

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 3. Практикум

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 – 5 я7

Ш 97

**Шутов А. В., Медведев Ю. А.**

Структуры и алгоритмы компьютерной обработки данных. Часть 3 (Практикум). – Владимир: ВлГУ, 2013. – 87 с.

 Учебное пособие адресовано студентам, обучающимся по направлению 010500 «Математическое обеспечение и администрирование информационных систем». Содержит теоретический материал, необходимый для проведения практических занятий.

 Пособие включает 18 практических занятий, сгруппированных по четырем темам: алгоритмы на графах, алгоритмы комбинаторного перебора, общие методы разработки алгоритомв и теория сложности алгоритмов. Материал может быть использован студентами физико-математических факультетов.

**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**

 Одним из важнейших аспектов подготовки будущего специалиста в области компьютерных технологий является знакомство с базовыми алгоритмами информатики, а также со структурами данных, позволяющими реализовывать эти алгоритмы. Среди этих алгоритмов особое место занимают алгоритмы решения задач, связанных с графами, а также методы организации перебора. Не менее важным является знание общих методов и приемов создания алгоритмов, а также понимание общих теоретических сведений о сложности алгоритмов, позволяющих сравнивать алгоритмы и оценивать их эффективность.

 В пособии приведены материалы для проведения 18 практических занятий, сгруппированных по четырем темам: алгоритмы на графах, алгоритмы комбинаторного перебора, общие методы разработки алгоритмов и теория сложности алгоритмов. Каждое практическое занятие снабжено списком рекомендуемой литературы с указанием конкретных страниц. Материалы по первым трем темам содержат большое число практических заданий, которые могут быть использованы как во время занятий, так и для самостоятельной работы студентов. Все задания снабжены подробными решениями. В последней теме изложены теоретические сведения, связанные с так называемой *NP*-проблемой, считающейся в настоящее время главной проблемой теоретической информатики.

 Учебное пособие предназначено для проведения практических занятий по дисциплине «Структуры и алгоритмы компьютерной обработки данных» для студентов вузов, обучающихся по направлению 010500 «Математическое обеспечение и администрирование информационных систем». Отдельные работы из данного пособия также могут быть использованы при изучении дисциплин «Программирование», «Теоретические основы информатики» студентами вузов, обучающимися на физико-математических факультетах по направлению «Педагогическое образование», а также в школах при проведении факультативов по информатике и при подготовке учащихся к олимпиадам по программированию.

**Тема 1. АЛГОРИТМЫ НА ГРАФАХ (6 часов)**

**Цель**: Изучение базовых алгоритмов на графах.

**Задачи**:

– Закрепление материала лекций.

– Изучение ряда дополнительных алгоритмов на графах (алгоритмы поиска кратчайших путей, алгоритмы нахождения потоков в сетях и т.п.).

– Знакомство с существующей литературой, посвященной алгоритмам на графах.

– Выработка умений по практическому созданию алгоритмов на графах.

– Выработка умений по решению стандартных задач, связанных с графами.

– Выработка умений по сравнительному анализу вычислительной сложности алгоритмов на графах.

– Подготовка предварительных версий реализаций алгоритмов на графах для последующего использования в процессе лабораторных занятий.

– Текущий контроль успеваемости и выполнения плана самостоятельной работы.

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1. *Кормен*, *Т*. Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. - М.: Вильямс, 2007. 1296 с.
2. *Окулов*, *С.М.* Программирование в алгоритмах / С.М. Окулов. - М.: Бином, 2007. 341с.
3. *Седжвик, Р*. Фундаментальные алгоритмы на С++ / Р. Седжвик. - М.: Вильямс, 2011. 1056 с.
4. *Новиков, Ф.А.* Дискретная математика для программистов / Ф.А. Новиков. - СПб.: Питер, 2012. 304 с.

**Рассматриваемые темы с указанием литературы**

1. **Понятие графа**.

[1] с. 607-609; [2] с. 158; [3] с. 18-38; [4] с. 189-200.

1. **Способы задания графов**.

[1] с. 609-613; [2] с. 159; [3] с. 38-48; [4] с. 201-203.

1. **Поиск в глубину**.

[1] с. 622-632; [2] с. 159-161; [3] с. 99-131; [4] с. 203-206.

1. **Поиск в ширину**.

[1] с. 613-621; [2] с. 161-162; [3] с. 132-141; [4] с. 203-206.

1. **Эйлеровы и гамильтоновы циклы**.

[2] с. 76-180 [4] с. 263-268

1. **Алгоритмы поиска кратчайших путей**.

[1] с. 663-671, 694-706; [2] с. 180-182; [3] с. 279-294; [4] с. 228-229.

1. **Алгоритм Беллмана-Форда**.

[1] с. 672-677

1. **Алгоритм Дейкстры**.

[1] с. 680-686; [2] с. 182-183; [3] с. 294-304; [4] с. 230-232.

1. **Алгоритм Флойда-Варшалла**.

[1] с. 718-726; [2] с. 186; [3] с. 304-311; [4] с. 229-230.

1. **Минимальные остовные деревья**.

[1] с. 644-651 [2] с. 162-165; [3] с. 231-250; [4] с. 255-257.

1. **Алгоритмы Крускала и Прима**.

[1] с. 651-662 [2] с. 165-170; [3] с. 250-266; [4] с. 257-259.

1. **Потоки в сетях**.

[1] с. 734-742 [2] с. 208-210; [3] с. 359-376; [4] с. 220-222.

1. **Алгоритм Форда-Фалкерсона**.

[1] с. 742-751 [2] с. 210-212; [4] с. 222-224.

1. **Алгоритм Эдмондса-Карпа**.

[1] с. 752-755.

Педагогические технологии

– фронтальный опрос,

– индивидуальные доклады (с презентациями),

– рассмотрение проблемных ситуаций,

– интерактивное написание компьютерных программ,

– индивидуальные и групповые задания,

– совместное решение задач.

Вопросы и задачи

*Пример 1.1.*

На рис. 1.1 изображен *неориентированный* граф *G =* ( *X,*  *A* ) .

*X*= { *x*1 , *x*2 , *x*3 , *x*4 },

*A =* { *a*1*=* ( *x*1 *, x*2 ) *, a*2 *=* ( *x*2 *, x*3 ) *, a*3 *=* ( *x*1 *, x*3 ) *, a*4 *=* ( *x*3 *, x*4 ) }*.*



Рис. 1.1

*Пример 1.2.*

На рис. 1.2 изображен *ориентированный* граф *G* = ( *X*, *A* ) .

*X* = { *x*1 , *x*2 , *x*3 , *x*4 },

*A*= { *a*1= ( *x*1,*x*2) ,*a*2= ( *x*1,*x*3) ,*a*3= ( *x*3,*x*4) ,*a*4= ( *x*3,*x*2) }.



Рис. 1.2

*Пример 1.3.*

На рис. 1.3 изображен *ориентированный* граф *G* = ( *X*, *A* ) .

*X* = { *x*1, *x*2,*x*3, *x*4}***,***

*A*= .

**

Ри*c*. 1.3

*Пример 1.4.*

В графе, изображенном на рис. 1.1, концами ребра *a*1являются вер­шины *x*1, *x*2; вершина *x*2инцидентна ребрам *a*1, *a*2; степень вершины *x*3равна3*;* вершины *x*1и *x*3смежные; ребра *a*1и *a*2смежные; вершина *x*4висячая. В ориентированном графе, изображен­ном на рис. 1.2, началом дуги *a*1является вершина *x*1, а ее концом - вершина *x*2;вершина *x*1инцидентна дугам *a*1и *a*2;

*G* ( *x*1 ) *=* { *x*2 *, x*3 }*, G* ( *x*2 ) *=* ∅, *G -*1( *x*3 ) *=* { *x*1 }*, G -*1( *x*1) *=* ∅.

*Пример 1.5.*

Матрица смежности графа , изображенного на рис. 1.1, имеет вид:

  *A* =  .

*Пример 1.6.*

Матрица смежности ориентированного графа , изображенного на рис. 1.2, имеет вид:

  *A* =  .

*Пример 1.7.*

Матрица инцидентности графа , изображенного на рис. 1.1, имеет вид:

 *B =*  .

*Пример 1.8.*

Матрица инцидентности ориентированного графа , изображенного на рис. 1.2 , имеет вид:

 *B =*   *.*

*Пример 1.9*

Графы , изображенные на рис. 1.4 , являются изоморфными.



Рис. 1.4

*Пример 1.10.*

В изображенном на рис. 1.5 графе рассмотрим два маршрута из вершины *x*1в вершину *x*4 : *M* 1= ( *a*1, *a*2, *a*4 ) и *M* 2= ( *a*1, *a*2, *a*5, *a*6 ). Длина маршрута *M* 1 равна 3, а длина маршрута  *M* 2 равна 4.



Рис.1.5

*Пример 1.11.*

В приведенном на рис 1.6 графе выделим следующие маршруты:

( *a*1 , *a*3 , *a*4) – простая элементарная цепь длины 3, т.к. все ребра и вершины попарно различны ;

( *a*2 , *a*4 , *a*3) – простой элементарный цикл, т.к. это замкнутый маршрут, у которого все ребра и верши­ны, кроме первой и последней, различны ;

( *a*1 , *a*2 , *a*4 , *a*3 )– цепь, которая является простой, но не элементарной, т.к. все ребра различны, но вершина *x*2 встречается дважды ;

( *a*1 , *a*2 , *a*2 ) – маршрут длины 3, не являющийся ни простой, ни элементарной цепью, т.к. ребро *a*2 и вершина *x*2 встречаются дважды.



Рис.1.6

*Пример 1.12.*

В приведенном на рис 1.7 графе выделим следующие пути:

( *x*1 , *x*2 , *x*3 , *x*4 ) – простой элементарный путь, т.к. каждая вершина и каждая дуга используются не более одного раза ;

( *x*2 , *x*5 , *x*6 , *x*7 , *x*2 ) – простой элементарный контур, т.к. это замкнутый путь, в котором все дуги и вершины, кроме первой и последней, различны.



Рис. 1.7

*Пример 1.3.*

У неориентированного графа, изображенного на рис. 1.8 две компоненты связности. Первая компонента связности включает вершины *x*1 , *x*2 , *x*4 , *x*5 ,а вторая состоит из одной вершины *x*1.



Рис.1.8

Матрица связности этого графа имеет вид:

 *S* =   *.*

Мы видим , что 1-ая , 2-ая , 4-ая и 5-ая строки матрицы *S*  одинаковы.

*Пример 1.14.*

У ориентированного графа , изображенного на рис. 1.9 две компоненты сильной связности. Первая компонента связности включает вершины *x*1 , *x*2 , *x*3 , *x*5 ,а вторая состоит из одной вершины *x*4 . Действительно, для любой пары вершин из множества { *x*1 , *x*2 , *x*3 , *x*5 }существует хотя бы один путь, соединяющий эти вершины. Например, путь ( *x* 1, *x* 2, *x* 5, *x* 3, *x* 1 ) соединяет все эти вершины. Из вершины *x*4 нет пути ни в одну вершину графа .



Рис. 1.9

Матрица сильной связности этого графа имеет вид:

 *S* =  .

Мы видим , что 1-я , 2-я , 3-я и 5-я строки матрицы *S*  одинаковы.

*Пример 1.15.*

С помощью алгоритма *1.1* найдем минимальный путь из верши­ны *х* 1 в вершину *х* 3 в графе , изображенном на рис. 1.10.



### Рис. 1.10

Рассмотрим подробно работу алгоритма Форда – Беллмана для этого примера. Значения индексов * i* ( *k* ) будем заносить в таблицу индексов ( табл. 1.1).

*Шаг 1.* Введем число вершин графа *n* = 5. Матрица весов этого графа имеет вид:

 *C* = .

*Шаг 2.* Положим *k* = 0, ** 1 (0) = 0, ** 2 (0) = ** 3 (0) = ** 4 (0) = ** 5 (0) = ∞. Эти значения занесем в первый столбец табл. 1.1.

*Шаг 3.*

*k =* 1.

** 1 (1) = 0.

* i* (1) = **{* j* (0) + *c j i*}.

** 2 (1) = *min*{** 1 (0) + *c* 1 2; ** 2 (0) + *c* 2 2 ; ** 3 (0) + *c* 3 2 ; ** 4 (0) + *c* 4 2 ; ** 5 (0) + *c* 5 2 ;} = *min*{0 + 1; ∞ + ∞; ∞ + ∞; ∞ + ∞; ∞ + ∞} = 1.

** 3 (1) = *min*{** 1 (0) + *c* 1 3 ; ** 2 (0) + *c* 2 3 ; ** 3 (0) + *c* 3 3 ; ** 4 (0) + *c* 4 3 ; ** 5 (0) + *c* 5 3 ;} = *min*{0 + ∞ ; ∞ + 8; ∞ + ∞; ∞ + 2; ∞ + ∞} = ∞ .

** 4 (1) = *min*{** 1 (0) + *c* 1 4 ; ** 2 (0) + *c* 2 4 ; ** 3 (0) + *c* 3 4 ; ** 4 (0) + *c* 4 4 ; ** 5 (0) + *c* 5 4 ;} = *min*{0 + ∞ ; ∞ + 7; ∞ + 1; ∞ + ∞; ∞ + 4} = ∞ .

** 5 (1) = *min*{** 1 (0) + *c* 1 5 ; ** 2 (0) + *c* 2 5 ; ** 3 (0) + *c* 3 5 ; ** 4 (0) + *c* 4 5 ; ** 5 (0) + *c* 5 5 ;} = *min*{0 + 3; ∞ + 1; ∞ – 5; ∞ + ∞; ∞ + ∞} = 1.

Полученные значения * i* (1) занесем во второй столбец табл. 1.1. Убеждаемся , что второй столбец , начиная со второго элемента , совпадает с первой строкой матрицы весов, что легко объясняется смыслом величин *i*(1), которые равны длине минимального пути из первой вершины в *i*-ую, содержащего не более одной дуги.

*k =* 2.

** 1 (2) = 0.

Равенство (1.1) для *k* = 2 имеет вид:

* i* (2) = **{* j* (1) + *c j i* }.

** 2 (2) = *min*{0 + 1; 1 + ∞; ∞ + ∞; ∞ + ∞; 3 + ∞} = 1.

** 3 (2) = *min*{0 + ∞ ; 1 + 8; ∞ + ∞; ∞ + 2; 3 + ∞} = 9 .

** 4 (2) = *min*{0 + ∞ ; 1 + 7; ∞ + 1; ∞ + ∞; 3 + 4} = 7 .

** 5 (2) = *min*{0 + 3; 1 + 1; ∞ – 5; ∞ + ∞; 3 + ∞} = 2.

Полученные значения * i* (2) занесем в третий столбец табл. 1.1. Величины * i* (2) равны длине минимального пути из первой вершины в *i*-ую, содержащего не более двух дуг.

*k =* 3.

** 1 (3) = 0.

Равенство (1.1) для *k* = 3 имеет вид:

* i* (3) = **{*j*(2) + *cji*}.

** 2 (3) = *min*{0 + 1; 1 + ∞; 9 + ∞; 7 + ∞; 2 + ∞} = 1.

** 3 (3) = *min*{0 + ∞ ; 1 + 8; 9 + ∞; 7 + 2; 2 + ∞} = 9 .

** 4 (3) = *min*{0 + ∞ ; 1 + 7; 9 + 1; 7 + ∞; 2 + 4} = 6 .

** 5 (3) = *min*{0 + 3; 1 + 1; 9 – 5; 7 + ∞; 2 + ∞} = 2.

Полученные значения * i* (3) занесем в четвертый столбец табл. 1.1. Величины * i* (3) равны длине минимального пути из первой вершины в *i*-ую, содержащего не более трех дуг.

*k =* 4.

** 1 (4) = 0.

Равенство (1.1) для *k* = 4 имеет вид:

* i* (4) = **{*j* (3) + *cji* }.

** 2 (4) = *min*{0 + 1; 1 + ∞; 9 + ∞; 6 + ∞; 2 + ∞} = 1.

** 3 (4) = *min*{0 + ∞ ; 1 + 8; 9 + ∞; 6 + 2; 2 + ∞} = 8 .

** 4 (4) = *min*{0 + ∞ ; 1 + 7; 9 + 1; 6 + ∞; 2 + 4} = 6 .

** 5 (4) = *min*{0 + 3; 1 + 1; 9 – 5; 6 + ∞; 2 + ∞} = 2 .

Полученные значения * i* (4) занесем в пятый столбец табл. 1.1. Величины * i* (4) равны длине минимального пути из первой вершины в *i*-ую, содержащего не более четырех дуг.

###  Таблица 1.1

|  |  |
| --- | --- |
| *I* (номер вершины) | * i* (0) * i* (1) * i*(2) * i* (3) * i* (4) |
| 12345 | 0 0 0 0 0∞ 1 1 1 1∞ ∞ 9 9 8∞ ∞ 7 6 6∞ 3 2 2 2 |

*Шаг 5.* Восстановление минимального пути.

Для последней вершины *x*3предшествующая ей вершина *xr* определяется из соотношения (1.2), полученного при *s* = 3:

* r* (3) *+* *c r* 3 = ** 3 (4), *x r* ∈ *G -*1( *x* 3 )*,* (1.3)

где *G -*1( *x* 3 ) *-*  прообраз вершины *x* 1.

*G -*1( *x* 3 )= { *x* 2, *x* 4 }.

Подставим в (1.3) последовательно *r* = 2 и *r* = 4, чтобы определить, для какого *r* это равенство выполняется:

** 2 (3) *+* *c* 2 3 = 1 + 8 ≠ ** 3 (4) = 8 ,

** 4 (3) *+* *c* 4 3 = 6 + 2 = ** 3 (4) = 8.

Таким образом, вершиной, предшествующей вершине *x* 3, является вершина *x* 4.

Для вершины *x* 4предшествующая ей вершина *x r* определяется из соотношения (1.2), полученного при *s* = 4:

* r* (2) *+* *c r* 4 = ** 4 (3) , *x r* ∈ *G -*1( *x* 4 ) *,* (1.4)

где *G -*1( *x* 4 ) *-*  прообраз вершины *x* 4 *.*

*G -*1( *x* 4 )= {*x* 2, *x* 3, *x* 5 }.

Подставим в (1.4) последовательно *r* = 2, *r* = 3 и *r* = 5, чтобы определить, для какого *r* это равенство выполняется:

** 2 (2) *+* *c* 2 4 = 1 + 7 ≠ ** 4 (3) = 6 ,

** 3 (2) *+* *c* 3 4 = 1 + 1 ≠ ** 4 (3) = 6 ,

** 5 (2) *+* *c* 5 4 = 2 + 4 = ** 4 (3) = 6 ,

Таким образом, вершиной, предшествующей вершине *x* 4, является вершина *x* 5.

Для вершины *x* 5предшествующая ей вершина *x r* определяется из соотношения (1.2), полученного при *s* = 5:

* r* (1) *+* *c r* 5 = ** 5 (2) , *x r  G -*1( *x* 5 )*,* (1.5)

где *G -*1( *x* 5 ) *-*  прообраз вершины *x* 5 *.*

*G -*1( *x* 5 )= { *x* 1, *x* 2 } .

Подставим в (1.5) последовательно *r* = 1 и *r* = 2, чтобы определить, для какого *r* это равенство выполняется:

** 1 (1) *+* *c* 1 5 = 0 + 3 ≠ ** 5 (2) = 2 ,

** 2 (1) *+* *c* 2 5 = 1 + 1 = ** 5 (2) = 2 .

Таким образом, вершиной, предшествующей вершине *x* 5, является вершина *x* 2.

Для вершины *x* 2предшествующая ей вершина *x r* определяется из соотношения (1.2), полученного при *s* = 2:

* r* (0) *+* *c r* 2 = ** 2 (1), *x r  G -*1( *x* 2 ) *,*  (1.6)

где *G -*1( *x* 2 ) *-*  прообраз вершины *x* 2 *.*

*G -*1( *x* 2 )= { *x* 1 } .

Подставим в (1.6) *r* = 1, чтобы определить, выполняется ли это равенство:

** 1 (0) *+* *c* 1 2 = 0 + 1 = ** 2 (1) = 1.

Таким образом, вершиной, предшествующей вершине *x*2 , является вершина *x* 1.

Итак, найден минимальный путь – *x* 1*, x* 2*, x* 5*, x* 4*, x* 3, его длина равна 8.

*Пример 1.16.*

С помощью модифицированного алгоритма 1.1 найдем максимальный путь из верши­ны *х* 1 в вершину *х* 3 в графе, изображенном на рис. 1.11.



Рис. 1.11

*Шаг 1.* Введем число вершин графа *n* = 5. Матрица весов этого графа после замены знаков при длинах дуг на противоположные имеет вид:

 *C* = .

*Шаг 2.* Положим *k* = 0, ** 1 (0) = 0, ** 2 (0) = ** 3 (0) = ** 4 (0) = ** 5 (0) = . Эти значения занесем в первый столбец табл. 1.2.

*Шаг 1.*

*k =* 1.

**1 (1) = 0.

Равенство (1.1) для *k* = 1 имеет вид:

* i* (1) = **{* j* (0) + *c j i* }.

** 2 (1) = *min*{** 1 (0) + *c* 1 2 ; ** 2 (0) + *c* 2 2 ; ** 3 (0) + *c* 3 2 ; ** 4 (0) + *c* 4 2 ; ** 5 (0) + *c* 5 2 ;} = *min*{0 – 1; ∞ + ∞; ∞ + ∞; ∞ + ∞; ∞ + ∞} = –1.

** 3 (1) = *min*{** 1 (0) + *c* 1 3 ; ** 2 (0) + *c* 2 3 ; ** 3 (0) + *c* 3 3 ; ** 4 (0) + *c* 4 3 ; ** 5 (0) + *c* 5 3 ;} = *min*{0 + ∞ ; ∞ – 8; ∞ + ∞; ∞ – 2; ∞ + ∞} = ∞ .

** 4 (1) = *min*{** 1 (0) + *c* 1 4 ; ** 2 (0) + *c* 2 4 ; ** 3 (0) + *c* 3 4 ; ** 4 (0) + *c* 4 4 ; ** 5 (0) + *c* 5 4 ; } = *min*{0 + ∞ ; ∞ – 7; ∞ + ∞; ∞ + ∞; ∞ – 4} = ∞ .

** 5 (1) = *min*{** 1 (0) + *c* 1 5 ; ** 2 (0) + *c* 2 5 ; ** 3 (0) + *c* 3 5 ; ** 4 (0) + *c* 4 5 ; ** 5 (0) + *c* 5 5 ; } = *min*{0 – 3; ∞ – 1; ∞ + 5; ∞ + ∞; ∞ + ∞} = –1.

Полученные значения * i* (1) занесем во второй столбец табл. 1.2. Убеждаемся, что второй столбец, начиная со второго элемента, совпадает с первой строкой матрицы весов, что легко объясняется смыслом величин *i*(1), которые равны длине минимального пути из первой вершины в *i*-ую, содержащего не более одной дуги.

*k =* 2.

** 1 (2) = 0.

Равенство (1.1) для *k* = 2 имеет вид:

* i* (2) = **{* j* (1) + *c j i* }.

** 2 (2) = *min*{0 – 1; –1 + ∞; ∞ + ∞; ∞ + ∞; –3 + ∞} = –1.

** 3 (2) = *min*{0 + ∞ ; –1 – 8; ∞ + ∞; ∞ – 2; –3 + ∞} = –9 .

** 4 (2) = *min*{0 + ∞ ; –1 – 7; ∞ + ∞; ∞ + ∞; –3 – 4} = –8 .

** 5 (2) = *min*{0 – 3; –1 – 1; ∞ + 5; ∞ + ∞; –3 + ∞} = –1.

Полученные значения * i* (2) занесем в третий столбец табл. 1.2. Величины * i* (2) равны длине минимального пути из первой вершины в *i*-ую, содержащего не более двух дуг.

*k =* 1.

** 1 (3) = 0.

Равенство (1.1) для *k* = 3 имеет вид:

* i* (3) = **{* j* (2) + *c j i* }.

** 2 (3) = *min*{0 – 1; – 1 + ∞; – 9 + ∞; –8 + ∞; – 3 + ∞} = – 1.

** 3 (3) = *min*{0 + ∞ ; – 1 – 8; – 9 + ∞; –8 – 2; – 3 + ∞} = – 10 .

** 4 (3) = *min*{0 + ∞ ; – 1 – 7; – 9 + ∞; –8 + ∞; – 3 – 4} = – 8 .

** 5 (3) = *min*{0 – 3; – 1 – 1; – 9 + 5; –8 + ∞; – 3 + ∞} = – 4.

Полученные значения * i* (3) занесем в четвертый столбец табл. 1.2. Величины * i* (3) равны длине минимального пути из первой вершины в *i*-ую, содержащего не более трех дуг.

*k =* 4.

** 1 (4) = 0.

Равенство (1.1) для *k* = 4 имеет вид:

* i* (4) = **{* j* (3) + *c j i* }.

** 2 (4) = *min*{0 – 1; – 1 + ∞ ; – 10 + ∞; – 8 + ∞; – 4 + ∞} = – 1.

** 3 (4) = *min*{0 + ∞ ; – 1 – 8; – 10 + ∞; – 8 – 2; – 4 + ∞} = – 10 .

** 4 (4) = *min*{0 + ∞ ; – 1 – 7; – 10 + ∞; – 8 + ∞; – 4 – 4} = – 8 .

** 5 (4) = *min*{0 – 3; – 1 – 1; – 10 + 5; – 8 + ∞; – 4 + ∞} = – 5.

Полученные значения * i* (4) занесем в пятый столбец табл. 1.2. Величины * i* (4) равны длине минимального пути из первой вершины в *i*-ую, содержащего не более четырех дуг.

###

###  Таблица 1.2

|  |  |
| --- | --- |
| *i* (номер вершины) | * i* (0) * i* (1) * i*(2) * i* (3) * i* (4) |
| 12345 | 0 0 0 0 0∞ – 1 – 1 – 1 1∞ ∞ – 9 – 10 – 10∞ ∞ – 8 – 8 – 8∞ – 3 –3 – 4 – 5 |

Заменив в табл. 1.2 отрицательные числа положительными, получим таблицу индексов максимальных путей ( табл. 1.3). При этом * i* ( *k* ) определяет длину максимального пути из первой вершины в *i*-ую, содержащего не более *k* дуг.

###  Таблица 1.3

|  |  |
| --- | --- |
| *i*(номер вершины) | * i* (0) * i* (1) * i*(2) * i* (3) * i* (4) |
| 12345 | 0 0 0 0 0∞ 1 1 1 1∞ ∞ 9 10 10∞ ∞ 8 8 8∞ 3 3 4 5 |

*Шаг 5.* Восстановление максимального пути производится по тому же правилу, что и для минимального пути.

Длина максимального пути равна 10. Этот путь состоит из трех дуг, т. к. * i* (3) = * i* (4) = 10. Поэтому в соотношении (1.2) будет выполнено, начиная с *n* – 1.

Учитывая это замечание, для последней вершины *x* 3предшествующую ей вершину *xr* определим из соотношения (1.2), полученного при *s* = 3:

* r* (2) *+* *c r* 3 = ** 3 (3), (1.7)

*x r*∈ *G -*1( *x* 3 )*,* где *G -*1( *x* 3 ) *-*  прообраз вершины *x*1.

*G -*1( *x* 3 )= { *x* 2,  *x* 4 }.

Подставим в (1.7) последовательно *r* = 2 и *r* = 4, чтобы определить, для какого *r* это равенство выполняется:

** 2 (2) *+* *c* 2 3 = 1 + 8 ≠ ** 3 (4) = 10,

** 4 (2) *+* *c* 4 3 = 8 + 2 = ** 3 (4) = 10.

Таким образом, вершиной, предшествующей вершине *x* 3, является вершина *x* 4.

Для вершины *x* 4предшествующая ей вершина *x r* определяется из соотношения (1.2), полученного при *s* = 4:

* r* (1) *+* *c r* 4 = ** 4 (2), *x r* ∈ *G -*1 ( *x* 4 )*,* (1.8)

где *G -*1 (*x* 4 ) *-*  прообраз вершины *x* 4*.*

*G -*1( *x* 4 )= { *x* 2, *x* 5 }.

Подставим в (1.8) последовательно *r* = 2, *r* = 3 и *r* = 5, чтобы определить, для какого *r* это равенство выполняется:

** 2 (1) *+* *c* 2 4 = 1 + 7 = ** 4 (3) = 8,

** 5 (1) *+* *c* 5 4 = 3 + 4 ≠ ** 4 (3) = 8,

Таким образом, вершиной, предшествующей вершине *x* 4, является вершина *x* 2.

Для вершины *x* 2предшествующая ей вершина *x r* определяется из соотношения (1.2), полученного при *s* = 2:

* r* (0) *+* *c r* 2 = ** 2 (1), *x r G -*1 (*x* 2 )*,* (1.9)

где *G -*1( *x* 2 ) *-*  прообраз вершины *x* 2*.*

*G -*1 ( *x* 2 )= { *x* 1}.

Подставим в (1.9) *r* = 1, чтобы определить, выполняется ли это равенство:

** 1 (1) *+* *c* 1 2 = 0 + 1 = ** 2 (1) = 1.

Таким образом, вершиной, предшествующей вершине *x* 2, является вершина *x* 1.

Итак, найден максимальный путь – *x* 1*, x* 2*, x* 4*, x* 3, его длина равна 10.

*Пример 1.17.*

Графы, изображенные на рис. 1.12, являются деревьями.

******

Рис. 1.12

*Пример 1.18.*

Для графа, изображенного на рис. 1.13*а*, графы на рис. 1.13*б* и 1.13*в* являются остовными деревьями.



Рис. 1.13

*Пример 1.19.*

Найдем минимальное остовное дерево для графа, изображенного на рис. 1.14.



Рис. 1.14

*Шаг 1*. Установка начальных значений.

Введем матрицу длин ребер *C*:

*С* =  .

*Шаг 2*. Выберем ребро минимальной длины. Минимальная длина ребра равна единице. Таких ребер три: ( *x* 1, *x* 2 ), ( *x* 1, *x* 4 ), (*x* 2 , *x* 4 ). В этом случае можно взять любое. Возьмем ( *x* 1, *x* 2 ). Построим граф *G* 2, состоящий из данного ребра и инцидентных ему вершин *x* 1 и *x* 2. Положим *i* = 2.

*Шаг 1.* Так как *n* = 5, то *i* ≠ *n*, поэтому переходим к шагу 4.

*Шаг 4.* Строим граф *G* 3, добавляя к графу *G* 2новое ребро мини­мальной длины, выбранное среди всех ребер графа *G*, каждое из которых инцидентно одной из вершин *x* 1, *x* 2 и одновременно инцидентно какой-нибудь вершине графа *G*, не содержащейся в *G* 2 т. е. одной из вершин *x* 3, *x*4, *x* 5. Таким образом, нужно выбрать ребро мини­мальной длины из ребер (*x* 1, *x*4), ( *x* 1, *x* 5 ), ( *x* 2, *x* 3 ), ( *x* 2, *x* 4 ), ( *x* 2, *x* 5 ). Таких ребер длины единица два: (*x*1, *x* 4 ) и ( *x* 2, *x* 4 ). Можно выбрать любое. Возьмем (*x* 1, *x* 4 ). Вместе с этим ребром включаем в *G* 3 вершину *x* 4, не содержащуюся в *G* 2. Полагаем *i* =3 и переходим к шагу 1.

*Шаг 1.* Так как *i* ≠ *n*, поэтому переходим к шагу 4.

*Шаг 4.* Строим граф *G* 4, добавляя к графу *G* 3новое ребро мини­мальной длины из ребер ( *x* 1, *x* 5 ), ( *x* 2, *x* 3 ), ( *x* 2, *x* 5 ), ( *x* 4, *x* 5 ). Такое ребро длины два одно: ( *x* 2, *x* 3 ). Вместе с этим ребром включаем в *G* 4 вершину *x*3, не содержащуюся в *G* 1. Полагаем *i* =4 и переходим к шагу 1.

*Шаг 1.* Так как *i* ≠ *n*, поэтому переходим к шагу 4.

*Шаг 4.* Строим граф *G* 5, добавляя к графу *G* 3новое ребро мини­мальной длины из ребер ( *x* 1, *x* 5 ), ( *x* 2, *x* 5 ), ( *x* 4, *x* 5 ). Таких ребер длины три два: ( *x* 2, *x* 5 ) и ( *x* 4, *x* 5 ). Возьмем ( *x* 2, *x* 5 ). Вместе с этим ребром включаем в *G* 5 вершину *x* 5, не содержащуюся в *G* 4. Полагаем *i* =5 и переходим к шагу 1.

*Шаг 1.* Так как *i* = *n*, то граф *G* 5 – искомое минимальное остовное дерево. Суммарная длина ребер равна 1 + 1 + 2 + 3 = 7.

Процесс построения минимального остовного дерева изображен на рис. 1.15.



Рис. 1.15

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

**Цель**: Изучение базовых алгоритмов комбинаторного перебора.

**Задачи**:

– Закрепление материала лекций.

– Изучение ряда дополнительных алгоритмов комбинаторного перебора, не рассмотренных на лекциях.

– Знакомство с существующей литературой, посвященной алгоритмам комбинаторного перебора.

– Выработка умений по практическому созданию алгоритмов комбинаторного перебора.

– Выработка умений по решению ряда стандартных задач, связанных с основными понятиями комбинаторики.

– Выработка умений по сравнительному анализу вычислительной сложности алгоритмов комбинаторного перебора.

– Подготовка предварительных версий реализаций алгоритмов комбинаторного перебора для последующего использования в процессе лабораторных занятий.

– Изучение общих принципов комбинаторной генерации.

– Текущий контроль успеваемости и выполнения плана самостоятельной работы.

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1. *Окулов*, *С.М.* Программирование в алгоритмах / С.М. Окулов. - М.: Бином, 2007. 341с.
2. *Кнут, Д*. Искусство программирования. Т. 4 Вып.2. Генерация всех кортежей и перестановок / Д. Кнут. - М.: Вильямс, 2008. 160 с.
3. *Кнут, Д*. Искусство программирования. Т. 4 Вып.3. Генерация всех сочетаний и разбиений / Д. Кнут. - М.: Вильямс, 2008. 208 с.
4. *Кнут, Д.* Искусство программирования. Т. 4 Вып.4. Генерация всех деревьев. История комбинаторной генерации / Д. Кнут. - М.: Вильямс, 2008. 160 с.

**РАССМАТРИВАЕМЫЕ ТЕМЫ С УКАЗАНИЕМ ЛИТЕРАТУРЫ**

1. **Основные комбинаторные объекты**.

[1] с. 27-34; [2] с. 12.

1. **Кортежи, перестановки, сочетания, размещения**.

[1] с. 27-34; [2] с. 12-13.

1. **Генерация кортежей**.

[2] с. 12-53.

1. **Коды Грея**.

[2] с. 12-53.

1. **Генерация перестановок**.

[1] с. 34-44; [2] с. 54-90.

1. **Генерация сочетаний**.

[1] с. 50-58; [3] с. 10-51.

1. **Генерация размещений**.

[1] с. 44-50

1. **Генерация разбиений**.

[1] с. 58-63; [3] с. 52-82

1. **Генерация деревьев**.

[4] с. 12-66.

ПЕДАГОГИЧЕСКИЕ ТЕХНОЛОГИИ

– фронтальный опрос,

– индивидуальные доклады (с презентациями),

– рассмотрение проблемных ситуаций,

– интерактивное написание компьютерных программ,

– индивидуальные и групповые задания,

– совместное решение задач.

ВОПРОСЫ И ЗАДАЧИ

Выборку по *К* элементов из множества *А* можно производить, руководствуясь правилами: считать разными выборки, в которых один и тот же элемент занимает разные позиции (либо не считать), допускать повторение одного и того же элемента в выбираемой группе (либо не допускать) и др. В зависимости от наложения определенных ограничений (правил) при выборе элементов разделяют три типа формирования комбинаторных групп. Рассмотрим их на примере:

Пусть *N* = 4, *K* = 2. Элементы исходного множества будем хранить в массиве *А* (рис. 2.1)



**Рис. 2.1.**

Тогда группы по 2 элемента, выбираемые из множества *А* можно сформировать так, как показано в [таблице 2.1](http://www.intuit.ru/department/school/olympr/11/#table.11.1#table.11.1):

|  |
| --- |
| **Таблица 2.1** |
| **Размещения** | **Сочетания** |
| **С повторениями** | **Без повторений** | **С повторениями** | **Без повторений** |
| 00 | 01 | 00 | 01 |
| 01 | 02 | 01 | 02 |
| 02 | 03 | 02 | 03 |
| 03 | 10 | 03 | 12 |
| 10 | 12 | 11 | 13 |
| 11 | 13 | 12 | 23 |
| 12 | 20 | 13 |  |
| 13 | 21 | 22 |  |
| 20 | 23 | 23 |  |
| 21 | 30 | 33 |  |
| 22 | 31 |  |  |
| 23 | 32 |  |  |
| 30 |  |  |  |
| 31 |  |  |  |
| 32 |  |  |  |
| 33 |  |  |  |
| Группы {01} и {10} считаются различными | Исключаются группы, где один и тот же элемент стоит в разных позициях | Группы {01} и {10} считаются одинаковыми | Исключаются группы, где один и тот же элемент стоит в разных позициях |

Существует еще и третий способ формирования комбинаторных групп: "Перестановки". В перестановках участвуют все элементы исходного множества (*K* = *N* ). Перестановки с повторениями возможны, когда в исходном множестве есть повторяющиеся элементы.

#### Типовые алгоритмы формирования групп:

**Размещения с повторениями:**

|  |  |
| --- | --- |
| Программная реализация на Бейсикеfor i=1 to nfor j=1 to nprint A(i), A(j);next j, i | Программная реализация на Паскалеfor i:=1 to n dofor j:=1 to n dowriteln (A[i], A[j]); |

**Размещения без повторений:**

|  |  |
| --- | --- |
| Программная реализация на Бейсикеfor i=1 to nfor j=1 to nif i<>j then print A(i), A(j);next j,i | Программная реализация на Паскалеfor i:=1 to n dofor j:=1 to n doif i<>j then writeln (A[i], A[j]); |

**Сочетания с повторениями:**

|  |  |
| --- | --- |
| Программная реализация на Бейсике...for i=1 to nfor j=i to nprint A(i,j);next j, i | Программная реализация на Паскале...for i:=1 to n dofor j:=i to n dowriteln (A[i], A[j]); |

**Сочетания без повторений:**

|  |  |
| --- | --- |
| Программная реализация на Бейсике...for i=1 to n-1for j=i+1 to nprint A(i), A(j);next j, i | Программная реализация на Паскале...for i:=1 to n-1 dofor j:=i+1 to n dowriteln (A[i], A[j]);  |

##### Задача "Кодовый замок сейфа"

Из 10 букв нужно набрать 3. Повторение букв допустимо. Подсчитать количество возможных комбинаций кодов.

**Идея решения:** Необходимо применить типовой алгоритм формирования групп РАЗМЕЩЕНИЯ С ПОВТОРЕНИЯМИ.

Программа на Бейсике

dim a$(10)

for i=1 to 10

 input "введите букву"; a$(i)

next

for х1=1 to 10

 for х2=1 to 10

 for х3=1 to 10

 print a$(х1); a$(х2); a$(х3);

 k=k+1

next x3,x2,x1

print "k="; k

Программа на Паскале:

var a: array [1.10] of char;

 x1, x2, x3, k, i: integer;

begin

 for i:=1 to 10 do

 readln (a[i]);

 for х1:=1 to 10 do

 for х2:= 1 to 10 do

 for х3:=1 to 10 do

 begin

 writeln (a[х1], a[х2], a[х3]);

 k:=k+1;

 end;

 writeln ('k:=', k);

end.

**Тест:**

|  |  |
| --- | --- |
| Результат: | 1000 |

**Задача**

В ассортименте кондитерской сегодня 10 видов пирожных. Каких три пирожных продавец может предложить покупателю?

**Идея решения:** Необходимо применить типовой алгоритм формирования групп *СОЧЕТАНИЯ С ПОВТОРЕНИЯМИ*.

Программа на Бейсике

dim a$ (10)

for i=1 to 10

 input "введите название пирожного"; a$ (i)

next

for х1=1 to 10

 for х2=1 to 10

 for х3=1 to 10

 print a$(х1); a$(х2); a$(х3)

next x3,x2,x1

Программа на Паскале:

var a: array [1.10] of string;

 x1, x2, x3, i: integer;

begin

 for i:=1 to 10 do

 begin

 writeln ('введите название пирожного');

 readln (a[i]);

 end;

 for х1:=1 to 10 do

 for х2:= 1 to 10 do

 for х3:=1 to 10 do

 writeln (a[х1], a[х2], a[х3]);

end.

##### Задачи из "Теории чисел"

*Найти все трехзначные числа, сумма цифр которых равна К* (*введенному с клавиатуры*)*.*

**Идея решения:** Необходимо применить типовой алгоритм формирования групп РАЗМЕЩЕНИЯ С ПОВТОРЕНИЯМИ.

Программа на Бейсике

Input "k="; k

for х1=1 to 9

 for х2= 0 to 9

 for х3=0 to 9

 if х1+х2+х3=k then print х1; х2; х3

next x3,x2,x1

Программа на Паскале:

var x1, x2, x3, k: integer;

begin

 writeln ('k=');

 readln (k);

 for х1:=1 to 9 do

 for х2:=0 to 9 do

 for х3:=0 to 9 do

 if х1+х2+х3=k then

 writeln (х1, х2, х3);

end.

**Тест:**

|  |  |
| --- | --- |
| Дано: | 3 |
| Результат: | 102111120201210300 |

*Подсчитать количество "счастливых" троллейбусных билетов ("счастливые" номера билетов - шестизначные числа, в которых сумма первых трех цифр равна сумме вторых трех цифр).*

**Идея решения:** Необходимо применить типовой алгоритм формирования групп РАЗМЕЩЕНИЯ С ПОВТОРЕНИЯМИ.

Программа на Бейсике:

for х1=1 to 9

 for х2=0 to 9

 for х3=0 to 9

 for х4=0 to 9

 for х5=0 to 9

 for х6=0 to 9

 if x1+x2+x3=x4+x5+x6 then k=k+1

next x6, x5, x4, x3, x2, x1

print "k="; k

Программа на Паскале:

var x1, x2, x3, k: integer;

begin

 k:=0;

 for х1:=1 to 9 do

 for х2:=0 to 9 do

 for х3:=0 to 9 do

 for х4:=0 to 9 do

 for х5:=0 to 9 do

 for х6:=0 to 9 do

 if x1+x2+x3=x4+x5+x6 then k:=k+1;

 writeln ('k=', k);

end.

**Тест:**

|  |  |
| --- | --- |
| Результат: | 46242 |

##### Геометрические задачи

*На плоскости N точек заданы своими координатами. Найти "центральную" точку (точку, сумма расстояний от которой до остальных точек максимальна).*

**Идея решения:** Необходимо применить типовой алгоритм формирования групп *РАЗМЕЩЕНИЯ БЕЗ ПОВТОРЕНИЙ*. Для поиска максимальной суммы расстояний применим типовой алгоритм *ПОИСКА МАКСИМАЛЬНОГО ЭЛЕМЕНТА*.

Программа на Бейсике:

input "введите количество точек"; n

dim x (n), y (n)

for i=1 to n

 input "введите пары координат"; x(i), y(i)

next

for i=1 to n

 s=0

 for j:=1 to n

 s=s+ sqr ((x(i)-x(j))^2+(y(i)-y(j))^2)

 next

 if s>max then max=s: num=i

next

print "это точка с координатами-"; x(num), y(num)

Программа на Паскале:

var x,y:array [1..10] of integer;

 i,j,n,num,max:integer;

begin

 writeln ('введите количество точек');

 readln (n);

 for i:=1 to n do

 begin

 writeln ('введите пары координат');

 readln (x[i], y[i]);

 end;

 for i:=1 to n do

 begin

 s:=0;

 for j:=1 to n do

 s:=s+ sqrt (sqr(x[i]-x[j])+sqr(y[i]-y[j]));

 if s>max then

 begin

 max:=s;

 num:=i;

 end;

 end;

 writeln (x[num], y[num]);

End.

**Тест:**

|  |  |
| --- | --- |
| Дано: | N=51,64,26,57,1010,3 |
| Результат: | 7,10 |

*На плоскости N точек заданы своими координатами. Найти 2 наиболее удаленные друг от друга точки.*

**Идея решения:** Необходимо применить типовой алгоритм формирования групп *СОЧЕТАНИЯ БЕЗ ПОВТОРЕНИЙ*. Для поиска двух наиболее удаленных точек применим типовой алгоритм *ПОИСКА МАКСИМАЛЬНОГО ЭЛЕМЕНТА*.

Программа на Бейсике:

input "введите количество точек"; n

dim x(n), y(n)

for i = 1 to n

 input "введите пары координат"; x(i), y(i)

next

for i = 1 to n - 1

 for j = i + 1 to n

 ras = sqr((x(i)-x(j))^ + (y(i)-y(j))^2)

 if ras < max then max = ras: num1 = i: num2 = j

next j, i

print x(num1); y(num1); "-"; x(num2); y(num2)

Программа на Паскале:

var x,y:array [1..10] of integer;

 n,i,j,num1,num2:integer;

 ras,max:real;

begin

 writeln ('введите количестов точек');

 readln (n);

 for i:=1 to n do

 begin

 writeln ('введите пары координат');

 readln (x[i], y[i]);

 end;

 {=================================}

 for i:=1 to n-1 do

 for j:=i+1 to n do

 begin

 ras:= sqrt(sqr(x[i] - x[j]) +sqr (y[i] - y[j])) ;

 if ras > max then

 begin

 max:=ras;

 num1:=i;

 num2:=j;

 end;

 end;

writeln (x[num1], y[num1], '-', x[num2], y[num2]);

end.

**Тест:**

|  |  |
| --- | --- |
| Дано: | N=63,56,87,69,1215,1013,3 |
| Результат: | 3,5 - 15,10 |

*На плоскости N точек заданы своими координатами. Найти минимальный радиус окружности, включающей в себя все точки.*

**Идея решения:**

Минимальной будет окружность, на которой находятся хотя бы три точки ( [рис. 2.2](http://www.intuit.ru/department/school/olympr/11/2.html#image.11.2#image.11.2) ). Необходимо найти такие три точки, сумма расстояний между которыми максимальна. Необходимо применить типовой алгоритм формирования групп *СОЧЕТАНИЯ БЕЗ ПОВТОРЕНИЙ*.



**Рис. 2.2.**

Радиус описанной окружности: *R* = (*abc*) / (4*S*)

*В приведенных ниже программах находятся координаты трех, наиболее удаленных друг от друга точек. Вычислить R не составит труда (проделайте это самостоятельно).*

Программа на Бейсике:

input "n="; n

dim x(n), y(n)

print "x=, y="

for i = 1 to n

 input x(i), y(i)

next

for i = 1 to n - 2

 for j = i + 1 to n - 1

 for r = j + 1 to n

 ras1 = sqr((x(r) - x(j)) ^ 2 + (y(r) - y(j)) ^ 2)

 ras2 = sqr((x(j) - x(i)) ^ 2 + (y(j) - y(i)) ^ 2)

 ras3 = sqr((x(i) - x(r)) ^ 2 + (y(i) - y(r)) ^ 2)

 if (ras1+ras2+ras3)>max then max=(ras1+ras2+ras3): num1=i: num2=j: num3=r

next r, j, i

print x(num1); y(num1); "-"; x(num2); y(num2); "-"; x(num3); y(num3)

Программа на Паскале:

var x,y: array [1..10] of integer;

 n,i,j,r,num1,num2,num3: integer;

 ras1,ras2,ras3,max: real;

begin

 writeln ('введите количестов точек'); readln (n);

 for i:=1 to n do

 begin

 writeln ('введите пары координат'); readln (x[i], y[i]);

 end;

 max:=0;

 for i:= 1 to (n - 2) do

 for j:= i + 1 to (n - 1) do

 for r:= j + 1 to n do

 begin

 ras1:=sqrt(sqr(x[r]-x[j])+sqr(y[r]-y[j]));

 ras2:=sqrt(sqr(x[j]-x[i])+sqr(y[j]-y[i]));

 ras3:=sqrt(sqr(x[i]-x[r])+sqr(y[i]-y[r]));

 if (ras1+ras2+ras3)>max then

 begin

 max:=(ras1+ras2+ras3); num1:=i; num2:=j; num3:=r;

 end;

 end;

 writeln (x[num1],' ',y[num1], '-',x[num2],' ',y[num2], '-', x[num3],' ',y[num3]);

end.

**Тест:**

|  |  |
| --- | --- |
| Дано: | N=84,113,74,37,106,76,211,89,5 |
| Результат: | 4,11-6,2-11,8 |

Пусть *N* = 4. Необходимо сформировать различные группы элементов, выбираемых из исходного множества. Количество элементов в выборке от 1 до *N*. Элементы исходного множества будем хранить в массиве *А* ([рис. 2.](http://www.intuit.ru/department/school/olympr/12/#image.12.1#image.12.1)3):



**Рис. 2.3.**

Составим таблицу, в которой выбранный элемент отметим "1", невыбранный - "0":

|  |
| --- |
| **Таблица 2.2.**  |
| **1** | **2** | **3** |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| 1 | 1 | 1 |

**Итак**, сформированы группы: {3}, {2}, {2, 3}, {1}, {1,3}, {1, 2}, {1, 2, 3}.

Для формирования групп потребовалось перебрать все варианты комбинаций "0" и "1". Такой метод формирования комбинаторных групп называется "Двоичным перебором", а количество групп будет равно 2 ***n*** -1.

**Идея решения:** для выбора элементов из исходного множества необходимо получить двоичный код (на единицу больше предыдущего). Первый вариант получения нового двоичного кода - перевод счетчика цикла *i* из десятичной в двоичную систему счисления. Второй вариант получения очередного двоичного кода - ищем в массиве двоичных кодов *d* последний нулевой элемент , заменяем его на единицу и обнуляем все следующие за ним элементы (этот метод называется лексигрофическим порядком).

Количество возможных комбинаций двоичных кодов 2 ^ *n*-1 (исключаем двоичный код, состоящий из одних нулей).

Программная реализация на Бейсике:

input "введите количество элементов исх. множества="; n

for i=1 to n

 input "введите элемент"; a(i)

next

for i=1 to 2^n-1

 rem=поиск первого нулевого элемента=========

 for j=1 to n

 if d(j)=0 then x=j

 next

 rem=формирование двоичного кода===========

 for z=x to n

 d(z)=0

 next z

 d(x)=1

 rem=печать элементов====================

 for j=1 to n

 if d(j) <> 0 then print a(j);

 next j

 print

next i

Программная реализация на Паскале:

const nn=10;

var a,d: array [1..nn] of integer;

 i,n,x,j,z,st: integer;

begin

 writeln ('количество элементов');

 readln (n);

 for i:= 1 to n do

 begin

 writeln ('введите элемент');

 readln (a[i]);

 end;

 {=формирование двоичного кода===}

 st:=1;

 for i:=1 to n do

 st:=st\*2;

 for i:= 1 to (st-1) do

 begin

 for j:= 1 to n do

 if d[j]= 0 then x:= j;

 for z:= x to n do

 d[z]:=0;

 d[x]:=1;

 {=печать элементов========}

 for j:= 1 to n do

 if d[j]<>0 then write (a[j]);

 writeln;

 end;

end.

**Тест:**

|  |  |
| --- | --- |
| Дано: | *N*=3{ 1,2,3} |
| Результат: | 322,311,31,21,2,3 |

Если предположить, что каждый элемент из исходного набора может повторяться во вновь созданной комбинаторной группе от 0 до *n* раз, то необходимо организовать *n*-ный перебор.

**Задачи:**

* *"Размен монет": дана купюра достоинством X. Требуется разменять ее монетами по 1, 5, 10, 50 рублей.*
* *Даны гири массами M1, M2, M3, M4. Как можно взвесить предмет массой X ?*
* *Даны N чисел. Выделите из них группы, содержащие от 1 до N элементов, каждая из которых имеет сумму X.*

Программная реализация на Бейсике:

input "x="; x

input "количество элементов в исходном множестве"; n

for i = 1 to n

 input "введите элемент"; a(i)

next

for i = 1 to (2^n - 1)

 rem==получение следующего двоичного кода==

 for j = 1 to n

 if d(j) = 0 then k = j

 next j

 for z = k to n

 d(z) = 0

 next z

 d(k) = 1

 rem=============================

 s = 0

 for j = 1 to n

 if d(j) <> 0 then s = s + a(j)

 next j

 rem========вывод результата==========

 if s = x then

 for ii = 1 to n

 if d(ii) <> 0 then print " "; a(ii);

 next ii

 end if

 print

next i

Программная реализация на Паскале:

const nn=10;

var a,d: array [1..nn] of integer;

 ii,i,n,x,j,z,st,k: integer;

begin

 writeln ('введите с чем сравнивать);

 readln (х);

 writeln ('введите количество элементов');

 readln (n);

 for i:= 1 to n do

 begin

 writeln ('введите элемент');

 readln (a[i]);

 end;

 {=вычисление количества возможных комбинаций=}

 st:=1;

 for i:=1 to n do

 st:=st\*2;

 {=================================}

 for i:= 1 to (st-1) do

 begin

 {=получение следующего двоичного кода===}

 for j:= 1 to n do

 if d[j]= 0 then k:= j;

 for z:= k to n do

 d[z]:=0;

 d[k]:=1;

 {=============================}

 s: = 0;

 for j: = 1 to n do

 if d[j] <> 0 then s:= s + a[j];

 if s = x then

 {=====вывод результата=========}

 for ii:= 1 to n do

 if d[ii] <> 0 then write (a[ii]);

 writeln;

 end;

end.

**Тест:**

|  |  |
| --- | --- |
| Дано: | x=5n=31,2,3 |
| Результат: | 2,3 |

**Тема 3. Общие методы разработки алгоритмов (4 часа)**

**Цель**: Изучение общих методов построения алгоритмов и их применение к решению задач.

**Задачи**:

– Закрепление материала лекций.

– Изучение ряда дополнительных методов разработки алгоритмов (динамическое программирование, жадные алгоритмы и т.п.).

– Выработка умений по практическому созданию новых типов алгоритмов.

– Выработка умений по сравнительному анализу вычислительной сложности алгоритмов.

–– Подготовка предварительных версий реализаций алгоритмов для последующего использования в процессе лабораторных занятий.

– Текущий контроль успеваемости и выполнения плана самостоятельной работы.

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1. *Кормен*, *Т*. Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. - М.: Вильямс, 2007. 1296 с.
2. *Ху, Т.Ч*. Комбинаторные алгоритмы / Т.Ч. Ху, М.Т. Шинг. - Нижний Новгород: Изд-во Нижегородского госуниверситета им. Н.И. Лобачевского, 2004. 330 с.
3. *Ахо*, *А*. Структуры данных и алгоритмы / А. Ахо, Д. Хопкрофт, Д. Ульман. - М.: Вильямс, 2007. 384 с.

**РАССМАТРИВАЕМЫЕ ТЕМЫ С УКАЗАНИЕМ ЛИТЕРАТУРЫ**

1. **Динамическое программирование**.

[1] с. 386-440; [2] с. 109-136. [3] с. 280-288

1. **«Жадные» алгоритмы**.

[1] с. 442-480; [2] с. 190-196. [3] с. 288-291

1. **Перебор с возвратом**.

[2] с. 137-155. [3] с. 291-302.

1. **Алгоритмы «Разделяй и властвуй»**.

[3] с. 276-280.

1. **Амортизационный анализ**.

[1] с. 482-510.

1. **Эвристические алгоритмы**

[2] с. 190-222.

ПЕДАГОГИЧЕСКИЕ ТЕХНОЛОГИИ

– фронтальный опрос,

– индивидуальные доклады (с презентациями),

– рассмотрение проблемных ситуаций,

– интерактивное написание компьютерных программ,

– индивидуальные и групповые задания,

– совместное решение задач.

ВОПРОСЫ И ЗАДАЧИ

*Использовать метод* обхода дерева *для решения следующей задачи: дан* массив *из n целых положительных чисел и число s ; требуется узнать, может ли число s быть представлено как сумма некоторых из чисел массива a. (Каждое число можно использовать не более чем по одному разу.)*

**Решение**. Будем задавать *k* –позицию последовательностью из *k* булевских значений, определяющих, входят ли в сумму числа или не входят. Позиция допустима, если ее сумма не превосходит *s*.

**Замечание**. По сравнению с полным перебором всех подмножеств тут есть некоторый выигрыш. Можно также предварительно отсортировать массив *a* в убывающем порядке, а также считать недопустимыми те позиции, в которых сумма отброшенных членов больше, чем разность суммы всех членов и *s*. Последний прием называют «методом ветвей и границ». Традиционное название этой задачи – «**задача о рюкзаке**» (рюкзак общей грузоподъемностью *s* нужно упаковать «*под завязку*», располагая предметами веса ). Перечислить все последовательности из нулей, единиц и двоек, в которых никакая группа цифр не повторяется два раза подряд (нет куска вида ).

Аналогичная задача для последовательностей нулей и единиц, в которых никакая группа цифр не повторяется три раза подряд (нет куска вида ).

К этой же категории относятся задачи типа «можно ли сложить данную фигуру из пентамино» и им подобные. В них важно умелое сокращение перебора (вовремя распознать, что имеющееся расположение фигурок уже противоречит требованиям, и по этой ветви поиск не продолжать).

*Написать рекурсивную программу возведения в целую неотрицательную степень.*

То же, если требуется, чтобы глубина рекурсии не превосходила , где - показатель степени.

**Решение**.

function power (a,n: integer): integer;

begin

| if n = 0 then begin

| | power:= 1;

| end else if n mod 2 = 0 then begin

| | power:= power(a\*a, n div 2);

| end else begin

| | power:= power(a, n-1)\*a;

| end;

end;

*Что будет, если изменить программу, приведенную в решении предыдущей задачи, заменив строку*

power:= power(a\*a, n div 2)

на

power:= power(a, n div 2)\* power(a, n div 2)?

**Решение**. Программа останется правильной. Однако она станет работать медленнее. Дело в том, что теперь вызов может породить два вызова (хотя и одинаковых) вместо одного - и число вызовов быстро растет с глубиной рекурсии. Программа по-прежнему имеет логарифмическую глубину рекурсии, но число шагов работы становится линейным вместо логарифмического.

Этот недостаток можно устранить, написав

t:= power(a, n div 2);

power:= t\*t;

или воспользовавшись функцией возведения в квадрат .

*Используя команды лишь при , написать рекурсивную программу печати десятичной записи целого положительного числа .*

**Решение**. Здесь использование рекурсии облегчает жизнь (проблема была в том, что цифры легче получать с конца, а печатать надо с начала).

procedure print (n:integer); {n>0}

begin

| if n<10 then begin

| | write (n);

| end else begin

| | print (n div 10);

| | write (n mod 10);

| end;

end;

### Порождение комбинаторных объектов, перебор

Рекурсивные программы являются удобным способом порождения комбинаторных объектов заданного вида. Мы решим заново несколько задач соответствующей лекции.

*Написать программу, которая печатает по одному разу все последовательности длины , составленные из чисел (их количество равно ).*

**Решение**. Программа будет оперировать с массивом и числом . Рекурсивная процедура печатает все последовательности, начинающиеся на ; после ее окончания и имеют то же значение, что и в начале:

procedure generate;

| var i,j : integer;

begin

| if t = n then begin

| | for i:=1 to n do begin

| | | write(a[i]);

| | end;

| | writeln;

| end else begin {t < n}

| | for j:=1 to k do begin

| | | t:=t+1;

| | | a[t]:=j;

| | | generate;

| | | t:=t-1;

| | end;

| end;

end;

Основная программа теперь состоит из двух операторов:

t:=0; generate;

**Замечание**. Команды и для экономии можно вынести из цикла .

*Написать программу, которая печатала бы все* перестановки *чисел по одному разу.*

**Решение**. Программа оперирует с массивом , в котором хранится перестановка чисел . Рекурсивная процедура в такой ситуации печатает все перестановки, которые на первых позициях совпадают с перестановкой ; по выходе из нее переменные и имеют те же значения, что и до входа. Основная программа такова:

for i:=1 to n do begin a[i]:=i; end;

t:=0;

generate;

Вот описание процедуры:

procedure generate;

| var i,j : integer;

begin

| if t = n then begin

| | for i:=1 to n do begin

| | | write(a[i]);

| | end;

| | writeln;

| end else begin {t < n}

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

| | | поменять местами a[t+1] и a[j]

| | | t:=t+1;

| | | generate;

| | | t:=t-1;

| | | поменять местами a[t+1] и a[j]

| | end;

| end;

end;

*Напечатать (по одному разу) все последовательности из нулей и единиц, содержащие ровно единиц.*

*Напечатать все возрастающие последовательности длины , элементами которых являются* натуральные числа *от до . (Предполагается, что , иначе таких последовательностей не существует.)*

**Решение**. Программа оперирует с массивом и целой переменной . Предполагая, что - возрастающая последовательность натуральных чисел из отрезка , рекурсивно определенная процедура печатает все ее возрастающие продолжения длины . (При этом и в конце такие же, как в начале.)

procedure generate;

| var i: integer;

begin

| if t = k then begin

| | печатать a[1]..a[k]

| end else begin

| | t:=t+1;

| | for i:=a[t-1]+1 to t-k+n do begin

| | | a[t]:=i;

| | | generate;

| | end;

| | t:=t-1;

| end;

end;

**Замечание**. Цикл мог бы иметь верхней границей (вместо ). Наш вариант экономит часть работы, учитывая тот факт, что предпоследний ( -ый) член не может превосходить , -ой член не может превосходить и т.п.

Основная программа теперь выглядит так:

t:=1;

for j:=1 to 1-k+n do begin

| a[1]:=j;

| generate;

end;

Можно было бы добавить к массиву слева фиктивный элемент , положить и ограничиться единственным вызовом процедуры .

*Перечислить все представления положительного целого числа в виде суммы последовательности невозрастающих целых положительных слагаемых.*

**Решение**. Программа оперирует с массивом (максимальное число слагаемых равно ) и с целой переменной }. Предполагая, что - невозрастающая последовательность целых чисел, сумма которых не превосходит , процедура печатает все представления требуемого вида, продолжающие эту последовательность. Для экономии вычислений сумма хранится в специальной переменной .

procedure generate;

| var i: integer;

begin

| if s = n then begin

| | печатать последовательность a[1]..a[t]

| end else begin

| | for i:=1 to min(a[t], n-s) do begin

| | | t:=t+1;

| | | a[t]:=i;

| | | s:=s+i;

| | | generate;

| | | s:=s-i;

| | | t:=t-1;

| | end;

| end;

end;

Основная программа при этом может быть такой:

t:=1;

for j:=1 to n do begin

| a[1]:=j

| s:=j;

| generate;

end;

**Замечание**. Можно немного сэкономить, вынеся операции увеличения и уменьшения из цикла, а также не возвращая каждый раз к исходному значению (увеличивая его на и возвращая к исходному значению в конце). Кроме того, добавив фиктивный элемент , можно упростить основную программу:

*t* := 0; s := 0; a [ 0 ] := n; generate;

*Написать рекурсивную программу* обхода дерева.

**Решение**. Процедура обработать\_над обрабатывает все листья над текущей вершиной и заканчивает работу в той же вершине, что и начала. Вот ее рекурсивное описание:

procedure обработать\_над;

begin

| if есть\_сверху then begin

| | вверх\_налево;

| | обработать\_над;

| | while есть\_справа do begin

| | | вправо;

| | | обработать\_над;

| | end;

| | вниз;

| end else begin

| | обработать;

| end;

end;

*Дан выпуклый -угольник (заданный* координатами *своих вершин в порядке обхода). Его разрезают на треугольники диагоналями, для чего необходимо диагонали (это можно доказать* индукцией *по ).* Стоимостью *разрезания назовем сумму длин всех использованных диагоналей. Найти минимальную* стоимость *разрезания. Число действий должно быть ограничено некоторым* многочленом *от . (Перебор не подходит, так как число вариантов не ограничено* многочленом*.)*

**Решение**. Будем считать, что вершины пронумерованы от до и идут по часовой стрелке. Пусть , - номера вершин, причем . Через обозначим многоугольник, отрезаемый от нашего хордой - . (Эта хорда разрезает многоугольник на два, один из которых включает сторону - ; через мы обозначаем другой, см. рис. 3.1) Исходный многоугольник естественно обозначить . При получается "двуугольник" с совпадающими сторонами.

Рис. 3.1

Через обозначим стоимость разрезания многоугольника диагоналями на треугольники. Напишем рекуррентную формулу для . При получается двуугольник, и мы полагаем . При получается треугольник, и в этом случае также . Пусть .

 Рис. 3.2

Хорда - является стороной многоугольника и, следовательно, стороной одного из треугольников, на которые он разрезан. Противоположной вершиной этого треугольника может быть любая из вершин , и минимальная стоимость разрезания может быть вычислена как



по всем . При этом надо учесть, что при хорда - - не хорда, а сторона, и ее длину надо считать равной (по стороне разрез не проводится).

Составив таблицу для и заполняя ее в порядке возрастания числа вершин (равного ), мы получаем программу, использующую память порядка и время порядка (однократное применение рекуррентной формулы требует выбора минимума из не более чем чисел).

*Матрицей размера называется прямоугольная таблица из строк и столбцов, заполненная числами. Матрицу размера можно умножить на матрицу размера (ширина левого сомножителя должна равняться* высоте *правого), и получается* матрица *размером . Ценой такого* умножения *будем считать* произведение *(таково число* умножений*, которые нужно выполнить при стандартном способе* умножения *- но сейчас это нам не важно).* Умножение матрицассоциативно*, поэтому* произведение **матриц *можно вычислять в разном порядке. Для каждого порядка подсчитаем суммарную цену всех матричных* умножений*. Найти минимальную цену вычисления произведения, если известны размеры всех* матриц*. Число действий должно быть ограничено* многочленом *от числа* матриц*.*

Матрицы размером , , можно перемножать двумя способами. В первом цена равна , во втором цена равна .

**Решение**. Представим себе, что первая матрица написана на отрезке , вторая - на отрезке , -ая - на отрезке . Матрицы на отрезках и имеют общий размер, позволяющий их перемножить. Обозначим его через . Таким образом, исходным ***данным*** в задаче является массив .

Через обозначим минимальную цену вычисления произведения матриц на участке (при ). Искомая величина равна . Величины равны нулю (матрица одна и перемножать ничего не надо). Рекуррентная формула будет такой:



где минимум берется по всем возможных местам последнего умножения, то есть по всем . В самом деле, произведение матриц на отрезке есть матрица размера , произведение матриц на отрезке имеет размер , и цена вычисления их произведения равна .

**Замечание**. Две последние задачи похожи. Это сходство станет яснее, если написать матрицы-множители на сторонах , многоугольника, а на каждой хорде написать произведение всех матриц, стягиваемых этой хордой.

*Использовать замену* рекурсиистеком *отложенных заданий в рекурсивной программе печати десятичной записи целого числа.*

**Решение**. Цифры добываются с конца и закладываются в стек, а затем печатаются в обратном порядке.

*Написать нерекурсивную программу, печатающую все вершины* двоичного дерева.

**Решение**. В этом случае стек отложенных заданий будет содержать заказы двух сортов: "напечатать данную вершину" и "напечатать все вершины поддерева с ***данным*** корнем" (при этом nil считается корнем пустого дерева). Таким образом, элемент стека есть пара: тип заказа, номер вершины .

Вынимая элемент из стека, мы либо сразу исполняем его (если это заказ первого типа), либо помещаем в стек три порожденных им заказа - в одном из шести возможных порядков.

*Что изменится, если требуется не печатать вершины* двоичного дерева*, а подсчитать их количество?*

**Решение**. Печатание вершины следует заменить прибавлением единицы к счетчику. Другими словами, инвариант таков: (общее число вершин = счетчик + сумма чисел вершин в поддеревьях, корни которых лежат в стеке).

*Для некоторых из шести возможных порядков возможны упрощения, делающие ненужным* ***хранение*** *в* стеке *элементов двух видов. Указать некоторые из них.*

**Решение**. Если требуемый порядок таков:



то заказ на печатание корня можно не закладывать в стек, а выполнять сразу.

Несколько более сложная конструкция применима для порядка



В этом случае все заказы в стеке, кроме самого первого (напечатать поддерево) делятся на пары:



( = поддерево с корнем в правом сыне *x* ). Объединив эти пары в заказы специального вида и введя переменную для отдельного *хранения* первого заказа, мы обойдемся стеком однотипных заказов.

То же самое, разумеется, верно, если поменять местами левое и правое - получается еще два порядка.

**Тема 4. Теория сложности алгоритмов (2 часа)**

**Цель**: Знакомство с основными понятиями и проблемами теории сложности алгоритмов.

**Задачи**:

– Знакомство с основными понятиями теории сложности алгоритмов.

– Знакомство с основными классами сложности алгоритмов.

– Формирование навыков по определению класса сложности алгоритма.

– Знакомство с проблемой *P* < > *NP*.

– Знакомство с понятием *NP*-полной задачи и с примерами *NP*-полных задач.

– Знакомство с существующей литературой, посвященной вычислительной сложности алгоритмов.

– Текущий контроль успеваемости и выполнения плана самостоятельной работы.

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1. *Кормен*, *Т*. Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. - М.: Вильямс, 2007. 1296 с.
2. *Ху, Т.Ч*. Комбинаторные алгоритмы / Т.Ч. Ху, М.Т. Шинг. - Нижний Новгород: Изд-во Нижегородского госуниверситета им. Н.И. Лобачевского, 2004. 330 с.

**РАССМАТРИВАЕМЫЕ ТЕМЫ С УКАЗАНИЕМ ЛИТЕРАТУРЫ**

1. **Понятие сложности алгоритма**.

[1] с. 52-55, [2] с. 252-255.

1. **Классы сложности**.

[1] с. 52-55, 1087-1088; [2] с. 252-257.

1. **Полиномиальные и экспоненциальные алгоритмы**.

[1] с. 1091-1100, [2] с. 255-257.

1. **Класс *NP***.

[1] с. 1100-1105, [2] с. 257-258.

1. ***NP* - полные алгоритмы**.

[1] с. 1106-1127, [2] с. 258-263.

1. **Примеры *NP* - полных задач**.

[1] с. 1127-1145.

1. **Проблема *P* < > *NP***.

[1] с. 1085-1088.

ПЕДАГОГИЧЕСКИЕ ТЕХНОЛОГИИ

– фронтальный опрос,

– индивидуальные доклады (с презентациями),

– рассмотрение проблемных ситуаций.

КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

## Введение

В процессе решения некоторых задач человечество пришло к выводу, что для этих задач не существует эффективного алгоритма, во всех случаях находящего правильный ответ. Чтобы подтвердить эти предположения, достаточно научиться оценивать снизу время, необходимое для решения этих задач. Доказательство таких утверждений приводит к понятию класса задач, обладающих одинаковой вычислительной сложностью. Некоторые классы задач обладают столь высокой сложностью, что человечество не обладает достаточным количеством ресурсов для решения этих задач.

Кроме того, работы XX века в области математической логики показали, что существуют задачи, для решения которых теоретически не может существовать алгоритма. Классическим примером такой задачи является проблема останова: «Дан исходный код программы, требуется определить, остановится ли она за конечное время, если ее запустить».

Исходя из этих соображений, хочется ограничить сверху вычислительные возможности алгоритмов и вычислительных устройств, поскольку становится ясно, что существуют задачи, не имеющие эффективных алгоритмов решения. Во второй половине XX века изучение этой области привело к созданию математической дисциплины — теории вычислительной сложности.

## Машина Тьюринга

Чтобы формально ввести и изучать понятия «алгоритм», «разрешимая проблема», необходимо ввести формальное вычислительное устройство, про которое можно строго доказывать математические факты. Исторически для этих целей была выбрана машина Тьюринга, описанная Аланом Тьюрингом в 1936 году.

Следует отметить, что для этих целей пригодна любая модель вычислительной машины, эквивалентная по мощности машине Тьюринга. Таких моделей известно немало — регистровые машины, счетчиковые машины, машины Поста-Тьюринга и многие другие. При использовании любой из этих моделей, результаты теории сложности не изменились бы, поскольку все перечисленные устройства эквивалентны в плане вычислительной мощности.

Кроме того, немаловажным является следующий факт: работу персонального компьютера можно эмулировать на машине Тьюринга за полиномиальное время, и наоборот. Не будь это верно, результаты теории сложности не имели бы никакого практического смысла.

Опишем формально машину Тьюринга. Далее в тексте будем использовать сокращение МТ.

Информация, подаваемая на вход МТ, хранится на ленте, бесконечной в обе стороны. В каждой клетке ленты хранится один символ из заранее зафиксированного алфавита входных символов. Входные данные располагаются на ленте одной группой, а все остальное пространство ленты заполнено зарезервированными символами — пробелами.

В каждый момент времени одна из клеток ленты является выделенной — говорят, что на эту клетку указывает ленточная головка. В начальный момент головка указывает на первый (непробельный) символ входных данных.

Также в каждый момент времени МТ находится в одном из конечного множества состояний. Одно из них является начальным. Некоторое множество состояний помечены как допускающие. Именно так МТ по завершении работы сообщит о полученном результате.

Каждый дискретный шаг происходит следующим образом. МТ имеет в данный момент некоторое состояние, а ленточная головка указывает на некоторый символ. Для этой пары — (состояние, символ) — в МТ однозначно определена соответствующая ей тройка: (новое состояние, новый символ, сдвиг). Соответственно, МТ переходит в новое состояние, на ленте вместо прочитанного символа оставляет новый символ (возможно, тот же самый) и ленточная головка сдвигается вдоль ленты либо влево, либо вправо на одну позицию.

## Недетерминированная машина Тьюринга

Определим иную, не менее важную для теории вычислительной сложности, модель вычислительной машины.

*Недетерминированная машина Тьюринга* *отличается от обычной (детерминированной) следующими правилами*:

* для каждой пары (состояние, символ) определены несколько переходов ,
* НМТ выбирает любой переход ,
* вход w допускается НМТ, если хотя бы одна последовательность переходов приводит НМТ к допускающему состоянию.

Недетерминированные машины Тьюринга решают то же множество проблем, что и детерминированные. Во-первых, из определения видно, что ДМТ является частным случаем НМТ. Кроме того, можно (хотя и технически сложно) показать, что работу любой НМТ можно эмулировать на некоторой ДМТ. То есть для любой НМТ можно сконструировать некоторую ДМТ, которая допускает в точности то же множество входных цепочек.

Принципиальная сложность, встающая при этом на пути, такова: соответствующая ДМТ может работать за экспоненциально большее время. Эту сложность человечество на данный момент так и не научилось обходить.

## Проблема = язык

Вскоре мы введем некоторые классы проблем, позволяющие провести границу между экспоненциальным временем решения и полиномиальным. Выберем для этого иную формулировку.

Рассмотрим какую-нибудь проблему, например «является ли данный граф связным?». Решить эту проблему — значит, привести алгоритм, принимающий на вход некоторую запись графа, и отвечающий «да» или «нет». Но действие алгоритма можно сформулировать и иначе: будем говорить, что данный алгоритм принимает язык записей связных графов.

Итак, *пусть проблема имеет вид*:

* Принадлежит ли объект X множеству Y ?
* Обладает ли объект X свойством Y ?

*Будем формулировать проблему так*:

* Принадлежит ли запись объекта X языку Y ?

## Классы P и NP

### Определение

*Язык* L *принадлежит классу P, если существует ДМТ* M*, которая допускает язык* L *и останавливается за полиномиальное время.*

### Определение

*Язык* L *принадлежит классу NP, если существует НМТ* M*, которая допускает язык* L *и останавливается за полиномиальное время.*

Приведем пример языка, принадлежащего классу *P* :

* Язык записей графов, в которых есть *эйлеров цикл*.

Соответствующая ДМТ несложна: она проверяет связность данного графа, и проверяет, что степени всех вершин — четные . Если оба условия выполнены , ДМТ возвращает ответ «да».

Приведем пример языка , принадлежащего классу *NP*:

* Язык записей графов, в которых есть *гамильтонов цикл*.

Соответствующую НМТ тоже придумать несложно. НМТ угадывает порядок обхода вершин в *гамильтоновом цикле*, после чего проверяет, все ли соседние вершины (то есть идущие подряд в угаданном обходе) соединены ребром. Если хотя бы один порядок обхода является *гамильтоновым циклом*, то НМТ возвращает ответ «да», в противном случае, действительно, ответ: «нет».

Поскольку всякая ДМТ является частным случаем НМТ, ясно, что

.

Важнейшим неразрешенным вопросом для *Computer Science* до сих является: верно ли, что

*P* ≠ *NP.*

## Полиномиальное сведeние

Предположим, что некоторый язык P1 не принадлежит классу *P*. То есть P1 — это некоторая заведомо сложная проблема.

Предположим также, что имеется некоторый язык P2 , и все попытки построить алгоритм , решающий P2 , не увенчались успехом. В таком случае, возникает предположение , что P2 не принадлежит *P*. Для доказательства этого предположения используем метод полиномиального свед**e**ния:

* Найдем полиномиальный алгоритм , строящий по экземпляру P1 экземпляр P2 с таким же ответом ( сведем P1 к P2 ).

После этого приведем доказательство от противного:

Пусть P2 принадлежит *P*. Тогда, по определению, существует ДМТ M, допускающая язык P2. Зная это , построим следующую ДМТ M:



Рис. 4.1

Эта ДМТ допускает язык P1. Язык P1 оказывается принадлежащим классу *P*. Получили противоречие , следовательно , P2 не принадлежит P, что и требовалось доказать.

## Классы труднорешаемых проблем

### Определение

Проблема P называется *NP*-полной, если P принадлежит *NP* и для всякого L из *NP* существует полиномиальное сведение L к P.

### Определение

Проблема P называется *NP*-трудной, если для всякого L из *NP* существует полиномиальное сведение L к P. Взаимоотношение между введенными классами сложности изображено на рис. 4.2.

Рис. 4.2

## Проблема SAT

Приведем пример очень важной для теории сложности *NP*–полной задачи. В англоязычной литературе для нее принято сокращение *SAT* (от англ. *satisfiability* — выполнимость), в русскоязычной, соответственно, ВЫП.

Пусть дана формула, содержащая булевы переменные, операторы «» («и»), «» («или») и «¬» («не»), и скобки. Формула называется выполнимой, если существует такой набор значений переменных, при подстановке которого формула принимает значение «**истина»**.

*SAT* — это язык выполнимых формул. В 1971 году Стивен Кук доказал *NP*–полноту задачи *SAT*.

### Теорема (Кук)

Проблема *SAT NP*-полна.

### Доказательство

Докажем сначала, что *SAT*  *NP*.

Для этого достаточно привести недетерминированную машину Тьюринга, которая принимает язык *SAT*. Простейшая НМТ действует так:

* Угадать значения всех переменных.
* Подставить эти значения в формулу.
* Если формула приняла значение «**истина»**, перейти в допускающее состояние.

Видно, что эта НМТ допускает формулу w тогда и только тогда, когда существует подстановка значений, выполняющая формулу w.

Рассмотрим любой язык L из *NP*. Необходимо теперь доказать, что существует полиномиальное сведѐние L к *SAT*, то есть по экземпляру L построить формулу для *SAT* с тем же ответом. При этом, поскольку L принадлежит *NP*, существует НМТ M, которая допускает язык L, совершая не более p (n) переходов, где p (n) — некоторый полином.

Искомая формула имеет следующую структуру:

( М начинает, имея на входе данный экземпляр L ) 
( Первый шаг M происходит по правилам ) 
(…) 
( Шаг M номер p(n) происходит по правилам ) 
( М пришло в допускающее состояние ).

К сожалению, технические детали этой формулы весьма громоздки, поэтому ограничимся лишь приведенным выше каркасом.

Предположим, что рассматриваемый экземпляр действительно принадлежит L. Значит, машина Тьюринга M , получив на вход этот экземпляр, может при определенном выборе переходов прийти в допускающее состояние. Тогда формула, описывающая процесс работы M, должна выполняться, поскольку все утверждения про работу M, закодированные в ней, верны.

И наоборот, если формула выполнима, то, анализируя значения переменных в каждой скобке, можно понять, какой переход был выбран на каждом шаге, и убедиться, что машина Тьюринга проделывала только разрешенные переходы и пришла в итоге в допускающее состояние, а значит, входной экземпляр принадлежит языку L.

## Проблема 3-SAT

Если некоторая проблема является *NP*–полной, то любое ее обобщение тем более является *NP*–полной проблемой. В то же время, иногда частный случай *NP*–полной проблемы может сам по себе обладать *NP*–полнотой. Важный пример — задача 3-*SAT*. Это задача о выполнимости некоторого класса булевых формул, но не столь общего, как в задаче *SAT*. Для определения 3-*SAT* введем несколько простых определений.

Литералом *называется выражение вида* «x» или «¬x», где x — переменная.

3-дизъюнктом *называется выражение вида* «a b c», где a, b и c — литералы.

*Булева формула записана в* 3-конъюнктивной нормальной форме, если *она является логическим «И» произвольного количества 3-дизъюнктов.* Например:

(x1 ¬x2 x3) (¬x1 x2 ¬x4) (x2 ¬x3 x4)

Проблема 3-*SAT* формулируется так: является ли данная формула, записанная в 3-конъюнктивной нормальной форме, выполнимой.

Проблема 3-*SAT* также *NP*–полна, что чрезвычайно удобно использовать при доказательстве *NP*–полноты других задач.

## Проблема IS

Следующая *NP*–полная проблема относится к теории графов. Сокращение *IS* расшифровывается как *Independent Set* — Независимое Множество, сокращенно — НМ.

*Множество вершин графа называется* независимым, *если никакие две вершины из этого множества не соединены ребром.*

Дан граф G и число k. Проблема *IS* такова: Существует ли в графе G независимое множество из k вершин ?

Докажем *NP*–полноту проблемы *IS*.

Для начала убедимся, что *IS*  *NP*. Для этого представим недетерминированный алгоритм, допускающий язык *IS*. Его суть такова:

* угадать про каждую вершину, принадлежит ли она искомому множеству ;
* убедиться, что выбрано ровно k вершин ;
* проверить, что нет двух выбранных вершин, соединенных ребром;
* если все проверки закончились удовлетворительно, перейти в допускающее состояние.

Теперь сведем 3-*SAT* к *IS*, чем докажем *NP*-полноту *IS*. Для формулы в 3-конъюнктивной нормальной форме приведем граф и число, принадлежащие языку *IS* тогда и только тогда, когда формула принадлежит языку 3-*SAT*, то есть является выполнимой.

Каждому дизъюнкту в булевой формуле (например, ( x1 ¬x 2 x 3 ) ) поставим в соответствие 3 смежные вершины, по одной вершине на литерал. Тем самым мы потребуем, чтобы в решении задачи про независимое множество из этих вершин было выбрано не более одной.

Вершину, соответствующую литералу x i в одном дизъюнкте , и вершину, соответствующую литералу ¬x i в другом , соединим ребром. Тем самым мы удостоверимся, что в разных дизъюнктах не будут выбраны противоречащие друг другу литералы.

Положим k = количество дизъюнктов в формуле.

*Поясним приведенные выше манипуляции на примере*. Так выглядят формула и соответствующий ей граф (**k = 3**)(см. рис. 4.3):

(x1¬x2x3)(¬x1x2x4)(¬x2¬x3¬x4)

 Рис. 4.3

Вершины независимого множества соответствуют истинным литералам в формуле. В каждом дизъюнкте обязан оказаться хотя бы один истинный литерал (ведь k равно количеству дизъюнктов) и противоречий возникнуть не может благодаря ребрам, проведенным между дизъюнктами.

## Проблема 3-COL

*Еще одна красивая* *NP*–*полная задача из теории графов относится к раскраске графа* или определению хроматического числа графа.

Дан граф G. Можно ли раскрасить вершины G в три цвета так, чтобы не было двух смежных вершин, раскрашенных в один цвет?

Проблема 3-*COL* *NP*–полна, как и любая проблема k-*COL* при условии

k ≥ 3.

## Проблема ILP

*Следующая NP–полная задача имеет алгебраический вид*. Сокращение *ILP* расшифровывается как *Integer Linear Programming* — Целочисленное Линейное Программирование.

Дан набор ограничений вида

.

При этом a i и c — целые числа.

Существует ли целочисленный вектор x, удовлетворяющий всем неравенствам ?

## Класс проблем co-NP

В определении недетерминированной машины Тьюринга мы договорились, что НМТ говорит «**да**», если хотя бы одна последовательность переходов приводит к допускающему состоянию. Таким образом, НМТ говорит «**нет**», если все последовательности переходов приводят к недопускающим состояниям. Видна определенная асимметрия между ответами «**да**» и «**нет**». Как выясняется, эта асимметрия принципиальна.

Дополнение **языка** **L** — *это множество цепочек, не принадлежащих языку* L.

### Определение

*Язык* L *принадлежит классу co-NP, если его дополнение принадлежит классу NP.*

Например, язык **не**выполнимых формул (всегда принимающих значение ложь) принадлежит, как несложно понять, классу co-*NP*. Кроме того, если рассмотреть невыполнимую формулу, заключить ее в скобки и приписать «**не**», то получиться формула, всегда принимающая значение истина, и наоборот. Такие формулы называются *тавтологиями*, и мы только что убедились, что язык тавтологий *TAUT* принадлежит классу co-*NP*.

На данный момент в *Computer Science* открытым остается следующий вопрос: верно ли, что

*NP* ≠ co-*NP.*

Возможно, для ответа на этот вопрос, удобно будет применить следующую теорему.

### Теорема

*NP = сo-NP тогда и только тогда, когда существует NP -полная задача, принадлежащая co-NP.*

В теории вычислительной сложности открыты многие вопросы о равенстве определенных классов задач. Общепринятая позиция во многих случаях такова: предполагается неравенство соответствующих классов задач, и дальнейшие рассуждения ведутся на основе этого предположения. Если же в какой-то момент будет доказано обратное, теорию вычислительной сложности придется развивать заново. Если поверить в предположения, говорящие что *P* ≠ *NP* ≠ co-*NP*, тогда классы, рассмотренные в данном пособии, соотносятся между собой следующим образом (см. рис. 4.4) :



Рис. 4.4

**БИБЛИОГРАФИЧЕСКИЙ СПИСОК**

*1. Кормен*, *Т*. Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. - М.: Вильямс, 2007. 1296 с.

*2. Окулов*, *С.М.* Программирование в алгоритмах / С.М. Окулов. - М.: Бином, 2007. 341с.

 *3. Седжвик, Р*. Фундаментальные алгоритмы на С++ / Р. Седжвик. - М.: Вильямс, 2011. 1056 с.

 *4. Новиков, Ф.А.* Дискретная математика для программистов / Ф.А. Новиков. - СПб.: Питер, 2012. 304 с.

**СОДЕРЖАНИЕ**

**ВВЕДЕНИЕ**…………………..………………………………………………….…3

**Тема 1**. **АЛГОРИТМЫ НА ГРАФАХ**……..……………………………6

**Рассматриваемые темы с указанием литературы**………6

Понятие графа. Поиск в глубину. Поиск в ширину. Эйлеровы и Гамильтоновы циклы. Алгоритмы поиска кратчайших путей. Алгоритм Беллмана-Форда. Алгоритм Дейкстры. Алгоритм Флойда-Варшалла. Минимальные остовные деревья. Алгоритмы Крускала и Прима. Потоки в сетях. Алгоритм Форда-Фалкерсона. Алгоритм Эдмондса-Карпа.

**Тема 2**. **АЛГОРИТМЫ КОМБИНАТОРНОГО ПЕРЕБОРА**………..25

**Рассматриваемые темы с указанием литературы**………..26

Основные комбинаторные объекты. Кортеж, перестановки, сочетания, различия. Генерация кортежей. Коды Грея. Генерация перестановок. Генерация сочетаний. Генерация размещений. Генерация разбиений. Генерация деревьев.

**Тема 3**. **ОБЩИЕ МЕТОДЫ РАЗРАБОТКИ АЛГОРИТМОВ**………48

**Рассматриваемые темы с указанием литературы**………..48

Динамическое программирование. «Жадные» алгоритмы. Перебор с возвратом. Алгоритмы «Разделяй и властвуй». Амортизационный анализ. Эвристические алгоритмы.

**Тема 4. ТЕОРИЯ СЛОЖНОСТИ АЛГОРИТМОВ**…………………...62

**Рассматриваемые темы с указанием литературы**………..62

Понятие сложности алгоритма. Классы сложности. Полиномиальные и экспоненциальные алгоритмы. Класс *NP. NP* – полные алгоритмы. Примеры *NP* – полных задач. Проблема *P* < > *NP.*

**БИБЛИОГРАФИЧЕСКИЙ СПИСОК**……………………………………75

ШУТОВ Антон Владимирович

МЕДВЕДЕВ Юрий Алексеевич

СТРУКТУРЫ И АЛГОРИТМЫ КОМПЬЮТЕРНОЙ ОБРАБОТКИ ДАННЫХ

Практикум

Издается в авторской редакции

Подписано в печать Формат 60х84 1/16

Усл. печ. л. 4,75 Уч. изд.л.- 4.95

Заказ № 08-13 Тираж 50 экз.

Отпечатано в отделе оперативной полиграфии ВлГУ

600014 г. Владимир, ул. Университетская, д. 2

Телефон: (4922) 33-87-40