И. В. Красиков, И. Е. Красикова

Алгоритмы

ПРОСТО

КАК

МОСКВА 2007

УДК 004.021
ББК 22.18
К 78

Выпускающий редактор В.В. Александров

К 78 **Красиков И.В.**

Алгоритмы. Просто как дважды два / И. В. Красиков, И. Е. Красикова. — М. : Эксмо, 2007. — 256 с. — (Просто как дважды два).

ISBN 978-5-699-21047-3

Программирование невозможно без знания языков программирования, но не менее невозможно оно без знания алгоритмов. Эта книга познакомит вас со многими алгоритмами для решения часто встречающихся в программистской практике задач. В книге собраны самые разные алгоритмы - от сортировки и работы с графами до численных методов и работы с календарем; имеется много примеров использования алгоритмов для решения конкретных задач, а также реализация описанных алгоритмов на языке программирования C++.

УДК 004.021
ББК 22.18

ISBN 978-5-699-21047-3

Содержание

Посвящается первому человеку,
который не только сумел сделать что-то,
но и пояснил другим, как это выполнить,
став первым автором алгоритма

Благодарности

Самая большая благодарность — авторам всех описанных в книге алгоритмов, без которых, само собой, этой книги просто не могло бы быть.

И. и И. Красиковы

Введение

Данную книгу можно рассматривать как краткий справочник по основным, наиболее распространенным компьютерным алгоритмам. Она рассчитана в первую очередь на начинающих программистов, но может также служить справочником для более опытных специалистов.

Алгоритм описывается с помощью слов, псевдокода и, как правило, реализации алгоритма (или примера его использования) на языке программирования C++.

В псевдокоде для выделения блоков используются как отступы, так и явное указание блоков **begin-end**. Инструкции циклов и условных операторов трактуются так же, как и в языке Pascal, однако переменная цикла при выходе из него остается действительной, а ее значение на единицу превышает верхнюю границу цикла. Функциональность оператора «=» зависит от контекста — в условиях циклов и конструкции **if** он означает равенство, в остальных частях псевдокода — присваивание. Все переменные, если не оговорено иное, являются локальными по отношению к данной процедуре.

Доступ к элементам массива осуществляется путем указания имени массива, за которым в квадратных скобках следует индекс. Например, $A[i]$ — это обозначение i-го элемента массива A. С помощью обозначения «..» указывается интервал значений, которые принимает индекс массива. Таким образом, обозначение $A[1..j]$ свидетельствует о том, что данное подмножество массива A состоит из j элементов $A[1],A[2],...,A[j]$. Однако в ряде случаев (особенно при использовании двойных индексов) для экономии места может применяться эквивалентное обозначение с нижним индексом: A_j, A_{jk}.

Сложные данные представляются в виде объектов, содержащих поля. Доступ к определенному полю осуществляется с помощью имени поля, после которого в квадратных скобках указывается имя объекта. Например, узел в списке имеет поле *next*, так что значение поля *next* узла x записывается как $next[x]$. Несмотря на то что квадратные скобки, как сказано выше, используются и для индексирования элементов массива, их интерпретация всегда будет понятна из контекста.

Переменная, которая используется в качестве имени массива или объекта, трактуется как указатель на данные, представляющие этот массив или объект. Для всех полей f объекта x присваивание $y = x$

приводит к тому, что $f[x] = f[y]$, а переменные x и y указывают на один и тот же объект.

Иногда переменная вообще не ссылается ни на какой объект. В таком случае она имеет значение 0.

Параметры передаются в процедуру по значению: в вызывающей процедуре создается своя собственная копия параметров, и если в вызванной процедуре какому-то параметру присваивается значение, то в вызывающей процедуре *не* происходит никаких изменений. Если передаются объекты, то происходит копирование указателя на данные, представляющие этот объект, но поля объекта не копируются.

Логические операторы **and** и **or** вычисляются сокращенно, как в языке программирования C/C++. Это означает, что при вычислении выражения «x **and** y» сначала определяется значение выражения x. Если оно ложно, то все выражение не может быть истинным, и значение выражения y не вычисляется. Если же выражение x истинно, то для определения значения всего выражения необходимо вычислить выражение y. Аналогично в выражении «x **or** y» величина y вычисляется только в том случае, если выражение x ложно. Укороченные операторы позволяют составлять такие логические выражения, как «$x \neq 0$ **and** $f[x] = y$», не беспокоясь о том, что произойдет при попытке вычислить выражение $f[x]$, если $x = 0$.

«Изюминкой» книги является реализация практически всех приведенных в ней алгоритмов на C++. Относиться к реализации алгоритмов на C++ в этой книге можно двояко — с одной стороны, вы можете применять готовые реализации алгоритмов в своих программах (хотя масса их более эффективно реализована в библиотеках C++), с другой — вы можете использовать эти реализации как учебный материал при изучении языка C++.

Следует сделать несколько замечаний по поводу реализации алгоритмов на C++.

При реализации алгоритмов не преследовалась цель получить максимально эффективный или элегантный код. Основной упор делался на корректность и понятность реализации.

Там, где это представлялось разумным, использовалось обобщенное программирование (в первую очередь, это алгоритмы сортировки, поиска и т.п.), однако в книге имеются и реализации с использованием C++ в качестве «улучшенного C», там, где обобщенность не принципиальна или решается конкретный экземпляр задачи.

Использование реализаций алгоритмов предполагает знание по крайней мере основ языка программирования C++. Многие тривиа-

альные с точки зрения языка вещи опущены (в качестве примера можно привести реализации алгоритмов сортировки, где без пояснений используются такие вещи, как итераторы или компараторы. Там же приведены только версии реализаций алгоритмов с компараторами, поскольку написать перегрузку шаблонной функции сортировки с использованием компаратора `less<type>` для знающего C++ программиста — задача тривиальная[1]). В качестве справочного пособия, облегчающего понимание реализаций алгоритмов на C++, можно посоветовать книгу [14].

Поскольку главной задачей является реализация алгоритма, обработка ошибок (там, где она имеется) выполняется простейшим образом, с помощью языкового средства **assert** — заинтересованный читатель должен применять вместо этого весьма грубого средства свою обработку ошибок, например, с использованием исключений. В ряде реализаций самоочевидные проверки опущены. Говоря об исключениях, следует также отметить, что вопросы безопасности исключений при реализации отодвигались на второй план.

Для компиляции программ использовались компиляторы Open Watcom 1.4 и Visual C++, однако в исходных текстах реализаций алгоритмов нет никаких особенностей, которые бы препятствовали их компиляции другими компиляторами, поддерживающими стандарт C++.

В тексте, отмеченном данной пиктограммой, рассматривается применение описанного алгоритма для решения конкретного экземпляра задачи.

Данная пиктограмма указывает на исходные тексты на C++, реализующие алгоритм так, чтобы его можно было применить для решения различных экземпляров задачи.

Этой пиктограммой отмечаются дополнительные сведения по рассматриваемой теме.

[1] На всякий случай приведем двухстрочное решение для функции пузырьковой сортировки:

```
template<typename Iter> void BubleSort(Iter b, Iter e) {
  BubleSort(b,e,less<iterator_traits<Iter>::value_type>());}
```

Глава 1

Структуры данных

В этой главе вы познакомитесь с основными структурами данных, которые играют в алгоритмах крайне важную роль, так как использование той или иной структуры данных зачастую определяет временны*е* характеристики алгоритма и его требования к памяти.

Предполагается, что читатель знаком со статическими структурами данных, такими как массивы или записи (структуры), которые имеются практически в любом языке программирования. Здесь будут рассматриваться только динамические структуры данных, т.е. такие структуры, у которых в процессе вычисления изменяются не только значения переменных, но даже и сама их структура. Естественно, что на определенном уровне детализации компоненты таких объектов представляют собой статические объекты, принадлежащие к базовым типам данных.

Поскольку с самого начала книги для описания эффективности алгоритмов используются асимптотические обозначения O, Θ, Ω, первый раздел этой главы дает краткое описание этих обозначений (в связи с краткостью этого материала выносить его в отдельную главу не имеет смысла).

Описание эффективности и асимптотические обозначения

Перед тем как перейти к собственно структурам данных, вкратце напомним об обозначениях, используемых при описании эффективности алгоритмов. Временн*а*я эффективность алгоритма обычно выражается как функция размера входных данных n (в ряде алгоритмов для оценки размера входных данных может использоваться несколько параметров одновременно — например, в алгоритмах для работы с графами такими параметрами являются количество вершин и количество ребер графа). В большинстве случаев выбор такого параметра не составляет труда. Например, для задач сортировки, поиска, нахождения наименьшего элемента и многих других алгоритмов обработки

списков таким параметром является размер списка. При вычислении значения многочлена степени n $p(x)=a_n x^n+\cdots+a_0$ таким параметром может быть степень многочлена или количество его коэффициентов, которое на единицу больше степени многочлена. Подобные небольшие отличия не влияют на результаты анализа эффективности алгоритма.

Поскольку конкретные временны*е* характеристики алгоритма зависят от его реализации, использованного компилятора и компьютера, для оценки эффективности алгоритмов применяется такая характеристика, как асимптотическая зависимость количества базовых операций от размера входных данных при очень больших величинах последнего. Следует учесть, что для разных входных данных количество базовых операций может весьма существенно различаться. Например, рассмотрим поиск в массиве из n элементов, последовательно перебирающий элементы от первого до n-го. В *наилучшем случае*, когда искомый элемент — первый в массиве, нам потребуется только одно сравнение. В *наихудшем*, когда искомый элемент — последний, потребуется n сравнений. В *среднем* же случае потребуется $n/2$ сравнений. Заметим, что и в наихудшем, и в среднем случае количество требующихся сравнений линейно зависит от размера входных данных, так что они оба имеют одинаковый *порядок роста*.

Имеется еще один вид эффективности — так называемая амортизированная эффективность, когда рассматриваются не конкретные операции, а их последовательности. Возможны ситуации, когда конкретная операция над структурой данных занимает длительное время, но совокупность операций занимает меньше времени, чем сумма времен выполнения в наихудшем случае. Амортизационный анализ алгоритмов (как и обычный) детальнейшим образом рассмотрен в книге [12], к которой и рекомендуется обратиться заинтересованному читателю.

Для того чтобы можно было классифицировать и сравнивать между собой порядки роста, введены три условных обозначения: O, Θ и Ω. Ниже через $t(n)$ обозначено время выполнения некоторого алгоритма (выражающееся как количество базовых операций); $g(n)$ — некоторая простая функция, с которой будет проводиться сравнение количества операций $t(n)$.

Говоря нестрого, обозначение $O(g(n))$ — это множество всех функций, порядок роста которых при достаточно больших n не превышает некоторую константу, умноженную на значение функции $g(n)$.

$\Omega(g(n))$ – это множество всех функций, порядок роста которых при достаточно больших n не меньше некоторой константы, умноженной на значение функции $g(n)$. И, наконец, $\Theta(g(n))$ – это множество всех функций, порядок роста которых при достаточно больших n равен некоторой константе, умноженной на значение функции $g(n)$. Более строгие определения выглядят следующим образом:

$t(n) \in O(g(n))$, если существует положительная константа c и неотрицательное целое число n_0 такое, что $t(n) \le cg(n)$ для всех $n \ge n_0$;

$t(n) \in \Omega(g(n))$, если существует положительная константа c и неотрицательное целое число n_0, такое, что $t(n) \ge cg(n)$ для всех $n \ge n_0$;

$t(n) \in \Theta(g(n))$, если существуют положительные константы c_1 и c_2, а также неотрицательное целое число n_0 такое, что $c_2 g(n) \le t(n) \le c_1 g(n)$ для всех $n \ge n_0$.

Нетрудно доказать ряд свойств указанных обозначений. Это, в частности, транзитивность:

из $f(n)=\Theta(g(n))$ и $g(n)=\Theta(h(n))$ следует $f(n)=\Theta(h(n))$,

из $f(n)=O(g(n))$ и $g(n)=O(h(n))$ следует $f(n)=O(h(n))$,

из $f(n)=\Omega(g(n))$ и $g(n)=\Omega(h(n))$ следует $f(n)=\Omega(h(n))$,

из $f(n)=o(g(n))$ и $g(n)=o(h(n))$ следует $f(n)=o(h(n))$,

из $f(n)=\omega(g(n))$ и $g(n)=\omega(h(n))$ следует $f(n)=\omega(h(n))$,

рефлексивность:

$$f(n)=\Theta(f(n)),\ f(n)=O(f(n)),\ f(n)=\Omega(f(n)),$$

симметричность:

$f(n)=\Theta(g(n))$ справедливо тогда и только тогда, когда $g(n)=\Theta(f(n))$,

$f(n)=O(g(n))$ справедливо тогда и только тогда, когда $g(n)=\Omega(f(n))$.

Кроме того, если $t_1(n) \in O(g_1(n))$ и $t_2(n) \in O(g_2(n))$, то

$$t_1(n)+t_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

Аналогичные утверждения справедливы также для обозначений Ω и Θ.

Несмотря на то что без строгих определений множеств O, Ω, Θ нельзя обойтись при доказательстве их абстрактных свойств, они редко используются для сравнения порядков роста конкретных функций. Существует более удобный метод выполнения этой оценки, основанный на вычислении предела отношения двух рассматриваемых функций. Могут существовать три основные ситуации (не считая достаточно редко встречающегося на практике случая, когда предел не существует):

$$\lim_{n\to\infty}\frac{t(n)}{g(n)}=\begin{cases}0, & \text{если } t(n) \text{ имеет меньший порядок роста, чем } g(n),\\ c, & \text{если } t(n) \text{ имеет тот же порядок роста, чем } g(n),\\ \infty, & \text{если } t(n) \text{ имеет больший порядок роста, чем } g(n).\end{cases}$$

Обратите внимание на то, что для двух первых случаев $t(n)\in O(g(n))$, для двух последних — $t(n)\in\Omega(g(n))$, и для второго — $t(n)\in\Theta(g(n))$.

В этой книге мы ограничимся только приведенным кратким описанием асимптотических обозначений; более полное описание их свойств можно найти, например, в [12, 21].

Стеки и очереди

Стеки и очереди — это динамические множества, в которые можно вставлять и из которых можно удалять элементы при помощи соответствующих операций вставки и удаления, однако вставка и удаление не могут производиться в произвольном порядке. При добавлении элемента нельзя указать конкретное место в множестве, в которое он вносится, как нельзя указать, какой именно элемент удаляется из множества. Из *стека* первым всегда удаляется элемент, который был помещен туда последним, т.е. в стеке реализуется стратегия «последним вошел — первым вышел». Аналогично в *очереди* всегда первым удаляется элемент, который содержится в ней дольше других: в очереди реализуется стратегия *«первым вошел — первым вышел»*. Существует несколько эффективных способов реализации стеков и очередей в компьютере, но здесь рассматривается только один из них, а именно — способ реализации при помощи обычного массива.

Стек

Операция вставки в стек часто называется *Push* (запись в стек), а операция удаления — *Pop* (снятие со стека). На рис. 1.1 схематично показана работа стека. При выполнении операции *Push* на *вершину стека* добавляется новый элемент (на рисунке это число 3), а при снятии со стека удаляется элемент из его вершины.

Стек с максимальным объемом n элементов можно реализовать в виде массива, например $S[1..n]$, и индекса последнего заполненного элемента массива *top*. Таким образом, значение *top*, равное 0, соответствует пустому стеку.

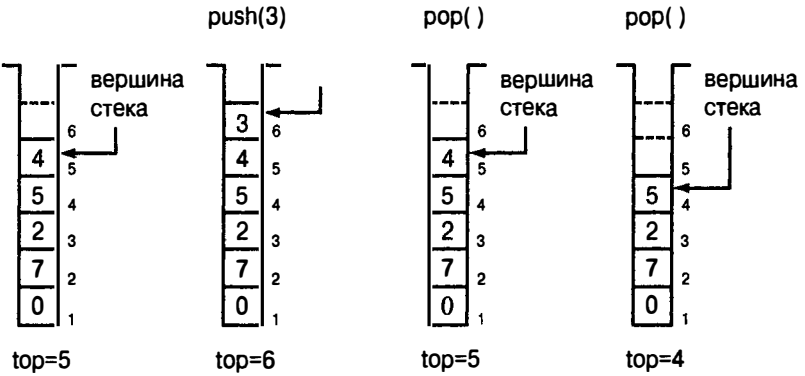


Рис. 1.1. *Реализация стека S в виде массива*

Протестировать стек на наличие в нем элементов можно с помощью операции *StackEmpty*, псевдокод которой приведен далее, и которая сводится к проверке значения индекса *top*. Если элемент снимается с пустого стека, говорят, что стек *опустошается*, что обычно приводит к ошибке. Если значение *top* превосходит n, то стек *переполняется*. В представленном ниже псевдокоде возможное переполнение стека во внимание не принимается.

Псевдокоды описанных операций над стеком состоят всего лишь из нескольких строк каждый.

```
StackEmpty(S)
if top = 0 then return true
else return false
```

```
Push(S,x)
top = top + 1
S[top] = x

Pop(S)
if StackEmpty(S) then error "Опустошение стека"
else top = top - 1
return S[top + 1]
```

На рис. 1.1 показано состояние стека для последовательности операций *Push*(3), *Pop*(), *Pop*(). Индексы массива, с помощью которого реализуется стек, приведены справа от изображения стека. Любая из рассмотренных операций со стеком выполняется за время $O(1)$.

Реализация стека на C++ тривиальна. Поскольку реализация стека имеется в стандартной библиотеке C++, здесь приведен простейший код, реализующий стек с фиксированным количеством элементов.

```
template<typename T, int N>
class Stack
{
public:
    Stack():top(0){}
    bool empty() { return top == 0; }
    void Push(const T& t);
    T Pop();
private:
    int top;
    T S[N];
};

template<typename T, int N>
void Stack<T,N>::Push(const T& t)
{
    assert((top < N) && "Переполнение стека");
    S[top++] = t;
}
template<typename T, int N>
T Stack<T,N>::Pop()
```

```
{
    assert(!empty() && "Опустошение стека");
    return S[--top];
}
```

По поводу данной реализации следует сделать несколько замечаний: поскольку она служит лишь иллюстрацией, в ней отсутствует масса необходимых вещей — в частности, у класса нет копирующего конструктора, оператора присваивания. Говорить об обобщенности такого решения можно лишь в том смысле, что в нем используется параметр типа `T`, но на самом деле, по сути, этот код не обеспечивает и сколь-нибудь значительной доли той гибкости, которая присуща настоящему обобщенному программированию. Обратите также внимание на небезопасность приведенного кода по отношению к исключениям (подробнее об этом можно прочесть в [18]).

Очереди

Применительно к очередям операция вставки называется *Enqueue* (поместить в очередь), а операция удаления — *Dequeue* (вывести из очереди). Так же, как и для стека, вы не можете указать, куда следует вставить новый элемент или какой элемент следует удалить из очереди. Благодаря стратегии *«первым вошел — первым вышел»* очередь действует как, например, живая очередь в магазине. У нее имеется *начало* и *конец*. Когда элемент ставится в очередь, он занимает место в ее конце. Из очереди всегда выводится элемент, который находится в ее начале.

На рис. 1.2 показан способ реализации очереди не более чем из $n-1$ элементов при помощи массива $Q[1..n]$. Для реализации очереди используются две индексные переменные — *head*, указывающая первый элемент очереди (который будет удален из нее очередной операцией *Dequeue*), и *tail*, указывающая позицию, в которую будет добавляться новый элемент. Элементы очереди расположены в ячейках $Q[head]$, $Q[head+1]$, ..., $Q[tail-1]$, которые циклически замкнуты в том смысле, что ячейка 1 следует сразу же после ячейки n в циклическом порядке. При условии $head = tail$ очередь пуста. Изначально выполняется соотношение $head = tail = 1$. Если очередь пуста, то при попытке удалить из нее элемент происходит ошибка опустошения. Если же $head = tail+1 \pmod n$, то очередь заполнена, и попытка добавить в нее элемент приводит к ее переполнению.

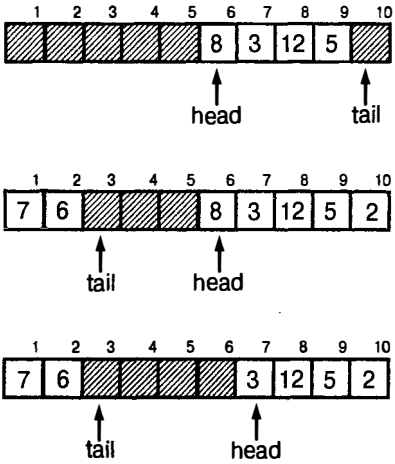


Рис. 1.2. *Реализация очереди с помощью массива*

В псевдокоде процедур *Enqueue* и *Dequeue* проверка на ошибки опустошения и переполнения не производится — вы легко можете добавить необходимый код самостоятельно.

```
Enqueue(Q,x)
Q[tail] = x
if tail = n then tail = 1
else tail = tail + 1

Dequeue(Q)
x = Q[head]
if head = n then head = 1
else head = head + 1
return x
```

На рис. 1.2 приведено состояние очереди после добавления трех и удаления одного элемента. Заштрихованы свободные ячейки массива, в которые можно выполнять вставку элементов. Любая из рассмотренных операций со стеком выполняется за время $O(1)$.

Реализация очереди на C++ приведена ниже. К ней применимы все те же замечания, которые были сделаны по поводу реализации на C++ стека.

```
template<typename T, int N>
class Queue
{
public:
    Queue():head(0),tail(0){}
    bool empty() { return head == tail; }
    void Enqueue(const T& t);
    T Dequeue();
private:
    int head, tail;
    T Q[N];
};

template<typename T, int N>
void Queue<T,N>::Enqueue(const T& t)
{
    assert((head%N!=(tail+1)%N) &&
           "Переполнение очереди");
    Q[tail] = t;
    tail = (tail+1)%N;
}

template<typename T, int N>
T Queue<T,N>::Dequeue()
{
    assert(!empty() && "Опустошение очереди");
    int oldhead = head;
    head = (head+1)%N;
    return Q[oldhead];
}
```

Связанные списки

Связанный список — это структура данных, в которой объекты расположены в линейном порядке. Однако, в отличие от массива, в котором этот порядок определяется индексами, порядок в связанном списке определяется указателями на объекты.

Элемент (узел) связанного списка помимо поля данных имеет поле *next*, в котором содержится указатель на следующий элемент списка (если это последний элемент списка, поле *next* принимает нулевое значение). Каждый список содержит помимо своих элементов указатель *head* на первый элемент списка. Если этот указатель равен 0, значит, список пуст.

Такой односвязный список схематически показан на рис. 1.3. Односвязный список отличается тем, что пройти по нему можно только в одном направлении – от начала в конец списка. Это оказывается достаточно неудобно, поэтому гораздо большее распространение получили *дважды связанные списки* (рис. 1.4), отличающиеся тем, что узлы такого списка содержат по два указателя — на следующий и предыдущий элементы списка. Кроме указателя *head* на первый элемент списка может существовать также указатель *tail* на последний элемент списка.

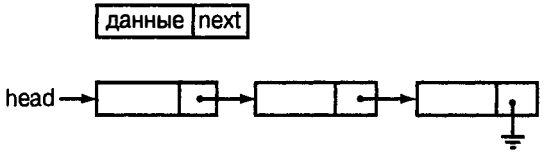


Рис. 1.3. *Односвязный список*

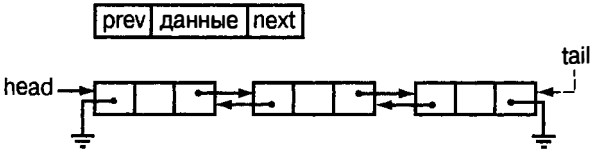


Рис. 1.4. *Дважды связанный список*

Частный случай дважды связанного списка — замкнутый (кольцевой) список, указатель *next* последнего элемента которого указывает на первый элемент, а указатель *prev* первого элемента — на последний элемент списка.

Главная особенность списка — быстрое выполнение операций вставки и удаления в произвольном месте списка. Эти операции требуют модификации указателей максимум у трех узлов — узла, с которым выполняется операция, и окружающих. Изменения значений указателей очевидны; схематично они показаны на рис. 1.5.

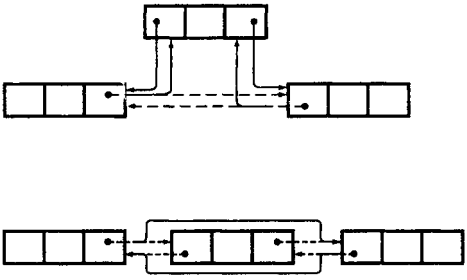


Рис. 1.5. *Вставка и удаление в дважды связанном списке*

Псевдокод этих операций не сложнее их описания.

```
ListInsert(L,x)
// Входные данные:  Список L, вставляемый узел x
// Выходные данные: Список L, в который вставлен
//                  узел x
next[x] = head[L]
if head[L] ≠ 0 then prev[head[L]] = x
head[L] = x
prev[x] = 0

ListDelete(L,x)
// Входные данные:  Список L, удаляемый узел x
// Выходные данные: Список L, из которого удален
//                  узел x
if prev[x] ≠ 0 then next[prev[x]] = next[x]
               else head[L] = next[x]
if next[x] ≠ 0 then prev[next[x]] = prev[x]
```

Что касается реализации списков на C++, то в стандартной библиотеке C++ содержится шаблонный класс **list<T>**, который реализует список. Это полная и профессионально выполненная реализация, с которой невозможно тягаться, особенно при ограниченном объеме книги. Поэтому порекомендуем заинтересованному читателю обратиться к классу **list<T>** из стандартной библиотеки, а здесь будет приведена демонстрационная программа, в которой строится список целых чисел с использованием процедур вставки и удаления, а также показаны поиск узла (в процедуре удаления) и обход списка.

```
#include <iostream>
using namespace std;
class List
{
    struct Node         // Внутренняя структура узла
    {
        int data;
        Node * prev;
        Node * next;
        Node(int i):data(i),next(0),prev(0){}
    };
public:
    List():head(){}
    void insert(int x);  // Вставка узла
    void remove(int x);  // Удаление узла
    void out();          // Вывод списка
private:
    Node * head;
};
void List::insert(int x)
{
    Node * n = new Node(x);
    n->next = head;
    if (head) head->prev = n;
    head = n;
}

void List::remove(int x)
{
    Node * n;
    // Поиск удаляемого узла
    for(n = head; n; n = n->next)
        if (n->data == x) break;
    if (n == 0) return;
    if (n->prev) n->prev->next = n->next;
    else head = n->next;
    if (n->next) n->next->prev = n->prev;
    delete n;
}
```

```
void List::out()
{
    if (head == 0) cout << "Список пуст\n";
    else
    {
        Node * n;
        for(n = head; n; n = n->next)
            cout << n->data << endl;
    }
}

int main()
{
    List l;
    l.out();
    l.insert(1);
    l.insert(2);
    l.insert(3);
    l.remove(2);
    l.out();
}
```

Вывод данной демонстрационной программы на экран имеет вполне ожидаемый вид:

```
Список пуст
3
2
```

Заметим, что использование указателей — не единственный способ представления списков (как и других структур данных). Например, если язык программирования не позволяет пользоваться указателями и допускает только работу с массивами, то тот же дважды связанный список можно получить при помощи трех массивов. В первом массиве хранятся значения узлов списка, во втором и третьем — индексы предыдущего и последующего элементов в списке. Заметим, что таким образом одни и те же массивы могут использоваться для представления нескольких списков одновременно. Свободные элементы в этих массивах также объединяются в один (возможно, односвязный) список. Пример такого представления дважды связанных списков показан на рис. 1.6. Здесь представлены три списка — два списка с

данными и список свободных ячеек. Значения в массивах *prev* и *next* указывают индексы ячеек массивов, соответствующие предыдущему и следующему узлам списка. Значение 0 означает конец списка (нет предыдущего или последующего узла).

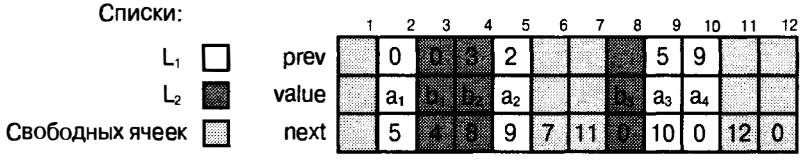


Рис. 1.6. *Представление списков в виде трех массивов*

Бинарные деревья поиска

Хотя деревья принято определять как частный случай графа (дерево является связным ациклическим неориентированным графом), здесь мы дадим рекурсивное определение дерева таким образом: *дерево* — это либо пустое дерево, либо узел с конечным числом связанных с ним отдельных деревьев, именуемых поддеревьями. Однако несмотря на все изобилие деревьев здесь мы рассмотрим только небольшое, но крайне важное их подмножество, а именно — *бинарные деревья поиска*. У такого дерева каждый узел имеет не более двух дочерних узлов, причем левый и правый узлы различаются. У каждого узла дерева имеется поле значения, хранящегося в узле, и поля, указывающие на левый и правый потомки данного узла, а также на родительский узел. Это — бинарное дерево. Бинарным деревом поиска его делает следующее свойство: значения в узлах дерева располагаются таким образом, что в любой момент для любого узла x значения всех узлов в его левом поддереве не превышают значения узла x, а значения всех узлов в его правом поддереве не меньше значения узла x. На рис. 1.7 показано несколько деревьев. Дерево *а* не является бинарным, так как у узла 5 — три дочерних узла; дерево *б* является бинарным, но не является бинарным деревом поиска — так как узел 2 является правым дочерним узлом по отношению к узлу 3, и тем самым нарушается свойство бинарного дерева поиска. Дерево *в* представляет собой корректное бинарное дерево поиска, а на рис. 1.7, *г* изображено возможное представление бинарного дерева поиска с использованием полей указателей.

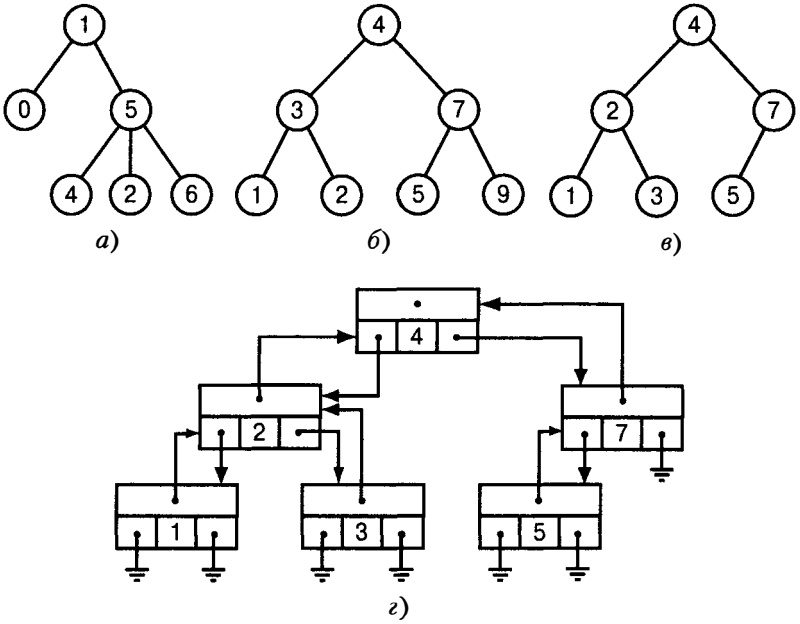


Рис. 1.7. *а) дерево; б) бинарное дерево; в) бинарное дерево поиска; г) одно из представлений бинарного дерева поиска*

Посетить все узлы дерева очень легко с помощью рекурсивной процедуры обхода. Основные варианты обхода бинарного дерева — симметричный обход дерева, когда для каждого узла сначала рекурсивно выполняется посещение его левого поддерева, затем самого узла, а после этого — узлов его правого поддерева. Для дерева на рис. 1.7, *в* симметричный обход дает следующую последовательность узлов: 1,2,3,4,5,7. Как видите, таким образом можно получить отсортированную последовательность значений узлов.

Два других распространенных метода обхода дерева — обход в прямом порядке, при котором сначала выводится корень, а потом — значения левого и правого поддеревьев (для уже рассматривавшегося дерева это дает последовательность 4,2,1,3,7,5), и обход в обратном порядке, при котором сначала выводятся значения узлов левого и правого поддеревьев, а затем — корня (для нашего дерева это дает последовательность 1,3,2,5,7,4).

Псевдокоды описанных обходов дерева очень просты. Вот псевдокод симметричного обхода:

```
InorderTreeWalk(x,f)
// Входные данные:  x — корневая вершина дерева,
//                  f — функция, вызываемая для
//                  значений
//                  каждого узла при обходе дерева
// Выходные данные: выполнение функции f для
//                  значения
//                  каждого узла
if x = 0 then return
InorderTreeWalk(left[x],f)
f(value[x])
InorderTreeWalk(right[x],f)
```

Написать самостоятельно соответствующие псевдокоды для обходов в прямом и обратном порядке не должно составить для вас никакого труда. Очевидно, что требуется $O(n)$ времени для обхода всего дерева (в предположении, что эффективность выполнения функции f составляет $O(1)$).

Основные операции при работе с бинарным деревом поиска — это поиск в нем определенного значения, а также поиск наименьшего и наибольшего элемента дерева, предшествующего и последующего элементов для данного.

Выполнение операции поиска основано на том, что, находясь на определенной вершине, можно всегда однозначно указать, в каком из поддеревьев находится искомое значение (если таковое имеется в данном дереве) — так как согласно свойству бинарного дерева поиска все значения узлов в левом поддереве не больше, а в правом — не меньше значения в корне. Таким образом, псевдокод рекурсивного варианта процедуры поиска заданного значения в бинарном дереве поиска имеет следующий вид:

```
TreeSearch(x,v)
// Входные данные:  x — корневой узел дерева, в
//                  котором выполняется поиск
//                  значения v
// Выходные данные: узел, значение которого равно v,
//                  либо 0, если такого узла нет
if x = 0 or value[x] = v then return x
if v < value[x] then return TreeSearch(left[x],v)
                else return TreeSearch(right[x],v)
```

Заметим, что этот псевдокод легко преобразовать из рекурсивного в итеративный:

```
TreeSearch(x,v)
// Входные данные:  x — корневой узел дерева, в котором
//                     выполняется поиск значения v
// Выходные данные: узел, значение которого равно v,
//                     либо 0, если такого узла нет
while x ≠ 0 and value[x] ≠ v do
begin
    if v < value[x] then x = left[x]
                    else x = right[x]
end
return x
```

В случае, если искомое значение в дереве отсутствует, возвращается нулевое значение.

Что касается поиска наименьшего и наибольшего элементов в бинарном дереве поиска, то из свойства бинарного дерева поиска очевидно, что, чтобы достичь наименьшего (наибольшего) элемента, надо двигаться по левым (или соответственно правым) ветвям дерева до тех пор, пока это возможно. Таким образом легко записать псевдокоды этих операций.

```
TreeMin(x)
// Входные данные:  x — корневой узел дерева
// Выходные данные: узел с минимальным значением
while left[x] ≠ 0 do x = left[x]
return x

TreeMax(x)
// Входные данные:  x — корневой узел дерева
// Выходные данные: узел с максимальным значением
while right[x] ≠ 0 do x = right[x]
return x
```

Поиск очередного и предшествующего узла — задача несколько более сложная, но весьма важная, например, для реализации итераторов в контейнере — бинарном дереве поиска. Интересно, что эта задача решается без выполнения непосредственного сравнения узлов, с использованием исключительно знаний о структуре дерева. Рассмотрим поиск следующего за x элемента. Если правое поддерево

x непустое, то очевидно, что следующий за x элемент — это минимальный элемент правого поддерева. Если же правое поддерево x пустое, и у x имеется следующий за ним элемент y, то y — наименьший предок x, левый наследник которого также является предком x. На рис. 1.7, *в* следующим за элементом 3 идет элемент 4, левый потомок которого 2 является предком 3. Для поиска такого предка мы просто идем по дереву в направлении корня, пока не найдем узел, который является левым дочерним узлом своего родителя (им может оказаться и сам текущий узел), так что псевдокод этой операции выглядит следующим образом.

```
TreeSuccessor(x)
// Входные данные:  x – узел дерева
// Выходные данные: узел дерева, следующий за x при
//                  симметричном обходе
if right[x] ≠ 0 then return TreeMin(right[x])
y = parent[x]
while y ≠ 0 and x = right[y] do
begin
    x = y
    y = parent[y]
end;
return y
```

Процедура поиска узла, предшествующего данному, симметрична процедуре поиска последующего узла.

```
TreePredecessor(x)
// Входные данные:  x – узел дерева
// Выходные данные: узел дерева, предшествующий
//                  x при симметричном обходе
if left[x] ≠ 0 then return TreeMax(left[x])
y = parent[x]
while y ≠ 0 and x = left[y] do
begin
    x = y
    y = parent[y]
end;
return y
```

Последние две рассматриваемые операции над бинарным деревом поиска — это вставка узла в дерево и его удаление из дерева. При

вставке узла в дерево мы сначала выполняем поиск места, куда следует вставить новый узел, а затем вставляем его, изменяя поля у вставляемого узла и его родительского узла.

```
TreeInsert(root,z)
// Входные данные:  узел z, добавляемый в дерево
//                    с корневым узлом root
// Выходные данные: дерево с добавленным в него узлом z
y = 0
x = root
while(x ≠ 0) do
begin
    y = x;
    if (value[z] < value[x]) then x = left[x]
                             else x = right[x]
end;
parent[z] = y
if y = 0 then root = z    // Вставка в пустое дерево
         else if value[z] < value[y]
                  then left[y] = z
                  else right[y] = z
```

Процедура начинает работу с корневого узла и идет вниз, перемещая указатель *x*. При проходе вниз указатель *y* постоянно указывает на родительский по отношению к *x* узел, а сам указатель перемещается в соответствии с результатами сравнения значений в текущем и вставляемом узлах. После того как указатель *x* становится равным 0, он находится именно в той позиции, куда следует поместить новый узел *z*.

Процедура удаления узла из дерева несколько сложнее, поскольку должна рассматривать различные варианты. Так, если у удаляемого узла нет дочерних узлов, то удаление сводится к тому, чтобы в родительском узле обнулить тот указатель на дочерний узел, который в настоящий момент указывает на удаляемый узел. Если у удаляемого узла только один дочерний узел, то удаление также легко осуществить — в этом случае соответствующий указатель в родительском узле должен указывать на дочерний по отношению к удаляемому узел. Соответственно должен быть исправлен и указатель на родительский узел в дочернем по отношению к удаляемому.

Если же у удаляемого узла два дочерних, то надо найти следующий за ним узел (у которого не может быть левого дочернего узла в

силу свойства бинарного дерева поиска), извлечь его из дерева и заменить им удаляемый узел. Все три рассмотренных варианта схематично представлены на рис. 1.8.

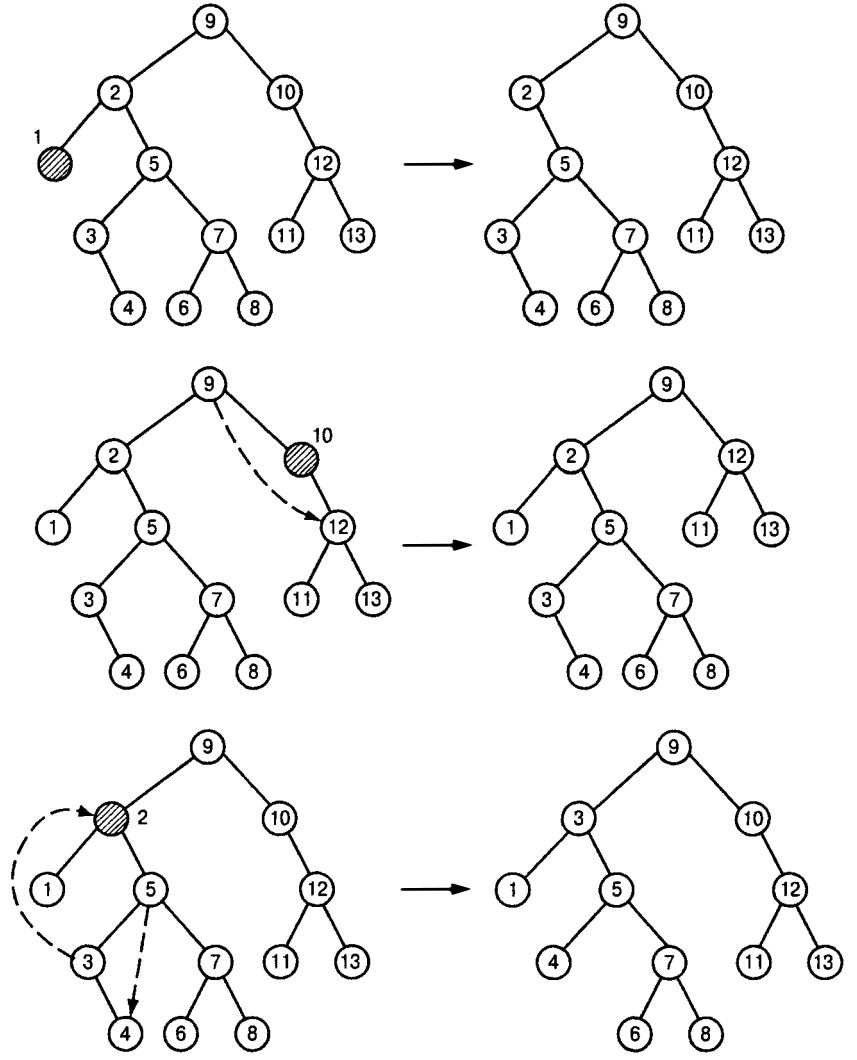


Рис. 1.8. *Удаление узла из бинарного дерева поиска*

Вот как выглядит псевдокод алгоритма удаления узла из бинарного дерева поиска.

```
TreeRemove(root,z)
// Входные данные: узел z, удаляемый из дерева с
//                 корневым узлом root
// Выходные данные: дерево с удаленным из него узлом
//                  z, указатель на удаленный узел
if left[z] = 0 or right[z] = 0
            then y = z
            else y = TreeSussessor(z)
if left[y] ≠ 0 then x = left[y]
               else x = right[y]
if x ≠ 0 then parent[x] = parent[y]
if parent[y] = 0 then root = x
    else if y = left[parent[y]]
            then left[parent[y]] = x
            else right[parent[y]] = x
if y ≠ z then value[z] = value[y]
return y
```

В приведенном псевдокоде два первых рассмотренных выше случая удаления узла объединены в один. Псевдокод возвращает удаленный узел для того, чтобы вызывающая программа могла, например, при необходимости освободить занимаемую им память.

В псевдокоде не происходит реальная замена одного узла другим — заменяется только значение узла. Однако если такая замена — дорогостоящая операция, можно реально заменить узел; при этом предпоследняя строка псевдокода должна быть заменена на обновление указателей в родительском и дочерних узлах (само собой, процедура при этом возвращает в качестве удаленного не узел *y*, а узел *z*):

```
if y ≠ z then begin
    left[y] = left[z]
    right[y] = right[z]
    parent[y] = parent[z]
    if left[z] ≠ 0 then parent[left[z]] = y
    if right[z] ≠ 0 then parent[right[z]] = y
    if root = z then root = y
        else if z = left[parent[z]]
                then left[parent[z]] = y
                else right[parent[z]] = y
end
```

Все приведенные в этом разделе операции с бинарным деревом поиска выполняются за время $O(h)$, где h – высота дерева. В наилучшем случае, когда дерево приближается к полному, т.е. когда у каждого внутреннего узла дерева по два потомка, высота дерева составляет примерно $\log_2 n$, где n – общее количество узлов в дереве. Таким образом, в наилучшем случае эффективность всех описанных операций над бинарным деревом поиска составляет $O(\log n)$. Однако в вырожденном, наихудшем случае, когда дерево вырождается в одну цепочку, эффективность падает до $O(n)$. К счастью, математическое ожидание высоты случайного бинарного дерева поиска с n узлами равно $O(\log n)$.

Количество различных бинарных деревьев с n узлами определяется n-м числом Каталана $b_n = \frac{1}{n+1} C_{2n}^{n}$. О том, как сгенерировать все возможные деревья, рассказывается в разделе «Генерация всех деревьев» на стр. 203.

Далее приведена реализация бинарного дерева поиска на C++. В данной реализации используется только один вид обхода бинарного дерева – симметричный, реализованный двумя способами – посредством итератора (причем данный итератор – крайне упрощенная версия, никак не согласующаяся с итераторами стандартной библиотеки и приведенная исключительно в методических целях) и процедуры обхода *InorderTreeWalk*, которой в качестве параметра передается функтор. Заметим, что в данной реализации обход при помощи итератора оказывается не самым эффективным, поскольку общее время обхода в этом случае составляет $O(nh)$ (n вызовов процедуры *TreeSuccessor*), в то время как обход при помощи процедуры *InorderTreeWalk* требует времени $\Theta(n)$.

```
template<typename T, typename Less = std::less<T> >
class BST
{
    struct Node
    {   // Внутренняя структура, представляющая
        // узел дерева
        T      value;
        Node * left;
```

```
        Node * right;
        Node * parent;
        Node(const T& t):value(t),left(0),
                         right(0),parent(0){}
        ~Node(){ delete left; delete right; }
    };
public:
    class Iterator
    {          // Итератор (крайне ограниченный,
    public:    // приведен с методическими целями
        Iterator(BST<T,Less>*T = 0, Node * N = 0)
            :tree(T),curr(N){};
        T& operator*() { return curr->value; }
        void operator++();
        void operator--();
        bool operator == (const Iterator& i)
                { return curr == i.curr; }
        bool operator != (const Iterator& i)
                { return curr != i.curr; }
    private:
        BST<T,Less>*tree;
        Node * curr;
    };
public:
    BST(const Less& L = Less()):root(0),comp(L){}
    ~BST(){ delete root; }
    void Insert(const T& t);    // Вставка объекта
    void Remove(const T& t);    // Удаление объекта
    BST& operator<<(const T& t)// Оператор для
            { Insert(t);       // цепочной вставки
              return *this; }
    T& min() const          // Минимальный элемент
      { assert(!empty()); return min(root)->value;}
    T& max() const          // Максимальный элемент
      { assert(!empty()); return max(root)->value;}
    bool empty() const { return root == 0; }
    template<typename Func> void InorderWalk(Func f)
      { InorderWalk(root,f); }
```

```
    Iterator begin()
      { return Iterator(this,min(root)); }
    Iterator end()   { return Iterator(this); }
    Iterator find(const T&);
private:
    static Node* min(Node *);
    static Node* max(Node *);
    static Node* succ(Node *);
    void         Remove(Node *);
    Node * search(const T&);
    Node * root;
    Less   comp;
    template<typename Func>
        void InorderWalk(Node *, Func f);
    friend class Iterator;
};

template<typename T, typename Less>
void BST<T,Less>::Insert(const T& t)
{
    Node * z = new Node(t);
    Node * y = 0, * x = root;
    while(x)
    {
      y = x;
      x = (comp(z->value,x->value)) ?
                x->left : x-> right;
    }
    z->parent = y;
    if (y)
        (comp(z->value,y->value) ?
                y->left : y->right) = z;
    else root = z;
}

template<typename T, typename Less>
typename BST<T,Less>::Node*
            BST<T,Less>::search(const T& t)
```

```
{
    if (root == 0) return 0;
    Node * x = root;
    while(x && (comp(x->value,t)||
          (comp(t,x->value))))
        x = comp(t,x->value) ? x->left : x->right;
    return x;}

template<typename T, typename Less>
typename BST<T,Less>::Iterator
            BST<T,Less>::find(const T& t)
{
    return Iterator(this,search(t));
}

template<typename T, typename Less>
typename BST<T,Less>::Node*
    BST<T,Less>::min(typename BST<T,Less>::Node*x)
{
    if (x) for(; x->left; x = x->left);
    return x;
}

template<typename T, typename Less>
typename BST<T,Less>::Node*
    BST<T,Less>::max(typename BST<T,Less>::Node*x)
{
    if (x) for(; x->right; x = x->right);
    return x;
}

template<typename T, typename Less>
typename BST<T,Less>::Node*
  BST<T,Less>::succ(typename BST<T,Less>::Node * x)
{
    if (x->right) return min(x->right);
    Node * y = x->parent;
    while(y && x==y->right)
```

```
    {
        x = y;
        y = y->parent;
    }
    return y;
}

template<typename T, typename Less>
void BST<T,Less>::Iterator::operator++()
{
    if (curr == 0) return;
    curr = succ(curr);
}

template<typename T, typename Less>
void BST<T,Less>::Iterator::operator--()
{
    if (curr == 0) {
        curr = max(tree->root);
        return;
    }
    if (curr->left) {
        curr=max(curr->left);
        return;
    }
    Node * y = curr->parent;
    while(y && curr == y->left)
    {
        curr = y;
        y = y->parent;
    }
    curr = y;
}

template<typename T, typename Less>
void BST<T,Less>::Remove(Node * z)
{
    Node * y =
```

```
      ((z->left == 0)||(z->right == 0))? z:succ(z);
    Node * x = (y->left) ? y->left : y->right;
    if (x) x->parent = y->parent;
    if (y->parent == 0) root = x;
    else ((y == y->parent->left) ?
            y->parent->left : y->parent->right) = x;
#if ValueCopy   // Вариант с копированием значения
    if (y != z) z->value = y->value;
    y->left = y->right = 0;
    delete y;
#else           // Вариант с переносом узла
    if (y != z)
    {
        y->left   = z->left;
        y->right  = z->right;
        y->parent = z->parent;
        if (z->left) z->left->parent = y;
        if (z->right) z->right->parent = y;
        if (root == z) root = y; else
            ((z == z->parent->left) ?
            z->parent->left : z->parent->right) = y;
    }
    z->left = z->right = 0;
    delete z;
#endif
}

template<typename T, typename Less>
void BST<T,Less>::Remove(const T& t)
{
    Node * z = search(t);
    if (z) Remove(z);
}

template<typename T, typename Less>
template<typename Func> void
    BST<T,Less>::InorderWalk(
        typename BST<T,Less>::Node*x, Func f)
```

```
{
    if (x == 0) return;
    InorderWalk(x->left,f);
    f(x->value);
    InorderWalk(x->right,f);
}
```

Вот краткий пример использования приведенной реализации бинарного дерева поиска.

```
void out(int x)
{
    std::cout << "Value = " << x << "\n";
}
...

BST<int> B;
B <<  9 << 10 << 3 <<  2 <<  6 <<  7 <<  5 <<  8
  << 11 <<  1 << 4 << 12 << 13 << 15 << 14 << 16;
std::cout << B.min() << "  " << B.max() << "\n";

// Вывод в прямом и обратном порядке
// с использованием итераторов
BST<int>::Iterator i, j = B.end(); --j;
for(i = B.begin(); i != B.end(); ++i, --j)
{
    std::cout << *i << "    " << *j << "\n";
}
// Вывод упорядоченного списка с использованием
// процедуры симметричного обхода бинарного дерева
B.InorderWalk(out);

// Поиск элемента
i = B.find(5);
if (i != B.end())
        std::cout << "Found: "<< *i << "\n";
    else std::cout << "Not found 5\n";
```

```
i = B.find(13);
if (i != B.end())
        std::cout << "Found: " << *i << "\n";
    else std::cout << "Not found 13\n";

// Удаление элементов
B.Remove(9);
B.Remove(1);
B.Remove(6);
B.InorderWalk(out);
```

Красно-черные деревья

Как упоминалось в предыдущем разделе, обычное бинарное дерево поиска в наихудшем случае представляет собой одномерную цепочку узлов, а высота такого дерева становится равной n. Такое дерево образуется, например, при внесении в него возрастающей последовательности значений.

Однако путем достаточно небольших модификаций можно гарантировать, что даже в наихудшем случае высота бинарного дерева поиска будет равна $\Theta(\log n)$. Заметим, что операции поиска минимального, максимального, последующего и предшествующего элементов, а также поиска элемента с заданным значением (соответственно, и реализация итераторов) зависят только от свойства бинарного дерева поиска и, таким образом, остаются неизменными для любых бинарных деревьев поиска — будь то рассмотренные выше простейшие бинарные деревья поиска или рассматриваемые далее красно-черные деревья. Однако того же нельзя сказать об операциях вставки и удаления, поскольку именно при их выполнении нарушается свойство сбалансированности дерева, которое и подлежит восстановлению.

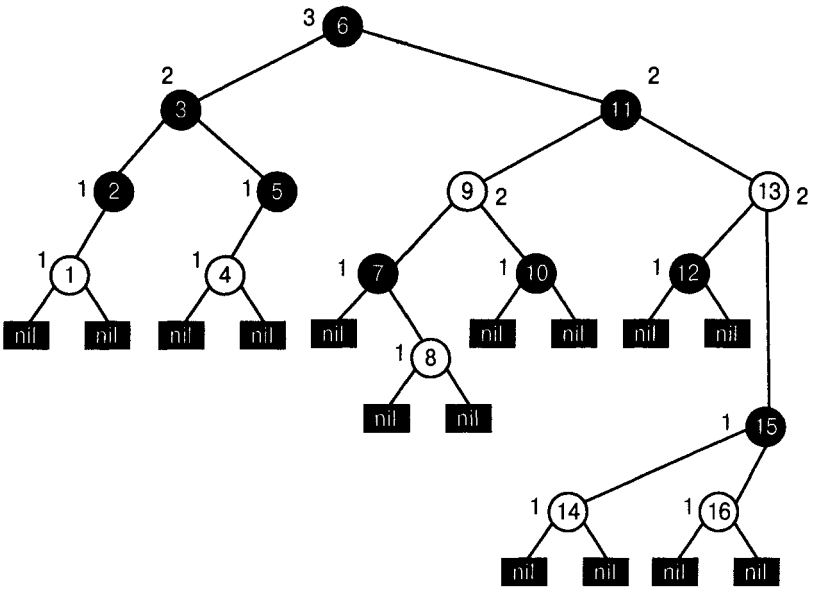
Красно-черные деревья гарантируют, что ни один путь в дереве от корня к вершине не отличается от другого по длине более чем в два раза, так что красно-черное дерево является приближенно сбалансированным и имеет высоту не более, чем $2\log_2(n+1)$. Каждый узел красно-черного дерева содержит, помимо указателей, дополнительное поле цвета, который может быть либо красным, либо черным (откуда и происходит название данного дерева). Если дочерний или родительский по отношению к данному узел не существует, соответствующий указатель принимает специальное значение *nil*. Эти значения

nil можно рассматривать как указатели на внешние узлы (листья) бинарного дерева поиска. При этом все «нормальные» узлы, содержащие поле значения, становятся внутренними узлами дерева.

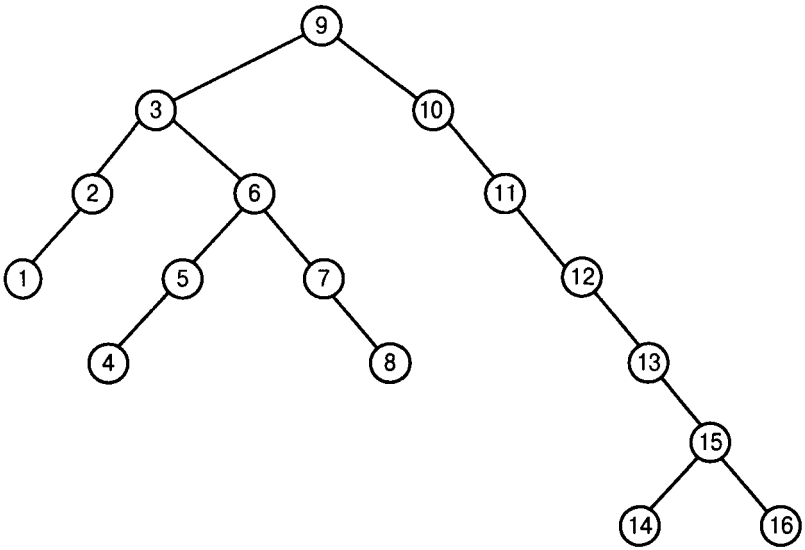
Бинарное дерево поиска является красно-черным деревом, если оно удовлетворяет следующим красно-черным свойствам.

1. Каждый узел является красным или черным.
2. Корень дерева является черным.
3. Каждый лист дерева (*nil*) является черным.
4. Если узел — красный, то оба его дочерних узла — черные.
5. Для каждого узла все пути от него до листьев, являющихся потомками данного узла, содержат одно и то же количество черных узлов.

На рис. 1.9, *а* представлен пример красно-черного дерева. Рядом с узлами стоят цифры, указывающие количество черных узлов на пути от листьев к данному узлу.



а)



б)

Рис. 1.9. *а) красно-черное дерево и б) бинарное дерево поиска для входных данных 9, 10, 3, 2, 6, 7, 5, 8, 11, 1, 4, 12, 13, 15, 14, 16*

Можно доказать, что высота красно-черного дерева с n узлами не превышает $2\log_2(n+1)$, так что все операции поиска минимального, максимального, последующего и предшествующего элементов, а также поиска элемента с заданным значением обладают эффективностью $O(\log n)$ в наихудшем случае.

Для удобства работы с красно-черным деревом все листья заменяются единым ограничивающим узлом, представляющим значение *nil*. Этот узел — черный (значения прочих полей не имеют значения). Кроме того, поскольку, как уже говорилось, в случае отсутствия родительского узла соответствующий указатель также принимает значение *nil*, этот узел-ограничитель выполняет функции родительского узла по отношению к корню красно-черного дерева.

Процедуры вставки узла в красно-черное дерево и удаления из него требуют определенных модификаций. Дело в том, что, если применять рассмотренные ранее процедуры *TreeInsert* и *TreeRemove*, корректно вставляя и удаляя узлы, будут нарушены свойства красно-черного дерева. Для исправления ситуации используется процедура,

именуемая *поворотом*, которая представляет собой локальную операцию в бинарном дереве поиска, сохраняющую его свойство. На рис. 1.10 показаны два типа поворотов — левый и правый (здесь a, b и c — произвольные поддеревья). При выполнении левого поворота в узле x предполагается, что его правый дочерний узел y не является листом *nil*. Левый поворот выполняется вокруг связи между x и y, делая y новым корнем поддерева, левым дочерним узлом которого становится x, а бывший левый потомок узла y — правым потомком x.

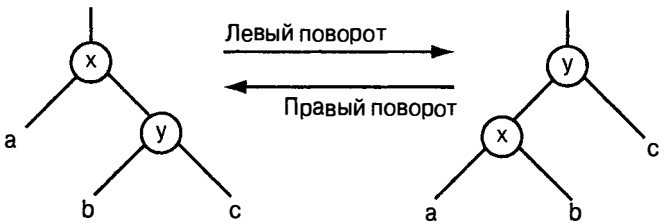


Рис. 1.10. *Операции поворота в бинарном дереве поиска*

Ниже приведен псевдокод левого поворота (правый поворот полностью симметричен левому). В псевдокоде процедуры *LeftRotate* предполагается, что $right[x] \neq nil$, и что *nil* также является родителем корневого узла.

```
LeftRotate(root,x)
// Входные данные: узел x, вокруг которого
//                 выполняется левый поворот в
//                 дереве с корневым узлом root
// Выходные данные: дерево с выполненным поворотом
y = right[x]          // Присваивание y
right[x] = left[y]    // Левое поддерево y становится
                      // правым поддеревом x
parent[left[y]] = x
parent[y] = parent[x] // Перенос родителя x в y
if parent[x] = nil then root = y
    else begin
        if x = left[parent[x]]
            then left[parent[x]]  = y
            else right[parent[x]] = y
    end
```

```
left[y] = x              // x - левый дочерний y
parent[x] = y
```

Очевидно, что данная процедура (как и процедура правого поворота) выполняется за время $O(1)$.

Вставка узла в красно-черное дерево с n узлами выполняется, как и в обычное бинарное дерево поиска, за время $O(\log n)$. Для вставки узла в красно-черное дерево используется модифицированная версия процедуры *TreeInsert*, которая вставляет узел в дерево, как если бы это было обычное бинарное дерево поиска (только вместо нулевых указателей теперь используется значение *nil*), а затем окрашивает его в красный цвет. Для того чтобы вставка сохраняла красно-черные свойства дерева, после нее вызывается вспомогательная процедура *RBInsertFixup*, которая перекрашивает узлы и выполняет повороты.

```
RBInsert(root,z)
// Входные данные: узел z, добавляемый в дерево с
//                 корневым узлом root
// Выходные данные: красно-черное дерево с
//                  добавленным в него узлом z
y = nil
x = root
while x ≠ nil do
begin
    y = x
    if value[z] < value[x] then x = left[x]
                           else x = right[x]
end
parent[z] = y
if y = nil then root = z
    else begin
        if value[z] < value[y] then left[y] = z
                               else right[y] = z
    end
left[z]  = nil
right[z] = nil
color[z] = RED
RBInsertFixup(root,z)
```

Описание нарушений свойств красно-черного дерева после вставки узла и методов их исправления при помощи поворотов и перекрашивания достаточно громоздко, так что здесь будет приведен только псевдокод указанной процедуры. Разобраться в том, какие свойства красно-черного дерева могут оказаться нарушены и как процедура *RBInsertFixup* справляется с этим, можно самостоятельно (это будет отличной практикой), либо обратившись к литературе, например [12].

```
RBInsertFixup(root,z)
// Входные данные: узел z, добавленный в дерево с
//                 корневым узлом root
// Выходные данные: красно-черное дерево с
//                 восстановленными после
//                 добавления узла z красно-черными
//                 свойствами
while color[parent[z]] = RED do
begin
    if parent[z] = left[parent[parent[z]]]
        then begin
            y = right[parent[parent[z]]]
            if color[y] = RED
                then begin
                    color[parent[z]] = BLACK
                    color[y] = BLACK
                    color[parent[parent[z]]] = RED
                    z = parent[parent[z]]
                end
                else begin
                    if z = right[parent[z]]
                        then begin
                            z = parent[z]
                            LeftRotate(root,z)
                        end
                    color[parent[z]] = BLACK
                    color[parent[parent[z]]] = RED
                    RightRotate(root,
                                parent[parent[z]])
                end
        else  begin
            Здесь код такой же, как и в части
```

```
            "then", но все "left" в нем заменяются
            на "right" (включая направления
            поворотов) и наоборот
        end
end
color[root] = BLACK
```

Еще более сложной оказывается процедура удаления узла из красно-черного дерева. Несмотря на свою сложность, она также выполняется за время $O(\log n)$. Процедура *RBRemove* представляет собой немного измененную процедуру *TreeRemove*. После удаления узла в ней вызывается вспомогательная процедура *RBRemoveFixup*, которая изменяет цвета и выполняет повороты для восстановления красно-черных свойств дерева.

```
RBRemove(root,z)
// Входные данные:  узел z, удаляемый из дерева с
//                    корневым узлом root
// Выходные данные: дерево с удаленным из него узлом z,
//                    указатель на удаленный узел
if left[z] = nil or right[z] = nil
        then y = z
        else y = TreeSuccessor(z)
if left[y] ≠ nil then x = left[y]
                 else x = right[y]
parent[x] = parent[y]
if parent[y] = nil then root = x
    else begin
        if y = left[parent[y]]
                then left[parent[y]] = x
                else right[parent[y]] = x
    end
if y ≠ z then value[z] = value[y]
if color[y] = BLACK then RBRemoveFixup(root,x)
return y
```

Что касается процедуры *RBRemoveFixup*, восстанавливающей красно-черные свойства дерева после удаления узла, то к ней применимы все те же замечания, которые были сделаны по поводу процедуры *RBInsertFixup*.

```
RBRemoveFixup(root,x)
// Входные данные:  узел x, для которого в дереве с
//                  корневым узлом root следует
//                  восстановить свойства
//                  красно-черного дерева
// Выходные данные: красно-черное дерево с
//                  восстановленными красно-черными
//                  свойствами, нарушенными после
//                  удаления узла
while x ≠ root and color[x] = BLACK do
begin
    if x = left[parent[x]]
        then begin
           w = right[parent[x]]
            if color[w] = RED then begin
                    color[w] = BLACK
                    color[parent[x]] = RED
                    LeftRotate(T,parent[x])
                    w = right[parent[x]]
                end
            if color[left[w]] = BLACK and
               color[right[w]] = BLACK
                then begin
                     color[w] = RED
                     x = parent[x]
                end
                else begin
                  if color[right[w]] = BLACK
                    then begin
                         color[left[w]] = BLACK
                         color[w] = RED
                         RightRotate(root,w)
                         w = right[parent[x]]
                    end
                    color[w] = color[parent[x]]
                    color[parent[x]] = BLACK
                    color[right[w]] = BLACK
                    LeftRotate(root,parent[x])
                    x = root
                end
        end
```

```
        else begin
            Здесь код такой же, как и в части
            "then", но все "left" в нем заменяются
            на "right" (включая направление
            поворотов) и наоборот
        end
end
color[x] = BLACK
```

На этом рассмотрение алгоритмов красно-черного дерева завершается. Итак, мы получили бинарное дерево поиска, все основные операции над которым выполняются за время $O(\log n)$ в наихудшем случае.

Реализация красно-черного дерева на C++ очень похожа на реализацию обычного бинарного дерева, с очевидными изменениями (в частности, заменой нулевых указателей на *nil*) и добавлениями (например, поля цвета в структуру *Node* или функций для восстановления красно-черных свойств дерева при вставке и удалении).

```
template<typename T, typename Less = std::less<T> >
class RBT
{
    struct Node
    {
        enum { RED = true, BLACK = false};
        T      value;
        Node * left;
        Node * right;
        Node * parent;
        bool   color;
        Node(const T& t):value(t),left(0),right(0),
                         parent(0),color(BLACK){}
        ~Node(){}
    };
public:
    class Iterator
    {
    public:
        Iterator(RBT<T,Less>*T = 0, Node * N = 0)
            :tree(T),curr(N){};
        T& operator*() { return curr->value; }
        void operator++();
```

```
        void operator--();
        bool operator == (const Iterator& i)
                { return curr == i.curr; }
        bool operator != (const Iterator& i)
                { return curr != i.curr; }
    private:
        RBT<T,Less>*tree;
        Node * curr;
    };
public:
    RBT(const Less& L = Less()):comp(L)
            {nil = new Node(T()); root = nil; }
    ~RBT(){ Destroy(root); delete nil; }
    void Insert(const T& t);
    void Remove(const T& t);
    RBT& operator << (const T& t)
            { Insert(t); return *this; }
    T& min() { assert(!empty());
               return min(root)->value; }
    T& max() { assert(!empty());
               return max(root)->value; }
    Iterator begin()
             { return Iterator(this,min(root)); }
    Iterator end()
             { return Iterator(this,nil); }
    Iterator find(const T&);
    bool empty() const { return root == nil; }
    template<typename Func> void InorderWalk(Func f)
            { InorderWalk(root,f); }
private:
    void   LRotate(Node *);
    void   RRotate(Node *);
    void   InsertFixup(Node*);
    void   RemoveFixup(Node*);
    Node*  min(Node *);       Node* max(Node *);
    Node*  succ(Node *);      void  Remove(Node *);
    void   Destroy(Node *);   Node* search(const T&);
    template<typename Func>
        void InorderWalk(Node *, Func f);
    Node * root, * nil;
    Less   comp;
```

```
    friend class Iterator;
};

template<typename T, typename Less>
void RBT<T,Less>::Insert(const T& t)
{
    Node * z = new Node(t);
    Node * y = nil, * x = root;
    while(x != nil)
    {
        y = x;
        x = (comp(z->value,x->value)) ?
                    x->left : x-> right;
    }
    z->parent = y;
    if (y == nil)
    {
        root = z;
    } else {
        (comp(z->value,y->value) ?
                    y->left : y->right) = z;
    }
    z->left = z->right = nil;
    z->color = Node::RED;
    InsertFixup(z);
}

template<typename T, typename Less>
typename RBT<T,Less>::Node*
            RBT<T,Less>::search(const T& t)
{
    if (root == nil) return nil;
    Node * x = root;
    while((x != nil) &&
          (comp(x->value,t)||(comp(t,x->value))))
        x = comp(t,x->value) ? x->left : x->right;
    return x;
}

template<typename T, typename Less>
typename RBT<T,Less>::Iterator
```

```
            RBT<T,Less>::find(const T& t)
{
    return Iterator(this,search(t));
}

template<typename T, typename Less>
typename RBT<T,Less>::Node*
    RBT<T,Less>::min(typename RBT<T,Less>::Node*x)
{
    if (x != nil) for(; x->left != nil;
                      x = x->left);
    return x;
}

template<typename T, typename Less>
typename RBT<T,Less>::Node*
    RBT<T,Less>::max(typename RBT<T,Less>::Node*x)
{
    if (x != nil) for(; x->right != nil;
                      x = x->right);
    return x;
}

template<typename T, typename Less>
typename RBT<T,Less>::Node*
  RBT<T,Less>::succ(typename RBT<T,Less>::Node * x)
{
    if (x->right != nil) return min(x->right);
    Node * y = x->parent;
    while(y != nil && x==y->right)
    {
        x = y;
        y = y->parent;
    }
    return y;
}

template<typename T, typename Less>
void RBT<T,Less>::Iterator::operator++()
{
    if (curr == tree->nil) return;
```

```
    curr = tree->succ(curr);
}

template<typename T, typename Less>
void RBT<T,Less>::Iterator::operator--()
{
    if (curr == tree->nil) {
        curr = tree->max(tree->root);
        return;
    }
    if (curr->left != tree->nil) {
        curr = tree->max(curr->left);
        return;
    }
    Node * y = curr->parent;
    while(y != tree->nil && curr == y->left)
    {
        curr = y;
        y = y->parent;
    }
    curr = y;
}

template<typename T, typename Less>
void RBT<T,Less>::Remove(Node * z)
{
    Node * y =
        ((z->left == nil)||(z->right == nil))?
                    z : succ(z);
    Node * x = (y->left != nil) ?
                    y->left : y->right;
    x->parent = y->parent;
    if (y->parent == nil) root = x;
    else ((y == y->parent->left) ?
                    y->parent->left :
                    y->parent->right) = x;
    if (y != z) z->value = y->value;
    if (y->color == Node::BLACK) RemoveFixup(x);
    delete y;
}
```

```
template<typename T, typename Less>
void RBT<T,Less>::Remove(const T& t)
{
    Node * z = search(t);
    if (z) Remove(z);
}

template<typename T, typename Less>
template<typename Func>
    void RBT<T,Less>::InorderWalk(
        typename RBT<T,Less>::Node*x,Func f)
{
    if (x == nil) return;
    InorderWalk(x->left,f);
    f(x->value);
    InorderWalk(x->right,f);
}

template<typename T, typename Less>
void RBT<T,Less>::LRotate(
            typename RBT<T,Less>::Node *x)
{
    Node * y = x->right;
    x->right = y->left;
    if (y->left != nil) y->left->parent = x;
    y->parent = x->parent;
    if (x->parent == nil) root = y;
    else ((x == x->parent->left) ?
                    x->parent->left :
                    x->parent->right) = y;
    y->left = x;
    x->parent = y;
}

template<typename T, typename Less>
void RBT<T,Less>::RRotate(
            typename RBT<T,Less>::Node *x)
{
    Node * y = x->left;
    x->left = y->right;
```

```
    if (y->right != nil) y->right->parent = x;
    y->parent = x->parent;
    if (x->parent == nil) root = y;
    else ((x == x->parent->right) ?
                x->parent->right :
                x->parent->left) = y;
    y->right = x;
    x->parent = y;
}

template<typename T, typename Less>
void RBT<T,Less>::InsertFixup(
            typename RBT<T,Less>::Node*z)
{
    Node * y;
    while(z->parent->color == Node::RED)
    {
        if (z->parent == z->parent->parent->left)
        {
            y = z->parent->parent->right;
            if (y->color == Node::RED)
            {
                z->parent->color = Node::BLACK;
                y->color = Node::BLACK;
                z->parent->parent->color =
                                    Node::RED;
                z = z->parent->parent;
            } else {
                if (z == z->parent->right)
                {
                    z = z->parent;
                    LRotate(z);
                }
                z->parent->color = Node::BLACK;
                z->parent->parent->color =
                                    Node::RED;
                RRotate(z->parent->parent);
            }
        } else {
            y = z->parent->parent->left;
```

```
            if (y->color == Node::RED)
            {
                z->parent->color = Node::BLACK;
                y->color = Node::BLACK;
                z->parent->parent->color =
                                    Node::RED;
                z = z->parent->parent;
            } else {
                if (z == z->parent->left)
                {
                    z = z->parent;
                    RRotate(z);
                }
                z->parent->color = Node::BLACK;
                z->parent->parent->color =
                                    Node::RED;
                LRotate(z->parent->parent);
            }
        }
    }
    root->color = Node::BLACK;
}

template<typename T, typename Less>
void RBT<T,Less>::RemoveFixup(
            typename RBT<T,Less>::Node*x)
{
    Node * w;
    while(x != root && x->color==Node::BLACK)
    {
        if (x == x->parent->left)
        {
            w = x->parent->right;
            if (w->color == Node::RED)
            {
                w->color = Node::BLACK;
                x->parent->color = Node::RED;
                LRotate(x->parent);
                w = x->parent->right;
            }
```

```
        if (w->left->color == Node::BLACK &&
            w->right->color == Node::BLACK)
        {
            w->color = Node::RED;
            x = x->parent;
        }
        else
        {
            if (w->right->color == Node::BLACK)
            {
                w->left->color = Node::BLACK;
                w->color = Node::RED;
                RRotate(w);
                w = x->parent->right;
            }
            w->color = x->parent->color;
            x->parent->color = Node::BLACK;
            w->right->color = Node::BLACK;
            LRotate(x->parent);
            x = root;
        }
    }
    else
    {
        w = x->parent->left;
        if (w->color == Node::RED)
        {
            w->color = Node::BLACK;
            x->parent->color = Node::RED;
            RRotate(x->parent);
            w = x->parent->left;
        }
        if (w->right->color == Node::BLACK &&
            w->left->color == Node::BLACK)
        {
            w->color = Node::RED;
            x = x->parent;
        }
        else
        {
```

```
                if (w->left->color == Node::BLACK)
                {
                    w->right->color = Node::BLACK;
                    w->color = Node::RED;
                    LRotate(w);
                    w = x->parent->left;
                }
                w->color = x->parent->color;
                x->parent->color = Node::BLACK;
                w->left->color = Node::BLACK;
                RRotate(x->parent);
                x = root;
            }
        }
    }
    x->color = Node::BLACK;
}

template<typename T, typename Less>
void RBT<T,Less>::Destroy(
            typename RBT<T,Less>::Node * x)
{
    if (x == nil) return;
    Destroy(x->left);
    Destroy(x->right);
    delete x;
}
```

При последовательной вставке в дерево **RBT<int>** значений 9, 10, 3, 2, 6, 7, 5, 8, 11, 1, 4, 12, 13, 15, 14, 16 мы получим красно-черное дерево, изображенное на рис. 1.9, *а*. Обычное бинарное дерево для данной входной последовательности приведено на рис. 1.9, *б*.

Пирамиды

Пирамида — это структура данных, представляющая собой массив, который можно рассматривать как почти полное бинарное дерево. Каждый узел этого дерева соответствует определенному элементу массива, причем на всех уровнях, кроме (возможно) последнего, дерево полностью заполнено. Последний уровень дерева заполняет-

ся слева направо до тех пор, пока в массиве не закончатся элементы. В корне дерева находится первый элемент массива, а остальные элементы подчиняются следующему принципу: если какому-то узлу соответствует индекс i, то индекс его родительского узла — $\lfloor i/2 \rfloor$, индекс левого дочернего узла — $2i$, а правого дочернего узла — $2i+1$.

Имеется два вида пирамид — неубывающие и невозрастающие. В пирамидах обоих типов значения, расположенные в узлах, удовлетворяют *свойству пирамиды*. Свойство невозрастающих пирамид заключается в том, что для каждого отличного от корневого узла с индексом i его значение не превышает значение родительского по отношению к нему узла. Принцип неубывающей пирамиды прямо противоположный. Здесь мы рассмотрим только невозрастающие пирамиды (далее, говоря о пирамидах, мы имеем в виду именно их); преобразование алгоритмов для невозрастающих пирамид в алгоритмы для неубывающих пирамид — задача тривиальная.

Основные процедуры, связанные с пирамидами, — это создание пирамиды, поддержка ее свойства, вставка нового значения в пирамиду, извлечение максимального значения и увеличение значения элемента. Все они, за исключением создания пирамиды, выполняются за время $O(\log n)$. Создание пирамиды требует времени $O(n)$.

Поддержка свойства пирамиды осуществляется процедурой *Heapify*, на вход которой подается массив и индекс i в этом массиве. При вызове процедуры предполагается, что бинарные деревья, корнями которых являются узлы, дочерние по отношению к узлу i, являются пирамидами. Процедура опускает значение i-го элемента вниз по пирамиде до тех пор, пока поддерево с корнем, отвечающим индексу i, не становится пирамидой.

```
Heapify(A,i)
// Входные данные:  Массив A[1..n], индекс i
// Выходные данные: Пирамида, в которой i-й элемент
//                    перемещается в положение, при
//                    котором для него выполняется
//                    свойство пирамиды
l = 2*i
r = 2*i+1
if l <= n and A[l]>A[i] then max = l; else max = i
if r <= n and A[r]>A[max] then max = r
if max ≠ i then begin
    swap(A[i],A[max])
```

```
    Heapify(A,max)
end
```

Процедура *Heapify* позволяет преобразовать массив $A[1..n]$ в невозрастающую пирамиду снизу вверх. Поскольку элементы подмассива $A[\lfloor n/2 \rfloor+1..n]$ являются листьями, их можно рассматривать как одноэлементные пирамиды. Процедура *MakeHeap* проходит по остальным узлам и для каждого из них вызывает процедуру *Heapify*.

```
MakeHeap(A)
// Входные данные:  массив A[1..n]
// Выходные данные: пирамида A
for i = ⌊n/2⌋ downto 1 do Heapify(A,i)
```

В соответствии со свойством пирамиды ее максимальный элемент находится в первом элементе. Извлечение его из пирамиды выполняется путем обмена первого элемента с последним, уменьшения размера пирамиды на 1 и выполнения процедуры *Heapify* для нового первого элемента. Данная процедура возвращает значение извлеченного элемента.

```
ExtractMax(A)
// Входные данные:  массив A[1..n]
// Выходные данные: пирамида A[1..n-1] с извлеченным
//                    максимальным элементом
max = A[1]
A[1] = A[n]
n = n - 1
Heapify(A,1)
return max
```

При увеличении значения элемента i процедура проходит путь от этого элемента до корня в поисках места для нового ключа. Если оказывается, что значение текущего элемента превышает значение родительского элемента, то происходит обмен значений, и процедура продолжает работу на более высоком уровне. В противном случае свойство пирамиды полностью восстановлено, и процедура завершает работу.

```
Increase(A,i,value)
// Входные данные:  пирамида A[1..n], индекс i
//                    элемента, новое значение
//                    которого становится равным
```

```
//                    value (Внимание! Проверка
//                    того, что значение не
//                    уменьшается, процедурой не
//                    производится - ответственность
//                    за это лежит на вызывающей
//                    процедуре)
// Выходные данные: пирамида A[1..n] с увеличенным
//                    элементом
A[i] = value
while i > 1 and A[⌊i/2⌋] < A[i] do begin
    swap(A[i],A[⌊i/2⌋])
    i = ⌊i/2⌋
end
```

Вставка нового элемента в пирамиду выполняется предельно просто — для этого достаточно вставить в пирамиду новый лист и «увеличить» его значение до требуемого.

```
Insert(A,value)
// Входные данные:  пирамида A[1..n] и новый элемент
//                    value, вставляемый в пирамиду с
//                    сохранением ее свойства
// Выходные данные: пирамида A[1..n+1] со
//                    вставленным в нее элементом
n = n+1
A[n] = value
Increase(A,n,value)
```

Далее приведен пример реализации пирамиды (само собой, она может быть реализована и совершенно иначе; это лишь один из возможных вариантов ее представления). Следует отметить, что это единственный случай в книге, когда исходный текст компилируется Visual C++, но не Open Watcom, который не поддерживает шаблонные функции в шаблонах классов.

```
template<typename T,
    typename Comparator = less<T> >
class Heap
{
public:
    typedef typename vector<T>::size_type Int;
```

```
    template <typename Iter> Heap(Iter b, Iter e,
                Comparator comp = Comparator());
    ~Heap() {};
    void Heapify(Int i);
    Int Parent(Int i) const { return (i-1)/2; }
    Int Left(Int i)   const { return 2*i+1; }
    Int Right(Int i)  const { return 2*i+2; }
    T Max() const { assert(heapsize > 0);
                    return V[0]; }
    T ExtractMax();
    void Increase(Int i, T value);
    void Insert(T value);
    Int size() const { return heapsize; }
private:
    Comparator c;
    Int heapsize;
    vector<T> V;
};

template<typename T, typename Comparator>
template<typename Iter>
Heap<T,Comparator>::Heap(Iter b, Iter e,
                         Comparator comp)
:c(comp)
{
    heapsize = e - b;
    V.resize(heapsize);
    copy(b,e,V.begin());
    for(Int i = heapsize/2; i >= 1; --i)
    {
        Heapify(i-1);
    }
}

template<typename T, typename Comparator>
T Heap<T,Comparator>::ExtractMax()
{
    assert(heapsize > 0);
    T max = V[0];
```

```
    V[0] = V[--heapsize];
    Heapify(0);
    return max;
}

template<typename T, typename Comparator>
void Heap<T,Comparator>::Heapify(Int i)
{
    Int l = Left(i);
    Int r = Right(i);
    Int largest;
    largest =
        (l < heapsize && c(V[i],V[l])) ? l : i;
    if (r < heapsize && c(V[largest],V[r]))
        largest = r;
    if (largest != i)
    {
        swap(V[i],V[largest]);
        Heapify(largest);
    }
}

template<typename T, typename Comparator>
void Heap<T,Comparator>::Increase(Int i, T value)
{
    assert(i >= 0 && i < heapsize && V[i] <= value);
    V[i] = value;
    while(i > 0 && V[Parent(i)] < V[i])
    {
        swap(V[i],V[Parent(i)]);
        i = Parent(i);
    }
}

template<typename T, typename Comparator>
void Heap<T,Comparator>::Insert(T value)
{
    if (heapsize < V.size()) V[heapsize++] = value;
    else {
```

```
        V.push_back(value);
        ++heapsize;
    }
    Increase(heapsize-1,value);
```

Для простоты пирамида строится не в том же контейнере, где содержатся исходные данные, а копируется во внутренний массив. В принципе, этого можно избежать, однако это приведет к определенному усложнению исходного текста. Поскольку данная реализация, несмотря на ее полную функциональность, все же носит исключительно демонстрационный характер, было решено оставить такую реализацию с передачей контейнера читателям в качестве самостоятельной работы. Подобная задача решается позже, в разделе, посвященном пирамидальной сортировке, которая, наряду с реализацией очередей с приоритетами, составляет основное применение пирамид.

Глава 2

Сортировка и поиск

В этой главе мы рассмотрим решение задач, встающих перед программистами, пожалуй, наиболее часто — это задачи сортировки и поиска. Данные задачи применяются как сами по себе, так и входят в состав более сложных задач. Представим, например, что нам дан массив некоторых упорядочиваемых элементов, из которого надо удалить все дублирующиеся элементы. Решение сравнения каждого элемента с остальными потребует $O(n^2)$ времени. Однако если предварительно отсортировать массив (на что, как вы увидите далее, требуется $O(n\log n)$ времени), то найти все дубли можно за $O(n)$ времени, сравнивая только соседние элементы, так что общее время решения задачи — $O(n\log n)$. Здесь задача сортировки вошла в другую задачу в качестве подзадачи.

Как более сложный пример можно рассмотреть задачу поиска в большом тексте самой длинной повторяющейся подстроки. Очевидное решение сравнения всех подстрок требует $O(n^2)$ времени. Однако если использовать массив указателей на подстроки (т.е. на каждый символ текста) и отсортировать его по значению указываемых подстрок (не трогая сам текст), то понятно, что после этого нам достаточно просканировать полученный массив, сравнивая соседние элементы для выявления повторяющейся подстроки максимальной длины. Здесь сортировка вновь выполняет роль подзадачи, способной снизить время решения задачи до $O(n\log n)$ (с использованием $\Theta(n)$ дополнительной памяти).

Сортировка

Задача сортировки формулируется следующим образом. На вход алгоритма подается последовательность из n элементов $a_1, a_2, ..., a_n$; на выходе требуется получить некоторую перестановку входной последовательности $a'_1, a'_2, ..., a'_n$ такую, что $a'_1 \le a'_2 \le ... \le a'_n$.

Алгоритмы сортировки можно разделить на алгоритмы внутренней сортировки — для сортировки данных, хранящихся во внутренней,

оперативной памяти компьютера, и внешней сортировки — для сортировки больших объемов данных, хранящихся в файлах внешней (например, дисковой) памяти. Здесь будут рассматриваться только алгоритмы внутренней сортировки; о внешней сортировке можно прочесть, например, в [10].

Сортировка вставкой

Сортировка вставкой представляет собой один из тривиальнейших алгоритмов сортировки, напоминающий действия игрока, раскладывающего карты по возрастанию. Вся последовательность делится на две части — содержащую уже отсортированные элементы (изначально содержит только первый элемент последовательности) и еще неотсортированную последовательность. Элементы из второй части поочередно вставляются в первую часть так, чтобы поддерживалось свойство отсортированности первой части.

Псевдокод алгоритма сортировки вставкой выглядит следующим образом.

```
Алгоритм InsertionSort(A[1..n])
// Входные данные: массив A[1..n]
// Выходные данные: массив, элементы которого
//                  отсортированы
for i = 2 to n do begin
    Вставка элемента A[i] в последовательность
    A[1..i-1]; при наличии элементов с данным
    значением в отсортированной последовательности
    элемент A[i] вставляется после них
end
```

Временная эффективность данного алгоритма — $O(n^2)$. Сортировка выполняется на месте, без привлечения дополнительной памяти. Алгоритм сортировки вставкой в данной реализации *устойчив*, т.е. относительный порядок одинаковых элементов в отсортированном массиве сохраняется.

Вот как выглядит пример работы алгоритма (полужирным курсивом указана уже отсортированная часть массива, а символ | разделяет части).

```
3 | 4   1   5   2
3   4 | 1   5   2
1   3   4 | 5   2
1   3   4   5 | 2
1   2   3   4   5
```

Вставка элемента при реализации алгоритма может осуществляться различными способами — например, проверкой всех значений отсортированной части массива, начиная с первого элемента, но проверка с конца отсортированной части массива является более предпочтительной в случае, если массив почти отсортирован. Сама вставка также может осуществляться разными способами. Наиболее обобщенный способ состоит в выполнении ряда обменов элементов местами с использованием стандартного алгоритма **swap**. Используемые итераторы — двунаправленные, так что эта сортировка вполне применима, например, для связанного списка.

```
template<typename Iter, typename Less>
void InsertionSort(Iter b, Iter e, Less c)
{
    Iter i = b;
    for(++i; i != e; ++i)
    {
        Iter j = i, k = i;
        for(--j; (k != b) && c(*k,*j); --j, --k)
            swap(*k,*j);
    }
}
```

Естественным улучшением указанной реализации алгоритма является отказ от функции **swap**. При этом значение вставляемого элемента сохраняется в дополнительной переменной, и все элементы из отсортированной части, которые не меньше вставляемого, сдвигаются, после чего сохраненный элемент вставляется на место последнего сдвинутого.

```
template<typename Iter, typename Less>
void InsertionSort(Iter b, Iter e, Less c)
{
    typedef typename
            iterator_traits<Iter>::value_type Value;
    Iter i = b;
```

```
    for(++i; i != e; ++i)
    {
        Value x = *i;
        Iter j = i, k = i;
        for(--j; (k != b) && c(x,*j); --j, --k)
            *k = *j;
        *k = x;
    }
}
```

Сортировка выбором

Сортировка выбором также состоит в разделении исходного массива на две части, однако в этом случае отсортированная часть изначально пустая. На каждой итерации алгоритма в неотсортированной части находится наименьший элемент, после чего он обменивается местами с первым элементом неотсортированной части, и этот первый элемент присоединяется к отсортированной части массива. Очевидно, что данный алгоритм сортировки устойчивым не является.

Псевдокод алгоритма сортировки выбором выглядит следующим образом.

```
Алгоритм SelectionSort(A[1..n])
// Входные данные: массив A[1..n]
// Выходные данные: массив, элементы которого
//                  отсортированы
for i = 1 to n-1 do begin
    k = индекс наименьшего элемента из A[i..n]
    Обменять местами A[i] и A[k]
end
```

Временная эффективность данного алгоритма — $\Theta(n^2)$. Сортировка выполняется на месте, без привлечения дополнительной памяти.

Вот как выглядит пример работы алгоритма (полужирным курсивом указана уже отсортированная часть массива, а символ | разделяет части).

```
3    4    1    5    2
1  | 4    3    5    2
1    2  | 3    5    4
1    2    3  | 5    4
1    2    3    4  | 5
```

Реализация данного алгоритма достаточно проста. В нем, как и при сортировке вставкой, используются двунаправленные итераторы. Единственным более-менее тонким местом является применение в качестве границы между частями передаваемого функции итератора, указывающего начало сортируемого диапазона. Такое использование вполне оправданно, поскольку на каждой итерации элементы отсортированной части массива занимают свои окончательные места и больше не перемещаются.

```
template<typename Iter, typename Less>
void SelectionSort(Iter b, Iter e, Less c)
{
    for(; b!= e; ++b)
    {
        Iter mini = b, i = b;
        for(++i; i != e; ++i)
            if (c(*i,*mini)) mini = i;
        if (b != mini) swap(*b,*mini);
    }
}
```

Пузырьковая и шейкерная сортировки

Еще один алгоритм сортировки основан на выполнении проходов от последнего элемента массива к первому, причем при каждом проходе каждые два соседних элемента, оказывающиеся неупорядоченными, меняются местами. Таким образом при каждом проходе малые элементы продвигаются к левому концу массива, как легкие пузырьки, поднимающиеся вверх, — откуда и произошло название метода *пузырьковой сортировки*. Очевидно, что при первой итерации наименьший элемент окажется в первой позиции, после второй — второй наименьший элемент окажется во второй позиции и т.д. Псевдокод описанного алгоритма пузырьковой сортировки имеет следующий вид.

```
Алгоритм BubbleSort(A[1..n])
// Входные данные:  массив A[1..n]
// Выходные данные: массив, элементы которого
//                  отсортированы
for i = 2 to n do begin
```

```
    for j = n to i by -1 do begin
        if A[j] < A[j-1] then begin
            swap(A[j],A[j-1])
        end
    end
end
```

Временная эффективность данного алгоритма — $O(n^2)$. Сортировка выполняется на месте, без привлечения дополнительной памяти. Алгоритм сортировки вставкой в данной реализации устойчив, т.е. относительный порядок одинаковых элементов в отсортированном массиве сохраняется.

Вот как выглядит пример работы алгоритма (полужирным курсивом указаны всплывшие «пузырьки»):

```
4     3     1     5     2
1     4     3     2     5
1     2     4     3     5
1     2     3     4     5
```

Реализация описанного алгоритма тривиальна. Как и в ранее рассмотренных алгоритмах сортировки, здесь используются двунаправленные итераторы.

```
template<typename Iter, typename Less>
void BubleSort(Iter b, Iter e, Less c)
{
    --e;
    for(; b != e; ++b)
        for(Iter k = e, j = k--; j != b; j = k--)
            if (c(*j,*k)) swap(*j,*k);
}
```

Однако при более внимательном рассмотрении алгоритма становится очевидным, что если при какой-то итерации перестановки не выполнялись, значит, массив находится в отсортированном состоянии, и выполнение алгоритма следует прекратить. Кроме того, если последний обмен произошел в некоторой позиции k, то массив слева от этой позиции уже отсортирован и подвергаться изменениям не будет, так что при очередной итерации можно ограничиться проходом только до позиции последнего обмена на предыдущей итерации. Эти

замечания приводят к реализации улучшенной пузырьковой сортировки, но никак не влияют на ее временную эффективность.

```
template<typename Iter, typename Less>
void EnhBubleSort(Iter b, Iter e, Less c)
{
    --e;
    for(; b != e; ++b)
    {
        Iter last = e;
        for(Iter k = e, j = k--; j != b; j = k--)
            if (c(*j,*k))
            {
                swap(*j,*k);
                last = k;
            }
        if (last == e) break; else b = last;
    }
}
```

Следующее замечание по поводу пузырьковой сортировки связано с тем наблюдением, что малый элемент в конце массива проходит к началу массива очень быстро, в то время как большой элемент в начале массива может перемещаться в его конец со скоростью не более чем на одну позицию за итерацию. Это наблюдение приводит к разработке так называемой *шейкерной сортировки*, которая представляет собой поочередные итерации пузырьковой сортировки в одну и другую сторону. Реализация шейкерной сортировки представляет собой, по сути, две последовательные итерации, реализованные в пузырьковой сортировке, причем у второй из них попросту изменено направление прохода.

```
template<typename Iter, typename Less>
void ShakerSort(Iter b, Iter e, Less c)
{
    Iter last = --e;
    for(;b != e;)
    {
        for(Iter k = e, j = k--; j != b; j = k--)
        {
            if (c(*j,*k))
```

```
            {
                swap(*j,*k);
                last = k;
            }
        }
        if (last == e) break; else b = last;
        for(Iter k = b, j = k++; j != e; j = k++)
        {
            if(c(*k,*j))
            {
                swap(*k,*j);
                last = k;
            }
        }
        if (last == b) break; else e = last;
    }
}
```

Быстрая сортировка

Быстрая сортировка, разработанная Чарльзом Хоаром в 1962 году, представляет собой рекурсивный алгоритм, основанный на принципе декомпозиции. Исходный массив разбивается на две части таким образом, чтобы все элементы одной части были меньше некоторого значения, а второй — больше. Затем выполняется рекурсивная сортировка полученных частей. Если такая разбивка может быть произведена быстро, то алгоритм будет иметь высокую временну*ю* эффективность. Вот псевдокод алгоритма быстрой сортировки.

```
Алгоритм QuickSort(A[1..n],p,r)
// Входные данные:  массив A[1..n], индексы p и r,
//                    определяющие диапазон
//                    сортируемых элементов
// Выходные данные: массив, элементы которого в
//                    диапазоне от p до r
//                    отсортированы
if p < r then begin
    q = Partition(A,p,r)
    QuickSort(A,p,q-1)
    QuickSort(A,q+1,r)
end
```

Алгоритмом используется вспомогательная процедура *Partition*, которая разделяет массив на два подмассива с описанными выше свойствами:

```
Partition(A,p,r)
x = A[r] // Возможен другой алгоритм выбора опорного
         // элемента, относительно которого
         // выполняется разделение массива
repeat
    while A[p] < x do p = p+1;
    while x < A[r] do r = r-1;
    if p <= r then begin
        swap(A[p],A[r])
        p = p + 1
        r = r - 1
    end;
until p > r;
return r;
```

Временная эффективность данного алгоритма в среднем случае — $O(n\log n)$, в наихудшем — $O(n^2)$. Сортировка выполняется на месте, без привлечения дополнительной памяти. Алгоритм быстрой сортировки неустойчив.

Ключевым моментом алгоритма является выбор опорного элемента, относительно которого происходит разбиение диапазона массива на две части. Идеален такой выбор элемента, когда диапазон разбивается на два равных поддиапазона, однако, естественно, в общем случае такой выбор нереален. Сам Хоар предполагал, что опорный элемент следует выбирать случайным образом. Интересно отметить, что при использовании в качестве опорного крайнего (слева или справа) элемента наихудшим является случай отсортированного массива, поскольку диапазон из n элементов при этом разделяется на диапазон из $n-1$ элементов и опорный элемент в качестве второго поддиапазона. Нетрудно подсчитать, что количество выполненных сравнений в этом случае пропорционально n^2.

Реализация указанного алгоритма требует использования итераторов с произвольным доступом.

```
template<typename RandIter, typename Less>
void QuickSort(RandIter b, RandIter e, Less c)
{
    typedef typename
        iterator_traits<RandIter>::value_type Value;
    RandIter i = b;
    RandIter j = e;
    --j;
    Value x = *i; // Рандомизированный вариант:
                  // Value x = *(i + rand()%(j-i+1));
    do
    {
        while(c(*i,x)) ++i;
        while(c(x,*j)) --j;
        if (j >= i)
        {
            swap(*i,*j);
            ++i;
            --j;
        }
    } while(j >= i);
    if (j > b)   QuickSort(b,j+1,c);
    if (e > i+1) QuickSort(i,e,c);
}
```

Сортировка слиянием

Все рассмотренные до сих пор алгоритмы сортировки имели эффективность $O(n^2)$, по крайней мере в наихудшем случае. Алгоритм *сортировки слиянием* обеспечивает эффективность $\Theta(n \log n)$ в любом случае, достигая, таким образом, теоретического предела эффективности сортировки, основанной на операции сравнения. Сортировка слиянием также основана на декомпозиции — массив разбивается на две примерно равные части, и после их рекурсивной сортировки выполняется операция слияния, которая имеет эффективность $O(n)$. Операция слияния двух отсортированных частей в одну состоит в том, что из каждой части выбирается по одному элементу, и меньший из них помещается в результирующий массив. Так продолжается до тех пор, пока не будет исчерпана одна из частей — в этом случае оставшаяся часть просто переносится в конец результирующего массива.

Ниже проиллюстрирован процесс слияния отсортированных последовательностей 1, 3, 7, 8 и 2, 4, 5, 6. Результирующая последовательность выделена полужирным шрифтом, а очередной выбираемый элемент — полужирным подчеркнутым курсивом.

1 2 3 ***2*** **1** ***3*** 4 **1** 7 ***4*** **1** 7 ***5*** **1** 7 ***6*** **1** ***7*** **1** **1**
3 4 7 4 7 5 **2** 8 5 **2** 8 6 **2** 8 **2** ***8*** **2** **2**
7 5 8 5 8 6 6 **3** **3** **3** **3**
8 6 6 **4** **4** **4** **4**
5 **5** **5**
6 **6**
7
8

Сначала рассмотрим псевдокод процесса слияния.

```
Алгоритм Merge(A[1..n],p,q,r)
// Входные данные: массив A[1..n], для двух частей
//                 которого между индексами p и q
//                 и q и r выполняется слияние
// Выходные данные: массив, элементы которого в
//                 диапазоне от p до r отсортирова-
ны
n = q - p + 1
m = r - q
Создаем массивы L[1..n] и R[1..m]
for i = 1 to n do L[i] = A[p+i-1]
for i = 1 to m do R[i] = A[q+i]
i = 1
j = 1
for k = p to r do begin
    if i = n+1 then begin
        A[k] = R[j];
        j = j+1
    end
    else if j = m+1 then begin
        A[k] = L[i]
        i = i+1
    end
    else if L[i] < R[j] then begin
```

```
        A[k] = L[i]
        i = i+1
    end
    else begin
        A[k] = R[j];
        j = j+1
    end
end
```

С учетом наличия процедуры слияния псевдокод сортировки слиянием выглядит очень просто:

```
Алгоритм MergeSort(A[1..n],p,r)
// Входные данные: массив A[1..n], индексы p и r
// Выходные данные: массив, элементы которого в
//                  диапазоне от p до r отсортированы
if p < r then begin
    q = (p+r)/2
    MergeSort(A,p,q)
    MergeSort(A,q+1,r)
    Merge(A,p,q,r)
end
```

Временная эффективность сортировки слиянием — $\Theta(n\log n)$, однако для работы алгоритму требуется $\Theta(n)$ дополнительной памяти для размещения временных массивов.

Реализация указанного алгоритма требует использования итераторов с произвольным доступом.

```
template<typename RandIter, typename Less>
void MergeSort(RandIter b, RandIter e, Less c)
{
    if (b >= e - 1) return;
    typedef typename
        iterator_traits<RandIter>::value_type Value;
    RandIter i = b + (e - b)/2;
    MergeSort(b,i,c);
    MergeSort(i,e,c);
    vector<Value> L,R;
```

```
    for(RandIter j = b; j != i; ++j)
        L.push_back(*j);
    for(RandIter j = i; j != e; ++j)
        R.push_back(*j);
    RandIter l = L.begin(), r = R.begin();
    for(RandIter j = b; j != e; ++j)
    {
        *j =
            (l == L.end()) ? *r++ :
            (r == R.end()) ? *l++ :
            (c(*l,*r)) ? *l++ : *r++;
    }
}
```

Пирамидальная сортировка

Еще один алгоритм сортировки, обеспечивающий теоретическую временную эффективность, — *пирамидальная сортировка*. Пирамидальная сортировка использует *пирамиду*, о которой шла речь в главе 1, «Структуры данных», в разделе «Пирамиды» (стр. 56). Из свойств пирамиды следует, что ее максимальный элемент — первый, так что мы можем обменять его с последним элементом пирамиды, уменьшить ее размер и выполнить действия, необходимые для восстановления свойства пирамиды. Эти действия легко осуществить, поскольку после обмена первого и последнего элементов пирамиды, дочерние по отношению к корневому узлу, сохраняют свои свойства пирамид. Это свойство может быть нарушено только для корневого узла, что исправляется процедурой *Heapify*, рассматривавшейся ранее в разделе, посвященном пирамидам.

Таким образом, при проведении описанных итераций будет выполняться поэлементное построение упорядоченной последовательности в конце массива, начиная с наибольшего элемента в последней ячейке массива.

Псевдокод пирамидальной сортировки выглядит следующим образом.

```
Алгоритм HeapSort(A)
// Входные данные:  массив A[1..n]
// Выходные данные: массив, элементы которого
//                  отсортированы
```

```
MakeHeap(A)
for i = n downto 2 do begin
    swap(A[1],A[i])
    Уменьшить размер пирамиды на 1
    Heapify(A,1)
end
```

Здесь использованы процедуры создания пирамиды на основе массива *MakeHeap* и поддержания свойства пирамиды *Heapify* из раздела, посвященного пирамидам (стр. 56).

Временная эффективность сортировки слиянием — $O(n\log n)$, дополнительная память не используется. К сожалению, определенная сложность операций с пирамидой увеличивает постоянный множитель в $O(n\log n)$, так что в среднем случае пирамидальная сортировка проигрывает быстрой по эффективности.

Несмотря на то что в упомянутом разделе имеется реализация пирамиды, она, как уже упоминалось, носит демонстрационный характер и неэффективна. Поэтому здесь для повышения эффективности реализуется только одна операция над пирамидой, а именно — *Heapify*, представленная отдельной шаблонной функцией, а пирамида строится непосредственно на переданном контейнере. Вот как выглядит описанная реализация пирамидальной сортировки.

```
template<typename RandIter, typename Difftype,
         typename Less>
void heapify(RandIter first,
             RandIter last,
             Difftype index,
             Less c)
{
    typedef typename
    iterator_traits<RandIter>::difference_type Int;
    Int size = last - first;
    Int L = ( 2*index + 1 ); // Левый дочерний узел
    Int R = ( 2*index + 2 ); // Правый дочерний узел
    Int largest = index;     // Индекс большего
                             // дочернего узла
    // Сравниваем левый дочерний узел с элементом
    // с индексом index
```

```
    if( L < size && c(first[largest],first[L]) )
        largest = L;
    // Сравниваем правый дочерний узел с элементом с
    // полученным элементом
    if( R < size && c(first[largest],first[R]))
        largest = R;
    // Если больший элемент - дочерний узел,
    // обмениваем узлы и повторяем процедуру
    if( largest != index ) {
        swap( first[index], first[largest] );
        heapify(first, last, largest, c);
    }
}

template< class Less, class RandIter>
void HeapSort(RandIter first, RandIter last, Less c)
{
    // Построение пирамиды
    typedef typename
     iterator_traits<RandIter>::difference_type Int;
    Int size = last - first;
    Int index;
    // Последовательность менее чем из двух
    // элементов не сортируется
    if( size <= 1 ) return;
    // Требуется подправить только узлы
    for( index = size/2 - 1; index >= 0; --index )
    {
        heapify(first, last, index, c);
    }
    // Сортировка
    for( index = size - 1; index > 0; --index )
    {
        // Первый элемент уходит в конец
        swap(first[0], first[index]);
        --last;
        heapify(first, last, 0, c);
    }
}
```

Сортировка вставкой в бинарное дерево поиска

Эта сортировка использует свойство бинарного дерева поиска, которое рассматривалось в разделе «Бинарные деревья поиска» (стр. 24). Как говорилось ранее, бинарное дерево поиска обладает тем свойством, что его обход в симметричном порядке выдает все его элементы в отсортированном порядке, так что, разместив поступающие на вход алгоритма данные в бинарном дереве поиска (обычном, красно-черном дереве или некоторой иной его разновидности), мы затем можем просто выполнить обход всех узлов дерева и получить отсортированные данные.

Оценить временную эффективность такого алгоритма несложно — поскольку вставка в бинарное дерево поиска выполняется за время $O(\log n)$ (будем для определенности считать, что мы имеем дело с красно-черным деревом, в котором такая эффективность обеспечивается в наихудшем случае), весь массив данных можно разместить в дереве за время $O(n \log n)$. Обход построенного дерева выполняется за время $\Theta(n)$, так что общее время сортировки вставкой в бинарное дерево равно $O(n \log n)$.

Вся необходимая информация для составления псевдокода и реализации данного алгоритма представлена в разделе, посвященном бинарным деревьям поиска, так что здесь приведено только его краткое текстовое описание.

Сортировки за линейное время

Можно достаточно легко доказать, что любая сортировка, основанная на сравнении элементов входных последовательностей, требует в наихудшем случае $\Omega(n \log n)$ сравнений. Получить меньшее время работы при использовании сравнения элементов невозможно, но для ряда задач со специфическими данными это время может быть улучшено до линейного (подумайте сами, почему невозможно достичь времени работы меньше линейного).

Сортировка подсчетом

В *сортировке подсчетом* предполагается, что все n входных элементов — целые числа, принадлежащие интервалу от 0 до k, где k — некоторая целая константа. Если $k = O(n)$, то время работы алгоритма сортировки подсчетом равно $\Theta(n)$.

Основная идея сортировки подсчетом заключается в том, чтобы для каждого входного элемента x определить количество элементов, которые меньше x. С помощью этой информации элемент x можно разместить в той позиции выходного массива, где он должен находиться. Например, если всего имеется 10 элементов, которые меньше x, то в выходной последовательности элемент x должен занимать одиннадцатую позицию. Если допускается ситуация, когда несколько элементов имеют одно и то же значение, то эту схему придется слегка модифицировать, поскольку разместить все подобные элементы в одной и той же позиции нельзя.

Для сортировки массива $A[1..n]$ требуются два дополнительных массива — $B[1..n]$, в котором будет записана отсортированная последовательность, и временный массив $C[0..k]$.

```
Алгоритм CountingSort(A,B,k)
// Входные данные: Массив A[1..n], элементы
//                  которого представляют собой
//                  целые числа в диапазоне от 0 до k
// Выходные данные: Массив B[1..n], элементы которого
//                  отсортированы
// Начальное обнуление временного массива
for i = 0 to k do C[i] = 0
for j = 1 to n do C[A[j]] = C[A[j]] + 1
// В C[i] накапливается количество элементов,
// равных i
for i = 1 to k do C[i] = C[i] + C[i - 1]
// В C[i] находится количество элементов,
// не превышающих i
for j = n downto 1 do begin
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
end
```

Время работы алгоритма — $\Theta(n+k)$. Таково же и количество дополнительной памяти, необходимой для работы алгоритма. Эффективность алгоритма сортировки подсчетом больше, чем у любого из ранее рассмотренных алгоритмов, поскольку в нем не сравниваются элементы последовательности — вместо этого непосредственно используются их значения, на основании которых вычисляются конкретные индексы элементов.

Алгоритм сортировки сравнением устойчив — элементы с одним и тем же значением находятся в выходном массиве в том же порядке, что и во входном.

Что касается реализации сортировки подсчетом на C++, то в обобщенном варианте она не имеет особого смысла в силу своей специфичности и привязки к конкретному типу. Тем не менее приведенный ниже код позволяет сортировать последовательность элементов, задаваемую итераторами произвольного доступа **b** и **e**. Для каждого элемента последовательности функтор **IndexOf v** возвращает его целое значение в диапазоне от 0 до **MaxValue** — именно в соответствии с этими значениями выполняется сортировка элементов последовательности.

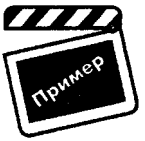
```
template<typename Iter, typename IndexOf>
void CountingSort(Iter b, Iter e, IndexOf v,
                  int MaxValue)
{
    typedef typename
        iterator_traits<Iter>::difference_type Int;
    typedef typename
        iterator_traits<Iter>::value_type Value;
    Int n = e - b;
    vector<Value> B;
    B.resize(n);
    vector<int> C;
    C.resize(MaxValue+1);
    for(Iter j = b; j !=e; ++j) ++C[v(*j)];
    for(int i = 1; i<=MaxValue; ++i) C[i]+=C[i-1];
    for(Iter j = e -1; j >= b; --j)
    {
        int i = v(*j);
        B[C[i]] = *j;
        --C[i];
    }
    copy(B.begin(),B.end(),b);
}
```

Сортировка, например, массива случайных целых чисел выполняется следующим образом.

```
inline int value(int x) { return x; }
...
int a[1000];
...
CountingSort(a,a+1000,value,500);
```

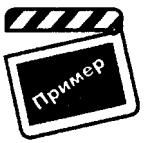
Поразрядная сортировка

С сортировкой подсчетом тесно связана *поразрядная сортировка* (в которой сортировка подсчетом используется в качестве подпрограммы). Поразрядная сортировка основана на том, что все числа сортируются при помощи устойчивой сортировки сначала по младшему разряду, затем по остальным разрядам в порядке их возрастания.



Вот пример поразрядной сортировки трехзначных чисел:

```
763             763             813             116
834             933             116             534
116             813             933             555
655 сортировка 834 сортировка 834 сортировка 655
534 по единицам 534 по десяткам 534 по сотням  763
933             655             655             813
813             555             555             834
555             116             763             933
```



Чтобы было понятно, что роль разрядов могут играть не только десятичные цифры, отсортируем таким же образом несколько дат:

```
09.05.1945              01.05.1886
30.12.1922              04.10.1957
05.12.1936              05.12.1936
01.05.1886    по дню    07.11.1917     по
12.04.1961   (младший   07.11.1917   месяцу
04.10.1957    разряд)   12.04.1961
07.11.1917              30.12.1922
```

```
12.04.1961                01.05.1886
01.05.1886                07.11.1917
09.05.1945                30.12.1922
04.10.1957       по       05.12.1936
07.11.1917       году     09.05.1945
05.12.1936                04.10.1957
30.12.1922                12.04.1961
```

Псевдокод поразрядной сортировки без преувеличения состоит из трех строк:

```
Алгоритм RadixSort(A,d)
// Входные данные: массив A, элементы которого
//                 представляют собой целые числа
//                 с d цифрами
// Выходные данные: массив A, элементы которого
//                  отсортированы
for i = 1 to d do begin
    Устойчивая сортировка массива A по i-й цифре,
      от младших к старшим
end
```

Временная эффективность поразрядной сортировки — $\Theta\left(d\left(n+k\right)\right)$, где d — количество разрядов, k — диапазон значений разряда и n — количество элементов в массиве.

В силу специфичности данного алгоритма обобщенная реализация на C++ не имеет смысла, поскольку будет крайне неэффективна. Поэтому в качестве примера приведем только код демонстрационной программы, которая поразрядно сортирует массив из 100 случайных трехзначных целых чисел, и использует в качестве устойчивой сортировки сортировку подсчетом.

```
inline int digit1(int x) { return x%10; }
inline int digit2(int x) { return (x/10)%10; }
inline int digit3(int x) { return x/100; }
int main()
{
    int a[100];
    for(int i = 0; i < 100; ++i) a[i] = rand()%1000;
    CountingSort(a,a+100,digit1,9);
    CountingSort(a,a+100,digit2,9);
    CountingSort(a,a+100,digit3,9);
```

```
    for(int i = 0; i < 100; ++i)
        cout << setw(4) << a[i];
}
```

На этом мы закончим рассмотрение алгоритмов сортировок в данной книге, тем более что тема сортировки упоминается практически в любой книге, посвященной алгоритмам. Особенно подробно сортировка рассмотрена в [10] и [12], а в [22] описание алгоритмов сортировки сопровождается как реализацией на языке программирования C++, так и массой конкретных примеров и увлекательных демонстрационных материалов.

Поиск

Поиск — также одно из часто встречающихся в программировании действий. Существует немало вариаций задачи поиска, но сейчас мы рассмотрим только поиск в фиксированной последовательности элементов. На вход алгоритма поиска подается последовательность из n элементов $a_1, a_2, ..., a_n$ и некоторый элемент b; задача состоит в том, чтобы получить на выходе индекс i элемента последовательности, равного заданному элементу b, или выяснить, что такого элемента в данной последовательности не существует.

Последовательный поиск

Тривиальным алгоритмом для решения поставленной задачи является *последовательный*, или *линейный*, *поиск*. Он заключается в том, что мы поочередно сравниваем b с элементами последовательности a_1, a_2, ..., a_n. Обнаружив совпадение, мы возвращаем индекс найденного элемента; если же по окончании сравнения b с элементами последовательности искомый элемент не найден, возвращается информация об этом (обычно для этого используется какое-либо значение, которое невозможно в качестве индекса — например, –1).

Алгоритм тривиален, и записать его псевдокод не составляет никакого труда:

```
Алгоритм SequentialSearch(A[1..n],b)
// Входные данные: массив A[1..n], элемент b
// Выходные данные: индекс элемента массива A,
//                  совпадающего с элементом b,
//                  или -1, если такой элемент
//                  не найден
```

```
i = 1
while i <= n and A[i] ≠ b do i = i + 1
if i <= n then return i
else return n
```

Очевидно, что при поиске элемента в массиве приходится проверить в среднем половину элементов массива в случае успешного поиска, и весь массив — в случае неудачного. В обоих случаях временная эффективность данного алгоритма — $O(n)$.

Реализация алгоритма последовательного поиска на C++ не более сложна, чем его псевдокод. Однако, прежде чем приступить к ней, заметим, что, поскольку последовательный поиск перебирает все элементы последовательности, его можно применять не только для того, чтобы найти элемент, равный заданному, но и для поиска элемента, отвечающего некоторому более общему условию. Соответственно, здесь приведены две реализации алгоритма последовательного поиска (здесь, как и в ряде прочих реализаций, приходится идти на определенные ухищрения, которые позволяют использовать представленный код как с компилятором Visual C++, так и с Open Watcom C++).

```
template<typename Iter, typename T>
Iter SeqSearch(Iter b, Iter e, const T& value)
{
    for (;b != e; ++b) if (*b == value) break;
    return b;
}

template<typename Iter, typename Predicate>
Iter SeqSearchIf(Iter b, Iter e, Predicate p)
{
    for (;b != e; ++b) if (p(*b)) break;
    return b;
}
```

Первая реализация ищет первый элемент, равный переданному, а вторая — первый элемент, для которого предикат **p** возвращает значение **true**. Как видно, единственные операции, которые выполняется с итератором в функции последовательного поиска, — это его инкремент и получение значения (разыменование), так что здесь может использоваться входной итератор, т.е. последовательный поиск может произво-

диться, например, среди элементов файла с помощью **istream_iterator** (о соотношениях разных типов итераторов см., например, [14]).

При неуспешном поиске возвращается итератор **e**, указывающий за границу диапазона, что согласуется с общепринятой практикой стандартной библиотеки C++.

Бинарный поиск

Если мы имеем дело с отсортированной последовательностью, то можем существенно уменьшить количество выполняемых проверок. Например, если мы узнаем, что искомый элемент меньше среднего элемента последовательности, то проверять все элементы от этого среднего и до конца отсортированной в порядке возрастания последовательности бессмысленно — все они не меньше среднего элемента, а значит, заведомо больше искомого. На этом наблюдении строится алгоритм *бинарного поиска*.

Вот его псевдокод.

```
Алгоритм BinarySearch(A[1..n],b)
// Входные данные:  упорядоченный в порядке
//                    возрастания массив A[1..n],
//                    элемент b
// Выходные данные: индекс элемента массива A,
//                    совпадающего с элементом b,
//                    или -1, если такой
//                    элемент не найден
l = 1, r = n
while l <= r do begin
    m = ⌊(l+r)/2⌋
    if b = A[m] then return m;
    else if b < A[m] then r = m - 1;
    else l = m + 1
end
return -1
```

Временная эффективность алгоритма бинарного поиска — $O(\log n)$.

Реализация алгоритма бинарного поиска на C++ тривиальна, она просто повторяет псевдокод.

```
template<typename Iter, typename T, typename Less >
Iter BinSearch(Iter b, Iter e,
               const T& value, Less c)
{
    Iter l = b, r = e - 1;
    while(l <= r)
    {
        Iter i = l + (r-l)/2;
        if (c(value,*i)) r = --i;
        else if (c(*i,value)) l = ++i;
        else return i;
    }
    return e;
}
```

Здесь есть только одна тонкость — в реализации на C++ изменен порядок проверок. Поскольку предикат **Less** дает значение **true** только тогда, когда его первый аргумент строго меньше второго, проверка условия равенства значений **x** и **y** выполняется как **!c(x,y)&&!c(y,x)**. Изменение порядка проверки позволяет уменьшить количество выполняемых сравнений при той же функциональности.

В реализации алгоритма не проверяется, действительно ли массив отсортирован, причем отсортирован с использованием той же функции сравнения, что и используемая при бинарном поиске.

Еще одно замечание по поводу последовательного и бинарного поиска. Если поиск в неотсортированном массиве выполняется разово, то, конечно же, легче воспользоваться последовательным поиском. Однако если в одном и том же массиве предполагается поиск большого количества элементов, то имеет смысл сначала отсортировать массив, а затем воспользоваться бинарным поиском. Если количество поисков — m, то использование последовательного поиска приводит ко времени работы $O(mn)$, в то время как использование сортировки с бинарным поиском — ко времени работы $O(n\log n)+O(m\log n)$, так что если $m=\Omega(\log n)$ (порядок роста m превышает $\log n$), то использование предварительной сортировки определенно имеет смысл. Тем более оправданно выполнение предварительной сортировки, если данные таковы, что их можно отсортировать за линейное время. Однако не следует забывать о том, что константа в асимптотической записи может оказаться весьма большой, так что сортировка оправдает себя только при больших размерах входных данных.

Интерполяционный поиск

Если в отсортированной последовательности выполняется поиск числовых значений, то можно воспользоваться интерполяционным поиском, который делит интервал поиска на неравные части в соответствии со значением искомого элемента. В качестве аналога можно привести пример поиска в телефонной книге — при поиске фамилии на букву «Б» вы не будете открывать справочник посредине, чтобы определить, в какой половине она находится, а откроете его существенно ближе к началу. Так и в случае поиска числового значения в отсортированной последовательности, принимая априори гипотезу линейного увеличения значения элемента с его номером, делить диапазон следует исходя из пропорции $(b-A[l])/(m-l)=(A[r]-A[l])/(r-l)$, т.е. строка $m=\lfloor (l+r)/2 \rfloor$ псевдокода заменяется строкой $m=(b-A[l])(r-l)/(A[r]-A[l])+l$, а выражение `Iter i = l + (r-l)/2;` в реализации алгоритма заменяется выражением `Iter i = (value-*l)*(r-l)/(*r-*l)+l;`. Теоретически *интерполяционный поиск* снижает количество делений диапазона в среднем до $\log_2 \log_2 N$, но при не слишком больших размерах данных выгода оказывается недостаточно велика, чтобы оправдать усложнение вычислений и увеличивающееся из-за этого время поиска.

Поиск в бинарном дереве поиска

Если нас интересует не просто поиск в заранее заданной последовательности элементов, а словарные операции с данными — т.е. когда мы размещаем данные таким образом, чтобы обеспечить быстрое выполнение основных операций над ними, а именно — вставки, удаления и поиска, — то неплохим решением является использование бинарного дерева поиска. Мы уже рассматривали бинарные деревья поиска ранее, в разделе «Бинарные деревья поиска» (стр. 24), поэтому здесь просто упоминается о такой возможности организации данных для последующего поиска. Эффективность поиска в бинарном дереве — $O(\log n)$, т.е. такая же, как и при бинарном поиске.

Отдельная интересная задача возникает, если поиск различных элементов в бинарном дереве поиска не равновероятен. Естественно, что, чем ближе к корню дерева будут находиться искомые элементы, тем меньшее количество сравнений придется выполнить для их поиска. Таким образом, если известны относительные частоты выполнения

поиска различных элементов в дереве, то можно построить дерево, при поиске в котором среднее количество сравнений будет минимально. Эта задача решается, например, методом динамического программирования (который вкратце рассматривается ниже, в разделе «Динамическое программирование» (стр. 213)). Решение задачи построения оптимального дерева поиска выходит за рамки данной книги, поэтому заинтересованному читателю рекомендуется обратиться к [12, раздел 15.5].

Описание отдельных операций над бинарным деревом поиска, их псевдокоды и реализация на C++ приведены в уже упоминавшемся разделе «Бинарные деревья поиска», поэтому во избежание дублирования здесь они не приводятся.

Хеширование

В этом разделе мы рассмотрим очень эффективный способ реализации словарей. Напомним, что *словарем* называется абстрактный тип данных, представляющий собой множество с операциями поиска, вставки и удаления, определенных над его элементами, которые обычно включают несколько полей. Среди полей записи имеется по крайней мере одно, именуемое ***ключом*** и используемое для идентификации элемента.

Хеширование основано на идее распределения ключей в одномерном массиве $H[0..m-1]$, называющемся хеш-таблицей. Распределение осуществляется путем вычисления для каждого ключа значения некоторой предопределенной хеш-функции h. Эта функция назначает каждому из ключей хеш-адрес, который представляет собой целое число от 0 до $m-1$. Например, если ключи представляют собой неотрицательные целые числа, то хеш-функция может иметь вид $h(K)=K \bmod m$ (ясно, что остаток от деления на m всегда находится в диапазоне от 0 до $m-1$). Если ключи — символы некоторого алфавита, то в качестве числовых значений символов можно использовать, например, их позиции в алфавите, а если это строки символов — то применять некоторую функцию от числовых значений символов, составляющих строку. В общем случае хеш-функция должна удовлетворять двум несколько противоречивым требованиям:

- распределять ключи по ячейкам хеш-таблицы как можно более равномерно (из-за этого требования m обычно выбирается простым. Это же требование делает желательной для большинства

приложений зависимость хеш-функции от всех битов ключа, а не только от некоторых из них);

- легко вычисляться.

Очевидно, что, если выбрать размер хеш-таблицы m меньше, чем количество ключей n, мы обречены на коллизии — ситуации, когда два (или несколько) ключей хешируются в одну и ту же ячейку хеш-таблицы, т.е. когда значения хеш-функции для разных элементов одинаковы. Коллизии могут наблюдаться независимо от соотношения размера таблицы и количества ключей; в наихудшем случае все ключи могут быть хешированы в одну ячейку хеш-таблицы. К счастью, при соответствующем выборе размера хеш-таблицы и хорошей хеш-функции такая ситуация встречается крайне редко. Тем не менее любая схема хеширования должна иметь механизм разрешения коллизий. Этот механизм различен в двух основных версиях — открытом хешировании (с раздельными цепочками) и закрытом хешировании (с открытой адресацией).

Открытое хеширование

При *открытом хешировании* ключи хранятся в связанных списках, присоединенных к ячейкам хеш-таблицы. Каждый список содержит все ключи, хешированные в данную ячейку (рис. 2.1).

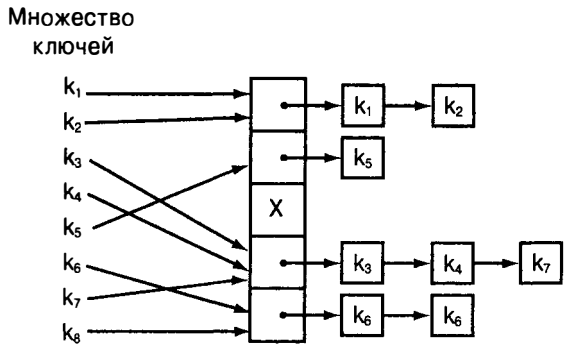


Рис. 2.1. *Разрешение коллизий с использованием цепочек*

Очевидно, что вставка элемента осуществляется за время $O(1)$ (если не выполнять проверку наличия данного элемента в таблице и вставлять новый элемент в начало списка); удаление элемента также выполняется за время $O(1)$ при использовании двусвязного списка. Для поиска элемента в таблице к ключу поиска применяется та же

функция, что и при создании таблицы. Значение функции указывает ячейку таблицы, а далее выполняется поиск искомого элемента в цепочке, связанной с данной ячейкой (последовательный поиск).

В общем случае эффективность поиска зависит от длины связанных списков, которая, в свою очередь, зависит от размеров словаря, таблицы и качества хеш-функции. Если хеш-функция распределяет n ключей по m ячейкам равномерно (или практически равномерно), то в каждом списке будет содержаться примерно около $\alpha = n/m$ ключей. Величина α называется *коэффициентом заполнения* хеш-таблицы и играет ключевую роль в эффективности хеширования. В частности, среднее количество проверяемых узлов цепочек при стандартных предположениях о поиске случайно выбранного элемента и хеш-функции, равномерно распределяющей ключи по ячейкам таблицы, для успешного поиска равно $1+\alpha/2$, а для неудачного — α. Полученные результаты, по сути, идентичны поиску в связанном списке: хеширование просто позволяет нам снизить размер связанного списка в m раз, заменив один большой список m меньшими.

Идеальное значение коэффициента заполнения — около 1. Слишком маленький коэффициент заполнения означает множество пустых ячеек и неэффективное использование памяти; слишком большой — длинные списки и продолжительный поиск. При значении коэффициента заполнения около 1 мы получаем наиболее эффективную схему, которая позволяет находить заданный ключ в среднем при помощи одного-двух сравнений. Само собой, при каждой операции вычисляется значение хеш-функции, но это операция с постоянным временем выполнения, не зависящим от n и m. Следует отметить, что высокая скорость достигается не только благодаря самому методу, но и ценой излишнего потребления памяти.

Закрытое хеширование

В случае *закрытого хеширования* все ключи хранятся в хеш-таблице без использования связанных списков (само собой, это приводит к требованию, чтобы размер хеш-таблицы m был не меньше количества ключей n). Для разрешения коллизий могут применяться разные стратегии. Простейшая из них — линейное исследование, когда в случае коллизии ячейки проверяются одна за другой. Если ячейка пуста, новый ключ вносится в нее; если заполнена — проверяется ячейка, следующая за ней. Если при проверке достигается конец таблицы, поиск переходит к первой ячейке таблицы, которая рассматривается как

циклический массив. На рис. 2.2 показано, как выглядит хеш-таблица при закрытом хешировании с линейным исследованием в случае вставки ключей с коллизиями.

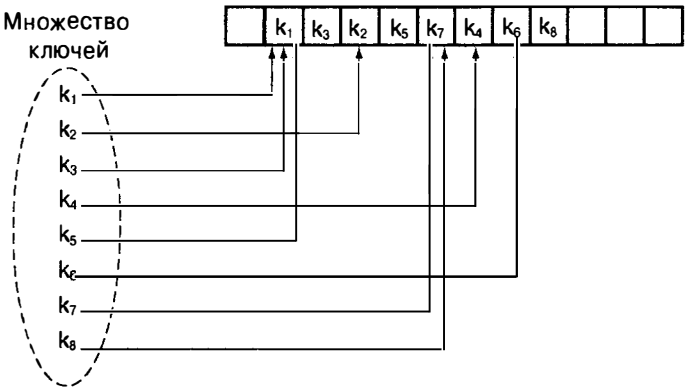


Рис. 2.2. *Закрытое хеширование с линейным исследованием. Порядок вставки ключей в таблицу — $k_1, k_2, ..., k_8$*

Поиск заданного ключа K мы начинаем с вычисления хеш-функции $h(K)$, использовавшейся при построении таблицы. Если ячейка $h(K)$ пуста, поиск неудачен. Если ячейка не пуста, мы сравниваем K с ключом, хранящимся в ячейке. Если они равны, то искомый ключ найден; если нет — то мы переходим к следующей ячейке и повторяем описанные действия до тех пор, пока не встретим искомый ключ (успешный поиск) или пустую ячейку (неудачный поиск).

В то время как операции поиска и вставки в такой версии хеширования очень просты, удаление оказывается очень сложным. Например, удалив ключ k_4 из таблицы на рис. 2.2, мы больше не сможем обнаружить в ней ключ k_8. Простейшим решением проблемы является «отложенное удаление», когда ранее занятая ячейка помечается специальным образом, чтобы можно было отличить ее от ячеек, которые никогда не были заняты.

Математический анализ линейного исследования — существенно более сложная задача, чем анализ хеширования с раздельными цепочками. Можно приближенно считать, что среднее количество обращений к хеш-таблице с коэффициентом заполнения α в случае успешного поиска равно $\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$, а для неудачного поиска —

$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$ (точность этого приближения увеличивается с ростом размера хеш-таблицы). Как нетрудно убедиться, получающиеся числа достаточно малы даже для плотно заполненных таблиц.

При разрешении коллизий методом цепочек у вас нет ограничения на количество элементов — просто большое их число ухудшит характеристики хеширования. Но при закрытом хешировании заполнение таблицы приводит к необходимости создания новой, большей таблицы. К сожалению, автоматический перенос данных из старой таблицы в новую невозможен, и требуется выполнение нового хеширования для каждого элемента таблицы.

По мере заполнения хеш-таблицы производительность линейного исследования снижается еще и из-за эффекта кластеризации. *Кластером* в линейном исследовании называется последовательность соседних заполненных ячеек (с возможным переходом из последней ячейки таблицы в первую). Например, в окончательном состоянии таблица на рис. 2.2 представляет собой один большой кластер. В хешировании кластеры представляют собой отрицательное явление, поскольку снижают эффективность словарных операций. Заметим также, что с ростом кластеров увеличивается вероятность того, что новый элемент будет добавлен к кластеру. Повышается и вероятность слияния кластеров при вставке нового ключа, что еще больше увеличивает кластеризацию.

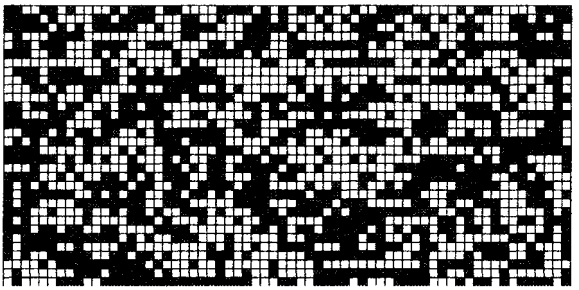
Для снижения эффекта кластеризации существует ряд стратегий разрешения коллизий. Наиболее простой — метод квадратичного исследования, когда при коллизии в ячейке $l=h(K)$ исследуются ячейки с номерами $(l+a_1i+a_2i^2) \bmod m$. Этот метод работает лучше линейного исследования, но требует подбора специальных значений параметров a_1 и a_2, а кроме того, подвержен эффекту вторичной кластеризации.

Одна из наиболее важных среди снижающих кластеризацию стратегий разрешения коллизий — двойное хеширование. В этой схеме для определения фиксированного значения шага последовательности исследований при коллизии в ячейке $l=h(K)$ используется другая хеш-функция $s(K)$, т.е. последовательно исследуются ячейки $(l+s(K)) \bmod m$, $(l+2s(K)) \bmod m$ и т.д. Для того чтобы такая последовательность исследования могла охватить всю таблицу, значение хеш-функции $s(K)$ должно быть взаимно простым с размером хеш-таблицы m. Один из вариантов обеспечения этого требования состоит в выборе m, равного степени двойки, а хеш-функция делается такой,

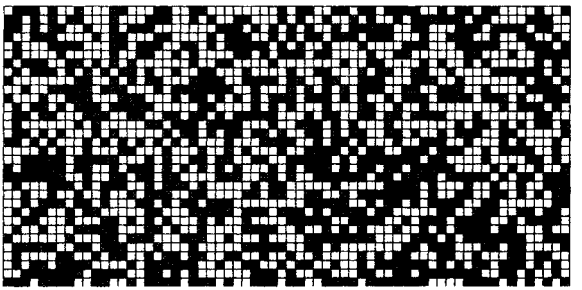
чтобы она всегда возвращала нечетные значения. Второй вариант состоит в выборе в качестве m простого числа, а хеш-функция всегда должна возвращать значения, которые меньше m.

Что касается выбора хеш-функции, то во многом это — искусство, в особенности для нечисловых данных (например, для строк рекомендуется использовать хеш-функцию, представляющую собой сумму по модулю m независимых хеш-функций для каждого символа строки). Для числовых значений, например, Кнут [10] рекомендует использовать функцию $h(K)=\left\lfloor M\left(\left(\frac{A}{w}K\right)\bmod 1\right)\right\rfloor$, где M представляет собой степень двойки, w — двойка в степени, равной количеству битов в машинном слове (для 32-битовых персональных компьютеров $w=2^{32}$), а целое число A Кнут предложил выбирать таким, чтобы $A/w\approx\left(\sqrt{5}-1\right)/2\approx 0{,}6180339887$. Операция mod 1 означает выделение дробной части числа, а $\lfloor x\rfloor$ -- наибольшее целое число, не превосходящее x (функция «пол»).

Обобщенная реализация хеширования достаточно громоздкая, поэтому вместо нее ниже приведен фрагмент эксперимента по сравнению закрытого хеширования с линейным исследованием и двойным хешированием. В коде создается таблица на 2048 элементов и используется заведомо плохая хеш-функция. Таблица наполовину заполняется случайными числами, после чего выполняется визуализация заполненности таблицы. Из рис. 2.3 видно, что линейное исследование характеризуется более выраженной кластеризацией.



а)



б)

Рис. 2.3. *Заполнение хеш-таблицы при а) линейном исследовании и б) двойном хешировании*

```
// Хеш-таблица на 2048 элементов
// У нас будут хешироваться только положительные
// числа, так что значение -1 говорит о том, что
// ячейка свободна, а -2 - о том, что из нее удален
// элемент (удаления в программе не используются)
int H[2048];

// Заведомо плохая хеш-функция
int h(int K)
{
    return (5*(K>>3)+7)%2048;
}

// Хеш-функция для двойного хеширования
int h2(int K)
{
    return K%97+1;
}

int main()
{
    // Количество заполняемых ячеек
    const int Count = 1024;
    // Инициализация таблицы
    for(int i = 0; i < 2048; ++i) H[i] = -1;
```

```
    // Заполняем случайными числами
    for(int i = 0; i < Count; ++i)
    {
        int value = rand();
        int j = h(value);
#if defined(LINEAR_PROBE)
        while(H[j] >= 0) j = (j+1)%2048;
#else
        int step = h2(value);
        while(H[j] >= 0) j = (j+step)%2048;
#endif
        H[j] = value;
    }
    // Визуализация - код данной функции
    // не приводится
    outPicture(H);
}
```

Поиск подстрок

Еще одна часто встречающаяся в программировании разновидность поиска — это поиск заданных подстрок в тексте.

Формально задачу поиска подстрок можно сформулировать следующим образом. Имеется некоторый текст длиной n символов, заданный в виде массива $T[1..n]$, и образец длиной $m \le n$ в виде массива $P[1..m]$. Если для некоторого значения $0 \le s \le n-m$ выполняется равенство $T[s+1..s+m] = P[1..m]$, т.е. если для всех $1 \le j \le m$ справедливо равенство $T[s+j] = P[j]$, то будем говорить, что образец входит в текст со сдвигом s. Задача поиска подстроки состоит в определении сдвига (первого или всех), с которым образец входит в текст (или установлении того факта, что данный образец в текст не входит).

Простейший алгоритм поиска состоит в непосредственной проверке всех возможных смещений. Проверка заключается в последовательном сравнении символов образца с символами текста; при первом же обнаруженном несовпадении символов проверка прекращается и сдвиг увеличивается на 1.

Псевдокод такого алгоритма можно записать следующим образом.

```
SimpleMatch(T,P)
// Входные данные: искомая подстрока P длиной m
//                 и текст T длиной n
// Выходные данные: сдвиги, с которыми образец
//                  входит в текст
for s = 0 to n-m do
begin
    for j = 1 to m do
    begin
        if P[j] ≠ T[s+j] then Выход из цикла
    end
    if j = m+1
        then print "Образец обнаружен со сдвигом ",s
end
```

Очевидно, что в наихудшем случае во внутреннем цикле приходится проверять все m символов, а внешний цикл выполняется $n-m+1$ раз, так что время работы всего алгоритма составляет $O(m(n-m+1))$.

Реализация такого алгоритма тривиальна. Для простоты будем считать, что мы имеем дело со строками C (состоящими из символов типа **char** и завершающимися нулевыми символами). Будем, кроме того, искать только первое вхождение образца в текст, а также опустим проверку корректности входных параметров (вообще говоря, следует убедиться в корректности передаваемых строк, что они не пусты и что образец короче текста). Еще одна особенность реализации такова: поскольку совпадение первого символа — ситуация достаточно редкая, сравнение первого символа образца вынесено в отдельный цикл.

```
char * strStr(char * text, char * pattern)
{
    // Ищем длины строк
    int lent = strlen(text);
    int lenp = strlen(pattern);
    // Первый символ образца не может быть ближе
    // к концу текста, чем длина образца
    int lenrange = lent - lenp + 1;
    for(int i = 0;;)
```

```
    {
        // Если не совпадает первый символ, можно не
        // продолжать
        while(i < lenrange &&
              text[i] != pattern[0]) ++i;
        if (i == lenrange) // Достигнут конец текста
            return 0;
        int k = i;          // Сохраняем сдвиг
        int j = 1; ++i;     // Первый символ проверен
        while (j < lenp && text[i] == pattern[j])
        {
            ++i;
            ++j;
        }
        if (j == lenp)
                return (char *)text + k;  // Найден
        i = k + 1;   // Не найден, продолжаем поиск
    }
}
```

Обратите внимание на то, что данный поиск никак не использует информацию, полученную при сравнении подстроки с текстом, — сколько бы символов не совпало, в любом случае сдвиг увеличивается только на единицу. При небольших размерах текста и строки простота такого поиска окупает себя, но при поиске длинных подстрок в больших массивах этот метод уступает другим, более эффективным.

Здесь мы рассмотрим только один из таких более эффективных методов, а именно алгоритм Хорспула, как один из наиболее простых. Однако это далеко не единственный эффективный алгоритм поиска подстрок. О других алгоритмах вы можете прочесть, например, в книгах [4] или [12].

Алгоритм Хорспула

Поиск в алгоритме Хорспула основан на необычном соображении — сравнение символов начинается не с начала образца, а с его конца. Рассмотрим в качестве примера поиск слова *АБРАКАДАБРА* в некотором тексте:

$$\overset{s_0}{\ldots}\ \ А\ Б\ Р\ А\ К\ А\ Д\ А\ Б\ Р\ \overset{c}{А}\ \ \overset{s_n}{\ldots}$$

Мы сравниваем пары символов из образца и текста, начиная с последнего символа *А* в образце и перемещаясь справа налево. Если все символы образца совпадают с символами текста, то искомая подстрока найдена (после этого поиск может либо завершиться, либо продолжиться, если требуется найти остальные вхождения подстроки в текст). Если же мы встретили несоответствие, то должны сдвинуть образец вправо, причем желательно как можно дальше, но, естественно, не пропустив при этом возможное вхождение подстроки в текст. Алгоритм Хорспула определяет величину такого сдвига, рассматривая символ *с* текста, который находится напротив последнего символа образца. В общем случае могут возникнуть четыре разные ситуации.

Ситуация 1. Если символа *с* в образце нет (например, если *с* в данном примере представляет собой символ *И*), то смело можно сдвигать образец на всю его длину (при сдвиге меньшей величины напротив символа *с* окажется некоторый символ образца, который заведомо не может быть таким же, как *с*, которого в образце нет):

```
s0 ...                И                    ... sn
                      ≠
   А Б Р А К А Д А Б Р А
               А Б Р А К А Д А Б Р А
```

Ситуация 2. Если символ *с* в образце есть, но он не последний (например, символ *Д* в нашем примере), то сдвиг должен выровнять образец так, чтобы напротив *с* в тексте было первое справа вхождение этого символа в образец:

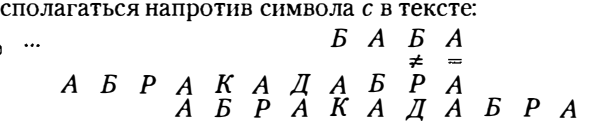
```
s0 ...                Д          ... sn
                      ≠
   А Б Р А К А Д А Б Р А
         А Б Р А К А Д А Б Р А
```

Ситуация 3. Если *с* — последний символ образца и среди остальных *m* – 1 символов образца такого символа больше нет, то сдвиг должен быть подобен сдвигу в ситуации 1 — образец следует сдвинуть на всю длину *m*:

```
s0 ...             Б А Р С                 ... sn
                   ≠ ≠ ≠
   А Б Р А К А Д А Б Р С
                  А Б Р А К А Д А Б Р С
```

Ситуация 4. И, наконец, если *с* — последний символ образца и среди остальных *m* – 1 символов образца имеются другие вхождения этого символа, то сдвиг должен быть подобен ситуации 2 — крайнее

справа вхождение c среди остальных $m-1$ символов образца должно располагаться напротив символа c в тексте:



Приведенные конкретные примеры показывают, что сравнение символов справа налево может привести к большим сдвигам, чем сдвиги на одну позицию в рассмотренном ранее простейшем алгоритме. Однако если такой алгоритм будет просматривать все символы образца при каждой проверке для выяснения, есть ли среди них искомый символ, то все его преимущество будет потеряно. Правда, можно предварительно вычислить величины сдвигов для всех символов алфавита и хранить их в таблице. Такая таблица индексируется всеми символами, которые могут встретиться в тексте (заметим, что для построения этой таблицы требуется информация только об образце, но не о тексте, в котором выполняется поиск). Элементы таблицы заполняются величинами сдвигов. В частности, для каждого символа c мы можем вычислить величину сдвига по формуле

$$t(c)=\begin{cases}\text{длина образца } m, \text{ если } c \text{ нет среди первых } m-1 \text{ символов образца;}\\ \text{в противном случае — расстояние от крайнего справа символа } c \text{ среди первых } m-1 \text{ символов образца до последнего символа образца.}\end{cases}$$

Так, для нашего образца *АБРАКАДАБРА* все элементы таблицы равны 11, кроме элементов для символов *А*, *Б*, *Р*, *К*, *Д*, для которых они равны соответственно 3, 2, 1, 6 и 4.

Далее приведен псевдокод простого алгоритма вычисления элементов таблицы сдвигов. Все значения в таблице инициализируются длиной образца m, а затем выполняется сканирование образца слева направо с повтором $m-1$ раз следующих действий: для j-го символа образца ($1 \le j \le m-1$) соответствующий ему элемент таблицы перезаписывается значением $m-j$, которое представляет собой расстояние от символа до правого конца образца. Заметим, что, поскольку алгоритм сканирует образец слева направо, последняя перезапись выполняется, когда встречается крайнее справа вхождение символа в образец, т.е. именно так, как и требуется.

```
ShiftTable(P[1..m])
// Входные данные: образец P[1..m] и алфавит
//                 возможных символов
// Выходные данные: таблица Table[1..size],
//                 индексированная символами
//                 алфавита и заполненная
//                 величинами сдвигов, вычисленных
//                 по приведенной выше формуле
// Инициализация всех элементов Table значениями m
for j = 1 to size do Table[j] = m
for j = 1 to m-1  do Table[P[j]] = m-j
return Table
```

Теперь пришло время для полного описания алгоритма Хорспула.

1. Для данного образца длиной m и алфавита, используемого в тексте и образце, описанным выше способом строим таблицу сдвигов.
2. Выравниваем начало образца с началом текста.
3. До тех пор, пока не будет найдена искомая подстрока или пока образец не достигнет последнего символа текста, повторяем следующие действия. Начиная с последнего символа образца, сравниваем соответствующие символы в шаблоне и тексте, пока не будет установлено равенство всех m символов (при этом поиск прекращается) либо пока не будет обнаружена пара разных символов. В последнем случае находим элемент $t(c)$ из таблицы сдвигов, где c — символ текста, находящийся напротив последнего символа образца, и сдвигаем образец вдоль текста на $t(c)$ символов вправо.

Вот как выглядит псевдокод алгоритма Хорспула.

```
Алгоритм HorspoolMatching(P[1..m], T[1..n])
// Входные данные: образец P[1..m] и текст T[1..n]
// Выходные данные: индекс левого конца первой
//                  найденной подстроки или 0, если
//                  искомой подстроки в тексте нет
ShiftTable(P[1..m]) // Генерация таблицы сдвигов
i = m               // Позиция правого конца образца
while i ≤ n do
begin
```

```
    k = 0                  // Количество совпадающих
                           // символов
    while k ≤ m and P[m-k] = T[i-k] do k = k + 1
    if k > m then return i - m
    else i = i + Table[T[i]]
end
return -1
```

А вот реальный поиск подстроки в тексте:

З А А Б Р А К А Д А Б Р И Л А С Ь _ А Б Р А К А Д А Б Р А
А Б Р А К А Д А Б Р А
А Б Р А К А Д А Б Р А
А Б Р А К А Д А Б Р А
А Б Р А К А Д А Б Р А

Эффективность алгоритма Хорспула в наихудшем случае составляет $\Theta(mn)$, но для случайных текстов его эффективность равна $\Theta(n)$. Затраты на построение таблицы сдвигов вполне окупаются, в особенности если поиск подстроки проводится неоднократно в разных текстах.

Реализация данного алгоритма использует не С-строки, а указатели **void***, чтобы расширить область применения поиска. Однако внутри функции используется представление данных как последовательности байтов. Заметим, что для других типов данных — например, поиска подпоследовательности чисел типа **int** в последовательности чисел **int** — алфавит оказывается чрезмерно большим, и практическое применение становится невозможным (так, для указанного поиска подпоследовательности целых 4-байтовых чисел требуется таблица общим размером 16 Гбайт).

```
vector<int> makeTable(void * p_, int len)
{
    assert(p_ && len > 0);
    unsigned char * p =
```

```
            static_cast<unsigned char *>(p_);
    vector<int> t;
    t.resize(numeric_limits<unsigned char>::max()+1,
             len);
    for(int i = 0; i < len - 1; ++i)
        t[p[i]] = len - i - 1;
    return t;
}

int HorspoolMatch(void * t_, int n, void * p_,
                  int m, const vector<int>& table)
{
    assert(t_ && p_ && n > 0 && m > 0);
    unsigned char * t =
        static_cast<unsigned char *>(t_);
    unsigned char * p =
        static_cast<unsigned char *>(p_);
    for(int i = m - 1; i < n; i += table[t[i]])
    {
        int k;
        for(k = 0; k < m && p[m-k-1] == t[i-k];
            ++k);
        if (k == m) return i - m;
    }
    return -1;
}
```

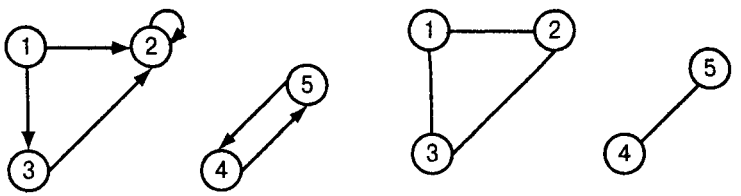
Глава 3

Графы

В этой главе будут кратко рассмотрены графы и некоторые основные алгоритмы для работы с ними.

Основные свойства графов

Ориентированный граф G определяется как пара (V,E), где V — конечное множество, а E — бинарное отношение на V. Множество V называется множеством вершин графа G, а его элементы — вершинами. Множество E называется множеством ребер графа G, а его элементы, соответственно, ребрами. На рис. 3.1, *а* изображен ориентированный граф с множеством вершин $\{1,2,3,4,5\}$. Вершины на рисунке показаны кружками, а ребра — стрелками. Обратите внимание на возможность существования петель — ребер, соединяющих вершину с самой собой.



а) *б)*

Рис. 3.1. *Примеры ориентированного и неориентированного графов*

В *неориентированном графе* $G=(V,E)$ множество ребер E состоит из неупорядоченных пар вершин, т.е. ребро является множеством $\{u,v\}$, где $u,v \in V$ и $u \neq v$. По соглашению для ребер используется запись (u,v). И (u,v), и (v,u) обозначают одно и то же ребро неориентированного графа, в то время как для ориентированного графа эти ребра различны. В неориентированном графе петли запрещены, так что каждое ребро содержит две разные вершины. На рис. 3.1, *б* показан неориентированный граф с множеством вершин $\{1,2,3,4,5\}$.

Многие определения выглядят одинаково и для ориентированных, и для неориентированных графов, хотя некоторые отличия, естественно, имеются. Если (u,v) — ребро ориентированного графа $G=(V,E)$, то ребро выходит из вершины u и входит в вершину v. Если (u,v) — ребро неориентированного графа $G=(V,E)$, то оно соединяет вершины u и v и называется *инцидентным* этим вершинам.

Если в графе G имеется ребро (u,v), то говорят, что вершина v смежна с вершиной u. Для неориентированных графов отношение смежности является симметричным; для ориентированных графов это утверждение неверно. Если вершина v смежна с вершиной u, это записывается как $u \to v$. На рис. 3.1, *а* и *б* вершина 2 смежна с вершинами 1 и 3, поскольку ребра $(1,2)$ и $(3,2)$ имеются в обоих графах; однако вершина 1 смежна с вершиной 2 только на рис. 3.1, *б*.

Степенью вершины в неориентированном графе называется количество ребер, соединяющих ее с другими вершинами. Вершина, степень которой равна 0, называется изолированной. В ориентированном графе различают исходящую степень, равную количеству выходящих из вершины ребер, и входящую степень, равную числу входящих в вершину ребер. Степень вершины в ориентированном графе равна сумме ее входящей и исходящей степеней.

Путь длины k от вершины u к вершине u' в графе $G=(V,E)$ представляет собой такую последовательность $\langle v_0,v_1,v_2,\ldots,v_k\rangle$ вершин, что $u=v_0$, $u'=v_k$ и $(v_{i-1},v_i)\in E$ для $i=1,2,\ldots,k$. Длиной пути называется количество составляющих его ребер. Путь содержит вершины $v_0,v_1,v_2,\ldots,v_k$ и ребра (v_0,v_1), (v_1,v_2), ..., (v_{k-1},v_k). Всегда существует путь нулевой длины из вершины в нее саму. Если имеется путь p из вершины u в вершину u', то говорят, что вершина u' достижима из u по пути p, что иногда в ориентированном графе G записывается как $u \overset{p}{\rightsquigarrow} u'$. Путь является простым, если все его вершины различны.

В ориентированном графе путь $\langle v_0,v_1,\ldots,v_k\rangle$ образует цикл, если $v_0=v_k$ и путь содержит по крайней мере одно ребро. Цикл называется простым, если, кроме того, все вершины $v_1,v_2,\ldots,v_k$ различны. Петля является циклом с длиной 1. Ориентированный граф, не содержащий петель, называется простым. Граф без циклов называется ациклическим.

Неориентированный граф является связным, если любая его вершина достижима из другой по некоторому пути. Для неориентированного графа отношение «быть достижимым из» является отношением

эквивалентности (т.е. оно рефлексивно, симметрично и транзитивно, как, например, отношение равенства между числами) на множестве вершин, а множества всех достижимых друг из друга вершин называются связными компонентами графа. Неориентированный граф связен тогда и только тогда, когда он состоит из единственного связного компонента.

Некоторые виды графов имеют свои специальные названия. Полным называется неориентированный граф, в котором каждая пара вершин являются смежными, т.е. который содержит все возможные ребра. Ациклический неориентированный граф называется лесом, а связный ациклический неориентированный граф — деревом.

Для представления графа в памяти компьютера обычно используется один из двух стандартных способов: как множества списков смежных вершин или в виде матрицы смежности. Оба способа применимы как для ориентированных, так и для неориентированных графов. Представление с помощью списков смежности обычно более предпочтительно, поскольку обеспечивает компактное представление разреженных графов, у которых количество ребер гораздо меньше квадрата количества вершин. Представление при помощи матрицы смежности предпочтительнее для плотных графов, когда количество ребер приближается к квадрату количества вершин или когда необходимо быстро определить, существует ли ребро, соединяющее две данные вершины.

Представление графа в виде списков смежности использует массив из $|V|$ списков, по одному для каждой вершины из V. Для каждой вершины такой список содержит все вершины, смежные с u в графе G (список может содержать и не сами вершины, а указатели на них). Вершины в каждом списке обычно хранятся в произвольном порядке.

На рис. 3.2, *а* показано представление ориентированного графа с рис. 3.1, *а* в виде списков смежности.

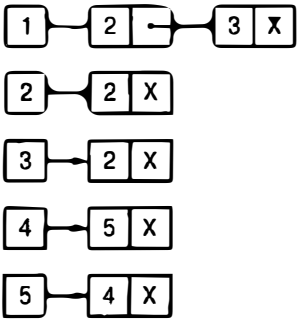
Как для ориентированных, так и для неориентированных графов представление в виде списков требует объема памяти, равного $\Theta(V+E)$. Главный недостаток представления при помощи списков смежности заключается в том, что при этом нет более быстрого способа определить, существует ли некоторое ребро (u,v) в графе, чем поиск v в списке смежности u.

Представление графа с помощью матрицы смежности предполагает, что вершины графа пронумерованы в некотором порядке числа-

ми $1,2,\ldots,|V|$. В этом случае представление графа G с использованием матрицы смежности представляет собой матрицу $A=(a_{ij})$ размером $|V|\times|V|$ такую, что

$$a_{ij}=\begin{cases}1, & \text{если } (i,j)\in E,\\ 0 & \text{в противном случае.}\end{cases}$$

На рис. 3.2, *б* показано представление ориентированного графа с рис. 3.1, *а* в виде матрицы смежности.



а)

	1	2	3	4	5
1	0	1	1	0	0
2	0	1	0	0	0
3	0	1	0	0	0
4	0	0	0	0	1
5	0	0	0	1	0

б)

Рис. 3.2. *Представления графа, показанного на рис. 3.1,* а

Матрица смежности графа требует объема памяти, равного $\Theta(V^2)$, независимо от количества ребер графа. Для неориентированного графа можно, воспользовавшись симметричностью отношения смежности, хранить в памяти только половину матрицы — поскольку матрица смежности для неориентированного графа симметрична.

Поиск в ширину

Поиск в ширину является одним из простейших алгоритмов для обхода графа и основой для многих других алгоритмов для работы с графами.

Пусть дан граф $G=(V,E)$ и выделена некоторая исходная вершина s. Алгоритм поиска в ширину систематически обходит все ребра G и «открывает» все вершины, достижимые из s, вычисляя при этом расстояние (минимальное количество ребер) от s к каждой достижимой

из s вершине. Кроме того, в процессе обхода строится «дерево поиска в ширину» с корнем s, содержащее все достижимые вершины. Для каждой достижимой из s вершины v путь в дереве поиска в ширину соответствует кратчайшему пути от s к v в G, т.е. пути, содержащему наименьшее количество ребер. Алгоритм применим как для ориентированных, так и для неориентированных графов.

Поиск в ширину называется так потому, что в процессе обхода, перед тем как приступить к поиску вершин на расстоянии $k+1$, выполняется обход всех вершин на расстоянии k.

Для отслеживания работы алгоритма поиск в ширину раскрашивает вершины графа в белый, серый и черный цвета. Изначально все вершины белые, а позже они могут стать серыми и черными. Когда в процессе поиска происходит открытие вершины, она окрашивается. Таким образом, серые и черные вершины — это вершины, которые уже были открыты, но алгоритм поиска в ширину по-разному работает с ними, чтобы обеспечить объявленный порядок обхода. Если $(u,v)\in E$ и вершина u черного цвета, то вершина v либо серая, либо черная, т.е. все вершины, смежные с черной, уже открыты. Серые вершины могут соседствовать с белыми, представляя собой границу между открытыми и неоткрытыми вершинами.

Поиск в ширину строит дерево поиска в ширину, которое изначально состоит из одного корня, которым является исходная вершина s. Если в процессе сканирования списка смежности уже открытой вершины u открывается белая вершина v, то вершина v и ребро (u,v) добавляются в дерево. В этом случае u является предшественником v в дереве поиска в ширину (родительским узлом). Поскольку вершина может быть открыта не более одного раза, она имеет не более одного родителя. Взаимоотношения предков и потомков определяются в дереве поиска в ширину как обычно — если u находится на пути от корня s к вершине v, то u является предком v, а v — потомком u.

Ниже приведен псевдокод поиска в ширину. Отличие в применении данного алгоритма к графу, представленному в виде списков смежности и в виде матриц, в том, что в первом случае проход по всем смежным с данной вершинам выполняется обычным проходом по списку, а во втором — проходом по соответствующей строке матрицы с отбором смежных вершин (элементы матрицы для которых ненулевые). В каждой вершине графа хранится также дополнительная информация — цвет вершины (в поле *color*) и ее предшественник (в поле p). Если предшественника у вершины нет, то значение ее поля p — 0.

Расстояние от s до вершины u, вычисляемое алгоритмом, хранится в поле d. Алгоритм использует очередь Q для работы с множеством серых вершин.

```
Алгоритм BFS(G,s)
// Входные данные:  Граф G=(V,E), исходная вершина s
// Выходные данные: в информационные поля всех
//                  достижимых из s вершин
//                  записываются расстояние от
//                  вершины s и предшественник в
//                  дереве поиска в ширину
for каждой вершины u ∈ V[G] - {s} do
begin
    color[u] = WHITE
        d[u] = ∞
        p[u] = 0
end
color[s] = GRAY
    d[s] = 0
    p[s] = 0
Q = ∅

Enqueue(Q,s)
while Q ≠ ∅ do
begin
    u = Dequeue(Q)
    for каждой вершины v, смежной с u do
    begin
        if color[v] = WHITE then
        begin
            color[v] = GRAY
            d[v] = d[u] + 1
            p[v] = u
            Enqueue(Q,v)
        end
    end
    color[u] = BLACK
end
```

Время работы данного алгоритма при использовании списков смежности составляет $O(V+E)$, а при использовании матрицы смеж-

ности — $O(V^2)$, так как при поиске смежных вершин приходится обойти все ее элементы.

Можно доказать, что приведенный алгоритм корректно определяет длины кратчайших путей к вершинам графа от вершины *s*. Вывести соответствующий путь легко с помощью представленной ниже процедуры.

```
Алгоритм PrintPath(G,s,v)
// Входные данные:  граф G с информацией, полученной
//                    в процессе поиска в ширину из
//                    вершины s, и вершина v
// Выходные данные: кратчайший путь из вершины s в
//                    вершину v
if v = s then print s
else if p[v] = 0
    then print "Путь из "s" в "v" отсутствует"
    else
    begin
        PrintPath(G,s,p[v])
        print v
    end
```

Поскольку алгоритм достаточно прост, не будем составлять его обобщенную реализацию. В качестве реализации алгоритма рассмотрим решение задачи о превращении мухи в слона. Давным-давно в одной из научно-популярных книг о вычислительных машинах и программировании [11] было сказано, что кратчайшая цепочка из четырехбуквенных слов, первое из которых — «муха», а последнее — «слон», и каждое из которых отличается от предшествующего одной буквой, состоит из 17 слов. Давайте проверим это. Приведенная ниже программа считывает файл **dictionary**, в котором собраны все четырехбуквенные существительные из словаря программы ispell (плюс добавленные для чистоты эксперимента и отсутствующие в словаре слова из [11]), и строит граф, в котором вершины представляют собой считанные слова, а ребра соединяют слова, отличающиеся ровно на одну букву.

Программа, как обычно, не оптимизирована, не обрабатывает ошибки и максимально проста. Для списков использован тип **vector**, так как тип **list** не имеет в этом случае никаких преимуществ; для очереди использован тип **deque**.

```
struct word_t          // Структура вершины графа
{
    char word[5];      // Мы имеем дело только с
                       // четырехбуквенными словами
    vector<int> adj; // Смежные вершины графа
    int  color;
    int  p;
    int  d;
    word_t(const char * w = ""):p(-1),color(0),
                    d(INT_MAX) { strcpy(word,w); }
};
typedef vector<word_t> graph;

// Отличаются ли слова ровно одной буквой?
bool adjacent(const char * a, const char * b)
{
    for(int i = 0, diffs = 0; a[i] && b[i]; ++i)
        if ((a[i] != b[i]) && (++diffs > 1))
            return false;
    return true;
}
// Чтение файла и создание графа
void readGraph(graph&G)
{
    char buf[7];
    ifstream dic("dictionary");
    while(dic.getline(buf,7) && dic.good())
        G.push_back(word_t(buf));
    // Создание списков смежности
    for(int i = 0; i < G.size() - 1; ++i)
        for(int j = i+1; j < G.size(); ++j)
            if (adjacent(G[i].word,G[j].word)) {
                G[i].adj.push_back(j);
                G[j].adj.push_back(i);
            }
}
// Вспомогательная функция поиска вершины по слову
int indexOf(graph&G, const char * word)
```

```
{
    for(int i = 0; i < G.size(); ++i)
        if (strcmp(G[i].word,word)==0) return i;
    return -1;
}
// Поиск в ширину
void BFS(graph&G,int s)
{
    // Цикл инициализации не нужен, так как она
    // выполнена в конструкторах вершин. Однако,
    // если планируется выполнение нескольких
    // поисков с одним графом, здесь нужна
    // инициализация полей color, d, p всех вершин
    G[s].color = 1; // GRAY
    G[s].d     = 0;
    deque<int> Q;
    Q.push_back(s);
    while(!Q.empty()) {
        int u = Q.front();
        Q.pop_front();
        for(int j= 0; j< G[u].adj.size(); ++j) {
            int v = G[u].adj[j];
            if (G[v].color == 0) {
                G[v].color = 1; // GRAY
                G[v].d = G[u].d + 1;
                G[v].p = u;
                Q.push_back(v);
            }
        }
        G[u].color = 2; // BLACK
    }
}

void printPath(graph&G, int s, int v)
{
    if (v==s) cout << G[s].word << endl;
    else if (G[v].p == -1)
        cout << "Пути от " << G[s].word << " к "
            << G[v].word << " нет" << endl;
```

```
    else {
        printPath(G,s,G[v].p);
        cout << G[v].word << endl;
    }
}

int main()
{
    graph G;
    readGraph(G);
    int fly      = indexOf(G,"муха");
    int elephant = indexOf(G,"слон");
    BFS(G,fly);
    printPath(G,fly,elephant);
    return;
}
```

В результате мы действительно получаем последовательность из 17 слов: муха-мура-фура-фара-кара-каре-кафе-кафр-каюр-каюк-каик-крик-крип-клип-клин-клон-слон, которая, впрочем, отличается от последовательности из упомянутой книги: муха-мура-фура-фара-кара-каре-кафе-кафр-каюр-каюк-крюк-урюк-урок-срок-сток-стон-слон. Заметим в этой связи, что поиск в ширину находит кратчайший путь, но нигде не сказано, что этот путь — единственный, и наша программа нашла один из возможных путей (стоит, кстати, убрать из словаря неизвестный в те годы «клип», как программа тут же находит второй путь). Это далеко не самая длинная цепочка — например, кратчайший путь от «тиши» к «грому» выглядит следующим образом: тишь-тушь-туша-душа-дура-фура-фара-кара-каре-кафе-кафр-каюр-каюк-крюк-урюк-урок-урод-брод-бром-гром (любопытно, что в обоих приведенных случаях имеется общая подцепочка из 11 слов: фура-...-урок).

Поиск в глубину

Стратегия *поиска в глубину* состоит в том, чтобы идти вглубь графа, пока это возможно. При выполнении поиска в глубину исследуются все ребра, выходящие из последней открытой вершины, и мы покидаем вершину только тогда, когда не остается неисследованных

выходящих из нее ребер. При этом происходит возврат в вершину, из которой была открыта текущая вершина. Этот процесс продолжается до тех пор, пока не будут открыты все вершины, достижимые из исходной. Если при этом остаются неоткрытые вершины, то одна из них выбирается в качестве новой исходной вершины, и поиск возобновляется из нее. Процесс повторяется до тех пор, пока не будут открыты все вершины графа.

Как и при поиске в ширину, когда в процессе сканирования списка смежности уже открытой вершины u открывается вершина v, процедура поиска записывает это событие, устанавливая поле предшественника v равным u. В отличие от поиска в ширину, где подграф предшествования образует дерево, при поиске в глубину подграф предшествования может состоять из нескольких деревьев, так как поиск может выполняться из нескольких исходных вершин. Поиск в ширину ограничивается только одной исходной вершиной, поскольку основное его предназначение — поиск кратчайшего пути из данной вершины. Поиск же в глубину самостоятельно используется редко и обычно является частью другого алгоритма. Подграф предшествования поиска в глубину, таким образом, образует лес, который состоит из нескольких деревьев поиска в глубину.

Как и при поиске в ширину, вершины графа раскрашиваются в разные цвета, указывающие их состояние. Каждая вершина изначально белая, затем при ее открытии в процессе поиска она окрашивается в серый цвет, а по завершении сканирования ее списка смежности она становится черной. Это гарантирует, что каждая вершина в конечном счете находится только в одном дереве поиска в глубину, так что деревья не пересекаются.

Помимо построения леса поиска в глубину, поиск в глубину также проставляет в вершинах метки времени. Каждая вершина имеет две такие метки — d, в которой указывается, когда вершина v открывается и окрашивается в серый цвет, и f, в которую записывается момент, когда завершается сканирование списка смежности вершины v, и она становится черной. Эти метки используются многими другими алгоритмами и полезны при рассмотрении поведения поиска в глубину. Само собой, для каждой вершины $d<f$.

Вот псевдокод алгоритма поиска в глубину. Граф G может быть как ориентированным, так и неориентированным. Переменная *time* — глобальная и используется для простановки меток времени.

```
Алгоритм DFS(G)
// Входные данные:  граф G
// Выходные данные: граф G, в котором вершины
//                   заполнены информацией,
//                   полученной при поиске в глубину
for каждой вершины u ∈ V[G] do
begin
    color[u] = WHITE
    p[u]     = 0
end
time = 0
for каждой вершины u ∈ V[G] do
begin
    if color[u] = WHITE
        then DFS_Visit(u)
end

DFS_Visit(u)
color[u] = GRAY
time = time +1
d[u] = time
for каждой вершины v, смежной с u do
begin
    if color[v] = WHITE then
    begin
        p[v] = u
        DFS_Visit(v)
    end
end
color[u] = BLACK
time = time +1
f[u] = time
```

При возврате из процедуры *DFS* каждой вершине u сопоставляются два момента времени — время открытия d и время завершения f. Время работы данного алгоритма при использовании списков смежности составляет $O(V+E)$, а при использовании матрицы смежности — $O(V^2)$, так как при поиске смежных вершин приходится обойти все ее элементы.

Поиск в глубину дает информацию о структуре графа. Подграф предшествования образует лес деревьев, поскольку структура деревьев поиска в глубину в точности отражает структуру рекурсивных вызовов процедуры *DFS_Visit*. Вершина v является потомком вершины u в лесу поиска в глубину тогда и только тогда, когда вершина u серая в момент открытия вершины v.

Еще одно важное свойство поиска в глубину заключается в том, что времена открытия и завершения образуют скобочную структуру, т.е. если открытие вершины u представить при помощи отрывающей скобки «(u», а завершение — при помощи закрывающей скобки «u)», то перечень открытий и завершений образует корректное выражение в смысле вложенности скобок.

При поиске в глубину в (ориентированном или неориентированном) графе $G = (V, E)$ для любых двух вершин u и v выполняется ровно одно из трех следующих утверждений.

- Отрезки $[d[u], f[u]]$ и $[d[v], f[v]]$ не пересекаются, и ни u не является потомком v в лесу поиска в глубину, ни v не является потомком u.
- Отрезок $[d[u], f[u]]$ полностью содержится в отрезке $[d[v], f[v]]$, и u является потомком v в дереве поиска в глубину.
- Отрезок $[d[v], f[v]]$ полностью содержится в отрезке $[d[u], f[u]]$, и v является потомком u в дереве поиска в глубину.

Соответственно, вершина v является потомком u (отличным от самого u) в лесу поиска в глубину в (ориентированном или неориентированном) графе G тогда и только тогда, когда $d[u] < d[v] < f[v] < f[u]$.

Здесь мы ограничимся только перечисленными свойствами поиска в глубину; дополнительную информацию можно найти в соответствующей литературе, например [12]. Пока же воспользуемся перечисленными свойствами, позволяющими нам решить одну головоломку из книги [6]. Вот вкратце ее суть — в некотором городке мэр назначил начальником отдела регулирования дорожного движения большого шутника, который развесил по городу множество знаков так, что движение по городу стало очень запутанным делом, приносившим в городскую казну большие доходы от штрафов за нарушения. Все ожидали, как проедет через город (и сколько штрафов при этом заплатит) один самый богатый фермер в округе. Но он сумел заранее достать план города со всеми дорожными знаками и разоча-

ровал всех, проехав город без единого нарушения. Как он это сделал? План города показан на рис. 3.3. Очутившись на любом перекрестке, вы имеете право двигаться в направлении одной из стрелок, т.е. поворачивать в нужную сторону разрешается лишь при условии, что имеется закругление, по которому можно свернуть, а следовательно, прямо — лишь при условии, что в нужную сторону идет прямая линия. Поворачивать, двигаясь задним ходом, запрещается; развороты также запрещены. Покидать перекресток разрешается только в направлении одной из стрелок.

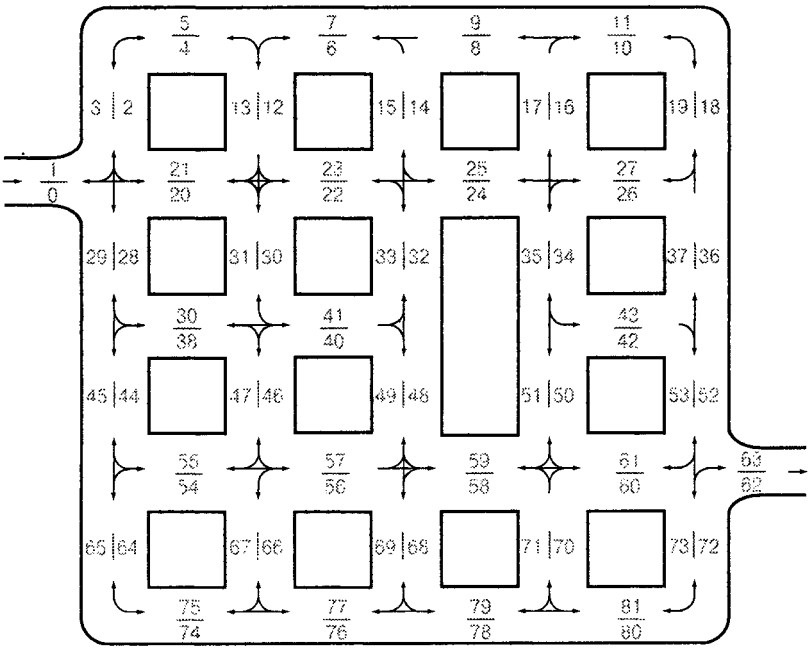


Рис. 3.3. *План города*

Давайте переведем нашу задачу на язык ориентированных графов. Перенумеруем все улицы между перекрестками, сделав их вершинами графов. Поскольку развороты запрещены, мы вынуждены использовать по две вершины для каждой улицы (сообразите сами, почему). Использованная нумерация показана на рис. 3.3. После этого для каждой вершины мы определяем все исходящие из нее ребра — ве-

дущие в другие вершины с учетом знаков разрешенных поворотов и правосторонности движения. Так, например, из вершины 32 мы можем попасть только в вершины 14 и 23.

Вот как выглядит программа для решения данной задачи. Как обычно, она не оптимизирована, не обрабатывает ошибки и максимально проста. В ней использован простейший класс для матрицы, обеспечивающий ее динамическое создание и возможность обращения к элементам с применением двойного индексирования (такой же класс будет использоваться и в главе, посвященной работе с матрицами).

```
// Класс для представления матрицы смежности
template<typename T>
class matrix
{
public:
    matrix(int m = 1, int n = 1):Rows_(m,
                                  vector<T>(n,0)) {}
    const vector<T>& operator[](int i) const
            { return Rows_[i]; }
    vector<T>& operator[](int i) { return Rows_[i]; }
    int rows() const { return Rows_.size(); }
    int cols() const { return Rows_[0].size(); }
private:
    vector< vector<T> > Rows_;
};
// Ребра графа
struct {
    int f,t;
} connections[] =
{
{ 0,20}, { 0, 2}, { 2, 4}, { 3, 1}, { 3,29}, { 3,20},
{ 4,13}, { 5, 3}, { 7,13}, { 8,10}, { 9, 7}, {10,19},
{11, 9}, {12, 6}, {12, 5}, {13,21}, {13,22}, {13,31},
{14, 7}, {15,24}, {15,33}, {16,10}, {17,35}, {18,11},
{19,27}, {20,31}, {20,22}, {21, 1}, {21, 2}, {22,24},
{22,33}, {23,21}, {23,31}, {24,26}, {25,14}, {25,23},
{26,18}, {27,25}, {27,35}, {28, 2}, {29,45}, {29,38},
{30,22}, {30,21}, {31,40}, {32,14}, {32,23}, {33,49},
```

```
    {34,26}, {34,16}, {35,42}, {35,51}, {36,18}, {37,53},
    {38,47}, {38,40}, {39,28}, {39,45}, {40,32}, {40,49},
    {41,30}, {41,39}, {41,47}, {42,53}, {43,34}, {44,28},
    {44,38}, {45,54}, {45,65}, {46,39}, {46,40}, {47,55},
    {47,56}, {48,32}, {49,58}, {50,34}, {51,59}, {51,60},
    {52,36}, {53,73}, {53,61}, {54,46}, {54,56}, {55,44},
    {55,65}, {56,58}, {56,48}, {57,46}, {57,67}, {57,55},
    {58,60}, {58,50}, {59,48}, {60,52}, {61,50}, {61,59},
    {64,44}, {64,54}, {65,74}, {66,56}, {67,75}, {67,76},
    {68,48}, {68,58}, {69,77}, {69,78}, {70,60}, {70,59},
    {71,79}, {71,80}, {72,52}, {72,62}, {73,81}, {74,66},
    {74,76}, {75,64}, {76,78}, {76,68}, {77,66}, {77,75},
    {78,80}, {78,70}, {79,68}, {79,77}, {80,72}, {81,70},
    {81,79}, {63,73},
    {-1,-1} // Ограничитель
};
// Вершина графа
struct node
{
    int color;
    int d, f;
    int p;
};
// Счетчик
int timer;
// Реализация поиска в глубину
void DFSVisit(int i, vector<node>&V, matrix<bool>&E)
{
    V[i].color = 1; // GRAY
    V[i].d = ++timer;
    for(int j = 0; j < V.size(); ++j) {
        if (E[i][j] && (V[j].color == 0)) {
            V[j].p = i;
            DFSVisit(j,V,E);
        }
    }
    V[i].color = 2;
    V[i].f = ++timer;
}
```

```
void DFS(vector<node>&V, matrix<bool>&E)
{
    for(int i = 0; i < V.size(); ++i) {
        V[i].color =  0; // WHITE
        V[i].p     = -1;
    }
    timer = 0;
    for(int i = 0; i < V.size(); ++i) {
        if (V[i].color == 0) DFSVisit(i,V,E);
    }
}
// Реализация поиска в ширину
void BFS(vector<node>&V, matrix<bool>&E, int s)
{
    for(int i = 0; i < V.size(); ++i) {
        if (i == s) continue;
        V[i].color =  0; // WHITE
        V[i].p     = -1;
        V[i].d     = INT_MAX;
    }
    V[s].color = 1; // GRAY
    V[s].d = 0;
    V[s].p = -1;
    deque<int> Q;
    Q.push_back(s);
    while(!Q.empty()) {
        int u = Q.front();
        Q.pop_front();
        for(int v = 0; v < V.size(); ++v) {
            if (E[u][v] && (V[v].color == 0)) {
                V[v].color = 1;
                V[v].d = V[u].d + 1;
                V[v].p = u;
                Q.push_back(v);
            }
        }
        V[u].color = 2;
    }
}
```

```
// Вывод найденного пути
void printPath(vector<node>&V,int N)
{
    if (V[N].p >= 0) {
        printPath(V,V[N].p);
        cout << " - ";
    }
    cout << setw(2) << N;
}

int main()
{
    matrix<bool> E(82,82);
    vector<node> V(82);
    // Инициализация матрицы смежности
    for(int i = 0; connections[i].f >= 0; ++i)
      E[connections[i].f][connections[i].t] = true;
    DFS(V,E);
    if ((V[0].d < V[62].d) && (V[62].f < V[0].f)) {
        cout << "Путь существует:\n";
        printPath(V,62);
    }

    BFS(V,E,0);
    cout << "\n\nКратчайший путь:\n";
    printPath(V,62);
}
```

В программе использован поиск как в глубину, так и в ширину — просто для сравнения полученных результатов. Поиск в глубину говорит нам о том, что проехать через город можно, и дает нам ответ, как именно:

```
 0 -  2 -  4 - 13 - 22 - 33 - 49 - 58 - 50 - 34 - 16 - 10 -
19 - 27 - 35 - 42 - 53 - 73 - 81 - 79 - 77 - 75 - 64 - 44 -
38 - 47 - 55 - 65 - 74 - 76 - 78 - 80 - 72 - 62
```

Поиск в ширину дает кратчайший путь проезда через город:

```
 0 - 20 - 22 - 33 - 49 - 58 - 50 - 34 - 16 - 10 - 19 - 27 -
35 - 42 - 53 - 73 - 81 - 79 - 77 - 75 - 64 - 44 - 38 - 47 -
55 - 65 - 74 - 76 - 78 - 80 - 72 - 62
```

Как видите, этот путь немного короче пути, найденного поиском в глубину. Убедитесь сами, что пути вполне корректны, т.е. согласуются с расставленными в городе знаками дорожного движения.

Топологическая сортировка

С поиском в глубину тесно связана так называемая топологическая сортировка. Топологическая сортировка ориентированного ациклического графа представляет собой такое линейное упорядочение всех его вершин, что если граф G содержит ребро (u,v), то u располагается до v (очевидно, что, если граф не является ацикличным, такая сортировка невозможна). Топологическую сортировку графа можно рассматривать как такое упорядочение его вершин вдоль горизонтальной линии, что все ребра направлены слева направо.

Алгоритм поиска в глубину позволяет легко и просто выполнить топологическую сортировку — достаточно выполнить поиск в глубину. Всякий раз при завершении работы над очередной вершиной ее надо внести в начало результирующего списка, т.е. в алгоритме *DFSVisit* за последней строкой следует добавить строку «внести вершину u в начало связанного списка» (вместо связанного списка можно воспользоваться стеком).

Вот топологическая сортировка в действии. Представим контору с чиновниками, пронумерованными от 0 до 6, причем чиновник 0 не выдаст справку без справки от чиновника 1, 1 — без справок от 2 и 5, 2 — от 5, 3 — от 0, 2 и 6, и 6 — без справки от 4. В каком порядке следует обходить чиновников?

Приведенный ниже листинг дает ответ на этот вопрос. (Определение класса **matrix**, идентичное уже рассматривавшемуся, опущено. Выделенный курсивом код поясняется немного позже.)

```
struct {
    int f,t;
} connections[] =
{
    {0,3},{1,0},{2,1},
    {2,3},{5,1},{5,2},
    {4,6},{6,3},
    {-1,-1}  // Ограничитель
```

```
};

struct node
{
    int color;
    int d, f;
    int p;
};

int timer;
bool hasCycle;

void DFSVisit(int i, vector<node>&V, matrix<bool>&E,
              stack<int>&Q)
{
    V[i].color = 1; // GRAY
    V[i].d = ++timer;
    for(int j = 0; j < V.size(); ++j) {
        if (E[i][j] && (V[j].color == 1)) // GRAY
                    hasCycle = true;
        if (E[i][j] && (V[j].color == 0)) {
            V[j].p = i;
            DFSVisit(j,V,E,Q);
        }
    }
    V[i].color = 2;
    Q.push(i);
    V[i].f = ++timer;
}

void DFS(vector<node>&V, matrix<bool>&E,
         stack<int>&Q)
{
    for(int i = 0; i < V.size(); ++i) {
        V[i].color =  0; // WHITE
        V[i].p     = -1;
    }
    timer = 0;
    hasCycle = false;
```

```
    for(int i = 0; i < V.size(); ++i) {
        if (V[i].color == 0) DFSVisit(i,V,E,Q);
    }
}

int main()
{
    matrix<bool> E(7,7);
    vector<node> V(7);
    for(int i = 0; connections[i].f >= 0; ++i)
        E[connections[i].f][connections[i].t]=true;
    stack<int> Q;
    DFS(V,E,Q);
    while(!Q.empty()) {
        cout << setw(4) << Q.top();
        Q.pop();
    }
    cout << endl;
}
```

Программа выдает список чиновников в топологически отсортированном порядке (одном из возможных — в общем случае топологически отсортированная последовательность — не единственная): 5 4 6 2 1 0 3.

Проверить, ациклический граф или нет, можно при помощи того же поиска в глубину. Если при рассмотрении очередного ребра (u,v) графа вершина v имеет серый цвет, то это ребро, соединяющее вершину u с ее предком v в дереве поиска в глубину, а значит, граф имеет цикл. Чтобы убедиться, что ребро (u,v) соединяет вершину u с ее предком v, заметим, что серые вершины всегда образуют линейную цепочку потомков, соответствующую стеку активных вызовов процедуры *DFSVisit*: количество серых вершин на единицу больше глубины последней открытой вершины в дереве поиска в глубину. Исследование всегда начинается с самой глубокой серой вершины, так что ребро, которое достигает другой серой вершины, достигает предка исходной вершины. Таким образом, если внести в код изменения, показанные в листинге курсивом, то мы будем знать, ацикличен ли граф (переменная **`hasCycle`** равна **`false`**), и если это не так, то результат топологической сортировки некорректен (данная процедура лишена смысла).

Кратчайшие пути

Из всего множества алгоритмов для работы с графами мы рассмотрим еще только два — для решения задачи о кратчайшем пути из одной вершины и для поиска кратчайших путей между всеми парами вершин. В силу ограниченности объема книги и сложности алгоритмов их детальное описание не приводится; заинтересованный читатель может прочесть об этих и многих других алгоритмах для работы с графами, не рассматривавшихся в этой книге, в [12].

Кратчайшие пути из одной вершины

В задаче о кратчайшем пути из одной вершины дан взвешенный ориентированный граф $G = (V, E)$ с весовой функцией, которая сопоставляет каждому ребру графа некоторый вес, представляющий собой действительное число. Вес пути представляет собой сумму весов входящих в этот путь ребер. Вес кратчайшего пути из одной вершины в другую определяется как минимальное значение веса пути среди всех возможных путей, соединяющих эти две вершины; если такого пути не существует, считаем, что вес кратчайшего пути равен бесконечности. Требуется определить кратчайшие пути из исходной вершины s во все остальные вершины графа.

Эта задача решается алгоритмом Беллмана–Форда, псевдокод которого приведен ниже. Алгоритм универсален в том смысле, что в состоянии работать с отрицательными весами ребер (что не могут делать некоторые другие алгоритмы для решения данной задачи) и определяет, разрешима ли данная задача (нет ли в графе цикла с отрицательным весом, достижимого из исходной вершины). Очевидно, что при наличии цикла с отрицательным весом, достижимого из исходной вершины, задача неразрешима, так как «накручивание» пути по этому циклу позволяет достичь значения, которое меньше любого наперед заданного.

```
Алгоритм BellmanFord(G,w,s)
// Входные данные:  граф G, весовая функция w
//                    и исходная вершина s
// Выходные данные: веса кратчайших путей к вершинам
//                    графа из исходной вершины, и
//                   логическое значение, указывающее,
//                    есть ли в графе цикл с
```

```
//                    отрицательным весом, достижимый
//                    из исходной вершины
for каждой вершины v∈V do
begin
    d[v] = ∞
    p[v] = 0
end
d[s] = 0
for i = 1 to |V|-1 do
    for каждого ребра (u,v)∈E do
    begin
        if d[v] > d[u] + w(u,v) then
        begin
            d[v] = d[u] + w(u,v)
            p[v] = u
        end
    end
for каждого ребра (u,v)∈E do
    if d[v] > d[u] + w(u,v) then return false
return true
```

Время работы алгоритма Беллмана–Форда — $O(VE)$ при использовании представления графа в виде списков смежности.

При реализации этого алгоритма, пожалуй, самым неприятным оказывается вопрос бесконечного значения. Есть два наиболее простых варианта его решения — либо использовать дополнительное поле, указывающее, что значение переменной равно ∞, либо в качестве такового значения принять некоторое заведомо недостижимое значение. В приведенной ниже программе использован второй метод.

В качестве конкретного примера мы воспользуемся графом, показанным на рис. 3.4.

Программа использует представление графа с помощью матрицы смежности. Матрица весов в программе одновременно служит и матрицей смежности — если значение веса превышает значение «бесконечности», значит, данное ребро отсутствует.

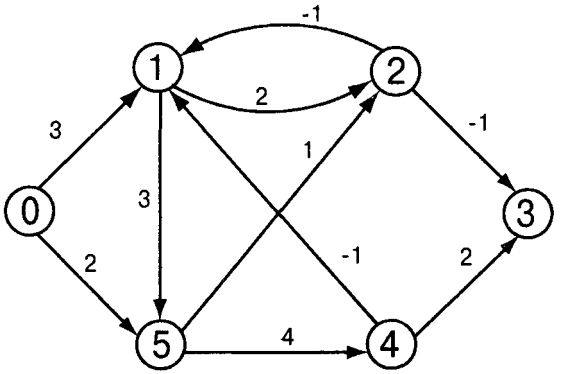


Рис. 3.4. *Пример ориентированного графа для поиска кратчайших путей*

```
struct {
    int f,t;
    double w;
} connections[] =
{
    {0,1, 3}, {1,2, 2}, {2,1,-1},
    {0,5, 2}, {1,5, 3}, {5,2, 1},
    {5,4, 4}, {4,1,-1}, {4,3, 2},
    {2,3,-1},
    {-1, -1, -1}
};

struct node
{
    double d;
    int p;
};

const double inf = 1e300;  // "Бесконечность"

bool BellmanFord(vector<node>&V,
                 matrix<double>&w, int s)
{
    for(int i = 0; i < V.size(); ++i) {
        V[i].d = inf;
```

```
        V[i].p = -1;
    }
    V[s].d = 0;
    for(int i = 0; i < V.size()-1; ++i) {
        for(int u = 0; u < V.size(); ++u)
            for(int v = 0; v < V.size(); ++v) {
                if (w[u][v] < inf) { // Ребро
                    double x = V[u].d + w[u][v];
                    if (V[v].d > x) {
                        V[v].d = x;
                        V[v].p = u;
                    }
                }
            }
    }
    for(int u = 0; u < V.size(); ++u)
        for(int v = 0; v < V.size(); ++v) {
            if (w[u][v] < inf) {
                if (V[v].d > V[u].d + w[u][v])
                    return false;
            }
        }
    return true;
}

void printPath(vector<node>&V, int s)
{
    if (V[s].p >= 0) {
        printPath(V,V[s].p);
        cout << " - ";
    }
    cout << s;
}

int main()
{
    vector<node> V(6);
    matrix<double> W(6,6);
    for(int i = 0; i < W.rows(); ++i)
```

```
        for(int j = 0; j < W.cols(); ++j)
            W[i][j] = 2*inf;
    for(int i = 0; connections[i].f >= 0; ++i) {
        W[connections[i].f][connections[i].t] =
                connections[i].w;
    }
    bool res = BellmanFord(V,W,0);
    cout << (res ?
        "Циклов с отрицательным весом нет" :
        "Есть цикл с отрицательным весом") << endl;
    if (res) {
        for(int i = 1; i < V.size(); ++i) {
            cout << "Вершина " << i
                 << ": расстояние "
                 << V[i].d << ", путь ";
            printPath(V,i);
            cout << endl;
        }
    }
}
```

Здесь используется представление графа с помощью матрицы смежности, но нет никаких проблем переписать ее для представления графа в виде списков смежности:

```
struct {
    int f,t;
    double w;
} connections[] =
{
    {0,1, 3}, {1,2, 2}, {2,1,-1},
    {0,5, 2}, {1,5, 3}, {5,2, 1},
    {5,4, 4}, {4,1,-1}, {4,3, 2},
    {2,3,-1},
    {-1, -1, -1}
};

struct edge     // Ребро
{
    int v;      // Вершина, в которую оно входит
```

```
    double w;   // Вес ребра
};

struct node
{
    double d;
    int p;
    vector<edge> adj;
};

const double inf = 1e300;

bool BellmanFord(vector<node>&V,int s)
{
    for(int i = 0; i < V.size(); ++i)
    {
        V[i].d = inf;
        V[i].p = -1;
    }
    V[s].d = 0;
    for(int i = 0; i < V.size()-1; ++i)
    {
        for(int u = 0; u < V.size(); ++u)
            for(int i = 0; i < V[u].adj.size(); ++i)
            {
                int v = V[u].adj[i].v;
                double x = V[u].d + V[u].adj[i].w;
                if (V[v].d > x)
                {
                    V[v].d = x;
                    V[v].p = u;
                }
            }
    }
    for(int u = 0; u < V.size(); ++u)
        for(int i = 0; i < V[u].adj.size(); ++i)
        {
            int v = V[u].adj[i].v;
            if (V[v].d > V[u].d + V[u].adj[i].w)
```

```
                return false;
            }
        return true;
}

void printPath(vector<node>&V, int s)
{
    if (V[s].p >= 0) {
        printPath(V,V[s].p);
        cout << " - ";
    }
    cout << s;
}

int main()
{
    vector<node> V(6);
    for(int i = 0; connections[i].f >= 0; ++i)
    {
        edge e;
        e.v = connections[i].t;
        e.w = connections[i].w;
        V[connections[i].f].adj.push_back(e);
    }
    bool res = BellmanFord(V,0);
    cout << (res ?
        "Циклов с отрицательным весом нет" :
        "Есть цикл с отрицательным весом") << endl;
    if (res)
    {
        for(int i = 1; i < V.size(); ++i)
        {
            cout << "Вершина " << i
                 << ": расстояние " << V[i].d
                 << ", путь ";
            printPath(V,i);
            cout << endl;
        }
    }
}
```

Курсивом показаны внесенные в программу изменения. Как видите, их не так уж много. Мы отказались от матрицы, так что вместо инициализации матрицы выполняется инициализация списков смежности (для создания которых введена новая структура **edge**), и в функции **BellmanFord()**, в которую теперь передается на один параметр меньше, внутренний цикл проходит не по всем вершинам в поисках существующих ребер, а только по реально имеющимся ребрам из списка смежности. Естественно, этот вариант программы работает быстрее. Первая программа приведена в учебных целях, чтобы еще раз продемонстрировать, что работать можно с любым представлением матрицы (хотя и с разной эффективностью).

При запуске любой из приведенных программ получается совершенно корректный ответ.

```
Циклов с отрицательным весом нет
Вершина 1: расстояние 2, путь 0 - 5 - 2 - 1
Вершина 2: расстояние 3, путь 0 - 5 - 2
Вершина 3: расстояние 2, путь 0 - 5 - 2 - 3
Вершина 4: расстояние 6, путь 0 - 5 - 4
Вершина 5: расстояние 2, путь 0 - 5
```

Кратчайшие пути между всеми парами вершин

Вторая задача — задача поиска кратчайших путей между всеми парами вершин графа. Такая задача возникает, например, при составлении таблицы расстояний между городами, нанесенными на атлас автомобильных дорог.

Как и в предыдущей задаче, дан взвешенный ориентированный граф $G = (V, E)$ с весовой функцией, которая сопоставляет с каждым ребром графа некоторый вес, представляющий собой действительное число. Очевидно, что один из способов решения данной задачи — применение к каждой вершине графа алгоритма для поиска кратчайших путей из одной вершины, например, алгоритма Беллмана–Форда.

Однако для решения поставленной задачи имеются и специализированные, более эффективные алгоритмы. Здесь мы рассмотрим один из таких алгоритмов — алгоритм Флойда–Воршалла.

В этом алгоритме используется представление графа в виде матрицы. Предполагается, что вершины пронумерованы как $1, 2, \ldots, |V|$, и в роли входных данных выступает матрица W размером $n \times n$,

представляющая веса ребер ориентированного графа $G=(V,E)$ с n вершинами. Другими словами, $W=(w_{ij})$, где

$$w_{ij}=\begin{cases}0, & \text{если } i=j,\\ \text{вес ориентированного ребра}(i,j), & \text{если } i\neq j \text{ и } (i,j)\in E,\\ \infty, & \text{если } i\neq j \text{ и } (i,j)\notin E.\end{cases}$$

Чтобы решить задачу о поиске кратчайших путей между всеми парами вершин со входной матрицей смежности, необходимо вычислить не только веса кратчайших путей, но и матрицу предшествования $P=(p_{ij})$, где величина p_{ij} имеет нулевое значение, если $i=j$ или путь из вершины i в вершину j отсутствует; в противном случае p_{ij} — предшественник вершины j на некотором кратчайшем пути из вершины i. Приведенная далее процедура, представляющая собой модифицированную версию рассматривавшейся ранее процедуры *PrintPath*, выводит кратчайший путь из вершины i в вершину j.

```
PrintPath(P,i,j)
if i = j then print i
else if p_ij = 0 then print "Не существует пути из" i
                          "в" j
     else begin
          PrintPath(P,i,p_ij)
          print j
     end
```

Алгоритм Флойда–Воршалла основан на применении динамического программирования, о котором вкратце рассказывается в главе 7, «Дополнительные вопросы». Описание вывода и принципов работы данного алгоритма выходит за рамки этой книги, так что ограничимся только изложением самого алгоритма. Полное описание алгоритма Флойда–Воршалла можно найти, например, в [12]. Алгоритм итеративно строит матрицу кратчайших расстояний между вершинами графа. Наличие ребер с отрицательным весом в графе допускается, однако предполагается, что циклы с отрицательными весами в графе отсутствуют.

```
Алгоритм FloydWarshall(W)
// Входные данные: матрица W размером nxn весов
//                 ребер ориентированного графа
//                 без циклов с отрицательным весом
```

```
// Выходные данные: матрица кратчайших расстояний
//                  между вершинами и матрица
//                  предшествования
D(0) = W    // Матрица кратчайших расстояний
// Инициализация матрицы предшествования
for i = 1 to n do for j = 1 to n do
begin
    if i ≠ j and W[i][j] < ∞
        then P(0)[i][j] = i;
        else P(0)[i][j] = 0;
end
// Итеративные вычисления матриц
// расстояний и предшествования
for k = 1 to n do
   for i = 1 to n do
      for j = 1 to n do
      begin
         D(k)[i][j] =
              min(D(k-1)[i][j],
                  D(k-1)[i][k]+D(k-1)[k][j])
         if D(k-1)[i][j] > D(k-1)[i][k]+D(k-1)[k][j]
            then P(k)[i][j] = P(k-1)[k][j]
            else P(k)[i][j] = P(k-1)[i][j]
      end
return D(n), P(n)
```

Из структуры алгоритма очевидно, что его временная эффективность равна $\Theta(V^3)$.

При реализации данного алгоритма незачем хранить все промежуточные матрицы. Можно обойтись лишь двумя матрицами, поочередно строя одну на основании другой. Это касается как матриц кратчайших расстояний, так и матрицы предшествования. В качестве конкретного примера применения алгоритма Флойда–Воршалла воспользуемся им для вычисления кратчайших путей между вершинами графа, показанного на рис. 3.4. Поскольку начало программы в точности такое же, как и в реализации алгоритма Беллмана–Форда, приведем здесь только исходный текст измененных функций.

```
void FloydWarshall(matrix<double>&W,
                   matrix<double>&Dret,
                   matrix<int>&Pret)

{
    int n = W.rows();
    matrix<double> D[2];
    D[0] = W; D[1] = W;

    matrix<int> P[2];
    P[0] = matrix<int>(W.rows(),W.rows());
    for(int i = 0; i < n; ++i)
        for(int j = 0; j < n; ++j)
            P[0][i][j] =
                ((i != j) &&
                 (W[i][j] < inf)) ? i : -1;
    P[1] = P[0];

    int no = 1;
    for(int k = 0; k < n; ++k) {
        for(int i = 0; i < n; ++i)
            for(int j = 0; j < n; ++j) {
                double x =
                    D[1-no][i][k]+D[1-no][k][j];
                D[no][i][j] = min(D[1-no][i][j],x);
                P[no][i][j] = (D[1-no][i][j]>x) ?
                    P[1-no][k][j] : P[1-no][i][j];
            }
        no = 1 - no; // Переключаемся на другие
    }                // матрицы D и P
    Dret = D[1-no];
    Pret = P[1-no];
}

void printPath(matrix<int>&P,int i, int j)
{
    if ( i == j) cout << setw(4) << i;
    else if (P[i][j] == -1) cout << "Нет пути из "
                                 << i << " в " << j;
```

```
    else {
        printPath(P,i,P[i][j]);
        cout << setw(4) << j;
    }
}

int main()
{
    matrix<double> W(6,6);
    // Инициализация матрицы весов
    for(int i = 0; i < W.rows(); ++i)
        for(int j = 0; j < W.cols(); ++j)
            W[i][j] = 2*inf;
    for(int i = 0; connections[i].f >= 0; ++i) {
        W[connections[i].f][connections[i].t] =
                        connections[i].w;
    }
    matrix<double> D;
    matrix<int> P;
    FloydWarshall(W,D,P);
    for(int i = 0; i < D.rows(); ++i) {
        for(int j = 0; j < D.cols(); ++j) {
            cout << "D(" << i << "," << j << "):";
            if (D[i][j] >= inf) cout << "   -";
            else cout << setw(4) << D[i][j];
            cout << "    Путь:    ";
            printPath(P,i,j);
            cout << endl;
        }
    }
}
```

Убедитесь самостоятельно, что алгоритм корректно вычисляет кратчайшие пути между вершинами графа (здесь приведена только часть таких путей).

```
D(0,0):    -    Путь:       0
D(0,1):    2    Путь:       0    5    2    1
D(0,2):    3    Путь:       0    5    2
D(0,3):    2    Путь:       0    5    2    3
```

```
D(0,4):   6    Путь:       0   5   4
D(0,5):   2    Путь:       0   5
D(1,0):   -    Путь:    Нет пути из 1 в 0
D(1,1):   1    Путь:       1
D(1,2):   2    Путь:       1   2
D(1,3):   1    Путь:       1   2   3
D(1,4):   7    Путь:       1   5   4
D(1,5):   3    Путь:       1   5
D(2,0):   -    Путь:    Нет пути из 2 в 0
D(2,1):  -1    Путь:       2   1
D(2,2):   1    Путь:       2
D(2,3):  -1    Путь:       2   3
D(2,4):   6    Путь:       2   1   5   4
D(2,5):   2    Путь:       2   1   5
D(3,0):   -    Путь:    Нет пути из 3 в 0
D(3,1):   -    Путь:    Нет пути из 3 в 1
...
```

Глава 4

Численные методы

С проникновением компьютеров в деловую и повседневную жизнь, когда им в основном приходится иметь дело с хранением и получением информации, относительная важность численных методов становится все меньше. Однако их приложения, усиленные мощью современных компьютеров, продолжают распространяться во всех областях фундаментальных исследований и технологий, так что здесь мы познакомимся хотя бы с несколькими базовыми численными методами, которые в полной мере могут рассматриваться как компьютерные алгоритмы.

К сожалению, объем книги не позволяет подойти всерьез к этой незаслуженно заброшенной в последнее время теме и дать ее в более-менее полном объеме, с раскрытием таких подтем, как устойчивость, погрешность и т.п. Поэтому здесь мы рассмотрим только некоторые базовые алгоритмы — решения нелинейных уравнений с одной переменной, численного интегрирования и дифференцирования, решения обыкновенных дифференциальных уравнений и некоторые другие. Там, где указываются какие-то дополнительные особенности тех или иных формул или алгоритмов, например погрешности вычислений, — это делается без вывода соответствующих формул. Заинтересованному читателю можно только посоветовать обратиться к соответствующей математической литературе.

В силу специфики численных методов здесь будут в основном приведены их описания, а псевдокоды алгоритмов и реализации на C++ будут встречаться в этой главе реже, чем в других.

Вычисление значений полиномов и интерполяция функций

Невозможно рассказывать о численных методах и не упомянуть наиболее известный и наверняка один из самых красивых численных алгоритмов — схему Горнера для вычисления значений полиномов. Вычисление значения полинома $P_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$ в

некоторой точке x_0 «в лоб» требует около $n(n+1)/2$ умножений. В то же время стоит переписать выражение для этого полинома в виде $P_n(x)=a_0+x\left(a_1+x\left(a_2+\cdots+x\left(a_{n-1}+xa_n\right)\cdots\right)\right)$, как становится очевидным, что для вычисления значения данного полинома в некоторой точке требуется всего n умножений.

Естественным образом возникают два усовершенствования схемы Горнера для частных случаев четных и нечетных функций. Если функция четная, то $n=2k$, и в полиноме присутствуют только четные степени x; в таком случае алгоритм вычисления естественным образом вытекает из записи полинома как

$$P_{2k}(x)=a_0+x^2\left(a_2+x^2\left(a_4+\cdots+x^2\left(a_{2k-2}+x^2a_{2k}\right)\cdots\right)\right).$$

В случае, если полином является нечетной функцией ($n=2k+1$), то для вычислений его следует привести к виду

$$P_{2k+1}(x)=x\left(a_1+x^2\left(a_3+x^2\left(a_5+\cdots+x^2\left(a_{2k-1}+x^2a_{2k+1}\right)\cdots\right)\right)\right).$$

В силу тривиальности алгоритма ни его псевдокод, ни реализация на C++ не приводятся и остаются читателю в качестве небольшого упражнения.

Интерполяция функций

Задача *интерполяции функции* заключается в определении значения функции в некоторой произвольной точке x, если известны значения этой функции в $n+1$ различных точках $f_i=f(x_i)$, $i=0,1,\ldots,n$ (например, найденные экспериментально или полученные в результате сложных вычислений).

Зачастую для решения этой задачи строится полином n-й степени, который принимает в точках x_i заданные значения, т.е. $L_n(x_i)=f_i$, $i=0,1,\ldots,n$. После этого значение функции в произвольной точке x вычисляется как $L_n(x)$. Такой полином существует, причем он является единственным и определяется как

$$L_n(x)=\sum_{i=0}^{n}p_{ni}(x)f_i,$$

где $p_{ni}(x)=\dfrac{(x-x_0)\cdots(x-x_{i-1})(x-x_{i+1})\cdots(x-x_n)}{(x_i-x_0)\cdots(x_i-x_{i-1})(x_i-x_{i+1})\cdots(x_i-x_n)}$, $i=0,1,\ldots,n$.

Такой интерполяционный полином носит имя интерполяционного полинома Лагранжа. Его максимальную погрешность на отрезке $[a,b]$ можно оценить следующим образом:

$$\max_{[a,b]}\left|f(x)-L_n(x)\right| \le \frac{M_{n+1}}{(n+1)!}\max_{[a,b]}\left|\omega_n(x)\right|,$$

где $M_{n+1}=\max_{[a,b]}\left|f^{(n+1)}(x)\right|$ — максимальное значение $(n+1)$-й производной функции на этом отрезке, а $\omega_n(x)\equiv(x-x_0)(x-x_1)\cdots(x-x_n)$.

Частным случаем полинома Лагранжа является линейная интерполяция, которая вычисляет значение функции на основании двух известных точек, интерполируя ее линейной функцией:

$$f(x)=\frac{f_1-f_0}{x_1-x_0}(x-x_0)+f_0.$$

Геометрически линейная интерполяция означает замену графика функции на отрезке $[x_0,x_1]$ хордой, соединяющей точки (x_0,f_0) и (x_1,f_1) (рис. 4.1).

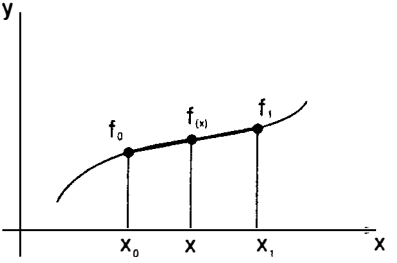


***Рис. 4.1.** Геометрический смысл линейной интерполяции*

Согласно приведенной ранее формуле максимальная погрешность линейной интерполяции на отрезке $[x_0,x_1]$ не превышает значения $\frac{(x_1-x_0)^2}{8}\max_{[x_0,x_1]}\left|f''(x)\right|$.

О других методах интерполяции функций (в частности, полиномах Чебышева, сплайнах и др.) и связанных с ними вопросах можно узнать из [5, 17]. Здесь же мы ограничились полиномом Лагранжа, поскольку он является основой для изложения материала раздела о численном дифференцировании и интегрировании.

Численное дифференцирование и интегрирование

С вопросами интерполяции функций тесно связаны вопросы численного дифференцирования и интегрирования. Задача *численного дифференцирования* заключается в поиске значений производной функции в некоторой произвольной точке x, если известны значения этой функции в $n+1$ различных точках $f_i = f(x_i)$, $i = 0, 1, \ldots, n$, а задача *численного интегрирования* — в поиске значения интеграла $\int_{x_0}^{x_1} f(x)dx$. Здесь мы везде полагаем, что все рассматриваемые производные существуют и непрерывны в рассматриваемых диапазонах области определения.

Обычно применяемые методы численного дифференцирования и интегрирования строятся на использовании полинома Лагранжа, т.е. мы используем его в качестве функции, для которой и ищем производную или интеграл. Как правило, в учебниках вопросы численного дифференцирования и интегрирования рассматриваются по отдельности. Здесь же мы пойдем иным путем — рассмотрим несколько первых полиномов Лагранжа и получим на их основе формулы численного интегрирования и дифференцирования.

Итак, начнем с полинома Лагранжа первой степени. В этом случае он представляет собой, как уже говорилось, линейную функцию

$$f(x) = \frac{f_1 - f_0}{x_1 - x_0}(x - x_0) + f_0.$$

Дифференцируя ее, мы получим, что $f'(x) = \frac{f_1 - f_0}{x_1 - x_0}$ для всех точек отрезка. Это значение и есть приближенное значение первой производной; естественно, что в таком случае значение второй и высших производных — нулевые. Для произвольной точки x исходного диапазона $[x_0, x_n]$ производная вычисляется следующим образом: находятся две соседние с x точки такие, что $x_i \le x \le x_{i+1}$, и значение производной в точке x считается равным $(f_{i+1} - f_i)/(x_{i+1} - x_i)$. Погрешность такого приближения составляет $(x_{i+1} - x_i)f''(\xi)/2$, где ξ — некоторая точка отрезка $y_{i+1} = y_i + hy'_{i+1}$.

Интегрируя данную функцию, получаем $\int_{x_0}^{x_1} f(x)dx = \frac{f_0 + f_1}{2}(x_1 - x_0)$, т.е. широко известную формулу трапеций. Погрешность такого при-

ближения составляет $h^3 f''(\xi)/12$, где ξ – некоторая точка отрезка $[x_{i+1}, x_i]$. Для всего диапазона $[x_0, x_n]$ значение интеграла вычисляется как $\frac{1}{2}\sum_{i=0}^{n-1}(f_i + f_{i+1})(x_{i+1} - x_i)$, или для равноотстоящих точек ($x_{i+1} - x_i = h$), $\int_{x_0}^{x_n} f(x)dx \approx h\left((f_0 + f_n)/2 + \sum_{i=1}^{n-1} f_i\right)$, а погрешность будет пропорциональна квадрату шага.

Заметим, что и дифференцирование, и интегрирование оказываются тем точнее, чем меньше величина шага вычислений – расстояния между соседними точками. Но здесь есть одно принципиальное отличие: в случае численного дифференцирования значение шага оказывается в знаменателе дроби, в числителе которой – разность двух близких значений. Пусть погрешность каждого из значений (связанная с неточностью измерений, метода вычисления или представления числа с плавающей точкой в компьютере) не превышает некоторого значения δ. Тогда погрешность вычисления числителя не превышает $2\delta/h$, т.е. при малых значениях h погрешность метода оказывается существенно меньше погрешности округлений. Оптимальным оказывается шаг, при котором величина погрешности метода равна погрешности округления.

Замечание о неточности, присущей численному дифференцированию, в еще большей степени относится к вычислению производных более высоких порядков. Это соображение никогда нельзя упускать из виду.

Теперь рассмотрим полином Лагранжа второй степени. Чтобы избежать громоздких вычислений, будем считать, что мы имеем дело с равноотстоящими точками ($x_{i+1} - x_i = h$). (Вычисления для общего случая не столько сложнее, сколько более громоздки.)

Полином в этом случае имеет вид

$$L_2(x) = \left((x - x_1)(x - x_2)f_0 - 2(x - x_0)(x - x_2)f_1 + (x - x_0)(x - x_1)f_2\right)/2h^2.$$

Дифференцируя в точке $x = x_1$, мы получим, что $f'(x_1) \approx \approx (f_2 - f_0)/2h$, т.е. производную в средней точке отрезка можно представить как отношение разности значений функции на концах отрезка к его длине. Заметим, что для полинома второго порядка это точная формула. Погрешность такого приближения составляет $h^2 f'''(\xi)/6$, где ξ – некоторая точка отрезка $[x_0, x_2]$. Нетрудно получить формулы и для производных на концах отрезка – например, производную в точке x_0 можно приближенно вычислить как $f'(x_0) \approx (4f_1 - 3f_0 - f_2)/2h$.

Вторая производная в точке $x = x_1$ равна $f''(x_1) \approx (f_2 - 2f_1 + f_0)/h^2$ с погрешностью порядка $h^2 f^{(4)}(\xi)/12$. Нетрудно сообразить, что третья и высшие производные при приближении функции полиномом Лагранжа второго порядка равны 0.

Интегрируя полином Лагранжа второго порядка, мы получаем формулу $\int_{x_0}^{x_2} f(x)dx = h(f_0 + 4f_1 + f_2)/3$ — формулу численного интегрирования Симпсона (ее можно записать и в ином, более привычном виде $\int_a^b f(x)dx = (b-a)(f(a) + 4f((a+b)/2) + f(b))/6$). Для всего диапазона $[x_0, x_{2n}]$ (обратите внимание на индекс $2n$, связанный с тем, что формула вычисляет интеграл на двух базовых отрезках) формулу Симпсона можно записать в виде

$$\int_{x_0}^{x_{2n}} f(x)dx \approx ((x_{2n} - x_0)/6n)(f_0 + f_{2n} + 2(f_2 + f_4 + \cdots + f_{2n-2}) + 4(f_1 + f_3 + \cdots + f_{2n-1})).$$

Погрешность формулы Симпсона можно оценить как

$$h^4 (x_n - x_0) \max_{[x_0, x_n]} \left| f^{(4)}(x) \right| / 2880.$$

На этом стоит остановиться, хотя, конечно же, можно использовать для вывода формул численного дифференцирования и интегрирования и полиномы Лагранжа более высоких степеней. Понимая, откуда и как выводятся эти формулы, вы сможете при необходимости вывести их самостоятельно.

Ниже приведена простейшая реализация вычисления определенного интеграла по формуле Симпсона с заданной точностью. Для этого интеграл вычисляется с заданным и удвоенным шагом, и полученные значения сравниваются. Если их значения отличаются больше, чем указывает параметр точности, шаг уменьшается в 4 раза, и вычисления повторяются; если меньше — выводится вычисленное значение интеграла.

Заметим также, что использование функтора решает проблему с возможной передачей параметров подынтегральной функции.

```
template<typename Double, typename Functor>
    Double Simpson(Double a, Double b,
                   Functor f, Double eps)
{
```

```
    eps *= eps;
    int N = 4;
    Double sum1, sum2, h;
    for(;;)
    {
        h = (b - a)/(2*N);
        sum1 = sum2 = (f(a) + f(b))/2;
        int idx1 = 1, idx2 = 1;
        Double x = a + h;
        Double fsave;
        for(int i = 1; i < 2*N; ++i)
        {
          sum2 += (fsave = f(x))*(idx2 = 3 - idx2);
          if (i % 2 == 0)
          {
              sum1 += fsave*(idx1 = 3 - idx1);
          }
          x += h;
        }.
        x = (sum2 - 2*sum1)/sum2;
        if (x*x < eps) return sum2*h*2/3; else N *= 4;
    }
}
```

С другими методами численного дифференцирования и интегрирования можно познакомиться в соответствующей литературе, например [5,13,17].

Решение нелинейных уравнений

Задача решения нелинейного уравнения состоит в поиске для заданной функции действительного переменного $f(x)$ значений корней уравнения $f(x)=0$. Обычно данная задача решается в два этапа. Сначала изучается расположение корней и выполняется их выделение — т.е. определяются отрезки области определения функции, содержащие ровно один корень, а на втором этапе на основе полученного начального приближения строится итерационный процесс, позволяющий уточнить значение искомого корня.

Каких-либо общих регулярных приемов решения задачи о расположении корней произвольной функции $f(x)$ не существует. Этот вопрос мы не будем рассматривать, считая, что действительные корни уже выделены, т.е. мы знаем, что на отрезке $[a,b]$ (на котором функция как минимум непрерывна, а для некоторых методов имеет также непрерывную производную) имеется один корень (если их несколько, то, разбивая отрезок на более мелкие, можно выделить все корни и указать отрезки, в которых содержится по одному корню).

Основной метод решения уравнения при заданных условиях — метод бисекции (деления пополам). Предположим, что в интервале (a,b) (мы говорим об интервале, чтобы исключить наличие корня на конце отрезка) имеется один корень уравнения $f(x)=0$. Очевидно, что тогда значения $f(a)$ и $f(b)$ имеют разные знаки. Пусть для определенности $f(a)<0$, $f(b)>0$. Вычислим середину интервала $x_0=(a+b)/2$ и $f(x_0)$. Если найденное значение отрицательно, то искомый корень находится в интервале (x_0,b), а если положительно — то в интервале (a,x_0). Выберем нужный нам интервал и повторим описанную процедуру. На каждом шаге длина интервала оказывается в два раза меньше предыдущего. Когда длина очередного полученного интервала окажется менее наперед заданной точности вычисления корня, процесс поиска прекращается.

Реализация такого алгоритма совершенно тривиальна (приведена ниже). Главным преимуществом данного алгоритма является его предсказуемость. Поскольку на каждом шаге длина отрезка уменьшается вдвое, для достижения заданной точности требуется $\lceil \log_2((a-b)/\varepsilon) \rceil$ итераций, где ε — требуемая точность вычисления.

```
template<typename Double, typename Functor>
Double BiSection(Double a,   Double b,
                 Double eps, Functor f)
{
    if (a > b) swap(a,b);
    Double fa = f(a), fb = f(b);
    assert(fa*fb < 0);
    Double x = (a + b)/2;
    for(; b - a > eps; x = (a + b)/2)
    {
        Double fx = f(x);
```

```
        if (fx*fa < 0)
        {
            b = x;
        }
        else
        {
            a  = x;
            fa = fx;
        }
    }
    return x;
}
```

Более быстрой сходимостью — квадратичной — обладает метод Ньютона. Квадратичная сходимость означает, что если на k-й итерации было получено приближение x_k точного корня x^*, то погрешность на следующей итерации можно записать как $|x_{k+1} - x^*| = O\left((x_k - x^*)^2\right)$. Суть самого метода Ньютона в том, что в окрестности точки приближенного решения функция заменяется первыми членами ряда Тейлора, т.е. $f(x) \approx f(x_0) + (x - x_0) f'(x_0)$. Отсюда мы находим очередное приближение корня

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}.$$

Геометрический смысл метода Ньютона показан на рис. 4.2: точка x_{k+1} — это точка пересечения с осью абсцисс касательной к графику функции $f(x)$, проведенной в точке $(x_k, f(x_k))$. Именно поэтому второе название данного метода — метод касательных.

Недостатком данного метода является то, что такая быстрая квадратичная сходимость метода гарантируется только при очень хороших, близких к точному решению начальных приближениях. Если начальное приближение выбрано неудачно, то метод может сходиться медленно либо вообще расходиться. Кроме того, в данном методе помимо вычисления значения функции требуется также вычисление значения производной, что не всегда возможно либо может быть вычислительно сложной задачей.

В этом случае можно воспользоваться модифицированным методом Ньютона, в котором производная вычисляется только в одной

точке: $x_{k+1} = x_k - \left(f(x_k)/f'(x_0)\right)$. Такой метод предъявляет меньшие требования к выбору начального приближения x_0, однако обладает всего лишь линейной сходимостью.

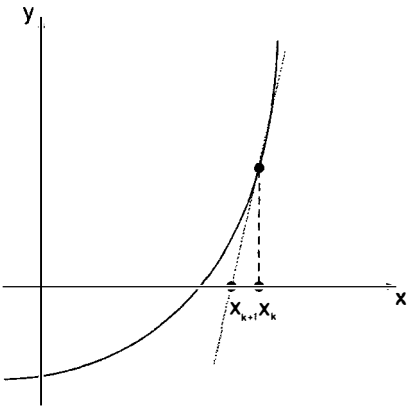


***Рис. 4.2.** Метод Ньютона решения нелинейного уравнения*

Если вместо вычисления производной воспользоваться ее приближенным значением $\left(f(x_k) - f(x_{k-1})\right)/(x_k - x_{k-1})$, то мы получим метод секущих:

$$x_{k+1} = x_k - \frac{(x_k - x_{k-1})}{\left(f(x_k) - f(x_{k-1})\right)} f(x_k).$$

Геометрическая интерпретация метода секущих состоит в следующем: через точки $\left(x_{k-1}, f(x_{k-1})\right)$ и $\left(x_k, f(x_k)\right)$ проводится прямая, и точка пересечения этой прямой с осью абсцисс является новым приближением x_{k+1}.

Этими методами мы и ограничимся в нашем рассмотрении решения нелинейных уравнений с одним неизвестным. К сожалению, описанные методы непосредственно не применимы для решения систем нелинейных уравнений с несколькими неизвестными. В силу сложности данной задачи здесь можно сделать лишь небольшое примечание об одном частном методе, который позволяет при некоторых условиях свести решение системы нелинейных уравнений к последовательному решению нелинейных уравнений с одной неизвестной.

Итак, пусть $\boldsymbol{f}(\boldsymbol{x}) = \left(f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), \ldots, f_n(\boldsymbol{x})\right)$, $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$. Система n уравнений $\boldsymbol{f}(\boldsymbol{x}) = 0$ эквивалентна одному уравнению $\Psi(\boldsymbol{x}) = 0$, где $\Psi(\boldsymbol{x}) = f_1^2(\boldsymbol{x}) + f_2^2(\boldsymbol{x}) + \cdots + f_n^2(\boldsymbol{x})$. Очевидно, что ре-

шениями уравнения $\Psi(x)=0$ являются точки нулевых минимумов функции $\Psi(x)$. Допустим, что эта функция дважды дифференцируема в области, содержащей изолированное решение x^*, в окрестности которого поверхности уровня функции Ψ имеют вид, показанный на рис. 4.3.

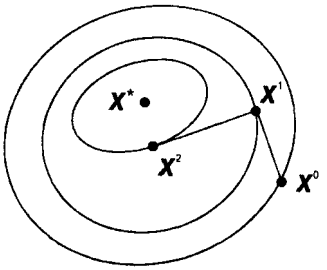


Рис. 4.3. *Метод градиентного спуска решения систем нелинейных уравнений*

Задавшись начальным приближением x^0, мы ищем минимум функции $\Psi\left(x^0-\lambda\nabla\Psi\left(x^0\right)\right)$ одной переменной λ, т.е. фактически применяем один из рассмотренных выше способов решения нелинейных уравнений с одной переменной для поиска минимального неотрицательного корня $\lambda=\lambda_0$ уравнения $\frac{d}{d\lambda}\Psi\left(x^0-\lambda\nabla\Psi\left(x^0\right)\right)=0$. Затем полагаем $x^1=x^0-\lambda_0\nabla\Psi\left(x^0\right)$, составляем аналогичное уравнение с использованием x^1 и находим очередное приближение, т.е. в общем случае на каждой очередной итерации:

$$x^k=x^{k-1}-\lambda_{k-1}\nabla\Psi\left(x^{k-1}\right),\ k=1,2,\ldots,$$

где λ_{k-1} — минимальный неотрицательный корень уравнения

$$\frac{d}{d\lambda}\Psi\left(x^{k-1}-\lambda\nabla\Psi\left(x^{k-1}\right)\right)=0.$$

Главная неприятность заключается в том, что сходимость последовательных приближений x^k к решению уравнения не гарантируется, поскольку можно попасть в точку относительного минимума.

Еще один метод поиска минимума функции $\Psi(x)$, а значит, и решения исходной системы уравнений, состоит в поочередном выборе каждого из компонентов вектора x и поиске такого его значения, которое минимизирует функцию $\Psi(x)$, т.е. решения уравнения

$\frac{d}{dx_i}\Psi(\boldsymbol{x})=0$, когда все значения x_j, $j=1, 2, ..., i+1, ..., n$ фиксированы. Поочередное решение n уравнений для $i=1, ..., n$ дает очередное приближение решения уравнения.

Решение обыкновенных дифференциальных уравнений с начальными условиями

Задача численного решения обыкновенного дифференциального уравнения с начальными условиями заключается в вычислении значений функции y, удовлетворяющей уравнению $y' = f(x,y)$ с начальным условием $y(x_0)=y_0$, в точках $x_i = x_0 + ih$, где h — некоторый шаг, с которым вычисляются значения функции. В данной книге будут рассмотрены только одношаговые методы типа Рунге--Кутта; с другими численными методами решения дифференциальных уравнений вы можете ознакомиться в книгах [3, 5, 13, 17].

Простейший численный метод очевиден — воспользоваться разложением функции y в ряд Тейлора в окрестности точки x_0:

$$y_1 = y_0 + (x_1 - x_0)y_0' + \frac{(x_1 - x_0)^2}{2!}y_0'' + \cdots = y_0 + hf(x_0,y_0) + \frac{h^2}{2!}y_0'' + \ldots.$$

Считая значение h достаточно малым и пренебрегая членами порядка h^2 и выше, получаем, что значение искомой функции в точке x_{i+1} можно приближенно получить, зная значение функции в точке x_i: $y_{i+1} = y_i + hf(x_i,y_i)$. Данный метод носит название *ломаной Эйлера* и имеет погрешность метода (вызванную заменой дифференциального выражения конечным выражением) порядка h^2.

Очевидны два пути улучшения метода ломаной Эйлера. Первый состоит в том, чтобы воспользоваться формулой для численного дифференцирования по трем точкам (см. стр. 141), т.е. рассмотреть дополнительную точку посредине между точками x_i и x_{i+1}: $y'_{i+1/2} = (y_{i+1} - y_i)/h$, откуда сразу же получаем $y_{i+1} = y_i + hf(x + h/2, y_{i+1/2})$. Для вычисления значения $y_{i+1/2}$ можно воспользоваться методом ломаной Эйлера, так что первое улучшение метода ломаной Эйлера выглядит следующим образом:

$$y_{i+1} = y_i + hf(x + h/2, y_i + hf(x_i,y_i)/2).$$

Второе улучшение состоит в том, чтобы записать разложение в ряд Тейлора не только для точки x_i, но и точки x_{i+1}: $y_i = y_{i+1} + (x_i - x_{i+1})y'_{i+1} + \cdots$, откуда мы сразу же получаем, что $y_{i+1} = y_i + (x_{i+1} - x_i)y'_{i+1}$. Теперь можно для поиска y'_{i+1} воспользоваться методом ломаной Эйлера и усреднить полученный результат. Таким образом, мы получаем второе улучшение метода ломаной Эйлера:

$$y_{i+1} = y_i + (h/2)\big(f(x_i, y_i) + f(x_i + h, y_i + hf(x_i, y_i))\big).$$

Оба приведенных метода имеют погрешность порядка h^3. Эти методы являются частными случаями методов Рунге–Кутта, в которых численное решение обыкновенного дифференциального уравнения $y' = f(x,y)$ получается по формуле $y_{i+1} = y_i + \hat{f}(x_i, y_i)$, где $\hat{f}(x_i, y_i)$ строится как весовое среднее значение функции $f(x,y)$ в определенным образом выбираемых точках так, что локальная ошибка метода имеет более высокий порядок. Вывод других формул для более высоких порядков погрешности здесь опущен, и в таблице ниже приводится только конечный результат.

Знание порядка локальной ошибки не имеет практического значения для оценки фактической ошибки обрыва или округления, поэтому для оценки ошибки и управления величиной шага обычно применяется метод Рунге, который заключается в следующем. Пусть ошибка метода имеет порядок k. Приближенное значение $y(x)$, вычисленное в точке x с величиной шага l, обозначим как $Y(x,l)$. Тогда в точке $x = x_0 + 2nh$ имеем

$$y(x) - Y(x,h) \approx A2nh^{k+1} = A(x - x_0)h^k,$$

$$y(x) - Y(x,2h) \approx An(2h)^{k+1} = A(x - x_0)2^k h^k,$$

откуда

$$y(x) - Y(x,h) \approx \frac{Y(x,h) - Y(x,2h)}{2^k - 1},$$

т.е. ошибка при шаге h выражается через приближенные значения при шагах h и $2h$.

Методы Рунге–Кутта без труда переносятся на системы обыкновенных дифференциальных уравнений. Так, для системы уравнений

$$\begin{cases} y' = f(x,y,z) \\ z' = g(x,y,z) \end{cases}$$

Название	Порядок ошибки	Формула	Вспомогательные величины
Метод ломаной Эйлера	h^2	$\hat{f} = f(x_i, y_i)$	
Улучшенная ломаная	h^3	$\hat{f} = \frac{k_1 + k_2}{2}$	$k_1 = f(x_i, y_i)$ $k_2 = f(x_i + h, y_i + hk_1)$
		$\hat{f} = f\left(x_i + \frac{h}{2}, y_i + \frac{h}{2}k_1\right)$	$k_1 = f(x_i, y_i)$
Формула Хойне	h^4	$\hat{f} = \frac{k_1 + 4k_2 + k_3}{6}$	$k_1 = f(x_i, y_i)$ $k_2 = f(x_i + h/2, y_i + (h/2)k_1)$ $k_3 = f(x_i + h, y_i + 2hk_2 - hk_1)$
		$\hat{f} = \frac{k_1 + 3k_3}{4}$	$k_1 = f(x_i, y_i)$ $k_2 = f(x_i + h/3, y_i + (h/3)k_1)$ $k_3 = f(x_i + 2h/3, y_i + (2h/3)k_2)$
Формула Рунге–Кутта	h^5	$\hat{f} = \frac{k_1 + 2k_2 + 2k_3 + k4}{6}$	$k_1 = f(x_i, y_i)$ $k_2 = f(x_i + h/2, y_i + (h/2)k_1)$ $k_3 = f(x_i + h/2, y_i + (h/2)k_2)$ $k_4 = f(x_i + h, y_i + hk_3)$
		$\hat{f} = \frac{k_1 + 4k_3 + k_4}{6}$	$k_1 = f(x_i, y_i)$ $k_2 = f(x_i + h/4, y_i + (h/4)k_1)$ $k_3 = f(x_i + h/2, y_i + (h/2)k_2)$ $k_4 = f(x_i + h, y_i + h(k_1 - 2k_2 + 2k_3))$

формула улучшенной ломаной будет выглядеть следующим образом:

$$y_{i+1} = y_i + h\hat{f}(x_i, y_i, z_i),$$

$$z_{i+1} = z_i + h\hat{g}(x_i, y_i, z_i),$$

где

$$\hat{f} = \frac{k_1 + k_2}{2}, \qquad \begin{aligned} k_1 &= f(x_i, y_i, z_i), \\ k_2 &= f(x_i + h, y_i + hk_1, z_i + hl_i) \end{aligned}$$

$$\hat{g} = \frac{l_1 + l_2}{2}, \qquad \begin{aligned} l_1 &= g(x_i, y_i, z_i), \\ l_2 &= g(x_i + h, y_i + hk_1, z_i + hl_i) \end{aligned}$$

Идея преобразования совершенно очевидна. Аналогично выполняются преобразования и для других формул Рунге–Кутта, так что никаких сложностей здесь не возникает.

Для решения обыкновенных дифференциальных уравнений второго и более высоких порядков можно выполнить их приведение к системам обыкновенных дифференциальных уравнений первого порядка. Так, например, уравнение

$$y'' = f(x, y, y'),\ y(x_0) = y_0,\ y_0'(x_0) = y_0'$$

можно легко привести к только что рассмотренной системе уравнений, выполнив подстановку $y' = z$. При использовании такой подстановки мы получаем систему уравнений

$$\begin{cases} y' = z \\ z' = f(x, y, z) \end{cases}, \text{ при } y(x_0) = y_0,\ z(x_0) = y_0'.$$

Здесь мы не будем рассматривать прочие методы решения как обыкновенных дифференциальных уравнений, так и, например, уравнений в частных производных. Как уже упоминалось, в поисках методов решения таких уравнений или иных, не одношаговых (например, многошаговых, разностных) методов вы можете обратиться к книгам [3, 5, 13, 17].

Далее будет приведена не реализация алгоритмов решения дифференциальных уравнений как таковых, а демонстрационная программа, которая решает одно дифференциальное уравнение — $y' = \cos x - y \operatorname{tg} x$ —

с начальным условием $y(0)=0$ (точное решение которого — $y=x\cos x$) различными методами — Эйлера, улучшенным методом Эйлера и Рунге–Кутта. Как видите, реализация алгоритмов в данной демонстрационной программе совершенно тривиальна, а результаты вычислений (был специально использован весьма грубый шаг просчета, равный 0.1) наглядно показывают точность каждого метода.

```
#include <cstdio>
#include <cmath>

using namespace std;

// Шаг
const double h = 0.1;
// Считаем на отрезке от 0 до 1
const int    steps = 1.0/h;

// Функция y'=f(x,y)
double f(double x, double y)
{
    return cos(x) - y*tan(x);
}

double Euler(double x, double y, double h)
{
    return y + h*f(x,y);
}

double AdvEuler(double x, double y, double h)
{
    double k1 = f(x,y);
    return y + h*f(x+h/2,y+h*k1/2);
}

double RK(double x, double y, double h)
{
    double k1 = f(x,y);
    double k2 = f(x+h/2,y+h*k1/2);
    double k3 = f(x+h/2,y+h*k2/2);
    double k4 = f(x+h,y+h*k3);
```

```
    return y + h*(k1+2*(k2+k3)+k4)/6;
}

int main()
{
    double y = 0.0, ye = 0.0, ya = 0.0, yr = 0.0;
    double x = 0.0;
    printf("   x      Точное    Эйлер     "
           "Улучшенный   Рунге-Кутта\n");
    printf("%8.6lf  %8.6lf  %8.6lf  "
           "%8.6lf     %8.6lf\n",
           x,y,ye,ya,yr);
    for(int i = 0; i < steps; ++i)
    {
        ye = Euler(x,ye,h);
        ya = AdvEuler(x,ya,h);
        yr = RK(x,yr,h);
        x += h;
        y = x*cos(x);              // Точное решение
        printf("%8.6lf  %8.6lf  %8.6lf  "
               "%8.6lf     %8.6lf\n",
               x,y,ye,ya,yr);
    }
}
```

Результат вычислений выглядит следующим образом:

```
   x       Точное     Эйлер      Улучшенный   Рунге-
                                              Кутта
0.000000  0.000000  0.000000  0.000000     0.000000
0.100000  0.099500  0.100000  0.099625     0.099500
0.200000  0.196013  0.198497  0.196252     0.196013
0.300000  0.286601  0.292480  0.286932     0.286601
0.400000  0.368424  0.378966  0.368813     0.368424
0.500000  0.438791  0.455050  0.439194     0.438791
0.600000  0.495201  0.517949  0.495565     0.495201
0.700000  0.535390  0.565047  0.535652     0.535388
0.800000  0.557365  0.593938  0.557458     0.557364
0.900000  0.559449  0.602455  0.559301     0.559446
1.000000  0.540302  0.588697  0.539839     0.540298
```

Глава 5

Матрицы

Здесь мы рассмотрим основные операции, связанные с матрицами, а именно — их умножение, LUP-разложение и решение систем линейных уравнений, а также вычисление определителей и поиск обратных матриц. Мы ограничимся здесь рассмотрением только матриц, элементы которых — действительные числа. Начнем с того, что напомним основные свойства матриц.

Свойства матриц

Матрица представляет собой прямоугольный массив чисел. Например,

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

является матрицей размера 2×3 $A = (a_{ij})$, где $i = 1,2$ и $j = 1,2,3$. Элемент на пересечении i-й строки и j-го столбца матрицы — a_{ij}. Для обозначения матриц будут использоваться прописные буквы, а их элементы будут обозначаться соответствующими строчными буквами с нижними индексами.

Транспонированная матрица A^{T} получается из матрицы A путем обмена местами ее строк и столбцов.

Вектор представляет собой одномерный массив чисел. Например,

$$x = \begin{pmatrix} 2 \\ 3 \\ 5 \end{pmatrix}$$

является вектором размером 3. Для обозначения векторов будут использоваться строчные буквы. Стандартной формой вектора будем считать *вектор-столбец*, представляющий собой матрицу размером $n\times1$. Соответствующий *вектор-строка* получается путем транспонирования. *Единичным вектором* e_i называется вектор, i-й элемент которого равен 1, а все остальные элементы — нулевые.

У *нулевой матрицы* все элементы равны 0. Такая матрица часто записывается просто как 0, поскольку понять, идет речь о числе 0 или нулевой матрице, легко из контекста. То же относится и к размеру матрицы.

Матрица называется квадратной, если число ее столбцов совпадает с числом строк, т.е. если ее размер $n \times n$. Некоторые частные случаи квадратных матриц имеют собственные названия.

Все элементы *диагональной матрицы*, кроме лежащих на главной диагонали, — нулевые, т.е. $a_{ij} = 0$ при $i \neq j$. Поскольку все недиагональные элементы такой матрицы равны 0, диагональную матрицу можно определить путем перечисления ее элементов вдоль диагонали:

$$\operatorname{diag}(a_{11}, a_{22}, \ldots, a_{nn}) = \begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix}.$$

Единичная матрица представляет собой частный случай диагональной, у которой все непулевые элементы равны 1: $I_n = \operatorname{diag}(1,1,\ldots,1)$. Если используется обозначение I без индекса, размер единичной матрицы определяется из контекста. Обратите внимание на то, что i-м столбцом единичной матрицы является единичный вектор e_i.

В ряде математических задач получаются *трехдиагональные матрицы*, в которых $t_{ij} = 0$, если $|i-j| > 1$. Все элементы такой матрицы нулевые, кроме элементов на главной диагонали, а также непосредственно над и под ней:

$$T = \begin{pmatrix} t_{11} & t_{12} & 0 & 0 & \cdots & 0 & 0 & 0 \\ t_{21} & t_{22} & t_{23} & 0 & \cdots & 0 & 0 & 0 \\ 0 & t_{32} & t_{33} & t_{34} & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & t_{n-2,n-2} & t_{n-2,n-1} & 0 \\ 0 & 0 & 0 & 0 & \cdots & t_{n-1,n-2} & t_{n-1,n-1} & t_{n-1,n} \\ 0 & 0 & 0 & 0 & \cdots & 0 & t_{n,n-1} & t_{nn} \end{pmatrix}.$$

Верхнетреугольной матрицей называется матрица, у которой все элементы ниже главной диагонали равны 0 ($u_{ij} = 0$ при $i > j$), а нижнетреугольной, соответственно, называется матрица, у которой нулю равны все элементы выше главной диагонали ($l_{ij} = 0$ при $i < j$). Эти

матрицы называются *единичными* верхне- или нижнетреугольными, если все их элементы на главной диагонали равны 1.

Особое значение имеет *матрица перестановки*, у которой в каждой строке и столбце ровно по одной 1, а на всех прочих местах располагаются 0. Вот пример такой матрицы перестановки:

$$P=\begin{pmatrix}0&1&0&0&0\\0&0&0&1&0\\1&0&0&0&0\\0&0&0&0&1\\0&0&1&0&0\end{pmatrix}.$$

Такие матрицы называются матрицами перестановки, потому что умножение вектора x на матрицу перестановки приводит к перестановке элементов вектора.

И, наконец, *симметричная матрица* A удовлетворяет условию $A=A^{\mathrm{T}}$.

Операции над матрицами

Результатом *сложения матриц* $A=(a_{ij})$ и $B=(b_{ij})$ одинакового размера $m\times n$ является матрица $C=(c_{ij})=A+B$ того же размера, определяемая соотношением $c_{ij}=a_{ij}+b_{ij}$ для $i=1,2,\ldots,m$ и $j=1,2,\ldots,n$, т.е. сложение матриц выполняется поэлементно. Очевидно, что нулевая матрица нейтральна по отношению к сложению матриц: $A+0=A=0+A$.

Если λ — некоторое число, а $A=(a_{ij})$ — матрица, то соотношение $\lambda A=(\lambda a_{ij})$ определяет *скалярное произведение* матрицы на число, которое также выполняется поэлементно.

Можно определить *вычитание* матриц как сложение с матрицей, умноженной на -1: $A-B=A+(-B)=A+(-1)B$.

Матричное умножение определяется следующим образом. Матрицы A и B могут быть перемножены, если они совместимы в том смысле, что число столбцов A равно числу строк B (в общем случае выражение, содержащее матричное произведение AB, всегда подразумевает совместимость матриц A и B). Если $A=(a_{ij})$ — матрица размером $m\times n$, а $B=(b_{ij})$ — матрица размером $n\times p$, то их произведение $C=AB$ представляет собой матрицу $C=(c_{ij})$ размером $m\times p$,

элементы которой определяются соотношением $c_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk}$ для $i = 1, 2, \ldots, m$ и $k = 1, 2, \ldots, p$.

Матрицы обладают многими (хотя и не всеми) алгебраическими свойствами, присущими обычным числам. Единичная матрица является нейтральным элементом по отношению к умножению:

$$I_m A = A I_n = A$$

для любой матрицы A размером $m \times n$. Умножение на нулевую матрицу дает нулевую матрицу:

$$A0 = 0A = 0.$$

Умножение матриц ассоциативно:

$$A(BC) = (AB)C$$

для любых совместимых матриц A, B и C. Умножение матриц дистрибутивно относительно сложения:

$$A(B + C) = AB + AC,$$
$$(B + C)D = BD + CD.$$

Для $n > 1$ умножение матриц размером $n \times n$ не коммутативно, т.е. в общем случае $AB \neq BA$.

Произведения матрицы и вектора или двух векторов легко вычисляются, если вспомнить, что вектор — это матрица размером $n \times 1$ (или $1 \times n$ для вектора-строки). Например, если x и y — векторы размера n, то произведение

$$x^{\mathrm{T}} y = \sum_{i=1}^{n} x_i y_i$$

представляет собой матрицу размером 1×1 (т.е., по сути, одно число), которое называется *скалярным произведением* векторов x и y. Матрица же $Z = xy^{\mathrm{T}}$ размером $n \times n$ с элементами $z_{ij} = x_i y_j$ является *тензорным произведением* этих же векторов. *Норма* $\|x\|$ вектора x размером n определяется как длина вектора в n-мерном евклидовом пространстве

$$\|x\| = \sqrt{x_1^2 + x_2^2 + \cdots x_n^2} = \sqrt{x^{\mathrm{T}} x}.$$

Обратные матрицы, ранги и определители

Матрицей, *обратной* к данной матрице A размером $n \times n$, является матрица того же размера A^{-1} (если таковая существует), такая что $AA^{-1} = I_n = A^{-1}A$. Некоторые ненулевые квадратные матрицы не имеют обратных матриц. Матрица, для которой не существует обратной матрицы, называется *вырожденной*. Если же обратная матрица существует, то она единственная. Если A и B — невырожденные матрицы размером $n \times n$, то $(AB)^{-1} = A^{-1}B^{-1}$. Кроме того, операция обращения коммутативна с операцией транспонирования:

$$\left(A^{-1}\right)^{\mathrm{T}} = \left(A^{\mathrm{T}}\right)^{-1}.$$

Векторы $x_1, x_2, \ldots, x_n$ *линейно зависимы*, если существуют такие коэффициенты $c_1, c_2, \ldots, c_n$, не равные одновременно нулю, что $c_1x_1 + c_2x_2 + \ldots + c_nx_n = 0$. Например, векторы $x_1 = (1\ \ 2\ \ 3)$, $x_2 = (2\ \ 6\ \ 4)$ и $x_3 = (4\ \ 11\ \ 9)$ линейно зависимы, поскольку $2x_1 + 3x_2 - 2x_3 = 0$.

Столбцовым рангом ненулевой матрицы A размером $m \times n$ называется размер наибольшего множества линейно независимых столбцов A. Аналогично *строчным рангом* той же матрицы называется размер наибольшего множества линейно независимых строк A. Фундаментальным свойством любой матрицы A является равенство ее строчного и столбцового рангов, так что обычно говорят о просто *ранге* матрицы. Эквивалентное определение ранга ненулевой матрицы A размером $m \times n$ — это наименьшее число r, для которого существуют матрицы B и C размером соответственно $m \times r$ и $r \times n$ такие, что $A = BC$.

Квадратная матрица размером $n \times n$ имеет *полный ранг*, если ее ранг равен n. Полный ранг квадратная матрица имеет тогда и только тогда, когда она невырождена.

Минором элемента a_{ij} (ij-минор) матрицы A размером $n \times n$ ($n > 1$) называется матрица $A_{[ij]}$ размером $(n-1) \times (n-1)$, которая получается из A удалением i-й строки и j-го столбца. *Определитель*, или *детерминант*, матрицы A размером $n \times n$ можно определить рекурсивно при помощи миноров следующим образом:

$$\det(A) = \begin{cases} a_{11} & \text{при } n = 1, \\ \sum_{j=1}^{n} (-1)^{1+j} a_{1j} \det\left(A_{[1j]}\right) & \text{при } n > 1. \end{cases}$$

Множитель $(-1)^{i+j}\det(A_{[ij]})$ называется *алгебраическим дополнением* элемента a_{ij}.

Определитель квадратной матрицы A обладает следующими свойствами.

- Если любая строка или любой столбец A нулевой, то $\det(A)=0$.
- Если все элементы одного произвольного столбца (или строки) матрицы умножаются на λ, то ее определитель также умножается на λ.
- Определитель матрицы A остается неизменным, если все элементы одной строки (или столбца) прибавить к элементам другой строки (столбца).
- Определитель матрицы A равен определителю транспонированной матрицы A^{T}.
- Определитель матрицы A умножается на -1, если обменять местами любые два ее столбца (или строки).

Кроме того, для любых квадратных матриц A и B $\det(AB)=\det(A)\det(B)$. Квадратная матрица A вырождена тогда и только тогда, когда $\det(A)=0$.

Умножение матриц

Как было сказано выше, если $A=(a_{ij})$ — матрица размером $m\times n$, а $B=(b_{ij})$ — матрица размером $n\times p$, то их произведение $C=AB$ представляет собой матрицу $C=(c_{ij})$ размером $m\times p$, элементы которой определяются соотношением $c_{ik}=\sum_{j=1}^{n}a_{ij}b_{jk}$ для $i=1,2,\ldots,m$ и $k=1,2,\ldots,p$. Отсюда непосредственно следует простейший алгоритм умножения матриц.

```
Алгоритм MatrixMultiply(A,B)
// Входные данные : матрица A размером m×n и матрица B
//                  размером n×p
// Выходные данные: матрица C размером m×p,
//                  представляющая собой матричное
//                  произведение AB
for i = 1 to m do
    for j = 1 to p do
```

```
    begin
        c_ij = 0
        for k = 1 to n do c_ij = c_ij + a_ik*b_kj
    end
return C
```

Для реализации как этого, так и прочих алгоритмов, связанных с матрицами, используем следующий простейший класс для представления матриц (мы уже сталкивались с таким классом при работе с графами).

```
class matrix
{
public:
    matrix(int m, int n)
        :Rows_(m, vector<double>(n,0.0)) {}
    const vector<double>& operator[](int i) const
                            { return Rows_[i]; }
    vector<double>& operator[](int i)
                            { return Rows_[i]; }
    int rows() const { return Rows_.size(); }
    int cols() const { return Rows_[0].size(); }
private:
    vector< vector<double> > Rows_;
};
```

Данный класс обеспечивает только обращение к элементам матрицы при помощи двойного индексирования, а также информацию о количестве строк и столбцов матрицы. Никакие другие действия с матрицами в данной главе выполняться не будут, так что с методической точки зрения использование такого непритязательного класса вполне оправданно. Методические же соображения поясняют и выбор класса **vector** в качестве хранилища данных (а не использования для этой цели класса **valarray**). При разработке собственных классов и реализаций алгоритмов на C++ для работы с матрицами можно обратиться к разделу 22.4 книги [19], где вы найдете немало полезных советов по этому поводу.

Умножение матриц с использованием определения операции матричного умножения реализуется функцией в несколько строк.

```
matrix MatrixMultiply(matrix& A, matrix& B)
{
    assert(A.cols() == B.rows());
    matrix C(A.rows(),B.cols());
    for(int i = 0; i < A.rows(); ++i)
        for(int j = 0; j < B.cols(); ++j)
        {
            C[i][j] = 0.0; // Данная строка излишня,
                           // так как при создании матрица
                           // инициализируется нулями
            for(int k = 0; k < A.cols(); ++k)
                C[i][j] += A[i][k]*B[k][j];
        }
    return C;
}
```

Три вложенных цикла ясно говорят о том, что временная эффективность данного алгоритма — $\Theta(n^3)$. Кажется, что улучшить эффективность алгоритма умножения невозможно, но это не так. Существует замечательный (в первую очередь с теоретической точки зрения) рекурсивный алгоритм умножения матриц размера $n \times n$, разработанный в 1969 году Штрассеном и имеющий время работы $\Theta(n^{\log_2 7}) \approx \Theta(n^{2.807})$. Здесь не будет приведен полный вывод алгоритма Штрассена; с ним можно познакомиться в [12]. Алгоритм основан на методе декомпозиции. Рассмотрим произведение матриц $C = AB$, где каждая матрица имеет размер $n \times n$. Считая, что n является точной степенью 2, поделим каждую из матриц на четыре матрицы размером $n/2 \times n/2$ и перепишем произведение следующим образом:

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & f \\ g & h \end{pmatrix}.$$

Алгоритм Штрассена вычисляет с помощью семи (а не восьми, как потребовалось бы в традиционном алгоритме) умножений следующие значения: $P_1 = a \cdot (f - h)$, $P_2 = (a + b) \cdot h$, $P_3 = (c + d) \cdot e$, $P_4 = d \cdot (g - e)$, $P_5 = (a + d) \cdot (e + h)$, $P_6 = (b - d) \cdot (g + h)$, $P_7 = (a - c) \cdot (e + f)$. После этого результат умножения матриц получается следующим образом:

$s = P_1 + P_2$, $t = P_3 + P_4$, $r = P_5 + P_4 - P_2 + P_6$ и $u = P_5 + P_1 - P_3 - P_7$.

Однако алгоритм Штрассена редко применяется на практике по следующим причинам.

1. Постоянный множитель, скрытый во времени работы алгоритма Штрассена, существенно превышает постоянный множитель во времени работы $\Theta(n^3)$ простого алгоритма умножения.
2. Для разреженных матриц имеются специализированные более эффективные методы умножения.
3. Алгоритм Штрассена несколько менее численно устойчив, чем простой алгоритм умножения матриц.
4. Построение подматриц на каждом шаге рекурсии приводит к повышенному расходу памяти.

Написать сколь-нибудь эффективную реализацию алгоритма Штрассена с использованием «игрушечного» демонстрационного класса `matrix` нереально — на каждом шаге рекурсии выигрыш от снижения количества умножений будет полностью нивелироваться «перетасовкой» большого количества данных в памяти, накладными расходами на обслуживание классов и т.п. Так что такая реализация будет проигрывать в эффективности реализации простого алгоритма перемножения матриц, основанного на определении этой операции. Таким образом, здесь мы ограничимся только изложением самого алгоритма. К этому следует добавить, что имеется теорема, согласно которой умножение матриц и обращение матрицы — задачи одного уровня сложности, так что обращение матрицы требует того же асимптотического времени, что и перемножение матриц.

В настоящее время наиболее эффективный алгоритм перемножения матриц Копперсмита–Винограда имеет эффективность $\Theta(n^{2.376})$. Вопрос о максимально возможной эффективности перемножения матриц остается открытым (пока что очевидна лишь нижняя граница — $O(n^2)$, но неизвестно, в какой мере она достижима).

Метод исключения Гаусса

Решение систем линейных уравнений представляет собой фундаментальную задачу, возникающую в различных приложениях. Итак, нам требуется решить систему из n уравнений с n неизвестными, где n — некоторое, обычно достаточно большое число:

$$
\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\
&\vdots \\
a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n
\end{aligned}
$$

Имеется элегантный алгоритм решения систем линейных уравнений, который называется *методом исключения Гаусса*. Идея этого метода заключается в преобразовании системы n линейных уравнений с n неизвестными в эквивалентную ей систему с верхнетреугольной матрицей коэффициентов:

$$
\begin{array}{ccccccccc}
a_{11}x_1 & + & a_{12}x_2 & + & \cdots & + & a_{1n}x_n & = & b_1 \\
a_{21}x_1 & + & a_{22}x_2 & + & \cdots & + & a_{2n}x_n & = & b_2 \\
\vdots & & & & & & & & \\
a_{n1}x_1 & + & a_{n2}x_2 & + & \cdots & + & a_{nn}x_n & = & b_n \\
 & & & & \Downarrow & & & & \\
a'_{11}x_1 & + & a'_{12}x_2 & + & \cdots & + & a'_{1n}x_n & = & b'_1 \\
 & & a'_{22}x_2 & + & \cdots & + & a'_{2n}x_n & = & b'_2 \\
 & & & & & & & & \vdots \\
 & & & & & & a'_{nn}x_n & = & b'_n
\end{array}
$$

Используя матричные обозначения, это действие можно записать как

$$Ax = b \;\Rightarrow\; A'x = b',$$

где

$$
A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & & \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix},\quad
b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix},
$$

$$
A' = \begin{bmatrix} a'_{11} & a'_{12} & \cdots & a'_{1n} \\ 0 & a'_{22} & \cdots & a'_{2n} \\ \vdots & & & \\ 0 & 0 & \cdots & a'_{nn} \end{bmatrix},\quad
b' = \begin{bmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_n \end{bmatrix}.
$$

(К элементам матрицы и свободным членам новой системы линейных уравнений добавлены штрихи для того, чтобы подчеркнуть отличие этих значений от значений их аналогов в исходной системе линейных уравнений.)

Система линейных уравнений с верхнетреугольной матрицей коэффициентов существенно лучше системы линейных уравнений с произвольной матрицей, потому что систему линейных уравнений с верхнетреугольной матрицей легко решить методом обратной подстановки. Сначала мы вычисляем значение x_n из последнего уравнения; затем подставляем полученное значение в предыдущее уравнение и получаем значение x_{n-1}. Выполняя такие подстановки вычисленных значений переменных в очередные уравнения, мы получим значения всех n переменных — от x_n до x_1.

Осталась «мелочь» — получить из системы линейных уравнений с произвольной матрицей коэффициентов A эквивалентную систему линейных уравнений с верхнетреугольной матрицей A'. Это можно сделать при помощи последовательности следующих элементарных операций:

- обмена двух уравнений системы линейных уравнений;
- умножения уравнения на ненулевую величину;
- замены уравнения на сумму или разность этого уравнения и другого уравнения, умноженного на некоторую величину.

Поскольку ни одна из перечисленных операций не изменяет решение системы линейных уравнений, любая система линейных уравнений, полученная из исходной при помощи серии описанных операций, будет иметь то же решение, что и исходная.

Начнем наши преобразования. Для начала используем в качестве опорного элемента a_{11} для того, чтобы сделать все коэффициенты при x_1 в строках ниже первой нулевыми. В частности, заменим второе уравнение разностью между ним и первым уравнением, умноженным на a_{21}/a_{11} для того, чтобы получить нулевой коэффициент при x_1. Выполняя то же для третьей, четвертой и далее строк и умножая первое уравнение соответственно на a_{31}/a_{11}, a_{11}/a_{11}, ..., a_{m1}/a_{11}, сделаем все коэффициенты при x_1 в уравнениях ниже первого равными 0. Затем обнулим все коэффициенты при x_2 в уравнениях ниже второго, вычитая из каждого из этих уравнений второе, умноженное на соответствующий коэффициент. Повторяя эти действия для каждой

из первых $n-1$ строк, получим систему линейных уравнений с верхнетреугольной матрицей коэффициентов.

Заметим, что можно работать не с двумя матрицами — коэффициентов и свободных членов, а только с матрицей коэффициентов, к которой в качестве $(n+1)$-го столбца добавлены свободные члены системы линейных уравнений. Другими словами, нет необходимости явно использовать имена переменных системы линейных уравнений или знаки + и =.

Вот как можно записать псевдокод только что рассмотренного этапа преобразования алгоритма решения систем линейных уравнений методом исключения Гаусса.

```
Алгоритм GaussElimination(A)
// Входные данные: Матрица A[1..n,1..n+1], в которой
//                  (n+1)-й столбец представляет
//                  собой столбец свободных членов
// Выходные данные: эквивалентная верхнетреугольная
//                  матрица на месте матрицы A со
//                  значениями в (n+1)-м столбце,
//                  соответствующими свободным
//                  членам новой системы линейных
//                  уравнений.
for i = 1 to n-1 do
    for j = i+1 to n do
        for k = i to n+1 do
            a_jk = a_jk - a_ik*a_ji/a_ii
return A
```

Однако этот метод не всегда корректен: если $a_{ii}=0$, то нельзя выполнить деление на этот элемент и, следовательно, использовать i-ю строку в качестве опорной на i-й итерации алгоритма. В этом случае мы должны воспользоваться первой из описанных ранее операций и обменять i-ю строку с одной из строк ниже ее, у которой в i-м столбце находится ненулевой элемент (если система линейных уравнений имеет единственное решение, то такая строка должна существовать).

Поскольку мы все равно должны быть готовы к возможному обмену строк, следует учесть еще одну потенциальную сложность — возможность того, что величина a_{ii} будет столь мала (и, соответственно, столь велик коэффициент a_{ji}/a_{ii}), что новое значение a_{jk} может оказаться искаженным ошибкой округления, связанной с вычитанием

двух сильно отличающихся чисел. Чтобы избежать этой проблемы, можно всегда выбирать строку с наибольшим абсолютным значением коэффициента в *i*-м столбце для обмена с *i*-й строкой, а затем использовать ее в качестве опорной на *i*-й итерации. Такая модификация алгоритма, называющаяся ***выбором ведущего элемента***, гарантирует, что значение масштабирующего множителя никогда не превысит 1.

Измененный с учетом рассмотренных замечаний псевдокод приведен ниже.

```
Алгоритм AdvGaussElimination(A)
// Входные данные:  матрица A[1..n,1..n+1], в которой
//                    (n+1)-ый столбец представляет
//                    собой столбец свободных членов
// Выходные данные: эквивалентная верхнетреугольная
//                    матрица на месте матрицы A со
//                    значениями в (n+1)-м столбце,
//                    соответствующими свободным
//                    членам новой системы линейных
//                    уравнений.
for i = 1 to n-1 do
begin
    pivotrow = i
    for j = i+1 to n do
    begin
        if |a_ji|>|a_pivotrow,i| pivotrow = j
    end
    for k = i to n+1 do swap(a_ik, a_pivotrow,k)
    for j = i+1 to n do
    begin
        temp = a_ji/a_ii
        for k = i to n+1 do a_jk = a_jk - a_ik*temp
    end
end
return A
```

Рассмотрение структуры псевдокода подсказывает, что временная эффективность данного алгоритма — $\Theta(n^3)$. Поскольку временная эффективность второй стадии решения системы линейных уравнений (обратной подстановки) алгоритма исключения Гаусса равна $\Theta(n^2)$, общее время работы алгоритма определяется кубиче-

ским временем стадии исключения, так что алгоритм исключения Гаусса — кубический.

Теоретически метод исключения Гаусса всегда либо дает точное решение системы линейных уравнений (если она имеет единственное решение), либо выясняет, что такого решения не существует. В последнем случае система линейных уравнений может либо не иметь решения вовсе, либо иметь бесконечно много решений. На практике решение систем большого размера данным методом наталкивается на трудности, в первую очередь связанные с накоплением ошибок округления, но этот вопрос выходит за рамки данной книги.

Поиск решения системы линейных уравнений после выполнения исключения Гаусса тривиален: переменные x_i вычисляются поочередно от x_n до x_1 подстановкой в соответствующие уравнения уже известных значений:

$$x_i = \frac{a_{i,n+1} - \sum_{j=i+1}^{n} a_{ij} x_j}{a_{ii}}.$$

Все изложенное выше с применением представленного ранее класса **matrix** и класса **vector** для вектора свободных членов реализуется в виде следующей функции.

```
vector<double> Gauss(const matrix& F,
                     vector<double> B)
{
    assert(F.rows()==F.cols());
    assert(F.rows()==B.size());
    int n = F.rows();
    matrix A(n,n+1);
    vector<double> x(n);
    // Создание новой матрицы
    for(int i = 0; i < n; ++i)
    {
        for(int j = 0; j <= n; ++j)
            A[i][j] = F[i][j];
        A[i][n] = B[i];
    }
    // Исключение Гаусса с выбором ведущего элемента
    for(int i = 0; i < n - 1; ++i)
    {
```

```
        int pivot = i;
        for(int j = i+1; j < n; ++j)
            if (fabs(A[j][i]) > fabs(A[pivot][i]))
                                        pivot = j;
        if (pivot != i) swap(A[i],A[pivot]);
        for(int j = i+1; j < n; ++j)
        {
            assert((A[i][i] != 0.0) &&
                   "Система решений не имеет");
            double t = A[j][i]/A[i][i];
            for(int k = i; k <= n; ++k)
                        A[j][k] -= A[i][k]*t;
        }
    }
    assert((A[n-1][n-1] != 0.0) &&
           "Система решений не имеет");
    // Просчет решения системы линейных уравнений
    for(int i = n-1; i >= 0; --i)
    {
        double sum = 0.0;
        for(int j = i+1; j < n; ++j)
            sum += A[i][j]*x[j];
        x[i] = (A[i][n] - sum)/A[i][i];
    }
    return x;
}
```

Обращение матрицы

Рассмотрим теперь применение метода исключения Гаусса для вычисления обратной матрицы. В соответствии с определением обратной матрицы для того, чтобы найти ее для невырожденной матрицы A размером $n \times n$, требуется найти n^2 чисел x_{ij}, $1 \le i, j \le n$ таких, что

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & & \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & & & \\ x_{n1} & x_{n2} & \cdots & x_{nn} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & & & \\ 0 & 0 & \cdots & 1 \end{bmatrix}.$$

Найти эти неизвестные числа можно, решая n систем линейных уравнений с одной и той же матрицей коэффициентов A, у которых векторы неизвестных представляют собой столбцы обратной матрицы, а векторы свободных членов — столбцы единичной матрицы. Эти системы линейных уравнений можно решить, применяя метод исключения Гаусса к матрице A, расширенной добавлением к ней единичной матрицы размером $n \times n$.

Временная эффективность этого алгоритма, как нетрудно заметить, также равна $\Theta(n^3)$, просто в этом случае обратная подстановка требует не квадратичного, а кубического времени работы.

Поскольку мы уже рассматривали псевдокод преобразования методом исключения Гаусса, нет смысла повторяться, так что далее приводится только реализация этого алгоритма обращения матрицы.

```
matrix Inverse(const matrix& F)
{
    assert(F.rows()==F.cols());
    int n = F.rows();
    matrix A(n,2*n);
    matrix I(n,n);
    vector<double> x(n);
    for(int i = 0; i < n; ++i)
    {
        for(int j = 0; j <= n; ++j)
            A[i][j] = F[i][j];
        A[i][i+n] = 1.0;
    }
    for(int i = 0; i < n - 1; ++i)
    {
        int pivot = i;
        for(int j = i+1; j < n; ++j)
            if (fabs(A[j][i]) > fabs(A[pivot][i]))
                pivot = j;
        if (pivot != i) swap(A[i],A[pivot]);
        for(int j = i+1; j < n; ++j)
        {
            assert((A[i][i] != 0.0) &&
                   "Матрица вырождена");
```

```
            double t = A[j][i]/A[i][i];
            for(int k = i; k < 2*n; ++k)
                        A[j][k] -= A[i][k]*t;
        }
    }
    assert((A[n-1][n-1] != 0.0) &&
           "Матрица вырождена");

    for(int column = 0; column < n; ++column)
    {
        for(int i = n-1; i >= 0; --i)
        {
            double sum = 0.0;
            for(int j = i+1; j < n; ++j)
                    sum += A[i][j]*I[j][column];
            I[i][column] =
                    (A[i][column+n] - sum)/A[i][i];
        }
    }
    return I;
}
```

Вычисление определителя

Обратившись к свойствам определителей в начале этой главы (см. стр. 160), можно увидеть, что в процессе применения метода исключения Гаусса определитель матрицы остается неизменным по абсолютному значению и только меняет знак при обмене местами двух строк матрицы. По окончании работы получается треугольная матрица, определитель которой легко вычисляется как произведение элементов, стоящих на ее главной диагонали. Таким образом, очень легко модифицировать код любой из приведенных функций, чтобы получить функцию для вычисления определителя квадратной матрицы.


```
double Det(const matrix& F)
{
    assert(F.rows()==F.cols());
    int n = F.rows();
    matrix A(F);
```

```
    double det = 1.0;
    for(int i = 0; i < n - 1; ++i)
    {
        int pivot = i;
        for(int j = i+1; j < n; ++j)
            if (fabs(A[j][i]) > fabs(A[pivot][i]))
                pivot = j;
        if (pivot != i)
        {
            det = -det;
            swap(A[i],A[pivot]);
        }
        for(int j = i+1; j < n; ++j)
        {
            assert((A[i][i] != 0.0) &&
                   "Матрица вырождена");
            double t = A[j][i]/A[i][i];
            for(int k = i; k < 2*n; ++k)
                A[j][k] -= A[i][k]*t;
        }
    }
    for(int i = 0; i < n; ++i)
    {
        det *= A[i][i];
    }
    return det;
}
```

Очевидно, что внесенные изменения никак не влияют на асимптотическое время работы функции, которое по-прежнему равно $\Theta(n^3)$.

Трехдиагональная система линейных уравнений

Метод исключения Гаусса решает системы линейных уравнений общего вида, однако зачастую приходится сталкиваться с различными, достаточно широко распространенными частными случаями систем линейных уравнений, которые могут быть решены существенно более эффективно. В качестве примера рассмотрим системы линейных уравнений с трехдиагональной матрицей коэффициентов.

Например, весьма часто при решении краевых задач разностным методом, при построении сплайнов и тому подобных задачах возникает система линейных уравнений следующего специального вида:

$$A_j x_{j-1} - C_j x_j + B_j x_{j+1} = F_j,\ j = 1, 2, \ldots, n-1,$$

$$x_0 = \lambda_0 x_1 + \nu_0,\ x_n = \lambda_n x_{n-1} + \nu_{n-1},$$

где $x_0, x_1, \ldots, x_n$ — неизвестные, $A_j, B_j, C_j, F_j, \lambda_j, \nu_j$ — заданные числа, причем $|C_j| \geq |A_j| + |B_j| \geq |A_j| > 0$, $|\lambda_0| < 1$, $|\lambda_n| \leq 1$. Последние три неравенства гарантируют существование единственного решения приведенной системы линейных уравнений (которая называется краевой задачей для трехточечного разностного уравнения).

Данную систему линейных уравнений можно записать в матричном виде как $Ax = b$, где $x = (x_0, x_1, \ldots, x_n)^T$, $b = (\nu_0, F_1, F_2, \ldots, F_n, \nu_n)^T$, и

$$A = \begin{pmatrix} 1 & -\lambda_0 & & & & 0 & \\ A_1 & -C_1 & B_1 & & & & \\ & A_2 & -C_2 & B_2 & & & \\ & & & \ddots & & & \\ & & & & A_{n-1} & -C_{n-1} & B_{n-1} \\ & 0 & & & & -\nu_n & 1 \end{pmatrix}.$$

Решить такую систему можно *методом прогонки*. Подставив уравнение $x_0 = \lambda_0 x_1 + \nu_0$ в первое уравнение системы, получим

$$A_1(\lambda_0 x_0 + \nu_0) - C_1 x_1 + B_1 x_2 = F_1$$

или

$$x_1 = \lambda_1 x_2 + \nu_1,$$

где

$$\lambda_1 = \frac{B_1}{C_1 - A_1\lambda_0},\ \nu_1 = \frac{A_1\nu_0 - F_1}{C_1 - A_1\lambda_0}.$$

Полученное для x_1 выражение можно подставить во второе уравнение системы и получить уравнение, связывающее x_2 и x_3, и т.д. Допустим, что уже найдено соотношение

$x_{k-1} = \lambda_{k-1} x_k + \nu_{k-1}$ ($k < n-1$). Подставляя это выражение в k-е уравнение системы, получим $A_k(\lambda_{k-1} x_k + \nu_{k-1}) - C_k x_k + B_k x_{k+1} = F_k$, так что

$$x_k = \lambda_k x_{k+1} + \nu_k, \text{ где } \lambda_k = \frac{B_k}{C_k - A_k \lambda_{k-1}}, \ \nu_k = \frac{A_k \nu_{k-1} - F_k}{C_k - A_k \lambda_{k-1}}.$$

Таким образом, мы осуществляем прямой проход по всем значениям $k = 1, 2, \ldots, n$, после чего, подставляя в последнее полученное соотношение уравнение $x_n = \lambda_n x_{n-1} + \nu_{n-1}$, получим уравнение $x_n = \lambda_n(\lambda_{n-1} x_n + \nu_{n-1}) + \nu_n$, из которого сможем найти неизвестное

$$x_n = \frac{\nu_n + \lambda_n \nu_{n-1}}{1 - \nu_n \nu_{n-1}}.$$

Теперь можно выполнить обратный проход, получая поочередно значения $x_{n-1}, x_{n-2}, \ldots, x_1, x_0$.

Совершенно очевидно, что каждый шаг как прямого, так и обратного прохода выполняется за время $\Theta(1)$, так что общее время решения трехдиагональной системы линейных уравнений методом прогонки составляет $\Theta(n)$, что существенно лучше времени метода исключения Гаусса $\Theta(n^3)$.

Указанные в начале этого подраздела условия существования единственного решения достаточные, но не необходимые. Однако большинство систем линейных уравнений, возникающих в реальных задачах, удовлетворяют этим условиям.

Что касается реализации метода прогонки, то в общем виде она, как правило, не имеет смысла, так как коэффициенты матрицы очень часто могут легко вычисляться в процессе решения системы линейных уравнений, а не храниться в матрице (или с целью сохранения памяти — в трех векторах). Именно с таким случаем мы сталкиваемся в демонстрационном примере, рассмотренном далее.

Рассмотрим численное решение уравнения $y'' = 2\cos x - y$ на отрезке $[0, \pi/2]$ с краевыми условиями $y(0) = y(\pi/2) = 0$ (точное решение поставленной задачи — $y = (x - \pi/2)\sin x$).

Из краевых условий очевидно, что $\lambda_0 = \lambda_n = 0$, $\nu_0 = \nu_n = 0$. Запишем уравнение в виде конечной разности с шагом h с использованием формулы для вычисления второй производной по трем точкам (см. стр. 141):

$$\frac{y_{j+1} - 2y_j + y_{j-1}}{h^2} = 2\cos x_j - y_j .$$

Если переписать эту формулу как

$$y_{j+1} - (2 - h^2)y_j + y_{j-1} = 2h^2 \cos x_j ,$$

становится очевидно, что для всех j $A_j = B_j = 1$, $C_j = 2 - h^2$, $F_j = 2h^2 \cos jh$, так что не имеет смысла хранить эти коэффициенты ни в векторах, ни тем более в матрице — их следует вычислять непосредственно в процессе решения системы линейных уравнений. Обратите внимание на то, что полученные коэффициенты удовлетворяют приведенным в начале раздела условиям существования единственного решения системы линейных уравнений.

Представленный ниже код решает рассмотренную выше систему линейных уравнений методом прогонки, после чего выводит на экран полученное и точное решения, что позволяет оценить степень погрешности нашего решения.

```
int main()
{
    const double Pi = 3.1415926;  // Число π
    const int    N  = 10;         // Количество
                                  // отрезков
    double       h  = Pi/(2.0*N); // Шаг расчета

    // Векторы решения, коэффициентов λ и ν
    vector<double> x(N+1), l(N+1), nu(N+1);

    // Краевые условия
    l[0] = 0.0; nu[0] = 0.0;
    l[N] = 0.0; nu[N] = 0.0;

    // Прямой проход
    for(int k = 1; k < N; ++k)
    {
        l[k] = 1.0/((2-h*h) - l[k-1]);
        nu[k] = (nu[k-1] - 2*h*h*cos(h*k))/
                        ((2-h*h) - l[k-1]);
    }
```

```
    // Обратный проход
    x[N] = (nu[N]+l[N]*nu[N-1])/(1.0-l[N]*l[N-1]);
    for(int k = N-1; k >= 0; --k)
    {
        x[k] = l[k]*x[k+1]+nu[k];
    }

    // Вывод результатов
    printf("  x       Решение     Точное решение\n");
    for(int k = 0; k <= N; ++k)
    {
        printf("%5.2lf    %8.5lf    %8.5lf\n",
               k*h, x[k], (k*h-Pi/2.0)*sin(k*h));
    }
}
```

Как видите, ничего принципиально сложного. Вот результат работы данной программы:

```
  x       Решение     Точное решение
 0.00      0.00000      0.00000
 0.16     -0.22219     -0.22115
 0.31     -0.39016     -0.38832
 0.47     -0.50157     -0.49919
 0.63     -0.55664     -0.55397
 0.79     -0.55804     -0.55536
 0.94     -0.51079     -0.50832
 1.10     -0.42192     -0.41988
 1.26     -0.30024     -0.29878
 1.41     -0.15590     -0.15515
 1.57      0.00000      0.00000
```

Как видите, полученное решение достаточно точное, несмотря даже на столь большой шаг вычислений.

Главный вывод, который следует сделать из данного материала, заключается в том, что существуют различные частные случаи систем линейных уравнений, для которых имеются существенно более эффективные алгоритмы решения, чем для решения систем линейных уравнений в общем случае.

LUP-разложение

При использовании метода исключения Гаусса мы привели исходную систему линейных уравнений в эквивалентную систему линейных уравнений с треугольной матрицей, что позволило легко найти решение при помощи обратной подстановки. Определенный минус такого решения в том, что оно, если можно так выразиться, разовое, и для нового столбца свободных членов при той же матрице коэффициентов требуется повторное применение метода исключения со временем работы $\Theta(n^3)$. Было бы неплохо, если бы можно было получить алгоритм, позволяющий один раз провести вычисления по методу исключения Гаусса, а затем для разных столбцов свободных членов быстро получать решения при помощи метода подстановки. Такой алгоритм существует и носит название LUP-разложения.

Идея, лежащая в основе LUP-разложения, состоит в поиске трех матриц L, U и P размером $n \times n$ таких, что $PA = LU$, где L — единичная нижнетреугольная матрица, U — верхнетреугольная матрица, а P — матрица перестановки. Матрицы L, U и P, удовлетворяющие уравнению $PA = LU$, называются *LUP-разложением* матрицы A. Любая невырожденная матрица A допускает такое разложение.

Преимущество LUP-разложения матрицы основано на простоте решения системы линейных уравнений с треугольной матрицей (каковыми и являются матрицы L и U). Найдя LUP-разложение матрицы A, мы можем решить исходную систему уравнений $Ax = b$ путем решения двух треугольных систем уравнений.

Умножая обе части уравнения $Ax = b$ на P, мы получим уравнение $PAx = Pb$, которое представляет собой исходную систему линейных уравнений с переставленными местами уравнениями. Поскольку $PA = LU$, получаем $LUx = Pb$.

Обозначим вектор Ux как y. Этот вектор y можно легко найти при помощи решения треугольной системы уравнений $Ly = Pb$ методом прямой подстановки, после чего, зная y, решение исходной системы линейных уравнений можно найти, решая еще одну треугольную систему линейных уравнений — $Ux = y$ методом обратной подстановки.

С обратной подстановкой мы уже имели дело в предыдущем разделе, так что нам надо просто заменить в соответствующей формуле элементы преобразованной матрицы a_{ij} коэффициентами верхнетреугольной матрицы u_{ij}, а свободные члены b_i — элементами вектора y:

$$x_i = \frac{y_i - \sum_{j=i+1}^{n} u_{ij} x_j}{u_{ii}}.$$

Применение прямой подстановки для решения системы линейных уравнений $Ly = Pb$ несколько осложняется тем, что сперва требуется вычислить произведение матрицы перестановки P и вектора свободных членов b. Однако, поскольку матрица P — это матрица перестановки, такое произведение сводится к перестановке элементов вектора b, которое можно выполнять прямо в процессе прямой подстановки. Кроме того, матрицу перестановки P в компактном виде удобно хранить как вектор $\pi[1..n]$. Элемент $\pi[i]$ ($i = 1, 2, \ldots, n$) указывает, что $P_{i,\pi[i]} = 1$ и $P_{ij} = 0$ при $j \neq \pi[i]$.

Поскольку L — единичная нижнетреугольная матрица, уравнение $Ly = Pb$ можно переписать как

$$\begin{aligned}
y_1 &= b_{\pi[1]},\\
l_{21}y_1 + y_2 &= b_{\pi[2]},\\
l_{31}y_1 + l_{32}y_2 + y_3 &= b_{\pi[3]},\\
&\vdots\\
l_{n1}y_1 + l_{n2}y_2 + l_{n3}y_3 + \ldots + y_n &= b_{\pi[n]}.
\end{aligned}$$

Значение y_1 определяется непосредственно, поскольку первое уравнение гласит, что $y_1 = b_{\pi[1]}$. Зная y_1, его можно подставить во второе уравнение и найти $y_2 = b_{\pi[2]} - l_{21}y_1$. Оба полученных значения при подстановке в третье уравнение дают $y_3 = b_{\pi[3]} - (l_{31}y_1 + l_{32}y_2)$, и так далее. Общая формула для элементов вектора y —

$$y_i = b_{\pi[i]} - \sum_{j=1}^{i-1} l_{ij} y_j.$$

Таким образом, если у нас есть матрицы L, U и представление матрицы перестановки P в виде вектора π, очень просто найти решение системы линейных уравнений для данного столбца свободных членов за время $\Theta(n^2)$. Вот псевдокод данного метода.

```
Алгоритм LUPSolution(L,U,π,b)
// Входные данные : матрицы L и U и вектор π,
//                  полученные при LUP-разложении
//                  матрицы системы линейных
//                  уравнений A, и столбец свободных
```

```
//                    членов b
// Выходные данные: вектор x, представляющий решение
//                    исходной системы линейных
//                    уравнений Ax=b
```

for i = 1 **to** n **do** $y_i = b_{\pi(i)} - \sum_{j=1}^{i-1} l_{ij} y_j$

for i = n **downto** 1 **do** $x_i = \left(y_i - \sum_{j=i+1}^{n} u_{ij} x_j\right) / u_{ii}$

Осталось вычислить само LUP-разложение матрицы исходной системы линейных уравнений. Математические основы приведенного далее алгоритма LUP-разложения здесь опущены, о них можно прочесть, например, в [12]. Сам алгоритм по сути представляет собой метод исключения Гаусса. Следует сделать только два замечания по поводу приведенного псевдокода. Первое — это то, что матрицу перестановок P мы получаем в компактном виде вектора перестановок π. Второе замечание касается того, что матрицы L и U имеют общие элементы только на главной диагонали, но так как матрица L — единичная нижнетреугольная матрица, ее элементы на главной диагонали предопределены — это единицы. Поэтому обе матрицы можно хранить в единственной матрице, что и делает приведенный алгоритм. Более того, матрицы L и U размещаются на месте переданной алгоритму матрице A, так что по окончании работы алгоритма

$$a_{ij} = \begin{cases} l_{ij} & \text{при } i > j, \\ u_{ij} & \text{при } i \le j. \end{cases}$$

```
Алгоритм LUPDecomposition(A)
// Входные данные : матрица A размером n×n
// Выходные данные: вектор π, представляющий матрицу
//                    перестановок, и матрицы L и U,
//                    размещающиеся на месте матрицы
//                    A, как описано выше
for i = 1 to n do π_i = i
for k = 1 to n do
begin
    p = 0
    for i = k to n do
    begin
        if |a_ik| > p then
```

```
        begin
            p = |a_ik|
            k' = i
        end
    end
    if p = 0 then error "Матрица вырождена"
    swap(π_k, π_k')
    swap(a_k, a_k')    // Обмен строк k и k' матрицы
    for i = k + 1 to n do
    begin
        a_ik = a_ik/a_kk
        for j = k + 1 to n do a_ij = a_ij - a_ik*a_kj
    end
end
```

Что касается реализации изложенных здесь алгоритмов, то здесь есть всего лишь одно отличие реализации от псевдокода — если внимательно рассмотреть псевдокод *LUPSolution*, то можно обнаружить, что при вычислении каждого значения x_i требуется только один элемент вектора y — а именно y_i, и больше ни в каких вычислениях этот элемент не участвует. Таким образом, при реализации можно обойтись без вспомогательного вектора y.

```
void LUPDecomposition(matrix& A, vector<int>& P)
{
    assert(A.rows() == A.cols());
    int n = A.rows();
    P.resize(n);
    for(int i = 0; i < n; ++i) P[i] = i;
    for(int k = 0; k < n; ++k)
    {
        double p = 0.0;
        int kk;
        for(int i = k; i < n; ++i)
        {
            if (fabs(A[i][k]) > p)
            {
                p = fabs(A[i][k]);
                kk = i;
```

```
            }
        }
        assert((p != 0.0) && "Матрица вырождена");
        swap(P[k],P[kk]);
        swap(A[k],A[kk]);
        for(int i = k+1; i < n; ++i)
        {
            A[i][k] /= A[k][k];
            for(int j = k + 1; j < n;++j)
                A[i][j] -= A[i][k]*A[k][j];
        }
    }
}

vector<double> LUPSolution(const matrix& A,
                           const vector<int>& p,
                           const vector<double>& b)
{
    assert(A.rows() == A.cols());
    assert(A.rows() == p.size());
    assert(A.rows() == b.size());
    vector<double> x(A.rows());
    int n = A.rows();
    for(int i = 0; i < n; ++i)
    {
        double sum = 0.0;
        for(int j = 0; j < i; ++j)
            sum += A[i][j]*x[j];
        x[i] = b[p[i]] - sum;
    }
    for(int i = n-1; i >= 0; --i)
    {
        double sum = 0.0;
        for(int j = i + 1; j < n; ++j)
        {
            sum += A[i][j]*x[j];
        }
        x[i] = (x[i] - sum)/A[i][i];
    }
```

```
    return x;
}
```

Как уже говорилось в предыдущем разделе, алгоритм решения системы линейных уравнений можно применить и для обращения матрицы. Само собой, сказанное в полной мере относится и к LUP-разложению.

Пусть имеется LUP-разложение матрицы A на три матрицы L, U и P такие, что $PA = LU$. Используя процедуру *LUPSolution*, можно решить уравнение вида $Ax = b$ за время $\Theta(n^2)$. Поскольку LUP-разложение зависит только от A, но не от b, можно использовать ту же процедуру для решения другой системы линейных уравнений вида $Ax = b'$ за то же время $\Theta(n^2)$. Таким образом, имея LUP-разложение матрицы A, можно решить k систем линейных уравнений с одной и той же матрицей A за время $\Theta(kn^2)$. Поскольку уравнение $AX = I_n$ можно рассматривать как множество из n различных систем линейных уравнений вида $Ax = b$, для их решения требуется время $\Theta(n^3)$. Поскольку LUP-разложение A также вычисляется за время $\Theta(n^3)$, задача обращения матрицы решается за время $\Theta(n^3)$.

Метод наименьших квадратов

Еще одна тема, связанная с решением систем линейных уравнений, — аппроксимация набора экспериментальных данных при помощи метода наименьших квадратов. Предположим, что у нас имеется множество из m точек $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$, где значения y_i содержат ошибки измерений. При этом требуется найти функцию $F(x)$ такую, что $y_i = F(x_i) + \eta_i$, причем эта функция минимизирует погрешности η_i (вопрос о том, что именно означает минимизация погрешностей, будет рассмотрен чуть позже). Мы ограничимся функциями, представляющими собой линейные комбинации базисных функций $f_i(x)$:

$$F(x) = \sum_{j=1}^{n} c_j f_j(x),$$

где количество слагаемых n и само множество базисных функций выбираются на основе знаний о предметной области рассматриваемой задачи. Если в качестве базисных функций выбираются $f_j(x) = x^{j-1}$, то искомая функция F представляет собой полином степени $n-1$:

$$F(x) = c_1 + c_2 x + c_3 x^2 + \cdots + c_n x^{n-1}.$$

При $n = m$ можно найти функцию F, которая удовлетворяет исходному соотношению с нулевыми погрешностями. Такой выбор функции F неудачен, поскольку учитывает и все погрешности измерений, что приводит к плохим результатам при использовании F для предсказания значений y для некоторого значения x. Обычно гораздо лучшие результаты получаются при значительно меньшем n, чем m, поскольку при этом происходит определенная «фильтрация» ошибок измерений. Для выбора значения n имеются определенные теоретические предпосылки, но данная тема лежит за пределами этой книги. Когда n выбрано, в результате получается переопределенная система линейных уравнений (т.е. система линейных уравнений, у которой количество уравнений превосходит количество неизвестных), приближенное решение которой требуется найти. Рассмотрим, каким образом это можно сделать.

Пусть A — матрица значений базисных функций в заданных точках:

$$A = \begin{pmatrix} f_1(x_1) & f_2(x_1) & \cdots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \cdots & f_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_m) & f_2(x_m) & \cdots & f_n(x_m) \end{pmatrix},$$

т.е. $a_{ij} = f_j(x_i)$, и пусть $c = (c_k)$ — искомый вектор коэффициентов размером n. Тогда

$$Ac = \begin{pmatrix} f_1(x_1) & f_2(x_1) & \cdots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \cdots & f_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_m) & f_2(x_m) & \cdots & f_n(x_m) \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix} = \begin{pmatrix} F(x_1) \\ F(x_2) \\ \vdots \\ F(x_m) \end{pmatrix}$$

представляет собой вектор размера m «предсказанных значений» y, а вектор $\eta = Ac - y$ — вектор *невязок* размера m.

Теперь мы вынуждены вновь вернуться к критерию минимизации ошибок приближения. Могут быть выбраны различные критерии — например, минимизация максимального отклонения или минимизация суммы абсолютных значений невязок, однако мы воспользуемся критерием минимальности нормы вектора ошибок η (что отражено в названии метода *наименьших квадратов*):

$$\|\eta\| = \left(\sum_{i=1}^{m} \eta_i^2 \right)^{1/2}.$$

Поскольку

$$\|\eta\|^2 = \|Ac - y\|^2 = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} c_j - y_i \right)^2,$$

можно минимизировать $\|\eta\|$, дифференцируя $\|\eta\|^2$ по всем c_k и приравнивая полученные производные к 0:

$$\frac{d\|\eta\|^2}{d c_k} = \sum_{i=1}^{m} 2 \left(\sum_{j=1}^{n} a_{ij} c_j - y_i \right) a_{ik} = 0.$$

Эти n уравнений эквивалентны одному матричному уравнению $(Ac - y)^{\mathrm{T}} A = 0$, которое в свою очередь эквивалентно уравнению $A^{\mathrm{T}}(Ac - y) = 0$, откуда $A^{\mathrm{T}} Ac = A^{\mathrm{T}} y$.

В математической статистике такое уравнение называется нормальным уравнением. Если матрица A имеет полный столбцовый ранг, то существует обратная матрица $(A^{\mathrm{T}} A)^{-1}$ (доказательство этого факта выходит далеко за рамки данной книги), и решение исходного уравнения имеет вид

$$c = \left((A^{\mathrm{T}} A)^{-1} A^{\mathrm{T}} \right) y = A^{+} y,$$

где $A^{+} = \left((A^{\mathrm{T}} A)^{-1} A^{\mathrm{T}} \right)$ — *псевдообратная* к A матрица. Понятие псевдообратной матрицы представляет собой естественное обобщение понятия обратной матрицы на случай не квадратной исходной матрицы.

На практике нормальное уравнение решается путем вычисления умножения $A^{\mathrm{T}} y$ с последующим поиском LUP-разложения $A^{\mathrm{T}} A$.

Зачастую к методу наименьших квадратов *методом линеаризации* приводятся сугубо нелинейные зависимости. Так, например, подбор коэффициентов для аппроксимации экспериментальных данных (x_i, y_i) функцией $y = ae^{bx}$ можно выполнить, прологарифмировав значения y и подбирая методом наименьших квадратов коэффициенты α и b в уравнении $\ln y = \alpha + bx$ для экспериментальных данных $(x_i, \ln y_i)$. После этого значение коэффициента a вычисляется как e^{α}. Однако необходимо заметить, что при линеаризации происходит изменение весов погрешностей измерений, так что решаемая задача несколько отличается от исходной. Кроме того, здесь ничего не было сказано о погрешности определения коэффициентов, их доверительных интервалах и т.п. Вы можете найти существенно более полное и строгое изложение этих вопросов в книге [15].

Глава 6

Комбинаторные алгоритмы

В некоторых задачах для решения приходится использовать метод исчерпывающего перебора всех возможных вариантов. Это может быть перебор всех перестановок из n элементов, всех сочетаний, содержащих m разных элементов из множества из n элементов, всех подмножеств данного множества и т.д. Метод исчерпывающего перебора может потребоваться для задач, для решения которых нет полиномиальных алгоритмов, задач небольшого размера, для которых проще сгенерировать все возможные варианты, чем реализовывать сложный полиномиальный алгоритм — словом, причины могут быть самыми разными; вопрос упирается в генерацию всех возможных комбинаторных элементов. Такие алгоритмы, как правило, пригодны только для небольших размеров задач в силу их экспоненциальной природы. Здесь мы рассмотрим генерацию только некоторых подобных комбинаторных объектов.

Вот пример задачи, требующей исчерпывающего перебора. Пусть у нас имеется ряд цифр 123456789, между которыми мы можем произвольным образом расставлять знаки арифметических операций, получая таким образом различные значения. Например, 123+4+5–6*7––89=1, 123+4–56–78+9=2 и т.д. Какое минимальное положительное число невозможно представить в виде такого выражения? Для какого числа имеется максимальное количество подобных представлений? Для решения задачи приходится прибегнуть к перебору всех возможных расстановок арифметических знаков между цифрами — всего 5^8=390625 вариантов, что не так уж и много для комбинаторных задач. Проще всего сгенерировать все возможные выражения при помощи восьми вложенных циклов, но такая простота не всегда возможна; простейший пример для данной задачи — при переменной длине строки цифр. (Здесь мы не будем решать эту задачу, приведем только ответ для заинтересовавшегося читателя: минимальное положительное число, которое нельзя представить указанным способом, — 910, наибольшее количество представлений — 147 — имеет число –9, по 145 способов имеют числа 10 и 45.)

Генерация всех подмножеств данного множества

Задача генерации всех подмножеств множества из n элементов тесно связана с генерацией последовательности всех 2^n n-битовых чисел. При сопоставлении каждому элементу множества своего бита в n-битовом числе любое подмножество однозначно отображается на n-битовое число, в котором единичные биты означают наличие в подмножестве соответствующих элементов множества, а нулевые — их отсутствие.

Естественно, проще всего получить все n-битовые числа простым прибавлением 1, так что простейшая программа для вывода всех подмножеств четырехэлементного множества $\{A_1, A_2, A_3, A_4\}$ выглядит следующим образом.

```
int main()
{
    const int N = 4;
    for(unsigned int L = 0; L < (1 << N); ++L)
    {
        cout << "{";
        unsigned int K = L;
        for(int i = 1; i <= N; ++i, K >>= 1)
        {
            if (K&0x1) cout << "  A" << i;
        }
        cout << "  }\n";
    }
}
```

Как и следовало ожидать, программа выводит все 16 подмножеств данного множества:

```
{  }
{  A1  }
{  A2  }
{  A1  A2  }
{  A3  }
{  A1  A3  }
{  A2  A3  }
{  A1  A2  A3  }
```

```
{  A4  }
{  A1  A4  }
{  A2  A4  }
{  A1  A2  A4  }
{  A3  A4  }
{  A1  A3  A4  }
{  A2  A3  A4  }
{  A1  A2  A3  A4  }
```

Однако с точки зрения реализации, пожалуй, более корректным было бы представление возвращаемой последовательности в виде некоторого итератора. Поскольку вся глава посвящена генерации той или иной последовательности, в ней будет использоваться представление последовательности в виде псевдоитератора, которое позволит использовать в программе при его применении конструкцию вида

```
for(Enumerator e; !e; ++e)
{
    ...=*e; // Получение очередного члена
            // последовательности
}
```

Естественно, нельзя считать такое представление универсальным, но для наших целей его вполне достаточно. Спорными являются вопросы корректности переопределения операторов `!` и `*`, но в данном случае их оказывается удобно переопределить именно так — как видите, при этом цикл выглядит очень просто, хотя, конечно, использование оператора `!` в состоянии привести в некоторое заблуждение.

В этом случае реализация выглядит следующим образом (само собой, максимальное значение `N` в этой реализации должно быть меньше количества битов в типе **`unsigned long`**).

```
class subset
{
public:
    subset(int N_):L(0),N(N_) {};
    bool operator          !(){ return L < (1<<N); }
    unsigned long operator *(){ return L; }
    void operator         ++(){ ++L; }
private:
    unsigned long L;
    int N;
```

```
};

int main()
{
    const int N = 4;
    for(subset s(4); !s; ++s)
    {
        cout << "{";
        unsigned long K = *s;
        for(int i = 1; K != 0; ++i, K >>= 1)
        {
            if (K&0x1) cout << "  A" << i;
        }
        cout << "  }\n";
    }
}
```

Упомянутое ограничение на размер множества не слишком строгое. Если воспользоваться 64-битовым целым числом, то это ограничение сведется к $N<64$. Это не так уж и мало — всего имеется 2^N различных подмножеств множества из N элементов, так что при $N=63$ мы получим последовательность из порядка $9 \cdot 10^{18}$ чисел. Даже если на генерацию каждого тратить один такт процессора с частотой 3 ГГц, то для полного перебора потребуется порядка 100 лет.

Однако на этом вопрос о генерации всех подмножеств данного множества рано считать закрытым. Обратите внимание на последовательность полученных подмножеств и их отличие друг от друга. Например, для перехода от второго к третьему подмножеству надо удалить из подмножества один элемент и добавить другой. Еще больше отличаются подмножества в восьмой и девятой строках — для получения очередного подмножества здесь надо удалить три элемента и добавить один. Естественным образом встает вопрос о получении такой последовательности чисел, в которой каждое очередное число отличается от предыдущего только одним битом. Такие последовательности носят название кодов Грея.

Хотя имеется очень много вариантов кодов Грея с указанным свойством, мы воспользуемся тем из них, который описан в [20] и наиболее быстро и просто реализуется с использованием нашего псевдоитератора. Двоичное

число B преобразуется в код Грея по формуле $G = B \oplus (B \quad 1)$, где оператор $\oplus$ – это оператор исключающего или, а битовый сдвиг – беззнаковый.

```
class graycode
{
public:
    graycode(int N_):L(0),N(N_) {};
    bool operator !(){ return L < (1<<N); }
    unsigned long operator *(){ return L^(L>>1); }
    void operator         ++(){ ++L; }
private:
    unsigned long L;
    int N;
};
```

Используя данный класс вместо **subset** в рассматривавшейся функции **main()**, мы получим следующий набор подмножеств.

```
{  }
{  A1  }
{  A1  A2  }
{  A2  }
{  A2  A3  }
{  A1  A2  A3  }
{  A1  A3  }
{  A3  }
{  A3  A4  }
{  A1  A3  A4  }
{  A1  A2  A3  A4  }
{  A2  A3  A4  }
{  A2  A4  }
{  A1  A2  A4  }
{  A1  A4  }
{  A4  }
```

Легко убедиться в том, что каждое последующее подмножество получается из предыдущего путем единственного добавления или удаления элемента.

Генерация всех перестановок

Как и ранее, будем считать, что множество переставляемых элементов — это просто множество целых чисел от 1 до n, которые в общем случае можно рассматривать как индексы элементов n-элементного множества.

Будем исходить из того, что задача меньшего размера, состоящая в генерации всех $(n-1)!$ перестановок, решена. Получить решение задачи генерации всех n перестановок можно путем вставки n в каждую из n возможных позиций среди элементов каждой из перестановок $n-1$ элементов. Все эти перестановки будут различны, а их общее количество будет равно $n(n-1)! = n!$. Так мы получаем все возможные перестановки исходного множества целых чисел от 1 до n.

Число n можно вставлять в ранее сгенерированные перестановки как слева направо, так и справа налево. Выгодно начинать вставку n в последовательность $12\ldots(n-1)$ справа налево и изменять направление всякий раз при переходе к новой перестановке множества $\{1,\ldots,n-1\}$.

Так, для двух чисел это дает перестановки

```
12  21
```

Применение этого же принципа для перестановок трех элементов даст нам

```
123 132 312  321 231 213
```

Преимущество этого способа генерации перестановок такое же, как и у кодов Грея — каждая перестановка получается из непосредственной предшественницы при помощи обмена местами только двух элементов.

Этот же порядок перестановок n элементов можно получить и без явной генерации перестановок для меньших значений n. Это можно сделать, связав с каждым компонентом k перестановки направление. Будем указывать это направление при помощи стрелки над рассматриваемым элементом, например:

$$\vec{3}\,\overleftarrow{2}\,\vec{4}\,\overleftarrow{1}.$$

Компонент k в такой перестановке с использованием стрелок называется мобильным, если стрелка указывает на меньшее соседнее число. Например, в перестановке $\vec{3}\,\overleftarrow{2}\,\vec{4}\,\overleftarrow{1}$ числа 3 и 4 мобильны, а 2 и 1 — нет.

Воспользовавшись понятием мобильного элемента, мы получаем описание алгоритма Джонсона–Троттера для генерации перестановок.

```
Алгоритм JohnsonTrotter(n)
// Входные данные: Натуральное число n
// Выходные данные: Список перестановок
//                  множества {1,...,n}
Инициализируем первую перестановку значением
while (пока) имеется мобильное число k do
begin
    Находим наибольшее мобильное число k
    Меняем местами k и соседнее целое число,
        на которое указывает стрелка у k
    Меняем направление стрелок у всех целых
        чисел, превышающих k
end
```

Вот пример использования этого алгоритма для $n = 3$ (наибольшее мобильное число показано полужирным шрифтом):

$$\overleftarrow{1}\,\overleftarrow{2}\,\overleftarrow{\mathbf{3}} \quad \overleftarrow{1}\,\overleftarrow{\mathbf{3}}\,\overleftarrow{2} \quad \overleftarrow{\mathbf{3}}\,\overleftarrow{1}\,\overleftarrow{2} \quad \overrightarrow{\mathbf{3}}\,\overleftarrow{2}\,\overleftarrow{1} \quad \overleftarrow{2}\,\overleftarrow{\mathbf{3}}\,\overleftarrow{1} \quad \overleftarrow{2}\,\overleftarrow{1}\,\overleftarrow{3}.$$

Вот (не самая эффективная, зато простая и понятная) реализация алгоритма Джонсона–Троттера:

```
class JTPermutation
{
public:
    JTPermutation(int N);
    bool operator        !() { return !done; }
    vector<int> operator *() { return v; }
    void operator       ++();
private:
    vector<int>  v;  // Текущая перестановка
    vector<bool> d;  // Массив стрелок
    bool done;
};

JTPermutation::JTPermutation(int N)
```

```
:v(N),d(N),done(false)
{
    for(int i = 0; i < N; ++i) v[i] = i;
}

void JTPermutation::operator ++()
{
    if (done) return;
    // Поиск максимального мобильного числа
    int maxmob = -1, idx = -1;
    for(int i = 0; i < v.size(); ++i)
    {
        if ((i > 0) && (v[i] > v[i-1]) &&
            (d[i] == false) && (maxmob < v[i]))
        {
            maxmob = v[i];
            idx = i;
        }
        if ((i < v.size()-1) && (v[i] > v[i+1]) &&
            (d[i] == true) && (maxmob < v[i]))
        {
            maxmob = v[i];
            idx = i;
        }
    }
    // Перестановки исчерпаны?
    if (maxmob < 0) { done = true; return; }
    // Обмен элементов
    swap(v[idx],v[idx+(d[idx]?1:-1)]);
    swap(d[idx],d[idx+(d[idx]?1:-1)]);
    // Замена стрелок
    for(int i = 0; i < v.size(); ++i)
        if (v[i] > maxmob) d[i] = !d[i];
}
```

Порядок перестановок, генерируемых алгоритмом Джонсона–Троттера, не совсем естественный; было бы более естественно, если бы последняя перестановка имела вид $n(n-1)\ldots 1$. Именно такая перестановка окажется последней, если перестановки будут упорядо-

чены в соответствии с лексикографическим порядком, т.е. порядком, в котором они были бы перечислены в словаре, если рассматривать цифры как буквы алфавита:

1 2 3 1 3 2 2 1 3 2 3 1 3 1 2 3 2 1

Каким образом можно сгенерировать перестановку, следующую за $a_1a_2\ldots a_{n-1}a_n$ в лексикографическом порядке? Если $a_{n-1} < a_n$, просто меняем местами два последних элемента (например, за 123 следует 132). Если $a_{n-1} > a_n$, следует обратиться к элементу a_{n-2}. Если $a_{n-2} < a_{n-1}$, мы должны переставить последние три элемента, минимально увеличивая $(n-2)$-й элемент, т.е. помещая на это место следующий превышающий a_{n-2} элемент, выбранный из a_{n-1} и a_n, и заполняя позиции $(n-1)$ и n оставшимися двумя из трех элементов a_{n-2}, a_{n-1} и a_n в возрастающем порядке. Например, за 132 следует 213, а за 231 — 312. В общем случае мы сканируем текущую перестановку справа налево в поисках первой пары соседних элементов a_i и a_{i+1} таких, что $a_i < a_{i+1}$ (и, следовательно, $a_{i+1} > \cdots > a_n$). Затем мы находим наименьший элемент из «хвоста», превышающий a_i, т.е. $\min\{a_j \mid a_j > a_i, j > i\}$, и помещаем его в позицию i; позиции с $(i+1)$-й по n-ю заполняются элементами $a_i, a_{i+1}, \ldots, a_n$, из которых изъят элемент для вставки в позицию i, в возрастающем порядке.

Этого описания алгоритма вполне достаточно для его реализации.

```
class LexPermutation
{
public:
    LexPermutation(int N);
    bool operator        !() { return !done; }
    vector<int> operator *() { return v; }
    void operator       ++();
private:
    vector<int>  v;
    bool done;
};

LexPermutation::LexPermutation(int N)
:v(N),done(false)
```

```
{
    for(int i = 0; i < N; ++i) v[i] = i;
}

void LexPermutation::operator ++()
{
    if (done) return;
    int i;
    for(i = v.size()-2; i >= 0; --i)
    {
        if (v[i] < v[i+1])
        {
            int j = v.size();
            while(v[--j] < v[i]);
            swap(v[i],v[j]);
            for(int k = i+1, l = v.size()-1; l > k;
                --l, ++k)
                    swap(v[k],v[l]);
            return;
        }
    }
    if (i < 0) done = true;
}
```

По поводу генерации всех перестановок в лексикографическом порядке можно заметить, что вся информация, которая нужна для получения очередной перестановки, содержится в предыдущей. Это означает, что можно реализовать генерацию в виде функции, получающей два итератора, которые определяют некоторый диапазон элементов множества, и выполняющей генерацию очередной перестановки «на месте», непосредственно в контейнере, указанном переданными итераторами. Именно такими функциями являются функции **`next_permutation`** и **`prev_permutation`** стандартной библиотеки C++.

Генерация всех сочетаний

Вот один из алгоритмов для генерации всех сочетаний из m элементов n-элементного множества в лексикографическом порядке.

```
Алгоритм LexCombinations(n,m)
// Входные данные: размер исходного множества n и
//                 размера генерируемых сочетаний
// Выходные данные: все сочетания из m элементов
//                  n-элементного множества в
//                  лексикографическом порядке
Создаем массив c[1..m+2]
for i = 1 to m do c[i] = i - 1
c[m+1] = n
c[m+2] = 0
while true do // Бесконечный цикл
begin
    Вывод сочетания c[m]...c[2]c[1]
    j = 1
    while c[j]+1 = c[j+1] do
    begin
        c[j] = j - 1
        j = j + 1
    end
    if j <= m then c[j] = c[j] + 1; else return
end
```

Элементы **c[m+1]** и **c[m+2]** введены искусственно и служат ограничителями. Обратите внимание на следующую особенность алгоритма: значение *n* в явном виде использовано в нем только один раз — при инициализации.

Реализация алгоритма, как и прочих рассматривавшихся ранее в этой главе, не содержит ничего особо сложного.

```
class Combinations
{
public:
    Combinations(int n, int m);
    vector<int> operator*();
    bool operator       !()
            { return done == false; }
    void operator      ++();
private:
```

```
    vector<int> c;
    int n, m;
    bool done;
};

Combinations::Combinations(int n_, int m_)
:c(m_+2),n(n_),m(m_),done(false)
{
    for(int j = 0; j < m; ++j) c[j] = j;
    c[m] = n; c[m+1] = 0;
    done = false;
}

vector<int> Combinations::operator*()
{
    vector<int> q(c);
    q.resize(m);
    return q;
}

void Combinations::operator ++()
{
    int j;
    for(j = 0; c[j]+1 == c[j+1]; ++j) c[j] = j;
    if (j >= m) { done = true; return; }
    ++c[j];
}
```

Возможно, проверяя, какая именно последовательность сочетаний генерируется данным алгоритмом, вы заметите, что она не вполне лексикографически отсортирована. Так, генерация всех сочетаний по 3 элемента из 5 дает нам следующие сочетания:

```
0   1   2
0   1   3
0   2   3
1   2   3
0   1   4
0   2   4
1   2   4
0   3   4
1   3   4
2   3   4
```

Однако лексикографический порядок, казалось бы, должен быть следующим:

```
0   1   2
0   1   3
0   1   4
0   2   3
0   2   4
0   3   4
1   2   3
1   2   4
1   3   4
2   3   4
```

Все становится на свои места, если мы рассмотрим сочетания как битовые числа, в которых *i*-й бит соответствует числу *i*. Тогда сгенерированная последовательность сочетаний приобретает такой вид:

```
00111
01011
01101
01110
10011
10101
10110
11001
11010
11100
```

И в этом смысле сгенерированная последовательность сочетаний лексикографически упорядочена. Тот же результат получится при записи исходных сочетаний в зеркальном виде (012—210 и т.д.).

Генерация всех разбиений числа

Здесь мы рассмотрим два алгоритма, связанных с представлением натурального числа в виде суммы натуральных чисел. Первый из них генерирует все возможные разбиения числа n $a_1 \geq a_2 \geq \cdots \geq a_m \geq 1$, такие что $a_1 + a_2 + \cdots + a_m = n$, где $1 \leq m \leq n$.

```
Алгоритм Partition(n)
// Входные данные: натуральное число n
// Выходные данные: все возможные представления числа n
```

```
//                    в виде суммы натуральных чисел
Создаем массив a[0..n]
a[0] = 0
m = 1
while true do     // Бесконечный цикл
begin
    a[m] = n
    q = m
    if n = 1 then q = q - 1
    Вывод разбиения a[1..m]
    if a[q] == 2 then
    begin
        a[q] = 1
        q = q - 1
        m = m + 1
        a[m] = 1
    end else begin
        if q = 0 then return
        x = a[q] - 1
        a[q] = x
        n = m - q + 1
        m = q + 1
        while n ≤ x do
        begin
            a[m] = x
            m = m + 1
            n = n - x
        end
    end
end
```

Как и в предыдущих разделах, данный алгоритм реализован в виде класса-перечислителя.

```
class partition
{
public:
    partition(int N_):n(N_),done(false) {
```

```
        a.resize(n);
        a[0] = n;
        m = 0; q = m - ((n==1) ? 1 : 0);
    }
    bool operator          !()
            { return done == false; }
    vector<int> operator  *();
    void operator        ++();
private:
    int m,n,q;
    vector<int> a;
    bool done;
};

vector<int> partition::operator*()
{
    vector<int> b(a);
    b.resize(m+1);
    return b;
}
void partition::operator++()
{
    if (q == -1) {done = true; return; }
    if (a[q] == 2) {
        a[q--] = 1;
        a[++m] = 1;
    } else {
        int x = a[q] - 1;
        a[q] = x;
        n = m - q + 1;
        m = q + 1;
        while (n > x) {
            a[m++] = x;
            n -= x;
        }
        a[m] = n;
        q = m - ((n==1) ? 1 : 0);
    }
}
```

Основное изменение в реализации по сравнению с исходным алгоритмом заключается в отказе от ограничителя **a[0]** путем переноса проверки завершения алгоритма в начало оператора **++**. Снижение эффективности за счет дополнительной проверки в данном случае невелико, а с учетом возврата результата в виде **vector<int>** полностью себя оправдывает, поскольку больше не требуется удалять при возврате первый элемент вектора.

Вот пример генерации всех разбиений для числа 6:

```
6
5   1
4   2
4   1   1
3   3
3   2   1
3   1   1   1
2   2   2
2   2   1   1
2   1   1   1   1
1   1   1   1   1   1
```

Второй алгоритм предназначается для генерации аналогичных разбиений на фиксированное число частей, т.е. он генерирует все возможные разбиения числа n $a_1 \ge a_2 \ge \cdots \ge a_m \ge 1$ такие, что $a_1 + a_2 + \cdots + a_m = n$, где m — фиксированное число, удовлетворяющее условию $n \ge m \ge 2$.

```
Алгоритм Partition(n, m)
// Входные данные: натуральное число n и количество
//                 частей m, на которое выполняется
//                 разбиение
// Выходные данные: все возможные представления
//                  числа n в виде суммы m
//                  натуральных чисел
Создаем массив a[1..m+1]
a[1] = n - m + 1
for j = 2 to m do a[j] = 1
a[m+1] = -1
while true do     // Бесконечный цикл
begin
    Вывод разбиения a[1..m]
    if a[2] < a[1] - 1 then
```

```
    begin
        a[1] = a[1] - 1
        a[2] = a[2] + 1
    end else begin
        j = 3
        s = a[1] + a[2] - 1
        while a[j] ≥ a[1] - 1 do
        begin
            s = s + a[j]
            j = j + 1
        end
        if j > m then return
        x = a[j] + 1
        a[j] = x
        j = j - 1
        while j > 1 do
        begin
            a[j] = x
            s = s - x
            j = j - 1
        end
        a[1] = s
    end
end
```

Реализация данного алгоритма точно так же, как и предыдущего, избавляется от дополнительного элемента-ограничителя путем дополнительной проверки:

```
class partition
{
public:
    partition(int N_, int M_)
        :n(N_),m(M_),done(false) {
        a.resize(m,1);
        a[0] = n - m + 1;
    }
    bool operator          !()
        { return done == false; }
    vector<int> operator  *()
```

```
        { return a; }
    void operator        ++();
private:
    int m,n,q;
    vector<int> a;
    bool done;
};

void partition::operator++()
{
    if (a[1] < a[0] - 1) {
        --a[0];
        ++a[1];
    } else {
        int k = 2;
        int s = a[0]+a[1]-1;
        while((k < m) && (a[k] >= a[0]-1))
        {
            s += a[k++];
        }
        if (k >= m) { done = true; return; }
        int x = a[k]+1;
        a[k--] = x;
        while(k > 0) {
            s -= (a[k--] = x);
        }
        a[0] = s;
    }
}
```

Вот что выводит данная программа при генерации всех троек натуральных чисел, в сумме составляющих 10:

```
8   1   1
7   2   1
6   3   1
5   4   1
6   2   2
5   3   2
4   4   2
4   3   3
```

Генерация всех деревьев

В данном разделе мы рассмотрим один из комбинаторных алгоритмов, который генерирует все возможные корректные расстановки *n* пар скобок (под корректной расстановкой мы подразумеваем расстановку, когда пары либо вложены одна в другую, либо не пересекаются). Естественно, возникает вопрос — как пары скобок связаны с деревьями? Оказывается, существует взаимнооднозначная связь между расстановкой пар скобок и деревьями. Рассмотрим, например, расстановку скобок, показанную на рис. 6.1. Все открывающие скобки пронумерованы цифрами над ними, закрывающие — под ними. Линии под строкой скобок соединяют соответствующие друг другу пары скобок — 12, 21, 38, 47, 53, 64, 76 и 85 (первая цифра указывает открывающую скобку, вторая — закрывающую). Показанная на рис. 6.1 строка соответствует лесу, приведенному на рис. 6.2.

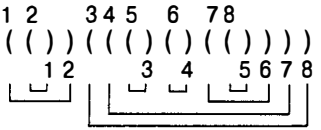


Рис. 6.1. *Пример расстановки 8 пар скобок*

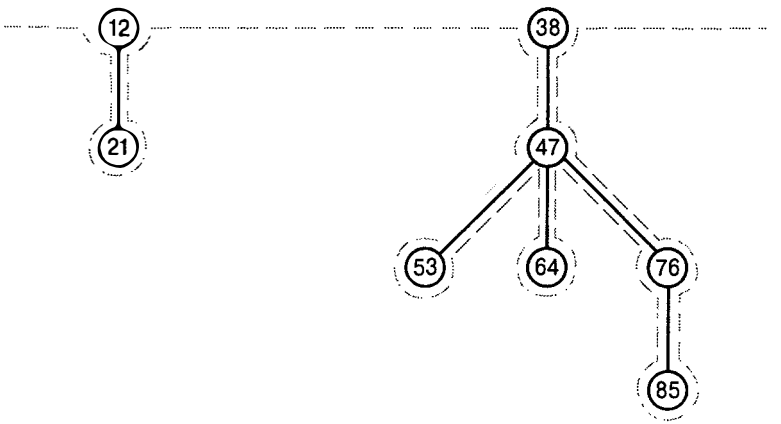


Рис. 6.2. *Лес, соответствующий расстановке скобок, приведенной на рис. 6.1*

Обратите внимание на то, что при обходе в прямом порядке оказываются корректно упорядочены открывающие скобки — 12, 21, 38, 47,

53, 64, 76, 85, а при обходе в обратном порядке — закрывающие скобки: 21, 12, 53, 64, 85, 76, 47, 38. Чтобы восстановить последовательность скобок для леса, можно обойти его так, как показано на рис. 6.2 пунктирной линией. Проходя слева от узла, мы ставим соответствующую открывающую скобку, а справа — закрывающую. Убедитесь сами в справедливости сказанного для приведенных на рис. 6.1 и 6.2 примеров.

В свою очередь, имеется однозначная связь между лесом и бинарным деревом (см., например, [8, раздел 2.3.2]) — для этого выполняется связывание всех потомков каждой семьи и удаление всех вертикальных связей, за исключением связи первого дочернего узла с родителем. В результате мы получим бинарное дерево, показанное на рис. 6.3.

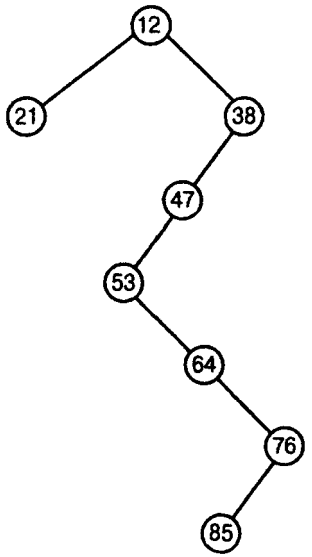


Рис. 6.3. *Представление леса на рис. 6.2 в виде бинарного дерева*

Теперь становится очевидно, что для генерации всех лесов с n узлами, как и для генерации всех бинарных деревьев с n внутренними узлами, достаточно сгенерировать все расстановки n пар скобок. Решить эту задачу можно при помощи следующего алгоритма расстановки вложенных скобок в лексикографическом порядке.

```
Алгоритм NestedParentheses(n)
// Входные данные: количество пар скобок n
// Выходные данные: все возможные расстановки скобок
Создаем массив a[0..2n] (a[0] - вспомогательный
```

```
элемент; расстановка располагается в элементах
массива a[1..2n]
for k = 1 to n do
begin
    a[2*k-1] = '('
    a[2*k]   = ')'
end
a[0] = ')'
m = 2*n - 1
while true do
begin
    Вывод строки a[1..2*n]
    a[m] = ')'
    if a[m-1] = ')' then
    begin
        a[m-1] = '('
        m = m - 1
    end
    else
    begin
        j = m - 1
        k = 2*n - 1
        while a[j] = '(' do
        begin
            a[j] = ')'
            a[k] = '('
            j = j - 1
            k = k - 2
        end
    end
    if j = 0 then return
    a[j] = '('
    m = 2*n - 1
end
```

Как обычно, никаких особых трудностей реализация этого алгоритма не представляет.

```
class parentheses
{
```

```
public:
    parentheses(int N_):n(N_),done(false) {
        a = new char[2*n+2];
        for(int i = 0; i <= n; ++i) {
            a[2*i+1] = '(';
            a[2*i]   = ')';
        }
        a[2*n+1] = 0;
        m = 2*n-1;
    }
    ~parentheses() { delete[] a; }
    bool operator         !()
        { return done == false; }
    const char * operator *()
        { return a + 1; }
    void operator        ++();
private:
    char * a;
    int n, m;
    bool done;
};

void parentheses::operator++()
{
    a[m] = ')';
    if (a[m-1] == ')') {
        a[m-1] = '(';
        --m;
        return;
    }
    int j = m - 1, k = 2*n-1;
    while(a[j] == '(')
    {
        a[j] = ')';
        a[k] = '(';
        --j;
        k-=2;
    }
    if (j == 0) { done = true; return; }
    a[j] = '(';
    m = 2*n-1;
}
```

Глава 7

Дополнительные вопросы

В этой главе нас интересуют не конкретные алгоритмы для решения тех или иных задач, а некоторые методы разработки алгоритмов. Это — рандомизированные алгоритмы, динамическое программирование, метод поиска с возвратом. Конкретные алгоритмы и их реализация в данной главе носят сугубо иллюстративный характер, однако вполне могут помочь вам в разработке собственных решений подобных задач.

В конце этой главы приведено несколько алгоритмов для работы с календарем. Эти алгоритмы оказались в данной главе постольку, поскольку они представляют собой скорее просто расчетные формулы, чем алгоритмы в общепринятом понимании (тем не менее, как показывает практика, вопросы работы с календарем зачастую ставят в тупик начинающих программистов).

Рандомизированные алгоритмы

В основном мы имеем дело с детерминистическими алгоритмами, т.е. такими алгоритмами, которые всегда для одних и тех же входных данных выполняют одну и ту же последовательность действий и приводят к одному и тому же конечному результату. Однако существует и другой класс алгоритмов — рандомизированные алгоритмы, которые в процессе работы используют случайные числа. Алгоритм называется *рандомизированным*, если его поведение определяется не только набором входных величин, но и значениями, которые выдает *генератор случайных чисел* (на практике в распоряжении программиста обычно имеется *генератор псевдослучайных чисел*, т.е. детерминистический алгоритм, который возвращает числа, ведущие себя при статистическом анализе как случайные).

Вспомним, например, алгоритм быстрой сортировки, который в среднем случае имеет эффективность $O(n \log n)$, а в худшем — $O(n^2)$,

причем если в качестве опорного брать первый элемент, то наихудший случай представляет собой уже отсортированный массив. Однако если перед применением быстрой сортировки случайным образом переставить элементы сортируемого массива, то вероятность того, что мы получим упорядоченный массив, исчезающе мала. Заметим, что в силу случайности перестановки два запуска алгоритма быстрой сортировки для одного и того же массива данных будут выполнять различные действия и потребуют различного количества сравнений.

Другой способ рандомизации алгоритма быстрой сортировки заключается в предложении Хоара выбирать опорный элемент случайным образом.

Главное в рандомизированных алгоритмах то, что никакие входные данные не могут вызвать наихудшее поведение алгоритма. Даже злейший враг не сможет подобрать для сортировки плохой входной массив, поскольку дальнейшая случайная перестановка приводит к тому, что порядок входных элементов становится несущественным. Рандомизированные алгоритмы плохо ведут себя лишь тогда, когда генератор случайных чисел выдаст «неудачную» перестановку.

Генераторы псевдослучайных чисел

Генерация псевдослучайных чисел — это очень сложный математический вопрос, который просто невозможно хотя бы сколь-нибудь полно рассмотреть в этой книге. Поэтому мы просто отсылаем читателей к [9, глава 3], а здесь приведем лишь пару реализаций генераторов случайных чисел, предложенных в книге Кнута, генерирующих равномерно распределенные случайные числа.

Первый генератор имеет такой вид:

```
const long MM = 2147483647;
const long AA = 48271;
const long QQ = 44488;  // long(MM/AA)
const long RR =  3399;  // MM % AA;  RR < QQ!

X = AA*(X%QQ) - RR*long(X/QQ);
if (X < 0) X += MM;
```

Как показывает непосредственный вычислительный эксперимент, период последовательности **x** равен 1 556 113 569, минимальное значение — 1, максимальное — 2 147 483 646.

Чтобы получить более длинную последовательность псевдослучайных чисел, можно скомбинировать приведенную программу с другой, дописав дополнительно несколько операций:

```
const long MMM = 2147483399;
const long AAA = 40692;
const long QQQ = 52774;  // long(MM/AA)
const long RRR =  3791;  // MM % AA;  RR < QQ!

Y = AAA*(Y%QQQ) - RRR*long(Y/QQQ);
if (Y < 0) Y += MMM;
Z = X - Y; if (Z <=0) Z += MM
```

Как **X**, так и **Y** должны быть установлены не равными 0, **Z** никогда не равно 0, лежит в диапазоне между 0 и 2^{31}. Длина периода последовательности **Z** приблизительно равна 74 квадриллионам.

Получение случайной перестановки

В качестве одного из способов рандомизации алгоритма быстрой сортировки была упомянута случайная перестановка входной последовательности чисел. Задача получения некоторой случайной перестановки элементов массива $A[1..n]$ встречается не только в рандомизированном алгоритме быстрой сортировки — это достаточно распространенная процедура.

Имеются различные методы «перетасовки» элементов массива, но нас интересуют только те, которые дают нам случайные перестановки с равномерным распределением (т.е. все возможные случайные перестановки должны быть равновероятны; вероятность получения каждой из возможных $n!$ перестановок должна быть равна $1/n!$). Один из простейших и быстрых алгоритмов получения случайной перестановки с равномерным распределением за время $O(n)$ без использования дополнительной памяти таков:

```
Алгоритм RandomizeArray(A)
// Входные данные: массив A размером n элементов
// Выходные данные: случайная перестановка A
for i = 1 to n do swap(A[i],A[Random(i,n)]
```

В этом алгоритме используется функция *Random*(*i*,*n*), представляющая собой генератор равномерно распределенных чисел в интервале от *i* до *n* включительно (о таких генераторах мы только что говорили в предыдущем разделе).

Метод Монте-Карло

К рандомизированным алгоритмам можно отнести численные методы Монте-Карло, формально заключающиеся в том, что задается случайная величина ξ, математическое ожидание которой равно искомой величине z, т.е. $\mathbf{M}[\xi]=z$, после чего осуществляется серия n независимых испытаний случайной величины ξ и приближенно полагается

$$z=\overline{\xi}=\frac{\sum_{i=1}^{n}\xi_i}{n}.$$

В силу того, что $\mathbf{M}[\xi]=z$, при любом натуральном n $\mathbf{M}\left[\overline{\xi}\right]=z$. Если дисперсия $\mathbf{D}[\xi]=\sigma^2$ конечна, то $\mathbf{D}\left[\overline{\xi}\right]=\sigma^2/n$, причем распределение случайной величины $\overline{\xi}$ асимптотически нормально (так что, например, при достаточно большом n (на практике — при $n>10$) неравенство $\left|z-\overline{\xi}\right|<3\sigma/\sqrt{n}$ выполняется с вероятностью около 0.997). На практике значение σ оценивают по формуле

$$\sigma\approx\sqrt{\frac{\sum_{i=1}^{n}\left(\xi_i-\overline{\xi}\right)^2}{n-1}}=\sqrt{\frac{n\sum_{i=1}^{n}\xi_i^2-\left(\sum_{i=1}^{n}\xi_i\right)^2}{n(n-1)}}.$$

Рассмотрим конкретный пример применения метода Монте-Карло — для вычисления определенного интеграла

$$I=\int_0^1 4\sqrt{1-x^2}\,dx.$$

Пусть η — равномерно распределенная на отрезке $[0,1]$ случайная величина, т.е. случайная величина с плотностью распределения

$$p_\eta(x)=\begin{cases}1, & 0\le x\le 1,\\ 0, & x\notin[0,1].\end{cases}$$

Тогда $\xi=4\sqrt{1-\eta^2}$ — тоже некоторая случайная величина, причем по определению математического ожидания

$$\mathbf{M}[\xi]=\int_0^1 4\sqrt{1-x^2}\,p_\eta(x)\,dx=\int_0^1 4\sqrt{1-x^2}\,dx=I.$$

Таким образом,

$$I\approx\overline{\xi}=\frac{1}{n}\sum_{i=1}^{n}4\sqrt{1-\eta_i^2}.$$

Значение σ оценим по приведенной выше формуле.

Вот как выглядит соответствующая программа для расчета значения интеграла с разными значениями n.

```
// Генератор случайных чисел от 0 до RandMax
const int RandMax = 2147483646-1;
int Rand(int seed = 0)
{
    const int MM = 2147483647;
    const int AA =      48271;
    const int QQ =      44488;
    const int RR =       3399;
    static int X = 1;
    if (seed != 0) X = seed;
    X = AA*(X%QQ)-RR*(X/QQ);
    if (X < 0) X += MM;
    return X-1;
}

// Подсчет среднего и дисперсии
void mc(int Count, double&M, double&sigma)
{
    double I = 0.0, I2 = 0.0;
    for(int i = 0; i < Count; ++i)
    {
        // x - случайное число от 0 до 1
        double x = double(Rand())/RandMax;
        x = 4.0*sqrt(1.0 - x*x);
        I  += x;
        I2 += x*x;

    }
    M = I/Count;
    sigma = sqrt((Count*I2-I*I)/Count/(Count-1));
}


int main()
{
```

```
    // Количество проб, для которых
    // проводится эксперимент
    int Counts[] =
    { 10, 100, 1000, 10000, 100000, 1000000,
      10000000, 100000000 };

    printf("        N    Интеграл   3*sigma/sqrt(N)"
           "     Отклонение\n");
    printf("----------------------------------------"
           "----------------\n");
    for(int i = 0;
        i<sizeof(Counts)/sizeof(Counts[0]); ++i)
    {
        double M,D;
        mc(Counts[i],M,D);
        printf("%9d  %10.6lf  %15.10lf  %15.10lf\n",
               Counts[i], M,
               D*3/sqrt(double(Counts[i])),
               fabs(M - 3.1415926));
    }
}
```

Как видно из представленных ниже результатов расчетов, значение определенного интеграла, полученное методом Монте-Карло, мало отличается от точного значения, равного π. Отклонения приближенного значения от точного, как видите, существенно меньше принятого в качестве границы значения $3\sigma/\sqrt{n}$.

N	Интеграл	3*sigma/sqrt(N)	Отклонение
10	3.155703	0.9746154298	0.0141108558
100	3.129528	0.2758984985	0.0120650626
1000	3.146651	0.0837720522	0.0050588183
10000	3.138165	0.0268280812	0.0034279958
100000	3.143550	0.0084560860	0.0019573032
1000000	3.142750	0.0026755951	0.0011572278
10000000	3.142041	0.0008467992	0.0004484336
100000000	3.141613	0.0002678445	0.0000202818

Кратные интегралы вычисляются методом Монте-Карло аналогично. Что касается сравнения метода Монте-Карло и применения ква-

дратурных формул, то преимущества метода Монте-Карло проявляются в многомерных случаях, при сложных областях интегрирования. Недостатком метода является его вероятностный характер, т.е. отсутствие строгих, стремящихся к нулю при $n \to \infty$ оценок погрешности.

Динамическое программирование

Динамическое программирование позволяет решать задачи, комбинируя решения вспомогательных задач. В данном случае исторически сложившееся название «программирование» означает метод, а не составление компьютерного кода. Динамическое программирование находит применение тогда, когда вспомогательные задачи используют решения одних и тех же подзадач, причем каждая вспомогательная задача решается только один раз, после чего ответ сохраняется в таблице.

Динамическое программирование, как правило, применяется к задачам оптимизации, в которых возможно наличие многих решений, и каждому из решений можно сопоставить некоторое значение. Задача состоит в том, чтобы найти решение с оптимальным (минимальным или максимальным) значением.

Обычно решение задачи методом динамического программирования состоит из четырех этапов. Сначала описывается структура оптимального решения, затем рекурсивно определяется значение, соответствующее оптимальному решению, после чего тем или иным способом вычисляется значение, соответствующее оптимальному решению, и при необходимости составляется само оптимальное решение.

В этом разделе применение динамического программирования будет проиллюстрировано на примере решения задачи об оптимальном перемножении матриц, которая формулируется следующим образом. Имеется последовательность из n матриц A_1, A_2, ..., A_n, и нам требуется вычислить их произведение $A_1 \cdot A_2 \cdot \ldots \cdot A_n$. Произведения матриц вычисляются при помощи стандартного алгоритма, основанного на определении произведения матриц, т.е. для перемножения матриц размером $m \times n$ и $n \times p$ требуется выполнить mnp умножений чисел. Поскольку произведение матриц ассоциативно, оно не зависит от порядка вычисления произведений отдельных матриц, т.е., например, $(A_1A_2)A_3 = A_1(A_2A_3)$, но количество умножений чисел при этом оказывается разным. Нам надо найти такую расстановку скобок для

последовательности из n матриц, чтобы количество выполненных умножений было минимальным. Само собой, цепочка согласованна, т.е. количество столбцов в каждой матрице последовательности совпадает с количеством строк матрицы, следующей за ней (размер i-й матрицы — $p_{i-1} \times p_i$.

Динамическое программирование применимо в первую очередь тогда, когда в оптимальном решении задачи содержатся оптимальные решения подзадач. Применительно к нашей задаче о перемножении последовательности матриц это означает, что если мы рассмотрим оптимальную расстановку скобок для произведения матриц $A_i \cdot \ldots \cdot A_j$, и если оно нетривиально ($i<j$), то оно обязательно будет разбито на две части $A_i \cdot \ldots \cdot A_k$ и $A_{k+1} \cdot \ldots \cdot A_j$ ($i \le k < j$), причем расстановки скобок в обеих частях оптимальны. Это легко доказать от противного — если расстановка скобок в какой-то из частей не оптимальна, заменив ее оптимальной, мы уменьшим количество умножений чисел в ней и, соответственно, количество умножений при вычислении всего произведения матриц. А это противоречит предположению об оптимальности исходной расстановки скобок.

Такое доказательство — необходимый шаг динамического программирования, входящий в качестве части в первый из перечисленных выше четырех этапов динамического программирования. Теперь рассмотрим общее количество умножений чисел, необходимых для определения произведения $A_i \cdot \ldots \cdot A_j$. Обозначим его как m_{ij}. Тогда в тривиальном случае ($i=j$) $m_{ij}=0$, а для нетривиального случая при показанном выше разбиении цепочки на две выполняется соотношение $m_{ij}=m_{ik}+m_{k+1,j}+p_{i-1}p_kp_{k+1}$. Следовательно, оптимальное количество умножений чисел для вычисления произведения можно определить как

$$m_{ij} = \begin{cases} 0 & \text{при } i = j, \\ \min\limits_{i \le k < j} \left\{ m_{i,k} + m_{k+1,j} + p_{i-1} p_k p_j \right\} & \text{при } i < j. \end{cases}$$

Решение исходной задачи представляет собой величину m_{1n}, которая может быть определена при помощи соответствующей рекурсивной процедуры. Проблема в том, что непосредственное применение рекурсии приведет к тому, что многие значения m_{ij} будут вычисляться многократно. Избежать этого можно при помощи использования массива $m[n,n]$, который заполняется вычисленными значениями «снизу вверх» — от значений для одноэлементных цепочек m_{ii} до последнего,

интересующего нас элемента массива $m[1,n]$, который и представляет собой окончательное решение поставленной задачи. Параллельно с массивом m мы используем массив s, в который в элементы $s[i,j]$ заносим индексы k элементов, при которых достигаются оптимальные стоимости — эта таблица затем используется при построении оптимального решения. Понятно, что так как $i<j$, заполняется только половина каждой таблицы. Заполнение в восходящем порядке приводит к тому, что при вычислении очередного значения m_{ij} все необходимые для этого элементы таблицы m уже определены.

```
MatrixChain(p,n)
// Входные данные: массив размеров матриц p для
//                 последовательности матриц длиной n
// Выходные данные: массивы m и s — количества
//                  умножений чисел и индексов
//                  разбиения в оптимальном случае
for i = 1 to do m[i,i] = 0
for l = 2 to n do
begin
    for i = 1 to n - l + 1 do
    begin
        j = i + l - 1
        m[i,j] = ∞
        for k = i to j - 1 do
        begin
            q = m[i,k] + m[k+1,j] + p[i-1]p[k]p[j]
            if q < m[i,j] then
            begin
                m[i,j] = q
                s[i,j] = k
            end
        end
    end
end
return m и s
```

Определив m и s, мы сразу же получаем требовавшееся количество умножений чисел для вычисления оптимальной цепочки $m[1,n]$, а построение оптимального решения — вывод расстановки скобок — легко осуществить при помощи очередной рекурсивной процедуры.

Зная индекс разбиения для цепочки $A_i \cdot \ldots \cdot A_j$, легко вывести расстановку скобок как

$$(\text{расстановка для } A_i \cdot \ldots \cdot A_k)(\text{расстановка для } A_{k+1} \cdot \ldots \cdot A_j),$$

т.е. использовать следующую рекурсивную процедуру:

```
PrintChain(s,i,j)
if i = j then print "A[",i,"]"
else begin
    print "("
        PrintChain(s,i,s[i,j])
        PrintChain(s,s[i,j]+1,j)
    print ")"
end
```

Построение таблиц можно осуществить и иначе, нисходящим методом. При этом мы непосредственно используем рекурсивное соотношение для m_{ij}, но когда мы сталкиваемся с необходимостью вычисления очередного значения, то начинаем с проверки — не вычислялось ли это значение ранее и нет ли его в таблице. Если есть — оно просто извлекается из таблицы; если нет — оно вычисляется и вносится в таблицу.

Здесь не приводится псевдокод такого решения, но показанная далее реализация решения данной задачи на C++ использует именно нисходящий подход. Следует заметить, что в этой реализации индикатором того, что значение еще не вычислено, служит нулевое значение (которое может возникать только в случае $i=j$, но этот случай рассматривается в программе отдельно, так что нулевое значение в качестве индикатора вполне оправданно).

Для иллюстративности в программе решаются последовательно две задачи — поиска минимального и максимального количества умножений чисел, необходимых для вычисления матричного произведения $A_1 \cdot A_2 \cdot \ldots \cdot A_n$. Доказательство оптимальности структуры решения для поиска максимального количества умножений остается читателям в качестве небольшого упражнения.

```
#include <iostream>
#include <limits>
using namespace std;
int P[] = {30, 35, 15, 5, 10, 20, 25};
```

```
// Размеры матриц:
// 30x35, 35x15, 15x5, 5x10, 10x20, 20x25
// Матрица i имеет размер P[i] x P[i+1]

// Массивы, описанные в тексте раздела
static long long
        M[sizeof(P)/sizeof(P[0])-1]
         [sizeof(P)/sizeof(P[0])-1] = { 0 };
static long long
        S[sizeof(P)/sizeof(P[0])-1]
         [sizeof(P)/sizeof(P[0])-1] = { 0 };

int MatrixChainMin(int i, int j)
{
    if (i == j) return 0;
    if (M[i][j] > 0) return M[i][j];
    int Count = numeric_limits<int>::max();
    for(int k = i; k < j; ++k)
    {
        int q =
            MatrixChainMin(i,k)+
            MatrixChainMin(k+1,j)+
            P[i]*P[k+1]*P[j+1];
        if (q < Count) {
            Count = q;
            S[i][j] = k;
        }
    }
   M[i][j] = Count;
    return Count;
}

int MatrixChainMax(int i, int j)
{
    if (i == j) return 0;
    if (M[i][j] > 0) return M[i][j];
    int Count = 0;
    for(int k = i; k < j; ++k)
    {
        int q =
```

```
            MatrixChainMax(i,k)+
            MatrixChainMax(k+1,j)+
            P[i]*P[k+1]*P[j+1];
        if (q > Count) {
            Count = q;
            S[i][j] = k;
        }
    }
    M[i][j] = Count;
    return Count;
}

void PrintChain(int i, int j)
{
    if (i == j) cout << "A["<<i+1<<"]";
    else {
        cout << "(";
        PrintChain(i,S[i][j]);
        PrintChain(S[i][j]+1,j);
        cout << ")";
    }
}

void main()
{
    // Поиск минимального количества умножений
    cout << "Min: " <<
         MatrixChainMin(0,sizeof(P)/sizeof(P[0])-2)
         << endl;
    PrintChain(0,sizeof(P)/sizeof(P[0])-2);
    cout << endl;

    // Подготовка к вычислению максимума
    memset(M,0,sizeof(M));
    memset(S,0,sizeof(S));

    // Поиск максимального количества умножений
    cout << "Max: " <<
         MatrixChainMax(0,sizeof(P)/sizeof(P[0])-2)
         << endl;
```

```
    PrintChain(0,sizeof(P)/sizeof(P[0])-2);
    cout << endl;
}
```

Применение метода нисходящего решения с запоминанием можно коротко и эффектно проиллюстрировать на примере вычисления чисел Фибоначчи. Забудем на секунду о том, что числа Фибоначчи можно вычислить очень быстро, за время O(1) при помощи явной зависимости от n, и воспользуемся рекуррентной формулой $F_i = F_{i-2} + F_{i-1}$. Это дает нам следующую функцию:

```
int fib(int n)
{
    if (n < 2) return n;
    return fib(n-1) + fib(n-1);
}
```

Теперь попробуем сделать то же, но с использованием подхода с запоминанием. Для простоты ограничимся числами Фибоначчи до пятидесятого.

```
int fibM(long long n)
{
    static int F[51] = {0};
    if (n < 2) return n;
    if (F[n]) return F[n];
    return (F[n] = fibM(n-1) + fibM(n-2));
}
```

Будет весьма поучительно — как с точки зрения демонстрации эффективности метода запоминания для такого рода задач, так и с точки зрения наглядной демонстрации экспоненциального алгоритма — сравнить время вычисления, например, сорокового или пятидесятого (правда, для этого тип **int** придется заменить на **long long**) числа Фибоначчи. Если вы решитесь на такой эксперимент, то, запустив программу, не волнуйтесь — она не зациклилась (например, процессору Celeron 2 ГГц потребовалось порядка 30 секунд для вычисления 45-го числа Фибоначчи таким способом)...

Более подробно динамическое программирование описано в [12, 21].

Жадные алгоритмы

С динамическим программированием тесно связаны так называемые жадные алгоритмы, которые в принципе можно рассматривать как частный случай динамического программирования. Связь их обусловлена наличием оптимальной подструктуры задачи, т.е. в оптимальном решении задачи находятся оптимальные решения подзадач.

Жадный подход строит решение посредством последовательности шагов, на каждом из которых получается частичное решение поставленной задачи, пока не будет получено полное решение. При этом на каждом шаге — и это является главным в рассматриваемом методе — выбор должен быть

- *допустимым*, т.е. удовлетворять ограничениям задачи;
- *локально оптимальным*, т.е. наилучшим локальным выбором среди всех допустимых вариантов, доступных на каждом шаге;
- *окончательным*, т.е., будучи сделан, он не может быть изменен последующими шагами алгоритма.

Эти требования поясняют название метода: на каждом шаге он предполагает «жадный» выбор наилучшей доступной альтернативы в надежде, что последовательность локально оптимальных выборов приведет к глобально оптимальному решению всей задачи. Существуют задачи, для которых последовательность локально оптимальных выборов приводит к оптимальному решению для любого экземпляра рассматриваемой задачи, но есть и другие задачи, для которых это не так; для задач такого рода жадный алгоритм может представлять интерес только в том случае, если нас устраивает приближенное решение.

Простейшим примером решаемой жадным алгоритмом задачи может служить задача о том, как выплатить сумму в 98 копеек монетами номиналом 1, 2, 5, 10 и 25 копеек так, чтобы общее количество монет было минимально. Жадный алгоритм в этом случае состоит в том, чтобы на каждом шаге построения решения использовать монеты максимального номинала, с тем чтобы их было как можно меньше (достижение локального минимума). Сначала мы берем три монеты по 25 копеек (4 монеты дают сумму, большую чем требуется). Остается выплатить 98–25*3=23 копейки. На втором шаге мы берем очередные наибольшие по номиналу монеты, которыми можно выдать недостающую сумму, — две монеты по 10 копеек. Два следующих шага дают нам по одной двух- и однокопеечной монете, тем самым позволяя выплатить всю сумму 7 монетами. (Заметим, что такой жадный алгоритм подходит не для любой

суммы и набора монет — например, сумму в 15 копеек монетами 1, 5 и 11 копеек можно выплатить тремя монетами по 5 копеек, но применение жадного алгоритма даст нам пять монет — 11 копеек и четыре монеты по 1 копейке. Однако, решая эту задачу методом динамического программирования, мы получим правильный ответ.)

Как правило, жадные алгоритмы интуитивно привлекательны и просты. Но, несмотря на несомненную простоту применения, для каждой задачи требуется подчас весьма сложное доказательство применимости жадного алгоритма для ее решения. Жадные алгоритмы основаны на сложной теории, базирующейся на абстрактной комбинаторной структуре, которая называется «матроид», однако эта тема выходит за рамки данной книги. Ограничимся лишь упоминанием о том, что этот вопрос разбирается в книге [12].

Поиск с возвратом

В ряде случаев сложные задачи, быстрый алгоритм решения которых неизвестен или отсутствует в принципе, и которые приходится решать методом исчерпывающего перебора, можно решить быстрее при помощи метода поиска с возвратом. Если решение задачи состоит из нескольких компонентов, то исчерпывающий перебор состоит в генерации всех возможных комбинаций решений и проверке для каждого из них, является оно корректным решением поставленной задачи или нет. Однако в этом случае можно поступить иначе — собирать решение покомпонентно с выяснением, можно ли получить корректное решение задачи при данном выборе первых компонентов или нет. Если нет — генерация всех возможных решений с зафиксированными первыми компонентами смысла не имеет, и можно переходить к следующему варианту. Алгоритм в этой ситуации возвращается к последнему построенному компоненту и заменяет его следующим возможным вариантом компонента этого уровня.

Поскольку здесь нет четко поставленной задачи, невозможно указать строгий алгоритм решения. Это метод разработки алгоритма, а не формальный алгоритм. Обобщенный псевдокод алгоритма поиска с возвратом можно записать следующим образом.

```
Алгоритм BackTrack(X[1..i])
Вход: Массив X[1..i], определяющий первые i
      допустимых компонентов решения
```

Выход: Все кортежи, представляющие решения задачи

```
if X[1..i] является решением then write X[1..i]
else
begin
    for каждого допустимого элемента x_{i+1},
        согласующегося с уже имеющимся множеством
        X[1..i] и удовлетворяющего ограничениям
        задачи
    begin
            X[i+1] = x
            Backtrack(X[1..i+1])
    end
end
```

Классическими примерами задач, решаемых при помощи такого алгоритма, является задача о расстановке ферзей на шахматной доске таким образом, чтобы они не угрожали друг другу, или задача об обходе конем шахматной доски таким образом, чтобы, сойдя с клетки **a1**, он посетил все клетки доски, причем ни одна клетка не была бы посещена дважды. В последней задаче компонентами решения являются последовательные ходы коня. При решении задачи поддерживается логический массив, соответствующий клеткам доски, в котором отмечается, в каких клетках конь уже побывал к настоящему моменту. На каждом шаге выбирается один из 8 возможных ходов коня и проверяется, возможен ли этот ход (не выходит ли он за пределы доски), и не приходится ли он на клетку, которая уже была посещена ранее. Если ход возможен, отмечаем в массиве новую посещенную клетку и переходим к проверке нового хода. Если нет — исследуем очередной возможный ход. Если исчерпаны все ходы, отмечаем текущую клетку как свободную и возвращаемся к предыдущему ходу.

Вот как выглядит простой листинг программы для решения задачи об обходе доски конем (выбрана доска размером 5×5 как минимальная, имеющая решение поставленной задачи).

```
#include <iostream>
using namespace std;

// Доска может иметь разный размер
```

```
const int Size = 5;
// Массив посещенных клеток
bool Desk[Size][Size] = {0};
// Количество ходов
int Moves = 0;
// Основная функция
bool nextMove(int currentX, int currentY) {
    // Все ходы сделаны!
    if (Moves==Size*Size) { return true; }
    // Ход за пределы доски?
    if ((currentX < 0)||(currentX >= Size))
        return false;
    if ((currentY < 0)||(currentY >= Size))
        return false;
    // Уже посещенная клетка?
    if (Desk[currentX][currentY]) return false;
    // Ход допустим, подбираем ход следующего уровня
    ++Moves;
    Desk[currentX][currentY] = true;
    if (nextMove(currentX+1,currentY+2) ||
        nextMove(currentX+1,currentY-2) ||
        nextMove(currentX-1,currentY+2) ||
        nextMove(currentX-1,currentY-2) ||
        nextMove(currentX+2,currentY+1) ||
        nextMove(currentX+2,currentY-1) ||
        nextMove(currentX-2,currentY+1) ||
        nextMove(currentX-2,currentY-1)) {
        // Мы нашли путь!
        cout << 'a'+[currentX]<<currentY+1<<"  ";
        return true;
    }
    // Увы, путь не найден...
    // Очищаем клетку
    Desk[currentX][currentY] = false;
    --Moves;
    return false;
}

int main() {
```

```
    if (nextMove(0,0)==false) cout << "No solution";
    cout << endl;
}
```

Эта задача рассмотрена такое множество раз в литературе, что уже не представляет особого интереса. Давайте в качестве второго примера рассмотрим интересную головоломку, приписываемую самому Эйнштейну. Она примечательна тем, что при ее решении будет в особенности наглядно видна суть поиска с возвратом. Итак, есть 5 домов разного цвета, в каждом из которых живет по одному человеку отличной друг от друга национальности. Каждый жилец пьет только определенный напиток, курит определенную марку сигарет и держит определенное животное, причем никто из 5 человек не пьет одинаковые с другим напитки, не курит одинаковые сигареты и не держит одинаковое животное. Вопрос: кому принадлежит рыба?

При этом известно следующее.

1. Англичанин живет в красном доме.
2. Швед держит собаку.
3. Датчанин пьет чай.
4. Зеленый дом стоит слева от белого.
5. Жилец зеленого дома пьет кофе.
6. Человек, который курит PallMall, держит птицу.
7. Жилец из среднего дома пьет молоко.
8. Норвежец живет в первом доме.
9. Жилец из желтого дома курит Dunhill.
10. Курильщик Marlboro живет около того, кто держит кошку.
11. Человек, который содержит лошадь, живет около того, кто курит Dunhill.
12. Курильщик сигарет Winfield пьет пиво.
13. Норвежец живет около голубого дома.
14. Немец курит Rothmans.
15. Курильщик Marlboro живет по соседству с человеком, который пьет воду.

Естественно, что путем логических рассуждений эту задачу можно решить минут за 10–15 и нереально решить методом исчерпываю-

щего перебора, так как всего имеется $(5!)^5$=24883200000 вариантов перестановок свойств жильцов.

Попробуем решить задачу методом поиска с возвратом. Компонентами решения являются пять перестановок пяти элементов — описаний цвета домов, национальностей проживающих, их напитков, сигарет и животных.

После выбора каждого уровня элементов (для реализации перебора перестановок использован алгоритм **next_permutation**) мы проверяем, можно ли получить решение для данного набора первых компонентов. Для этого мы проверяем выполнение тех условий, компоненты для которых зафиксированы. Так, определив перестановку национальностей, мы можем проверить выполнение правила 8, так как в нем задействована только национальность жильца, но никак не другое правило, например, правило 14 — так как расстановка сигарет пока что неизвестна.

Естественно, что чем больше условий мы сможем проверять на начальных этапах, тем больше заведомо некорректных комбинаций мы сможем отсечь. Именно поэтому после простейшего анализа условий был выбран именно такой порядок компонентов (впрочем, анализ его оптимальности не проводился).

Чтобы не быть голословными, мы добавили в программу счетчики количества отбрасываний комбинаций, которые не могут привести к решению, на каждом из уровней проверки — это позволит нам сравнить количество проверенных вариантов со всеми возможными. Заметим также, что после того, как решение найдено, работа программы продолжается, чтобы гарантировать единственность решения головоломки.

```
#include <algorithm>
#include <iostream>
#include <iomanip>
using namespace std;
// Типы для описания национальностей, цвета домов,
// напитков, сигарет и животных введены для удобства
enum Nation{Briton, German, Swede, Dane, Norwegian};
enum Color {Green, Blue, Yellow, Red, White};
enum Drink {Tea, Coffee, Beer, Water, Milk};
enum Smoke {Dunhill,Marlboro,Winfield,
            PallMall,Rothmans};
enum Pet   {Horse, Cat, Dog, Bird, Fish};
```

```
const char * NName[] = { "Briton","German","Swede",
                         "Dane","Norwegian"};
const char * CName[] = { "Green","Blue","Yellow",
                         "Red","White"};
const char * DName[] = { "Tea","Coffee","Beer",
                         "Water","Milk"};
const char * SName[] = { "Dunhill","Marlboro",
                         "Winfield",
                         "PallMall","Rothmans"};
const char * PName[] = { "Horse","Cat","Dog",
                         "Bird","Fish"};
// Функция для поиска номера дома, соответствующая
// определенному свойству
int House(int*array, int sign)
{
    for(int i = 0; i < 5; ++i)
        if (array[i]==sign) return i;
    return 0;
}
// Условия задачи
bool Rule01(int*N,int*C)
    { return C[House(N,Briton)]==Red; }
bool Rule02(int*N,int*P)
    { return P[House(N,Swede)] ==Dog; }
bool Rule03(int*N,int*D)
    { return D[House(N,Dane)] ==Tea; }
bool Rule04(int*C)
    { return House(C,Green)==House(C,White) - 1; }
bool Rule05(int*C,int*D)
    { return D[House(C,Green)]==Coffee; }
bool Rule06(int*S,int*P)
    { return P[House(S,PallMall)]==Bird; }
bool Rule07(int*D)
    { return House(D,Milk)==2; }
bool Rule08(int*N)
    { return House(N,Norwegian)==0; }
bool Rule09(int*C,int*S)
    { return C[House(S,Dunhill)]==Yellow; }
bool Rule10(int*P,int*S)
    { int i = House(S,Marlboro), j = House(P,Cat);
```

```
      return ((i==j + 1)||(i==j - 1)); }
bool Rule11(int*P,int*S)
    { int i = House(P,Horse), j = House(S,Dunhill);
      return ((i==j + 1)||(i==j - 1)); }
bool Rule12(int*D,int*S)
    { return D[House(S,Winfield)]==Beer; }
bool Rule13(int*C,int*N)
    { int i = House(N,Norwegian), j = House(C,Blue);
      return ((i==j + 1)||(i==j - 1)); }
bool Rule14(int*N,int*S)
    { return S[House(N,German)]==Rothmans; }
bool Rule15(int*D,int*S)
    { int i = House(S,Marlboro), j = House(D,Water);
      return ((i==j + 1)||(i==j - 1)); }
// Поскольку это не просто решение задачи, а
// вычислительный эксперимент, нам нужны счетчики
// количества отказов для каждого компонента
int Count1 = 0, Count2 = 0, Count3 = 0,
    Count4 = 0, Count5 = 0;

int main()
{   // Национальность
  int N[5] = {Briton,German,Swede,Dane,Norwegian};
  for(int flag = true; flag;
      flag = next_permutation(N,N+5))
  {
    if (Rule08(N)==false) {
      ++Count1;
      continue;
    }
    // Цвет дома
    int C[5] = { Green,Blue,Yellow,Red,White};
    for(int flag = true; flag;
        flag = next_permutation(C,C+5))
    {
      if ((Rule04(C)==false)||(Rule01(N,C)==false)||
          (Rule13(C,N)==false))
      {
        ++Count2;
        continue;
      }
```

```
// Напиток
int D[5] = { Tea,Coffee,Beer,Water,Milk};
for(int flag = true; flag;
    flag = next_permutation(D,D+5))
{
  if ((Rule07(D)  ==false)||
      (Rule03(N,D)==false) ||
      (Rule05(C,D)==false))
  {
    ++Count3;
    continue;
  }
  // Сигареты
  int S[5] = { Dunhill,Marlboro,
               Winfield,PallMall,Rothmans};
  for(int flag = true; flag;
      flag = next_permutation(S,S+5))
  {
    if ((Rule09(C,S)==false)||
        (Rule14(N,S)==false)||
        (Rule12(D,S)==false)||
        (Rule15(D,S)==false))
    {
      ++Count4;
      continue;
    }
    // Животные
    int P[5] = { Horse,Cat,Dog,Bird,Fish};
    for(int flag = true; flag;
        flag = next_permutation(P,P+5))
    {
      if ((Rule02(N,P)==false)||
          (Rule06(S,P)==false)||
          (Rule10(P,S)==false)||
          (Rule11(P,S)==false))
      {
        ++Count5;
        continue;
      }
      // Вывод решения и значений счетчиков
```

```
cout << "Houses:    " << setw(10)
  << 1 << setw(10) << 2 << setw(10)
  << 3 << setw(10) << 4 << setw(10)
  << 5 << endl;
cout << "Color:     "
    << setw(10) << CName[C[0]]
    << setw(10) << CName[C[1]]
    << setw(10) << CName[C[2]]
    << setw(10) << CName[C[3]]
    << setw(10) << CName[C[4]]
    << setw(10) <<endl;
cout << "Nations:   "
    << setw(10) << NName[N[0]]
    << setw(10) << NName[N[1]]
    << setw(10) << NName[N[2]]
    << setw(10) << NName[N[3]]
    << setw(10) << NName[N[4]]
    << setw(10) <<endl;
cout << "Drinks:    "
    << setw(10) << DName[D[0]]
    << setw(10) << DName[D[1]]
    << setw(10) << DName[D[2]]
    << setw(10) << DName[D[3]]
    << setw(10) << DName[D[4]]
    << setw(10) <<endl;
cout << "Smoke:     "
    << setw(10) << SName[S[0]]
    << setw(10) << SName[S[1]]
    << setw(10) << SName[S[2]]
    << setw(10) << SName[S[3]]
    << setw(10) << SName[S[4]]
    << setw(10) <<endl;
cout << "Pet:       "
    << setw(10) << PName[P[0]]
    << setw(10) << PName[P[1]]
    << setw(10) << PName[P[2]]
    << setw(10) << PName[P[3]]
    << setw(10) << PName[P[4]]
    << setw(10) <<endl;
cout << "------------" << endl;
```

```
                cout << "Count[1] = " << setw(5)
                     << Count1 << endl;
                cout << "Count[2] = " << setw(5)
                     << Count2 << endl;
                cout << "Count[3] = " << setw(5)
                     << Count3 << endl;
                cout << "Count[4] = " << setw(5)
                     << Count4 << endl;
                cout << "Count[5] = " << setw(5)
                     << Count5 << endl;
              }
            }
          }
        }
      }
      cout << "-----------" << endl;
      cout<<"Count[1] = "<<setw(5)<<Count1<<endl;
      cout<<"Count[2] = "<<setw(5)<<Count2<<endl;
      cout<<"Count[3] = "<<setw(5)<<Count3<<endl;
      cout<<"Count[4] = "<<setw(5)<<Count4<<endl;
      cout<<"Count[5] = "<<setw(5)<<Count5<<endl;
}
```

Вот как выглядит вывод этой программы:

```
Houses:          1        2        3        4        5
Color:      Yellow     Blue      Red    Green    White
Nations:  Norwegian    Dane   Briton   German    Swede
Drinks:       Water      Tea     Milk   Coffee     Beer
Smoke:      Dunhill Marlboro PallMall Rothmans Winfield
Pet:            Cat    Horse     Bird     Fish      Dog
-----------
Count[1] =    96
Count[2] =  2208
Count[3] =  1031
Count[4] =   603
Count[5] =    27
-----------
Count[1] =    96
Count[2] =  2868
Count[3] =  1432
```

```
Count[4] =    959
Count[5] =    119
```

Как видите, основная масса «отсевов» произошла на втором и третьем этапах. Решение найдено после проверки 3965 вариантов, а всего программа проверила 5474 варианта, что немногим менее 220 миллиардных долей общего числа вариантов.

За счет чего получилась такая экономия? В исходном тексте программы ясно видно, что это возможно только благодаря выносу проверок из внутренних циклов во внешние. Внесите все проверки во внутренний цикл — и вы получите тривиальный метод исчерпывающего перебора. В этой программе с итеративным решением (в отличие от рекурсивного решения задачи об обходе доски конем) это свойство проявилось особенно четко.

Оно же заставляет при разработке подобного рода алгоритмов особенно тщательно продумывать вложенность циклов, с тем, чтобы как можно больше отсевов заведомо некорректных вариантов выполнялось во внешних циклах. Какие-либо более конкретные рекомендации дать, увы, невозможно. Можно только посоветовать как следует изучить задачу и воспользоваться ее особыми свойствами — например, симметрией или какими-то другими. Например, если бы нас интересовало только получение решения (без проверки его единственности) в головоломке Эйнштейна, то можно было бы первым элементом массива национальностей поставить норвежца, так как из условия 8 известно, что он живет в первом доме (впрочем, в данной задаче это ненамного бы ускорило ее решение).

И напоследок следует заметить: несмотря на то, что мы легко решили пару задач, это далеко не означает, что все задачи будут решаться так же просто. Успешность этой стратегии колеблется в очень широких пределах. Одна и та же задача, легко разрешимая для одних начальных условий, может потребовать едва ли не исчерпывающего перебора при других.

Метод ветвей и границ

Метод ветвей и границ очень напоминает метод поиска с возвратом. Главная идея поиска с возвратом — прекращение работы в определенном направлении, как только можно сделать вывод, что это направление не может привести к решению задачи. Эта идея может быть усилена при работе с задачами оптимизации, которые должны

минимизировать или максимизировать некоторую целевую функцию, обычно при наличии определенных ограничений (длина маршрута, стоимость выбранных предметов, стоимость назначений и т.п.). При решении задач оптимизации *допустимое решение* означает решение, которое удовлетворяет всем ограничениям задачи, но не является *оптимальным*. Оптимальное решение — это допустимое решение с наилучшим значением целевой функции.

По сравнению с методом поиска с возвратом методу ветвей и границ требуется способ получить для каждого узла дерева пространства состояний границу наилучшего значения целевой функции (нижней границы в задаче минимизации и верхней — в задаче максимизации) для всех решений, которые могут быть получены путем дальнейшего добавления компонентов к имеющемуся частичному решению, и значения наилучшего решения, полученного к этому моменту.

Если такая информация доступна, мы можем сравнивать значение границы со значением наилучшего решения, полученного к этому моменту: если значение границы не лучше значения уже имеющегося наилучшего решения — т.е. не меньше в случае задачи минимизации или не больше в случае задачи максимизации, — то это направление бесперспективно, и его обработка может быть завершена (иногда говорят, что обрезается ветвь дерева решения задачи), поскольку ни одно получаемое таким образом решение не может оказаться лучше того, что уже имеется. В этом заключается основная идея метода ветвей и границ — работа прекращается не только тогда, когда для данного частичного решения получение полного решения невозможно в принципе, но и когда такое частичное решение заведомо оказывается неоптимальным (хуже некоторого уже имеющегося допустимого решения).

Давайте в качестве примера рассмотрим вопрос о том, как минимальным количеством монет достоинством 11, 5 и 1 копейка собрать сумму 15 копеек (мы уже сталкивались с тем, что эта задача не решается при помощи жадного алгоритма).

Каким образом оценить для данной суммы и набора монет нижнюю границу количества монет, составляющих искомую сумму? Понятно, что это количество не может быть меньшим, чем количество монет наибольшего достоинства (не превышающего сумму), составляющих данную сумму, т.е. для суммы S и максимального достоинства монеты N, не превышающего S, это количество равно $\lfloor (S-1)/N \rfloor + 1$.

Очевидно также, что желательно просматривать варианты в порядке убывания достоинства монет, т.е. чтобы первым решением было жадное решение — скорее всего, оно будет достаточно близким к оптимальному (если не совпадать с ним). Это и предыдущее соображение о нижней границе приводит к тому, что множество достоинств монет должно быть упорядочено по убыванию.

Заметим также, что одно допустимое решение напрашивается сразу же — это сумма, составленная из монет достоинством в 1 копейку. Будем считать это решение в начале работы программы текущим наилучшим решением.

После всего сказанного и с учетом нашего опыта в решении задач методом поиска с возвратом можно тут же приступить к написанию программы для поиска оптимального составления суммы — как обычно, не самым эффективным, зато самым понятным способом.

```
int coins[] = { 11, 5, 1 };  // Достоинства монет
int Sum = 15;                // Искомая сумма

vector<int> sol;             // Текущее решение
vector<int> bestSolution;    // Текущее наилучшее
                             // решение

// Нижняя граница количества монет для
// частичного решения sol
int estimate(vector<int>&sol, int sum)
{   // Имеем sol.size() монет и сумму, которую надо
    // добавить к частичному решению
    sum -= accumulate(sol.begin(),sol.end(),0);
    // Находим монету максимального достоинства, не
    // превышающую сумму
    for(int i = 0;
        i < sizeof(coins)/sizeof(coins[0]); ++i)
        if (sum >= coins[i])
            return sol.size()+(sum-1)/coins[i]+1;
    return sol.size() + sum;
}

// Добавляем к текущему решению монету - теперь
// в решении level монет
```

```
void makeSolutionAtLevel(vector<int>&sol, int level)
{
    if (level >= bestSolution.size())
        return;      // Новое решение не лучше
                     // уже имеющегося
    sol.resize(level); // Добавим место для
                       // еще одной монеты
    for(int i = 0;
        i < sizeof(coins)/sizeof(coins[0]); ++i)
    {   // Поочередно пробуем все монеты
        sol[level-1] = 0;
        if (Sum - accumulate(sol.begin(),sol.end(),0)
             >= coins[i]) sol[level-1] = coins[i];
        else continue;  // Монету добавлять некуда
        if (Sum ==
            accumulate(sol.begin(),sol.end(),0))
        {
            // Точное решение
            cout << "Solution: ";
            for(int j = 0; j < level; ++j)
                cout << setw(4) << sol[j];
            cout << endl;
            bestSolution = sol;
        }
        else
        {
            int count = estimate(sol,Sum); // При
                // данном частичном решении
                // потребуется никак не меньше
                // монет, чем count. Эту ветвь надо
                // отсекать, если это количество
                // превышает количество монет в
                // текущем наилучшем решении
            if (count >= bestSolution.size())
                continue;
            // Пробуем дальше
            makeSolutionAtLevel(sol,level+1);
        }
    }
```

```
    sol.resize(level-1); // Вернемся на
                         // уровень вверх
}

int main()
{
    // Составляем текущее наилучшее
    // решение из копеек
    for(int i = 0; i < Sum; ++i)
        bestSolution.push_back(1);
    // Ищем оптимальное решение
    makeSolutionAtLevel(sol,1);
}
```

Эта программа выводит все получаемые в процессе работы решения, каждое из которых лучше предыдущего. Для нашего входного набора данных это всего лишь два решения — жадное и оптимальное:

```
Solution:   11   1   1   1   1
Solution:    5   5   5
```

Алгоритмы для работы с календарем

Поскольку эта тема часто вызывает живой интерес у программистов, а высказываемые мнения часто грешат массой неточностей, в этом небольшом заключительном разделе будет рассказано о нескольких простейших алгоритмах, связанных с календарем.

Небольшое теоретическое введение. Год — это время, за которое Земля совершает один полный оборот по орбите вокруг Солнца. По общепринятому соглашению считается, что неделя содержит 7 дней, месяц — от 28 до 31 дня, а год состоит из 12 месяцев. Число и месяц однозначно определяют день года.

Трудность такого счета дней в году заключается в том, что календарный год всегда содержит целое число суток, в то время как Земля совершает один полный оборот по орбите вокруг Солнца за 365,2422 суток (эта величина называется *тропическим годом*). Если не учитывать расхождение в 0,2422 суток, то за 100 лет расхождение между реальным положением Земли и календарем достигло бы 24 суток, а через полторы тысячи лет времена года поменялись бы местами.

Во времена Юлия Цезаря был принят декрет, согласно которому три последовательных года содержат по 365 дней, а четвертый, номер года которого делится нацело на 4, — содержит 366 дней (один дополнительный день добавляется к февралю). При использовании такого календаря год в среднем содержит 365,25 суток, что достаточно близко к величине тропического года (больше истинной на 11 минут 14 секунд, что за 100 лет не превышает 1 суток). Такой календарь называется юлианским, и он применялся до 1582 года, когда значительное расхождение между датами заставило папу Григория XIII исключить из календаря дни с 5 по 14 октября 1582 года включительно, и ввести правило пропуска 3 дней каждые 400 лет (средняя продолжительность года в григорианском календаре на 26 секунд превышает истинную). В григорианском календаре года, содержащие целое число сотен (1800, 1900, 2000), считаются високосными только в том случае, если они делятся на 400.

Таким образом, проверка високосности года в григорианском календаре может быть выполнена следующим образом:

```
bool IsLeapYear(int year)
{
  return (year%400==0)|||((year%4==0)&&(year%100!=0));
}
```

Введение григорианского календаря в разных странах происходило в разное время; первыми в соответствии с буллой «Inter Gravissimas» от 24.02.1582 после 4 октября 1582 года в 15 октября вступили Италия, Польша, Португалия и Испания. В России за 31 января 1918 последовало 14 февраля 1918; на Украине григорианский календарь был введен 15 февраля 1918 года (правда, в период с сентября по декабрь 1919 года григорианский календарь на Украине был отменен).

Приведенные далее алгоритмы не учитывают эти детали, и григорианский календарь считается действующим с 4 октября 1582 года. Не учитываются также ошибки в расчетах, из-за которых правило високосных лет нарушалось между 45 г. до н.э. и 12 г. н.э.

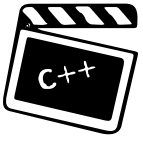
С исторической точки зрения интерес представляют также календарь, предложенный Омаром Хайямом, в котором високосными считались 8 лет из каждых 33 (погрешность такого календаря 19 секунд, что меньше, чем у григорианского), и календарь русского астронома И. Медлера, предложившего в 1864 г. поправку к юлианскому кален-

дарю, по которой пропускался один високосный год каждые 128 лет (погрешность такого календаря составляет всего лишь 1 секунду).

Несмотря на наличие различных календарных алгоритмов удобнее всего оказывается использовать так называемую юлианскую дату, принятую в астрономии. Это — количество суток, истекшее со времени гринвичского полудня 1 января 4713 г. до н.э. Поскольку здесь нас интересуют только календарные дни, а не часы или минуты, мы будем рассматривать юлианскую дату для данной календарной как целое число, т.е. считать, что мы вычисляем юлианскую дату для гринвичского полудня рассматриваемой календарной даты.

Рассчитать ее можно при помощи следующего алгоритма.

```
Алгоритм JD(y,m,d)
Вход:  календарная дата: y - год,
       m - месяц, d - день
Выход: юлианская дата
if m <= 2 then begin
    y = y - 1
    m = m + 12
end;
if Дата дана по григорианскому календарю then begin
    A = y div 100
    B = 2 - A + A div 4
end else B = 0
C = int(365.25*y)
D = int(30.6001*(m+1))
return B+C+D+d+1720995
```



Реализация на C++ очевидна:

```
long Jd(int year, int month, int day)
{
    int Grig = 0;  // Григорианская ли это дата?
    if (year > 1528) Grig = 0; else
    if (year < 1528) Grig = 1; else
    if (month > 10)  Grig = 0; else
    if (month < 10)  Grig = 1; else
```

```
    if (day >= 15)   Grig = 0; else
                     Grig = 1;
    if (month <= 2) {
        year--;
        month += 12;
    };
    unsigned long J;
    int A;
    if (Grig == 0) {
        A = year/100;
        A = 2 - A + (A / 4);
    } else A = 0;
    J = 1461L * long(year);
    J /= 4L;
    unsigned long K = 306001L*long(month + 1);
    K /= 10000L;
    J += K + day + 1720995L + A;
    return J;
};
```

Обратное преобразование юлианской даты в календарную выполняется следующим образом.

```
Алгоритм GDate(jd)
Вход: юлианская дата jd
Выход: календарная дата: day, month, year
if jd > 2299160 then begin
    A = int((jd-1867216.24)/36524.25)
    B = jd + 1 + A - int(A/4)
end else A = jd
C = B + 1524
D = (C-122.1)/365.25
E = int(365.25*D)
G = int((C-E)/30.6001)
day = C-E-int(30.6001*G)
month = G - 1
if month >= 13 then month = month - 1
year = D - 4715
if month > 2 then year = year - 1
```

Вот соответствующая функция на C++:

```
void GDate(long JD, int& y, int& m, int& d)
{
    unsigned long A=(JD*4L-7468865L)/146097L;
    A=(JD > 2299160)?JD+1+A-(A/4L):JD;
    long B=A+1524;
    long C=(B*20L-2442L)/7305L;
    long D=(C*1461L)/4L;
    long E=(10000L*(B-D))/306001L;
    d=int(B-D-((E*306001L)/10000L));
    m= int((E<=13) ? E-1 : E-13);
    y= int(C - ((m>2) ? 4716 : 4715 ));
};
```

Эти два алгоритма позволяют решить массу задач, связанных с календарем (несмотря на наличие более простых алгоритмов для решения конкретных задач). Например, порядковый номер дня в году можно получить, вычисляя разность между юлианской датой для данной календарной и юлианской датой 31 декабря предыдущего года. Количество дней между двумя датами также вычисляется как разность двух юлианских дат. И, наконец, день недели легко вычислить как остаток от деления увеличенной на 1 юлианской даты на 7:

```
int weekday(int JD) { return (JD+1)%7; }
```

При этом значение 0 соответствует воскресенью, 1 — понедельнику, ..., 6 — субботе.

И еще два алгоритма, связанных с календарем (они приводятся без описания, только в виде реализации на C++) — предвычисление католической и православной Пасхи. Используемое ныне определение даты Пасхи было дано на Первом Вселенском Соборе, состоявшемся в Византии в 325 году: *Пасха празднуется в первое воскресенье, следующее за первым полнолунием после весеннего равноденствия*. В расчетах за дату равноденствия принимают 21 марта (дата, соответствующая весеннему равноденствию в 325 году). Именно этой привязкой к действующему календарю обусловлено отличие католической Пасхи

от православной: первая использует 21 марта по григорианскому календарю, последняя — по юлианскому.

Обе функции получают в качестве аргумента год и возвращают месяц и день Пасхи. В функции предвычисления православной Пасхи включен блок перевода ее в григорианскую дату.

```
void EasterOrthodox(int year, int&mm, int&dd)
{
    int a = year % 4;
    int b = year % 7;
    int c = year % 19;
    int d = (19 * c + 15) % 30;
    int e = (2 * a + 4 * b - d + 34) % 7;
    a = d + e + 114;
    mm = a / 31;
    dd = a % 31 + 1;
    if (year > 1582) {
        d = 10 + ((year / 100 - 15) * 3) / 4;
        dd += d;
        if ((mm == 3) && (dd > 31)) {
            dd -= 31;
            mm++;
        } else if ((mm == 4) && (dd > 30)) {
            dd -= 30;
            mm++;
        };
    };
};

void EasterCatholic(int year, int&mm, int&dd)
{
    if (year <= 1582)
    {
        EasterOrthodox(year,mm,dd);
        return;
    }
    int a = year % 19;
    int b = year / 100;
    int c = year % 100;
```

```
    int g = (b - (b + 8) / 25 + 1) / 3;
    int h = (19 * a + b - b / 4 - g + 15) % 30;
    int l = (32 + 2 * (b % 4) + 2 * (c / 4) -
            h - (c % 4)) % 7;
    mm = (a + 11 * h + 22 * l) / 451;
    mm = h + l - 7 * mm + 114;
    dd = mm % 31 + 1;
    mm /= 31;
};
```

Многие другие алгоритмы, связанные с календарем, временем и астрономическими вычислениями, можно найти в книгах [7, 16].

Список литературы

1. Ахо А., Хопкрофт Д., Ульман Дж. *Структуры данных и алгоритмы.* — М.: Изд. дом «Вильямс», 2000.
2. Бентли Дж. *Жемчужины программирования, 2-е изд.* — СПб.: Питер, 2002.
3. Березин И.С., Жидков Н.П. *Методы вычислений, т. II.* — М.: Физматгиз, 1962.
4. Вирт Н. *Алгоритмы и структуры данных.* — СПб.: Невский Диалект, 2001.
5. Волков Е.А. *Численные методы.* — М.: Наука, 1982.
6. Гарднер М. *Математические досуги.* — М.: Мир, 1972.
7. Даффет-Смит П. *Практическая астрономия с калькулятором.* — М.: Мир, 1982.
8. Кнут Д. *Искусство программирования, том 1. Основные алгоритмы, 3-е изд.* — М.: Изд. дом «Вильямс», 2000.
9. Кнут Д. *Искусство программирования, том 2. Получисленные алгоритмы, 3-е изд.* — М.: Изд. дом «Вильямс», 2000.
10. Кнут Д. *Искусство программирования, том 3. Сортировка и поиск, 2-е изд.* — М.: Изд. дом «Вильямс», 2000.
11. Кобринский Н., Пекелис В. *Быстрее мысли.* — М.: Молодая гвардия, 1959.
12. Кормен, Т., Лейзерсон Ч., Ривест Р., Штайн К. *Алгоритмы: построение и анализ, 2-е изд.* — М.: Изд. дом «Вильямс», 2005.
13. Корн Г., Корн Т. *Справочник по математике (для научных работников и инженеров).* — М.: Наука, 1978.
14. Красикова И.Е., Красиков И.В. *C++. Просто как дважды два.* — М.: Эксмо, 2005.
15. Линник Ю.В. *Метод наименьших квадратов и основы математико-статистической теории обработки наблюдений.* — М.: Физматгиз, 1962.
16. Меёс Ж. *Астрономические формулы для калькуляторов.* — М.: Мир, 1988.
17. Самарский А.А., Гулин А.В. *Численные методы.* — М.: Наука, 1989.

18. Саттер Г. *Решение сложных задач на C++.* — М.: Изд. дом «Вильямс», 2002.
19. Страуструп Б. *Язык программирования C++, 3-е изд.* СПб.; М.: «Невский Диалект» — «Издательство БИНОМ», 1999 г.
20. Уоррен Г. С. *Алгоритмические трюки для программистов.* — М.: Изд. дом «Вильямс», 2003.
21. Levitin A. *Introduction to The Design & Analysis of Algorithms.* Addison-Wesley, Reading, MA, 2003.
22. Sedgewick, R. *Algorithms in C++. 3rd edition.* Addison-Wesley, Reading, MA, 1999.

Предметный указатель

Т

У

Ф

Х

Ч

Ш

Научно-популярное издание

Красиков Игорь Владимирович
Красикова Ирина Евгеньевна

АЛГОРИТМЫ

ПРОСТО КАК ДВАЖДЫ ДВА

Зав. редакцией *И. Е. Федосова*
Ответственный редактор *В. В. Александров*
Литературный редактор *А. А. Макиевская*
Художественный редактор *Н. С. Никонова*
Верстка *Р. А. Марчишин*
Корректор *А. А. Гловацкая*

ООО «Издательство «Эксмо»
127299, Москва, ул. Клары Цеткин, д. 18/5. Тел. 411-68-86, 956-39-21.
Home page: **www.eksmo.ru** E-mail: **info@eksmo.ru**

Оптовая торговля книгами «Эксмо»:
ООО «ТД «Эксмо». 142702, Московская обл., Ленинский р-н, г. Видное, Белокаменное ш., д. 1, многоканальный тел. 411-50-74.
E-mail: **reception@eksmo-sale.ru**

По вопросам приобретения книг «Эксмо» зарубежными оптовыми покупателями *обращаться в отдел зарубежных продаж ООО «ТД «Эксмо»*
E-mail: **foreignseller@eksmo-sale.ru**

International Sales: *For Foreign wholesale orders, please contact International Sales Department at* **foreignseller@eksmo-sale.ru**

По вопросам заказа книг «Эксмо» в специальном оформлении *обращаться в отдел корпоративных продаж ООО «ТД «Эксмо»* E-mail: **project@eksmo-sale.ru**

Оптовая торговля бумажно-беловыми и канцелярскими товарами для школы и офиса «Канц-Эксмо»: Компания «Канц-Эксмо»: 142700, Московская обл., Ленинский р-н, г. Видное-2, Белокаменное ш., д. 1, а/я 5. Тел./факс +7 (495) 745-28-87 (многоканальный). e-mail: **kanc@eksmo-sale.ru**, сайт: **www.kanc-eksmo.ru**

Полный ассортимент книг издательства «Эксмо» для оптовых покупателей:
В Санкт-Петербурге: ООО СЗКО, пр-т Обуховской Обороны, д. 84Е. Тел. (812) 365-46-03/04. **В Нижнем Новгороде:** ООО ТД «Эксмо НН», ул. Маршала Воронова, д. 3. Тел. (8312) 72-36-70. **В Казани:** ООО «НКП Казань», ул. Фрезерная, д. 5. Тел. (843) 570-40-45/46. **В Самаре:** ООО «РДЦ-Самара», пр-т Кирова, д. 75/1, литера «Е». Тел. (846) 269-66-70. **В Ростове-на-Дону:** ООО «РДЦ-Ростов», пр. Стачки, 243А. Тел. (863) 268-83-59/60. **В Екатеринбурге:** ООО «РДЦ-Екатеринбург», ул. Прибалтийская, д. 24а. Тел. (343) 378-49-45. **В Киеве:** ООО ДЦ «Эксмо-Украина», ул. Луговая, д. 9. Тел./факс: (044) 537-35-52. **Во Львове:** ТП ООО ДЦ «Эксмо-Украина», ул. Бузкова, д. 2. Тел./факс: (032) 245-00-19. **В Симферополе:** ООО «Эксмо-Крым» ул. Киевская, д. 153. Тел./факс (0652) 22-90-03, 54-32-99.

Мелкооптовая торговля книгами «Эксмо» и канцтоварами «Канц-Эксмо»:
117192, Москва, Мичуринский пр-т, д. 12/1. Тел./факс: (495) 411-50-76.
127254, Москва, ул. Добролюбова, д. 2. Тел.: (495) 745-89-15, 780-58-34.

Подписано в печать 06.02.2007.
Формат 84×108 $^{1}/_{32}$. Печать офсетная. Бумага писчая.
Усл. печ. л. 13,44. Тираж 3 000 экз. Заказ № 101.

Отпечатано в ОАО «ИПП «Уральский рабочий»
620041, ГСП-148, г. Екатеринбург, ул. Тургенева, 13
http://www.uralprint.ru
e-mail: book@uralprint.ru