Table[ChromaticNumber[RealizeDegreeSequence[{4, 4, 4, 4, 4, 4}]], {5}]
Out[3]= {3, 3, 3, 3, 3}
```

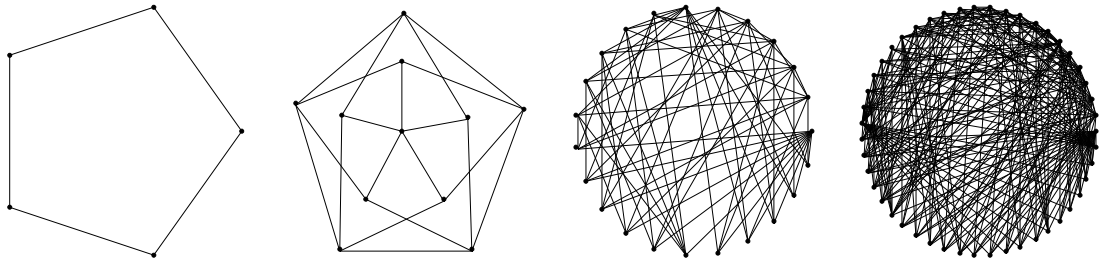
Минимальную окраску графа позволяет представить функция MinimumVertexColoring[g], а MinimumVertexColoring[g, k], возвращает k-окраску графа g, если таковая существует.
Каждая вершина полного графа должна быть окрашена в разные цвета:

```
In[4]:= Map[MinimumVertexColoring, Table[CompleteGraph[i], {i, 3, 10}]] // ColumnForm
```

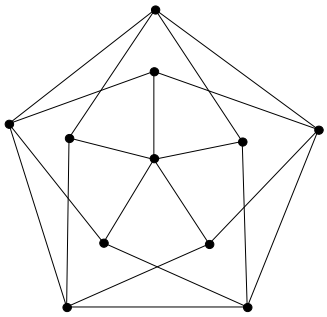
```
Out[4]= {1, 2, 3}
        {1, 2, 3, 4}
        {1, 2, 3, 4, 5}
        {1, 2, 3, 4, 5, 6}
        {1, 2, 3, 4, 5, 6, 7}
        {1, 2, 3, 4, 5, 6, 7, 8}
        {1, 2, 3, 4, 5, 6, 7, 8, 9}
        {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

Оказывается, для каждого k существуют k-хроматические графы без треугольников. Они называются графами Мыцельского. Встроенная функция MycielskiGraph[k] конструирует графы Мыцельского для любого целого положительного k.
Функция GrotztschGraph конструирует наименьший граф без треугольников с хроматическим числом 4. Он идентичен графу MycielskiGraph[4].

```
In[5]:= ShowGraphArray[Table[MycielskiGraph[i], {i, 3, 6}]]
```



```
Out[5]= - GraphicsArray -

In[6]:= IdenticalQ[MycielskiGraph[4], GrotztschGraph]
Out[6]= True

In[7]:= ShowGraph[GrotztschGraph]
```



Рассмотрим вершинные раскраски первых четырех графов Мыцельского

```
In[8]:= Table[MinimumVertexColoring[MycielskiGraph[i]], {i, 1, 4}]
Out[8]= {{1}, {1, 2}, {1, 2, 1, 2, 3}, {1, 2, 2, 1, 3, 3, 1, 2, 4, 1, 2}}
```

Использование тег All заставляет алгоритм раскраски вершин возвращать все раскраски, которые можно сконструировать с каждым числом цветов. Рассмотрим все раскраски в три цвета колеса на семи вершинах:

```
In[9]:= MinimumVertexColoring[Wheel[7], 3, All]
Out[9]= {{1, 2, 1, 2, 1, 2, 3}, {1, 3, 1, 3, 1, 3, 2}}
```

MinimumVertexColoring практически не работает на больших графах. В этом случае Combinatorica предлагает функцию BrelazColoring.
Функция BrelazColoring[g] возвращает вершинную раскраску, в которой вершины «жадно» окрашены в наименьшее число цветов в убывающем порядке степеней вершин.
Граф Мыцельского на десяти вершинах – «большой граф»:

```
In[10]:= {V[MycielskiGraph[10]], M[MycielskiGraph[10]]}
Out[10]= {767, 22196}
```

но BrelazColoring очень быстро работает на них.

```
In[11]:= Table[Max[BrelazColoring[MycielskiGraph[i]]], {i, 1, 10}]
Out[11]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

In[12]:= Timing[BrelazColoring[MycielskiGraph[10]];]
Out[12]= {1.281 Second, Null}
```

Полный к-дольный граф к-окрашиваем. Алгоритм Брелаза окрашивает такие графы с минимальным числом цветов

```
In[13]:= BrelazColoring[CompleteKPartiteGraph[1, 2, 4, 6]]
Out[13]= {1, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4}
```

Кроме того, Combinftorica предлагает функцию VertexColoring[g], которая возвращает не обязательно минимальную раскраску вершин. Функция допускает опцию Algorithm, которая принимает значения Brelaz или Optimum. По умолчанию Algorithm -> Brelaz.

```
In[14]:= {VertexColoring[DodecahedralGraph], ChromaticNumber[DodecahedralGraph]}
Out[14]= {{1, 2, 1, 2, 3, 2, 1, 2, 1, 1, 3, 3, 2, 3, 3, 1, 2, 1, 2, 4}, 3}
```

Установим опцию Algorithm ->Optimum и получим минимальную раскраску вершин додекаэдра

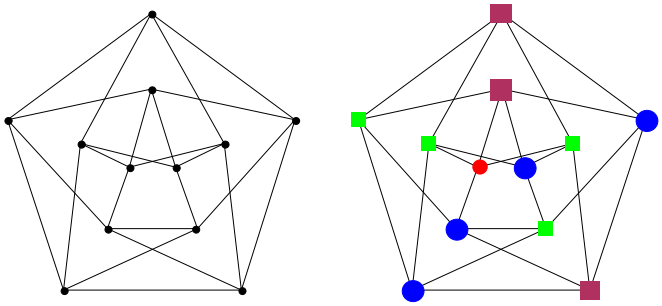
```
In[15]:= VertexColoring[DodecahedralGraph, Algorithm → Optimum]
Out[15]= {1, 2, 1, 2, 3, 2, 1, 2, 1, 1, 3, 3, 2, 3, 3, 1, 2, 3, 1, 2}
```

Функция ChvatalGraph возвращает наименьший 4-регулярный, 4-хроматический граф без треугольников.
Рассмотрим вершинную раскраску этого графа:

```
In[16]:= MinimumVertexColoring[ChvatalGraph]
Out[16]= {1, 2, 1, 1, 3, 1, 2, 3, 2, 3, 4, 2}
```

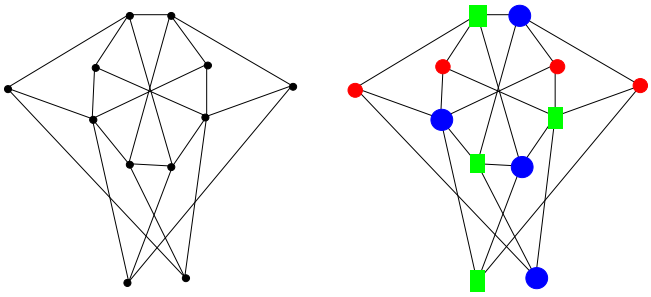
Окрасим вершины этого 4-регулярного графа в четыре цвета:

```
In[17]:= g = ChvatalGraph; s = {Red, Green, Blue, Maroon}; t = {Disk[0.05], Box[0.05], Disk[Large], Box[Large]};
         cv = {1, 2, 1, 1, 3, 1, 2, 3, 2, 3, 4, 2};
         ShowGraphArray[
          {g,
           SetGraphOptions[g,
            Table[ {i, VertexColor → s[[Mod[cv[[i]], 4] + 1]], VertexStyle → t[[Mod[cv[[i]], 4] + 1]]}, {i, V[g]}]]}]
```



Функция Uniquely3ColorableGraph возвращает 12-вершинный, без треугольников граф с хроматическим числом 3, который окрашиваем единственным образом.

```
In[18]:= g = Uniquely3ColorableGraph; s = {Red, Green, Blue}; t = {Disk[0.05], Box[0.05], Disk[Large]};
         cv = {1, 2, 1, 2, 1, 2, 3, 3, 3, 3, 1, 2};
         ShowGraphArray[
          {g,
           SetGraphOptions[g,
            Table[{i, VertexColor → s[[Mod[cv[[i]], 3] + 1]], VertexStyle → t[[Mod[cv[[i]], 3] + 1]]}, {i, V[g]}]]}]
```



| | |
|---|---|
| ChromaticNumber[g] | возвращает хроматическое число графа g |
| MinimumVertexColoring[g] | возвращает минимальную вершинную окраску графа g. |
| MinimumVertexColoring[g, k] | возвращает k-раскраску графа g, если таковая существует |
| BrelazColoring[g] | возвращает вершинную раскраску, в которой вершины «жадно» окрашены в наименьшее число цветов в убывающем порядке степеней вершин |
| VertexColoring[g] | возвращает не обязательно минимальную раскраску вершин. Функция допускает опцию Algorithm, которая принимает значения Brelaz или Optimum. По умолчанию Algorithm -> Brelaz. |
| ChromaticNumber[g] | возвращает хроматическое число графа g |
| MycielskiGraph[k] | конструирует графы Мыцельского для любого целого положительного k |
| GrotztschGraph | конструирует наименьший граф без треугольников с хроматическим числом 4 |
| ChvatalGraph | возвращает наименьший 4-регулярный, 4-хроматический граф без треугольников |
| Uniquely3ColorableGraph | возвращает 12-вершинный, без треугольников граф с хроматическим числом 3, который окрашиваем единственным образом |

### *6.5.4. Двудольные графы*

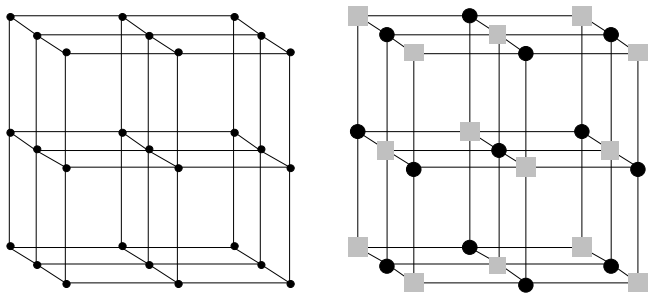
Граф **двудольный**, если его множество вершин может быть разбито на два непересекающихся множества, таких, что нет ребер, связывающих две вершины из разных множеств. Таким образом, "двуцветный граф" и "двудольный" – одно и то же свойство. Другую характеристику дву-

дольных графов дает теорема Кенига, которая утверждает, что граф двудольный тогда и только тогда, когда он не содержит циклов нечетной длины. Функция TwoColoring, приведенная ниже, присваивает каждой вершине один из двух цветов, так что раскраска правильная тогда и только тогда, когда граф двудольный.
Функция TwoColoring[g] находит двуцветную раскраску вершин графа g, если граф двудольный. Функция возвращает список из 1 и 2 (1, 2 - номера цветов) соответствующих вершинам графа. Рассмотрим 3×3×3-решетчатый граф. Убедимся, что он двудольный, и применим TwoColoring:

```
In[2]:= g = GridGraph[3, 3, 3]; If[BipartiteQ[g], col = TwoColoring[g]]
Out[2]= {1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1}
```

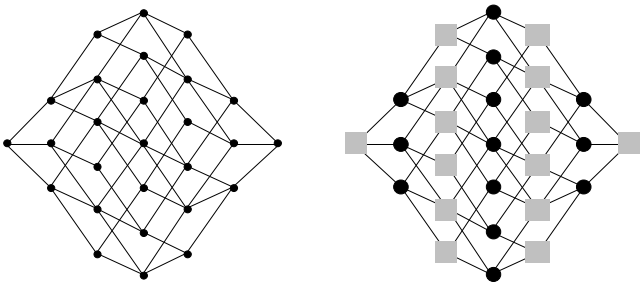
Теперь раскрасим вершины в серый и черный цвет:

```
In[3]:= s = {VertexStyle → Box[0.05], VertexColor → Gray}; s1 = {VertexStyle → Disk[0.05]};
        m = Flatten[Position[col, 1], 1]; m1 = Flatten[Position[col, 2], 1]; t = Join[m, s]; t1 = Join[m1, s1];
        ShowGraphArray[{g, h = SetGraphOptions[g, List[t, t1]]}]
```



Для удобства рассмотрения применим RankedEmbedding:

```
In[4]:= g = RankedEmbedding[GridGraph[3, 3, 3], {1}]; If[ BipartiteQ[g], col = TwoColoring[GridGraph[3, 3, 3]]];
        s = {VertexStyle → Box[0.05], VertexColor → Gray}; s1 = {VertexStyle → Disk[0.05]};
        m = Flatten[Position[col, 1], 1]; m1 = Flatten[Position[col, 2], 1]; t = Join[m, s]; t1 = Join[m1, s1];
        ShowGraphArray[{g, h = SetGraphOptions[g, List[t, t1]]}]
```

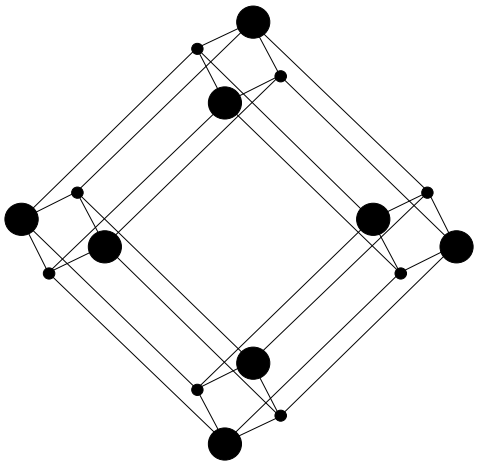


Деревья, решетчатые графы, гиперкубы, циклы четной длины являются двудольными.

```
In[5]:= {BipartiteQ[RandomTree[20]], BipartiteQ[Cycle[20]], BipartiteQ[Hypercube[5]], BipartiteQ[Cycle[19]]}
Out[5]= {True, True, True, False}
```

Рассмотрим двуцветную раскраску четырехмерного гиперкуба:

```
In[6]:= g = Hypercube[4]; col = TwoColoring[g]; ver1 = Flatten[Position[col, 1]]; ver2 = Flatten[Position[col, 2]];
        ShowGraph[Highlight[g, {ver1}]]
```

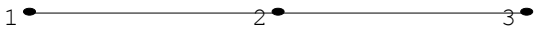


| | |
|---|---|
| TwoColoring[g] | находит двуцветную раскраску вершин графа g, если граф двудольный. Функция возвращает список из 1 и 2 (1, 2- номера цветов) соответствующих вершинам графа. Раскраска правильная тогда и только тогда, когда g – двудольный граф |

### *6.5.5. Хроматический полином*

Если граф z-раскрашиваемый, то его можно раскрасить в z цветов более чем одним способом. Две раскраски графа считаются различными, если хотя бы одной вершине присваиваются разные цвета.
Хроматический полином P(g,z) принимает значение для каждого целого z, равное числу различных правильных z-раскрасок графа g.

```
In[2]:= ShowLabeledGraph[Path[3]]
```




Рассмотрим, например, граф пути на трех вершинах. Среди данных z цветов мы можем выбрать любой для раскраски вершины 1. Вершину 2 можно раскрасить в один из оставшихся z-1 цветов. Для каждой раскраски вершины 2 существует z-1 различных способов раскраски вершины 3. Таким образом, путь на трех вершинах можно раскрасить $z(z-1)^2$ различными способами. Другими словами, хроматический полином графа равен $z(z-1)^2$.С помощью таких же рассуждений можно показать, что хроматический полином пути на n вершинах равен $z(z-1)^n$. В качестве другого примера рассмотрим полный граф $K_n$ на n вершинах. При наличии z цветов первую вершину можно раскрасить в любой из них, вторую в любой из оставшихся z-1 цветов, третью в любой из оставшихся z-2 цветов и т.д. Таким образом, $P(K_n, z)=z(z-1)\ldots(z-n+1)$.
Функция ChromaticPolynomial[g, z] возвращает хроматический полином P(z) графа g, который считает число способов раскраски графа g в по крайней мере z цветов.

```
In[3]:= {ChromaticPolynomial[Path[100], z], ChromaticPolynomial[EmptyGraph[100], z],
         ChromaticPolynomial[CompleteGraph[10], z]} // ColumnForm
Out[3]= (-1 + z)^99 z
        z^100
        (-9 + z) (-8 + z) (-7 + z) (-6 + z) (-5 + z) (-4 + z) (-3 + z) (-2 + z) (-1 + z) z
```

Хроматический полином несвязного графа есть произведение хроматических полиномов их связных компонент

```
In[4]:= ChromaticPolynomial[GraphUnion[CompleteGraph[5], Wheel[5]], z]
Out[4]= 336 z^2 - 1444 z^3 + 2616 z^4 - 2617 z^5 + 1588 z^6 - 601 z^7 + 139 z^8 - 18 z^9 + z^10
```

Разложим этот полином на множители:

```
In[5]:= Factor[%]
Out[5]= (-4 + z) (-3 + z) (-2 + z)^2 (-1 + z)^2 z^2 (7 - 5 z + z^2)
In[6]:= ChromaticPolynomial[CompleteGraph[5], z] Factor[ChromaticPolynomial[Wheel[5], z]]
Out[6]= (-4 + z) (-3 + z) (-2 + z)^2 (-1 + z)^2 z^2 (7 - 5 z + z^2)
```

Вычислим число различных раскрасок решетчатого 3x3 графа:

```
In[7]:= Table[ChromaticPolynomial[GridGraph[3, 3], z], {z, 1, 10}]
Out[7]= {0, 2, 246, 9612, 142820, 1166910, 6464682, 27350456, 95004072, 283982490}
```

| | |
|---|---|
| ChromaticPolynomial[g, z] | возвращает хроматический полином P(z) графа g, который считает число способов раскраски графа g в по крайней мере z цветов |

### *6.5.6. Реберная раскраска графов и хроматический индекс*

Реберной k-раскраской графа называется присвоение ребрам графа k различных цветов. Реберная раскраска называется правильной, если никакие два смежных ребра не получают в ней одинакового цвета. Граф называется реберно-k-раскрашиваемым, если он имеет правильную окраску. Не нарушая общности, допустим, что графы, рассматриваемые в этом разделе, не имеют петель.
Хроматический индекс или реберное хроматическое число $\chi_1(g)$ графа g – это минимальное число k, для которого граф g имеет правильную реберную k-окраску. Граф g называется реберно-k-хроматическим, если $\chi_1(g)=k$.
Легко видеть, что реберная k-раскраска графа g=(E,V) влечет разбиение $(E_1,E_2,\ldots,E_k)$ множества E, где $E_i$ – подмножество ребер, которым присвоен в раскраске цвет i. Аналогично любое разбиение множества E на k подмножеств соответствует реберной k-раскраске графа g. Поэтому реберную раскраску обозначают соответствующим разбиением графа g или номерами k. Если раскраска $(E_1,E_2,\ldots,E_k)$ правильная, то каждое $E_i$ является паросочетанием. Поэтому $\chi_1(g)$ можно рассматривать как наименьшее число паросочетаний, на которые разбивается множество ребер графа g.
Функция EdgeChromaticNumber[g] возвращает хроматический индекс графа g.
Поскольку в любой правильной раскраске ребра, инцидентные одной вершине, получают разные цвета, то $\chi_1(g)\geq\Delta$, где $\Delta$- максимальная степень в графе g. В общем случае $\chi_1(g)\neq\Delta$, но в случае двудольных графов $\chi_1(g)=\Delta$

```
In[2]:= {EdgeChromaticNumber[CompleteKPartiteGraph[5, 8]],
         Max[DegreeSequence[CompleteKPartiteGraph[5, 8]]]}
Out[2]= {8, 8}
```

а это хроматический индекс гиперкуба:

```
In[3]:= {BipartiteQ[g = Hypercube[4]], EdgeChromaticNumber[g]}
Out[3]= {True, 4}
```

В общем случае Δ цветов недостаточно для того, чтобы правильно раскрасить ребра графа, Визинг показал, что для любого простого графа достаточно Δ+1 цветов.
**Теорема** (Визинг). Если g=(E,V) – простой граф, то либо $\chi_1(g)=\Delta$, либо $\chi_1(g)=\Delta+1$.

```
In[4]:= {Max[DegreeSequence[PetersenGraph]], EdgeChromaticNumber[PetersenGraph]}
Out[4]= {3, 4}
```

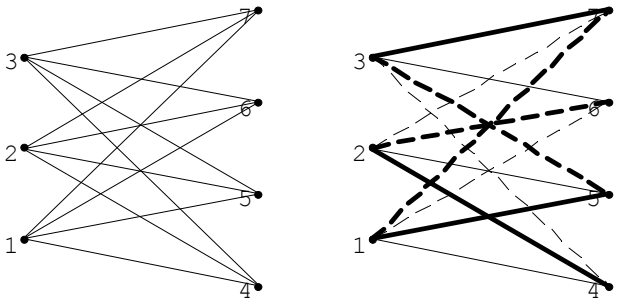
Точно определить, какие графы имеют хроматический индекс Δ, а какие Δ+1, практически невозможно, так как доказано, что это NP– полная задача.
Функция EdgeColoring[g] возвращает раскраску ребер графа g, не обязательно минимальную.

Рассмотрим, например раскраску двудольного графа:

```
In[5]:= EdgeColoring[CompleteKPartiteGraph[3, 4]]
Out[5]= {1, 2, 3, 4, 2, 1, 4, 3, 3, 4, 1, 2}
```

Теперь изобразим полученную реберную раскраску. Поскольку печать рисунков не цветная, различную раскраску ребер будем изображать обычными линиями, жирными линиями, обычным пунктиром и жирным пунктиром.

```
In[6]:= s = {Normal, Thick, NormalDashed, ThickDashed};
        t = EdgeColoring[CompleteKPartiteGraph[3, 4]]; e = Edges[g];
        ShowGraphArray[{g = CompleteKPartiteGraph[3, 4],
          SetGraphOptions[g, Table[{e[[i]], EdgeStyle → s[[t[[i]]]]}, {i, M[g]}]]},
         VertexNumber → True]
```




Нахождение минимальной раскраски ребер графа эквивалентно нахождению минимальной окраски вершин соответствующего реберного графа. Таким образом, хроматический полином и функции вершинной окраски могут быть применены к реберным окраскам.

Найдем, например, хроматический полином вышеприведенного двудольного графа.

```
In[6]:= ChromaticPolynomial[LineGraph[CompleteKPartiteGraph[3, 4]], z]
Out[6]= -166848 z + 615668 z^2 - 998412 z^3 + 955989 z^4 - 608616 z^5 +
         273307 z^6 - 89022 z^7 + 21144 z^8 - 3600 z^9 + 419 z^10 - 30 z^11 + z^12
```

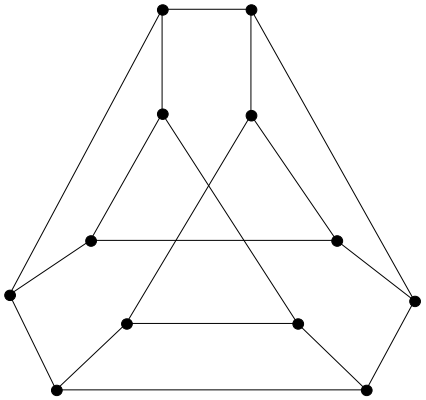
При изготовлении карт, для того чтобы отличать отдельные области, необходимо их раскрашивать таким образом, чтобы никакие две смежные области не были раскрашены одинаково. Легко

привести пример, показывающий, что трех цветов в общем случае недостаточно для правильной раскраски планарного графа. Следующая теорема известна под названием теоремы о пяти красках:

**Теорема (**Хивуд**).** Любой планарный граф 5-раскрашиваемый.

Возникает вопрос: можно ли улучшить теорему о пяти красках? Предполагалось, что любой планарный граф 4-раскрашиваемый. Эта гипотеза известна как гипотеза четырех красок. Она оставалась неразрешенной более 100 лет и была предметом интенсивных исследований. В 1976 году была доказана справедливость гипотезы.
Эти теоремы верны для плоских графах и, следовательно, для графов на сфере. В теории графов известна гипотеза Хивуда, которая дает верхнюю границу числа цветов, достаточных для раскраски графа на поверхности данного рода. Единственным контрпримером этому предположению является бутылка Клейна, для которого формула Хивуда дает число цветов равное 7, а на самом деле достаточно 6 цветов, что демонстрирует граф Франклина.

```
In[7]:= ShowGraph[FranklinGraph]
```



| EdgeChromaticNumber[g] | возвращает хроматический индекс графа g |
|---|---|
| FranklinGraph | возвращает граф Франклина |

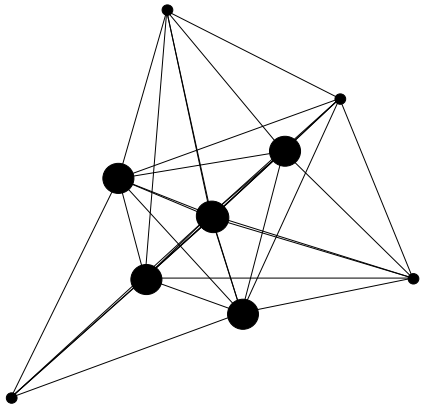
### *6.5.7. Максимальная клика*

**Кликой** в графе G называется подмножество вершин, которое индуцирует полный подграф. Любая вершина или ребро определяет клику размера 1 и 2 соответственно, но практический интерес вызывают максимальные клики.
Функция CliqueQ[g, s] возвращает True, если подмножество вершин s определяет клику в графе g, и False в противном случае.
Любое подмножество вершин в полном графе g образует клику:

```
In[2]:= CliqueQ[CompleteGraph[20], RandomSubset[Range[20]]]
Out[2]= True
```

Функция MaximumClique[g] находит максимальную клику в графе g. MaximumClique[g, k] возвращает k-клику, если таковая существует в g. В противном случае функция возвращает пустой список.

```
In[3]:= q = RandomGraph[10, 0.8]; ShowGraph[Highlight[q, MaximumClique[q]]]
```



Двудольный граф не может иметь клику большего размера, чем 2.

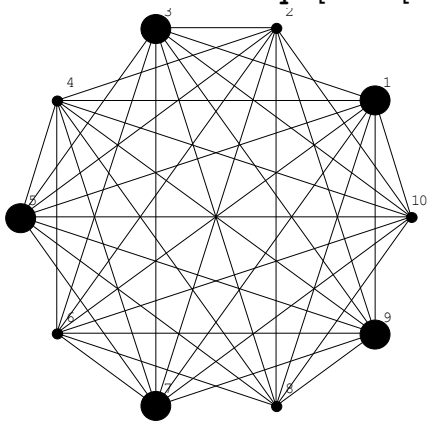
```
In[4]:= {MaximumClique[CompleteKPartiteGraph[5, 6]],
         MaximumClique[CompleteKPartiteGraph[5, 6], 3]}
Out[4]= {{1, 6}, {}}
```

Combinatorica содержит функцию Turan[n, p], которая конструирует граф Турана – наибольший граф на n вершинах, который не содержит $K_p$ как подграф.

```
In[5]:= MaximumClique[Turan[10, 6]]
Out[5]= {1, 3, 5, 7, 9}
```

Выделим с помощью Highlight максимальную клику:

```
In[6]:= ShowLabeledGraph[Highlight[CircularEmbedding[Turan[10, 6]],
          MaximumClique[Turan[10, 6]]], VertexNumberPosition → {0.04, 0.04}]
```




Граф называется **совершенным**, если для любого подграфа графа g размер наибольшей клики равен хроматическому числу.
Функция PerfectQ[g] возвращаетTrue, если g–совершенный граф и False в противном случае.
Для каждого подграфа решетчатого 3×3-графа подсчитаем размер наибольшей клики:

```
In[7]:= Map[Length,
          Map[MaximumClique, Map[InduceSubgraph[GridGraph[3, 3], #] &, Subsets[6]]]]
Out[7]= {0, 1, 2, 1, 2, 2, 1, 1, 1, 2, 2, 2, 1, 2, 2, 1, 2, 2, 2,
         2, 2, 2, 2, 2, 1, 1, 2, 2, 2, 2, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2,
         2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 1, 1}
```

Теперь рассмотрим хроматические числа всех этих подграфов:

```
In[8]:= Map[ChromaticNumber, Map[InduceSubgraph[GridGraph[3, 3], #] &, Subsets[6]]]
Out[8]= {0, 1, 2, 1, 2, 2, 1, 1, 1, 2, 2, 2, 1, 2, 2, 1, 2, 2, 2,
         2, 2, 2, 2, 2, 1, 1, 2, 2, 2, 2, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2,
         2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 1, 1}
```

Сравним эти списки:

```
In[9]:= SameQ[
          Map[Length, Map[MaximumClique,
            Map[InduceSubgraph[GridGraph[3, 3], #] &, Subsets[6]]]],
          Map[ChromaticNumber, Map[InduceSubgraph[GridGraph[3, 3], #] &, Subsets[6]]]]

Out[9]= True
```

| | |
|---|---|
| CliqueQ[g, s] | возвращает True, если подмножество вершин s определяет клику в графе g |
| MaximumClique[g] | возвращает максимальную клику в графе g |
| Turan[n, p] | конструирует граф Турана – наибольший граф на n вершинах, который не содержит $K_p$ как подграф |
| PerfectQ[g] | возвращает True, если g – совершенный граф и False в противном случае |

# Глава 7. Оптимизационные алгоритмы на графах

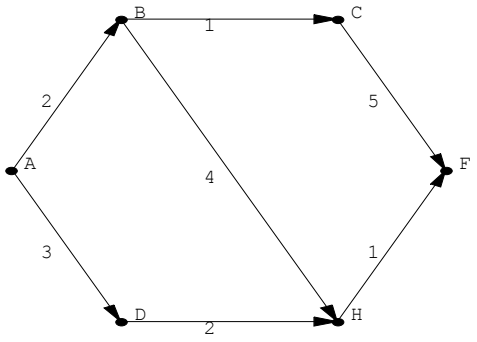
## ■7.1. Кратчайшие пути

Кратчайший путь – это путь минимального общего веса, соединяющий выбранные вершины. Если граф невзвешенный, то кратчайший путь между вершинами – путь, содержащий минимальное число ребер. Задача нахождения кратчайшего пути в невзвешенном графе может быть решена с помощью поиска в ширину. Но поиск в ширину не подходит для поиска кратчайшего пути во взвешенном графе, потому что кратчайший взвешенный путь между s и t необязательно содержит минимальное число ребер.

### *7.1.1. Алгоритмы Дейкстры и Беллмана-Форда*

Рассмотрим взвешенный граф:

```
In[2]:= v = {{-1, 0}, {-0.5, 0.8}, {0.5, 0.8}, {1, 0}, {0.5, -0.8}, {-0.5, -0.8}};
        g = FromOrderedPairs[{{1, 2}, {2, 3}, {3, 4}, {5, 4}, {6, 5}, {1, 6}, {2, 5}}, v];
        g = SetEdgeWeights[g, {2, 1, 5, 1, 2, 3, 4}];
        g = SetEdgeLabels[g, {2, 1, 5, 1, 2, 3, 4}];
        g = SetVertexLabels[g, {A, B, C, F, H, D}];
        ShowGraph[g, EdgeLabel → GetEdgeWeights[g], VertexLabel → True,
         TextStyle → {FontSize → {14}}]
```




Этот граф может представлять, например, дороги и их длины в километрах между шестью городами. Поскольку количество вершин в этом графе невелико, то можно перебрать все возможные пути между любой парой заданных вершин. При этом, естественно, найдется наиболее короткий путь, соединяющий соответствующие города. В реальной задаче, возникающей в профессиональной деятельности, число вершин, как правило, настолько велико, что такой упрощенный подход к поиску кратчайшего пути слишком неэффективен.

Для поиска кратчайшего пути в реберно-взвешенных графах или в графах со взвешенными вершинами предпочтительнее использовать алгоритм Дейкстры. Для заданной вершины A он находит кратчайший путь от A до всех остальных вершин. Перед формальным изложением алгоритма мы опишем его и проиллюстрируем работу алгоритма на вышеприведенном ориентированном графе.

Пусть g – связный ориентированный граф, в котором каждому ориентированному ребру сопоставлено положительное вещественное число, называемое длиной ребра. Длина ребра, направленного из вершины i в вершину j, обозначается через $w_{i,j}$. Если в графе отсутствует ребро, направленное из вершины i в вершину j, то $w_{i,j}=\infty$. Рассмотрим матрицу весов для нашего графа:

```
In[5]:= w = ToAdjacencyMatrix[g, EdgeWeight] // MatrixForm
```

Out[5]//MatrixForm=

$$\begin{pmatrix} \infty & 2 & \infty & \infty & \infty & 3 \\ \infty & \infty & 1 & \infty & 4 & \infty \\ \infty & \infty & \infty & 5 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 1 & \infty & \infty \\ \infty & \infty & \infty & \infty & 2 & \infty \end{pmatrix}$$

Рассмотрим алгоритм Дейкстры, который определяет кратчайшие пути из данной вершины ко всем другим вершинам связного ориентированного графа на n вершинах. В течение работы алгоритма каждой вершине v ориентированного графа присваивается число d[v], равное расстоянию от вершины A до v. Перед началом работы d[v] совпадает с весом ребра (A,v) , если ребро существует или равно ∞ в противном случае. Мы будем проходить вершины ориентированного графа и уточнять значения d[v].

На каждом шагу алгоритма отмечается одна вершина u, до которой уже найден кратчайший путь от A и расстояние d[u] до нее. Далее полученное значение d[u] отмеченной вершины не меняется. Для оставшихся, неотмеченных вершин v, число d[v] будет меняться с учетом того, что искомый кратчайший путь до них от A будут проходить через последнюю отмеченную вершину u. Алгоритм завершится в тот момент, когда все возможные вершины будут отмечены и получат свои окончательные значения d[v].

Для каждого шага алгоритма, описанного ниже, в соответствующую строку таблицы заносится отмеченная вершина, текущие значения d[v] и оставшиеся неотмеченные вершины. При этом жирным шрифтом выделяется наименьшее из значений d[v] среди неотмеченных вершин. Соответствующую вершину следует отметить. Кроме того, в таблице все значения d[v] для уже отмеченных вершин отделены от остальных ломаной линией.

| Шаг | Отмеченные вершины | Расстояние до вершины | | | | | | Неотмеченные вершины | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | A | B | C | D | H | F | | | | | |
| 0 | A | 0 | **2** | ∞ | 3 | ∞ | ∞ | B, | C, | D, | H, | F |
| 1 | B | 0 | 2 | 3 | **3** | 6 | ∞ | C, | D, | H, | F | |
| 2 | D | 0 | 2 | **3** | 3 | 5 | ∞ | C, | H, | F | | |
| 3 | C | 0 | 2 | 3 | 3 | **5** | 8 | H, | F | | | |
| 4 | H | 0 | 2 | 3 | 3 | 5 | **6** | F | | | | |
| 5 | F | 0 | 2 | 3 | 3 | 5 | 6 | | | | | |

**Шаг 0.** Поскольку нас интересует кратчайший путь от вершины A, мы ее отмечаем и используем первую строку матрицы w для определения начальных значений d[v]. Таким образом, получается первая строка таблицы. Наименьшее число из всех d[v] для неотмеченных вершин – d[B]=2.

**Шаг 1.** Отмечаем вершину B, так как она является ближайшей к A. Вычисляем длины путей, ведущих от A к неотмеченным вершинам через вершину B. Если новые значения d[v] оказываются меньше старых, то меняем последние на новый. Итак, при этом проходе цикла путь ABC имеет вес 3.

**Шаг 2.** Из оставшихся неотмеченными вершины C и D находятся ближе всех к A. Отметить можно любую из них. Возьмем вершину D. Так как длина пути ADH равна 5, текущее значение d[H] следует уменьшить до 5. Теперь можно заполнить третью строчку таблицы. Наименьшее значение d[v] среди неотмеченных к этому моменту вершин оказывается у вершины C.

**Шаг 3.** Отмечаем вершину C и подправляем значения d[v]. Теперь можно дойти и до вершины F, следуя путем ABCF. Его длина, а стало быть и значение d[F], равны 8. К этому моменту остались неотмеченными две вершины: H и F.

**Шаг 4.** Мы отмечаем вершину H, что позволяет нам уменьшить величину d[F] с 8 до 6.

**Шаг 5.** Отмечаем вершину F.

Встроенная функция Dijkstra[g, v] возвращает остовное дерево кратчайших путей графа g и соответствующие расстояния от вершины v. Остовное дерево кратчайших путей выдается списком, в котором элемент i – родитель i-й вершины в остовном дереве кратчайших путей.
Применим функцию Dijkstra к вышеприведенному графу g:

```
In[6]:= Dijkstra[g, 1]
Out[6]= {{1, 1, 2, 5, 6, 1}, {0, 2, 3, 6, 5, 3}}
```

Первый список, возвращаемый алгоритмом, содержит родительские отношения в остовном дереве кратчайших путей. Выделим кратчайший путь от первой вершины до четвертой:

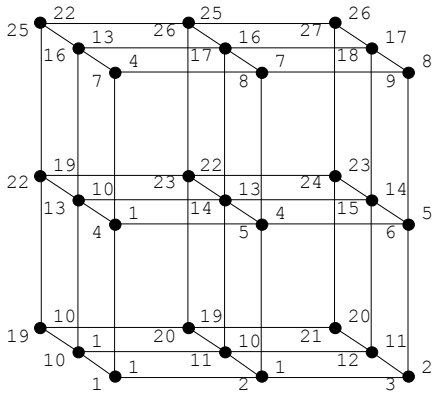
```
In[7]:= ParentsToPaths[{1, 1, 2, 5, 6, 1}, 1, 4]
Out[7]= {1, 6, 5, 4}
```

Второй список, возвращаемый Dijkstra, выдает расстояния от вершины v. Четвертый элемент этого списка 6 – есть кратчайшее расстояние от первой вершины до четвертой.
Рассмотрим кратчайшие пути из вершины 1 до остальных вершин решетчатого 3×3×3-графа.

```
In[8]:= Dijkstra[g = GridGraph[3, 3, 3], 1]
Out[8]= {{1, 1, 2, 1, 4, 5, 4, 7, 8, 1, 10, 11, 10,
          13, 14, 13, 16, 17, 10, 19, 20, 19, 22, 23, 22, 25, 26},
         {0, 1, 2, 1, 2, 3, 2, 3, 4, 1, 2, 3, 2, 3, 4, 3, 4, 5, 2, 3, 4, 3, 4, 5, 4, 5, 6}}
```

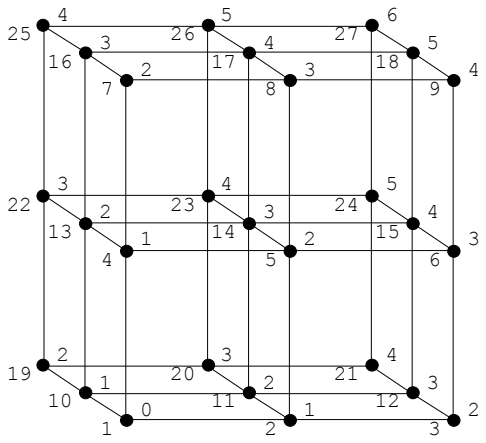
Поместим номер родителя каждой вершины на кратчайшем пути от первой вершины как метку справа и выше от вершины.

```
In[9]:= ShowGraph[g, VertexLabel → Dijkstra[g, 1][[1]], VertexNumber → True,
          PlotRange → 0.15, TextStyle → {FontSize → 11}]
```



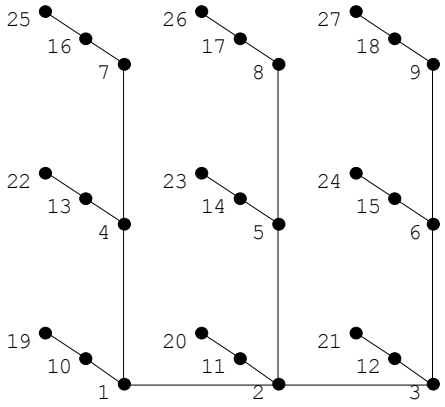

Второй список, возвращаемый алгоритмом, дает расстояния от первой вершины до всех других вершин графа. Поместим номера вершин в нижнем левом углу, а расстояния – в верхнем правом углу каждой вершины. Этот граф невзвешенный, каждое ребро имеет единичную длину и кратчайшие пути минимизируют число ребер.

```
In[10]:= ShowLabeledGraph[g, VertexLabel → Dijkstra[g, 1][[2]], PlotRange → 0.15,
           TextStyle → {FontSize → 12}]
```



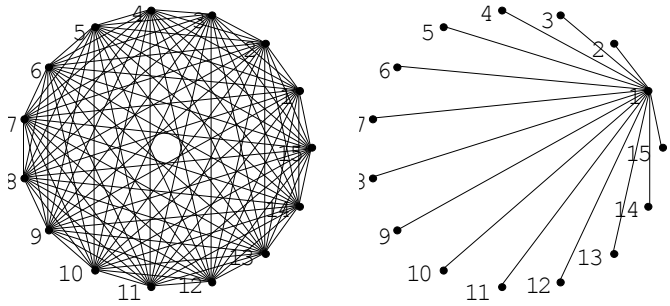

Функция ShortestPathSpanningTree[g, v] конструирует остовное дерево кратчайших путей с корнем в вершине v, так что кратчайший путь от v к любой другой вершине есть путь в этом дереве. Функция допускает опцию Algorithm, которая может принимать значения Automatic, Dijkstra или BellmanFord. По умолчанию Algorithm –> Automatic. В этом случае, в зависимости от наличия отрицательных весов или в зависимости от плотности графа, алгоритм выбирает между алгоритмом Дейкстры или алгоритмом Беллмана-Форда.
Построим остовное дерево кратчайших путей вышеприведенного графа g.

```
In[11]:= ShowLabeledGraph[ShortestPathSpanningTree[g, 1], PlotRange → 0.15]
```



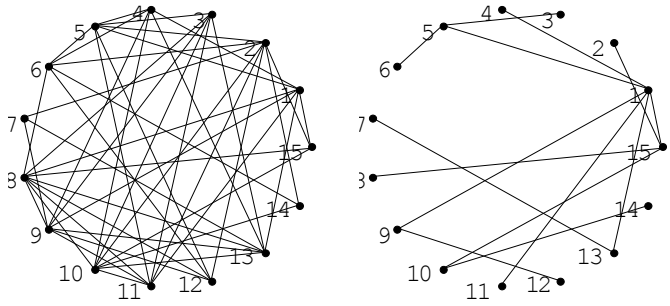

Рассмотрим случайный полный граф на двадцати вершинах и присвоим каждому ребру вес, равный евклидову расстоянию между его концевыми точками. Так как для весов ребер выполняется неравенство треугольника, то кратчайший путь от корня до вершины в полном графе с такими весами есть просто расстояние от корня до этой вершины.

```
In[12]:= g = SetEdgeWeights[RandomGraph[15, 1], WeightingFunction → Euclidean];
         ShowGraphArray[{g, ShortestPathSpanningTree[g, 1]}, VertexNumber → True];
```



Проделаем тот же эксперимент на редких случайных графах. Здесь необязательно существует прямой путь от корня к вершине, т.к. мы видим переходы и изгибы. Дерево отражают геометрию точек. Корень идентифицируется как вершина с наибольшей степенью.

```
In[13]:= g = SetEdgeWeights[RandomGraph[15, 0.5], WeightingFunction → Euclidean];
         ShowGraphArray[{g, ShortestPathSpanningTree[g, 1]}, VertexNumber → True];
```

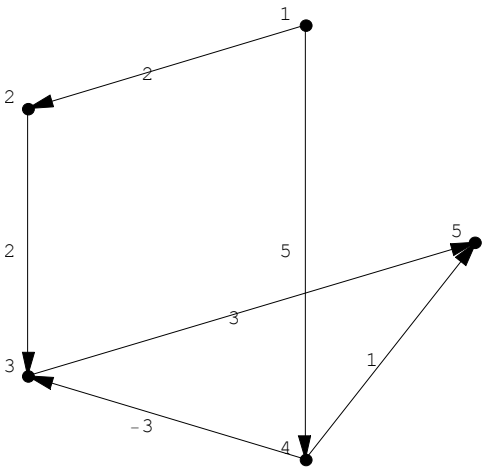


Алгоритм Дейсктры работает корректно и на несвязных графах. Если от корня до данной вершины нет пути, то кратчайшему пути присваивается ∞.

```
In[14]:= Dijkstra[GraphUnion[Star[10], Wheel[10]], 1]
Out[14]= {{1, 10, 10, 10, 10, 10, 10, 10, 10, 1, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20},
          {0, 2, 2, 2, 2, 2, 2, 2, 2, 1, ∞, ∞, ∞, ∞, ∞, ∞, ∞, ∞, ∞, ∞}}
```

Алгоритм Беллмана-Форда корректно вычисляет наикратчайшие пути даже при отрицательных весах ребер, при условии, что граф не содержит отрицательных циклов. Отрицательный цикл – это ориентированный цикл, чья сумма весов ребер отрицательна.
Функция BellmanFord[g, v] возвращает остовное дерево кратчайших путей и соответствующие расстояния от вершины v графа g. Остовное дерево кратчайших путей задается списком, в котором элемент i – родитель вершины i в остовном дереве кратчайших путей.
Рассмотрим ориентированный граф h, одно из ребер которого имеет отрицательный вес.

```
In[15]:= g = FromOrderedPairs[{{1, 2}, {2, 3}, {1, 4}, {4, 3}, {4, 5}, {3, 5}}];
         h = SetEdgeWeights[g, {2, 2, 5, -3, 1, 3}];
         ShowGraph[SetEdgeLabels[h, GetEdgeWeights[h]], VertexNumber → True,
          VertexNumberPosition → UpperLeft, TextStyle → {FontSize → 11}]
```

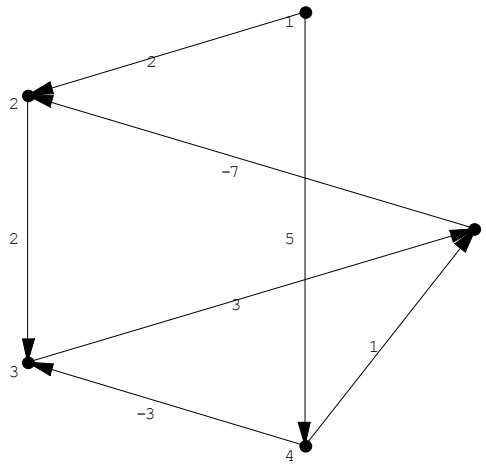



Вычислим остовное дерево кратчайших путей графа t из вершины 1, вычисленное с помощью функций Dijkstra и BellmanFord:

```
In[16]:= {Dijkstra[h, 1], BellmanFord[h, 1]} // ColumnForm
Out[16]= {{1, 5, 4, 1, 4}, {0, -1, 2, 5, 6}}
         {{1, 5, 2, 1, 3}, {0, -5, -1, 5, 2}}
```

Алгоритм Дейкстры показывает, что кратчайший путь из первой вершины к пятой в графе h проходит через четвертую вершину и имеет цену 6, тогда как алгоритм Беллмана-Форда указывает, что путь, проходящий через третью вершину, имеет цену 5.
Прибавление ребра (5,2) с весом (-7) создает отрицательный цикл (2,3,5,2). Функция в BellmanFord[h,1] возвращает ответ, но легко проверить, что ответ неверный, т.к. ребро (2,3) не релаксировано.

```
In[17]:= h = AddEdges[h, {{{5, 2}, EdgeWeight → -7}}];
         {par, dist} = BellmanFord[h, 1]
Out[17]= {{1, 5, 2, 1, 3}, {0, -5, -1, 5, 2}}

In[19]:= ShowLabeledGraph[SetEdgeLabels[h, GetEdgeWeights[h]]]
```




Чтобы определить наличие отрицательных циклов, может быть использован алгоритм Беллмана-Форда.

```
In[20]:= w = GetEdgeWeights[h]; e = Edges[h];
         Table[dist[[e[[i, 1]]]] + w[[i]] ≤ dist[[e[[i, 2]]]], {i, Length[e]}]
```

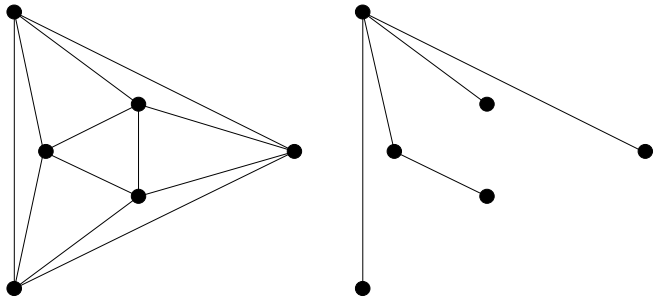
```
Out[20]= {False, True, True, False, False, True, True}
```

Алгоритмы Беллмана-Форда и Дейкстры одинаково быстро работают на редких графах и плотных графах, а на больших и плотных графах быстрее работает функция BellmanFord

```
In[21]:= p = SetEdgeWeights[GridGraph[20, 20]]; q = CompleteGraph[200];
         {{Timing[Dijkstra[p, 1];], Timing[BellmanFord[p, 1];]},
          {Timing[Dijkstra[q, 1];], Timing[BellmanFord[q, 1];]}}
Out[21]= {{{0.485 Second, Null}, {0.171 Second, Null}},
          {{1.375 Second, Null}, {2.422 Second, Null}}}
```

Как уже упоминалось, функция ShortestPathSpanningTree производит остовное дерево кратчайших путей из источника ко всем вершинам графа. Установка опции на Algorithm –> Automatic позволяет функции определить, будет ли использоваться алгоритм Дейкстры или Беллмана-Форда. Она выбирает решение, основываясь на существовании отрицательных весов ребер и разреженности графов.
Рассмотрим теперь остовное дерево евклидовых кратчайших путей от вершины 1 ко всем другим вершинам в графе октаэдра:

```
In[22]:= g = OctahedralGraph;
         t = ShortestPathSpanningTree[SetEdgeWeights[g, WeightingFunction → Euclidean], 1];
         ShowGraphArray[{g, t}, VertexStyle → Disk[0.05]]
```

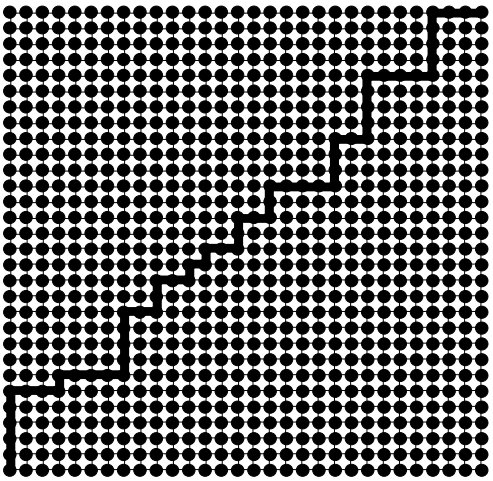


Функция ShortestPath[g, start, end] находит кратчайшие пути от вершины start до вершины end. Функция допускает опцию Algorithm, которая может принимать значения Automatic, Dijkstra или BellmanFord. По умолчанию Algorithm –> Automatic. В этом случае, в зависимости от наличия отрицательных весов или в зависимости от плотности графа, алгоритм выбирает между алгоритмом Дейкстры или алгоритмом Беллмана-Форда.
Рассмотрим кратчайший путь в решетчатом 30×30-решетчатом графе от первой вершины до девятисотой:

```
In[23]:= p = ShortestPath[g = SetEdgeWeights[GridGraph[30, 30]], 1, 900]
Out[23]= {1, 31, 32, 62, 92, 122, 152, 182, 212, 211, 241, 242, 243, 273, 303, 333,
          363, 393, 423, 453, 483, 513, 543, 573, 603, 604, 605, 606, 607, 608, 609,
          610, 611, 612, 613, 614, 615, 645, 675, 676, 677, 678, 679, 709, 710, 711,
          741, 742, 772, 773, 774, 804, 834, 864, 865, 866, 896, 897, 898, 899, 900}
```

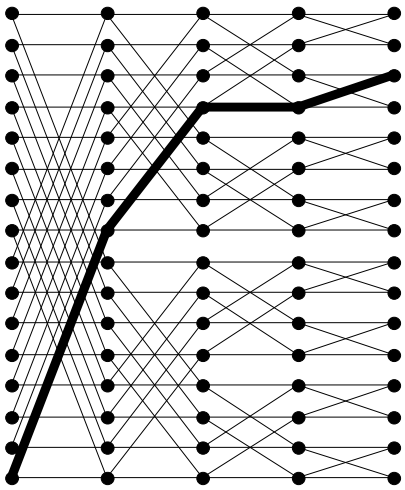
Выделим полученный кратчайший путь от нижнего левого к правому верхнему углу в решетчатом графе со случайными весами ребер. Т.к. веса ребер случайны, нет гарантии, что путь будет проходить всегда вверх или вправо.

```
In[24]:= ShowGraph[Highlight[g, {Map[Sort, Partition[p, 2, 1]]}]]
```



Выделим кратчайший путь между вершиной 0 уровня и вершиной r-го уровня в r-мерном графе бабочки. Путь проходит каждый уровень ровно один раз

```
In[24]:= ShowGraph[Highlight[g=ButterflyGraph[4],{Partition[Short
         estPath[g,1,70],2,1]}]]
```



Теперь рассмотрим еще один вид графов - Shuffle-Exchange графы.

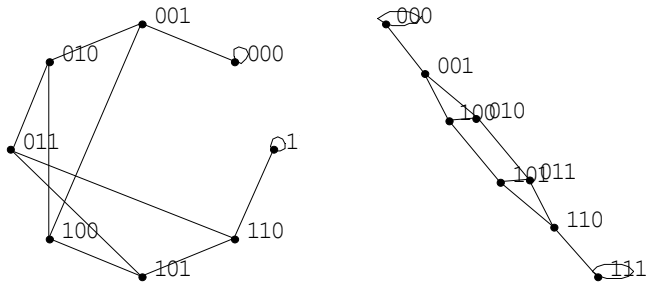
r-мерный shuffle-exchange граф имеет $n=2^r$ вершин и $3 \times 2^{r-1}$ ребер. Каждая вершина соответствует r-битовой бинарной строке, а две вершины u и v связаны ребром тогда и только тогда, когда

1. u и v отличаются только в последнем бите или
2. u являются левым или правым циклическим сдвигом v.

Если u и v отличаются в последнем бите, ребро называется обменным ребром, в противном случае, ребро называется перетасовочным ребром.

Рассмотрим, например, циркулярное вложение трехмерного shuffle-exchange графа и его SpringEmbedding:

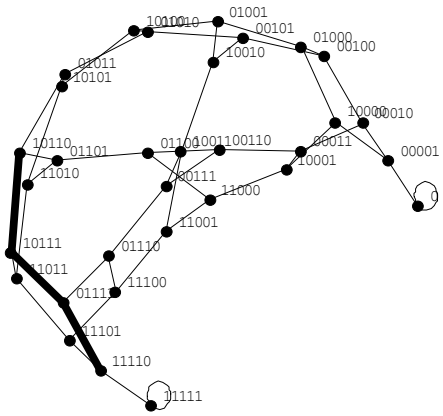
```
In[25]:= ShowGraphArray[{g=ShuffleExchangeGraph[3,VertexLabel->Tr
         ue], SpringEmbedding[g,100]},PlotRange->0.1]
```




Выделим кратчайший путь между парой случайно выбранных вершин в пятимерном shuffle-exchange графе.

```
In[26]:= g=ShuffleExchangeGraph[5,VertexLabel->True];s=Random[Int
         eger,{1,32}];t=Random[Integer,{1,32}];GetVertexLabels[g]
         [[ShortestPath[g,s,t]]]
Out[26]= {11110, 01111, 10111, 10110}
```

```
In[27]:= ShowGraph[SpringEmbedding[
           Highlight[g, {Partition[ShortestPath[g, s, t], 2, 1]}]]]
```

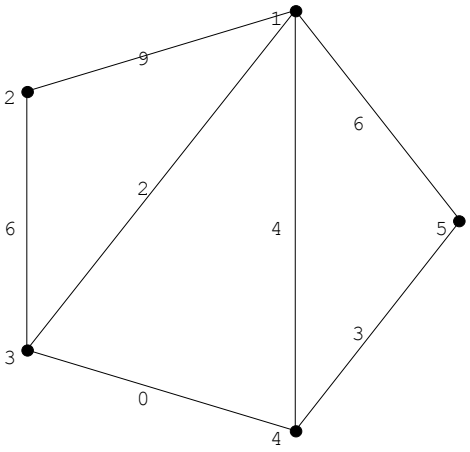


Многим приложениям требуется знать длины кратчайших путей между всеми парами вершин заданного графа. Например, если нам нужно найти диаметр графа – кратчайший путь между всеми парами вершин наибольшей длины.

Длины кратчайших путей могут быть вычислены повторениями алгоритма Дейкстры или алгоритма Беллмана-Форда для каждой из n возможных начальных вершин. Но Combinatorica предлагает гораздо более изящный алгоритм Флойда-Уоршелла для построения матрицы расстояний на основании начальной матрицы весов. Алгоритм Флойда-Уоршелла применяется в динамическом программировании.

Функция AllPairsShortestPath[g] возвращает матрицу, чей (i, j)-й элемент – длина кратчайшего пути в графе g между вершинами i и j. AllPairsShortestPath[g, Parent] возвращает трехмерную матрицу размера 2 * V[g] * V[g], в которой (1, i, j)-й элемент есть длина кратчайшего пути от i до j, а (2, i, j)-й элемент есть отец вершины j в кратчайшем пути от i до j. Эти функции работают на алгоритме Флойда-Уоршелла. Рассмотрим матрицу кратчайших путей в графе g, веса ребер которого – случайные целые числа из отрезка [0,10]:

```
In[25]:= g = SetEdgeWeights[AddEdges[Cycle[5], {{1, 4}, {1, 3}}],
            WeightingFunction → RandomInteger, WeightRange → {0, 10}];
          ShowLabeledGraph[SetEdgeLabels[g, GetEdgeWeights[g]], TextStyle → {FontSize → 12}]
```




Выделим матрицу кратчайших путей:

```
In[27]:= (s = AllPairsShortestPath[g]) // TableForm
```

Out[27]//TableForm=

| | | | | |
|---|---|---|---|---|
| 0 | 8 | 2 | 2 | 5 |
| 8 | 0 | 6 | 6 | 9 |
| 2 | 6 | 0 | 0 | 3 |
| 2 | 6 | 0 | 0 | 3 |
| 5 | 9 | 3 | 3 | 0 |

Например, длина кратчайшего пути от второй вершины к пятой равна 11. Применяя тэг Parent, помимо матрицы кратчайших путей получим и родительскую матрицу:

```
In[28]:= AllPairsShortestPath[g, Parent] // TableForm
```

Out[28]//TableForm=

| | | | | |
|---|---|---|---|---|
| 0 | 8 | 2 | 2 | 5 |
| 8 | 0 | 6 | 6 | 9 |
| 2 | 6 | 0 | 0 | 3 |
| 2 | 6 | 0 | 0 | 3 |
| 5 | 9 | 3 | 3 | 0 |
| 1 | 3 | 3 | 3 | 3 |
| 3 | 2 | 3 | 3 | 3 |
| 4 | 4 | 3 | 4 | 4 |
| 3 | 3 | 3 | 4 | 5 |
| 4 | 4 | 4 | 4 | 5 |

Из родительской матрицы мы можем узнать, что в кратчайшем пути из первой вершины до третьей родителем третьей вершины является четвертая вершина.
Посмотрим, для каких пар вершин, длина полученных кратчайших путей короче, чем длина прямого ребра, соединяющего эти вершины. Это следующие пары {1,5} и {5,1}

```
In[29]:= (ToAdjacencyMatrix[g, EdgeWeight] - s) /. Infinity → 0 // TableForm
```

Out[29]//TableForm=

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 0 | 2 | 1 |
| 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 |

Функция CostOfPath[g, p] суммирует веса всех ребер пути p графа g.
Просчитаем цены некоторых путей:

```
In[30]:= {CostOfPath[g, {1, 4, 5}], CostOfPath[g, {1, 5}]}
Out[30]= {7, 6}
```

Функция AllPairsShortestPath работает гораздо быстрее, чем алгоритм Дейкстры:

```
In[31]:= g = RandomGraph[100, 0.6];
          {Timing[Table[Dijkstra[g, i], {i, 1, V[g]}];][[1, 1]],
           Timing[AllPairsShortestPath[g];][[1, 1]]}
Out[31]= {26.515, 2.}
```

Как уже упоминалось (u,v)-й элемент k-й степени матрицы смежности графа g дает число путей длины k между вершинами u,v. Применяя это свойство матрицы смежности, работают функции NumberOfKPaths:
Функция NumberOfKPaths[g, v, k] возвращает список, который содержит число путей длины k от вершины до всех других вершин графа g.
NumberOfKPaths[al, v, k] делает то же самое, только вместо графа берет матрицу смежности al.
NumberOf2Paths[g, v] возвращает список, который содержит число путей длины 2 от вершины до всех других вершин графа g.

```
In[32]:= {NumberOfKPaths[g, 1, 3], NumberOf2Paths[g, 1]}
Out[32]= {{1392, 1406, 1430, 1446, 1471, 1493, 1519, 1551, 1570, 1572, 1581, 1595, 1602, 1603,
           1614, 1627, 1629, 1634, 1641, 1647, 1652, 1658, 1660, 1673, 1674, 1682, 1684,
           1686, 1687, 1687, 1692, 1698, 1700, 1705, 1705, 1707, 1708, 1720, 1722, 1727,
           1732, 1742, 1744, 1746, 1747, 1749, 1750, 1752, 1753, 1753, 1761, 1762,
           1763, 1767, 1768, 1769, 1779, 1781, 1790, 1791, 1796, 1798, 1799, 1802,
           1811, 1812, 1822, 1825, 1826, 1826, 1828, 1829, 1839, 1840, 1848, 1850,
           1856, 1860, 1864, 1864, 1866, 1869, 1880, 1888, 1890, 1897, 1900, 1900,
           1917, 1919, 1924, 1939, 1940, 1944, 1976, 1978, 2037, 2075, 2117, 2147},
          {22, 22, 23, 23, 24, 24, 24, 25, 25, 25, 25, 25, 25, 26, 26, 26, 26, 26, 26, 26,
           27, 27, 27, 27, 27, 27, 27, 27, 27, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28,
           29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 30, 30, 30, 30, 30, 30,
           30, 30, 30, 30, 30, 30, 30, 30, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31,
           32, 32, 32, 32, 32, 32, 32, 32, 33, 33, 34, 35, 35, 35, 35, 35, 35, 36, 37, 48}}
```

Рассмотрим теперь матрицу кратчайших путей гиперкуба:

```
In[33]:= AllPairsShortestPath[Hypercube[4]]
Out[33]= {{0, 1, 2, 1, 1, 2, 3, 2, 2, 3, 4, 3, 1, 2, 3, 2}, {1, 0, 1, 2, 2, 1, 2, 3, 3, 2, 3, 4, 2, 1, 2, 3},
          {2, 1, 0, 1, 3, 2, 1, 2, 4, 3, 2, 3, 3, 2, 1, 2}, {1, 2, 1, 0, 2, 3, 2, 1, 3, 4, 3, 2, 2, 3, 2, 1},
          {1, 2, 3, 2, 0, 1, 2, 1, 1, 2, 3, 2, 2, 3, 4, 3}, {2, 1, 2, 3, 1, 0, 1, 2, 2, 1, 2, 3, 3, 2, 3, 4},
          {3, 2, 1, 2, 2, 1, 0, 1, 3, 2, 1, 2, 4, 3, 2, 3}, {2, 3, 2, 1, 1, 2, 1, 0, 2, 3, 2, 1, 3, 4, 3, 2},
          {2, 3, 4, 3, 1, 2, 3, 2, 0, 1, 2, 1, 1, 2, 3, 2}, {3, 2, 3, 4, 2, 1, 2, 3, 1, 0, 1, 2, 2, 1, 2, 3},
          {4, 3, 2, 3, 3, 2, 1, 2, 2, 1, 0, 1, 3, 2, 1, 2}, {3, 4, 3, 2, 2, 3, 2, 1, 1, 2, 1, 0, 2, 3, 2, 1},
          {1, 2, 3, 2, 2, 3, 4, 3, 1, 2, 3, 2, 0, 1, 2, 1}, {2, 1, 2, 3, 3, 2, 3, 4, 2, 1, 2, 3, 1, 0, 1, 2},
          {3, 2, 1, 2, 4, 3, 2, 3, 3, 2, 1, 2, 2, 1, 0, 1}, {2, 3, 2, 1, 3, 4, 3, 2, 2, 3, 2, 1, 1, 2, 1, 0}}
```

Подсчитаем частоту ненулевых элементов в первой строке этой матрицы:

```
In[34]:= Distribution[First[AllPairsShortestPath[Hypercube[4]]]]
Out[34]= {1, 4, 6, 4, 1}
```

Обобщим этот результат и получим биномиальные коэффициенты:

```
In[35]:= Table[Distribution[First[AllPairsShortestPath[Hypercube[i]]]], {i, 0, 7}] // TableForm
```

```
Out[35]//TableForm=
1
1   1
1   2   1
1   3   3   1
1   4   6   4   1
1   5   10  10  5   1
1   6   15  20  15  6   1
1   7   21  35  35  21  7   1
```

| | |
|---|---|
| Dijkstra[g, v] | возвращает остовное дерево кратчайших путей и соответствующие расстояния от вершины v графа g. Остовное дерево кратчайших путей задается списком, в котором элемент i – родитель вершины i в остовном дереве кратчайших путей |
| BellmanFord[g, v] | возвращает остовное дерево кратчайших путей и соответствующие расстояния от вершины v графа g. Остовное дерево кратчайших путей задается списком, в котором элемент i – родитель вершины i в остовном дереве кратчайших путей. |
| ShortestPathSpanningTree[g, v] | конструирует остовное дерево кратчайших путей с корнем в вершине v так, что кратчайший путь от v к любой другой вершине есть путь в этом дереве. Функция допускает опцию Algorithm, которая может принимать значения Automatic, Dijkstra или BellmanFord. По умолчанию Algorithm -> Automatic. В этом случае, в зависимости от наличия отрицательных весов или в зависимости от плотности графа, алгоритм выбирает между алгоритмом Дейкстры или алгоритмом Беллмана-Форда |
| ShortestPath[g, start, end] | находит кратчайшие пути от вершины start до вершины end. Функция допускает опцию Algorithm, которая может принимать значения Automatic, Dijkstra, или BellmanFord. По умолчанию Algorithm -> Automatic. В этом случае, в зависимости от наличия отрицательных весов или в зависимости от плотности графа, алгоритм выбирает между алгоритмом Дейкстры или алгоритмом Беллмана- Форда |
| AllPairsShortestPath[g] | возвращает матрицу, чей (i, j)-й элемент – длина кратчайшего пути в графе g между вершинами i и j |
| AllPairsShortestPath[g, Parent] | возвращает трехмерную матрицу размера 2 * V[g] * V[g], в которой (1, i, j)-й элемент есть длина кратчайшего пути от i до j, а (2, i, j)-й элемент есть отец вершины j в кратчайшем пути от i до j |
| NumberOfKPaths[g, v, k] | возвращает список, который содержит число путей длины k от вершины до всех других вершин графа g |
| NumberOfKPaths[al, v, k] | делает то же самое, только вместо графа берет матрицу смежности al |
| NumberOf2Paths[g, v] | возвращает список, который содержит число путей длины 2 от вершины до всех других вершин графа g |

### *7.1.2. Метрические характеристики графа*

Некоторые инварианты графов связаны с матрицей кратчайших путей.
Пусть g – связный граф, u, v – две его несовпадающие вершины.
Длина кратчайшего (u,v)-маршрута называется **расстоянием** между u и v. Обозначим эту величину d(u,v). Если граф невзвешенный, то по умолчанию каждому ребру присваивается вес, равный 1, и расстояние между вершинами u и v – число ребер в (u,v)-маршруте.

Функция Distances[g, v] возвращает в неубывающем порядке расстояния от вершины v до всех вершин графа g, при этом g интерпретируется как невзвешенный граф.

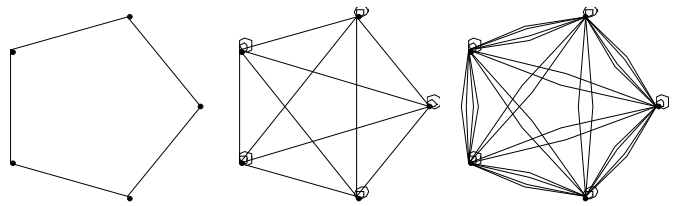
```
In[2]:= Distances[PetersenGraph, 5]

Out[2]= {0, 1, 1, 1, 2, 2, 2, 2, 2, 2}
```

Понятие расстояния между вершинами в связном графе позволяет определить k-ю степень графа $g^k$. Пусть g – связный граф, $k \in \mathbb{N}$.

1. Граф $g^k$ имеет то же множество вершин, что и g.
2. Вершины u,v смежны в $g^k$ тогда и только тогда, когда в g выполняется неравенство $d(u,v) \le k$.

Очевидно, если $k \ge |g|-1$, то $g^k$ - полный граф.
Функция GraphPower[g, k] возвращает k-ю степень графа

```
In[3]:= ShowGraphArray[Table[GraphPower[Cycle[5], i], {i, 1, 3}]]
```



Оказывается, что куб любого связного графа является гамильтоновым

```
In[4]:= HamiltonianQ[GraphPower[RandomTree[20], 3]]

Out[4]= True
```

Для фиксированной вершины u величина e(u)=maxd(u,v), где max берется по всем вершинам графа g, называется **эксцентриситетом** вершины u. Максимальный среди всех эксцентриситетов вершин называется **диаметром** графа g и обозначается d(g). Минимальный из эксцентриситетов вершин графа g называется **радиусом** g и обозначается через r(g).
Функция Eccentricity[g] возвращает эксцентриситет каждой вершины v графа g. Diameter[g], Radius[g] – выдают диаметр и радиус графа g соответственно

```
In[5]:= Eccentricity[Star[10]]

Out[5]= {2, 2, 2, 2, 2, 2, 2, 2, 2, 1}
```

```
In[6]:= {Diameter[Star[10]], Radius[Star[10]]}

Out[6]= {2, 1}
```

Вершина v называется центральной, если e(v)=r(g). Множество всех центральных вершин называется центром графа.
Граф может иметь единственную центральную вершину или несколько центральных вершин. Функция GraphCenter[g] возвращает список центральных вершин графа.
Центр простой цепи $P_n$ при четном n состоит ровно из двух вершин, а при нечетном n из одной вершины.

```
In[7]:= {GraphCenter[Path[10]], GraphCenter[Path[11]]}

Out[7]= {{5, 6}, {6}}
```

Для цикла $C_n$ все вершины центральные

```
In[8]:= GraphCenter[Cycle[10]]

Out[8]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

Центр любого дерева состоит из одной или двух смежных вершин:

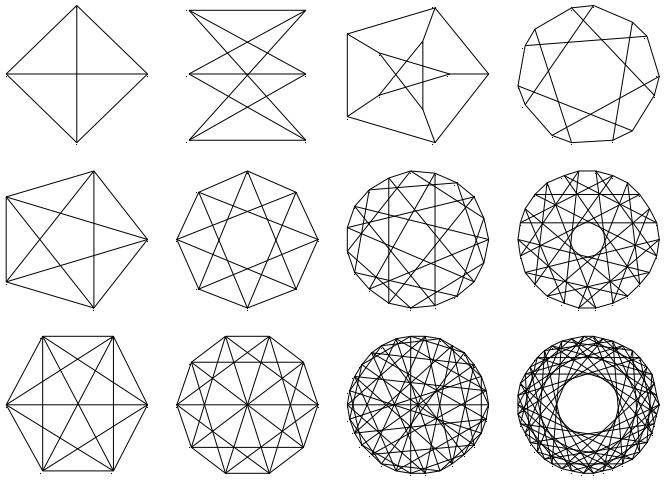
```
In[9]:= Table[GraphCenter[RandomTree[i]], {i, 1, 20}]

Out[9]= {{1}, {1, 2}, {2}, {1, 4}, {3, 4}, {4}, {5}, {8}, {6, 8}, {7},
        {2}, {6}, {10, 11}, {4}, {14}, {1, 5}, {7, 8}, {9, 13}, {8}, {20}}
```

Задача нахождения центральных вершин графа часто возникают в практической деятельности. Например, пусть g – граф дорог. Нужно оптимально разместить пункты обслуживания населения. В подобных ситуациях критерий оптимальности часто заключается в оптимизации наихудшего случая, то есть в минимизации расстояния от места обслуживания до наиболее удаленного пункта. Следовательно, местами размещения должны быть центральные вершины графа.
Под длиной (или весом) любого подграфа будем понимать сумму весов его ребер.
Минимальная из длин циклов называется его **обхватом.**
Функция Girth[g] вычисляет обхват простого графа g.
Обхват дерева есть ∞, т.к. кратчайшего цикла в графе дерева не существует

```
In[10]:= Girth[RandomTree[10]]

Out[10]= ∞
```

В Combinatorica встроенная функция CageGraph[k, r], которая конструирует наименьший k-регулярный граф с обхватом r для некоторых малых значений k и r. CageGraph[r] возвращает CageGraph[3, r]. Для k = 3, r может принимать значения 3, 4, 5, 6, 7, 8, или 10. Для k = 4 или 5, r может быть равным 3, 4, 5, или 6. CageGraph[k, r] известны для малых значений k,r.
Приведем таблицу (k,r)-клеточных графов для k=3,4,5 и r=3,4,5,6. Первые три графа в первой строке уже знакомые нам 3-регулярные графы: $K_4$, $K_{3,3}$ и граф Петерсена:

```
In[11]:= ShowGraphArray[p = Table[CageGraph[k, r], {k, 3, 5}, {r, 3, 6}]]
```



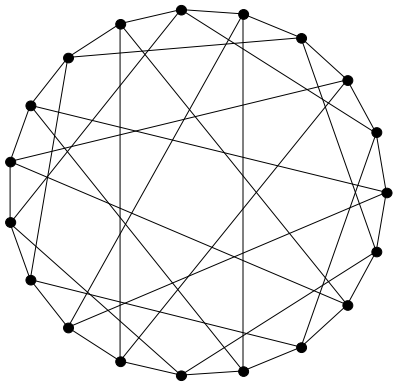
Проверим, что построенные графы действительно удовлетворяют свойству регулярности и имеют корректные степень и обхват.

```
In[12]:= Map[{RegularQ[#], Degrees[#][[1]], Girth[#]} &, Flatten[p, 1]] // ColumnForm
```

```
Out[12]= {True, 3, 3}
         {True, 3, 4}
         {True, 3, 5}
         {True, 3, 6}
         {True, 4, 3}
         {True, 4, 4}
         {True, 4, 5}
         {True, 4, 6}
         {True, 5, 3}
         {True, 5, 4}
         {True, 5, 5}
         {True, 5, 6}
```

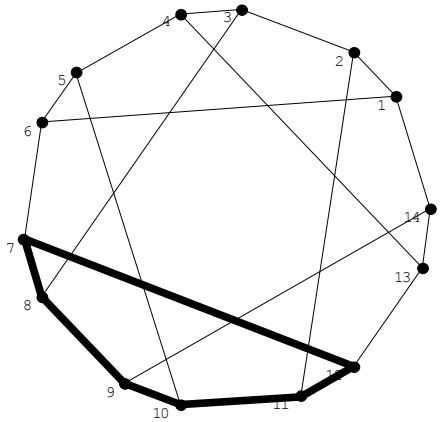
Combinatorica обеспечена рядом встроенных функций, возвращающих непараметризованные клеточные графы, которые являются элементами списка FiniteGraphs:
Встроенная функция RobertsonGraph возвращает 19-вершинный граф, это единственный (4,5)-клеточный граф:

```
In[13]:= ShowGraph[RobertsonGraph]
```



Встроенная функция HeawoodGraph возвращает единственный (6,3)-клеточный граф, 3-регулярный граф, чей обхват равен 6. Выделим в этом графе наибольший цикл:

```
In[14]:= ShowLabeledGraph[Highlight[HeawoodGraph, {Partition[{10, 9, 8, 7, 12, 11, 10}, 2, 1]}]]
```

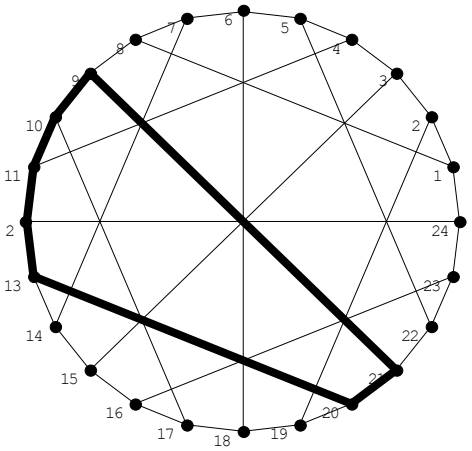



```
In[15]:= RegularQ[HeawoodGraph]

Out[15]= True
```

```
In[16]:= Degrees[HeawoodGraph]

Out[16]= {3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3}

In[17]:= Girth[HeawoodGraph]
Out[17]= 6
```

Встроенная функция McGeeGraph возвращает единственный (7,3)-клеточный граф, 3-регулярный граф, чей обхват равен 7:

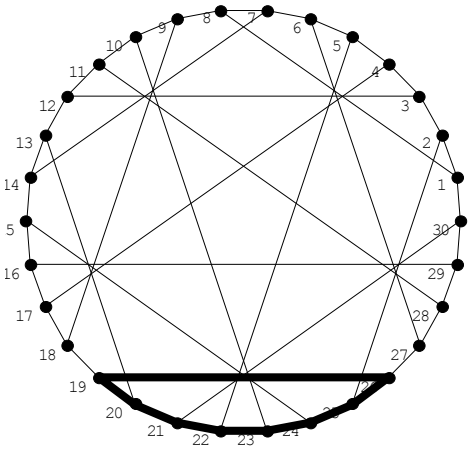
```
In[18]:= ShowLabeledGraph[
          Highlight[McGeeGraph, {Partition[{13, 12, 11, 10, 9, 21, 20, 13}, 2, 1]}]]
```




```
In[19]:= {RegularQ[McGeeGraph], Degrees[McGeeGraph]} // ColumnForm
Out[19]= True
         {3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3}
In[20]:= Girth[McGeeGraph]
Out[20]= 7
```

Встроенная функция LeviGraph возвращает единственный (8,3)-клеточный граф, 3-регулярный граф, чей обхват равен 8:

```
In[21]:= ShowLabeledGraph[
          Highlight[LeviGraph, {Partition[{22, 21, 20, 19, 26, 25, 24, 23, 22}, 2, 1]}]]
```




```
In[22]:= {RegularQ[LeviGraph], Degrees[LeviGraph], Girth[LeviGraph]} //
          ColumnForm
```

```
Out[22]= True
         {3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3}
         8
```

Пусть g – некоторый граф, $A_{ij}$ – его матрица смежности. Очевидно, что если графы изоморфны, то их матрицы смежности подобны, поэтому равны характеристические полиномы этих матриц. Таким образом, корректно определяется характеристический полином графа как характеристический полином его матрицы смежности. Спектр этой матрицы называется **спектром** графа. Как известно из линейной алгебры, вещественная симметрическая матрица определяется своим спектром однозначно, с точностью до подобия. Тем не менее из совпадения характеристических полиномов графов отнюдь не следует изоморфизм графов. Неизоморфные графы с равными характеристическими полиномами называются **коспектральными.**
Спектр полного графа на n вершинах состоит из {n-1, -1,….,-1}:

```
In[21]:= Map[Spectrum, Table[CompleteGraph[i], {i, 1, 11}]] // ColumnForm
Out[21]= {0}
         {-1, 1}
         {2, -1, -1}
         {3, -1, -1, -1}
         {4, -1, -1, -1, -1}
         {5, -1, -1, -1, -1, -1}
         {6, -1, -1, -1, -1, -1, -1}
         {7, -1, -1, -1, -1, -1, -1, -1}
         {8, -1, -1, -1, -1, -1, -1, -1, -1}
         {9, -1, -1, -1, -1, -1, -1, -1, -1, -1}
         {10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}
```

| | |
|---|---|
| Distances[g, v] | возвращает в неубывающем порядке расстояния от вершины v до всех вершин графа g, при этом g интерпретируется как неориентированный граф |
| GraphPower[g, k] | возвращает k-ю степень графа g |
| Eccentricity[g] | возвращает эксцентриситет каждой вершины v графа g |
| Diameter[g] | возвращает диаметр графа g |
| Radius[g] | возвращаетт радиус графа g |
| GraphCenter[g] | возвращает список центральных вершин графа |
| Spectrum[g] | возвращает спектр графа g |
| Girth[g] | вычисляет обхват простого графа g |
| CageGraph[k, r] | конструирует наименьший k-регулярный граф с обхватом r для некоторых малых значений k и r |
| CageGraph[r] | возвращает CageGraph[3, r] |
| RobertsonGraph | возвращает единственный (4,5)-клеточный граф |
| HeawoodGraph | возвращает единственный (6,3)-клеточный граф, 3-регулярный граф, чей обхват равен 6 |
| McGeeGraph | возвращает единственный (7,3)-клеточный граф, 3-регулярный граф, чей обхват равен 7 |
| LeviGraph | возвращает единственный (8,3)-клеточный граф, 3-регулярный граф, чей обхват равен 8 |

## ■7.2. Остов минимального веса

### *7.2.1. Алгоритм Краскала*

Рассмотрим следующую задачу: Во взвешенном связном графе найти остов минимального веса. Эта задача возникает при проектировании линий электропередач, трубопроводов, дорог и т.д., когда требуется заданные соединить центры некоторой системой каналов связи так, чтобы любые два центра были связаны либо непосредственно соединяющим их каналом, либо через другие центры и каналы, и чтобы общая длина была минимальной. В этой ситуации заданные центры можно считать вершинами полного графа с весами ребер, равными длинам (стоимости) соединяющих эти центры каналов. Тогда искомая сеть будет кратчайшим остовным подграфом полного графа. Очевидно, что этот кратчайший остовной подграф должен быть деревом. Т.к. полный граф $K_n$ содержит $n^{n-2}$ различных остовных деревьев, то решение этой задачи «слепым» перебором вариантов потребовало бы чрезвычайно больших вычислений даже при малых n. Однако для ее решения существуют эффективные алгоритмы. Опишем два из них – алгоритм Краскала (1956) и Прима (1957), применимые к произвольному связному графу.

Рассмотрим алгоритм Краскала.

1. Строим граф $O_n+\{e_1\}=T_1$, присоединяя к пустому графу на множестве V ребро $e_1$ минимального веса.
2. Если граф $T_i$ уже построен и $i<n-1$, то строим граф $T_{i+1} =T_i +\{e_{i+1}\}$, где $e_{i+1}$ – ребро графа g, имеющее минимальный вес среди ребер, не входящих в $T_i$ и не составляющих цикла с ребрами из $T_i$.

Следующая теорема утверждает, что алгоритм Краскала всегда приводит к остову минимального веса.

**Теорема**. При $i<n-1$ граф $T_{i+1}$ можно построить. Граф $T_{n-1}$ является остовом минимального веса в графе g.

Алгоритм Прима отличается от алгоритма Краскала только тем, что на каждом этапе строится не просто ациклический граф, но дерево. Именно:

1. Выбираем ребро $e_i=\{a,b\}$ минимального веса и строим дерево $T_1$.
2. Если дерево $T_i$ порядка i+1 уже построено и $i<n-1$, то среди ребер, соединяющих вершины этого дерева с вершинами графа g, не входящими в $T_i$ , выбираем ребро $e_{i+1}$ минимального веса. Строим дерево $T_{i+1}$, присоединяя к $T_i$ ребро $e_{i+1}$ вместе с его не входящим в $T_i$ концом. Для этого алгоритма также верна вышеприведенная теорема.

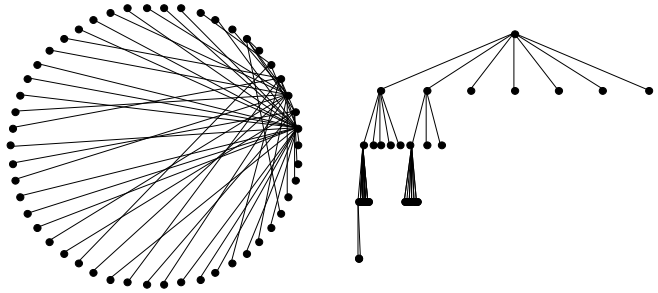
В некоторых ситуациях нужно построить остов не минимального, а максимального веса. К этой задаче также применимы алгоритмы Краскала и Прима. Следует только заменить минимальный вес на максимальный.

Функция MinimumSpanningTree[g] находит минимальное остовное дерево графа g. MaximumSpanningTree[g] находит максимальное остовное дерево графа g.

Функции используют алгоритм Краскала.

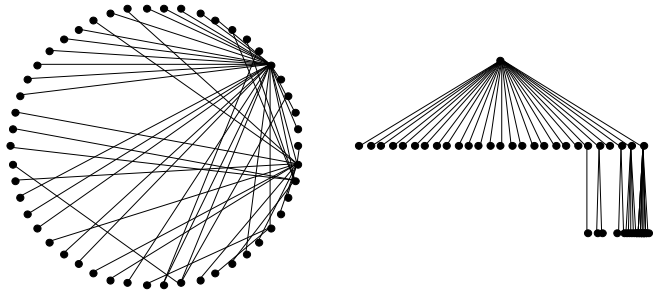
Рассмотрим минимальное остовное дерево случайного графа на пятидесяти вершинах.

```
In[2]:= g = RandomGraph[50, 0.5]; ShowGraphArray[{t = MinimumSpanningTree[g], RootedEmbedding[t]}]
```



А теперь выделим остовное дерево кратчайших путей этого же графа:

```
In[3]:= ShowGraphArray[{p = ShortestPathSpanningTree[g, 5], RootedEmbedding[p]}]
```

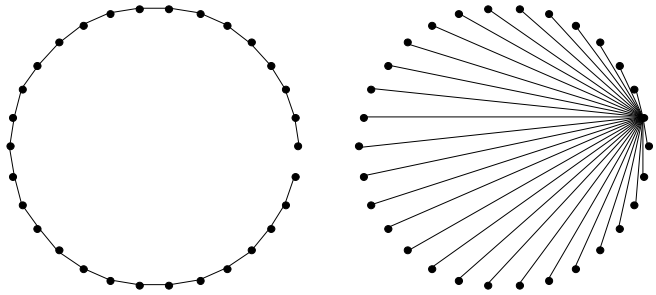


Вычислим длины минимального остовного дерева и остовного дерева кратчайших путей:

```
In[4]:= {Apply[Plus, GetEdgeWeights[g, Edges[t]]], Apply[Plus, GetEdgeWeights[g, Edges[p]]]}
Out[4]= {49, 49}
```

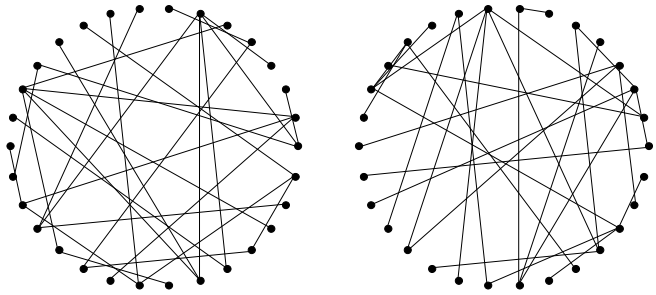
Евклидово минимальное остовное дерево – это минимальное остовное дерево полного графа, чьи вершины - точки евклидова пространства и вес каждого ребра – евклидово расстояние между точками. Евклидово минимальное остовное дерево для множества точек, равномерно расположенных на окружности, всегда является путем вдоль окружности, тогда как каждое остовное дерево наикратчайших путей есть звезда.

```
In[5]:= g = SetEdgeWeights[CompleteGraph[30], WeightingFunction → Euclidean];
        ShowGraphArray[{MinimumSpanningTree[g], ShortestPathSpanningTree[g, 1]}]
```



Один способ произвести случайные остовные деревья - присвоить случайные веса ребрам и произвести алгоритм минимального остовного дерева на нем.

```
In[6]:= g = CompleteGraph[30]; ShowGraphArray[Map[MinimumSpanningTree[SetEdgeWeights[#]] &, {g, g}]]
```



| | |
|---|---|
| MinimumSpanningTree[g] | находит минимальное остовное дерево графа g |
| MaximumSpanningTree[g] | находит максимальное остовное дерево графа g |

### *7.2.2. Число остовных деревьев*

В связном графе остовом служит любое остовное дерево, то есть остовной подграф является деревом. Число остовных деревьев связном графе порядка n>2 по теореме Кирхгофа равно алгебраическому дополнению любого элемента матрицы Кирхгофа. Матрица Кирхгофа графа на т вершинах называется n×n – матрица, элемент (i,j) которой равен -1, если вершины i,j смежны, нулю, если i≠j и вершины i,j смежны, deg[i], если i=j.
Функция Cofactor[m,i,j] считает алгебраическое дополнение элемента (i,j) матрицы m.
Функция NumberOfSpanningTrees[g] считает число помеченных остовных деревьев графа.
Число остовных деревьев полного графа равно $n^{n-2}$, как уже было показано

```
In[2]:= Table[{NumberOfSpanningTrees[CompleteGraph[n]], n^(n-2)}, {n, 4, 10}] // ColumnForm
Out[2]= {16, 16}
        {125, 125}
        {1296, 1296}
        {16807, 16807}
        {262144, 262144}
        {4782969, 4782969}
        {100000000, 100000000}
```

Любое дерево содержит ровно одно остовное дерево

```
In[3]:= Table[NumberOfSpanningTrees[Star[n]], {n, 3, 100}]
Out[3]= {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
         1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
         1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
```

Цикл с n вершинами содержит ровно n остовных деревьев, т.к. удаление любого ребра создает дерево

```
In[4]:= Map[Equal, Table[{NumberOfSpanningTrees[Cycle[n]], M[Cycle[n]]}, {n, 3, 10}]]
Out[4]= {True, True, True, True, True, True, True, True}
```

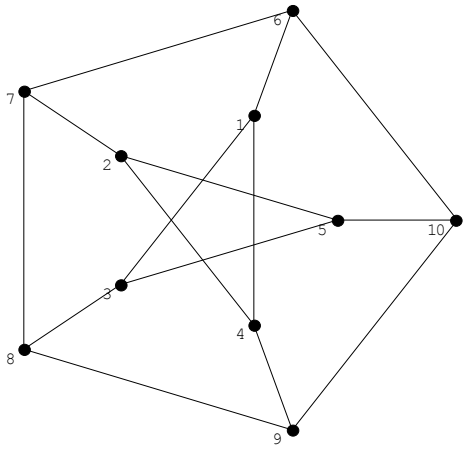
| | |
|---|---|
| NumberOfSpanningTrees[g] | считает число помеченных остовных деревьев графа |
| Cofactor[m,i,j] | считает алгебраическое дополнение элемента (i,j) матрицы m |

## ■7.3. Потоки в сети

Любой реберно-взвешенный граф можно рассматривать как сеть труб, причем вес ребра (i,j) задает пропускную способность трубы. Пропускную способность можно рассматривать как функцию площади поперечного сечения трубы. Для данного взвешенного графа g и двух вершин s и t в задаче потока в сети нужно узнать, какой максимальный поток может быть отправлен из s в t с учетом максимальной пропускной способности каждой трубы.
Более формально, сетевой поток состоит из реберно-взвешенного графа g, вершины источника s и вершины стока t. Вес каждого ребра означает его пропускную способность – максимальное количество потока, которое оно может пропустить. Задача максимального потока находит максимальный возможный поток от s к t, удовлетворяющий пропускной способности каждого ребра. К задаче сетевого потока могут быть легко сведены многие другие важные задачи, включая k-связность и двудольное покрытие.
Традиционно алгоритмы сетевого потока используют идею дополняющих путей Форда и Фалкерсона, которые все время находят дополняющие пути с положительной пропускной способностью из s в t, и добавляет их к текущему потоку.

Функция NetworkFlow[g, source, sink] возвращает значение максимального потока через граф g из source в sink.
NetworkFlow[g, source, sink, Edge] возвращает ребра в g, которые имеют положительный поток вместе с их потоками в максимальном потоке из source в sink.
NetworkFlow[g, source, sink, Cut] возвращает минимальный разрез между source и sink.
NetworkFlow[g, source, sink, All] дает список смежности графа g вместе с потоком на каждом ребре в максимальном потоке из source в sink. Граф g может быть ориентированным или неориентированным.
Рассмотрим граф Петерсена, в котором каждое ребро имеет пропускную способность 1, то есть граф невзвешенный.

```
In[2]:= ShowLabeledGraph[PetersenGraph]
```




Выделим список смежности графа Петерсена:

```
In[3]:= ToAdjacencyLists[PetersenGraph]
Out[3]= {{3, 4, 6}, {4, 5, 7}, {1, 5, 8}, {1, 2, 9},
         {2, 3, 10}, {1, 7, 10}, {2, 6, 8}, {3, 7, 9}, {4, 8, 10}, {5, 6, 9}}
```

А теперь рассмотрим список смежности графа g вместе с потоком на каждом ребре в максимальном потоке из первой вершины в десятую.

```
In[4]:= NetworkFlow[g = PetersenGraph, 1, 10, All] // ColumnForm
Out[4]= {{3, 1}, {4, 1}, {6, 1}}
        {{4, 0}, {5, 0}, {7, 0}}
        {{1, 0}, {5, 1}, {8, 0}}
        {{1, 0}, {2, 0}, {9, 1}}
        {{2, 0}, {3, 0}, {10, 1}}
        {{1, 0}, {7, 0}, {10, 1}}
        {{2, 0}, {6, 0}, {8, 0}}
        {{3, 0}, {7, 0}, {9, 0}}
        {{4, 0}, {8, 0}, {10, 1}}
        {{5, 0}, {6, 0}, {9, 0}}
```

Этот список показывает, что из первой вершины через ребра {1, 3}, {1, 4}, {1, 6} могут быть посланы по единице потока, по ребрам {2,4}, {2,5}, {2,6} нельзя послать ни одной единицы, из третьей вершины только через ребро {3,5} можно послать одну единицу потока и т.д.
Вызывая функцию NetworkFlow, получим, что из первой вершины в десятую могут быть посланы максимум три единицы потока:

```
In[5]:= NetworkFlow[g, 1, 10]
Out[5]= 3
```

Выделим ребра, которые имеют положительный поток вместе с их потоками в максимальном потоке из первой вершины в десятую:

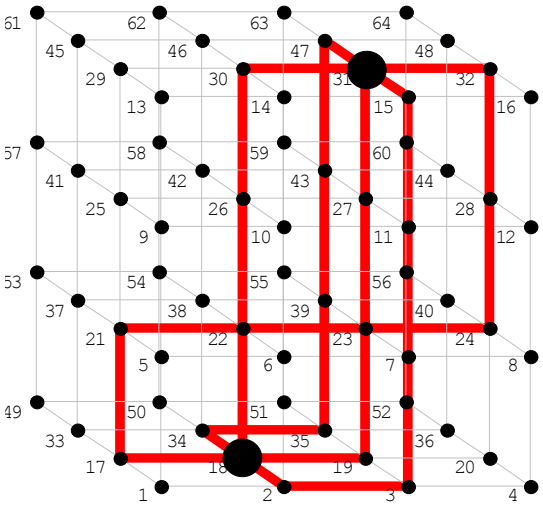
```
In[6]:= NetworkFlow[g, 1, 10, Edge]
Out[6]= {{{1, 3}, 1}, {{1, 4}, 1}, {{1, 6}, 1}, {{3, 5}, 1},
        {{4, 9}, 1}, {{5, 10}, 1}, {{6, 10}, 1}, {{9, 10}, 1}}
```

В любом графе, в котором пропускная способность ребер равна 1 (в невзвешенном графе), максимальный поток от s к t равен числу реберно-непересекающихся путей от s до t. Отметим, что ребра, вдоль которых проходит положительный поток, формирует пути от источника s к стоку t.
Многие задачи могут быть смоделированы как задачи сетевого потока. Проиллюстрируем вычисление числа реберно-непересекающихся путей между парами вершин, используя решение задачи максимального потока.
Рассмотрим 4× 4× 4-решетчатый граф. Подсчитаем, например, сколько единиц потока могут быть посланы вдоль реберно-непересекающихся путей от вершины 18 к вершине 31.

```
In[7]:= g = GridGraph[4, 4, 4]; NetworkFlow[g, 18, 31]
Out[7]= 5
```

NetworkFlow показывает, что от вершины 18 к вершине 31 могут быть посланы пять единиц потока и, следовательно, существует 5 реберно-непересекающихся путей между вершинами 18 и вершиной 31.
Выделим эти пути:

```
In[8]:= ShowLabeledGraph[Highlight[g, {{18, 31}, First[Transpose[NetworkFlow[g, 18, 31, Edge]]]}]]
```



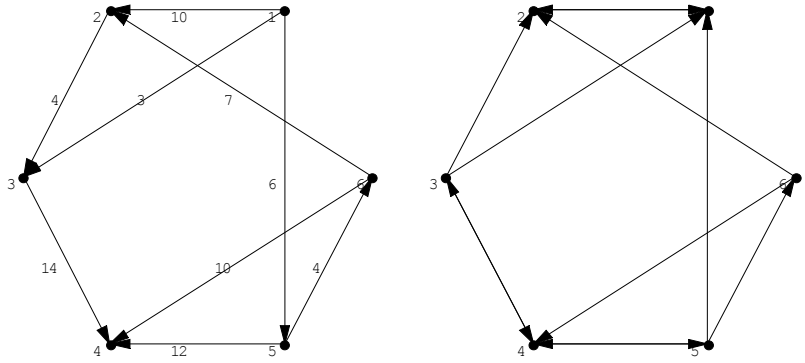

Ключевой структурой в теории сетевых потоков является граф остаточного потока, обозначаемый как R(g,f), где g – вводимый граф, а f – текущий поток через него. Этот ориентированный, реберно-взвешенный R(g,f), имеет те же вершины, что и g, и для каждого ребра (i,j) в g с пропускной способностью c(i,j) и потока f(i,j) R(g,f) может содержать два ребра:
1. ребро (i,j) с весом c(i,j)- f(i,j), если c(i,j)- f(i,j)>0
2. ребро (i,j) с весом f(i,j), если f(i,j)>0
Наличие ребра (i,j) в остаточном графе указывает на то, что из i в j может быть послано положительное количество потока. Вес ребра есть точное количество потока, которое может быть послано.
Функция ResidualFlowGraph[g, flow] возвращает ориентированный граф остаточного потока для графа g относительно потока flow.
Сконструируем ориентированный реберно-взвешенный граф h и вычислим максимальный поток из первой вершины в четвертую.

```
In[9]:= g = AddEdges[EmptyGraph[6, Type → Directed],
          {{1, 2}, {2, 3}, {1, 3}, {3, 4}, {1, 5}, {5, 6}, {5, 4}, {6, 4}, {6, 2}}];
        h = SetEdgeWeights[g, {10, 4, 3, 14, 6, 4, 12, 10, 7}]; h = SetEdgeLabels[h, GetEdgeWeights[h]];
        NetworkFlow[h, 1, 4]
Out[9]= 13
```

NetworkFlow показывает, что от вершины 18 к вершине 31 могут быть посланы 13 единиц потока.

Рассмотрим теперь исходный граф и его граф остаточного потока, соответствующий максимальному потоку от вершины 1 к вершине 4.

```
In[10]:= f = NetworkFlow[h, 1, 4, All];
         ShowGraphArray[{h, ResidualFlowGraph[h, f]}, VertexNumber → True]
```




В графе остаточного потока нет ориентированного пути из первой в четвертую вершину, поскольку наличие такого пути означало бы, что вычисленный поток не максимальный.

Минимальным s-t разрезом в реберно взвешенном графе называется множество ребер с минимальным общим весом, чье удаление разделяет вершины s и t. Теорема о максимальном потоке и минимального разрезе утверждает, что вес минимального s-t разреза равен максимальному потоку, который может быть послан из s к t.

Рассмотрим 1-4 разрез:

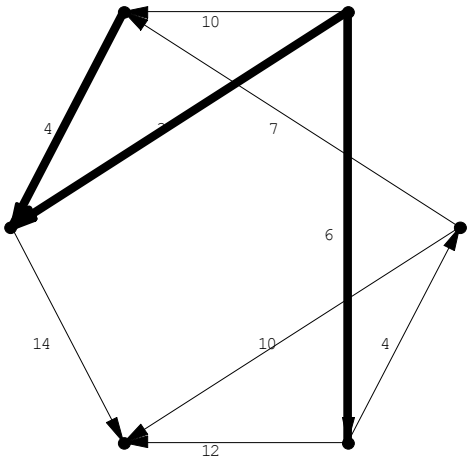
```
In[11]:= NetworkFlow[h, 1, 4, Cut]
Out[11]= {{2, 3}, {1, 3}, {1, 5}}
```

Вычислим вес этого разреза:

```
In[12]:= Apply[Plus, GetEdgeWeights[h, cut]]
Out[12]= 13
```

Выделим этот разрез:

```
In[13]:= cut = NetworkFlow[h, 1, 4, Cut];
         ShowGraph[Highlight[h, {cut}], TextStyle → {FontSize → 10}]
```




| | |
|---|---|
| NetworkFlow[g, source, sink] | возвращает значение максимального потока через граф g из source в sink |
| NetworkFlow[g, source, sink, Edge] | возвращает ребра в g, которые имеют положительный поток вместе с их потоками в максимальном потоке из source в sink |
| NetworkFlow[g, source, sink, Cut] | возвращает минимальный разрез между source и sink |
| | |
| NetworkFlow[g, source, sink, All] | возвращает список смежности графа g вместе с потоком на каждом ребре в максимальном потоке из source в sink. Граф g может быть ориентированным или неориентированным |
| ResidualFlowGraph[g, flow] | возвращает ориентированный граф остаточного потока для графа g относительно потока flow |

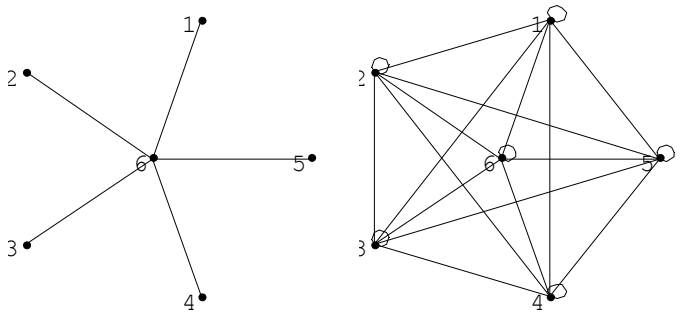
## ■7.4. Частичные порядки

Частичный порядок – это бинарное отношение, которое рефлексивно, транзитивно и антисимметрично. Интересные бинарные отношения между комбинаторными объектами часто бывают частичными порядками.
Функция PartialOrderQ[g] возвращает True, если бинарное отношение, определенное ребрами графа g является частичным порядком. PartialOrderQ[r] возвращает True, если бинарное отношение, заданное матрицей r определяет частичный порядок.

### *7.4.1. Транзитивное замыкание и транзитивная редукция*

Ориентированный граф называется **транзитивным,** если его ребра представляют транзитивное отношение или g=(E,V) транзититивен, если для любых трех вершин x,y,z∈V из существования ребер (x,y), (y,z)∈E следует, что (x,z)∈E. Это определение расширяется на неориентированные графы, если интерпретировать каждое неориентированное ребро как два ориентированных ребра, ориентированных в противоположных направлениях.
Любой граф g может быть сведен к транзитивному графу прибавлением достаточного числа дополнительных ребер. **Транзитивным замыканием** C(g) графа g называется граф, такой, что (u,v) ∈ C(g) тогда и только тогда, когда в g существует ориентированный путь из u в v .
Функция TransitiveClosure[g] возвращает транзитивное замыкание графа g.
Рассмотрим транзитивное замыкание звезды:

```
In[2]:= ShowGraphArray[{Star[6], TransitiveClosure[Star[6]]},
        VertexNumber → True]
```

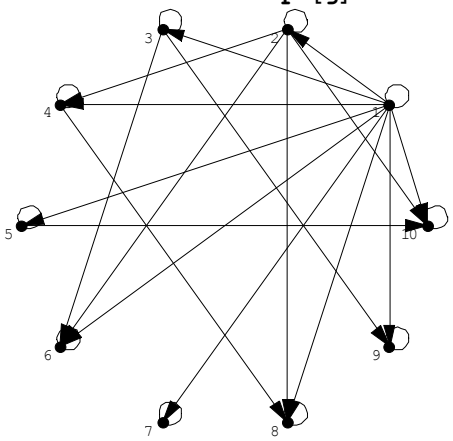



Транзитивное замыкание полного графа без петель есть полный граф:

```
In[3]:= Map[CompleteQ, Map[RemoveSelfLoops[TransitiveClosure[#]] &,
          Table[CompleteGraph[n], {n, 3, 20}]]]
Out[3]= {True, True, True, True, True, True, True, True,
         True, True, True, True, True, True, True, True, True, True}
```

Транзитивной редукцией ориентированного графа g называется граф $g^1=(E^1,V^1)$ с минимальным числом ребер, транзитивное замыкание которого совпадает с графом g.
Отношение делимости на множестве натуральных чисел определяет частичный порядок на этом множестве. Рассмотрим ориентированный граф на десяти вершинах, который определяет это отношение. Так как 1 является делителем любого натурального числа, то в этом графе существуют все ребра, соединяющие первую вершину со всеми остальными.

```
In[4]:= g = MakeGraph[Range[10], (Mod[#2, #1] == 0) &];
        ShowLabeledGraph[g]
```

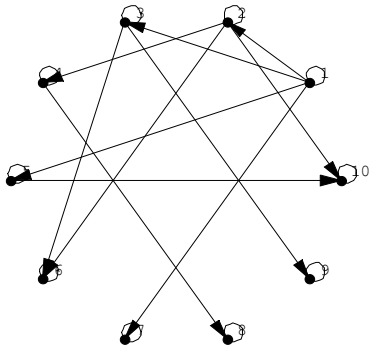



Проверим, что отношение делимости, определяемое ребрами этого графа, действительно является частичным порядком

```
In[5]:= {TransitiveQ[g] && ReflexiveQ[g] && PartialOrderQ[g]}
Out[5]= {True}
```

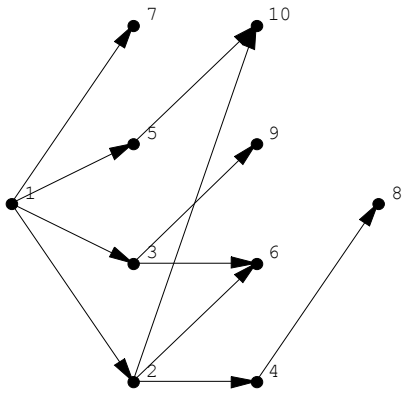
Функция TransitiveReduction[g] возвращает транзитивную редукцию графа g.
Применим TransitiveReduction к вышеприведенному графу. Это устранит ребра, такие как {4,8}, {1,4},{1,6}, {1,8} и т.д.

```
In[6]:= ShowGraph[h = TransitiveReduction[g], VertexNumber → True, VertexNumberPosition → UpperRight,
          TextStyle → {FontSize → 11}, PlotRange → 0.1]
```



Удалим петли и применим ранжированное вложение:

```
In[7]:= ShowLabeledGraph[ RankedEmbedding[RemoveSelfLoops[h], {1}], VertexNumberPosition→ UpperRight,
        TextStyle→ {FontSize→ 11}, PlotRange→ 0.1]
```




Транзитивная редукция, примененная  к редуцированным графам, не меняет его:

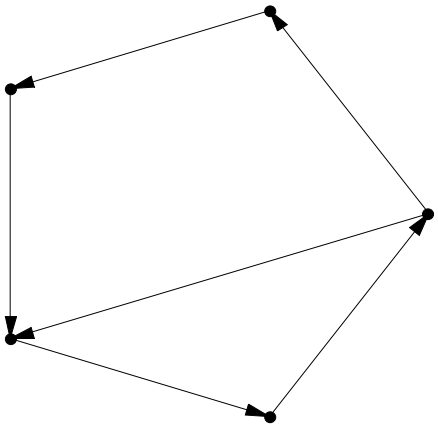
```
In[8]:= IdenticalQ[h, TransitiveReduction[h]]
Out[8]= True
```

Транзитивное замыкание транзитивной редукции графа есть тождественный оператор

```
In[9]:= IdenticalQ[g, TransitiveClosure[TransitiveReduction[h]]]
Out[9]= True
```

Транзитивное ограничение неациклического графа не обязательно минимально при применении TransitiveReduction. Ориентированный 5-цикл включает все ребра полного пятивершинного ориентированного графа. Поэтому мы имеем необязательное ребро.

```
In[10]:= ShowGraph[TransitiveReduction[CompleteGraph[5, Type→ Directed]]]
```



| | |
|---|---|
| TransitiveClosure[g] | возвращает транзитивное замыкание графа g |
| TransitiveReduction[g] | возвращает транзитивную редукцию графа g |
| PartialOrderQ[g] | возвращает True, если бинарное отношение, определенное ребрами графа g, является частичным порядком |
| PartialOrderQ[r] | возвращает True, если бинарное отношение, заданное матрицей r, определяет частичный порядок |

### *7.4.2. Диаграммы Хассе*

Частичный порядок изображается диаграммой Хассе, которая ясно отображает иерархию, заданную частичным порядком, и, чтобы избежать загромождения рисунка, содержит наименьшее возможное число ребер.

Более точно, диаграмма Хассе частично упорядоченного множества P обладает свойствами:

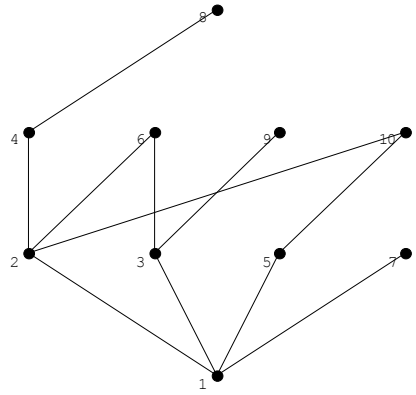
1. если i<j в P, то i появляется ниже j
2. диаграмма не содержит ребер, влекущих транзитивность. Так как все ребра в таких диаграммах идут вверх, направления ребер в диаграммах обычно опускаются.

Чтобы представить иерархию частично упорядоченного множества, может быть использовано ранжированное вложение графа – диаграммы. В основании иерархии находятся вершины, которые имеют входящую степень 0. Вершины, которые имеют выходящую степень 0, представляют максимум частичного порядка и ранжированы наверху. Применение транзитивной редукции минимизирует число ребер графа.

HasseDiagram[g] возвращает диаграмму Хассе отношения, определенного ориентированным ациклическим графом g.

Рассмотрим, например, диаграмму Хассе отношения делимости на множестве натуральных чисел:

```
In[2]:= ShowLabeledGraph[HasseDiagram[h]]
```

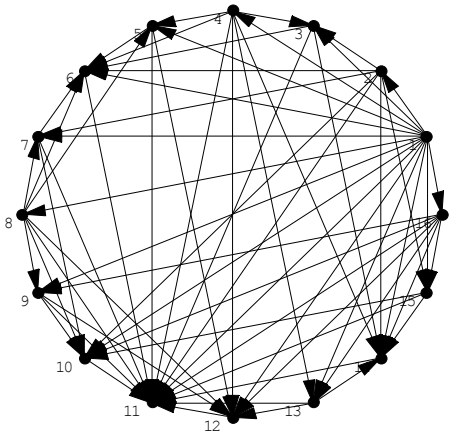



Булева алгебра - частичный порядок на подмножествах, определенный отношением включения. Рассмотрим, например, отношение включения на множестве всех подмножеств четырехэлементного множества.

```
In[3]:= g = MakeGraph[Subsets[4], ((Intersection[#2, #1] === #1) && (#1 =!= #2)) &]
Out[3]= -Graph:<65, 16, Directed>-
```

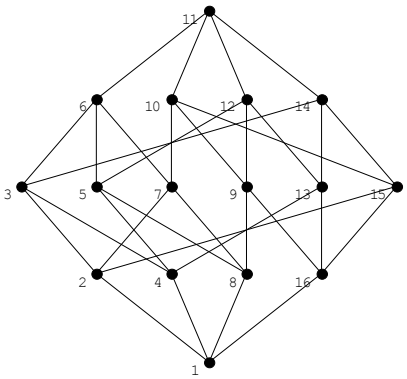
Combinatorica по умолчанию возвращает циркулярное вложение графа, из которого трудно понять строение булевой алгебры.

```
In[4]:= ShowLabeledGraph[g]
```




А теперь рассмотрим диаграмму Хассе, которая ясно показывает решетчатую структуру графа:

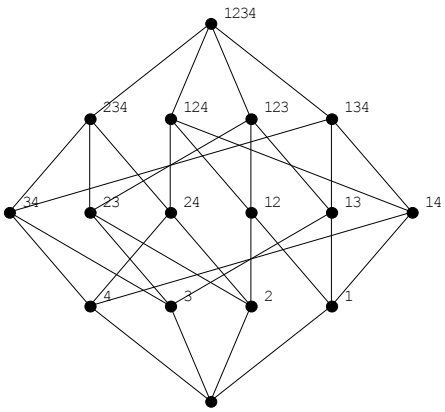
```
In[5]:= ShowLabeledGraph[HasseDiagram[g]]
```




Сформируем список меток: каждую вершину пометим соответствующим этой вершине подмножеством.

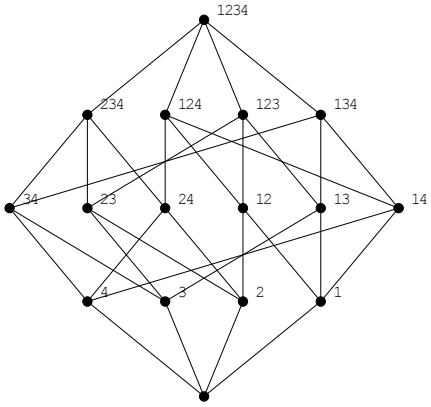
```
In[6]:= l = Map[StringJoin[Map[ToString, #]] &, Subsets[4]]
Out[6]= {, 4, 34, 3, 23, 234, 24, 2, 12, 124, 1234, 123, 13, 134, 14, 1}
```

```
In[7]:= ShowGraph[HasseDiagram[g], VertexLabel → l]
```



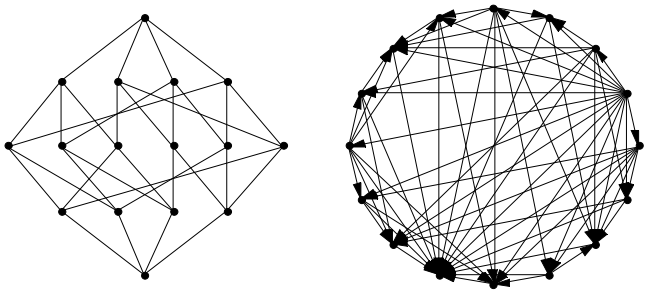

Combinatorica содержит функцию BooleanAlgebra[n], которая возвращает диаграмму Хассе для булевой алгебры на n элементах. Функция допускает две опции: Type и VertexLabel, которые принимают по умолчанию значения Undirected и False соответственно.

```
In[8]:= ShowGraph[BooleanAlgebra[4, VertexLabel → True]]
```




BooleanAlgebra принимает опцию Type, которая если устанавливается на Directed, то производит лежащий в основе ориентированный ациклический граф. Тогда как диаграмма Хассе более удобна при визуализации, лежащий в основе DAG более удобен для вычисления

```
In[9]:= ShowGraphArray[{BooleanAlgebra[4], BooleanAlgebra[4, Type → Directed]}]
```

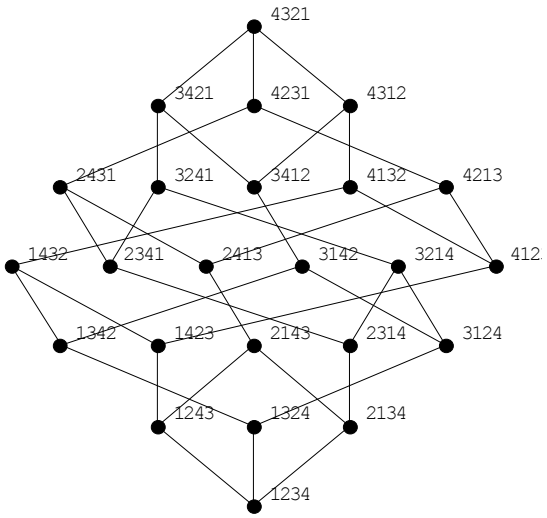


Оказывается, BooleanAlgebra[4] – другое вложение гиперкуба!

```
In[10]:= IsomorphicQ[Hypercube[5], BooleanAlgebra[5]]
Out[10]= True
```

Рассмотрим множество n-перестановок. Введем отношение порядка: для n-перестановок $\pi$ и $\pi_1$, $\pi < \pi_1$, если $\pi_1$ может быть получена из $\pi$ последовательностью смежных транспозиций так, что каждая транспозиция увеличивает число инверсий. Combinatorica обеспечена функцией InversionPoset, которая берет положительное целое n и возвращает диаграмму Хассе частично упорядоченного множества на n перестановках. Функция принимает две опции Type и VertexLabel, которые принимают по умолчанию значения Undirected и False соответственно.
Например, рассмотрим InversionPoset[4]:

```
In[11]:= ShowGraph[InversionPoset[4, VertexLabel → True]]
```

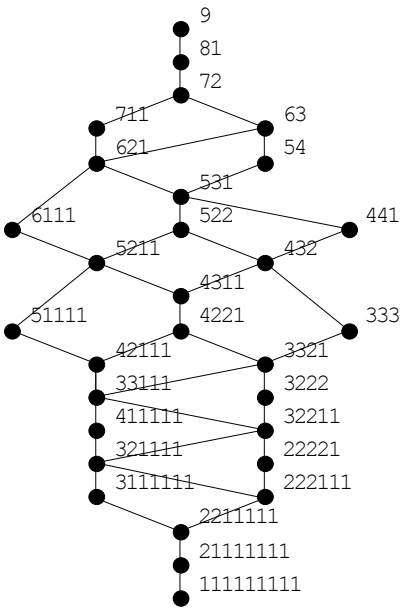



Рассмотрим целые разбиения $\lambda=(\lambda_1, \lambda_2,\ldots, \lambda_k)$ и $\mu=(\mu_1, \mu_2,\ldots,\mu_j)$ некоторого положительного целого числа n. Будем говорить, что $\lambda$ **доминирует** над $\mu$, если $\lambda_1+\lambda_2+\ldots+\lambda_t \geq \mu_1+\mu_2+\ldots+\mu_t$ для всех $t\geq 1$.
Функция DominatigIntegerPartitionQ[a,b] тестирует, будет ли целое разбиение a доминировать над целым разбиением b.
DominationLattice[n] возвращает диаграмму Хассе частично упорядоченного множества на множестве целых разбиений натурального числа n, в котором p<q, если q доминирует над p. Функция принимает две опции: Type и VertexLabel, которые по умолчанию принимают значения Undirected и False соответственно.
Так как 5+1 меньше, чем 4+3, первое разбиение не может доминировать над вторым, и т.к. 4 меньше, чем 5, два разбиения несравнимы.

```
In[12]:= {DominatingIntegerPartitionQ[{5, 1, 1, 1}, {4, 3, 1}],
          DominatingIntegerPartitionQ[{4, 3, 1}, {5, 1, 1, 1}]}
Out[12]= {False, False}
```

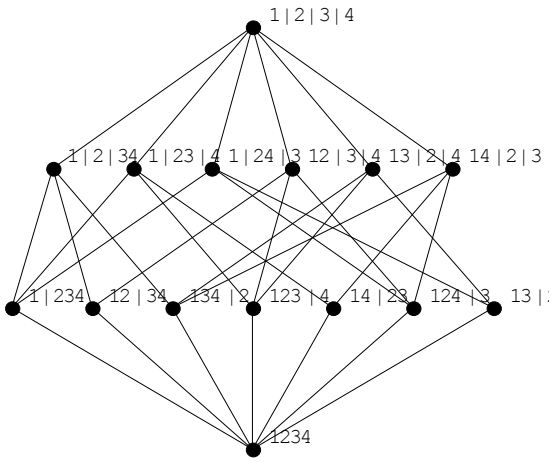
Рассмотрим DominationLattice[9]

```
In[13]:= ShowGraph[DominationLattice[9, VertexLabel → True]]
```




Последний чум, который мы здесь рассмотрим - решетка разбиений, которая является частичным порядком на разбиениях множества. Для двух разбиений p и q множества S  p ≤ q, если p "грубее", чем q. Другими словами, каждый блок в q содержится в некотором блоке в p.
Функция PartitionLattice[n] берет натуральное число n и вычисляет решетку разбиения на множестве разбиений множества {1,2,.....n}. Функция принимает две опции: Type и VertexLabel, которые по умолчанию принимают значения Undirected и False соответственно.
Рассмотрим PartitionLattice[5]. Элемент  в основании - грубейшее разбиение, тогда как верхний элемент - самое тонкое возможное разбиение. Второй уровень (снизу) соответствует множеству разбиений с двумя блоками, а уровень выше - с тремя блоками. Число элементов в каждом уровне, поэтому является числом Стирлинга второго рода.

```
In[14]:= ShowGraph[PartitionLattice[4, VertexLabel → True], PlotRange → 0.1]
```




| | |
|---|---|
| HasseDiagram[g] | возвращает диаграмму Хассе отношения, определенного ориентированным ациклическим графом g |
| BooleanAlgebra[n] | возвращает диаграмму Хассе для булевой алгебры на n элементах. Функция допускает две опции: Type и VertexLabel, которые принимают по умолчанию значения Undirected и False соответственно |

| | |
|---|---|
| InversionPoset[n] | берет положительное целое n и возвращает диаграмму Хассе частично упорядоченного множества на n перестановках. Функция принимает две опции Type и VertexLabel, которые принимают по умолчанию значения Undirected и False соответственно. |
| DominatigIntegerPartitionQ[a,b] | тестирует, будет ли целое разбиение a доминировать над целам разбиением b |
| DominationLattice[n] | возвращает диаграмму Хассе частично упорядоченного множества на множестве целых разбиений натурального числа n, в котором p<q, если q доминирует над p. Функция принимает две опции: Type и VertexLabel, которые по умолчанию принимают значения Undirected и False соответственно p |
| PartitionLattice[n] | берет натуральное n и вычисляет решетку разбиения на множестве разбиений множества {1,2,.....n}. Функция принимает две опции: Type и VertexLabel, которые по умолчанию принимают значения Undirected и False соответственно |

### *7.4.3. Теорема Дилворта*

Цепь в частично упорядоченном множестве – это множество элементов $\nu_1,\nu_2,\ldots,\nu_k$ , таких, что $\nu_i$ находится в отношении с $\nu_{i+1}$, (i<k). Антицепь – множество элементов, ни одна пара из которых не находится в данном отношении. Теорема Дилворта утверждает, что для любого частичного порядка максимальный размер антицепи равен минимальному числу цепей, на которые разбиваются элементы множества.
Функция MinimumChainPartition[g] разбивает частичный порядок g на минимальное число цепей.
Функция MaximumAntichain[g] возвращает максимальное число вершин, не находящихся в отношении частичного порядка g.
Рассмотрим минимальное разбиение на цепи решетки доминирования на целых разбиениях 8.

```
In[2]:= MinimumChainPartition[d = DominationLattice[8, Type → Directed]]

Out[2]= {{1, 2, 3, 4, 6, 7, 11, 12, 16, 17, 20, 21, 22}, {5, 8, 9, 10, 13, 14, 15, 18, 19}}
```

По теореме Дилворта, длина максимальной антицепи равна размеру минимального разбиения на цепи.

```
In[3]:= MaximumAntichain[d]

Out[3]= {4, 5}
```

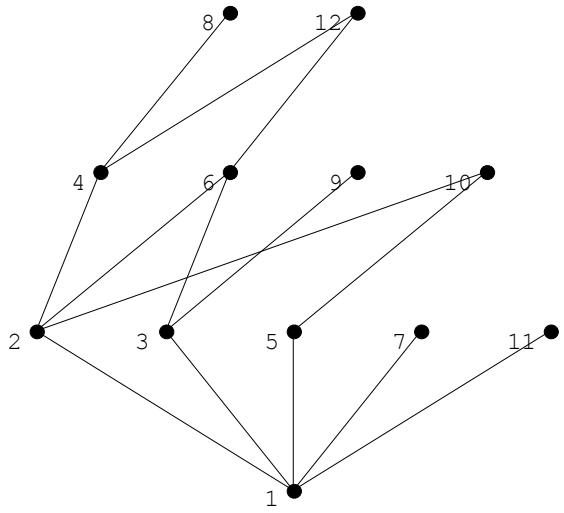
| | |
|---|---|
| MinimumChainPartition[g] | разбивает частичный порядок g на минимальное число цепей |
| MaximumAntichain[g] | возвращает максимальное число вершин, не находящихся в отношении частичного порядка g |

### *7.4.4. Топологическая сортировка*

Топологическая сортировка – это фундаментальная операция на ориентированных ациклических графах. С ее помощью вершины располагаются в таком порядке, что все направленные ребра идут «слева направо» (от вершины с меньшим номером к вершине с большим). Очевидно, что такой расстановки не существует, если граф содержит ориентированные циклы, так как нельзя идти по прямой вправо и вернуться туда, откуда вы начали.

Функция TopologicalSort[g] возвращает топологическую сортировку вершин ориентированного ациклического графа g.
Рассмотрим диаграмму Хассе отношения делимости на множестве натуральных чисел. Каждое отношение i<j в диаграмме представляется "верхним путем" от i к j. Так как существует верхний путь от 1 к любому другому, каждая топологическая сортировка этого частично упорядоченного множества начинает с 1. Любое перечисление целых в неубывающем порядке их рангов является топологической сортировкой.

```
In[2]:= ShowGraph[HasseDiagram[g = MakeGraph[Range[12], (Mod[#2, #1] == 0) &]],
          VertexNumber → True, TextStyle → {FontSize → 11}];
```




Такой "ранжированный " топологический порядок на элементах и является топологической сортировкой.

```
In[3]:= TopologicalSort[g]

Out[3]= {1, 2, 3, 5, 7, 11, 4, 6, 9, 10, 8, 12}
```

Любая перестановка на вершинах пустого графа определяет топологический порядок.

```
In[4]:= TopologicalSort[EmptyGraph[30]]

Out[4]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
         16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30}
```

С другой стороны, полный ориентированный ациклический граф определяет общий порядок:

```
In[5]:= TopologicalSort[MakeGraph[Range[30], (#1 > #2) &]]

Out[5]= {30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19,
         18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1}
```

| | |
|---|---|
| TopologicalSort[g] | возвращает топологическую сортировку вершин ориентированного ациклического графа g |

## ■7.5. Изоморфизмы графов

### *7.5.1. Нахождение изоморфизмов графов*

Как уже упоминалось, два графа изоморфны, если существует переименование вершин графа такое, что эти графы идентичны. Два помеченных графа идентичны, если они имеют идентичный список ребер.
Каждая перестановка полного графа представляет полный граф:

```
In[2]:= Isomorphism[CompleteGraph[20], CompleteGraph[20]]
Out[2]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}
```

Эти графы явно изоморфны, но не идентичны:

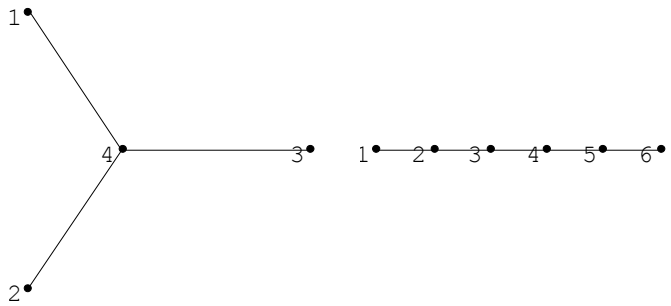
```
In[3]:= {IdenticalQ[g = CompleteGraph[5, 6], h = CompleteGraph[6, 5]], IsomorphicQ[g, h]}
Out[3]= {False, True}
```

Чтобы найти изоморфизм графов, нужно проверить n! Перестановок. Использование инвариантов графа позволяет сократить число перестановок.
Инварианты графов – меры графов, которые инвариантны при изоморфизмах. Например, никакие две вершины различной степени не могут отображаться друг на друга. Использование степеней вершин в соединении с некоторыми другими легко вычисляемыми инвариантами может помочь при тестировании графов на изоморфность. Функция Equivalences, показанная ниже, обеспечивает базовый механизм для сокращения числа кандидатов перестановок, которые нам нужно рассмотреть.
Функция Equivalences[g, h] перечисляет классы эквивалентности вершин графов g, h, определенные их степенями вершин. Equivalences[g] перечисляет классы эквивалентности вершин графа g, определенные их степенями вершин. Могут быть также использованы функции Equivalences[g, h, f1, f2, ...] , Equivalences[g, f1, f2, ...], где f1, f2, ...- другие инварианты вершин. Каждая функция fi[g, v] возвращает соответствующий инвариант вершины v в графе g. Функции f1, f2, ... вычисляются по порядку, причем вычисление прекращается, когда: либо все функции вычислены, либо найден пустой класс эквивалентности.

Первой, второй, третьей вершинам звезды, имеющим степень 1, эквивалентны концевые точки пути, которые имеют степень 1, а четвертой точке звезды степени 3 не соответствует ни одна точка пути. Следовательно, эти графы не изоморфны.

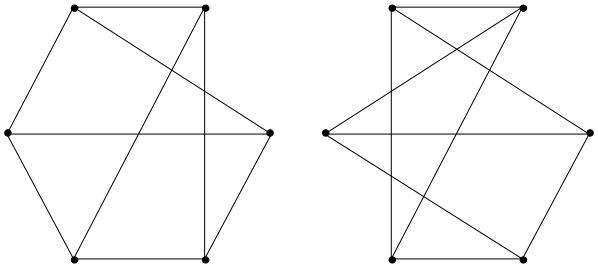
```
In[4]:= Equivalences[Star[4], Path[6]]
Out[4]= {{1, 6}, {1, 6}, {1, 6}, {}}
```

```
In[5]:= ShowGraphArray[{g = Star[4], h = Path[6]}, VertexNumber → True]
```




А в этом случае степени вершин бесполезны как инвариант:

```
In[6]:= g = RegularGraph[3, 6]; h = RegularGraph[3, 6]; Equivalences[g, h]
Out[6]= {{1, 2, 3, 4, 5, 6}, {1, 2, 3, 4, 5, 6}, {1, 2, 3, 4, 5, 6},
         {1, 2, 3, 4, 5, 6}, {1, 2, 3, 4, 5, 6}, {1, 2, 3, 4, 5, 6}}

In[7]:= ShowGraphArray[{RegularGraph[3, 6], RegularGraph[3, 6]}]
```



Combinatorica содержит несколько функций, которые вычисляют некоторые другие меры локальных структур графа. Эти меры инвариантны относительно изоморфизма графа и, следовательно, полезны при сокращении пространства поиска изоморфизма и в определении неизоморфных графов. Такими инвариантами являются, например, Neighborhood, DegreesOf2 Neighborhood, NumberOfKPaths, NumberOf2Paths.

Neighborhood[g,v,k] возвращает подмножество вершин графа g, которые находятся на расстоянии, по крайней мере, не большем k. Тот же результат можно получить, используя функцию Neighborhood[al,v,k], которая как ввод берет список смежности al графа g.

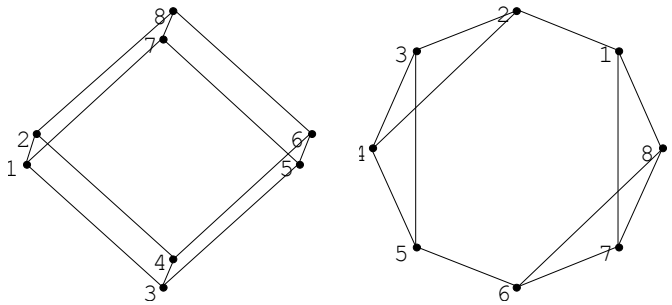
DegreesOf2Neighborhood[g,v] возвращает упорядоченный список степеней вершин в графе g, которые находятся на расстоянии двух ребер от вершины v.

NumberOfKPaths[g,v,k] возвращает упорядоченный список, который состоит из числа путей длины k от вершины v до остальных вершин графа. Тот же результат можно получить, используя функцию NumberOfKPaths[al,v,k], которая как ввод берет список смежности al графа g. Если взять k=2, получим NumberOf2Paths.

Distances[g, v] возвращает расстояния от вершины v до всех вершин графа в неубывающем порядке, при этом граф трактуется как невзвешенный граф.

Рассмотрим два 3-регулярных 8-вершинных графа. Изоморфны ли эти графы? Если нет, можем мы доказать их неизоморфность, используя некоторые простые инварианты?

```
In[8]:= h = DeleteEdges[CirculantGraph[8, {1, 2}], {{2, 8}, {4, 6}, {1, 3}, {5, 7}}];
        g = Hypercube[3]; ShowGraphArray[{g, h}, VertexNumber → True]
```




Для каждой вершины графа g вычислим список степеней вершин этого графа, которые находятся на расстоянии двух ребер. В трехмерном гиперкубе 7 вершин находятся в пределах двух ребер от каждой вершины:

```
In[10]:= Table[DegreesOf2Neighborhood[g, i], {i, V[g]}] // ColumnForm
```

```
Out[10]= {3, 3, 3, 3, 3, 3, 3}
         {3, 3, 3, 3, 3, 3, 3}
         {3, 3, 3, 3, 3, 3, 3}
         {3, 3, 3, 3, 3, 3, 3}
         {3, 3, 3, 3, 3, 3, 3}
         {3, 3, 3, 3, 3, 3, 3}
         {3, 3, 3, 3, 3, 3, 3}
         {3, 3, 3, 3, 3, 3, 3}
```

Граф h не так симметричен, как гиперкуб. Вычислим список степеней вершин графа h, которые находятся на расстоянии двух ребер от каждой вершины.

```
In[11]:= Table[DegreesOf2Neighborhood[h, i], {i, V[g]}] // ColumnForm
Out[11]= {3, 3, 3, 3, 3, 3, 3}
         {3, 3, 3, 3, 3, 3, 3}
         {3, 3, 3, 3, 3, 3}
         {3, 3, 3, 3, 3, 3}
         {3, 3, 3, 3, 3, 3, 3}
         {3, 3, 3, 3, 3, 3, 3}
         {3, 3, 3, 3, 3, 3}
         {3, 3, 3, 3, 3, 3}
```

Эта таблица показывает, что для третьей, четвертой, седьмой и восьмой вершин только шесть вершин находятся от них на расстоянии двух ребер, следовательно, эти графы неизоморфны, так как ни одна из этих четырех вершин не может быть отображена ни на одну из вершин гиперкуба g.

```
In[12]:= Equivalences[g, h, DegreesOf2Neighborhood]
Out[12]= {{1, 2, 5, 6}, {1, 2, 5, 6}, {1, 2, 5, 6},
          {1, 2, 5, 6}, {1, 2, 5, 6}, {1, 2, 5, 6}, {1, 2, 5, 6}, {1, 2, 5, 6}}
```

Рассмотрим число путей длины 2 от первой вершины до всех вершин графов g и h:

```
In[13]:= {NumberOf2Paths[g, 1], NumberOf2Paths[h, 1]}
Out[13]= {{2, 2, 2, 3}, {1, 1, 1, 1, 2, 3}}
```

Первый список для графа g показывает, что существует три 2 - пути от первой вершины к самой себе через каждую из смежных с ней вершин – 2, 3, 7; существует два 2-пути до пятой, четвертой и восьмой вершин. Второй список перечисляет число 2-путей из первой вершины графа h: три 2-пути из первой вершины к самой себе, два 2-пути из первой в шестую вершину и по одному из первой в третью, четвертую, седьмую и восьмую вершины. Различий в этих двух списках достаточно для того, чтобы утверждать, что графы g и h неизоморфны. Действительно,

```
In[14]:= Equivalences[g, h, NumberOf2Paths]
Out[14]= {{}, {}, {}, {}, {}, {}, {}, {}}
```

Еще один инвариант – это множество кратчайших расстояний между одной вершиной и всеми другими. Для большинства графов этого достаточно, чтобы установить неизоморфность графов, хотя существуют неизоморфные графы, которые реализуют то же множество расстояний.
Например, для вышеприведенных графов использование Distances как инварианта не так эффективно, как использование NumberOf2Paths:

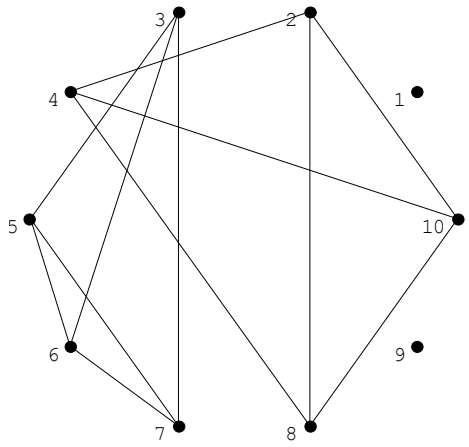
```
In[15]:= Equivalences[g, h, Distances]
Out[15]= {{1, 2, 5, 6}, {1, 2, 5, 6}, {1, 2, 5, 6},
          {1, 2, 5, 6}, {1, 2, 5, 6}, {1, 2, 5, 6}, {1, 2, 5, 6}, {1, 2, 5, 6}}
```

Функция IsomorphicQ[g,h] тестирует, являются ли графы g,h изморфными:

```
In[16]:= IsomorphicQ[g, h]
Out[16]= False
```

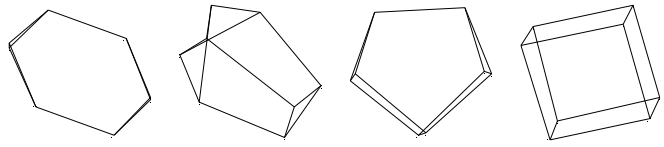
Рассмотрим десять полуслучайных 3 – регулярных графов и введем на этом множестве отношение изоморфизма. Построим новый граф, вершинами которых являются построенные десять графов, и две вершины соединяются ребром, если соответствующие этим вершинам графы изоморфны. Изображаемый граф есть множество несвязных клик, а каждая клика представляет множество неизоморфных исходных графов.

```
In[17]:= gt = Table[RegularGraph[3, 8], {10}];
         ShowLabeledGraph[h = MakeSimple[MakeGraph[gt, IsomorphicQ, Type → Undirected]],
          TextStyle → {FontSize → 12}]
```




Независимое множество в этом графе представляет множество взаимно неизоморфных графов. Изобразим эти неизоморфные графы

```
In[19]:= ShowGraphArray[Map[SpringEmbedding[#, 30] &, gt[[MaximumIndependentSet[h]]]]]
```

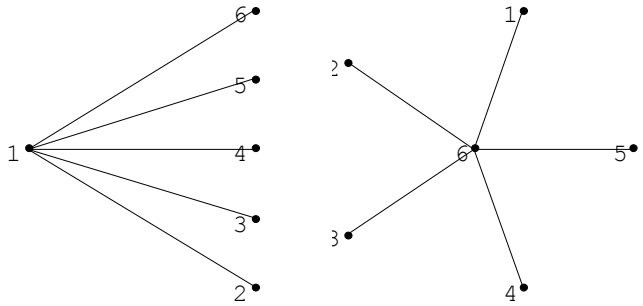


IsomorphismQ[g, h, p] тестирует, является ли перестановка p изоморфизмом графов g и h.
Функция Isomorphism[g, h] возвращает изоморфизм между графами g и h, если таковой существует.
Isomorphism[g, h, All] возвращает все изоморфизмы между графами g и h. Isomorphism[g] возвращает группу автоморфизмов графа g. Эти функции допускают опцию Invariants -> {f1, f2, ...}, где f1, f2, ...- функции, которые используются для вычисления инвариантов вершин, и они применяются в порядке, в котором они специфицированы. По умолчанию Invariants -> {DegreesOf2Neighborhood, NumberOf2Paths, Distances}.
Использование опции Invariants значительно сокращает время решения:

```
In[20]:= g = RegularGraph[3, 10]; h = RegularGraph[3, 10];
         {Timing[IsomorphicQ[g, h];], Timing[IsomorphicQ[g, h, Invariants → {}];]}
Out[20]= {{0.032 Second, Null}, {3.125 Second, Null}}
```

Полный двудольный граф $K_{n-1}$ изоморфен звезде на n вершинах

```
In[21]:= Isomorphism[CompleteKPartiteGraph[1, 5], Star[6]]
Out[21]= {6, 1, 2, 3, 4, 5}

In[22]:= ShowGraphArray[{CompleteKPartiteGraph[1, 5], Star[6]}, VertexNumber → True]
```




Тег All позволяет рассмотреть все изоморфизмы.

```
In[23]:= Isomorphism[Cycle[10], LineGraph[Cycle[10]], All]
Out[23]= {{1, 2, 10, 9, 8, 7, 6, 5, 4, 3}, {1, 3, 4, 5, 6, 7, 8, 9, 10, 2},
          {2, 1, 3, 4, 5, 6, 7, 8, 9, 10}, {2, 10, 9, 8, 7, 6, 5, 4, 3, 1}, {3, 1, 2, 10, 9, 8, 7, 6, 5, 4},
          {3, 4, 5, 6, 7, 8, 9, 10, 2, 1}, {4, 3, 1, 2, 10, 9, 8, 7, 6, 5}, {4, 5, 6, 7, 8, 9, 10, 2, 1, 3},
          {5, 4, 3, 1, 2, 10, 9, 8, 7, 6}, {5, 6, 7, 8, 9, 10, 2, 1, 3, 4}, {6, 5, 4, 3, 1, 2, 10, 9, 8, 7},
          {6, 7, 8, 9, 10, 2, 1, 3, 4, 5}, {7, 6, 5, 4, 3, 1, 2, 10, 9, 8}, {7, 8, 9, 10, 2, 1, 3, 4, 5, 6},
          {8, 7, 6, 5, 4, 3, 1, 2, 10, 9}, {8, 9, 10, 2, 1, 3, 4, 5, 6, 7}, {9, 8, 7, 6, 5, 4, 3, 1, 2, 10},
          {9, 10, 2, 1, 3, 4, 5, 6, 7, 8}, {10, 2, 1, 3, 4, 5, 6, 7, 8, 9}, {10, 9, 8, 7, 6, 5, 4, 3, 1, 2}}
```

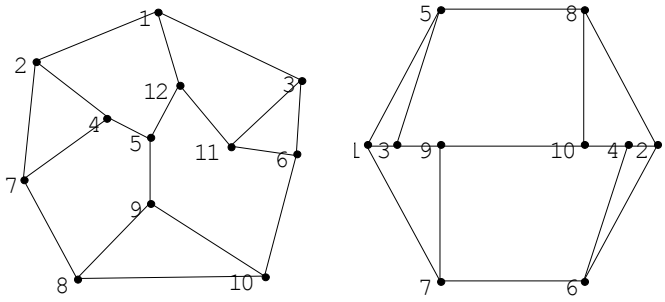
| | |
|---|---|
| Equivalences[g, h] | перечисляет классы эквивалентности вершин графов g, h, определенные их степенями вершин |
| Equivalences[g] | перечисляет классы эквивалентности вершин графа g, определенные их степенями вершин |
| Equivalences[g, h, f1, f2, ...] | перечисляет классы эквивалентности вершин графа g, определенные f1, f2, ...- другими инвариантами вершин. |
| Neighborhood[g,v,k] | возвращает подмножество вершин графа g, которые находятся на расстоянии, по крайней мере, не большем k от вершины v |
| Neighborhood[al,v,k] | производит то же самое, только как ввод берет список смежности al графа g |
| DegreesOf2Neighborhood[g,v] | возвращает упорядоченный список, который состоит из числа путей от вершины v до остальных вершин графа g |
| NumberOfKPaths[g,v,k] | возвращает упорядоченный список, который состоит из числа путей длины k от вершины v до остальных вершин графа |
| NumberOfKPaths[al,v,k] | производит то же самое, только как ввод берет список смежности al графа g |
| NumberOf2Paths [g, v] | возвращает упорядоченный список, который состоит из числа путей длины 2 от вершины v до остальных вершин графа |
| IsomorphismQ[g, h, p] | тестирует, является ли перестановка p изоморфизмом графов g и h |
| Isomorphism[g, h] | возвращает изоморфизм между графами g и h, если таковой существует |
| Isomorphism[g, h, All] | возвращает все изоморфизмы между графами g и h |
| Isomorphism[g] | возвращает группу автоморфизмов графа g. Эти функции допускают опцию Invariants -> {f1, f2, ...}, где f1, f2, ...- функции, которые используются для вычисления инвариантов вершин, и они применяются в порядке, в котором они специфицированы. По умолчанию Invariants -> {DegreesOf2Neighborhood, |

| | NumberOf2Paths, Distances} |
|---|---|

### *5.2. Нахождение автоморфизмов графов*

Каждый граф G, ориентированный или неориентированный, имеет группу автоморфизмов Γ=Γ(G), состоящую из изоморфизмов G на себя. Это значит, что группа автоморфизмов Γ графа G состоит из всех таких взаимно однозначных отображений $\alpha$ множества вершин Vна себя, что если {a,b} есть (ориентированное) ребро в G, то ребро {$\alpha$(a), $\alpha$(b)} также является (ориентированным) ребром, и наоборот. Таким образом, Γ можно рассматривать как группу перестановок на множестве V. Если граф G представляется матрицей смежности M(G), то есть автоморфизмами будут те перестановочные матрицы $M_\alpha$, перестановочные с M(G). Перестановочной называется матрица, в каждой строке и столбце которой находятся ровно по одному единичному элементу, а все остальные элементы равны 0. Умножение любой матрицы A на перестановочную матрицу P слева означает некоторую перестановку строк матрицы A. Умножение A на P справа равносильно некоторой перестановке столбцов A.
Оба графа на нижеприведенном рисунке являются однородными степени 3. Первый из них, FruchtGraph – наименьший однородный граф степени 3, группа автоморфизмов которого является единичной, а группа автоморфизмов второго графа – порядка 2, как показал Фрухт, определяется подстановкой - {{1,2},{3,4},{5,6},{7,8},{9,10}}.

```
In[2]:= v = {{-1, 0}, {1, 0}, {-0.8, 0}, {0.8, 0}, {-0.5, 0.8}, {0.5, -0.8}, {-0.5, -0.8},
          {0.5, 0.8}, {-0.5, 0}, {0.5, 0}};
        Fr = FromAdjacencyLists[{{5, 3, 7}, {8, 4, 6}, {5, 9}, {10, 6}, {8}, {7}, {9}, {10}, {10}, {4}}, v];

In[3]:= ShowGraphArray[{FruchtGraph, Fr}, VertexNumber → True]
```




```
In[4]:= {Automorphisms[FruchtGraph], Automorphisms[Fr]}
Out[4]= {{{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}},
         {{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, {2, 1, 4, 3, 6, 5, 8, 7, 10, 9}}}
```

Найдем автоморфизм графа Fr:

```
In[5]:= ToCycles[{2, 1, 4, 3, 6, 5, 8, 7, 10, 9}]
Out[5]= {{2, 1}, {4, 3}, {6, 5}, {8, 7}, {10, 9}}
```

Группой автоморфизмов простого цикла является диэдральная группа:

```
In[6]:= {Length[DihedralGroup[10]], Length[Automorphisms[Cycle[10]]],
         Complement[Automorphisms[Cycle[10]], DihedralGroup[10]]}
Out[6]= {20, 20, {}}
```

Группа автоморфизмов дополнения графа та же самая, что и у первоначального.

```
In[7]:= g = RandomGraph[10, 0.3]; SameQ[Automorphisms[g], Automorphisms[GraphComplement[g]]]
Out[7]= True
```

Вычислим группу перестановок, действующую на ребрах графа пятивершинного колеса.

```
In[8]:= g = Wheel[5]; G = Automorphisms[g]; KSubsetGroup[G, Edges[g]]
Out[8]= {{1, 2, 3, 4, 5, 6, 7, 8}, {1, 4, 3, 2, 8, 7, 6, 5}, {2, 1, 4, 3, 5, 8, 7, 6}, {2, 3, 4, 1, 6, 7, 8, 5},
        {3, 2, 1, 4, 6, 5, 8, 7}, {3, 4, 1, 2, 7, 8, 5, 6}, {4, 1, 2, 3, 8, 5, 6, 7}, {4, 3, 2, 1, 7, 6, 5, 8}}
```

Группа тетраэдра есть симметрическая группа $S_4$ , так как он является полным графом на четырех вершинах:

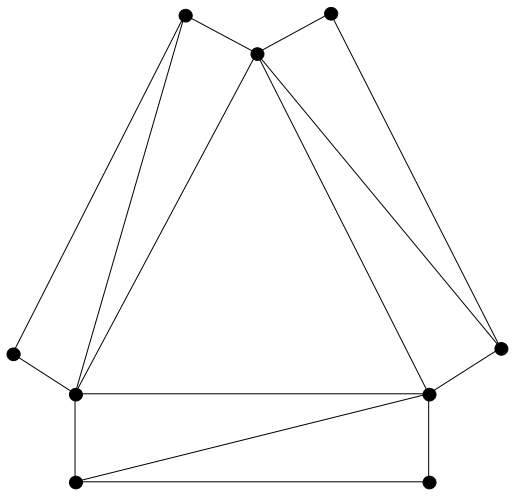
```
In[9]:= Automorphisms[TetrahedralGraph]
Out[9]= {{1, 2, 3, 4}, {1, 2, 4, 3}, {1, 3, 2, 4}, {1, 3, 4, 2}, {1, 4, 2, 3}, {1, 4, 3, 2},
         {2, 1, 3, 4}, {2, 1, 4, 3}, {2, 3, 1, 4}, {2, 3, 4, 1}, {2, 4, 1, 3}, {2, 4, 3, 1},
         {3, 1, 2, 4}, {3, 1, 4, 2}, {3, 2, 1, 4}, {3, 2, 4, 1}, {3, 4, 1, 2}, {3, 4, 2, 1},
         {4, 1, 2, 3}, {4, 1, 3, 2}, {4, 2, 1, 3}, {4, 2, 3, 1}, {4, 3, 1, 2}, {4, 3, 2, 1}}
```

Группы куба или октаэдра изоморфны прямому произведению группы $S_4$ и отражения порядка 2. Группы додекаэдра и икосаэдра изоморфны прямому произведению $A_5$ и отражения. Каждая из них состоит из 48 элементов.

```
In[10]:= Length[Automorphisms[OctahedralGraph]]
Out[10]= 48
```

Так называемый граф Петерсена – однородный граф степени 3 и порядка 10. Он был впервые введен Петерсеном в качестве примера графа степени 3 , который не является суммой трех графов первой степени. Как установил Фрухт, его группа изоморфна $S_5$.
Функция SmallestCyclicGroupGraph возвращает наименьший нетривиальный граф, чья группа автоморфизмов является циклической.

```
In[11]:= ShowGraph[SmallestCyclicGroupGraph]
```



Рассмотрим его группу автоморфизмов:

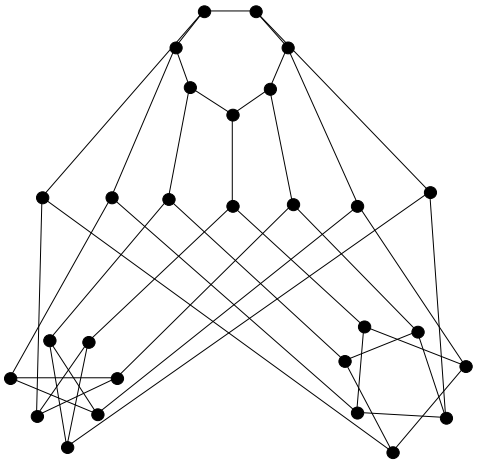
```
In[12]:= g = Automorphisms[SmallestCyclicGroupGraph]
Out[12]= {{1, 2, 3, 4, 5, 6, 7, 8, 9}, {2, 3, 1, 8, 9, 4, 5, 6, 7}, {3, 1, 2, 6, 7, 8, 9, 4, 5}}
```

Покажем, что g-циклическая группа:

```
In[13]:= SameQ[MultiplicationTable[CyclicGroup[3], Permute], MultiplicationTable[g, Permute]]
Out[13]= True
```

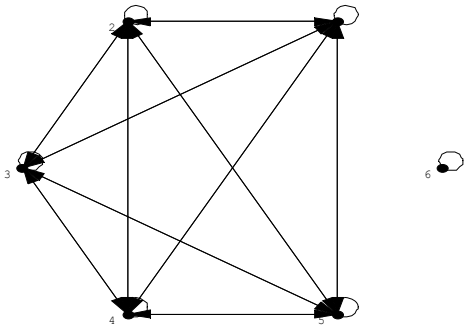
В связи с группами автоморфизмов графов следует также упомянуть о некоторых интересных исследованиях регулярных графов степени 3, которые провел Татт. Он рассмотрел задачу о нахождении таких графов, в которых любая ориентированная простая цепь длины s может быть переведена автоморфизмом в любую другую такую простую цепь. Это возможно при следующих условиях: $s \le 5$ и $s \le 0.5m+1$.
Функция CoxeterGraph возвращает негамильтонов граф с высокой степенью симметрии такой, что существует автоморфизм, переводящий любую цепь длины 3 в любую другую такую цепь.

```
In[14]:= ShowGraph[CoxeterGraph]
```



Рассмотрим граф 6-вершинного колеса. Определим бинарное отношение на вершинах графа: вершина i находится в отношении с вершиной j, если существует некоторый автоморфизм $\pi$, такой, что $\pi(i)=j$. Это отношение является отношением эквивалентности с двумя классами эквивалентности, один размера 5, а другой единичный. Единичный класс эквивалентности соответствует центральной вершине колеса, который отображается на себя при каждом автоморфизме.

```
In[15]:= ag = Automorphisms[g = Wheel[6]]; r = MemberQ[Table[ag[[i, #1]], {i, Length[ag]}], #2] &;
         ShowGraph[MakeGraph[Range[6], r, Type → Directed], VertexNumber → True]
```

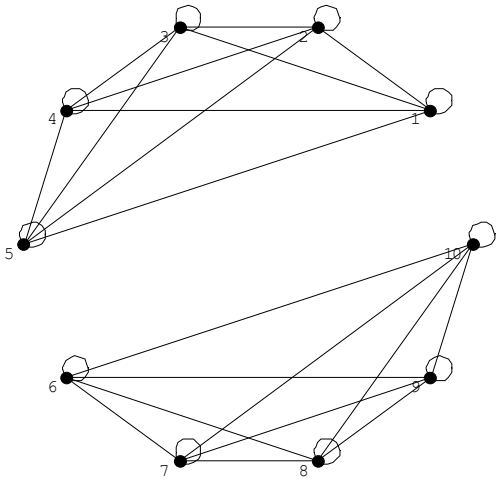



Вычислим то же самое отношение эквивалентности, только на этот раз на ребрах: ребро i находится в отношении с ребром j, если существует перестановка, переводящая ребро i в ребро j. Это еще одно отношение эквивалентности. Каждый класс эквивалентности содержит 5 ребер. Ребра, инцидентные центральной вершине, формируют одну группу, оставшиеся ребра - другую.
Граф называется **вершинно-транзитивным**, если для любой пары вершин (i,j) существует автоморфизм $\pi$ такой, что $\pi(i)=j$. Другими словами, граф называется вершинно-транзитивным, если его группа автоморфизмов транзитивна. Аналогично, граф называется **реберно-транзитивным**, если для любой пары ребер $(e_1,e_2)$ существует перестановка $\pi$ ребер, порожденная автоморфиз-

мом, такая, что $\pi(e_1)=e_2$. Если граф является и реберно-транзитивным и вершинно-транзитивным, то он называется симметрическим. Регулярный граф, который является реберно-транзитивным, но не вершинно-транзитивным, называется полусимметрическим.

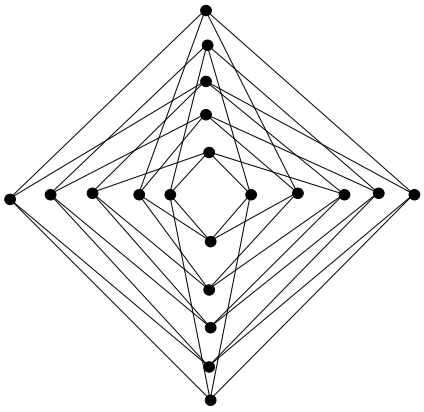
Граф колеса не является ни реберно-транзитивным, ни вершинно-транзитивным.

```
In[17]:= ag = KSubsetGroup[ag, Edges[g]];
         ShowGraph[MakeGraph[Range[10], r, Type → Undirected], VertexNumber → True]
```



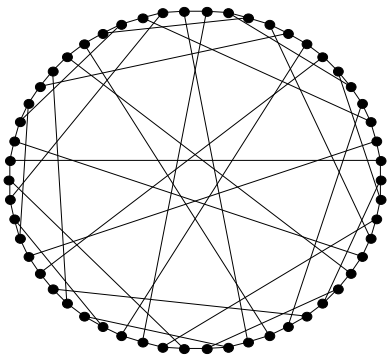

Граф Фолкмана, приведенный ниже – наименьший нетривиальный граф, который является реберно-транзитивным, но не вершинно-транзитивным графом. Это полусимметрический 4-регулярный граф.

```
In[19]:= ShowGraph[u = FolkmanGraph]
```



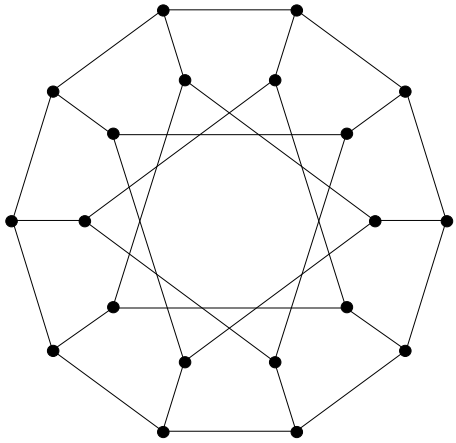
Еще один пример полусимметрического графа – граф Грэя: 3-регулярный граф на 54 вершинах. Это наименьший из известных примеров таких графов.

```
In[20]:= ShowGraph[GrayGraph]
```



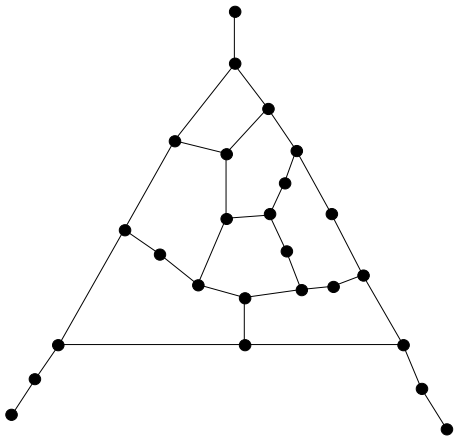
Граф g называется n-унитранзитивным, если он связный, 3-регулярный, n-транзитивный и если для любых n- маршрутов u и v существует ровно один автоморфизм $\alpha$ графа g такой, что $\alpha$ (u)=v
Функция UnitransitiveGraph возвращает 20-вершинный 3-унитранзитивный граф, открытый Коксетером, который не изоморфен 4- или 5- клеточному.

```
In[21]:= ShowGraph[UnitransitiveGraph]
```



Продемонстрируем еще один граф на 25 вершинах с единичной группой автоморфизмов – граф Уолтера:

```
In[22]:= ShowGraph[WaltherGraph]
```



```
In[23]:= Length[Automorphisms[WaltherGraph]]

Out[23]= 1
```

| | |
|---|---|
| Automorphisms[g] | возвращает группу автоморфизмов графа g |
| FruchtGraph | возвращает граф Фрухта |
| SmallestCyclicGroupGraph | возвращает наименьший нетривиальный граф, чья группа автоморфизмов является циклической |
| CoxeterGraph | возвращает граф Коксетера |
| FolkmanGraph | возвращает граф Фолкмана |
| GrayGraph | возвращает граф Грэя |
| UnitransitiveGraph | возвращает 20-вершинный 3-унитранзитивный граф, открытый Коксетером, который не изоморфен 4- или 5- клеточному |

### *7.5.3. Изоморфизм деревьев*

Более быстрый алгоритм тестирования деревьев на изоморфность использует сертификаты. Вычисляется сертификат n- вершинного дерева T, который является бинарной строкой длины 2n. Вначале присвоим каждой вершине дерева метку 01. Для каждой не висячей вершины x составим множество Y - это множество меток висячих вершин, смежных x и метка x, из которой удаляется начало 0 и конец 1. Следующий шаг повторяется до тех пор, пока останутся две или меньше вершин. Заменим метку x на конкатенацию меток в Y,отсортированных в возрастающем лексикографическом порядке, с 0 впереди и 1 сзади. Потом удаляем все висячие вершины, смежные x. После повторения этого достаточное число раз, мы имеем одну или две висячие вершины в T.Если существует только одна вершина x, то метка x сертификат. Если существуют две висячие вершины x и y, то эти две метки упорядочиваются в возрастающем лексикографическом порядке и их конкатенация есть сертификат. При этом деревья имеют одинаковый сертификат тогда и только тогда, когда они изоморфны.

Функция TreeToCertificate[t]-возвращает сертификат дерева t.
Функция TreeIsomorphismQ[t1, t2] возвращает True, если деревья t1 и t2 изоморфны, и False в противном случае.
Выделим сертификаты случайных деревьев.

```
In[2]:= Table[TreeToCertificate[Path[i]], {i, 10}]
Out[2]= {01, 0101, 001011, 00110011, 0001100111, 000111000111,
       00001110001111, 0000111100001111, 000001111000011111, 0000011111000000011111}
```

Использование TreeIsomorphismQ занимает меньшее время, чем использование IsomorphismQ для тестирования изоморфизмов деревьев.

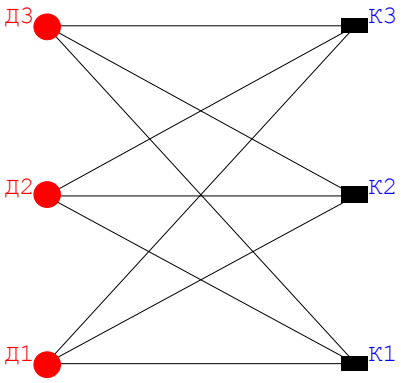
```
In[3]:= g = Star[100]; h = CompleteGraph[1, 99];
      {Timing[TreeIsomorphismQ[g, h];], Timing[IsomorphicQ[g, h];]}
Out[3]= {{0.031 Second, Null}, {17.375 Second, Null}}
```

| | |
|---|---|
| TreeToCertificate[t] | возвращает сертификат дерева t |
| TreeIsomorphismQ[t1, t2] | возвращает True, если деревья t1 и t2 изоморфны, и False в противном случае |

## ■7.6. Планарные графы

Рассмотрим известную задачу о трех домах и трех колодцах: В трех домах живут трое враждующих соседей, каждый из которых ходит за водой в один из трех колодцев. Можно ли протоптать тропинки от каждого дома к каждому колодцу таким образом, чтобы пути соседей не пересекались?
Абстрактный граф называется **планарным,** если он может быть реализован в виде плоского геометрического графа.
Встроенная функция PlanarQ[g] тестирует, является ли данный граф g планарным. С ее помощью получим ответ задачи о трех домах и трех колодцах. Рассмотрим $K_{3,3}$, три вершины в одной доли соответствуют трем домам, три вершины другой доли - колодцам.

```
In[2]:= q = SetGraphOptions[CompleteKPartiteGraph[3, 3],
          {{1, 2, 3, VertexLabel → {Д1, Д2, Д3}, VertexLabelPosition → UpperLeft,
            VertexLabelColor → RGBColor[1, 0, 0], VertexStyle → Disk[Large], VertexColor → Red},
           {4, 5, 6, VertexLabel → {К1, К2, К3}, VertexLabelColor → RGBColor[0, 0, 1],
            VertexStyle → Box[0.03]}}]
Out[2]= -Graph:<9, 6, Undirected>-

In[3]:= ShowGraph[q]
```



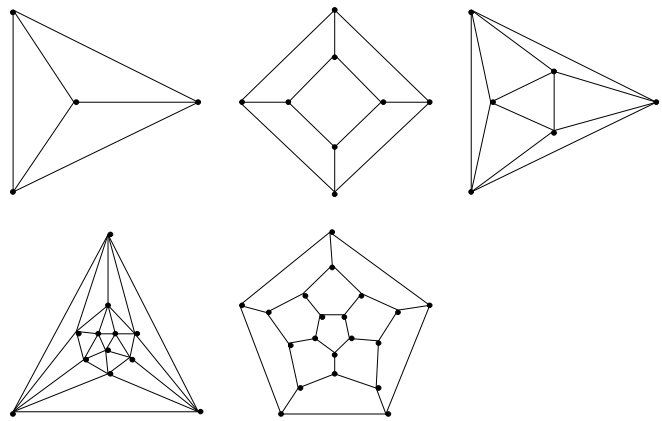

```
In[4]:= PlanarQ[q]
Out[4]= False
```

Все деревья планарны. Текущая геометрическая реализация не влияет на PlanarQ

```
In[5]:= Apply[And, Map[PlanarQ, Table[RandomTree[i], {i, 1, 50}]]]
Out[5]= True
```

Графы платоновых тел также планарны, это следует из того факта, что геометрическая реализация графа на плоскости эквивалентна геометрической рализации графа на сфере, что следует из свойств стереографической проекции.

```
In[6]:= ShowGraphArray[
          g = {{TetrahedralGraph, CubicalGraph, OctahedralGraph}, {IcosahedralGraph, DodecahedralGraph}}]
```



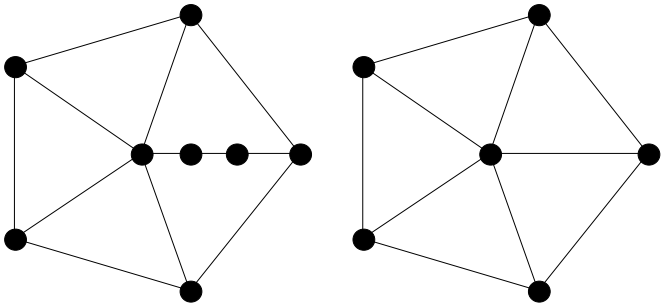
Доказательство того факта, что не существует других платоновых тел, следует из формулы Эйлера для любого планарного графа с V вершинами, E ребрами,G гранями, V-E+F=2. Из формулы Эйлера следует, что каждый планарный граф имеет, по крайней мере, 3n-6 ребер для n≥5

```
In[7]:= Apply[Or, Map[PlanarQ, Table[ExactRandomGraph[n, 3 n - 6], {n, 5, 15}]]]
```

```
Out[7]= True
```

Очевидно, что свойство планарности не нарушается, если некоторое ребро графа разбить на два ребра введением новой вершины второй степени или заменить два ребра, инцидентных одной вершине второй степени, одним ребром, удалив эту вершину. Два графа называются **гомеоморфными** (или изоморфными с точностью до вершин второй степени), если при помощи этих операций они становятся изоморфными.
Очевидно, что изоморфизм графов – частный случай гомеоморфизма. Пример гомеоморфных, но не изоморфных графов приведен ниже:

```
In[8]:= ShowGraphArray[{g = AddVertices[Wheel[6], {{0.3, 0}, {0.6, 0}}], Wheel[6]},
          VertexStyle → Disk[Large]]
```



Покажем, что эти графы неизоморфны

```
In[9]:= IsomorphicQ[g, Wheel[6]]
Out[9]= False
```

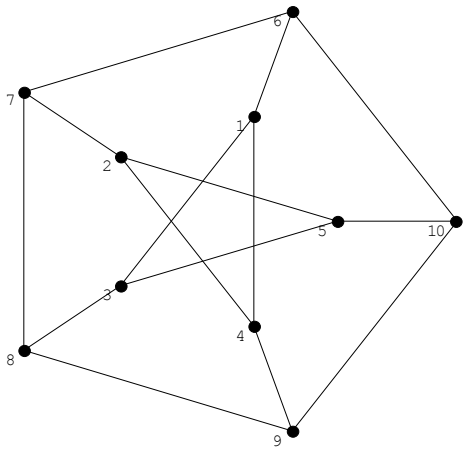
Необходимый и достаточный признак планарности графов следует из знаменитой теоремы Понтрягина – Куратовского, которая утверждает, что граф планарен тогда и только тогда, когда он не содержит подграфа, гомеоморфного $K_5$ или $K_{3,3}$.

Покажем, что $K_5$ и $K_{3,3}$ не планарны

```
In[10]:= Map[PlanarQ, {CompleteGraph[5], CompleteKPartiteGraph[3, 3]}]
Out[10]= {False, False}
```

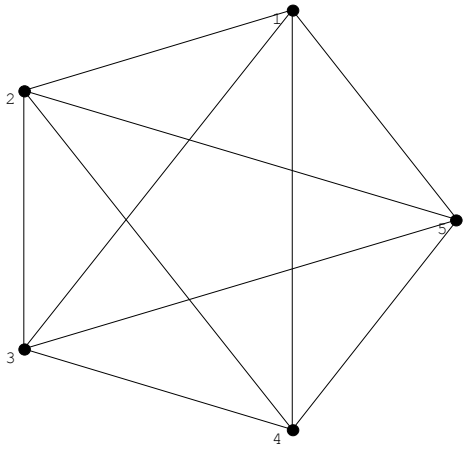
Рассмотрим граф Петерсена, который не содержит подграфа, изоморфного $K_5$ или $K_{3,3}$, однако, как известно, он не планарный.

```
In[11]:= ShowLabeledGraph[PetersenGraph]
```



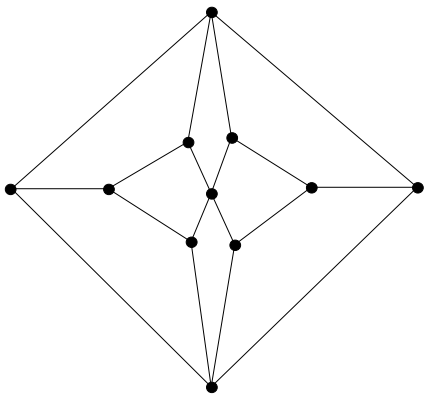
Граф Петерсена сводится к $K_5$ стягиванием вершин {i,7-i}, где i∈{1,2,3,4,5}

```
In[12]:= t = PetersenGraph; Do[t = Contract[t, {1, 7 - i}], {i, 1, 5}];
         ShowLabeledGraph[MakeSimple[t]]
```




Граф многогранника – это 3-связный простой планарный граф. Граф Гершеля – это наименьший негамильтонов граф многогранника. Это единственный такой граф на 11 вершинах и с 18 ребрами .

```
In[13]:= ShowGraph[HerschelGraph]
```



```
In[14]:= {HamiltonianQ[HerschelGraph], PlanarQ[HerschelGraph]}
Out[14]= {False, True}
```

| | |
|---|---|
| PlanarQ[g] | тестирует, является ли данный граф g планарным |
| HerschelGraph | возвращает граф Гершеля |

# Глава 8. Примеры решения задач

Приведем отрывок из книги Д. Пойа “Как решать задачу” [37]

Что нужно сделать, чтобы решить задачу?

1. Понять задачу.

• Что является неизвестным? Каковы исходные данные? Каковы условия задачи?

• Можно ли удовлетворить условиям задачи? Достаточно ли заданных условий для отыскания неизвестной? Являются ли условия задачи недостаточными, избыточными или противоречивыми?

• Сделайте рисунок. Введите соответствующие обозначения.

• Выделите отдельные части условия задачи. Можете ли вы их сформулировать?

2. Составить план решения.

• Определите взаимосвязь между исходными данными и неизвестной.

• Возможно, вам следует рассмотреть какие-то дополнительные задачи, если не удается непосредственно найти эту взаимосвязь.

• В конце концов, вы должны получить план решения.

• Вы уже встречали эту задачу? Или, может быть, вы встречали похожую задачу?

• Может быть, вы знаете задачу, связанную с этой? Известна ли вам какая-нибудь теорема, которая может быть полезной?

• Рассмотрите искомое и попытайтесь вспомнить задачу, которая вам уже известна и в которой отыскивалось то же самое или нечто подобное.

• Перед вами задача, которая связана с рассматриваемой и которую вы уже решали. Можете ли вы использовать результаты, полученные при её решении? Можете ли вы воспользоваться использованным методом решения? Может быть, следует ввести какой-то дополнительный элемент, чтобы воспользоваться этим методом?

• Могли бы вы сформулировать задачу по-другому? Смогли бы вы дать ещё одну формулировку задачи? Обратитесь мысленно к определениям.

• Если вы не можете решить задачу, которая вам предложена, попытайтесь решить задачу, связанную с ней. Не можете ли вы припомнить задачу, которая была бы связана с вашей, но была бы разрешима? Более общая или более частная задача или аналогичная вашей? Смогли бы вы решить часть задачи? Оставьте только часть условий, необходимых для другой части задачи. В какой мере может быть теперь определена неизвестная, как можно ею варьировать? Можете ли вы извлечь из данных что-то полезное? Не приходят ли вам на ум другие данные, которые могли бы помочь определить неизвестную? Можете ли вы изменить неизвестную, или исходные данные, или то и другое, если это необходимо, так, чтобы новая неизвестная и новые данные лучше соответствовали друг другу?

• Вы использовали все исходные данные? Вы использовали условия задачи в полном объеме? Вы учли все существенные стороны задачи?

3. Выполнить план.

•Выполняя принятый план решения, проверяйте каждый его этап, каждый элемент последовательно один за другим. Очевидно ли вам, что этот элемент плана корректен? Можете ли доказать его корректность?

4. Проверить полученное решение.

• Можете ли вы проверить полученный результат? Можете ли вы проверить проведенные рассуждения?

• Можете ли вы получить результат, отличающийся от уже полученного? Можете ли это понять с первого взгляда?

• Можете ли вы воспользоваться полученным результатом или методом решения для какой-нибудь другой задачи?

## Терминология и краткий обзор задач

Представление задачи означает формулировку задачи на языке некоторой теории. Например, функция f: A → {0, 1}, определенная на множестве A и принимающая значение 0 или 1, есть представление подмножества B = (x ∈ A : f(x) = 1) множества A. Граф с множеством вершин V представляет некоторое бинарное отношение на множестве V. Как следует из приведенного ниже списка NP-полных задач, задачи целочисленного программирования, комбинаторного анализа, булевой алгебры, доказательства теорем в формальных теориях первого порядка имеют эффективное представление задачами теории графов и наоборот. И, по-видимому, графы играют здесь такую же роль, что и графики функций в математическом анализе. Представление позволяет применять для решения задачи в исходной постановке, операции, теоремы, методы и алгоритмы той теории, в терминах которой она адекватно представлена. Необходимо отметить, что задача может иметь различные представления и в рамках одной теории (задачи “Новая группа”, “Помощь маляру”). В частности, во многих задачах математических олимпиад выбор определенного представления, или унификация задачи с некоторой теоремой, помогает быстро найти короткое доказательство требуемого утверждения.

Важным шагом этапа представления задачи является абстракция, игнорирование несущественных для задачи деталей. Например, в задаче “Сталкер” вершиной графа заменяется целая карта, а в задаче “Помощь маляру” ребра исходного графа рассматриваются как ребра нового графа. Или, замена страны планарного графа вершиной двойственного графа. Менее тривиальный пример абстракции показан при решении задачи “Два туриста”, где задача, представленная двумя графами, сводится к применению стандартного алгоритма на одном графе, а именно, на декартовом произведении исходных графов.

Унификация означает выбор такого представления, для которого можно применить адекватный задаче некоторый алгоритм или теорему. Пример, где унификация диктует представление исходной задачи, показан при решении задачи “Сталкер”, в которой граф в свою очередь представлен вершиной нового графа. Более хитрый пример унификации приведен в задаче “Курорт Even”, где задача формулируется так, чтобы ее решение достигалось с помощью алгоритма Дейкстры, хотя применение этого алгоритма к прямой задаче не дает нужного решения.

Для успешного решения задачи, естественно, необходима ориентация в той теории, в терминах которой представлена задача. В задаче “Маршруты модниц”, после выбора представления необходимо найти цепочку теорем, ведущую к реализованному алгоритму: теорема Менгера -> теорема Форда-Фалкерсона -> алгоритм Форда-Фалкерсона. Еще один пример - задача “Диверсант”, в которой требуется перечислить все циклы графа.

Задачу можно унифицировать к разным алгоритмам и теоремам, но желательно выбрать оптимальный по времени выполнения (задача “Новая группа”). Пример неудачной унификации приведен в задаче “восьми ферзей”.

Представляя задачу при помощи графов, для наглядности полезно улучшить вложение графа на плоскости, что позволяет визуально выявить структуру данных исходной задачи (задачи “Острова”, “Перемена мест”, “Диверсант”, ”Разноцветные фишки”). Иногда такое вложение помогает сразу решить задачу (задача “Перемена мест”) еще на этапе представления. В задаче “Ханойские башни и ковер Серпинского” улучшение вложения позволило установить занимательную связь задачи с геометрией, а именно с ковром Серпинского. Для визуального исследования больших графов полезно применить опцию PlotRange со значением Zoom, позволяющую рассмотреть окрестности интересующих нас вершин. В пакете функций “Combinanorica” реализованы несколько функций вложения графа на плоскость: CircularEmbedding, RankedEmbedding, RootedEmbedding, SpringEmbedding. Особый интерес вызывает вложение SpringEmbedding, при котором вершины графа располагаются на плоскости так, чтобы привести в равновесие систему связанных по ребрам законом Гука отрицательных электрических зарядов, расположенных в вершинах.

Очень часто легко решить задачу дискретной математики на компьютере можно при помощи рекурсии, если задачу данной размерности можно сформулировать посредством задач меньшей размерности (хрестоматийная задача “Ханойские башни”). В методе математической индукции

рекурсия эквивалентна индукционному шагу. В компьютерной графике рекурсия применяется при построении фракталов, красивых геометрических объектов, части которых имеют такую же структуру, как у всего объекта. Простым примером фрактала является дерево в теории графов, канторово множество и ковер Серпинского в геометрии. Интересный пример фрактала возникает при решении той же задачи "Ханойские башни и ковер Серпинского", что и неудивительно, ввиду ее рекурсивной структуры. Как правило, большинство объектов комбинаторики имеют рекурсивную структуру. Учитывая лавинообразный рост вычислений при вычислении рекурсивной функции, даже тогда, когда задача сформулирована в рекурсивной форме, необходимо вначале поискать другие методы ее вычисления: либо найти замкнутую формулу вычисления рекурсивной функции (задачи "Ханойские башни", "Задача Иосифа Флавия", "Ряды"), либо итерационным алгоритмом линейной сложности, как в задаче "Числа Фибоначчи". Рекурсивное вычисление можно применять в задачах, где требуется полный перебор различных вариантов, например, в комбинаторике, при генерации всех подмножеств, разбиений, перестановок, и т. д. В некоторых задачах рекурсия эффективна и по сложности вычислений, если удается быстро объединить решения подзадач, хрестоматийным примерами которой являются задача поиска элемента в отсортированном множестве, алгоритм "быстрой" сортировки множества. Такой же пример приведен в задаче "задача Иосифа Флавия". Рекурсия применяется при решении задач: "Рюкзак", "Оптимальная упаковка рюкзака", "Редактирование".

Необходимо знать, когда можно и когда нельзя свести задачу к задаче, для которой известен эффективный алгоритм. Например, в самой постановке задачи "Дефицит" скрывается явный подвох для решателя. Ее формулировка наталкивает на перебор подмножеств множества вершин графа, хотя теорема Кенига устанавливает связь дефицита двудольного графа с мощностью его максимального паросочетания, для поиска которого можно применить известный алгоритм Карпа – Хопкрофта полиномиальной сложности. Такой же пример представлен в решении задачи "Прямоугольники".

Использование перебора в этих случаях является грубой ошибкой и напоминает участие марафонца в гонках Формулы-1.

Задача относится к классу P, если ее можно решить за полиномиальное от размерности входных данных время. Причем степень полинома не зависит от размерности входных данных. Многие задачи теории графов относятся к классу P. Перечислим главные из них.

1. Найти эйлеров цикл на графе из m ребер. Сложность порядка O(m).
2. Построить остовное дерево минимальной стоимости. СложностьO(m log m), m – число ребер.
3. Найти кратчайший путь на графе, состоящем из n вершин и m ребер. Сложность O(mn).
4. Найти связные компоненты графа. Сложность $O(n^2)$.
5. Построить транзитивное замыкание. Сложность $O(n^2)$.
6. Построить максимальное паросочетание. Сложность $O(n^{5/2})$.
7. Найти максимальный поток. Сложность $O(n^3)$.
8. Проверка планарности графа. Сложность O(n).

Решение многих других задач сокрыто в доказательствах теорем, поскольку большинство теорем дискретной математики имеет конструктивное доказательство.

Для справки ниже приведен краткий список часто возникающих задач, сводимых друг к другу за полиномиальное от размерности входных данных задачи время вычислений, успешно решаемых полным перебором подмножеств множества входных данных, но для которых неизвестен эффективный алгоритм их решения. Это класс так называемых NP-полных задач.

1. Имеет ли данный неориентированный граф гамильтонов цикл.
2. Является ли данный неориентированный граф k-раскрашиваемым.
3. Найти в данном неориентированном графе максимальное независимое множество.
4. Найти в данном неориентированном графе минимальное вершинное покрытие.
5. Найти в данном неориентированном графе максимальную клику.
6. Выделить из семейства подмножеств минимальное подсемейство, являющегося покрытием множества.
7. Выделить из семейства множеств подсемейство непересекающихся множеств, являющегося покрытием объединения всех множеств семейства.

8. Задача о рюкзаке: найти решение в (0,1)-числах уравнения
   $\Sigma a_i x_i = b$ , где a, b – заданные целые числа.
9. Задача о камнях: разделить кучу камней на две равные по весу части.
10. Разложить натуральное число на простые множители.
11. Выполнимость логического выражения.
12. Построение минимальной ДНФ.

Более полный список NP-полных задач можно найти в [7].
NP-полные задачи тем более интересны, что если хотя бы одна из них будет решена за полиномиальное время, то и все остальные задачи из класса NP будут решены за полиномиальное время. Задача о том, совпадают ли множества P и NP, является основной нерешенной задачей теории сложности.
Задачи, которые можно решить перебором подмножеств множества входных данных, относятся к классу NP-трудных задач, к которому относятся NP-полные задачи. NP-полные задачи, формулируемые как задачи оптимизации, перестают принадлежать классу NP-полных (задача нахождения гамильтонова цикла является NP- полной, задача коммивояжера – нет). Для решения NP-трудных задач большой размерности обычно применяют различные эвристические алгоритмы, дающие не обязательно точное решение (задача "Помощь маляру"), а также полиномиальные алгоритмы, дающие приближенное решение, отличающегося от оптимального в отношении не более чем некоторое число $\varepsilon$. Примеры $\varepsilon$-приближенных алгоритмов приведены для задачи коммивояжера ("Алгоритм Кристофидеса" и "Алгоритм Эйлера").
Заметим, что для некоторых NP-трудных задач, для определенного класса данных есть эффективные алгоритмы. Например, задачи 3, 4, 5 вышеприведенного списка для двудольных графов.
Существенно сокращает перебор для NP-трудных задач метод динамического программирования, когда вновь не пересчитываются ранее полученные по ходу решения результаты. Для применения этого метода к задачам поиска оптимального пути, он должен удовлетворять принципу Беллмана: каждая его часть (s, t) является оптимальным путем от s к t. Так устроен, например, классический алгоритм Дейкстры нахождения кратчайшего пути на графе, ребрам которого приписаны положительные веса. На каждом шаге алгоритма решается подзадача, результат которой – постоянная пометка – далее не пересчитывается.
Метод динамического программирования представлен решениями задач "Слалом", "Оптимальная упаковка рюкзака", "Редактирование".
Для решения открытой и замкнутой задачи коммивояжера продемонстрирован метод ветвей и границ.
Остановимся на задаче поиска кратчайшего s-t пути. Пусть ребрам не приписаны веса. Требуется найти путь с наименьшим количеством ребер. В этом случае для поиска вершины t применятся стратегия обхода вершин графа в ширину, по которой рассматриваются напарники вершины s, затем непросмотренные напарники напарников s и т. д., пока не достигнем вершины t. Сложность алгоритма поиска в ширину пропорциональна числу ребер, а алгоритм Дейкстра имеет сложность $O(n^2)$. Поэтому в задаче "Ханойские башни" и ей подобных вместо функции ShortestPath лучше применять функцию BreadthFirstTraversal.
Для поиска длин кратчайших путей между всеми парами вершин графа взвешенного графа лучше применять алгоритм Флойда. Он применим и для графа с ребрами отрицательного веса, но без циклов отрицательной длины. Эффективный метод решения задачи без последнего ограничения пока неизвестен. Более того, доказано, что такая задача является NP-полной.
Задачи "Грэй-код", "Экономный обход",… , "Нумерация композиций" демонстрируют нумерации комбинаторных множеств построением гамильтонова пути графа, представляющего определенное бинарное отношение на множестве.
Задачи "Раскраски", "Ожерелья", "Простая группа $A_5$" демонстрируют применение теории групп для решения комбинаторных задач.

### *8.1. Задача Иосифа Флавия*

Дана группа из n человек, сидящих за круглым столом. При этом каждый m-й человек должен быть казнен. Казни продолжаются до тех пор, пока не останется только один человек. Найти позицию L(n,m), в которой нужно находиться, чтобы быть последним. Список, дающий позиции казненных, может быть получен с помощью функции Josephus[n,m].

**Решение**. В книге [7] приведена рекурсивная функция J(n), возвращающая номер уцелевшего при
m = 2

$J(1) = 1;$
$J(2n) = 2J(n) - 1;$
$J(2n+1) = 2J(n) + 1,$

и замкнутая формула ее вычисления:

$J((b_k b_{k-1} \ldots b_1 b_0)_2) = (b_{k-1} \ldots d_1 b_0 b_m)_2,$

где $(b_k b_{k-1} \ldots b_1 b_0)_2$ – двоичное представление числа n.
В случае m>2 определить такую же простую рекурсивно заданную функцию не удается. Воспользуемся функцией Josephus, реализованной в пакете Combinftorica.
Например, рассмотрим четырех человек, занумерованных от 1 до 4, причем каждый второй должен быть убит.

```
In[2]:= Josephus[4, 2]

Out[2]= {4, 1, 3, 2}
```

Этот пример показывает, что первого человека казнят последним, второго – первым, третьего – третьим, четвертого – вторым. Не совсем так. Первого конечно не казнят.

Обратная перестановка возвращает последовательность казненных:

```
In[3]:= InversePermutation[%]

Out[3]= {2, 4, 3, 1}
```

Первоначальная задача Иосифа Флавия: даны 41 человек, стоящих по кругу, причем каждый третий должен быть казнен, то есть n=41, m=3;

```
In[4]:= Josephus[41, 3]

Out[4]= {14, 36, 1, 38, 15, 2, 24, 30, 3, 16, 34, 4, 25, 17,
          5, 40, 31, 6, 18, 26, 7, 37, 19, 8, 35, 27, 9, 20,
          32, 10, 41, 21, 11, 28, 39, 12, 22, 33, 13, 29, 23}
```

Последовательность казненных:

```
In[5]:= InversePermutation[%]

Out[5]= {3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 1, 5, 10, 14, 19, 23, 28, 32,
         37, 41, 7, 13, 20, 26, 34, 40, 8, 17, 29, 38, 11, 25, 2, 22, 4, 35, 16, 31}
```

Проиллюстрируем это решение. Изобразим на внешнем круге порядковые номера казненных.

```
In[6]:= l = Vertices[EmptyGraph[41]]; m = Range[41];
        s = 1.2 l; t = Josephus[41, 3];
        g1 = Graphics[Text["Последний", l[[31]] + {0, 0.2}]];
        g2 = Graphics[Text["Предпоследний", l[[16]] + {0.6, 0.02}]];
        g3 = Table[Graphics[Text[m[[i]], l[[i]]]], {i, 41}];
        g4 = Table[Graphics[Text[t[[i]], s[[i]]]], {i, 41}];
        g5 = Graphics[Circle[l[[31]], 0.1]];
        g6 = Graphics[Circle[l[[16]], 0.1]];

In[7]:= Show[g1, g2, g3, g4, g5, g6, AspectRatio → Automatic, TextStyle → {FontSize → 12}]
```


Последним казнят 31-го человека, а предпоследним – шестнадцатого.

### *8.2. Задача "Грэй-код"*

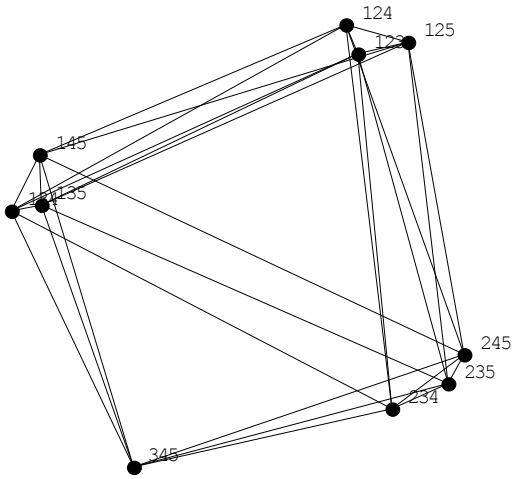
Перечислить все k-подмножества данного множества таким образом, чтобы от одного подмножества к другому можно перейти, по крайней мере, одной операцией вставки и одной операцией удаления (перечисление в порядке Грэй-кода).

**Решение.** Задача сводится к построению гамильтонова пути на графе. Сформируем бинарное отношение между парами подмножеств: два подмножества находятся в отношении, если одно может быть получено из другого вставкой и удалением одного элемента.

```
In[2]:= grC = ((Length[Complement[#1, #2]] == 1) && (Length[Complement[#2, #1]] == 1)) &;
```

Построим граф, вершины которого есть все 3-подмножества 5-элементного множества, а две вершины соединены ребром, если они находятся в отношении grC.

```
In[3]:= ShowGraph[
          g = SpringEmbedding[MakeGraph[KSubsets[Range[5], 3], grC, Type → Undirected,
              VertexLabel → True]], PlotRange → 0.1]
```



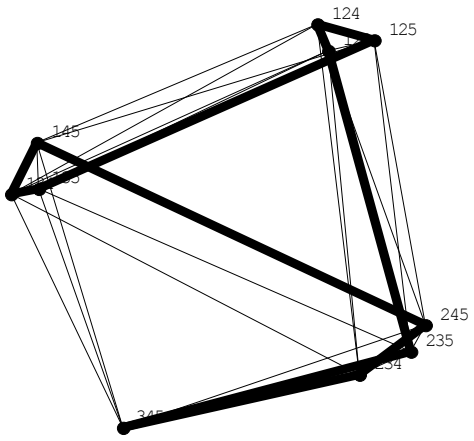

Построенный граф является гамильтоновым:

```
In[4]:= HamiltonianQ[g]
Out[4]= True
```

Выделим гамильтонов цикл:

```
In[5]:= HamiltonianCycle[g]
Out[5]= {1, 2, 3, 5, 4, 6, 9, 7, 10, 8, 1}
```

```
In[6]:= ShowGraph[Highlight[g, {Partition[HamiltonianCycle[g], 2, 1]}], PlotRange → 0.1]
```




Выведем метки гамильтонова цикла:

```
In[7]:= GetVertexLabels[g, HamiltonianCycle[g]]
Out[7]= {123, 124, 125, 135, 134, 145, 245, 234, 345, 235, 123}
```

Вычислим число гамильтоновых циклов:

```
In[8]:= Length[HamiltonianCycle[g, All]]
Out[8]= 6432
```

### *8.3. Задача "Экономный обход"*

Требуется перебрать все перестановки множества {1, 2, …, n} в таком порядке, чтобы каждая последующая перестановка отличалась от предыдущей ровно одной транспозицией.

**Решение.** Достаточно свести задачу к нахождению гамильтонова пути на графе.
На множестве всех перестановок определим бинарное отношение P: две перестановки находятся в отношении P, если они отличаются друг от друга ровно одной транспозицией. Как известно, бинарное отношение можно представить графом, вершины которого соответствуют элементам множества и две вершины смежны, если они находятся в данном отношении.
Заметим, что две перестановки находятся в отношении P, если они отличаются как упорядоченные множества ровно в двух позициях. Следующая функция определяет P:

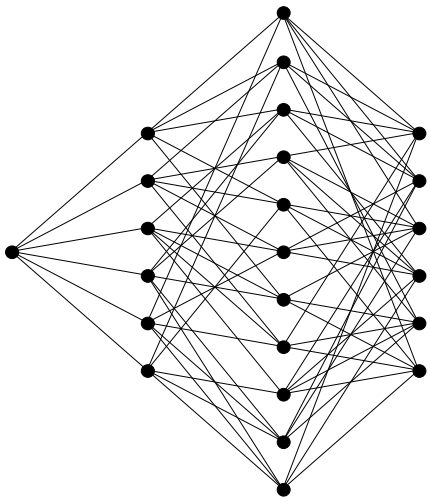
```
In[2]:= P = (p = #1; q = #2; k = 0; Do[If[p[[i]] == q[[i]], , k = k + 1], {i, 1, Length[#1]}];
          If[k == 2, True, False]) &;
```

Мы определили функцию P от двух аргументов, здесь p=#1 –первая перестановка, q=#2- вторая перестановка, далее в цикле Do[] подсчитываем количество k несовпадающих позиций в этих перестановках. Функция возвращает True, если k=2.
Рассмотрим пример для перестановок длины 4.

```
In[3]:= g = MakeGraph[Permutations[4], P, Type → Undirected]
Out[3]= -Graph:<72, 24, Undirected>-
```

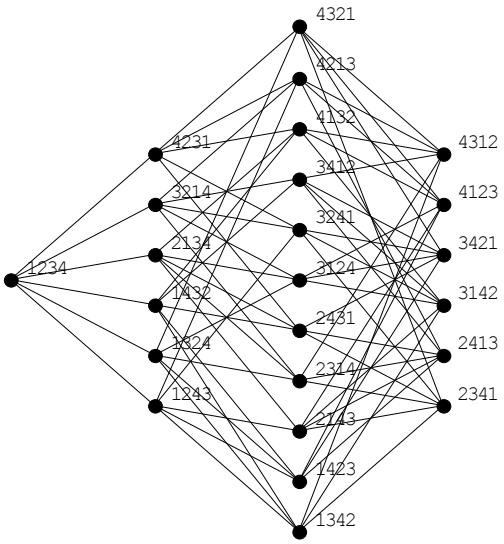
Для наглядности построим ранжированное вложение вершин этого графа относительно первой вершины:

```
In[4]:= ShowGraph[RankedEmbedding[g, {1}]]
```



Если мы хотим определить перестановку, соответствующую каждой вершине графа g , то достаточно в теле графа установить опцию VertexLabel->True, но метки сильно загромождают рисунок.

```
In[5]:= h = MakeGraph[Permutations[4], P, Type → Undirected, VertexLabel → True];
        ShowGraph[RankedEmbedding[h, {1}]]
```

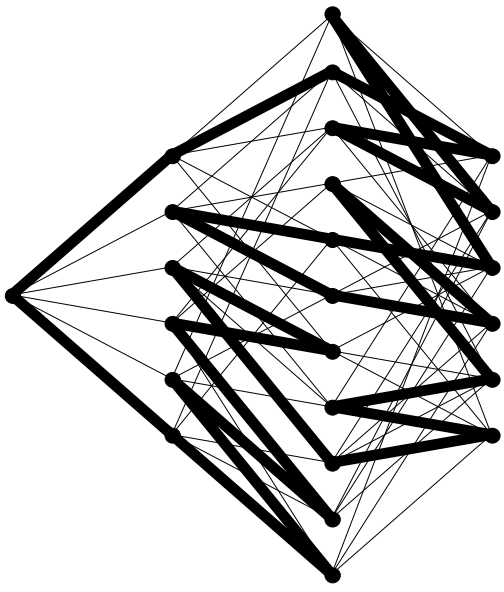



Проверим, является ли граф g гамильтоновым:

```
In[6]:= HamiltonianCycle[g]
Out[6]= {1, 2, 4, 3, 5, 6, 12, 7, 8, 10, 9, 11, 17, 14, 13, 15, 16, 18, 24, 19, 20, 23, 21, 22, 1}
```

Выделим этот гамильтонов цикл в графе:

```
In[7]:= ShowGraph[RankedEmbedding[Highlight[g, {Partition[HamiltonianCycle[g], 2, 1]}], {1}]]
```



Пройдя по вершинам этого гамильтонова цикла, получим все перестановки размера 4, причем любые две смежные перестановки отличаются ровно одной транспозицией.

```
In[8]:= GetVertexLabels[h, HamiltonianCycle[h]]
Out[8]= {1234, 1243, 1342, 1324, 1423, 1432, 2431, 2134, 2143, 2341, 2314, 2413,
         3412, 3142, 3124, 3214, 3241, 3421, 4321, 4123, 4132, 4312, 4213, 4231, 1234}
```

Вычислим расстояния в графе от первой вершины:

```
In[9]:= RankGraph[g, {1}]
Out[9]= {1, 2, 2, 3, 3, 2, 2, 3, 3, 4, 4, 3, 3, 4, 2, 3, 3, 4, 4, 3, 3, 2, 4, 3}
```

Рассмотрим распределение расстояний:

```
In[10]:= Distribution[RankGraph[g, {1}]]
Out[10]= {1, 6, 11, 6}
```

Отсюда следует, что шесть перестановок можно получить из тождественной перестановки одной транспозицией, одиннадцать перестановок – двумя транспозициями и шесть – тремя транспозициями.
То же самое можно вычислить с помощью числа Стирлинга первого рода:

```
In[11]:= Table[StirlingFirst[4, i], {i, 4, 1, -1}]
Out[11]= {1, 6, 11, 6}
```

В предыдущей задаче перестановки перечислялись в порядке минимального отличия друг от друга. Поставим обратную задачу:

### *8.4. Задача "Непохожие соседи"*

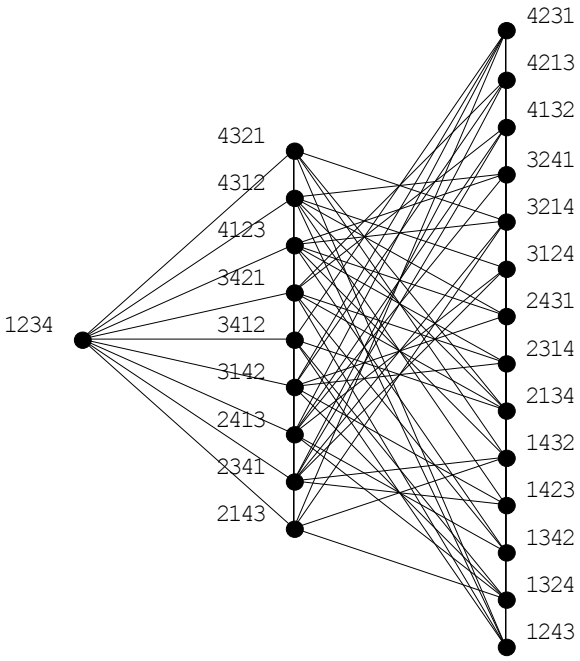
Требуется перебрать все перестановки множества {1,2,…,n} в таком порядке, чтобы каждая последующая перестановка отличалась от предыдущей во всех позициях.

**Решение.**Задача аналогична задаче "Экономный обход".
Известно, что перестановки $\pi_1$ и $\pi_2$ отличаются друг от друга во всех позициях, если произведение $(\pi_1)^{-1}\times\pi_2$ есть перестановка без неподвижных точек.
Построим булеву функцию и граф, вершинами которого являются перестановки размера 4, а две вершины смежны, если соответствующие им перестановки отличаются во всех позициях:

```
In[2]:= d = DerangementQ[Permute[InversePermutation[#1], #2]] &;
        dgraph = MakeGraph[Permutations[4], d, Type → Undirected, VertexLabel → True]
Out[2]= -Graph:<108, 24, Undirected>-
```

Чтобы метки перестановок сильно не загромождали рисунок, установим опции меток:

```
In[3]:= h = SetGraphOptions[dgraph, {1, 8, 10, 11, 14, 17, 18, 19, 23, 24, VertexLabelPosition → UpperLeft}]
Out[3]= -Graph:<108, 24, Undirected>-

In[4]:= ShowGraph[RankedEmbedding[h, {1}]]
```

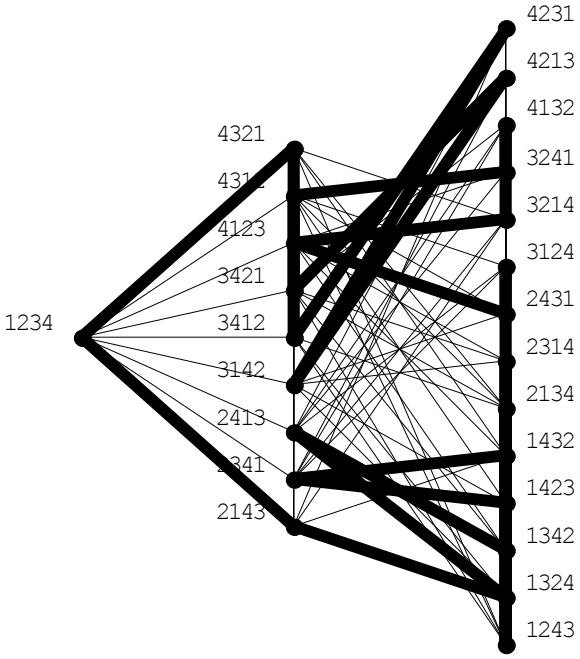



Проверим, будет ли этот граф гамильтоновым:

```
In[5]:= HamiltonianQ[h]
Out[5]= True
```

Выделим гамильтонов цикл в графе h:

```
In[6]:= l = HamiltonianCycle[dgraph]
Out[6]= {1, 8, 3, 11, 4, 7, 2, 9, 5, 10, 6, 13, 12, 19, 15, 20, 16, 23, 18, 21, 14, 22, 17, 24, 1}

In[7]:= ShowGraph[RankedEmbedding[Highlight[h, {Partition[HamiltonianCycle[h], 2, 1]}], {1}]]
```

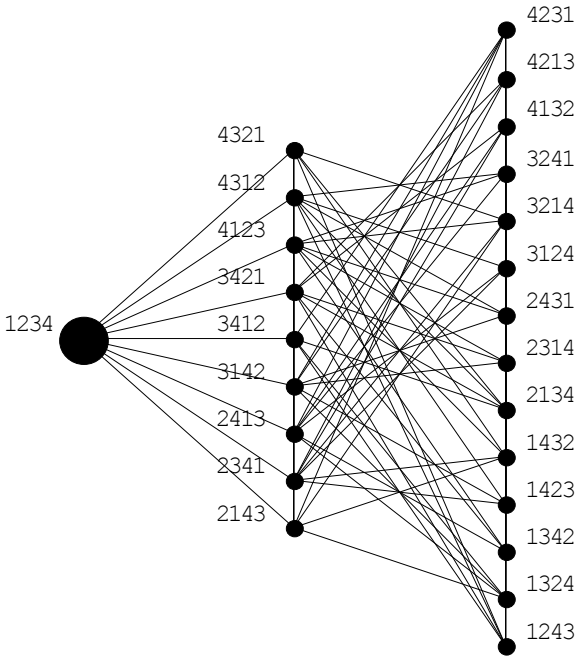



Чтобы последовательность вершин и ребер в гамильтоновом цикле была явно выделена, лучше выделить гамильтонов цикл анимацией. Для этого выделим последовательность вершин и ребер гамильтонова цикла:

```
In[8]:= s = {1, {1, 8}, 8, {8, 3}, 3, {3, 11}, 11, {11, 4}, 4, {4, 7}, 7, {7, 2}, 2, {2, 9}, 9,
        {9, 5}, 5, {5, 10}, 10, {10, 6}, 6, {6, 13}, 13, {13, 12}, 12, {12, 19}, 19, {19, 15},
        15, {15, 20}, 20, {20, 16}, 16, {16, 23}, 23, {23, 18}, 18, {18, 21}, 21, {21, 14}, 14,
        {14, 22}, 22, {22, 17}, 17, {17, 24}, 24, {24, 1}};
```

Эта команда в динамике покажет прохождение вершин цикла:

```
In[9]:= AnimateGraph[RankedEmbedding[h, {1}], s]
```




Дважды кликнув по этому рисунку, можно увидеть прохождение по всем вершинам и ребрам гамильтонова цикла.
Выделим перестановки, соответствующие вершинам гамильтонова цикла:

```
In[10]:= GetVertexLabels[h, HamiltonianCycle[h]]
Out[10]= {1234, 2143, 1324, 2413, 1342, 2134, 1243, 2314, 1423, 2341, 1432, 3124,
         2431, 4123, 3214, 4132, 3241, 4312, 3421, 4213, 3142, 4231, 3412, 4321, 1234}
```

Задача решена.
Эту задачу можно решить с помощью бинарного отношения Q:

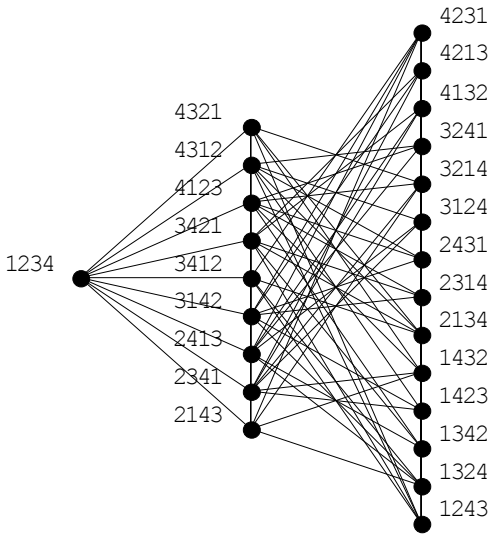
```
In[11]:= Q = (p = #1; q = #2; k = 0; Do[If[p[[i]] =!= q[[i]], , k = k + 1], {i, 1, Length[#1]}];
          If[k > 0, False, True]) &;
```

Или проще:

```
In[12]:= Q = (Count[#1 - #2, 0] == 0) &;
```

Построим граф отношения Q:

```
In[13]:= ShowGraph[
          RankedEmbedding[
           SetGraphOptions[g = MakeGraph[Permutations[4], Q, Type → Undirected, VertexLabel → True],
            {1, 8, 10, 11, 14, 17, 18, 19, 23, 24, VertexLabelPosition → UpperLeft}], {1}],
          PlotRange → 0.25]
```




Выведем вершины гамильтонова цикла:

```
In[14]:= GetVertexLabels[g, HamiltonianCycle[g]]
Out[14]= {1234, 2143, 1324, 2413, 1342, 2134, 1243, 2314, 1423, 2341, 1432, 3124,
          2431, 4123, 3214, 4132, 3241, 4312, 3421, 4213, 3142, 4231, 3412, 4321, 1234}
```

### *8.5. Задача "Похожие соседи"*

Требуется перебрать все перестановки множества {1,2,…,n} в таком порядке, чтобы каждая последующая перестановка отличалась от предыдущей ровно одной транспозицией смежных элементов.

**Решение.** Сформируем бинарное отношение R на множестве 4-перестановок: две перестановки находятся в отношении R, если они отличаются друг от друга транспозицией смежных элементов.

```
In[2]:= R = MemberQ[Table[p = #1; {p[[i]], p[[i + 1]]} = {p[[i + 1]], p[[i]]}; p, {i, 1, Length[#1] - 1}],
          #2] &;
```

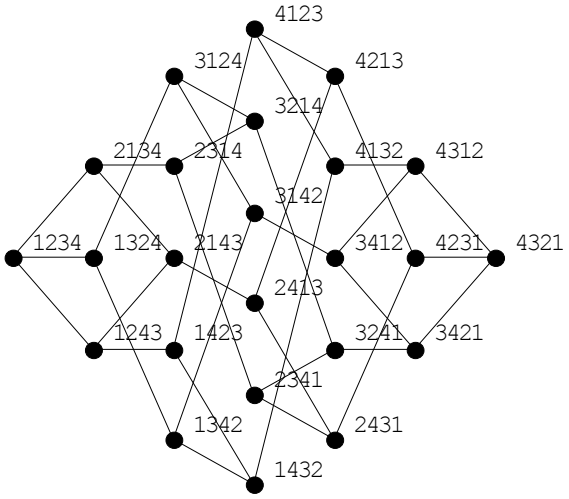
Сконструируем граф отношения R:

```
In[3]:= g = MakeGraph[Permutations[4], R, Type → Undirected, VertexLabel → True]
Out[3]= -Graph:<36, 24, Undirected>-
```

Получаем граф с 24 вершинами и 36 ребрами, связывающими перестановки, отличающиеся друг от друга транспозицией смежных элементов перестановки. Этот граф смежных транспозиций

является остовным подграфом графа транспозиций, так как он содержит все вершины графа транспозиций и некоторые его ребра.

```
In[4]:= ShowGraph[RankedEmbedding[g, {1}], PlotRange → 0.25]
```




Этот граф является гамильтоновым:

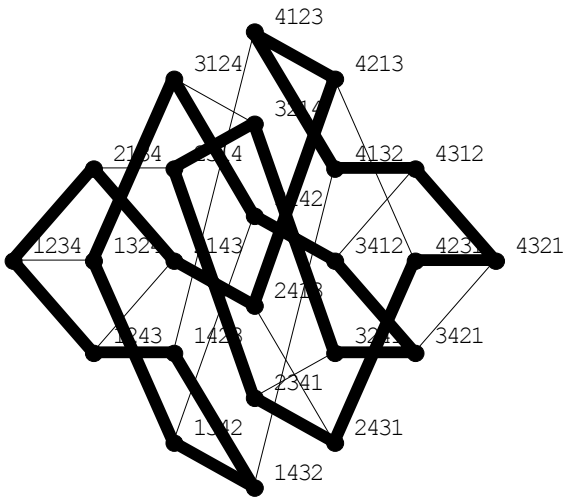
```
In[5]:= HamiltonianQ[g]
Out[5]= True
```

Вычислим гамильтонов цикл:

```
In[6]:= HamiltonianCycle[g]
Out[6]= {1, 2, 5, 6, 4, 3, 13, 14, 17, 18, 16, 15, 9, 10, 12, 22, 24, 23, 20, 19, 21, 11, 8, 7, 1}
```

Выделим гамильтонов цикл на графе:

```
In[7]:= ShowGraph[RankedEmbedding[Highlight[g, {Partition[HamiltonianCycle[g], 2, 1]}], {1}],
        PlotRange → 0.25]
```




Выведем перестановки, соответствующие вершинам гамильтонова пути:

```
In[8]:= GetVertexLabels[g, HamiltonianCycle[g]]
Out[8]= {1234, 1243, 1423, 1432, 1342, 1324, 3124, 3142, 3412, 3421, 3241, 3214,
         2314, 2341, 2431, 4231, 4321, 4312, 4132, 4123, 4213, 2413, 2143, 2134, 1234}
```

### *8.6. Задача "Инволюции"*

Можно ли перечислить все инволюции в таком порядке, чтобы каждая отличалась от предыдущей ровно одной транспозицией?

**Решение.** Рассмотрим все инволюции множества перестановок размера 4.

```
In[2]:= s = Involutions[4]
Out[2]= {{2, 1, 4, 3}, {2, 1, 3, 4}, {3, 4, 1, 2}, {3, 2, 1, 4}, {4, 3, 2, 1},
         {4, 2, 3, 1}, {1, 3, 2, 4}, {1, 4, 3, 2}, {1, 2, 4, 3}, {1, 2, 3, 4}}
```

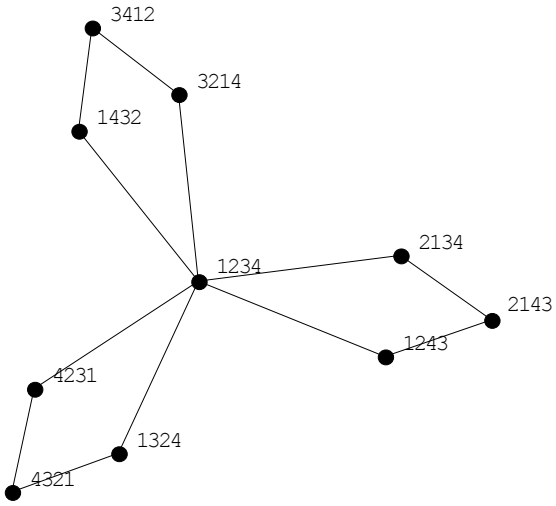
Построим бинарное отношение на множестве s: две инволюции находятся в отношении br, если одна инволюция отличается от другой ровно одной транспозицией.

```
In[3]:= br = (Count[#1 - #2, 0] == (Length[#1] - 2)) &;
```

Построим граф с вершинами из множества s, причем две вершины связаны ребром, если они находятся в отношении br.

```
In[4]:= g = MakeGraph[Involutions[4], br, Type → Undirected, VertexLabel → True];
```

```
In[5]:= ShowGraph[SpringEmbedding[g, 50], PlotRange → 0.2]
```




Этот граф, очевидно, не является гамильтоновым:

```
In[6]:= HamiltonianQ[g]
Out[6]= False
```

Следовательно, нельзя перечислить все инволюции размера 4 в требуемом порядке.

### *8.7. Задача "Нумерация разбиений"*

Перечислить все разбиения целого числа 7 так, что в каждом последующем разбиении одна часть разбиения увеличивается на 1, а другая уменьшается на 1.

**Решение.** Составим список разбиений числа 7одинаковой длины 7, для этого дополним каждое разбиение нулями.

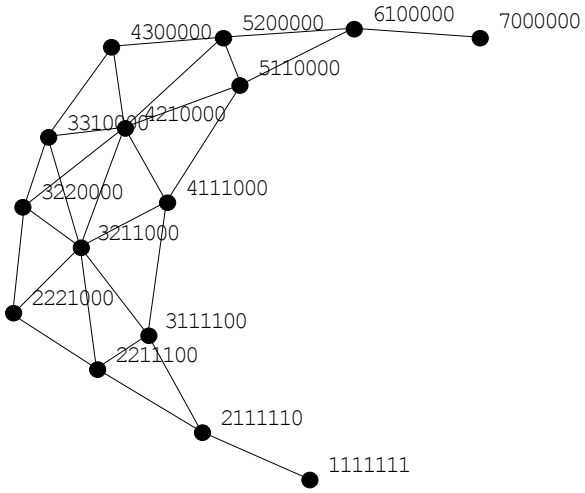
```
In[2]:= l = Map[Join[#, Table[0, {7 - Length[#]}]] &, Partitions[7]]
Out[2]= {{7, 0, 0, 0, 0, 0, 0}, {6, 1, 0, 0, 0, 0, 0}, {5, 2, 0, 0, 0, 0, 0},
        {5, 1, 1, 0, 0, 0, 0}, {4, 3, 0, 0, 0, 0, 0}, {4, 2, 1, 0, 0, 0, 0}, {4, 1, 1, 1, 0, 0, 0},
        {3, 3, 1, 0, 0, 0, 0}, {3, 2, 2, 0, 0, 0, 0}, {3, 2, 1, 1, 0, 0, 0}, {3, 1, 1, 1, 1, 0, 0},
        {2, 2, 2, 1, 0, 0, 0}, {2, 2, 1, 1, 1, 0, 0}, {2, 1, 1, 1, 1, 1, 0}, {1, 1, 1, 1, 1, 1, 1}}
```

Составим бинарное отношение между элементами списка l: два элемента находятся в отношении rel, если в каждом последующем разбиении одна часть разбиения увеличивается на1, а другая уменьшается на 1.

```
In[3]:= rel = Module[{t = #1 - #2}, Count[t, 1] == 1 && Count[t, -1] == 1 && Count[t, 0] == Length[t] - 2] &;
```

Построим граф отношения rel:

```
In[4]:= ShowGraph[g = SpringEmbedding[MakeGraph[l, rel, Type → Undirected, VertexLabel → True]],
        PlotRange → 0.3]
```




Этот граф негамильтонов

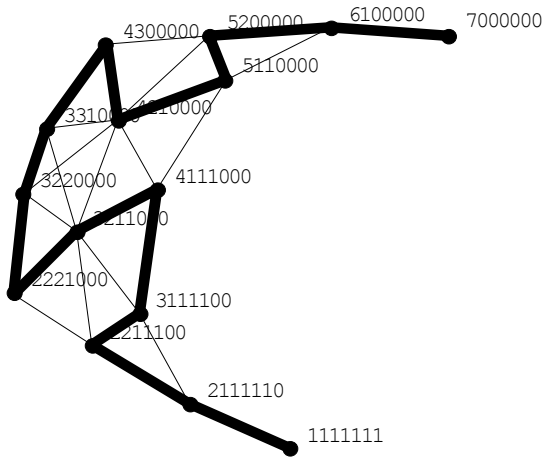
```
In[5]:= HamiltonianQ[g]
Out[5]= False
```

...но имеет гамильтонов путь:

```
In[6]:= HamiltonianPath[g]
Out[6]= {1, 2, 3, 4, 6, 5, 8, 9, 12, 10, 7, 11, 13, 14, 15}
```

Выделим гамильтонов путь в графе.

```
In[7]:= ShowGraph[Highlight[g, {Partition[HamiltonianPath[g], 2, 1]}], PlotRange → 0.3]
```

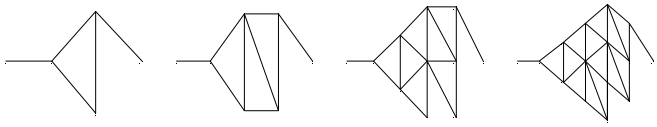



Перечислим разбиения в нужном порядке:

```
In[8]:= GetVertexLabels[g, HamiltonianPath[g]]
Out[8]= {7000000, 6100000, 5200000, 5110000, 4210000, 4300000, 3310000,
         3220000, 2221000, 3211000, 4111000, 3111100, 2211100, 2111110, 1111111}
```

Рассмотрим графы отношения rel для различных n:

```
In[9]:= ShowGraphArray[
          h =
           Table[RankedEmbedding[MakeGraph[l1 = Map[Join[#, Table[0, {7 - Length[#]}]] &, Partitions[n]],
              rel, Type → Undirected], {1}], {n, 4, 7}]]
```



Выведем гамильтоновы пути в этих графах:

```
In[10]:= Map[HamiltonianPath, h] // ColumnForm
Out[10]= {1, 2, 3, 4, 5}
         {1, 2, 3, 4, 5, 6, 7}
         {1, 2, 4, 3, 5, 6, 8, 9, 7, 10, 11}
         {1, 2, 3, 4, 6, 5, 8, 9, 12, 10, 7, 11, 13, 14, 15}
```

### *8.8. Задача "Нумерация композиций"*

Можно ли перечислить композиции целого числа в таком порядке, что каждая последующая композиция отличается от предыдущей увеличением одной части на единицу и соответственным уменьшением на единицу другой части.

**Решение.** Рассмотрим, например, композицию числа 4 на три части.

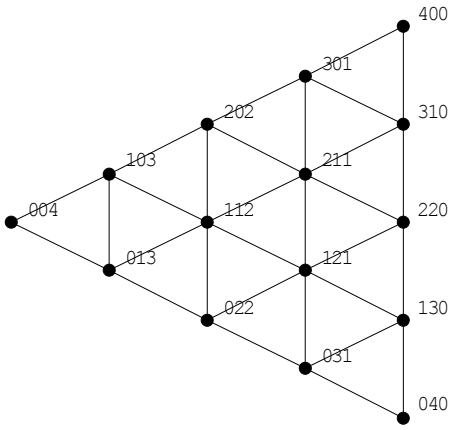
```
In[2]:= s = Compositions[4, 3]
Out[2]= {{0, 0, 4}, {0, 1, 3}, {0, 2, 2}, {0, 3, 1}, {0, 4, 0}, {1, 0, 3}, {1, 1, 2},
         {1, 2, 1}, {1, 3, 0}, {2, 0, 2}, {2, 1, 1}, {2, 2, 0}, {3, 0, 1}, {3, 1, 0}, {4, 0, 0}}
```

Сформируем нужное бинарное отношение:

```
In[3]:= cr = Module[{d = #1 - #2}, Count[d, -1] == 1 && Count[d, 1] == 1 && Count[d, 0] == Length[d] - 2] &;
```

Граф этого отношения – решетчатый граф

```
In[4]:= ShowGraph[g = RankedEmbedding[MakeGraph[s, cr, Type → Undirected, VertexLabel → True], {1}],
         PlotRange → 0.25]
```

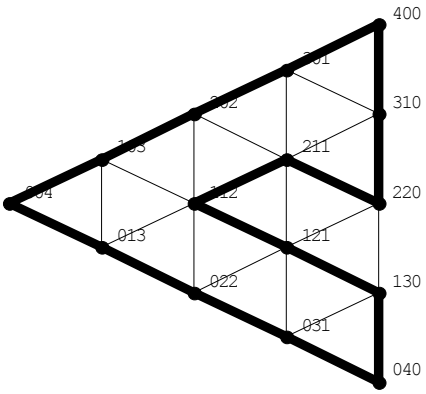



Граф g – гамильтонов

```
In[5]:= HamiltonianQ[g]
Out[5]= True
```

Выделим гамильтонов цикл графа g

```
In[6]:= ShowGraph[Highlight[g, {Partition[HamiltonianCycle[g], 2, 1]}], PlotRange → 0.25]
```




Выделим метки вершин гамильтонова цикла

```
In[7]:= GetVertexLabels[g, HamiltonianCycle[g]]
Out[7]= {004, 013, 022, 031, 040, 130, 121, 112, 211, 220, 310, 400, 301, 202, 103, 004}
```

В данном случае гамильтонов цикл находится быстро:

```
In[8]:= Timing[HamiltonianCycle[g];]
Out[8]= {0.047 Second, Null}
```

Граф g содержит  пятьдесят два гамильтоновых цикла

```
In[9]:= Length[HamiltonianCycle[g, All]]
Out[9]= 52
```

### *8.9. Задача « Разлив вина»*

Два человека имеют полный кувшин вина в 8 литров, а также два пустых кувшина в 5 и 3 литра. Как они могут поделить вино поровну?

**Решение.** Рассмотрим список возможных наполнений трех данных кувшинов:

```
In[2]:= j = {{8, 0, 0}, {5, 0, 3}, {3, 5, 0}, {0, 5, 3}, {5, 3, 0}, {0, 5, 3}, {3, 2, 3}, {6, 2, 0},
          {6, 0, 2}, {1, 5, 2}, {1, 4, 3}, {4, 4, 0}};
```

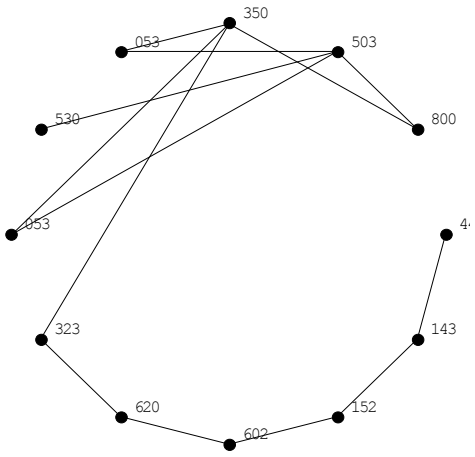
Сформируем бинарное отношение на элементах множества j:

```
In[3]:= pr =
        Module[{t = #1 - #2}, (Count[t, -1] == 1 && Count[t, -1] == 1 && Count[t, 0] == 1) && t == {0, 1, -1} ||
           (Count[t, -2] == 1 && Count[t, -2] == 1 && Count[t, 0] == 1) && t =!= {2, -2, 0} &&
            t =!= {2, 0, -2} && t =!= {-2, 2, 0} ||
           (Count[t, 0] == Length[t] - 2) && Count[t, -3] == 1 && Count[t, 3] == 1 && t =!= {3, -3, 0} &&
            t =!= {-3, 3, 0} || (Count[t, -5] == 1 && Count[t, 5] == 1 && Count[t, 0] == 1)] &;
```

Построим граф этого отношения:

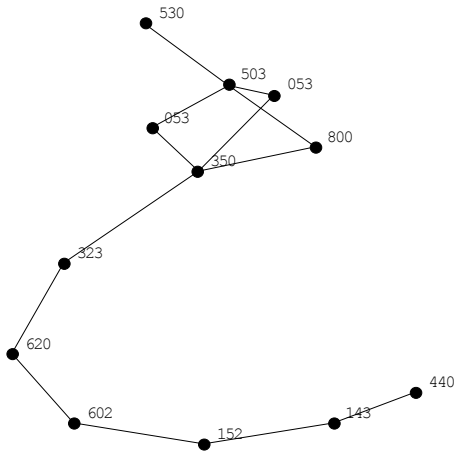
```
In[4]:= g = MakeGraph[j, pr, Type → Undirected, VertexLabel → True]
Out[4]= -Graph:<13, 12, Undirected>-
```

```
In[5]:= ShowGraph[g]
```




Улучшим вложение с помощью SpringEmbedding:

```
In[6]:= ShowGraph[SpringEmbedding[g]]
```




Вычислим кратчайший путь от первой вершины до двенадцатой

```
In[7]:= ShortestPath[g, 1, 12]
Out[7]= {1, 3, 7, 8, 9, 10, 11, 12}
```

Выделим соответствующие этому кратчайшему пути элементы множества j:

```
In[8]:= Map[j[[#]] &, ShortestPath[g, 1, 12]]
Out[8]= {{8, 0, 0}, {3, 5, 0}, {3, 2, 3}, {6, 2, 0}, {6, 0, 2}, {1, 5, 2}, {1, 4, 3}, {4, 4, 0}}
```

### *8.10. Задача "Ревнивые мужья "*

Три ревнивых мужа и их жены должны переправиться через реку. Имеется только один маленький бот, который может выдержать одновременно только двоих человек. Как могут переправиться все шестеро, если никакой муж не оставит жену в присутствии других мужчин?

**Решение**. Пусть нужно переправиться с левого берега на правый берег. Сформируем множество l всех возможных сочетаний людей на левом берегу, причем в каждом сочетании последний символ b означает наличие бота у группы оставшихся, а символ o – отсутствие такового.

```
In[2]:= l = {{m1, m2, m3, v1, v2, v3, b}, {m1, m2, m3, v1, b}, {m1, m2, m3, v2, b}, {m1, m2, m3, v3, b},
        {m1, m2, v1, v2, b}, {m1, m3, v1, v3, b}, {m2, m3, v2, v3, b}, {m1, m2, m3, v2, b},
        {m1, m2, m3, v3, b}, {m1, m2, m3, v2, v3, b}, {m1, m2, m3, v1, v3, b}, {m1, m2, m3, v1, v2, b},
        {m1, v1, b}, {m2, v2, b}, {m3, v3, b}, {v1, v2, v3, b}, {v1, v2, b}, {v1, v3, b},
        {v2, v3, b}, {m1, m2, m3, v1, o}, {m1, m2, m3, v2, o}, {m1, m2, m3, v3, o},
        {m1, m2, v1, v2, o}, {m1, m3, v1, v3, o}, {m2, m3, v2, v3, o}, {m1, m2, m3, v2, o},
        {m1, m2, m3, v3, o}, {m1, m2, m3, o}, {m1, v1, o}, {m2, v2, o}, {m3, v3, o}, {v1, v2, o},
        {v1, v3, o}, {v2, v3, o}, {v1, o}, {v2, o}, {v3, o}, {o}};
```

Этот список содержит 38 элементов

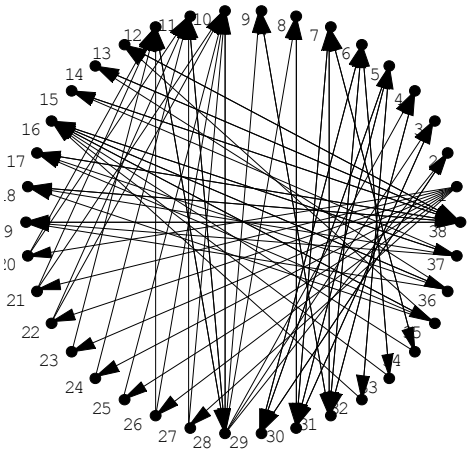
```
In[3]:= Length[l]
Out[3]= 38
```

Построим  бинарное отношение на множестве l:

```
In[4]:= kr =
        Module[{t = Complement[#1, #2]},
          Length[#2] - Length[#1] == 1 && Last[#1] =!= Last[#2] && Last[#1] == o && Length[#2] =!= 7 &&
            Length[Complement[#2, #1]] == 2 ||
           Length[#2] - Length[#1] == 2 && Last[#1] =!= Last[#2] && Last[#1] == o && Length[#2] =!= 7 &&
            Length[Complement[#2, #1]] == 3 ||
           (Length[#1] - Length[#2] == 2 && Last[#1] =!= Last[#2] && Length[Complement[#1, #2]] == 3 &&
              Last[#1] == b)] &;
```

Построим граф с вершинами в точках множества l, причем две вершины связаны ребром, если они удовлетворяют отношению kr:

```
In[5]:= ShowLabeledGraph[g = MakeGraph[l, kr], EdgeDirection → True]
```



```
In[6]:= g
Out[6]= -Graph:<90, 38, Directed>-
```

Рассмотрим кратчайший путь из первой вершины в тридцать восьмую:

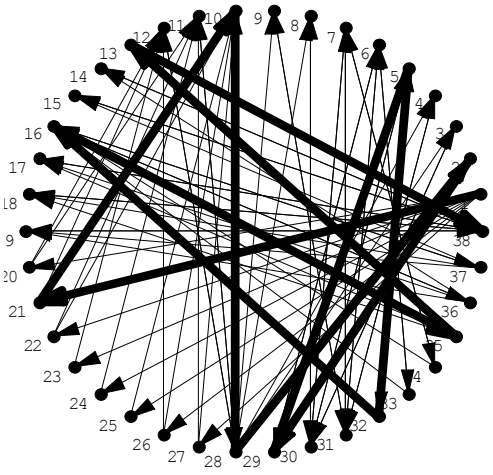
```
In[7]:= ShortestPath[g, 1, 38]
Out[7]= {1, 21, 10, 28, 2, 29, 5, 32, 16, 35, 13, 38}
```

Рассмотрим исходные метки соответствующих вершин:

```
In[8]:= Map[l[[#]] &, ShortestPath[g, 1, 38]]
Out[8]= {{m1, m2, m3, v1, v2, v3, b}, {m1, m2, m3, v2, o},
         {m1, m2, m3, v2, v3, b}, {m1, m2, m3, o}, {m1, m2, m3, v1, b}, {m1, v1, o},
         {m1, m2, v1, v2, b}, {v1, v2, o}, {v1, v2, v3, b}, {v1, o}, {m1, v1, b}, {o}}
```

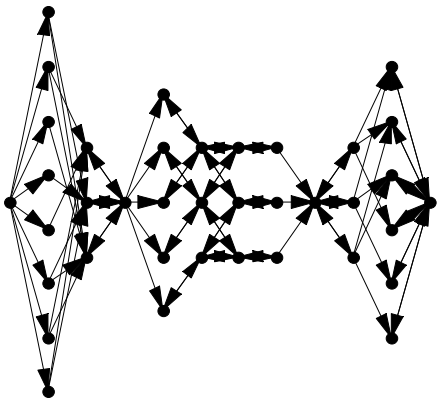
Выделим кратчайший путь от первой до тридцать восьмой вершины:

```
In[9]:= ShowLabeledGraph[Highlight[g, {Partition[ShortestPath[g, 1, 38], 2, 1]}]]
```



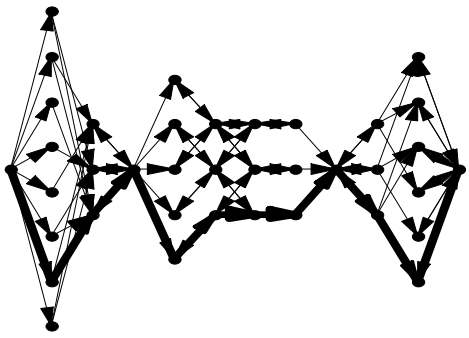
Улучшим вложение графа с помощью RankedEmbedding:

```
In[10]:= ShowGraph[RankedEmbedding[g, {1}], EdgeDirection → True]
```



Выделим гамильтонов цикл:

```
In[11]:= ShowGraph[Highlight[RankedEmbedding[g, {1}], {Partition[ShortestPath[g, 1, 38], 2, 1]}]]
```



### *8.11. Задача "Самолеты"*

Имеется некоторый город M, который связан маршрутами с городами $A_1,A_2,\ldots,An$. Пусть, согласно расписанию, маршрут $MA_iM$ обслуживается в интервале времени $[a_i,b_i]$. Таким образом, расписание рейсов в аэропорту задано списком интервалов $[a_1,b_1]$, $[a_2,b_2],\ldots,[a_n,b_n]$, где $a_i$ – время

вылета, $b_i$ – время возврата. Найти наименьшее количество самолетов, требующееся для обслуживания всех рейсов.

**Решение.** При решении этой задачи требуется найти хроматическое число графа. Действительно, все рейсы, имеющие одинаковый цвет, могут быть обслужены одним самолетом. С другой стороны, если имеется некоторое число самолетов для обслуживания всех рейсов, то, окрасив одним цветом все рейсы, обслуживаемые одним самолетом, мы получим некоторую правильную раскраску графа g.
Строим граф, вершины которого соответствуют интервалам $[a_i,b_i]$, и две вершины смежны, если соответствующие им интервалы пересекаются.
Легко видеть, что хроматическое число этого графа есть решение этой задачи.
Например, пусть есть расписание в виде следующего списка:

```
In[2]:= l = {{3.03, 21.07}, {9.53, 10.36}, {2.34, 9.43}, {5.29, 10.08}, {10.78, 20.00},
        {3.63, 16.37}, {14.04, 20.00}, {19.58, 22.23}, {12.24, 19.58}, {15.56, 16.37},
        {2.34, 21.12}, {15.13, 15.34}, {2.34, 5.39}, {12.28, 20.00}, {5.29, 9.29}};
```

С помощью функции IntervalGraph по данному списку построим граф и вычислим его хроматическое число:

```
In[3]:= ChromaticNumber[IntervalGraph[l]]
Out[3]= 8
```

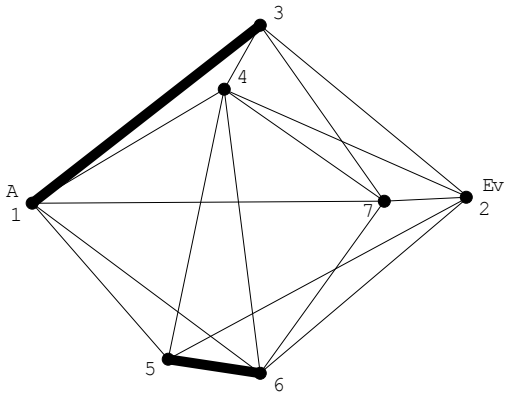
То есть для обслуживания данного расписания достаточно 8 самолетов.
**Примечание**. Задача определения хроматического числа относится к классу NP-трудных.
Время вычисления хроматического числа этого графа с 15-ю вершинами и 63-ребрами занимает около 5 минут:

```
In[4]:= Timing[ChromaticNumber[IntervalGraph[l]];]
Out[4]= {334.984 Second, Null}
```

### *8.12. Задача "Курорт Even"*

Расписание поездов задано тройками {A, t, B}, где A – пункт отправления, B – пункт назначения, а целое число t- количество дней в пути. Известно, что на курорте Even в гостиницы заселяют только по нечетным числам. Группе туристов, выезжающей в определенный день, требуется за минимальное количество дней прибыть на курорт Even в приемный день.

**Решение**. Можно построить взвешенный граф с вершинами во всех городах расписания и ребрами {A, B} веса t, соответствующих тройкам {A, t, B}.
Далее применяем алгоритм Дейкстры для нахождения кратчайшего пути между двумя вершинами, но кратчайший путь может оказаться таким, что время прибытия – нечетный день.
Чтобы алгоритм Дийкстры возвращал нужный путь, построим новый граф, состоящий из двух копий построенного графа, в котором ребра нечетного веса {A, t, B} заменяются на два ребра: $\{A, t, B_{copy}\}$ и $\{A_{copy}, t, B\}$, где $A_{copy}$, $B_{copy}$ есть копии вершин A, B.
Например, по некоторому расписанию построен следующий граф g




со следующими весами ребер (на рисунке выделены ребра нечетного веса)

```
In[2]:= Edges[g, EdgeWeight]
Out[2]= {{{3, 4}, 2}, {{3, 7}, 4}, {{4, 5}, 2}, {{4, 6}, 2}, {{4, 7}, 2},
        {{5, 6}, 3}, {{6, 7}, 4}, {{1, 3}, 1}, {{1, 4}, 4}, {{1, 5}, 6}, {{1, 6}, 2},
        {{1, 7}, 4}, {{2, 3}, 6}, {{2, 4}, 4}, {{2, 5}, 2}, {{2, 6}, 4}, {{2, 7}, 2}}
```

Найдем кратчайший путь от пункта A до пункта Ev

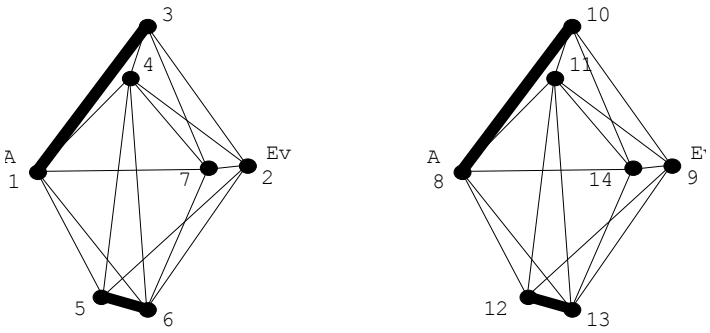
```
In[3]:= путь = ShortestPath[g, 1, 2, Algorithm → Dijkstra]
Out[3]= {1, 6, 2}
```

Цена этого пути:

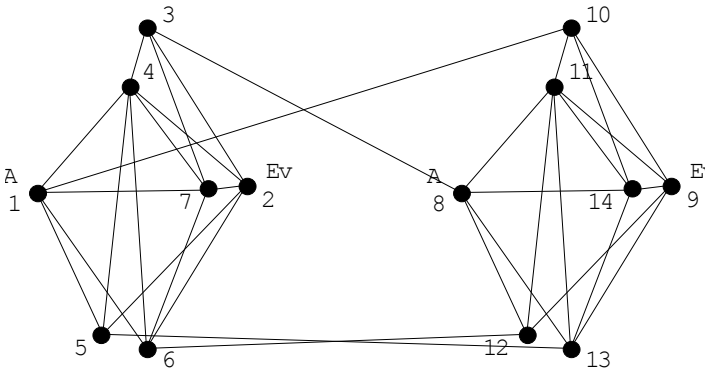
```
In[4]:= CostOfPath[g, путь]
Out[4]= 6
```

Предположим, что группа выезжает из пункта A в четный день. Тогда она прибудет в Ev в четный день, когда нельзя поселиться в гостинице.
Построим новый граф, создавая две копии графа g:

```
In[5]:= ShowGraph[новыйГраф = GraphUnion[2, g]]
```




При помощи функций удаления и добавления ребер проведем вышеописанную операцию замены ребер нечетного веса




Теперь находим кратчайший путь до вершины 9=Ev в копии

```
In[7]:= нечетныйпуть = ShortestPath[новыйГраф, 1, 9, Algorithm → Dijkstra]
Out[7]= {1, 10, 9}
```

Цена этого пути:

```
In[8]:= CostOfPath[новыйГраф, нечетныйпуть]
Out[8]= 7
```

Из списка весов  нового графа можно проверить сумму весов ребер {1, 10} и {10, 9}:

```
In[9]:= Edges[новыйГраф, EdgeWeight]
```

```
Out[10]= {{{3, 4}, 2}, {{3, 7}, 4}, {{4, 5}, 2}, {{4, 6}, 2}, {{4, 7}, 2}, {{6, 7}, 4}, {{1, 4}, 4},
          {{1, 5}, 6}, {{1, 6}, 2}, {{1, 7}, 4}, {{2, 3}, 6}, {{2, 4}, 4}, {{2, 5}, 2}, {{2, 6}, 4},
          {{2, 7}, 2}, {{10, 11}, 2}, {{10, 14}, 4}, {{11, 12}, 2}, {{11, 13}, 2}, {{11, 14}, 2},
          {{13, 14}, 4}, {{8, 11}, 4}, {{8, 12}, 6}, {{8, 13}, 2}, {{8, 14}, 4}, {{9, 10}, 6}, {{9, 11}, 4},
          {{9, 12}, 2}, {{9, 13}, 4}, {{9, 14}, 2}, {{1, 10}, 1}, {{3, 8}, 1}, {{5, 13}, 3}, {{6, 12}, 3}}
```

### *8.13. Задача "Острова"*

В прямоугольной матрице, состоящей из нулей и единиц, две клетки считаются соседними, если они соседствуют по вертикали, горизонтали, или по диагоналям. Островом будем считать множество клеток состоящих из единиц, если любые две клетки острова можно соединить путем из единиц. Требуется определить количество островов и их площади.

**Решение.** Например, пусть дана матрица b:

$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Построим граф, вершины которого есть клетки матрицы b, содержащие единицы, и две вершины смежны, если соответствующие клетки соседние.
Построим список всех вершин:
Расположим элементы матрицы b в одной строке l:

```
In[3]:= l = Flatten[b, 1]
Out[3]= {0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0,
         1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0,
         1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0}
```

Выделим позиции единиц в списке l:

```
In[4]:= v = Flatten[Position[l, 1]]
Out[4]= {2, 4, 6, 7, 8, 10, 12, 14, 15, 16, 17, 18, 20, 21, 25, 26, 28, 30, 31, 32, 33, 35, 36, 38, 39, 45,
         46, 47, 48, 49, 51, 52, 55, 57, 60, 61, 68, 69, 70, 71, 81, 85, 86, 87, 88, 92, 94, 95, 98, 99}
```

Список v есть список вершин графа.
Построим функции, которые по номеру вершины определяет строку и столбец соответствующей клетки матрицы.

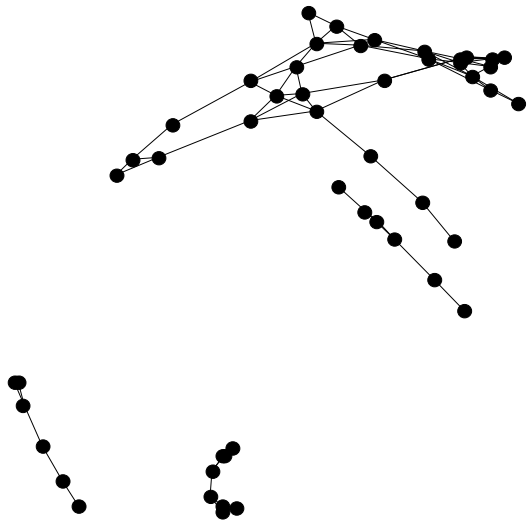
```
In[5]:= i[n_] = Floor[(n - 1) / 10] + 1;
        j[k_] = Mod[k - 1, 10] + 1;
```

Сформируем бинарное отношение соседства.

```
In[7]:= rel = Max[Abs[i[#1] - i[#2]], Abs[j[#1] - j[#2]]] == 1 &;
```

Рассмотрим граф отношения rel:

```
In[8]:= g = MakeGraph[v, rel, Type → Undirected];

In[9]:= ShowGraph[SpringEmbedding[g, 10]]
```



Выделим связные компоненты этого графа, т.е. искомые острова

```
In[10]:= Острова = ConnectedComponents[g]

Out[10]= {{1, 7, 14, 19, 20, 21}, {2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 15, 16, 17, 18, 22, 23, 24, 25, 26, 27,
          28, 29, 30, 33, 34, 35, 37, 38, 39}, {31, 32, 36, 40, 41, 46}, {42, 43, 44, 45, 47, 48, 49, 50}}
```

Количество связных компонент есть число островов

```
In[11]:= Length[Острова]
Out[11]= 4
```

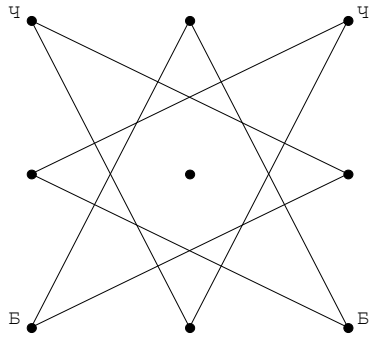
Площадь островов легко вычислить функцией  Length , применяя ее к элементам списка острова:

```
In[12]:= Map[Length, Острова]
Out[12]= {6, 30, 6, 8}
```

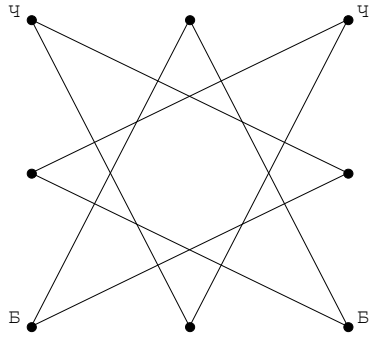
### *8.14. Задача "Перемена мест"*

Требуется за минимальное количество ходов поменять местами белых и черных коней.

**Решение**. Построим граф g ходов коня функцией KnightsTourGraph[3,3]

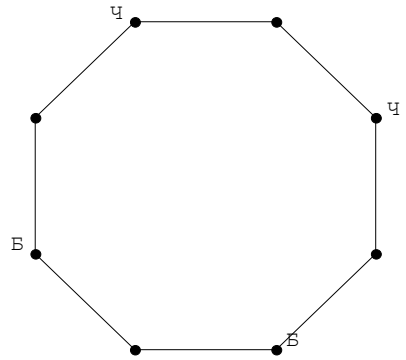



Удалим из него центральную вершину




Улучшим его вложение, многократно применяя функцию SpringEmbedding:

```
In[2]:= ShowGraph[SpringEmbedding[g, 100]]
```



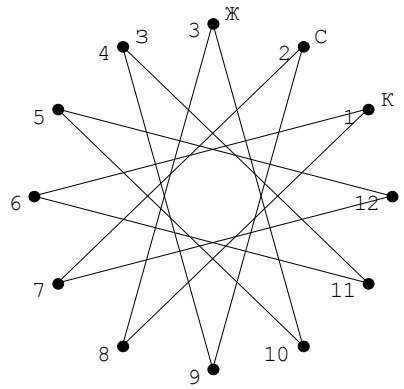

Повернув этот рисунок на 180 градусов, получим решение задачи. При этом каждый конь совершает четыре хода, то есть всего нужно совершить 16 ходов.

### *8.15. Задача «Разноцветные фишки»*

12 полей расположены по кругу и на четырех соседних полях стоят четыре разноцветные фишки: красная, желтая, зеленая и синяя.
Одним ходом можно передвинуть любую фишку с поля, на котором она стоит, через четыре любых поля на пятое (если оно свободно) в любом из двух возможных направлений. После нескольких ходов фишки могут стать снова снова на те же четыре поля. Как они могут при этом переставиться?

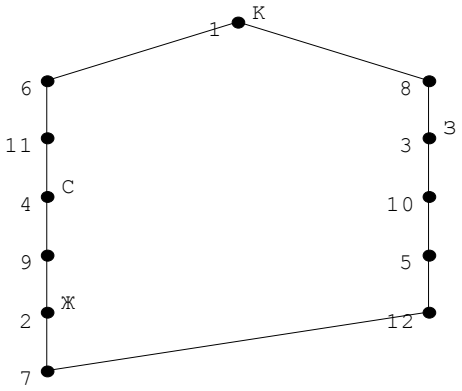
**Решение.** Будем считать двенадцать полей вершинами графа, ребра которого соединены через четыре вершины в каждом направлении. Это обычный циркулянт граф. Построим этот граф:

```
In[2]:= ShowLabeledGraph[
          g = SetGraphOptions[CirculantGraph[12, 5],
            {{1, 2, 3, 4, VertexLabel → {К, С, Ж, З}}}], PlotRange → 0.15,
          TextStyle → {FontSize → 14}]
```




Изменим порядок расположения полей по кругу, а именно расположим их в порядке, при котором можно переходить с одного поля на соседнее. Мы можем считать, что имеем 12 полей, расположенных именно таким образом (ведь фактически занимаемое полем место значения не имеет). Для этого достаточно изменить вложение графа на корневое вложение с корнем, например, в первой вершине:

```
In[3]:= ShowGraph[RootedEmbedding[g, 1], {1, 2, 3, 4, VertexLabel → {"К", "Ж", "З", "С"}},
          PlotRange → 0.15, TextStyle → {FontSize → 13}, VertexNumber → True]
```




Правило движения фишек при таком расположении полей оказывается чрезвычайно простым – каждая фишка может сдвинуться на одно поле влево или вправо, если только это соседнее поле не занято.

Теперь совершенно ясно, что единственный способ, каким фишки могут поменяться местами – это двигаться по кругу в одном или другом направлении: ведь ни одна фишка не может «перегнать» другую, так как другая преграждает ей путь. Таким образом, если фишка К займет поле 4, то фишка С должна будет занять поле 2, фишка Ж –поле 3 и фишка З – поле 1. Если фишка К займет поле 2, то фишка С должна будет занять поле 3, фишка Ж – поле 1, фишка З – поле 4. Если фишка К займет поле 3, то фишка С должна будет занять поле 1, фишка Ж – поле 4 и фишка З – поле 2.

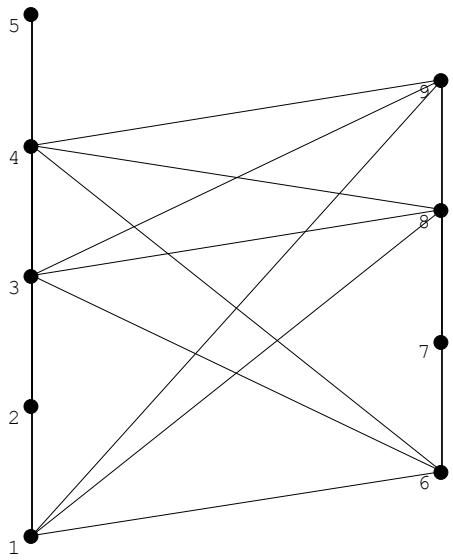
Никакие другие новые расположения фишек невозможны.

### *8.16. Задача “Новая группа”*

В каждой из двух групп отдыхающих все взаимно знакомы друг с другом. Кроме того, некоторые отдыхающие одной группы имеют знакомых из другой группы. Требуется из этих двух групп сформировать наибольшую группу взаимно знакомых отдыхающих.

**Решение.** Представим отдыхающих вершинами графа, а знакомства ребрами. Тогда условию задачи соответствует задача нахождения в этом графе максимальной клики. Но она является классической NP-полной задачей, т. е. для ее решения требуется значительное время. А можно ли свести ее к задаче полиномиальной сложности?
Да. Функцией GraphComplement[g] построим дополнение графа и ищем в нем максимальное независимое множество. В результате операции дополнения получается двудольный граф и функция MaximumIndependentSet[g] в этом случае вычисляется за полиномиальное время.
Проверим результаты. Пусть знакомства представлены графом g:

```
In[2]:= g = FromAdjacencyLists[{{2, 3, 4, 5, 6, 8, 9}, {3, 4, 5}, {4, 5, 6, 8, 9},
          {6, 5, 8, 9}, {4}, {7, 8, 9}, {8, 9}, {9}, {8}},
         {{-2, -2}, {-2, -1}, {-2, 0}, {-2, 1}, {-2, 2}, {2, -1.5}, {2, -0.5},
          {2, 0.5}, {2, 1.5}}]
Out[2]= -Graph:<25, 9, Undirected>-

In[3]:= ShowLabeledGraph[g]
```




Найдем максимальную клику в графе g:

```
In[4]:= l = MaximumClique[g]
Out[4]= {1, 3, 4, 6, 8, 9}
```

и сравним с максимальным независимым множеством в дополнении графа g:

```
In[5]:= l = MaximumIndependentSet[GraphComplement[g]]
Out[5]= {1, 3, 4, 6, 8, 9}
```

Ответы совпадают, но время вычисления второй функции на двудольном графе полиномиально, а первой – экспоненциально.

```
In[6]:= {Timing[MaximumClique[g];],
         Timing[MaximumIndependentSet[GraphComplement[g]];]} // ColumnForm
Out[6]= {0.063 Second, Null}
        {0.031 Second, Null}
```

### *8.17. Задача "Сталкер"*

Сталкер получил задание добраться до склада в промзоне. Он нашел в электронном архиве несколько карт территории промзоны, на каждой из которых есть информация только о некоторых дорогах промзоны. Одна и та же дорога может присутствовать на нескольких картах.
В пути сталкер может загружать из архива на мобильный телефон по одной карте. При загрузке новой карты предыдущая карта в памяти телефона стирается. Сталкер может перемещаться лишь по дорогам, отмеченным на карте, загруженной на данный момент. Каждая загрузка карты стоит 1 рубль. Сталкеру требуется как можно меньшее число раз загружать карты.

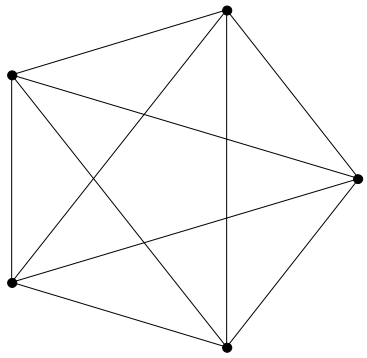
**Решение.** Унифицируем задачу к задаче нахождения кратчайшего пути на графе. Карты соответствуют вершинам графа; вершины графа смежны, если соответствующие карты имеют общие дороги. Весам ребер присваиваем значение 1. При этом загрузке карты соответствует переход по ребру. Определяем номера s и t карт, на которых отмечены старт и склад, и выполняем функцию ShortestPath[g, s, t, Algorithm->Dijkstra].

### *8.18. Задача "Помощь маляру"*

"Однорукий" маляр умеет так раскрасить вершины графа, что смежные вершины будут окрашены разными цветами, но не умеет так же раскрасить ребра. Помогите ему.

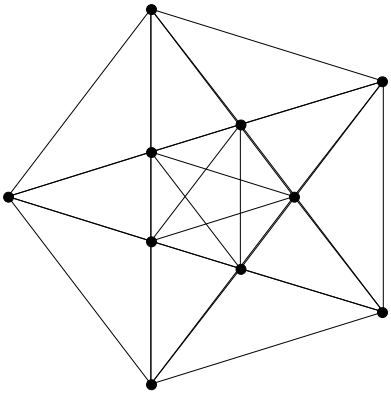
**Решение.** Например, пусть g есть полный граф:

```
In[2]:= ShowGraph[g = CompleteGraph[5]];
```



Ребра графа g представим вершинами нового графа gНовый. Вершины этого графа смежны, если и только если смежны ребра исходного графа. Он строится функцией LineGraph:

```
In[3]:= ShowGraph[gНовый = LineGraph[g]]
```

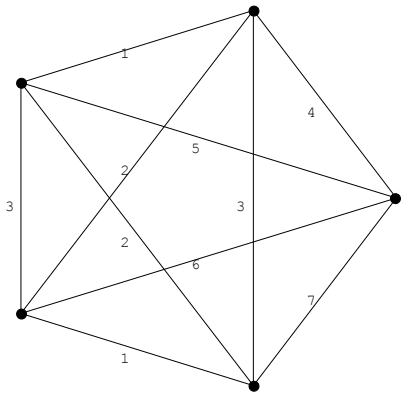


Отдаем маляру для покраски новый граф функцией MinimumVertexColoring [gНовый]:

```
In[4]:= MinimumVertexColoring[gНовый]
Out[4]= {1, 2, 3, 4, 3, 4, 5, 5, 1, 2}
```

Здесь выведена минимальная правильная раскраска 12 вершин нового графа 5-ю красками. Проверим, как сработает функция EdgeColoring на старом графе:

```
In[5]:= EdgeColoring[g]
Out[5]= {1, 2, 3, 4, 3, 2, 5, 1, 6, 7}
```



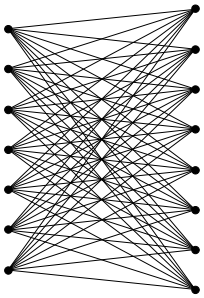

Потребовалось 7 красок. Но маляра можно понять. Правильная раскраска графа есть NP-трудная задача. Функция EdgeColoring для поиска правильной раскраски использует эвристический алгоритм Брелаза.

### *8.19. Задача "Обед"*

После окончания некоторого фестиваля организаторы решили устроить совместный обед для всех делегаций. Могут ли они усадить за столики участников таким образом, чтобы за каждым столом находились участники разных делегаций? Количество участников каждой делегации и количество и вместимость каждого стола организаторам известны.

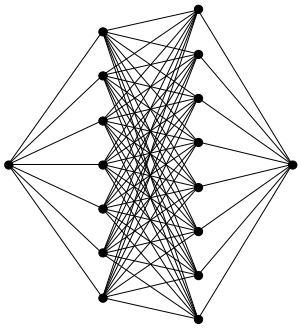
**Решение.** Построим полный двудольный граф g. Вершины одной доли соответствуют делегациям, а вершины второй доли соответствуют обеденным столам.

```
In[2]:= g = CompleteKPartiteGraph[7, 8]; ShowGraph[g];
```



Добавим к графу источник и сток по следующей схеме: источник соединим ребрами с делегациями, сток – со столами.

```
In[3]:= ShowGraph[g];
```



Присвоим веса ребрам, соединяющим источник с делегациями (в каждой делегации ровно семь человек )

```
In[4]:= g = SetEdgeWeights[g, Table[{16, i}, {i, 1, 7}], {7, 7, 7, 7, 7, 7, 7}];
```

Теперь присвоим веса ребрам, соединяющие столы со стоком:

```
In[5]:= g = SetEdgeWeights[g, Table[{17, i}, {i, 8, 15}], {6, 7, 7, 10, 5, 10, 7, 4}];
```

Вычислим максимальный поток графа g:

```
In[6]:= NetworkFlow[g, 16, 17]
Out[6]= 49
```

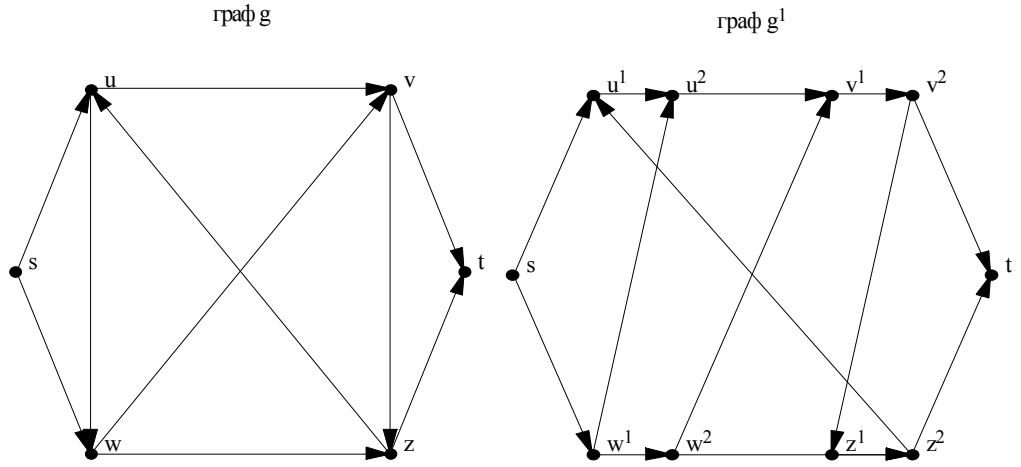
Получили, что все сорок девять участников распределились требуемым образом, т.к. величина стока равна 49.

### *8.20. Задача “Маршруты модниц”*

А. Обычно модницы, увидев на улице другую модницу с такой же шляпкой, испытывают дискомфорт. Помогите проложить максимальное число путей, связывающих пункты s и t некоторого города так, чтобы исключить подобные коллизии.
Б. То же самое, но маршруты не пересекаются на перекрестках.

**Решение.**
А) В теории графов такие пути называются реберно-непересекающимися s-t путями. По теореме Менгера, максимальное число реберно-непересекающимися s-t путей в графе G равно мощности минимального s-t разреза, минимальному числу ребер, удаление которых из связного графа разделяет вершины s и t. А по теореме Форда-Фалкерсона, пропускная способность минимального s-t разреза равна величине максимального потока между вершинами s и t. В свою очередь поме-

чивающий алгоритм Форда-Фалкерсона позволяет эффективно вычислить величину максимального потока.
Для решения исходной задачи достаточно присвоить ребрам графа вес 1, кстати, принятого по умолчанию, и передать его функции NetworkFlow[g, s, t]. Для примера смотреть задачу "Диверсант".
Замечание. Задача для нахождения максимального потока применяется для ориентированных графов. Для неориентированного графа g алгоритм строит граф d(g) – ориентированный граф, полученный заменой каждого ребра e графа g парой противоположно ориентированных ребер, инцидентных тем же вершинам, что и e. Можно легко показать, что
1).Существует взаимно-однозначное соответствие между путями графа g и ориентированными путями в графе d(g) и
2). Для любых двух вершин s и t минимальное число ребер, удаление которых из графа разрушает все s–t-пути в графе g, равно минимальному числу ребер, удаление которых разрушает все ориентированные пути в графе d(g).
Б) Решаем задачу аналогично задаче А) на другом графе.
Пусть s и t – две несмежные вершины в ориентированном графе g=(V,E). Построим из графа g ориентированный граф $g^1$ следующим образом:
1.Расщепим вершину $v \in V-\{s,t\}$ на две новые вершины $v^1$ и $v^2$ и соединим их ориентированным ребром $(v^1, v^2)$.
2. Заменим каждое ребро графа g с конечной вершиной $v \in V-\{s,t\}$ на новое ребро, имеющее $v^1$ в качестве конечной вершины.
3. Заменим каждое ребро графа g с начальной вершиной $v \in V-\{s,t\}$ на новое ребро, имеющее $v^2$ в качестве начальной вершины.
Граф g и соответствующий ему граф $g^1$ представлены ниже.




Нетрудно доказать, что:

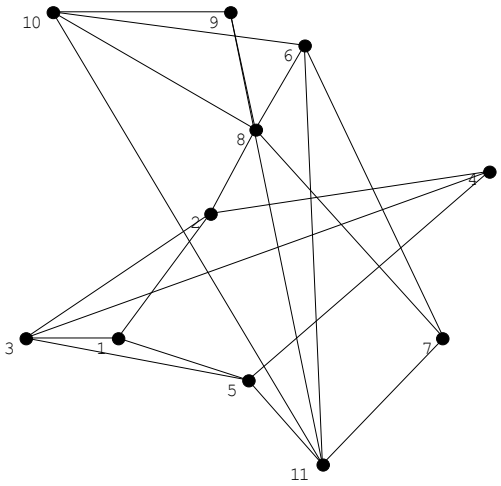
1. Каждый ориентированный s–t-путь в графе $g^1$ соответствует ориентированному s–t-пути в графе g, который получается стягиванием всех ребер вида $(v^1,v^2)$, и наоборот, каждый ориентированный s–t-путь в графе g соответствует s–t-пути в графе $g^1$, полученному расщеплением всех вершин пути, отличных от s и t.
2. Два ориентированных s–t-пути в графе $g^1$ не пересекаются по ребрам тогда и только тогда, когда соответствующие им пути в графе g не пересекаются по вершинам.
3. Максимальное число не пересекающихся по ребрам ориентированных s–t-путей в графе $g^1$ равно максимальному числу не пересекающихся по вершинам ориентированных s–t-путей в графе g.
4. Минимальное число ребер, удаление которых разрушает все ориентированные s–t-пути в графе $g^1$, равно минимальному числу вершин, удаление которых разрушает все ориентированные s–t-пути в графе g.

### *8.21. Задача “Диверсант”*

А. Как диверсанту определить минимальное количество мостов, которые надо уничтожить для того, чтобы разделить два пункта некоторого города.
Б. Может ли он пройти по мостам, по дороге сжигая их, и вернуться назад.

**Решение.**
А. Решается аналогично задаче “Маршруты модниц “: присвоить ребрам графа вес 1, только вызывать функцию NetworkFlow поиска максимального потока между вершинами s и t, следует с тегом Cut:
NetworkFlow[g, s, t, Cut] возвращает минимальный s-t разрез.
Пусть g есть изображенный ниже граф:

```
In[4]:= ShowLabeledGraph[g]
```

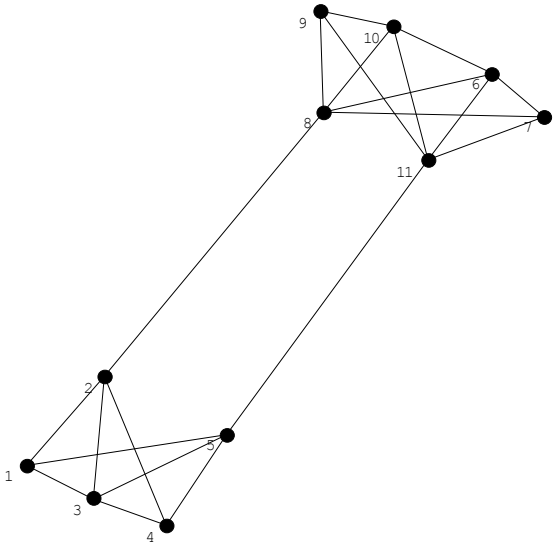



По умолчанию, веса ребер графа, трактуемые здесь как пропускные способности, принимаются равными 1.
Вычислим величину максимального потока от вершины 3 до вершины 11:

```
In[5]:= NetworkFlow[g, 3, 11]
Out[5]= 2
```

По теореме Форда-Фалкерсона эта величина есть мощность минимального 3-11 разреза. Визуально найти этот разрез довольно трудно. Призовем на помощь следующую функцию улучшения вложения:

```
In[6]:= ShowLabeledGraph[SpringEmbedding[g, 20]]
```

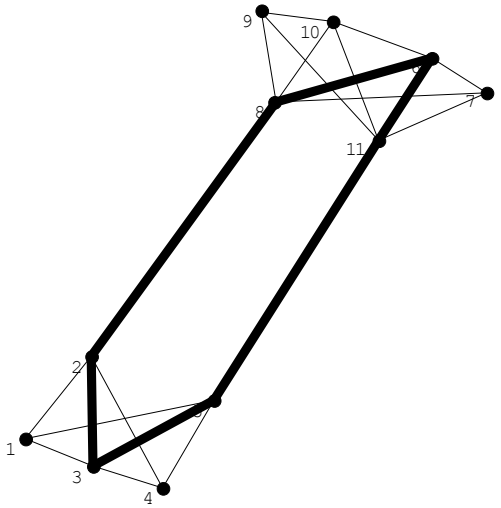



Требуемый разрез виден невооруженным глазом: {{2, 8},{5, 11}}
Для интереса посмотрим на ребра с положительной величиной потоком вдоль них:

```
In[7]:= NetworkFlow[g, 3, 11, Edge]
Out[7]= {{{2, 8}, 1}, {{3, 2}, 1}, {{3, 5}, 1}, {{5, 11}, 1}, {{6, 11}, 1}, {{8, 6}, 1}}
```

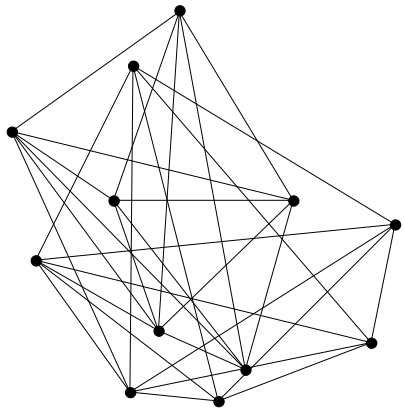
Выделим ребра этого потока:

```
In[8]:= flow = Map[First, NetworkFlow[g, 3, 11, Edge]]
Out[8]= {{2, 8}, {3, 2}, {3, 5}, {5, 11}, {6, 11}, {8, 6}}

In[9]:= ShowLabeledGraph  [Highlight [g, {flow }] ];
```




Б. Пусть дан граф t

```
In[10]:= ShowGraph[t]
```

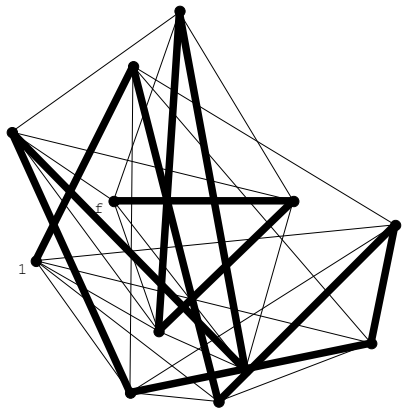


Построим список ребер графа, получаемом при поиске в глубину, начиная, допустим, с вершины 1:

```
In[11]:= tr = DepthFirstTraversal[t, 1, Edge]
Out[11]= {{1, 2}, {2, 3}, {3, 4}, {4, 5}, {5, 6},
          {6, 9}, {9, 7}, {7, 8}, {8, 10}, {10, 11}, {11, 12}}
```

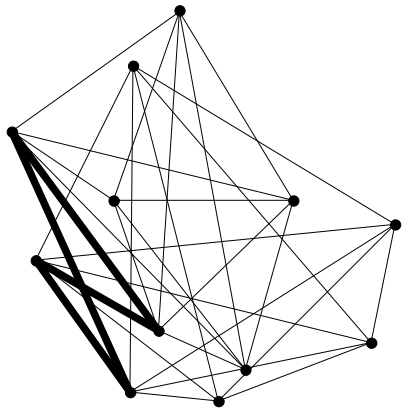
Выделим их:

```
In[12]:= dt = Highlight [t, {tr}];
In[13]:= dt1 = SetGraphOptions[dt,
           {1, 12, VertexLabel → {1, f}, VertexLabelPosition → LowerLeft }];

In[14]:= ShowGraph[dt1];
```



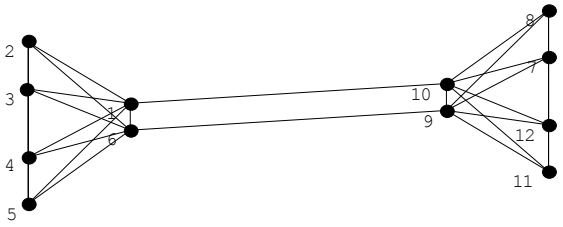

В данном графе мы видим, что дерево обхода есть простой путь, проходящий через все вершины. Попутно мы можем заключить, что граф связен и гамильтонов. Далее каждое невыделенное ребро {s, t} образует вместе с s-t путем дерева базисный цикл. Все остальные циклы получаются сложением по модулю 2 различных подмножеств множества базисных циклов. Аналогично, каждое выделенное ребро {s, t} вместе с ребрами, соединяющими предков вершины s в дереве обхода с потомками вершины t, образует базисное разрезающее множество. Здесь невыделенному ребру {1, 10} соответствует базисный цикл {1, 2, 3, 4, 5, 6, 9, 7, 8, 10, 1}, ребру {1, 6} – {1, 2, 3, 4, 5, 6}, ребру {9, 10} – {9, 7, 8, 10}. Суммируя по модулю 2 ребра этих циклов, получаем цикл ct = {1, 6, 9, 10, 1}, который является разрезом.

```
In[15]:= ShowGraph [Highlight [t, {ct}]];
```



Улучшим вложение функцией SpringEmbedding:

```
In[16]:= t = SpringEmbedding[t, 1];
In[17]:= ShowLabeledGraph[t];
```



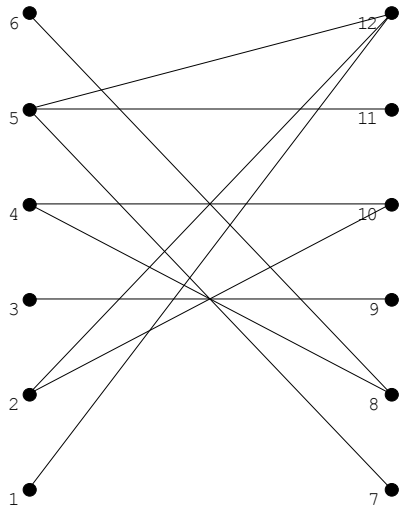

Легко видеть, что цикл {1, 6, 9, 10, 1} является разрезом.

### *8.22. Задача "Дефицит"*

Дан двудольный граф G = (X, Y, E), S – произвольное подмножество X, L(S) – множество вершин, смежных с вершинами S. Определим дефицит S как σ(S) = $\left|S\right|$ – $\left|L(S)\right|$ и дефицит графа σ(G) как максимум дефицитов всех подмножеств множества X. Требуется вычислить σ(G).

**Решение.** Число всех подмножеств множества X равно $2^{|X|}$. Можно ли обойтись без перебора подмножеств? Да, можно. По теореме Кенига число ребер максимального паросочетания двудольного графа G равно $\left|X\right|$ – σ(G). Найти паросочетание двудольного графа можно за полиномиальное время от количества вершин графа. Например, вычислим дефицит следующего графа

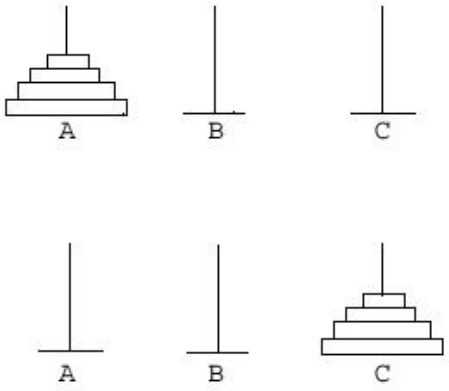
```
In[3]:= ShowLabeledGraph[g]
```




```
In[4]:= mtch = BipartiteMatching[g]
Out[4]= {{1, 12}, {2, 10}, {3, 9}, {4, 8}, {5, 7}}
```

Количество ребер паросочетания Length[mtch] равно 5. Следовательно, дефицит данного графа равен 1.

### 8.23 Задача “Ханойские башни и ковер Серпинского”




На трех стержнях A,B,C расположены различные диски. На каждом стержне диски расположены в убывающем порядке, как на рисунках. За один ход разрешается переместить меньший диск на больший. Требуется
а) перенести башню со стержня A на стержень C;
б) за минимальное количество ходов из любой одной конфигурации перейти к любой другой.

**Решение**.
а) На рис. 1 представлена классическая постановка задачи о ханойских башнях, решаемая простой рекурсией. Пусть F(X,Y,n-1) – порядок ходов для перемещения башни высотой n-1 со стержня X на стерженьY. Тогда, очевидно, F(A,C,n) состоит в выполнении следующих функций: F(A,B,n-1); F(A,C,1); F(B,C,n-1).
Достаточно определить F(X,Y,1)={X,Y}, где {X,Y} есть перемещение диска с X на Y. Например, построим рекурсивную функцию, вычисляющую только количество ходов и вычислим ее значение для n=4:

```
In[2]:= F[1] = 1; F[n_] := 2 F[n - 1] + 1; F[4]

Out[2]= 15
```

Все просто, но нет ли “свернутой”, замкнутой формулы вычисления функции F?
Если положить $T_n = F[n] + 1$, то получим

$T_1 = 1;$
$T_n = 2 \cdot T_{n-1}.$

Отсюда сразу получается, что решение этой рекурсии есть просто Tn = 2n; следовательно, F[n] = 2n – 1. Проверим:

```
In[3]:= F[10]

Out[3]= 1023
```

и, наконец, как обманули индийского раджу в известной притче:

```
In[4]:= F[64]

Out[4]= 18446744073709551615
```

б) в этом случае найти рекурсию гораздо сложнее. Представим задачу на языке теории графов. Каждую возможную конфигурацию дисков представим разбиением множества {1, 2, … ,n ) на три блока, например: {{1,2,3,4},{},{}}; {{2,3,4},{1},{}}: {{3,4},{1},{2}} и сопоставим различным разбиениям вершины неориентированного графа, в котором две вершины V и W смежны, если из соответствующей вершине V конфигурации существует ход, приводящий к конфигурации, соответствующей вершине W. Тогда задача состоит в нахождении кратчайшего пути на графе из вершины V до вершины W. Для примера, V – {}, W – {}. Легко построить рекурсивную функцию, возвращающую следующий список всех требуемых разбиений множества Range[4]:

```
v = {{{}, {}, {1, 2, 3, 4}}, {{}, {4}, {1, 2, 3}}, {{}, {3, 4}, {1, 2}}, {{}, {3}, {1, 2, 4}},
 {{}, {2, 3}, {1, 4}}, {{}, {2, 3, 4}, {1}}, {{}, {2, 4}, {1, 3}}, {{}, {2}, {1, 3, 4}},
 {{}, {1, 2}, {3, 4}}, {{}, {1, 2, 4}, {3}}, {{}, {1, 2, 3, 4}, {}}, {{}, {1, 2, 3}, {4}},
 {{}, {1, 3}, {2, 4}}, {{}, {1, 3, 4}, {2}}, {{}, {1, 4}, {2, 3}}, {{}, {1}, {2, 3, 4}},
 {{4}, {}, {1, 2, 3}}, {{4}, {3}, {1, 2}}, {{4}, {2, 3}, {1}}, {{4}, {2}, {1, 3}},
 {{4}, {1, 2}, {3}}, {{4}, {1, 2, 3}, {}}, {{4}, {1, 3}, {2}}, {{4}, {1}, {2, 3}},
 {{3, 4}, {}, {1, 2}}, {{3, 4}, {2}, {1}}, {{3, 4}, {1, 2}, {}}, {{3, 4}, {1}, {2}},
 {{3}, {}, {1, 2, 4}}, {{3}, {4}, {1, 2}}, {{3}, {2, 4}, {1}}, {{3}, {2}, {1, 4}},
 {{3}, {1, 2}, {4}}, {{3}, {1, 2, 4}, {}}, {{3}, {1, 4}, {2}}, {{3}, {1}, {2, 4}},
 {{2, 3}, {}, {1, 4}}, {{2, 3}, {4}, {1}}, {{2, 3}, {1, 4}, {}}, {{2, 3}, {1}, {4}},
 {{2, 3, 4}, {}, {1}}, {{2, 3, 4}, {1}, {}}, {{2, 4}, {}, {1, 3}}, {{2, 4}, {3}, {1}},
 {{2, 4}, {1, 3}, {}}, {{2, 4}, {1}, {3}}, {{2}, {}, {1, 3, 4}}, {{2}, {4}, {1, 3}},
 {{2}, {3, 4}, {1}}, {{2}, {3}, {1, 4}}, {{2}, {1, 3}, {4}}, {{2}, {1, 3, 4}, {}},
 {{2}, {1, 4}, {3}}, {{2}, {1}, {3, 4}}, {{1, 2}, {}, {3, 4}}, {{1, 2}, {4}, {3}},
 {{1, 2}, {3, 4}, {}}, {{1, 2}, {3}, {4}}, {{1, 2, 4}, {}, {3}}, {{1, 2, 4}, {3}, {}},
 {{1, 2, 3, 4}, {}, {}}, {{1, 2, 3}, {}, {4}}, {{1, 2, 3}, {4}, {}}, {{1, 3}, {}, {2, 4}},
 {{1, 3}, {4}, {2}}, {{1, 3}, {2, 4}, {}}, {{1, 3}, {2}, {4}}, {{1, 3, 4}, {}, {2}},
 {{1, 3, 4}, {2}, {}}, {{1, 4}, {}, {2, 3}}, {{1, 4}, {3}, {2}}, {{1, 4}, {2, 3}, {}},
 {{1, 4}, {2}, {3}}, {{1}, {}, {2, 3, 4}}, {{1}, {4}, {2, 3}}, {{1}, {3, 4}, {2}},
 {{1}, {3}, {2, 4}}, {{1}, {2, 3}, {4}}, {{1}, {2, 3, 4}, {}}, {{1}, {2, 4}, {3}},
 {{1}, {2}, {3, 4}}}
```

Из структуры элементов вышеприведенного списка видна следующая рекурсия: на первом месте разбиения находятся все подмножества множества {1, 2, 3, 4}, далее для каждого подмножества следуют разбиения его дополнения на два блока. На множестве v строим бинарное отношение P:

```
In[5]:= P = (p = #1; q = #2; d1 = Length[p[[1]]] - Length[q[[1]]]; d2 = Length[p[[2]]] - Length[q[[2]]];
          d3 = Length[p[[3]]] - Length[q[[3]]]; i1 = -1; i2 = -1; If[d1 == 1, i1 = 1]; If[d2 == 1, i1 = 2];
          If[d3 == 1, i1 = 3]; If[d1 == -1, i2 = 1]; If[d2 == -1, i2 = 2]; If[d3 == -1, i2 = 3];
          If[(i1 > 0) && (i2 > 0), r = First[p[[i1]]]; p[[i1]] = Rest[p[[i1]]]; p[[i2]] = Prepend[p[[i2]], r]]; p == q) &;
```

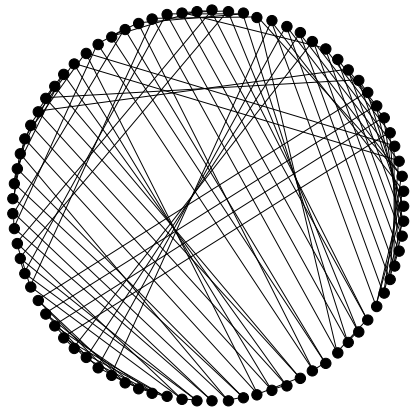
Создаем граф:

```
In[7]:= g = MakeGraph [v, P, Type -> Undirected ];
```

Так как мы реализовали в функции P рефлексивное отношение, уберем петли:

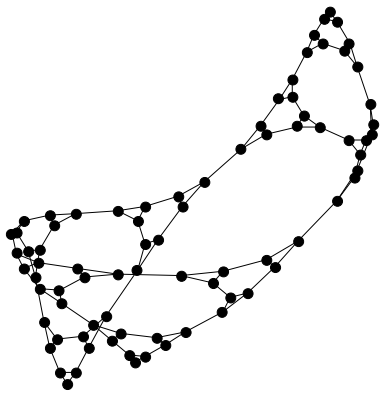
```
In[9]:= g1 = MakeSimple[g];
```

и изобразим получившийся граф:



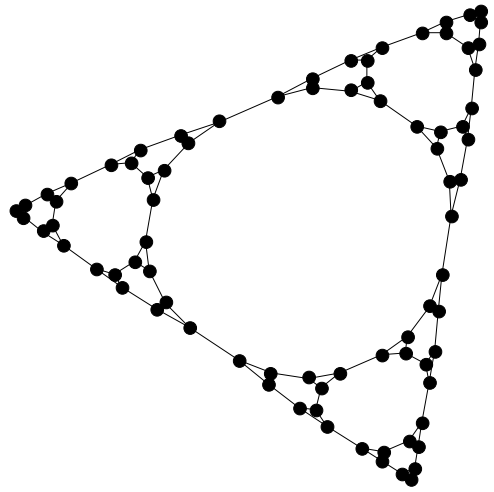
Попробуем улучшить вложение:

```
In[10]:= ShowGraph[SpringEmbedding[g1, 100]];
```



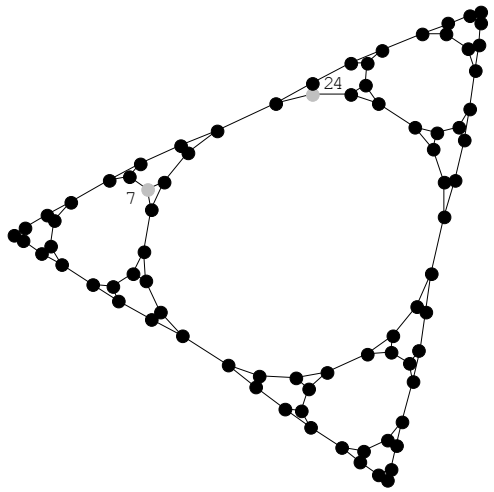
Уже лучше, но еще раз:

```
In[11]:= ShowGraph[SpringEmbedding[g2, 100]];
```



Скрытое стало явным.

Отметим две вершины, допустим 7 и 24 :




Конфигурации {{}, {2, 4}, {1, 3}} отвечает вершина 7 в списке v:

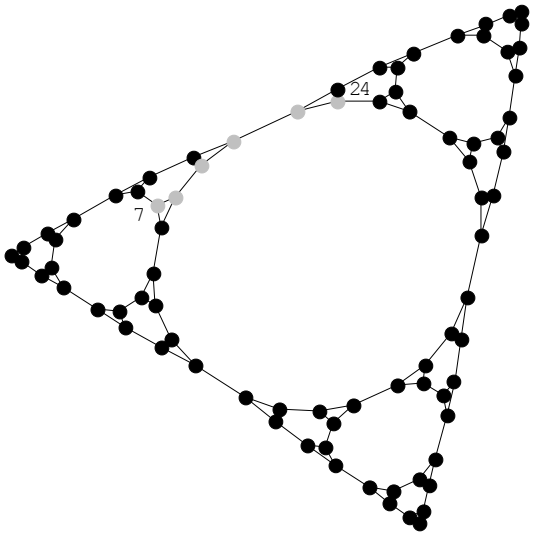
```
In[13]:= v[[7]]
Out[13]= {{}, {2, 4}, {1, 3}}
```

А конфигурации {{4}, {1}, {2, 3}} – вершина 24:

```
In[14]:= v[[24]]
Out[14]= {{4}, {1}, {2, 3}}
```

Выделим кратчайший путь:

```
In[15]:= ShowGraph[Highlight[g, ShortestPath[g, 7, 24], HighlightedVertexStyle → Disk[Normal],
           HighlightedVertexColors → {Gray}]];
```




Выведем список вершин кратчайшего пути:

```
In[16]:= ShortestPath[g, 7, 24]
Out[16]= {7, 80, 75, 2, 17, 24}
```

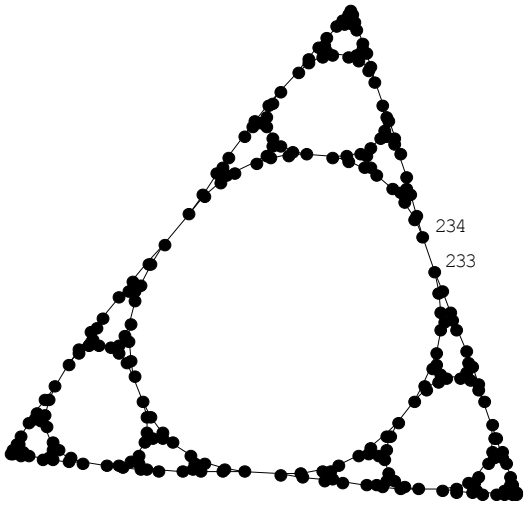
Протестируем на нашем графе задачу а):
Конфигурации слева соответствует вершина 61, справа – вершина 1.

```
In[17]:= Length[ShortestPath[g, 61, 1]] - 1
Out[17]= 15
```

здесь мы отняли 1, т. к. количество ребер в пути на 1 меньше количества вершин.
На рисунке ясно видна циклическая структура графа.
Для интереса найдем в нем максимальное число реберно-непересикающихся циклов:

```
In[18]:= ExtractCycles[g]
Out[18]= {{58, 50, 51, 58}, {57, 49, 52, 57}, {56, 48, 53, 56}, {55, 47, 54, 55}, {60, 44, 45, 60},
          {59, 43, 46, 59}, {61, 41, 42, 61}, {63, 38, 39, 63}, {65, 30, 35, 65}, {66, 31, 34, 66},
          {64, 29, 36, 64}, {68, 25, 28, 68}, {69, 26, 27, 69}, {71, 18, 23, 71}, {72, 19, 22, 72},
          {70, 17, 24, 70}, {73, 20, 21, 73}, {77, 4, 13, 77}, {78, 5, 12, 78}, {76, 3, 14, 76}, {79, 6, 11, 79},
          {75, 2, 15, 75}, {80, 7, 10, 80}, {40, 62, 37, 40}, {67, 32, 33, 67}, {74, 1, 16, 74}, {81, 8, 9, 81}}
```

На рисунке хорошо видны эти 27  длины 3.
Такую же структуру циклов имеет граф ходов для произвольного количества дисков. Ниже изображен граф для пяти дисков:

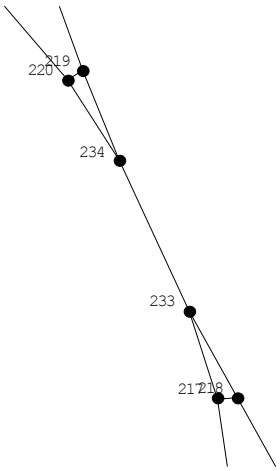



Снова вызовем функцию:

```
In[20]:= Length[ExtractCycles[g]]
Out[20]= 81
```

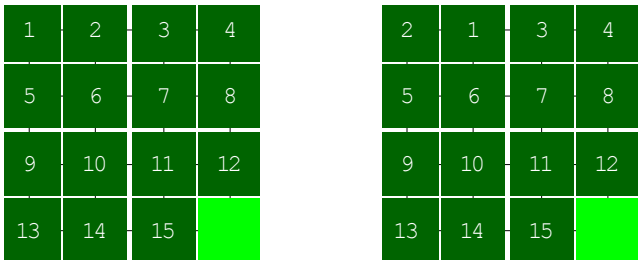
Количество реберно-непересекающихся циклов увеличилось в три раза. Это подводит нас к тому, что граф состоит из трех подграфов, изоморфных предыдущему графу, каждый из которых, в свою очередь, устроен также. С геометрической точки зрения эти графы являются фракталами. В конечном счете, граф состоит из $3^n$ непересекающихся циклов длины 3, покрывающих все вершины графа. На предыдущем рисунке вершина 234 кажется не входящей в цикл длины 3. Рассмотрим окрестность двух вершин 233 и 234:

```
In[22]:= ShowLabeledGraph[g, VertexLabel → True, VertexNumberPosition → UpperLeft, PlotRange → Zoom[{233, 234}]];
```




В действительности, вершины 233, 234 входят в состав циклов длины 3.
Как фрактальные m-угольники с глубиной рекурсии устроены графы m ханойских башен с n дисками. И также сплетены из простых треугольников.

### *8.24. Задача “Игра в пятнадцать”*




В 30-х годах в Америке была очень популярной головоломка, за решение которой полагался солидный приз. Требуется, передвигая фишки на свободное место, из начального расположения фишек перейти к конечному расположению. Оказывается, что задача разрешима тогда, и только когда перестановки, соответствующие начальному и конечному расположению фишек имеют одинаковую четность. Двум расположениям фишек, изображенным на рисунке, соответствуют две перестановки 16-ти чисел:

{2, 1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}, и единичная перестановка
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}.

Вычислим четность первой перестановки:

```
In[2]:= Signature[{2, 1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}]

Out[2]= -1
```

четность единичной перестановки равна 1. Следовательно, задача неразрешима.

### *8.25. Задача “Два туриста”*

Один турист должен дойти из пункта A до пункта B в стране X, его друг – из пункта C до пункта D в соседней стране Y. При этом во время передвижения они должны находиться друг от друга на дистанции радиосвязи.
Какие маршруты они должны выбрать, чтобы совершить совместно наименьшее количество переходов?

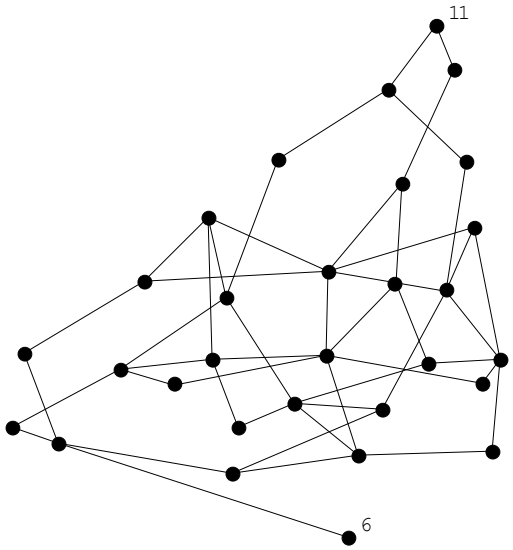
**Решение.** Функциями FromOrderedPairs, AddEdges строим графы g1, g2, представляющие две страны, соответственно. Строим произведение GraphProduct[g1, g2]. Далее, на множестве V= {(x,y); x∈X, y∈Y} вершин этого графа, удовлетворяющих условию d(x,y)≤l, функцией InduceSubgraph [g, V] строим вершинно-порожденный подграф g.
Осталось при помощи функции ConnectedComponents[g] выделить связные компоненты графа g, проверить принадлежность вершин v=(A, B) и w= (C, D) одной компоненте, и вызвать функцию ShortestPath[g,v,w] для нахождения кратчайшего пути между вершинами v и w.

### 8.26. Задача ”Погоня”

Некоторый отряд преследует беглеца. Отряд и беглец передвигаются по имеющейся у них карте дорог, соединяющих пункты ночевок. За один световой день беглец преодолевает один переход, в то время как преследующий его отряд – два, причем оставляя в пройденных пунктах ловушки. Может ли беглец дойти до пункта C, если он выходит из пункта A, и одновременно с ним из пункта B – преследователи?

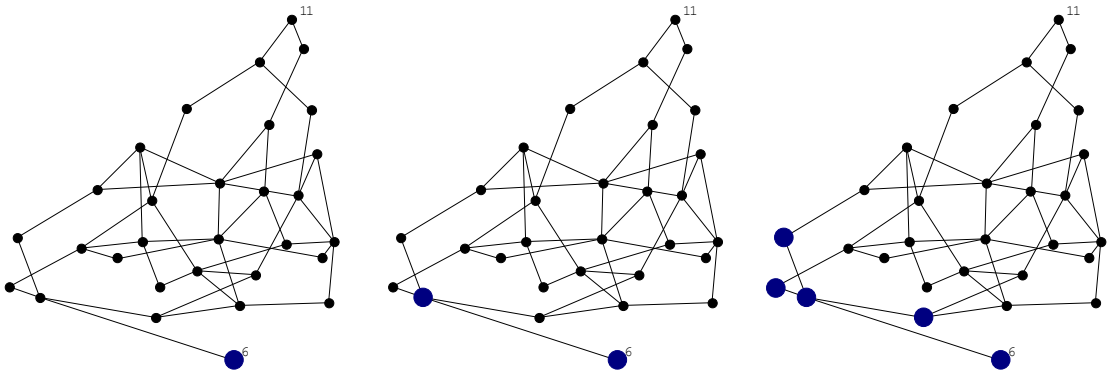
**Решение.** Нужно в цикле по переменной Level, обозначающей количество дней, применить функцию поиска в ширину (волновой алгоритм) BreadthFirstTraversal[g, v, Level] для формирования множеств V1[level], V2[2*Level] вершин графа g(V,E), проходимых волнами, исходящими из вершин графа, соответствующих пунктам А и Б карты; проверить принадлежит ли вершина С множествам V1,V2; проверить связность подграфа, порожденного множеством вершин V1-V2 и V-V2.
Множества V1 и V2 проще построить функцией Neighborhood[g, v, level], которая возвращает подмножество вершин графа g, находящихся на расстоянии, меньшем или равным level от вершины v. Пусть дан граф g:

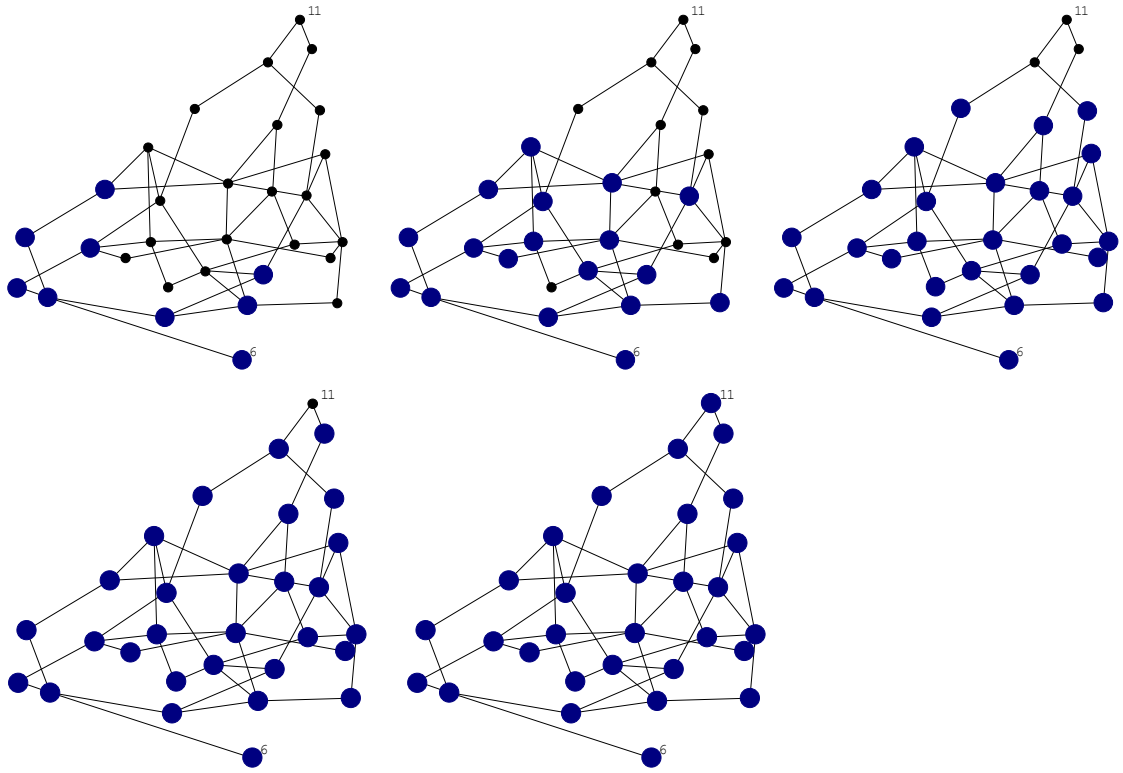
```
In[2]:= ShowGraph[g];
```




Проследим, например, распространение волны из вершины 6.
Последовательно вызывая функцию Neighborhood с увеличивающимся параметром level, подсветим затопленные волной вершины:

```
In[3]:= Do[ShowGraph [Highlight [g, Neighborhood [g, 6, i],
          HighlightedVertexStyle    -> Disk [0.051 ],
      HighlightedVertexColors    -> {Navy }]
       ]; , {i, 0, 7}];
```



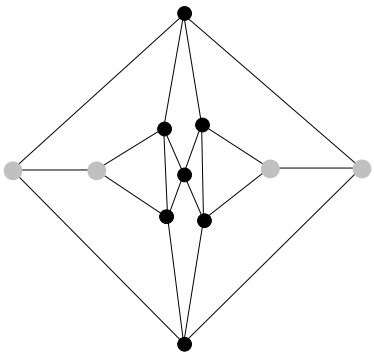


### *8.27. Задача “Железная дорога”*

Требуется разделить изображенную ниже сеть железных дорог между двумя компаниями так, чтобы в каждом пункте у каждой компании имелась своя дорога.

**Решение.**

```
In[2]:= ShowGraph[g]
```

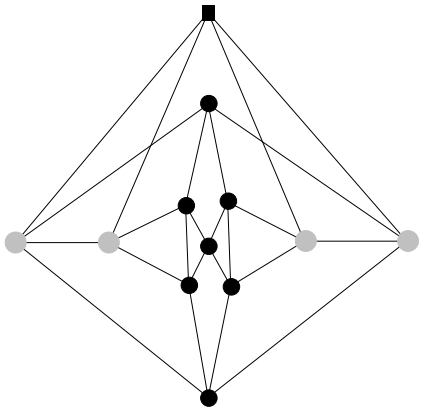


Выделим список вершин графа g с нечетной степенью:

```
In[3]:= t = Degrees[g]; o = {}; Do[If[Mod[t[[i]], 2] == 1, o = Append[o, i]], {i, 1, Length[t]}]; o
Out[3]= {3, 4, 10, 11}
```

Количество таких вершин в любом графе четно. Добавим к графу одну вершину 12 и соединим ее ребрами с вершинами полученного списка

```
In[4]:= gt = AddVertex[g, {0, 0.4}]; gt = AddEdges[gt, {{12, 4}, {12, 3}, {12, 11}, {12, 10}}];

In[5]:= ShowGraph[gt];
```



Получившийся граф, очевидно, эйлеров:

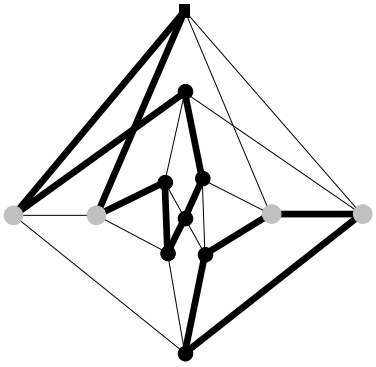
```
In[6]:= EulerianQ[gt]
Out[6]= True
```

Построим в нем эйлеров цикл:

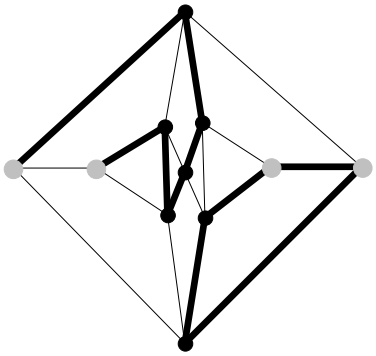
```
In[7]:= c = EulerianCycle[gt]
Out[7]= {5, 7, 2, 8, 6, 1, 3, 11, 6, 9, 5, 10, 7, 9, 8, 11, 12, 10, 4, 12, 3, 2, 4, 1, 5}
```

Легко видеть, что все вершины графа являются его внутренними вершинами. Построим список нечетных ребер этого цикла и выделим его одной компании, а четные ребра оставим другой компании:

```
In[8]:= odEd = {}; Do[odEd = Append[odEd, Take[c, {i, i + 1}]], {i, 1, Length[c] - 1, 2}]; odEd

In[9]:= ShowGraph [Highlight [gt, {odEd }] ];
```



```
In[10]:= ShowGraph[DeleteVertex[gt, 12]];
```



### *8.28. Задача “Восемь ферзей”*

Рассмотрим известную задачу о восьми ферзях, которую связывают с именем К.Гаусса. Требуется так расставить на шахматной доске наибольшее число ферзей, чтобы они не атаковали друг друга.

**Решение.** Требуемая расстановка ферзей на шахматной доске должна удовлетворять трем условиям:

1). никакие два ферзя не должны находиться на одной вертикали;
2). никакие два ферзя не должны находиться на одной горизонтали;
3). никакие два ферзя не должны находиться на одной диагонали.

Каждой перестановке восьмого порядка соответствует расстановка ферзей на шахматной доске, удовлетворяющая первым двум условиям. Необходимо расположить ферзей на шахматной доске таким образом, чтобы они не атаковали друг друга по диагоналям. Это означает, что никакие два ферзя на шахматной доске не должны находиться в вершинах равнобедренного прямоугольного треугольника, катеты которого горизонтальны или вертикальны. Для поиска соответствующих перестановок переберем все 8! перестановок восьмого порядка с помощью встроенной функции системы Mathematica - NextPermutation. Функция NextPermutation[p] возвращает перестановку, следующую за перестановкой p в лексикографическом порядке. Система Mathematica. выполняет этот перебор за считанные секунды. Обозначим множество всех перестановок, удовлетворяющее условию 3 через sp.

```
In[2]:= per = {1, 2, 3, 4, 5, 6, 7, 8}; num = 1; sp = {};
        Do[y := 0; per = NextPermutation[per];
        Do[Do[If[Abs[i – j] == Abs[per[[i]] – per[[j]]], y := 1, x = 1], {j, i + 1, 8}], {i, 1, 7}];
        If[y == 0, Print[num, "-", per]; sp = Append[sp, per]; num = num + 1;, x = 1];, {8 ! }];
```

Получили 92 перестановки, удовлетворяющие третьему условию. Заметим, что некоторые из соответствующих этим перестановкам расстановок ферзей можно получить друг из друга, например, при повороте шахматной доски на 180°.

Определение. Две перестановки из списка sp назовем изоморфными, если они принадлежат одной орбите при действии группы самосовмещений квадрата.

Известно, что группа самосовмещений квадрата состоит из:
-поворотов вокруг центра на углы 0°,90°, 180°,270°;
-симметрии относительно вертикальной оси, проходящей через центр квадрата;
-симметрии относительно горизонтальной оси, проходящей через центр квадрата;
-симметрий относительно двух диагоналей квадрата.

Выделим вначале те пары перестановок, которые получаются друг из друга при симметрии относительно вертикальной и горизонтальной осей симметрии. В списке sp им соответствуют перестановки с обратным порядком расположения элементов.

Аналогично выделим в списке sp перестановки, получающиеся друг из друга при симметрии относительно диагонали квадрата, соединяющую нижнюю левую и верхнюю правую вершины. В списке sp таким парам соответствуют прямая и обратная перестановки. Чтобы выделить такие пары воспользуемся встроенной функцией InversePermutation[p], которая возвращает перестановку, обратную перестановке p.
Далее выделим все перестановки, соответствующие первым элементам пар перестановок, симметричных относительно второй диагонали.
Рассмотрим теперь все перестановки в списке sp, получающиеся друг из друга при повороте относительно центра квадрата по часовой стрелке на 90°: В списке sp им соответствуют обратные перестановки, перечисленные в обратном порядке.
Осталось рассмотреть только пары перестановок, получающиеся друг из друга при повороте на 180°. В списке sp4 таким парам соответствуют такие пары перестановок, что второй элемент пары является обратной перестановкой первого элемента пары, перечисленного в обратном порядке.
Введем бинарное отношение rel:

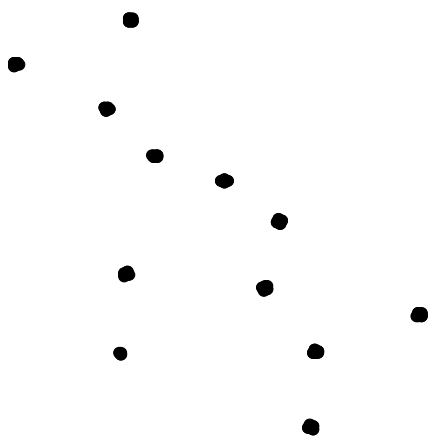
```
In[3]:= rel = (#1 == Reverse[#2] || #1 == InversePermutation[#2] ||
          #1 == Reverse[InversePermutation[Reverse[#2]]] || #1 == Reverse[InversePermutation[#2]] ||
          #1 == InversePermutation[Reverse[#2]]) &;
```

Построим граф g, вершины которого – элементы множества sp, две вершины графа g соединены ребром, если они связаны отношением rel. Количество связных компонет этого графа дает нам число неизоморфных перестановок в sp.

```
In[4]:= g = MakeGraph[sp, rel, Type → Undirected]
Out[4]= -Graph:<226, 92, Undirected>-
```

Изобразим граф g:

```
In[5]:= ShowGraph[SpringEmbedding[g, 100]]
```



Выделим связные компоненты графа g:

```
In[6]:= s = ConnectedComponents[g] // ColumnForm
```

```
Out[6]= {1, 4, 22, 33, 60, 71, 89, 92}
        {2, 3, 15, 42, 51, 78, 90, 91}
        {5, 28, 36, 41, 52, 57, 65, 88}
        {6, 19, 29, 39, 54, 64, 74, 87}
        {7, 20, 23, 45, 48, 70, 73, 86}
        {8, 13, 16, 37, 56, 77, 80, 85}
        {9, 27, 31, 46, 47, 62, 66, 84}
        {10, 12, 30, 43, 50, 63, 81, 83}
        {11, 21, 32, 44, 49, 61, 72, 82}
        {14, 38, 55, 79}
        {17, 25, 26, 35, 58, 67, 68, 76}
        {18, 24, 34, 40, 53, 59, 69, 75}

In[7]:= Length[s]

Out[7]= 12
```

Итак, существует ровно 12 различных способов расстановок ферзей.
Рассмотрим, например, один из таких двенадцати наборов:

```
In[8]:= l = Table[s[[i]][[1]], {i, Length[s]}]

Out[8]= {1, 2, 5, 6, 7, 8, 9, 10, 11, 14, 17, 18}
```

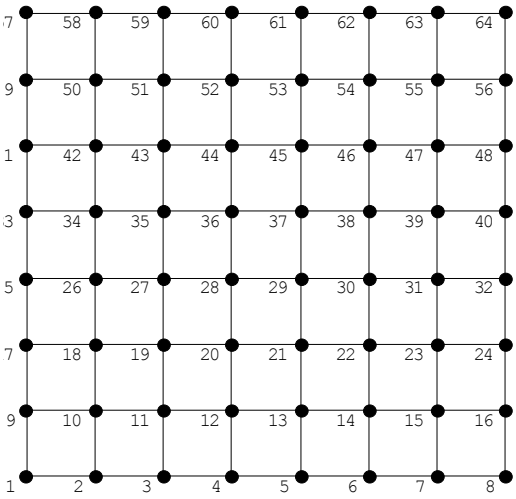
и соответствующие им перестановки списка sp:

```
In[9]:= Map[sp[[#]] &, l]
Out[9]= {{1, 5, 8, 6, 3, 7, 2, 4}, {1, 6, 8, 3, 7, 4, 2, 5}, {2, 4, 6, 8, 3, 1, 7, 5},
         {2, 5, 7, 1, 3, 8, 6, 4}, {2, 5, 7, 4, 1, 8, 6, 3}, {2, 6, 1, 7, 4, 8, 3, 5},
         {2, 6, 8, 3, 1, 4, 7, 5}, {2, 7, 3, 6, 8, 5, 1, 4}, {2, 7, 5, 8, 1, 4, 6, 3},
         {3, 5, 2, 8, 1, 7, 4, 6}, {3, 5, 8, 4, 1, 7, 2, 6}, {3, 6, 2, 5, 8, 1, 7, 4}}
```

Рассмотрим еще одно решение этой задачи.

Построим граф, состоящий из 64 вершин, соответствующих клеткам шахматной доски, которые соединены вертикальными и горизонтальными ребрами. Для построения графа, вершины которого соответствуют клеткам шахматной доски, воспользуемся встроенной функцией GridGraph[m,n], которая строит решетчатый граф размера m×n.

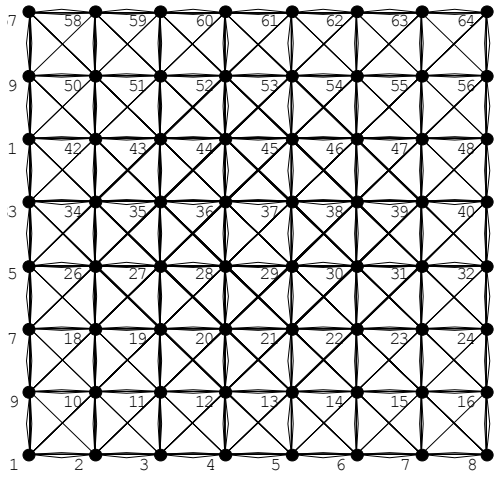
```
In[10]:= Chess = GridGraph[8, 8];

In[11]:= ShowLabeledGraph[Chess]
```




Соединим вершины диагональными ребрами с помощью встроенной функции AddEdges[g,l], которая прибавляет к графу g ребра из списка l. Сформируем список ребер, соединяющих вершины клеток шахматной доски по диагоналям. Обозначим полученный граф, состоящий из 64 вершин и 840 ребер, снова через Chess.

```
In[12]:= l = {};
         Do [
          Do [If [( Quotient [j - 1, 8] - Quotient [i - 1, 8]) == Abs [Mod [j - 1, 8] - Mod [i - 1, 8]]  ||
             Quotient [j - 1, 8] - Quotient [i - 1, 8] == 0  ||
         Mod [j - 1, 8] - Mod [i - 1, 8] == 0, l = Append [l, {i, j}], y = 0], {j, i + 1, 64}], {i, 1, 64}]
         Chess = AddEdges [Chess , l]
```

```
In[14]:= ShowLabeledGraph  [Chess ]
```




Ясно, что требуемой в задаче расстановке ферзей соответствует максимальное независимое множество в графе Chess.

Каждой расстановке ферзей соответствует максимальное независимое множество вершин графа Chess. Вызывая функцию MaximumIndependentSet[Chess], получим одно из решений задачи о восьми ферзях.

```
In[15]:= MaximumIndependentSet[Chess]
Out[15]= {1, 13, 24, 30, 35, 47, 50, 60}
```

Заметим, однако, что в этом решении выдается лишь одно из возможных решений. Задача нахождения максимального независимого множества является NP-полной, так что неудивительно, что поиск решения здесь занимает достаточно большое время

```
In[16]:= Timing[MaximumIndependentSet[Chess];]
```

### *8.29. Задача "Три ферзя"*

Требуется расставить на шахматной доске шестого порядка три ферзя таким образом, чтобы все свободные клетки доски оказались под боем.

**Решение.**

Определим шесть строк Rw[i], шесть столбцов Cl[j] и матрицу Ds порядка шесть, которые соответствуют горизонталям, вертикалям и клеткам шахматной доски шестого порядка.

```
In[2]:= Ds = Flatten[Table[{i, j}, {i, 6}, {j, 6}], 1];
        Do[Rw[i] = Table[{i, k}, {k, 6}];
          Cl[j] = Table[{k, j}, {k, 6}], {j, 6}, {i, 6}];
```

Для каждой клетки (i,j) сформируем две диагонали Diag1[i,j] и Diag2[i,j], пересекающиеся в клетке (i,j). Построим две чистые функции FormDiag1[i_,j_] и FormDiag2[i_,j_]:

```
In[4]:= FormDiag1[i_, j_] :=
          Do[If[ i + k ≥ 1 && j + k ≥ 1 && i + k ≤ 6 && j + k ≤ 6,
            Diag1[i, j] = Append[Diag1[i, j], {i + k, j + k}]], {k, -5, 5}];
        FormDiag2[i_, j_] :=
          Do[If[ i - k ≥ 1 && j + k ≥ 1 && i - k ≤ 6 && j + k ≤ 6,
            Diag2[i, j] = Append[Diag2[i, j], {i - k, j + k}]], {k, -5, 5}];
```

Для каждой клетки (i,j) построим множество клеток Dn[i,j], состоящее из соответствующих i-ой строки Rw[i], j-го столбца Cl[j] и двух диагоналей Diag1[i,j], Diag2[i,j] , пересекающихся в клетке (i,j).Объединение этих множеств есть множество клеток, нахождение ферзей на которых позволяет им атаковать клетку(i,j).

```
In[6]:= Do[Diag1[i, j] = {}; Diag2[i, j] = {}; FormDiag1[i, j]; FormDiag2[i, j];
         Dn[i, j] = Union[Diag1[i, j], Diag2[i, j], Rw[i], Cl[j]], {i, 6},
         {j, 6}]
```

Возьмем первую тройку клеток из набора Ds :

```
In[7]:= s = {{1, 1}, {1, 2}, {1, 3}}; num = 1;
```

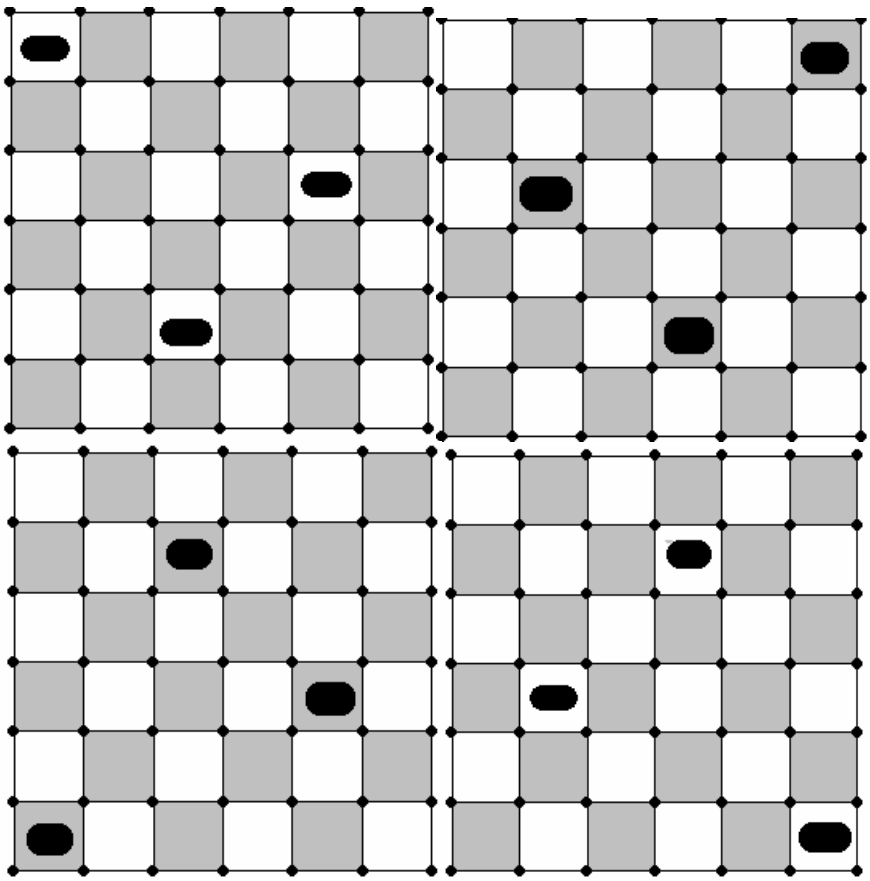
Для каждой тройки клеток проверим совпадение множества ее "опасных" клеток с множеством Ds. Для этого воспользуемся встроенной функцией NextKSubset[l,s], возвращающую k-подмножество списка l, следующее за k-подмножеством s в лексикографическом порядке. С помощью этой функции переберем все возможные тройки клеток:

```
In[8]:= Do[
         If[
          Complement[Ds, Union[Dn[Part[Part[s, 1], 1], Part[Part[s, 1], 2]],
             Dn[Part[Part[s, 2], 1], Part[Part[s, 2], 2]],
             Dn[Part[Part[s, 3], 1], Part[Part[s, 3], 2]]]] == {},
          Print[num, "   -  ", s]; num = num + 1;, y = 0]; s = NextKSubset[Ds, s],
         {k, Binomial[36, 3]}]

        1   -  {{1, 1}, {3, 5}, {5, 3}}

        2   -  {{1, 6}, {3, 2}, {5, 4}}

        3   -  {{2, 3}, {4, 5}, {6, 1}}

        4   -  {{2, 4}, {4, 2}, {6, 6}}
```

Определение. Два набора клеток шахматной доски шестого порядка назовем изоморфными, если соответствующие им позиции ферзей на шахматной доске инвариантны относительно действия группы самосовмещений квадрата.

Группа самосовмещений квадрата состоит из поворотов относительно центра квадрата на углы 0°,90°,180°,270°, симметрии относительно вертикальной оси симметрии, проходящей через центр квадрата, симметрии относительно горизонтальной оси симметрии, проходящей через центр квадрата, симметрии относительно диагоналей квадрата. Таким образом, группа самосовмещений квадрата является собственной подгруппой симметрической группы S4 и называется диэдральной группой D4.

На нижнем рисунке показаны четыре полученных решения:



Видим, что полученные решения задачи о трех ферзях изоморфны: первое и четвертое множества соответствуют клеткам доски, симметричным относительно одной диагонали шахматной доски, а второе и третье соответствуют клеткам, симметричным относительно второй диагонали.
Остались два множества

{{1,1},{3,5},{5,3}}
{{1,6},{3,2},{5,4}}

Эти оставшиеся наборы получаются при повороте шахматной доски на девяносто градусов. Получается, что задача о трех ферзях имеет одно базовое решение.
Таким образом, существует единственный способ разместить три ферзя на шахматной доске шестого порядка так, чтобы все свободные клетки доски оказались под угрозой.
На первый взгляд кажется, что все три ферзя не должны находится на одной диагонали, однако приведенное решение показывает, что эта гипотеза ошибочна.

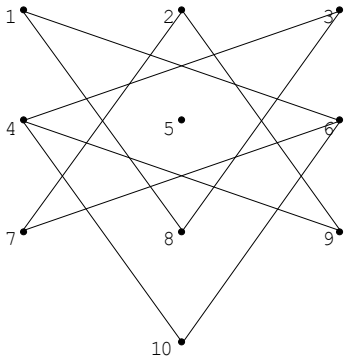
### *8.30. Задача " Телефонные номера"*

Сколько можно составить семизначных телефонных номеров, которые можно набрать ходом коня?

**Решение**. Построим граф с десятью вершинами, соответствующими цифровой клавиатуре телефонного аппарата, причем ребра этого графа соединяют вершины ходом коня на шахматной доске. Используем встроенную функцию FromAdjacencyLists[e,v], которая возвращает граф с ребрами из списка e и вершинами из списка v.

```
In[2]:= p := FromAdjacencyLists[{{6, 8}, {7, 9}, {4, 8}, {3, 9, 10}, {}, {1, 7, 10},
          {2, 6}, {1, 3}, {2, 4}, {4, 6}},
         {{2, 4}, {3, 4}, {4, 4}, {2, 3}, {3, 3}, {4, 3}, {2, 2}, {3, 2}, {4, 2}, {3, 1}}]
```

```
In[3]:= ShowGraph[p,VertexNumber ->True]
```




```
Out[3]= (- Graphics -)
```

Построим матрицу смежности этого графа:

```
In[4]:= (a = ToAdjacencyMatrix[p] ) // MatrixForm
```

Out[4]//MatrixForm=

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Элемент $a_{ij}$ есть количество маршрутов длины 1, соединяющих вершины i и j, а элементы К-ой степени матрицы равны количеству маршрутов длины К и соединяющих вершины i и j.
Возведем матрицу смежности в 7 - ю степень.

```
In[5]:= P = a; Do[P = P.a, {6}]; P // MatrixForm
```

Out[5]//MatrixForm=

$$\begin{pmatrix} 0 & 48 & 0 & 80 & 0 & 88 & 0 & 56 & 0 & 0 \\ 48 & 0 & 48 & 0 & 0 & 0 & 56 & 0 & 56 & 64 \\ 0 & 48 & 0 & 88 & 0 & 80 & 0 & 56 & 0 & 0 \\ 80 & 0 & 88 & 0 & 0 & 0 & 80 & 0 & 88 & 104 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 88 & 0 & 80 & 0 & 0 & 0 & 88 & 0 & 80 & 104 \\ 0 & 56 & 0 & 80 & 0 & 88 & 0 & 48 & 0 & 0 \\ 56 & 0 & 56 & 0 & 0 & 0 & 48 & 0 & 48 & 64 \\ 0 & 56 & 0 & 88 & 0 & 80 & 0 & 48 & 0 & 0 \\ 0 & 64 & 0 & 104 & 0 & 104 & 0 & 64 & 0 & 0 \end{pmatrix}$$

Просуммируем все элементы матрицы P:

```
In[6]:= P1 = Flatten[P]; nnum = 0; Do[nnum = nnum + P1[[i]], {i, 1, 100}]
```

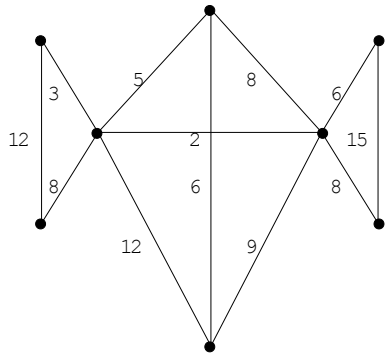
Число nnum – число требуемых телефонных номеров;

```
In[7]:= nnum
Out[7]= 2848
```

Применяя встроенную функцию NumberOfKPaths, эту задачу можно решить в одно действие:

```
In[8]:= l = Flatten[Table[NumberOfKPaths[p, i, 7], {i, 10}]]; Apply[Plus, l]
Out[8]= 2848
```

### *8.31. Задача “Уборка дорог”*



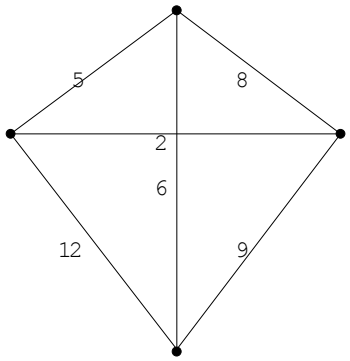

Эта задача известна как задача “о китайском почтальоне".
На рисунке изображена сеть дорог района некоторого города. Уборочная машина должна пройти по всем дорогам хотя бы один раз. Число на каждой стороне показывает время, которое должна потратить уборочная машина, чтобы проехать эту дорогу. Требуется найти необходимый путь для уборки дорог за наименьшее время.

**Решение.** Так как сеть дорог определяет некоторый граф, то определим в этом графе множество вершин с нечетной степенью. Количество таких вершин четно, так как сумма степеней вершин любого графа четна.

```
In[2]:= od = {}; d = Degrees[g]; Do[If[Mod[d[[i]], 2] == 1, od = Append[od, i]],
          {i, 1, Length[d]}];
        od
Out[2]= {1, 2, 3, 4}
```

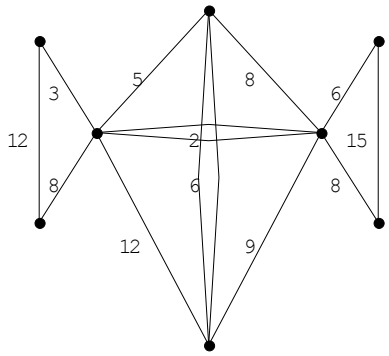
Определим на этом множестве вершинно-порожденный взвешенный граф, вес ребра (i, j) которого равен кратчайшему пути между вершинами i и j в исходном графе. В нашем случае эти ребра и их веса совпадают с соответствующими ребрами и их весами в исходном графе.

```
In[3]:= ShowGraph[sg = InduceSubgraph[g, od], PlotRange → 0.15];
```



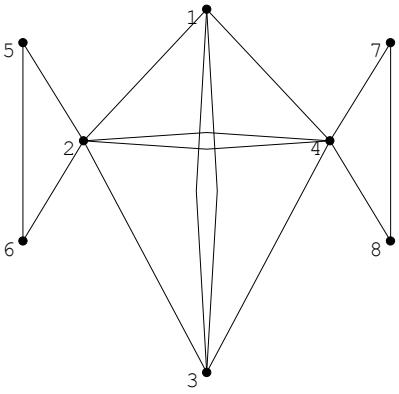

Строим на этом графе совершенное паросочетание минимального веса. Здесь его составляют ребра {1, 3} и {2, 4} с весами 6 и 2 соответственно. Добавим эти ребра к исходному графу:

```
In[4]:= gt = AddEdges[g, {{1, 3}, {2, 4}}];
```




Так как степени вершин этого графа четны, то граф имеет эйлеров цикл:

```
In[5]:= EulerianCycle[gt]

Out[5]= {5, 6, 2, 3, 4, 7, 8, 4, 1, 3, 1, 2, 4, 2, 5}
```




в котором ребра {1, 3} и {2, 4} исходного графа проходятся дважды.

### *8.32. Задача "Числа Фибоначчи"*

Требуется вычислить значение следующей рекурсивной функции:

```
In[2]:= F[0] = 1; Fib[1] = 1; Fib[n_] := F[n - 2] + Fib[n - 1];
```

**Решение.** Не торопитесь вычислять так заданную функцию. Здесь кроется маленькая ловушка с большими последствиями. Каждое слагаемое в определении этой функции порождает, в свою очередь вычисление двух ее значений, что ведет к лавинообразному росту вычислений. Для вычисления i-го числа достаточно хранить тройку (i, Fib[i-2], Fib[i-1]) и применить для вычисления итерационный процесс:

```
In[3]:= t = {2, 1, 1}; Do[s = t[[2]] + t[[3]]; t[[1]] = i; t[[2]] = t[[3]]; t[[3]] = s,
         {i, 3, n + 1}];
        t[[3]]
In[4]:= t = {2, 1, 1}; Do[s = t[[2]] + t[[3]]; t[[1]] = i; t[[2]] = t[[3]]; t[[3]] = s,
         {i, 3, n + 1}];
        t[[3]]
```

Оформим эти вычисления в виде функции от n (n>2) с локальными переменными s и t:

```
In[5]:= Fib[n_] := Module[{s, t}, t = {2, 1, 1};
          Do[s = t[[2]] + t[[3]]; t[[1]] = i; t[[2]] = t[[3]]; t[[3]] = s, {i, 3, n + 1}];
          t[[3]]]

In[6]:= Fib[20]

Out[6]= 10946
```

Для вычисления n-го числа Фибоначи можно также применить формулу Бине:

```
In[7]:= Fib[n] = [(√5 + 1)^n - (1 - √5)^n] / (2^n √5).
```

### *8.33. Задача "Прямоугольники"*

На плоскости даны n прямоугольников ($x_i$, $y_i$, $d_i$, $h_i$) со сторонами, параллельными осям координат. Здесь $x_i$, $y_i$ – координаты левого нижнего угла прямоугольника, $d_i$, $h_i$ – ширина и высота соответственно. Требуется найти площадь объединения данных прямоугольников.

**Решение.** Если воспользоваться формулой включения и исключения для объединения множеств, то потребуется перебор 2n вариантов, что неприемлемо по времени вычисления. Следует представить двойной интеграл по площади повторным интегралом по интервалам. Строится последовательность точек событий $x_{k1}$, $x_{k2}$, …, $x_{km}$, состоящая из строго возрастающей последовательности абсцисс вершин прямоугольников. В каждой точке x оси абсцисс проходящая через нее вертикальная прямая, называемая заметающей прямой, определяет ее статус - объединение отрезков пересечений прямой с прямоугольниками. В интервале между соседними точками событий статус заметающей прямой постоянен, а при переходе через точку события ее статус меняется: к нему добавляются или из него удаляются некоторые отрезки. Искомая площадь есть сумма площадей прямоугольников с основаниями $x_{k(i+1)} - x_{ki}$, и высотами, равными длине статуса, допустим, средней точки интервала ($x_{k(i+1)}$ , $x_{ki}$ ).

### *8.34. Задача "Слалом"*

```
   3
 1   3
5   2   1
2   6   4   7
```

Из каждой позиции треугольника, составленного из чисел, можно перейти на нижележащий ряд либо влево, либо вправо. Требуется найти такой путь от вершины до основания треугольника, что сумма чисел вдоль пути наибольшая (наименьшая).

**Решение**. Количество всех таких путей равно $2^n$, где n – высота треугольника. Эта задача удовлетворяет условию Беллмана, поэтому для ее решения можно применить метод динамического программирования, вычисляя рекурсивную функцию, или представить треугольник взвешенным ориентированным графом g, функцией SetEdgeWeigths присвоить ребрам соответствующие веса, для каждой вершины t нижнего ряда вызывать функцию ShortestPath [g, 1, t, Algorithm -> Dijkstra].
Пусть c(n, k) – число треугольника, стоящее k-м в n-м ряду.

```
In[2]:= c = {{3}, {1, 3}, {5, 2, 1}, {2, 6, 4, 7}}

Out[2]= {{3}, {1, 3}, {5, 2, 1}, {2, 6, 4, 7}}
```

Построим рекурсивную функцию F(n, k), вычисляющую цену искомого пути до этой позиции:

```
In[3]:= F[1, 1] = c[[1, 1]]; F[n_, 1] := c[[n, 1]] + F[n - 1, 1];
        F[n_, n_] := c[[n, n]] + F[n - 1, n - 1];
        F[n_, k_] := c[[n, k]] + Max[F[n - 1, k - 1], F[n - 1, k]];
```

Вычислим функцию для элементов 4-го ряда:

```
In[4]:= sums = {}; Do[sums = Append[sums, F[4, i]], {i, 4}]; sums
Out[4]= {11, 15, 12, 14}

In[5]:= Max[sums]
Out[5]= 15
```

Заменив в функции F max на min:

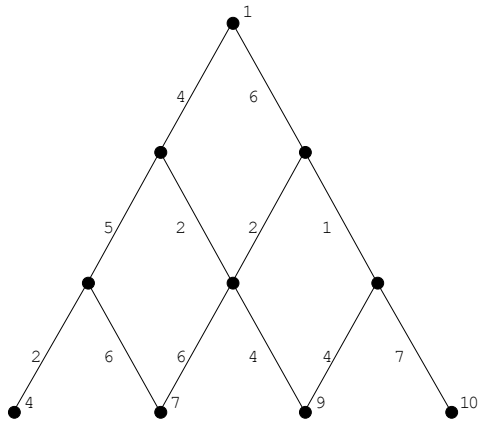
```
In[6]:= F[n_, k_] := c[[n, k]] + Min[F[n - 1, k - 1], F[n - 1, k]];
```

найдем минимальную цену:

```
In[7]:= Min[F[4, 1], F[4, 2], F[4, 3], F[4, 4]]
Out[7]= 10
```

Повторим этот результат на графе. Задаче соответствует следующий взвешенный граф:

```
In[8]:= ShowGraph[tr];
```




Здесь у середины ребра указан его вес. Вычислим цены кратчайших путей от вершины 1 до вершин нижнего ряда:

```
In[9]:= cs = (CostOfPath [tr, ShortestPath [tr, #1, #2, Algorithm -> Dijkstra ]]) &

In[9]:= cs = (CostOfPath [tr, ShortestPath [tr, #1, #2, Algorithm -> Dijkstra ]]) &
Out[9]= {11, 12, 10, 14}

In[10]:= Min[l]
Out[10]= 10
```

### *8.35. Задача "Рюкзак"*

Даны n коробок, объемы которых есть целые числа a1, …, an. Можно ли полностью заполнить какими то из этих коробок рюкзак, объем которого есть целое число b?

**Решение.** Это есть классическая NP-полная задача. Ее можно решить полным перебором подмножеств множества всех коробок. Для сокращения перебора применим метод динамического программирования. Рассмотрим вычисления подробно, шаг за шагом, не смотря на то, что для их получения можно применить рекурсию. Например, пусть v есть список чисел, представляющих объемы шести коробок:

```
In[2]:= v = {1, 3, 3, 3, 5, 5};
```

Далее, пусть vs1, vs2, …, vs6 – списки чисел, которые можно получить суммами элементов списков v1 = {1}, v2 = {1, 3}, vs3 = {1, 3, 3}, vs4 = {1, 3, 3, 3}, vs5 = {1, 3, 3, 3, 5}, vs6 = {1, 3, 3, 3, 5, 5} соответственно:

```
In[3]:= vs1 = {1}; vs2 = Union[vs1, {3}, vs1 + 3]
Out[3]= {1, 3, 4}

In[4]:= vs3 = Union[vs2, {3}, vs2 + 3]
Out[4]= {1, 3, 4, 6, 7}

In[5]:= vs4 = Union[vs3, {3}, vs3 + 3]
Out[5]= {1, 3, 4, 6, 7, 9, 10}

In[6]:= vs5 = Union[vs4, {5}, vs4 + 5]
Out[6]= {1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 15}

In[7]:= vs5 = Union[vs5, {5}, vs5 + 5]
Out[7]= {1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 19, 20}
```

Здесь приведена 6-и шаговая итерация, на каждом шаге которой объединяются предыдущий список сумм, очередное число из списка v и предыдущие полученные суммы, увеличенные на очередное число. Из окончательного списка видно, что можно полностью заполнить рюкзак любого объема, не большего 20, за исключением рюкзаков объемом 2 и 18.

### *8.36. Задача "Оптимальная упаковка рюкзака"*

Пусть задано множество типов коробок $y_i = (v_i, c_i)$, $1 \le i \le n$, где целые числа $v_i$, $c_i$, – объем коробки и его цена соответственно. Целое число V объем рюкзака. Требуется упаковать рюкзак так, чтобы закрывался и сумма стоимостей упакованных коробок была бы наибольшей.

**Решение**. Эта задача поставлена как задача линейного целочисленного программирования. Максимизировать линейный функционал:

$$P_n(x) = \sum x_i c_i \to \max, \quad \sum x_i v_i \le V, \quad \text{где xi – неотрицательные целые числа.}$$

Решим задачу методом линейного программирования. Для целых чисел k, v через C(k, v) обозначим стоимость оптимальной упаковки рюкзака объемом v коробками типов $y_1$, …, $y_k$. Функцию C(k, v) можно определить рекурсивно:

$C(0, v) = C(k, 0) = 0$. Далее, $C(k, v) = \max\{x_k c_k + C(k - 1, v - x_k v_k); 0 \le x_k \le v/v_k)\}$

```
In[2]:= f[0, v_] = 0; f[k_, 0] = 0;
        f[k_, v_] := Module[{sp, t, i}, sp = {};
          Do[t = i * y[[k, 2]] + f[k - 1, v - i * y[[k, 1]]]; sp = Append[sp, t],
           {i, 0, Floor[v / y[[k, 1]]]}]; Max[sp]]
```

Пусть даны три типа коробок:

```
In[3]:= y = {{2, 3}, {3, 6}, {5, 7}}
Out[3]= {{2, 3}, {3, 6}, {5, 7}}
```

Вычислим цены упаковок рюкзака объема 11 коробками:

(2, 3);
(2, 3), (3, 6);
(2, 3), (3, 6), (5, 7)

соответственно:

```
In[4]:= {f[1, 11], f[2, 11], f[3, 11]}
Out[4]= {15, 21, 21}
```

### *8.37. Задача "Редактирование"*

Даны две строки в некотором алфавите. Разрешены операции редактирования – вставка (I), удаление (D), замена (R) символа и и пустая операция (NOP). Редакционное расстояние между двумя строками есть минимальное количество операций редактирования, необходимых для преобразования первой строки во вторую. Требуется вычислить редакционное расстояние.

**Решение.** Решаем задачу методом динамического программирования. Построим две строки:

```
In[2]:= s1 = "jewel"; s2 = "gold";
```

Определим функцию Ed[i, j], возвращающую редакционное расстояние от строки, составленной из i первых символов строки s1, до строки, составленной из j первых символов строки s2. Очевидно, что

```
In[3]:= Ed[i_, 0] = i; Ed[0, j_] = j;
```

Для остальных значений аргументов определим функцию рекурсивно:

```
In[4]:= Ed[i_, j_] := Module[{t}, t = If[StringTake[s1, {i}] == StringTake[s2, {j}], 0, 1];
          Min[{Ed[i - 1, j] + 1, Ed[i, j - 1] + 1, Ed[i - 1, j - 1] + t}]];
```

Первому элементу списка функции Min[{ , , }] соответствует операция удаления i-го символа, после чего за Ed[i-1, j] операций редактирования строка s1[1, …,i-1] преобразуется в строку s2[1, …,j]; второму элементу – вставка символа в конец строки s2[1, …,j-1], в свою очередь полученной из s1[1, …,i] за Ed[i, j-1] операций; третьему элементу – операция замены i-го символа строки s1[1, …,i] j-м символом строки s2 в случае их несовпадения, и пустая операция иначе.

```
In[5]:= Ed[5, 4]
Out[5]= 5
```

то есть строка “jewel” преобразуется в строку “gold” пятью операциями редактирования. Проверим функцию на более простом примере:

```
In[6]:= s1 = "golod"; s2 = "gold";

In[7]:= Ed[5, 4]

Out[7]= 1
```

### *8.38. Задача “Раскраски куба”*

Сколькими способами можно раскрасить вершины куба в три цвета, например в красный, зеленый и синий {R, G, B}?

**Решение.** На первый взгляд может показаться, что задача совсем простая. Поскольку каждую из восьми вершин куба можно раскрасить тремя способами, причем независимо от того, как раскрашены другие вершины, то множество всех вершин куба можно раскрасить 3^8=6561 способами.
Однако при таком подходе к решению задачи молчаливо предполагается, что мы умеем различать вершины куба перед окраской, т.е., скажем, куб жестко закреплен или его вершины занумерованы. При этом полученный ответ можно интерпретировать следующим образом: можно так раскрасить 6561 абсолютно одинаковых, жестко закрепленных кубов, что они все будут различаться. Для 3^8+1 кубов это сделать уже нельзя (т.е. пара кубов из них будет одинакова). Ситуация существенно меняется, если мы откажемся от предположения о том, что кубы жестко закреплены, т.к. по-разному окрашенные кубы можно повернуть в пространстве так, что в новом положении их раскраски совпадут. Естественно считать, что два куба раскрашены одинаково, если их раскраски совпадают после некоторого вращения одного из кубов в пространстве. Будем говорить, что такие раскраски кубов геометрически неотличимы. Поэтому естественным уточнением задачи о раскраске является следующая задача:
Сколькими геометрически различными способами можно раскрасить вершины куба в три цвета?
Пусть M – 6561 - элементное множество всевозможных по-разному раскрашенных кубов одного размера, положение которых в пространстве фиксировано, а gs – группа всех вращений куба, состоящая из 24 перестановок. Группа gs естественным образом определяет группу перестановок на множестве M. Именно: если $\alpha \epsilon$gs - некоторое вращение, то каждому кубу из M можно сопоставить некоторый, вообще говоря, другой куб, который получается из первого при вращении $\alpha$.Это соответствие является, очевидно, перестановкой на множестве M, которую будем также обозначать через $\alpha$.
То, что два куба из M раскрашены геометрически одинаково, означает, что один из них можно перевести вращением в такое положение, в котором они геометрически неотличимы. Иными словами, существует такая перестановка, что оба куба находятся в одной орбите действия группы gs, действующей на множестве M. Таким образом, для того чтобы определить число геометрически неразличимых способов раскраски вершин куба, нужно найти количество орбит действия группы gs на множестве M.
Рассмотрим группу симметрий куба gs, выраженную как перестановки вершин. Первые девять перестановок в этом списке получены вращением куба вокруг трех осей, которые проходят через центры противоположных граней. Следующие шесть перестановок получены вращением куба вокруг шести осей, проходящих через центры противоположных ребер. Последующие восемь получены вращением куба вокруг четырех диагоналей куба. Последняя перестановка - тождественная.

```
In[2]:= gs = {{2, 3, 4, 1, 6, 7, 8, 5}, {3, 4, 1, 2, 7, 8, 5, 6}, {4, 1, 2, 3, 8, 5, 6, 7},
    {2, 8, 5, 3, 6, 4, 1, 7},
   {8, 7, 6, 5, 4, 3, 2, 1}, {7, 1, 4, 6, 3, 5, 8, 2}, {4, 3, 5, 6, 8, 7, 1, 2},
    {6, 5, 8, 7, 2, 1, 4, 3}, {7, 8, 2, 1, 3, 4, 6, 5}, {2, 1, 7, 8, 6, 5, 3, 4},
    {5, 6, 4, 3, 1, 2, 8, 7}, {7, 6, 5, 8, 3, 2, 1, 4},
   {5, 8, 7, 6, 1, 4, 3, 2}, {4, 6, 7, 1, 8, 2, 3, 5}, {5, 3, 2, 8, 1, 7, 6, 4},
    {1, 7, 8, 2, 5, 3, 4, 6},
   {1, 4, 6, 7, 5, 8, 2, 3}, {8, 2, 1, 7, 4, 6, 5, 3}, {3, 2, 8, 5, 7, 6, 4, 1},
    {8, 5, 3, 2, 4, 1, 7, 6},
   {6, 4, 3, 5, 2, 8, 7, 1}, {6, 7, 1, 4, 2, 3, 5, 8}, {3, 5, 6, 4, 7, 1, 2, 8},
    {1, 2, 3, 4, 5, 6, 7, 8}};
```

Проверим с помощью встроенного теста PermutationGroupQ, является ли gs группой перестановок:

```
In[3]:= PermutationGroupQ[gs]

Out[3]= True
```

Применяя встроенную функцию MultiplicationTable легко посмотреть таблицу умножения элементов группы gs.
Это очень удобный способ проверки того, является ли множество S с бинарной операцией ▲ группой. Это двумерная таблица размера |S|×|S|, чей элемент, стоящий на пересечении i-ой строки и j-го столбца равен элементу k тогда и только тогда, когда i-й элемент, умноженный на j-й элемент в S, равен k-му элементу в S. Например, в нижеприведенной таблице элементов произведение двадцать третьего элемента на двадцать второй элемент равно двадцать четвертому элементу группы gs.

```
In[4]:= MultiplicationTable[gs, Permute] // MatrixForm
```

$$\begin{pmatrix}
2 & 3 & 24 & 23 & 13 & 18 & 17 & 12 & 20 & 19 & 22 & 5 & 8 & 16 & 21 & 4 & 10 & 15 & 7 & 11 & 6 & 9 & 14 & 1 \\
3 & 24 & 1 & 14 & 8 & 15 & 10 & 5 & 11 & 7 & 9 & 13 & 12 & 4 & 6 & 23 & 19 & 21 & 17 & 22 & 18 & 20 & 16 & 2 \\
24 & 1 & 2 & 16 & 12 & 21 & 19 & 13 & 22 & 17 & 20 & 8 & 5 & 23 & 18 & 14 & 7 & 6 & 10 & 9 & 15 & 11 & 4 & 3 \\
20 & 15 & 19 & 5 & 6 & 24 & 23 & 14 & 16 & 18 & 21 & 17 & 22 & 2 & 8 & 10 & 1 & 9 & 13 & 12 & 7 & 3 & 11 & 4 \\
12 & 8 & 13 & 6 & 24 & 4 & 11 & 2 & 10 & 9 & 7 & 1 & 3 & 15 & 14 & 18 & 20 & 16 & 22 & 17 & 23 & 19 & 21 & 5 \\
17 & 14 & 22 & 24 & 4 & 5 & 21 & 15 & 18 & 16 & 23 & 20 & 19 & 8 & 2 & 9 & 12 & 10 & 3 & 1 & 11 & 13 & 7 & 6 \\
23 & 11 & 21 & 19 & 10 & 17 & 8 & 9 & 24 & 2 & 5 & 16 & 18 & 22 & 20 & 3 & 14 & 1 & 15 & 4 & 12 & 6 & 13 & 7 \\
13 & 5 & 12 & 15 & 2 & 14 & 9 & 24 & 7 & 11 & 10 & 3 & 1 & 6 & 4 & 21 & 22 & 23 & 20 & 19 & 16 & 17 & 18 & 8 \\
18 & 10 & 16 & 20 & 11 & 22 & 24 & 7 & 8 & 5 & 2 & 21 & 23 & 17 & 19 & 12 & 6 & 13 & 4 & 15 & 3 & 14 & 1 & 9 \\
16 & 9 & 18 & 17 & 7 & 19 & 5 & 11 & 2 & 24 & 8 & 23 & 21 & 20 & 22 & 1 & 4 & 3 & 6 & 14 & 13 & 15 & 12 & 10 \\
21 & 7 & 23 & 22 & 9 & 20 & 2 & 10 & 5 & 8 & 24 & 18 & 16 & 19 & 17 & 13 & 15 & 12 & 14 & 6 & 1 & 4 & 3 & 11 \\
8 & 13 & 5 & 21 & 3 & 16 & 20 & 1 & 17 & 22 & 19 & 24 & 2 & 18 & 23 & 6 & 9 & 14 & 11 & 7 & 4 & 10 & 15 & 12 \\
5 & 12 & 8 & 18 & 1 & 23 & 22 & 3 & 19 & 20 & 17 & 2 & 24 & 21 & 16 & 15 & 11 & 4 & 9 & 10 & 14 & 7 & 6 & 13 \\
22 & 6 & 17 & 8 & 15 & 2 & 16 & 4 & 23 & 21 & 18 & 19 & 20 & 24 & 5 & 7 & 3 & 11 & 12 & 13 & 10 & 1 & 9 & 14 \\
19 & 4 & 20 & 2 & 14 & 8 & 18 & 6 & 21 & 23 & 16 & 22 & 17 & 5 & 24 & 11 & 13 & 7 & 1 & 3 & 9 & 12 & 10 & 15 \\
9 & 18 & 10 & 12 & 21 & 3 & 4 & 23 & 14 & 6 & 15 & 7 & 11 & 1 & 13 & 17 & 24 & 22 & 5 & 8 & 19 & 2 & 20 & 16 \\
14 & 22 & 6 & 7 & 19 & 10 & 12 & 20 & 1 & 3 & 13 & 4 & 15 & 9 & 11 & 24 & 16 & 2 & 21 & 23 & 5 & 18 & 8 & 17 \\
10 & 16 & 9 & 1 & 23 & 13 & 6 & 21 & 15 & 4 & 14 & 11 & 7 & 12 & 3 & 20 & 5 & 19 & 24 & 2 & 22 & 8 & 17 & 18 \\
4 & 20 & 15 & 10 & 17 & 7 & 13 & 22 & 3 & 1 & 12 & 14 & 6 & 11 & 9 & 2 & 23 & 24 & 18 & 16 & 8 & 21 & 5 & 19 \\
15 & 19 & 4 & 11 & 22 & 9 & 1 & 17 & 12 & 13 & 3 & 6 & 14 & 10 & 7 & 5 & 18 & 8 & 23 & 21 & 24 & 16 & 2 & 20 \\
7 & 23 & 11 & 3 & 16 & 12 & 15 & 18 & 6 & 14 & 4 & 9 & 10 & 13 & 1 & 22 & 8 & 17 & 2 & 24 & 20 & 5 & 19 & 21 \\
6 & 17 & 14 & 9 & 20 & 11 & 3 & 19 & 13 & 12 & 1 & 15 & 4 & 7 & 10 & 8 & 21 & 5 & 16 & 18 & 2 & 23 & 24 & 22 \\
11 & 21 & 7 & 13 & 18 & 1 & 14 & 16 & 4 & 15 & 6 & 10 & 9 & 3 & 12 & 19 & 2 & 20 & 8 & 5 & 17 & 24 & 22 & 23 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24
\end{pmatrix}$$

Найдем число орбит действия группы gs на перестановках длины 3. Воспользуемся функцией OrbitRepresentatives[g, x], которая возвращает представителя каждой орбиты, порожденной дей-

ствием группы g на x. Количество элементов в этом множестве есть число орбит, и это число дает нам искомый результат.

```
In[5]:= Length[OrbitRepresentatives[gs, Strings[{R, G, B}, 8]]]
Out[5]= 333
```

Таким образом, число геометрически неразличимых способов раскраски вершин куба в три цвета равно 333.

Эту задачу решим теперь, применяя очень красивую лемму Коши-Фробениуса, часто называемую леммой Бернсайда, по имени английского математика – алгебраиста В.Бернсайда (1852-1927), который, по-видимому, первым опубликовал ее доказательство в своей книге по теории конечных групп (1911 г.). Это простое утверждение является основой теории перечисления, разработанной Д.Пойа и рядом других математиков, - теории, находящей широкое применение в кибернетике, технике, органической химии, биологии и т.д.

Пусть k($\alpha$) – число неподвижных точек перестановки $\alpha$ , t(G) - число орбит группы перестановок G, действующей на множестве M={1,2,….,n}. Лемма Бернсайда утверждает, что число орбит t(G) равно

$$\frac{1}{|G|}\sum_{\alpha \in G} k(\alpha),$$

где суммирование берется по всем $\alpha \in$ G.

Типом перестановки называется список {k1, k2, …., kn}, где ki - число циклов длины i. Встроенная функция PermutationType[s] возвращает тип перестановки s. Рассмотрим типы перестановок из группы gs:

```
In[6]:= p = Map[PermutationType, gs]
Out[6]= {{0, 0, 0, 2, 0, 0, 0, 0}, {0, 4, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 2, 0, 0, 0, 0},
         {0, 0, 0, 2, 0, 0, 0, 0}, {0, 4, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 2, 0, 0, 0, 0},
         {0, 0, 0, 2, 0, 0, 0, 0}, {0, 4, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 2, 0, 0, 0, 0},
         {0, 4, 0, 0, 0, 0, 0, 0}, {0, 4, 0, 0, 0, 0, 0, 0}, {0, 4, 0, 0, 0, 0, 0, 0},
         {0, 4, 0, 0, 0, 0, 0, 0}, {0, 4, 0, 0, 0, 0, 0, 0}, {0, 4, 0, 0, 0, 0, 0, 0},
         {2, 0, 2, 0, 0, 0, 0, 0}, {2, 0, 2, 0, 0, 0, 0, 0}, {2, 0, 2, 0, 0, 0, 0, 0},
         {2, 0, 2, 0, 0, 0, 0, 0}, {2, 0, 2, 0, 0, 0, 0, 0}, {2, 0, 2, 0, 0, 0, 0, 0},
         {2, 0, 2, 0, 0, 0, 0, 0}, {2, 0, 2, 0, 0, 0, 0, 0}, {8, 0, 0, 0, 0, 0, 0, 0}}
```

Перечислим различные типы перестановок:

```
In[7]:= t = Union[p]
Out[7]= {{0, 0, 0, 2, 0, 0, 0, 0}, {0, 4, 0, 0, 0, 0, 0, 0},
         {2, 0, 2, 0, 0, 0, 0, 0}, {8, 0, 0, 0, 0, 0, 0, 0}}
```

Подсчитаем количество перестановок каждого типа в группе gs:

```
In[8]:= Map[{#, Apply[Plus, Distribution[{#}, p]]} &, t] // ColumnForm
Out[8]= {{0, 0, 0, 2, 0, 0, 0, 0}, 6}
        {{0, 4, 0, 0, 0, 0, 0, 0}, 9}
        {{2, 0, 2, 0, 0, 0, 0, 0}, 8}
        {{8, 0, 0, 0, 0, 0, 0, 0}, 1}
```

Итак, в группе gs вращений куба имеется
6 перестановок типа {0,0,0,2,0,0,0,0},
9 перестановок типа {0,4,0,0,0,0,0,0},
8 перестановок типа {2,0,2,0,0,0,0,0},
1 перестановка типа {8,0,0,0,0,0,0,0} (тождественная).

Подсчитаем число неподвижных точек. Каждая перестановка типа {0,0,0,2,0,0,0,0} имеет 3^2 неподвижных точек, типа {0,4,0,0,0,0,0,0} и {2,0,2,0,0,0,0,0} имеют 3^4 неподвижных точек и последнего типа 3^8 неподвижных точек. По лемме Бернсайда имеем:

```
In[9]:= (1/24) (6 3^2 + 9 3^4 + 8 3^4 + 3^8)
Out[9]= 333
```

### *8.39. Задача "Раскраска зонта"*

Сколькими способами можно раскрасить четырьмя красками круг разделенный на 6 равных секторов? При этом способы, приводящие к совпадению при повороте вокруг центра окружности, из рассмотрения исключаются.

**Решение.** Так как красок 6, а число частей, на которые разделили круг равно 4, то всего возможных способов раскраски равно:

```
In[2]:= 4^6
Out[2]= 4096
```

Пусть M – множество всевозможных раскрасок круга, состоящее из 4096 элементов, G-группа шестого порядка поворотов круга, то есть циклическая группа порядка 6. Рассмотрим действие группы G на множестве M, тогда совпадающие способы раскраски – это точки одной орбиты действия циклической группы шестого порядка на перестановках длины 4. Таким образом, необходимо подсчитать число орбит.

```
In[3]:= Length[OrbitRepresentatives[CyclicGroup[4], Strings[{A, B, C, D, E, F}, 4]]]
Out[3]= 336
```

Гораздо проще эту задачу можно решить одним нажатим клавиши, применяя встроенную функцию NumberOfNecklaces

```
In[4]:= NumberOfNecklaces[4, 6, Cyclic]
Out[4]= 336
```

### *8.40. Задача "Раскраски граней куба"*

Сколькими различными способами можно окрасить грани куба в шесть различных цветов?

**Решение.** Если мы игнорируем симметрии куба, существует 720 различных способов раскраски граней. Симметрии куба могут быть выражены как группа перестановок вершин куба. Группа симметрий вершин куба действует на множестве граней куба, индуцируя группу перестановок симметрий граней куба. Загрузим стандартный пакет расширения <<Graphics `Polyhedra`, содержащий определения различных полиэдров, включая пять платоновых тел.

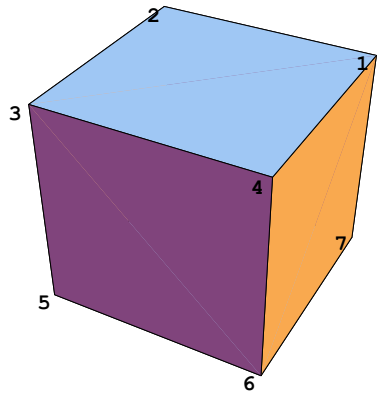
```
In[2]:= << Graphics`Polyhedra`
```

Система "Mathematica"обладает изумительными графическими возможностями. В частности, этот пакет вполне может заменить множество картонных и каркасных объемных фигур, которые можно еще встретить в кабинетах математики школ и вузов. Он является прекрасным инструментом для визуализации задач, допускающих представление результатов в графической форме. При этом полиэдры имеют разнообразную функциональную окраску, их можно интерактивно вращать, выбирать наиболее удобную точку обзора, добиваясь лучшей выразительности и лучшего обзора многогранников. Кроме того, этот пакет несет обширную информацию о полиэдрах

(координаты вершин, количество ребер, граней и т.д.). Все трехмерные изображения в “Mathematica” можно модифицировать, используя многочисленные опции и директивы трехмерной графики. Их применение позволяет легко управлять характером и типом графических объектов, придавая им вид, удобный для заданного применения.

Здесь изображен объект куб, определенный в пакете, выводится список координат вершин куба, и соответствующая нумерация вершин куба.

```
In[3]:= Show[Graphics3D[{Cube[], v = Vertices[Cube];
        Table[ Text[i, v[[i]] – 0.05], {i, 7}]}], Boxed –> False, TextStyle –> {FontWeight
        –> "Bold", FontSize –> 16}]
```




Группа перестановок симметрий вершин куба, перечисленная выше, действует на гранях куба и порождает группу перестановок граней. Чтобы построить эту группу, начнем с множества граней куба, используем встроенную функцию Faces в пакете <<Graphics`Polyhedra`, которая перечисляет грани куба.

```
In[4]:= Faces[Cube]
Out[4]= {{1, 2, 3, 4}, {1, 4, 6, 7}, {1, 7, 8, 2}, {2, 8, 5, 3}, {5, 8, 7, 6}, {3, 5, 6, 4}}
```

Упорядочим элементы этого множества:

```
In[5]:= s = Map[Sort, Faces[Cube]]
Out[5]= {{1, 2, 3, 4}, {1, 4, 6, 7}, {1, 2, 7, 8}, {2, 3, 5, 8}, {5, 6, 7, 8}, {3, 4, 5, 6}}
```

Функция Faces возвращает грани куба в следующем порядке: верх, справа, зад, слева, низ и фасад относительно изображения куба, показанного выше. Грани занумерованы от 1 до 6 в этом порядке.

Рассмотрим группу симметрий граней, порожденную группой симметрий вершин. Заметим, что каждая перестановка в группе симметрий граней является перестановкой 6-го порядка, так как куб имеет 6 граней. Воспользуемся функцией KSubsetGroup[g,s], которая возвращает группу из n элементов, порожденную действием группы g на множестве s всех k-подмножеств n-элементного множества.

```
In[6]:= g = KSubsetGroup[gs, s]
Out[6]= {{1, 3, 4, 6, 5, 2}, {1, 4, 6, 2, 5, 3}, {1, 6, 2, 3, 5, 4}, {4, 1, 3, 5, 2, 6},
        {5, 4, 3, 2, 1, 6}, {2, 5, 3, 1, 4, 6}, {6, 2, 1, 4, 3, 5}, {5, 2, 6, 4, 1, 3},
        {3, 2, 5, 4, 6, 1}, {3, 4, 1, 2, 6, 5}, {6, 4, 5, 2, 3, 1}, {5, 3, 2, 6, 1, 4},
        {5, 6, 4, 3, 1, 2}, {2, 1, 6, 5, 4, 3}, {4, 5, 6, 1, 2, 3}, {3, 1, 2, 5, 6, 4},
        {2, 3, 1, 6, 4, 5}, {3, 5, 4, 1, 6, 2}, {4, 6, 1, 3, 2, 5}, {4, 3, 5, 6, 2, 1},
        {6, 5, 2, 1, 3, 4}, {2, 6, 5, 3, 4, 1}, {6, 1, 4, 5, 3, 2}, {1, 2, 3, 4, 5, 6}}
```

Таблица умножения элементов множества g показывает, что индуцированное множество перестановок действительно является группой. Проверим это и с помощью теста:

```
In[7]:= PermutationGroupQ[g]
Out[7]= True
```

Используя OrbitRepresentatives, получим число способов раскраски граней куба в шесть различных цветов:

```
In[8]:= Length[OrbitRepresentatives[g, Permutations[6]]]
Out[8]= 30
```

Следовательно, грани куба можно раскрасить в шесть различных цветов тридцатью способами. Приведем список этих способов:

```
In[9]:= OrbitRepresentatives[g, Permutations[6]]
Out[9]= {{1, 3, 4, 6, 5, 2}, {1, 3, 4, 5, 6, 2}, {1, 3, 5, 6, 4, 2},
        {1, 3, 5, 4, 6, 2}, {1, 3, 6, 5, 4, 2}, {1, 3, 6, 4, 5, 2},
        {1, 4, 3, 6, 5, 2}, {1, 4, 3, 5, 6, 2}, {1, 4, 5, 6, 3, 2}, {1, 4, 5, 3, 6, 2},
        {1, 4, 6, 5, 3, 2}, {1, 4, 6, 3, 5, 2}, {1, 5, 3, 6, 4, 2}, {1, 5, 3, 4, 6, 2},
        {1, 5, 4, 6, 3, 2}, {1, 5, 4, 3, 6, 2}, {1, 5, 6, 4, 3, 2}, {1, 5, 6, 3, 4, 2},
        {1, 6, 3, 5, 4, 2}, {1, 6, 3, 4, 5, 2}, {1, 6, 4, 5, 3, 2}, {1, 6, 4, 3, 5, 2},
        {1, 6, 5, 4, 3, 2}, {1, 6, 5, 3, 4, 2}, {1, 4, 5, 6, 2, 3}, {1, 4, 6, 5, 2, 3},
        {1, 5, 4, 6, 2, 3}, {1, 5, 6, 4, 2, 3}, {1, 6, 4, 5, 2, 3}, {1, 6, 5, 4, 2, 3}}
```

### *8.41. Задача "Ожерелья"*

Сколько различных ожерелий можно составить из девяти бусин трех различных цветов?

**Решение.** Вычислим число строк длины 9 , которые можно получить, используя алфавит {R, G, B}:

```
In[2]:= Length[Strings[{R, G, B}, 9]]
Out[2]= 19683
```

Подсчитаем число различных ожерелий из девяти бусин, окрашенных, по крайней мере, в три цвета, причем ожерелья считаются геометрически неотличимыми, если получаются одно из другого вращением или отражением относительно осей симметрии или суперпозицией этих преобразований:

```
In[3]:= Length[OrbitRepresentatives[DihedralGroup[9], Strings[{R, G, B}, 9]]]
Out[3]= 1219
```

Получаем, что список из 19683 ожерелий содержит только 1219 различных ожерелий. Подсчитаем теперь число различных ожерелий, причем геометрически неотличимыми будем считать ожерелья, которые получаются одно из другого только преобразованием вращения

```
In[4]:= Length[OrbitRepresentatives[CyclicGroup[9], Strings[{R, G, B}, 9]]]
Out[4]= 2195
```

Пакет расширения <<DiscreteMath `Combinatorica` содержит функции для непосредственного решения задачи о перечислении ожерелий. Используя эти функции нужное количество ожерелий можно получить гораздо проще.

Функция NecklacePolynomial [n, c, Cyclic] возвращает многочлен от переменной x, коэффициенты которого представляют собой число способов раскраски ожерелья из n бусин в цвета из списка c, причем два ожерелья считаются геометрически неотличимыми, если могут быть получены один из другого преобразованием вращения. Применяя аргумент Dihedral вместо Cyclic, можно получить тот же самый многочлен, но два ожерелья считаются геометрически неотличимыми, если могут быть получены один из другого преобразованием вращения или симметрии относительно осей.

Функция NumberOfNecklaces[n, c, Cyclic] возвращает число различных ожерелий, которые можно составить из n бусин, окрашенных в c различных цветов, причем два ожерелья неотличимы, если одно может быть получено из другого вращением. Если же мы считаем два ожерелья неотличимыми, при условии, что одно ожерелье может быть получено из другого вращением или отражением относительно осей, то нужно применить функцию NecklacePolynomial:

```
In[5]:= NecklacePolynomial[9, 3, Dihedral]
Out[5]= 1219

In[6]:= NecklacePolynomial[9, 3, Cyclic]

Out[6]= 2195

In[7]:= NumberOfNecklaces[9, 3, Dihedral]

Out[7]= 1219

In[8]:= NumberOfNecklaces[9, 3, Cyclic]

Out[8]= 2195
```

Если необходимо получить все различные ожерелья, бусины которых окрашены в цвета из списка c, то можно вызвать функцию ListNecklaces[n, c ,Cyclic] (или ListNecklaces[n, c, Dihedral] в зависимости от того, какие ожерелья считаются геометрически неотличимыми).

```
In[9]:= ListNecklaces[4, {R, G, B, R}, Dihedral]

Out[9]= {{R, G, B, R}, {B, R, G, R}}

In[10]:= ListNecklaces[4, {R, G, B, R}, Cyclic]

Out[10]= {{R, G, B, R}, {B, G, R, R}, {B, R, G, R}}
```

Рассмотрим встроенную функцию <<DiscreteMath`Combinatorica` NecklacePolynomial[n, c, Cyclic (Dihedral)]. Она возвращает полином от цветов из списка c, коэффициенты этого полинома представляют собой число ожерелий из n бусин, цвета которых выбраны из списка цветов c. Полученный многочлен содержит в себе обширную информацию, интересующую нас в задаче о перечислении ожерелий. Рассмотрим эту функцию на конкретном примере перечисления ожерелий, составленных из семи бусин, окрашенных по крайней мере в три цвета:

```
In[11]:= NecklacePolynomial[7, {R, G, B}, Dihedral]

Out[11]= B^7 + B^6 G + 3 B^5 G^2 + 4 B^4 G^3 + 4 B^3 G^4 + 3 B^2 G^5 + B G^6 + G^7 + B^6 R + 3 B^5 G R + 9 B^4 G^2 R +
         10 B^3 G^3 R + 9 B^2 G^4 R + 3 B G^5 R + G^6 R + 3 B^5 R^2 + 9 B^4 G R^2 + 18 B^3 G^2 R^2 + 18 B^2 G^3 R^2 +
         9 B G^4 R^2 + 3 G^5 R^2 + 4 B^4 R^3 + 10 B^3 G R^3 + 18 B^2 G^2 R^3 + 10 B G^3 R^3 + 4 G^4 R^3 +
         4 B^3 R^4 + 9 B^2 G R^4 + 9 B G^2 R^4 + 4 G^3 R^4 + 3 B^2 R^5 + 3 B G R^5 + 3 G^2 R^5 + B R^6 + G R^6 + R^7
```

Здесь мы получили многочлен от переменных R,G,B, для которого степень каждого члена суммы равныа7. Коэффициент перед слагаемым $R^mB^nG^k$ есть число ожерелий из семи бусин, в котором m бусин – цвета R, n бусин – цвета B и k бусин – цвета G. Этот полином показывает

нам, что существует ровно 18 ожерелий из семи бусин, с тремя бусинами цвета R, двумя бусинами цвета B и двумя бусинами цвета G. Коэффициент при $R^7$ , $G^7$, $B^7$ равен единице, поскольку существует ровно одно монохроматическое ожерелье каждого цвета.

Заменяя символы R, G, B единицей, получим число различных ожерелий из семи бусин, окрашенных по крайней мере в три цвета:

```
In[12]:= NecklacePolynomial[7, {R, G, B}, Dihedral] /. {R → 1, G → 1, B → 1}

Out[12]= 198
```

Если в функции NecklacePolynomial переменную специфицировать как число цветов, в результате получим многочлен следующего вида:

```
In[13]:= NecklacePolynomial[7, m, Dihedral]

Out[13]= 3 m/7 + m^4/2 + m^7/14

In[14]:= % /. {m → 3}

Out[14]= 198
```

Таким образом, вызывая функцию NecklacePolynomial, задачу раскраски зонта можно решить и таким способом:

```
In[15]:= NecklacePolynomial[4, {R, G, B, A, C, D}, Cyclic] /. {R → 1, G → 1,
          B → 1, A → 1, C → 1, D → 1}
Out[15]= 336
```

### *8.42. Задача "Простая группа $A_5$"*

Напомним, что знакопеременная группа $A_n$ при $n \geq 5$ является простой (результат Э.Галуа).
Этот факт можно установить индукцией по n, если доказать, что знакопеременная группа $A_5$ – простая.

Утверждение. Знакопеременная группа $A_5$ – простая.

Рассмотрим все четные перестановки пятого порядка, вызвав встроенную функцию AlternatingGroup[5]:

```
In[2]:= A5 = AlternatingGroup[5];
```

Эта команда выдает нам мощность группы A5, равную 60.

```
In[3]:= Length[A5]

Out[3]= 60
```

Проверим, является ли A5 группой относительно умножения перестановок:

```
In[4]:= PermutationGroupQ[A5]

Out[4]= True
```

Это можно проверить также, построив таблицу умножения, вызвав встроенную функцию MultiplicationTable. Это очень удобный способ проверки того, является ли множество с данной бинарной операцией группой.

```
In[5]:= Multipl = MultiplicationTable[A5, Permute];
```

Из таблицы умножения Multipl можно увидеть, например, что произведение тридцать девятого и пятьдесят восьмого элементов A5 есть пятнадцатый элемент A5.

```
In[6]:= Multipl[[59, 31]]

Out[6]= 15
```

Типом перестановки размера n называется список {k1,k2,....,kn}, где ki – число циклов перестановки длины i.

```
In[7]:= s = Map[PermutationType, A5];
```

Рассмотрим различные типы перестановок в A5:

```
In[8]:= t = Union[Map[PermutationType, A5]]

Out[8]= {{0, 0, 0, 0, 1}, {1, 2, 0, 0, 0}, {2, 0, 1, 0, 0}, {5, 0, 0, 0, 0}}
```

Найдем количество перестановок каждого типа:

```
In[9]:= Map [{#, Apply [Plus , Distribution  [{#}, s]]} &, t] // ColumnForm
Out[9]= {{0, 0, 0, 0, 1}, 24}
        {{1, 2, 0, 0, 0}, 15}
        {{2, 0, 1, 0, 0}, 20}
        {{5, 0, 0, 0, 0}, 1}
```

Видим, что в группе A5 ровно 24 перестановки, состоящих из одного цикла длины 5, 15 перестановок, раскладываемых в произведение одного цикла длины 1 и двух циклов длины 2, 20 перестановок, состоящих из двух циклов длины 1 и одного цикла длины 3, и одна тождественная перестановка – произведение пяти циклов длины 1.
Выделим четные перестановки типа {1, 2, 0, 0, 0} в список sp22:

```
In[10]:= sp22 = Select[A5, PermutationType[#][[2]] > 0 &]

Out[10]= {{1, 3, 2, 5, 4}, {1, 4, 5, 2, 3}, {1, 5, 4, 3, 2}, {2, 1, 3, 5, 4}, {2, 1, 4, 3, 5},
         {2, 1, 5, 4, 3}, {3, 2, 1, 5, 4}, {3, 4, 1, 2, 5}, {3, 5, 1, 4, 2}, {4, 2, 5, 1, 3},
         {4, 3, 2, 1, 5}, {4, 5, 3, 1, 2}, {5, 2, 4, 3, 1}, {5, 3, 2, 4, 1}, {5, 4, 3, 2, 1}}
```

Этот список содержит 15 перестановок. Эти перестановки раскладываются в произведение двух транспозиций. Аналогично получим список перестановок типа{2, 0, 1, 0, 0}:

```
In[11]:= sp3 = Select[A5, PermutationType[#][[3]] > 0 &]

Out[11]= {{1, 2, 4, 5, 3}, {1, 2, 5, 3, 4}, {1, 3, 4, 2, 5}, {1, 3, 5, 4, 2}, {1, 4, 2, 3, 5},
          {1, 4, 3, 5, 2}, {1, 5, 2, 4, 3}, {1, 5, 3, 2, 4}, {2, 3, 1, 4, 5}, {2, 4, 3, 1, 5},
          {2, 5, 3, 4, 1}, {3, 1, 2, 4, 5}, {3, 2, 4, 1, 5}, {3, 2, 5, 4, 1}, {4, 1, 3, 2, 5},
          {4, 2, 1, 3, 5}, {4, 2, 3, 5, 1}, {5, 1, 3, 4, 2}, {5, 2, 1, 4, 3}, {5, 2, 3, 1, 4}}
```

Этот список содержит 20 перестановок. Перестановки этого типа содержат один цикл длины 3.
Выделим перестановки типа {0, 0, 0, 0, 5}:

```
In[12]:= sp5 = Select[A5, PermutationType[#][[5]] > 0 &]
```

```
Out[12]= {{2, 3, 4, 5, 1}, {2, 3, 5, 1, 4}, {2, 4, 1, 5, 3}, {2, 4, 5, 3, 1}, {2, 5, 1, 3, 4},
          {2, 5, 4, 1, 3}, {3, 1, 4, 5, 2}, {3, 1, 5, 2, 4}, {3, 4, 2, 5, 1}, {3, 4, 5, 1, 2},
          {3, 5, 2, 1, 4}, {3, 5, 4, 2, 1}, {4, 1, 2, 5, 3}, {4, 1, 5, 3, 2}, {4, 3, 1, 5, 2},
          {4, 3, 5, 2, 1}, {4, 5, 1, 2, 3}, {4, 5, 2, 3, 1}, {5, 1, 2, 3, 4}, {5, 1, 4, 2, 3},
          {5, 3, 1, 2, 4}, {5, 3, 4, 1, 2}, {5, 4, 1, 3, 2}, {5, 4, 2, 1, 3}}
```

Возьмем четную перестановку из sp22:

```
In[13]:= p2 = {1, 3, 2, 5, 4};
```

Построим орбиту Orb5 этой перестановки при действии сопряжением группы A5. Для этого составим функцию orb:

```
In[14]:= orb[p_] := (Orb = {}; Do[g = A5[[i]];
                    Orb = Append[Orb, Permute[Permute[InversePermutation[g], p], g]],
                    {i, 1, 60}]; Union[Orb]);
```

Рассмотрим орбиту элемента p:

```
In[15]:= O1 = orb[p2]
Out[15]= {{1, 3, 2, 5, 4}, {1, 4, 5, 2, 3}, {1, 5, 4, 3, 2}, {2, 1, 3, 5, 4}, {2, 1, 4, 3, 5},
          {2, 1, 5, 4, 3}, {3, 2, 1, 5, 4}, {3, 4, 1, 2, 5}, {3, 5, 1, 4, 2}, {4, 2, 5, 1, 3},
          {4, 3, 2, 1, 5}, {4, 5, 3, 1, 2}, {5, 2, 4, 3, 1}, {5, 3, 2, 4, 1}, {5, 4, 3, 2, 1}}
```

Легко увидеть, что O1 совпадает с sp22
Действительно,

```
In[16]:= Complement [sp22 , O1 ]
Out[16]= {}
```

Таким образом, sp22 и есть класс сопряженных элементов.
Рассмотрим перестановку из списка sp3:

```
In[17]:= p3 = {1, 2, 4, 5, 3};
```

Построим орбиту O3 этой перестановки при действии сопряжением группы A5:

```
In[18]:= O3 = orb[p3]

Out[18]= {{1, 2, 4, 5, 3}, {1, 2, 5, 3, 4}, {1, 3, 4, 2, 5}, {1, 3, 5, 4, 2}, {1, 4, 2, 3, 5},
          {1, 4, 3, 5, 2}, {1, 5, 2, 4, 3}, {1, 5, 3, 2, 4}, {2, 3, 1, 4, 5}, {2, 4, 3, 1, 5},
          {2, 5, 3, 4, 1}, {3, 1, 2, 4, 5}, {3, 2, 4, 1, 5}, {3, 2, 5, 4, 1}, {4, 1, 3, 2, 5},
          {4, 2, 1, 3, 5}, {4, 2, 3, 5, 1}, {5, 1, 3, 4, 2}, {5, 2, 1, 4, 3}, {5, 2, 3, 1, 4}}

In[19]:= Complement[sp3, O3]

Out[19]= {}
```

Элементы sp3 также все сопряжены.
Возьмем четную перестановку p5 из sp5 и снова построим ее орбиту:

```
In[20]:= p5 = {2, 3, 4, 5, 1};
In[21]:= O5 = orb[p5]
```

```
Out[21]= {{2, 3, 4, 5, 1}, {2, 4, 1, 5, 3}, {2, 5, 4, 1, 3}, {3, 1, 5, 2, 4},
          {3, 5, 2, 1, 4}, {3, 5, 4, 2, 1}, {4, 1, 5, 3, 2}, {4, 3, 1, 5, 2},
          {4, 3, 5, 2, 1}, {5, 1, 2, 3, 4}, {5, 4, 1, 3, 2}, {5, 4, 2, 1, 3}}
```

Эта орбита содержит 12 элементов. Рассмотрим оставшиеся 12 элементов в sp5:

```
In[22]:= Comp5 = Complement[sp5, O5]

Out[22]= {{2, 3, 5, 1, 4}, {2, 4, 5, 3, 1}, {2, 5, 1, 3, 4}, {3, 1, 4, 5, 2},
          {3, 4, 2, 5, 1}, {3, 4, 5, 1, 2}, {4, 1, 2, 5, 3}, {4, 5, 1, 2, 3},
          {4, 5, 2, 3, 1}, {5, 1, 4, 2, 3}, {5, 3, 1, 2, 4}, {5, 3, 4, 1, 2}}
```

Получаем, что элементы из sp5 распадаются на два класса с представителями {2,3,4,5,1} и {2,3,5,1,4}.
Объединим эти множества:

```
In[23]:= J = Join [O5 , Comp5 ]

Out[23]= {{2, 3, 4, 5, 1}, {2, 4, 1, 5, 3}, {2, 5, 4, 1, 3}, {3, 1, 5, 2, 4}, {3, 5, 2, 1, 4},
          {3, 5, 4, 2, 1}, {4, 1, 5, 3, 2}, {4, 3, 1, 5, 2}, {4, 3, 5, 2, 1}, {5, 1, 2, 3, 4},
          {5, 4, 1, 3, 2}, {5, 4, 2, 1, 3}, {2, 3, 5, 1, 4}, {2, 4, 5, 3, 1}, {2, 5, 1, 3, 4},
          {3, 1, 4, 5, 2}, {3, 4, 2, 5, 1}, {3, 4, 5, 1, 2}, {4, 1, 2, 5, 3}, {4, 5, 1, 2, 3},
          {4, 5, 2, 3, 1}, {5, 1, 4, 2, 3}, {5, 3, 1, 2, 4}, {5, 3, 4, 1, 2}}

In[24]:= Complement [sp5 , J]

Out[24]= {}
```

Перемножим две перестановки из sp2:

```
In[25]:= p11 = {2, 1, 3, 5, 4}; p12 = {2, 1, 4, 3, 5}; Permute[p11, p12]

Out[25]= {1, 2, 5, 3, 4}
```

Рассмотрим тип произведения этих перестановок:

```
In[26]:= PermutationType  [Permute [p11 , p12 ]]

Out[26]= {2, 0, 1, 0, 0}
```

Видим, что произведение лежит в списке sp3.
Рассмотрим все попарные произведения перестановок из sp3:

```
In[29]:= Union[Map[PermutationType, mult]]

Out[29]= {{0, 0, 0, 0, 1}, {1, 2, 0, 0, 0}, {2, 0, 1, 0, 0}, {5, 0, 0, 0, 0}}

In[29]:= Union[Map[PermutationType, mult]]

Out[29]= {{0, 0, 0, 0, 1}, {1, 2, 0, 0, 0}, {2, 0, 1, 0, 0}, {5, 0, 0, 0, 0}}
```

Аналогично все попарные произведения элементов из sp22:

```
In[30]:= mult22 = {}; Do[
         Do[mult22 = Append[mult22, Permute[sp22[[i]], sp22[[j]]]],
         {j, Length[sp22]}], {i, Length[sp22]}];

In[31]:= Union[Map[PermutationType, mult22]]

Out[31]= {{0, 0, 0, 0, 1}, {1, 2, 0, 0, 0}, {2, 0, 1, 0, 0}, {5, 0, 0, 0, 0}}
```

Рассмотрим все попарные произведения перестановок из sp3:

```
In[32]:= mult5 = {}; Do[Do[mult5 = Append[mult5, Permute[sp5[[i]], sp5[[j]]]], {j, Length[sp5]}],
          {i, Length[sp5]}];

In[33]:= Union[Map[PermutationType, mult5]]
Out[33]= {{0, 0, 0, 0, 1}, {1, 2, 0, 0, 0}, {2, 0, 1, 0, 0}, {5, 0, 0, 0, 0}}
```

Следовательно, если K- нормальная подгруппа в A5, то она, очевидно, совпадает с A5, то есть группа A5 простая. Доказательство простоты группы A5 также очевидно следует из теоремы Лагранжа. Мощность нормальной подгруппы K должна быть равна

$$|K|=\delta 1\times 1+\delta 2\times 15+\delta 3\times 20+\delta 4\times 12+\delta 5\times 12$$

где $\delta 1=1$ (так как $e\in K$) и $|=\delta i$ равно нулю или единице при i=2,3,4,5. По теореме Лагранжа (мощность подгруппы |K| должно быть делителем порядка |A|=60) возможно лишь два варианта:

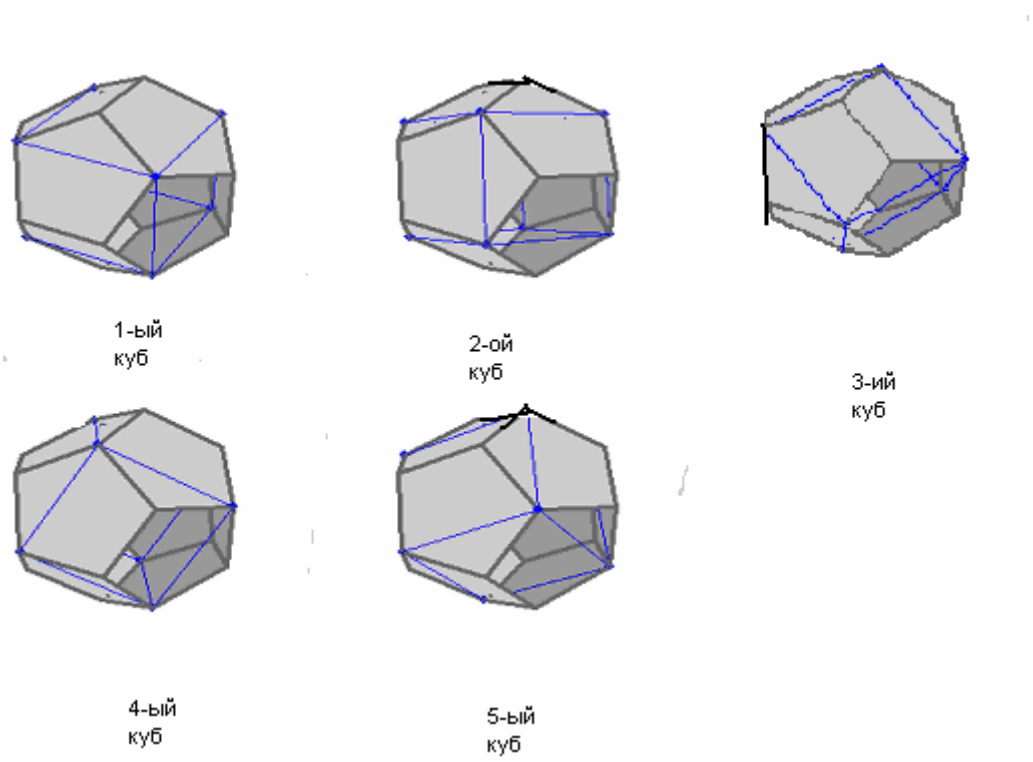
а) $\delta 2=\delta 3=\delta 4=\delta 5=0$; K- единичная подгруппа,

б) $\delta 2=\delta 3=\delta 4=\delta 5=1$; K=A5.

Это также доказывает, что A5 простая группа.

Все перестановки группы A5 представляют группу вращений правильного додекаэдра вокруг осей, проходящих через центр додекаэдра, причем элементам списка sp5 соответствуют вращения вокруг осей, проходящих через центры противоположных граней, элементам списка sp3 соответствуют вращения вокруг осей, проходящих через противоположные вершины, а элементам списка sp22 соответствуют вращения вокруг осей, проходящих через середины противоположных ребер.

В правильный додекаэдр можно вписать пять различных кубов, если считать додекаэдр жестко закрепленным в пространстве. На рисунке приведены все пять кубов, вписанных в один и тот же додекаэдр. Занумеруем их числами от 1 до 5. При каждом вращении правильного додекаэдра куб переходит либо в себя, либо в другой куб. Каждому вращению додекаэдра соответствует перестановка номеров этих пяти кубов, то есть элемент знакопеременной группы A5. Легко показать, что это соответствие является изоморфизмом.




### *8.43. Задача "Ряды"*

Для натурального числа n вычислить суммы
$S_1 = 1 + 3 + \ldots + (2n - 1)$, $S_2 = 1/1\cdot2 + 1/2\cdot3 + \ldots + 1/n\cdot(n + 1)$, $S_3 = 1/1! + 1/2! + \ldots + 1/n!$

**Решение**. $S_1 = n^2$, $S_2 = (1 - 1/2) + (1/2 - 1/3) + \ldots + (1/n - 1/(n + 1)) = 1 - 1/(n + 1)$.
Формулы для подобных сумм в основном получаются методом математической индукции.
Чтобы эффективно вычислить $S_3$ , заметим, что $1/(k + 1)! = 1/((k + 1)\cdot k!)$. Таким образом, зная значение слагаемого 1/k!, нет нужды заново пересчитывать к! в следующем слагаемом суммы. Принцип динамического программирования в действии! Для вычисления суммы применяем простую итерацию:

```
In[2]:= n = 3;

In[3]:= For[p = 1; S = p; i = 2, i < n + 1, i++, p = p / i; S += p]; S
```

$$\text{Out[3]=}\ \frac{5}{3}$$

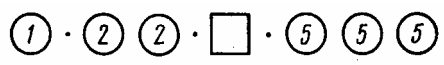
```
In[4]:= n = 10;

In[5]:= For[p = 1; S = p; i = 2, i < n + 1, i++, p = p / i; S += p]; S
```

$$\text{Out[5]=}\ \frac{6235301}{3628800}$$

### *8.44. Задача « Размен монет»*

Сколькими способами можно разменять монету достоинством 20 копеек на медные монеты достоинством в одну, две, три и пять копеек?

**Решение.**Обозначим через $P_n$ – число способов заплатить сумму в n копеек этими четырьмя типами монет. Очевидно, что $P_1=1$, $P_2=2$, $P_3=3$, $P_4=4$, $P_5=6$. Нужно найти $P_{20}$. Положим для удобства $P_0=1$. Один из способов размена монеты в 20 копеек можно представить в виде символического произведения:



Все возможные способы размена монет будут содержаться в произведении, изображенном на рисунке 1:

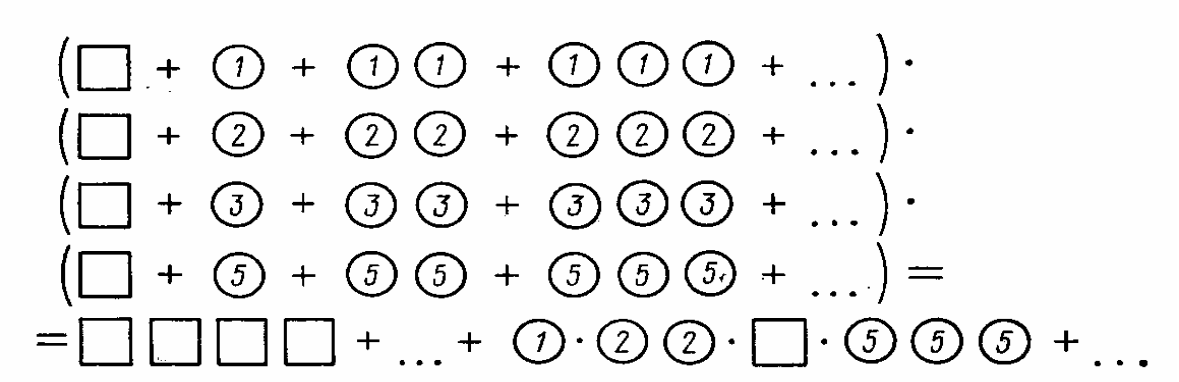
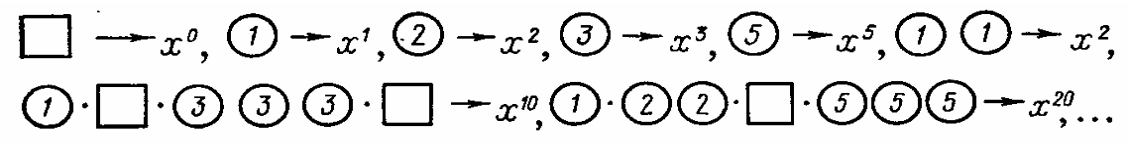


Рис.1

Для того чтобы извлечь из последней строки рисунка 1 те и только те рисунки, которые отвечают способам размена монеты в 20 копеек, сопоставим каждой фигуре переменную x в степени, равной общей стоимости монет фигуры.



Рассмотрим $P(x)=P_0 + P_1x + P_2x^2 +\dots +P_nx^n + \dots$,
Весь рисунок 1 соответствует равенству:

$$(1-x)^{-1}(1-x^2)^{-1}(1-x^3)^{-1}(1-x^5)^{-1}= P_0 + P_1x + P_2x^2 +\dots +P_nx^n + \dots,$$

Нужно вычислить коэффициент $P_{20}$ перед $x^{20}$ в функции

$F(x)= (1-x)^{-1}(1-x^2)^{-1}(1-x^3)^{-1}(1-x^5)^{-1}$:

```
In[2]:= Coefficient[Series[(1/(1-x))(1/(1-x^2))(1/(1-x^3))(1/(1-x^5)),{x,0,20}],
        x,20]

Out[2]= 91
```

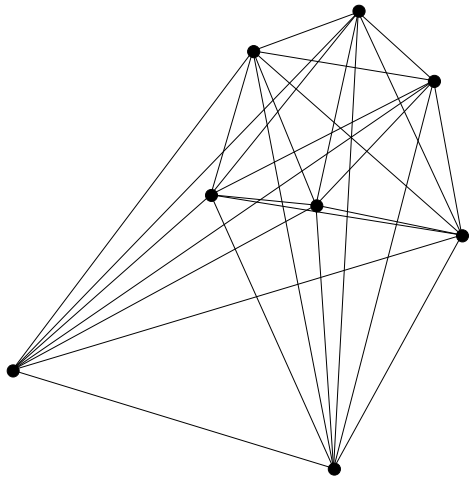
Таким образом, монету достоинством в 20 копеек можно разменять на медные монеты 91 способом.

## 8.45. Алгоритм Кристофидеса нахождения приближенного решения задачи коммивояжера на полном графе с неравенством треугольника

**Пример.**

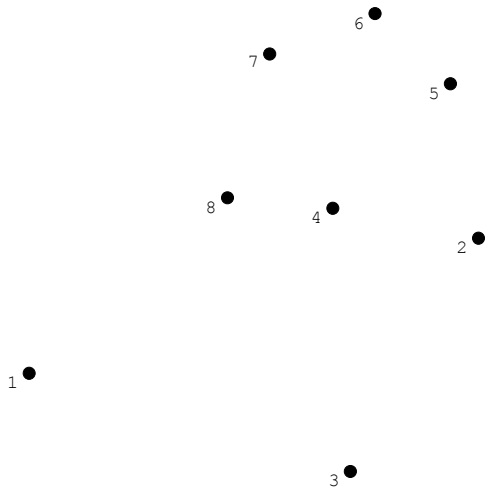
Пусть g – произвольный полный граф. Построим некоторое вложение графа g на плоскость:

```
In[2]:= ShowGraph[g = RandomVertices[CompleteGraph[8]]];
```



Изобразим на плоскости только вершины этого графа, изменяя цвет всех ребер:

```
In[3]:= ShowLabeledGraph[g = SetGraphOptions[g, EdgeColor → White]];
```

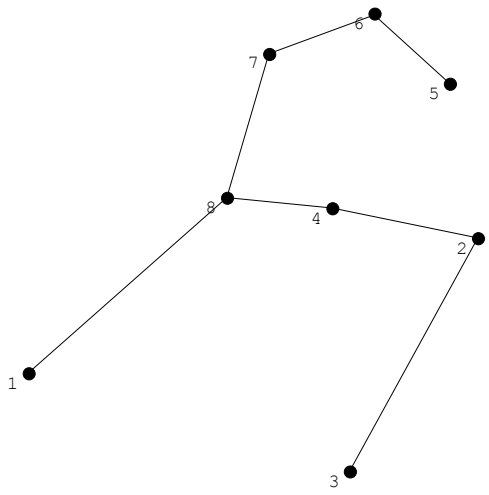



Между всеми точками зададим евклидовы расстояния:

```
In[4]:= g = SetEdgeWeights[g, WeightingFunction → Euclidean];
```

Построим минимальный остов:

```
In[5]:= Ost = SetGraphOptions[MinimumSpanningTree[g],
          EdgeColor → Black];

In[6]:= ShowLabeledGraph[Ost];
```



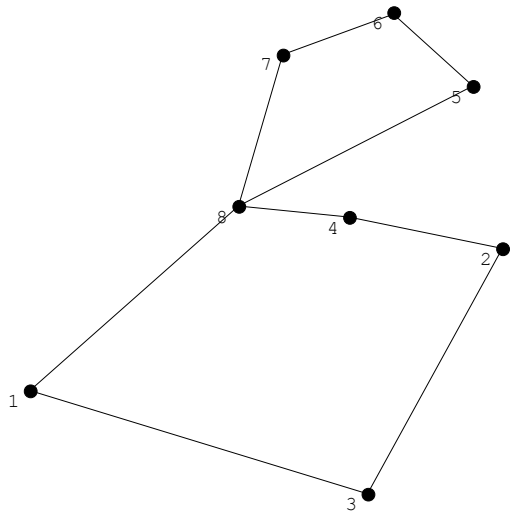

```
In[7]:= Degrees[Ost]
Out[7]= {1, 2, 1, 2, 1, 2, 2, 3}
```

В остове выделим список вершин нечетной степени – {1, 3, 5, 8}. Так как в любом графе количество вершин нечетной степени четно, то в вершинно-порожденном подграфе на этих вершинах существует совершенное паросочетание. Выбираем паросочетание минимального веса:

```
In[8]:= MaximalMatching[InduceSubgraph[g, {1, 3, 5, 8}]]
Out[8]= {{1, 2}, {3, 4}}
```

Здесь ребро {1, 2} подграфа соответствует ребру {1, 3} графа g, а ребро {3, 4} – ребру{5,8}. Добавляем ребра этого паросочетания к остову:

```
In[9]:= gr = AddEdges[Ost, {{1, 3} , {5, 8}}];

In[10]:= ShowLabeledGraph[gr];
```

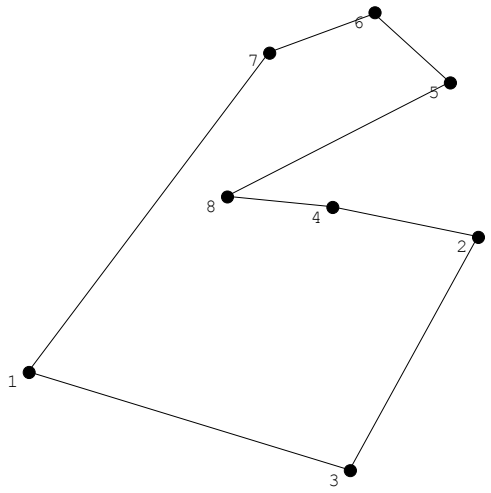



У получившегося графа степени всех вершин – четные, следовательно, в нем существует эйлеров цикл:

```
In[11]:= Ecycle = EulerianCycle[gr]
Out[11]= {1, 3, 2, 4, 8, 5, 6, 7, 8, 1}
```

Проходим по вершинам эйлерова цикла, и если очередная вершина ранее встречалась, то пропускаем ее и переходим к следующей. В конце добавляем начальную вершину цикла. В итоге получим гамильтонов цикл, длина которого, в силу неравенства треугольника меньше чем длина исходного цикла:

```
In[12]:= Gcycle = {1, 3, 2, 4, 8, 5, 6, 7, 1}
Out[12]= {1, 3, 2, 4, 8, 5, 6, 7, 1}
```



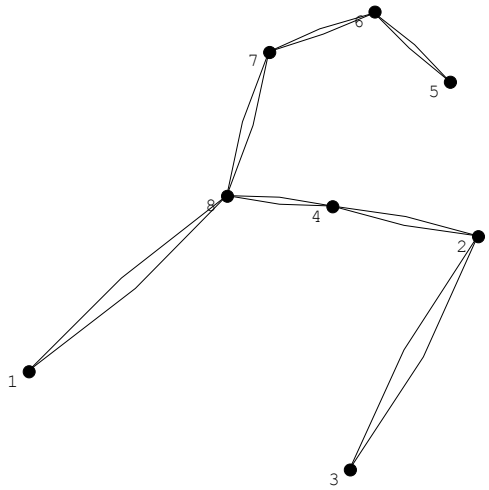

```
In[13]:= CostOfPath[g, Gcycle]
Out[13]= 2.63433
```

Известно, что для этого метода отношение стоимости полученного пути к стоимости оптимального пути не более 1.5. Трудоемкость данного алгоритма – линейное время от числа ребер для нахождения минимального остова плюс трудоемкость нахождения паросочетания.

## 8.46. Алгоритм Эйлера нахождения приближенного решения задачи коммивояжера на полном графе с неравенством треугольника

**Пример.** Рассмотрим алгоритм на графе из предыдущего примера. Этот алгоритм отличается от предыдущего только логикой построения эйлерова цикла на вершинах остова и легкостью программирования.
Ребра остова дублируются и получается эйлеров граф:

```
In[2]:= ShowLabeledGraph[DblOst = AddEdges[Ost, Edges[Ost]]];
```

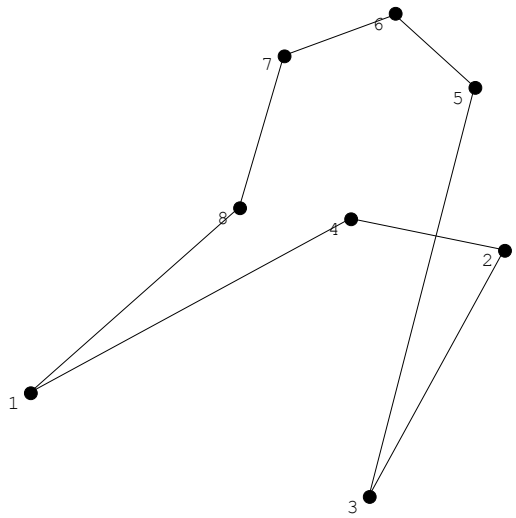



Далее аналогично, строится эйлеров цикл:

```
In[3]:= Ecycle = EulerianCycle[DblOst]
Out[3]= {5, 6, 7, 8, 1, 8, 4, 2, 3, 2, 4, 8, 7, 6, 5}
```

и по нему строится гамильтонов цикл

```
In[4]:= Gcycle = {5, 6, 7, 8, 1, 4, 2, 3, 5}

Out[4]= {5, 6, 7, 8, 1, 4, 2, 3, 5}
```




```
In[5]:= CostOfPath[g, Gcycle]
Out[5]= 2.47542
```

Стоимость Cost полученного пути, в силу неравенства треугольника, не превосходит удвоенной стоимости 2·CostFr минимального остова. Так как при удалении ребра из оптимального пути коммивояжера получается остов, то стоимость минимального остова CostFr меньше стоимости OptCost оптимального пути коммивояжера. Отсюда получается оценка $\text{Cost} \leq 2 \cdot \text{OptCost}$. Трудоемкость данного алгоритма – линейное время от числа ребер для нахождения минимального остова плюс линейное время от числа вершин для построения преобразования эйлерова цикла.

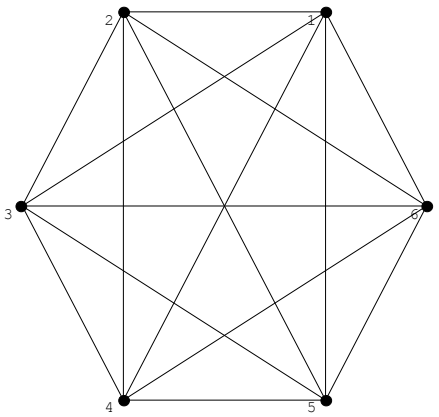
### *8.47. Открытая задача коммивояжера*

Найти минимальную гамильтонову цепь в неориентированном графе с заданной матрицей весов ребер.

Применим для ее поиска метод ветвей и границ.
**Пример**. Построим полный граф на шести вершинах со следующей матрицей весов

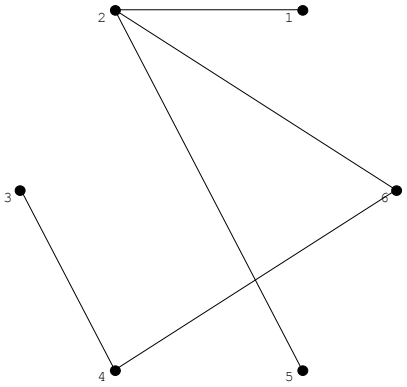
$$\begin{pmatrix} \infty & 4 & 10 & 18 & 5 & 10 \\ 4 & \infty & 12 & 8 & 2 & 6 \\ 10 & 12 & \infty & 4 & 18 & 16 \\ 18 & 8 & 4 & \infty & 14 & 6 \\ 5 & 2 & 18 & 14 & \infty & 16 \\ 10 & 6 & 16 & 6 & 16 & \infty \end{pmatrix}$$

```
In[2]:= c = {{Infinity, 4, 10, 18, 5, 10}, {4, Infinity, 12, 8, 2, 6},
           {10, 12, Infinity, 4, 18, 16}, {18, 8, 4, Infinity, 14, 6},
           {5, 2, 18, 14, Infinity, 16}, {10, 6, 16, 6, 16, Infinity}};

In[3]:= g = FromAdjacencyMatrix[c, EdgeWeight];

In[4]:= ShowLabeledGraph[g];
```




Построим остов минимального веса:

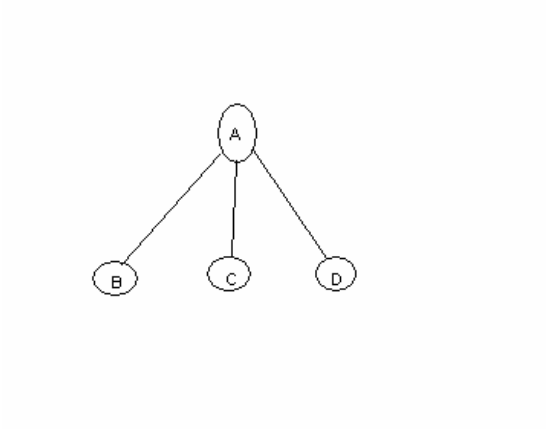
```
In[5]:= ShowLabeledGraph[ost = MinimumSpanningTree[g]];
```




Вычислим его стоимость:

```
In[6]:= CostOfPath[g, {1, 2, 6, 4, 3}] + CostOfPath[g, {2, 5}]

Out[6]= 22

In[7]:= Degrees[ost]

Out[7]= {1, 3, 1, 2, 1, 2}
```

Так как степени вершин цепи не превышают 2, то полученный остов не является цепью, следовательно, можно сказать, что в окончательном ответе должно отсутствовать по крайней мере одно из ребер {2,1},{2,6},{2,5}.
Таким образом, решение первоначальной задачи A является решением по крайней мере одной из следующих трех частныйх задач изображаемыми узлами B, C, D дерева решений:

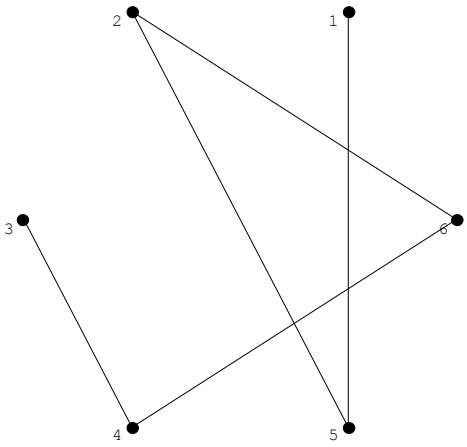



На этом рисунке узел A представляет первоначальную задачу, а узлы B, C и D отвечают задачам, матрицы весов которых те же самые, что и матрица весов задачи A, но только ребрам {2, 1}, {2, 6}или {2, 5} соответственно приписан бесконечный вес.
Последовательно решаем задачи B, C, D. Каждая из этих задач заключается в нахождении минимального остова с дальнейшим анализом полученного остова.
Для задачи B корректируем вес ребра {2, 1}

```
In[8]:= c21 = c; c21[[2, 1]] = Infinity; c21[[1, 2]] = Infinity;
        g = FromAdjacencyMatrix[c21, EdgeWeight];
```

и строим минимальный остов

```
In[9]:= ShowLabeledGraph[ost = MinimumSpanningTree[g]];
```



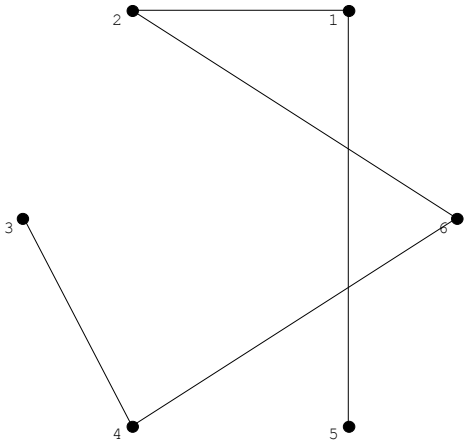
```
In[10]:= Degrees[ost]

Out[10]= {1, 2, 1, 2, 2, 2}
```

По степеням вершин, и из того что остов является связным графом, заключаем что он является гамильтоновой цепью стоимости:

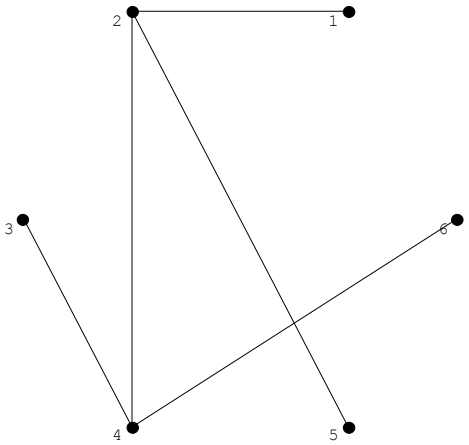
```
In[11]:= CostOfPath[g, {1, 5, 2, 6, 4, 3}]

Out[11]= 23
```

Аналогично для задач C и D получаем

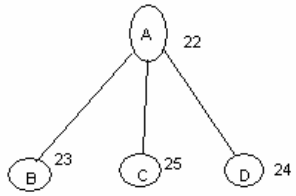
```
In[12]:= c25 = c; c25[[2, 5]] = Infinity; c25[[5, 2]] = Infinity;
         g = FromAdjacencyMatrix[c25, EdgeWeight];

In[13]:= ShowLabeledGraph[ost = MinimumSpanningTree[g]];
```




```
In[14]:= CostOfPath[g, {5, 1, 2, 6, 4, 3}]

Out[14]= 25

In[15]:= c26 = c; c26[[2, 6]] = Infinity; c26[[6, 2]] = Infinity;
         g = FromAdjacencyMatrix[c26, EdgeWeight];

In[16]:= ShowLabeledGraph[ost = MinimumSpanningTree[g]];
```



```
In[17]:= CostOfPath[g, {1, 2, 4, 3}] + CostOfPath[g, {2, 5}] + CostOfPath[g, {4, 6}]

Out[17]= 24
```




При ветвлении из вершин из вершины A на вершину B мы сразу получили решение стоимости 23. При выборе вершины C получилось решение стоимости 25. Ветвление из вершины D можно не производить, так как нижняя граница стоимостей возможных решений равна 24, что больше полученного ранее решения стоимости 23. Этот этап называется отсечением ветвей, что позволяет существенно сократить перебор возможных решений.

### *8.48. Замкнутая задача коммивояжера.*

Найти минимальный гамильтонов цикл (тур) в полном ориентированном графе с заданной матрицей весов ребер (матрицей стоимостей).

**Решение задачи методом ветвей и границ**. Решаем аналогично предыдущей задаче. В этой задаче применяется другая процедура ветвления и другой метод вычисления нижних границ.
Пусть R множество всех (n-1)! возможных туров, T – текущее множество. Вначале T = R. В процессе поиска минимального тура множество T разбивается на два подмножества Y{k, l} и $\underline{Y}${k, l}, определяемые выбором ребра (k, l). Во множество Y{k, l} входят все туры из T, содержащие ребро (k, l), а во множество $\underline{Y}${k, l} – туры, не содержащие это ребро.
Для каждого из этих двух подмножеств вычисляется нижняя граница, гарантирующая, что стоимость каждого тура подмножества больше или равна нижней границе.
После этого разделения из всех получившихся подмножеств выбирается подмножество, которое будет текущим, и процесс деления повторяется до тех пор, пока одно из вновь получившихся подмножеств не будет состоять из одного тура.
Если стоимость этого тура меньше или равна нижней границе некоторого подмножества, к которому еще не применена процедура деления, то это подмножество исключается из поиска. Иначе в качестве текущего подмножества выбирается некоторое подмножество туров с нижней границей, меньшей, чем стоимость получившегося тура, и поиск продолжается, начиная с этого подмножества. Если такого подмножества выбрать нельзя, то поиск минимального тура завершен.
Выбор текущего подмножества и метод его деления на два подмножества называется процедурой ветвления.
Вычисление нижней границы подмножеств в данной задаче производится процедурой приведения матрицы стоимостей. Эта процедура основана на следующих двух соображениях:

1. В терминах матрицы стоимостей каждый тур содержит только один элемент (ребро и соответствующую стоимость) из каждого столбца и каждой строки матрицы. Заметим, что обратное утверждение не всегда верно – множество, содержащее один и только один элемент из каждой строки и из каждого столбца, не обязательно представляет тур.
2. Если вычесть константу h из каждого элемента строки или столбца матрицы стоимостей, то стоимость любого тура новой матрицы будет ровно на h меньше. Поскольку

любой тур должен содержать ребро из данной строки или данного столбца, стоимость всех туров уменьшается на h. Это вычитание называется приведением строки (или столбца) матрицы стоимостей. Пусть t – оптимальный тур при матрице стоимостей C, z(t) – его стоимость.

Если матрица C` получается из матрицы C приведением строки (или столбца), то тур t должен оставаться оптимальным туром и в матрице C`, при этом стоимость туров связана соотношением

$$z(t) = h + z`(t)$$

где z`(t) – стоимость тура t в матрице C`.
Под процедурой приведения всей матрицы стоимостей понимается следующее: мы последовательно проходим строки и вычитаем значение наименьшего элемента $h_i$ каждой строки из каждого элемента этой строки. Затем аналогичные действия выполняются для каждого столбца. Если для некоторого столбца или строки значение $h_i = 0$, то рассматриваемый столбец или строка уже приведены и мы переходим к следующему столбцу или строке матрицы стоимостей.
Сумма h всех $h_i$ будет нижней границей стоимости всех туров из множества R.
Полученную в результате матрицу стоимостей назовём приведенной из C. Каждой вершине поискового дерева сопоставляется своя приведенная матрица стоимости.
Приведем пример для полного ориентированного графа, матрица стоимостей которого приведена ниже:

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| 1 | ∞ | 25 | 40 | 31 | 27 | $h_1 = 25$ |
| 2 | 5 | ∞ | 17 | 30 | 25 | $h_2 = 5$ |
| 3 | 19 | 15 | ∞ | 6 | 1 | $h_3 = 1$ |
| 4 | 9 | 50 | 24 | ∞ | 6 | $h_4 = 6$ |
| 5 | 22 | 8 | 7 | 10 | ∞ | $h_5 = 7$ |
| | $h_6 = 0$ | $h_7 = 0$ | $h_8 = 0$ | $h_9 = 3$ | $h_{10} = 0$ | |

Общее приведение составляет h = 25 + 5 + 1 + 6 + 7 + 3 = 47, следовательно, нижняя граница стоимости любого тура из R равна 47.
Следующий вопрос – как вычисляются нижние границы для некорневых вершин дерева решений. Предположим, что ветвление происходит по ребру (3, 5)
По определению, ребро (3, 5) содержится в каждом туре множества Y{3, 5}. Этот факт препятствует выбору ребра (5 ,3), так как рёбра (3, 5) и (5, 3) образуют цикл, а это не допускается ни для какого тура. Поэтому ребро (5, 3) исключаем из рассмотрения, положив значение $c_{53} = \infty$. Строку 3 и столбец 5 также можно исключить из дальнейшего рассмотрения по отношению к множеству туров Y{3 ,5}, потому что уже есть ребро из вершины 3 в вершину 5. Часть матрицы стоимостей, которая будет необходима для дальнейшего поиска на множестве туров {3, 5}, показана в табл. 1. Она может быть приведена к матрице стоимостей, показанной в табл. 2 со значением h = 15. Теперь нижняя граница для любого тура из множества Y{3,5} равна 47+15 = 62, что указано около вершины Y{3,5} на рис.
В дальнейшем при формировании матрицы стоимостей для множества Y(k, l), чтобы исключить возврат в начальную вершину, следует положить также $c_{sl} = \infty$, где s – начальная вершина тура.

**Матрица стоимостей**
для множества {3,5}

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | ∞ | 0 | 15 | 3 |
| 2 | 0 | ∞ | 12 | 22 |
| 4 | 3 | 44 | 18 | ∞ |
| 5 | 5 | 1 | ∞ | 0 |

**Приведённая матрица стоимостей**
для множества {3,5}

| | 1 | 2 | 3 | 4 | |
|---|---|---|---|---|---|
| 1 | ∞ | 0 | 3 | 3 | 0 |
| 2 | 0 | ∞ | 0 | 22 | 0 |
| 4 | 0 | 41 | 3 | ∞ | $h_3$=3 |
| 5 | 15 | 1 | ∞ | 0 | 0 |
| | 0 | 0 | h=12 | 0 | |

Нижняя граница для множества $\underline{Y}\{3,5\}$ получается несколько иным способом. Ребро (3,5) не может находиться в этом множестве, поэтому полагаем $c_{35} = \infty$. В любой тур из множества $\underline{Y}\{3,5\}$ будет входить какое-то ребро из вершины 3 и какое-то ребро к вершине 5. Самое дешевое ребро из вершины 3, исключая старое значение (3,5), имеет стоимость 2, а самое дешевое ребро к вершине 5 имеет стоимость 0. Следовательно, нижняя граница любого тура во множестве Y{3,5} равна 47+2+0 = 49.

При приведении матрицы стоимости для множества Y нам удалось сократить её размер. Очевидно, что это сокращение будет происходить каждый раз для множества Y, так что мы значительно сокращаем при этом вычислительные затраты. Ещё один вывод состоит в том, что если мы сможем найти тур из множества $\underline{Y}$ со стоимостью, меньшей или равной 62, то тогда мы отбрасываем вершину поискового дерева решений Y и в этом случае будем говорить, что вершина Y в дереве отработана. Тогда следующей целью может быть ветвление из вершины $\underline{Y}$ в надежде найти тур со стоимостью в пределах $49 \le Ct \le 62$.

Какое ребро выбрать при делении текущего множества?

Эвристика для выбора ребра. Ребро (k,l) для обеспечения сокращения перебора нужно выбирать так, чтобы попытаться получить большую по величине нижнюю границу на множестве $\underline{Y}$, что облегчит проведение оценки для множества Y. Таким образом, алгоритм выбирает на каждом шаге множество Y, чтобы решить задачу за n шагов. Как применить эти идеи к выбору конкретного ребра ветвления (k,l) ? В приведенной матрице стоимостей $C'$, связанной с вершиной X , каждая строка и столбец имеют хотя бы по одному нулевому элементу (если это не так, то матрица $C'$ не полностью приведена ) . Можно предположить, что ребра, соответствующие этим нулевым стоимостям, будут с большей вероятностью входить в оптимальный тур, чем ребра с большими стоимостями. Поэтому в качестве ребра ветвления выбирается одно из них, при этом нужно, чтобы у множества $\underline{Y}$ ={k,l} была как можно большая нижняя граница. Пусть ребро (k,l) имеет $c_{kl}$=0, и, обращаясь к процедуре вычисления нижней границы, мы видим, что для множества $\underline{Y}$ эта граница задается в виде:

$$w\left\{\bar{Y}\right\} = w(X) + \min_{i,i\neq k} c_{il} + \min_{j,j\neq l} c_{k,j}$$

Следовательно, из всех ребер (k,l) , у которых $c_{kl}$=0 в текущей матрице $C'$, мы выбираем то, которое дает наибольшее значение для нижней границы – w($\underline{Y}$).

Эвристика для выбора текущей вершины. Теперь нужно выбрать следующую вершину X, от которой необходимо провести ветвление. Этот выбор довольно очевиден. Мы выбираем ту вершину, которая имеет в данный момент наименьшую нижнюю границу, и из которой в данный момент выходят ветви, т.е. – это лист поискового дерева решений с минимальной нижней границей.

Продемонстрируем решение задачи. Построим приведенную матрицу, соответствующую вершине R:

$$\begin{pmatrix} \infty & 0 & 15 & 3 & 2 \\ \mathbf{0} & \infty & 12 & 22 & 20 \\ 18 & 14 & \infty & 2 & 0 \\ 3 & 44 & 18 & \infty & 0 \\ 15 & 1 & 0 & 0 & \infty \end{pmatrix}$$

Согласно эвристике выбора ребра, выбираем ребро (2, 1). При этом нижняя граница множества $\underline{Y}\{2,1\}$ получается равной 47 + (12 + 3) = 62.
Строим приведенную матрицу для множества Y{2, 1}

$$\begin{pmatrix} \infty & 15 & 3 & 2 \\ 14 & \infty & 2 & 0 \\ 44 & 18 & \infty & 0 \\ 1 & 0 & 0 & \infty \end{pmatrix} \qquad \begin{pmatrix} \infty & 13 & 1 & 0 \\ 13 & \infty & 2 & 0 \\ 43 & 18 & \infty & \mathbf{0} \\ 0 & 0 & 0 & \infty \end{pmatrix}$$

Общее приведение равно 2 + 1 = 3. Следовательно, нижняя граница для множества Y{2, 1} равна 47 + 3 = 50.
Согласно эвристике выбора текущей вершины, выбираем вершину, соответствующую множеству Y{2, 1}. Далее, делим это множество по ребру (4, 5). Нижняя граница множества $\underline{Y}\{4,5\}$ получается равной 50 + (18 + 0) = 68. Для множества Y{4, 5} получим приведенную матрицу

$$\begin{pmatrix} \infty & 13 & 1 \\ 13 & \infty & 2 \\ 0 & 0 & \infty \end{pmatrix} \qquad \begin{pmatrix} \infty & 12 & \mathbf{0} \\ 11 & \infty & 0 \\ 0 & 0 & \infty \end{pmatrix}$$

Приведение равно 1 + 2 =3, и нижняя граница получается равной 50 + 3 =53. В качестве текущей выбираем вершину, соответствующую множеству Y(4,5) и делим его по ребру (1, 4). Для множества $\underline{Y}$(1,4) нижняя граница будет равна 53 +12=65. Для множества Y(1,4) матрица стоимостей:

$$\begin{pmatrix} 11 & \infty \\ 0 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & \infty \\ 0 & 0 \end{pmatrix}$$

Приведение равно 11 и нижняя граница составляет 53 + 11 = 64. Множество Y{1,4} состоит из одного тура {(2, 1), (1, 4), (4, 5), (5, 3), (3, 2)} стоимости 5 + 31 + 6 + 7 + 15 = 64.
Из дальнейшего поиска будут исключены вершины с большими нижними границами.
Теперь текущей вершиной будет вершина , соответствующая множеству $\underline{Y}$(2,1), с нижней границей, равной 62 и матрицей стоимостей:

$$\begin{pmatrix} \infty & 0 & 15 & 3 & 2 \\ \infty & \infty & 0 & 10 & 8 \\ 15 & 14 & \infty & 2 & 0 \\ \mathbf{0} & 44 & 18 & \infty & 0 \\ 12 & 1 & 0 & 0 & \infty \end{pmatrix}$$

Она делится по ребру (4, 1). Нижняя граница вершины $\underline{Y}\{4, 1\}$ равна 62 + 12 =74, а нижняя граница вершины Y{4, 1} остается равной 62:

$$\begin{pmatrix} 0 & 15 & \infty & 2 \\ \infty & \mathbf{0} & 10 & 8 \\ 14 & \infty & 2 & 0 \\ 1 & 0 & 0 & \infty \end{pmatrix}$$

Деление этой вершины по ребру (2, 3) приводит к вершинам с нижними границами, равными 80.
Поиск завершен. Найденный тур – {1, 4, 5, 3, 2, 1}.
Функция Mathematica TravelingSalesman[g] работает только на неориентированных графах.

## Некоторые нерешенные задачи

### *Близнецы*

Пары (2, 3), (3, 5), (5, 7),(11, 13), (17, 19), (29, 31), … называются близнецами.
Вопрос: сколько близнецов?

### *Задача*

Пусть m – некоторое натуральное число и $(a_s, \ldots, a_1)$ – его десятичная запись. Пусть преобразование T переводит число m в произведение его десятичных цифр: $T(m) = a_1 \cdot \ldots \cdot a_s$. После этого к полученному числу T(m) может быть применено то же самое преобразование. Ясно, что после некоторого числа шагов v(m) мы получим либо ноль, либо число, не превосходящее девяти.

Вопрос: существует ли такая константа c, что $v(m) < c$ для всех m?

### *Задача*

Пусть m – некоторое натуральное число. Пусть преобразование T переводит число m в m/2, если m четно, и в 3m+1, если m нечетно. После этого к полученному числу T(m) может быть применено то же самое преобразование. На некотором шаге мы можем вернуться к ранее полученному числу, тем самым попадая в цикл.

Вопрос: существует ли такое m, для которого последовательность преобразований не зацикливается?

**Примечание**: за решение можете получить приз в 60$.

### *Совершенные числа*

Найти все натуральные числа, которые равны сумме своих делителей. Например:

$6 = 1 + 2 + 3,$
$28 = 1 + 2 + 4 + 7 + 14,$
$496 = 1 + 2 + 4 + 8 + 16 + 31 + 62 + 124 + 248.$

Найдите хотя бы одно нечетное совершенное число.

### *Задача Грэхема*

Пусть A – произвольное множество из n натуральных чисел.

Вопрос: верно ли неравенство $\max(a/НОД(a,b)), a,b \in A) \geq n$?

## Некоторые алгоритмы арифметики и теории чисел.

### Алгоритм Евклида вычисления НОД(a,b) для a, b ≥0

1. Если b = 0, то Result := a и алгоритм заканчивает свою работу.
2. Положить r := a mod b, a := b, b := r и вернуться на шаг 1.

### Бинарный алгоритм Евклида вычисления НОД(a,b) для a, b ≥0

Следующие свойства целых чисел и наибольшего общего делителя помогают улучшить алгоритм Евклида:

(а) если числа a b и четные, то НОД(a,b) = НОД(a/2,b/2) · 2,
(б) если a четное и b нечетное, то НОД(a,b) = НОД(a/2,b),
(в) НОД(a,b) = НОД(a-b,b),
(г) если числа a и b нечетные, то |a-b| четное и a-b < max(a,b).

1. Если a < b, то поменять местами a и b. Если b = 0, то Result :=a и выход. Иначе k := 0 и, пока числа a и b оба четные, k:=k+1; a := a div 2; b := b div 2.

2. Пока a четное, повторять a := a div 2. Или пока b четное, повторять b := b div 2.

3. t := (a-b)/2. Если t = 0, то Result := $2^k$a и завершить работу алгоритма.

4. Пока t четное, повторять t := t div 2. Если t > 0, то a := t, иначе b := -t. Вернуться на шаг 3.

**Вычисление НОД($n_1$,…, $n_r$)** рекурсивно сводится к случаю двух аргументов:

$$НОД(n_1,\ldots, n_r) = НОД(n_1, НОД(n_2,\ldots, n_r)).$$

**Вычисление НОК(a, b**) происходит из соотношения:

$$НОК(a,b) \cdot НОД(a,b) = |ab|.$$

**Вычисление НОК($n_1$,…, $n_r$)** рекурсивно сводится к случаю двух аргументов:

$$НОК(n_1,\ldots, n_r) = НОК(n_1, НОК(n_2,\ldots, n_r)).$$

Справедлив дистрибутивный закон:

$$НОД(a, НОК(b, c)) = НОК(НОД(a, b), НОД(a, c)).$$
$$НОК(a, НОД(b, c)) = НОД(НОК(a, b), НОК(a, c)).$$

### Расширенный алгоритм Евклида вычисления НОД(a,b) и таких чисел U и V, что НОД(a, b) = U·a + V·b, для a, b ≥ 0

1. Положить U := 1; d := a. Если b = 0, то V := 0 и завершить алгоритм, иначе $v_1$:= 0; $v_3$:= b.

2. t3 := d mod $v_3$; q := d div $v_3$; t1:= U – q·$v_1$; U:=$v_1$; d := $v_3$; $v_1$ := $t_1$; $v_3$:=$t_3$.

3. Если $v_3 = 0$, то V :=(d - Ua)/b и завершить алгоритм, иначе вернуться на шаг 2.

**Алгоритм вычисления целой части r квадратного корня из натурального числа n ≥0**

1. $x := \lfloor 2^{(e+2)/2} \rfloor$, где $e := \lfloor \log_2 n \rfloor$.

2. $y := \lfloor (x + c/x)/2 \rfloor$.

3. Если y < x, то x := y и вернуться на шаг 2. Иначе результат в x и алгоритм завершен.

**Алгоритм, определяющий, является ли целое число n > 0 полным квадратом. Если да, то результат алгоритма – квадратный корень из числа n.**

0. Строятся четыре массива: q11, q63, q64 и q65:
   q11[k] := 0 для k = 0, …, 10; q11[$k^2$ mod11] := 1 для k = 0, …, 5;
   q63[k] := 0 для k = 0, …, 62; q63[$k^2$ mod11] := 1 для k = 0, …, 31;
   q64[k] := 0 для k = 0, …, 63; q64[$k^2$ mod11] := 1 для k = 0, …, 31;
   q65[k] := 0 для k = 0, …, 64; q65[$k^2$ mod11] := 1 для k = 0, …, 32;

1. Положить t := n mod 64. Если q64[t], то число n не является полным квадратом и алгоритм заканчивает работу. Иначе положить r := n mod (11·63·65).

2. Если q63[r mod 63], то число n не является полным квадратом и алгоритм заканчивает работу.

3. Если q65[r mod 65], то число n не является полным квадратом и алгоритм заканчивает работу.

4. Если q11[r mod 11], то число n не является полным квадратом и алгоритм заканчивает работу.

5. $q := \lfloor sqrt(n) \rfloor$; если q2 ≠ n, то число n не является полным квадратом и алгоритм заканчивает работу. Иначе n является полным квадратом и q – квадратный корень из n.

На первый взгляд, этот алгоритм может показаться странным: что за константы 11, б3, 64, 65? Внесем ясность: если целое число n является квадратом целого числа, то оно будет полным квадратом и по модулю произвольного целого числа m. Здесь используется обратное утверждение: если n не является квадратом по модулю m, то оно не является квадратом и в целых числах. Всего существует 12 квадратов по модулю 64, 16 квадратов по модулю 63, 21 квадрат по модулю 65 и 6 квадратов по модулю 11, то есть вероятность пропустить с 1 – 4 шагов алгоритма на 5-й шаг число, не являющееся полным квадратом, равна

(1 - 52/64)(1 - 47/63)(1 - 44/65)(1 - 5/11) = 6/715

**Алгоритм поиска всех простых чисел, меньших или равных натуральному числу N (решето Эратосфена)**

1. Положить $L := \lfloor (N-1)/2 \rfloor$; $B := \lfloor \sqrt{N}/2 \rfloor$. Для $1 \le i \le L$ положить $f_i := 1$. Положить i := 1; p := 3; s:= 4.
2. Если $f_i = 0$, то перейти к шагу 4. Иначе печатать p; k := s.

3. При $k \le L$ положить $f_k := 0$; $k := k + p$ и повторить этот шаг.
4. Если $i \le B$, то положить $I := I + 1$; $s := s + 2p$; $p := p + 2$ и вернуться на шаг 2. Иначе закончить алгоритм.

Число $(1041870^{32768} - 1)$ – простое, 2000 г. В ноябре 2001 г. было найдено 39-е простое число Мерсенна – $(2^{13466917} - 1)$ http://www.mersenne.org

### Бинарный алгоритм вычисления $a^e$ по модулю m

0. Пусть $(e_{n-1}e_{n-2}\ldots e_0)$, где старший разряд $e_{n-1} > 0$ есть двоичное представление показателя e.
1. $p := a^{e_{n-1}}$ ; $i := n-2$.
2. $p := p^2 v \bmod m$.
3. Если $e_i = 1$, то $p := p \cdot a \bmod m$.
4. $i := i-1$, при $i \ge 0$ вернуться на шаг 2.
5. Результат := p.

### Вычисление коэффициентов призведения биномов

$$(a_1 \cdot x + a_0) \cdot (b_1 \cdot x + b_0) = c_2 \cdot x^2 + c_1 \cdot x + c_0$$

1. $c_0 := a_0 \cdot b_0$, $c_2 := a_2 \cdot b_2$, $c_1 := (a_1 + a_0) \cdot (b_1 \cdot x + b_0) - c_2 - c_1$.

### Вычисление значения полинома

$$s = a_0 \cdot x_n + a_1 \cdot x_{n-1} + \ldots + a_n$$

1. $s := a0$, $i := 0$.

2. Если $i = n$, то выход, иначе $i := i + 1$, $s := s \cdot x + a \cdot i$ и повторить шаг 2.

### Китайская теорема об остатках

Пусть натуральные числа $m_1, \ldots, m_r$ попарно взаимно просты и числа $a_1, \ldots, a_r$ – произвольные целые. Тогда существует решение системы сравнений $x \equiv a_i \bmod m_i$, причем это решение единственно по модулю произведения $m_1 \cdot m_2 \ldots m_r$.

*Доказательство.* $M := m_1 \cdot m_2 \ldots m_r$; $M_j := m/m_j$. Тогда НОД$(M_j, m_j) = 1$ и расширенным алгоритмом Евклида находим целые числа $U_j$ и $V_j$ такие, что $M_j \cdot U_j + m_j \cdot V_j = 1$, то есть $M_j \cdot U_j \equiv 1 \bmod m_j$.
Сумма $x_0 := \sum M_j \cdot U_j \cdot a_j$ есть решение.

# Упражнения

1. Квадрат графа g =( E, V) есть граф $G^2$ =($E^2$,V), где {u,v}$\in E^2$ тогда и только тогда, когда существует путь длины 2 между u и v в G. Описать алгоритм для построения квадрата данного графа, используя
1. список смежности
2. матрицу смежности
3. бинарное отношение

2. Найти связь между матрицей инцидентности графа G и матрицей смежности реберного графа L(G).

3. Используя функции для представления графов, определенные в этой главе, сформировать несколько различных функций для конструирования полных графов.

4.Построить булеву функцию для тестирования планарности графа, проверяя, пересекаются ли любые два ребра.

5. Показать, что дополнение несвязного графа может быть связным.

6. Показать, что граф не имеет циклов нечетной длины тогда и только тогда, когда он двудольный.

7. Доказать, что любая вершина, принадлежащая более чем одной связной компоненте есть точка сочленения.

8. Доказать, что 3-регулярный граф содержит мост тогда и только тогда , когда он содержит точку сочленения.

9. Доказать, что любой слабо связный граф, все вершины которого имеют одинаковую входящую и выходящую степени, является сильно связным

10. Доказать, что операция произведения графов коммутативна, то есть граф G×H изоморфен H ×G.

11. Является ли произведение двух гамильтоновых (эйлеровых) графов гамильтоновым (эйлеровым) графом? Доказать.

12. Доказать, что реберный граф эйлерова графа является и эйлеровым и гамильтоновым, а реберный граф гамильтонова графа всегда гамильтонов.

13. Доказать, что центр дерева состоит по крайней мере из двух смежных вершин.

14. Начертить двудольный граф для отношения a~ A, где A пробегает все подмножества множества V из 3 или 4 элементов

15. Построить матрицы смежности и инцидентности для графов правильных многогранников.

16. Определить радиусы, диаметры и эксцентриситеты для графов правильных многогранников.

17. Найти степени и числа вершин для графов правильных многогранников.

18. Может ли звезда быть реберным графом другого графа?

19. Каково наименьшее число ребер в связном графе  с n вершинами?

20 .Доказать, что если граф G связен, то связен и реберный граф L(G).

21. На шахматной доске помесить короля, ладью пешку в некоторую позицию и определить, существует ли последовательность ходов, содержащая все возможные переходы по одному разу.

22. Когда реберный граф данного графа имеет эйлеров граф?

23. Определить реберную и вершинную связности  для графов правильных многогранников, звезды на вершинах, колеса на n вершинах, решетчатого m×n графа.

24. Найти хроматическое число для графов правильных многогранников, звезды на вершинах, колеса на n вершинах, решетчатого m×n графа.

# Список встроенных функций пакета <<DiscreteMath`Combinatorica`

## ▪ AcyclicQ
AcyclicQ[g] возвращает True, если граф g ациклический. См. С.106

## ▪ AddEdge
AddEdge[g, e] возвращает граф g с новым добавленным ребром e, где e - {a, b} или {{a, b}, options}. См. С.134

## ▪ AddEdges
AddEdges[g, edgeList] возвращает граф g с новыми добавленными ребрами из списка edgeList, имеющего форму {a, b} или {{a, b}, {c, d}, ...}, или { {{a, b}, x}, {{c, d}, y}, ...}, где x и y -опции соответствующих ребер. См. С.134

## ▪ AddVertex
AddVertex[g] добавляет одну изолированную вершину к графу g.
AddVertex[g, v] добавляет к g вершину с координатами, заданными вектором v. См. С.136

## ▪ AddVertices
AddVertices[g, n] добавляет n изолированных вершин к графу g.
AddVertices[g, vList] добавляет вершины из списка vList к g. vList - {x, y} или {{x1, y1}, {x2, y2}...}, или {{{x1, y1}, g1}, {{x2, y2}, g2},...}, {x, y}, где
{x1, y1}, и {x2, y2} - координаты точек, g1 и g2 - опции вершин. См. С.136

## ▪Algorithm
Algorithm - опция таких функций как ShortestPath, VertexColoring, VertexCover для указания алгоритма их вычисления. См. С.201, 204, 218, 221

## ▪ AllPairsShortestPath
AllPairsShortestPath[g] возвращает матрицу, где (i, j)-й элемент есть длина кратчайшего пути в g между вершинами i и j.
AllPairsShortestPath[g, Parent] возвращает матрицу размерностью 2 * V[g] * V[g], (1, i, j)-й элемент которой есть длина кратчайшего пути от i до j и (2, i, j)-й элемент есть предшественник j в кратчайшем пути от i до j. См. С.224

## ▪ AlternatingGroup
AlternatingGroup[n] генерирует множество четных перестановок множества {1,2,...,n}, группу An.
AlternatingGroup[l] генерирует множество четных перестановок множества l. См. С.55, 330

## ▪ AlternatingGroupIndex
AlternatingGroupIndex[n, x] возвращает циклический индекс знакопеременной группы An как полином от x. См. С.61

## ▪ AlternatingPaths
AlternatingPaths[g, start, ME] возвращает дополняющие пути в графе g, относительно паросочетания ME, начинающиеся в вершинах списка start в виде леса корневых деревьев. См. С.196

## ▪ AnimateGraph
AnimateGraph[g, l] изображает граф g с последовательно подсвеченными элементами списка l, содержащим вершины и ребра g. Флаг опций, принимающий значения All и One, указывает, будут ли ранее подсвеченные элементы гаснуть или нет. По умолчанию флаг All, т. е. подсветка сохраняется. См. С.92

## ▪ AntiSymmetricQ
AntiSymmetricQ[g] возвращает True, если матрица смежности графа g
представляет антисимметричное бинарное отношение. См. С. 132

## ▪ Approximate
Approximate - значение опции Algorithm в таких функциях, как VertexCover, для использования приближенного алгоритма. См. С.201

## ▪ ApproximateVertexCover
ApproximateVertexCover[g] строит вершинное покрытие графа g, используя приближенный алгоритм. См. С.204

## ▪ ArticulationVertices
ArticulationVertices[g] возвращает список точек сочленения графа g. Точка сочленения, это вершина графа g, удаление которой приводит к увеличению числа связных компонент. См. С.183

## ▪ Automorphisms
Automorphisms[g] строит группу автоморфизмов графа g. См. С.256

## ▪ Backtrack
Backtrack[s, partialQ, solutionQ] выполняет поиск в глубину в пространстве состояний s, расширяя частичное решение до тех пор, пока выполняется условие partialQ.

## ▪ BellB
BellB[n] возвращает n-е число Белла. См. С.72

## ▪ BellmanFord
BellmanFord[g, v] строит дерево кратчайших путей от вершины v до всех вершин графа g. Это дерево представляется списком, i-й элемент которого есть предшественник вершины i в дереве. Функция BellmanFord возвращает корректный результат и для отрицательных весов ребер, при отсутствии в графе циклов отрицательной длины. См. С.218

## ▪ BiconnectedComponents
BiconnectedComponents[g] возвращает список двусвязных компонент графа g. См. С.183

## ▪ BiconnectedQ
BiconnectedQ[g] возвращает True, если граф g двусвязный. См. С.183

## ▪ BinarySearch
BinarySearch[l, k] ищет в отсортированном списке l ключ k и возвращает позицию в l, содержащую k, если k есть в l. Если ключа k нет в l и k попадает между элементами в позициях p и p+1, то возвращается (p + 1/2). См. С.38

## ▪ BinarySubsets
BinarySubsets[l] возвращает все подмножества множества l в порядке возрастания характеристического вектора. Для каждого n>0 BinarySubsets[n] возвращает все подмножества множества {1, 2,.., n}. См. С.47

## ▪ BipartiteMatching
BipartiteMatching[g] возвращает список ребер максимального паросочетания двудольного графа g. Для взвешенного графа функция возвращает паросочетание максимального веса. См. С.197

## ▪ BipartiteMatchingAndCover
BipartiteMatchingAndCover[g] возвращает паросочетание максимального веса вместе с двойственным вершинным покрытием двудольного графа g. Если граф невзвешенный, то веса ребер полагаются равными 1. См. С.204

## ▪ BipartiteQ
BipartiteQ[g] возвращает True, если граф g двудольный. См. С.102,208

## ▪ BooleanAlgebra
BooleanAlgebra[n] возвращает диаграмму Хассе булевой алгебры n элементов. Функция имеет две опции: Type и VertexLabel, с значениями по умолчанию Undirected и False соответственно. Когда Type -> Directed, функция строит ориентированный ациклический граф. Если VertexLabel -> True, вершины помечаются. См. С.242

## ▪ Box
Box есть значение опции VertexStyle функции ShowGraph. См. С.84

## ▪ BreadthFirstTraversal
BreadthFirstTraversal[g, v] выполняет поиск в ширину в графе g, начиная с вершины v, и возвращает номера вершин в порядке обхода.
BreadthFirstTraversal[g, v, Edge] возвращает ребра графа, проходимые при поиске.
BreadthFirstTraversal[g, v, Tree] возвращает дерево поиска.
BreadthFirstTraversal[g, v, Level] возвращает уровень вершины в дереве поиска. См. С.165

## ▪ Brelaz
Brelaz есть значение опции Algorithm функции VertexColoring. См. С.205

## ▪ BrelazColoring
BrelazColoring[g] возвращает раскраску, при которой вершинам присваивается цвет с наименьшим возможным номером, в порядке убывания степеней вершин. См. С.205

## ▪ Bridges
Bridges[g] возвращает список мостов графа g, где мост – это ребро, удаление которого приводит к увеличению числа связных компонент. См. С.187

## ▪ ButterflyGraph
ButterflyGraph[n] возвращает ориентированный граф, вершины которого есть пары (w, i), где w есть двоичная строка длины n, i - целое число от 0 до n, и ребра которого соединяют вершину (w, i) с вершиной (w', i+1), если w' равна w во всех битах с возможным неравенством в (i+1)-м бите. Допускается опция VertexLabel, по умолчанию имеющая значение False. Если VertexLabel -> True, то вершины помечаются строкой (w, i). См. С.151

## ▪ CageGraph
CageGraph[k, r] возвращает наимньший k-регулярный граф обхвата r
для некоторых малых значений k и r. CageGraph[r] возвращает CageGraph[3, r]. Для k = 3, r может быть 3, 4, 5, 6, 7, 8 или 10. Для k = 4 или 5 r может быть 3, 4, 5 или 6. См. С.228

## ▪ CartesianProduct
CartesianProduct[l1, l2] возвращает декартово произведение списков l1 и l2. См. С.157

## ▪ Center
Center есть значение опций VertexNumberPosition, VertexLabelPosition, и EdgeLabelPosition функции ShowGraph. См. С.84

## ▪ ChangeEdges
ChangeEdges[g, e] заменяет ребра графа g ребрами списка e. Здесь e - список {{s1, t1}, {s2, t2}, ...} или {{{s1, t1}, gr1}, {{s2, t2}, gr2}, ...}, где {s1, t1}, {s2, t2}, ... ребра и gr1, gr2, ... есть графические опции ребра. См. С.132

## ▪ ChangeVertices
ChangeVertices[g, v] заменяет вершины графа g вершинами списка v. Здесь v - список {{x1, y1}, {x2, y2}, ...} или {{{x1, y1}, gr1}, {{x2, y2}, gr2}, ...}, где {x1,
y1}, {x2, y2}, ... координаты вершин и gr1, gr2, ... есть графические опции вершины. См. С.132

## ▪ ChromaticNumber
ChromaticNumber[g] возвращает хроматическое число графа g, т. е. наименьшее количество цветов для правильной раскраски вершин графа. См. С.205

## ▪ ChromaticPolynomial
ChromaticPolynomial[g, z] возвращает хроматический полином P(z) графа g, определяющий количество вариантов раскраски. См. С.210

## ▪ ChvatalGraph
ChvatalGraph возвращает наименьший, не содержащий треугольников,
4-регулярный, 4-хроматический граф Хватала. См. С.207

## ▪ CirculantGraph
CirculantGraph[n, l] строит граф-циркулянт с n вершинами, в котором вершина i смежна с (i+j)-ой и (i-j)-й вершинами для всех j в списке l. См. С.105

## ▪ CircularEmbedding
CircularEmbedding[n] строит n точек равномерно расположенных по окружности. CircularEmbedding[g] строит такое же вложение графа g. См. С.139

## ▪ CircularVertices
CircularVertices[n] и CircularVertices[g] эквивалентны функциям CircularEmbedding. См. С.139

## ▪ CliqueQ
CliqueQ[g, c] возвращает True, если вершины списка c образуют клику в графе g. См. С.213

## ▪CoarserSetPartitionQ
CoarserSetPartitionQ[a, b] возвращает True, если разбиение множества b грубее, чем разбиение множества a, т. е. каждый блок из разбиения a содержится в некотором блоке из b. См. С.74

## ■ CodeToLabeledTree
CodeToLabeledTree[l] по коду Прюфера l строит помеченное дерево с n вершинами. Код Прюфера l есть список из n-2 целых чисел от 1 до n. См. С.108

## ■ Cofactor
Cofactor[m, {i, j}] вычисляет алгебраическое дополнение элемента(i, j) матрицы m. См.233

## ■ CompleteBinaryTree
CompleteBinaryTree[n] возвращает полное двоичное дерево из n вершин. См. С.109

## ■ CompleteGraph
CompleteGraph[n] строит полный n-вершинный граф. Можно указать опцию Type со значениями Directed или Undirected. Значение по умолчанию есть Type -> Undirected. См. С.100

## ■ CompleteKaryTree
CompleteKaryTree[n, k] возвращает полное k-арное дерево с n вершинами. См. С.109

## ■ CompleteKPartiteGraph
CompleteKPartiteGraph[a, b, c, ...] создает полный k-дольный граф. Можно указать опцию Type со значениями Directed или Undirected. Значение по умолчанию есть Type -> Undirected. См. С.102

## ■ CompleteQ
CompleteQ[g] возвращает True, если граф g полный. См. С.100

## ■ Compositions
Compositions[n, k] возвращает список всех композиций целого числа n из k частей. См. С.69

## ■ ConnectedComponents
ConnectedComponents[g] возвращает вершины графа g, разбитые на связные компоненты. См. С.97

## ■ ConnectedQ
ConnectedQ[g] возвращает True, если неориентированный граф g связен.
Если g ориентирован, то функция возвращает True, если связен соответствующий неориентированный граф.
ConnectedQ[g, Strong] и ConnectedQ[g, Weak] возвращают True, если ориентированный граф g, соответственно сильно или слабо связен. См. С.97, 191

## ■ ConstructTableau
ConstructTableau[p] для каждого элемента перестановки p повторяет несколько раз «прыгающий алгоритм» и возвращает результат – табло Янга, ассоциированное с перестановкой p. См. С.80

## ■ Contract
Contract[g, {x, y}] возвращает граф, полученный из графа g стягиванием вершин x, y. См. С.149

## ■ CostOfPath
CostOfPath[g, p] суммирует веса ребер вдоль пути p. См. С.180

## ▪ CoxeterGraph
CoxeterGraph возвращает негамильтонов граф с высокой степенью симметрии, в котором существует автоморфизм, отображающий любой путь длины 3 на любой другой путь длины 3. См. С.255

## ▪ CubeConnectedCycle
CubeConnectedCycle[d] возвращаетг граф, полученный заменой каждой вершины d-мерного куба циклом длины d. См. С.162

## ▪ CubicalGraph
CubicalGraph возвращает граф куба. См. С.102

## ▪ Cut
Cut - тег функции NetworkFlow для возврата ею минимального разреза. См. С.182, 187, 235

## ▪ CycleIndex
CycleIndex[pg, x] возвращает полином от x[1], x[2], ..., x[index[g]], цикловой группы перестановок pg. Здесь index[pg] – это длина перестановки в pg. См. С.61

## ▪ Cycle
Cycle[n] строит цикл из n вершин. Можно указывать опцию Type, принимающую значения Directed или Undirected. По умолчанию Type -> Undirected. См. С.103

## ▪ Cycles
Cycles есть опция функции Involutions. См. С.44

## ▪ CycleStructure
CycleStructure[p, x] возвращает моном от x[1], x[2], ..., x[длина [p]], циклическую структуру перестановки p. См. С.43

## ▪ Cyclic
Cyclic есть аргумент функций теории перечисления Пойа -ListNecklaces, NumberOfNecklace и NecklacePolynomial, подсчитывающих или перечисляющих различные ожерелья. При этом два ожерелья считаются эквивалентными, если они получаются друг из друга некоторым вращением. См. С.60

## ▪ CyclicGroup
CyclicGroup[n] возвращает циклическую группу перестановок n символов. См. С.55

## ▪ CyclicGroupIndex
CyclicGroupIndex[n, x] возвращает циклический индекс циклической группы как полином от x[1], x[2], ..., x[n]. См. С.61

## ▪ DeBruijnGraph
DeBruijnGraph[m, n] строит n-мерный граф Де Брёйна с m символами для целых чисел m > 0 и n > 1. См. С.174

## ▪ DeBruijnGraph
DeBruijnGraph[alph,n] строит n-размерный граф Де Брёйна с символами из alph для непустого множества alph и целого числа n > 1 . Допускается опция VertexLabel, с значением по умолчанию False. Если VertexLabel-> True, то вершины помечаются строками из alph. См. С.174

## ▪ DeBruijnSequence
DeBruijnSequence[a, n] возвращает последовательность Де Брёйна
на алфавите a.
DeBruijnSequence [g] возвращает отсортированную последовательность степеней
графа g. См. С.176

## ▪ Degrees
Degrees[g] возвращает степени вершин 1, 2, 3, ... в этом же порядке. См. С.83

## ▪ DegreesOf2Neighborhood
DegreesOf2Neighborhood[g, v] возвращает отсортированный список вершин графа g с расстоянием 2 от вершины v. См. С.83

## ▪ DeleteCycle
DeleteCycle[g, c] удаляет простой цикл c из графа g. Цикл c задается последовательностью вершин, в которой первая и последняя вершины одинаковы. Если g не содержит c, то возвращается граф g. См. С.95

## ▪ DeleteEdge
DeleteEdge[g, e] возвращает граф g без ребра e. Если граф g ориентированный, то и ребро понимается как ориентированное. Если ребра кратные, то удаляется одно из них. См. С.134

## ▪ DeleteEdge
DeleteEdge[g, e, All] удаляет все ребра, задаваемые парой вершин.
DeleteEdges[g, edgeList] возвращает граф g без ребер списка edgeList. Если граф g ориентированный, то и ребра понимаются как ориентированные. Если ребра кратные, то удаляется одно из них.
DeleteEdges[g, edgeList, All] удаляет и кратные ребра. См. С.134

## ▪ DeleteFromTableau
DeleteFromTableau[t, r] удаляет последний элемент ряда r из табло Янга t. См. С.80

## ▪ DeleteVertex
DeleteVertex[g, v] удаляет одну вершину v из графа g. v есть номер вершины в графе. См. С.136

## ▪ DeleteVertices
DeleteVertices[g, vList] удаляет вершины списка vList из графа g. vList есть список {i, j, ...}, где i, j, ... номера вершин. См. С.136

## ▪ DepthFirstTraversal
DepthFirstTraversal[g, v] выполняет поиск в глубину в графе g, начиная с вершины v, и возвращает список вершин в порядке обхода.
DepthFirstTraversal[g, v, Edge] возвращает ребра графа в порядке обхода.
DepthFirstTraversal[g, v, Tree] возвращает дерево поиска графа. См. С.170

## ▪ DerangementQ
DerangementQ[p] возвращает True, если перестановка p является перестановкой без неподвижных точек. См. С.46

## ▪ Derangements
Derangements[p] строит все перестановки без неподвижных точек размера, равного размеру перестановки p. См. С.46

## ▪ Diameter
Diameter[g] возвращает диаметр графа g. См. С.227

## ▪ Dihedral
Dihedral – аргумент функций теории перечисления Пойа- ListNecklaces, NumberOfNecklace и NecklacePolynomial, считающих или перечисляющих различные ожерелья. Он указывает, что на ожерельях действует диэдральная группа, повороты и зеркальное отражение. См. С.60

## ▪ DihedralGroup
DihedralGroup[n] возвращает диэдральную группу n символов. См. С.55

## ▪ DihedralGroupIndex
DihedralGroupIndex[n, x] возвращает циклический индекс диэдральной группы Dn как полином от x[1], x[2], ..., x[n].См. С.61

## ▪ Dijkstra
Dijkstra[g, v] строит дерево кратчайших путей от вершины v до всех вершин графа g. Это дерево представляется списком, i-й элемент которого есть предшественник вершины i в дереве. Алгоритм Дейкстра работает некорректно для графов с отрицательными весами ребер; в этом случае следует применять функцию BellmanFord[g, v]. См. С.216

## ▪ DilateVertices
DilateVertices[v, d] умножает каждую координату всех вершин списка v на d.
DilateVertices[g, d] умножает каждую координату всех вершин графа на d. См. С.145

## ▪ Directed
Directed – значение опции Type. См. С.98, 242

## ▪ Disk
Disk – значение опции VertexStyle в функции ShowGraph. См. С.89.

## ▪ Distances
Distances[g, v] возвращает в неубывающем порядке расстояния от вершины v до всех вершин графа g, рассматривая g как невзвешенный граф. См. С.227

## ▪ DistinctPermutations
DistinctPermutations[l] возвращает все перестановки мультимножества l. См. С.30

## ▪ Distribution
Distribution[l, set] возвращает частоту каждого элемента множества set в списке l.

## ▪ DodecahedralGraph
DodecahedralGraph возвращает граф додекаэдра, одного из пяти платоновых тел. См. С.102

## ▪DominatingIntegerPartition
DominatingIntegerPartitionQ[a, b] возвращает True, если разбиение a доминирует над разбиением b. См. С.66

## ▪ DominationLattice
DominationLattice[n] возвращает диаграмму Хассе частично упорядоченного множества разбиений целого числа n; p < q если q доминирует над p. Функция имеет две опции: Type и VertexLabel, со значениями по умолчанию Undirected и False соответственно. Если Type -> Directed, строится ориентированный ациклический граф. Если VertexLabel -> True, то вершинам присваиваются метки. См. С.244

## ▪ DurfeeSquare
DurfeeSquare[p] возвращает количество рядов в квадрате Дюрфи разбиения p, наибольшего квадрата, содержащего диаграмму Феррерса разбиения p. См. С.67

## ▪ Eccentricity
Eccentricity[g] возвращает эксцентриситет каждой вершины v графа g - максимум расстояний от v до остальных вершин. См. С.227

## ▪ Edge
Edge есть необязательный аргумент некоторых функций для работы с ребрами вместо вершин. См. С.165

## ▪ EdgeChromaticNumber
EdgeChromaticNumber[g] возвращает наименьшее количество цветов, необходимых для раскраски ребер графа g, так чтобы два ребра, инцидентные одной вершине, имели разные цвета. См. С.211

## ▪ EdgeColor
EdgeColor есть опция, позволяющая присваивать цвета ребрам. Black - значение по умолчанию. EdgeColor может быть частью структуры графа или указываться в функции ShowGraph. См. С.86

## ▪ EdgeColoring
EdgeColoring[g] использует эвристику Брелаза для поиска реберной раскраски, не обязательно минимальной. См. С.211

## ▪ EdgeConnectivity
EdgeConnectivity[g] возвращает минимальное количество ребер, удаление которых из графа g нарушает его связность.
EdgeConnectivity[g, Cut] возвращает минимальное множество ребер, удаление которых из графа g нарушает его связность. См. С.187

## ▪ EdgeDirection
EdgeDirection есть опция со значениями True или False, позволяющая устанавливать ориентацию графа. EdgeDirection может быть частью структуры графа, или указываться в функции ShowGraph. См. С.86

## ▪ EdgeLabel
EdgeLabel есть опция, принимающая значения True или False, и позволяющая присваивать ребрам метки. По умолчанию, ребра не имеют меток. EdgeLabel может быть частью структуры графа, или указываться в функции ShowGraph. См. С.86

## ■ EdgeLabelColor
EdgeLabelColor есть опция, позволяющая присваивать цвета меткам ребер. Black - цвет по умолчанию. EdgeLabelColor может быть частью структуры графа или указываться в функции ShowGraph. См. С.86

## ■ EdgeLabelPosition
EdgeLabelPosition есть опция, позволяющая указывать позицию метки ребра относительно середины ребра. LowerLeft - значение по умолчанию. EdgeLabelPosition может быть частью структуры графа или указываться в функции ShowGraph. См. С.86

## ■ Edges
Edges[g] возвращает список ребер графа g
Edges[g, All] возвращает список ребер графа g вместе с графической информацией каждого ребра.
Edges[g, EdgeWeight] возвращает список ребер графа g вместе с их весами. См. С.112

## ■ EdgeStyle
EdgeStyle есть опция, позволяющая указывать размер и форму ребер. EdgeStyle может быть частью структуры графа или указываться в функции ShowGraph. См. С.86

## ■ EdgeWeight
EdgeWeight есть опция, позволяющая присваивать ребрам веса. 1 – значение по умолчанию. EdgeWeight может быть частью структуры графа или указываться в функции ShowGraph. См. С.86

## ■ Element
Element [a,p] возвращает p-й элемент вложенного списка a. См. С.38.

## ■ EmptyGraph
EmptyGraph[n] строит пустой граф с n вершинами. Возможна опция Type, принимающая значения Directed или Undirected. Значение по умолчанию - Type -> Undirected. См. С.83

## ■ EmptyQ
EmptyQ[g] возвращает True, если граф g не содержит ребер. См. С.83

## ■ EncroachingListSet
EncroachingListSet[p] возвращает упорядоченное множество упорядоченных подмножеств перестановки p, причем каждый последующий список должен быть вложен в предыдущий. См. С.38

## ■ EquivalenceClasses
EquivalenceClasses[r] определяет классы эквивалентности среди элементов матрицы r. См. С.54

## ■ EquivalenceRelationQ
EquivalenceRelationQ[r] возвращает True, если матрица r определяет отношение эквивалентности.
EquivalenceRelationQ[g] возвращает True, если матрица смежности графа g определяет отношение эквивалентности. См. С.54

## ▪ Equivalences
Equivalences[g, h] перечисляет классы эквивалентности вершин между графами g и h по степеням вершин.
Equivalences[g] перечисляет классы эквивалентности вершин графа g по степеням вершин.
Можно использовать формы Equivalences[g, h, f1, f2, ...] и Equivalences[g, f1, f2, ...], где f1, f2, ... функции, вычисляющие другие инварианты вершины.
Предполагается, что вызов fi[g, v] возвращает соответствующий инвариант вершины v графа g. Например, функции DegreesOf2Neighborhood, NumberOf2Paths, и Distances. См. С.247

## ▪ Euclidean
Euclidean есть опция функции SetEdgeWeights. См. С.117

## ▪ Eulerian
Eulerian[n, k] возвращает количество перестановок длины n, состоящих из k непрерывных возрастающих подпоследовательностей. См. С.37

## ▪ EulerianCycle
EulerianCycle[g] находит эйлеров цикл графа g, если он существует. См. С.173

## ▪ EulerianQ
EulerianQ[g] возвращает True, если граф g эйлеров. См. С.173

## ▪ ExactRandomGraph
ExactRandomGraph[n, e] строит случайный помеченный граф с e ребрами и n вершинами. См. С.111

## ▪ ExtractCycles
ExtractCycles[g] возвращает максимальный список реберно непересекающихся циклов графа g. См. С.95

## ▪ FerrersDiagram
FerrersDiagram[p] рисует диаграмму Феррерса разбиения p. См. С.67

## ▪ FindCycle
FindCycle[g] находит список вершин, образующих цикл в графе g.См. С.95

## ▪ FindSet
FindSet[n, s] возвращает расстояние от корня до n в дереве поиска s.

## ▪ FiniteGraphs
FiniteGraphs возвращает список непараметризованных графов. См. С.91

## ▪ FirstLexicographicTableau
FirstLexicographicTableau[p] конструирует первое табло Янга формы разбиения p. См. С.78

## ▪ FolkmanGraph
FolkmanGraph возвращает наименьший реберно-транзитивный граф Фолкмана, не являющийся вершинно-транзитивным. См. С.256

## ▪ FranklinGraph
FranklinGraph возвращает 12-вершинный граф 6-цветной карты на бутылке Клейна. Известно, что любую карту на плоскости можно раскрасить 4 цветами. См. С.212

## ▪ FromAdjacencyLists
FromAdjacencyLists[l] строит граф по списку смежности l. Опция Type, принимающая значения Directed или Undirected, указывает, ориентирован граф или нет. Значение по умолчанию Type -> Undirected. См. С.122

## ▪ FromAdjacencyMatrix
FromAdjacencyMatrix[m] строит граф по матрице смежности m. Опция Type, принимающая значения Directed или Undirected, указывает, ориентирован граф или нет. Значение по умолчанию Type ->Undirected.
FromAdjacencyMatrix[m, EdgeWeight] интерпретирует элементы m как веса ребер, с весом бесконечность для отсутствующих ребер. См. С.125

## ▪ FromCycles
FromCycles[{c1, c2, ...}] возвращает перестановку данной циклической структуры. См. С.41

## ▪ FromInversionVector
FromInversionVector[v] строит перестановку по вектору инверсий v. См. С.35

## ▪ FromOrderedPairs
FromOrderedPairs[l] строит граф по списку упорядоченных пар l.
FromOrderedPairs[l, v] строит граф по списку упорядоченных пар l с вложением v. Опция Type, принимающая значения Directed или Undirected, указывает ориентирован граф или нет. Значение по умолчанию Type -> Directed. См. С.120

## ▪ FromUnorderedPairs
FromUnorderedPairs[l] строит граф из списка неупорядоченных пар l.
FromUnorderedPairs[l, v] строит граф из списка неупорядоченных пар l с вложением v. Опция Type, принимающая значения Directed или Undirected, указывает, ориентирован граф или нет. Значение по умолчанию
Type ->Undirected. См. С.120

## ▪ FruchtGraph
FruchtGraph возвращает наименьший 3-регулярный граф, группа автоморфизмов которого тривиальна. См. С.252

## ▪ FunctionalGraph
FunctionalGraph[f, v] по множеству v и функции f : v -> v строит ориентированный граф с множеством вершин v и ребрами (x, f(x)) для всех x из v.
FunctionalGraph[f, v], где f есть список функций, строит граф с множеством вершин v и ребрами (x, fi(x)) для каждой функции fi из списка f. Опция Type, принимающая значения Directed или Undirected, указывает, ориентирован граф или нет. Значение по умолчанию Type ->Directed, если Type -> Undirected, то возвращается соответствующий неориентированный граф.
FunctionalGraph[f, n] по функции f, отображающей множество {0,1,..., n-1} на себя, строит ориентированный граф с множеством вершин {0, 1,..., n-1} и ребрами {x, f(x)} для каждой вершины x. См. С.131

## ▪ GeneralizedPetersenGraph
GeneralizedPetersenGraph[n, k] возвращает обобщенный граф Петерсена для целых чисел n > 1 и k > 0, граф с вершинами {u1, u2, ..., un} и {v1, v2, ..., vn} и ребрами {ui, u(i+1)}, {vi, v(i+k)}, и {ui, vi}. При n = 5 и k = 2 получается граф Петерсена. См. С.140

## ▪ GetEdgeLabels
GetEdgeLabels[g] возвращает список меток ребер графа g.
GetEdgeLabels[g, es] возвращает список меток ребер из es. См. С.117

## ▪ GetEdgeWeights
GetEdgeWeights[g] возвращает список весов ребер графа g.
GetEdgeWeights[g, es] возвращает список весов ребер из списка es. См. С.120

## ▪ GetVertexLabels
GetVertexLabels[g] возвращает список меток вершин графа g.
GetVertexLabels[g, vs] возвращает список меток вершин из vs. См. С.115

## ▪ GetVertexWeights
GetVertexWeights[g] возвращает список весов вершин графа g.
GetVertexWeights[g, vs] возвращает список весов вершин из vs. См. С.117

## ▪ Girth
Girth[g] возвращает обхват графа g. См. С.228

## ▪ Graph
Graph[e, v, opts] представляет объект граф, где e есть список ребер с опциями графики, v есть список вершин с опциями графики и opts есть множество глобальных опций графа.
e имеет форму {{{i1, j1}, opts1}, {{i2, j2}, opts2},...}, где {i1, j1}, {i2, j2},... ребра графа и opts1, opts2,... есть опции ребер. v имеет форму {{{x1, y1}, opts1}, {{x2, y2}, opts2},...}, где {x1, y1}, {x2, y2},... координаты на плоскости вершины1, вершины2,... и opts1, opts2,... есть опции вершин. Опциями ребер являются EdgeWeight, EdgeColor, EdgeStyle, EdgeLabel, EdgeLabelColor и EdgeLabelPosition.
Опции вершин – VertexWeight, VertexColor, VertexStyle, VertexNumber, VertexNumberColor, VertexNumberPosition, VertexLabel, VertexLabelColor и VertexLabelPosition. Третий элемент opts - последовательность глобальных опций применимые для всех вершин или всех ребер или к графу в целом. Все опции ребер и вершин могут быть использованы как глобальные опции. Если глобальная опция отличается от локальной, то применяется локальная опция. В дополнение к этим опциям есть еще две глобальные опции: LoopPosition и EdgeDirection. Более того, все опции функции Plot могут появляться как глобальные опции в объекте Graph. См. С. 114

## ▪ GraphCenter
GraphCenter[g] возвращает список вершин графа g с минимальным эксцентриситетом. См. С.231

## ▪ GraphComplement
GraphComplement[g] возвращает дополнение графа g. См. С.157

## ▪ GraphDifference
GraphDifference[g, h] строит граф, полученный удалением ребер графа h из ребер графа g. См. С.157

## ▪ GraphicQ
GraphicQ[s] возвращает True, если список целых чисел s есть графическая последовательность, т. е. последовательность степеней вершин некоторого графа. См. С.128

## ▪ GraphIntersection
GraphIntersection[g1, g2, ...] строит пересечение графов, граф с ребрами, встречающимися во всех графах g1, g2, ... . См. С.154

## ▪ GraphJoin
GraphJoin[g1, g2, ...] строит соединение графов g1, g2, ... . Этот граф получается добавлением всех возможных ребер между графами g1, g2, .... См. С.157

## ▪ GraphOptions
GraphOptions[g] возвращает опции графа g.
GraphOptions[g, v] возвращает опции вершины v.
GraphOptions[g, {u, v}] возвращает опции ребра {u, v}. См. С.114

## ▪ GraphPolynomial
GraphPolynomial[n, x] возвращает полином от x, в котором коэффициент при x^m есть количество неизоморфных графов с n вершинами и m ребрами.
GraphPolynomial[n, x, Directed] возвращает то же для ориентированных графов. См. С.98

## ▪ GraphPower
GraphPower[g, k] возвращает k-ю степень графа g. Это граф, множество вершин которого совпадает с множеством вершин g и содержит ребро между вершинами i и j, если в g существует путь от i до j длины, равной или меньшей чем k. См. С.227

## ▪ GraphProduct
GraphProduct[g1, g2, ...] строит произведение графов g1, g2,... См. С.159

## ▪ GraphSum
GraphSum[g1, g2, ...] строит граф объединением ребер графов g1, g2,... См. С.156

## ▪ GraphUnion
GraphUnion[g1, g2, ...] строит объединение графов g1, g2, ... .
GraphUnion[n, g] строит n копий графа g. См. С.156

## ▪ GrayCode
GrayCode[l] строит код Грэя множества l. Вместо этой функции рекомендуется использовать GrayCodeSubsets[l] См. С.49

## ▪ GrayCodeKSubsets
GrayCodeKSubsets[l, k] генерирует k-подмножества множества l в порядке возрастания их кода Грэя. См. С.52

## ▪ GrayCodeSubsets
GrayCodeSubsets[l] строит код Грэя множества l. См. С.49

## ▪ GrayGraph
GrayGraph возвращает 3-регулярный, 54-вершинный реберно транзитивный, но не вершинно транзитивный граф; наименьший известный пример. См.с. 256

## ▪ Greedy
Greedy есть значение опции Algorithm такой функции, как VertexCover, для указания того, чтобы при вычислении применялся жадный алгоритм. См. С.201

## ▪ GreedyVertexCover
GreedyVertexCover[g] возвращает вершиное покрытие графа g, полученное при помощи жадного алгоритма, которое может и не быть в точности вершинным покрытием. См. С.201

## ▪ GridGraph
GridGraph[n, m] строит n×m-решетчатый граф, произведение путей из n вершин и m вершин.
GridGraph[p, q, r] строит p×q×r-решетчатый граф, произведение GridGraph[p, q] и пути из r вершин. См. С.105

## ▪ GrotztschGraph
GrotztschGraph возвращает наименьший граф без треугольников с хроматическим числом 4. Она идентична функции MycielskiGraph[4]. См. С.205

## ▪ HamiltonianCycle
HamiltonianCycle[g] находит гамильтонов цикл в графе g, если он существует.
HamiltonianCycle[g, All] возвращает все гамильтоновы циклы графа g. См. С.176

## ▪ HamiltonianPath
HamiltonianPath[g] находит гамильтонов путь графа g, если он существует.
HamiltonianPath[g, All] возвращает все гамильтоновы пути графа g. См. С.176

## ▪ HamiltonianQ
HamiltonianQ[g] возвращает True, если в графе g существует гамильтонов цикл. См. С.176

## ▪ Harary
Harary[k, n] строит минимальный k-связный граф с n вершинами. См. С.185

## ▪ HasseDiagram
HasseDiagram[g] строит диаграмму Хассе бинарного отношения, заданного ациклическим ориентированным графом g. См. С.241

## ▪ Heapify
Heapify[p] строит heap из перестановки p. Последовательность $\{a_n\}_{n=1}^{N}$ называется heap, если оно удовлетворяет условию $a_{[j/2]} \le a_j$ для $2 \le j \le N$, где [x]- целая часть x. Heap можно представить как помеченное бинарное дерево, в котором метка i-й вершины меньше, чем метка ее потомка См. С.38

## ▪ HeapSort
HeapSort[l] выполняет heap сортировку на элементах списка l. См. С.38

## ▪ HeawoodGraph
HeawoodGraph возвращает наименьший (6, 3)-клеточный, 3-регулярный граф с обхватом 6. См. С.228

## ▪ HerschelGraph

HerschelGraph возвращает граф Гершеля. См. с.260

## ▪ HideCycles
HideCycles[c] переводит циклическую структуру c в единственную перестановку. См. С.41

## ▪ Highlight
Highlight[g, p] изображает граф g с выделенными элементами из p. p есть список {s1, s2,...} из непересекающихся подмножеств вершин и ребер графа g. Применяются опции HighlightedVertexStyle, HighlightedEdgeStyle, HighlightedVertexColors, и HighlightedEdgeColors для изображения выделенных элементов графа. По по умолчанию HighlightedVertexStyle->Disk[Large] и HighlightedEdgeStyle->Thick. Опции HighlightedVertexColors и HighlightedEdgeColors принимают значения подмножеств множества
{Black, Red, Blue, Green, Yellow, Purple, Brown, Orange, Olive, Pink,
DeepPink, DarkGreen, Maroon, Navy}. Цвета выбираются из палитры так: color1 для s1, color2 для s2 и т.д. Если выделенных элементов большечем цветов, то цвета выбираются циклически. Функция допускает все опции функции SetGraphOptions , например, VertexColor, VertexStyle, EdgeColor, и EdgeStyle. Эти опции используются для невыделенных элементов. См. С.92

## ▪ HighlightedEdgeColors
HighlightedEdgeColors есть опция функции Highlight, определяющая цвет ребер. См. С.92

## ▪ HighlightedEdgeStyle
HighlightedEdgeStyle есть опция функции Highlight, определяющая размер и форму выделенных ребер. См. С.92

## ▪ HighlightedVertexColors
HighlightedVertexColors есть опция функции Highlight, определяющая цвет выделенных вершин. См. С.92

## ▪ HighlightedVertexStyle
HighlightedVertexStyle есть опция функции Highlight, определяющая размер и форму выделенных вершин. См. С.92

## ▪ Hypercube
Hypercube[n] строит n-мерный гиперкуб. См. С.160

## ▪ IcosahedralGraph
IcosahedralGraph возвращает граф икосаэдра. См. С.102

## ▪ IdenticalQ
IdenticalQ[g, h] возвращает True, если графы g и h имеют одинаковые списки ребер. См. С.98

## ▪ IdentityPermutation
IdentityPermutation[n] возвращает единичную перестановку размера n. См. С.33

## ▪ IncidenceMatrix
IncidenceMatrix[g] возвращает (0, 1)-матрицу графа g, ряды которой соответствуют вершинам, а столбцы ребрам, и (v, e) = 1 если и только если вершина v инцидентна ребру e. Для ориентированного графа (v, e) = 1, если ребро e исходит из v. См. С.128

## ▪ InDegree
InDegree[g, n] возвращает полустепени захода вершины n в ориентированном графе g.
InDegree[g] возвращает последовательность полустепеней захода в ориентированном графе g. См. С.190

## ▪ IndependentSetQ
IndependentSetQ[g, i] возвращает True, если вершины списка i образуют независимое множество графа g. См. С.199

## ▪ Index
Index[p] возвращает индекс перестановки p, сумму всех таких j, для которых p[j] больше, чем p[j+1].См. С.37

## ▪ InduceSubgraph
InduceSubgraph[g, s] строит вершинно порожденный подграф графа g из списка вершин s. См. С.94

## ▪ InitializeUnionFind
InitializeUnionFind[n] инициализирует дерево поиска для множества из n элементов.

## ▪ InsertIntoTableau
InsertIntoTableau[e, t] вставляет целое число e в табло Янга t, используя «прыгающий алгоритм».
InsertIntoTableau[e, t, All] вставляет элемент e в табло Янга t и возвращает новое табло Янга и номер строки, размер которой увеличился за счет вставки. См. С.78

## ▪ IntervalGraph
IntervalGraph[l] строит интервальный граф по списку интервалов l. См. С.129

## ▪ Invariants
Invariants есть опция функций Isomorphism и IsomorphicQ для определения инвариантов для вычисления эквивалентности вершин. См. С.129

## ▪ InversePermutation
InversePermutation[p] возвращает обратную перестановку перестановки p. См. С.33

## ▪ InversionPoset
InversionPoset[n] возвращает диаграмму Хассе частично упорядоченного
множества перестановок длины n, в котором p < q если q может быть
получена из p транспозицией соседних элементов, ставящей больший элемент перед меньшим. Функция имеет две опции: Type и VertexLabel, со значениями по умолчанию Undirected и False, соответственно. Если Type -> Directed, то строится сответствующий ориентированный ациклический граф. Если VertexLabel -> True, то вершины помечаются соответствующими перестановками. См. С.242

## ▪ Inversions
Inversions[p] считает число инверсий в перестановке p. См. С.35

## ▪ InvolutionQ
InvolutionQ[p] возвращает True, если перестановка p совпадает со своей обратной. См. С.44

## ▪ Involutions
Involutions[l] возвращает список инволюций из множества перестановок l.
Involutions[l, Cycles] возвращает инволюции в виде произведения циклов.
Involution[n] возвращает все инволютивные перестановки длины n.
Involutions[n, Cycles] возвращает все инволютивные перестановки длины n в виде произведения циклов. См. С.44

## ▪IsomorphicQ
IsomorphicQ[g, h] возвращает True, если графs g и h изоморфны. Функция имеет опцию Invariants -> {f1, f2, ...}, где f1, f2, ... есть функции вычисляющие инварианты вершин. Эти функции используются по порядку. Значение по умолчанию есть {DegreesOf2Neighborhood, NumberOf2Paths, Distances}. См. С.98

## ▪ Isomorphism
Isomorphism[g, h] возвращает изоморфизм графов g и h, если он существует.
Isomorphism[g, h, All] возвращает все изоморфизмы графов g и h.
Isomorphism[g] возвращает группу автоморфизмов графа g. Эти функции имеют опцию Invariants -> {f1, f2, ...}, где f1, f2, ... есть функции, вычисляющие инварианты вершин. Эти функции используются по порядку. Значение по умолчанию есть {DegreesOf2Neighborhood, NumberOf2Paths, Distances}. См. С.250

## ▪ IsomorphismQ
IsomorphismQ[g, h, p] возвращает True, если перестановка p есть изоморфизм между графами g и h. См. С.250

## ▪ Josephus
Josephus[n, m] решает задачу Иосифа Флавия. См. С.264

## ▪ KnightsTourGraph
KnightsTourGraph[m, n] возвращает граф с m*n вершинами, представляющий ходы коня на шахматной доске размера m x n. См. С.287

## ▪ KSetPartitions
KSetPartitions[set, k] возвращает список разбиений множества set на k частей.
KSetPartitions[n, k] возвращает список разбиений множества {1, 2, ..., n} на k частей. Для получения всех разбиений используется функция SetPartitions. См. С.71

## ▪ KSubsetGroup
KSubsetGroup[pg, s] возвращает группу, порожденную действием
группы перестановок pg на множестве s k-элементных подмножеств множества {1, 2, ..., n}, где n есть порядок группы pg. Опция Type со значениями Ordered или Unordered указывает на то, что есть s - множество или кортеж. См. С.327

## ▪ KSubsetGroupIndex
KSubsetGroupIndex[g, s, x] возвращает цикловой индекс группы KSubsetGroup[pg, s] в виде полинома от x[1], x[2], .... Опция Type со значениями Ordered или Unordered указывает на то, что есть s - множество или кортеж. См. С.61

## ▪ KSubsets
KSubsets[l, k] возвращает в лексикографическом порядке все k-подмножества. См. С.50

## ■ LabeledTreeToCode
LabeledTreeToCode[g] возвращает код Прюфера дерева g. См. С.108

## ■ Large
Large есть символ для обозначения размера вершин. Опция VertexStyle может иметь значения Disk[Large] или Box[Large]. См. С.84

## ■ LastLexicographicTableau
LastLexicoграфicTableau[p] строит последнее табло Янга формы, определяемой разбиением p. См. С.78

## ■ Level
Level есть опция функции BreadthFirstTraversal для возврата уровней вершин. См. С.

## ■ LeviGraph
LeviGraph возвращает единственный (8, 3)-клеточный, 3-регулярный граф
обхвата 8. См. С.230

## ■ LexicographicPermutations
LexicographicPermutations[l] возвращает в лексикографическом порядке все перестановки множества l. См. С.32

## ■ LexicographicSubsets
LexicographicSubsets[l] возвращает в лексикографическом порядке все подмножества множества l.
LexicographicSubsets[n] возвращает в лексикографическом порядке все подмножества множества {1,2,...,n}. См. С.47

## ■ LineGraph
LineGraph[g] строит реберный граф графа g. См. С.163, 291

## ■ ListGraphs
ListGraphs[n, m] возвращает все неизоморфные неориентированные
графы с n вершинами и m ребрами.
ListGraphs[n, m, Directed] возвращает все неизоморфные ориентированные
графы с n вершинами и m ребрами.
ListGraphs[n] возвращает все неизоморфные неориентированные
графы с n вершинами.
ListGraphs[n, Directed] возвращает все неизоморфные ориентированные
графы с n вершинами. См. С.98

## ■ ListNecklaces
ListNecklaces[n, c, Cyclic] возвращает все различные ожерелья, раскрашенные цветами из списка c. Здесь c есть список n необязательно различных цветов. Два ожерелья считаются эквивалентными, если одно может быть получено вращением другого.
ListNecklaces[n, c, Dihedral] аналогична предыдущей функции, если к вращениям добавить зеркальное отображения. См. С.58, 329

## ■ LNorm
LNorm[p] есть значение опции WeightingFunction функции SetEdgeWeights. См. С.116

## ■ LongestIncreasingSubsequence
LongestIncreasingSubsequence[p] находит самую длинную возрастающую непрерывную подпоследовательность в перестановке p. См. С.38

## ■ LoopPosition
LoopPosition есть опция функции ShowGraph, значение s которой определяет позицию петли у вершины. Эта опция имеет значения UpperLeft, UpperRight, LowerLeft, и LowerRight. См. С.86

## ■ LowerLeft
LowerLeft есть значение опций VertexNumberPosition, VertexLabelPosition, и EdgeLabelPosition функции ShowGraph. См. С.86

## ■ LowerRight
LowerRight есть значение опций VertexNumberPosition, VertexLabelPosition, и EdgeLabelPosition функции ShowGraph. См. С.86

## ■ M
M[g] возвращает количество ребер в графе g. См. С.114

## ■ MakeDirected
MakeDirected[g] строит ориентированный граф из неориентированного графа g, заменяя каждое ребро двумя ориентированными ребрами противоположных направлений. См. С.190

## ■ MakeGraph
MakeGraph[v, f] строит граф с вершинами из множества v и ребрами (x , y) в v, если значения булевой функции f(x,y) есть True. Опция Type и VertexLabel. Функция имеет две опции: Type и VertexLabel, со значениями по умолчанию Directed и False соответственно. Если Type -> Directed, то строится ориентированный граф. Если VertexLabel -> True, то вершины помечаются метками из v. См. С.129

## ■ MakeSimple
MakeSimple[g] возвращает неориентированный граф без петель и кратных ребер, полученный из графа g. См. С.137

## ■ MakeUndirected
MakeUndirected[g] возвращает соответствующий неориентированный граф ориентированного графа g. См. С.190

## ▪ MaximalMatching
MaximalMatching[g] возвращает список ребер максимального паросочетания графа g. См. С.193

## ▪ MaximumAntichain
MaximumAntichain[g] возвращает наибольшее множество вершин , не находящихся в отношении в частичном порядке g.См. С.245

## ▪ MaximumClique
MaximumClique[g] находит максимальную клику графа g.
MaximumClique[g, k] возвращает k-клику, если таковая существует в g;
иначе она возвращает пустой список {}.См. С.213

## ■ MaximumIndependentSet
MaximumIndependentSet[g] находит наибольшее независимое множество графа g. См. С.201

## ■ MaximumSpanningTree
MaximumSpanningTree[g] использует алгоритм Краскала для нахождения максимального остова графа g. См. С.233

## ■ McGeeGraph
McGeeGraph возвращает единственный (7, 3)-клеточный, 3-регулярный граф с обхватом 7. См. С.229

## ■ MeredithGraph
MeredithGraph возвращает 4-регулярный, 4-связный негамильтонов граф. См. С.179

## ■ MinimumChainPartition
MinimumChainPartition[g] разбивает частичный порядок g на минимальное количество цепей. См. С.245

## ■ MinimumChangePermutations
MinimumChangePermutations[l] генерирует все перестановки множества l так, что каждая следующая перестановка получается из предыдущей одной транспозицией. См. С.44

## ■ MinimumSpanningTree
MinimumSpanningTree[g] использует алгоритм Краскала нахождения минимального остова графа g. См. С.231

## ■ MinimumVertexColoring
MinimumVertexColoring[g] возвращает минимальную вершинную раскраску графа g.
MinimumVertexColoring[g, k] возвращает k-раскраску графа g, если таковая существует. См. С.205, 292

## ■ MinimumVertexCover
MinimumVertexCover[g] находит минимальное вершинное покрытие графа g. Для двудольных графов используется полиномиальной сложности венгерский алгоритм. Иначе используется полный перебор. См. С.202

## ■ MultipleEdgesQ
MultipleEdgesQ[g] возвращает True, если g имеет кратные ребра, иначе - False. См. С.83

## ■ MultiplicationTable
MultiplicationTable[l, f] строит полную транзитивную таблицу, определенную бинарной булевой функцией f на множестве l.См. С.55,324

## ■ MycielskiGraph
MycielskiGraph[k] возвращает граф с без треугольников с хроматическим числом k, граф Мыцельского. См. С.205

## ▪ NecklacePolynomial

NecklacePolynomial[n, c, Cyclic] возвращает полином, чьи коэффициенты равны числу способов раскраски n-ожерелий цветами, выбираемыми из c, при этом две раскраски эквивалентны, если могут быть получены друг из друга вращением.
NecklacePolynomial[n, c, Dihedral] в этой функции две раскраски эквивалентны, если могут быть получены друг из друга вращением и зеркальным отражением. См. С.58, 328

## ▪ Neighborhood

Neighborhood[g, v, k] возвращает подмножество вершин графа g, находящееся на расстоянии, меньшем или равным k от вершины v.
Neighborhood[al, v, k] отличается от функции Neighborhood[g, v, k] только заменой графа g матрицей смежности al. См. С.248

## ▪ NetworkFlow

NetworkFlow[g, source, sink] возвращает значение максимального потока графа g от source до sink.
NetworkFlow[g, source, sink, Edge] возвращает ребра в g с положительными значениями потока вместе с их потоками в максимальном потоке от source до sink.
NetworkFlow[g, source, sink, Cut] возвращает минимальный разрез между source и sink.
NetworkFlow[g, source, sink, All] возвращает список смежности графа g вдоль потока. g может быть как ориентированным, так и неориентированным графом. См. С.235, 295

## ▪ NetworkFlowEdges

NetworkFlowEdges[g, source, sink] возвращает ребра графа с положительным потоком, показывая распределение максимального потока из источника source до стока sink. Эта функция устарела, рекомендуется использовать NetworkFlow[g, source, sink, Edge]. См. С.235

## ▪ NextBinarySubset

NextBinarySubset[l, s] строит подмножество множества l, следующее за s в порядке возрастания характеристических векторов. См. С.47

## ▪ NextComposition

NextComposition[l] возвращает целочисленную композицию, следующую за l в каноническом порядке. См. С.69

## ▪ NextGrayCodeSubset

NextGrayCodeSubset[l, s] строит следующее за s, в порядке возрастания кода Грэя, подмножество множества l. См. С.49

## ▪ NextKSubset

NextKSubset[l, s, k] возвращает следующее за s, в лексикографическом порядке, k-подмножество множества l. См. С.50

## ▪ NextLexicographicSubset

NextLexicographicSubset [l,s] возвращает подмножество, следующее за подмножеством s в множестве LexicographicSubsets[l]. См.С. 47.

## ▪ NextPartition

NextPartition[p] возвращает целочисленное разбиение, следующее за p в порядке, обратном лексикографическому. См. С.64

## ■ NextPermutation
NextPermutation[p] возвращает перестановку, следующую за p, в лексикографическом порядке. См. С.32, 309

## ■ NextSubset
NextSubset[l, s] возвращает следующее за s, в каноническом порядке, подмножество множества l. См. С.50

## ■ NextTableau
NextTableau[t] возвращает следующее в лексикографическом порядке за табло Янга формы t. См. С.78

## ■ NoMultipleEdges
NoMultipleEdges есть значение опции для Type. См. С.137

## ■ NonLineGraphs
NonLineGraphs возвращает девять графов. Граф является реберным тогда и только тогда, когда он не содержит любой из девяти возвращаемых функцией NonLineGraphs графов как подграфы. См. С.163

## ■ NoPerfectMatchingGraph
NoPerfectMatchingGraph возвращает 16-вершинный связный граф, не содержащий совершенного паросочетания. См. С.196

## ■ Normal
Normal есть значение опций VertexStyle, EdgeStyle и PlotRange функции ShowGraph. См. С.84

## ■ NormalDashed
NormalDashed есть значение опции EdgeStyle. См. С.86

## ■ NormalizeVertices
NormalizeVertices[v] возвращает список вершин с таким же вложением, как и v, со всеми координатами, сжатыми до диапазона [0 ,1]. См. С.148

## ■ NoSelfLoops
NoSelfLoops есть значение опции Type. См. С.137

## ■ NthPair
NthPair[n] возвращает n-ую неупорядоченную пару положительных целых чисел, упорядоченных по значению большего элемента пары.
Пары с одинаковыми большими элементами упорядочены по значению меньшего элемента пары.

## ■ NthPermutation
NthPermutation[n, l] возвращает n-ую в лексикографическом порядке перестановку списка l.

## ■ NthSubset
NthSubset[n, l] возвращает n-ое в каноническом порядке подмножество списка l. См. С.47

## ▪ NumberOf2Paths
NumberOf2Paths[g, v] возвращает упорядоченный список, содержащий пути длины 2 из вершины v. См. С.225

## ▪ NumberOfCompositions
NumberOfCompositions[n, k] возвращает количество композиций целого числа n из k частей. См. С.69

## ▪ NumberOfDerangements
NumberOfDerangements[n] возвращает число перестановок без неподвижных точек размера n. См. С.46

## ▪NumberOfDirectedGraphs
NumberOfDirectedGraphs[n] возвращает число неизоморфных ориентированных n-вершиных графов.
NumberOfDirectedGraphs[n, m] возвращает число неизоморфных ориентированных n-вершиных графов с m ребрами. См. С.99

## ▪ NumberOfGraphs
NumberOfGraphs[n] возвращает число неизоморфных неориентированных n-вершиных графов.
NumberOfGraphs[n, m] возвращает число неизоморфных неориентированных n-вершиных графов с m ребрами. См. С.99

## ▪ NumberOfInvolutions
NumberOfInvolutions[n] возвращает количество инволюций на n элементах. См. С.46

## ▪ NumberOfKPaths
NumberOfKPaths[g, v, k] возвращает сортированный список, содержащий количество путей длины k от вершины v до всех вершин графа g.
NumberOfKPaths[al, v, k] идентична предыдущей, только al есть матрица смежности. См. С.225

## ▪ NumberOfNecklaces
NumberOfNecklaces[n, nc, Cyclic] возвращает число способов раскраски n-ожерелий nc цветами, при этом две раскраски эквивалентны, если могут быть получены друг из друга вращением.
NumberOfNecklaces[n, nc, Dihedral] в этой функции две раскраски эквивалентны, если могут быть получены друг из друга вращением и зеркальным отражением. См. С.58, 326

## ▪ NumberOfPartitions
NumberOfPartitions[n] считает количество разбиений числа n. См. С.64

## ▪ NumberOfPermutationsByCycles
NumberOfPermutationsByCycles[n, m] возвращает количество перестановок длины n, состоящих точно из m циклов. См. С.43

## ▪ NumberOfPermutationsByInversions
NumberOfPermutationsByInversions[n, k] возвращает количество перестановок длины n с точно k инверсиями.
NumberOfPermutationsByInversions[n] возвращает таблицу количества перестановок с k инверсиями для всех k. См. С.37

## ▪ NumberOfPermutationsByTypes
NumberOfPermutationsByTypes[l] возвращает количество перестановок типа l. См. С.43

## ▪ NumberOfSpanningTrees
NumberOfSpanningTrees[g] возвращает количество помеченных остовов графа g. См. С.233

## ▪ NumberOfTableaux
NumberOfTableaux[p] возвращает количество табло Янга формы, определенной разбиением p. См. С.81

## ▪ OctahedralGraph
OctahedralGraph возвращает граф октаэдра. См. С.102

## ▪ OddGraph
OddGraph[n] возвращает граф, вершины которого есть (n-1)-подмножества (2n-1)-элементного множества и ребра которого соединяют пару вершин, если соответствующие подмножества не пересекаются. OddGraph[3] граф Петерсена. См. С.129

## ▪ One
One есть тег некоторых функций, для указания того, что выбирается один объект, а не все, или возвращается одно решение из многих. См. С.119

## ▪ Optimum
Optimum есть значение опции Algorithm функций VertexColoring и VertexCover. См. С.201

## ▪ OrbitInventory
OrbitInventory[ci, x, w] возвращает значение циклового индекса ci, заменяя переменную x[i] значением w.
OrbitInventory[ci, x, weights] берет цикловой индекс в переменных x[1], x[2]..… и список weights весов $w_1, w_2$,…….и вычисляет полином, полученный подстановкой в цикловой индекс суммы ($w_1^i+w_2^i$+…..) вместо x[i]. См. С.58

## ▪ OrbitRepresentatives
OrbitRepresentatives[pg, x] возвращает представление каждой орбиты x группы pg, действующей на x. pg есть множество перестановок и x -множество функций на {1, 2, ..., n}. Каждая функция из x представляется n-кортежем. См. С.58, 324

## ▪ Orbits
Orbits[pg, x] возвращает орбит множества x действия группы pg на x. pg полагается как множество перестановок первых n натуральных чисел и x есть множество функций на {1, 2, ..., n}. Каждая функция из x представляется n-кортежем. См. С.58

## ▪ Ordered
Ordered есть опция функций KSubsetGroup и KSubsetGroupIndex, указывающая тип входных данных: множества или кортежи. См. С.60

## ▪ OrientGraph
OrientGraph[g] ориентирует каждое ребро неориентированного графа g без мостов, так что он становится сильно связным. См. С.191

## ■ OutDegree

OutDegree[g, n] возвращает полустепень исхода вершины n в ориентированном графе g.
OutDegree[g] возвращает последовательность полустепеней исхода вершин в ориентированном графе g. См. С.191

## ■ PairGroup

PairGroup[g] возвращает группу, порожденную двухэлементными подмножествами группы g.
PairGroup[g, Ordered] возвращает группу, порожденную упорядоченными парами различающихся элементов группы g. См. С.58

## ■ PairGroupIndex

PairGroupIndex[g, x] возвращает цикловой индекс парной группы, порожденной группой g как полином от x[1], x[2], ....
PairGroupIndex[ci, x] по цикловому индексу ci группы g, с формальными переменными x[1], x[2], ... возвращает цикловой индекс парной группы, порожденной группой g.
PairGroupIndex[g, x, Ordered] возвращает цикловой индекс упорядоченной парной группы, порожденной g как полином от x[1], x[2], ....
PairGroupIndex[ci, x, Ordered] по цикловому индексу ci группы g, с формальными переменными x[1], x[2], ..., и возвращает цикловой индекс упорядоченной парной группы, порожденной g.

## ■ Parent

Parent есть аргумент функции AllPairsShortestPath для вывода информации о предшественниках конечных вершин в кратчайших путях. См. С.224

## ■ ParentsToPaths

ParentsToPaths[l, i, j] по списку предшественников l возвращает пути от i до j .
ParentsToPaths[l, i] возвращает пути от i до всех вершин по заданному списку предшественников. См. С.197

## ■ PartialOrderQ

PartialOrderQ[g] возвращает True, если бинарное отношение, определяемое ребрами графа g , есть отношение частичного порядка, т. е. транзитивно, рефлексивно и антисимметрично.
PartialOrderQ[r] возвращает True, если бинарное отношение, определяемое квадратной матрицей r , есть отношение частичного порядка. См. С.240

## ■ PartitionLattice

PartitionLattice[n] возвращает диаграмму Хассе частично упорядоченного множества разбиений множества {1, 2, ..., n}, где p < q , если каждый блок в q содержится в некотором блоке разбиения p. Функция имеет две опции: Type и VertexLabel, с значениями по умолчанию Undirected и False соответственно. Если Type -> Directed, то возвращается ориентированный ациклический граф. Если VertexLabel -> True, то вершины помечаются соответствующими разбиениями. См. С.244

## ■ PartitionQ

PartitionQ[p] возвращает True, если p есть разбиение целого числа.
PartitionQ[n, p] возвращает True, если p есть разбиение целого числа n. См. С.245

## ■ Partitions

Partitions[n] строит все разбиения целого числа n в порядке, обратном лексикографическому.
Partitions[n, k] строит все разбиения целого числа n с числом частей, не превосходящим k, в порядке, обратном лексикографическому. См. С.64

## ■ Path
Path[n] конструирует путь на n вершинах. Допускается опция Type, которая принимает значенияDirected или Undirected. По умолчанию Type-> Directed. См. С. 107.

## ■ PerfectQ
PerfectQ[g] возвращает True, если граф g совершенный, т. е. для каждого подграфа мощность максимальной клики равна хроматическому числу подграфа. См. С.214

## ■ PermutationGraph
PermutationGraph[p] возвращает граф перестановки p. См. С.37

## ■ PermutationGroupQ
PermutationGroupQ[l] возвращает True , если список перестановок l
образует группу. См. С.54

## ■ PermutationQ
PermutationQ[p] возвращает True, если список p есть перестановка, иначе False. См. С.33

## ■ PermutationToTableaux
PermutationToTableaux[p] возвращает пару табло Янга, построенного из p, используя Робинсона-Шенстеда-Кнута соответствие. См. С.81

## ■ PermutationType
PermutationType[p] возвращает тип перестановки p. См. С. 44

## ■ PermutationWithCycle
PermutationWithCycle[n, {i, j, ...}] возвращает перестановку длины n, в которой {i, j, ...} есть цикл и все остальные позиции неподвижны. См. С.43

## ■ Permute
Permute[l, p] переставляет элементы списка l согласно перестановке p. См. С.32

## ■ PermuteSubgraph
PermuteSubgraph [g, p] переставляет вершины вершинно-порожденного подграфа, заданного списком p, согласно перестановке p. См. С.94

## ■ PetersenGraph
PetersenGraph возвращает граф Петерсена, чьи вершины могут рассматриваться
как 2-элементные подмножества множества {1, 2, 3, 4, 5} с ребрами,
соединяющими непересекающие подмножества. См. С.140

## ■ PlanarQ
PlanarQ[g] возвращает True, если граф g планарный, т. е. может быть
изображен на плоскости без пересечения ребер. См. С.256

## ■ Polya
Polya[g, m] возвращает полином степени m (m цветов ), структуры, определяемой группой перестановок g. Лучше использовать OrbitInventory. С. 58

## ■ PseudographQ
PseudographQ[g] возвращает True, если граф g псевдограф, т. е. содержит петли. См. С.83

## ■ RadialEmbedding
RadialEmbedding[g, v] строит радиальное вложение графа g, в котором все вершины располагаются на концентрических окружностях с центром в вершине v в зависимости от их расстояний от v.
RadialEmbedding[g] строит радиальное вложение графа g, где v – центр графа. См. С.108

## ■ Radius
Radius[g] возвращает радиус графа g, минимальный из эксцентриситетов всех вершин. См. С.227

## ■ RandomComposition
RandomComposition[n, k] строит случайную композицию целого n на k частей. См. С.69

## ■ RandomGraph
RandomGraph[n, p] строит случайный помеченный граф с n вершинами и с pn(n-1)/2 ребрами. Опция Type, принимающая значения Directed или Undirected, указывает, ориентирован граф или нет. Значение по умолчанию Type ->Undirected. Type->Directed строит случайный ориентированный граф.
Используйте функцию SetEdgeWeights для присвоения случайных весов. См. С.109

## ■ RandomHeap
RandomHeap[n] строит случайный heap из n элементов (см Heapify). См. С.40

## ■ RandomInteger
RandomInteger есть значение опции WeightingFunction функции SetEdgeWeights. См. С.116

## ■ RandomKSetPartition
RandomKSetPartition[set, k] возвращает случайное разбиение множества set на k частей. RandomKSetPartition[n, k] возвращает случайное разбиение множества
{1,2,...,n} на k частей. См. С.72

## ■ RandomKSubset
RandomKSubset[l, k] возвращает случайное k-элементное подмножество множества l. См. С.50

## ■ RandomPartition
RandomPartition[n] строит случайное разбиение целого числа n. См. С.64

## ■ RandomPermutation
RandomPermutation[n] возвращает случайную перестановку первых n натуральных чисел. См. С.32

## ■ RandomRGF
RandomRGF[n] возвращает случайную ограниченную возрастающую функцию (RGF), определенную на множестве первых n натуральных чисел. RandomRGF[n, k] возвращает случайную RGF определенную на множестве первых n натуральных чисел, с максимальным значением k. См. С.75

## ■ RandomSetPartition
RandomSetPartition[set] возвращает случайное разбиение множества set.
RandomSetPartition[n] возвращает случайное разбиение множества первых n натуральных чисел. См. С.72

## ■ RandomSubset
RandomSubset[l] создает случайное подмножество множества l. См. С.47

## ■ RandomTableau
RandomTableau[p] строит случайное табло Янга формы p. См. С.81

## ■ RandomTree
RandomTree[n] строит случайное помеченное дерево с n вершинами. См. С.109

## ■ RandomVertices
RandomVertices[g] присваивает случайное вложение графу g. См. С.148

## ■RankBinarySubset
RankBinarySubset[l, s] возвращает ранг подмножества s множества l в порядке, полученном интерпретацией подмножества s как двоичного числа. См. С.149

## ■ RankedEmbedding
RankedEmbedding[l] по разбиению l множества вершин {1, 2,..., n} возвращает вложение вершин на плоскости так, что вершины каждого блока располагаются на отдельной вертикали.
RankedEmbedding[g, l] по разбиению l множества вершин графа g возвращает граф g с вершинами, вложенными согласно RankedEmbedding[l].
RankedEmbedding[g, s] располагает вершины из списка s в блоке 1, вершины на расстоянии 1 от любой вершины блока 1 в блоке 2 и т. д. См. С.139

## ■ RankGraph
RankGraph[g, l] разбивает вершины графа g на части, каждый элемент которых наиболее близок к одной из точек из l. См. С.140

## ■ RankGrayCodeSubset
RankGrayCodeSubset[l, s] выдает ранг подмножества s множества l в Грэйкоде l. См. С.50

## ■ RankKSetPartition
RankKSetPartition[sp, s] возвращает ранг sp в списке всех разбиений на k- блоки множества s .
RankKSetPartition[sp] возвращает ранг разбиения sp в списке всех разбиений множества, являющегося объединением элементов sp на k-блоки. См. С.72

## ■ RankKSubset
RankKSubset[s, l] возвращает ранг k-подмножества множества l, где k подмножества перечислены в лексикографическом порядке. См. С.50

## ■ RankPermutation
RankPermutation[p] возвращает ранг перестановки p, где перестановки перечислены в лексикографическом порядке. См. С.33

## ■ RankRGF
RankRGF[f] возвращает ранг ограниченной возрастающей функции (RGF) f в лексикографическом порядке. См. С.75

## ■ RankSetPartition
RankSetPartition[sp, s] возвращает ранг разбиения sp в списке всех разбиений множества s.
RankSetPartition[sp] возвращает ранг разбиения sp в списке всех разбиений множества, являющегося объединением элементов sp. См. С.72

## ■ RankSubset
RankSubset[l, s] возвращает ранг подмножества s множества l в каноническом порядке. См. С.47

## ■ ReadGraph
ReadGraph[f] считывает граф представленный как список ребер из файла f, и возвращает граф.

## ■ RealizeDegreeSequence
RealizeDegreeSequence[s] строит полуслучайный граф с последовательностью степеней (графической последовательностью) s. См. С.128

## ■ ReflexiveQ
ReflexiveQ[g] возвращает True, если матрица смежности графа g представляет рефлексивное бинарное отношение. См. С.132

## ■ RegularGraph
RegularGraph[k, n] строит полуслучайный k-регулярный граф с n вершинами, если таковой существует. См. С.100

## ■ RegularQ
RegularQ[g] возвращает True, если g есть регулярный граф. См. С.100

## ■ RemoveMultipleEdges
RemoveMultipleEdges[g] возвращает граф, полученный удалением кратных ребер из графа g. См. С.137

## ■ RemoveSelfLoops
RemoveSelfLoops[g] возвращает граф, полученный удалением петель из графа g. См. С.137

## ■ ResidualFlowGraph
ResidualFlowGraph[g, flow] возвращает ориентированный граф остаточного потока графа g относительно flow. См. С.236

## ■ RevealCycles
RevealCycles[p]возвращает каноническую циклическую структуру перестановки См. С.41

## ■ ReverseEdges
ReverseEdges[g] меняет ориентацию всех ребер в ориентированном графе g.

## ■ RGFQ
RGFQ[l] возвращает True, если l есть ограниченная возрастающая функция, иначе она возвращает False.См. С.75

## ■ RGFs
RGFs[n] перечисляет все ограниченные возрастающие функции первых n натуральных чисел в лексикографическом порядке. См. С.75

## ■ RGFToSetPartition
RGFToSetPartition[rgf, set] преобразует ограниченную возрастающую функцию
RGF в соответствующее разбиение множества set. См. С.77

## ■ RobertsonGraph
RobertsonGraph возвращает 19-вершинный граф, единственный (4, 5)-клеточный граф. См. С.228

## ■ RootedEmbedding
RootedEmbedding[g, v] строит корневое вложение графа g с корневой вершиной v. RootedEmbedding[g] строит корневое вложение графа g с корнем в центре. См. С.142

## ■ RotateVertices
RotateVertices[v, theta] поворачивает позицию каждой вершины списка v на угол theta (радиан) вокруг начала координат (0, 0). RotateVertices[g, theta] поворачивает вершины графа g на угол theta вокруг начала координат (0, 0). См. С.145

## ■ Runs
Runs[p] разбивает перестановку p на непрерывные возрастающие подпоследовательности. См. С.37

## ■ SamenessRelation
SamenessRelation[l] строит бинарное отношение из списка l перестановок. См. С.54

## ■ SelectionSort
SelectionSort[l, f] сортирует список l по функции f. См. С.32

## ■ SelfComplementaryQ
SelfComplementaryQ[g] возвращает True, если граф g самодополняемый, т. е. он изоморфен своему дополнению. См. С.157

## ■ SelfLoopsQ
SelfLoopsQ[g] возвращает True, если граф g содержит петли.См. С.83

## ■ SetEdgeLabels
SetEdgeLabels[g, l] присваивает метки из l ребрам графа g. Если длина списка l меньше, чем количество ребер в g, то метки присваиваются циклически. Если длина l больше, чем количество ребер в g, то лишние метки игнорируются. См. С.115

## ■ SetEdgeWeights
SetEdgeWeights[g] присваивает случайные веса из диапазона [0, 1] ребрам графа g. SetEdgeWeights принимает опции WeightingFunction и WeightRange. WeightingFunction может иметь значения Random, RandomInteger, Euclidean или LNorm[n] для положительного n, или любую чистую функцию двух аргументов, каждый из которых имеет форму {Integer, {Number, Number}}.
По умолчанию значение для WeightingFunction есть Random и для WeightRange – [0, 1].
SetEdgeWeights[g, e] присваивает веса ребрам списка e.

SetEdgeWeights[g, w] присваивает веса из списка w ребрам графа g. SetEdgeWeights[g, e, w] присваивает веса из списка w ребрам списка e. См. С.116

## ■ SetGraphOptions

SetGraphOptions[g, opts] возвращает g с опциями opts.
SetGraphOptions[g, {v1, v2, ..., vopts}, gopts] возвращает граф с опциями vopts множества вершин v1, v2, ... и gopts есть опции графа g. SetGraphOptions[g, {e1, e2,..., eopts}, gopts] возвращает граф с опциями vopts множества вершин v1, v2, ... и gopts есть опции графа g, а eopts есть опции ребер. Можно указывать перед каждой опцией тег со значениями One или All. По умолчанию принимается значение All и тогда опция применяется ко всем параллельным ребрам. См. С.118

## ■ SetPartitionListViaRGF

SetPartitionListViaRGF[n] перечисляет все разбиения множества {1, 2, ...,n}, вначале перечисляет все ограниченные возрастающие функции
(RGFs) на них и отображает эти RGFs в соответствующие разбиения.
SetPartitionListViaRGF[n, k] перечисляет все RGFs с максимальным элементом k, затем отображает эти RGFs в соответствующие разбиения, каждое из которых состоит из k блоков. См. С.75

## ■ SetPartitionQ

SetPartitionQ[sp, s] возвращает True, если sp есть разбиение множества s.
SetPartitionQ[sp] возвращает True, если sp состоит из непересекающихся множеств. См. С.69

## ■ SetPartitions

SetPartitions[set] возвращает список разбиений множества set. SetPartitions[n] возвращает список разбиений множества {1, 2, ..., n}. Если требуется получить список разбиений множества на определенное число подмножеств, следует использовать функцию KSetPartitions. См. С.69

## ■ SetPartitionToRGF

SetPartitionToRGF[sp, set] преобразует разбиение sp множества set в соответствующую ограниченную возрастающую функцию. См. С.75

## ■ SetVertexLabels

SetVertexLabels[g, l] присваивает метки из l вершинам графа g. Если длина списка l меньше чем количество вершин, то метки присваиваются циклически. Если длина l больше, чем количество вершин, то лишние метки игнорируются. См. С.119

## ■ SetVertexWeights

SetVertexWeights[g] присваивает случайные веса из диапазона [0, 1] вершинам графа g. SetVertexWeights принимает опции WeightingFunction и WeightRange. WeightingFunction может иметь значения Random, RandomInteger или любую чистую функцию двух аргументов, первый из которых есть Integer, а второй есть пара{Number, Number}.
По умолчанию значение для WeightingFunction есть Random и для WeightRange – [0, 1]. См. С.117

## ■ ShakeGraph

ShakeGraph[g, d] выполняет случайное перемещение вершин графа g на расстояние, не большее, чем d. См. С.145

## ■ ShortestPath
ShortestPath[g, start, end] находит кратчайший путь между вершинами start и end в графе g. Опция Algorithm принимает значения Automatic, Dijkstra, или BellmanFord. По умолчанию Algorithm -> Automatic. В этом случае, в зависимости от знаков весов ребер или плотности графа, функция выбирает алгоритм Беллмана-Форда или Дейкстра. См. С.221

## ■ ShortestPathSpanningTree
ShortestPathSpanningTree[g, v] строит остовное дерево кратчайших путей с корнем v так, что кратчайший путь в граф g от вершины v до любой другой вершины есть путь по этому дереву path intree. Опция Algorithm принимает значения Automatic, Dijkstra или BellmanFord. По умолчанию Algorithm -> Automatic. В этом случае, в зависимости от знаков весов ребер или плотности графа, функция выбирает алгоритм Беллмана-Форда или Дейкстра. См. С.218

## ■ ShowGraph
ShowGraph[g] изображает граф g. ShowGraph[g, options] модифицирует изображение согласно списку опций. Опции: VertexColor, VertexStyle, VertexNumber, VertexNumberColor, VertexNumberPosition, VertexLabel, VertexLabelColor, VertexLabelPosition, EdgeColor, EdgeStyle, EdgeLabel, EdgeLabelColor, EdgeLabelPosition, LoopPosition, и EdgeDirection.
Кроме того, используются все опции "Mathematica"- функции Plot и графического примитива Arrow . Эти опции имеют меньший приоритет, чем аналогичные опции объекта g. См. С.89

## ■ ShowGraphArray
ShowGraphArray[{g1, g2, ...}] изображает ряд графов.
ShowGraphArray[{ {g1, ...}, {g2, ...}, ...}] изображает графы в виде 2-мерной таблицы. ShowGraphArray принимает все опции функции ShowGraph и дополнительно опцию GraphicsSpacing -> d. См. С.89

## ■ ShowLabeledGraph
ShowLabeledGraph[g] изображает граф g с номерами вершин в качестве меток.
ShowLabeledGraph[g, l] использует в качестве меток вершин элементы списка l. См. С.89

## ■ ShuffleExchangeGraph
ShuffleExchangeGraph[n] возвращает граф, вершины которого есть двоичные строки длины n и ребрами (w , w') если :
(i) w' отличается от w в последнем бите или
(ii) w' получается из w циклическим сдвигом влево или вправо.
Если нужно прсвоить вершинам соответствующие строки, можно указать опцию VertexLabel -> True. См. С.222

## ■ SignaturePermutation
SignaturePermutation[p] возвращает сигнатуру перестановки p. См. С.35

## ■ Simple
Simple есть значение опции Type. См. С.122

## ▪ SimpleQ
SimpleQ[g] возвращает True, если граф g простой, т. е. не содержит петель и параллельных ребер. См. С.83

## ■ Small
Small есть символ для обозначения размера объектов, таких как вершины. Значениями опции VertexStyle могут быть Disk[Small] или Box[Small]. См. С.84

## ■ SmallestCyclicGroupGraph
SmallestCyclicGroupGraph возвращает наименьший нетривиальный граф с циклической группой автоморфизмов. См. С.253

## ■ Spectrum
Spectrum[g] возвращает cj, собственные значения графа g. См. С.231

## ■ SpringEmbedding
SpringEmbedding[g] изменяет вложение графа g, моделируя систему прыжков.
SpringEmbedding[g, step, increment] используется для усовершенствования алгоритма. Значение step указывает, сколько итераций пробегает алгоритм, а increment указывает расстояние, на которое передвигаются вершины при каждом шаге. По умолчанию значения step и increment равны 10 и 0.15 соответственноСм. С.145

## ■ StableMarriage
StableMarriage[mpref, fpref] находит оптимальный устойчивый марьяж, определенный перестановками, описывающими предпочтения юношей и девушек. См. С.199

## ■ Star
Star[n] строит звезду с n  вершинами. См. С.105

## ■ StirlingFirst
StirlingFirst[n, k] возвращает число Стирлинга первого рода. См. С.74

## ■ StirlingSecond
StirlingSecond[n, k] возвращает число Стирлинга второго рода. См. С.74

## ■ Strings
Strings[l, n] строит все строки длины n из элементов списка l. См. С.52

## ■ StronglyConnectedComponents
StronglyConnectedComponents[g] возвращает сильно связные компоненты  ориентированного графа g как списки вершин. См. С.191

## ■ Strong
Strong есть опция функции ConnectedQ для определения сильной связности ориентированного графа. См. С.191

## ■ Subsets
Subsets[l] возвращает все подмножества множества l. См. С.47

## ■ SymmetricGroup
SymmetricGroup[n] возвращает симметрическую группу Sn. См. С.54

## ■ SymmetricGroupIndex
SymmetricGroupIndex[n, x] возвращает цикловой индекс симметрической группы Sn как полином от x[1], x[2], ..., x[n]. См. С.61

## ■ SymmetricQ
SymmetricQ[r] возвращает True, если матрица r симметрическая.
SymmetricQ[g] возвращает True, если ребра графа g представляют симметричное бинарное отношение. См. С.132

## ■ TableauClasses
TableauClasses[p] разбивает элементы перестановки p на классы в соответствии их первоначальным столбцам во время конструирования табло Янга. См. С.81

## ■ TableauQ
TableauQ[t] возвращает True, если t – табло Янга и False в противном случае. См. С.77

## ■ Tableaux
Tableaux[p] строит все табло Янга с формой, заданной целочисленным разбиением p. См. С.77

## ■ TableauxToPermutation
TableauxToPermutation[t1, t2] возвращает единственную перестановку, ассоциированную с парой табло t1, t2, которые имеют одну и ту же форму. См. С.81

## ■ TetrahedralGraph
TetrahedralGraph возвращает граф тетраэдра. См. С.102

## ■ Thick
Thick есть значение опции EdgeStyle. См. С.92

## ■ ThickDashed
ThickDashed есть значение опции EdgeStyle. См. С.92

## ■ Thin
Thin есть значение опции EdgeStyle. См. С.92

## ■ ThinDashed
ThinDashed есть значение опции EdgeStyle. См. С.92

## ■ThomassenGraph
ThomassenGraph возвращает гипоследный граф– граф g, который не имеет гамильтонова пути, но любой подграф g \ {v}имеет гамильтонов путь для каждой вершины v. См.С.178

## ■ ToAdjacencyLists
ToAdjacencyLists[g] строит списки смежности графа g. Применяется опция Type со значениями All или Simple. Type -> All значение по умолчанию, и отображает петли и кратные ребра в списках смежности. Значение Type -> Simple удаляет петли и параллельные ребра.
ToAdjacencyLists[g, EdgeWeight] возвращает список смежности графа g с весами ребер. См. С.123

## ■ ToAdjacencyMatrix

ToAdjacencyMatrix[g] строит матрицу смежности графа g. Значение опции Type ->Simple указывает на исключение петель и параллельных ребер.
ToAdjacencyMatrix[g, EdgeWeight] возвращает в матрице смежности веса ребер. Веса отсутствующих ребер равны бесконечности. См. С.125

## ■ ToCanonicalSetPartition

ToCanonicalSetPartition[sp,p] переупорядочивает sp в каноническом порядке относительно s. В каноническом порядке элементы каждого подмножества в множестве разбиений упорядочиваются в таком порядке, в каком они появляются в s, а сами подмножества упорядочиваются по их первым элементам.
ToCanonicalSetPartition[sp] переупорядочивает sp в каноническом порядке, предполагая, что Mathematica знает порядок на множестве, для которого sp является множеством разбиения. См. С. 70.

## ■ ToCycles

ToCycles[p] возвращает циклическую структуру перестановки p списком циклов. См. С.41

## ■ ToInversionVector

ToInversionVector[p] возвращает вектор инверсий, ассоциированный с перестановкой p. См. С.35

## ■ ToOrderedPairs

ToOrderedPairs[g] строит список упорядоченных пар, представляющих ребра графа g. Если граф неориентированный, то каждое ребро представляется двумя упорядоченными парами. Опция Type -> Simple удаляет петли и параллельные ребра. Type -> All принято по умолчанию и сохраняет структуру графа. См. С.122

## ■ TopologicalSort

TopologicalSort[g] возвращает перестановку вершин ориентированного ациклического графа g, так что направление ребра (i, j) влечет то, что вершина i предшествует в перестановке вершине j. См. С.245

## ■ ToUnorderedPairs

ToUnorderedPairs[g] строит список неупорядоченных пар, представляющих
ребра графа g. Опция Type -> Simple игнорирует петли и параллельные ребра.
Type -> All принято по умолчанию и сохраняет структуру графа. См. С.122

## ■ TransitiveClosure

TransitiveClosure[g] строит транзитивное замыкание граф g, суперграф графа g,
содержащий ребро {x, y}, если и только если существует путь от вершины x к y. См. С.238

## ■ TransitiveQ

TransitiveQ[g] возвращает True, если граф g определяет транзитивное отношение. См. С.54

## ■ TransitiveReduction

TransitiveReduction[g] находит наименьший граф с таким же транзитивным
замыканием, как у графа g. См. С.239

## ▪TranslateVertice
TranslateVertices[g, v, {x, y}] прибавляет вектор {x, y} к каждой вершине из списка v.
TranslateVertices[g, {x, y}] осуществляет параллельный перенос графа g на вектор {x, y}. См. С.145

## ▪ TransposePartition
TransposePartition[p] берет разбиение p натурального n, состоящее из k частей, и отражает его относительно центрального элемента множества разбиений Partition[n] , создавая разбиение с максимальной частью k. См. С.66

## ▪ TransposeTableau
TransposeTableau[t] отражает табло относительно главной диагонали (строки табло становятся столбцами), конструируя тем самым другое табло. См. С.81

## ▪ TravelingSalesman
TravelingSalesman[g] находит оптимальный маршрут задачи коммивояжера в графе g. См. С.180

## ▪ TravelingSalesmanBounds
TravelingSalesmanBounds[g] возвращает верхнюю и нижнюю границу минимального маршрута задачи коммивояжера в графе g. См. С.180

## ▪ Tree
Tree есть опция некоторых функций для вывода результата в виде дерева. См. С.106

## ▪ TreeIsomorphismQ
TreeIsomorphismQ[t1, t2] возвращает True, если деревья t1 и t2 изоморфны.
Иначе возвращается False. См. С.256

## ▪ TreeQ
TreeQ[g] возвращает True, если граф g есть дерево. См. С.106

## ▪ TreeToCertificate
TreeToCertificate[t] возвращает двоичную строку, однозначно сопоставленную с деревом t. См. С.256

## ▪ TriangleInequalityQ
TriangleInequalityQ[g] возвращает True, если веса ребер графа g удовлетворяют неравенству треугольника.

## ▪ Turan
Turan[n, p] строит граф Турана, экстремального графа с n вершинами, не содержащего полного p-вершинного подграфа. См. С.103

## ▪ TutteGraph
TutteGraph возвращает граф Татта, пример 3-связного, 3-регулярного планарного негамильтонова графа. См. С.180

## ▪ TwoColoring
TwoColoring[g] возвращает 2-раскраску двудольного графа g в виде списка меток 1 и 2 соответствующих вершин. См. С.208

## ■ Type
Type есть опция многих функций. В зависимости от функций она может принимать такие значения, как Directed, Undirected, Simple, и т. д. См. С.60

## ■ UndirectedQ
UndirectedQ[g] возвращает True, если граф g неориентирован. См. С.83

## ■ Undirected
Undirected есть опция некоторых функций для указания того, что граф неориентирован. См. С.60

## ■UnionSet
UnionSet[a, b, s] объединяет подмножества, содержащие a и b в поисковой структуре s.

## ■ Uniquely3ColorableGraph
Uniquely3ColorableGraph возвращает 12-вершинный, без треугольников граф, раскрашиваемый единственным способом в 3 цвета. См. С.207

## ■ UnitransitiveGraph
UnitransitiveGraph возвращает 20-вершинный, 3-унитранзитивныйе граф Коксетера, который неизоморфен 4-клеточному или 5-клеточному. См. С.257

## ■ UnrankBinarySubset
UnrankBinarySubset[n, l] возвращает n-е подмножество множества l, в порядке возрастания двоичных чисел, представляющих подмножества. См. С.49

## ■UnrankGrayCodeSubset
UnrankGrayCodeSubset[n, l] возвращает n-е подмножество множества l, в порядке возрастания кода Грэя. См. С.50

## ■ UnrankSetPartition
UnrankSetPartition[r, s, k] находит k-блок разбиения ранга r множества s.
UnrankSetPartition[r, n, k] находит k-блок разбиения ранга r множества {1, 2,..., n}. См. С.74

## ■ UnrankKSubset
UnrankKSubset[m, k, l] возвращает m-ое, в лексикографическом порядке, k-подмножество множества l. См. С.50

## ■ UnrankPermutation
UnrankPermutation[r, l] возвращает r-ую, в лексикографическом порядке, перестановку множества l.
UnrankPermutation[r, n] возвращает r-ую, в лексикографическом порядке, перестановку множества {1,2,...,n}. См. С.33

## ■ UnrankRGF
UnrankRGF[r, n] возвращает ограниченную возрастающую функцию
ранга r на множестве {1,2,...,n}. См. С.75

## ■ UnrankSetPartition
UnrankSetPartition[r, set] возвращает разбиение множества set ранга r.
UnrankSetPartition[r, n] возвращает разбиение множества {1,2,...,n} ранга r. См. С.74

## ■ UnrankSubset
UnrankSubset[n, l] возвращает n-ое, в каноническом порядке, подмножество множества l. См. С.47

## ■ UnweightedQ
UnweightedQ[g] возвращает True, если все ребра имеют вес 1 и False иначе. См. С.117

## ■ UpperLeft
UpperLeft есть значение опций VertexNumberPosition, VertexLabelPosition и EdgeLabelPosition функции ShowGraph. См. С.84

## ■ UpperRight
UpperRight есть значение опций VertexNumberPosition, VertexLabelPosition и EdgeLabelPosition функции ShowGraph. См. С.84

## ■ V
V[g] возвращает количество вершин графа g. См. С.114

## ■ Value
Value есть опция функции NetworkFlow для возврата значения максимального потока. См. С.295

## ■ VertexColor
VertexColor есть опция, позволяющая устанавливать цвета вершин. Цвет по умолчанию - Black. VertexColor может быть установлена как часть структуры графа или в функции ShowGraph. См. С.84

## ■ VertexColoring
VertexColoring[g] использует эвристический алглритм Брелаза для нахождения не обязательно минимальной, но достаточно хорошей раскраски вершин графа g. При установке опции Algorithm -> Optimum применяется полный перебор для нахождения минимальной раскраски. См. С.205

## ■ VertexConnectivity
VertexConnectivity[g] возвращает минимальное число вершин, удаление которых приводит к несвязному графу.
VertexConnectivity[g, Cut] возвращает множество вершин минимальной мощности, удаление которых приводит к несвязному графу. См. С.182

## ■ VertexConnectivityGraph
VertexConnectivityGraph[g] возвращает ориентированный граф, содержащий ребро для каждой вершины графа g, и в котором реберно-непересекающиеся пути соответствуют вершинно-непересекающимся путям в g. См. С.188

## ■ VertexCover
VertexCover[g] возвращает вершинное покрытие графа g. Опция Algorithm, принимающая значения Greedy, Approximate или Optimum, задает алгоритм вычисления. По умолчанию принимается Algorithm -> Approximate. См. С.200

## ■ VertexCoverQ
VertexCoverQ[g, c] возвращает True, если вершины списка c
образуют вершинное покрытие графа g. См. С.200

## ■ VertexLabel

VertexLabel есть опция со значениями True или False, для меток вершин. По умолчанию, вершины не помечены. VertexLabel может быть установлена как часть структуры графа или в функции ShowGraph. См. С.84

## ■ VertexLabelColor

VertexLabelColor есть опция, позволяющая устанавливать цвета меткам вершин. Цвет по умолчанию - Black.
VertexLabelColor может быть установлена как часть структуры графа или в функции ShowGraph. См. С.84

## ■ VertexLabelPosition

VertexLabelPosition есть опция, позволяющая устанавливать позиции меток вершин относительно положения вершин.
По умолчанию VertexLabelPosition –> Upper Right. VertexLabelPosition может быть установлена как часть структуры графа или в функции ShowGraph. См. С.84

## ■ VertexNumber

VertexNumber есть опция, принимающая значения True (изображает номера вершин) или False. По умолчанию номера скрыты. VertexNumber может быть установлена как часть структуры графа или в функции ShowGraph. См. С.84

## ■ VertexNumberColor

VertexNumberColor есть опция, позволяющая устанавливать цвета номеров вершинt. Black - цвет по умолчанию. VertexNumberColor может быть установлена как часть структуры графа или в функции ShowGraph. См. С.84

## ■ VertexNumberPosition

VertexNumberPosition есть опция, позволяющая устанавливать позиции номеров вершин относительно положения вершины. По умолчанию номер вершины изображается снизу и слева от вершины. VertexNumberPosition может быть установлена как часть структуры графа или в функции ShowGraph. См. С.84

## ■ VertexStyle

VertexStyle есть опция, определяющая формы вершин. По умолчанию форма вершины есть круг. VertexStyle может быть установлена как часть структуры графа или в функции ShowGraph. См. С.84

## ■ VertexWeight

VertexWeight есть опция, сопоставляющая вершинам веса. 0 – значение по умолчанию. VertexWeight может быть частью структуры графа.См. С.84

## ■ Vertices

Vertices[g] возвращает координаты вершин графа g. Vertices[g, All] возвращает координаты вершин графа g вместе с графическими опциями. См. С.114

## ■ WaltherGraph

WaltherGraph возвращает граф Уолтера.

## ■ Weak
Weak есть опция функции ConnectedQ для проверки ориентированного графа на слабую связность. См. С.191

## ■ WeaklyConnectedComponents
WeaklyConnectedComponents[g] возвращает слабосвязные компоненты ориентированного графа g как списки вершин. См. С.191

## ■ WeightingFunction
WeightingFunction есть опция функций SetEdgeWeights и SetVertexWeights и указывает, как вычислять веса ребер и вершин соответственно. По умолчанию значение этой опции -> Random. См. С.116

## ■ WeightRange
WeightRange есть опция функций SetEdgeWeights и SetVertexWeights, указывающая диапазон весов. По умолчанию диапазон [0, 1] принимается как для вещественных, так и для целых чисел. См. С.116

## ■ Wheel
Wheel[n] строит граф колеса на n вершинах, соединение CompleteGraph[1] и Cycle[n-1]. См. С.105

## ■ WriteGraph
WriteGraph[g, f] записывает граф g в файл f как список ребер.

## ■ Zoom
Zoom[{i, j, k, ...}] есть значение опции PlotRange функции ShowGraph и позволяет увеличить изображения для того, чтобы на нем поместились вершины i, j, k, .... См. С.89, 261

# Библиографический список

1.Агапова О.И., Кривошеев А.О., Ушаков А.С. О трех поколениях компьютерных технологий обучения // Информатика и образование.- 1994.- №2.- С.34-40.

2.Акимов О.Е. Дискретная математика: логика, группы, графы, фракталы. – М.: Издатель Акимова, 2005. – 656 с.

3.Аладьев В.З., Шишаков М.Л. Введение в среду пакета Mathematica 2.2. – М.: Информационно-издательский дом «Филин», 1997.–368 с.

4. Апатова Н.В. Влияние информационных технологий на содержание и методы обучения в средней школе: дис. докт. пед. наук.- М., 1994.

5.Апатова Н.В. Информационные технологии в школьном образовании.-М., 1994. – 228 с.

6.Архангельский С.И. Лекции по научной организации учебного процесса в высшей школе. – М.: Высшая школа, 1976.– 200 с.

7.Ахо А., Хопкрофт Дж, Ульман Дж. Построение и анализ вычислительных алгоритмов. – М.: Мир, 1979.

8.Берже М. Геометрия. – М.: Мир, 1984. Ч.1. – С.5.

9.Беспалько В.П. Образование и обучение с участием компьютеров (Педагогика третьего тысячелетия).- М.: Издательство Московского психолого-социального института; Воронеж: Издательство НПО «МОТЭК», 2002. – 352 с. [серия «Библиотека педагога-практика»].

10.Бордковский Г.А, Извозчиков В.А. Новые технологии обучения. Вопросы терминологии//Педагогика.- 1993.-№5. –С.12-15.

11. Булгаков М.В., Пушкин А.Е., Фомин С.С Технологические аспекты создания компьютерных обучающих программ//Компьютерные технологии в высшем образовании/ ред. кол.: А.Н. Тихонов, В.В. Садовничий и др. – М.: Изд-во Московского университета, 1994. –С.147-152.

12. Вирт Н. Алгоритмы + структуры данных = программы. – М.: Мир, 1985. –280 с.

13. Воробьев Е.М. Введение в систему «Mathematica »: учеб. пособие. –М.:Финансы и статистика, 1998. – 262 с.

14. Гаврилов Г., Сапоженко А. Задачи и упражнения по дискретной математике. – М.:Физматлит, 2005. – 156 с.

15. Гарднер М. Крестики – Нолики. – М.: Мир, 1988. – 300 с.

16. Гнеденко Б.В., Гнеденко Д.Б. Об обучении математике в университетах и педвузах на рубеже двух тысячелетий. -Изд.3-е, испр. и доп. – М.: Дом. книга, 2006. -160 с. [Психология, педагогика, технология обучения: математика]

17. Головешкин В., Ульянов М. Теория рекурсии для программистов. – М.: Физматлит, 2006. – 200 с.

18. Государственный образовательный стандарт высшего профессионального образования. – М., 2004.

19. Грэхем Р., Кнут Д., Поташник О. Конкретная математика. Основание информатики. – М.: Мир, 1998.

20. Гэри М., Джонсон Д. Вычислительные машины и труднорешаемые задачи. – М.: Мир, 1982.

21. Долинский М. Решение сложных и олимпиадных задач по программированию. – СПб: Питер, 2006.

22. Дьяконов В. Mathematica 4: учеб. курс. – СПб: Питер, 2001. – 654 с.

23. Евстигнеев В., Мельников Л. Задачи и упражнения по теории графов и комбинаторике. – Новосибирск: Изд-во НГУ, 1981.

24. Емеличев В.А., Мельников О.И., Сарванов В.И., Тышкевич Р.И. Лекции по теории графов.– М.: Наука, 1990.– 384 с.

25. Зыков А. Основы теории графов. – М.: Наука, 1987.

26. Капустина Т.В. Компьютерная система Mathematica 3.0 в вузовском образовании.-М.: Солон, 2000.

27. Кнут. Искусство программирования на ЭВМ. Т. 1. Основные алгоритмы. – М.: Мир, 1976.

28. Кнут. Искусство программирования на ЭВМ. Т. 3. Сортировка и поиск. – М.: Мир, 1978.

29. Кострикин А.И. Введение в алгебру.-М: Наука, 1977.

30. Кристофидес Н. Теория графов: Алгоритмический подход. – М.: Мир, 1978.

31.Лавров И., Максимова Л. Задачи по теории множеств, математической логике и теории алгоритмов. – М.: Физматлит, 2001.

32. Машбиц Е.И. Компьютеризация обучения: проблемы и перспективы. – М.: Знание, 1986.-80 с.

33.Машбиц Е.И. Психолого-педагогические проблемы компьютеризации обучения. –М.: Педагогика, 1988. –192 с.

34.Монахов В.М.Проектирование и внедрение новых технологий обучения // Советская педагогика, 1990, №7. – С. 17-23.

35. Окулов С. Программирование в алгоритмах. – М.: БИНОМ. Лаборатория знаний, 2002.

36. Оре О. Теория графов. – М.: Мир, 1984.

37. Пойа Д. Как решать задачу. – М.: Мир, 1961.

38. Препарата Ф., Шеймос М. Вычислительная геометрия: Введение. – М.: Мир, 1989.

39.Роберт И.В. Современные информационные технологии в образовании: дидактические проблемы; перспективы использования. – М.: Школа-Пресс, 1994. – 205 с.

40. Свами М., Тхуласираман К. Графы, сети и алгоритмы.–М.: Мир, 1984. – 455 с.

41.Свириденко С.С Информационные технологии в интеллектуальной деятельности: учеб. пособие. –М.: Изд-во МНЭПУ, 1995. –239 с.

42. Сигал И.Х., Иванова А.П. Введение в прикладное дискретное программирование: модели и вычислительные алгоритмы: учеб. пособие. – М.: ФИЗМАТЛИТ, 2002. – 240 с.

43. Хаггарти З. Дискретная математика для программистов. – М.: Техносфера, 2005.

44. Харари Ф. Теория графов. – М.: Мир, 1973.

45. Компьютер и задачи выбора. – М.: Наука, 1989.

46. Перечислительные задачи комбинаторного анализа. – М.: Наука, 1979.

47. Шапорев С.Д. Дискретная математика. Курс лекций и практических занятий. – СПб.: БХВ=Петербург, 2006. – 400с.

48.Pemmaraju S., Skiena S. Computational Discrete Mathematics. – Cambridge Univercity Press, 2003.

49. Skiena S., Revilla M. Programming Challenges.– Springer, 2005.

50.Wolfram S. Mathematica: A System for Doing Mathematics by Computer.-Addison-Wesley Publishing Company, 1988.

51.Wolfram S. Mathematica: A System for Doing Mathematics by Computer. Second Edition.-Addison-Wesley Publishing Company, 1991.

52. Wolfram S. The Mathematica Book: Third Edition. Mathematica Version 3.– Cambridge University Press, 1998.

53.Wolfram S. The Mathematica Book: Fourth Edition. Mathematica Version 4.– Cambridge University Press, 1999.

# Оглавление

## Глава 5. Построение графов



## Глава 6. Анализ графов

## Глава 7. Оптимизационные алгоритмы на графах



## Глава 8. Примеры решения задач