

Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего профессионального образования

«Владимирский государственный университет

им. А. Г. и Н. Г. Столетовых» (ВлГУ)

А. В. ШУТОВ

Ю.А. МЕДВЕДЕВ

СТРУКТУРЫ И АЛГОРИТМЫ КОМПЬЮТЕРНОЙ

ОБРАБОТКИ ДАННЫХ

Часть 1. Курс лекций

по дисциплине «Структуры и алгоритмы компьютерной обработки данных» для студентов, обучающихся по направлению 010500 «Математическое обеспечение и администрирование информационных систем»

Владимир – 2013

УДК 004.31

ББК 32.988-5я7

Ш 97

**Шутов А. В., Медведев Ю. А.**

Структуры и алгоритмы компьютерной обработки данных. Часть 1 (Курс лекций). – Владимир: ВлГУ, 2013. – 101 с.

Учебное пособие адресовано студентам вузов, обучающимся по направлению 010500 «Математическое обеспечение и администрирование информационных систем».

Курс включает 9 лекция по 3 темам: алгоритмы на графах, алгоритмы комбинаторного перебора, общие методы разработки алгоритмов. Материал систематизирован и может быть использован студентами физико-математических факультетов вузов.

**Рецензенты:** доктор технических наук, профессор **Монахов М. Ю.**, зав. кафедрой информатикии защитыинформацииВлГУ;

доктор физико-математических наук, профессор ВлГУ

**Алхутов Ю. А.**

*Печатается по решению Редакционно-*

*издательского совета ВлГУ*

© ФГБОУ ВПО «Владимирский государственный университет», 2013

© Шутов А. В., Медведев Ю. А., 2013

**ВВЕДЕНИЕ.**

1. **ЦЕЛИ ОСВОЕНИЯ ДИСЦИПЛИНЫ**

Целью освоения дисциплины «Структуры и алгоритмы компьютерной обработки данных» является формирование системы понятий, знаний, умений и навыков в области использования классических алгоритмов и структур данных.

1. **МЕСТО ДИСЦИПЛИНЫ В СТРУКТУРЕ ООП ВПО**

Раздел образовательной программы: Б.3. Профессиональный цикл. Базовая часть.

Для изучения курса необходимы начальные знания по следующим дисциплинам:

- информатика,

- программирование,

- алгебра,

- дискретная математика,

- математический анализ.

Для того чтобы приступить к изучению курса «Структуры и алгоритмы компьютерной обработки данных», студент должен знать:

- основные понятия алгебры, математического анализа, дискретной математики и информатики;

- основные управляющие структуры,

- один из языков программирования.

Знания и умения, полученные в ходе освоения данной дисциплины, понадобятся при изучении таких последующих дисциплин ООП, как:

- объектно-ориентированное программирование;

- параллельное программирование;

- базы данных и СУБД;

- технология разработки ПО.

1. **КОМПЕТЕНЦИИ ОБУЧАЮЩЕГОСЯ, ФОРМИРУЕМЫЕ В РЕЗУЛЬТАТЕ ОСВОЕНИЯ ДИСЦИПЛИНЫ (МОДУЛЯ)**

В результате освоения данной дисциплины формируются следующие компетенции:

ОК-10 – демонстрировать фундаментальную подготовку по основам профессиональных знаний;

ОК-12 – демонстрировать владение основными методами, способами и средствами получения, хранения, переработки информации, иметь навыки работы с компьютером как средством управления информацией;

ОК-14 – демонстрировать способность к анализу и синтезу;

ПК-2 – демонстрировать умение понять поставленную задачу;

ПК-3 – демонстрировать умение формулировать результат;

ПК-7 – демонстрировать умение грамотно пользоваться языком предметной области;

ПК-11 – демонстрировать самостоятельное построение алгоритма и его анализ;

ПК-14 – демонстрировать контекстную обработку информации.

В результате освоения дисциплины обучающийся должен:

Знать:

• основные абстрактные типы данных и способы их реализации

• основные алгоритмы работы с деревьями

• основные алгоритмы сортировки и поиска

• основные алгоритмы комбинаторного перебора

• основные алгоритмы работы с графами

• базовые методы разработки алгоритмов

• основные результаты и проблемы теории сложности вычислений

Уметь:

• реализовывать на одном из языков программирования основные абстрактные типы данных

• реализовывать на одном из языков программирования основные алгоритмы работы с деревьями

• реализовывать на одном из языков программирования основные алгоритмы сортировки и поиска

• реализовывать на одном из языков программирования основные алгоритмы комбинаторного перебора

• реализовывать на одном из языков программирования основные алгоритмы работы с графами

• использовать базовые методы разработки алгоритмов

**Тема 1. Алгоритмы на графах (6 часов).**

План лекций.

1. Начальные понятия теории графов.

Начальные понятия теории графов. Определение графа. Графы и бинарные отношения. Откуда берутся графы. Число графов. Смежность, инцидентность, степени. Некоторые специальные графы. Графы и матрицы. Взвешенные графы. Изоморфизм. Инварианты. Операции над графами. Локальные операции. Подграфы. Алгебраические операции.

1. Поиск в глубину и ширину.

Поиск в ширину. Процедура поиска в ширину. BFS-дерево и вычисление расстояний. Процедура поиска в глубину. DFS-дерево. Глубинная нумерация. Построение каркаса. Шарниры.

1. Эйлеровы и гамильтоновы циклы.

Маршруты, пути, циклы. Связность и компоненты. Метрические характеристики графов. Маршруты и связность в орграфах. Эйлеровы пути и циклы. Построение эйлерова цикла. Гамильтоновы пути и циклы.

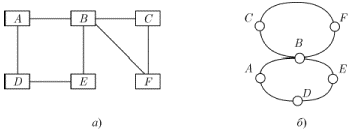
**Лекция 1. Начальные понятия теории графов.**

Графы являются существенным элементом математических моделей в самых разнообразных областях науки и практики. Они помогают наглядно представить взаимоотношения между объектами или событиями в сложных системах. Многие алгоритмические задачи дискретной математики могут быть сформулированы как задачи, так или иначе связанные с графами, например задачи, в которых требуется выяснить какие-либо особенности устройства графа, или найти в графе часть, удовлетворяющую некоторым требованиям, или построить граф с заданными свойствами.

Цель этой и двух следующих лекций - дать краткое введение в теорию графов. В них приводится минимум понятий, необходимый для того, чтобы можно было начать какую-либо содержательную работу с графами или приступить к более глубокому изучению теории графов. Доказательства приводятся только в тех случаях, когда их сложность не превышает некоторого интуитивно ощущаемого порога. Поэтому, например, такие важные факты, как теорема Кирхгофа или теорема Понтрягина-Куратовского, сообщаются без доказательств.

### Определение графа

Для описания строения различных систем, состоящих из связанных между собой элементов, часто используют графические схемы, изображая элементы точками (кружками, прямоугольниками и т.д.), а связи между ними - линиями или стрелками, соединяющими элементы. При этом получаются диаграммы вроде тех, что показаны на [рис. 1.1](http://localhost:3232/department/algorithms/gaa/1/gaa_1.html#image.1.1#image.1.1).



**Рис. 1.1.**

На таких диаграммах часто ни способ изображения элементов, ни форма или длина линий не имеют значения - важно лишь, какие именно пары элементов соединены линиями. Если посмотреть внимательно, то можно заметить, что [рисунки 1.1а](http://localhost:3232/department/algorithms/gaa/1/gaa_1.html#image.1.1#image.1.1) и [1.1 б](http://localhost:3232/department/algorithms/gaa/1/gaa_1.html#image.1.1#image.1.1) изображают одну и ту же структуру связей между элементами A, B, C, D, E, F. Эту же структуру можно описать, не прибегая к графическому изображению, а просто перечислив пары связанных между собой элементов: (A,B), (A,D), (B,C), (B,E), (B,F), (C,F), (D,E). Таким образом, когда мы отвлекаемся от всех несущественных подробностей, у нас остаются два списка: список элементов и список пар элементов. Вместе они составляют то, что математики называют графом. Из этого примера видно, что понятие графа само по себе не связано напрямую с геометрией или графикой. Тем не менее, возможность нарисовать граф - одна из привлекательных черт этого математического объекта.

Термин "граф" неоднозначен, это легко заметить, сравнивая приводимые в разных книгах определения. Однако во всех этих определениях есть кое-что общее. В любом случае граф состоит из двух множеств - множества вершин и множества ребер, причем для каждого ребра указана пара вершин, которые это ребро соединяет. Вершины и ребра называются элементами графа. Здесь будут рассматриваться только конечные графы, то есть такие, у которых оба множества конечны. Чтобы получить законченное определение графа того или иного типа, необходимо уточнить еще три момента.

1. Ориентированный или неориентированный?

Прежде всего, нужно договориться, считаем ли мы пары \left(a,b\right)и \left(b,a\right)различными. Если да, то говорят, что рассматриваются упорядоченные пары (порядок элементов в паре важен), если нет - неупорядоченные. Если ребро eсоединяет вершину aс вершиной bи пара \left(a,b\right)считается упорядоченной, то это ребро называется ориентированным, вершина a- его началом, вершина b- концом. Если же эта пара считается неупорядоченной, то ребро называется неориентированным, а обе вершины - его концами. Чаще всего рассматривают графы, в которых все ребра имеют один тип - либо ориентированные, либо неориентированные. Соответственно и весь граф называют ориентированным или неориентированным. На рисунках ориентацию ребра (направление от начала к концу) указывают стрелкой. На [рис. 1.1](http://localhost:3232/department/algorithms/gaa/1/gaa_1.html#image.1.1#image.1.1) показаны неориентированные графы, а на [рис. 1.2](http://localhost:3232/department/algorithms/gaa/1/gaa_1.html#image.1.2#image.1.2) - ориентированные.

1. Кратные ребра.

Следующий пункт, требующий уточнения, - могут ли разные ребра иметь одинаковые начала и концы? Если да, то говорят, что в графе допускаются кратные ребра. Граф с кратными ребрами называют также мультиграфом. На [рис. 1.2](http://localhost:3232/department/algorithms/gaa/1/gaa_1.html#image.1.2#image.1.2) изображены два графа, левый является ориентированным мультиграфом, а правый - ориентированным графом без кратных ребер.

1. Петли.

Ребро, которому поставлена в соответствие пара вида \left(a,a\right), то есть ребро, соединяющее вершину aс нею же самой, называется петлей. Если такие ребра не допускаются, то говорят, что рассматриваются графы без петель.



**Рис. 1.2.**

Комбинируя эти три признака, можно получить разные варианты определения понятия графа. Особенно часто встречаются неориентированные графы без петель и кратных ребер. Такие графы называют обыкновенными. Если в графе нет кратных ребер, то можно просто отождествить ребра с соответствующими парами вершин - считать, что ребро это и есть пара вершин. Чтобы исключить петли, достаточно оговорить, что вершины, образующие ребро, должны быть различны. Это приводит к следующему определению обыкновенного графа.

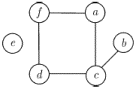
**Определение.** Обыкновенным графом называется пара G=\left(V,E\right), где V- конечное множество, E- множество неупорядоченных пар различных элементов из V. Элементы множества Vназываются вершинами графа, элементы множества E- его ребрами.

Слегка модифицируя это определение, можно получить определения других типов графов без кратных ребер: если заменить слово "неупорядоченных" словом "упорядоченных", получится определение ориентированного графа без петель, если убрать слово "различных", получится определение графа с петлями. Ориентированный граф часто называют орграфом.

В дальнейшем термин "граф" мы будем употреблять в смысле "обыкновенный граф", а рассматривая другие типы графов, будем специально это оговаривать.

Множество вершин графа Gбудем обозначать через VG, множество ребер - EG, число вершин - n\left(G\right), число ребер - m\left(G\right).

Из определения видно, что для задания обыкновенного графа достаточно перечислить его вершины и ребра, причем каждое ребро должно быть парой вершин. Положим, например, VG = \{a, b, c, d, e, f\}, EG \!=\! \{(a, c), (a, f), (b, c), (c, d), (d, f)\}. Тем самым задан граф Gс n(G)\!=\!6, m\left(G\right)=5. Если граф не слишком велик, то более наглядно представить его можно с помощью рисунка, на котором вершины изображаются кружками или иными значками, а ребра - линиями, соединяющими вершины. Заданный выше граф Gпоказан на [рисунке 1.3](http://localhost:3232/department/algorithms/gaa/1/gaa_1.html#image.1.3#image.1.3). Мы будем часто пользоваться именно этим способом представления графа, при этом обозначения вершин иногда будут помещаться внутри кружков, изображающих вершины, иногда рядом с ними, а иногда, когда имена вершин несущественны, и вовсе опускаться.



**Рис. 1.3.**

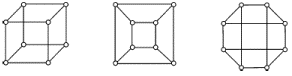
### Графы и бинарные отношения

Напомним, что бинарным отношением на множестве Aназывается любое подмножество Rмножества A^{2}, состоящего из всевозможных упорядоченных пар элементов множества A. Каждому такому отношению можно поставить в соответствие граф отношения G=(A,R). Сравнивая с тем, что говорилось выше об определениях различных типов графов, видим, что понятие бинарного отношения эквивалентно понятию ориентированного графа с петлями. Другие типы графов без кратных ребер - это частные виды бинарных отношений. Отношение Rназывается рефлексивным, если для любого x\in Aпара (x,x)принадлежит R, и антирефлексивным, если ни одна такая пара не принадлежит R. Отношение называется симметричным, если из \left(x,y\right)\in Rследует, что \left(y,x\right)\in
R. В графе антирефлексивного и симметричного отношения нет петель и для каждой пары вершин либо нет ни одного, либо есть два ребра, соединяющих эти вершины. Если в таком графе каждую пару ориентированных ребер, соединяющих одни и те же две вершины, заменить одним неориентированным ребром, то получится обыкновенный граф.

### Откуда берутся графы

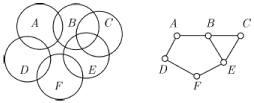
Легко найти примеры графов в самых разных областях науки и практики. Сеть дорог, трубопроводов, электрическая цепь, структурная формула химического соединения, блок-схема программы - в этих случаях графы возникают естественно и видны "невооруженным глазом". При желании графы можно обнаружить практически где угодно. Это наглядно показано в книге Д.Кнута [D.E.Knuth, "The Stanford GraphBase"] - графы извлекаются из романа "Анна Каренина", из картины Леонардо да Винчи, из материалов Бюро Экономического Анализа США и из других источников.

Немало поводов для появления графов и в самой математике. Наиболее очевидный пример - любой многогранник в трехмерном пространстве. Вершины и ребра многогранника можно рассматривать как вершины и ребра графа. При этом мы отвлекаемся от того, как расположены элементы многогранника в пространстве, оставляя лишь информацию о том, какие вершины соединены ребрами. На [рис. 1.4](http://localhost:3232/department/algorithms/gaa/1/gaa_1.html#image.1.4#image.1.4) показаны три способа изобразить один и тот же граф трехмерного куба.



**Рис. 1.4.**

Еще один способ образования графов из геометрических объектов иллюстрирует [рис. 1.5](http://localhost:3232/department/algorithms/gaa/1/gaa_1.html#image.1.5#image.1.5). Слева показаны шесть кругов на плоскости, а справа - граф, в котором каждая вершина соответствует одному из этих кругов и две вершины соединены ребром в том и только том случае, когда соответствующие круги пересекаются. Такие графы называют графами пересечений. Можно построить граф пересечений семейства интервалов на прямой, или дуг окружности, или параллелепипедов. Вообще, для любого семейства множеств \{S_{1}\dts S_{n}\}можно построить граф пересечений с множеством вершин \{1\dts n\}, в котором ребро (i,j)имеется тогда и только тогда, когда i\ne jи S_{i} \cap S_{j} \ne
\varnothing. Известно, что любой граф можно представить как граф пересечений некоторого семейства множеств.



**Рис. 1.5.**

### Число графов

Возьмем какое-нибудь множество V, состоящее из nэлементов, и будем рассматривать всевозможные (обыкновенные!) графы с множеством вершин V. Обозначим число таких графов через g_{n}. Эти графы различаются только множествами ребер, а каждое ребро - это неупорядоченная пара различных элементов из V. В комбинаторике такие пары называются сочетаниями из nпо 2, их число равно

\begin{pmatrix}{n}\\
{2} \end{pmatrix}=\frac{n\left(n-1\right)}{2}

Каждая пара может быть включена или не включена в множество ребер графа. Применяя правило произведения, приходим к следующему результату:

**Теорема 1.** g_{n} =2^{n(n-1)/2}.

### Смежность, инцидентность, степени

Если в графе имеется ребро e=(a,b), то говорят, что вершины aи bсмежны в этом графе, ребро eинцидентно каждой из вершин a, b, а каждая из них инцидентна этому ребру.

Множество всех вершин графа, смежных с данной вершиной a, называется окрестностью этой вершины и обозначается через V(a).

На практике удобным и эффективным при решении многих задач способом задания графа являются так называемые списки смежности. Эти списки могут быть реализованы различными способами в виде конкретных структур данных, но в любом случае речь идет о том, что для каждой вершины aперечисляются все смежные с ней вершины, т.е. элементы множества V(a). Такой способ задания дает возможность быстрого просмотра окрестности вершины.

Число вершин, смежных с вершиной a, называется степенью вершины aи обозначается через \deg
\left(a\right).

Если сложить степени всех вершин некоторого графа, то каждое ребро внесет в эту сумму вклад, равный 2, поэтому справедливо следующее утверждение:

**Теорема 2.** \suml_{a\in VG}\deg (a) =2m\left(G\right).

Это равенство известно как "лемма о рукопожатиях". Из него следует, что число вершин нечетной степени в любом графе четно.

Вершину степени 0называют изолированной.

Граф называют регулярным степени d, если степень каждой его вершины равна d.

Набор степеней графа - это последовательность степеней его вершин, выписанных в неубывающем порядке.

### Некоторые специальные графы

Рассмотрим некоторые особенно часто встречающиеся графы.

Пустой граф - граф, не содержащий ни одного ребра. Пустой граф с множеством вершин \{1,2\dts n\}обозначается через O_{n}.

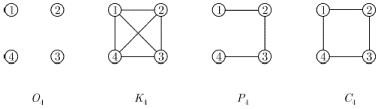
Полный граф - граф, в котором каждые две вершины смежны. Полный граф с множеством вершин \{1,2\dts n\}обозначается через K_{n}.

Граф K_{1}, в частности, имеет одну вершину и ни одного ребра. Очевидно, K_{1} =O_{1}. Будем считать также, что существует граф K_{0}, у которого VG=EG=\varnothing.

Цепь(путь) P_{n}- граф с множеством вершин \{1,2\dts n\}и множеством ребер \left\{(1,2),(2,3)\dts (n-1,n)\right\}.

Цикл C_{n}- граф, который получается из графа P_{n}добавлением ребра (1,n).

Все эти графы при n=4показаны на [рис. 1.6](http://localhost:3232/department/algorithms/gaa/1/gaa_1.html#image.1.6#image.1.6)



**Рис. 1.6.**

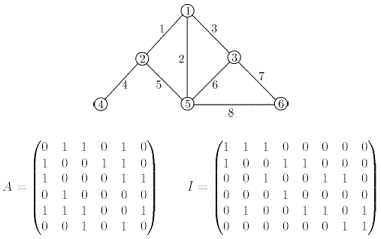
### Графы и матрицы

Пусть G- граф с nвершинами, причем VG=\left\{1,2\dts n\right\}. Построим квадратную матрицу Aпорядка n, в которой элемент A_{ij}, стоящий на пересечении строки с номером iи столбца с номером j, определяется следующим образом:


\eq*{
A_{ij} =\left\{\begin{aligned} & {1,\ \t{если } \left(i,j\right)\in EG,}
\\ & {0,\ \t{если } \left(i,j\right)\notin EG.}
\end{aligned}
\right.
}


Она называется матрицей смежности графа. Матрицу смежности можно построить и для ориентированного графа, и для неориентированного, и для графа с петлями. Для обыкновенного графа она обладает двумя особенностями: из-за отсутствия петель на главной диагонали стоят нули, а так как граф неориентированный, то матрица симметрична относительно главной диагонали. Обратно, каждой квадратной матрице порядка n, составленной из нулей и единиц и обладающей двумя указанными свойствами, соответствует обыкновенный граф с множеством вершин \left\{1,2\dts
n\right\}.

Другая матрица, ассоциированная с графом - это матрица инцидентности. Для ее построения занумеруем вершины графа числами от 1 до n, а ребра - числами от 1 до m. Матрица инцидентности Iимеет nстрок и mстолбцов, а ее элемент I_{ij}равен 1, если вершина с номером iинцидентна ребру с номером j, в противном случае он равен нулю. На [рис. 1.7](http://localhost:3232/department/algorithms/gaa/1/gaa_1.html#image.1.7#image.1.7) показан граф с занумерованными вершинами и ребрами и его матрицы смежности и инцидентности.



**Рис. 1.7.**

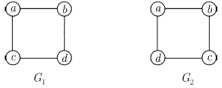
Для ориентированного графа матрица инцидентности определяется несколько иначе: ее элемент I_{ij}равен 1, если вершина iявляется началом ребра j, и равен -1, если она является концом этого ребра, и он равен 0, если эта вершина и это ребро не инцидентны друг другу.

### Взвешенные графы

Часто, особенно когда графы используются для моделирования реальных систем, их вершинам, или ребрам, или и тем, и другим приписываются некоторые числа. Природа этих чисел может быть самая разнообразная. Например, если граф представляет собой модель железнодорожной сети, то число, приписанное ребру, может указывать длину перегона между двумя станциями, или наибольший вес состава, который допустим для этого участка пути, или среднее число поездов, проходящих через этот участок в течение суток и т.п. Что бы ни означали эти числа, сложилась традиция называть их весами, а граф с заданными весами вершин и /или ребер - взвешенным графом.

### Изоморфизм

На [рис. 1.8](http://localhost:3232/department/algorithms/gaa/1/gaa_1.html#image.1.8#image.1.8) изображены два графа с одним и тем же множеством вершин \left\{a,b,c,d\right\}. При внимательном рассмотрении можно обнаружить, что это разные графы - в левом имеется ребро (a,c), в правом же такого нет. В то же время, если не обращать внимания на наименования вершин, то эти графы явно одинаково устроены: каждый из них - цикл из четырех вершин. Во многих случаях при исследовании строения графов имена или номера вершин не играют роли, и такие графы, один из которых получается из другого переименованием вершин, удобнее было бы считать одинаковыми. Для того чтобы это можно было делать "на законном основании", вводится понятие изоморфизма графов.



**Рис. 1.8.**

**Определение.** Графы G_{1}и G_{2}называются изоморфными, если существует такая биекция fмножества VG_{1}на множество VG_{2}, что (a,b)\in EG_{1}тогда и только тогда, когда \left(f(a),f(b)\right)\in EG_{2}. Отображение fв этом случае называется изоморфизмом графа G_{1}на граф G_{2}.

Тот факт, что графы G_{1}и G_{2}изоморфны, записывается так: {G_{1} \!\cong\! G_{2}}.

Для графов, изображенных на [рис. 1.8](http://localhost:3232/department/algorithms/gaa/1/gaa_1.html#image.1.8#image.1.8), изоморфизмом является, например, отображение, задаваемое следующей таблицей:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| x(вершина графа G_1) | a | b | c | d |
| f(x)(вершина графа G_2) | a | b | d | c |

Заметим, что в этом примере есть и другие изоморфизмы первого графа на второй.

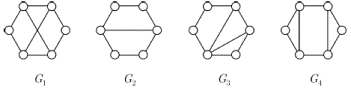
Сформулированное определение изоморфизма годится и для ориентированных графов, нужно только обе упоминаемые в нем пары вершин считать упорядоченными.

Изоморфизм - бинарное отношение на множестве графов. Очевидно, что это отношение рефлексивно, симметрично и транзитивно, то есть является отношением эквивалентности. Классы эквивалентности называются абстрактными графами. Когда говорят, что рассматриваются абстрактные графы, это означает, что изоморфные графы считаются одинаковыми. Абстрактный граф можно представлять себе как граф, у которого стерты имена (пометки) вершин, поэтому абстрактные графы иногда называют также непомеченными графами.

### Инварианты

В общем случае узнать, изоморфны ли два графа, достаточно сложно. Если буквально следовать определению, то нужно перебрать все биекции множества вершин одного из них на множество вершин другого и для каждой из этих биекций проверить, является ли она изоморфизмом. Для nвершин имеется n! биекций и эта работа становится практически невыполнимой уже для не очень больших n(например, 20!>2\cdot
10^{18}). Однако во многих случаях бывает довольно легко установить, что два данных графа неизоморфны. Рассмотрим, например, графы, изображенные на [рис. 1.9](http://localhost:3232/department/algorithms/gaa/1/gaa_1.html#image.1.9#image.1.9).

Так как при изоморфизме пара смежных вершин переходит в пару смежных, а пара несмежных - в пару несмежных, то ясно, что число ребер у двух изоморфных графов должно быть одинаковым. Поэтому сразу можно сказать, что графы G_{1}и G_{2}, у которых разное количество ребер, неизоморфны. У графов G_{1}и G_{3}одинаковое число ребер, но они тоже неизоморфны. Это можно установить, сравнивая степени вершин. Очевидно, при изоморфизме каждая вершина переходит в вершину той же степени. Следовательно, изоморфные графы должны иметь одинаковые наборы степеней, а у графов G_{1}и G_{3}эти наборы различны. С графами G_{1}и G_{4}дело обстоит немного сложнее - у них и наборы степеней одинаковы. Все же и эти графы неизоморфны: можно заметить, что в графе G_{4}есть цикл длины 3, а в графе G_{1}таких циклов нет. Ясно, что при изоморфизме каждый цикл длины 3 переходит в цикл длины 3.



**Рис. 1.9.**

Характеристики графов, которые сохраняются при изоморфизме, называются инвариантами. В этом примере мы видели некоторые простые инварианты - число ребер, набор степеней, число циклов заданной длины, в дальнейшем будут введены еще некоторые другие. Если удается установить, что для двух исследуемых графов некоторый инвариант принимает разные значения, то эти графы неизоморфны. Для того чтобы доказать, что два графа изоморфны, необходимо предъявить соответствующую биекцию.

### Операции над графами

Для получения новых графов можно использовать разнообразные операции над графами. Здесь мы рассмотрим два вида операций - локальные, при которых заменяются, удаляются или добавляются отдельные элементы графа, и алгебраические, когда новый граф строится по определенным правилам из нескольких имеющихся.

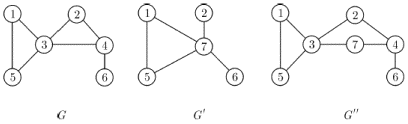
### Локальные операции

Простейшая операция - удаление ребра. При удалении ребра сохраняются все вершины графа и все его ребра, кроме удаляемого. Обратная операция - добавление ребра.

При удалении вершины вместе с вершиной удаляются и все инцидентные ей ребра. Граф, получаемый из графа Gудалением вершины a, обозначают G-a. При добавлении вершины к графу добавляется новая изолированная вершина. С помощью операций добавления вершин и ребер можно "из ничего", то есть из графа K_{0}, построить любой граф.

Операция стягивания ребра (a,b)определяется следующим образом. Вершины aи bудаляются из графа, к нему добавляется новая вершина cи она соединяется ребром с каждой вершиной, с которой была смежна хотя бы одна из вершин a,b.

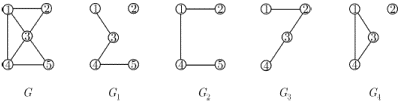
Операция подразбиения ребра (a,b)действует следующим образом. Из графа удаляется это ребро, к нему добавляется новая вершина cи два новых ребра (a,c)и (b,c). На [рис. 1.10](http://localhost:3232/department/algorithms/gaa/1/gaa_1.html#image.1.10#image.1.10) изображены исходный граф G, граф G', полученный из него стягиванием ребра (3,4)и G'', полученный подразбиением того же ребра. В обоих случаях вновь добавленная вершина обозначена цифрой 7.



**Рис. 1.10.**

### Подграфы

Граф G'называется подграфом графа G, если {VG'\subseteq VG}, {EG'\subseteq EG}. Всякий подграф может быть получен из графа удалением некоторых вершин и ребер. На [рис. 1.11](http://localhost:3232/department/algorithms/gaa/1/gaa_1.html#image.1.11#image.1.11) изображены граф Gи его подграфы G_{1}, G_{2}, G_{3}, G_{4}.



**Рис. 1.11.**

Подграф G'графа Gназывается остовным, если VG'=VG. Остовный подграф может быть получен из графа удалением некоторых ребер, вершины же остаются в неприкосновенности. На [рис. 1.11](http://localhost:3232/department/algorithms/gaa/1/gaa_1.html#image.1.11#image.1.11) G_{1}- остовный подграф графа G, а G_{2}, G_{3}и G_{4}не являются остовными подграфами.

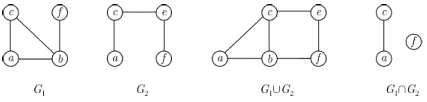
Другая важная разновидность подграфов - порожденные подграфы. Пусть задан граф G=(V,E)и в нем выбрано множество вершин U\subseteq V. Рассмотрим подграф G'=(U,E'), где E'состоит из всех тех ребер графа G, у которых оба конца принадлежат U. Говорят, что этот подграф порожден множеством вершин U. Он обозначается через G\langle U\rangle. Порожденный подграф может быть получен из графа удалением "лишних" вершин, т.е. вершин, не принадлежащих U.

Можно определить также подграф, порожденный множеством ребер F\subseteq E. Это подграф G'=(V',F), где V'состоит из всех вершин, инцидентных ребрам из F.

На [рис. 1.11](http://localhost:3232/department/algorithms/gaa/1/gaa_1.html#image.1.11#image.1.11) G_{2}- подграф графа G, порожденный множеством вершин \{1, 2, 4, 5\}, т.е. G_{2} =G\langle \left\{1,\;
2,\; 4,\;
5\right\}\rangle, он же порождается множеством ребер \{(1, 2), (1,
4), (4, 5)\}; подграф G_{3}не порождается множеством вершин, но порождается множеством ребер \{(1, 2), (2, 3), (3, 4)\}; подграф G_{4}не является ни остовным, ни порожденным в каком-либо смысле.

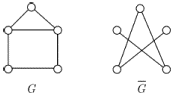
### Алгебраические операции

Поскольку граф состоит из двух множеств (вершины и ребра), то различные операции над множествами естественным образом порождают соответствующие операции над графами. Например, объединение двух графов G_{1}и G_{2}определяется как граф G=G_{1} \cup G_{2}, у которого VG=VG_{1} \cup VG_{2}, EG=EG_{1} \cup
EG_{2}, а пересечение - как граф G=G_{1} \cap G_{2}, у которого VG=VG_{1} \cap
VG_{2}, EG=EG_{1} \cap EG_{2}. Обе операции иллюстрирует [рис. 1.12](http://localhost:3232/department/algorithms/gaa/1/gaa_1.html#image.1.12#image.1.12).



**Рис. 1.12.**

Дополнением (дополнительным графом) к графу G=(V,E)называется граф \ol{G}, у которого множество вершин то же, что у G, а множество ребер является дополнением множества Eдо множества всех неупорядоченных пар вершин. Иначе говоря, две различные вершины смежны в графе \ol{G} тогда и только тогда, когда они несмежны в графе G. Например, \ol{O}_{n} =K_{n}. Другой пример показан на [рис. 1.13](http://localhost:3232/department/algorithms/gaa/1/gaa_1.html#image.1.13#image.1.13). Очевидно, что всегда \ol{\ol{G}}=G.



**Рис. 1.13.**

Под суммой G_{1} +
G_{2}двух абстрактных графов понимают объединение графов с непересекающимися множествами вершин. Точнее говоря, имеется в виду следующее. Сначала вершинам графов-слагаемых присваиваются имена (пометки, номера) так, чтобы множества вершин не пересекались, затем полученные графы объединяются. Операция сложения ассоциативна, то есть (G_{1} +G_{2})+G_{3} =G_{1} +(G_{2} +G_{3})для любых трех графов. Поэтому можно образовывать сумму любого числа графов, не указывая порядка действий с помощью скобок. Если складываются kэкземпляров одного и того же графа G, то полученный граф обозначается через kG. Например, O_{n} \cong nK_{1}. На [рис. 1.14](http://localhost:3232/department/algorithms/gaa/1/gaa_1.html#image.1.14#image.1.14) изображен граф C_{4} +2K_{2} +4K_{1}.

1-14

**Рис. 1.14.**

1-15

**Рис. 1.15.**

Соединением двух графов G_{1}и G_{2}называется граф, получаемый из их суммы добавлением всех ребер, соединяющих вершины первого слагаемого с вершинами второго. Будем записывать эту операцию как G_{1} \circ G_{2}. На [рис. 1.15](http://localhost:3232/department/algorithms/gaa/1/gaa_1.html#image.1.15#image.1.15) представлен граф P_{3} \circ O_{2}. Легко видеть, что операции сложения и соединения графов связаны друг с другом следующими простыми соотношениями:

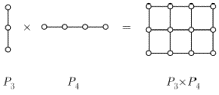

\eq*{
\overline{G_{1} +G_{2}}=\overline{G_{1}}\circ
\overline{G_{2}},\qq
\overline{G_{1} \circ G_{2}}=\overline{G_{1} }+\overline{G_{2}}.
}


Введем еще два типа специальных графов, которые легко описываются с помощью операции соединения. Первый - полный двудольный граф K_{p,q} =O_{p} \circ
O_{q}. В этом графе множество вершин разбито на два подмножества (доли), в одном из которых pвершин, в другом q, и две вершины в нем смежны тогда и только тогда, если они принадлежат разным подмножествам. Второй - колесо W_{n} =C_{n} \circ K_{1}
. На [рис. 1.16](http://localhost:3232/department/algorithms/gaa/1/gaa_1.html#image.1.16#image.1.16) показаны графы K_{3,4}и W_{6}.



**Рис. 1.16.**

Произведение G=G_{1}
\times G_{2}графов G_{1}и G_{2}определяется следующим образом. Множеством вершин графа Gявляется декартово произведение множеств VG_{1}и VG_{2}, то есть вершины этого графа - упорядоченные пары \left(x,y\right), где x- вершина первого сомножителя, y- вершина второго. Вершины \left(x_{1},y_{1}\right)и \left(x_{2},y_{2}\right)в Gсмежны тогда и только тогда, если x_{1} =x_{2}и y_{1}смежна с y_{2}в графе G_{2}, или y_{1}
=y_{2}и x_{1}смежна с x_{2}в графе G_{1}. С помощью операции произведения можно выразить некоторые важные графы через простейшие. Например, произведение двух цепей дает прямоугольную решетку (см. [рис. 1.17](http://localhost:3232/department/algorithms/gaa/1/gaa_1.html#image.1.17#image.1.17)). Если один из сомножителей превратить в цикл, добавив одно ребро, то прямоугольная решетка превратится в цилиндрическую, а если и второй сомножитель превратить в цикл, то получится тороидальная решетка.

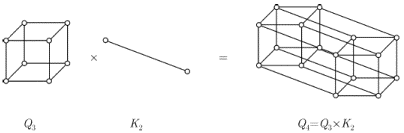


**Рис. 1.17.**

Другой пример - k- мерный куб Q_{k}, определяемый следующим образом. Вершинами его являются всевозможные упорядоченные двоичные наборы длины k. Всего, таким образом, в этом графе 2^{k}вершин. Вершины x=\left(x_{1}\dts x_{k}
\right)и y=\left(y_{1}\dts y_{k} \right)смежны в нем тогда и только тогда, когда наборы xи yразличаются точно в одной координате. С помощью операции произведения граф Q_{k}можно определить рекурсивно:


\eq*{
Q_{1} =K_{2}, \qq Q_{k} =Q_{k-1} \times K_{2}.
}

На [рис. 1.18](http://localhost:3232/department/algorithms/gaa/1/gaa_1.html#image.1.18#image.1.18) показано, как получается Q_{4}из Q_{3}.



**Рис. 1.18.**

**Лекция 2. Поиск в глубину и ширину.**

### Поиск в ширину

С этой лекции мы начинаем рассматривать алгоритмы для решения различных задач на графах. Сначала речь пойдет о задачах анализа графов с целью выявления их структуры и вычисления метрических характеристик. Многие задачи такого рода могут быть решены путем систематического обхода графа с посещением всех его вершин и исследованием всех его ребер. Такой обход можно выполнить многими способами, в действительности же широкое распространение благодаря своей простоте, а в большей степени своей полезности, получили две стратегии - поиск в глубину и поиск в ширину. Рассмотрение этих алгоритмов и примеров их применения к задачам анализа графов составляет содержание этой и двух следующих лекций.

### Процедура поиска в ширину

Работа всякого алгоритма обхода состоит в последовательном посещении вершин и исследовании ребер. Какие именно действия выполняются при посещении вершины и исследовании ребра - зависит от конкретной задачи, для решения которой производится обход. В любом случае, однако, факт посещения вершины запоминается, так что с момента посещения и до конца работы алгоритма она считается посещенной. Вершину, которая еще не посещена, будем называть новой. В результате посещения вершина становится открытой и остается такой, пока не будут исследованы все инцидентные ей ребра. После этого она превращается в закрытую.

Идея поиска в ширину состоит в том, чтобы посещать вершины в порядке их удаленности от некоторой заранее выбранной или указанной стартовой вершины a. Иначе говоря, сначала посещается сама вершина a, затем все вершины, смежные с a, то есть находящиеся от нее на расстоянии 1, затем вершины, находящиеся от aна расстоянии 2, и т.д.

Рассмотрим алгоритм поиска в ширину с заданной стартовой вершиной a. Вначале все вершины помечаются как новые. Первой посещается вершина a, она становится единственной открытой вершиной. В дальнейшем каждый очередной шаг начинается с выбора некоторой открытой вершины x. Эта вершина становится активной. Далее исследуются ребра, инцидентные активной вершине. Если такое ребро соединяет вершину xс новой вершиной y, то вершина yпосещается и превращается в открытую. Когда все ребра, инцидентные активной вершине, исследованы, она перестает быть активной и становится закрытой. После этого выбирается новая активная вершина, и описанные действия повторяются. Процесс заканчивается, когда множество открытых вершин становится пустым.

Основная особенность поиска в ширину, отличающая его от других способов обхода графов, состоит в том, что в качестве активной вершины выбирается та из открытых, которая была посещена раньше других. Именно этим обеспечивается главное свойство поиска в ширину: чем ближе вершина к старту, тем раньше она будет посещена. Для реализации такого правила выбора активной вершины удобно использовать для хранения множества открытых вершин очередь - когда новая вершина становится открытой, она добавляется в конец очереди, а активная выбирается в ее начале. Схематически процесс изменения статуса вершин изображен на [рис. 2.1](http://localhost:3232/department/algorithms/gaa/4/gaa_4.html#image.4.1#image.4.1). Черным кружком обозначена активная вершина.



**Рис. 2.1.**

Опишем процедуру поиска в ширину (BFS - от английского названия этого алгоритма - Breadth First Search) из заданной стартовой вершины a. В этом описании V(x)обозначает множество всех вершин, смежных с вершиной x, Q- очередь открытых вершин. Предполагается, что при посещении вершины она помечается как посещенная и эта пометка означает, что вершина уже не является новой.

**Procedure BFS(a)**

1. посетить вершину a
2. a\Rightarrow Q
3. **while** Q\ne
   \varnothing**do**
4. x\Leftarrow Q
5. **for** y\in V(x)**do**
6. исследовать ребро (x,y)
7. **if** вершина yновая
8. **then** посетить вершину y
9. y\Rightarrow Q

Отметим некоторые свойства процедуры BFS.

1. Процедура BFS заканчивает работу после конечного числа шагов.

Действительно, при каждом повторении цикла **while** из очереди удаляется одна вершина. Вершина добавляется к очереди только тогда, когда она посещается. Каждая вершина может быть посещена не более одного раза, так как посещаются только новые вершины, а в результате посещения вершина перестает быть новой. Таким образом, число повторений цикла **while** не превосходит числа вершин.

1. В результате выполнения процедуры BFS будут посещены все вершины из компоненты связности, содержащей вершину a, и только они.

Очевидно, что вершина может быть посещена только в том случае, когда существует путь, соединяющий ее с вершиной a(так как посещается всегда вершина, смежная с уже посещенной). То, что каждая такая вершина будет посещена, легко доказывается индукцией по расстоянию от данной вершины до вершины a.

1. Время работы процедуры BFS есть O(m), где m- число ребер в компоненте связности, содержащей вершину a.

Из предыдущих рассуждений видно, что каждая вершина из этой компоненты становится активной точно один раз. Внутренний цикл **for** для активной вершины xвыполняется \deg (x)раз. Следовательно, общее число повторений внутреннего цикла будет равно \suml_{x\in VG}\deg (x) =2m.

Итак, процедура BFS(a) производит обход компоненты связности, содержащей вершину a. Чтобы перейти к другой компоненте, достаточно выбрать какую-нибудь новую вершину (если такие вершины еще имеются), в качестве стартовой. Пусть V- множество вершин графа. Следующий алгоритм осуществляет полный обход графа методом поиска в ширину.

**Алгоритм 1.** Поиск в ширину.

1. пометить все вершины как новые
2. создать пустую очередь Q
3. **for** a\in V**do** **if** aновая **then** BFS(a)

Учитывая, что цикл **for** в строке 3повторяется nраз, где n- число вершин графа, получаем общую оценку трудоемкости O(m+n). Необходимо отметить, что эта оценка справедлива в предположении, что время, требуемое для просмотра окрестности вершины, пропорционально степени этой вершины. Это имеет место, например, если граф задан списками смежности. Если же граф задан матрицей смежности, то для просмотра окрестности любой вершины будет затрачиваться время, пропорциональное n. В этом случае общее время работы алгоритма будет оцениваться как O(n^{2}). Наибольшее значение величины mпри данном nравно \frac{n(n-1)}{2}, т.е. имеет порядок n^{2}. Таким образом, трудоемкость алгоритма поиска в ширину при задании графа списками смежности не выше, чем при задании матрицей смежности. В целом же первый способ задания предпочтительнее, так как дает выигрыш для графов с небольшим числом ребер.

В качестве простейшего примера применения поиска в ширину для графа рассмотрим задачу выявления компонент связности. Допустим, мы хотим получить ответ в виде таблицы, в которой для каждой вершины xуказан номер \comp(x)компоненты, которой принадлежит эта вершина. Компоненты будут получать номера в процессе обхода. Для решения этой задачи достаточно ввести переменную cсо значением, равным текущему номеру компоненты, и каждый раз при посещении новой вершины xполагать \comp(x)=c. Значение cпервоначально устанавливается равным 0и модифицируется при каждом вызове процедуры BFS.

### BFS-дерево и вычисление расстояний

Другая простая задача, для решения которой можно применить поиск в ширину, - построение каркаса. Напомним, что каркасом графа называется остовный лес, у которого области связности совпадают с областями связности графа. Каркас связного графа - остовное дерево.

Ребра, исследуемые в процессе обхода графа, можно разделить на две категории: если ребро соединяет активную вершину xс новой вершиной y, то оно классифицируется как прямое, в противном случае - как обратное. В зависимости от решаемой задачи прямые и обратные ребра могут подвергаться различной обработке.

Предположим, что алгоритм поиска в ширину применяется к связному графу. Покажем, что в этом случае по окончании обхода множество всех прямых ребер образует дерево. Действительно, допустим, что на некотором шаге работы алгоритма обнаруживается новое прямое ребро (x,y), а множество прямых ребер, накопленных к этому шагу, образует дерево F. Тогда вершина xпринадлежит дереву F, а вершина yне принадлежит ему. Поэтому при добавлении к дереву Fребра (x,y)связность сохранится, а циклов не появится.

Итак, если применить поиск в ширину к связному графу и запомнить все прямые ребра, то получим каркас графа. Для произвольного графа будет получен лес, также, очевидно, являющийся каркасом.

Каркас, который будет построен описанным образом в результате поиска в ширину в связном графе, называется BFS-деревом. Его можно рассматривать как корневое дерево с корнем в стартовой вершине a. BFS-дерево с заданным корнем a, вообще говоря, не единственное - зависит от того, в каком порядке просматриваются окрестности вершин. Однако всякое BFS-дерево обладает свойством, на котором и основаны наиболее важные применения поиска в ширину. Каркас Tсвязного графа Gс корнем aназовем геодезическим деревом, если для любой вершины xпуть из xв aв дереве Tявляется кратчайшим путем между xи aв графе G.

**Теорема 1.** Любое BFS-дерево является геодезическим деревом.

**Доказательство.** Обозначим через D(i)множество всех вершин графа, находящихся на расстоянии iот стартовой вершины a. Работа алгоритма начинается с посещения стартовой вершины, т.е. единственной вершины, составляющей множество D(0). При первом выполнении цикла **while** будут посещены и помещены в очередь все вершины из множества D(1). Затем эти вершины будут одна за другой извлекаться из очереди, становиться активными, и для каждой из них будут исследоваться все смежные вершины. Те из них, которые еще не посещались, будут посещены и помещены в очередь. Но это как раз все вершины из множества D(2)(когда начинается исследование окрестностей вершин из D(1), ни одна вершина из D(2)еще не посещалась и каждая из них смежна хотя бы с одной вершиной из D(1)). Следовательно, каждая вершина из D(2)будет посещена после всех вершин из D(1). Рассуждая далее таким образом, приходим к следующему выводу.

(А) Все вершины из D(i+1)будут посещены после всех вершин из D(i), i=0,1,\ldots.

Строгое доказательство легко провести индукцией по i. Отметим еще следующий факт.

(Б) Если активной является вершина из D(i), то в этот момент все вершины из D(i)уже посещены.

В самом деле, из (А) следует, что вершины из D(i)попадут в очередь после вершин из D(i-1). Поэтому, когда первая вершина из D(i)становится активной, все вершины из D(i-1)уже закрыты. Значит, к этому моменту окрестности всех вершин из D(i-1)полностью исследованы, и, следовательно, все вершины из D(i)посещены.

Рассмотрим теперь момент работы алгоритма, когда активной является вершина x\in D(i)и обнаруживается смежная с ней новая вершина y. В BFS-дереве расстояние между yи aна 1 больше, чем расстояние между xи a. В графе расстояние между yи aне больше, чем i+1, так как xи yсмежны. Ввиду (А) это расстояние не может быть меньше i, а ввиду (Б) оно не может быть равно i. Значит, y\in D(i+1), т.е. в графе расстояние между yи aтоже на 1 больше, чем расстояние между xи a. Следовательно, если до какого-то момента работы алгоритма расстояния от каждой из посещенных вершин до стартовой вершины в графе и в дереве были равны, то это будет верно и для вновь посещаемой вершины. Поскольку это верно вначале, когда имеется единственная посещенная вершина a(оба расстояния равны 0), то это останется верным и тогда, когда будут посещены все вершины.

Итак, мы можем применить поиск в ширину для вычисления расстояний от стартовой вершины aдо всех остальных вершин графа - нужно только в процессе обхода для каждой посещаемой вершины yопределять расстояние от yдо aв BFS-дереве. Это сделать легко: d(a,y)=d(a,x)+1, где x- активная вершина. Вначале устанавливаем d(a,a)=0.

Если граф несвязен, некоторые расстояния будут бесконечными. Чтобы учесть эту возможность, положим вначале d(a,x)=\inftyдля всех x\ne a. Пока вершина xостается новой, для нее сохраняется значение d(a,x)=\infty, когда же она посещается, d(a,x)становится равным расстоянию между aи xи больше не меняется. Таким образом, бесконечность расстояния можно использовать как признак того, что вершина новая. Если по окончании работы d(a,x)=\inftyдля некоторой вершины x, это означает, что xне достижима из a, то есть принадлежит другой компоненте связности.

Для того чтобы не только определять расстояния, но и находить кратчайшие пути от aдо остальных вершин, достаточно для каждой вершины yзнать ее отца F(y)в BFS-дереве. Очевидно, что F(y)=x, где x- вершина, активная в момент посещения вершины y. Заполнение таблицы Fфактически означает построение BFS-дерева.

Модифицируя процедуру BFS с учетом сделанных замечаний, получаем следующий алгоритм:

**Алгоритм 2.**Построение BFS-дерева и вычисление расстояний от вершины aдо всех остальных вершин:

1. **for** x\in V**do** d(a,x)\; :=\;
   \infty
2. d(a,a)\; :=\; 0
3. a\Rightarrow Q
4. **while** Q\ne \varnothing
   \quad**do**
5. x\Leftarrow Q
6. **for** y\in
   V\left(x\right)\quad \quad**do**
7. **if** d(a,y)=\infty
8. **then** d(a,y):=d(a,x)+1
9. F(y):=x
10. y\Rightarrow Q

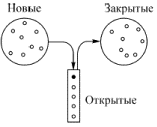
### Процедура поиска в глубину

Поиск в глубину - вероятно, наиболее важная ввиду многочисленности приложений стратегия обхода графа. Идея этого метода - идти вперед в неисследованную область, пока это возможно, если же вокруг все исследовано, отступить на шаг назад и искать новые возможности для продвижения вперед. Метод поиска в глубину известен под разными названиями, например, "бэктрекинг", "поиск с возвращением".

Понятия новой, открытой, закрытой и активной вершин для поиска в глубину имеют такой же смысл, как и для поиска в ширину. Отметим, что всегда имеется не более чем одна активная вершина.

Обход начинается с посещения заданной стартовой вершины a, которая становится активной и единственной открытой вершиной. Затем выбирается инцидентное вершине aребро (a,y)и посещается вершина y. Она становится открытой и активной. Заметим, что при поиске в ширину вершина aоставалась активной до тех пор, пока не были исследованы все инцидентные ей ребра. В дальнейшем, как и при поиске в ширину, каждый очередной шаг начинается с выбора активной вершины из множества открытых вершин. Если все ребра, инцидентные активной вершине x, уже исследованы, она превращается в закрытую. В противном случае выбирается одно из неисследованных ребер (x,y), это ребро исследуется. Если вершина yновая, то она посещается и превращается в открытую.

Главное отличие от поиска в ширину состоит в том, что при поиске в глубину в качестве активной выбирается та из открытых вершин, которая была посещена последней. Для реализации такого правила выбора наиболее удобной структурой хранения множества открытых вершин является стек: открываемые вершины складываются в стек в том порядке, в каком они открываются, а в качестве активной выбирается последняя вершина. Схематически это показано на рис. 2.2.



**Рис. 2.2.**

Обозначим стек для открытых вершин через S, остальные обозначения сохраняют тот же смысл, что и в предыдущем разделе. Через {\rm
top}(S)обозначается верхний элемент стека (т.е. последний элемент, добавленный к стеку). Тогда процедура обхода одной компоненты связности методом поиска в глубину со стартовой вершиной aможет быть записана следующим образом (DFS - Depth First Search).

**Procedure** DFS(a)

1. посетить вершину a
2. a\Rightarrow S
3. **while** S\ne
   \varnothing**do**
4. x :={\rm top}(S)
5. **if** имеется неисследованное ребро (x,y)
6. **then** исследовать ребро (x,y)
7. **if** вершина yновая
8. **then** посетить вершину y
9. y\Rightarrow S
10. **else** удалить xиз S

Еще раз обратим внимание на основное отличие этой процедуры от аналогичной процедуры поиска в ширину. При поиске в ширину вершина, став активной, остается ею, пока не будет полностью исследована ее окрестность, после чего она становится закрытой. При поиске в глубину, если в окрестности активной вершины xобнаруживается новая вершина y, то yпомещается в стек и при следующем повторении цикла **while** станет активной. При этом xостается в стеке и через какое-то время снова станет активной. Иначе говоря, ребра, инцидентные вершине x, будут исследованы не подряд, а с перерывами.

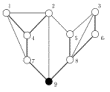
Алгоритм обхода всего графа - тот же, что и в случае поиска в ширину (алгоритм 1), только нужно очередь заменить стеком, а процедуру BFS - процедурой DFS.

Свойства 1 и 2 поиска в ширину, отмеченные в предыдущем разделе, сохраняются и для поиска в глубину. Остается верной и оценка трудоемкости O(m+n), но ее доказательство требует несколько иных рассуждений, так как каждая вершина теперь может становиться активной несколько раз. Однако каждое ребро рассматривается только два раза (один раз для каждой инцидентной ему вершины), поэтому в операторе **if** в строке 5 ветвь **then** (строки 6-9) повторяется O(m)раз. В этом же операторе ветвь **else** (строка 10) повторяется O(n)раз, так как каждая вершина может быть удалена из стека только один раз. В целом получается O(m+n), причем остаются справедливыми сделанные замечания об условиях, при которых имеет место эта оценка.

### DFS-дерево

Поиск в глубину можно применить для нахождения компонент связности графа или для построения каркаса точно таким же образом, как поиск в ширину. Понятия прямого и обратного ребра определяются так же, как в предыдущем разделе и так же доказывается, что прямые ребра при поиске в глубину образуют каркас графа. Для связного графа каркас, получаемый поиском в глубину, называется DFS-деревом. DFS-дерево рассматривается как корневое дерево с корнем в стартовой вершине a. Это дерево обладает особыми свойствами, на использовании которых основаны многочисленные применения метода поиска в глубину. Рассмотрим наиболее важное из этих свойств.

Относительно любого корневого остовного дерева все ребра графа, не принадлежащие дереву, можно разделить на две категории. Ребро назовем продольным, если одна из его вершин является предком другой, в противном случае ребро назовем поперечным. В примере на [рис. 2.](http://localhost:3232/department/algorithms/gaa/5/gaa_5.html#image.5.2#image.5.2)3 ребра каркаса выделены жирными линиями, корень - черным кружком. Обратные ребра показаны тонкими линиями, из них продольными являются ребра (1, 7), (2, 9), (3, 8), а поперечными - ребра (1, 2), (2, 5), (3, 5).



**Рис. 2.3.**

**Теорема 1.** Пусть G- связный граф, T- DFS-дерево графа G. Тогда относительно Tвсе обратные ребра являются продольными.

**Доказательство.** Убедимся сначала, что после того, как стартовая вершина aпомещена в стек, на каждом последующем шаге работы алгоритма последовательность вершин, хранящаяся в стеке, образует путь с началом в вершине a, а все ребра этого пути принадлежат дереву. Вначале это, очевидно, так. В дальнейшем всякий раз, когда новая вершина yпомещается в стек, к дереву добавляется прямое ребро (x,y), причем вершина xнаходится в стеке перед вершиной y. Значит, если указанное свойство имело место до добавления вершины в стек, то оно сохранится и после добавления. Удаление же вершины из стека, конечно, не может нарушить этого свойства.

Пусть теперь (x,y)- обратное ребро. Каждая из вершин xи yв ходе работы алгоритма когда-либо окажется в стеке. Допустим, xокажется там раньше, чем y. Рассмотрим шаг алгоритма, на котором yпомещается в стек. В этот момент xеще находится в стеке. Действительно, вершина исключается из стека только тогда, когда в ее окрестности нет непосещенных вершин. Но непосредственно перед помещением в стек вершина yявляется новой и принадлежит окрестности вершины x. Таким образом, вершина xлежит на пути, принадлежащем дереву и соединяющем вершины aи y. Но это означает, что вершина xявляется предком вершины yв дереве Tи, следовательно, ребро (x,y)- продольное.

Таким образом, каркас, изображенный на рис. 2.3, не мог быть построен методом поиска в глубину. Кстати, он не мог быть построен и с помощью поиска в ширину (почему?).

### Глубинная нумерация

Ввиду важности этого метода опишем еще два варианта алгоритма поиска в глубину. Первый из них - рекурсивный, и, как обычно, рекурсия дает возможность представить алгоритм в наиболее компактной форме. Для того чтобы алгоритм выполнял какую-то полезную работу, будем нумеровать вершины в том порядке, в каком они встречаются при обходе. Номер, получаемый вершиной x, обозначается через \Dnum(x)и называется ее глубинным номером. Вначале полагаем \Dnum(x)=0для всех x. Это нулевое значение сохраняется до тех пор, пока вершина не становится открытой, в этот момент ей присваивается ее настоящий глубинный номер. Таким образом, нет необходимости в какой-либо специальной структуре для запоминания новых вершин - они отличаются от всех других нулевым значением \Dnum. Переменная cхранит текущий номер. Рекурсивная процедура DFSR обходит одну компоненту связности, а алгоритм 1 обходит весь граф и присваивает вершинам глубинные номера.

**Алгоритм 1.** Поиск в глубину с вычислением глубинных номеров - рекурсивный вариант

1. **for** x\in V**do** \Dnum\left(x\right)
   :=0
2. c:=0
3. **for** x\in V**do**
4. **if** \Dnum\left(x\right)=0**then** DFSR(x)

**Procedure** DFSR(x) 

1. c:=c+1
2. \Dnum(x):=c
3. **for** y\in V(x)**do**
4. **if** \Dnum(y)=0**then** DFSR(y)

### Построение каркаса

Следующий вариант алгоритма поиска в глубину отличается тем, что не использует стека для хранения открытых вершин. Стек нужен для того, чтобы в момент, когда окрестность активной вершины xисследована и необходимо сделать "шаг назад", можно было определить вершину, в которую нужно вернуться. Но это та вершина, которая является отцом вершины xв DFS-дереве. Поэтому, если решение задачи предусматривает построение DFS-дерева, то это дерево можно использовать и для организации "возвратных движений" в процессе обхода. Описываемый ниже алгоритм строит каркас произвольного графа, каждая компонента связности этого каркаса является DFS-деревом соответствующей компоненты связности графа. Через F(x)обозначается отец вершины xв этом DFS-дереве, при этом для корня дерева (стартовой вершины) aполагаем F(a)=a. Здесь и далее в описаниях алгоритмов инструкция "открыть (закрыть) вершину" означает, что вершина каким-то образом помечается как открытая (закрытая).

**Алгоритм 2.** Поиск в глубину с построением каркаса

1. пометить все вершины как новые
2. **for** a\in V**do**
3. **if** вершина aновая **then** DFST(a)

**Procedure** DFST(a)

1. F(a)\, :=a
2. открыть вершину a
3. x\, :=a
4. **while** xоткрытая **do**
5. **if** имеется неисследованное ребро (x,y)
6. **then** исследовать ребро (x,y)
7. **if** вершина yновая
8. **then** F(y):=x
9. открыть вершину y
10. x:=y
11. **else** закрыть вершину x
12. x:=F(x)

### Шарниры

В качестве примера задачи, для эффективного решения которой можно использовать основное свойство DFS-дерева, выражаемое теоремой 1, рассмотрим задачу выявления шарниров в графе. Напомним, что шарниром называется вершина, при удалении которой увеличивается число компонент связности. Отсутствие поперечных ребер относительно DFS-дерева позволяет очень просто узнать, является ли стартовая вершина a(корень этого дерева) шарниром.

**Лемма 1.** Стартовая вершина а является шарниром графа тогда и только тогда, когда ее степень в DFS-дереве больше 1.

**Доказательство.** Если вершину aудалить из дерева, то оно распадется на поддеревья, называемые ветвями. Число ветвей равно степени вершины aв дереве. Так как поперечных ребер нет, то вершины из разных ветвей не могут быть смежными в графе и каждый путь из одной ветви в другую обязательно проходит через вершину a. Следовательно, если степень вершины aв DFS-дереве больше 1, то эта вершина - шарнир. Если же степень вершины aв DFS-дереве равна 1, то в дереве имеется единственная вершина b, смежная с a, и каждая из остальных вершин графа соединена с вершиной bпутем, не проходящим через a. Поэтому в данном случае удаление вершины aне нарушает связности графа и эта вершина не является шарниром.

Это свойство корня DFS-дерева можно было бы использовать для выявления всех шарниров, просто выполнив nраз поиск в глубину, стартуя поочередно в каждой вершине. Оказывается, все шарниры можно выявить однократным поиском в глубину. Следующая теорема характеризует все шарниры, отличные от корня DFS-дерева. Напомним, что каждая вершина дерева является и предком, и потомком самой себя. Предок (потомок) вершины, отличный от самой этой вершины, называется собственным предком (потомком).

**Теорема 2.** Пусть T- DFS-дерево графа Gс корнем a. Вершина x\ne aявляется шарниром графа тогда и только тогда, когда у нее в дереве Tимеется такой сын y, что ни один потомок вершины yне соединен ребром ни с одним собственным предком вершины x.

**Доказательство.** Если y- сын вершины xи ни один потомок вершины yне соединен ребром ни с одним собственным предком вершины x, то, ввиду отсутствия поперечных ребер, любой путь, соединяющий вершину yс корнем, проходит через x. Следовательно, в этом случае вершина x- шарнир. Если же для каждого сына yвершины xимеется ребро, соединяющее вершину yс каким-либо собственным предком вершины x, то каждый сын вершины xсоединен с корнем дерева путем, не проходящим через x. Поэтому при удалении вершины xграф останется связным и xв этом случае не является шарниром.

Для применения этого критерия к поиску шарниров введем на множестве вершин функцию \Low, связанную с DFS-деревом: значением \Low(x)является наименьший из глубинных номеров вершин, смежных с потомками вершины x. Если вершина yявляется сыном вершины x, то \Low(y)\le \Dnum(x)(так как вершина yявляется потомком самой себя и смежна с вершиной x). Из теоремы 2 следует, что вершина x, отличная от a, является шарниром тогда и только тогда, когда у нее имеется сын yтакой, что \Low(y)=\Dnum(x).

Функцию \Lowможно определить рекурсивно - если мы знаем ее значения для всех сыновей вершины xи глубинные номера всех вершин, смежных с xи не являющихся ее сыновьями, то \Low(x)есть минимум из всех этих величин, то есть


\Low(x)=\min \left(\min_{y \in A} \Low(y),\quad \min_{y\in B}
\Dnum(y)\right)\!,


где Aобозначает множество всех сыновей вершины x, а B- множество всех остальных вершин, смежных с x. Нетрудно видеть, что это определение эквивалентно первоначальному. Исходя из него, можно вычислять значения функции \Lowв процессе поиска в глубину с помощью следующей рекурсивной процедуры. Предполагается, что вначале всем элементам массива \Dnumприсвоены нулевые значения.

**Procedure** ComputeLow(x)

1. c:=c+1
2. \Dnum(x):=c
3. \Low(x):=c
4. **for** y\in V(x)**do**
5. **if** \Dnum(y)=0
6. **then** ComputeLow(y)
7. \Low(x):=\min (\Low(x),\Low(y))
8. **else** \Low(x):=\min
   (\Low(x),\Dnum(y))

**Лекция 3. Эйлеровы и гамильтоновы циклы**.

### Маршруты, пути, циклы

Маршрут в графе - это последовательность вершин x_{1},x_{2}\ldots x_{n}, такая, что для каждого i=1,2 \ldots n-1вершины x_{i}и x_{i+1}соединены ребром. Эти n-1ребер называются ребрами маршрута. Говорят, что маршрут проходит через них, а число n-1называют длиной маршрута. Говорят, что маршрут соединяет вершины x_{1}и x_{n}, они называются соответственно началом и концом маршрута, вершины x_{2}\ldots x_{n-1}называются промежуточными. Маршрут называется замкнутым, если x_{1} =x_{n}.

Путь - это маршрут, в котором все ребра различны. Путь называется простым, если и все вершины в нем различны.

Цикл - это замкнутый путь. Цикл x_{1}, x_{2}\ldots x_{n-1},x_{1}называется простым, если все вершины x_{1} ,x_{2}\ldots x_{n-1}попарно различны.

В графе на [рисунке 3.1](http://localhost:3232/department/algorithms/gaa/2/gaa_2.html#image.2.1#image.2.1) последовательность вершин

* 2, 3, 5, 4- не маршрут;
* 2, 3, 4, 5, 1, 4, 3- маршрут, но не путь;
* 3, 1, 4, 5, 1, 2- путь, но не простой;
* 2, 3, 1, 4, 3, 1, 2- замкнутый маршрут, но не цикл;
* 2, 3, 1, 4, 5, 1, 2- цикл, но не простой;
* 2, 3, 4, 5, 1, 2- простой цикл.

2-1

**Рис. 3.1.**

Установим некоторые простые свойства маршрутов.

**Теорема 1.** В любом маршруте, соединяющем две различные вершины, содержится простой путь, соединяющий те же вершины. В любом цикле, проходящем через некоторое ребро, содержится простой цикл, проходящий через это ребро.

**Доказательство.**

Пусть x_{1},x_{2}\ldots x_{n}- маршрут. Если все его вершины различны, то это уже простой путь. В противном случае, пусть x_{i} =x_{j}, i \lt j. Тогда последовательность x_{1},x_{2} \ldots x_{i-1}, x_{i}, x_{j+1}\ldots
x_{n}, полученная из этого маршрута удалением отрезка последовательности от x_{i+1}до x_{j}, тоже является маршрутом. Новый маршрут соединяет те же вершины и имеет меньшую длину. Продолжая действовать таким образом, после конечного числа "спрямлений" получим простой путь, соединяющий x_{1}и x_{n}. Второе утверждение теоремы доказывается аналогично.

Отметим, что в формулировке теоремы 1 нельзя заменить слово "цикл" словами "замкнутый маршрут". Действительно, если \left(a,b\right)- ребро графа, то последовательность a,b,a- замкнутый маршрут, проходящий через это ребро, но никакого цикла в нем нет.

**Теорема 2.** Если в графе степень каждой вершины не меньше 2, то в нем есть цикл.

**Доказательство.**

Найдем в графе простой путь наибольшей длины. Пусть это x_{1}, x_{2} \ldots x_{n}. Вершина x_{n}смежна с x_{n-1}, а так как ее степень не меньше двух, то она смежна еще хотя бы с одной вершиной, скажем, с y. Если бы yбыла отлична от всех вершин пути, то последовательность x_{1},x_{2} \ldots x_{n},yбыла бы простым путем большей длины. Следовательно, y- это одна из вершин пути, y=x_{i}, причем i \lt n-1. Но тогда x_{i},x_{i+1}\ldots x_{n},x_{i}- цикл.

### Связность и компоненты

Граф называется связным, если в нем для любых двух вершин имеется маршрут, соединяющий эти вершины. Заметим, что ввиду теоремы 1 можно в этом определении заменить слово "маршрут" словами "простой путь".

Для произвольного графа определим на множестве вершин отношение соединимости: вершина aсоединима с вершиной b, если существует соединяющий их маршрут. Легко видеть, что это отношение рефлексивно, симметрично и транзитивно, то есть является отношением эквивалентности. Классы эквивалентности называются областями связности, а порождаемые ими подграфы - компонентами связности графа. В связном графе имеется только одна компонента связности - весь граф. Компоненты связности можно определить также как максимальные по включению связные подграфы данного графа.

У графа на [рис. 3.2](http://localhost:3232/department/algorithms/gaa/2/gaa_2.html#image.2.2#image.2.2) имеется четыре области связности - \{1, 2, 9\}, \{3, 10,
11\}, \{4\}, \{5, 6, 7, 8, 12, 13, 14,
15\}.

2-2

**Рис. 3.2.**

Вершина называется шарниром (или точкой сочленения), если при ее удалении число компонент связности увеличивается. У графа на [рис. 3.2](http://localhost:3232/department/algorithms/gaa/2/gaa_2.html#image.2.2#image.2.2) имеется четыре шарнира - это вершины 3, 6, 7, 8.

Ребро, при удалении которого увеличивается число компонент связности, называется перешейком. Перешейками графа, изображенного на [рис. 3.2](http://localhost:3232/department/algorithms/gaa/2/gaa_2.html#image.2.2#image.2.2), являются ребра (3, 10), (3, 11), (6,
7), (7, 8), (7, 13).

Легко доказываются следующие свойства шарниров и перешейков:

**Теорема 3.**Вершина aявляется шарниром тогда и только тогда, когда в графе имеются такие отличные от aвершины bи c, что любой путь, соединяющий bи c, проходит через a.

**Теорема 4.** Ребро является перешейком в том и только том случае, если в графе нет простого цикла, содержащего это ребро.

### Метрические характеристики графов

Расстоянием между двумя вершинами графа называется наименьшая длина пути, соединяющего эти вершины. Расстояние между вершинами aи bобозначается через d\left(a,b\right). Если в графе нет пути, соединяющего aи b, то есть эти вершины принадлежат разным компонентам связности, то расстояние между ними считается бесконечным.

Функция d\left(x,y\right)обладает следующими свойствами:

1. d\left(x,y\right)\ge 0, причем d\left(x,y\right)=0тогда и только тогда, когда x=y;
2. d\left(x,y\right)=d\left(y,x\right);
3. d\left(x,y\right)+d\left(y,z\right)\ge d\left(x,z\right)(неравенство треугольника).

В математике функцию двух переменных, определенную на некотором множестве и удовлетворяющую условиям 1 - 3, называют метрикой, а множество, на котором задана метрика, - метрическим пространством. Таким образом, множество вершин любого графа можно рассматривать как метрическое пространство.

Расстояние от данной вершины aдо наиболее удаленной от нее вершины называется эксцентриситетом вершины aи обозначается через \ecc\left(a\right). Таким образом,

ecc(a)=\max_{x\in VG} d(a,x).

Вершину с наименьшим эксцентриситетом называют центральной, а вершину с наибольшим эксцентриситетом - периферийной. Множество всех центральных вершин называется центром графа. Сама величина наименьшего эксцентриситета называется радиусом графа и обозначается через rad(G), а величина наибольшего - диаметром и обозначается diam(G). Иначе говоря,


rad(G)=\min_{x\in VG} \max_{y\in VG} d(x,y),\\

diam(G)=\max_{x\in VG} \max_{y\in VG} d(x,y).

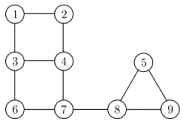
Наименьший диаметр имеет полный граф - его диаметр равен 1. Среди связных графов с nвершинами наибольший диаметр, равный n-1, имеет цепь P_{n}.

Если расстояние между двумя вершинами равно диаметру графа, то кратчайший путь, соединяющий эти вершины, называется диаметральным путем, а подграф, образованный вершинами и ребрами этого пути, - диаметральной цепью.

Для графа, изображенного на [рис. 3.3](http://localhost:3232/department/algorithms/gaa/2/gaa_2.html#image.2.3#image.2.3), эксцентриситеты вершин приведены в следующей таблице:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| x | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| \ecc(x) | 5 | 4 | 4 | 3 | 5 | 3 | 3 | 4 | 5 |

Центр этого графа составляют вершины 4, 6, 7; периферийные вершины - 1, 5и 9; радиус его равен 3, а диаметр 5. Одна из диаметральных цепей порождается множеством вершин \{1, 3, 6, 7, 8, 9\}.

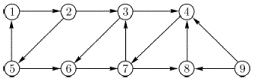


**Рис. 3.3.**

### Маршруты и связность в орграфах

Для ориентированного графа можно определить два типа маршрутов. Неориентированный маршрут (или просто маршрут) - это чередующаяся последовательность x_{1},e_{1},x_{2}, e_{2} \ldots e_{k-1},x_{k}вершин и ребер графа, такая, что для каждого i=1,2\ldots k-1выполняется одно из двух: e_{i} =(x_{i},x_{i+1})или e_{i} =\quad \left(x_{i+1},x_{i} \right). Маршрут называется ориентированным (или ормаршрутом), если e_{i} =(x_{i},x_{i+1})для каждого i. Таким образом, при движении вдоль маршрута в орграфе ребра могут проходиться как в направлении ориентации, так и в обратном направлении, а при движении вдоль ормаршрута - только в направлении ориентации. Это различие очевидным образом распространяется на пути и циклы, так что в орграфе можно рассматривать пути и орпути, циклы и орциклы. Будем говорить, что маршрут соединяет вершины x_{1}и x_{k}, а ормаршрут ведет из x_{1}в x_{k}.

Соответственно двум типам маршрутов определяются и два типа связности орграфов. Орграф называется связным (или слабо связным), если для каждой пары вершин в нем имеется соединяющий их маршрут; он называется сильно связным, если для каждой упорядоченной пары вершин (a,b) в нем имеется ормаршрут, ведущий из aв b. Максимальные по включению подмножества вершин орграфа, порождающие сильно связные подграфы, называются его областями сильной связности, а порождаемые ими подграфы - компонентами сильной связности. Очевидно, разные области сильной связности не могут иметь общих вершин, так что множество вершин каждого орграфа разбивается на области сильной связности. Областями сильной связности орграфа на [рис. 3.4](http://localhost:3232/department/algorithms/gaa/2/gaa_2.html#image.2.4#image.2.4) являются множества \{1, 2, 5\}, \{3, 4, 6, 7, 8\}, \{9\}.



**Рис. 3.4.**

### Эйлеровы пути и циклы

Первая теорема теории графов была доказана задолго до того, как стало употребляться словосочетание "теория графов". В 1736 г. появилась работа Эйлера, в которой не только была решена предложенная ему задача о кенигсбергских мостах, но и сформулировано общее правило, позволяющее решить любую задачу такого рода. Интересно, что в одном из писем Эйлер писал по этому поводу:

*"... это решение по своему характеру, по-видимому, имеет мало отношения к математике, и мне непонятно, почему следует скорее от математика ожидать этого решения, нежели от какого-нибудь другого человека ..."*

На языке теории графов задача состоит в том, чтобы определить, имеется ли в графе путь, проходящий через все его ребра (напомним, что путь, по определению, не может дважды проходить по одному ребру). Такой путь называется эйлеровым путем, а если он замкнут, то эйлеровым циклом. В графе, изображенном на [рис. 3.5 а](http://localhost:3232/department/algorithms/gaa/2/gaa_2.html#image.2.5#image.2.5), эйлеров цикл существует - например, последовательность вершин 1, 2, 4, 5, 2, 3, 5, 6, 3, 1образует такой цикл. В графе же на [рисунке 3.5 б](http://localhost:3232/department/algorithms/gaa/2/gaa_2.html#image.2.5#image.2.5) эйлерова цикла нет, но есть эйлеровы пути, например, 2, 4, 5, 2, 1, 3, 5, 6, 3.



**Рис. 3.5.**

Рассмотрим сначала условия существования эйлерова цикла в обыкновенном графе. Ясно, что в несвязном графе эйлеров цикл может существовать только в том случае, когда все его ребра принадлежат одной компоненте связности, а все остальные компоненты - просто изолированные вершины. Поэтому достаточно рассматривать связные графы.

**Теорема 5.** Эйлеров цикл в связном графе существует тогда и только тогда, когда в нем степени всех вершин четны.

**Доказательство.**

Необходимость условия очевидна, так как при каждом прохождении цикла через какую-либо вершину используются два ребра - по одному из них маршрут входит в вершину, по другому выходит из нее (это относится и к стартовой вершине - в ней ведь маршрут должен закончиться). Докажем его достаточность.

Пусть G- связный граф, в котором больше одной вершины и степени всех вершин четны. Значит, степень каждой вершины не меньше 2, поэтому по теореме 2 в графе Gимеется цикл Z_{1}. Если удалить все ребра этого цикла из графа G, то получится граф G_{1}, в котором степени вершин также четны. Если в G_{1}нет ни одного ребра, то Z_{1}- эйлеров цикл. В противном случае, применяя ту же теорему 2 к графу, полученному из G_{1}удалением всех изолированных вершин, заключаем, что в G_{1}имеется цикл Z_{2}. Удалив из G_{1}все ребра цикла Z_{2}, получим граф G_{2}. Продолжая действовать таким образом, пока не придем к пустому графу, получим в итоге систему циклов Z_{1} \ldots Z_{k}, причем каждое ребро графа принадлежит в точности одному из них. Покажем теперь, что из этих циклов можно составить один цикл. Действительно, из того, что исходный граф связен, следует, что хотя бы один из циклов Z_{1} \ldots Z_{k-1}имеет общую вершину с Z_{k}. Допустим, для определенности, что таков цикл Z_{k-1}. Пусть Z_{k} =x_{1},x_{2} \ldots x_{p}, Z_{k-1} = y_{1},y_{2}
\ldots y_{q}, и x_{i} =y_{j}для некоторых iи j. Тогда последовательность вершин

 Z'_{k-1} =x_{1},x_{2} \ldots x_{i},y_{j+1},y_{j+2} \ldots y_{q},y_{2} \ldots
y_{j},x_{i+1}\ldots x_{p} 

очевидно, является циклом, а множество ребер этого цикла есть объединение множеств ребер циклов Z_{k-1}и Z_{k}. Таким образом, получаем систему из меньшего числа циклов, по-прежнему обладающую тем свойством, что каждое ребро графа принадлежит в точности одному из них. Действуя далее таким же образом, в конце концов получим один цикл, который и будет эйлеровым.

Теорема 5 верна и для мультиграфов (кстати, в задаче о кенигсбергских мостах ситуация моделируется именно мультиграфом). Она остается верной и при наличии петель, если при подсчете степеней вершин каждую петлю считать дважды.

Теперь нетрудно получить и критерий существования эйлерова пути.

**Теорема 6.** Эйлеров путь в связном графе существует тогда и только тогда, когда в нем имеется не более двух вершин с нечетными степенями.

**Доказательство.**

Если в графе нет вершин с нечетными степенями, то, по предыдущей теореме, в нем имеется эйлеров цикл, он является и эйлеровым путем. Не может быть точно одной вершины с нечетной степенью - это следует из теоремы 2. Если же имеются точно две вершины с нечетными степенями, то построим новый граф, добавив ребро, соединяющее эти вершины. В новом графе степени всех вершин четны и, следовательно, существует эйлеров цикл (возможно, что при добавлении нового ребра получатся кратные ребра, но, как отмечалось выше, теорема об эйлеровом цикле верна и для мультиграфов). Так как циклический сдвиг цикла - тоже цикл, существует и такой эйлеров цикл, в котором добавленное ребро - последнее. Удалив из этого цикла последнюю вершину, получим эйлеров путь в исходном графе.

В ориентированном графе под эйлеровым путем (циклом) понимают ориентированный путь (цикл), проходящий через все ребра графа. Ориентированный вариант критерия существования эйлерова цикла формулируется следующим образом.

**Теорема 7.** Эйлеров цикл в связном орграфе существует тогда и только тогда, когда у каждой его вершины число входящих в нее ребер равно числу выходящих.

### Построение эйлерова цикла

Напомним, что эйлеровым циклом называется замкнутый маршрут, в котором каждое ребро графа встречается точно один раз. Согласно теореме 5, для существования такого маршрута в связном графе необходимо и достаточно, чтобы степени всех вершин были четными. Теперь рассмотрим алгоритм, который находит эйлеров цикл в заданном графе при условии, что условия связности и четности степеней выполнены.

Этот алгоритм похож на алгоритм поиска в глубину: начиная с произвольно выбранной стартовой вершины a, строим путь, выбирая каждый раз для дальнейшего продвижения еще не пройденное ребро. Главное отличие от поиска в глубину состоит в том, что как пройденные помечаются именно ребра, а не вершины. Поэтому одна и та же вершина может посещаться несколько раз, но каждое ребро проходится не более одного раза, так что в полученном маршруте ребра не будут повторяться. Вершины пути накапливаются в стеке S. Через некоторое количество шагов неизбежно наступит тупик - все ребра, инцидентные активной (последней посещенной) вершине x, уже пройдены. Так как степени всех вершин графа четны, в этот момент x=aи пройденные ребра образуют цикл, но он может включать не все ребра графа. Для обнаружения еще не пройденных ребер возвращаемся по пройденному пути, перекладывая вершины из стека Sв другой стек C, пока не встретим вершину x, которой инцидентно непройденное ребро. Так как граф связен, такая вершина обязательно встретится. Тогда возобновляем движение вперед по непройденным ребрам, пока не дойдем до нового тупика и т.д. Процесс заканчивается, когда в очередном тупике обнаруживается, что Sпуст. В этот момент в стеке Cнаходится последовательность вершин эйлерова цикла.

**Алгоритм 1.** Построение эйлерова цикла

1. выбрать произвольно вершину a
2. a\Rightarrow S
3. **while** S\ne \emptyset**do**
4. x:=top(S)
5. **if** имеется непройденное ребро (x,y)
6. **then** пометить ребро (x,y)как пройденное
7. y\Rightarrow S
8. **else** переместить вершину xиз Sв C

Для обоснования алгоритма заметим сначала, что первой в стек Sпомещается вершина a, и она будет последней перемещена из Sв C. Следовательно, она будет последней вершиной в стеке C. Далее, как было отмечено выше, первый раз, когда обнаружится, что все инцидентные активной вершине ребра пройдены (т.е. будет выполняться ветвь **else** в строке 8), активной будет стартовая вершина a. Значит, эта вершина будет первой перемещена из Sв C. Итак, по окончании работы алгоритма в начале и в конце последовательности вершин, содержащейся в стеке C, находится вершина a. Иначе говоря, если эта последовательность представляет маршрут (а далее будет показано, что так оно и есть), то этот маршрут замкнут.

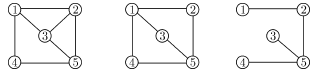
Далее отметим, что в конечном итоге каждое ребро будет пройдено. Действительно, допустим, что в момент окончания работы алгоритма имеются еще не пройденные ребра. Поскольку граф связен, должно существовать хотя бы одно непройденное ребро, инцидентное посещенной вершине. Но тогда эта вершина не могла быть удалена из стека S, и Sне мог стать пустым.

Будем говорить, что ребро (x,y)представлено в стеке (S или C), если в какой-то момент работы алгоритма в стеке рядом находятся вершины xи y. Ясно, что каждое ребро графа будет представлено в стеке Sи что каждые две вершины, расположенные рядом в этом стеке, образуют ребро. Допустим, в какой-то момент из стека Sв стек Cперемещается вершина x, а непосредственно под ней в стеке Sнаходится вершина y. Возможно, что вершина yбудет перемещена из Sв Cпри следующем повторении цикла **while**, тогда ребро (x,y)будет представлено в стеке C. Другая возможность - между перемещением вершины xи следующим перемещением, т.е. следующим выполнением ветви **else**, будет несколько раз выполнена ветвь **then** (строки 6,\,7). Это означает, что будет пройдена некоторая последовательность ребер, начинающаяся в вершине y. Ввиду четности степеней эта последовательность может закончиться только в вершине y. Значит, и в этом случае следующей за вершиной xбудет перемещена из Sв Cвершина y. В любом случае ребро (x,y)будет представлено в стеке C. Из этого рассуждения видно, что последовательность вершин в стеке Cявляется маршрутом и что каждое ребро графа в конечном итоге будет содержаться в этом маршруте, причем один раз.

При каждом повторении цикла **while** в рассмотренном алгоритме либо проходится одно ребро, либо одна вершина перемещается из Sв C. Последнее можно трактовать как прохождение уже пройденного однажды ребра в обратном направлении. Каждое ребро в каждом направлении будет пройдено один раз, поэтому общая трудоемкость этого алгоритма оценивается как O(m). Необходимо только оговориться, что этот вывод, как и аналогичные заключения об алгоритмах обхода в первых разделах этой главы, справедлив лишь при определенных предположениях о том, как задан граф. Способ задания должен обеспечить возможность быстрого просмотра множества ребер, инцидентных данной вершине. Подходящим является, например, задание графа списками инцидентности, в которых для каждой вершины перечисляются инцидентные ей ребра. Необходимо также иметь возможность быстро пометить ребро как пройденное или проверить, пройдено ли данное ребро. Для этого подходящей структурой может служить характеристический массив на множестве ребер.

### Гамильтоновы пути и циклы

Гамильтоновым циклом (путем) называют простой цикл (путь), содержащий все вершины графа. В графе, изображенном на [рис. 3.](http://localhost:3232/department/algorithms/gaa/8/gaa_8.html#image.8.1#image.8.1)6 слева, гамильтоновым циклом является, например, последовательность 1, 2, 3, 5, 4, 1. В графе, изображенном в центре, нет гамильтоновых циклов, но есть гамильтоновы пути, например, 2,1,3,5,4. В правом графе нет и гамильтоновых путей.



**Рис. 3.6.**

Внешне определение гамильтонова цикла похоже на определение эйлерова цикла. Однако есть кардинальное различие в сложности решения соответствующих задач распознавания и построения. Мы видели, что имеется достаточно простой критерий существования эйлерова цикла и эффективный алгоритм его построения. Для гамильтоновых же циклов (и путей) неизвестно никаких просто проверяемых необходимых и достаточных условий их существования, а все известные алгоритмы требуют для некоторых графов перебора большого числа вариантов.

Гамильтонов цикл представляет собой, с комбинаторной точки зрения, просто перестановку вершин графа. При этом в качестве начальной вершины цикла можно выбрать любую вершину, так что можно рассматривать перестановки с фиксированным первым элементом. Самый бесхитростный план поиска гамильтонова цикла состоит в последовательном рассмотрении всех этих перестановок и проверке для каждой из них, представляет ли она цикл в данном графе. Такой способ действий уже при не очень большом числе вершин становится практически неосуществимым ввиду быстрого роста числа перестановок - имеется (n-1)!перестановок из nэлементов с фиксированным первым элементом.

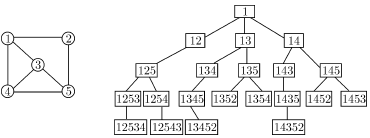
Более рациональный подход состоит в рассмотрении всевозможных простых путей, начинающихся в произвольно выбранной стартовой вершине a, до тех пор, пока не будет обнаружен гамильтонов цикл или все возможные пути не будут исследованы. По сути дела, речь тоже идет о переборе перестановок, но значительно сокращенном - если, например, вершина bне смежна с вершиной a, то все (n-2)!перестановок, у которых на первом месте стоит a, а на втором b, не рассматриваются.

Рассмотрим этот алгоритм подробнее. Будем считать, что граф задан окрестностями вершин: для каждой вершины xзадано множество вершин, смежных с x. На каждом шаге алгоритма имеется уже построенный отрезок пути, он хранится в стеке **PATH**. Для каждой вершины x, входящей в **PATH**, хранится множество N(x)всех вершин, смежных с x, которые еще не рассматривались в качестве возможных продолжений пути из вершины x. Когда вершина xдобавляется к пути, множество N(x)полагается равным V(x). В дальнейшем рассмотренные вершины удаляются из этого множества. Очередной шаг состоит в исследовании окрестности последней вершины xпути **PATH**. Если N(x)\ne \varnothingи в N(x)имеются вершины, не принадлежащие пути, то одна из таких вершин добавляется к пути. В противном случае вершина xисключается из стека. Когда после добавления к пути очередной вершины оказывается, что путь содержит все вершины графа, остается проверить, смежны ли первая и последняя вершины пути, и при утвердительном ответе выдать очередной гамильтонов цикл.

**Алгоритм 2.** Поиск гамильтоновых циклов

1. выбрать произвольно вершину a
2. a\Rightarrow PATH
3. N(a):=V(a)
4. **while** PATH\ne
   \varnothing**do**
5. x\, :={\rm top}(PATH)
6. **if** N(x)\ne \varnothing
7. **then** взять y\in
   N(x)
8. N(x):=N(x)-y
9. **if** вершина yне находится в **PATH**
10. **then** y\Rightarrow PATH
11. N(y):=V(y)
12. **if** **PATH** содержит все вершины
13. **then** **if** yсмежна с a
14. **then** выдать цикл
15. **else** удалить вершину xиз **PATH**

Этот алгоритм очень похож на алгоритм поиска в глубину и отличается от него по существу только тем, что открытая вершина, когда вся ее окрестность исследована, не закрывается, а опять становится новой (исключается из стека). В начале все вершины новые. Процесс заканчивается, когда все вершины опять станут новыми. На самом деле это и есть поиск в глубину, только не в самом графе, а в дереве путей. Вершинами этого дерева являются всевозможные простые пути, начинающиеся в вершине a, а ребро дерева соединяет два пути, один из которых получается из другого добавлением одной вершины в конце. На [рис. 3.](http://localhost:3232/department/algorithms/gaa/8/gaa_8.html#image.8.2#image.8.2)7 показаны граф и его дерево путей из вершины 1.



**Рис. 3.7.**

В худшем случае время работы этого алгоритма тоже растет с факториальной скоростью. Например, для графа K_{n-1} +K_{1}(граф с двумя компонентами связности, одна из которых - полный граф с n-1вершиной, другая - изолированная вершина), если в качестве стартовой выбрана не изолированная вершина, то будут рассмотрены все {(n-2)!}простых путей длины n-2в большой компоненте. Вместе с тем, если перед поиском гамильтонова цикла исходный граф проверить на связность, то ответ будет получен быстро. Можно пойти дальше и при обходе дерева путей поверять на связность каждый встречающийся "остаточный граф", т.е. граф, получающийся из исходного удалением всех вершин рассматриваемого пути. Если этот граф несвязен, то этот путь не может быть продолжен до гамильтонова пути. Поэтому можно не исследовать соответствующую ветвь дерева, а вернуться к рассмотрению более короткого пути, удалив последнюю вершину (т.е. сделать "шаг назад" в поиске в глубину). Можно пойти еще дальше и заметить, что если некоторая вершина xDFS-дерева с корнем aявляется развилкой, т.е. имеет не менее двух сыновей, то в подграфе исходного графа, полученном удалением всех предков этой вершины, кроме нее самой, она будет шарниром. Поэтому путь от aдо xв DFS-дереве не может быть продолжен до гамильтонова пути. Эти соображения приводят к следующей модификации алгоритма: обходим граф поиском в глубину с построением DFS-дерева, затем находим в этом дереве самую нижнюю развилку (развилку с наименьшим глубинным номером). Если ни одной развилки нет, то само DFS-дерево представляет собой гамильтонов путь и остается только проверить наличие ребра, соединяющего начало и конец пути. Если же x- развилка, то возвращаемся из xв предшествующую вершину пути, помечаем все вершины, кроме собственных предков вершины x, как непосещенные и возобновляем поиск в глубину с этого места.

Рассмотрим другой алгоритм, выясняющий существование гамильтонова цикла, по сути близкий к поиску в ширину и имеющий не столь быстро (хотя все же быстро) растущую оценку трудоемкости.

Пусть граф Gзадан матрицей смежности A=\left\|
A(i,j)\right\|. Выберем произвольно стартовую вершину aи определим для каждого k=0,1\ldots, n-2функцию H_{k} (x,X), где значениями переменной xявляются вершины, отличные от a, а значениями переменной X- k-элементные подмножества множества VG-\{ a\}, причем вершина xне должна принадлежать множеству X. Эти функции определяются так: полагаем H_{k} (x,X)=1, если существует простой путь длины k+1из вершины aв вершину x, проходящий только через вершины из множества X, и H_{k} (x,X)=0, если такого пути не существует. Тогда


H_{0} (x,\varnothing)=A(a,x) \qquad \text{для всех } x,


а для k>0


H_{k} (x,X)=\mathop{\vee}\limits_{y\in X} H_{k-1}
(y,X-y)A(y,x).
}

Таким образом, зная все значения функции H_{k-1}, мы можем вычислить все значения функции H_{k}, причем для вычисления одного значения требуется выполнить 2k-1логических операций. Общее количество логических операций для вычисления всех этих функций составит


(n-1)\suml_{k=1}^{n-2}\begin{pmatrix} {n-2}
\\ k \end{pmatrix}
(2k-1)=(n-1)(2^{n-2} (n-3)+1)=O(n^{2} 2^{n}).


После того, как будет вычислена функция H_{n-2}, останется только для всех x, для которых H_{n-2} (x,X)=1(X в этом случае определяется однозначно), выяснить, чему равно A(x,a)- если хотя бы в одном случае это 1, то гамильтонов цикл существует. Очевидный недостаток данного алгоритма - необходимость хранения большого количества промежуточной информации.

**Тема 2. Алгоритмы комбинаторного перебора (6 часов).**

План лекций.

1. Базовые комбинаторные объекты.

Размещения. Размещения с повторениями. Перестановки. Методы генерации. Подмножества. Разбиения.

2. Коды Грея.

Понятие кодов Грея. Построение кодов Грея. Применения кодов Грея.

3. Применение методов комбинаторного перебора.

Числа Каталана. Расстановка скобок. Подсчет количеств. Комбинаторные методы в олимпиадных задачах.

**Лекция 4. Базовые комбинаторные объекты**.

### Размещения с повторениями

Напечатать все последовательности длины k из чисел 1..n.

**Решение**. Будем печатать их в лексикографическом порядке (последовательность a предшествует последовательности b, если для некоторого s их начальные отрезки длины s равны, а (s+1) -ый член последовательности a меньше). Первой будет последовательность <1,1,...,1>, последней - последовательность <n,n,...,n>. Будем хранить последнюю напечатанную последовательность в массиве x[1]..x[k].

...x[1]...x[k] положить равными 1

...напечатать x

...last[1]...last[k] положить равным n

{напечатаны все до x включительно}

while x <> last do begin

| ...x := следующая за x последовательность

| ...напечатать x

end;

Опишем, как можно перейти от x к следующей последовательности. Согласно определению, у следующей последовательности первые s членов должны быть такими же, а (s+1) -ый - больше. Это возможно, если x[s+1] меньше n. Среди таких s нужно выбрать наибольшее (иначе полученная последовательность не будет непосредственно следующей). Соответствующее x[s+1] нужно увеличить на 1. Итак, надо, двигаясь с конца последовательности, найти самый правый член, меньший n (он найдется, т.к по предположению x<>last ), увеличить его на 1, а идущие за ним члены положить равными 1.

p:=k;

while not (x[p] < n) do begin

| p := p-1;

end;

{x[p] < n, x[p+1] =...= x[k] = n}

x[p] := x[p] + 1;

for i := p+1 to k do begin

| x[i]:=1;

end;

**Замечание**. Если членами последовательности считать числа не от 1 до n, а от 0 до n-1, то переход к следующему соответствует прибавлению единицы в n -ичной системе счисления.

В предложенном алгоритме используется сравнение двух массивов ( x <> last ). Устранить его, добавив булевскую переменную l и включив в инвариант соотношение

6f2b2544400dbae842c05ba9c8d36d59

Напечатать все подмножества множества {1...k}.

**Решение**. Подмножества находятся во взаимно однозначном соответствии с последовательностями нулей и единиц длины k.

Напечатать все последовательности положительных целых чисел длины k, у которых i -ый член не превосходит i.

### Перестановки

Напечатать все перестановки чисел 1..n (то есть последовательности длины n, в которые каждое из этих чисел входит по одному разу).

**Решение**. Перестановки будем хранить в массиве x[1]..x[n] и печатать в лексикографическом порядке. (Первой при этом будет перестановка 5d00829d71a8b37a8f5c4d2893d5d768, последней - 121952df47e7507e20faaf1d8a17182e. Для составления алгоритма перехода к следующей перестановке зададимся вопросом: в каком случае k -ый член перестановки можно увеличить, не меняя предыдущих? Ответ: если он меньше какого-либо из следующих членов (т.е. членов с номерами больше k ). Мы должны найти наибольшее k, при котором это так, т.е. такое k, что

5afb283d28f89769a3da53ff620d1de8

После этого значение x[k] нужно увеличить минимальным возможным способом, т.е. найти среди x[k+1]..x[n] наименьшее число, большее его. Поменяв x[k] с ним, остается расположить числа с номерами k+1..n так, чтобы перестановка была наименьшей, т.е. в возрастающем порядке. Это облегчается тем, что они уже расположены в убывающем порядке.

Алгоритм перехода к следующей перестановке:

{<x[1]...x[n]> <> <n...2,1>}

k:=n-1;

{последовательность справа от k убывающая: x[k+1]>...>x[n]}

while x[k] > x[k+1] do begin

| k:=k-1;

end;

{x[k] < x[k+1] > ... > x[n]}

t:=k+1;

{t <=n, все члены отрезка x[k+1] > ... > x[t] больше x[k]}

while (t < n) and (x[t+1] > x[k]) do begin

| t:=t+1;

end;

{x[k+1] > ... > x[t] > x[k] > x[t+1] > ... > x[n]}

... обменять x[k] и x[t]

{x[k+1] > ... > x[n]}

... переставить участок x[k+1] ... x[n] в обратном порядке

**Замечание**. Программа имеет знакомый дефект: если t=n, то x[t+1] не определено.

### Подмножества

Для заданных n и k ( 0b4ac9d367f75b90ea55a50922b2cf46) перечислить все k -элементные подмножества множества {1..n} }.

**Решение**. Будем представлять каждое подмножество последовательностью x[1]..x[n] нулей и единиц длины n, в которой ровно k единиц. (Другой способ представления разберем позже.) Такие последовательности упорядочим лексикографически (см. выше). Очевидный способ решения задачи - перебирать все последовательности как раньше, а затем отбирать среди них те, у которых k единиц - мы отбросим, считая его неэкономичным (число последовательностей с k единицами может быть много меньше числа всех последовательностей). Будем искать такой алгоритм, чтобы получение очередной последовательности требовало не более {C d09d4c86aabb57a0587725bb6e992616n} действий.

В каком случае s -ый член последовательности можно увеличить, не меняя предыдущие? Если x[s] меняется с 0 на 1, то для сохранения общего числа единиц нужно справа от х[s] заменить 1 на 0. Для этого надо, чтобы справа от x[s] единицы были. Если мы хотим перейти к непосредственно} следующему, то x[s] должен быть первым справа} нулем, за которым стоят единицы. Легко видеть, что х[s+1]=1 (иначе х[s] не первый). Таким образом надо искать наибольшее s, для которого х[s]=0, x[s+1]=1:

02_01

За х[s+1] могут идти еще несколько единиц, а после них несколько нулей. Заменив х[s] на 1, надо выбрать идущие за ним члены так, чтобы последовательность была бы минимальна с точки зрения нашего порядка, т.е. чтобы сначала шли нули, а потом единицы. Вот что получается:

23bbc0b69f667b2314a206ff8ae71fe7

feb117dd244d91f7056b9a51e5c26eae

f59837330f4752608d6bb6a6a5dba59e

s := n - 1;

while not ((x[s]=0) and (x[s+1]=1)) do begin

| s := s - 1;

end;

{s - член, подлежащий изменению с 0 на 1}

num:=0;

for k := s to n do begin

| num := num + x[k];

end;

{num - число единиц на участке x[s]...x[n], число нулей

равно (длина - число единиц), т.е. (n-s+1) - num}

x[s]:=1;

for k := s+1 to n-num+1 do begin

| x[k] := 0;

end;

{осталось поместить num-1 единиц в конце}

for k := n-num+2 to n do begin

| x[k]:=1;

end;

Другой способ представления подмножеств - это перечисление их элементов. Чтобы каждое подмножество имело ровно одно представление, договоримся перечислять элементы в возрастающем порядке. Приходим к такой задаче.

Перечислить все возрастающие последовательности длины k из чисел 1..n в лексикографическом порядке. (Пример: при n=5, k=2 получаем: 12 13 14 15 23 24 25 34 35 45.)

**Решение**. Минимальной будет последовательность 43635876957b3e4e28c1187b1985a692; максимальной - c23cdb38e84994afa912dd3613de6a40. В каком случае s -ый член последовательности можно увеличить? Ответ: если он меньше n-k+s. После увеличения s -го элемента все следующие должны возрастать с шагом 1. Получаем такой алгоритм перехода к следующему:

s:=n;

while not (x[s] < n-k+s) do begin

| s:=s-1;

end;

{s - номер элемента, подлежащего увеличению};

x[s] := x[s]+1;

for i := s+1 to n do begin

| x[i] := x[i-1]+1;

end;

Пусть мы решили представлять k -элементные подмножества множества {1..n} убывающими последовательностями длины k, упорядоченными по-прежнему лексикографически. (Пример: eefd73812899a290fe946c38bee0c543.) Как выглядит тогда алгоритм перехода к следующей?

**Ответ**. Ищем наибольшее s, для которого х[s+1]+1 < x[s]. (Если такого s нет, полагаем s=0.) Увеличив x[s+1] на 1, кладем остальные минимально возможными ( x[t]=k+1-t для t>s ).

Решить две предыдущие задачи, заменив лексикографический порядок на обратный (раньше идут те, которые больше в лексикографическом порядке).

Перечислить все вложения (функции, переводящие разные элементы в разные) множества \{1..k} в {1..n} } (предполагается, что 5971074b9deff57f754f68976061cacd). Порождение очередного элемента должно требовать не более 454ba716a3ff862d81e869f636c24137действий.

**Указание**. Эта задача может быть сведена к перечислению подмножеств и перестановок элементов каждого подмножества.

### Разбиения

Перечислить все разбиения целого положительного числа n на целые положительные слагаемые (разбиения, отличающиеся лишь порядком слагаемых, считаются за одно). (Пример: n=4, разбиения 1+1+1+1, 2+1+1, 2+2, 3+1, 4.)

**Решение**. Договоримся, что (1) в разбиениях слагаемые идут в невозрастающем порядке, (2) сами разбиения мы перечисляем в лексикографическом порядке. Разбиение храним в начале массива x[1]..x[n], при этом количество входящих в него чисел обозначим k. В начале x[1]=...=x[n]=1, k=n, в конце x[1]=n, k=1.

В каком случае x[s] можно увеличить, не меняя предыдущих? Во-первых, должно быть x[s-1]>x[s] или s=1. Во-вторых, s должно быть не последним элементом (увеличение s надо компенсировать уменьшением следующих). Увеличив s, все следующие элементы надо взять минимально возможными.

s := k - 1;

while not ((s=1) or (x[s-1] > x[s])) do begin

| s := s-1;

end;

{s - подлежащее увеличению слагаемое}

x [s] := x[s] + 1;

sum := 0;

for i := s+1 to k do begin

| sum := sum + x[i];

end;

{sum - сумма членов, стоявших после x[s]}

for i := 1 to sum-1 do begin

| x [s+i] := 1;

end;

k := s+sum-1;

Представляя по-прежнему разбиения как невозрастающие последовательности, перечислить их в порядке, обратном лексикографическому (для n=4, например, должно быть 391fefa24da008714ac544b9aa016e4b, 99731179dbf51771f6aad0a040ccb960, 9519ccb642bc8d732b72305a873e61a5, c5f20fb9aebb6fa5be15de6a437ece93, 9d4e5d5fb1a2e979f36a0d4097ac16c6).

**Указание**. Уменьшать можно первый справа член, не равный 1 ; найдя его, уменьшим на 1, а следующие возьмем максимально возможными (равными ему, пока хватает суммы, а последний - сколько останется).

Представляя разбиения как неубывающие последовательности, перечислить их в лексикографическом порядке. Пример для 6b324a31fc723f28e9fcd8d3cd89d2dc.

**Указание**. Последний член увеличить нельзя, а предпоследний - можно; если после увеличения на 1 предпоследнего члена за счет последнего нарушится возрастание, то из двух членов надо сделать один, если нет, то последний член надо разбить на слагаемые, равные предыдущему, и остаток, не меньший его.

Представляя разбиения как неубывающие последовательности, перечислить их в порядке, обратном лексикографическому. Пример для f3021166662b8ec7418a09c066a6ee02.

**Указание**. Чтобы элемент x[s] можно было уменьшить, необходимо, чтобы s=1 или x[s-1] < x[s]. Если x[s] не последний, то этого и достаточно. Если он последний, то нужно, чтобы cc7021f5cb43683a431302f1799c5c95или s=1. (Здесь 938c7c4438ce41cf271ca729899e5cb6обозначает целую часть a7bfda5fcc7189c6d8c11e536498a81a.)

**Лекция 5. Коды Грея.**

### Коды Грея и аналогичные задачи

Иногда бывает полезно перечислять объекты в таком порядке, чтобы каждый следующий минимально отличался от предыдущего. Рассмотрим несколько задач такого рода.

Перечислить все последовательности длины n из чисел 1..k в таком порядке, чтобы каждая следующая отличалась от предыдущей в единственной цифре, причем не более, чем на 1.

**Решение**. Рассмотрим прямоугольную доску ширины n и высоты k. На каждой вертикали будет стоять шашка. Таким образом, положения шашек соответствуют последовательностям из чисел 1..k длины n ( s -ый член последовательности соответствует высоте шашки на s -ой вертикали). На каждой шашке нарисуем стрелочку, которая может быть направлена вверх или вниз. Вначале все шашки поставим на нижнюю горизонталь стрелочкой вверх. Далее двигаем шашки по такому правилу: найдя самую правую шашку, которую можно подвинуть в направлении (нарисованной на ней) стрелки, двигаем ее на одну клетку в этом направлении, а все стоящие правее нее шашки (они уперлись в край) разворачиваем кругом.

Ясно, что на каждом шаге только одна шашка сдвигается, т.е. один член последовательности меняется на 1. Докажем индукцией по n, что проходятся все последовательности из чисел 1..k. Случай n=1 очевиден. Пусть n>1. Все ходы поделим на те, где двигается последняя шашка, и те, где двигается не последняя. Во втором случае последняя шашка стоит у стены, и мы ее поворачиваем, так что за каждым ходом второго типа следует k-1 ходов первого типа, за время которых последняя шашка побывает во всех клетках. Если мы теперь забудем о последней шашке, то движения первых n-1 по предположению индукции пробегают все последовательности длины n-1 по одному разу; движения же последней шашки из каждой последовательности длины n-1 делают k последовательностей длины n.

В программе, помимо последовательности x[1]..x[n], будем хранить массив d[1]..d[n] из чисел +1 и -1 ( +1 соответствует стрелке вверх, -1 - стрелке вниз).

Начальное состояние: x[1]=...=x[n]=1 ; d[1]=...=d[n]=1.

Приведем алгоритм перехода к следующей последовательности (одновременно выясняется, возможен ли переход - ответ становится значением булевской переменной p ).

{если можно, сделать шаг и положить p := true, если нет,

положить p := false }

i := n;

while (i > 1) and

| (((d[i]=1) and (x[i]=n)) or ((d[i]=-1) and (x[i]=1)))

| do begin

| i:=i-1;

end;

if (d[i]=1 and x[i]=n) or (d[i]=-1 and x[i]=1) then begin

| p:=false;

end else begin

| p:=true;

| x[i] := x[i] + d[i];

| for j := i+1 to n do begin

| | d[j] := - d[j];

| end;

end;

**Замечание**. Для последовательностей нулей и единиц возможно другое решение, использующее двоичную систему. (Именно оно связывается обычно с названием "коды Грея".)

Запишем подряд все числа от 7e691cd98b064ce17c0b6a2d0614dd5dдо 6f7ea36fe0f86b00fe26df2728cc4bebв двоичной системе. Например, для 4d2ed561cbb95b61e71c6a6a7637a3bcнапишем:

44b90124e7e0c313c9c53455643573a3

Затем каждое из чисел подвергнем преобразованию, заменив каждую цифру, кроме первой, на ее сумму с предыдущей цифрой (по модулю c90fe44538b32061a458791169cb8983). Иными словами, число 45696ab863b8b4e4adaef7c1771acff6преобразуем в 95cd32e07a71e48cb5a648f9e7a6bbbd(сумма по модулю c90fe44538b32061a458791169cb8983). Для 4d2ed561cbb95b61e71c6a6a7637a3bcполучим:

2e4856fe178321bdec7d894c7896283d

Легко проверить, что описанное преобразование чисел обратимо (и тем самым дает все последовательности по одному разу). Кроме того, двоичные записи соседних чисел отличаются заменой конца fbd45cbc19f668ad4c7be7fcb62bac73на конец b975e5e1dde20eaefd1d8327d15094a5, что - после преобразования - приводит к изменению единственной цифры.

**Применение кода Грея**. Пусть есть вращающаяся ось, и мы хотим поставить датчик угла поворота этой оси. Насадим на ось барабан, выкрасим половину барабана в черный цвет, половину в белый и установим фотоэлемент. На его выходе будет в половине случаев 7e691cd98b064ce17c0b6a2d0614dd5d, а в половине 51226c368dab44d96ea9d6af652f63d8(т.е. мы измеряем угол "с точностью до dbef04edb9fcfa721a5974a83030409b").

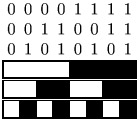
Развертка барабана:

02_02

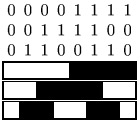
Сделав рядом другую дорожку из двух черных и белых частей и поставив второй фотоэлемент, получаем возможность измерить угол с точностью до 0a2c6e18d73b09c181f0f222c09f5503:



Сделав третью,



мы измерим угол с точностью до 6df90eee4b1037857aba1c2d87ee678bи т.д. Эта идея имеет, однако, недостаток: в момент пересечения границ сразу несколько фотоэлементов меняют сигнал, и если эти изменения произойдут не совсем одновременно, на какое-то время показания фотоэлементов будут бессмысленными. Коды Грея позволяют избежать этой опасности. Сделаем так, чтобы на каждом шаге менялось показание лишь одного фотоэлемента (в том числе и на последнем, после целого оборота).



Написанная нами формула позволяет легко преобразовать данные от фотоэлементов в двоичный код угла поворота.

Заметим также, что геометрически существование кода Грея означает наличие "гамильтонова цикла" в 2d87b8a32ece94992b848596ff06ce88-мерном кубе (возможность обойти все вершины куба по разу, двигаясь по ребрам, и вернуться в исходную вершину).

Напечатать все перестановки чисел 1..n так, чтобы каждая следующая получалась из предыдущей перестановкой (транспозицией) двух соседних чисел. Например, при n=3 допустим такой порядок:

da7232cd9541d39a927ee440c5bed5b5

(между переставляемыми числами вставлены точки).

**Решение**. Наряду с множеством перестановок рассмотрим множество последовательностей y[1]..y[n] целых неотрицательных чисел, для которых a9b43bec91c8eca4f28aa5092f023e16, 947b22204fe71dc8625633c86e4514ec, ce59e81d2662232fab47468046f5aaae. В нем столько же элементов, сколько в множестве всех перестановок, и мы сейчас установим между ними взаимно однозначное соответствие. Именно, каждой перестановке поставим в соответствие последовательность y[1]..y[n], где y[i] - количество чисел, меньших i и стоящих левее i в этой перестановке. Взаимная однозначность вытекает из такого замечания. Перестановка чисел 1..n получается из перестановки чисел 1..n-1 добавлением числа n, которое можно вставить на любое из n мест. При этом к сопоставляемой с ней последовательности добавляется еще один член, принимающий значения от 0 до n-1, а предыдущие члены не меняются. При этом оказывается, что изменение на единицу одного из членов последовательности y соответствует транспозиции двух соседних чисел, если все следующие числа последовательности y принимают максимально или минимально возможные для них значения. Именно, увеличение y[i] на 1 соответствует транспозиции числа i с его правым соседом, а уменьшение - с левым.

Теперь вспомним решение задачи о перечислении всех последовательностей, на каждом шаге которого один член меняется на единицу. Заменив прямоугольную доску доской в форме лестницы (высота i -ой вертикали равна i ) и двигая шашки по тем же правилам, мы перечислим все последовательности y, причем i -ый член будет меняться как раз только если все следующие шашки стоят у края. Надо еще уметь параллельно с изменением y корректировать перестановку. Очевидный способ требует отыскания в ней числа i ; это можно облегчить, если помимо самой перестановки хранить функцию

fe02027ad6417d93f63a8eb3510a7bb9

т.е. обратное к перестановке отображение, и соответствующим образом ее корректировать. Вот какая получается программа:

program test;

| const n=...;

| var

| x: array [1..n] of 1..n; {перестановка}

| inv\_x: array [1..n] of 1..n; {обратная перестановка}

| y: array [1..n] of integer; {y[i] < i}

| d: array [1..n] of -1..1; {направления}

| b: boolean;

|

| procedure print\_x;

| | var i: integer;

| begin

| | for i:=1 to n do begin

| | | write (x[i], ' ');

| | end;

| | writeln;

| end;

|

| procedure set\_first;{первая: y[i]=0 при всех i}

| | var i : integer;

| begin

| | for i := 1 to n do begin

| | | x[i] := n + 1 - i;

| | | inv\_x[i] := n + 1 - i;

| | | y[i]:=0;

| | | d[i]:=1;

| | end;

| end;

|

| procedure move (var done : boolean);

| | var i, j, pos1, pos2, val1, val2, tmp : integer;

| begin

| | i := n;

| | while (i > 1) and (((d[i]=1) and (y[i]=i-1)) or

| | | ((d[i]=-1) and (y[i]=0))) do begin

| | | i := i-1;

| | end;

| | done := (i>1); {упрощение: первый член нельзя менять}

| | if done then begin

| | | y[i] := y[i]+d[i];

| | | for j := i+1 to n do begin

| | | | d[j] := -d[j];

| | | end;

| | | pos1 := inv\_x[i];

| | | val1 := i;

| | | pos2 := pos1 + d[i];

| | | val2 := x[pos2];

| | | {pos1, pos2 - номера переставляемых элементов;

| | | val1, val2 - их значения; val2 < val1}

| | | tmp := x[pos1];

| | | x[pos1] := x[pos2];

| | | x[pos2] := tmp;

| | | tmp := inv\_x[val1];

| | | inv\_x[val1] := inv\_x[val2];

| | | inv\_x[val2] := tmp;

| | end;

| end;

|

begin

| set\_first;

| print\_x;

| b := true;

| {напечатаны все перестановки до текущей включительно;

| если b ложно, то текущая - последняя}

| while b do begin

| | move (b);

| | if b then print\_x;

| end;

end.

**Лекция 6. Применение методов комбинаторного перебора.**

Посмотрим еще раз на использованные нами приемы. Вначале удавалось решить задачу по такой схеме: определяем порядок на подлежащих перечислению объектах и явно описываем процедуру перехода от данного объекта к следующему (в смысле этого порядка). В задаче о кодах Грея потребовалось хранить, помимо текущего объекта, и некоторую дополнительную информацию (направления стрелок). Наконец, в задаче о перечислении перестановок (на каждом шаге допустима одна транспозиция) мы применили такой прием: установили взаимно однозначное соответствие между перечисляемым множеством и другим, более просто устроенным. Таких соответствий в комбинаторике известно много. Мы приведем несколько задач, связанных с так называемыми " числами Каталана ".

Перечислить все последовательности длины 2n, составленные из n единиц и n минус единиц, у которых сумма любого начального отрезка неотрицательна, т.е. число минус единиц в нем не превосходит числа единиц. (Число таких последовательностей называют числом Каталана ; формулу для чисел Каталана см. в следующем разделе.)

**Решение**. Изображая единицу вектором (1,1), а минус единицу вектором (1,-1), можно сказать, что мы ищем пути из точки (0,0) в точку (n,0), не опускающиеся ниже оси абсцисс.

Будем перечислять последовательности в лексикографическом порядке, считая, что -1 предшествует 1. Первой последовательностью будет "пила"

d816d22ac11f58a9c914b37aa418baf1

а последней - "горка"

4ad0ccacef77c6ea9d14b4f8546b3062

Как перейти от последовательности к следующей? До некоторого места они должны совпадать, а затем надо заменить -1 на 1. Место замены должно быть расположено как можно правее. Но заменять -1 на 1 можно только в том случае, если справа от нее есть единица (которую можно заменить на -1 ). После замены -1 на 1 мы приходим к такой задаче: фиксирован начальный кусок последовательности, надо найти минимальное продолжение. Ее решение: надо приписывать -1, если это не нарушит условия неотрицательности, а иначе приписывать 1. Получаем такую программу:

...

type array2n = array [1..2n] of integer;

...

procedure get\_next (var a: array2n; var last: Boolean);

| {в a помещается следующая последовательность, если}

| {она есть (при этом last:=false), иначе last:=true}

| var k, i, sum: integer;

begin

| k:=2\*n;

| {инвариант: в a[k+1..2n] только минус единицы}

| while a[k] = -1 do begin k:=k-1; end;

| {k - максимальное среди тех, для которых a[k]=1}

| while (k>0) and (a[k] = 1) do begin k:=k-1; end;

| {a[k] - самая правая -1, за которой есть 1;

| если таких нет, то k=0}

| if k = 0 then begin

| | last := true;

| end else begin

| | last := false;

| | i:=0; sum:=0;

| | {sum = a[1]+...+a[i]}

| | while i< >k do begin

| | | i:=i+1; sum:= sum+a[i];

| | end;

| | {sum = a[1]+...+a[k], a[k]=-1}

| | a[k]:= 1; sum:= sum+2;

| | {вплоть до a[k] все изменено, sum=a[1]+...+a[k]}

| | while k < > 2\*n do begin

| | | k:=k+1;

| | | if sum > 0 then begin

| | | | a[k]:=-1

| | | end else begin

| | | | a[k]:=1;

| | | end;

| | | sum:= sum+a[k];

| | end;

| | {k=2n, sum=a[1]+...a[2n]=0}

| end;

end;

Перечислить все расстановки скобок в произведении n сомножителей. Порядок сомножителей не меняется, скобки полностью определяют порядок действий. Например, для n=4 есть 5 расстановок:

40b21606e64eb6e796d334b965b39a15

**Указание**. Каждому порядку действий соответствует последовательность команд стекового калькулятора, описанного в [пункте 8.3.](http://www.intuit.ru/department/se/prmalgs/8/)

На окружности задано 2n точек, пронумерованных от 1 до 2n. Перечислить все способы провести n непересекающихся хорд с вершинами в этих точках.

Перечислить все способы разрезать n -угольник на треугольники, проведя n-2 его диагонали.

(Мы вернемся к разрезанию многоугольника в разделе о динамическом программировании, [пункт 8.1.](http://www.intuit.ru/department/se/prmalgs/2/8))

Еще один класс задач на перечисление всех элементов заданного множества мы рассмотрим ниже, обсуждая метод поиска с возвратами (backtracking).

### Подсчет количеств

Иногда можно найти количество объектов с тем или иным свойством, не перечисляя их. Классический пример: 8cf1852454cdaa5fcb81594dc54d93e5- число всех 5b9712f800c303699c96c43db144124e-элементных подмножеств 2d87b8a32ece94992b848596ff06ce88-элементного множества - можно найти, заполняя таблицу по формулам

92f72c147e4fbc8c9a185b0c4e7dbfd3

или по формуле

78a2793726682959660833ce79fc3861

(Первый способ эффективнее, если надо вычислить много значений 8cf1852454cdaa5fcb81594dc54d93e5.)

Приведем другие примеры.

Число разбиений; предлагалась на Всесоюзной олимпиаде по программированию 1988 года. Пусть b026f2805333f8d95dbd1a4196ac3bfd- число разбиений целого положительного 2d87b8a32ece94992b848596ff06ce88на целые положительные слагаемые (без учета порядка, 3a29b4367270de17d18aa38804d92652и f055f18ceeef785ab388764a4f79e8b6- одно и то же разбиение). При ef7c428d11771c0256e336a440147c1fположим e930922214eeec4092821007f895ee15(единственное разбиение не содержит слагаемых). Построить алгоритм вычисления b026f2805333f8d95dbd1a4196ac3bfdдля заданного 2d87b8a32ece94992b848596ff06ce88.

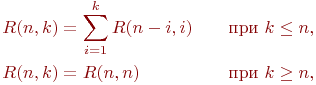
**Решение**. Можно доказать (это нетривиально) такую формулу для b026f2805333f8d95dbd1a4196ac3bfd:

bfe8395335cf7e8c1ef392e380c2c860

(знаки у пар членов чередуются, вычитаемые в одной паре равны 576f53f75cbf449636d4a469937a2685и 9dacd13e6bd70d4c264ccc793aec2ca1; сумма конечна - мы считаем, что bec81ddf26a9b1fdbcabb924c402a88fпри 789147c9e34861bbddfed6550e7ef291).

Однако и без ее использования можно придумать способ вычисления b026f2805333f8d95dbd1a4196ac3bfd, который существенно эффективнее перебора и подсчета всех разбиений.

Обозначим через 30a83c4dfae7c3f94734e397d7571cb4(для da42acab19ce35cd32c0df01b273027a, cf247a4230dfb846abd3a28537da5a7b) число разбиений 2d87b8a32ece94992b848596ff06ce88на целые положительные слагаемые, не превосходящие 5b9712f800c303699c96c43db144124e. (При этом e9833c31e83ca903f23d974ceefcfb78считаем равным 51226c368dab44d96ea9d6af652f63d8для всех 0493799d62c46c756a1dd34211b6cef7.) Очевидно, 4bd972d01d9e865f87677fbafadc7f5c. Все разбиения 2d87b8a32ece94992b848596ff06ce88на слагаемые, не превосходящие 5b9712f800c303699c96c43db144124e, разобьем на группы в зависимости от максимального слагаемого (обозначим его d0197c95725ed8778c167e033323ed81). Число 30a83c4dfae7c3f94734e397d7571cb4равно сумме (по всем d0197c95725ed8778c167e033323ed81от 51226c368dab44d96ea9d6af652f63d8до 5b9712f800c303699c96c43db144124e) количеств разбиений со слагаемыми не больше 5b9712f800c303699c96c43db144124eи максимальным слагаемым, равным d0197c95725ed8778c167e033323ed81. А разбиения 2d87b8a32ece94992b848596ff06ce88на слагаемые не более 5b9712f800c303699c96c43db144124eс первым слагаемым, равным d0197c95725ed8778c167e033323ed81, по существу представляют собой разбиения 965cd3f435efa37825ecd332d79c4c36на слагаемые, не превосходящие d0197c95725ed8778c167e033323ed81(при caa661e8ec2277cde8e39f312631018e). Так что



что позволяет заполнять таблицу значений функции 535ce254337d5dd7d8fbe544578bc8bf.

Счастливые билеты; предлагалась на Всесоюзной олимпиаде по программированию 1989 года. Последовательность из 58e798c45c2a315122657141c9dfec30цифр (каждая цифра от 7e691cd98b064ce17c0b6a2d0614dd5dдо 1a67a3a2ba9905623da1d4a9422d8dbf) называется счастливым билетом, если сумма первых 2d87b8a32ece94992b848596ff06ce88цифр равна сумме последних 2d87b8a32ece94992b848596ff06ce88цифр. Найти число счастливых последовательностей данной длины.

**Решение**. (Сообщено одним из участников олимпиады; к сожалению, не могу указать фамилию, так как работы проверялись зашифрованными.) Рассмотрим более общую задачу: найти число последовательностей, где разница между суммой первых 2d87b8a32ece94992b848596ff06ce88цифр и суммой последних 2d87b8a32ece94992b848596ff06ce88цифр равна 5b9712f800c303699c96c43db144124e( 1986df2cb666d90f8ef48e3082eb2da7). Пусть d42ea24cbc2ea70079bc9f131d19bc0b- число таких последовательностей.

Разобьем множество таких последовательностей на классы в зависимости от разницы между первой и последней цифрами. Если эта разница равна a2bbb443dffd9549210057fab0b62b12, то разница между суммами групп из оставшихся e693a9f00e8bf2ca253e994be40bb56dцифр равна dc792ab51ff13da70a36e0f7ef97240a. Учитывая, что пар цифр с разностью a2bbb443dffd9549210057fab0b62b12бывает 51e9486ae9510546bb448e05b4ffa388, получаем формулу

bc55d41e6920abb59f8142a7f4fe8742

(Некоторые слагаемые могут отсутствовать, так как dc792ab51ff13da70a36e0f7ef97240aможет быть слишком велико.)

В некоторых случаях ответ удается получить в виде явной формулы.

Доказать, что число Каталана (количество последовательностей длины 58e798c45c2a315122657141c9dfec30из 2d87b8a32ece94992b848596ff06ce88единиц и 2d87b8a32ece94992b848596ff06ce88минус единиц, в любом начальном отрезке которых не меньше единиц, чем минус единиц) равно 64c9893229e7f87b7c2f20d520f83c34.

**Указание**. Число Каталана есть число ломаных, идущих из d66a6a33bcb14777646342e27196c676в 553f4dd046fd4dfb2a955e2ecb4aad3aшагами 83c9739d4251f2fefd9d44f81fd25376и d266a959c994e9a970eff89404396dec, не опускающихся в нижнюю полуплоскость, т.е. разность числа всех ломаных (которое есть c3b49fd8fb9aebc76fd198fec327be2b) и числа ломаных, опускающихся в нижнюю полуплоскость. Последние можно описать также как ломаные, пересекающие прямую cf421769d81786aef5e1b626d62f750f. Отразив их кусок справа от самой правой точки пересечения относительно указанной прямой, мы установим взаимно однозначное соответствие между ними и ломаными из d66a6a33bcb14777646342e27196c676в 46550763610022096baf3cf765cf3f18. Остается проверить, что a92c41807e3fc85b4d4b3221f10581d9.

**Тема 3. Общие методы разработки алгоритмов (6 часов).**

План лекций.

1. Обход дерева и перебор с возвратом.

[Ферзи, не бьющие друг друга: обход дерева позиций](http://www.intuit.ru/department/se/prmalgs/3/#sect1#sect1). [Обход дерева в других задачах](http://www.intuit.ru/department/se/prmalgs/3/4.html#sect2).

1. Рекурсия.

[Примеры рекурсивных программ](http://www.intuit.ru/department/se/prmalgs/7/#sect1#sect1).  [Рекурсивная обработка деревьев](http://www.intuit.ru/department/se/prmalgs/7/2.html#sect2). [Порождение комбинаторных объектов, перебор](http://www.intuit.ru/department/se/prmalgs/7/3.html#sect3).  [Другие применения рекурсии](http://www.intuit.ru/department/se/prmalgs/7/4.html#sect4).

1. Построение итеративных алгоритмов по рекурсивным.

Динамическое программирование. Стек отложенных заданий.

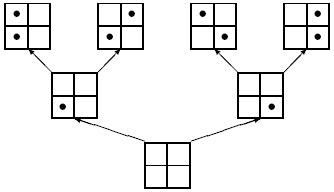
**Лекция 7. Обход дерева и перебор с возвратом**

### Ферзи, не бьющие друг друга: обход дерева позиций

Ранее мы рассматривали несколько задач одного и того же типа: «перечислить все элементы некоторого множества 8ec02840af196154a5480bc564e5008c». Схема решения была такова: на множестве 8ec02840af196154a5480bc564e5008cвводился порядок и описывалась процедура перехода от произвольного элемента множества 8ec02840af196154a5480bc564e5008cк следующему за ним (в этом порядке). Такую схему не всегда удается реализовать непосредственно, и в этой лекции мы рассмотрим другой полезный прием перечисления всех элементов некоторого множества. Его называют « поиск с возвратами «, « метод ветвей и границ «, « backtracking «. На наш взгляд, наиболее точное название этого метода – обход дерева.

Перечислить все способы расстановки 2d87b8a32ece94992b848596ff06ce88ферзей на шахматной доске 0d966f886c2e1e8b987680d6217251e6, при которых они не бьют друг друга.

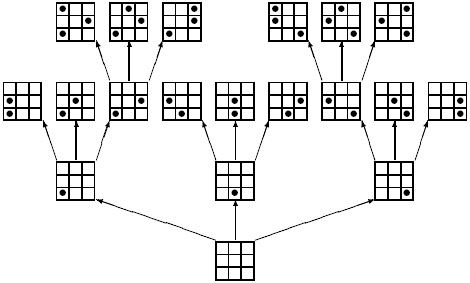
**Решение**. Очевидно, на каждой из 2d87b8a32ece94992b848596ff06ce88горизонталей должно стоять по ферзю. Будем называть k-позицией (для 7d3dc94e2a31b551333f380746606c54) произвольную расстановку 5b9712f800c303699c96c43db144124eферзей на 5b9712f800c303699c96c43db144124eнижних горизонталях (ферзи могут бить друг друга). Нарисуем «дерево позиций»: его корнем будет единственная 7e691cd98b064ce17c0b6a2d0614dd5d-позиция, а из каждой 5b9712f800c303699c96c43db144124e-позиции выходит 2d87b8a32ece94992b848596ff06ce88стрелок вверх в af7374c7487f499b475c9509c4b04c50-позиции. Эти 2d87b8a32ece94992b848596ff06ce88позиций отличаются положением ферзя на af7374c7487f499b475c9509c4b04c50-ой горизонтали. Будем считать, что расположение их на рисунке соответствует положению этого ферзя: левее та позиция, в которой ферзь расположен левее.



Дерево позиций для n=2

Среди позиций этого дерева нам надо отобрать те 2d87b8a32ece94992b848596ff06ce88-позиции, в которых ферзи не бьют друг друга. Программа будет «обходить дерево» и искать их. Чтобы не делать лишней работы, заметим вот что: если в какой-то 5b9712f800c303699c96c43db144124e-позиции ферзи бьют друг друга, то ставить дальнейших ферзей смысла нет. Поэтому, обнаружив это, мы будем прекращать построение дерева в этом направлении.

Точнее, назовем 5b9712f800c303699c96c43db144124e-позицию допустимой, если после удаления верхнего ферзя оставшиеся не бьют друг друга. Наша программа будет рассматривать только допустимые позиции.



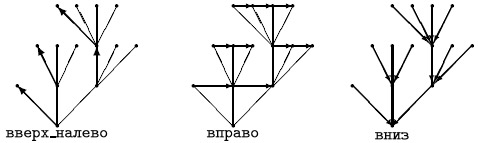
Дерево допустимых позиций для n=3

Разобьем задачу на две части: (1) обход произвольного дерева и (2) реализацию дерева допустимых позиций.

Сформулируем задачу обхода произвольного дерева. Будем считать, что у нас имеется Робот, который в каждый момент находится в одной из вершин дерева (вершины изображены на рисунке кружочками). Он умеет выполнять команды:

* вверх\_налево (идти по самой левой из выходящих вверх стрелок)
* вправо (перейти в соседнюю справа вершину)
* вниз (спуститься вниз на один уровень)

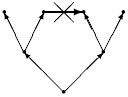
(На рисунках стрелками показано, какие перемещения соответствуют этим командам.)



Кроме того, в репертуар Робота входят проверки (соответствующие возможности выполнить каждую из команд):

* есть\_сверху ;
* есть\_справа ;
* есть\_снизу ;

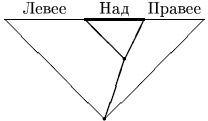
(последняя проверка истинна всюду, кроме корня). Обратите внимание, что команда вправо позволяет перейти лишь к «родному брату», но не к «двоюродному».



Так команда вправо не действует!

Будем считать, что у Робота есть команда **обработать** и что его задача – обработать все листья (вершины, из которых нет стрелок вверх, то есть где условие есть\_сверху ложно). Для нашей шахматной задачи команде обработать будет соответствовать проверка и печать позиции ферзей.

Доказательство правильности приводимой далее программы использует такие определения. Пусть фиксировано положение Робота в одной из вершин дерева. Тогда все листья дерева разбиваются на три категории: **над** Роботом, **левее** Робота и **правее** Робота. (Путь из корня в лист может проходить через вершину с Роботом, сворачивать влево, не доходя до нее и сворачивать вправо, не доходя до нее.) Через **(ОЛ)** обозначим условие «обработаны все листья левее Робота», а через **(ОЛН)** – условие « обработаны все листья левее и над Роботом».



Нам понадобится такая процедура:

procedure вверх\_до\_упора\_и\_обработать;

| {дано: (ОЛ), надо: (ОЛН)}

begin

| {инвариант: ОЛ}

| while есть\_сверху do begin

| | вверх\_налево;

| end

| {ОЛ, Робот в листе}

| обработать;

| {ОЛН}

end;

Основной алгоритм:

дано: Робот в корне, листья не обработаны

надо: Робот в корне, листья обработаны

{ОЛ}

вверх\_до\_упора\_и\_обработать;

{инвариант: ОЛН}

while есть\_снизу do begin

| if есть\_справа then begin {ОЛН, есть справа}

| | вправо;

| | {ОЛ}

| | вверх\_до\_упора\_и\_обработать;

| end else begin

| | {ОЛН, не есть\_справа, есть\_снизу}

| | вниз;

| end;

end;

{ОЛН, Робот в корне => все листья обработаны}

Осталось воспользоваться следующими свойствами команд Робота (в каждой строке в первой фигурной скобке записаны условия, в которых выполняется команда, во второй – утверждения о результате ее выполнения):

(1) { ОЛ, не есть\_сверху } обработать { ОЛН }

(2) { ОЛ, есть\_сверху } вверх\_налево {ОЛ}

(3) { есть\_справа ОЛН } вправо { ОЛ }

(4) {не есть\_справа }, есть\_снизу, ОЛН } вниз { ОЛН }

Доказать, что приведенная программа завершает работу (на любом конечном дереве).

**Решение**. Процедура вверх\_до\_упора\_и\_обработать } завершает работу (высота Робота не может увеличиваться бесконечно). Если программа работает бесконечно, то, поскольку листья не обрабатываются повторно, начиная с некоторого момента ни один лист не обрабатывается. А это возможно, только если Робот все время спускается вниз. Противоречие. (Об оценке числа действий см. далее.)

Доказать правильность следующей программы обхода дерева:

var state: (WL, WLU);

state := WL;

while есть\_снизу or (state <> WLU) do begin

| if (state = WL) and есть\_сверху then begin

| | вверх\_налево;

| end else if (state = WL) and not есть\_сверху then begin

| | обработать; state := WLU;

| end else if (state = WLU) and есть\_справа then begin

| | вправо; state := WL;

| end else begin {state = WLU, not есть\_справа, есть\_снизу}

| | вниз;

| end;

end;

**Решение**. Инвариант цикла:

a4b5fcae530609300e1b0f1d0562b804

Доказательство завершения работы: переход из состояния ОЛ в ОЛН возможен только при обработке вершины, поэтому если программа работает бесконечно, то с некоторого момента значение state не меняется, что невозможно.

Написать программу обхода дерева, использующую процедуру перехода в следующий лист (с выходным параметром, сообщающим, удалось ли это сделать или лист оказался последним).

Решить задачу об обходе дерева, если мы хотим, чтобы обрабатывались все вершины (не только листья).

**Решение**. Пусть f8d5aef0bc3bee6562c1285421844ab8- некоторая вершина. Тогда любая вершина 96f94aa4af7ba6a280977b83a2f3bbabотносится к одной из четырех категорий. Рассмотрим путь из корня в 96f94aa4af7ba6a280977b83a2f3bbab. Он может:

(а) быть частью пути из корня в f8d5aef0bc3bee6562c1285421844ab8( 0671e653801b817ce2898451b370846b);

(б) свернуть налево с пути в f8d5aef0bc3bee6562c1285421844ab8( 9524c963912e340ebbbeab19b7bdd1ac);

(в) пройти через f8d5aef0bc3bee6562c1285421844ab8( d7998e205628a671e9e707f2f00b5130);

(г) свернуть направо с пути в f8d5aef0bc3bee6562c1285421844ab8( bf8d70981315bb7dbc2db718f50a782e);

В частности, сама вершина f8d5aef0bc3bee6562c1285421844ab8относится к категории (в). Условия теперь будут такими:

(ОНЛ) обработаны все вершины ниже и левее;

(ОНЛН) обработаны все вершины ниже, левее и над.

Вот как будет выглядеть программа:

procedure вверх\_до\_упора\_и\_обработать;

| {дано: (ОНЛ), надо: (ОНЛН)}

begin

| {инвариант: ОНЛ}

| while есть\_сверху do begin

| | обработать;

| | вверх\_налево;

| end

| {ОНЛ, Робот в листе}

| обработать;

| {ОНЛН}

end;

Основной алгоритм:

дано: Робот в корне, ничего не обработано

надо: Робот в корне, все вершины обработаны

{ОНЛ}

вверх\_до\_упора\_и\_обработать;

{инвариант: ОНЛН}

while есть\_снизу do begin

| if есть\_справа then begin {ОНЛН, есть справа}

| | вправо;

| | {ОНЛ}

| | вверх\_до\_упора\_и\_обработать;

| end else begin

| | {ОЛН, не есть\_справа, есть\_снизу}

| | вниз;

| end;

end;

{ОНЛН, Робот в корне => все вершины обработаны}

.Приведенная только что программа обрабатывает вершину до того, как обработан любой из ее потомков. Как изменить программу, чтобы каждая вершина, не являющаяся листом, обрабатывалась дважды: один раз до, а другой раз после всех своих потомков? (Листья по-прежнему обрабатываются по разу.)

**Решение**. Под «обработано ниже и левее» будем понимать «ниже обработано по разу, слева обработано полностью (листья по разу, остальные по два)». Под «обработано ниже, левее и над» будем понимать « ниже обработано по разу, левее и над – полностью».

Программа будет такой:

procedure вверх\_до\_упора\_и\_обработать;

| {дано: (ОНЛ), надо: (ОНЛН)}

begin

| {инвариант: ОНЛ}

| while есть\_сверху do begin

| | обработать;

| | вверх\_налево;

| end

| {ОНЛ, Робот в листе}

| обработать;

| {ОНЛН}

end;

Основной алгоритм:

дано: Робот в корне, ничего не обработано

надо: Робот в корне, все вершины обработаны

{ОНЛ}

вверх\_до\_упора\_и\_обработать;

{инвариант: ОНЛН}

while есть\_снизу do begin

| if есть\_справа then begin {ОНЛН, есть справа}

| | вправо;

| | {ОНЛ}

| | вверх\_до\_упора\_и\_обработать;

| end else begin

| | {ОЛН, не есть\_справа, есть\_снизу}

| | вниз;

| | обработать;

| end;

end;

{ОНЛН, Робот в корне => все вершины обработаны полностью}

Доказать, что число операций в этой программе по порядку равно числу вершин дерева. (Как и в других программах, которые отличаются от этой лишь пропуском некоторых команд обработать.)

**Указание**. Примерно каждое второе действие при исполнении этой программы – обработка вершины, а каждая вершина обрабатывается максимум дважды.

Вернемся теперь к нашей задаче о ферзях (где из всех программ обработки дерева понадобится лишь первая, самая простая). Реализуем операции с деревом позиций. Позицию будем представлять с помощью переменной k: 0..n (число ферзей) и массива c: array[1..n] of 1..n ( c[i] – координаты ферзя на i –ой горизонтали; при 5e060cd55a135e75e6d19e2e454eada8значение c[i] роли не играет). Предполагается, что все позиции допустимы (если убрать верхнего ферзя, остальные не бьют друг друга).

Program queens;

| const n = ...;

| var

| k: 0..n;

| c: array [1..n] of 1..n;

|

| procedure begin\_work; {начать работу}

| begin

| | k := 0;

| end;

|

| function danger: oolean; {верхний ферзь под боем}

| | var b: boolean; i: integer;

| begin

| | if k <= 1 then begin

| | | danger := false;

| | end else begin

| | | b := false;

| | | i := 1;

| | | {b ⬄ верхний ферзь под боем ферзей с номерами < i}

| | | while I <> k do begin

| | | | b := b or (c[i]=c[k]) {вертикаль}

| | | | or (abs(c[i]-c[k]))=abs(i-k)); {диагональ}

| | | | i := i+1;

| | | end;

| | | danger := b;

| | end;

| end;

|

| function is\_up: oolean; {есть\_сверху}

| begin

| | is\_up := (k < n) and not danger;

| end;

|

| function is\_right: oolean; {есть\_справа}

| begin

| | is\_right := (k > 0) and (c[k] < n);

| end;

| {возможна ошибка: при k=0 не определено c[k]}

|

| function is\_down: oolean; {есть\_снизу}

| begin

| | is\_down := (k > 0);

| end;

|

| procedure up; {вверх\_налево}

| begin {k < n, not danger}

| | k := k + 1;

| | c [k] := 1;

| end;

|

| procedure right; {вправо}

| begin {k > 0, c[k] < n}

| | c [k] := c [k] + 1;

| end;

|

|

| procedure down; {вниз}

| begin {k > 0}

| | k := k – 1;

| end;

|

| procedure work; {обработать}

| | var i: integer;

| begin

| | if (k = n) and not danger then begin

| | | for I := 1 to n do begin

| | | | write (‘<’, I, ‘,’ , c[i], ‘> ‘);

| | | end;

| | | writeln;

| | end;

| end;

|

|

| procedure UW; {вверх\_до\_упора\_и\_обработать}

| begin

| | while is\_up do begin

| | | up;

| | end

| | work;

| end;

|

begin

| begin\_work;

| UW;

| while is\_down do begin

| | if is\_right then begin

| | | right;

| | | UW;

| | end else begin

| | | down;

| | end;

| end;

end.

Приведенная программа тратит довольно много времени на выполнение проверки есть\_сверху (проверка, находится ли верхний ферзь под боем, требует числа действий порядка n ). Изменить реализацию операций с деревом позиций так, чтобы все три проверки есть\_сверху/справа/снизу и соответствующие команды требовали бы количества действий, ограниченного не зависящей от n константой.

**Решение**. Для каждой вертикали, каждой восходящей и каждой нисходящей диагонали будем хранить булевское значение – сведения о том, находится ли на этой линии ферзь (верхний ферзь не учитывается). (Заметим, что в силу допустимости позиции на каждой из линий может быть не более одного ферзя.)

**Лекция 8. Рекурсия.**

### Примеры рекурсивных программ

При анализе рекурсивной программы возникает, как обычно, два вопроса:

* почему программа заканчивает работу?
* почему она работает правильно, если заканчивает работу?

Для (2) достаточно проверить, что (содержащая рекурсивный вызов) программа работает правильно, предположив, что вызываемая ею одноименная программа работает правильно. В самом деле, в этом случае в цепочке рекурсивно вызываемых программ все программы работают правильно (убеждаемся в этом, идя от конца цепочки к началу).

Чтобы доказать (1), обычно проверяют, что с каждым рекурсивным вызовом значение какого-то параметра уменьшается, и это не может продолжаться бесконечно.

Написать рекурсивную процедуру вычисления факториала целого положительного числа 2d87b8a32ece94992b848596ff06ce88(т. е. произведения ee9be15562b855e5dece484e67108959, обозначаемого 2d87b8a32ece94992b848596ff06ce88!).

**Решение**. Используем равенства a9aaa705ac1ce940e2fd51b7690672af, bd042e4119ce6c7c70604cd3b76872eb.

procedure factorial (n: integer; var fact: integer);

| {положить fact равным факториалу числа n}

begin

| if n=1 then begin

| | fact:=1;

| end else begin {n>1}

| | factorial (n-1, fact);

| | {fact = (n-1)!}

| | fact:= fact\*n;

| end;

end;

С использованием процедур-функций можно написать так:

function factorial (n: integer): integer;

begin

| if n=1 then begin

| | factorial:=1;

| end else begin {n>1}

| | factorial:= factorial (n-1)\*n;

| end;

end;

Обратите внимание на некоторую двойственность использования имени d79808be2cc2230b0278db8c4e8d1b64внутри описания функции: оно обозначает как переменную, так и вызываемую рекурсивно функцию. К счастью, в нашем случае они различаются по скобкам после имени, но если бы функция была без параметров, то дело было бы плохо. (Стандартная, но трудно находимая ошибка возникает, если автор программы на паскале полагает, что он использует значение переменной, а компилятор в этом месте видит рекурсивный вызов.)

Обычно факториал определяют и для нуля, считая, что c50952ff06fe75429776f53e1843671b. Изменить программы соответственно.

Игра "Ханойские башни" состоит в следующем. Есть три стержня. На первый из них надета пирамидка из 6705ff0251d7806d002b2a7c5ebf40acколец (большие кольца снизу, меньшие сверху). Требуется переместить кольца на другой стержень. Разрешается перекладывать кольца со стержня на стержень, но класть большее кольцо поверх меньшего нельзя. Составить программу, указывающую требуемые действия.

**Решение**. Напишем рекурсивную процедуру перемещения 04a28832344530e89d67bf5db2702a74верхних колец с ed3d65ea700b96161424bf54c16be51e-го стержня на a6f0c8d9a8181170eb78a6481bc60648-ый (остальные кольца предполагаются большими по размеру и лежат на стержнях без движения).

procedure move(i,m,n: integer);

| var s: integer;

begin

| if i = 1 then begin

| | writeln ('сделать ход ', m, '->', n);

| end else begin

| | s:=6-m-n; {s - третий стержень: сумма номеров равна 6}

| | move (i-1, m, s);

| | writeln ('сделать ход ', m, '->', n);

| | move (i-1, s, n);

| end;

end;

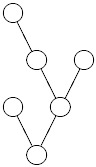
(Сначала переносится пирамидка из c456a41e3125ad4fc5dfdd36141599d8колец на третью палочку. После этого 6faeb6ba04719988390a716064200812кольцо освобождается, и его можно перенести куда следует. Остается положить на него пирамидку.)

Написать рекурсивную программу суммирования массива 567f1e224762c62381c5a0051e6c7182.

**Указание**. Рекурсивно определяемая функция должна иметь дополнительный параметр - число складываемых элементов.

### Рекурсивная обработка деревьев

Двоичным деревом называется картинка вроде такой:



Нижняя вершина называется корнем. Из каждой вершины могут идти две линии: влево вверх и вправо вверх. Вершины, куда они ведут, называются левым и правым сыновьями } исходной вершины. Вершина может иметь двух сыновей, а может иметь только одного сына (левого или правого). Она может и вовсе не иметь сыновей, и в этом случае называется листом.

Пусть f8d5aef0bc3bee6562c1285421844ab8- какая-то вершина двоичного дерева. Она сама вместе с сыновьями, внуками, правнуками и т.д. образует поддерево с корнем в f8d5aef0bc3bee6562c1285421844ab8- поддерево потомков x.

В следующих задачах мы предполагаем, что вершины дерева пронумерованы целыми положительными числами, причем номера всех вершин различны. Мы считаем, что номер корня хранится в переменной 90e9c68214b8fb98812010486482d77a. Мы считаем, что имеются два массива

l,r: array [1..N] of integer

и левый и правый сын вершины с номером 04a28832344530e89d67bf5db2702a74имеют соответственно номера 59fa447594e286261c01d0683121aa5cи e1e908c66cce91f5f5b6145dd1caf223. Если вершина с номером 04a28832344530e89d67bf5db2702a74не имеет левого (или правого) сына, то 59fa447594e286261c01d0683121aa5c(соответственно f4663ada0095cb0cc04ab4ec53215cf1равно 4b96b7549c98cb94ac966c6a7d5393b0. (По традиции при записи программ мы используем вместо нуля константу 9f0a8caa43bc55f4eda00488ba69a1ea, равную нулю.)

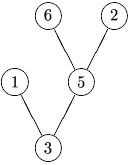
Здесь 472f2b8ab71800270cd65716c89aa231- достаточно большое натуральное число (номера всех вершин не превосходят 8c54245ef3fbf9fc147a6a9eb5f1a884. Отметим, что номер вершины никак не связан с ее положением в дереве и что не все числа от 7c6cb714d105ac96ac21b3fe8f32be89до 472f2b8ab71800270cd65716c89aa231обязаны быть номерами вершин (и, следовательно, часть данных в массивах 0c6acc241cbeae07d17b30112a631844и 3832aca10445268c13901059c71b9044- это мусор).

Пусть 4f6adc34d09b95472bf43348c04e2c45, 140e8d53937118a4e13a4226778b47f3, массивы 1a949fbfae8d9657811c85e1ddf35e47и 3832aca10445268c13901059c71b9044таковы:



Нарисовать соответствующее дерево.

**Ответ**.



Написать программу подсчета числа вершин в дереве.

**Решение**. Рассмотрим функцию dec0a34fa65e5d42223e5b770c6a927b, равную числу вершин в поддереве с корнем в вершине номер af71fd91e28cc481849405f1a34fd044. Считаем, что acff95c3b77a57a4c785d54205d81f70(полагая соответствующее поддерево пустым), и не заботимся о значениях 432d9644e92c8180fc8800013b2d3098для чисел 2e9f50f77a6a4ea207764d324ca97b1d, не являющихся номерами вершин. Рекурсивная программа для a6f0c8d9a8181170eb78a6481bc60648такова:

function n(x:integer):integer;

begin

| if x = nil then begin

| | n:= 0;

| end else begin

| | n:= n(l[x]) + n(r[x]) + 1;

| end;

end;

(Число вершин в поддереве над вершиной af71fd91e28cc481849405f1a34fd044равно сумме чисел вершин над ее сыновьями плюс она сама.) Глубина рекурсии конечна, так как с каждым шагом высота соответствующего поддерева уменьшается.

Написать программу подсчета числа листьев в дереве.

**Ответ**.

function n (x:integer):integer;

begin

| if x = nil then begin

| | n:= 0;

| end else if (l[x]=nil) and (r[x]=nil) then begin {лист}

| | n:= 1;

| end else begin

| | n:= n(l[x]) + n(r[x]);

| end;

end;

Написать программу подсчета высоты дерева (корень имеет высоту 7e691cd98b064ce17c0b6a2d0614dd5d, его сыновья - высоту 51226c368dab44d96ea9d6af652f63d8, внуки - c90fe44538b32061a458791169cb8983и т.п.; высота дерева - это максимум высот его вершин).

**Указание**. Рекурсивно определяется функция 88ffa22d4460c7f75eb62be9595ab7abвысота поддерева с корнем в af71fd91e28cc481849405f1a34fd044.

Написать программу, которая по заданному a6f0c8d9a8181170eb78a6481bc60648считает число всех вершин высоты a6f0c8d9a8181170eb78a6481bc60648(в заданном дереве).

Вместо подсчета количества вершин того или иного рода можно просить напечатать список этих вершин (в том или ином порядке).

Написать программу, которая печатает (по одному разу) все вершины дерева.

**Решение**. Процедура 8cc27b96de29909acb515df84c719a09печатает все вершины поддерева с корнем в af71fd91e28cc481849405f1a34fd044по одному разу; главная программа содержит вызов 2052c5f2912d47e05707e8699a4c83f6.

procedure print\_subtree (x:integer);

begin

| if x = nil then begin

| | {ничего не делать}

| end else begin

| | writeln (x);

| | print\_subtree (l[x]);

| | print\_subtree (r[x]);

| end;

end;

Данная программа печатает сначала корень поддерева, затем поддерево над левым сыном, а затем над правым. Три строки в bdae7c340a4fba0afcb13292df45412b-части могут быть переставлены 5e02c55102d3afa265247354c66158dcспособами, и каждый из этих способов дает свой порядок печати вершин.

**Топологическая сортировка**. Представим себе 2d87b8a32ece94992b848596ff06ce88чиновников, каждый из которых выдает справки определенного вида. Мы хотим получить все эти справки, соблюдая установленные ограничения: у каждого чиновника есть список справок, которые нужно собрать перед обращением к нему. Дело безнадежно, если схема зависимостей имеет цикл (справку 8ec02840af196154a5480bc564e5008cнельзя получить без 0b943ae98b3244d3f480e24ad6eb6868, 0b943ae98b3244d3f480e24ad6eb6868без 992dc219b8e42322eca3d04018b5dbc2без 67e77d783e275e9fb759d42deb1aca57и 67e77d783e275e9fb759d42deb1aca57без 8ec02840af196154a5480bc564e5008c). Предполагая, что такого цикла нет, требуется составить план, указывающий один из возможных порядков получения справок.

Изображая чиновников точками, а зависимости - стрелками, приходим к такой формулировке. Имеется 2d87b8a32ece94992b848596ff06ce88точек, пронумерованных от 51226c368dab44d96ea9d6af652f63d8до 2d87b8a32ece94992b848596ff06ce88. Из каждой точки ведет несколько (возможно, 7e691cd98b064ce17c0b6a2d0614dd5d) стрелок в другие точки. (Такая картинка называется ориентированным графом.) Циклов нет. Требуется расположить вершины графа (точки) в таком порядке, чтобы конец любой стрелки предшествовал ее началу. Эта задача называется топологической сортировкой.

Доказать, что это всегда возможно.

**Решение**. Из условия отсутствия циклов вытекает, что есть вершина, из которой вообще не выходит стрелок (иначе можно двигаться по стрелкам, пока не зациклимся). Ее будем считать первой. Выкидывая все стрелки, в нее ведущие, мы сводим задачу к графу с меньшим числом вершин и продолжаем рассуждение по индукции.

Предположим, что ориентированный граф без циклов хранится в такой форме: для каждого 04a28832344530e89d67bf5db2702a74от 7c6cb714d105ac96ac21b3fe8f32be89до a6f0c8d9a8181170eb78a6481bc60648в c665e3621b9fc4eeeeb212735a740de6хранится число выходящих из 04a28832344530e89d67bf5db2702a74стрелок, в 5072554f017d578223d4ac365c139bc9- номера вершин, куда эти стрелки ведут. Составить (рекурсивный) алгоритм, который производит топологическую сортировку не более чем за 4f68e2fee7bad98f8a1cfab238b4b3c0действий, где ed3d65ea700b96161424bf54c16be51e- число ребер графа (стрелок).

**Замечание**. Непосредственная реализация приведенного выше доказательства существования не дает требуемой оценки; ее приходится немного подправить.

**Решение**. Наша программа будет печатать номера вершин. В массиве

printed: array[1..n] of boolean

мы будем хранить сведения о том, какие вершины напечатаны (и корректировать их одновременно с печатью вершины). Будем говорить, что напечатанная последовательность вершин корректна, если никакая вершина не напечатана дважды и для любого номера 04a28832344530e89d67bf5db2702a74, входящего в эту последовательность, все вершины, в которые ведут стрелки из 04a28832344530e89d67bf5db2702a74, напечатаны, и притом до 04a28832344530e89d67bf5db2702a74.

procedure add (i: 1..n);

| {дано: напечатанное корректно;}

| {надо: напечатанное корректно и включает вершину i}

begin

| if printed [i] then begin {вершина i уже напечатана}

| | {ничего делать не надо}

| end else begin

| | {напечатанное корректно}

| | for j:=1 to num[i] do begin

| | | add(adr[i][j]);

| | end;

| | {напечатанное корректно, все вершины, в которые из

| | i ведут стрелки, уже напечатаны - так что можно

| | печатать i, не нарушая корректности}

| | if not printed[i] then begin

| | | writeln(i); printed [i]:= TRUE;

| | end;

| end;

end;

Основная программа:

for i:=1 to n do begin

| printed[i]:= FALSE;

end;

for i:=1 to n do begin

| add(i)

end;

К оценке времени работы мы вскоре вернемся.

В приведенной программе можно выбросить проверку, заменив

if not printed[i] then begin

| writeln(i); printed [i]:= TRUE;

end;

на

writeln(i); printed [i]:= TRUE;

Почему? Как изменится спецификация процедуры?

**Решение**. Спецификацию можно выбрать такой:

дано: напечатанное корректно

надо: напечатанное корректно и включает вершину i;

все вновь напечатанные вершины доступны из i.

Где использован тот факт, что граф не имеет циклов?

**Решение**. Мы опустили доказательство конечности глубины рекурсии. Для каждой вершины рассмотрим ее "глубину" - максимальную длину пути по стрелкам, из нее выходящего. Условие отсутствия циклов гарантирует, что эта величина конечна. Из вершины нулевой глубины стрелок не выходит. Глубина конца стрелки по крайней мере на 51226c368dab44d96ea9d6af652f63d8меньше, чем глубина начала. При работе процедуры 89a17514502fd02db92587dfc6a5fa85все рекурсивные вызовы 93b332b29656d002db1b0bab43467abeотносятся к вершинам меньшей глубины.

Вернемся к оценке времени работы. Сколько вызовов 89a17514502fd02db92587dfc6a5fa85возможно для какого-то фиксированного 04a28832344530e89d67bf5db2702a74? Прежде всего ясно, что первый из них печатает 04a28832344530e89d67bf5db2702a74, остальные сведутся к проверке того, что 04a28832344530e89d67bf5db2702a74уже напечатано. Ясно также, что вызовы 89a17514502fd02db92587dfc6a5fa85индуцируются "печатающими" (первыми) вызовами 93b332b29656d002db1b0bab43467abeдля тех 36039cbc401c1253925967070306ab76, из которых в 04a28832344530e89d67bf5db2702a74ведет ребро. Следовательно, число вызовов 89a17514502fd02db92587dfc6a5fa85равно числу входящих в 04a28832344530e89d67bf5db2702a74ребер (стрелок). При этом все вызовы, кроме первого, требуют 320c036ca13b0828a5cc285e4e1a6bd4операций, а первый требует времени, пропорционального числу исходящих из 04a28832344530e89d67bf5db2702a74стрелок. (Не считая времени, уходящего на выполнение 93b332b29656d002db1b0bab43467abeдля концов 36039cbc401c1253925967070306ab76выходящих ребер.) Отсюда видно, что общее время пропорционально числу ребер (плюс число вершин).

**Связная компонента графа**. Неориентированный граф - набор точек (вершин), некоторые из которых соединены линиями (ребрами). Неориентированный граф можно считать частным случаем ориентированного графа, в котором для каждой стрелки есть обратная.

Связной компонентой вершины 04a28832344530e89d67bf5db2702a74называется множество всех тех вершин, в которые можно попасть из 04a28832344530e89d67bf5db2702a74, идя по ребрам графа. (Поскольку граф неориентированный, отношение " 36039cbc401c1253925967070306ab76принадлежит связной компоненте 04a28832344530e89d67bf5db2702a74" является отношением эквивалентности.)

Дан неориентированный граф (для каждой вершины указано число соседей и массив номеров соседей, как в задаче о топологической сортировке). Составить алгоритм, который по заданному 04a28832344530e89d67bf5db2702a74печатает все вершины связной компоненты 04a28832344530e89d67bf5db2702a74по одному разу (и только их). Число действий не должно превосходить e8d5f2d083a181fa3c31820708a01731(общее число вершин и ребер в связной компоненте).

**Решение**. Программа в процессе работы будет "закрашивать" некоторые вершины графа. Незакрашенной частью графа будем называть то, что останется, если выбросить все закрашенные вершины и ведущие в них ребра. Процедура 89a17514502fd02db92587dfc6a5fa85закрашивает связную компоненту 04a28832344530e89d67bf5db2702a74в незакрашенной части графа (и не делает ничего, если вершина 04a28832344530e89d67bf5db2702a74уже закрашена).

procedure add (i:1..n);

begin

| if вершина i закрашена then begin

| | ничего делать не надо

| end else begin

| | закрасить i (напечатать и пометить как закрашенную)

| | для всех j, соседних с i

| | | add(j);

| | end;

| end;

end;

Докажем, что эта процедура действует правильно (в предположении, что рекурсивные вызовы работают правильно). В самом деле, ничего, кроме связной компоненты незакрашенного графа, она закрасить не может. Проверим, что вся она будет закрашена. Пусть 04ddc8b4942feb59d800213d8f4fa056- вершина, доступная из вершины 04a28832344530e89d67bf5db2702a74по пути fc18350f30742c70387ec3533bc6e6b1, проходящему только по незакрашенным вершинам. Будем рассматривать только пути, не возвращающиеся снова в 04a28832344530e89d67bf5db2702a74. Из всех таких путей выберем путь с наименьшим 36039cbc401c1253925967070306ab76(в порядке просмотра соседей в процедуре). Тогда при рассмотрении предыдущих соседей ни одна из вершин пути ef944bd789d550b35d22b19d63e0311dне будет закрашена (иначе 36039cbc401c1253925967070306ab76не было бы минимальным) и потому 04ddc8b4942feb59d800213d8f4fa056окажется в связной компоненте незакрашенного графа к моменту вызова 93b332b29656d002db1b0bab43467abe. Что и требовалось.

Чтобы установить конечность глубины рекурсии, заметим, что на каждом уровне рекурсии число незакрашенных вершин уменьшается хотя бы на 51226c368dab44d96ea9d6af652f63d8.

Оценим число действий. Каждая вершина закрашивается не более одного раза - при первым вызове 89a17514502fd02db92587dfc6a5fa85с данным 04a28832344530e89d67bf5db2702a74. Все последующие вызовы происходят при закрашивании соседей - количество таких вызовов не больше числа соседей - и сводятся к проверке того, что вершина 04a28832344530e89d67bf5db2702a74уже закрашена. Первый же вызов состоит в просмотре всех соседей и рекурсивных вызовах 93b332b29656d002db1b0bab43467abeдля всех них. Таким образом, общее число действий, связанных с вершиной 04a28832344530e89d67bf5db2702a74, не превосходит константы, умноженной на число ее соседей. Отсюда и вытекает требуемая оценка.

Решить ту же задачу для ориентированного графа (напечатать все вершины, доступные из данной по стрелкам; граф может содержать циклы).

**Ответ**. Годится по существу та же программа (строку "для всех соседей" надо заменить на "для всех вершин, куда ведут стрелки").

Следующий вариант задачи о связной компоненте имеет скорее теоретическое значение (и называется теоремой Сэвича).

Ориентированный граф имеет 030350cafe467a6a01d0a32119557ddcвершин (двоичные слова длины 2d87b8a32ece94992b848596ff06ce88) и задан в виде функции есть\_ребро, которая по двум вершинам f8d5aef0bc3bee6562c1285421844ab8и 96f94aa4af7ba6a280977b83a2f3bbabсообщает, есть ли в графе ребро из f8d5aef0bc3bee6562c1285421844ab8в 96f94aa4af7ba6a280977b83a2f3bbab. Составить алгоритм, который для данной пары вершин 3c7a01a45843b4d4f0110e85c492f2dfи 2b68aed7755296c7c996236d76ee37b1определяет, есть ли путь (по ребрам) из 3c7a01a45843b4d4f0110e85c492f2dfв 2b68aed7755296c7c996236d76ee37b1, используя память, ограниченную многочленом от 2d87b8a32ece94992b848596ff06ce88. (Время при этом может быть - и будет - очень большим.)

**Указание**. Использовать рекурсивную процедуру, выясняющую, существует ли путь из f8d5aef0bc3bee6562c1285421844ab8в 96f94aa4af7ba6a280977b83a2f3bbabдлины не более 52bb517d158f07167df848fd93a8288f(и вызывающую себя с уменьшенным на единицу значением 5b9712f800c303699c96c43db144124e).

**Быстрая сортировка Хоара**. В заключение приведем рекурсивный алгоритм сортировки массива, который на практике является одним из самых быстрых. Пусть дан массив 3974f965cd67a969719f0b9487c4707e. Рекурсивная процедура daa3e624bc83f1aacee34489c93b48fcсортирует участок массива с индексами из полуинтервала 303e4552e876311b5a8b9ca4adfb7721, то есть a9d9093863d641773f255b67099a68b8, не затрагивая остального массива.

procedure sort (l,r: integer);

begin

| if l = r then begin

| | ничего делать не надо - участок пуст

| end else begin

| | выбрать случайное число s в полуинтервале (l,r]

| | b := a[s]

| | переставить элементы сортируемого участка так, чтобы

| | сначала шли элементы, меньшие b - участок (l,ll]

| | затем элементы, равные b- участок (ll,rr]

| | затем элементы, большие b - участок (rr,r]

| | sort (l,ll);

| | sort (rr,r);

| end;

end;

Разделение элементов сортируемого участка на три категории (меньшие, равные, больше) рассматривалась в [лекции 1](http://www.intuit.ru/department/se/prmalgs/1/) (это можно сделать за время, пропорциональное длине участка). Конечность глубины рекурсии гарантируется тем, что длина сортируемого участка на каждом уровне рекурсии уменьшается хотя бы на 51226c368dab44d96ea9d6af652f63d8.

|  |
| --- |
| Доказать, что математическое ожидание числа операций при работе этого алгоритма не превосходит 67a3bb1c7f68575c83e388e88c6fb0e5, причем константа df6b1e3b4fc1cfa73dcf132ec7f1eaefне зависит от сортируемого массива.  **Указание**. Пусть 7453a183b2234accb2f3bdc5cfe3e6df- максимум математического ожидания числа операций для всех входов длины 2d87b8a32ece94992b848596ff06ce88. Из текста процедуры вытекает такое неравенство:  38cecf036080b4bccdea1d19db87a01b  Первый член соответствует распределению элементов на меньшие, равные и большие. Второй член - это среднее математическое ожидание для всех вариантов случайного выбора. (Строго говоря, поскольку среди элементов могут быть равные, в правой части вместо 38cebf50bc882659e45ee43856d7ce02и 98a30931c33e0d85e20adae372178161должны стоять максимумы 0620592f78262d98bf7e6556195f9478по всем f8d5aef0bc3bee6562c1285421844ab8, не превосходящим 5b9712f800c303699c96c43db144124eили 8c0b2ac22716eb425ca9486081e89f50, но это не мешает дальнейшим рассуждениям.) Далее индукцией по 2d87b8a32ece94992b848596ff06ce88нужно доказывать оценку d5989a7ca7384352b0d6e7454bb06799. При этом для вычисления среднего значения 2b9a1e028f9dc72868f5d4a7d9ca1a78по всем 023ea2e32b249f2e3442e405215d5963нужно вычислять 8fcf197d2bf44f5d5db099f1a7b44362по частям как 66911fec07ed200a6971b51e5286c4d1. При достаточно большом 1fa0597e1f4689d1d399da5dac0c1156член 4a06bf549096a518c84c43d1be9f2bf7в правой части перевешивается за счет интеграла 181aa3450bab37da7ca585d8ec506111, и индуктивный шаг проходит.  Имеется массив из 2d87b8a32ece94992b848596ff06ce88различных целых чисел 3974f965cd67a969719f0b9487c4707eи число 5b9712f800c303699c96c43db144124e. Требуется найти 5b9712f800c303699c96c43db144124e-ое по величине число в этом массиве, сделав не более 4a06bf549096a518c84c43d1be9f2bf7действий, где df6b1e3b4fc1cfa73dcf132ec7f1eaef- некоторая константа, не зависящая от 5b9712f800c303699c96c43db144124eи 2d87b8a32ece94992b848596ff06ce88.  **Замечание**. Сортировка позволяет очевидным образом сделать это за e2df7ac804e4bbcc75cbe7c865cfa496действий. Очевидный способ: найти наименьший элемент, затем найти второй, затем третий, df6322ff524ba5b143fad0b7e9461255-ый требует порядка 3ef766915d5d41c3d40ee9258054e973действий, то есть не годится (константа при 2d87b8a32ece94992b848596ff06ce88зависит от 5b9712f800c303699c96c43db144124e).  **Указание**. Изящный (хотя практически и бесполезный - константы слишком велики) способ сделать это таков:  А. Разобьем наш массив на e8061d93a1ffabc23554df1b3a567bd7групп, в каждой из которых по 825128405ebb10d31a552b6898bc4cfaэлементов. Каждую группу упорядочим.  Б. Рассмотрим средние элементы всех групп и перепишем их в массив из e8061d93a1ffabc23554df1b3a567bd7элементов. С помощью рекурсивного вызова найдем средний по величине элемент этого массива.  В. Сравним этот элемент со всеми элементами исходного массива: они разделятся на большие его и меньшие его (и один равный ему). Подсчитав количество тех и других, мы узнаем, в какой из этих частей должен находится искомый ( 5b9712f800c303699c96c43db144124e-ый) элемент и каков он там по порядку.  Г. Применим рекурсивно наш алгоритм к выбранной части.  Пусть 7453a183b2234accb2f3bdc5cfe3e6df- максимально возможное число действий, если этот способ применять к массивам из не более чем 2d87b8a32ece94992b848596ff06ce88элементов ( 5b9712f800c303699c96c43db144124eможет быть каким угодно). Имеем оценку:  cb55a3899b918aaa2924c9975455db09  Последнее слагаемое объясняется так: при разбиении на части каждая часть содержит не менее fdd0e8c2d2977e71927aa72e70822d16элементов. В самом деле, если f8d5aef0bc3bee6562c1285421844ab8- средний из средних, то примерно половина всех средних меньше f8d5aef0bc3bee6562c1285421844ab8. А если в пятерке средний элемент меньше f8d5aef0bc3bee6562c1285421844ab8, то еще два заведомо меньше f8d5aef0bc3bee6562c1285421844ab8. Тем самым по крайней мере 3b6ddf29d13b0fbab4e121baa299a716от половины элементов меньше f8d5aef0bc3bee6562c1285421844ab8.  Теперь по индукции можно доказать оценку 058a2cbc033ddf83d5def6877fba7d06(решающую роль при этом играет то обстоятельство, что 72105a6ddb96050d0e6c1d9d9ef7a554). |
| empty |

**Лекция 9. Построение итеративных алгоритмов по рекурсивным.**

Для универсальных языков программирования (каковым является паскаль) рекурсия не дает ничего нового: для всякой рекурсивной программы можно написать эквивалентную программу без рекурсии. Мы не будем доказывать этого, а продемонстрируем некоторые приемы, позволяющие избавиться от рекурсии в конкретных ситуациях.

Зачем это нужно? Ответ прагматика мог бы быть таким: во многих компьютерах (в том числе, к сожалению, и в современных, использующих так называемые RISC-процессоры), рекурсивные программы в несколько раз медленнее соответствующих нерекурсивных программ. Еще один возможный ответ: в некоторых языках программирования рекурсивные программы запрещены. А главное, при удалении рекурсии возникают изящные и поучительные конструкции.

### Таблица значений (динамическое программирование)

Следующая рекурсивная процедура вычисляет числа сочетаний (биномиальные коэффициенты). Написать эквивалентную нерекурсивную программу.

function C(n,k: integer):integer;

| {n >= 0; 0 <= k <=n}

begin

| if (k = 0) or (k = n) then begin

| | C:=1;

| end else begin {0<k<n}

| | C:= C(n-1,k-1)+C(n-1,k)

| end;

end;

**Замечание**. 8cf1852454cdaa5fcb81594dc54d93e5- число 5b9712f800c303699c96c43db144124e-элементных подмножеств 2d87b8a32ece94992b848596ff06ce88-элементного множества. Соотношение fef4152085e8342eca076ec173201a22получится, если мы фиксируем некоторый элемент 2d87b8a32ece94992b848596ff06ce88-элементного множества и отдельно подсчитаем 5b9712f800c303699c96c43db144124e-элементные подмножества, включающие и не включающие этот элемент. Таблица значений 8cf1852454cdaa5fcb81594dc54d93e5



называется треугольником Паскаля (того самого). В нем каждый элемент, кроме крайних единиц, равен сумме двух стоящих над ним.

**Решение**. Можно воспользоваться формулой

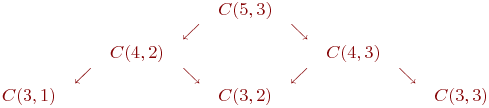
399b0eb3b038594ccb036711a803849a

Мы, однако, не будем этого делать, так как хотим продемонстрировать более общие приемы устранения рекурсии. Вместо этого составим таблицу значений функции 9475008a0492e5f3c604cd563ea9587f, заполняя ее для 7619a85a9f73e5a308d8b89629a76587, пока не дойдем до интересующего нас элемента.

Что можно сказать о времени работы рекурсивной и нерекурсивной версий в предыдущей задаче? Тот же вопрос о памяти.

**Решение**. Таблица занимает место порядка b573f76fd49ad902fe89155a5cc9733a, его можно сократить до 2d87b8a32ece94992b848596ff06ce88, если заметить, что для вычисления следующей строки треугольника Паскаля нужна только предыдущая. Время работы остается порядка b573f76fd49ad902fe89155a5cc9733a. Рекурсивная программа требует существенно большего времени: вызов C(n,k) сводится к двум вызовам для C(n-1,..), т.е. - к четырем вызовам для C(n-2,..) и так далее. Таким образом, время оказывается экспоненциальным (порядка 030350cafe467a6a01d0a32119557ddc). Используемая рекурсивной версией память пропорциональна 2d87b8a32ece94992b848596ff06ce88- умножаем глубину рекурсии ( 2d87b8a32ece94992b848596ff06ce88) на количество памяти, используемое одним экземпляром процедуры (константа).

Кардинальный выигрыш во времени при переходе от рекурсивной версии к нерекурсивной связан с тем, что в рекурсивном варианте одни и те же вычисления происходят много раз. Например, вызов C(5,3) в конечном счете порождает два вызова C(3,2):



Заполняя таблицу, мы каждую клетку заполняем только однажды - отсюда и экономия. Этот прием называется динамическим программированием, и применим в тех случаях, когда объем хранимой в таблице информации оказывается не слишком большим.

Порассуждать на ту же тему на примере рекурсивной и (простейшей) нерекурсивной программ для вычисления чисел Фибоначчи, заданных соотношением

1e2565777822a6af6720aedab6e3543d

Железная дорога с односторонним движением имеет 2d87b8a32ece94992b848596ff06ce88станций. Известны цены билетов от d0197c95725ed8778c167e033323ed81-ой станции до 947449428e406f59b8052aee3f91e2dd-ой (при ca96a4f4d9a84a56258aa4d68f17addc- в обратную сторону проезда нет). Найти минимальную стоимость проезда от начала до конца (с учетом возможной экономии за счет пересадок).

Задано конечное множество с бинарной операцией (вообще говоря, не коммутативной и даже не ассоциативной). Имеется 2d87b8a32ece94992b848596ff06ce88элементов 9b59fbac0a81a55c7e8036a2f85d2304этого множества и еще один элемент f8d5aef0bc3bee6562c1285421844ab8. Проверить, можно ли так расставить скобки в произведении df82bfeff6541b6f417cd430d4ef46a4, чтобы в результате получился f8d5aef0bc3bee6562c1285421844ab8. Число операций должно не превосходить ef8bf324e05201a82afeea072b9d43f9для некоторой константы df6b1e3b4fc1cfa73dcf132ec7f1eaef(зависящей от числа элементов в выбранном конечном множестве).

**Решение**. Заполняем таблицу, в которой для каждого участка 6424e5496012441eb81e24284c1bb04aнашего произведения хранится список всех возможных его значений (при разной расстановке скобок).

Имеется 2d87b8a32ece94992b848596ff06ce88положительных целых чисел eb35af40d56000cf46c4f9607181e958и число 6705ff0251d7806d002b2a7c5ebf40ac. Выяснить, можно ли получить 6705ff0251d7806d002b2a7c5ebf40ac, складывая некоторые из чисел f4444bd211194b2a3f383e9b25a128ca. Число действий должно быть порядка 7ad9ba2aa528bb3ca68d5d9b51fe16b0.

**Указание**. После d0197c95725ed8778c167e033323ed81шагов хранится множество тех чисел на отрезке f6b86a5d92be663ef42b257dae0912b1, которые представимы в виде суммы некоторых из 88404be1d60b73d6989890603ac860b7.

**Замечание**. Мы видели, что замена рекурсивной программы на заполнение таблицы значений иногда позволяет уменьшить число действий. Примерно того же эффекта можно добиться иначе: оставить программу рекурсивной, но в ходе вычислений запоминать уже вычисленные значения, а перед очередным вычислением проверять, нет ли уже готового значения.

### Стек отложенных заданий

Другой прием устранения рекурсии продемонстрируем на примере задачи о ханойских башнях.

Написать нерекурсивную программу для нахождения последовательности перемещений колец в задаче о ханойских башнях.

**Решение**. Вспомним рекурсивную программу, перекладывающую i верхних колец с m на n:

procedure move(i,m,n: integer);

| var s: integer;

begin

| if i = 1 then begin

| | writeln ('сделать ход ', m, '->', n);

| end else begin

| | s:=6-m-n; {s - третий стержень: сумма номеров равна 6}

| | move (i-1, m, s);

| | writeln ('сделать ход ', m, '->', n);

| | move (i-1, s, n);

| end;

end;

Видно, что задача "переложить i верхних дисков с m -го стержня на n -ый" сводится к трем задачам того же типа: двум задачам с i-1 дисками и к одной задаче с единственным диском. Занимаясь этими задачами, важно не позабыть, что еще осталось сделать.

Для этой цели заведем стек отложенных заданий, элементами которого будут тройки 2f33b685914f92dba79a48717599cd04. Каждая такая тройка интерпретируется как заказ "переложить i верхних дисков с m -го стержня на n -ый". Заказы упорядочены в соответствии с требуемым порядком их выполнения: самый срочный - вершина стека. Получаем такую программу:

procedure move(i,m,n: integer);

begin

| сделать стек заказов пустым

| положить в стек тройку <i,m,n>

| {инвариант: осталось выполнить заказы в стеке}

| while стек непуст do begin

| | удалить верхний элемент, переложив его в <j,p,q>

| | if j = 1 then begin

| | | writeln ('сделать ход', p, '->', q);

| | end else begin

| | | s:=6-p-q;

| | | {s - третий стержень: сумма номеров равна 6}

| | | положить в стек тройки <j-1,s,q>, <1,p,q>, <j-1,p,s>

| | end;

| end;

end;

(Заметим, что первой в стек кладется тройка, которую надо выполнять последней.) Стек троек может быть реализован как три отдельных стека. (Кроме того, в паскале есть специальный тип, называемый "запись" ( record ), который может быть применен.)

Для задачи о ханойских башнях есть и другие нерекурсивные алгоритмы. Вот один из них: простаивающим стержнем (не тем, с которого переносят, и не тем, на который переносят) должны быть все стержни по очереди. Другое правило: поочередно перемещать наименьшее кольцо и не наименьшее кольцо, причем наименьшее - по кругу.

Написать нерекурсивный вариант программы быстрой сортировки. Как обойтись стеком, глубина которого ограничена 17a1227901197c942bf4a2ee1b334678, где 2d87b8a32ece94992b848596ff06ce88- число сортируемых элементов?

**Решение**. В стек кладутся пары 33a57ea872d3cbca277ae4b7870b97a8, интерпретируемые как отложенные задания на сортировку соответствующих участков массива. Все эти заказы не пересекаются, поэтому размер стека не может превысить 2d87b8a32ece94992b848596ff06ce88. Чтобы ограничиться стеком логарифмической глубины, будем придерживаться такого правила: глубже в стек помещать больший из возникающих двух заказов. Пусть 7db7b4cc7c6221592d27f5ad3a9d1e7a- максимальная глубина стека, которая может встретиться при сортировке массива из не более чем 2d87b8a32ece94992b848596ff06ce88элементов таким способом. Оценим 7db7b4cc7c6221592d27f5ad3a9d1e7aсверху таким способом: после разбиения массива на два участка мы сначала сортируем более короткий (храня в стеке более длинный про запас), при этом глубина стека не больше 84dc4060b7cc279e101790a6765fe56b, затем сортируем более длинный, так что

b6b0ecfb1ac8cd33f75022bb69ec6700

откуда очевидной индукцией получаем 43d6326a87ea17d495e74de14c2e4ef7.

### Более сложные случаи рекурсии

Пусть функция 7b769f8f37e64ceb2b9df048d4161c2cс натуральными аргументами и значениями определена рекурсивно условиями

0af38271858bd8a8c66009a5932d135b

где 64573212f7cab8a2ee7b77ed2c8e2f91- некоторое число, а cab3e53428032dad6fe89328f6c2f8b7и 8c0b2ac22716eb425ca9486081e89f50- известные функции. Другими словами, значение функции 7b769f8f37e64ceb2b9df048d4161c2cв точке f8d5aef0bc3bee6562c1285421844ab8выражается через значение 7b769f8f37e64ceb2b9df048d4161c2cв точке 83d7ef9e9acdd59f4fdeebfa11f811ea. При этом предполагается, что для любого f8d5aef0bc3bee6562c1285421844ab8в последовательности

7c80b12d6756e91d21bc42f8d38eae11

рано или поздно встретится 7e691cd98b064ce17c0b6a2d0614dd5d.

Если дополнительно известно, что 20aad0b7eb35f4aa255c21006b2fa159для всех f8d5aef0bc3bee6562c1285421844ab8, то вычисление 7b769f8f37e64ceb2b9df048d4161c2cне представляет труда: вычисляем последовательно 25e585ee5c3852360041e87ed4192554

Написать нерекурсивную программу вычисления 7b769f8f37e64ceb2b9df048d4161c2cдля общего случая.

**Решение**. Для вычисления 46488a6f3918ac5b32b4ab0ae9ae6a43вычисляем последовательность

6c808f4167b5c9251562f8034789a6e2

до появления нуля и запоминаем ее, а затем вычисляем значения 7b769f8f37e64ceb2b9df048d4161c2cв точках этой последовательности, идя справа налево.

Еще более сложный случай из следующей задачи вряд ли встретится на практике (а если и встретится, то проще рекурсию не устранять, а оставить). Но тем не менее: пусть функция 7b769f8f37e64ceb2b9df048d4161c2cс натуральными аргументами и значениями определяется соотношениями

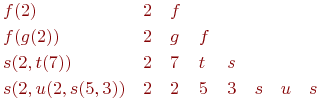
0003e9b47d8493fdb9a5983de1a47439

где 64573212f7cab8a2ee7b77ed2c8e2f91- некоторое число, а 8c0b2ac22716eb425ca9486081e89f50, 6b92a8949607d7391ec6c9fcaf3db645и cab3e53428032dad6fe89328f6c2f8b7- известные функции. Предполагается, что если взять произвольное число и начать применять к нему функции 8c0b2ac22716eb425ca9486081e89f50и 6b92a8949607d7391ec6c9fcaf3db645в произвольном порядке, то рано или поздно получится 7e691cd98b064ce17c0b6a2d0614dd5d.

Написать нерекурсивную программу вычисления 7b769f8f37e64ceb2b9df048d4161c2c.

**Решение**. Можно было бы сначала построить дерево, у которого в корне находится f8d5aef0bc3bee6562c1285421844ab8, а в сыновьях вершины d0197c95725ed8778c167e033323ed81стоят de2c6cbc6562c60fe105aa541aca3f8fи d2d48e6d9f69ac59e23ac7596079d7f4- если только d0197c95725ed8778c167e033323ed81не равно нулю. Затем вычислять значения функции, идя от листьев к корню. Однако есть и другой способ.

Обратной польской записью (или постфиксной записью ) выражения называют запись, где знак функции стоит после всех ее аргументов, а скобки не используются. Вот несколько примеров:



Постфиксная запись выражения позволяет удобно вычислять его с помощью 00f46a5a8d76ae744919d7c911ca9c35. Этот калькулятор имеет стек, который мы будем представлять себе расположенным горизонтально (числа вынимаются и кладутся справа), и клавиши - числовые и функциональные. При нажатии на клавишу с числом это число кладется в стек. При нажатии на функциональную клавишу соответствующая функция применяется к нескольким аргументам у вершины стека. Например, если в стеке были числа

adb8b0d69d32ebee24d5b2746f247d58

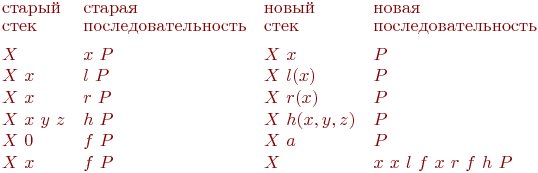
и нажата функциональная клавиша 39f2466f09ac6f1bf559b78d92020d29, соответствующая функции от двух аргументов, то в стеке окажутся числа

46209d533a6585b429fbb693c3fb0212

Перейдем теперь к нашей задаче. В процессе вычисления значения функции 7b769f8f37e64ceb2b9df048d4161c2cмы будем работать со стеком чисел, а также с последовательностью чисел и символов f, l, r, h, которую мы будем интерпретировать как последовательность нажатий клавиш на стековом калькуляторе. Инвариант такой:

8b53e5ea2978ba0451a68884282eb965

Пусть нам требуется вычислить значение 46488a6f3918ac5b32b4ab0ae9ae6a43. Тогда вначале мы помещаем в стек число f8d5aef0bc3bee6562c1285421844ab8, а последовательность содержит единственный символ f. (При этом инвариант соблюдается.) Далее с последовательностью и стеком выполняются такие преобразования:



Здесь f8d5aef0bc3bee6562c1285421844ab8, 96f94aa4af7ba6a280977b83a2f3bbab, b73e0b27b7db780bfb9cd147043c46f1- числа, 0afc66567c0c6d6c0ae30f599f4a5a38- последовательность чисел, 96605b0d144b65ef1d7a24ed64c0e451- последовательность чисел и символов f, l, r, h. В последней строке предполагается, что 40f60b72418556c8b7aca2454e654b7e. Эта строка соответствует равенству

00e8f2dbde4c09d7ee1cca3b186a396a

Преобразования выполняются, пока последовательность не станет пуста. В этот момент в стеке окажется единственное число, которое и будет ответом.

**БИБЛИОГРАФИЧЕСКИЙ СПИСОК**

1. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. М.: Вильямс, 2007.
2. Ахо А., Хопкрофт Д., Ульман Д. Структуры данных и алгоритмы. М.: Вильямс, 2007.
3. Окулов С.М. Программирование в алгоритмах. М.: Бином, 2007.
4. Кнут Д. Искусство программирования. Т. 4 Вып.2. Генерация всех кортежей и перестановок. М.: Вильямс, 2008.
5. Кнут Д. Искусство программирования. Т. 4 Вып.2. Генерация всех сочетаний и разбиений. М.: Вильямс, 2008.
6. Кнут Д. Искусство программирования. Т. 4 Вып.2. Генерация всех деревьев. История комбинаторной генерации. М.: Вильямс, 2008.
7. Седжвик Р. Фундаментальные алгоритмы на С++. М.: Вильямс, 2011.
8. Ху Т.Ч., Шинг М.Т. Комбинаторные алгоритмы Нижний Новгород: Изд-во Нижегородского госуниверситета им. Н.И. Лобачевского, 2004.
9. Новиков Ф.А. Дискретная математика для программистов. СПб.: Питер, 2012.

**СОДЕРЖАНИЕ**

Введение 4

**Тема 1. АЛГОРИТМЫ НА ГРАФАХ.** 6

Лекция 1. Начальные понятия теории графов. 7

Лекция 2. Поиск в глубину и ширину. 21

Лекция 3. Эйлеровы и гамильтоновы циклы. 35

**Тема 2. АЛГОРИТМЫ КОМБИНАТОРНОГО ПЕРЕБОРА**. 48

Лекция 4. Базовые комбинаторные объекты. 49

Лекция 5. Коды Грея. 55

Лекция 6. Применение методов комбинаторного перебора. 61

**Тема 3. ОБЩИЕ МЕТОДЫ РАЗРАБОТКИ АЛГОРИТМОВ**. 66

Лекция 7. Обход дерева и перебор с возвратом. 67

Лекция 8. Рекурсия. 78

Лекция 9. Построение итеративных алгоритмов по рекурсивным. 90

**БИБЛИОГРАФИЧЕСКИЙ СПИСОК** 99

**Шутов Антон Владимирович**

**Медведев Юрий Алексеевич**

СТРУКТУРЫ И АЛГОРИТМЫ КОМПЬЮТЕРНОЙ

ОБРАБОТКИ ДАННЫХ

Курс лекций

Издается в авторской редакции

|  |  |
| --- | --- |
| Подписано в печать 20.12.2013  Усл. п. л. – 7,25  Заказ 07 - 12 | Формат 84 x 108 1/32  Уч. –изд. л. – 7,45  Тираж 50 экз. |

Отпечатано в отделе оперативной полиграфии ВГГУ

600014, г. Владимир, ул. Университетская, 2, тел. 33-87-40