

Н.И. Костюкова

Комбинаторные алгоритмы для программистов



ИНТУИТ

НАЦИОНАЛЬНЫЙ ОТКРЫТЫЙ УНИВЕРСИТЕТ

Костюкова Н.И.

Комбинаторные алгоритмы для программистов

2-е издание, исправленное

Костюкова Н.И.

Национальный Открытый Университет "ИНТУИТ"

2016

УДК 519.1(07)

ББК 16

К72

Графы и их применение. Комбинаторные алгоритмы для программистов / Костюкова Н.И. - М.: Национальный Открытый Университет "ИНТУИТ", 2016 (Основы информационных технологий)

ISBN 978-5-9556-0069-7

Курс начинается с азов комбинаторики и охватывает все основные алгоритмы, их анализ и реализацию на языках программирования, а так же рассматриваются алгоритмы на графах с точки зрения комбинаторных методов их реализации и анализа.

Курс описывает различные способы представлений конечных последовательностей и операций над ними; множества и мультимножества; производящие функции и рекуррентные соотношения; абстрактные структуры данных; алгоритмы рекуррентных соотношений; комбинаторные задачи теории информации; алгоритмы на абстрактных структурах данных; различные типы поисков (последовательный, логарифмический в статических и динамических таблицах, бинарный, по сбалансированным сильно ветвящимся деревьям); все виды сортировок (внутренняя, вставка, обменная сортировка, выбор, распределяющая сортировка, цифровая распределяющая сортировка, частичная сортировка-выбор, частичная сортировка-слияние); алгоритмы на графах Дейкстры и алгоритм Флойда. В конце курса приводится программная реализация на языках программирования Паскаль, Си, С++ классических комбинаторных алгоритмов.

(с) ООО "ИНТУИТ.РУ", 2007-2016

(с) Костюкова Н.И., 2007-2016

Комбинаторные вычисления

Введение. Проблема представления: коды, сохраняющие разности. Классы алгоритмов. Анализ алгоритмов. Программа.

Введение

Комбинаторная математика является старой дисциплиной. Она получила свое наименование в 1666 г. от Лейбница в его "Dissertation de Arte Combinatori". Комбинаторные алгоритмы с их акцентом на разработку, анализ и реализацию практических алгоритмов являются продуктом века вычислительных машин.

Предмет теории комбинаторных алгоритмов - вычисления на дискретных математических структурах. Это новое направление исследований. Лишь в последние несколько лет из наборов искусных приемов и разрозненных алгоритмов сформировалась система знаний о разработке, реализации и анализе алгоритмов.

Комбинаторные вычисления находятся в таком же отношении к комбинаторной математике (дискретной, конечной математике), как численные методы анализа - к анализу. Комбинаторные вычисления развиваются в следующем направлении:

- интенсивно изобретаются новые алгоритмы;
- происходит быстрый прогресс (главным образом в математическом плане) в понимании алгоритмов, их разработки и анализа;
- происходит переход от изучения отдельных алгоритмов к исследованию свойств, присущих классам алгоритмов.

В отличие от некоторых других разделов математики, комбинаторные вычисления не имеют "ядра", то есть некоторого количества "фундаментальных теорем", составляющих суть предмета, из которых выводится большинство результатов. Сначала может показаться, что в целом эта область состоит из наборов специальных методов и хитрых приемов. Однако после того, как было исследовано достаточно много комбинаторных алгоритмов, стали вырисовываться некоторые общие

принципы. Именно эти принципы делают комбинаторные вычисления связной областью знаний и позволяют изложить ее в систематизированном виде.

Проблема представления: коды, сохраняющие разности

Чрезвычайно важной проблемой в комбинаторных вычислениях является задача эффективного представления объектов, подлежащих обработке. Она возникает потому, что обычно имеется много возможных способов представления сложных объектов более простыми структурами, которые можно заложить в языки программирования, но не все такие представления в одинаковой степени эффективны с точки зрения времени и памяти. Более того, идеальное представление зависит от вида производимых операций.

Приведем примеры, в которых задаются весьма специфические операции над целыми. Целые определяются как данные простейшего типа почти во всех вычислительных устройствах и языках программирования. Таким образом, проблема представления, как правило, не возникает. Имеющееся представление почти всегда наилучшее. Однако существуют некоторые заслуживающие внимания исключения, когда выгодно или даже необходимо использовать представление целых в вычислительном устройстве иным способом. Эти исключения появляются в следующих случаях:

1. Необходимы целые, больше имеющихся непосредственно в аппаратном оборудовании.
2. Необходимы только небольшие целые, и требуется сэкономить память, упаковывая их по несколько в одну ячейку.
3. Действия с целыми производятся не общепринятыми арифметическими операциями.
4. Целые используются для представления других типов объектов, и необходимо иметь возможность легко обращаться целое в соответствующий ему объект и обратно.

Проблемы кодов, сохраняющих разности, касаются случаев 2 и 3. В задачах распознавания образов и классификации для решения вопроса, будут ли два объекта X, Y эквивалентными, стандартной является

следующая процедура. X, Y представляются векторами признаков $(x_1, x_2, \dots, x_f), (y_1, y_2, \dots, y_f)$ соответственно, где каждая компонента означает признак объекта, выраженный целым значением. Считается, что X, Y эквиваленты тогда и только тогда, если

$$\sum_{i=1}^f |x_i - y_i| \leq t,$$

где t — целое, называемое порогом.

Классы алгоритмов

Одной из причин быстрого прогресса комбинаторных вычислений является усиление внимания к исследованию классов алгоритмов в противоположность изучению отдельных из них. Для того чтобы утверждать, например, что "все алгоритмы, предназначенные для выполнения того-то и того-то, должны обладать такими-то и такими-то свойствами" или "не существует алгоритма, удовлетворяющего тому-то и тому-то", необходимо иметь дело с четко определенным классом алгоритмов. Именно при таком определении становится возможным говорить, что данный алгоритм является оптимальным по отношению к некоторому свойству, если он работает по крайней мере так же хорошо (относительно этого свойства), как любой другой алгоритм из рассматриваемого класса.

Как можно строго определить, возможно, бесконечный, класс алгоритмов? Исследуем этот вопрос на примере задачи о фальшивой монете. Рассматриваемый в этом примере класс алгоритмов порождает более обширный и более важный класс алгоритмов - так называемые деревья решений.

Задача. Имеется n монет, о которых известно, что $n - 1$ из них являются настоящими и не более чем одна монета, фальшивая (легче или тяжелее остальных монет). Дополнительно к группе из n сомнительных монет дается еще одна монета, причем заведомо известно, что она настоящая. Имеются также весы, с помощью которых можно сравнить общий вес любых m монет с общим весом любых

других m монет и тем самым установить, имеют ли две группы по m монет одинаковый вес либо одна из групп легче другой. Задача состоит в том, чтобы найти фальшивую монету, если она есть, за наименьшее число взвешиваний, или сравнений.

Решение Пусть сомнительные монеты занумерованы числами $1, 2, \dots, n$. Монете, о которой известно, что она настоящая, поставим в соответствие номер 0. Пусть $\{0, 1, 2, \dots, n\}$ - множество монет. Если S_1, S_2 — непересекающиеся непустые подмножества множества S , то через $S_1 : S_2$ обозначим операцию сравнения весов множества S_1, S_2 . При сравнении возможны три исхода, которые обозначим следующим образом: $S_1 < S_2, S_1 = S_2, S_1 > S_2$ в зависимости от того, является ли вес S_1 меньшим, равным или большим веса S_2 .

Рассматриваемые алгоритмы можно представить в форме дерева решений.

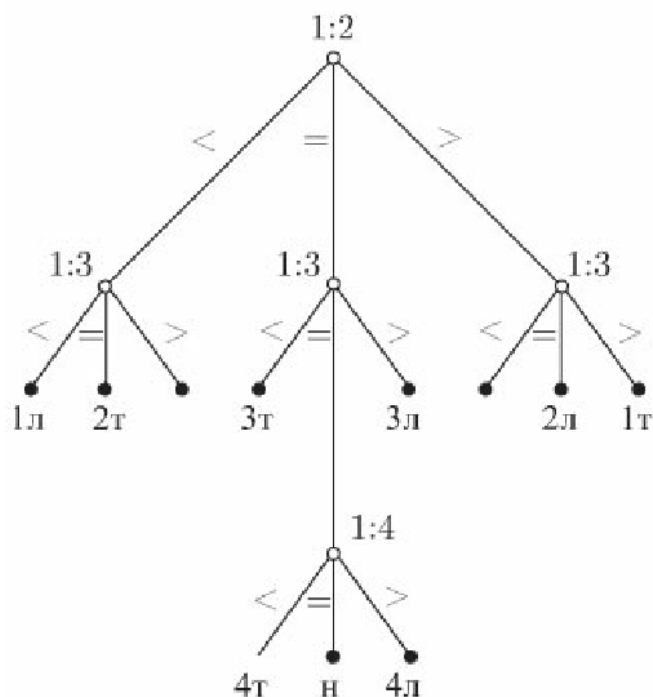


Рис. 1.1. Дерево решений для задачи о фальшивой монете с четырьмя

Корень дерева на рисунке изображен полой окружностью и помечен отношением $1 : 2$. Это означает, что алгоритм начинает работу сравнением весов монет с номерами 1 и 2. Три исходящие из корня ветви ведут к поддеревьям, определяющим продолжение работы алгоритма после каждого из трех возможных исходов первого сравнения. Окружности, залитые черной краской, называются листьями дерева и означают, что работа алгоритма заканчивается. Метки соответствуют исходам: "1л" - монета 1 легкая, "1т" - монета 1 тяжелая, "н" - все монеты настоящие. Непомеченная вершина дерева означает, что при наших предположениях этот случай возникнуть не может.

Алгоритм, приведенный на [рис. 1.1](#), требует двух сравнений в одних случаях и трех - в других. Скажем, что он требует "трех сравнений в худшем случае". Обычно важно знать, сколько работы требует алгоритм в среднем, однако для этого требуется задать вероятности различных исходов. Если предположим, что все девять исходов 1л, 1т, 2л, 2т, 3л, 3т, 4л, 4т, - равновероятны, то тогда этот алгоритм требует в среднем $7/3$ сравнений.

На одну чашку весов можем положить больше одной монеты. Например, можно начать сравнения, положив на одну чашку весов монеты 1 и 2, а на другую - монеты 3 и 4 ([рис. 1.2](#)).

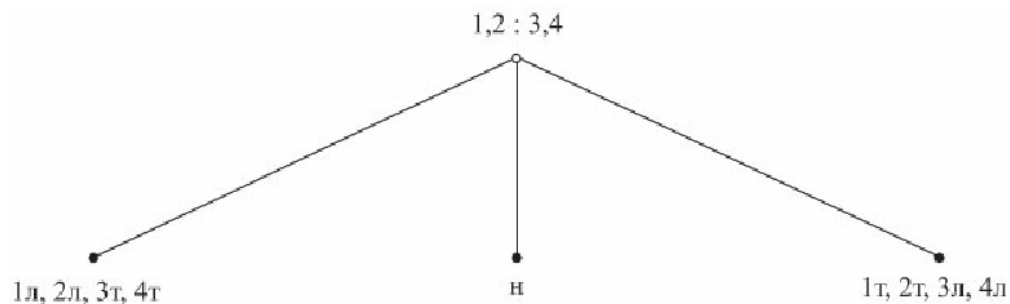


Рис. 1.2. Корень другого дерева решений для задачи о четырех монетах

Если посчастливится, задачу можно решить за одно сравнение - это может произойти, когда все монеты настоящие. Независимо от того, как дополняется это дерево решений, в худшем случае задача все равно потребует тех же трех сравнений, поскольку единственное тернарное

решение не может идентифицировать один из четырех исходов, которые возможны на ветви, помеченной символом " $<$ ", так же как и один из четырех исходов на ветви, помеченной символом " $>$ ". К тому же, независимо от того, как дополняется это дерево решений, оно потребует в среднем по крайней мере $7/3$ сравнений, и в этом случае оно не лучше, чем дерево на [рис.1.1](#).

Используя монету 0, о которой известно, что она настоящая, можно получить приведенное на [рис. 1.3](#) дерево решений (полное двухъярусное тернарное дерево), которое и в худшем, и в среднем случае требует двух сравнений.

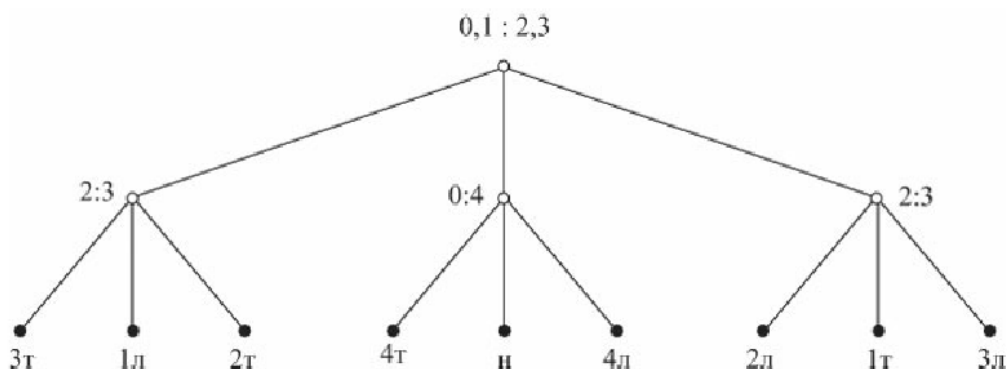


Рис. 1.3. Оптимальное дерево решений для задачи о четырех монетах

Рассматриваемый класс алгоритмов решения задачи о фальшивой монете есть множество тернарных деревьев решений (примеры на [рис.1.1](#), [рис.1.2](#), [рис.1.3](#)), обладающих следующими свойствами:

- каждый узел помечен сравнением $S_1 : S_2$, где S_1 и S_2 - непересекающиеся непустые подмножества множества $S = \{0, 1, 2, \dots, n\}$ всех монет;
- каждый лист либо не помечен, что соответствует невозможному исходу в предположении существования не более чем одной фальшивой монеты, либо помечен одним из исходов $л$, $Г$, $н$, означающим соответственно, что все монеты настоящие. Четко определив подлежащий дальнейшему рассмотрению класс алгоритмов, можно исследовать свойства, которыми должно обладать каждое дерево из этого класса, и определить, как найти

алгоритмы, являющиеся в некотором смысле оптимальными. Решим эту проблему в начале для четырех монет, а затем перейдем к общему случаю.

Поскольку в задаче о четырех монетах требуется различить девять возможных исходов, любое дерево решений для этой задачи должно иметь, по крайней мере, девять листьев и, следовательно, не менее двух ярусов. Поэтому дерево на [рис.1.3](#) является оптимальным и для худшего случая, и для среднего. Существуют ли другие оптимальные деревья? Для ответа на этот вопрос нужно рассмотреть множество всех деревьев решений для задачи о четырех монетах. Попытаемся исключить из дальнейшего рассмотрения какую-либо часть этого множества. Прежде всего видно, что путем любой перестановки множества $\{1, 2, \dots, n\}$ сомнительных монет из одного дерева, приведенного на [рис.1.3](#), можно получить другие оптимальные деревья. Все они будут изоморфны дереву на [рис.1.3](#). Исходя из этого, уточним постановку задачи и будем интересоваться попарно неизоморфными деревьями.

Рассмотрим затем, существует ли оптимальное дерево среди тех, у которых в корне не используется монета с номером n . При таком ограничении в корне дерева можно сделать только два различных сравнения, а именно $1 : 2$ и $1, 2 : 3, 4$. Рассмотрим разбиение исходов по трем ветвям, выходящим из корня, как показано на [рис.1.2](#). Для получения такого, как на [рис.1.3](#), полного двухъярусного тернарного дерева, девять возможных исходов должны были бы быть разбиты в отношении $(3, 3, 3)$. Они же вместо этого разбиваются, соответственно, в отношении $(2, 5, 2)$ и $(4, 1, 4)$. Таким образом, заключаем, что задачу для четырех монет нельзя решить за два сравнения, не используя дополнительную настоящую монету.

Наконец, рассмотрим те деревья решений, которые используют монету 0 в корне. В этом случае видно, что в корне фактически возможны только два сравнения: $(0 : 1)$ и $0, 1 : 2, 3$. Для первого сравнения набор исходов будет $(1, 7, 1)$, в связи с чем все алгоритмы, начинающиеся таким способом, для нас непригодны. Набор исходов $(3, 3, 3)$ приводит к оптимальному дереву, показанному на [рис.1.3](#). Аналогичным образом устанавливается, что для оптимального дерева сравнения в первом от корня ярусе определяются единственным образом. Отсюда заключаем,

что для задачи о четырех монетах фактически существует только одно оптимальное дерево.

Когда используемые идеи анализа задачи о четырех монетах переносятся на произвольный случай, в некоторой степени все идеи обобщаются на случай любого числа монет. Однако некоторые из них не имеют практического значения, когда n значительно больше четырех. В принципе, оптимальные деревья решений всегда можно найти путем систематического поиска в множестве деревьев, поскольку для любого заданного n в качестве кандидатов требуется рассмотреть лишь конечное число деревьев решений. В лекции обсуждается техника исчерпывающего поиска в таких конечных множествах. Однако, если даже поиск организован разумно и рассматриваются лишь существенно различные (не изоморфные) деревья, эта процедура не может служить практическим способом отыскания оптимальных деревьев решений. С ростом n число деревьев растет экспоненциально, и поэтому техника исчерпывающего поиска имеет практическое значение только для малых значений n .

Поскольку число листьев в дереве решений должно быть по крайней мере таким же, как и число возможных исходов задачи ($2^n + 1$ для задачи о монетах), сразу же можно получить нижнюю оценку необходимого числа сравнений (или, что эквивалентно, верхнюю оценку числа монет для данного сравнения).

Анализ алгоритмов

В процессе разработки и реализации алгоритма естественным образом раскрываются некоторые его свойства. По мере того как алгоритмы становятся все более и более сложными, все менее и менее вероятно, что их важные свойства проявятся на стадиях разработки и реализации. Как правило, некоторые важные аспекты поведения алгоритма, такие как его корректность, необходимое число операций или объем памяти, определить трудно. Поэтому обычно глубокое понимание нового алгоритма предваряется очень длинной стадией его анализа.

Из-за трудностей анализа им зачастую просто пренебрегают. Вместо этого программа выполняется для того, чтобы увидеть, что получается (например, измеряется время работы). Такой подход можно признать

удовлетворительным, если есть основание полагать, что тестовые задачи достаточно хорошо характеризуют работу алгоритма в общем случае; если же это не так, то описанный подход даст мало ценной информации. Даже если тест прекрасно характеризует работу алгоритма, он никогда не даст ответ на придирчивый вопрос, могут ли существовать лучшие алгоритмы для решения той же самой задачи. Проблему оптимальности алгоритма можно решить только путем его анализа.

В анализе алгоритмов существуют две фундаментальные проблемы:

- Какими свойствами обладает данный алгоритм?
- Какие свойства должен иметь любой алгоритм, решающий данную проблему?

Фундаментальная разница между этими двумя вопросами состоит в подходе к ответу на них. В первом случае алгоритм задан и заключения выводятся путем изучения свойств, присущих ему. Во втором случае задается проблема и точно определяется структура алгоритма. Заключения выводятся на основе изучения существа проблемы по отношению к данному классу алгоритмов.

Программа

Программа 1. Поиск фальшивой монеты

```
//Монета ищется простым перебором.  
//Алгоритм реализован на языке программирования Turbo-C++.  
#include <stdlib.h>  
#include <conio.h>  
#include <stdio.h>  
int main() {  
    clrscr();  
    int k, n, flag = -1;  
    int *mas;  
    printf("Введите число монет: ");  
    scanf("%i", &n);  
    mas = (int*)malloc(sizeof(n));
```



```
randomize();
k = random (10) + 1;
for (int i = 0; i < n; i++) {
    mas[i] = k;
}
for (i = k; i == k; i = random (20));
mas[random (n)] = i;
printf("Масса монет: ");
for (i = 0; i < n; i++) {
    printf("%5i", mas[i]);
    if (mas[i] != k) {
        flag = i;
    }
}
printf("\nФальшивая монета под номером %i ее вес %i", flag+1,
mas[flag]);
getch();
free(mas);
return 0;
}
```

Целые и последовательности (последовательное распределение)

Введение. Целые. Последовательности. Различные способы представлений конечных последовательностей (или начальных сегментов бесконечных последовательностей) и операции над ними.

Введение

Большинство вычислительных устройств в качестве основных объектов допускает только двоичные наборы, целые и символы, поэтому, прежде чем работать с более сложными объектами, их необходимо представить двоичными наборами, целыми или символами. Например, числа с плавающей запятой кодируются целыми - мантиссой и порядком этого числа, но такое кодирование обычно незаметно для пользователя. В противоположность этому, рассмотренные во второй и третьей лекции способы кодирования объектов (множества, последовательности, деревья) всегда адресованы пользователю.

Любой заданный класс объектов может иметь несколько возможных представлений, и выбор наилучшего из них решающим образом зависит от того, каким образом объект будет использован, а также от типа производимых над ним операций. Поэтому рассмотрим не только свойства самих представлений, но также и некоторые приложения.

Целые

Целые являются основными объектами в вычислительной комбинаторике. В различных вычислительных теоретико-числовых исследованиях изучаются сами целые числа, но мы будем использовать их главным образом при подсчете и индексировании. В последнее время установлено, что полезны различные представления. В этой лекции обсудим общий класс позиционных представлений.

Мы будем рассматривать только неотрицательные целые. Кроме того, к любому представлению неотрицательных целых легко присоединить одиночный знаковый двоичный разряд.

Позиционные системы для представления целых чисел очень широко известны, поскольку они встречаются во многих разделах математики, начиная с "новой математики" и кончая углубленным курсом теории чисел. В системе счисления с основанием r каждое положительное целое число имеет единственное представление в виде конечной последовательности

$$(d_0, d_1, d_2, d_3, \dots, d_k), \quad (2.1)$$

в которой каждое d_i - целое, удовлетворяющее условию $0 \leq d_i < r$ и $d_k \neq 0$. Нуль представляется последовательностью (0) . r называется основанием системы ($r > 1$). Целое, соответствующее последовательности (2.1), имеет вид

$$N = d_0 + d_1 r + d_2 r^2 + d_3 r^3 + \dots + d_k r^k,$$

что принято выражать следующим образом:

$$N = (d_k d_{k-1} \dots d_1 d_0)_r.$$

На протяжении истории использовались различные значения r . Например, древние вавилоняне использовали $r = 60$, а индейцы племени Майя — $r = 20$. Сегодня наиболее широко используется $r = 10$ - десятичная система, которую мы унаследовали от арабов, и $r = 2$ - двоичная система, которая лежит в основе современных вычислительных устройств. В действительности она применяется лишь на самом низком уровне аппаратного оборудования; в сложных вычислительных устройствах и базисных языках удобнее использовать $r = 8$ или $r = 16$. Единственность этого представления можно доказать методом от противного. Числа $N = 0$ и $N = 1$, очевидно, имеют единственное представление. Предположим, что представление не единственно, и пусть $N > 1$ будет наименьшим целым числом, имеющим два различных представления:

$$N = (d_k d_{k-1} \dots d_0)_r = (e_l e_{l-1} \dots e_0)_r.$$

Если $k \neq l$, то без потери общности предположим, что $k > l$. Тогда,

поскольку

$$\sum_{i=0}^l (r-1)r^i = r^{l+1} - 1 < r^{l+1} < r^k$$

и поскольку $d_k \neq 0$, мы заключаем, что

$$(d_k d_{k-1} \dots d_0)_r > (e_l e_{l-1} \dots e_0)_r, \quad (2.2)$$

что невозможно. Таким образом, мы должны иметь $k = l$. Аналогично, если $d_k > e_k$, мы имели бы снова неравенство (2.2) и отсюда с необходимостью $d_k = e_k$. Следовательно, число

$$N - d_k r^k = (d_{k-1} d_{k-2}, \dots, d_0 = (e_{k-1} e_{k-2} \dots e_0)_r$$

имеет два различных представления, что противоречит предположению, что N - наименьшее из таких чисел.

Для доказательства того, что каждое положительное целое имеет представление по основанию r , достаточно задать алгоритм, конструирующий (с необходимостью единственное) представление данного числа N .

Алгоритм 1. Преобразование числа N в его представление $(d_k d_{k-1} \dots d_1 d_0)_r$ в системе счисления с основанием r .

Он строит последовательность $d_0, d_1, d_2, \dots, d_k$ путем повторения деления на r и записи остатков. Пусть на первом шаге при делении N на r остаток будет d_0 . Частное, полученное в результате первого шага, делим на r , вновь полученное частное делим на r и так далее. Полученная в результате такого процесса последовательность остатков и будет требуемым представлением N по основанию r .

Важным обобщением систем счисления с основанием r являются смешанные системы счисления, в которых задается не единственное основание r , а последовательность оснований $r_0, r_1, r_2, \dots$, и последовательность (2.2) соответствует целому

$$N = d_0 + d_1 r_0 + d_2 r_0 r_1 + d_3 r_0 r_1 r_2 + \dots + d_k \prod_{i=0}^{k-1} r_i,$$

где теперь каждое d_i удовлетворяет неравенству $0 \leq d_i < r_i$ и $d_k \neq 0$, если $N \neq 0$ - тот факт, что каждая такая последовательность соответствует единственному числу и каждое положительное целое число имеет единственное представление, следует из простого обобщения результатов для обычных систем счисления, которые являются частным случаем смешанных систем при $r_i = r, i \geq 0$.

Смешанные системы счисления могут вначале показаться странными, но в действительности в повседневной жизни они встречаются почти так же часто, как и десятичные.

Пример. Рассмотрим нашу систему измерения времени: секунды, минуты, часы, дни недели и годы. Это - в точности смешанная система с $r_0 = 60, r_1 = 60, r_2 = 24, r_3 = 7, r_4 = 52$.

Представление целого N в смешанной системе счисления $(r_0, r_1, \dots)$ осуществляется с помощью алгоритма 2, который является простым обобщением алгоритма 1. Вместо того, чтобы для получения d_k в качестве делителя всегда использовалась r , в алгоритме 2 используется r_k .

Алгоритм 2. Преобразование числа N в его представление $(d_0, d_1, \dots, d_k)$ в смешанной системе счисления $(r_0, r_1, r_2, \dots)$.

Последовательности

Бесконечная последовательность

$$s_1, s_2, s_3, \dots$$

формально определяется как функция f , областью определения которой является множество положительных целых чисел: $f(i) = s_i, i \geq 1$. Во многих случаях индексирование последовательности более удобно

начинать с нуля; тогда областью определения f будет множество целых неотрицательных чисел. Аналогично определим конечную последовательность или список

$$s_1, s_2, \dots, s_n$$

как функцию, областью определения которой является множество $\{1, 2, \dots, n\}$. Примером бесконечной последовательности являются простые числа

$$\begin{aligned} i &: 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10 \ \dots \quad (2.3) \\ p_i &: 2 \ 3 \ 5 \ 7 \ 11 \ 13 \ 17 \ 19 \ 23 \ 29 \ \dots, \end{aligned}$$

а перестановка

$$\Pi(1, 2, 3, 4, 5, 6) = (6, 2, 5, 1, 3, 4)$$

представляет собой пример конечной последовательности.

В комбинаторных алгоритмах часто приходится встречаться с представлениями конечных последовательностей (или начальных сегментов бесконечных последовательностей) и операциями над ними.

Различные способы представлений конечных последовательностей (или начальных сегментов бесконечных последовательностей) и операции над ними

Последовательное распределение. С вычислительной точки зрения простейшим представлением конечной последовательности является список ее членов, расположенных по порядку в последовательных ячейках памяти.

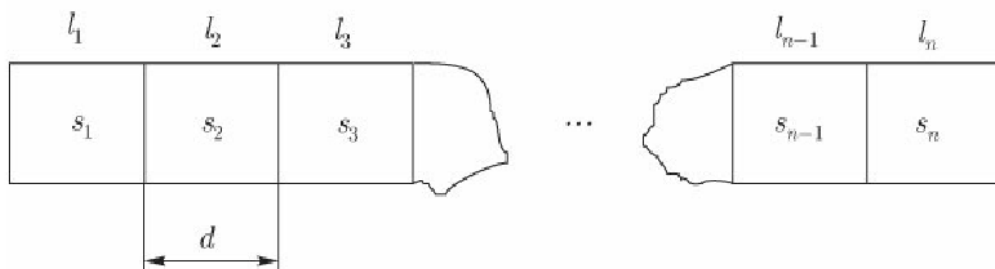


Рис. 2.1.

Так, s_1 хранится, начиная с ячейки l_1 , s_2 хранится, начиная с ячейки $l_2 = l_1 + d$, s_3 хранится, начиная с ячейки $l_3 = l_1 + 2d$ и так далее, где d - число ячеек, требуемых для хранения одного элемента последовательности.

Описанное выше представление последовательности имеет ряд преимуществ. Во-первых, оно легко осуществимо и требует небольших расходов в смысле памяти. Кроме того, оно полезно и потому, что существует простое соотношение между i и адресом ячейки, в которой хранится s_i :

$$l_i = l_1 + (i - 1)d.$$

Это соотношение позволяет организовать прямой доступ к любому элементу последовательности. Наконец, последовательное представление имеет достаточно широкий диапазон и включает в себя в качестве специального случая представление многомерных массивов. Например, чтобы представить массив размером $n \times m$

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \cdot & \cdot & \cdot & \cdot \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{pmatrix} \quad (2.4)$$

будем рассматривать его как последовательность $s_1, s_2, \dots, s_n$, в которой каждое s_i в свою очередь является последовательностью из m элементов i -й строки нашей матрицы. Таким образом, число ячеек, требуемых для записи элемента s_i (будем обозначать это число

символом $\bar{d}$), равно $m\bar{d}$, где $\bar{d}$ - число ячеек, требуемых для записи элемента a_{ij} . Поскольку последовательность s_i начинается в ячейке

$$l_i = l_1 + (i - 1)d = l_1 + (i - 1)m\bar{d},$$

ячейка для a_{ij} будет иметь следующий адрес:

$$l_i + (j - 1)\bar{d} = l_1 + [(i - 1)m + (j - 1)]\bar{d}.$$

Это представление известно как построчная запись матрицы; постолбцовая запись получается, если (2.4) рассматривать как последовательность $t_1, t_2, \dots, t_m$ в которой каждое t_i в свою очередь является последовательностью из элементов i -го столбца матрицы.

Последовательное распределение, наряду с преимуществами, имеет значительные недостатки. Например, такое представление становится неудобным, если требуется изменить последовательность путем включения новых и исключения имеющихся там элементов. Включение между s_i и s_{i+1} нового элемента требует сдвига $s_{i+1}, s_{i+2}, \dots, s_n$ вправо на одну позицию; аналогично, исключение s_i требует сдвига тех же элементов на одну позицию влево. С точки зрения времени обработки, такое передвижение элементов может оказаться дорогостоящим, и в случае динамических последовательностей лучше использовать технику связанного распределения, рассматриваемую в следующей лекции.

Характеристические векторы. Важной разновидностью последовательного распределения является случай, когда такому представлению подвергается последовательность некоторой основной последовательности $s_1, s_2, s_3 \dots$. В этом случае можно представить последовательность более удобно, используя характеристический вектор - последовательность из нулей и единиц, где i -й разряд равен единице, если s_i принадлежит рассматриваемой последовательности, и нулю в противном случае.

Например, характеристический вектор начального сегмента последовательности (2.3)

$$s_i : \quad 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10$$

характеристический вектор для простых чисел:

0 1 1 0 1 0 1 0 0 0

Здесь основной последовательностью является последовательность целых положительных чисел. В ЭВМ с 32-разрядными словами для запоминания простых чисел, меньших 10^6 , потребуется $10^6 / 32 = 31250$ слов. Замечая далее, что для $i > 1$ число 2^i не простое, можно сэкономить половину этого поля, выписывая разряды только для чисел видов $2i + 1, i \geq 1$, и запоминая, что простое число 2 отсутствует. Таким образом, простые числа, меньшие чем 10^6 , можно записать только 15625 словами. Поскольку число простых чисел, меньших 10^6 , равно 78498, последовательное представление, описанное ранее, потребовало бы поля в пять раз меньшего размера.

Характеристические векторы полезны в ряде случаев. Их полезность вытекает из их компактности, существования простого фиксированного соотношения между i и адресом i -го разряда и возможности при таком представлении очень легко исключать элементы.

Главное неудобство характеристических векторов состоит в том, что они не экономичны. Исключение составляют "плотные" последовательности последовательностей $s_1, s_2, s_3 \dots$. Кроме того, их трудно использовать, если не существует простого соотношения между i и s_i . Если такое соотношение сложное, то использование характеристических векторов может быть очень не экономичным в смысле времени обработки. Если последовательности недостаточно плотные, то значительным может оказаться объем памяти. В случае простых чисел между i и s_i имеется простое соотношение: $s_i = i$ (или $s_i = 2i + 1$, если использовать только нечетные числа). Теорема о простых числах утверждает, что число простых чисел, меньших n , приблизительно равно $n / \ln n$; таким образом, простые числа относительно плотно распределены в множестве целых чисел.

Последовательности (связанное распределение, стеки и очереди)

Связанное распределение. Разновидности связанных списков. Стеки и очередь. Задачи. Программы.

Связанное распределение

В лекции 11 даны примеры и программные реализации списков, стеков и очередей. Неудобство включения и исключения элементов при последовательном распределении происходит из-за того, что порядок следования элементов задается неявно требованием, чтобы смежные элементы последовательности находились в смежных ячейках памяти. В результате многие элементы последовательности во время включения или исключения должны передвигаться. Если это требование опущено, то можно выполнить операции включения и исключения без того, чтобы передвигать элементы последовательности. При связанном распределении последовательности (связанном списке) каждому s_i поставлен в соответствии указатель (ссылка) P_i , отмечающий ячейку, в которой записаны s_{i+1} и P_{i+1} . Существует также указатель P_0 , который указывает начальную ячейку последовательности, то есть ячейку с символами s_1 и P_1 . Все сказанное выше проиллюстрировано на рис. 3.1.

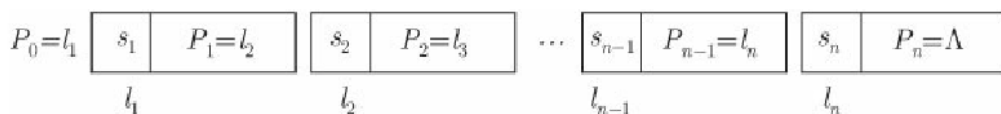


Рис. 3.1.

Здесь каждый узел состоит из двух полей. Под узлом понимается одно или несколько последовательных слов в памяти машины, которые выступают как единое целое и разделены на части, именуемые полями. В поле *INFO* размещен сам элемент последовательности, а в поле *LINK* - указатель на следующий за ним элемент.

Linked list - список с использованием указателей: список, в котором каждый элемент содержит указатель на следующий элемент или два

указателя - на следующий и предыдущий. Поскольку для $s_n = INFO(l_n)$ следующего элемента не существует, будем использовать обозначение $P_n = LINK(l_n) = \Lambda$, где Λ - пустой, или нулевой указатель. Так как точные значения $l_1, l_2, \dots, l_n$ для программиста не существенны, то в более общем виде связанное представление, показанное на рис. 3.1, можно изобразить так, как показано на рис.3.2.

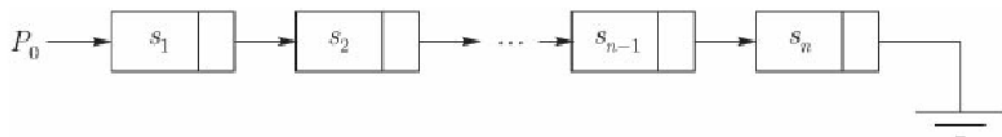


Рис. 3.2. Другой, более употребительный, способ представления списка

Связанное представление последовательностей облегчает операции включения элемента после некоторого s_i и исключения элемента s_{i+1} , если ячейка для s_i известна. Для этого необходимо лишь изменить значения некоторых указателей. Например, чтобы исключить элемент s_2 из последовательности, изображенной на рис. 3.2, необходимо только положить $LINK(l_1) = LINK(l_2)$; после такой операции элемента s_2 в последовательности больше не будет (рис.3.3). Чтобы в последовательность, изображенную на рис. 3.2, включить новый элемент s_5 , необходимо только воспроизвести новый элемент в некоторой ячейке l_5 с $INFO(l_5) = s_5$ и $LINK(l_5) = LINK(l_1)$ и присвоить $LINK(l_1) \leftarrow l_5$. Это изображено на рис.3.4.

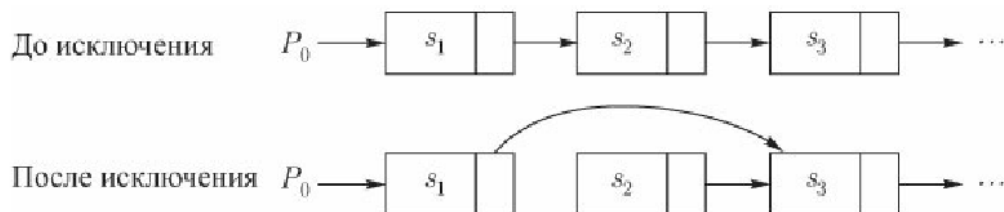
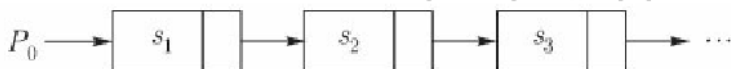


Рис. 3.3. Исключение элемента из связанного списка

До добавления



После добавления

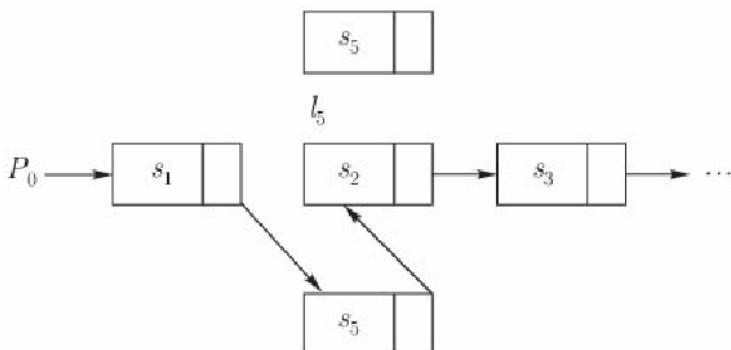


Рис. 3.4. Включение элемента в связанный список

Использование связанного представления предполагает существование некоторого механизма, который по мере надобности поставляет новые ячейки и собирает старые ячейки, когда они освобождаются, но эта проблема выходит за рамки нашего курса.

Будем предполагать, с целью использования в различных алгоритмах, существование операции порождения ячейки `get_cell`; если эта операция присутствует в правой части оператора присваивания, то она дает адрес (место) новой неиспользованной ячейки памяти. Таким образом, чтобы добавить элемент s_5 , как показано на рис. 3.4 нужно фактически использовать оператор `get_cell` для того чтобы найти значение l_5 . Проблему сбора ненужных ячеек памяти будем полностью игнорировать, предполагая лишь, что их каким-то образом собирают для последующего использования; такой процесс носит название сбора мусора.

Недостаток при связанном распределении - приходится тратить память на указатели P_i . В приложениях при выборе последовательного или связанного представления нужно сначала проанализировать типы операций, которые будут выполняться над последовательностью. Если операции производятся преимущественно над случайными элементами, осуществляют поиск специфических элементов или производят упорядочение элементов, то обычно лучше применять

последовательное распределение. Связанное распределение предпочтительнее, если в значительной степени используются операции включения и(или) исключения элементов, а также сцепления и (или) разбиения последовательностей.

Разновидности связанных списков

Тривиальной модификацией связанного списка, изображенного на [рис. 3.2](#), будет следующее несколько более гибкое представление последовательности: если P_n указывает на s_1 , как показано на [рис.3.5](#), то мы имеем так называемый циклический список. Такая форма списка дает возможность достигнуть любой элемент из любого другого элемента последовательности. Включение и исключение элементов здесь осуществляется так же, как и в нециклических списках, в то время как сцепление и разбиение реализуются несколько более сложно.

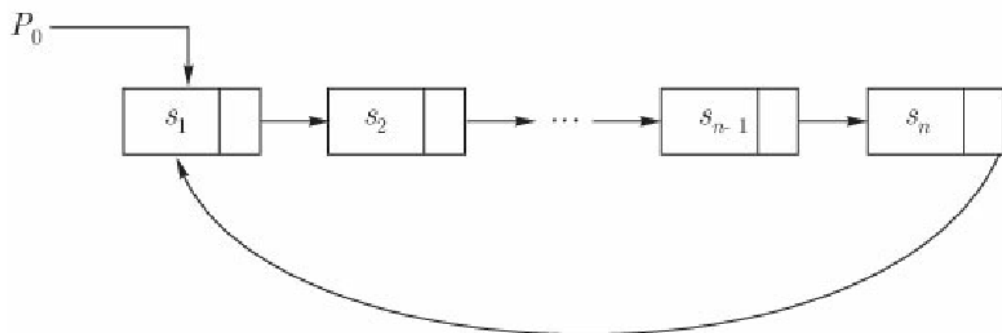


Рис. 3.5. Циклический список

Еще большая гибкость достигается, если использовать дважды связанный список, когда каждый элемент s_i последовательности вместо одного имеет два связанных с ним указателя. Как показано на [рис. 3.6](#), они указывают на элементы s_{i-1} и s_{i+1} . В таком списке для любого элемента имеется мгновенный прямой доступ к предыдущему и последующему элементам, в связи с чем облегчаются такие операции, как включение нового элемента перед s_i и исключение элемента s_i без предварительного знания его предшественника. Если есть необходимость, дважды связанный список очевидным образом можно сделать циклическим.

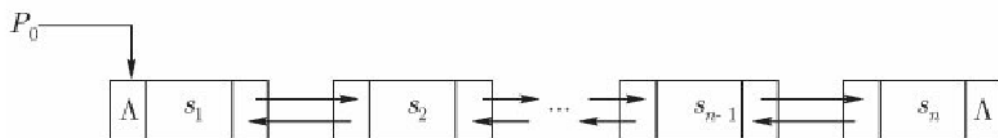


Рис. 3.6. Дважды связанный список

Стеки и очередь

В комбинаторных алгоритмах особую важность представляют две структуры данных, основанные на динамических последовательностях, т.е. последовательностях, которые изменяются вследствие включения новых и исключения имеющихся элементов. В обоих случаях операции включения и исключения, которым подвергается последовательность, имеют ограниченный вид: они производятся только в концах последовательности. Стек есть последовательность, у которой все включения и исключения происходят только в ее правом конце, называемом вершиной стека (соответственно, левый конец последовательности называется основанием). Таким образом, элементы включаются в стек и исключаются из него в соответствии с правилом "Первым пришел - последним ушел". Очередь - это последовательность, в которой все включения производятся на правом конце списка (в конце очереди), в то время как все исключения производятся на левом конце (в начале очереди). В противоположность стеку очередь оперирует в режиме "Первым пришел - первым ушел".

Стеки и очереди имеют важное значение. Для выполнения какой-либо определенной задачи может потребоваться выполнение ряда подзадач. Каждая подзадача может также привести к другим требующим выполнения подзадачам. И стеки, и очереди являются механизмом, посредством которого запоминаются подзадачи, подлежащие выполнению, а также порядок, в котором они должны быть выполнены. В некоторых случаях порядок таков: "Первым пришел - последним ушел"; тогда удобно использовать стеки. Если порядок подчиняется правилу "Первым пришел - первым ушел", то подходящим инструментом являются очереди.

Задачи

Задача 1. Создать список, элементами которого являются числа: 1 2 3 4 5 6 7 8 9. Вывести список на экран терминала. Включить в связанный список элемент 2005 после каждого элемента, который делится на 3. Модифицированный список вывести на экран терминала.

Задача 2. Очередью с приоритетом называется линейный список, который оперирует в режиме "первым включается - с высшим приоритетом исключается"; иными словами, каждому элементу очереди сопоставлено некоторое число - приоритет. Включения производятся в конец очереди, а исключения - в любом месте очереди, поскольку исключаемый элемент - это всегда элемент с высшим приоритетом. Нужно описать алгоритм (и его реализацию) включения и исключения для очередей с приоритетом.

Программы

Программа 1. Создание списка.

// Алгоритм реализован на языке программирования Turbo-C++.

```
#include <stdio.h>
```

```
#include <conio.h>
```

```
#include <dos.h>
```

```
struct List{int i;
```

```
    List*next;
```

```
};
```

```
List*head=NULL;
```

```
void Hed(int i)
```

```
{if(head==NULL){head=new List;
```

```
    head->i=i;
```

```
    head->next=NULL;
```

```
    }else
```

```
{
```

```
    struct List*p,*p1;
```

```
    p=head;
```

```
while(p->next!=NULL)
```

```
p=p->next;
```

```
p1=new List;
```

```
p1->i=i;
```

```
p1->next=NULL;
```

```
p->next=p1;
```

```
}
```

```
}
```

```
int s=0;
```

```
void Print(List*p)
```

```
{printf("%d",p->i);
```

```
if(p->next!=NULL)Print(p->next);
```

```
}
```

```
void delist()
```

```
{List*p;
```

```
while(head!=NULL)
```

```
{p=head;
```

```
head=head->next;
```

```
delete(p);
```

```
}
```

```
}
```

```
void Vstavka(int i1,int c)
```

```
{List*p=head,*p1;
```

```
while(p->i!=i1)
```

```
p=p->next;
```

```
p1=new List;
```

```
p1->i=c;
```

```
p1->next=p->next;
```

```
p->next=p1;
```

```
}
```

```
void main()
```

```
{
```

```
clrscr();
```

```
for(int i=1;i<=10;i++)
```



```
Hed(i);
textcolor(12);
Print(head);
textcolor(1);
Vstavka(10,11);
printf("\n");
Print(head);
textcolor(11);
Vstavka(3,12);
printf("\n");
Print(head);
textcolor(14);
Vstavka(5,13);
printf("\n");
Print(head);
delist();
getch();
}
```

Программа 2. Создание стека и работа со стеком.

```
//Работа со стеком
// Алгоритм реализован на языке программирования Turbo-C++.
#include <stdio.h>
#include <dos.h>
#include <iostream.h>
#include <PROCESS.H>
#include <STDLIB.H>
#include <conio.H>
#define max_size 200
// char s[max_size]; //компоненты стека
int s[max_size];
int next=0; // позиция стека
int Empty()
{
    return next==0;
}
int Full()
{

```

```
        return next==max_size;
    }
    void Push()
    { if (next==max_size)
      {
        cout <<"Ошибка: стек полон"<<endl;}

    else { next++;cout <<"Добавлен"<<endl;
          cout <<"Что поместить в стек?"<<endl;
          cin <<s[next-1];
        }
    }
    void OUTst()
    {int i=0;
      if (next==0)
        {
          cout <<"Стек пуст"<<endl;}

    else { for(i=0;i <next;i++)
          cout <<s[i] <<" " <<endl;
        }
    }
    void Clear()
    {
      next=0;
    }
    Poz()
    {
      return next;
    }
    void Del()
    {
      int a;
      if (next==0) cout <<"Ошибка: стек пуст" <<endl; else
        {next--;cout <<"Удален" <<endl;}
    }
    void menu(){
      cout <<"0: распечатать стек" <<endl;
      cout <<"1: добавить в стек" <<endl;
```

```
    cout <<"2: удалить из стека" <<endl;
    cout <<"3: узнать номер позиции в стеке" <<endl;
    cout <<"4: узнать, пуст ли стек" <<endl;
    cout <<"5: узнать, полон ли стек" <<endl;
    cout <<"6: очистить стек" <<endl;
    cout <<"7: выход" <<endl;
}
main()
{
    char c;
    clrscr();
    textcolor(11);
    do {
        menu();
        cin>>c;
        clrscr();
        switch (c) {
            case "0":OUTst();getch();break;
            case "1":Push();break;
            case "2":Del();getch();break;
            case "3":cout <<"Номер" <<Poz() <<endl;getch();break;
            case "4":if (Empty()==1) cout <<"Пуст" <<endl; else cout <<"Не
пуст" <<endl;getch();break;
            case "5":if (Full()==1)cout <<"Полон" <<endl; else cout <<"Не
полон" <<endl;getch();break;
            case "6":Clear();cout <<"Стек очищен" <<endl;getch();break;
            case "7":exit(1);
        }
        delay(200);
    }
    while (c!=7);
    return 0;
}
```

Последовательности (деревья)

Деревья. Представления. Прохождения. Длина путей. Задача. Программа.

Деревья

Конечное корневое дерево T формально определяется как непустое множество упорядоченных узлов, таких, что существует один выделенный узел, называемый корнем дерева, а оставшиеся узлы разбиты на $m \geq 0$ поддеревьев $T_1, T_2, \dots, T_m$. Будем рассматривать только корневые деревья. Узлы, не имеющие поддеревьев, называются листьями ; остальные узлы называются внутренними узлами.

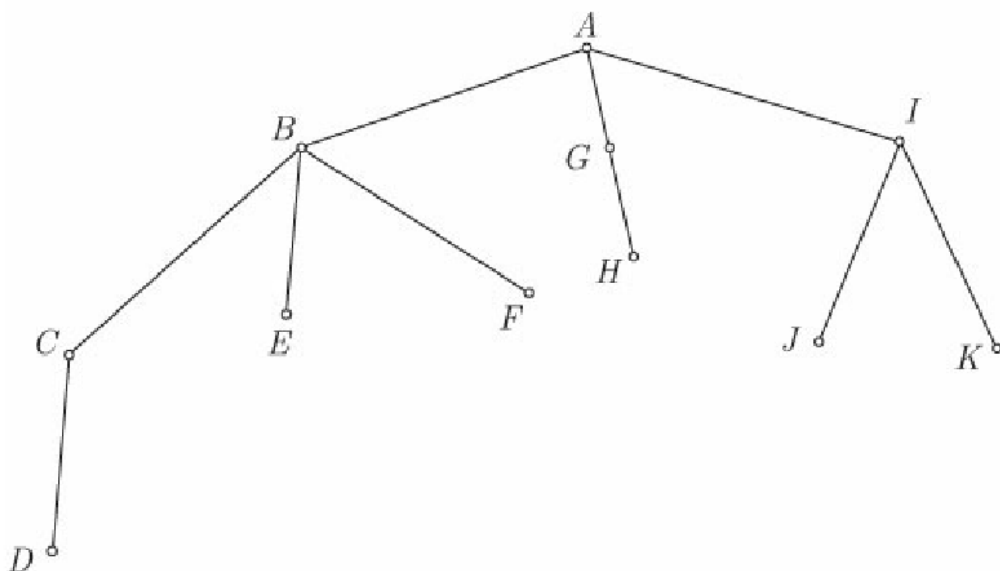


Рис. 4.1.

В первой лекции уже было использовано дерево - при изучении необходимого числа взвешиваний в задаче о фальшивой монете с n монетами. Так "рано" деревья появились в тексте не случайно, поскольку понятие дерева используется в различных важных аспектах данного курса. Посредством деревьев изображаются иерархические организации, поэтому они являются наиболее важными нелинейными

структурами в комбинаторных алгоритмах.

В описании соотношений между узлами дерева используем терминологию, принятую в генеалогических деревьях. Так, говорят, что в дереве или поддереве все узлы являются потомками его корня, и наоборот, корень есть предок всех своих потомков. Корень именуют отцом корней его поддеревьев, которые в свою очередь будут сыновьями корня. Например, на рис. 4.1 узел A является отцом узлов B, G и I ; J, K - сыновья I , а C, E, F - братья.

Все рассматриваемые нами деревья будут упорядочены, то есть для них будет важен относительный порядок поддеревьев каждого узла. Таким образом, деревья считаются различными.

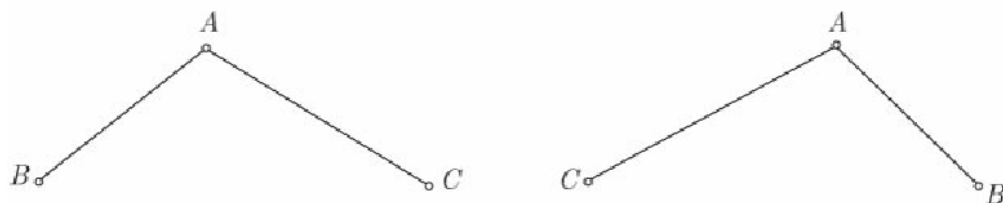


Рис. 4.2. Различные деревья

Определим лес как упорядоченное множество деревьев; в связи с этим можно перефразировать определение дерева: дерево есть непустое множество узлов, такое, что существует один выделенный узел, называемый корнем дерева, а оставшиеся узлы образуют лес с $m \geq 0$ поддеревьями корня. Важной разновидностью корневых деревьев является класс бинарных деревьев. Бинарное дерево T либо пустое, либо состоит из выделенного узла, называемого корнем, и двух бинарных поддеревьев: левого T_l и правого T_r . Бинарные деревья не являются подмножеством множества деревьев, они полностью отличаются по своей структуре, поскольку два следующих рисунка не изображают одно и то же бинарное дерево.



Рис. 4.3. Различные бинарные деревья

Как деревья, однако, они не отличаются от дерева, изображенного на рис. 4.4.



Рис. 4.4. Не бинарное дерево

Различие между деревом и бинарным деревом состоит в том, что дерево не может быть пустым, а каждый узел дерева может иметь произвольное число поддеревьев; в то же время, бинарное дерево может быть пустым. Каждая из вершин бинарного дерева может иметь 0, 1 или 2 поддерева, и существует различие между левым и правым поддеревьями.

Представления

Почти все машинные представления деревьев основаны на связанных распределениях. Каждый узел состоит из поля *INFO* и нескольких полей для указателей. Например, представление, которое будет удобным для изложения множества и мультимножества, для каждого узла имеет единственное поле для указателя *FATHER*, указывающего на отца

данного узла. При этом приведенное на [рис. 4.1](#) дерево будет выглядеть так, как показано на [рис. 4.5](#). Такое представление полезно, если необходимо подниматься по дереву от потомков к предкам. Такая операция встречается довольно редко. Чаще требуется опуститься по дереву от предков к потомкам.

Представление дерева (или леса) с использованием указателей, ведущих от предков к потомкам, довольно сложно, поскольку узел, имея не более чем одного отца, может в то же время иметь произвольно много сыновей. Другими словами, при таком представлении узлы должны различаться по размеру, что является определенным неудобством. Один из путей обхода этой трудности состоит в том, чтобы определить соответствие между деревьями и бинарными деревьями, поскольку бинарные деревья легко представить узлами фиксированного размера.

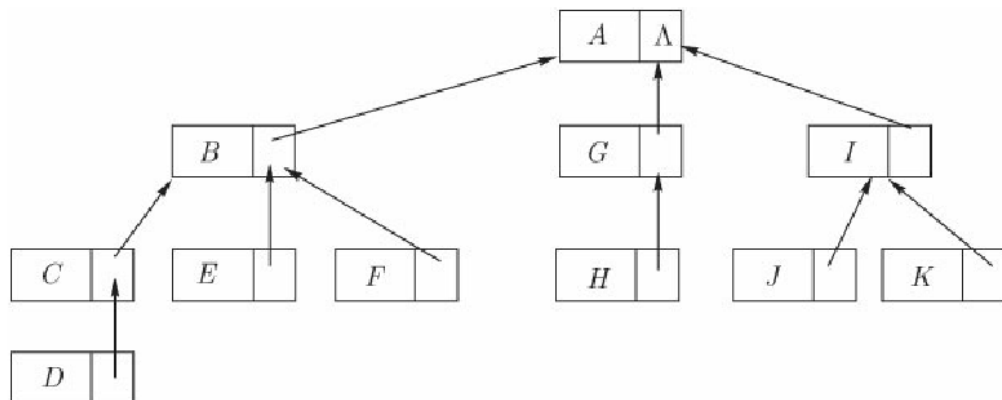


Рис. 4.5.

Каждый узел в этом случае имеет три поля: *LEFT*, указатель местоположения корня левого поддерева, *INFO*, содержимое узла, и *RIGHT*, указатель местоположения корня правого поддерева. Все сказанное выше проиллюстрировано на [рис. 4.6](#).

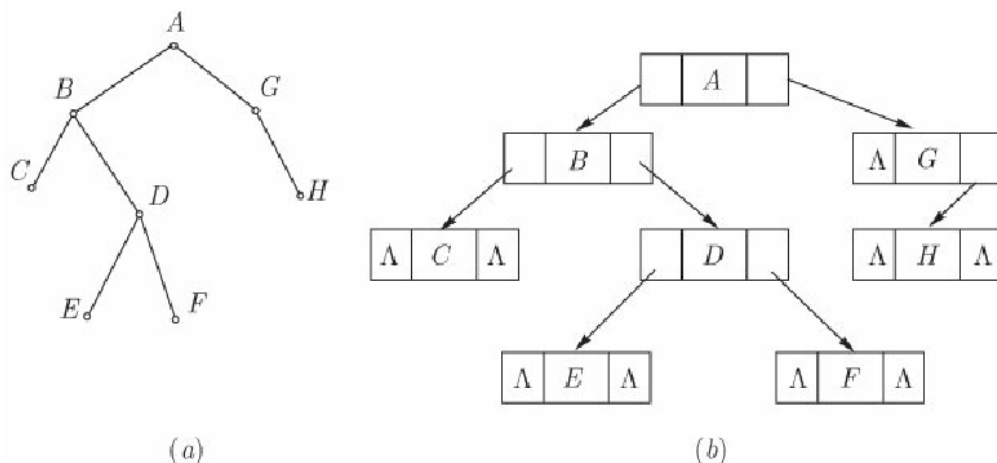


Рис. 4.6.

Можно представлять деревья как бинарные, используя узлы фиксированного размера, представляя каждый узел леса в виде узла, состоящего из полей *LEFT*, *INFO*, *RIGHT*. При этом *LEFT* предназначается для указания самого левого сына данного узла, а поле *RIGHT* - для указания следующего брата данного дочернего/сыновнего узла.

Прохождения

Во многих приложениях необходимо пройти лес, заходя в узлы, то есть обрабатывая их некоторым систематическим образом. Посещение каждого узла может быть связано с простой операцией, такой как печать содержимого, или со сложной, такой как вычисление функции. Будем предполагать, что при посещении узла структура леса не меняется. Рассмотрим четыре основных способа прохождения леса: в глубину, снизу вверх, в горизонтальном порядке и для бинарных деревьев - в симметричном порядке.

При прохождении в глубину, известном также как прохождение в прямом порядке, узлы леса проходятся в соответствии со следующей рекурсивной процедурой:

1. Посетить корень первого дерева.

2. пройти в глубину поддерева первого дерева (если оно есть).
3. Пройти в глубину оставшиеся деревья, если они есть.

Например, для леса, показанного на рис. 4.7, узлы будут проходиться в следующем порядке:

A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S.

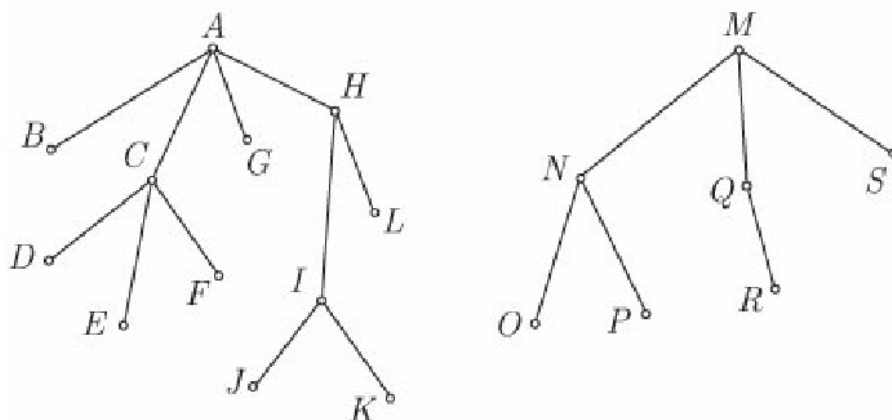


Рис. 4.7. Лес

Название "в глубину" отражает тот факт, что после посещения некоторого узла мы продолжаем прохождение в глубь дерева всякий раз, когда это возможно. Такой порядок особенно полезен в процедурах поиска.

Для бинарных деревьев эта процедура упрощается и выглядит следующим образом.

1. Посетить корень.
2. Пройти в глубину левое поддерево
3. Пройти в глубину правое поддерево.

Прохождение снизу вверх, известное также как обратный порядок или концевой порядок, осуществляется согласно следующей рекурсивной процедуре:

- Пройти снизу вверх поддерева первого дерева, если они есть.

- Посетить корень первого дерева.
- Пройти снизу вверх оставшиеся деревья, если они есть.

Название "снизу вверх" связано с тем, что в момент посещения произвольного узла его потомки оказываются уже пройденными. Такой порядок прохождения полезен, в частности, потому, что он позволяет вычислять рекурсивно определенные функции на лесах. При этом порядке прохождения узлы леса, показанного на рис. 4.7, проходятся в такой последовательности:

B, D, E, F, C, G, J, K, I, L, H, A, O, P, N, R, Q, S, M.

Рекурсивная процедура прохождения снизу вверх применительно к бинарным деревьям имеет следующий вид:

1. Пройти снизу вверх левое дерево.
2. Пройти снизу вверх правое дерево.
3. Посетить корень.

Симметричный порядок для бинарных деревьев определяется рекурсивно следующим образом:

1. Пройти в симметричном порядке левое поддерево.
2. Посетить корень.
3. Пройти в симметричном порядке правое поддерево.

Такой способ прохождения известен также как лексикографический порядок или внутренний порядок. Заметим, что прохождение леса снизу вверх эквивалентно прохождению в симметричном порядке бинарного дерева, соответствующего этому лесу (при естественном соответствии).

Сравнивая рекурсивные процедуры прохождения бинарных деревьев в глубину, снизу вверх и в симметричном порядке, можно обнаружить их значительное сходство:

прохождение в глубину	симметричный порядок	прохождение снизу вверх
посетить корень	левое поддерево	левое поддерево
левое поддерево	посетить корень	правое поддерево

правое поддереву

правое поддереву

посетить корень

Это сходство позволяет построить общий нерекурсивный алгоритм, который может быть применен к каждому из этих порядков прохождения бинарных деревьев.

Горизонтальный порядок прохождения. При таком способе узлы леса проходятся слева направо, уровень за уровнем от корня вниз. Таким образом, в соответствии с этой процедурой узлы леса, показанного на рис.4.7, будут проходиться в следующем порядке: *A, M, B, C, G, H, N, Q, S, D, E, F, I, L, O, P, R, J, K*. Такое прохождение дерева полезно в определенных алгоритмах на графах.

Длина путей

Деревья можно использовать не только как способ представления структуры данных, но также как средство для анализа поведения определенных алгоритмов. В связи с этим возникает потребность в количественных измерениях различных характеристик деревьев и, в частности, бинарных деревьев.

Наиболее важные количественные характеристики деревьев связаны с уровнями узлов. Уровень P определяется рекурсивно и считается равным нулю, если P корень T ; в противном случае уровень P определяется как $1 + \text{уровень}(FATHER(p))$. Понятие уровня дает возможность определить высоту $h(T)$ дерева T :

$$h(T) = \max_{p \in T} \text{уровня}(p).$$

Другими словами, высота дерева есть максимальное число ребер, образующих путь от корня к листу дерева.

Задача

Задача 1. Построить алгоритм обхода бинарного дерева (см. рис. 4.6,(a)) в глубину.

Программа

Программа 1. Обход бинарного дерева в глубину

```
//Обход ориентированных графов - поиск в глубину -
//обобщение обхода дерева в прямом порядке
#include <stdio.h>
#include <string.h>
#include <conio.h>
#include <stdlib.h>

int matr_sm[50][50];
int mark[50];
int n;

void vvod ()
{int v1,v2;
printf("Введите кол-во вершин в графе: ");
do
{
scanf("%d",&n);
if (n>51) printf("Ошибка!! Введите кол-во вершин в графе: ");
}
while (n>51);

for (int i=0;i <50;i++) for (int j=0;j <50;j++) matr_sm[i][j]=0;

printf("\nВведите связанные вершины : \n");
do
{ scanf("%d",&v1);
if (v1>n) //исходящая вершина
{ printf("ОШИБКА ВВОДА !!!!!!!!!");
abort();
}
if (v1==0){break;} // конец ввода
scanf(" %d",&v2);
if (v2>n) // входящая вершина
{ printf("ОШИБКА ВВОДА !!!!!!!!!");
abort();
```

```
    }
    if (v2==0){break;} //конец ввода
    matr_sm[v2-1][v1-1]=1;
} while(1);
}

void vivod()
{
    for (int j=0;j <n;j++)
    {
        for (int i=0;i<n;i++) printf("%d ",matr_sm[j][i]);
        printf("\n");
    }
}

void dfs(int v)
{
    int w;
    mark[v]=1; //посетили
    printf("%i ",v+1); //и выдали на экран
    for (w=0;w<n;w++)
        if (matr_sm[v][w]==1);
        else if (mark[w]==0) dfs(w);
}

void main()
{ clrscr();
  vvod();
  // printf("\n"); printf("МАТРИЦА СМЕЖНОСТИ\n");
  // vivod();

  printf("\n РЕЗУЛЬТАТ ПОИСКА В ГЛУБИНУ \n");
  for (int v=0;v<50;v++) mark[v]=0;
  for (v=0;v<n;v++) //если вершина не посещалась, то посетить
    if (mark[v]==0) dfs(v);

  getch();
}
```

Комбинаторика разбиений

Введение. Задачи. Разные статистики. Деревья и перестановки из n элементов. Число сочетаний S_{mn} . Задачи на разбиение чисел. Комбинаторные задачи теории информации.

Введение

В задачах, которые мы сейчас рассмотрим, элементы делятся на группы, и надо найти все способы такого раздела. При этом могут встретиться различные случаи. Иногда существенную роль играет порядок элементов в группах: например, когда сигнальщик вывешивает сигнальные флаги на нескольких мачтах, то для него важно не только то, на какой мачте окажется тот или иной флаг, но и то, в каком порядке эти флаги развешиваются. В других же случаях порядок элементов в группах никакой роли не играет. Когда игрок в домино выбирает кости из кучи, ему безразлично, в каком порядке они придут, а важен лишь окончательный результат.

Отличаются задачи и по тому, играет ли роль порядок самих групп. При игре в домино игроки сидят в определенном порядке, и важно не только то, как разделились кости, но и то, кому какие кости достались. Если раскладывать фотографии по одинаковым конвертам, чтобы разослать их, то существенно, как распределяются фотографии по конвертам, но порядок самих конвертов совершенно несущественен.

Играет роль и то, различаем ли мы между собой сами элементы или нет, а также различаем ли между собой группы, на которые делятся элементы. Наконец, в одних задачах некоторые группы могут оказаться пустыми, то есть не содержащими ни одного элемента, а в других такие группы недопустимы. В соответствии со всем сказанным возникает целый ряд различных комбинаторных задач на разбиение.

Задачи

Общая постановка этих задач:

Задача 1. Раскладка по ящикам

Даны n различных предметов и k ящиков. Надо положить в первый ящик n_1 предметов, во второй - n_2 предметов,..., в k -й - n_k предметов, где $n_1 + n_2 + \dots + n_k = n$. Сколькими способами можно сделать такое распределение?

Число различных раскладок по ящикам равно

$$P(n_1, n_2, \dots, n_k) = \frac{n!}{n_1! n_2! \dots n_k!}.$$

Эту формулу можно получить при решении следующей, на первый взгляд, совсем непохожей задачи:

Задача 2. Перестановки с повторением.

Имеются предметы k различных типов. Сколько различных перестановок можно сделать из n_1 предметов первого типа, n_2 предметов второго типа, ..., n_k предметов k -го типа? Число элементов в каждой перестановке равно $n_1 + n_2 + \dots + n_k = n$. Поэтому если бы все элементы были различны, то число перестановок равнялось бы $n!$. Но из-за того, что некоторые элементы совпадают, получится меньшее число перестановок. В самом деле, возьмем, например, перестановку

$$\frac{aa\dots a}{n_1} \frac{bb\dots b}{n_2} \dots \frac{xx\dots x}{n_3}, \quad (5.1)$$

в которой сначала выписаны все элементы первого типа, потом все элементы второго типа, ..., наконец, все элементы k -го типа. Элементы первого типа можно переставлять друг с другом $n_1!$ способами. Но так как все эти элементы одинаковы, то такие перестановки ничего не меняют. Точно так же ничего не меняют $n_2!$ перестановок элементов второго типа, ..., $n_k!$ перестановок элементов k -го типа. Перестановки элементов первого типа, второго типа и так далее можно делать независимо друг от друга. Поэтому элементы перестановки 5.1. можно переставлять друг с другом $n_1! n_2! \dots n_k!$ способами так, что она остается неизменной. То же самое верно и для любого другого расположения элементов. Поэтому множество всех $n!$

перестановок распадается на части, состоящие из $n_1!n_2!\dots n_k!$ одинаковых перестановок каждая. Значит, число различных перестановок с повторениями, которые можно сделать из данных элементов, равно

$$P(n_1, n_2, \dots, n_k) = \frac{n!}{n_1!n_2!\dots n_k!}, \quad (5.2)$$

где $n_1 + n_2 + \dots + n_k = n$. Пользуясь формулой 5.2, можно ответить на вопрос: сколько перестановок можно сделать из букв слова "Миссисипи"? Здесь у нас одна буква "м", четыре буквы "и", три буквы "с" и одна буква "п", а всего 9 букв. Значит, по формуле 5.2 число перестановок равно

$$P(4, 3, 1, 1) = \frac{9!}{4! \cdot 3! \cdot 1! \cdot 1!} = 2520.$$

Чтобы установить связь между этими задачами, занумеруем все n мест, которые могут занимать наши предметы. Каждой перестановке соответствует распределение номеров мест на k классов. В первый класс попадают номера тех мест, на которые попали предметы первого типа, во второй - номера мест предметов второго типа и так далее. Тем самым устанавливается соответствие между перестановками с повторениями и раскладкой номеров мест по "ящикам". Понятно, что формулы решения задач оказались одинаковыми.

В рассмотренных задачах мы не учитывали порядок, в котором расположены элементы каждой части. В некоторых задачах этот порядок надо учитывать.

Задача 3. Флаги на мачтах.

Имеется n различных сигнальных флагов и k мачт, на которые их вывешивают. Значение сигнала зависит от того, в каком порядке развешены флаги. Сколькими способами можно развесить флаги, если все флаги должны быть использованы, но некоторые из мачт могут оказаться пустыми?

Каждый способ развешивания флагов можно осуществить в два этапа.

На первом этапе мы переставляем всеми возможными способами данные n флагов. Это можно сделать $n!$ способами. Затем берем один из способов распределения n одинаковых флагов по k матчам (число этих способов C_{n+k-1}^{k-1}). Пусть этот способ заключается в том, что на первую мачту надо повесить n_1 флагов, на вторую - n_2 флагов, ..., на k -ю n_k флагов, где $n_1 + n_2 + \dots + n_k = n$. Тогда берем первые n_1 флагов данной последовательности и развешиваем в полученном порядке на первой мачте; следующие n_2 флагов развешиваем на второй мачте и т.д. Ясно, что используя все перестановки n флагов и все способы распределения n одинаковых флагов по k матчам, получим все способы решения поставленной задачи. По правилу произведения получаем, что число способов развешивания флагов равно

$$n! C_{n+k-1}^{k-1} = \frac{(n+k-1)!}{(k-1)!} = A_{n+k-1}^n. \quad (5.3)$$

Вообще, если имеется n различных вещей, то число способов распределения этих вещей по k различным ящикам равно A_{n+k-1}^n .

Разные статистики

Задачи о раскладке предметов по ящикам весьма важны для статистической физики. Эта наука изучает, как распределяются по своим свойствам физические частицы; например, какая часть молекул данного газа имеет при данной температуре ту или иную скорость. При этом множество всех возможных состояний распределяют на большое число k маленьких ячеек (фазовых состояний), так что каждая из n частиц попадет в одну из ячеек.

Вопрос о том, какой статистике подчиняются те или иные частицы, зависит от вида этих частиц. В классической статистической физике, созданной Максвеллом и Больцманом, частицы считаются различимыми друг от друга. Такой статистике подчиняются, например, молекулы газа. Известно, что n различных частиц можно распределить по k ячейкам k^n способами. Если все эти k^n способов при заданной энергии имеют равную вероятность, то говорят о статистике Максвелла-Больцмана.

Оказалось, что этой статистике подчиняются не все физические объекты. Фотоны, атомные ядра и атомы, содержащие четное число элементарных частиц, подчиняются иной статистике, разработанной Эйнштейном и индийским ученым Бозе. В статистике Бозе-Эйнштейна частицы считаются неразличимыми друг от друга. Поэтому имеет значение лишь то, сколько частиц попало в ту или иную ячейку, а не то, какие именно частицы туда попали.

Однако для многих частиц, например таких как электроны, протоны и нейтроны, не годится и статистика Бозе-Эйнштейна. Для них в каждой ячейке может находиться не более одной частицы, причем различные распределения, удовлетворяющие указанному условию, имеют равную вероятность. В этом случае может быть C_k^n различных распределений. Эта статистика называется статистикой Дирака-Ферми.

Деревья и перестановки из n элементов

С помощью леса можно представить перестановки из n элементов множества $M = \{a; b; c; d\}$ (множество мы определяем так: множество - это неупорядоченная совокупность различных объектов или структура данных, используемая для представления множества). Подсчитаем, сколько можно получить перестановок. Для n такой лес изображен на рис. 5.1.

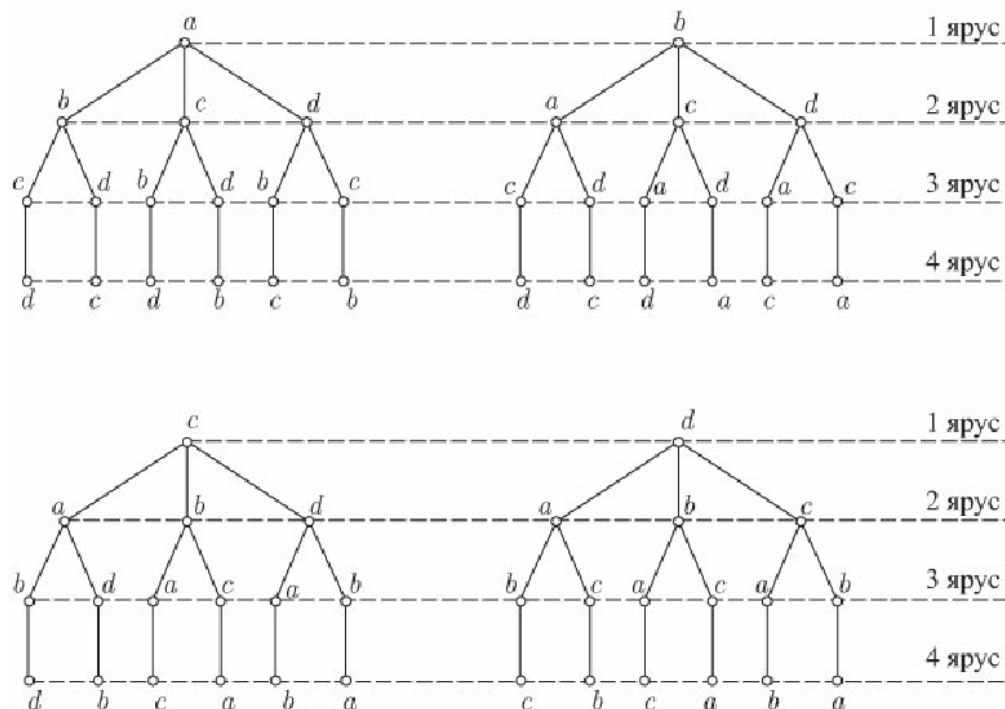


Рис. 5.1. Всевозможные перестановки прочитываются по этой схеме от корневой до висячей вершины соответствующего дерева. Ярус показывает номер места, на котором расположен элемент. Число висячих вершин леса равно числу перестановок

Число сочетаний

Рассмотрим подмножества множества, состоящего из пяти элементов, и подсчитаем их число. При этом записывать подмножества будем не с помощью букв, как обычно, а в виде последовательностей длиной пять, составленных из нулей и единиц. Каждая из единиц указывает на наличие в подмножестве соответствующего элемента. Например, подмножества, содержащие один элемент, будут изображаться следующими последовательностями: 10000, 01000, 00100, 00010, 00001. Пустое подмножество $\emptyset$ будет соответствовать последовательности 00000. Подмножества, содержащие по два элемента из пяти, запишутся с помощью следующих последовательностей: 11000, 10100, 10010, 10001, 01100, 01010, 01001, 00110, 00101, 00011. Всего их $C_5^2 = 10$.

Вообще, число сочетаний из n элементов по m равно числу всевозможных последовательностей из m единиц и $n - m$ нулей.

Задачи на разбиение чисел

Теперь мы переходим к задачам, в которых все разделяемые предметы совершенно одинаковы. В этом случае можно говорить не о разделении предметов, а о разбиении натуральных чисел на слагаемые (которые, конечно, тоже должны быть натуральными числами).

Здесь возникает много различных задач. В одних задачах учитывается порядок слагаемых, в других - нет.

Задача 4. Отправка бандероли.

За пересылку бандероли надо уплатить 18 рублей. Сколькими способами можно оплатить ее марками стоимостью 4, 6, и 10 рублей, если два способа, отличающиеся порядком марок, считаются различными (запас марок различного достоинства считаем неограниченным)?

Обозначим через $f(N)$ число способов, которыми можно наклеить марки в 4, 6 и 10 рублей так, чтобы общая стоимость этих марок равнялась N . Тогда для $f(N)$ справедливо следующее соотношение:

$$f(N) = f(N - 4) + f(N - 6) + f(N - 10). \quad (5.4)$$

Пусть имеется некоторый способ наклейки марок с общей стоимостью N , и пусть последней наклеена марка стоимостью 4 рубля. Тогда все остальные марки стоят $(N - 4)$ рубля. Наоборот, присоединяя к любой комбинации марок общей стоимостью $(N - 4)$ рубля одну четырехрублевую марку, получаем комбинацию марок стоимостью N рублей. При этом из разных комбинаций стоимостью $(N - 4)$ рублей получается разные комбинации стоимостью N рублей. Итак, число искомых комбинаций, где последней наклеена марка стоимостью 4 рубля, равно $f(N - 4)$.

Точно так же доказывается, что число комбинаций, оканчивающихся на шестирублевую марку, равно $f(N - 6)$, а на десятирублевую марку оканчиваются $f(N - 10)$ комбинацией. Поскольку любая комбинация оканчивается на марку одного из указанных типов, то по правилу суммы получаем соотношение 5.4.

Соотношение 5.4 позволяет свести задачу о наклеивании марок на сумму N рублей к задачам о наклеивании марок на меньшие суммы. Но при малых значениях N задачу легко решить непосредственно. Простой подсчет показывает, что

$$\begin{aligned} f(0) &= 1, f(1) = f(2) = f(3) = 0, f(4) = 1, f(5) = 0, \\ f(6) &= 1, f(7) = 0, f(8) = 1, f(9) = 0. \end{aligned}$$

Равенство $f(0) = 1$ означает, что сумму в 0 рублей можно уплатить единственным образом: совсем не наклеивая марок. А сумму в 1,2,3,5,7 и 9 рублей вообще никак нельзя получить с помощью марок стоимостью 4, 6 и 10 рублей. Используя значения $f(N)$ для $N = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9$, легко найти $f(10)$:

$$f(10) = f(6) + f(4) + f(0) = 3.$$

После этого находим

$$\begin{aligned} f(11) &= f(7) + f(5) + f(1) = 0, \\ f(12) &= f(8) + f(6) + f(2) = 2 \end{aligned}$$

и т.д. Наконец, получаем значение $f(18) = 8$. Таким образом, марки можно наклеить восемью способами. Эти способы таковы: 10, 4, 4 ; 4, 10, 4 ; 4, 4, 10 ; 6, 4, 4, 4 ; 4, 6, 4, 4 ; 4, 4, 6, 4 ; 4, 4, 4, 6 ; 6, 6, 6.

Отметим, что значения $f(N)$ для $N = 1, 2, 3, 4, 5, 6, 7, 8, 9$ можно было получить иначе, не приводя непосредственно проверки. Дело в том, что при $N < 0$ имеем $f(N) = 0$, поскольку отрицательную сумму нельзя уплатить, наклеивая неотрицательное количество марок. В то же время, как мы видели, $f(0) = 1$. Поэтому

$$f(1) = f(-3) + f(-5) + f(-9) = 0.$$

Точно так же получаем значение $f(2) = 0, f(3) = 0$, а для $N = 4$ имеем $f(4) = f(0) + f(-2) + f(-6) = 1$.

Задача 5. Общая задача о наклейке марок.

Разобранная задача является частным случаем следующей общей задачи: Имеются марки достоинством в $n_1, n_2, \dots, n_k$. Сколькими способами можно оплатить с их помощью сумму в N рублей, если два способа, отличающиеся порядком, считаются различными? Все числа $n_1, n_2, \dots, n_k$ различны, а запас марок неограничен. Здесь на первом месте мы будем указывать число слагаемых, на втором – разбиваемое число и на последнем – ограничения на величину слагаемых.

В этом случае число $f(N)$ способов удовлетворяет соотношению

$$f(N) = f(N - n_1) + f(N - n_2) + \dots + f(N - n_k). \quad (5.5)$$

При этом $f(N) = 0$, если $f(0) = 1$ и $N < 0$. С помощью соотношения 5.5 можно найти $f(N)$ для любого N , последовательно вычисляя $f(1), f(2), \dots, f(N - 1)$. Рассмотрим частный случай этой задачи, когда $n_1 = 1, n_2 = 2, \dots, n_k = k$. Мы получаем всевозможные разбиения числа N на слагаемые $1, 2, \dots, k$, причем разбиения, отличающиеся порядком слагаемых, считаются различными.

Обозначим число этих разбиений через $\varphi(k; N)$. (На первом месте мы будем указывать число слагаемых, на втором – разбиваемое число и на последнем – ограничения на величину слагаемых.) Из соотношения 5.5 следует, что

$$\varphi(k; N - 1) + \varphi(k; N - 2) + \dots + \varphi(k; N - k). \quad (5.6)$$

При этом $\varphi(k; 0) = 1$ и $\varphi(k; N) = 0$, если $N < 0$. Вычисление $\varphi(N; k)$ можно упростить, если заметить, что

$$\varphi(N-1; k) = \varphi(N-2; k) + \varphi(N-k-1; k),$$

и потому

$$\varphi(N; k) = 2\varphi(N-1; k) - \varphi(N-k-1; k). \quad (5.7)$$

Ясно, что слагаемые не могут быть больше N . Поэтому $\varphi(N, N)$ равно числу всех разбиений на N на натуральные слагаемые (включая и "разбиение" $N = N$). Если число слагаемых равно s , то получаем

C_{N-1}^{s-1} разбиений. Поэтому

$$\varphi(N, N) = C_{N-1}^0 + C_{N-1}^1 + \dots + C_{N-1}^{N-1} = 2^{N-1}.$$

Итак, мы доказали, что натуральное число N можно разбить на слагаемые 2^{N-1} способами. Напомним, что при этом учитывается порядок слагаемых. Например, число 5 можно разбить на слагаемые $2^{5-1} = 16$ способами.

5 = 5	5 = 3 + 1 + 1	5 = 1 + 2 + 2
5 = 4 + 1	5 = 1 + 3 + 1	5 = 2 + 1 + 1 + 1
5 = 1 + 4	5 = 1 + 1 + 3	5 = 1 + 2 + 1 + 1
5 = 2 + 3	5 = 2 + 2 + 1	5 = 1 + 1 + 2 + 1
5 = 3 + 2	5 = 2 + 1 + 2	5 = 1 + 1 + 1 + 2
		5 = 1 + 1 + 1 + 1 + 1

Комбинаторные задачи теории информации

Информация - сведения, неизвестные до их получения, или данные, или значения, приписанные данным.

Теория информации - математическая дисциплина, изучающая количественные свойства информации.

Задачу, похожую на только что решенную, приходится решать в теории информации. Предположим, что сообщение передается с помощью

сигналов нескольких типов. Длительность передачи сигнала первого типа равна t_1 , второго типа - $t_2, \dots, k$ -го типа - t_k единиц времени.

Задача 6. Сколько различных сообщений можно передать с помощью этих сигналов за T единиц времени? При этом учитываются лишь "максимальные" сообщения, то есть сообщения, к которым нельзя присоединить ни одного сигнала, не выйдя за рамки отведенного для передачи времени.

Обозначим число сообщений, которые можно передать за время T через $f(T)$. Рассуждая точно так же, как и в задаче о марках, получаем, что $f(T)$ удовлетворяет соотношению

$$f(T) = f(T - t_1) + \dots + f(T - t_k). \quad (5.8)$$

При этом снова $f(T) = 0$, если $T < 0$ и $f(0) = 1$.

Последовательности (множества и мультимножества)

Множества и мультимножества. Формула включений и исключений. Решето Эратосфена. Примеры программ.

Множества и мультимножества

Не существует формального определения множества ; считается что это понятие первичное и не определяется. Так, можно говорить, что множество есть объединение различных элементов, но при этом мы оставляем неопределяемыми понятия "объединение" и "элементы". Дадим следующее определение множеству: множество - это неупорядоченная совокупность различных объектов или структура данных, используемая для представления множества. Мультимножество есть объединение не обязательно различных элементов; его можно считать множеством, в котором каждому элементу поставлено в соответствие положительное целое число, называемое кратностью.

Конечное множество S будем записывать в следующем виде:

$$S = \{s_1, s_2, \dots, s_n\},$$

где $s_1, s_2, \dots, s_n$ - элементы S , обязательно различные! Мощность множества S обозначается как $|S|$, для выписанного выше множества мощность записывается так $|S| = n$. Если S - конечное мультимножество, то будем записывать его в следующем виде:

$$S = \{ \underbrace{s_1, s_1, \dots, s_1}_{m_1 \text{ раз}}, \underbrace{s_2, s_2, \dots, s_2}_{m_2 \text{ раз}}, \underbrace{s_3, s_3, \dots, s_3}_{m_3 \text{ раз}} \} = \{m_1 \bullet s_1, m_2 \bullet s_2, \dots, m_n \bullet s_n\}.$$

Здесь все s_i различны и m_i - кратность элемента s_i . В этом случае мощность S равна

$$|S| = \sum_{i=1}^n m_i.$$

Наиболее общими операциями на множествах и мультимножествах являются операции объединения и пересечения. Для множеств эти

операции будем обозначать $\cup$ и $\cap$, а для мультимножеств - $\overset{+}{\cup}$ и $\overset{+}{\cap}$. Последовательное и связанное представление последовательностей можно использовать для множеств и мультимножеств очевидным способом. Индуцируя искусственный порядок элементов множества или используя собственный порядок, если он существует, можно рассматривать множество как последовательность. Аналогично, как последовательность можно рассматривать и мультимножество, или, для того чтобы сэкономить место, его можно рассматривать как последовательность пар, каждая из которых состоит из элемента и его кратности.

Как и для последовательностей, наилучший метод представления множеств или мультимножеств существенно зависит от операций, которые выполняются над ними. Предположим, например, что имеем дело с непересекающимися подмножествами множества $S = \{s_1, s_2, \dots, s_n\}$ и что над ними необходимо выполнить две следующие операции: объединение двух множеств и отыскание подмножества, содержащего данное s_i . Таким образом, в любой момент времени имеем разбиение S на непустые непересекающиеся подмножества. Рассмотрим эти операции в конце данной лекции.

С целью идентификации считаем, что каждое из непересекающихся подмножеств множества S имеет имя. Имя - это просто один из элементов подмножества, или, иначе, - представитель подмножества. Когда мы будем ссылаться на имя подмножества, то будем под этим подразумевать его представителя. Рассмотрим, например, множество

$$S = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\},$$

разбитое на четыре непересекающихся подмножества

$$\{1, 6, \langle 7 \rangle, 8, 11\} \{ \langle 2 \rangle \} \{ \langle 3 \rangle, 4, 5 \} \{ 9, \langle 10 \rangle \} \quad (6.1)$$

В каждом из подмножеств, взятый в скобки элемент является его именем. Если нам нужно найти подмножество, в котором содержится восьмерка, искомым ответом будет 7, то есть имя подмножества, содержащего восьмерку. Если нужно взять объединение подмножеств с именами 2 и 10, получим разбиение множества S следующего вида:

$$\{1, 6, \langle 7 \rangle, 8, 11\} \{ \langle 3 \rangle, 4, 5 \} \{ \langle 2 \rangle \} \cup \{9, \langle 10 \rangle\}$$

Именем множества $\{ \langle 2 \rangle \} \cup \{9, \langle 10 \rangle\}$ может быть или 2, или 10. Предполагаем, что вначале имеется разбиение множества $S = \{s_1, s_2, \dots, s_n\}$ на n подмножеств, каждое из которых состоит из одного элемента

$$\{ \langle s_1 \rangle \} \{ \langle s_2 \rangle \}, \dots, \{ \langle s_n \rangle \} \quad (6.2)$$

и имя каждого из них есть просто этот единственный элемент. Это разбиение преобразуется путем применения операций объединения вперемешку с операциями отыскания. Такая кажущаяся на первый взгляд надуманной задача чрезвычайно полезна в определенных комбинаторных алгоритмах; пример ее полезности виден в "жадном" алгоритме (лекция 16). Для реализации операций и объединения, и отыскания опишем процедуры (операции) $UNION(x, y)$ и $FIND(x)$. Процедура (операция) $UNION(x, y)$ по именам двух различных подмножеств x и y образует новое подмножество, содержащее все элементы множеств x и y . Процедура (операция) $FIND(x)$ выдает имя множества, содержащего x . Например, если нужно множество, содержащее a , объединить с множеством, содержащим b , необходимо выполнить следующую последовательность операторов:

```

 $x \leftarrow FIND(a)$ 
 $y \leftarrow FIND(b)$ 
if  $x \neq y$  then  $UNION(x, y)$ .

```

Предположим, что мы имеем u операций объединения, перемешанных с f операциями отыскания, и что начинаем алгоритм с множества $S = \{s_1, s_2, \dots, s_n\}$, которое разбито на подмножества, состоящие из одного элемента (см. 6.2.). Найдем такую структуру данных для представления непересекающихся подмножеств множества S , чтобы последовательность операций можно было производить эффективно. Такой структурой данных является представление в виде леса с указателями отца, как показано на рис. 4.5 лекции 4. Каждый элемент

S_i множества будет узлом леса, а отцом его будет элемент из того же подмножества, что и S_i . Если элемент не имеет отца, то есть является корнем, то он будет именем своего подмножества. В соответствии с этим разбиение 6.1 может быть представлено так:

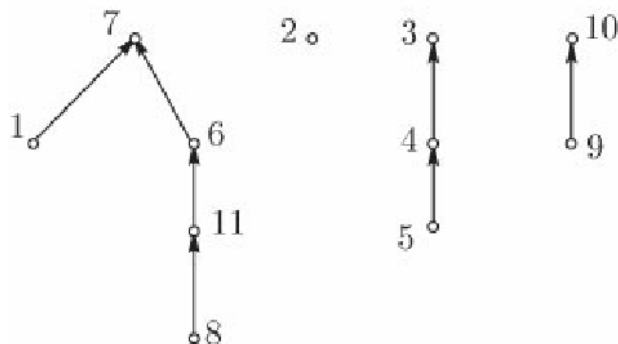


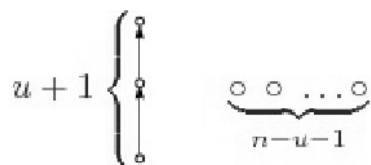
Рис. 6.1. Представление разбиения

При таком представлении процедура (операция) $FIND(x)$ состоит в переходах по указателям отцов от x до корня, то есть имени, его подмножества. Процедура (операция) $UNION(x, y)$ состоит в связывании вместе некоторым образом деревьев, имеющих корни x и y . Например, такую связь можно осуществить, сделав y отцом x .

После u операций объединения наибольшее из возможных подмножеств, получающихся в результате разбиения S , будет содержать $u + 1$ элементов. Поскольку каждое объединение уменьшает число подмножеств на единицу, последовательность операций может содержать не более $n - 1$ объединений, откуда $u \leq n - 1$. Так как каждая операция объединения изменяет имя подмножества, содержащего некоторые элементы, можно считать, что каждому объединению предшествует по крайней мере одно отыскание, в связи с чем естественно предположить, что $f \geq u$. Выясним, насколько эффективно можно выполнить последовательность из $u \leq n - 1$ операций объединения, перемешанных с $f \geq u$ операциями отыскания. Время, требуемое на операции объединения, очевидно, пропорционально u , потому что необходимая для каждой операции объединения переделка некоторых указателей требует фиксированного

количества работы. Поэтому сосредоточим свое внимание на времени, требуемом для f операций отыскания.

Если операция $UNION(x, y)$ выполняется путем назначения x отцом y , то после u операций объединения может получиться лес, показанный ниже.



В этом случае, если f операций отыскания выполняются после всех операций объединения и каждый поиск начинается внизу цепи из $u + 1$ элементов множества, то ясно, что время, требуемое на операции отыскания, будет пропорционально $f \cdot (u + 1)$. Очевидно, оно не может быть больше, чем константа, умноженная на $f \cdot (u + 1)$. Можно существенно уменьшить эту оценку.

Формула включений и исключений

Пусть имеется N предметов, некоторые из которых обладают свойствами $\alpha_1, \alpha_2, \dots, \alpha_n$. При этом каждый предмет может либо не обладать ни одним из этих свойств, либо обладать одним или несколькими свойствами. Обозначим через $N(\alpha_i \alpha_j \dots \alpha_k)$ количество предметов, обладающих свойствами $\alpha_i, \alpha_j, \dots, \alpha_k$ (и быть может, еще некоторыми из других свойств). Если нужно взять предметы, не обладающие некоторым свойством, то эти свойства пишем со штрихом. Например, через $N(\alpha_1 \alpha_2 \alpha'_4)$ обозначено количество предметов, обладающих свойствами α_1, α_2 , но не обладающих свойством α_4 (вопрос об остальных свойствах останется открытым). Число предметов, не обладающих ни одним из указанных свойств, обозначается по этому правилу через $N(\alpha'_1 \alpha'_2 \dots \alpha'_n)$. Общий закон состоит в том, что

$$N(\alpha'_1 \alpha'_2 \dots \alpha'_n) = N - N(\alpha_1) - N(\alpha_2) - \dots - N(\alpha_n) + N(\alpha_1 \alpha_2) + \\ + N(\alpha_1 \alpha_3) + \dots + N(\alpha_1 \alpha_n) + \dots + N(\alpha_{n-1} \alpha_n) - N(\alpha_1 \alpha_2 \alpha_3) - \dots - \\ - N(\alpha_{n-2} \alpha_{n-1} \alpha_n) + \dots + (-1)^n N(\alpha_1 \alpha_2 \dots \alpha_n).$$

Здесь алгебраическая сумма распространена на все комбинации свойств $\alpha_1, \alpha_2, \dots, \alpha_n$ (без учета их порядка), причем знак $+$ ставится, если число учитываемых свойств четно, и знак $-$, если это число нечетно.

Например, $N(\alpha_1 \alpha_3 \alpha_6 \alpha_8)$ входит со знаком $+$, а $N(\alpha_3 \alpha_4 \alpha_{10})$ со знаком $-$. Формулу 6.3 называют формулой включений и исключений - сначала исключаются все предметы, обладающие хотя бы одним из свойств $\alpha_1, \alpha_2, \dots, \alpha_n$, потом включаются предметы, обладающие, по крайней мере, двумя из этих свойств, затем исключаются имеющие, по крайней мере, три и т.д.

Решето Эратосфена

Одной из самых больших загадок математики является расположение простых чисел в ряду всех натуральных чисел. Иногда два простых числа идут через одно, (например, 17 и 19, 29 и 31), а иногда подряд идет миллион составных чисел. Сейчас ученые знают уже довольно много о том, сколько простых чисел содержится среди N первых натуральных чисел. В этих подсчетах весьма полезным оказался метод, восходящий еще к древнегреческому ученому Эратосфену. Он жил в третьем веке до новой эры в Александрии.

Эратосфен занимался самыми различными вопросами - ему принадлежат интересные исследования в области математики, астрономии и других наук. Впрочем, такая разносторонность привела его к некоторой поверхностности. Современники несколько иронически называли Эратосфена "во всем второй": второй математик после Евклида, второй астроном после Гиппарха и т.д.

В математике Эратосфена интересовал как раз вопрос о том, как найти все простые числа среди натуральных чисел от 1 до N . (Эратосфен считал 1 простым числом. Сейчас математики считают 1 числом особого вида, которое не относится ни к простым, ни к составным числам.) Он придумал для этого следующий способ. Сначала

вычеркивают все числа, делящиеся на 2 (исключая само число 2). Потом берут первое из оставшихся чисел (а именно 3). Ясно, что это число - простое. Вычеркивают все идущие после него числа, делящиеся на 3. Первым оставшимся числом будет 5. Вычеркивают все идущие после него числа, делящиеся на 5, и т.д. Числа, которые уцелеют после всех вычеркиваний, и являются простыми. Так как во времена Эратосфена писали на восковых табличках и не вычеркивали, а "выкалывали" цифры, то табличка после описанного процесса напоминала решето. Поэтому метод Эратосфена для нахождения простых чисел получил название "решето Эратосфена".

Подсчитаем, сколько останется чисел в первой сотне, если мы вычеркнем по методу Эратосфена числа, делящиеся на 2, 3 и 5. Иными словами, поставим такой вопрос: сколько чисел в первой сотне не делится ни на одно из чисел 2, 3, 5? Эта задача решается по формуле включения и исключения.

Обозначим через α_1 свойство числа делиться на 2, через α_2 - свойство делимости на 3 и через α_3 - свойство делимости на 5. Тогда $\alpha_1\alpha_2$ означает, что число делится на 6, $\alpha_1\alpha_3$ означает, что оно делится на 10, и $\alpha_2\alpha_3$ - оно делится на 15. Наконец, $\alpha_1\alpha_2\alpha_3$ означает, что число делится на 30. Надо найти, сколько чисел от 1 до 100 не делится ни на 2, ни на 3, ни на 5, то есть не обладает ни одним из свойств α_1 , α_2 , α_3 . По формуле 6.3 имеем

$$N(\alpha'_1\alpha'_2\alpha'_3) = 100 - N(\alpha_1) - N(\alpha_2) - N(\alpha_3) + N(\alpha_1\alpha_2) + N(\alpha_1\alpha_3) + N(\alpha_2\alpha_3) - N(\alpha_1\alpha_2\alpha_3).$$

Но чтобы найти, сколько чисел от 1 до N делится на n , надо разделить N на n и взять целую часть получившегося частного. Поэтому

$$N(\alpha_1) = 50, N(\alpha_2) = 33, N(\alpha_3) = 20, \\ N(\alpha_1\alpha_2) = 16, N(\alpha_1\alpha_3) = 10, N(\alpha_2\alpha_3) = 6, N(\alpha_1\alpha_2\alpha_3) = 3,$$

и значит,

$$N(\alpha'_1\alpha'_2\alpha'_3) = 32.$$

Таким образом, 26 числа от 1 до 100 не делятся ни на 2, ни на 3, ни на

5. Эти числа и уцелеют после первых трех шагов процесса Эратосфена. Кроме них останутся сами числа 2, 3 и 5. Всего останется 35 чисел. А из первой тысячи после первых трех шагов процесса Эратосфена останется 335 чисел. Это следует из того, что в этом случае

$$N(\alpha_1) = 500, N(\alpha_2) = 333, N(\alpha_3) = 200, \\ N(\alpha_1\alpha_2) = 166, N(\alpha_1\alpha_3) = 100, N(\alpha_2\alpha_3) = 66, N(\alpha_1\alpha_2\alpha_3) = 33.$$

Примеры программы

Программа 1. Решето Эратосфена.

{В примере, иллюстрирующем работу с множествами, реализуется алгоритм выделения из первой сотни натуральных чисел всех простых чисел. В алгоритме лежит прием "решета Эратосфена".

Алгоритм написан на языке программирования Turbo-Pascal.}

```
Uses crt;
Const
  N=100; {количество элементов исходного множества}
Type
  SetN=set of 1..N;
var
  n1, next, I : word; {вспомогательные переменные }
  BeginSet,      {исходное множество }
  PrimerSet: SetN; {множество простых чисел }
Begin
  Clrscr; {почистить экран}
  BeginSet:=[2..N]; {создать исходное множество}
  PrimerSet:=[1];   {первое простое число}
  next:=2;          {следующее простое число}
  while BeginSet <> [ ] do {начало основного цикла}
    begin
      n1:=next; {n1-число, кратное очередному простому (next)}
      while n1<=N do
        {цикл удаления из исходного множества непростых чисел}
        begin
          BeginSet:=BeginSet-[n1];
```



```

    n1:=n1+next {следующее кратное}
    end;          {конец цикла удаления}
repeat {получить следующее простое число, которое есть первое
        не вычеркнутое из исходного множества}
inc(next)
until(next in BeginSet) or (next > N)
end; {конец основного цикла}
{вывод результата}
textcolor(15); {задание цвета}
for I:=1 to N do
    if i in PrimerSet then write(I:8);
    readln;
end.

```

Программа 2. Простые числа в порядке убывания от 200.

{Находит и пишет все простые числа в порядке убывания от 2 до 200.
Алгоритм написан на языке программирования Turbo-Pascal.}

```

Uses crt;
const
n=197;
var

i,q,w,e,r,t:integer;
prost:array[1..n] of integer;

begin
clrscr;
e:=1;
r:=0;
for q:=1 to n do begin
r:=0;
for w:=2 to n-1 do
if (q<>w) and (q mod w = 0) then r:=1;
{prost[e]:=q; e:=e+1;}

if r=0 then begin{begin write(q, ' ');}

```

```
prost[e]:=q;  
e:=e+1;  
end;  
end;  
  
for i:=e downto 2 do begin  
write (prost[i], ' ');  
if wherex>70 then writeln;  
  
end;  
readln;  
end.
```

Программа 3. Поиск литер в строке.

{Поиск числа вхождений в данную строку литер а, с, е, h.
Алгоритм написан на языке программирования Turbo-Pascal.}

```
Uses crt;  
type  
liter_set = set of char;  
  
var  
c:integer;  
let: liter_set;  
a:char;  
  
begin  
clrscr;  
let:=['a','c','e','h'];  
  
repeat  
a:=readkey;  
write(a);  
if a in let then c:=c+1;  
until a = '.';  
writeln;  
writeln('Общее число вхождений литер а,с,е,h в вашу запись:',c);  
readln;
```

end.

Рекуррентные соотношения

Размещения без повторений. Перестановки. Сочетания. Рекуррентные соотношения. Другой метод доказательства. Процесс последовательных разбиений. Задача: "Затруднение мажордома".

Размещения без повторений

Имеется n различных предметов. Сколько из них можно составить k - расстановок? При этом две расстановки считаются различными, если они либо отличаются друг от друга хотя бы одним элементом, либо состоят из одних и тех же элементов, но расположенных в разном порядке. Такие расстановки называют размещениями без повторений, а их число обозначают A_n^k . При составлении k -размещений без повторений из n предметов нам надо сделать k выборов. На первом шагу можно выбрать любой из имеющихся n предметов. Если этот выбор уже сделан, то на втором шагу приходится выбирать из оставшихся $n - 1$ предметов. На k - м шагу $n - k + 1$ предметов. Поэтому по правилу произведения получаем, что число k -размещений без повторения из n предметов выражается следующим образом:

$$A_n^k = n(n - 1) \dots (n - k + 1).$$

Перестановки

При составлении размещений без повторений из n элементов по k мы получили расстановки, отличающиеся друг от друга и составом, и порядком элементов. Но если брать расстановки, в которые входят все n элементов, то они могут отличаться друг от друга лишь порядком входящих в них элементов. Такие расстановки называют перестановками из n элементов, или, короче, n - перестановками.

Сочетания

В тех случаях, когда нас не интересует порядок элементов в комбинации, а интересует лишь ее состав, говорят о сочетаниях. Итак,

k - сочетаниями из n элементов называют всевозможные k - расстановки, составленные из этих элементов и отличающиеся друг от друга составом, но не порядком элементов. Число k -сочетаний, которое можно составить из n элементов, обозначают через C_n^k .

Формула для числа сочетаний получается из формулы для числа размещений. В самом деле, составим сначала все k -сочетания из n элементов, а потом переставим входящие в каждое сочетание элементы всеми возможными способами. При этом получается, что все k - размещений из n элементов, причем каждое только по одному разу. Но из каждого k - сочетания можно сделать $k!$ перестановок, а число этих сочетаний равно C_n^k . Значит справедлива формула

$$k!C_n^k = A_n^k.$$

Из этой формулы находим, что

$$C_n^k = \frac{A_n^k}{k!} = \frac{n!}{(n-k)!k!}.$$

Рекуррентные соотношения

При решении многих комбинаторных задач пользуются методом сведения данной задачи к задаче, касающейся меньшего числа предметов. Метод сведения к аналогичной задаче для меньшего числа предметов называется методом рекуррентных соотношений (от латинского "recurrere" - "возвращаться").

Понятие рекуррентных соотношений проиллюстрируем классической проблемой, которая была поставлена около 1202 года Леонардо из Пизы, известным как Фибоначчи. Важность чисел Фибоначчи для анализа комбинаторных алгоритмов делает этот пример весьма подходящим.

Фибоначчи поставил задачу в форме рассказа о скорости роста популяции кроликов при следующих предположениях. Все начинается с одной пары кроликов. Каждая пара становится фертильной через месяц,

после чего каждая пара рождает новую пару кроликов каждый месяц. Кролики никогда не умирают, и их воспроизводство никогда не прекращается.

Пусть F_n - число пар кроликов в популяции по прошествии n месяцев, и пусть эта популяция состоит из N_n пар приплода и O_n "старых" пар, то есть $F_n = N_n + O_n$. Таким образом, в очередном месяце произойдут следующие события: $O_{n+1} = O_n + N_n = F_n$. Старая популяция в $(n+1)$ -й момент увеличится на число родившихся в момент времени n . $N_{n+1} = O_n$. Каждая старая пара в момент времени n производит пару приплода в момент времени $(n+1)$. В последующий месяц эта картина повторяется:

$$\begin{aligned} O_{n+2} &= O_{n+1} + N_{n+1} = F_{n+1}, \\ N_{n+2} &= O_{n+1} \end{aligned}$$

Объединяя эти равенства, получим следующее рекуррентное соотношение:

$$\begin{aligned} F_{n+2} &= O_{n+2} + N_{n+2} = F_{n+1} + O_{n+1}, \\ F_{n+2} &= F_{n+1} + F_n \quad (7.1) \end{aligned}$$

Выбор начальных условий для последовательности чисел Фибоначчи не важен; существенное свойство этой последовательности определяется рекуррентным соотношением. Будем предполагать $F_0 = 0, F_1 = 1$ (иногда $F_0 = F_1 = 1$).

Рассмотрим эту задачу немного иначе.

Пара кроликов приносит раз в месяц приплод из двух крольчат (самки и самца), причем новорожденные крольчата через два месяца после рождения уже приносят приплод. Сколько кроликов появится через год, если в начале года была одна пара кроликов?

Из условия задачи следует, что через месяц будет две пары кроликов. Через два месяца приплод даст только первая пара кроликов, и получится 3 пары. А еще через месяц приплод дадут и исходная пара кроликов, и пара кроликов, появившаяся два месяца тому назад.

Поэтому всего будет 5 пар кроликов. Обозначим через $F(n)$ количество пар кроликов по истечении n месяцев с начала года. Ясно, что через $n + 1$ месяцев будут эти $F(n)$ пар и еще столько новорожденных пар кроликов, сколько было в конце месяца $n - 1$, то есть еще $F(n - 1)$ пар кроликов. Иными словами, имеет место рекуррентное соотношение

$$F(n + 1) = F(n) + F(n - 1) \quad (7.2)$$

Так как, по условию, $F(0) = 1$ и $F(1) = 2$, то последовательно находим

$$F(2) = 3, F(3) = 5, F(4) = 8$$

и т.д.

В частности, $F(12) = 377$.

Числа $F(n)$ называются числами Фибоначчи. Они обладают целым рядом замечательных свойств. Теперь выведем выражение этих чисел через C_m^k . Для этого установим связь между числами Фибоначчи и следующей комбинаторной задачей.

Найти число n последовательностей, состоящих из нулей и единиц, в которых никакие две единицы не идут подряд.

Чтобы установить эту связь, возьмем любую такую последовательность и сопоставим ей пару кроликов по следующему правилу: единицам соответствуют месяцы появления на свет одной из пар "предков" данной пары (включая и исходную), а нулями - все остальные месяцы. Например, последовательность 010010100010 устанавливает такую "генеалогию": сама пара появилась в конце 11-го месяца, ее родители - в конце 7-го месяца, "дед" - в конце 5-го месяца и "прадед" - в конце второго месяца. Исходная пара кроликов тогда зашифровывается последовательностью 000000000000.

Ясно, что при этом ни в одной последовательности не могут стоять две единицы подряд - только что появившаяся пара не может, по условию,

принести приплод через месяц. Кроме того, при указанном правиле различным последовательностям отвечают различные пары кроликов, и обратно, две различные пары кроликов всегда имеют разную "генеалогию", так как, по условию, крольчиха дает приплод, состоящий только из одной пары кроликов.

Установленная связь показывает, что число n -последовательностей, обладающих указанным свойством, равно $F(n)$.

Докажем теперь, что

$$F(n) = C_{n+1}^0 + C_n^1 + C_{n-1}^2 + \dots + C_{n-p+1}^p, \quad (7.3)$$

где $p = \frac{n+1}{2}$, если n нечетно, и $p = \frac{n}{2}$, если n четно. Иными словами, p - целая часть числа $\frac{n+1}{2}$ (в дальнейшем будем обозначать целую часть числа α через $E(\alpha)$; таким образом, $p = E(\frac{n+1}{2})$).

В самом деле, $F(n)$ - это число всех n - последовательностей из 0 и 1, в которых никакие две единицы не стоят рядом. Число же таких последовательностей, в которые входит ровно k единиц и $n - k$ нулей, равно C_{n-k+1}^k . Так как при этом должно выполняться неравенство $k \leq n - k + 1$, то k изменяется от 0 до $E(\frac{n+1}{2})$. Применяя правило суммы, приходим к соотношению (7.3).

Равенство (7.3) можно доказать и иначе. Положим $G(n) = C_{n+1}^0 + C_n^1 + C_{n-1}^2 + \dots + C_{n-p+1}^p$, где $p = \frac{n+1}{2}$. Из равенства $C_n^k = C_{n-1}^k + C_{n-1}^{k-1}$ легко следует, что

$$G(n) = G(n-1) + G(n-2). \quad (7.4)$$

Кроме того, ясно, что $G(1) = 2 = F(1)$ и $G(2) = 3 = F(2)$. Так как обе последовательности $F(n)$ и $G(n)$ удовлетворяют рекуррентному соотношению $X(n) = X(n-1) + X(n-2)$, то имеем

$$G(3) = G(2) + G(1) = F(2) + F(1) = F(3),$$

и, вообще, $G(n) = F(n)$.

Другой метод доказательства

В предыдущем разделе непосредственно установлена связь между задачей Фибоначчи и комбинаторной задачей. Эту связь можно установить и иначе, непосредственно доказав, что число $T(n)$ решений комбинаторной задачи удовлетворяет тому же рекуррентному соотношению

$$T(n+1) = T(n) + T(n-1), \quad (7.5)$$

что и числа Фибоначчи. В самом деле, возьмем любую $(n+1)$ -последовательность нулей и единиц, удовлетворяющую условию, что никакие две единицы не идут подряд. Она может оканчиваться или на 0, или на 1. Если она оканчивается на 0, то, отбросив его, получим n -последовательность, удовлетворяющую нашему условию. Если взять любую n -последовательность нулей и единиц, в которой подряд не идут две единицы, и приписать к ней нуль, то получим $(n+1)$ -последовательность с тем же свойством. Таким образом доказано, что число последовательностей, оканчивающихся на нуль, равно $T(n)$.

Пусть теперь последовательность оканчивается на 1. Так как двух единиц подряд быть не может, то перед этой единицей стоит нуль. Иными словами, последовательность оканчивается на 01. Остающаяся же после отбрасывания 0 и 1 $(n-1)$ -последовательность может быть любой, лишь бы в ней не шли подряд две единицы. Поэтому число последовательностей, оканчивающихся единицей, равно $T(n-1)$. Но каждая последовательность оканчивается или на 0, или на 1. В силу правила суммы получаем, что $T(n+1) = T(n) + T(n-1)$.

Таким образом, получено то же самое рекуррентное соотношение. Отсюда еще не вытекает, что числа $T(n)$ и $F(n)$ совпадают.

Процесс последовательных разбиений

Для решения комбинаторных задач часто применяют метод, использованный в предыдущем пункте: устанавливают для данной задачи рекуррентное соотношение и показывают, что оно совпадает с рекуррентным соотношением для другой задачи, решение которой нам уже известно. Если при этом совпадают и начальные члены последовательностей в достаточном числе, то обе задачи имеют одинаковые решения.

Применим описанный прием для решения следующей задачи.

Пусть дано некоторое множество из n предметов, стоящих в определенном порядке. Разобьем это множество на две непустые части так, чтобы одна из этих частей лежала левее второй (то есть, скажем, одна часть состоит из элементов от первого до m - го, а вторая - из элементов от $(m + 1)$ -го до n -го). После этого каждую из частей таким же образом разобьем на две непустые части (если одна из частей состоит уже из одного предмета, она не подвергается дальнейшим разбиениям). Этот процесс продолжается до тех пор, пока не получим части, состоящие из одного предмета каждая. Сколько существует таких процессов разбиения (два процесса считаются различными, если хотя бы на одном шагу они приводят к разным результатам)?

Обозначим число способов разбиения для множества из $n + 1$ предметов через B_n . На первом шагу это множество может быть разбито n способами (первая часть может содержать один предмет, два предмета, ..., n предметов). В соответствии с этим множество всех процессов разбиений распадается на n классов - в s - класс входят процессы, при которых первая часть состоит из s предметов.

Подсчитаем число процессов в s -м классе. В первой части содержится s элементов. Поэтому ее можно разбивать далее B_{s-1} различными процессами. Вторая же часть содержит $n - s + 1$ элементов, и ее можно разбивать далее B_{n-s} процессами. По правилу произведения получаем, что s - класс состоит из $B_{s-1} B_{n-s}$ различных процессов. По правилу суммы отсюда вытекает, что

$$B_n = B_0 B_{n-1} + B_1 B_{n-2} + \dots + B_{n-1} B_0. \quad (7.6)$$

Таким образом получено рекуррентное соотношение для B_n . Двоичный поиск, поиск делением пополам. Поиском по числам Фибоначчи называется поиск, основанный на том, что область поиска делится в точках, являющихся числами Фибоначчи.

Задача: "Затруднение мажордома"

Бывают комбинаторные задачи, в которых приходится составлять не одно рекуррентное соотношение, а систему соотношений, связывающую несколько последовательностей. Эти соотношения выражают $(n + 1)$ - у члены последовательностей через предыдущие члены не только данной, но и остальных последовательностей.

Задача: "Затруднение мажордома". Однажды мажордом короля Артура обнаружил, что к обеду за круглым столом приглашено 6 пар враждующих рыцарей. Сколькими способами можно рассадить их так, чтобы никакие два врага не сидели рядом?

Если мы найдем какой-то способ рассадки рыцарей, то, пересаживая их по кругу, получим еще 11 способов. Мы не будем сейчас считать различными способы, получающиеся друг из друга такой циклической пересадкой.

Введем следующие обозначения. Пусть число рыцарей равно $2n$. Через A_n обозначим число способов рассадки, при которых никакие два врага не сидят рядом. Через B_n обозначим число способов, при которых рядом сидит ровно одна пара врагов, и через C_n - число способов, при которых есть ровно две пары враждующих соседей.

Выведем сначала формулу, выражающую A_{n+1} через A_n, B_n, C_n . Пусть $n + 1$ пар рыцарей посажены так, что никакие два врага не сидят рядом. Мы будем считать, что все враждующие пары рыцарей занумерованы. Попросим встать из-за стола пару рыцарей с номером $n + 1$. Тогда возможны три случая: среди оставшихся за столом нет одной пары соседей- врагов, есть одна такая пара и есть две такие пары

(ушедшие рыцари могли разделять эти пары). Мы считаем, что $n > 1$. При $n = 1$ последующие рассуждения теряют силу.

Выясним теперь, сколькими способами можно снова посадить ушедших рыцарей за стол так, чтобы после этого не было одной пары соседей-врагов.

Проще всего посадить их, если за столом рядом сидят две пары врагов. В этом случае один из вновь пришедших садится между рыцарями первой пары, а другой – между рыцарями второй пары. Это можно сделать двумя способами. Но так как число способов рассадки $2n$ рыцарей, при которых две пары соседей оказались врагами, равно C_n , то всего получилось $2C_n$ способов.

Пусть теперь рядом сидит только одна пара врагов. Один из вернувшихся должен сесть между ними. Тогда за столом окажутся $2n + 1$ рыцарей, между которыми есть $2n + 1$ мест. Из них два места – рядом с только что севшим гостем – запретны для второго рыцаря, и ему остается $2n - 1$ мест. Так как первым может войти любой из двух вышедших рыцарей, то получается $2(2n - 1)$ способов рассадки. Но число случаев, когда $2n$ рыцарей сели так, чтобы ровно одна пара врагов оказалась соседями, равно B_n . Поэтому мы получаем $2(2n - 1)B_n$ способов посадить гостей требуемым образом.

Наконец, пусть никакие два врага не сидели рядом. В этом случае первый рыцарь садится между любыми двумя гостями – это он может сделать $2n$ способами. После этого для его врага останется $2n - 1$ мест – он может занять любое место, кроме двух мест, соседних с только что севшим рыцарем. Таким образом, если $2n$ рыцарей уже сидели нужным образом, то вернувшихся гостей можно посадить $2n(2n - 1)A_n$ способами. Как уже отмечалось, разработанными случаями исчерпываются все возможности. Поэтому имеет место рекуррентное соотношение

$$A_{n+1} = 2n(2n - 1)A_n + 2(2n - 1)B_n + 2C. \quad (7.7)$$

Этого соотношения пока недостаточно, чтобы найти A_n для всех

значений n . Надо еще узнать, как выражаются B_{n+1}, C_{n+1} через A_n, B_n, C_n .

Предположим, что среди $2n + 2, n > 1$ рыцарей оказалась ровно одна пара врагов-соседей. Мы знаем, что это может произойти в B_{n+1} случаях. Во избежание ссоры попросим их удалиться из-за стола. Тогда останется $2n$ рыцарей, причем возможно одно из двух: либо среди оставшихся нет врагов-соседей, либо есть ровно одна пара таких врагов - до ухода покинувших зал они сидели по обе стороны от них и теперь оказались рядом. Во втором случае ушедших можно посадить обратно только на старое место - иначе появится вторая пара враждующих соседей. Но так как $2n$ рыцарей можно посадить B_n способами так, чтобы была только одна пара враждующих соседей, то мы получаем $2B_n$ вариантов (возвратившихся рыцарей можно поменять местами). В первом же случае можно посадить ушедших между любыми двумя рыцарями, то есть $2n$ способами, а так как их еще можно поменять местами, то получится $4n$ способов. Комбинируя их со всеми способами посадки n пар рыцарей, при которых нет соседей врагов, получаем $4nA_n$ способов. Наконец, номер ушедшей и вернувшейся пары рыцарей мог быть любым от 1 до $n + 1$. Отсюда вытекает, что рекуррентное соотношение для B_{n+1} имеет вид

$$B_{n+1} = 4n(n + 1)A_n + 2(n + 1)B_n. \quad (7.8)$$

Наконец, разберем случай, когда среди $2n + 2$ рыцарей было две пары врагов-соседей. Номера этих пар можно выбрать $C_{n+1}^2 = \frac{n(n+1)}{2}$ способами. Заменяем каждую пару одним новым рыцарем, причем будем считать новых двух рыцарей врагами. Тогда за столом будут сидеть $2n$ рыцарей, причем среди них либо не будет ни одной пары врагов-соседей (если новые рыцари не сидят рядом), либо только одна такая пара.

Первый вариант может быть в A_n случаях. Вернуться к исходной компании мы можем 4 способами благодаря возможности изменить порядок рыцарей в каждой паре. Поэтому первый вариант приводит к $4C_{n+1}^2 A_n = 2n(n + 1)A_n$ способами.

Второй же вариант может быть в $\frac{1}{n}B_n$ случаях. Имеется B_n случаев, когда какая-нибудь пара врагов сидит рядом. Если указать, какая именно пара должна сидеть рядом, получим в n раз меньше случаев.

Здесь тоже можно вернуться к исходной компании 4 способами, и мы получаем всего $2(n+1)B_n$ способов. Отсюда вытекает, что при $n \geq 1$

$$C_{n+1} = 2n(n+1)A_n + 2(n+1)B_n. \quad (7.9)$$

Мы получили систему рекуррентных соотношений

$$A_{n+1} = 2n(2n-1)A_n + 2(2n-1)B_n + 2C \quad (7.10)$$

$$B_{n+1} = 4n(n+1)A_n + 2(n+1)B_n. \quad (7.11)$$

$$C_{n+1} = 2n(n+1)A_n + 2(n+1)B_n. \quad (7.12)$$

Они справедливы при $n \geq 2$. Но простой подсчет показывает, что $A_2 = 2, B_2 = 0, C_2 = 4$. Поэтому из соотношений 7.10-7.12 вытекает, что $A_3 = 32, B_3 = 48, C_3 = 24$. Продолжая далее, находим, что гостей можно посадить за стол требуемым образом $A_6 = 12771840$ способами.

Алгоритмы рекуррентных соотношений

Решение рекуррентных соотношений. Линейные рекуррентные соотношения с постоянными коэффициентами. Случай равных корней характеристического уравнения. Производящие функции.

Решение рекуррентных соотношений

Будем говорить, что рекуррентное соотношение имеет порядок k , если оно позволяет выразить $f(n+k)$ через $f(n), f(n+1), \dots, f(n+k-1)$. Например,

$$f(n+2) = f(n)f(n+1) - 3f^2(n+1) + 1$$

рекуррентное соотношение второго порядка, а

$$f(n+3) = 6f(n)f(n+2) + f(n+1)$$

- рекуррентное соотношение третьего порядка. Если задано рекуррентное соотношение k -го порядка, то ему удовлетворяет бесконечно много последовательностей. Дело в том, что первые k элементов последовательности можно задать совершенно произвольно - между ними нет никаких соотношений. Но если первые k элементов заданы, то все остальные элементы определяются совершенно однозначно - элемент $f(k+1)$ выражается в силу рекуррентного соотношения через $f(1), \dots, f(k)$ элемент $f(k+2)$ - через $f(2), \dots, f(k+1)$ и т.д. Пользуясь рекуррентным соотношением и начальными членами, можно один за другим выписывать члены последовательности, причем рано или поздно получим любой ее член. Однако при этом придется выписать и все предыдущие члены - ведь не узнав их, мы не узнаем и последующих членов. Но во многих случаях нужно узнать только один определенный член последовательности, а остальные не нужны. В этих случаях удобнее иметь явную формулу для n -го члена последовательности. Некоторая последовательность является решением данного рекуррентного соотношения, если при подстановке этой последовательности соотношение тождественно

выполняется. Например, последовательность

$$2, 4, 8, \dots, 2^n, \dots$$

является одним из решений рекуррентного соотношения

$$f(n+2) = 3f(n+1) - 2f(n).$$

В самом деле, общий член этой последовательности имеет вид $f(n) = 2^n$. Значит, $f(n+2) = 2^{n+2}$, $f(n+1) = 2^{n+1}$. Но при любом n имеет место тождество $2^{n+2} = 3 \cdot 2^{n+1} - 2 \cdot 2^n$. Поэтому 2^n является решением указанного соотношения. Решение рекуррентного соотношения k -го порядка называется общим, если оно зависит от k произвольных постоянных $C_1, C_2, \dots, C_k$ и путем подбора этих постоянных можно получить любое решение данного соотношения. Например, для соотношения

$$f(n+2) = 5f(n+1) - 6f(n) \quad (8.1)$$

общим решением будет

$$f(n) = C_1 2^n + C_2 3^n. \quad (8.2)$$

В самом деле, легко проверить, что последовательность (8.2) обращает (8.1) в тождество. Поэтому нам надо только показать, что любое решение нашего соотношения можно представить в виде (8.2). Но любое решение соотношения (8.1) однозначно определяется значениями $f(1)$ и $f(2)$. Поэтому нам надо доказать, что для любых чисел a и b найдутся такие значения C_1 и C_2 , что

$$2C_1 + 3C_2 = a$$

и

$$2^2 C_1 + 3^2 C_2 = b.$$

Но легко видеть, что при любых значениях a и b система уравнений

$$2C_1 + 3C_2 = a$$

$$4C_1 + 9C_2 = b$$

имеет решение. Поэтому (8.2) действительно является общим решением соотношения (8.1).

Линейные рекуррентные соотношения с постоянными коэффициентами

Для решения рекуррентных соотношений общих правил, вообще говоря, нет. Однако существует весьма часто встречающийся класс соотношений, решаемый единообразным методом. Это - рекуррентные соотношения вида

$$f(n+k) = a_1 f(n+k-1) + a_2 f(n+k-2) + \dots + a_k f(n), \quad (8.3)$$

где $a_1, a_2, \dots, a_k$ - некоторые числа. Такие соотношения называют линейными рекуррентными соотношениями с постоянными коэффициентами. Сначала рассмотрим, как решаются такие соотношения при $k=2$, то есть изучим соотношение вида

$$f(n+2) = a_1 f(n+1) + a_2 f(n). \quad (8.4)$$

Решение этих соотношений основано на следующих двух утверждениях.

1. Если $f_1(n)$ и $f_2(n)$ являются решениями рекуррентного соотношения (8.4), то при любых числах A и B последовательность $f(n) = Af_1(n) + Bf_2(n)$ также является решением этого соотношения. В самом деле, по условию, имеем

$$f_1(n+2) = a_1 f_1(n+1) + a_2 f_1(n)$$

$$f_2(n+2) = a_1 f_2(n+1) + a_2 f_2(n)$$

Умножим эти равенства на A и B соответственно и сложим полученные тождества. Получим, что

$$Af_1(n+2) + Bf_2(n+2) = a_1[Af_1(n+1) + Bf_2(n+1)] + a_2[Af_1(n) + Bf_2(n)].$$

А это означает, что $Af_1(n) + Bf_2(n)$ является решением соотношения (8.4).

2. Если r_1 является корнем квадратного уравнения

$$r^2 = a_1r + a_2,$$

то последовательность

$$1, r_1, r_1^2, \dots, r_1^{n-1}, \dots$$

является решением рекуррентного соотношения

$$f(n+2) = a_1f(n+1) + a_2f(n).$$

В самом деле, если $f(n) = r_1^{n-1}$, то $f(n+1) = r_1^n$ и $f(n+2) = r_1^{n+1}$. Подставляя эти значения в соотношение (8.4), получаем равенство

$$1^{n+1} = a_1r_1^n + a_2r_1^{n-1}.$$

Оно справедливо, так как по условию имеем $r^2 = a_1r + a_2$.

Заметим, что наряду с последовательностью $\{r_1^{n-1}\}$ любая последовательность вида

$$f(n) = r_1^{n+m}, n = 1, 2, \dots$$

также является решением соотношения (8.4). Для доказательства достаточно использовать утверждение (8.4), положив в нем

$$A = r_1^{m+1}, B = 0.$$

Из утверждений 1 и 2 вытекает следующее правило решения линейных рекуррентных соотношений второго порядка с постоянными коэффициентами.

Пусть дано рекуррентное соотношение

$$f(n+2) = a_1 f(n+1) + a_2 f(n) \quad (8.5)$$

Составим квадратное уравнение

$$r^2 = a_1 r + a_2, \quad (8.6)$$

которое называется характеристическим для данного соотношения. Если это уравнение имеет два различных корня r_1, r_2 , то общее решение соотношения (8.5) имеет вид

$$f(n) = C_1 r_1^{n-1} + C_2 r_2^{n-1}.$$

Чтобы доказать это правило, заметим сначала, что по утверждению 2 $f_1(n) = r_1^{n-1}, f_2(n) = r_2^{n-1}$ являются решениями нашего соотношения. А тогда по утверждению 1 и $C_1 r_1^{n-1} + C_2 r_2^{n-1}$ является его решением. Надо только показать, что любое решение соотношения (8.5) можно записать в этом виде. Но любое решение соотношения второго порядка определяется значениями $f(1), f(2)$. Поэтому достаточно показать, что система уравнений

$$\begin{aligned} C_1 + C_2 &= a, \\ C_1 r_1 + C_2 r_2 &= b \end{aligned}$$

имеет решение при любых a, b . Этими решениями являются

$$\begin{aligned} C_1 &= \frac{b - ar_2}{r_1 - r_2}, \\ C_2 &= \frac{ar_1 - b}{r_1 - r_2}. \end{aligned}$$

(Случай, когда оба корня уравнения (8.6) совпадут друг с другом, разберем в следующем пункте.)

Пример на доказанное правило.

При изучении чисел Фибоначчи мы пришли к рекуррентному

соотношению

$$f(n) = f(n-1) + f(n-2). \quad (8.7)$$

Для него характеристическое уравнение имеет вид

$$r^2 = r + 1.$$

Корнями этого квадратного уравнения являются числа

$$r_1 = \frac{1 + \sqrt{5}}{2}, r_2 = \frac{1 - \sqrt{5}}{2}.$$

Поэтому общее решение соотношения Фибоначчи имеет вид

$$f(n) = C_1 \left(\frac{1 + \sqrt{5}}{2} \right)^n + C_2 \left(\frac{1 - \sqrt{5}}{2} \right)^n. \quad (8.8)$$

(Мы воспользовались сделанным выше замечанием и взяли показатели n вместо $n-1$). Мы называли числами Фибоначчи решения соотношения (8.7), удовлетворяющее начальным условиям $f(0) = 1, f(1) = 2$ (то есть последовательность 1, 2, 3, 5, 8, 13, ...). Часто бывает более удобно добавить к этой последовательности вначале числа 0 и 1, то есть рассматривать последовательность 0, 1, 1, 2, 3, 5, 8, 13, ... Ясно, что эта последовательность удовлетворяет тому же самому рекуррентному соотношению (8.6) и начальным условиям $f(0) = 1, f(1) = 2$. Полагая в формуле (8.8) $n = 0, n = 1$, получаем для C_1, C_2 систему уравнений

$$\begin{aligned} C_1 + C_2 &= 0 \\ \frac{\sqrt{5}}{2}(C_1 - C_2) &= 1 \end{aligned}$$

Отсюда находим, что $C_1 = -C_2 = \frac{1}{\sqrt{5}}$ и потому

$$f(n) = \frac{1}{\sqrt{5}} \left[\left(\frac{1 + \sqrt{5}}{2} \right)^n - \left(\frac{1 - \sqrt{5}}{2} \right)^n \right]. \quad (8.9)$$

На первый взгляд кажется удивительным, что это выражение при всех натуральных значениях n принимает целые значения.

Случай равных корней характеристического уравнения

Рассмотрим случай, когда оба корня характеристического уравнения совпадают: $r_1 = r_2$. В этом случае выражение $C_1 r_1^{n-1} + C_2 r_2^{n-1}$ уже не будет общим решением. Ведь из-за того, что $r_1 = r_2$, это решение можно записать в виде

$$f(n) = (C_1 + C_2) r_1^{n-1} = C r_1^{n-1}.$$

Остается только одно произвольное постоянное ; и выбрать его так, чтобы удовлетворить двум начальным условиям $f(1) = a$, $f(2) = b$, вообще говоря, невозможно. Поэтому надо найти какое-нибудь второе решение, отличное от $f_1(n) = r_1^{n-1}$. Таким решением является $f_2(n) = n r_1^{n-1}$. В самом деле, если квадратное уравнение $r^2 = a_1 r + a_2$ имеет два совпадающих корня $r_1 = r_2$, то по теореме Виета $a_1 = 2r_1$, $a_2 = -r_1^2$. Поэтому уравнение записывается так:

$$r^2 = 2r_1 r - r_1^2.$$

А тогда рекуррентное соотношение имеет такой вид:

$$f(n+2) = 2r_1 f(n+1) - r_1^2 f(n). \quad (8.10)$$

Проверим, что $f_2(n) = n r_1^{n-1}$ действительно являются его решением. Имеем $f_2(n+2) = (n+2) r_1^{n+1}$, а $f_2(n+1) = (n+1) r_1^n$. Подставляя эти значения в соотношение (8.10), получаем очевидное тождество

$$(n+2) r_1^{n+1} = 2(n+1) r_1^{n+1} - n r_1^{n+1}.$$

Значит, nr_1^{n-1} - решение рассматриваемого соотношения.

Итак, имеются два решения $f_1(n) = r_1^{n-1}$ и $f_2(n) = nr_1^{n-1}$ заданного соотношения. Его общее решение запишется так:

$$f(n) = C_1 r_1^{n-1} + C_2 nr_1^{n-1} = r_1^{n-1}(C_1 + C_2 n).$$

Теперь уже путем подбора C_1, C_2 можно удовлетворить любым начальным условиям. Линейные рекуррентные соотношения с постоянными коэффициентами, порядок которых больше двух, решаются таким же способом. Пусть соотношение имеет вид

$$f(n+k) = a_1 f(n+k-1) + \dots + a_n f(n). \quad (8.11)$$

Составим характеристическое уравнение

$$r^k = a_1 r^{k-1} + \dots + a_k.$$

Если все корни $r_1, r_2, \dots, r_k$ этого алгебраического уравнения k -й степени различны, то общее решение соотношения (8.3) имеет вид

$$f(n) = C_1 r_1^{n-1} + C_2 r_2^{n-1} + \dots + C_k r_k^{n-1}.$$

Если же, например, $r_1 = r_2 = \dots = r_s$, то этому корню соответствуют решения

$$\begin{aligned} f_1(n) &= r_1^{n-1}, f_2(n) = nr_1^{n-1}, \\ f_3(n) &= n^2 r_1^{n-1}, \dots, f_s(n) = n^{s-1} r_1^{n-1} \end{aligned}$$

рекуррентного соотношения (8.11). В общем решении этому корню соответствует часть

$$r_1^{n-1} [C_1 + C_2 n + C_3 n^2 + \dots + C_s n^{s-1}]$$

Составляя такое выражение для всех корней и складывая их, получаем общее решение соотношения (8.3).

Например, решим рекуррентное соотношение

$$f(n+4) = 5f(n+3) - 6f(n+2) - 4f(n+1) + 8f(n).$$

Характеристическое уравнение в этом случае имеет вид

$$r^4 - 5r^3 + 6r^2 + 4r - 8 = 0.$$

Решая его, получим корни

$$r_1 = 2, r_2 = 2, r_3 = 2, r_4 = -1.$$

Значит, общее решение нашего соотношения имеет следующий вид:

$$f(n) = 2^{n-1}[C_1 + C_2n + C_3n^2] + C_4(-1)^{n-1}.$$

Производящие функции

В комбинаторных задачах на подсчет числа объектов при наличии некоторых ограничений искомым решением часто является последовательность $a_0, a_1, a_2, \dots$, где a_k - число искомых объектов "размерности" k . Например, если мы ищем число разбиений числа, то можем принять $a_k = P(k)$, если ищем число подмножеств n - элементного множества, то $a_k = \binom{n}{k}$ и т.д. В этом случае удобно последовательности $a_0, a_1, a_2, \dots$, поставить в соответствие формальный ряд

$$A(x) = \sum_{k=0}^{\infty} a_k x^k, \quad (8.12)$$

называемый производящей функцией для данной последовательности. Название формальный ряд для данной последовательности означает, что (8.12) мы трактуем только как удобную запись нашей последовательности - в данном случае несущественно, для каких (действительных или комплексных) значений переменной x он сходится. Поэтому мы никогда не будем вычислять значение такого ряда для конкретного значения переменной x , мы будем только выполнять некоторые операции на таких рядах, а затем определять коэффициенты при отдельных степенях переменной x . Для произвольных рядов

$$A(x) = \sum_{k=0}^{\infty} a_k x^k, B(x) = \sum_{k=0}^{\infty} b_k x^k$$

мы определим операцию сложения:

$$A(x) + B(x) = \sum_{k=0}^{\infty} (a_k + b_k) x^k, \quad (8.13)$$

операцию умножения на число (действительное или комплексное):

$$pA(x) = \sum_{k=0}^{\infty} pa_k x^k \quad (8.14)$$

и произведение Коши

$$A(x) \cdot B(x) = \sum_{k=0}^{\infty} c_k x^k, \quad (8.15)$$

где

$$c_k = a_0 b_k + a_1 b_{k-1} + \dots + a_k b_0 = \sum_{i=0}^k a_i b_{k-i}, \quad (8.16)$$

Если $a_k = 0$ для $k > n$, то ряд (8.12) будем отождествлять с многочленом $a_n x^n + \dots + a_0$. Из математического анализа известно, что если ряд (8.12) сходится в некоторой окрестности нуля, то его сумма $A(x)$ является аналитической функцией в этой окрестности и

$$a_k = A^{(k)}(0)/k!, \quad k = 0, 1, 2, \dots \quad (8.17)$$

($A^{(k)}(0)$ обозначает значение k -й производной функции $A(x)$ для $x = 0$; ряд 8.12 - это не что иное, как ряд Маклорена функции $A(x)$). Более того, когда $A(x), B(x)$ являются аналитическими функциями в

окрестности нуля, то формулы (8.13)-(8.16) будут справедливы, если $A(x), B(x)$ трактовать как значения функций A, B в точке x , а ряды понимать в обычном смысле, т.е. так, как в математическом анализе. Это сохраняющее операции взаимно однозначное соответствие между рядами, сходящимися в окрестности нуля, и функциями, аналитическими в окрестности нуля, позволяет отождествить формальный ряд (8.12) с определенной через него аналитической функцией в случае рядов, сходящихся в окрестности нуля (несмотря на то, что ряды мы будем трактовать всегда как формальные ряды, то есть только как формальную запись их коэффициентов). Таким образом, будем писать, например,

$$\sum_{k=0}^{\infty} x^k = (1 - x)^{-1},$$
$$\sum_{k=0}^{\infty} \frac{1}{k!} x^k = e^x$$

и т.д.

Комбинаторика и ряды

Введение. Деление многочленов. Алгебраические дроби и степенные ряды. Действия над степенными рядами.

Введение

Метод рекуррентных соотношений позволяет решать многие комбинаторные задачи. Но в целом ряде случаев рекуррентные соотношения довольно трудно составить, а еще труднее решить. Зачастую эти трудности удается обойти, используя производящие функции. Поскольку понятие производящей функции связано с бесконечными степенными рядами, познакомимся с этими рядами.

Деление многочленов

Если заданы два многочлена $f(x)$ и $\varphi(x)$, то всегда существуют многочлены $q(x)$ (частное) и $r(x)$ (остаток), такие, что $f(x) = \varphi(x)q(x) + r(x)$, причем степень $r(x)$ меньше степени $\varphi(x)$ или $r(x) = 0$. При этом $f(x)$ называется делимым, а $\varphi(x)$ - делителем. Если же мы хотим, чтобы деление выполнялось без остатка, то придется допустить в качестве частного не только многочлены, но и бесконечные степенные ряды. Для получения частного надо расположить многочлены по возрастающим степеням x и делить "углом", начиная с младших членов. Рассмотрим, например, деление 1 на $1 - x$

$$\begin{array}{r}
 1 \\
 \hline
 \mp 1 \pm x \\
 \hline
 x \\
 \mp x \pm x^2 \\
 \hline
 x^2 \\
 \mp x^2 \pm x^3 \\
 \hline
 x^3 \dots
 \end{array}
 \begin{array}{l}
 | 1 - x \\
 \hline
 | 1 + x + x^2 + \dots
 \end{array}$$

Ясно, что процесс деления никогда не закончится (так же, например,

как при обращении числа $\frac{1}{3}$ в бесконечную десятичную дробь). С помощью индукции легко убедиться, что все коэффициенты частного равны единице. Поэтому в качестве частного получается бесконечный ряд $1 + x + x^2 + \dots + x^n + \dots$. Вообще, если $f(x)$ и $\varphi(x)$ - два многочлена

$$\begin{aligned} f(x) &= a_0 + \dots + a_n x^n, \\ \varphi(x) &= b_0 + \dots + b_m x^m, \end{aligned}$$

причем свободный член b_0 многочлена $\varphi(x)$ отличен от нуля, $b_0 \neq 0$, то при делении $f(x)$ на $\varphi(x)$ получается бесконечный ряд

$$c_0 + c_1 x + \dots + c_k x^k + \dots \quad (9.1)$$

Лишь в случае, когда $f(x)$ делится без остатка на $\varphi(x)$, ряд (9.1) обрывается и мы получаем многочлен.

Алгебраические дроби и степенные ряды

При делении многочлена $f(x)$ на многочлен $\varphi(x)$ мы получаем бесконечный степенной ряд. Возникает вопрос: как связан этот ряд с алгебраической дробью $\frac{f(x)}{\varphi(x)}$, то есть какой смысл можно придать записи

$$\frac{f(x)}{\varphi(x)} = c_0 + c_1 x + \dots + c_n x^n + \dots \quad (9.2)$$

Рассмотрим, например, разложение

$$\frac{1}{1-x} \cong 1 + x + x^2 + \dots + x^n + \dots \quad (9.3)$$

Мы не пишем здесь знака равенства, так как не знаем, какой смысл имеет стоящая справа сумма бесконечного числа слагаемых. Чтобы

выяснить это, попробуем подставлять в обе части соотношения (9.3) различные значения x . Сначала положим $x = \frac{1}{10}$. Тогда левая часть соотношения примет значение $\frac{10}{9}$, а правая превратится в бесконечный числовой ряд $1 + 0,1 + 0,01 + \dots + 0.000\dots01 + \dots$. Так как мы не умеем складывать бесконечно много слагаемых, попробуем взять сначала одно слагаемое, потом - два, потом - три и так далее слагаемых. Мы получим такие суммы: $1; 1,1; 1,11; \dots; 1,111\dots1;n$. Ясно, что с возрастанием n эти суммы приближаются к значению $\frac{10}{9} = 1,11\dots$, которое приняла левая часть соотношения (9.3) при $x = \frac{1}{10}$.

То же самое получится, если вместо x подставить в обе части (9.3) число $\frac{1}{2}$. Левая часть равенства примет значение 2, а правая превратится в бесконечный числовой ряд $1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots + \frac{1}{2^n} + \dots$. Беря последовательно одно, два, три, четыре, слагаемых, мы получим числа $1; 1\frac{1}{2}; 1\frac{3}{4}; 1\frac{7}{8}, \dots, 2 - \frac{1}{2^n}$. Ясно, что с возрастанием n эти числа стремятся к числу 2.

Однако, если взять $x = 4$, то левая часть (9.3) примет значение $-\frac{1}{3}$, а в правой получим ряд $1 + 4 + 4^2 + \dots + 4^n + \dots$. Если последовательно складывать члены этого ряда, то получаются суммы $1; 5; 21; 85; \dots$. Эти суммы неограниченно увеличиваются и не приближаются к числу $-\frac{1}{3}$.

Мы встретились, таким образом, с двумя случаями. Чтобы их различать, введем общее понятие о сходимости и расходимости числового ряда. Пусть задан бесконечный числовой ряд

$$a_1 + a_2 + \dots + a_n + \dots \quad (9.4)$$

Говорят, что бесконечный числовой ряд сходится к числу b , если разность $b - (a_1 + a_2 + \dots + a_n)$ стремится к нулю при неограниченном увеличении n . Иными словами, какое бы число $\varepsilon > 0$ мы ни указали, отклонение суммы $a_1 + \dots + a_n$ от b , начиная

с некоторого номера N , окажется меньше ε :

$$|b - (a_1 + \dots + a_n)| < \varepsilon \text{ если } n \geq N.$$

В этом случае число b называют суммой бесконечного ряда $a_1 + a_2 + \dots + a_n + \dots$ и пишут

$$b = a_1 + a_2 + \dots + a_n + \dots$$

Если не существует числа b , к которому сходится данный ряд (9.4), то этот ряд называют расходящимся. Проведенное выше исследование показывает, что

$$\begin{aligned} \frac{10}{9} &= 1 + 0,1 + 0,01 + \dots + 0,00\dots01 + \dots, \\ 2 &= 1 + \frac{1}{2} + \frac{1}{4} + \dots + \frac{1}{2^n} + \dots, \end{aligned}$$

в то время как ряд $1 + 4 + 16 + \dots + 4^n + \dots$ расходится. Более тщательное исследование показывает, что если $|x| < 1$, то ряд $1 + x + \dots + x^n + \dots$ сходится к $\frac{1}{1-x}$, а если $|x| > 1$, то он расходится. Чтобы доказать это утверждение, достаточно заметить, что

$$1 + x + \dots + x^n = \frac{1 - x^{n+1}}{1 - x}$$

и что при $n \rightarrow \infty$ выражение x^{n+1} стремится к нулю, если $|x| < 1$, и к бесконечности, если $|x| \geq 1$. При $x = \pm 1$ получаем расходящиеся числовые ряды $1 + 1 + \dots + 1 + \dots$ и $1 - 1 + \dots + 1 - 1 + \dots$. Итак, если $|x| < 1$, то

$$\frac{1}{1-x} = 1 + x + \dots + x^n + \dots \quad (9.5)$$

Отметим, что равенство (9.5) - это известная из школьного курса математики формула для суммы бесконечно убывающей геометрической прогрессии.

Мы выяснили, таким образом, смысл записи

$$\frac{1}{1-x} = 1 + x + \dots + x^n + \dots$$

Она показывает, что для значений x , лежащих в некоторой области, а именно при $|x| < 1$, стоящий справа ряд сходится к $\frac{1}{1-x}$. Говорят, что функция $\frac{1}{1-x}$ при $|x| < 1$ разлагается в степенной ряд $1 + x + \dots + x^n + \dots$. Теперь уже можно выяснить и более общий вопрос. Пусть при делении многочлена $f(x)$ на многочлен $\varphi(x)$ получился степенной ряд

$$c_0 + c_1 x + \dots + c_n x^n + \dots \quad (9.6)$$

Оказывается, что тогда при достаточно малых значениях x ряд (9.6) сходится к $f(x)/\varphi(x)$. Размеры области сходимости зависят от корней знаменателя, то есть чисел, при которых знаменатель обращается в нуль. Именно, если эти числа равны $x_1, \dots, x_k$ и r - наименьшее из чисел $|x_1|, \dots, |x_k|$, то ряд сходится в области $|x| < r$. Иными словами, всегда есть область $|x| < r$, в которой выполняется равенство

$$\frac{f(x)}{\varphi(x)} = c_0 + c_1 x + \dots + c_n x^n + \dots \quad (9.7)$$

В степенные ряды можно разлагать не только алгебраические дроби, но и многие другие функции. В математическом анализе доказывают, например, что

$$\begin{aligned} \sin x &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots, \\ \cos x &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \dots, \\ e^x &= 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots \end{aligned}$$

Отметим еще следующее важное утверждение: функция $f(x)$ не может иметь двух различных разложений в степенные ряды.

Действия над степенными рядами

Перейдем теперь к действиям над степенными рядами. Пусть функции $f(x)$ и $\varphi(x)$ разложены в степенные ряды

$$f(x) = a_0 + a_1x + \dots a_nx^n + \dots \quad (9.12)$$

$$\varphi(x) = b_0 + b_1x + \dots b_nx^n \dots \quad (9.13)$$

Тогда

$$f(x) + \varphi(x) = (a_0 + a_1x + \dots a_nx^n + \dots) + (b_0 + b_1x + \dots b_nx^n \dots).$$

Оказывается, что слагаемые в правой части равенства можно переставить и сгруппировать вместе члены с одинаковыми степенями x . Это утверждение совсем не так очевидно, как кажется на первый взгляд. Ведь в правой части равенства у нас бесконечные суммы, а в бесконечных суммах переставлять слагаемые можно далеко не всегда. После этой перегруппировки мы получим

$$f(x) + \varphi(x) = (a_0 + b_0) + (a_1 + b_1)x + \dots + (a_n + b_n)x^n + \dots$$

Ряд, стоящий в правой части равенства (9.14), называется суммой степенных рядов (9.12) и (9.13). Посмотрим теперь, как разлагается в степенной ряд произведение функций $f(x)$ и $\varphi(x)$. Мы имеем

$$f(x)\varphi(x) = (a_0 + a_1x + \dots a_nx^n + \dots) \times (b_0 + b_1x + \dots b_nx^n \dots).$$

Оказывается, что как и в случае многочленов, ряды, стоящие в правой части равенства (9.15), можно почленно перемножать. Мы опускаем доказательство этого утверждения. Найдем ряд, получающийся после почленного перемножения. Свободный член этого ряда равен a_0b_0 . Члены, содержащие x , получатся дважды: при умножении a_0 на b_1x и при умножении a_1x на b_0 . Они дают

$$a_0 b_1 x + a_1 b_0 x = (a_0 b_1 + a_1 b_0) x.$$

Точно так же вычисляются члены, содержащие x^2 . Таким образом,

$$f(x)\varphi(x) = a_0 b_0 + (a_0 b_1 + a_1 b_0)x + \dots + (a_0 b_n + \dots + a_n b_0)x^n + \dots$$

Ряд, стоящий в правой части равенства (9.16), называется произведением рядов (9.12) и (9.13). В частности, возводя ряд (9.12) в квадрат, получаем

$$f^2(x) = a_0^2 + 2a_0 a_1 x + (a_1^2 + 2a_0 a_2)x^2 + 2(a_0 a_3 + a_1 a_2)x^3 + \dots$$

Посмотрим теперь, как делят друг на друга степенные ряды. Пусть свободный член ряда (9.13) отличен от нуля. Покажем, что в этом случае существует такой степенной ряд

$$c_0 + c_1 x + \dots + c_n x^n + \dots, \quad (9.18)$$

что

$$(b_0 + b_1 x + \dots + b_n x^n + \dots) \times (c_0 + c_1 x + \dots + c_n x^n + \dots) = a_0 + a_1 x + \dots + a_n x^n + \dots$$

Для доказательства перемножим ряды в левой части этого равенства. Мы получим ряд

$$b_0 c_0 + (b_0 c_1 + b_1 c_0)x + \dots + (b_0 c_n + \dots + b_n c_0)x^n + \dots$$

Для того чтобы этот ряд совпадал с рядом (9.12), необходимо и достаточно, чтобы выполнялись равенства

$$\begin{aligned} b_0 c_0 &= a_0, \\ \underbrace{b_0 c_1 + b_1 c_0}_{\dots\dots\dots} &= a_1, \\ \underbrace{b_0 c_n + \dots + b_n c_0}_{\dots\dots\dots} &= a_n, \end{aligned}$$

Эти равенства дают бесконечную систему уравнений для отыскания коэффициентов. Из первого уравнения системы получаем $c_0 = \frac{a_0}{b_0}$.

Подставим полученное значение во второе уравнение. Мы получим уравнение

$$b_0 c_1 = a_1 - \frac{b_1 a_0}{b_0},$$

из которого находим, что $c_1 = \frac{a_1 b_0 - b_1 a_0}{b_0^2}$. Вообще, если уже найдены коэффициенты $c_0, \dots, c_{n-1}$, то для отыскания c_n имеем уравнение

$$b_0 c_n = a_n - b_1 c_{n-1} - \dots - b_n c_0.$$

Это уравнение разрешимо, поскольку $b_0 \neq 0$. Итак, мы доказали существование ряда (9.18), удовлетворяющего соотношению (9.19). Ряд (9.18) называют частным при делении рядов (9.12) и (9.13). Можно доказать, что он получается при разложении функции $f(x)/\varphi(x)$. Таким образом, степенные ряды можно складывать, умножать и делить (последнее - при условии, что свободный член делителя отличается от нуля). Эти действия соответствуют действиям над разлагаемыми функциями.

Производящие функции и рекуррентные соотношения

Применение степенных рядов для доказательства тождеств. Производящие функции. Бином Ньютона. Ряд Ньютона. Производящие функции и рекуррентные соотношения. О едином нелинейном рекуррентном соотношении.

Применение степенных рядов для доказательства тождеств

С помощью степенных рядов можно доказывать многие тождества. Для этого берут некоторую функцию и двумя способами разлагают ее в степенной ряд. Поскольку функция может быть представлена лишь единственным образом в виде степенного ряда, то коэффициенты при одинаковых степенях x в обоих рядах должны совпадать. Это и приводит к доказываемому тождеству.

Рассмотрим, например, известное нам разложение

$$\frac{1}{1-x} = 1 + x + x^2 + \dots + x^n + \dots$$

Возведя обе части этого разложения в квадрат, получаем

$$\frac{1}{(1-x)^2} = 1 + 2x + 3x^2 + \dots + (n+1)x^n + \dots \quad (10.1)$$

Если заменить здесь x на $-x$, то получим, что

$$\frac{1}{(1+x)^2} = 1 - 2x + 3x^2 - \dots + (-1)^n(n+1)x^n + \dots \quad (10.2)$$

Перемножив разложения (10.1) и (10.2), выводим, что

$$\begin{aligned} \frac{1}{(1-x)^2} \frac{1}{(1+x)^2} &= 1 + [1(-2) + 2 \cdot 1]x + [1 \cdot 3 + 2(-2) + 3 \cdot 1]x^2 + \\ &\dots + [1(-1)^n(n+1) + 2(-1)^{n-2}n + \dots + (-1)^n(n+1) \cdot 1]x^n + \dots \end{aligned}$$

Очевидно, что коэффициенты при нечетных степенях x обращаются в нуль, каждое слагаемое дважды входит в эти коэффициенты с противоположными знаками. Коэффициент же x^{2n} равен

$$1(2n+1) - 2 \cdot 2n + 3(2n-1) - \dots + (2n+1).$$

Но функцию $\frac{1}{(1-x)^2(1+x)^2}$ можно разложить в степенной ряд и иным образом. Мы имеем

$$\frac{1}{(1-x)^2(1+x)^2} = \frac{1}{(1-x^2)^2},$$

А разложение для $\frac{1}{(1-x^2)^2}$ получается из разложения (10.1), если заменить в нем x на x^2 :

$$\frac{1}{(1-x^2)^2} = 1 + 2x^2 + 3x^4 + \dots + (n+1)x^{2n} + \dots \quad (10.4)$$

Мы знаем, что никакая функция не может иметь двух различных разложений в степенные ряды. Поэтому коэффициент при x^{2n} в разложении (10.3) должен равняться коэффициенту при x^{2n} в разложении (10.4). Отсюда вытекает следующее тождество:

$$1(2n+1) - 2 \cdot 2n + 3(2n-1) - \dots + (2n+1) = n+1.$$

Производящие функции

Пусть дана некоторая последовательность чисел $a_0, a_1, \dots, a_n, \dots$. Образует степенной ряд

$$a_0 + a_1x + \dots + a_nx^n + \dots$$

Если этот ряд сходится в какой-то области к функции $f(x)$, то эту функцию называют производящей для последовательности чисел $a_0, a_1, \dots, a_n, \dots$. Например, из формулы

$$\frac{1}{1-x} = 1 + x + \dots + x^n + \dots$$

вытекает, что функция $\frac{1}{1-x}$ является производящей для последовательности чисел. А формула (10.1) показывает, что для последовательности чисел $1, 2, 3, 4, \dots, n, \dots$ производящей является функция $\frac{1}{(1-x)^2}$.

Нас будут интересовать производящие функции для последовательностей $a_0, a_1, \dots, a_n, \dots$, так или иначе связанных с комбинаторными задачами. С помощью таких функций удастся получить самые разные свойства этих последовательностей. Кроме того, мы рассмотрим, как связаны производящие функции с решением рекуррентных соотношений.

Бином Ньютона

Получим производящую функцию для конечной последовательности чисел $C_n^0, C_n^1, \dots, C_n^n$. Известно, что

$$(a+x)^2 = a^2 + 2ax + x^2$$

и

$$(a+x)^3 = a^3 + 3a^2x + 3ax^2 + x^3.$$

Эти равенства являются частными случаями более общей формулы, дающей разложение для $(a+x)^n$. Запишем $(a+x)^n$ в виде

$$(a+x)^n = \underbrace{(a+x)(a+x)\dots(a+x)}_{n \text{ раз}}. \quad (10.4)$$

Раскроем скобки в правой части этого равенства, причем будем записывать все множители в том порядке, в котором они нам встретятся. Например, $(a+x)^2$ запишем в виде

$$(a + x)^2 = (a + x)(a + x) = aa + ax + xa + xx, \quad (10.5)$$

а $(a + x)^3$ - в виде

$$\begin{aligned} (a + x)^3 &= (a + x)(a + x)(a + x) = \\ &= aaa + aax + axa + axx + xaa + xax + xxa + xxx. \end{aligned} \quad (10.6)$$

Видно, что в формулу (10.5) входят все размещения с повторениями, составленные из букв x и a по две буквы в каждом размещении, а в формулу (10.6) - размещения с повторениями из тех же букв, но состоящие из трех букв каждое. То же самое и в общем случае — после раскрытия скобок в формуле (10.4) мы получим всевозможные размещения с повторениями букв x и a , состоящие из n элементов. Приведем подобные члены. Подобными будут члены, содержащие одинаковое количество букв x (тогда и букв a в них будет поровну). Найдем, сколько будет членов, в которые входит k букв x и, следовательно, $n - k$ букв a . Эти члены являются перестановками с повторениями, составленными из k букв x и $n - k$ букв a . Поэтому их число равно

$$P(k, n - k) = C_n^k = \frac{n!}{k!(n - k)!}.$$

Отсюда вытекает, что после приведения подобных членов выражение $x^k a^{n-k}$ войдет с коэффициентом $C_n^k = \frac{n!}{k!(n-k)!}$. Итак, мы доказали, что

$$(a + x)^n = C_n^0 a^n + C_n^1 a^{n-1} x + \dots + C_n^k a^{n-k} x^k + \dots + C_n^n x^n.$$

Равенство (10.7) принято называть формулой бинома Ньютона. Если положить в этом равенстве $a = 1$, то получим

$$(1 + x)^n = C_n^0 + C_n^1 x + \dots + C_n^k x^k + \dots + C_n^n x^n. \quad (10.8)$$

Мы видим, что $(1 + x)^n$ является производящей функцией для чисел C_n^k , $k = 0, 1, \dots$. С помощью этой производящей функции можно

сравнительно просто доказать многие свойства чисел C_n^k .

Ряд Ньютона

Мы назвали, как это обычно делают, формулу $(a+x)^n$ биномом Ньютона. Это наименование с точки зрения истории математики неверно. Формулу для $(a+x)^n$ хорошо знали среднеазиатские математики Омар Хайям, Гиясэдди и другие. В Западной Европе задолго до Ньютона она была известна Блэзу Паскалю. Заслуга же Ньютона была в ином - ему удалось обобщить формулу $(a+x)^n$ на случай нецелых показателей. Именно, он доказал, что если a - положительное число и $|x| < a$, то для любого действительного значения α имеет место равенство

$$\begin{aligned} (x+a)^\alpha = a^\alpha + \alpha a^{\alpha-1}x + \frac{\alpha(\alpha-1)}{1 \cdot 2} a^{\alpha-2}x^2 + \dots \\ \dots + \frac{\alpha(\alpha-1) \dots (\alpha-k+1)}{1 \cdot 2 \dots k} a^{\alpha-k}x^k + \dots \end{aligned} \quad (10.9)$$

Только теперь получилось не конечное число слагаемых, а бесконечный ряд. В случае, когда n - натуральное число, $(n-n)$ обращается в нуль. Но эта скобка входит в коэффициент всех членов, начиная с $(n+2)$ - го, и потому все эти члены разложения равны нулю. Поэтому при натуральном n ряд (10.9) превращается в конечную сумму.

Производящие функции и рекуррентные соотношения

Теория производящих функций тесно связана с рекуррентными соотношениями. Вернемся снова к делению многочленов. Пусть функции $f(x)$ и $\varphi(x)$ разложены в степенные ряды,

$$\begin{aligned} f(x) &= a_0 + a_1x + \dots a_nx^n + \dots \\ \varphi(x) &= b_0 + b_1x + \dots b_nx^n \dots \end{aligned}$$

- два многочлена, причем $b_0 \neq 0$. Мы будем, кроме того, предполагать, что $n < m$, то есть, что алгебраическая дробь $\frac{f(x)}{\varphi(x)}$ правильна (в противном случае мы всегда можем выделить из нее целую часть). Мы знаем, что если

$$\frac{f(x)}{\varphi(x)} = c_0 + c_1x + \dots + c_kx^k + \dots, \quad (10.10)$$

то

$$\begin{aligned} a_0 + a_1x + \dots + a_nx^n = \\ = (b_0 + b_1x + \dots + b_mx^m)(c_0 + c_1x + \dots + c_kx^k + \dots). \end{aligned}$$

Раскроем в правой части этого равенства скобки и сравним коэффициенты при одинаковых степенях x слева и справа. Сначала мы получим m соотношений такого вида:

$$\begin{aligned} b_0c_0 &= a_0, \\ b_0c_1 + b_1c_0 &= a_1, \\ b_0c_2 + b_1c_1 + b_2c_0 &= a_2, \end{aligned} \quad (10.11)$$

.....

$$b_0c_{m-1} + b_1c_{m-2} + \dots + b_{m-1}c_0 = a_{m-1}$$

(если $n < m - 1$, то мы считаем, что $a_{n+1} = \dots = a_{m-1} = 0$). А дальше все соотношения имеют один и тот же вид:

$$b_0c_{m+k} + b_1c_{m+k-1} + \dots + b_mc_k = 0, \quad k = 0, 1, \dots \quad (10.12)$$

(ведь в $f(x)$ нет членов, содержащих x^m, x^{m+1} и т.д.). Таким образом, коэффициенты $c_0, c_1, \dots, c_k, \dots$ ряда (10.10) удовлетворяют рекуррентному соотношению (10.12). Коэффициенты этого соотношения зависят лишь от знаменателя дроби. Числитель же дроби нужен для нахождения первых членов $c_0, c_1, \dots, c_{m-1}$ рекуррентной последовательности. Обратно, если дано рекуррентное соотношение (10.12) и заданы члены $c_0, c_1, \dots, c_{m-1}$, то мы сначала по формулам

(10.11) вычислим значения $a_0, \dots, a_{m-1}$. А тогда производящей функцией для последовательности чисел $c_0, c_1, \dots, c_k, \dots$ является алгебраическая дробь

$$\frac{f(x)}{\varphi(x)} = \frac{a_0 + ax + \dots + a_{m-1}x^{m-1}}{b_0 + b_1x + \dots + b_mx^m}. \quad (10.13)$$

На первый взгляд кажется, что мы мало выиграли при замене рекуррентного соотношения производящей функции. Ведь все равно придется делить числитель на знаменатель, а это приведет к тому же самому рекуррентному соотношению (10.12). Но дело в том, что над дробью (10.13) можно выполнять некоторые алгебраические преобразования, а это облегчит отыскание чисел c_k .

Об едином нелинейном рекуррентном соотношении

При решении задачи о разбиении последовательности мы пришли к рекуррентному соотношению

$$T_n = T_0T_{n-1} + T_1T_{n-2} + \dots + T_{n-1}T_0, \quad (10.14)$$

где $T_0 = 1$. Покажем, как решить соотношение (10.14). Для этого составим производящую функцию.

$$f(x) = T_0 + T_1x + T_2x^2 + \dots + T_nx^n + \dots \quad (10.15)$$

Положим

$$F(x) \equiv xf(x) = T_0x + T_1x^2 + \dots + T_nx^{n+1} + \dots \quad (10.16)$$

и возведем $F(x)$ в квадрат. Мы получим, что

$$\begin{aligned} F^2(x) &= T_0^2 + (T_0T_1 + T_1T_0)x^3 + \dots \\ &\dots + (T_0T_{n-1} + \dots + T_{n-1}T_0)x^{n+1} + \dots \end{aligned}$$

Но по рекуррентному соотношению (10.14),

$$T_0 T_{n-1} + \dots + T_{n-1} T_0 = T_n.$$

Значит,

$$F^2(x) = T_1 x^2 + T_2 x^3 + \dots + T_n x^{n+1}.$$

Полученный ряд есть не что иное, как $F(x) - T_0 x$; поскольку $T_0 = 1$, он равен

$$F^2(x) = F(x) - x.$$

Для функции $F(x)$ получилось квадратное уравнение (10.17). Решая его, находим, что

$$F(x) = \frac{1 - \sqrt{1 - 4x}}{2}.$$

Мы выбрали перед корнем знак минус, так как в противном случае при $x = 0$ мы имели бы $F(0) = 2$, а из разложения (10.16) видно, что $F(0) = 0$.

Алгоритмы на абстрактных структурах данных

Введение. Стеки. Очереди. Связанные списки. Двоичные деревья.

Введение

Н. Вирт определил программирование как алгоритм + структуры данных. При этом структура данных может не зависеть от конкретных языковых конструкций (абстрактная структура данных).

Рассмотрим некоторые основные структуры данных.

Стеки

Стеком называется одномерная структура данных, загрузка или увеличение элементов для которой осуществляется с помощью указателя стека в соответствии с правилом LIFO ("last-in, first-out" "последним введен, первым выведен").

Указатель стека `sp` (stack pointer) содержит в любой момент времени индекс (адрес) текущего элемента, который является единственным элементом стека, доступным в данный момент времени для обработки.

Существуют следующие основные базисные операции для работы со стеком (для случая, когда указатель стека всегда задает ячейку, находящуюся непосредственно над его верхним элементом).

1. Начальная установка:

```
Sp:=1;
```

1. Загрузка элемента x в стек:

```
Stack[sp]:=x;  
Sp:=sp+1;
```

1. Извлечение элемента из стека:

```
Sp:=sp-1;  
X:=stack[sp];
```

1. Проверка на переполнение и загрузка элемента в стек:

```
If sp<=sd then  
Begin stack[sp]:=x; sp:=sp+1 end  
Else  
\{ переполнение \};  
Здесь sd - размерность стека.
```

1. Проверка наличия элементов и извлечение элемента стека:

```
If sp>1 then  
Begin sp:=sp-1; x:=stack[sp] end  
Else  
\{ антипереполнение \}
```

1. Чтение данных из указателя стека без извлечения элемента:

```
x:=stack[sp-1].
```

Программа 1. Работа со стеком.

{Реализованы основные базисные операции для работы со стеком.
Программа написана на языке программирования Turbo-Pascal }

```
uses crt,graph;  
type PEI=^EI;  
EI=record  
  n:byte;  
  next:PEI;  
end;
```

```
var ster:array[1..3] of PEI;  
  number: byte;  
  p:PEI;  
  th,l: integer;  
  i:integer;
```

```

    nhod:word;
    s:string;

procedure hod(n,f,t:integer);
begin
if n>1 then begin
    hod(n-1,f,6-(f+t));
    hod(1,f,t);
    hod(n-1,6-(f+t),t);
end else begin
    p:=ster[f];
    ster[f]:=ster[f]^next;
    p^.next:=ster[t];
    ster[t]:=p;
    inc(nhod);
    str(nhod,s);
{ **** }
    setfillstyle(1,0);bar(0,0,50,10);
    setcolor(2);outtextxy(0,0,s);
    setfillstyle(1,0);setcolor(0);p:=ster[f];i:=1;
    while p<>nil do begin p:=p^.next;inc(i);end;
    fillellipse(160*f,460-(i-1)*th,(number-ster[t]^n+1)*l,10);
    setfillstyle(1,4);setcolor(4);p:=ster[t];i:=1;
    while p<>nil do begin fillellipse(160*t,460-(i-1)*th,(number-
ster[t]^n+1)*l,10);inc(i);p:=p^.next;end;
{ **** }
    { readkey; } { delay(50); }
end;
end;

procedure start;
var i:integer;grD,grM: Integer;
begin
clrscr;write('Enter the number of rings, please. ');readln(number);
for i:=1 to 3 do ster[i]:=nil;
for i:=1 to number do begin new(p);p^.n:=i;p^.next:=ster[1];ster[1]:=p;end;
nhod:=0;
grD:=Detect;{ InitGraph(grD,grM,"); } InitGraph(grD,grM,'c:\borland\tp\bgi');
th:=20;l:=round(50/number);

```

```
setfillstyle(1,4);setcolor(4);
for i:=1 to number do begin fillellipse(160,460-(i-1)*th,(number-
i+1)*l,10);end;
end;

begin
start;
{readkey;}
hod(number,1,3);
{closegraph;}
end.
```

Программа 2. Ханойская башня.

На стержне A в исходном порядке находится N дисков, уменьшающихся по размеру снизу вверх. Диски должны быть переставлены на стержень в исходном порядке при использовании в случае необходимости промежуточного стержня B для временного хранения дисков. В процессе перестановки дисков обязательно должны соблюдаться правила: одновременно может быть переставлен только один самый верхний диск (с одного из стержней на другой); ни в какой момент времени диск не может находиться на другом диске меньшего размера.

Программа реализована с помощью абстрактного типа данных – стек для произвольного числа дисков.

{Программа написана на языке программирования Turbo-Pascal}

```
uses crt,graph;
type PEI=^EI;
  EI=record
    n:byte;
    next:PEI;
  end;

var ster:array[1..3] of PEI;
    number: byte;
    p:PEI;
```

```

    th,l: integer;
    i:integer;
    nhod:word;
    s:string;

procedure hod(n,f,t:integer);
begin
if n>1 then begin
    hod(n-1,f,6-(f+t));
    hod(1,f,t);
    hod(n-1,6-(f+t),t);
end else begin
    p:=ster[f];
    ster[f]:=ster[f]^next;
    p^.next:=ster[t];
    ster[t]:=p;
    inc(nhod);
    str(nhod,s);
    {*****}
    setfillstyle(1,0);bar(0,0,50,10);
    setcolor(2);outtextxy(0,0,s);
    setfillstyle(1,0);setcolor(0);p:=ster[f];i:=1;
    while p<>nil do begin p:=p^.next;inc(i);end;
    fillellipse(160*f,460-(i-1)*th,(number-ster[t]^n+1)*l,10);
    setfillstyle(1,4);setcolor(4);p:=ster[t];i:=1;
    while p<>nil do begin fillellipse(160*t,460-(i-1)*th,(number-
ster[t]^n+1)*l,10);inc(i);p:=p^.next;end;
    {*****}
    { readkey;} {delay(50);}
end;
end;

procedure start;
var i:integer;grD,grM: Integer;
begin
clrscr;write('Enter the number of rings, please. ');readln(number);
for i:=1 to 3 do ster[i]:=nil;
for i:=1 to number do begin new(p);p^.n:=i;p^.next:=ster[1];ster[1]:=p;end;
nhod:=0;

```

```
grD:=Detect;{InitGraph(grD,grM,"");}InitGraph(grD,grM,'c:\borland\tp\bgi');
th:=20;l:=round(50/number);
setfillstyle(1,4);setcolor(4);
for i:=1 to number do begin fillellipse(160,460-(i-1)*th,(number-
i+1)*l,10);end;
end;

begin
start;
{readkey;}
hod(number,1,3);
{closegraph;}
end.
```

Очереди

Очередь - одномерная структура данных, для которой загрузка или извлечение элементов осуществляется с помощью указателей начала извлечения (head) и конца (tail) очереди в соответствии с правилом FIFO ("first-in, first-out" - "первым введен, первым выведен").

1. Начальная установка:

```
Head:=1; tail:=1;
```

1. Добавление элемента x:

```
Queue[tail]:=x; tail:=tail+1;
If tail>qd then tail:=1;
Здесь qd - размерность очереди.
```

1. Исключение элемента x:

```
x:=queue[head]; head:=head+1;
if head>qd then head:=1;
```

1. Проверка переполнения очереди и включение в нее элемента:

```
Temp:=tail+1;  
If temp>qd then temp:=1;  
If temp=head then \{переполнение\<}  
Else btgin queue[tail]:=x; tail:=temp end;
```

1. Проверка элементов и исключение элемента:

```
If head:=tail then  
\{очередь пуста\<}  
else begin  
x:=queue[head]; head:=head+1;  
if yead>qd then head:=1;  
end;
```

Отметим, что при извлечении элемента из очереди все элементы могут также перемещаться на один шаг к ее началу.

Связанные списки

Связанный список представляет собой структуру данных, которая состоит из узлов (как правило, записей), содержащих указатели на следующий узел. Указатель, который ни на что не указывает, снабжается значением *nil*. Таким образом, в каждый элемент связанного списка добавляется указатель (звено связи).

Приведем основные базисные операции для работы с однонаправленным связанным списком.

1. Включение элемента после элемента:

```
Link[q]:=link[p];  
Link[p]:=q;
```

Здесь *q* – индекс элемента, который должен быть вставлен в список после элемента с индексом *p*.

1. Исключение преемника элемента *x*:


```

If link[x]<>null then
Link[x]:=link[x]]
else
\{Элемент x не имеет преемника\};

```

Отметим, что элемент, следующий в списке за элементом x , называется преемником элемента x , а элемент, расположенный перед элементом x , называется предшественником элемента x . Если элемент x не имеет преемника, то содержащемуся в нем указателю присваивается значение `nil`.

1. Включение элемента y перед элементом x :

```

Prev:=0;
While(link[prev]<>nil)and(link[prev]<>x)do
Prev:=link[prev];
If link[prev]=x then
Btgin link[prev]:=y; link[y]:=x end
Else
\{Элемент x не найден\};
Здесь link[0]является началом списка.

```

Отметим, что исключение последнего элемента из однонаправленного списка связано с просмотром всего списка.

В двунаправленном связанном списке каждый элемент имеет два указателя (`succlink` - описывает связь элемента с преемником, `predlink` - с предшественником).

Приведем основные базисные операции для работы с двунаправленным связанным списком.

Ответ 1 Включение y перед элементом x :

```

Succlink[y]:=x;
Predlink[y]:=predlink[x];
Succlink[predlink[x]]:=y;
Predlink[x]:=y;

```

Ответ 2 Включение элемента y после элемента x :

```

Succlink[y]:=succlink[x];
Predlink[y]:=x;
Predlink[succlink[x]]:=y;
Succlink[x]:=y;

```

Ответ 3 Исключение элемента x.

```

Predlink[succlink[x]]:=predlink[x];
Succlink[predlink[x]]:=succlink[x];

```

Программа 3.Список целых чисел.

{Создается список целых чисел. Числа выбираются случайным образом из интервала 0..9999, затем он упорядочивается, сначала - по возрастанию, затем - по убыванию.

Программа написана на языке программирования Turbo-Pascal}

```

uses crt;
type TLink=^Link;
  Link=record
    v : integer;
    p, n : TLink
  end;

var i : integer;
    p, q, w : TLink;
    s1,s2,rs : TLink;

procedure Sort( sp : TLink; t : integer );
  var temp : integer;
begin
  q:=sp;
  while q^.n<>nil do begin
    q:=q^.n;
    p:=sp;
    while p^.n<>nil do begin
      if (p^.v-p^.n^.v)*t>0 then begin
        temp:=p^.v;
        p^.v:=p^.n^.v;

```

```
    p^.n^.v:=temp;
end;
p:=p^.n;
end;
end;
end;
```

```
function CreatRndSpis(deep : integer):TLink;
begin
new(q);
for i:=1 to deep do begin
if i=1 then begin
p:=q;q^.p:=nil;
end;
q^.v:=random(9999);
new(q^.n);
q^.n^.p:=q;
q:=q^.n;
end;
q^.p^.n:=nil;
dispose(q);
CreatRndSpis:=p;
end;
```

```
function CreatSortDawnSpis(deep : integer):TLink;
begin
if deep<9999 then begin
new(q);
for i:=1 to deep do begin
if i=1 then begin
q^.p:=nil;p:=q;
end;
q^.v:=random(round(9999/deep))+round(9999*(1-i/deep));
new(q^.n);
q^.n^.p:=q;
q:=q^.n;
end;
q^.p^.n:=nil;
dispose(q);
```

```

end else p:=nil;
CreatSortDawnSpis:=p;
end;

```

```

procedure Show( s : TLink; sp: integer );
var i : integer;
begin
p:=s;
i:=1;
while p<>nil do begin
gotoxy(sp,i);write(' ' : 5); gotoxy(sp,i);writeln(p^.v);
p:=p^.n;
inc(i);
end;
end;

```

```

function min( c1, c2 : integer ) : integer;
begin
case c1<c2 of
true : min:=c1;
false: min:=c2;
end;
end;

```

```

function CreatConcSortUpSpis( sp1, sp2 : TLink ) : TLink;
begin
q:=sp1;while q^.n<>nil do q:=q^.n;
w:=sp2;while w^.n<>nil do w:=w^.n;
new(p);

```

```

CreatConcSortUpSpis:=p;
p^.p:=nil;
while(w<>nil)and(q<>nil)do begin
if(w<>nil)and(q<>nil)then begin
p^.v:=min(q^.v,w^.v);
case p^.v=q^.v of
true : q:=q^.p;
false: w:=w^.p;
end;

```

```

    new(p^.n);
    p^.n^.p:=p;
    p^.n^.n:=nil;
    p:=p^.n;
end;
if(w=nil)and(q<>nil)then begin
    while q<>nil do begin
        p^.v:=q^.v;q:=q^.p;
        new(p^.n);
        p^.n^.p:=p;
        p^.n^.n:=nil;
        p:=p^.n;
    end;
end;
if(w<>nil)and(q=nil)then begin
    while w<>nil do begin
        p^.v:=w^.v;w:=w^.p;
        new(p^.n);
        p^.n^.p:=p;
        p^.n^.n:=nil;
        p:=p^.n;
    end;
end;
p^.p^.n:=nil;
dispose(p);
end;

begin
clrscr;
randomize;
s1:=CreatRndSpis(15);Sort(s1,-1);
s2:=CreatRndSpis( 5);Sort(s2,-1);
rs:=CreatConcSortUpSpis(s1,s2);
Show(s1,10);
Show(s2,20);
Show(rs,30);
Sort(rs,-1);
Show(rs,40);

```

```
readln;  
end.
```

Двоичные деревья

Основные определения и понятия о графах даются в лекции 12. В лекции 16 и 17 рассматриваются комбинаторные алгоритмы на графах. В данной лекции приведены несколько понятий, необходимых для описания абстрактной структуры данных, - двоичное дерево.

При решении многих задач математики используется понятие графа. Граф - набор точек на плоскости (эти точки называются вершинами графа), некоторые из которых соединены отрезками (эти отрезки называются ребрами графа). Вершины могут располагаться на плоскости произвольным образом. Причем неважно, является ли ребро, соединяющее две вершины, отрезком прямой или кривой линией. Важен лишь факт соединения двух данных вершин графа ребром. Примером графа может служить схема линий железных дорог. Если требуется различать вершины графа, их нумеруют или обозначают разными буквами. В изображении графа на плоскости могут появляться точки пересечения ребер, не являющиеся вершинами.

Говорят, что две вершины графа соединены путем, если из одной вершины можно пройти по ребрам в другую вершину. Путей между двумя данными вершинами может быть несколько. Поэтому обычно путь обозначают перечислением вершин, которые посещают при движении по ребрам. Граф называется связным, если любые две его вершины соединены некоторым путем.

Состоящий из различных ребер замкнутый путь называется циклом. Связный граф, в котором нет циклов, называется деревом. Одним из основных отличительных свойств дерева является то, что в нем любые две вершины соединены единственным путем. Дерево называется ориентированным, если на каждом его ребре указано направление. Следовательно, о каждой вершине можно сказать, какие ребра в нее входят, а какие - выходят. Точно так же о каждом ребре можно сказать, из какой вершины оно выходит, и в какую - входит. Двоичное дерево - это такое ориентированное дерево, в котором:

1. Имеется ровно одна вершина, в которую не входит ни одного ребра. Эта вершина называется корнем двоичного дерева.
2. В каждую вершину, кроме корня, входит одно ребро.
3. Из каждой вершины (включая корень) исходит не более двух ребер.

Граф задается аналогично спискам через записи и указатели.

Программа 4. Создание и работа с деревом.

```
//Алгоритм реализован на языке Turbo-C++.
//Вершины дерева задаются структурой: поле целых,
//поле для размещения адреса левого "сына" и поле для размещения
//адреса правого "сына"
//Значение целого выбирается случайным образом из интервала 0..99.
//Число уровней дерева равно N. В примере N = 5.
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <time.h>
#define N 5
struct tree{int a;
            tree* left;
            tree* right;};

void postr(tree* root,int h)
{
    root->a=random(100);
    if (h!=0){
        if (random(3)){
            root->left=new(tree);
            postr(root->left,h-1);}
        else root->left=NULL;
        if (random(3))
            {root->right=new(tree);postr(root->right,h-1);}
        else root->right=NULL;}
    else {root->right=NULL;root->left=NULL;}
}

void DFS(tree* root)
```

```
{printf("%d ",root->a);  
  if (root->left!=NULL) DFS(root->left);  
  if (root->right!=NULL) DFS(root->right);}
```

```
void main()  
{clrscr();  
  randomize();  
  tree* root1;  
  root1=new(tree);  
  postr(root1,N);  
  DFS(root1);  
  getch();  
}
```


Что такое граф? Определения и примеры

Введение. Представления. Связность и расстояние. Остовные деревья. Клики. Изоморфизм. Планарность.

Введение

Множество самых разнообразных задач естественно формулируется в терминах графов. Так, например, могут быть сформулированы задачи составления расписаний в исследовании операций, анализа сетей в электротехнике, установления структуры молекул в органической химии, сегментации программ в программировании, анализа цепей Маркова в теории вероятностей. В задачах, возникающих в реальной жизни, соответствующие графы часто оказываются так велики, что их анализ неосуществим без ЭВМ. Таким образом, решение прикладных задач с использованием теории графов возможно в той мере, в какой возможна обработка больших графов на ЭВМ, и поэтому эффективные алгоритмы решения задач теории графов имеют большое практическое значение. В 16 и 17 лекциях мы излагаем несколько эффективных алгоритмов на графах и используем их для демонстрации некоторой общей техники решения задач на графах с помощью ЭВМ.

Конечный граф $G = (V, E)$ состоит из конечного множества вершин $V = \{v_1, v_2, \dots\}$ и конечного множества ребер $E = \{e_1, e_2, \dots\}$. Каждому ребру соответствует пара вершин: если ребро (v, w) соответствует ребру e , то говорят, что e инцидентно вершинам v и w . Граф $G = (V, E)$ изображается следующим образом: каждая вершина представляется точкой и каждое ребро представляется отрезком линии, соединяющим его концевые вершины. Граф называется ориентированным, если пара вершин (v, w) , соответствующая каждому ребру, упорядочена. В таком случае говорят, что ребро e ориентировано из вершины v в вершину w , а направление обозначается стрелкой на ребре. Мы будем называть ориентированные графы орграфами. В неориентированном графе концевые вершины каждого ребра не упорядочены, и ребра не имеют направления. Ребро называется петлей, если оно начинается и кончается в одной и той же вершине. Говорят,

что два ребра параллельны, если они имеют одну и ту же пару концевых вершин (и если они имеют одинаковую ориентацию в случая ориентированного графа). Граф называется простым, если он не имеет ни петель, ни параллельных ребер. Если не указывается противное, будем считать, что рассматриваемые графы являются простыми. Всюду в 16 и 17 лекции будем использовать символы $|V|$ и $|E|$ для обозначения соответственно числа вершин и числа ребер в графе $G = (V, E)$.

Представления

Наиболее известный способ представления графа на бумаге состоит в геометрическом изображении точек и линий. В ЭВМ граф должен быть представлен дискретным способом, причем возможно много различных представлений. Простота использования, так же как и эффективность алгоритмов на графе, зависит от подходящего выбора представления графа. Рассмотрим различные структуры данных для представления графов.

Матрица смежностей. Одним из наиболее распространенных машинных представлений простого графа является матрица смежностей или соединений. Матрица смежностей графа $G = (V, E)$ есть $|V| \times |V|$ -матрица $A = [a_{ij}]$, в которой $a_{ij} = 1$, если в G существует ребро, идущее из i -й вершины в j -ю, и $a_{ij} = 0$ в противном случае. Орграф и его матрица смежностей представлены на рис. 12.1.

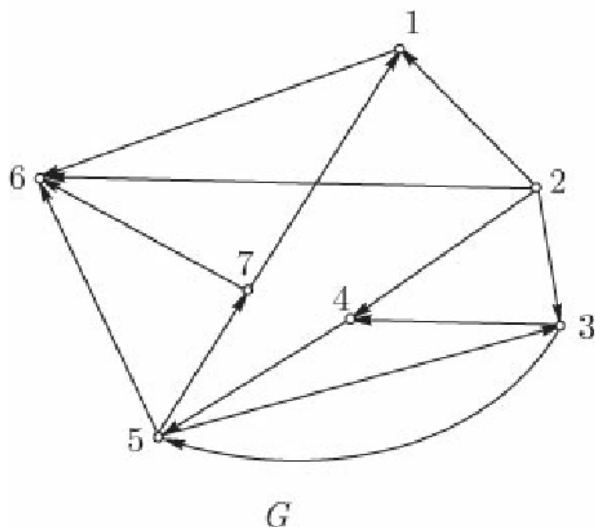


Рис. 12.1. Ориентированный граф и его матрица смежностей

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Заметим, что в матрице смежностей петля может быть представлена соответствующим единичным диагональным элементом. Кратные ребра можно представить, позволив элементу матрицы быть больше 1, но это не принято, так как обычно удобно представлять каждый элемент матрицы одним двоичным разрядом.

Для задания матрицы смежностей требуется $|V|^2$ двоичных разрядов. У неориентированного графа матрица смежностей симметрична, и для ее представления достаточно хранить только верхний треугольник. В результате экономится почти 50% памяти, но время вычислений может при этом немного увеличиться, потому что каждое обращение к a_{ij} должно быть заменено следующим: if $i > j$ then a_{ji} else a_{ij} . В

случае представления графа его матрицей смежностей для большинства алгоритмов требуется время вычисления, по крайней мере пропорциональное $|V|^2$.

Матрица весов. Граф, в котором ребру (i, j) сопоставлено число w_{ij} , называется взвешенным графом, а число w_{ij} называется весом ребра (i, j) . В сетях связи или транспортных сетях эти веса представляют некоторые физические величины, такие как стоимость, расстояние, эффективность, емкость или надежность соответствующего ребра. Простой взвешенный граф может быть представлен своей матрицей весов $W = [w_{ij}]$, где w_{ij} есть вес ребра, соединяющего вершины i и j . Веса несуществующих ребер обычно полагают равными ∞ или 0 в зависимости от приложений. Когда вес несуществующего ребра равен 0, матрица весов является простым обобщением матрицы смежностей.

Список ребер. Если граф является разреженным, то возможно, что более эффективно представлять ребра графа парами вершин. Это представление можно реализовать двумя массивами $g = (g_1, g_2, \dots, g_{|E|})$ и $h = (h_1, h_2, \dots, h_{|E|})$. Каждый элемент в массиве есть метка вершины, а i -е ребро графа выходит из вершины g_i и входит в вершину h_i . Например, орграф, изображенный на рис. 12.1, будет представляться следующим образом:

$$g = (1, 2, 2, 2, 2, 3, 3, 4, 5, 5, 5, 7, 7),$$

$$h = (6, 1, 3, 4, 6, 4, 5, 5, 3, 6, 7, 1, 6).$$

Ясно, что при этом легко представимы петли и кратные ребра.

Структура смежности. В ориентированном графе вершина y называется последователем другой вершины x , если существует ребро, направленное из x в y . Вершина x называется тогда предшественником y . В случае неориентированного графа две вершины называются соседями, если между ними есть ребро. Граф может быть описан его структурой смежности, то есть списком всех последователей (соседей) каждой вершины; для каждой вершины v задается $Adj(v)$ - список всех последователей (соседей) вершины v . В большинстве

алгоритмов на графах относительный порядок вершин, смежных с вершиной v в $Adj(v)$, не важен, и в таком случае удобно считать $Adj(v)$ мультимножеством (или множеством, если граф является простым) вершин, смежных с v . Структура смежности орграфа, представленного на рис. 12.1, такова:

```
Adj(v)
1: 6
2: 1, 3, 4, 6
3: 4, 5
4: 5
5: 3, 6, 7
6:
7: 1, 6
```

Если для хранения метки вершины используется одно машинное слово, то структура смежности ориентированного графа требует $|V| + |E|$ слов. Если граф неориентированный, нужно $|V| + 2|E|$ слов, так как каждое ребро встречается дважды.

Структуры смежности могут быть удобно реализованы массивом из $|V|$ линейно связанных списков, где каждый список содержит последователей некоторой вершины. Поле данных содержит метку одного из последователей, и поле указателей указывает следующего последователя. Хранение списков смежности в виде связанного списка желательно для алгоритмов, в которых в графе добавляются или удаляются вершины.

Матрица инцидентности - M задает граф : $m_{ij} = 1$, если ребро j выходит из вершины i , $m_{ij} = -1$, если ребро j входит в вершину i , и $m_{ij} = 0$ в остальных случаях.

Связность и расстояние

Говорят, что вершины x и y в графе смежны, если существует ребро, соединяющее их. Говорят, что два ребра смежны, если они имеют

общую вершину. Простой путь, или для краткости, просто путь, записываемый иногда как $(v_1, v_2, \dots, v_k)$, - это последовательность смежных ребер $(v_1, v_2), (v_2, v_3), \dots, (v_{k-2}, v_{k-1}), (v_{k-1}, v_k)$, в которой все вершины $v_1, v_2, \dots, v_k$ различны, исключая, возможно, случай $v_1 = v_k$. В орграфе этот путь называется ориентированным из v_1 в v_k , в неориентированном графе он называется путем между v_1 и v_k . Число ребер в пути называется длиной пути. Путь наименьшей длины называется кратчайшим путем. Замкнутый путь называется циклом. Граф, который не содержит циклов, называется ациклическим.

Подграф графа $G = (V, E)$ есть граф, вершины и ребра которого лежат в G . Подграф G , индуцированный подмножеством S множества V вершин графа G , - это подграф, который получается в результате удаления всех вершин из $V - S$ и всех ребер, инцидентных им.

Неориентированный граф G связан, если существует хотя бы один путь в G между каждой парой вершин v_i и v_j . Ориентированный граф G связан, если неориентированный граф, получающийся из G путем удаления ориентации ребер, является связным. Граф, состоящий из единственной изолированной вершины, является (тривиально) связным. Максимальный связный подграф графа G называется связной компонентой или просто компонентой G . Несвязный граф состоит из двух или более компонент. Максимальный сильно связный подграф называется сильно связной компонентой.

Иногда недостаточно знать, что граф связан; нас может интересовать, насколько "сильно связан" связный граф. Например, связный граф может содержать вершину, удаление которой вместе с инцидентными ей ребрами разъединяет оставшиеся вершины. Такая вершина называется точкой сочленения или разделяющей вершиной. Граф, содержащий точку сочленения, называется разделимым. Граф без точек сочленения называется двусвязным или неразделимым. Максимальный двусвязный подграф графа называется двусвязной компонентой или блоком.

Большинство основных вопросов о графах касается связности, путей и расстояний. Нас может интересовать вопрос, является ли граф связным; если он связан, то может оказаться нужным найти кратчайшее расстояние между выделенной парой вершин или определить

кратчайший путь между ними. Если граф несвязен, то может потребоваться найти все его компоненты. В нашем курсе строятся алгоритмы для решения этих и других подобных вопросов.

Остовные деревья

Связный неориентированный ациклический граф называется деревом, множество деревьев называется лесом. В связном неориентированном графе G существует по крайней мере один путь между каждой парой вершин; отсутствие циклов в G означает, что существует самое большее один такой путь между любой парой вершин в G . Поэтому, если G - дерево, то между каждой парой вершин в G существует в точности один путь. Рассуждение легко обратимо, и поэтому неориентированный граф G будет деревом тогда и только тогда, если между каждой парой вершин в G существует в точности один путь. Так как наименьшее число ребер, которыми можно соединить n вершин, равно $n - 1$ и дерево с n вершинами содержит в точности $n - 1$ ребер, то деревья можно считать минимально связными графами. Удаление из дерева любого ребра превращает его в несвязный граф, разрушая единственный путь между по крайней мере одной парой вершин.

Особый интерес представляют остовные деревья графа G , то есть деревья, являющиеся подграфами графа G и содержащие все его вершины. Если граф G несвязен, то множество, состоящее из остовных деревьев каждой компоненты называется остовным лесом графа. Для построения остовного дерева (леса) данного неориентированного графа G , мы последовательно просматриваем ребра G , оставляя те, которые не образуют циклов с уже выбранными.

Во взвешенном графе $G = (V, E)$ часто интересно определить остовное дерево (лес) с минимальным общим весом ребер, то есть дерево (лес), у которого сумма весов всех его ребер минимальна. Такое дерево называется минимумом оставных деревьев или минимальное остовное дерево. Другими словами, на каждом шаге мы выбираем новое ребро с наименьшим весом (наименьшее ребро), не образующее циклов с уже выбранными ребрами; этот процесс продолжаем до тех

пор, пока не будет выбрано $|V| - 1$ ребер, образующих остовное дерево T . Этот процесс известен как жадный алгоритм.

Жадный алгоритм может быть выполнен в два этапа. Сначала ребра сортируются по весу и затем строится остовное дерево путем выбора наименьших из имеющихся в распоряжении ребер.

Существует другой метод получения минимума остовных деревьев, который не требует ни сортировки ребер, ни проверки на цикличность на каждом шаге, - так называемый алгоритм ближайшего соседа. Мы начинаем с некоторой произвольной вершины a в заданном графе.

Пусть (a, b) - ребро с наименьшим весом, инцидентное a ; ребро (a, b) включается в дерево. Затем среди всех ребер, инцидентных либо a , либо b , выбираем ребро с наименьшим весом и включаем его в частично построенное дерево. В результате этого в дерево добавляется новая вершина, например, c . Повторяя процесс, ищем наименьшее ребро, соединяющее a , b или c с некоторой другой вершиной графа. Процесс продолжается до тех пор, пока все вершины из G не будут включены в дерево, то есть пока дерево не станет остовным.

Наихудшим для этого алгоритма будет случай, когда G - полный граф (то есть когда каждая пара вершин в графе соединена ребром); в этом случае для того, чтобы найти ближайшего соседа, на каждом шаге нужно сделать максимальное число сравнений. Чтобы выбрать первое ребро, мы сравниваем веса всех $|V| - 1$ ребер, инцидентных вершине a , и выбираем наименьшее; этот шаг требует $|V| - 2$ сравнений. Для выбора второго ребра мы ищем наименьшее среди возможных $2(|V| - 2)$ ребер (инцидентных a или b) и делаем для этого $2(|V| - 2) - 1$ сравнений. Таким образом, ясно, что для выбора i -го ребра требуется $i(|V| - i) - 1$ сравнений, и поэтому в сумме потребуется

$$\sum_{i=1}^{|V|-1} [i(|V| - i) - 1] = \frac{1}{6} |V|^3 + O(|V|^2)$$

сравнений для построения минимума остовных деревьев.

Клики

Максимальный полный подграф графа G называется кликой графа G ; другими словами, клика графа G есть подмножество его вершин, такое, что между каждой парой вершин этого подмножества существует ребро и, кроме того, это подмножество не принадлежит никакому большему подмножеству с тем же свойством. Например, на [рис. 12.3](#) показан граф и его клики.

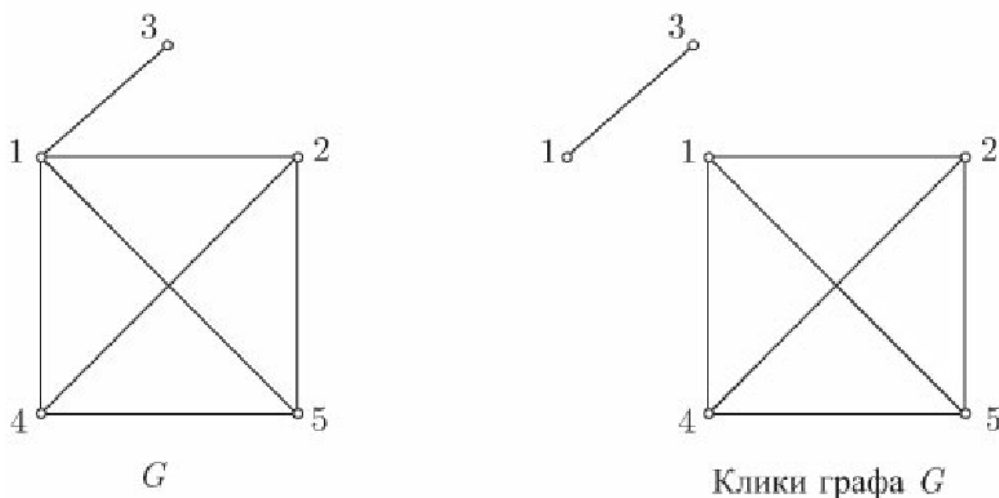


Рис. 12.2. Граф G и все его клики

Клики графа представляют "естественные" группировки вершин, и определение клик графа полезно в кластерном анализе в таких областях, как информационный поиск и социология.

Изоморфизм

Два графа $G_A = (V_A, E_A), G_B = (V_B, E_B)$ называются изоморфными, если существует взаимно однозначное соответствие $f: V_A \rightarrow V_B$, такое, что $(v, w) \in E_A$ тогда и только тогда, если $(f(v), f(w)) \in E_B$, то есть существует соответствие между

вершинами графа G_A и вершинами графа G_B , сохраняющее отношение смежности. Например, на рис. 12.3 показаны два изоморфных орграфа: вершины a, b, c, d, e, f в орграфе G_2 соответствуют вершинам 2, 3, 6, 1, 4, 5 в указанном порядке в орграфе G_1 . Вообще говоря, между V_A и V_B может быть более чем одно соответствие, и на рис. 12.3 графы имеют на самом деле второй изоморфизм: a, b, c, d, e, f соответствуют в указанном порядке вершинам 2, 3, 6, 1, 5, 4. Изоморфные графы отличаются только метками вершин, в связи с чем задача определения изоморфизма возникает в ряде практических ситуаций, таких, как информационный поиск и определение химических соединений.

Заметим, что можно ограничиться орграфами. Любой неориентированный граф превращается в орграф заменой каждого ребра двумя противоположно направленными ребрами. Два полученные таким образом орграфа, очевидно, изоморфны тогда и только тогда, если изоморфны исходные графы.

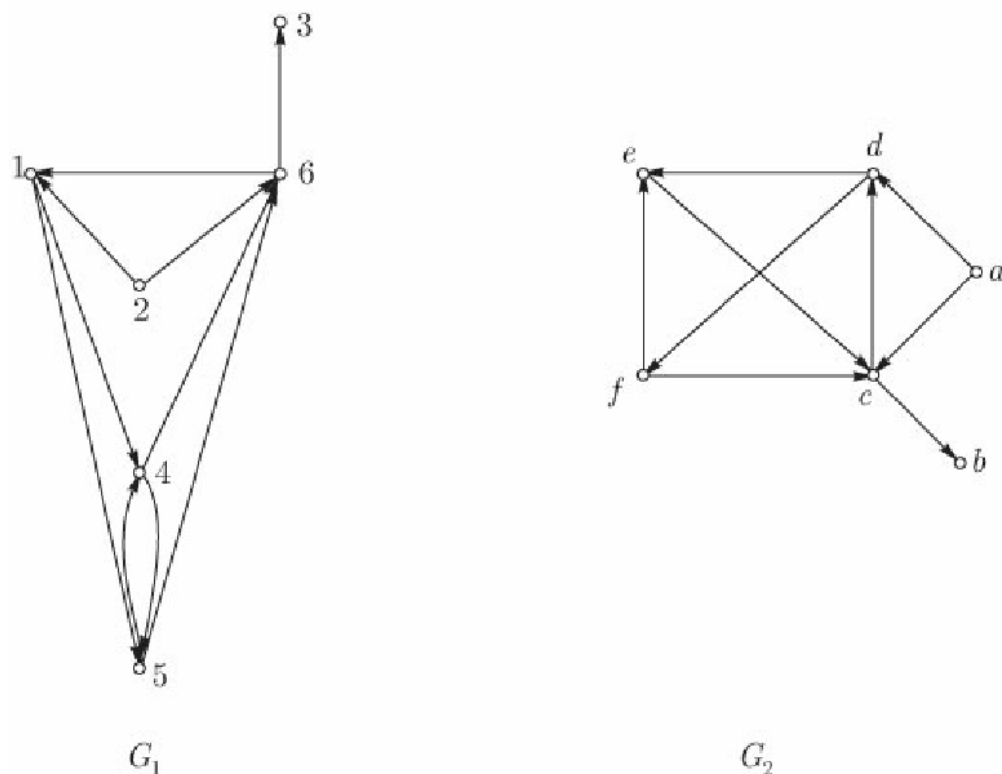


Рис. 12.3. Изоморфные орграфы

Планарность

Граф называют планарным, если существует такое изображение на плоскости его вершин и ребер, что:

- каждая вершина v изображается отдельной точкой v' на плоскости;
- каждое ребро (v, w) изображается простой кривой, имеющей концевые точки (v', w') ;
- эти кривые пересекаются только в общих концевых точках. Задача определения того, можно ли изобразить граф на плоскости без пересечения ребер, имеет большой практический интерес (например, при конструировании интегральных схем или печатных плат необходимо выяснить, можно ли окончательную

схему вложить в плоскость).

Определение планарности графа отличается от других рассмотренных нами задач, поскольку при изображении точек и линий на плоскости приходится больше иметь дело с непрерывными, а не с дискретными величинами. Взаимосвязь между дискретными и непрерывными аспектами планарности заинтересовала математиков и привела к различным характеристикам планарных графов. С точки зрения математики эти характеристики изящны, но эффективных алгоритмов определения планарности они не дают. Наиболее успешный подход к определению планарности состоит просто в том, что граф разбивается на подграфы и затем делается попытка разместить его на плоскости, добавляя подграфы один за другим и сохраняя при размещении планарность.

Вначале сделаем несколько простых, но полезных наблюдений. Поскольку оргграф планарен тогда и только тогда, если планарен соответствующий неориентированный граф, полученный игнорированием направления ребер, то достаточно рассматривать только неориентированные графы, поскольку неориентированный граф планарен тогда и только тогда, если все его двусвязные компоненты планарны. Поэтому если неориентированный граф является разделимым, мы можем разложить его на двусвязные компоненты и рассматривать их отдельно. Наконец, поскольку параллельные ребра и петли всегда можно добавить к графу или удалить из него без нарушения свойства планарности, нам достаточно рассматривать только простые графы. Поэтому при определении планарности будем предполагать, что граф неориентированный, простой и двусвязный.

Наша основная стратегия состоит прежде всего в том, чтобы в графе G найти цикл C , разместить C на плоскости в виде простой замкнутой кривой, разложить оставшуюся часть $G - C$ на непересекающиеся по ребрам пути и затем попытаться разместить каждый из этих путей либо целиком внутри C , либо целиком вне C . Если нам удалось разместить так весь граф G , то он планарен, в противном случае он непланарен. Трудность этого способа заключается в том, что при размещении путей можно выбирать либо внутренность, либо внешность C , и мы должны проконтролировать, чтобы неправильный выбор области размещения на ранней стадии не устранял возможности размещения последующих

путей, - это могло бы привести нас к неверному заключению, что планарный граф непланарен.

Поиск

Введение. Поиск и другие операции над таблицами. Последовательный поиск. Логарифмический поиск в статических таблицах. Бинарный поиск. Оптимальные деревья бинарного поиска. Логарифмический поиск в динамических таблицах. Сбалансированные сильно ветвящиеся деревья.

Введение

Задача поиска является фундаментальной в комбинаторных алгоритмах, так как она формулируется в такой общности, что включает в себя множество задач, представляющих практический интерес. При самой общей постановке "Исследовать множество T с тем чтобы найти элемент, удовлетворяющий некоторому условию C ", о задаче поиска едва ли можно сказать что-либо стоящее. Удивительно, однако, что достаточно незначительных ограничений на структуру множества T , чтобы задача стала интересной: возникает множество разнообразных стратегий поиска различной степени эффективности. Мы сделаем некоторые предположения о структуре множества T , позволяющие исследовать T $O(\log n)$ или $O(1)$. Большинство алгоритмов поиска попадает в одну из трех категорий, характеризуемых временем поиска $O(n)$.

Поиск и другие операции над таблицами

Любой способ поиска оперирует с элементами, которые будем называть именами, взятыми из множества имен S - оно называется пространством имен. Это пространство имен может быть конечным или бесконечным. Самыми распространенными пространствами имен являются множества целых чисел с их числовым порядком (нумерацией), и множества последовательностей символов над некоторым конечным алфавитом с их лексикографическим (то есть словарным) порядком. Каждый из алгоритмов поиска, обсуждаемых в этой лекции, основан на одном из трех следующих предположений о пространстве S .

Предположение 1. На S определен линейный порядок, называемый естественным порядком и обозначаемый знаком $<$. Такой порядок имеет следующие свойства.

1. Любые два элемента $x, y \in S$ сравнимы, то есть должно выполняться в точности одно из трех условий: $x < y$, $x = y$, $y < x$.
2. Порядок обладает транзитивностью, то есть если $x < y$ и $y < z$, то $x < z$ для любых элементов $x, y, z \in S$. Мы используем обозначения $>$, $\leq$, $\geq$ в очевидном смысле. При анализе эффективности алгоритма поиска полагаем, что исход ($<$, $=$ или $>$) зависит от сравнения.

Предположение 2. Каждое имя в S есть последовательность символов или цифр над конечным линейно упорядоченным алфавитом $A = \{a_1, a_2, \dots, a_c\}$. Естественным порядком на S является лексикографический порядок, индуцированный линейным порядком на A . Мы полагаем, что исход ($<$, $=$ или $>$) сравнения двух символов (не имен) получается за время, не зависящее от $|S|$ или n .

Предположение 3. Имеется функция $h: S \rightarrow \{0, 1, \dots, m-1\}$, которая равномерно отображает пространство имен S в множество $\{0, 1, \dots, m-1\}$, то есть все целые $i, 0 \leq i \leq m-1$, приблизительно с одинаковой частотой являются образами имен из S при отображении h . Мы полагаем, что функция h не зависела от $|S|$, это с теоретической точки зрения выглядит шатко, но с практической - довольно реально.

Как уже было отмечено, поиск производится не в самом пространстве S имен, а в конечном подмножестве $T = \{x_1, x_2, \dots, x_n\}$ множества S , называемом таблицей. На большинстве таблиц, которые мы рассматриваем, определен линейный порядок, называемый табличным порядком: ему соответствует нижний индекс имени (то есть x_1 есть первое имя в таблице, x_2 - второе и тому подобное). Табличный порядок часто совпадает с естественным порядком,

определенным на пространстве имен, однако такое совпадение не обязательно.

Мощность таблицы T обычно намного меньше, чем мощность пространства имен S , даже если S конечно.

Представление о таблице как об упорядоченном множестве имен существенно для многих целей. Но иногда бывает необходимо рассматривать таблицу как множество ячеек, каждая из которых может содержать одно имя. Например, если процесс включения нового имени рассматривать более подробно, то обнаружится, что сначала нужно расширить таблицу добавлением ячейки, для того чтобы заготовить место для новой записи: только потом туда заносится новое имя.

Мы будем предполагать, что имена появляются в таблице не больше одного раза (исключение составляет переходный период, в течение которого заносится новое имя; в таблице допускается два вхождения одного имени). В большинстве случаев вследствие такого предположения таблица с n именами имеет ровно n ячеек. Однако важный класс алгоритмов, основанных на вычислении адреса, опирается на предположение о том, что таблица содержит больше ячеек, чем имен. Эти алгоритмы должны четко принимать во внимание наличие пустых ячеек.

Существует ряд синонимов для объектов, именуемых здесь таблицей и именем. В обработке данных существуют файлы, элементами которых являются записи; каждая запись есть последовательность полей, одно из которых (участвующее в поиске) называется ключом. Если сам файл проходится во время поиска, "файл" и "ключ" представляют собой то, что мы называем "таблицей" и "именем" соответственно. (Эта терминология несколько двусмысленна, поскольку понятие "ключ" можно отнести либо к самой ячейке, либо к содержимому ячейки.) Однако при поиске в большом файле часто не подразумевается просмотр самого файла; вместо этого поиск осуществляется на справочнике или указателе файла. При успешном поиске найденная в указателе отдельная запись указывает на соответствующую запись в файле. В таком типе организации файлов нашему понятию таблицы соответствует указатель или справочник.

Мы рассматриваем только четыре табличные операции: поиск, включение, исключение и распечатка. Подробное определение того, что должны делать эти операции над таблицей $T = \{x_1, x_2, \dots, x_n\}$, зависит от структуры данных, использованной для реализации таблицы.

поиск z : если $z \in T$, то отметить его указателем, то есть переменной i присвоить такое значение, что $z = x_i$; в противном случае указать, что $z \notin T$;

включение z : если $z \notin T$, то поместить его на соответствующее место.

Включение в общем случае предполагает прежде всего поиск соответствующего места, поэтому иногда удобно разделить операцию на две фазы. Сначала используем процедуру поиска для отыскания места, куда должно быть помещено z , и затем помещаем z на это место.

включить z на i -е место: включить z сразу после имени x_{i-1} . До включения $T = \{x_1, \dots, x_{i-1}, \dots, x_{i-1}z, x_i, \dots, x_n\}$.

исключить z : если $z \in T$, то исключить его.

Как и включение, исключение иногда реализуется процедурой поиска для получения места z и последующей процедурой:

исключить с i -го места: исключить x_i из T . До исключения $T = \{x_1, \dots, x_{i-1}, x_i, x_{i+1}, \dots, x_n\}$ \ \& После исключения $T = \{x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n\}$

Таблица, в которой осуществляются включения или исключения, называется динамической; в противном случае она носит название статической.

Последней операцией, которую мы будем рассматривать для каждой табличной структуры, является

распечатка: & напечатать все имена из T в их естественном порядке

Среди всех операций, которые можно производить над таблицами, четыре, рассматриваемые в этой лекции (поиск, включение, исключение и распечатка), и сортировка ([лекции 14, 15](#)) - наиболее важные.

Последовательный поиск

Под последовательным поиском мы подразумеваем исследование имен в том порядке, в котором они встречаются в таблице. При таком поиске в таблице в худшем случае получается просмотр всей таблицы; даже в среднем последовательный поиск имеет тенденцию к использованию числа операций, пропорционального n . Для больших таблиц его не следует относить к методам быстрого поиска, поскольку последовательный поиск асимптотически гораздо медленнее других алгоритмов, описанных в этой лекции. Несмотря на его низкие асимптотические возможности, имеется ряд причин, по которым этот метод следует обсудить вначале. Во-первых, хотя идея его проста, он позволяет нам ввести важные понятия и методы, применимые к поиску вообще. Во-вторых, последовательный поиск является единственным методом поиска, применимым к отдельным устройствам памяти и к тем таблицам, которые строятся на пространстве имен без линейного порядка. Наконец, последовательный поиск является быстрым для достаточно малых таблиц и для больших таблиц, организованных иерархическим способом: более быстрый метод используется для исследования окрестности верхушки иерархии, а последовательный поиск – для подтаблицы на нижнем уровне иерархии.

Для последовательного поиска по таблице $T = \{x_1, x_2, \dots, x_n\}$ мы предполагаем, что имеется указатель i , значение которого принадлежит отрезку $1 \leq i \leq n$ или, возможно, $0 \leq i \leq n + 1$. Над этим указателем разрешается производить только следующие операции; первоначальное присваивание ему значения 1 или n (или, если удобнее, 0 или $n + 1$), увеличение и /или уменьшение его на единицу и сравнение его с 0, 1, n или $n + 1$. При таких соглашениях наиболее очевидный алгоритм поиска в таблице T первого вхождения данного имени z имеет вид алгоритма 13.1. Здесь, как и во всех других алгоритмах поиска, изложенных в настоящей лекции, мы полагаем, что

алгоритм останавливается немедленно по отыскании z или установлении, что z в таблице нет.

Алгоритм 13.1. Последовательный поиск

for $i = 1$ *to* n *do* *if* $z = x_i$ *then* { найдено: i указывает на z }
не найдено: z не входит в T }

Рассмотрим некоторые аспекты эффективности последовательного поиска, начиная со стандартных методов программирования. В программе, построенной в виде одного цикла, как алгоритм 13.1, любое значительное ускорение должно быть следствием улучшения кода в цикле. Для того чтобы увидеть, какие операции выполняются внутри цикла, необходимо переписать алгоритм 13.1 в форме, близкой к языку машины:

```
i <- 1
цикл: if  $z = x_i$  then найдено
      if  $i = n$  then не найдено
      i <- i + 1
      goto цикл
```

За каждую итерацию выполняется до четырех команд: два сравнения, одна операция увеличения и одна передача управления.

Для ускорения внутреннего цикла общим приемом является добавление в таблицу специальных строк, которые делают необязательной явную проверку того, достиг ли указатель границ таблицы. Это можно сделать в алгоритме 13.1. Если перед поиском мы добавим искомое имя z в конце таблицы, то цикл всегда будет завершаться отысканием вхождения z ; таким образом, нам не нужно в цикле каждый раз делать проверку $i = n$. В конце цикла проверка условия $i > n$, выполняемая лишь однажды, говорит о том, является ли найденное вхождение z истинным или специальным элементом таблицы. Это демонстрируется в алгоритме 13.2.

$x_{n+1} \leftarrow z$

```
i <- 1
```

```

while  $z \neq x_i$  do  $i \leftarrow i+1$ 
if  $i \leq n$  then {найденно:  $i$  указывает на  $z$ }
    else {не найденно}

```

Алгоритм 13.2. Улучшенный последовательный поиск

Улучшение алгоритма 13.1 будет наиболее очевидным, если мы перепишем алгоритм 13.2 в тех же близких к языку машины обозначениях, которые использовались раньше:

```

 $x_{n+1} \leftarrow z$ 

 $i \leftarrow 1$ 
цикл: if  $z = x_i$  then goto возможно
         $i \leftarrow i+1$ 
        goto цикл
возможно: if  $i \leq n$  then {найденно:  $i$  указывает на  $z$ }
    else {не найденно}

```

При каждой итерации выполняются лишь три действия вместо четырех, как это было в алгоритме 13.1. Таким образом, в большинстве вычислительных устройств цикл в алгоритме 13.2 будет выполняться гораздо быстрее, чем в алгоритме 13.1, и поскольку скорость цикла определяет скорость всей программы, такое же сравнение имеет место для двух программ.

Печально то, что фокус с добавлением z в конец таблицы перед поиском удастся, только если мы имеем прямой непосредственный доступ к концу таблицы. Это возможно, если таблица хранится в памяти с произвольным доступом, но невозможно в общем случае, когда используется связанное размещение или память с последовательным доступом.

Единственным недостатком алгоритма 13.2 является то, что при безуспешном поиске (поиске имен, которых нет в таблице) всегда просматривается вся таблица. Если такой поиск возникает часто, то имена надо хранить в естественном порядке; это позволяет завершать поиск, как только при просмотре попало первое имя, большее или равное аргументу поиска. В этом случае в конец таблицы следует добавить фиктивное имя ∞ для того, чтобы гарантировать

выполнение условия завершения (∞ - это новое имя, которое по предположению больше любого имени из пространства имен S). Таким образом получаем алгоритм 13.3.

```

 $x_{n+1} \leftarrow \infty$ 

i<-1
  while  $z > x_i$  do i<- i+1
  if  $z=x_i$  then{найденi указывает на  $z$ }
    else{не найдено}

```

Алгоритм 13.3.Последовательный поиск по таблице, хранимой в естественном порядке

Логарифмический поиск в статических таблицах

Мы говорим о логарифмическом времени поиска, как только возникает возможность за время c , не зависящее от n , последовательно свести задачу поиска в таблице, содержащей n имен, к задаче поиска в таблице, содержащей не более αn имен, где $\alpha < 1$ - константа. В этом случае время $t(n)$, требующееся для поиска в таблице с n именами, удовлетворяет рекуррентному соотношению

$$t(n) = c + t(\alpha n),$$

решение, которого имеет вид

$$t(n) = c \log_{1/\alpha} n + b,$$

где b определяется начальными условиями и коэффициент c при логарифме есть время, требуемое для уменьшения размера таблицы от n до αn .

Самыми распространенными предположениями, которые дают возможность уменьшить размер таблицы от n до αn за время, не зависящее от n , являются предположения о том, что пространство имен S линейно упорядочено и что сравнение двух имен x, y из S (для определения $x < y, x = y, x > y$) есть элементарная операция,

требующая постоянного количества времени, не зависящего от n . В результате время, необходимое для большинства логарифмических алгоритмов поиска, естественно измеряется числом сравнений (с тремя исходами) пар имен. Для некоторых алгоритмов, однако, более естественны сравнения с большим, но фиксированным числом исходов.

В этом разделе мы рассматриваем только статические таблицы, то есть таблицы, в которых включение и исключение либо не встречаются, либо так редки, что когда они появляются, строится новая таблица. Динамические структуры таблиц, допускающие логарифмическое время поиска, так же как и эффективные алгоритмы включения и исключения, обсуждаются в конце этой лекции. Для статических таблиц нужно обсудить лишь алгоритмы поиска и построения таблицы. Алгоритмы поиска достаточно просты, но некоторые из алгоритмов построения таблицы сложны. Такая ситуация возникает потому, что в случае статической таблицы разумно считать частоты обращения известными, и, может быть, стоит затратить существенные усилия на построение оптимальной таблицы – таблицы с минимальным средним временем поиска (относительно данных частот обращения). Алгоритмы построения таблиц и их анализ являются наиболее важными темами этого раздела.

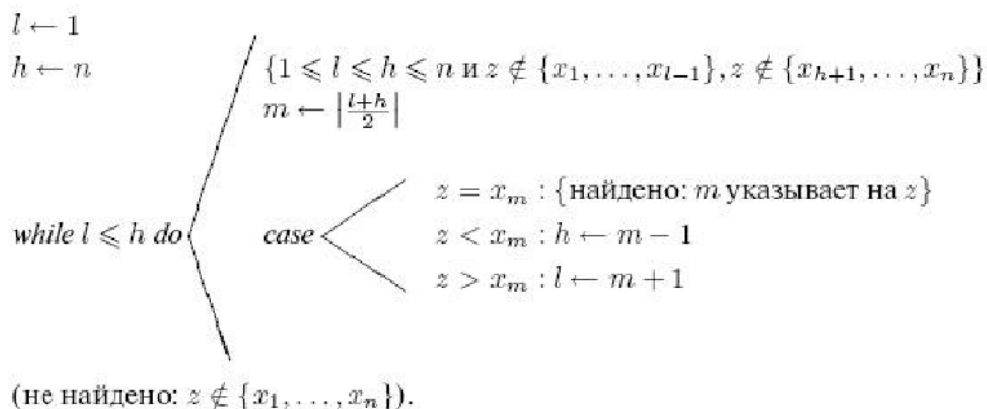
Бинарный поиск

Когда ячейки таблицы последовательно распределены в памяти с произвольным доступом и имена хранятся в таблице в их естественном порядке, возможен бинарный поиск - один из наиболее широко используемых методов поиска. Идея этого метода состоит в том, чтобы искать имя z в интервале, крайними точками которого являются два заданных указателя l (для "низа") и h (для "верха"). Новый указатель m (для "средней точки") устанавливается где-то около середины интервала, и либо z с именем в этой ячейке сводит интервал поиска к одному из интервалов $[l, m - 1]$ или $[m + 1, h]$. Если интервал становится пустым, поиск завершается безуспешно.

Для получения логарифмического времени поиска существенно устанавливать указатель m за время, не зависящее от длины интервала; это требование делает непригодным бинарный поиск на большинстве

вспомогательных запоминающих устройств. Требование, чтобы m помещалось точно в середине интервала, несущественно, хотя выбор средней точки в качестве m обычно дает самый эффективный алгоритм. В некоторых частных случаях полезно разбить интервал на подинтервалы длины $\alpha(h-l+1)$ и $(1-\alpha)(h-l+1)$ для фиксированного значения α , отличного от $\frac{1}{2}$. Когда таблица размещена не последовательно, а хранится в виде списка древовидной структуры, доля α должна, вероятно, меняться от интервала к интервалу.

Бинарный поиск по идее прост, но с деталями условия завершения поиска нужно обращаться осторожно. Частные случаи $h-l=1$ и $h-l=0$ требуют пристального внимания в любой программе бинарного поиска. В алгоритме 13.4 эти случаи обрабатываются тем же кодом, что и в общем случае, и поучительно посмотреть, как это делается, проследив за выполнением алгоритма для $n=2$ и $n=1$.



Корректность алгоритма 13.4 следует из утверждения, данного в комментарии в начале тела цикла. Он устанавливает, что если z находится где-либо в таблице, то оно должно находиться в интервале $[l, h]$; иначе говоря, при нашем предположении, что имя появляется в таблице не больше одного раза, утверждается, что z не встречается ни в интервале $[1, l-1]$, ни в интервале $[h+1, n]$. Это утверждение очевидно первый раз, когда мы входим в цикл при $l=1$ и $h=n$, и непосредственно по индукции проверяется, что оно выполняется при каждом проходе через цикл. Когда мы выходим из цикла, то должно

быть $l > h$, и поэтому утверждение принимает вид $z \notin \{x_1, \dots, x_{l-1}\}$ и $z \notin \{x_{h+1}, \dots, x_n\}$, откуда следует, что $z \notin \{x_1, \dots, x_n\}$.

Оптимальные деревья бинарного поиска

Бинарный поиск в последовательно распределенной таблице (алгоритм 13.4.) обеспечивает очень быстрое нахождение имен, которые являются средними точками на раннем этапе процесса деления пополам, именно имен, близких к вершине дерева $T(1, n)$. Таким образом, любое имя в таблице можно выбрать примерно за $\lg n$ сравнений.

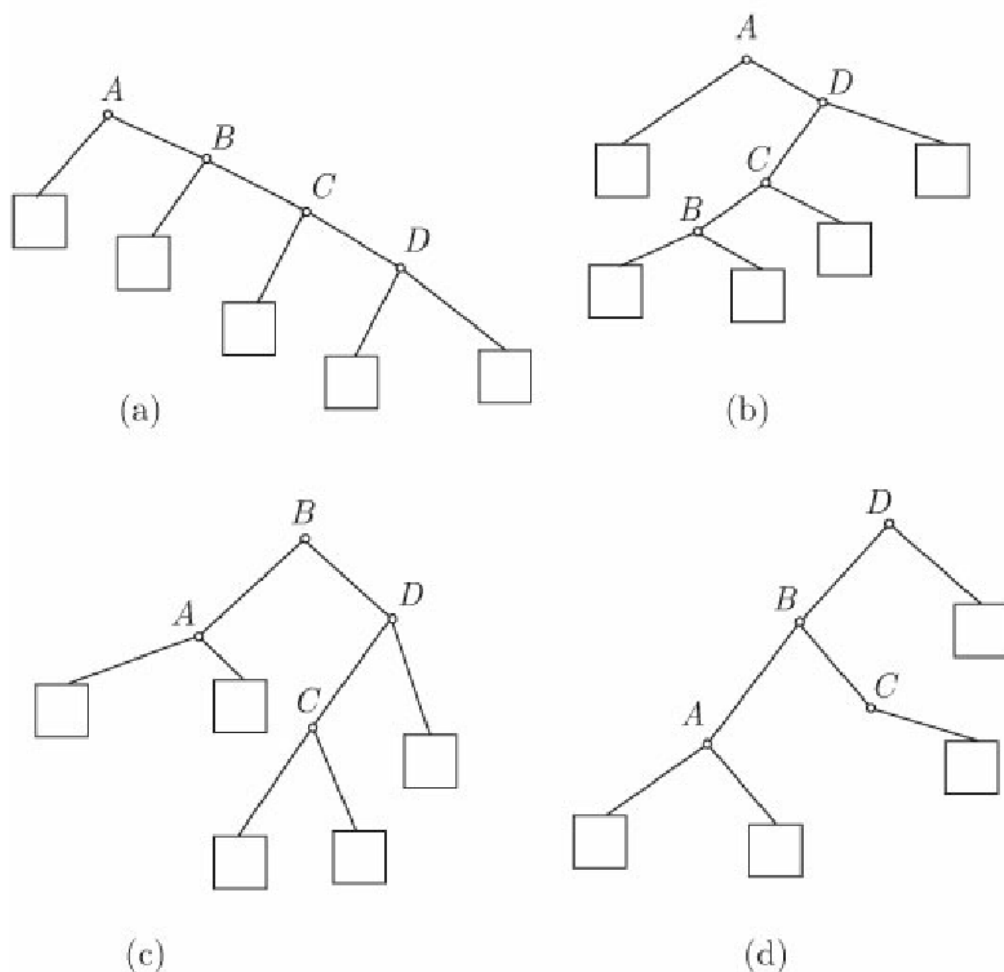


Рис. 13.1. Четыре дерева бинарного поиска над множеством имен {A,B,C,D}

На практике в большинстве таблиц встречаются имена, к которым обращаются гораздо чаще, чем к другим, и "привилегированные" места в таблице разумно постараться использовать для наиболее часто вызываемых имен, а не для имен, выбранных для этих мест в результате бинарного поиска. Это невозможно осуществить для последовательно распределенных таблиц, поскольку место имени определяется его положением относительно естественного порядка имен в таблице. Введем структуру данных, легко приспособляемую как к месту бинарного поиска, так и к возможности выделять точки, в которых

таблица делится на две части.

Деревом бинарного поиска над именами $x_1, x_2, \dots, x_n$ называется расширенное бинарное дерево, все внутренние узлы которого помечены различными именами из списка $x_1, x_2, \dots, x_n$ таким образом, что симметричный порядок узлов совпадает с естественным порядком. Каждый из $n + 1$ внешних узлов соответствует промежутку в таблице. На рис. 13.1 показаны четыре различных дерева бинарного поиска на множестве имен $\{A, B, C, D\}$. Деревья (a) и (b) - вырожденные, поскольку они по существу являются линейными списками, которые должны просматриваться последовательно.

Поиск имени z в дереве бинарного поиска осуществляется путем сравнения z с именем, стоящим в корне. Тогда

1. Если корня нет (дерево пусто), то z в таблице отсутствует и поиск завершается безуспешно.
2. Если z совпадает с именем в корне, поиск завершается успешно.
3. Если z предшествует имени в корне, поиск продолжается ниже в левом поддереве корня.
4. Если z следует за именем в корне, поиск продолжается ниже в правом поддереве корня.

Пусть бинарное дерево имеет вид, в котором каждый узел представляет собой тройку (LEFT, NAME, RIGHT), где LEFT и RIGHT содержат указатели левого и правого сыновей соответственно, и NAME содержит имя, хранящееся в узле. Указатели могут иметь значение Λ , означающее, что поддерево, на которое они указывают пусто. Если указатель корня дерева есть Λ , то само дерево пусто. Как и следовало ожидать, успешный поиск завершается во внутреннем узле дерева бинарного поиска и безуспешный поиск завершается во внешнем узле.

Эта процедура нахождения имени z в таблице, организованная в виде дерева бинарного поиска T , показана в алгоритме 13.5. Отметим его сходство с алгоритмом 13.4 (бинарный поиск).

```

 $p \leftarrow \text{корень } T$ 
while  $p \neq \Lambda$  do case  $\left\{ \begin{array}{l} z = \text{NAME}(p) : \{ \text{найденно: } p \text{ указывает на } z \} \\ z < \text{NAME}(p) : p \leftarrow \text{LEFT}(p) \\ z > \text{NAME}(p) : p \leftarrow \text{RIGHT}(p) \end{array} \right.$ 
{ не найдено:  $z$  нет в таблице}

```

Алгоритм 13.5. Поиск в дереве бинарного поиска

Логарифмический поиск в динамических таблицах

Здесь мы рассмотрим организацию в виде деревьев для таблиц, в которых часто встречаются включения и исключения. Что происходит с временем поиска в дереве, которое модифицировалось путем включения и исключения? Если включенные и исключенные имена выбраны случайно, то оказывается, что в среднем время поиска мало изменяется; но в худшем случае поведение плохое - деревья могут вырождаться в линейные списки, поиск в которых нужно осуществлять последовательно. Проблема вырождения дерева в линейный список, приводящая к времени поиска $O(n)$ вместо $O(\log n)$ в практических приложениях выражена более резко, чем это указывается теоретическим анализом. Такой анализ обычно предполагает, что включения и исключения появляются случайным образом, но на практике часто это не так.

Случайные деревья бинарного поиска. Как ведут себя деревья бинарного поиска без ограничений в качестве динамических деревьев? Другими словами, предположим, что дерево бинарного поиска меняется при случайных включениях и исключениях.

Для включения z мы используем незначительную модификацию алгоритма 13.5. Если z не было найдено, мы получаем для z новую ячейку и связываем ее с последним узлом, пройденным во время безуспешного поиска z . Соответствующее предложение нельзя однако просто добавить в конце алгоритма 13.5, поскольку при нормальном окончании цикла *while* указатель p не указывает больше на последний пройденный узел, а вместо этого имеет значение Λ . В связи с этим мы должны производить включение до того, как выполняется

предположение $p \leftarrow LEFT(p)$ или $p \leftarrow RIGHT(p)$; мы можем осуществить это, сделав процедуру включения рекурсивной. Процедура $INSERT(z, T)$ выдает в качестве значения указатель на дерево, в которое добавлено z . Таким образом, $T \leftarrow INSERT(z, T)$ используется для включения z в T .

```

 $T \leftarrow INSERT(z, T)$ 
procedure  $INSERT(z, T)$ 
   $p \leftarrow \text{корень } T$ 
  if  $p = \Lambda$  then
     $p \leftarrow \text{get-cell}$ 
     $NAME(p) \leftarrow z$ 
     $LEFT(p) \leftarrow RIGHT(p) \leftarrow \Lambda$ 
  else
    case
       $z = NAME(p) : \{\text{уже в таблице}\}$ 
       $z < NAME(p) : LEFT(p) \leftarrow INSERT(z, LEFT(p))$ 
       $z > NAME(p) : RIGHT(p) \leftarrow INSERT(z, RIGHT(p))$ 
   $INSERT \leftarrow p$ 
  return
  
```

Алгоритм 13.6. Включение в дерево бинарного поиска

Исключение гораздо сложнее включения, и мы изложим здесь только основную идею. Если подлежащее удалению имя z имеет самое большее одного сына, то при исключении z его сын (если он вообще есть) объявляется сыном отца z . Если z имеет двух сыновей, его прямо удалить нельзя. Вместо этого мы находим в таблице либо имя y_1 , которое непосредственно предшествует z , либо имя y_2 , которое непосредственно следует за z в естественном порядке. Оба имени принадлежат узлам, которые имеют не больше одного сына, и, таким образом, z можно исключить заменой его либо именем y_1 , либо y_2 , и затем исключением узла, который содержал y_1 или y_2 , соответственно.

Сбалансированные сильно ветвящиеся деревья

Деревья бинарного поиска естественным образом обобщаются до m -арных деревьев поиска, в которых каждый узел имеет $k \leq m$ сыновей и содержит $k - 1 \leq m - 1$ имен. Имена в узле делят множество имен

на k подмножеств, каждое подмножество соответствует одному из k поддеревьев узла. На рис. 13.2 показано полностью заполненное 5-арное дерево с двумя уровнями. Заметим, что мы не можем требовать, чтобы каждый узел m -арного дерева имел ровно m сыновей и включал равно $m - 1$ имен; если мы захотим включить Z в дерево на рисунке 13.2, то должны будем создать узлы с меньше чем m сыновьями и меньше чем $m - 1$ именами. Таким образом, определение m -арного дерева утверждает только, что каждый узел имеет не более m сыновей и содержит не более $m - 1$ имен. Ясно, что на m -арных деревьях можно осуществлять поиск так же, как и на бинарных деревьях.

Как и в деревьях бинарного поиска, полезно различать внутренние узлы и листья. Внутренний узел содержит $k \geq 1$ имен, записанных в естественном порядке, и имеет $k + 1$ сыновей, каждый из которых может быть либо внутренним узлом, либо листом. Лист не содержит имен (разве что временно в процессе включения), и, как раньше, в листьях - завершаются безуспешные поиски. Обычно за очевидностью мы на рисунке их опускаем.

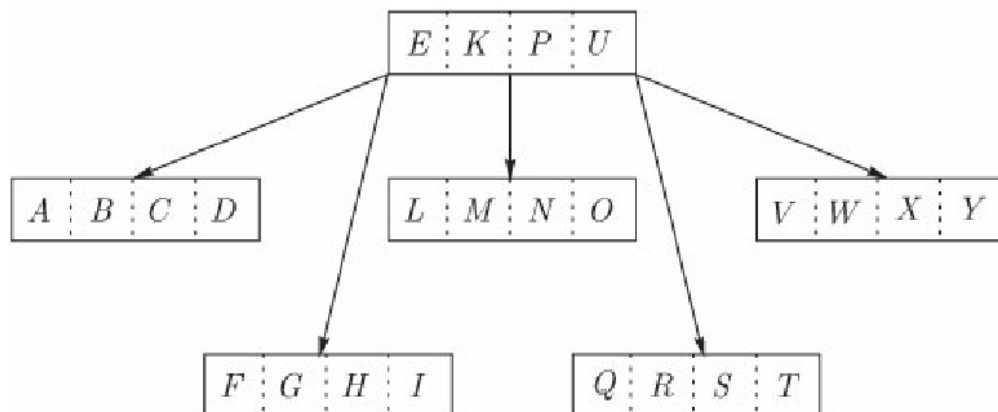


Рис. 13.2. Абсолютно сбалансированное, полностью заполненное 5-арное дерево поиска

Сбалансированное сильно ветвящееся дерево порядка m есть m -арное дерево в котором:

1. Все листья расположены на одном уровне.

2. Корень имеет k сыновей, $2 \leq k$.
3. Другие внутренние узлы имеют k' сыновей, $m/2 \leq k' \leq m$.

Сортировка (часть 1)

Введение. Внутренняя сортировка. Вставка. Обменная сортировка.

Введение

Рассматриваемые здесь задачи можно отнести к наиболее часто встречающимся классам комбинаторных задач. Почти во всех машинных приложениях множество объектов должно быть переразмещено в соответствии с некоторым заранее определенным порядком. Например, при обработке коммерческих данных часто бывает необходимо расположить их по алфавиту или по возрастанию номеров. В числовых расчетах иногда требуется знать наибольший корень многочлена, и т.д.

Будем считать заданной таблицу с n именами, обозначаемыми $x_1, x_2, \dots, x_n$. Каждое имя x_i принимает значение из пространства имен, на котором определен линейный порядок. Будем считать, что никакие два имени не имеют одинаковых значений; то есть любые x_i, x_j обладают тем свойством, что если $i \neq j$, то либо $x_i < x_j$, либо $x_i > x_j$. Ограничение $x_i \neq x_j$ при $i \neq j$ упрощает анализ без потери общности, ибо и при наличии равных имен корректность идей и алгоритмов не нарушается. Наша цель состоит в том, чтобы выяснить что-нибудь относительно перестановки $\Pi = (\pi_1, \pi_2, \dots, \pi_n)$ для которой $x_{\pi_1} < x_{\pi_2} < \dots < x_{\pi_n}$. В задаче полной сортировки требуется полностью определить Π , хотя обычно это делается неявно путем переразмещения имени в порядке возрастания. В задачах частичной сортировки требуется либо извлечь частичную информацию о Π (например, Π_i для нескольких значений i), либо полностью определить Π по некоторой заданной частичной информации о ней (так обстоит дело при слиянии двух упорядоченных таблиц).

При внутренней сортировке решается задача полной сортировки для случая достаточно малой таблицы, уместящейся непосредственно в адресной памяти. Внешняя сортировка представляет собой задачу полной сортировки для случая такой большой таблицы, что доступ к ней организован по частям, расположенным на внешних

запоминающих устройствах. Частичная сортировка - задачи выбора i -наибольшего имени и слияния двух упорядоченных таблиц.

Внутренняя сортировка

Существует по крайней мере пять широких классов алгоритмов внутренней сортировки.

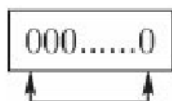
1. Вставка. На i -м этапе i -е имя помещается на надлежащее место между $i-1$ уже отсортированными именами:



Не отсортированы

Уже отсортированы

1. Обмен. Два имени, расположение которых не соответствует порядку, меняются местами (обмен). Эти процедуры повторяются до тех пор, пока остаются такие пары.



1. Выбор. На i -м этапе из неотсортированных имен выбирается i -е наибольшее (наименьшее) имя и помещается на соответствующее место.

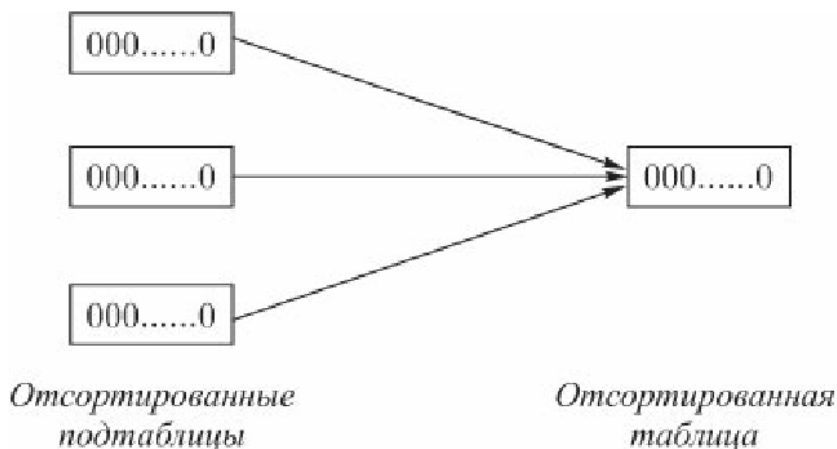


Не отсортированы

Уже отсортированы

1. Распределение. Имена распределяются по группам, и содержимое групп затем объединяется таким образом, чтобы частично отсортировать таблицу; процесс повторяется до тех пор, пока таблица не будет отсортирована полностью.

2. Слияние. Таблица делится на подтаблицы, которые сортируются по отдельности и затем сливаются в одну.



Эти классы нельзя назвать ни взаимоисключающими, ни исчерпывающими: одни алгоритмы сортировки можно с полным основанием отнести более чем к одному классу (пузырьковую сортировку можно рассматривать и как выбор, и как обмен), а другие не укладываются ни в один из классов. Тем не менее, перечисленные пять классов достаточно удобны для классификации обсуждаемых алгоритмов сортировки.

Сосредоточим внимание на первых четырех классах алгоритмов сортировки. Алгоритмы, основанные на слиянии, приемлемы для внутренней сортировки, но более естественно рассматривать их как методы внешней сортировки.

В описываемых алгоритмах сортировки имена образуют последовательность, которую будем обозначать $x_1, x_2, \dots, x_n$ независимо от возможных пересылок данных; таким образом, значением x_i является любое текущее имя в i -й позиции последовательности. Многие алгоритмы сортировки наиболее применимы к массивам; в этом случае x_i обозначает i -й элемент массива. Другие алгоритмы более приспособлены для работы со связанными списками: здесь x_i обозначает i -й элемент списка. Следующие обозначения используются для пересылок данных:

$x_i \leftrightarrow x_j$ значения x_i и x_j меняются местами. $x_i \leftarrow y$ значение y

присваивается имени x_i . $y \leftarrow x_j$ значение имени x_j присваивается y .

Таким образом, операция $x_i \leftarrow x_j$, которая встречается в различных алгоритмах сортировки, временно нарушает предположение о том, что никакие два имени не имеют одинаковых значений. Однако это условие всегда обязательно восстанавливается.

В каждом из рассматриваемых алгоритмов будем считать, что имена нужно сортировать на месте. Другими словами, перерасмещение имен должно происходить внутри последовательности $x_1, x_2, \dots, x_n$; при этом существуют одна или две дополнительные ячейки, в которых временно размещается значение имени. Ограничение "на месте" основано на предположении, будто число имен настолько велико, что во время сортировки не допускается перенос их в другую область памяти. Если в распоряжении имеется память, достаточная для такого переноса, то некоторые из обсуждаемых алгоритмов можно значительно ускорить. Эти рассуждения заставляют нас в алгоритмах распределяющей сортировки и сортировки слиянием реализовать последовательность $x_1, x_2, \dots, x_n$ как связанный список.

Вставка

Вставка - простейшая сортировка вставками проходит через этапы $j = 2, 3, \dots, n$: на этапе j имя x_j вставляется на свое правильное место среди $x_1, x_2, \dots, x_{j-1}$.

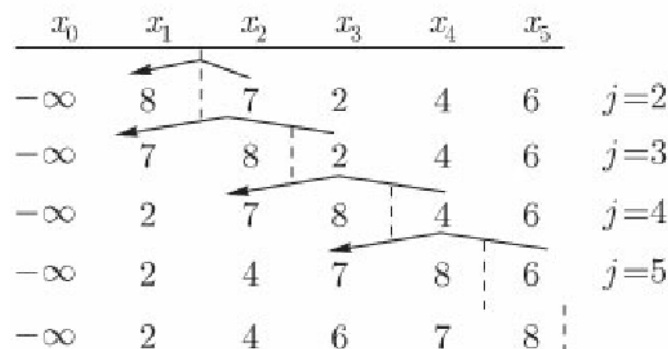
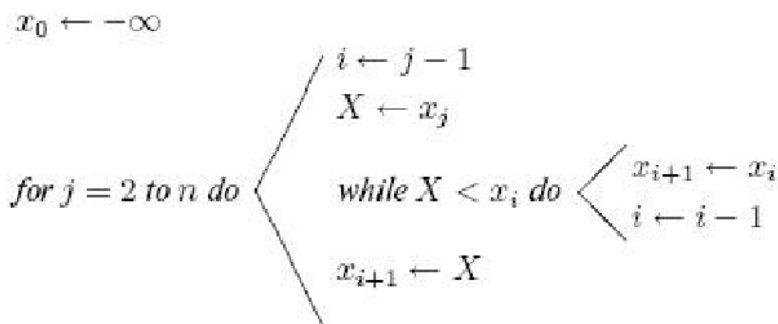


Рис. 14.1. Простая сортировка вставками, используемая на таблице из $n = 5$ имен. Пунктирные вертикальные линии разделяют уже отсортированную часть таблицы и еще не отсортированную

При вставке имя x_j временно размещается в X , и просматриваются имена $x_{j-1}, x_{j-2}, \dots, x_i$; они сравниваются с X и сдвигаются вправо, если обнаруживается, что они больше X . Имеется фиктивное имя x_0 , значение которого $-\infty$ служит для остановки просмотра слева. На рис. 14.1 работа этого алгоритма проиллюстрирована на примере таблицы из пяти имен.



Алгоритм 14.1. Простая сортировка вставками

Эффективность этого алгоритма, как и большинства алгоритмов сортировки, зависит от числа сравнений имен и числа пересылок данных, осуществляемых в трех случаях: худшем, среднем (в предположении, что все $n!$ перестановок равновероятны) и лучшем.

Обменная сортировка

Обменная сортировка некоторым систематическим образом меняет местами пары имен, не отвечающие порядку, до тех пор, пока такие пары существуют. Фактически алгоритм 14.1 можно рассматривать как обменную сортировку, в которой имя x_j меняется местами со своим соседом слева, пока не оказывается на правильном месте. В этом разделе мы обсуждаем два типа обменных сортировок: хорошо известную, но относительно неэффективную пузырьковую сортировку и быструю сортировку — один из лучших со всех точек зрения

алгоритмов внутренней сортировки.

Пузырьковая сортировка. Наиболее очевидный метод систематического обмена местами имен с неправильным порядком состоит в просмотре пар смежных имен последовательно слева направо и перемене мест тех имен, которые не отвечают порядку.

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8		d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8
	4	7	3	1	5	8	2	6		0	0	2	3	1	0	5	2
1-й проход	4	3	1	5	7	2	6	8		0	1	2	0	0	4	1	0
2-й проход	3	1	4	5	2	6	7	8		0	1	0	0	3	0	0	0
3-й проход	1	3	4	2	5	6	7	8		0	0	0	2	0	0	0	0
4-й проход	1	3	2	4	5	6	7	8		0	0	1	0	0	0	0	0
5-й проход	1	2	3	4	5	6	7	8		0	0	0	0	0	0	0	0
6-й проход	1	2	3	4	5	6	7	8		0	0	0	0	0	0	0	0

Рис. 14.2. Пузырьковая сортировка, примененная к таблице. Показан вектор инверсии таблицы после каждого прохода

Эта техника получила название пузырьковой сортировки, так как большие имена "пузырьками всплывают" вверх (то есть на правый конец) таблицы. В алгоритме 14.2 эта простая идея реализуется с одним небольшим усовершенствованием: ясно, что не имеет смысла продолжать просмотр для больших имен (в правом конце таблицы), про которые известно, что они находятся на своих окончательных позициях. В алгоритме 14.2 используется переменная b , значение которой в начале цикла *while* равно наибольшему индексу t , такому, что имя x_t еще не известно, стоит ли оно в окончательной позиции. На рис. 14.2 показана работа алгоритма на примере таблицы с $n = 8$ именами.

```

 $b \leftarrow n$ 
while  $b \neq 0$  do
     $t \leftarrow 0$ 
    for  $j = 1$  to  $b - 1$  do if  $x_j > x_{j+1}$  then
         $x_j \leftrightarrow x_{j+1}$ 
         $t \leftarrow j$ 
     $b \leftarrow t$ 

```

Алгоритм 14.2. Пузырьковая сортировка

Анализ пузырьковой сортировки зависит от трех факторов: числа проходов (то есть числа выполнений тела цикла *while*), числа сравнений $x_j > x_{j+1}$ и числа обменов $x_j \leftrightarrow x_{j+1}$. Число обменов равно, как в алгоритме 14.1, числу инверсий: 0 в лучшем случае, $\frac{1}{2}n(n-1)$ в худшем случае и $\frac{1}{4}n(n-1)$ - в среднем. Рисунок 14.2 дает возможность предположить, что каждый проход пузырьковой сортировки, исключая последний, уменьшает на единицу каждый ненулевой элемент вектора инверсий и циклически сдвигает вектор на одну позицию влево; легко доказать, что это верно в общем случае, и поэтому число проходов равно единице плюс наибольший элемент вектора инверсий. В лучшем случае имеется всего один проход, в худшем случае - n проходов и в среднем - $\sum kP_k$ проходов, где P_k - вероятность того, что наибольшим элементом вектора инверсии является $k-1$. Общее число сравнений имен трудно определить, но можно показать, что оно равно $n-1$ в лучшем случае, $\frac{1}{2}n(n-1)$ в худшем случае и $\frac{1}{2}(n^2 - n \ln n) + O(n)$ - в среднем.

Пузырьковую сортировку можно несколько улучшить, но при этом она все еще не сможет конкурировать с более эффективными алгоритмами сортировки. Ее единственным преимуществом является простота.

Как в простой сортировке вставками, так и в пузырьковой сортировке (алгоритм 14.2) основной причиной неэффективности является тот факт, что обмены дают слишком малый эффект, так как в каждый момент времени имена сдвигаются только на одну позицию. Такие алгоритмы непременно требуют порядка n^2 операций, как в среднем, так и в худшем случаях.

Быстрая сортировка. Идея метода быстрой сортировки состоит в том, чтобы выбрать одно из имен в таблице и использовать его для разделения таблицы на две подтаблицы, составленные соответственно из имен меньших и больших выбранного, которые затем рекурсивно сортируются с использованием быстрой сортировки. Разделение можно реализовать, одновременно просматривая таблицу и слева направо, и справа налево, меняя местами имена в неправильных частях таблицы. Имя, используемое для расщепления таблицы, затем помещается между двумя подтаблицами, и две подтаблицы сортируются рекурсивно.

В алгоритме 14.3 показаны детали быстрой сортировки для сортировки таблицы $(x_f, x_{f+1}, \dots, x_l)$, где x_j используется для разбиения таблицы на подтаблицы. На рис. 14.3 показано, как алгоритм 14.3 использует два указателя i и j для просмотра таблицы во время разбиения. В начале цикла "*while $i < j$* " i и j указывают соответственно на первое и последнее имена, о которых известно, что они находятся не в тех частях файла, в которых требуется. Когда $i < j$ встречаются, то есть когда $i \geq j$, все имена находятся в соответствующих частях таблицы и x_f помещается между двумя частями, меняясь при этом местами с x_j , алгоритм предполагает, что имя x_{i+1} определено и больше, чем $x_f, x_{f+1}, \dots, x_l$.

```

procedure QUICKSORT( $f, l$ )
  {отсортировать  $x_f, x_{f+1}, \dots, x_l$ }
  if  $f \geq l$  then return
  {разделить таблицу}
   $i \leftarrow f + 1$ 
  while  $x_i < x_f$  do  $i \leftarrow i + 1$ 
   $j \leftarrow l$ 
  while  $x_j < x_f$  do  $j \leftarrow j - 1$ 
  {В ЭТОТ МОМЕНТ ИМЕЕМ  $i < j, x_i \geq x_f \geq x_j$ }
   $x_i \leftrightarrow x_j$ 
  while  $i < j$  do
     $i \leftarrow i + 1$ 
    while  $x_i < x_f$  do  $i \leftarrow i + 1$ 
     $j \leftarrow j - 1$ 
    while  $x_j < x_f$  do  $j \leftarrow j - 1$ 
   $x_f \leftrightarrow x_j$ 
  {отсортировать подтаблицы рекурсивно}
  QUICKSORT( $f, j - 1$ )  QUICKSORT( $j + 1, l$ )  return

```

	x_f	x_{f+1}	x_{f+2}									x_{l-1}	x_l
Начало	27	99	0	8	13	64	86	16	7	10	88	25	90
		$\uparrow$ i										$\uparrow$ j	
Первый обмен	27	99	0	8	13	64	86	16	7	10	88	25	90
Второй обмен	27	25	0	8	13	64	86	16	7	10	88	99	90
						$\uparrow$ i				$\uparrow$ j			
Третий обмен	27	25	0	8	13	10	86	16	7	64	88	99	90
						$\uparrow$ i			$\uparrow$ j				
Просмотры встретились	27	25	0	8	13	10	7	16	86	64	88	99	90
								$\uparrow$ i	$\uparrow$ j				
x_f помещается на свое место	27	25	0	8	13	10	7	16	86	64	88	99	90
								$\uparrow$ j	$\uparrow$ i				
Разделенная таблица	16 25 0 8 13 10 7							27	86 64 88 99 90				

Рис. 14.3.

Алгоритм 14.3 изящен, но непрактичен. Проблема состоит в том, что рекурсия используется для записи подтаблиц, которые рассматриваются на более поздних этапах, и в худших случаях (когда таблица уже отсортирована) глубина рекурсии может равняться n . Следовательно, для стека, реализующего рекурсию, необходима память, пропорциональная n ; для больших n такое требование становится неприемлемым. Кроме того, второе рекурсивное обращение к быстрой сортировке в алгоритме 14.3 может быть легко исключено. По этим причинам мы предлагаем алгоритм 14.4, итерационный вариант быстрой сортировки, в которой стек ведется явно. Элементом стека является пара (f, l) : когда пара находится в стеке, это значит, что нужно сортировать соответствующие $x_f, \dots, x_l$. Алгоритм 14.4 помещает в стек большую из двух подтаблиц и немедленно применяет алгоритм к меньшей подтаблице. Это уменьшает глубину стека в худшем случае примерно до $\lg n$. Заметим, что подтаблицы длины 1 игнорируются и что расщепление подтаблицы делается с использованием случайно выбранного имени в этой подтаблице.


```

 $S \leftarrow$  пустой стек
 $S \leftarrow (0, 0)$ 
 $f \leftarrow 1$ 
 $x_{n+1} \leftarrow \infty$ 
 $l \leftarrow n$ 
  while  $f < l$  do
     $x_f \leftarrow x_{\text{rand}(f, l)}$ 
     $i \leftarrow f + 1$ 
    while  $x_i < x_f$  do  $i \leftarrow i + 1$ 
     $j \leftarrow l$ 
    while  $x_j > x_f$  do  $j \leftarrow j - 1$ 
    while  $i < j$  do
       $x_i \leftrightarrow x_j$ 
       $i \leftarrow i + 1$ 
      while  $x_i < x_f$  do  $i \leftarrow i + 1$ 
       $j \leftarrow j - 1$ 
      while  $x_j > x_f$  do  $j \leftarrow j - 1$ 
     $x_f \leftrightarrow x_j$ 
    Case
       $j - 1 \leq f$  and  $l \leq j + 1$ : {обе таблицы тривиальны}
       $(f, l) \leftarrow S$ 
       $j - 1 \leq f$  and  $l \leq j + 1$ : {нетривиальна только правая таблица}
       $f \leftarrow j + 1$ 
       $j - 1 \leq f$  and  $l \leq j + 1$ : {нетривиальна только левая таблица}
       $l \leftarrow j - 1$ 
       $j - 1 > f$  and  $l > j + 1$ : {никакая подтаблица не является: тривиальной: поместить большую в  $S$ }
      if  $j - f > l - j$  then {левая подтаблица длиннее}
         $S \leftarrow (f, j - 1)$ 
         $f \leftarrow j + 1$ 
      else {правая таблица длиннее}
         $S \leftarrow (j + 1, l)$ 
         $l \leftarrow j - 1$ 

```

Алгоритм 14.4. Итерационный вариант быстрой сортировки

Сортировка (часть 2)

Выбор. Распределяющая сортировка. Цифровая распределяющая сортировка. Внешняя сортировка. Частичная сортировка. Частичная сортировка (выбор). Частичная сортировка (слияние).

Выбор

В сортировке посредством выбора основная идея состоит в том, чтобы идти по шагам $i = 1, 2, \dots, n$, находя i -е наибольшее (наименьшее) имя и помещая его на его место на i -ом шаге. Простейшая форма сортировки выбором представлена алгоритмом 15.1: i -е наибольшее имя находится очевидным способом просмотром оставшихся $n - i + 1$ имен. Число сравнений имен на i -ом шаге равно $n - i$, что приводит к общему числу сравнений имен $(n - 1) + (n - 2) + \dots + 1 = \frac{1}{2}n(n - 1)$ независимо от входа, поэтому ясно, что это не очень хороший способ сортировки.

$$\text{for } j = n \text{ to } 2 \text{ by } -1 \text{ do } \begin{cases} i \leftarrow 1 \\ \text{for } k = 2 \text{ to } j \text{ do if } x_i < x_k \text{ then } i \leftarrow k \\ x_i \leftrightarrow x_j \end{cases}$$

Алгоритм 15.1. Простая сортировка выбором

Несмотря на неэффективность алгоритма 15.1, идея выбора может привести и к эффективному алгоритму сортировки. Весь вопрос в том, чтобы найти более эффективный метод определения i -го наибольшего имени, чего можно добиться, используя механизм турнира с выбыванием. Суть его такова: сравниваются $x_1 : x_2, x_3 : x_4, x_5 : x_6, \dots, x_{n-1} : x_n$, затем сравниваются "победители" (то есть большие имена) этих сравнений и т.д.; эта процедура для $n = 16$ показана на рис. 15.1. Заметим, что для определения наибольшего имени этот процесс требует $n - 1$ сравнений имен; но, определив наибольшее имя, мы обладаем большим объемом информации о втором по величине (в порядке убывания) имени: оно должно быть одним из тех, которые "потерпели поражение"

от наибольшего имени. Таким образом, второе по величине имя теперь можно определить, заменяя наибольшее имя на $-\infty$ и вновь осуществляя сравнение вдоль пути от наибольшего имени к корню. На рис. 15.2 эта процедура показана для дерева из рис. 15.1.

Идея турнира с выбыванием прослеживается при сортировке весьма отчетливо, если имена образуют пирамиду. Пирамида - это полностью сбалансированное бинарное дерево высоты h , в котором все листья находятся на расстоянии h или $h - 1$ от корня и все потомки узла меньше его самого; кроме того, в нем все листья уровня максимально смещены влево. На рис. 15.3 показано множество имен, организованных в виде пирамиды. Чтобы получить удобное линейное представление дерева, пирамиду можно хранить по уровням в одномерном массиве: сыновья имени из i -ой позиции есть имена в позициях $2i$ и $2i + 1$. Таким образом, пирамида, представленная на рисунке 15.3, принимает вид

i:	1	2	3	4	5	6	7	8	9	10	11	12
x_i :	94	93	75	91	85	44	51	18	48	58	10	34

Заметим, что в пирамиде наибольшее имя должно находиться в корне и, таким образом, всегда в первой позиции массива, представляющего пирамиду. Обмен местами первого имени с n -м помещает наибольшее имя в его правильную позицию, но нарушает свойство пирамидальности в первых $n - 1$ именах. Если мы можем сначала построить пирамиду, а затем эффективно восстановить ее, то все в порядке, так как тогда можно производить сортировку следующим образом: построить пирамиду из $x_1, x_2, \dots, x_n$,

$$\text{for } i = n \text{ to } 2 \text{ by } -1 \text{ do } \begin{cases} x_1 \leftrightarrow x_i \\ \text{восстановить пирамиду} \\ \text{в } x_1, \dots, x_{i-1} \end{cases}$$

Это общее описание пирамидальной сортировки.

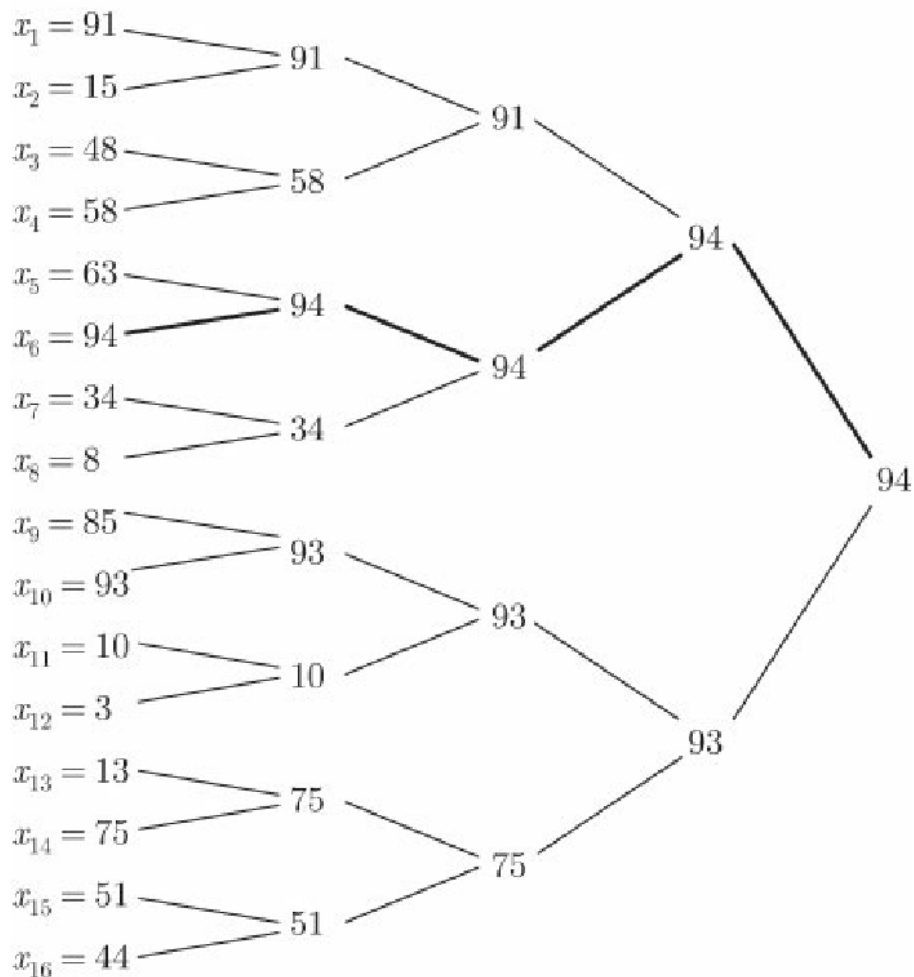


Рис. 15.1. Использование турнира с выбыванием для отыскания наибольшего имени. Путь наибольшего имени показан жирной линией

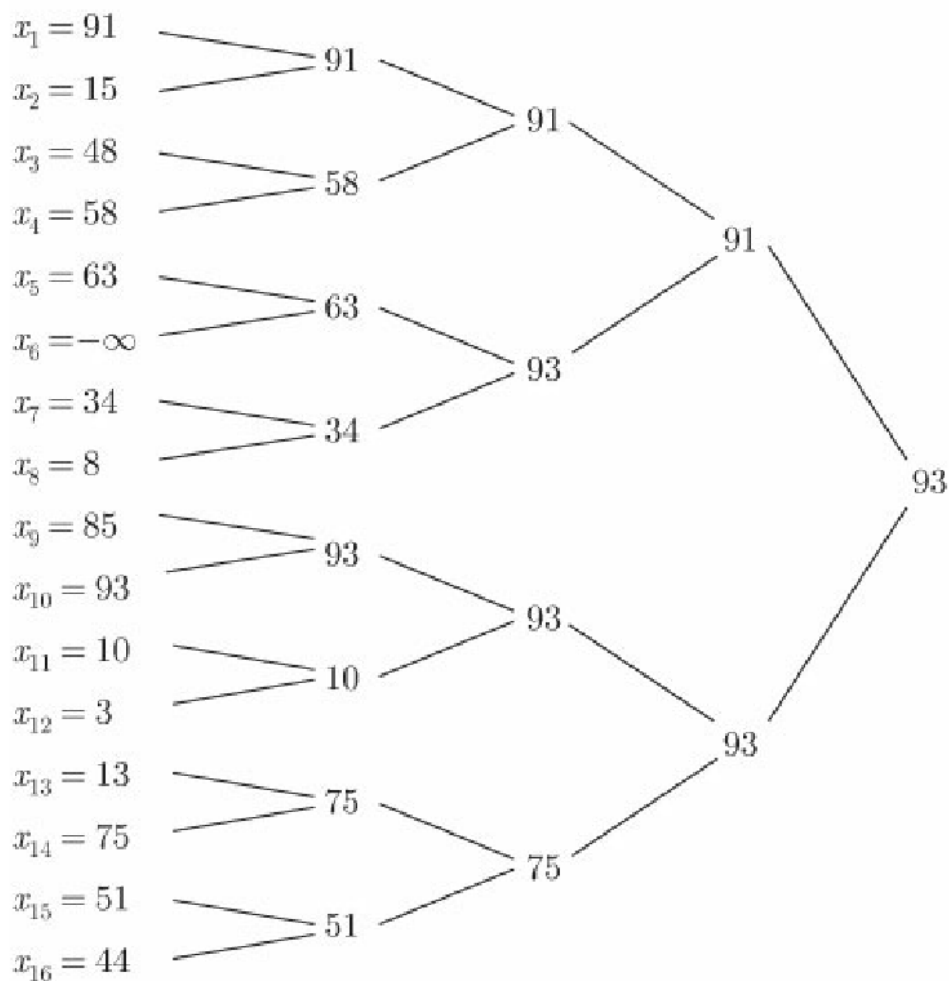


Рис. 15.2.

Процедура **RESTORE** (j, k) восстановления пирамиды из последовательности $x_j, x_{j+1}, \dots, x_k$ в предположении, что все поддеревья суть пирамиды, такова:

```

procedure RESTORE $j, k$ 
  if  $x_j \neq lucm$  then  $\left\{ \begin{array}{l} \text{пусть } x_m \text{ есть больший из сыновей } x_j \\ \text{if } x_m > x_j \text{ then } \left\{ \begin{array}{l} x + m \leftrightarrow x_j \\ \text{RESTORE}(m, k) \end{array} \right. \end{array} \right.$ 
  return

```

Переписывая это итеративным способом и дополняя деталями, мы получим алгоритм 15.2.

```

procedure RESTORE $f, l$ 
   $i \leftarrow f$ 
  while  $j \leq [\frac{1}{2}l]$  do  $\left\{ \begin{array}{l} \text{if } 2j < l \text{ and } x_{2j} < x_{2j+1} \text{ then } m \leftarrow 2j + 1 \\ \hspace{10em} \text{else } m \leftarrow 2j \\ \text{if } x_m > x_j \text{ then } \left\{ \begin{array}{l} x_m \leftrightarrow x_j \\ j \leftarrow m \end{array} \right. \\ \hspace{10em} \text{else } j \leftarrow m \end{array} \right.$ 
  return

```

Алгоритм 15.2. Восстановление пирамиды из дерева, поддеревья которого суть пирамиды

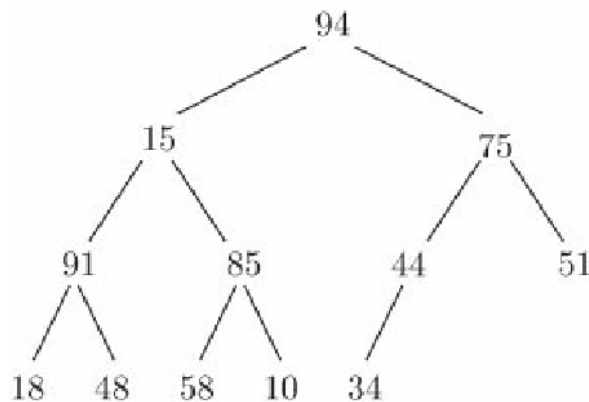


Рис. 15.3. Пирамида, содержащая 12 имен

```

for  $i = \lceil \frac{1}{2}n \rceil$  to 1 by -1 do RESTORE( $i, n$ )
for  $i = n$  to 2 by -1 do  $\begin{cases} x_1 \leftrightarrow x_i \\ \text{RESTORE}(1, i-1) \end{cases}$ 

```

Алгоритм 15.3. Пирамидальная сортировка

Распределяющая сортировка

Обсуждаемый здесь алгоритм сортировки отличается от рассматривавшихся до сих пор тем, что он основан не на сравнениях между именами, а на представлении имен. Мы полагаем, что каждое из имен $x_1, x_2, \dots, x_n$ имеет вид

$$x_i = (x_{i,p}, x_{i,p-1}, \dots, x_{i,1})$$

и их нужно отсортировать в возрастающем лексикографическом порядке, то есть

$$x_i = (x_{i,p}, x_{i,p-1}, \dots, x_{i,1}) < (x_{j,p}, x_{j,p-1}, \dots, x_{j,1}) = x_j$$

тогда и только тогда, если для некоторого $t \leq p$ имеем $x_{i,l} = x_{j,l}$ для $l > t$ и $x_{i,t} < x_{j,t}$. Для простоты будем считать, что $0 \leq x_{i,l} < r$, и поэтому имена можно рассматривать как целые, представленные по основанию r , так что каждое имя имеет pr -ичных цифр. Более короткие имена дополняются нулями.

Цифровая распределяющая сортировка

Цифровая распределяющая сортировка основана на наблюдении, что если имена уже отсортированы по младшим разрядам $l, l-1, \dots, 1$, то их можно полностью отсортировать, сортируя только по старшим разрядам $p, p-1, \dots, i+1$ при условии, что сортировка осуществляется таким образом, чтобы не нарушить относительный порядок имен с одинаковыми цифрами в старших разрядах. Заметим, что к самой таблице обращаются по правилу "первым включается –

первым исключается", и поэтому лучшим способом представления являются очереди. В частности, предположим, что с каждым ключом x_i ассоциируется поле связи $LINK_i$; тогда эти поля связи можно использовать для сцепления всех имен в таблице вместе во входную очередь Q . При помощи полей связи можно также сцеплять имена в очереди $Q_0, Q_1, \dots, Q_{r-1}$, используемые для представления стопок. После того как имена распределены по стопкам, очереди, представляющие эти стопки, связываются вместе для получения вновь таблицы Q . Алгоритм 15.4. представляет эту процедуру в общих чертах (очереди описаны в [лекции 11](#)). В результате применения алгоритма очередь Q будет содержать имена в порядке возрастания; то есть имена будут связаны в порядке возрастания полями связи, начиная с головы очереди Q .

Использовать поля связи $LINK_1, \dots, LINK_n$ для формирования $x_1, \dots, x_n$ во входную очередь Q

```

for j = 1 to p do {
  Вначале сделать очереди  $Q_0, Q_1, \dots, Q_{r-1}$  пустыми
  while  $Q$  не пуста do {
     $X \Leftarrow Q$ 
    пусть  $X = (x_p, x_{p-1}, \dots, x_1)$ 
     $Q_{x_j} \Leftarrow X$ 
  }
  сцепить очереди  $Q_0, Q_1, \dots, Q_{r-1}$  вместе
  для формирования новой очереди
}

```

Алгоритм 15.4. Цифровая распределяющая сортировка

Внешняя сортировка

В методах сортировки, обсуждавшихся в предыдущем разделе, мы полагали, что таблица умещается в быстродействующей внутренней памяти. Хотя для большинства реальных задач обработки данных это предположение слишком сильно, оно, как правило, выполняется для комбинаторных алгоритмов. Сортировка обычно используется только для некоторого сокращения времени работы алгоритмов, в которых сортировка применяется только для некоторого сокращения времени работы алгоритмов, когда оно недопустимо велико даже для задач

"умеренных" размеров. Например, часто бывает необходимо сортировать отдельные предметы во времени исчерпывающего поиска (лекция 13), но поскольку такой поиск обычно требует экспоненциального времени, маловероятно, что подлежащая сортировке таблица будет настолько большой, чтобы потребовалось использование запоминающих устройств. Однако задача сортировки таблицы, которая слишком велика для основной памяти, служит хорошей иллюстрацией работы с данными большого объема, и поэтому в этом разделе мы обсудим важные идеи внешней сортировки. Более того, будем рассматривать только сортировку таблицы путем использования вспомогательной памяти с последовательным доступом. Условимся называть эту память лентой.

Общей стратегией в такой внешней сортировке является использование внутренней памяти для сортировки имен из ленты по частям так, чтобы производить исходные отрезки (известные также как цепочки) имен в возрастающем порядке. По мере порождения эти отрезки распределяются по t рабочим лентам, и затем производится их слияние по t отрезков обратно на исходную $(t + 1)$ -ю ленту так, что она будет содержать меньшее число более длинных отрезков. Затем отрезки снова распределяются по остальным t лентам, и снова производится их слияние по t штук обратно на $(t + 1)$ -ю ленту. Процесс продолжается до тех пор, пока не получится один отрезок, то есть пока таблица не будет полностью отсортирована. Имеются, таким образом, две отдельные проблемы: как порождать исходные отрезки и как осуществлять слияние.

Самый очевидный метод для получения исходных отрезков состоит в том, что можно просто считывать $(t + 1)$ -ю ленту m имен, рассортировывать их во внутренней памяти и записывать их на ленту в виде отрезка, продолжая процесс до тех пор, пока не будут исчерпаны все имена. Все полученные таким образом исходные отрезки содержат m имен (исключая, возможно, последний отрезок). Поскольку число исходных отрезков в конце концов определяет время слияния, мы хотели бы найти некоторый метод образования более длинных исходных отрезков и, следовательно, меньшего их количества. Это можно сделать, используя для сортировки идею турнира (пирамидальную сортировку). При этом подходе m имен, которые умещаются в памяти, хранятся в

виде такой пирамиды, что сыновья узла больше узла (вместо того, чтобы быть меньше его). Этот метод отвечает определению "победителя" при сравнении имен в сортировках типа турнира как меньшего из двух имен, и это позволяет нам следить за наименьшим именем.

Порождение исходных отрезков продолжается следующим образом. Из входной ленты считываются первые m имен, и затем из них формируется пирамида, как описано выше. Наименьшее имя выводится как первое в первом отрезке и заменяется в пирамиде следующим именем из входной ленты в соответствии с алгоритмом 15.2. модифицированным так, чтобы для восстановления пирамиды следить за наименьшим, а не за наибольшим именем. Процесс, известный как выбор с замещением, продолжается таким образом, что к текущему отрезку всегда добавляется наименьшее в пирамиде имя, большее или равное имени, которое последним добавлено к отрезку; при этом добавленное имя заменяется на следующее из входной ленты и восстанавливается пирамида. Когда в пирамиде нет имен, больших, чем последнее имя в текущем отрезке, отрезок обрывается и начинается новый. Этот процесс продолжается до тех пор, пока все имена не сформируются в отрезки.

Разумный путь реализации этой процедуры состоит в том, чтобы рассматривать каждое имя x как пару (r, x) , где r есть номер отрезка, в котором находится x . Иначе говоря, считается, что пирамида состоит из пар $(r_1, x_1), (r_2, x_2), \dots, (r_m, x_m)$; сравнения между парами осуществляются лексикографически. Когда считывается имя, меньшее последнего имени в текущем отрезке, оно должно быть в следующем отрезке, и благодаря наличию номера отрезка это имя будет ниже всех имен пирамиды, которые входят в текущий отрезок.

Порождает ли этот выбор с замещением длинные отрезки? Ясно, что он делает это, по крайней мере, не хуже, чем очевидный метод, так как все отрезки (кроме, возможно, последнего) содержат не меньше m имен. На самом деле можно показать, что средняя длина отрезков, порождаемых выбором с замещением, равна $2m$, и это вполне можно считать улучшением по сравнению с очевидным методом, в котором средняя длина отрезков равна m . Конечно, в лучшем случае все

заканчивается только одним исходным отрезком, то есть отсортированной таблицей.

После того, как порождены исходные отрезки, возникает задача повторного распределения их по рабочим лентам и слияния их вместе до тех пор, пока в конце концов мы не получим окончательную отсортированную таблицу. Простейший метод состоит в том, чтобы распределить отрезки по лентам $1, 2, \dots, t$ равномерно и произвести их слияние на ленту $t + 1$. Получаемые в результате отрезки снова равномерно распределить по лентам $1, 2, \dots, t$ и снова произвести слияние, образуя более длинные отрезки на ленте $t + 1$. Процесс продолжается до тех пор, пока на ленте $t + 1$ не останется только один отрезок (отсортированная таблица).

Частичная сортировка

Ранее мы изучали проблему полного упорядочения множества имен, не имея а priori информации об абстрактном порядке имен. Имеется два очевидных уточнения этой проблемы. Вместо полного упорядочения требуется только определить k -е наибольшее имя (то есть k -е имя в порядке убывания) (выбор) или, вместо того чтобы начинать процесс, не располагая информацией о порядке, начинать с двух отсортированных подтаблиц (слияние). Мы рассматриваем обе проблемы частичной сортировки.

Частичная сортировка (выбор)

Как при данных именах $x_1, x_2, \dots, x_n$ можно найти k -е из наибольших в порядке убывания? Задача, очевидно, симметрична: отыскание $(n - k + 1)$ -го наибольшего (k -го наименьшего) имени можно осуществить, используя алгоритм отыскания k -го наибольшего, но меняя местами действия, предпринимаемые при результатах $<$ и $>$ сравнения имен. Таким образом, отыскание наибольшего имени ($k = 1$) эквивалентно отысканию наименьшего имени ($k = n$) ; отыскание второго наибольшего имени ($k = 2$) эквивалентно отысканию второго наименьшего ($k = n - 1$) и т.д.

Конечно, все перечисленные варианты задачи выбора можно решить, используя любой из методов полной сортировки имен и затем тривиально обращаясь к k -му наибольшему. Такой подход потребует порядка $n \log n$ сравнений имен независимо от значений k .

При использовании алгоритма сортировки для выбора наиболее подходящим будет один из алгоритмов, основанных на выборе: либо простая сортировка выбором (алгоритм 15.1) либо пирамидальная сортировка (алгоритм 15.3). В каждом случае мы можем остановиться после того, как выполнены первые k шагов. Для простой сортировки выбором это означает использование

$$(n-1) + (n-2) + \dots + (n-k) = kn - \frac{k(k+1)}{2}$$

сравнений имен, а для пирамидальной — использование $n + k \lg n$ сравнений имен. В обоих случаях мы получаем больше информации, чем нужно, потому что мы полностью определяем порядок k наибольших имен.

Частичная сортировка (слияние)

Вторым направлением исследования частичной сортировки является задача слияния двух отсортированных таблиц $x_1 \leq x_2 \leq \dots \leq x_n$ и $y_1 \leq y_2 \leq \dots \leq y_m$ в одну отсортированную таблицу $z_1 \leq z_2 \leq \dots \leq z_{n+m}$. Существует очевидный способ это сделать: таблицы, подлежащие слиянию, просматривать параллельно, выбирая на каждом шаге меньшее из двух имен и помещая его в окончательную таблицу. Этот процесс немного упрощается добавлением имен-сторожей $x_{n+1} = y_{m+1} = \infty$, как в алгоритме 15.5. В этом алгоритме i и j указывают, соответственно, на последние имена в двух входных таблицах, которые еще не были помещены в окончательную таблицу.

$$x_{n+1} \leftarrow y_{m+1} \leftarrow \infty$$

$$i \leftarrow j \leftarrow 1$$

$$\text{for } k = 1 \text{ to } n + m \text{ do } \left\langle \begin{array}{l} \text{if } x_i < y_j \text{ then } \left\langle \begin{array}{l} z_k \leftarrow x_i \\ i \leftarrow i + 1 \end{array} \right. \\ \\ \text{else } \left\langle \begin{array}{l} z_k \leftarrow y_j \\ j \leftarrow j + 1 \end{array} \right. \end{array} \right.$$

Алгоритм 15.5. Прямое слияние

Алгоритмы на графах

Поиск в глубину. Алгоритм Дейкстры нахождения кратчайшего пути. Алгоритм Флойда нахождения кратчайших путей между парами вершин. Программы.

Поиск в глубину

При решении многих задач, касающихся ориентированных графов, необходим эффективный метод систематического обхода вершин и дуг орграфов. Таким методом является метод поиск в глубину. Метод поиска в глубину является основой многих эффективных алгоритмов работы с графами. Предположим, что есть ориентированный граф G , в котором первоначально все вершины помечены меткой "*unvisited*". Поиск в глубину начинается с выбора начальной вершины v орграфа G , для этой вершины метка "*unvisited*" меняется на метку "*visited*". Затем для каждой вершины, смежной с вершиной v и не посещаемой раньше, рекурсивно применяется поиск в глубину. Когда все вершины, которых можно достичь из вершины v , будут рассмотрены, поиск заканчивается. Если некоторые вершины остались не посещенными, то выбирается одна из них и алгоритм повторяется. Этот процесс продолжается до тех пор, пока не будут обойдены все вершины орграфа G .

Метод получил свое название - поиск в глубину, поскольку поиск не посещенных вершин идет в направлении вглубь до тех пор, пока это возможно. Например, пусть x является последней посещенной нами вершиной. Выбираем очередную дугу (x, y) (ребро), выходящую из вершины x . Возможна следующая альтернатива: вершина y помечена меткой "*unvisited*"; вершина y помечена меткой "*visited*". Если вершина y уже посещалась, то отыскивается другая вершина, смежная с вершиной x ; иначе вершина y метится меткой "*visited*" и поиск начинается заново от вершины y . Пройдя все пути, которые начинаются в вершине y , возвращаемся в вершину x , то есть в ту вершину, из которой впервые была достигнута вершина y . Затем процесс повторяется, то есть продолжается выбор нерассмотренных дуг, исходящих из вершины x , и так до тех пор, пока не будут исчерпаны все

Алгоритм Дейкстры нахождения кратчайшего пути

Рассмотрим алгоритм нахождения путей в ориентированном графе. Пусть есть ориентированный граф $G(V, E)$, у которого все дуги имеют неотрицательные метки (веса дуг), а одна вершина определена как источник. Задача состоит в нахождении весов кратчайших путей от источника ко всем другим вершинам графа $G(V, E)$. Здесь длина пути определяется как сумма весов дуг, составляющих путь. Эта задача часто называется задачей нахождения кратчайшего пути с одним источником. Отметим, что мы будем говорить о длине пути даже тогда, когда она измеряется в других, не линейных, единицах измерения, например, во временных единицах или в денежном эквиваленте.

Можно представить орграф $G(V, E)$ в виде карты маршрутов рейсовых полетов из одного города в другой. Каждая вершина соответствует городу, а ребро (дуга) (v, w) - рейсовому маршруту из города v в город w . Вес дуги (v, w) - это время полета из города v в город w . В этом случае решение задачи нахождения кратчайшего пути с одним источником для ориентированного графа трактуется как минимальное время перелета между различными городами.

Для решения поставленной задачи будем использовать "жадный" алгоритм, который называют алгоритмом Дейкстры (Dijkstra). Алгоритм строит множество S вершин, для которых кратчайшие пути от источника уже известны. На каждом шаге к множеству S добавляется та из оставшихся вершин, расстояние до которой от источника меньше, чем для других оставшихся вершин. Если веса всех дуг неотрицательны, то можно быть уверенным, что кратчайший путь от источника к конкретной вершине проходит только через вершины множества S . Назовем такой путь особым. На каждом шаге алгоритма используется также массив M , в который записываются длины кратчайших особых путей для каждой вершины. Когда множество S будет содержать все вершины орграфа, то есть для всех вершин будут найдены особые пути, тогда массив M будет содержать длины кратчайших путей от

источника к каждой вершине.

Алгоритм Флойда нахождения кратчайших путей между парами вершин

Предположим, что мы имеем помеченный орграф, который содержит время полета по маршрутам, связывающим определенные города, и мы хотим построить таблицу, где приводилось бы минимальное время перелета из одного (произвольного) города в любой другой. В этом случае мы сталкиваемся с общей задачей нахождения кратчайших путей, то есть нахождения кратчайших путей между всеми парами вершин орграфа.

Формулировка задачи. Есть ориентированный граф $G(V, E)$, каждой дуге (ребру) (v, w) этого графа сопоставлен неотрицательный вес C_{vw} . Общая задача нахождения кратчайших путей заключается в нахождении для каждой упорядоченной пары вершин v, w любого пути от вершины v в вершину w , длина которого минимальна среди всех возможных путей от v к w .

Можно решать эту задачу, последовательно применяя алгоритм Дейкстры для каждой вершины, объявляемой в качестве источника. Но мы для решения поставленной задачи воспользуемся алгоритмом, предложенным Флойдом (R.W. Floyd). Пусть все вершины орграфа последовательно пронумерованы от 1 до n . Алгоритм Флойда использует матрицу $A(n \times n)$, в которой находятся длины кратчайших путей:

$$A_{ij} = C_{ij}, \text{ если } i \neq j ;$$

$$A_{ij} = 0, \text{ если } i = j ;$$

$$A_{ij} = \infty \text{ если отсутствует путь из вершины } i \text{ в вершину } j .$$

Над матрицей A выполняется n итераций. После k -й итерации A_{ij} содержит значение наименьшей длины пути из вершины i в вершину

j , причем путь не проходит через вершины с номерами большими k .

Вычисление на k -ой итерации выполняется по формуле:

$A_{ij}^k = \min(A_{ij}^{k-1}, A_{ik}^{k-1} + A_{kj}^{k-1})$ Верхний индекс k обозначает значение матрицы A после k -ой итерации.

Для вычисления A_{ij}^k проводится сравнение величины A_{ij}^{k-1} (то есть стоимость пути от вершины i к вершине j без участия вершины k или другой вершины с более высоким номером) с величиной $A_{ik}^{k-1} + A_{kj}^{k-1}$ (стоимость пути от вершины i к вершине k плюс стоимость пути от вершины k до вершины j). Если путь через вершину k дешевле, чем A_{ij}^{k-1} , то величина A_{ij}^k изменяется. Рассмотрим орграф:

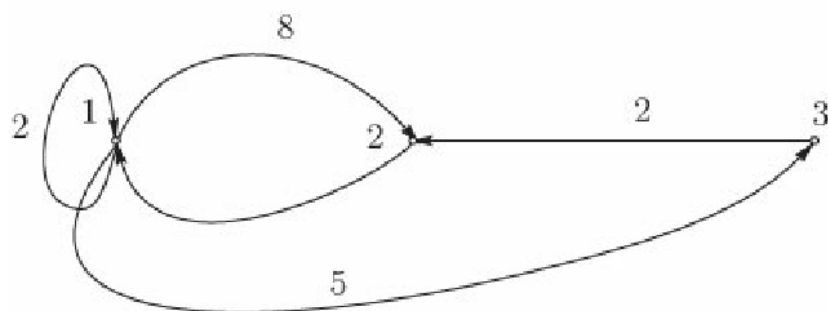


Рис. 16.1. Помеченный орграф

Матрица $A(3 \times 3)$ на нулевой итерации ($k = 0$)

$$\begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & \infty \\ \infty & 2 & 0 \end{bmatrix}$$

Матрица $A(3 \times 3)$ после первой итерации ($k = 1$)

$$\begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & 8 \\ \infty & 2 & 0 \end{bmatrix}$$

Матрица $A(3 \times 3)$ после второй итерации ($k = 2$)

$$\begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & 8 \\ 5 & 2 & 0 \end{bmatrix}$$

Матрица $A(3 \times 3)$ после третьей итерации ($k = 3$)

$$\begin{bmatrix} 0 & 7 & 5 \\ 3 & 0 & 8 \\ 5 & 2 & 0 \end{bmatrix}$$

Программы

Программа 1. Построение матрицы инцидентности.

//Построение матрицы инцидентности

//Программа реализована на языке программирования Turbo-C++

```
\begin{verbatim}
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>
#include <iostream.h>
```

```
struct elem
{
    int num; /* Номер вершины */
    int suns; /* Количество сыновей */
    char str[20]; /* Строка с номерами сыновей */
    elem *next; /* Указатель на следующую вершину */
} *head, *w1, *w2;
```

```
int Connected(int i, int j)
{
    int k;
    char *str1;
    w2 = head;
    if(i == j) return 0;
    for(k=1; k<i; k++)
        w2 = w2->next;
    if( strchr(w2->str, j) ) return 1;
    return 0;
}

void main()
{

    int tops;
    int i,j,k,l;
    char *str1;

    clrscr();
    printf("Введите количество вершин \n");
    scanf("%d", &tops);

    head = (elem *)malloc(sizeof(elem));
    head->num = 1;
    head->suns = 0;
    head->str[0] = '\0';
    head->next = NULL;

    w1 = head;

    for(i=2;i<=tops;i++)
    {
        w2 = (elem *)malloc(sizeof(elem));
        w2->num = i;
        w2->suns = 0;
        w2->str[0] = '\0';
        w2->next = NULL;
```

```
w1->next = w2;
    w1 = w2;
}

w1 = head;

for(i=1; i<=tops; i++)
{
// clrscr();
printf("Введите количество путей из вершины  %d\n", i);
scanf("%d", &k);

for(j=1; j<=k; j++)
{
    printf("Введите связь %d\n", j);
    scanf("%d", &l);
    if((l<=0) || (l > tops))
    {
        printf("Такой вершины нет, повторите попытку\n");
        l = 0;
        j--;
        continue;
    }
    w1->str[w1->suns++] = l;
    w1->str[w1->suns] = '\0';
    if(w1->suns == 49)
    {
        printf("Слишком много связей !");
        exit(1);
    }
}
    w1 = w1->next;
}
clrscr();
printf("\n\n Матрица инцидентности :\n");
for(i=1; i<=tops; i++)
{
    printf("\n %d ", i);
```

```

        for(j=1; j<=tops; j++)
        {
            printf("%d ", Connected(i, j));
        }
    }
    printf("\n\n Нажмите любую клавишу\ldots ");
    getch();
}

```

Программа 2.Поиск вершин, недостижимых из заданной вершины графа.

//Поиск вершин, недостижимых из заданной вершины графа.
 //Программа реализована на языке программирования Turbo-C++

```

\begin{verbatim}
#include <iostream.h>
#include <fstream.h>
//- -----
int n,s;
int c[20][20];
int r[20];
//- -----
int load();
int save();
int solve();
//- -----
int main(){
    load();
    solve();
    save();
    return 0;
}
//- -----
int load(){
    int i,j;
    ifstream in("input.txt");
    in>>n>s;
    s--;

```

```

    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            in"c[i][j];
    in.close();
    return 0;
}

int save(){
    int i;
    ofstream out("output.txt");
    for (i=0; i<n; i++)
        if (r[i]==0)
            out"i+1 "" ";
    out.close();
    return 0;
}
//-----
int solve(){
    int i,h,t;
    int q[400];
    for (i=0; i<n+1; i++) q[i]=0;
    r[s]=1;
    h=0;
    t=1;
    q[0]=s;
    while (h<t){
        for (i=0;i<n;i++)
            if ((c[q[h]][i]>0)&(r[i]==0)){
                q[t]=i;
                t++;
                r[i]=1;
            }
        h++;
    }
    return 0;
}

```

Программа 3. Поиск циклов в графе.

{>

Реализация на Turbo-Pascal.

Поиск циклов в графе

<}

{R-,I-,S-,Q-}

```
const MAXN      = 40;
      QUERYSIZE = 600;
```

```
type vert = record x: integer; s: array [1..MAXN] of integer; end;
```

```
var c  : array [1..MAXN,1..MAXN] of integer;
      n : integer;
```

```
wr : vert;
```

```
res : array [1..MAXN] of string;
      resv: integer;
      ss : string;
```

```
procedure load;
var i,j: integer;
begin
  assign(input,'input.txt');
  reset(input);
  read(n);
  for i:=1 to n do
    for j:=1 to n do
      read(c[i][j]);
  close(input);
end;
```

```
function saveway(i:integer):string;
var e:string;
begin
  str(i,e);
  if (wr.s[i]=-1) then
    saveway:=e+' '
```

```

    else
        saveway:=saveway(wr.s[i])+e+' ';
    end;

p>function findss(s: string): boolean;
var i      : integer;
    l1,l2,rs : string;
    i1,i2,i22 : integer;

begin
    findss:=false;
    l2:=copy(s,1,pos(' ',s)-1);
    i2:=length(l2);
    i22:=length(s);
    for i:=1 to resv do begin
        l1:=copy(res[i],1,pos(' ',res[i])-1);
        i1:=length(l1);
        rs:=copy(res[i],1,length(res[i])-i1)+res[i];
        if (length(res[i])+i2=i22+i1)and(pos(s,rs)>0)
        then begin
            findss:=true;
            exit;
        end;
    end;
end;

procedure solve;
var h,t,i,j: integer;
    q      : array [1..QUERYSIZE] of vert;
    e      : string;
begin
    resv:=0;
    fillchar(res,sizeof(res),0);

    for i:=1 to n do begin
        fillchar(q[i],sizeof(q[i]),0);
        q[i].x:=i;
        q[i].s[i]:=-1;
    end;
end;

```



```
t:=n+1;
h:=1;
while h<t do begin
  for i:=1 to n do
    if (c[q[h].x,i]>0) then begin
      if (q[h].s[i]=-1) then begin
        wr:=q[h];
        str(i,e);
        ss:=saveway(q[h].x)+e;
        if (not findss(ss)) then begin
          inc(resv);
          res[resv]:=ss;
        end;
      end;
    end;
  if (q[h].s[i]=0) then begin
    q[t]:=q[h];
    q[t].x:=i;
    q[t].s[i]:=q[h].x;
    inc(t);
  end;
end;
inc(h);
end;
```

```
close(output);
end;
```

```
procedure save;
var i: integer;
begin
  assign(output,'output.txt');
  rewrite(output);
  for i:=1 to resv do
    writeln(res[i]);
  close(output);
end;
```

```
begin
```

```
load;  
solve;  
save;  
end.
```

Калейдоскоп из комбинаторных алгоритмов

Автоматическое построение лабиринтов. Бинарное дерево. Задача о восьми ферзях. Сортировки. Задача о назначениях (задачи выбора). Ханойская башня.

Автоматическое построение лабиринтов

Тезей должен был найти выход из Критского лабиринта или погибнуть, убитый Минотавром. Но что поразительно: найти вход в лабиринт - задача не менее трудная.

Здесь не представляется возможным описать все мыслимые лабиринты, да это и не требуется. Мы займемся простыми лабиринтами, построенными на прямоугольнике $m \times n$, где m, n — положительные целые числа. Внутри и на границах прямоугольника поставлены стенки по ребрам покрывающей его единичной квадратной сетки. Чтобы построить из прямоугольника лабиринт, выйдем одну единичную стенку на одной из сторон прямоугольника (получится вход в лабиринт); выйдем одну единичную стенку на противоположной стороне (получится выход) и еще удалим какое-то число строго внутренних стенок. Говорят, что лабиринт имеет решение, если между входом и выходом внутри лабиринта есть путь в виде ломаной, не имеющей общих точек со стенками. Решение единственно, если любые два таких пути проходят через одни и те же внутренние ячейки сетки. На рис. 17.1 приведен пример лабиринта 5×5 .

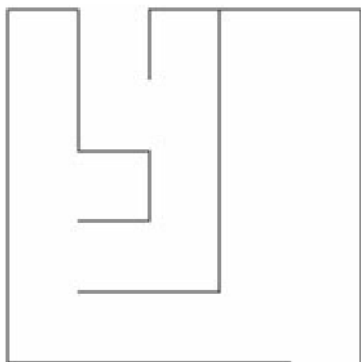


Рис. 17.1. Пример лабиринта

Один из возможных подходов к решению таков. Выбираем вход; затем, начав от него, добавляем по одной ячейке к главному пути-решению, пока он не достигнет выходной стороны. После этого удаляем некоторые внутренние стенки так, чтобы все клетки оказались соединенными с главным путем. Чтобы главный путь не получился прямым коридором, следует при его построении предусмотреть случайные повороты. Программа должна также следить за тем, чтобы при построении главного пути или при открытии боковых ячеек не нарушалась единственность решения. Наблюдательный читатель заметит, что определение единственности решения не годится в случае, когда путь заходит в боковой тупик и затем возвращается.

Программу можно написать почти на любом из процедурных языков.

Программа 1. Лабиринт.

{Программно задаются вход и выход. Нажимая на клавишу "Enter", перебираем всевозможные пути от входа до выхода в лабиринте. Выход из программы по клавише "Esc".
Алгоритм реализован на языке программирования Turbo-Pascal}

```
program Maze;
uses
  Graph, Crt;

var
  m,n: Integer;
  Matrix: array [1..100,1..100] of Boolean;
  Start,Finish: Integer;

procedure PrepareGraph;
var
  Driver,Mode: Integer;
begin
  Driver:=VGA;
  Mode:=VGAHi;
  InitGraph(Driver,Mode,'c:\borland\tp\bgi');
end;
```

```
procedure DisplayMaze(x1,y1,x2,y2: Integer);
var
  i,j: Integer;
  dx,dy: Real;
begin
  SetFillStyle(1,8);
  SetColor(15);
  dx:=(x2-x1)/m;
  dy:=(y2-y1)/n;
  for i:=1 to n do
    for j:=1 to m do
      if not Matrix[i,j] then
        Rectangle(Round(x1+(i-1)*dx),Round(y1+(j-1)*dy),
          Round(x1+i*dx),Round(y1+j*dy));
end;
```

```
function CreatesPath(i,j: Integer): Boolean;
var
  Result: Boolean;
  Count: Integer;
  ii,jj: Integer;
begin
  Count:=0;
  if (i>1) and Matrix[i-1,j] then Inc(Count);
  if (i<m) and Matrix[i+1,j] then Inc(Count);
  if (j>1) and Matrix[i,j-1] then Inc(Count);
  if (j<m) and Matrix[i,j+1] then Inc(Count);
  if Count>1 then Result:=true else Result:=false;
  CreatesPath:=Result;
end;
```

```
function DeadEnd(i,j: Integer): Boolean;
var
  Result: Boolean;
  Count: Integer;
begin
  Count:=0;
  if (i=2) or CreatesPath(i-1,j) then Inc(Count);
  if (i=m-1) or CreatesPath(i+1,j) then Inc(Count);
```

```
    if (j=2) or CreatesPath(i,j-1) then Inc(Count);  
    if (j=n-1) or CreatesPath(i,j+1) then Inc(Count);  
    if Count=4 then Result:=true else Result:=false;  
    DeadEnd:=Result;  
end;
```

```
function CreateMaze: Boolean;  
var  
    i,j: Integer;  
    di,dj: Integer;  
    Result: Boolean;  
begin  
    Randomize;  
    for i:=1 to n do  
        for j:=1 to m do Matrix[i,j]:=false;  
    Start:=Random(m-2)+2;  
    i:=Start;  
    j:=2;  
    Matrix[Start,1]:=true;  
    repeat  
        Matrix[i,j]:=true;  
        di:=0;  
        dj:=0;  
        while (di=0) and (dj=0) do begin  
            di:=1-Random(3);  
            if (i+di=1) or (i+di=m) then di:=0;  
            if di=0 then dj:=1-Random(3);  
            if j+dj=1 then dj:=0;  
            if CreatesPath(i+di,j+dj) then begin  
                di:=0;  
                dj:=0;  
            end;  
        end;  
        i:=i+di;  
        j:=j+dj;  
    until DeadEnd(i,j) or (j=n);  
    Finish:=i;  
    Matrix[Finish,n]:=true;  
    if j<n then Result:=false else Result:=true;
```

```
CreateMaze:=Result;  
end;  
  
begin  
  m:=6;  
  n:=6;  
  
  PrepareGraph;  
  repeat  
    ClearDevice;  
    repeat until CreateMaze;  
    DisplayMaze(120,40,520,440);  
    repeat until KeyPressed;  
  until ReadKey=#27;  
  CloseGraph;  
end.
```

Программа 2. Лабиринт.

{Лабиринт реализуется автоматически, без участия пользователя.

Алгоритм реализован на языке программирования Turbo-Pascal }

```
uses graph,crt;
```

```
var
```

```
mpos,npos,m,n,dex,x,y,t,gd,gm,i,k:integer;
```

```
begin
```

```
  randomize;
```

```
  writeln('Input labyrinth size (x and y)');
```

```
  readln(m,n);
```

```
  writeln('Input entrance&exit coordinates (mpos<m and npos<m)');
```

```
  readln(mpos,npos);
```

```
  initgraph(gd,gm,'c:\borland\tp\bgi');
```

```
  for i:=1 to m do
```

```
  begin
```

```
    for k:=1 to n do
```

```
    begin
```

```
      rectangle(90+10*i,90+10*k,90+10*i+10,90+10*k+10);
```

```
    end;
```

```
  end;
```

```
setfillstyle(1,0);
setcolor(0);
line(100+(mpos-1)*10+1,100,100+(mpos-1)*10+9,100);
line(100+(npos-1)*10+1,100+n*10,100+(npos-1)*10+9,100+n*10);
y:=n;
x:=npos;
readln;

while y>1 do
begin
delx:=random(m)-x+1;
if y=2 then delx:=mpos-x;
i:=91+x*10;
if i<90+(x+delx)*10 then
begin
while i<>90+(x+delx)*10 do
begin
i:=i+1;
line(i,91+y*10,i,99+y*10);
end;
end;

if i>91+(x+delx)*10 then
begin
while i<>91+(x+delx)*10 do
begin
i:=i-1;
line(i,91+y*10,i,99+y*10);
end;
end;

x:=x+delx;
line(91+10*x,90+y*10,99+10*x,90+y*10);

y:=y-1;

end;

readln;
```


end.

Бинарное дерево

Строится бинарное дерево. В узлы дерева засылаются целые положительные числа, выбранные случайным образом. После задания значений вершин на каждом уровне эти значения выводятся на экран, начиная с корневой. Пользователю предлагается сделать выбор номера уровня, в нем определяется максимальное значение и выводится на экран.

Программа 3. Поиск максимального элемента.

{ Алгоритм реализован на языке программирования Turbo-Pascal}

```
uses crt;
type sp=^tree;
tree=record
    val:integer;
    l:sp;
    r:sp;
end;
var
    t:sp;
    nh, max,h,i:integer;
procedure find(t:sp; h,nh:integer);
begin
    if t=nil then exit;
    if h=nh then
        begin
            if t^.val > max then max:=t^.val;
        end
    else
        begin
            find(t^.l,h+1,nh);
            find(t^.r,h+1,nh);
        end;
    end;
end;
procedure zadtree(var t:sp; h,nh:integer);
begin
```

```
    if h=5 then
    begin
        new(t);
        t^.l:=nil;
        t^.r:=nil;
        t^.val:=random(100);
    end
else
begin
    new(t);
    zadtree(t^.l, h+1,nh);
    zadtree(t^.r, h+1,nh);
    t^.val:=random(100);
end;
end;
procedure writetree(t:sp; h,nh:integer);
begin
    if t=nil then exit;
    if h=nh then
        begin
            write(t^.val, ' ');
        end
    else
        begin
            writetree(t^.l,h+1,nh);
            writetree(t^.r,h+1,nh);
        end;
    end;
end;
begin
    clrscr;
    randomize;
    t:=nil;
    zadtree(t,1,nh);
    for i:=1 to 5 do
        begin
            writetree(t,1,i);
            writeln;
        end;
    max:=0;
```

```
write('vvedite uroven ');  
readln(nh);  
find(t,1,nh);  
write('max= ',max);  
readln;  
end.
```

Задача о восьми ферзях

Условие задачи. Найти все такие расстановки восьми ферзей на шахматной доске, при которых ферзи не бьют друг друга.

Анализ задачи. Пусть A - множество искомых расстановок (конфигураций). Рассмотрим следующий подход к решению задачи. Будем искать множество конфигураций B со следующими свойствами:

1. $A \subset B$.
2. Имеется условие, позволяющее по элементу из B определить, принадлежит ли он A .
3. Имеется процедура, генерирующая все элементы из B .

С помощью процедуры из пункта 3 будем генерировать по очереди все элементы из B ; для элементов из B проверяем (см. пункт 2) принадлежит ли он A : в результате в силу 1 свойства будут порождены все элементы A .

Заметим теперь, что ферзи, которые не бьют друг друга, должны располагаться на разных горизонталях. Поэтому можно упорядочить ферзи и всегда ставить k -го ферзя на k -ю горизонталь. Тогда в качестве B можно взять множество конфигураций, в которых на каждой из первых N горизонталей стоит ровно по одному ферзю, причем никакие два ферзя не бьют друг друга.

Программа 4. Расстановка восьми ферзей на шахматной доске.

```
{ Программа выдает все комбинации ферзей, которые не бьют друг дру  
Алгоритм реализован на языке программирования Turbo-Pascal }  
program ferz;
```

```
uses crt;
const desksize=8;
type sizeint=1..desksize;
    unuses=set of sizeint;
    combines=array[shortint] of sizeint;
var num:byte;
    combine:combines;
    unuse:unuses;

function attack(combine:combines):boolean;
var i,j:byte;
    rightdiag,leftdiag:combines;
begin
    attack:=false;
    for i:=1 to desksize do
        begin
            leftdiag[i]:=i+combine[i];
            rightdiag[i]:=i-combine[i];
        end;
    for j:=1 to desksize do
        for i:=1 to desksize do
            begin
                if (i<>j) and ((leftdiag[i]=leftdiag[j])or(rightdiag[i]=rightdiag[j]))
            then
                begin
                    attack:=true;
                    exit;
                end;
            end;
        end;
    end;

procedure output(combine:combines);
var i,j:byte;
begin

    for i:=1 to desksize do
        for j:=1 to desksize do
            begin
                gotoxy(i,j);
```

```
    if(combine[i]=j) then write(#2) else write(' ');
  end;
  readln;
end;

procedure create(num:byte; unuse:unuses; combine:combinates);
var i:byte;
begin
  if num<=desksize then
    for i:=1 to desksize do
      begin
        if i in unuse then
          begin
            combine[num]:=i;
            create(num+1,unuse-[i],combine);
          end;
        end
      else if not attack(combine) then output(combine);

end;

begin
  textmode(c40);

  clrscr;
  unuse:=[1..desksize];
  create(1,unuse,combine);
end.
```

Сортировки

В лекциях 14 и 15 мы рассматривали более подробно различные способы сортировки. Здесь мы напоминаем некоторые из них и приводим пример программ.

Сортировка упорядочивает совокупность объектов в соответствии с заданным отношением порядка. Ключ сортировки - поле или группа полей элемента сортировки, которые используются при сравнении во

время сортировки. Сортирующая последовательность - схема упорядочивания. Например, можно взять последовательность символов алфавита, задающую способ упорядочения строк этого алфавита.

Способ сортировки: сортировка по возрастанию, сортировки по убыванию. Методы сортировки:

- метод прямого выбора;
- метод пузырька;
- метод по ключу;
- сортировки слиянием;
- сортировки Батчера.

Сортировка по возрастанию - это сортировка, при которой записи упорядочиваются по возрастанию значений ключевых полей.

Сортировка по убыванию - это сортировка, при которой записи упорядочиваются по убыванию значений ключевых полей.

Сортировка методом пузырька (пузырьковая сортировка) - способ сортировки, заключающийся в последовательной перестановке соседних элементов сортируемого массива.

Сортировка по ключу - это сортировка записей с упорядочением по значению указанного поля или группы полей.

Сортировка слиянием - это внешняя сортировка, при которой на первом этапе группы записей сортируются в оперативной памяти и записываются на несколько внешних носителей; на втором этапе упорядоченные группы сливаются с нескольких внешних носителей на один носитель данных. Носитель данных - материальный объект, предназначенный для хранения данных, или среда передачи данных.

Сортировка Батчера - это сортировка, внутренний алгоритм которой работает за время $O(N \cdot \log(N))$.

Программа 5. Сортировка массива по возрастанию методом пузырька.

//Данные, которые нужно отсортировать, берутся из файла "massiv.txt",

```
//результат записывается в массив int mas['K'] и выводится на экран
// Алгоритм реализован на Turbo C++.
#include <conio.h>
#include <stdio.h>
#define K 1000; //Размер массива

int mas['K'];
int n;
void puzirek()//функция сортирует массив по возрастанию методом пузы
{
int i,j,t;
    for(i=0;i<n;i++)
        for(j=1;j<n-i;j++)
            if(mas[j]<mas[j-1])
            {
                t=mas[j];
                mas[j]=mas[j-1];
                mas[j-1]=t;
            }
}

int main()
{
    clrscr();
    FILE *filePointer=fopen("massiv.txt","r");
    int i=0;
    while (!feof(filePointer))
    {
        fscanf(filePointer,"%d",&mas[i]);
        i++;
    }
    n=i;
    puzirek();
    for(i=0;i<n;i++)
        printf("%d ",mas[i]);
    //scanf("%d",&n);
    getch();
    return 0;
}
```

Программа 6. Пузырьковая сортировка и сортировка методом прямого выбора.

{Сортировка. Алгоритм реализован на языке программирования Turbo-
uses crt;

var

M, N : array[0..10] of integer;

i:integer;

procedure Input;

begin

for i := 0 to 10 do

begin

writeln('Число');

readln(M[i]); {Ввод массива}

end;

end;

Procedure Sort1; {Пузырьковый метод сортировки}

var

q,i,x:integer;

begin

for i:=10 downto 0 do

begin

for q:=0 to 10 do

if M[q]<M[q+1] then

begin

x:=M[q];

M[q]:=M[q+1];

M[q+1]:=x

end;

end;

end;

procedure Sort2; {Метод прямого выбора}

var

i,j,k,x:integer;

begin


```
for i:=0 to 9 do
begin
  k:=i;
  x:=M[i];
  for j:=i+1 to 10 do
  if M[j] > x then begin k:=j; x:=M[k];
  end;
  M[k]:= M[i];
  M[i]:=x;
end;
end;

{-----}
begin
  clrscr;
  input; {Ввод исходного массива}
  writeln('Исходный массив');
  for i:=0 to 10 do write(M[i], ' '); {Вывод исходного массива}
  writeln;
  Sort1;{Сортировка массива методом пузырька}
  writeln ('Сортированный массив');
  for i:=0 to 10 do write(M[i], ' '); {Вывод отсортированного массива}
  input; {Ввод исходного массива}
  writeln('Исходный массив');
  for i:=0 to 10 do write(M[i], ' '); {Вывод исходного массива}
  writeln;

  sort2;
  writeln ('Сортированный массив методом прямого выбора');
  for i:=0 to 10 do write(M[i], ' '); {Вывод отсортированного массива}
  readln;
end.
```

Программа 7. Сестры.

```
//Две строки матрицы назовем сестрами, если совпадают
//множества чисел, встречающихся в этих строках. Программа
//определяет всех сестер матрицы, если они есть,
//и выводит номера строк. Алгоритм реализован на Turbo C++.
```

```
    #include <graphics.h>
    #include <stdlib.h>
    #include <string.h>
    #include <stdio.h>
    #include <conio.h>
    #include <math.h>
    #include <dos.h>
    #include <values.h>
    #include <iostream.h>

    const n=4, //кол-во строк
          m=4; //столбцов

    int m1[n][m];
    //исходный массив
    struct mas{int i, i1;};
    //i-индекс сравниваемой строки с i1 строкой
    mas a[n*2];
    //массив типа mas, где будут лежать сестры, пример) a[1].i и a[1].i1 -
    сестры

    void main()
    {clrscr();
    int i, j;
    randomize();
    for( i=0; i<n; i++)
    for( j=0; j<m; j++)
    m1[i][j]=random(2);
    //случайным образом в массив заносим цифры

    for(i=0; i<n; i++)
    {printf("\n %d", i);
    for(int j=0; j<m; j++)
    printf(" %d", m1[i][j]);
    //распечатываем этот массив
    }
    int min,
    p;
    //индекс минимального элемента после s-го элемента i-ой строки
```

```

//сортировка строк массива по возрастанию
for(i=0;i<n;i++)//i-сортировка i-ой строки
{
    for(int s=0;s<m-1;s++)
    {min=m1[i][s+1];
    for(int j=s;j<m;j++)
        if(m1[i][j]<=min){min=m1[i][j];p=j;}
//запоминаем минимальный элемент в ряде после s-го элемента
    if(m1[i][s]>=min)
        {m1[i][p]=m1[i][s];m1[i][s]=min;}
//меняем местами s-й и p-й элемент,если s-й>p-го(минимального)
    }

}

printf("\n");
for(i=0;i<n;i++)
{printf("\n %d ",i);
for(int j=0;j<m;j++)
    printf(" %d",m1[i][j]);
//выводим отсортированный массив
}
int s=0 //сколько элементов в i-й строке совпадают с
        эл-ми i1 строки, k=0;
//сколько строк совпали
int i1;
for(i=0;i<n-1;i++)          //верхняя строка i
for( i1=i+1;i1<n;i1++)      //нижняя строка i1
{s=0;
for(int j=0;j<m;j++)
    //сравнение идет по j-му столбцу
//      !      !
    if(m1[i][j]==m1[i1][j])s++;    //если соответствующие элементы в
//i-й и i1-й строки совпадают, то кол-во совпавших увеличивается на 1
    if(s==m){a[k].i=i;a[k].i1=i1;k++;}
    //если все элементы i-й и i1-й строки совпали, то они сестры
}

printf("\nСестры :");

```

```
for(i=0;i<k;i++)
printf("\n %d и %d",a[i].i,a[i].i1);
//распечатываем a[i].i-ю и a[i].i1-ю сестру
getch();
}
```

Программа 8. Поиск узоров из простых чисел.

//Построить матрицу A(15 X 15)таким образом: A(8,8)=1, затем
//по спирали против часовой стрелки,
//увеличивая значение очередного элемента на единицу
//и выделяя все простые числа красным цветом, заполнить матрицу
//Алгоритм реализован на Turbo C++.

```
#include <stdio.h>
#include <conio.h>

void main(void)
{
clrscr();
int mas[15][15];
int n=1,x=6,y=6,k=1;
int i,j;
while(1){
mas[x][y]=k++;
switch(n){
case 1: x++;break;
case 2: y--;break;
case 3: x--;break;
case 4: y++;break;
}
if(x==15) break;

if(x==y && x<6) n=4;
else if(x+y==12 && x<6) n=1;
else if(x+y==12 && x>6) n=3;
else if(x==y+1 && x>6) n=2;

}
```

```

for(i=0;i<15;i++)
{
    for(j=0;j<15;j++)
    {
        textcolor(12);
        if(mas[j][i]>2)
            for(k=2;k<mas[j][i];k++)
                if(mas[j][i]%k==0) textcolor(15);
        printf("%3d ",mas[j][i]);
    }
    printf("\n");
}

getch();
}

```

Программа 9. Сортировка строк матрицы.

//Сортировка строк матрицы. В каждой строке подсчитывается сумма
 //простых чисел. Полученный вектор упорядочивается по возрастанию.
 //Строки матрицы переставляются по новому вектору.
 //Алгоритм реализован на Turbo C++.

```

#include<stdio.h>
#include<conio.h>

```

```

#define n 5

```

```

struct summa
{
    int value;
    int idx;
} sum,massum[n],a;

```

```

void main(void){
    clrscr();
    int mas1[n][n],mas[n][n]={ {1,1,1,1,1},
                                {3,16,11,6,4},
                                {8,10,15,23,1},

```

```
{3,8,10,15,3},  
{7,3,20,15,10} };
```

```
int i,j,k,flag;
```

```
for(i=0;i<n;i++){  
    sum.value=0;  
    for(j=0;j<n;j++){  
        flag=0;  
        if(mas[i][j]>2)  
            for(k=2;k<mas[i][j];k++)  
                if(mas[i][j]%k==0) flag=1;  
        if(flag==0) sum.value=sum.value+mas[i][j];  
    }  
    sum.idx=i;  
    massum[i]=sum;  
}
```

```
for(i=0;i<n-1;i++){  
    for(j=0;j<n-1-i;j++){  
        if (massum[j].value>massum[j+1].value){  
            a=massum[j];  
            massum[j]=massum[j+1];  
            massum[j+1]=a;  
        }  
    }  
}
```

```
for(i=0;i<n;i++){  
    for(j=0;j<n;j++){  
        mas1[i][j]=mas[massum[i].idx][j];
```

```
for(i=0;i<n;i++){  
    for(j=0;j<n;j++){  
        printf("%3d ",mas[i][j]);  
        printf("\n");  
    }  
}
```

```
printf("\n\n\n");
```

```

for(i=0;i<n;i++){
    for(j=0;j<n;j++){
        printf("%3d ",mas1[i][j]);
        printf("\n");
    }
    getch();
}

```

Задача о назначениях (задачи выбора)

Эта задача состоит в следующем. Пусть имеется n работ и n кандидатов для выполнения этих работ. Назначение кандидата i на работу j связано с затратами c_{ij} ($i, j = 1, 2, \dots, n$). Требуется найти назначение кандидатов на все работы, дающее минимальные суммарные затраты; при этом каждого кандидата можно назначить только на одну работу и каждая работа может быть занята только одним кандидатом.

Иначе говоря, решение этой задачи представляет собой перестановку $(p_1, p_2, \dots, p_n)$ чисел $(1, 2, \dots, n)$; каждое из производимых назначений описывается соответствием $i \rightarrow p_i$ ($i = 1, \dots, n$). Указанные условия единственности при этом автоматически выполняются, и нашей целью является минимизация суммы

$$\sum_{i=1}^n c_{ip_i} \quad (17.1)$$

по всем перестановкам $(p_1, p_2, \dots, p_n)$.

Перед нами типичная экстремальная комбинаторная задача. Ее решение путем прямого перебора, то есть вычисления значений функции 17.1 на всех перестановках и сравнения, практически невозможно при сколь угодно больших n , поскольку число перестановок равно $n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot (n-1) \cdot n$. Попытаемся свести дело к линейному программированию.

Конечное множество, на котором задана целевая функция 17.1,

представляет собой множество всех перестановок чисел $(1, 2, \dots, n)$. Как известно, каждая такая перестановка может быть описана точкой в n^2 -мерном евклидовом пространстве; эту точку удобнее всего представить в виде $n \times n$ -матрицы $X = \|x_{ij}\|$. Элементы x_{ij} интерпретировать следующим образом:

$x_{ij} = 1$, если i -й кандидат назначается на j -ю работу,

$x_{ij} = 0$, в противном случае.

Элементы матрицы должны быть подчинены двум условиям:

$$\sum_{j=1}^n x_{ij} = 1, \quad i = 1, 2, \dots, n, \quad (17.3)$$

$$\sum_{i=1}^n x_{ij} = 1, \quad j = 1, 2, \dots, n. \quad (17.4)$$

Условия 17.3 и 17.4 говорят о том, что в каждой строке и в каждом столбце матрицы X имеется ровно по одной единице. Говоря неформально, условие 17.3 означает, что каждый кандидат может быть назначен только на одну работу, а условие 17.4 — что каждая работа предназначена только для одного кандидата. (Матрицу перестановок можно получить из единичной матрицы путем некоторой перестановки ее строк.) Теперь задача заключается в нахождении чисел x_{ij} , удовлетворяющих условиям 17.2, 17.3, 17.4 и минимизирующих суммарные затраты 17.1, которые теперь можно переписать в виде

$$\sum_{i=1}^n \sum_{j=1}^n c_{ij} x_{ij}. \quad (17.5)$$

Казалось бы, что к полученной задаче методы линейного программирования непосредственно применить нельзя, ибо в силу условий 17.2 она формально является целочисленной. Заменим условие 17.2 на условие неотрицательности переменных

$$x_{ij} > 0. \quad (17.6)$$

Тем самым мы получаем обычную задачу линейного программирования. В нашем случае требование целочисленности 17.2 будет выполняться автоматически.

Программа 10. Назначение на работу.

```

program one; {Назначение на работу.
Рассматривается случай: 10 работ и 10 желающих.
реализовано на Turbo-Pascal}
uses crt;
const n=10;
var C : array [1..n,1..n] of integer;
    T : array [1..n] of integer;
    M : array [1..n,1..4] of integer;
    Sum,tmj,z,min,i,j,tmp:integer;

begin
  clrscr;
  randomize;
  write('work - ');
  for i:=1 to n do write(i:2, ' ');
  for i:=1 to n do begin
    writeln;
    write(i:2, ' man ');
    for j:=1 to n do begin
      C[i,j]:=random(100);
      {if M[i,j]>max then max:=M[i,j];}
      {if C[i,j]<min then begin M[1]:=C[i,j]; M[2]:=i; M[3]:=j; end; }
      write(C[i,j]:2, ' ');
    end;
    writeln;
  end;

  for j:=1 to n do T[j]:=0;
  Sum:=0;
  for i:=1 to n do begin

```

```

writeln;
write(i:2, ' man ');
min:=100;
for j:=1 to n do begin
  if (C[i,j]<min) and (T[j]=0) then begin min:=C[i,j]; M[i,1]:=i;
  M[i,2]:=j; M[i,3]:=C[i,j]; tmj:=j;

end;

write(C[i,j]:2, ' ');
end;
T[tmj]:=1;
{M[i,3]:=min;}
Sum:=Sum+M[i,3];
write('=',M[i,3]:2, ' man=',M[i,1], ' job=',M[i,2]);
end;
writeln;
{for i:=1 to n do begin
  for j:=1 to n do begin
    if (i<>j) and (M[i,2]=M[j,2]) then begin
      M[j,3]:=C[j,1];
      for z:=1 to n do begin
        if (M[j,3]>C[j,z]) and (z<>M[j,2]) then begin M[j,3]:=C[j,z];
M[j,2]:=z; end;
        end;
      end;
    end;
  }
  write('sum=',Sum);
  readln;
end.

```

Программа 11.Назначение на работу.

```
/*
```

Назначение на работу.

Рассматривается случай: 6 работ и 6 желающих.

*/

//Назначение на работу. Реализовано на Turbo C++.

#include <stdio.h>

#include <iostream.h>

#include <stdlib.h>

#include <conio.h>

#define k 6

int Sum,tmj,i,j,zj,min,tmp,min2,tmj2,p,q,ki;

int M[k][4], C[k][k], T[k][2], Temper[k][2];

char a;

/*struct myst

{int cel;

float rac;

};

myst ctpyk[k];*/

main()

{

Sum=0;

min=100;

for(i=1;i<k;i++)

{ T[i][1]=0;

printf("\n");

for(j=1;j<k;j++)

{C[i][j]=rand()/1000 +1;

// printf(" %d ", C[i][j]);

}

}

for(i=1;i<k;i++)

{

min=100;

printf("\n");

for(j=1;j<k;j++)

{

if(C[i][j]<min/* && T[j][1]==0*/)

{

```

        if(T[j][1]==0)
        {

min=C[i][j]; //m[i][1] - 4el, m[i][2] -job, m[i][3]
- stoimost.
        M[i][1]=i;
        M[i][2]=j;
        M[i][3]=C[i][j];
        tmj=j;
    }
    /* else
    {
        if(C[i][j]<C[T[j][2]][j])
        {
            ki=T[j][2];
            T[j][2]=0;
            // T[j][1]=0;
            min=C[i][j];
            M[i][1]=i;
            M[i][2]=j;
            M[i][3]=C[i][j];
            tmj=j;
            for(zj=1;zj<k;zj++)
            {
                min2=100;
                if(C[ki][zj]<min2 && zj!=tmj &&
T[zj][1]==0)
                {
                    min2=C[ki][zj];
                    tmj2=zj;
                    M[ki][1]=ki;
                    M[ki][2]=zj;
                    M[ki][3]=C[ki][zj];
                }

            }

            T[tmj2][2]=ki;
            T[tmj2][1]=min2;
        }
    }

```

```

*/

}

printf(" %d ", C[i][j]);
}

T[tmj][2]=i;
T[tmj][1]=min;
//na4alo mega funkcii
/* if(C[i][j]<min && T[j][1]!=0)
{
    for(p=1;pk;p++)
    {
        if(C[T[tmj][2]][p]

    }

}
*/
//konec.

Sum=Sum+M[i][3];
printf("    $= %d, man= %d, job= %d ",M[i][3],M[i][1],M[i][2]);

}

/* for(i=0;i<k;i++)
{ctpyk[i].cel=rand();
ctpyk[i].rac=rand()/1000;
printf("%d %f\n", ctpyk[i].cel, ctpyk[i].rac);}

*/
scanf("%d",a);
return 0;
}

```

Ханойская башня

В лекции 11 дана реализация алгоритма "Ханойская башня". Здесь используется другой подход. В программе реализован пользовательский интерфейс с развитым эргономическим компонентом. Пользователю предоставляется возможность самому решить поставленную задачу. В программе использована работа с видеостраницами.

Постановка задачи.

На стержне в исходном порядке находится дисков, уменьшающихся по размеру снизу вверх. Диски должны быть переставлены на стержень в исходном порядке при использовании в случае необходимости промежуточного стержня для временного хранения дисков. В процессе перестановки дисков обязательно должны соблюдаться правила: одновременно может быть переставлен только один самый верхний диск (с одного из стержней на другой); ни в какой момент времени диск не может находиться на другом диске меньшего размера.

Программа 12. Ханойская башня.

{Программа реализована с помощью абстрактного типа данных – стек для произвольного числа дисков. Число колец задается константой maxc. Программа написана на языке программирования Turbo-Pascal}

```
program Tower;  
uses Crt, Graph;
```

```
const maxc = 4; {Максимальное число колец на одной башне}
```

```
type TTower = record  
    num: byte;  
    sizes: array[1..maxc] of byte;  
    up: byte;  
end;
```

```
var Towers: array[1..3] of TTower;  
    VisVP, ActVP: byte; {видимая и активная видеостраницы}  
    ActTower: byte;  
    Message: String;
```

```
Win: boolean;
font1: integer;

function CheckEnd: boolean;
var res: boolean;
    i: byte;
begin
    res:=False;
    if (Towers[2].num=maxc) or (Towers[3].num=maxc) then res:=true;
    CheckEnd:=res;
end;

procedure BeginDraw;
begin
    SetActivePage(ActVP);
end;

procedure EndDraw;
begin

VisVP:=ActVP;
    SetVisualPage(VisVP);
    if VisVP=1 then ActVP:=0 else ActVP:=1;
end;

procedure Init;
var grDr, grM: integer;

ErrCode: integer;
    i: integer;
begin
    grDr:=VGA;
    grM:=VGAMed;
    InitGraph(grDr, grM, 'c:\borland\tp\bgi');
    ErrCode:=GraphResult;
    if ErrCode <> grOk then
    begin
        Writeln('Graphics error:', GraphErrorMsg(ErrCode));
        Halt;
```

```
end;

Towers[1].num:=maxc;
Towers[1].up:=0;
for i:=0 to maxc-1 do
  Towers[1].sizes[i+1]:=maxc-i;

Towers[2].num:=0;
Towers[2].up:=0;
for i:=0 to maxc-1 do
  Towers[2].sizes[i+1]:=0;

p>Towers[3].num:=0;
Towers[3].up:=0;
for i:=0 to maxc-1 do
  Towers[2].sizes[i+1]:=0;

ActTower:=1;
VisVP:=0; ActVP:=1;
SetVisualPage(VisVP);
SetActivePage(ActVP);
Message:="";
Win:=False;
end;

procedure Close;
begin
  closegraph;
end;

procedure DrawTower(x, y: integer; n: integer);
var i: integer;
begin
  if n=ActTower then
    SetColor(yellow);
    Line(x, y, x, y+15+maxc*15);
    for i:=1 to Towers[n].num do
      begin
        Rectangle(x-10*Towers[n].sizes[i], y+15+15*(maxc-i+1),
```



```
x+10*Towers[n].sizes[i], y+15+15*(maxc-i))
end;

if Towers[n].up<>0 then
begin
  Rectangle(x-10*Towers[n].up, y-15, x+10*Towers[n].up, y-30);
end;

SetColor(White);
end;

procedure DrawInfo;
begin
  OutTextXY(50, 20, 'Ханойская башня. ');
  OutTextXY(80, 40, 'Работа с программой: стрелки влево-вправо - выбо
башни');
  OutTextXY(130, 60, 'текущая башня выделяется желтым цветом');
  OutTextXY(60, 80, 'стрелка вверх - поднять кольцо, стрелка вниз -
положить кольцо');
  OutTextXY(80, 100, '(две последние операции выполняются для активн
башни)');

end;

procedure Draw;
begin
  BeginDraw;
  ClearDevice;
  OutTextXY(180, 140, Message); Message:="";
  DrawTower(150, 200, 1);
  DrawTower(300, 200, 2);
  DrawTower(450, 200, 3);

if win then
begin
  SetTextStyle(GothicFont, HorizDir, 8);
  SetColor(Red);
  Outtextxy(70, 0, 'Congratulations');
  Outtextxy(160, 70, 'You win');
```

```
    SetTextStyle(DefaultFont, HorizDir, 1);
    SetColor(White);
    OutTextXY(250, 330, 'Press any key');
end
else
    DrawInfo;
EndDraw;
end;
```

```
procedure MainCycle;
var ch: char;
    ex: boolean;
    up: byte;
begin
    ex:=False;
    repeat
        if KeyPressed then
            begin
                ch:=ReadKey;
                case ch of
                    #27: begin
                        Ex:=True;
                        end;
                    #77: begin
                        up:=Towers[ActTower].up;
                        Towers[ActTower].up:=0;
                        inc(ActTower);
                        if ActTower>3 then ActTower:=1;
                        Towers[ActTower].up:=up;
                        end;
                    #75: begin
                        up:=Towers[ActTower].up;
                        Towers[ActTower].up:=0;
                        dec(ActTower);
                        if ActTower<1 then ActTower:=3;
                        Towers[ActTower].up:=up;
                        end;
                    #80: begin    {вниз}
                        if Towers[ActTower].up<>0 then
```

```
begin
  if Towers[ActTower].num=0 then
    begin
      Towers[ActTower].num:=Towers[ActTower].num+1;

Towers[ActTower].sizes[Towers[ActTower].num]:=Towers[ActTower].up;
      Towers[ActTower].up:=0;
    end
  else
    begin
      if
Towers[ActTower].sizes[Towers[ActTower].num]>Towers[ActTower].up then
        begin
          Towers[ActTower].num:=Towers[ActTower].num+1;

Towers[ActTower].sizes[Towers[ActTower].num]:=Towers[ActTower].up;
          Towers[ActTower].up:=0;
        end
      else
        Message:='Это кольцо сюда опускать нельзя';
      end;
    end;
  end;
end;
#72: begin      {вверх}
  if Towers[ActTower].num<>0 then
    begin
      Towers[ActTower].num:=Towers[ActTower].num-1;

Towers[ActTower].up:=Towers[ActTower].sizes[Towers[ActTower].num+1];
      Towers[ActTower].sizes[Towers[ActTower].num+1]:=0;
    end;
  end;
end;
if CheckEnd then
  begin
    Win:=True;
    ex:=True;
  end;
Draw;
```

```
    end;  
    until ex;  
end;  
  
begin  
    Init;  
    Draw;  
    MainCycle;  
    if win then repeat until keypressed;  
    Close;  
end.
```

Содержание

Титульная страница	2
Выходные данные	3
Лекция 1. Комбинаторные вычисления	4
Лекция 2. Целые и последовательности (последовательное распределение)	14
Лекция 3. Последовательности (связанное распределение, стеки и очереди)	22
Лекция 4. Последовательности (деревья)	32
Лекция 5. Комбинаторика разбиений	42
Лекция 6. Последовательности (множества и мультимножества)	53
Лекция 7. Рекуррентные соотношения	64
Лекция 8. Алгоритмы рекуррентных соотношений	75
Лекция 9. Комбинаторика и ряды	86
Лекция 10. Производящие функции и рекуррентные соотношения	94
Лекция 11. Алгоритмы на абстрактных структурах данных	102
Лекция 12. Что такое граф? Определения и примеры	117
Лекция 13. Поиск	130
Лекция 14. Сортировка (часть 1)	147
Лекция 15. Сортировка (часть 2)	158
Лекция 16. Алгоритмы на графах	170
Лекция 17. Калейдоскоп из комбинаторных алгоритмов	183