# ГЛАВА 2

# Разработка эффективных алгоритмов

Цель этой главы имеет двоякий характер. Во-первых, мы вводим в рассмотрение фундаментальные структуры данных, которые полезны при разработке эффективных алгоритмов для широких классов задач. Во-вторых, описываем методы программирования, такие как рекурсия и динамическое программирование, которые используются во многих эффективных алгоритмах.

В главе l описаны основные модели вычислений. Несмотря на то что нашей первичной моделью является RAM, как правило, мы не будем описывать алгоритмы в терминах такого устройства. По этой причине мы ввели в рассмотрение псевдо-Алгол (см. раздел 1.8). Но даже этот язык оказывается слишком примитивным, если не использовать структуры данных, более сложные, чем массивы. В начале главы рассматриваются такие элементарные структуры данных, как списки и стеки, которые часто используются в эффективных алгоритмах. Мы продемонстрируем, как с помощью этих структур можно представлять множества, графы и деревья. Наше изложение по необходимости будет лаконичным, и читателям, не знакомым с методами обработки списков, следует обратиться к одному из первоисточников, указанных в конце главы, или уделить особое внимание упражнениям.

Мы включили в главу раздел о рекурсии. Один из важных аспектов рекурсии — концептуальное упрощение алгоритмов. Несмотря на то что примеры, приведенные в этой главе, слишком просты, чтобы считать их полноценным обоснованием этого утверждения, в последующих главах рекурсия позволит избежать громоздких подробностей без потери точности изложения сложных алгоритмов. Рекурсия сама по себе не всегда гарантирует эффективность алгоритмов. Однако в сочетании с другими методами, такими как балансировка, “разделяй и властвуй“ и алгебраические упрощения, она, как мы увидим, часто позволяет создать эффективные и элегантные алгоритмы.

## 2.1. Структуры данных: списки, очереди и стеки

Мы предполагаем, что читатели знакомы с элементарными понятиями математики такими, как множества и отношения, а также основными типами данных, такими как целые числа, строки и массивы. В этом разделе мы приводим краткий обзор основных операций над списками.

С математической точки зрения *список* — это конечная последовательность элементов, извлеченных из некоего множества. В описаниях алгоритмов часто используются списки, в которые добавляются и из которых удаляются элементы. В частности, мы можем пожелать добавить или удалить элемент где-то в середине списка. По этой причине нам необходимы структуры данных, позволяющие реализовать списки, в которых можно удалять и добавлять новые элементы без ограничений.

Рассмотрим список

Элемент 1, Элемент 2, Элемент 3, Элемент 4. (2.1)

Простейшей его реализацией является односвязная структура, изображенная на рис. 2.1. Каждый элемент в этой структуре состоит из двух ячеек памяти. Первая ячейка содержит сам элемент,[1] а вторая — указатель на следующий элемент. Эту структуру можно реализовать в виде двух массивов, которые на рис. 2.2 названы NAME и NEXT.[2] Если ITEM — это индекс в рассматриваемом массиве, то NAME[ITEM] — элемент, хранящийся в массиве, а NEXT[ITEM] — индекс следующего элемента в списке, если такой элемент существует. Если ITEM — индекс последнего элемента в списке, то NEXT[ITEM] = 0.

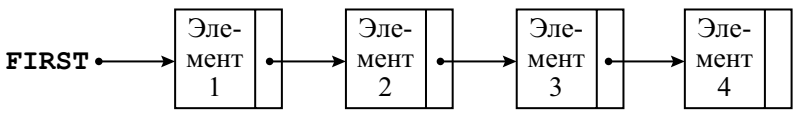



**Рис. 2.1.** Связанный список

На рис. 2.2 выражение NEXT[0] означает постоянный указатель на первый элемент списка. Заметим, что порядок элементов в массиве NAME не совпадает с порядком элементов в списке. Однако рис. 2.2 правильно изображает список, показанный на рис. 2.1, поскольку массив NEXT располагает элементы в том же порядке, в котором они следуют в списке (2.1).

Следующая процедура вставляет элемент в список. Предполагается, что FREE — это индекс свободной ячейки в массивах NAME и NEXT, а POSITION — индекс элемента в списке, после которого следует вставить ITEM.

---

[1] Если этот элемент сам является сложной структурой, то первая ячейка может содержать указатель на него.

[2] Существует альтернативная (и эквивалентная) точка зрения, в соответствии с которой считается, что каждый элемент хранится в отдельной ячейке. Каждая ячейка обладает адресом, представляющим собой номер первого (возможно, единственного) регистра памяти в группе регистров, зарезервированных для данного элемента. Ячейка состоит из одного или нескольких полей. В данном случае ячейка состоит из полей NAME и NEXT, а выражения NAME[ITEM] и NEXT[ITEM] — это ссылки на эти поля в ячейке с адресом ITEM.

```
procedure INSERT(ITEM, FREE, POSITION):
begin
      NAME[FREE] ← ITEM
      NEXT[FREE] ← NEXT[POSITION]
      NEXT[POSITION] ← FREE
end
```

Трансляция этой процедуры в команды RAM приведет к тому, что время выполнения процедуры `INSERT` не будет зависеть от размера списка.

**Пример 2.1.** Допустим, мы хотим вставить в список (2.1) новый элемент после второго элемента и в результате получить список

Элемент 1, Элемент 2, Новый элемент, Элемент 3, Элемент 4.

Если пятая ячейка в каждом массиве на рис. 2.2 свободна, то можно вставить новый элемент после Элемента 2 (позиция 3), вызвав процедуру `INSERT`(Новый элемент, `5`, `3`). В результате выполнения трех инструкций в процедуре `INSERT` получим: `NAME[5]`= Новый элемент, `NEXT[5]=4` и `NEXT[3]=5`. В итоге будут созданы массивы, показанные на рис. 2.3.

| | NAME | NEXT |
|---|---|---|
| 0 | — | 1 |
| 1 | Элемент 1 | 3 |
| 2 | Элемент 4 | 0 |
| 3 | Элемент 2 | 4 |
| 4 | Элемент 3 | 2 |

**Рис. 2.2.** Представление списка из четырех элементов

| | NAME | NEXT |
|---|---|---|
| 0 | — | 1 |
| 1 | Элемент 1 | 3 |
| 2 | Элемент 4 | 0 |
| 3 | Элемент 2 | 5 |
| 4 | Элемент 5 | 2 |
| 5 | Новый элемент | 4 |

**Рис. 2.3.** Список со вставленным новым элементом

Для того чтобы удалить элемент, следующий за элементом в ячейке $I$, можно выполнить присваивание `NEXT`[$I$] = `NEXT`(`NEXT`[$I$]). При желании индекс удаленного элемента можно записать в список свободных ячеек памяти.

Часто в одном и том же массиве хранятся несколько списков. Как правило, один из этих списков состоит из свободных ячеек. Назовем его *свободным списком*. Для

добавления элемента к списку $A$ можно изменить процедуру INSERT, так чтобы свободная ячейка возникала с помощью удаления первой ячейки в свободном списке. При удалении элемента из списка $A$ соответствующая ячейка возвращается в свободный список для повторного использования.

Этот способ организации памяти не единственно возможный, но он описан здесь для того, чтобы показать, что операции добавления и удаления элементов списка можно выполнить за конечное число шагов, если местоположение элемента, который мы хотим добавить или удалить, определено заранее.

Существуют еще две базовые операции над списками — конкатенация двух списков, т.е. объединение их в один список, и обратная к ней операция разделения списка, после которой исходный список разделяется на два списка, один из которых начинается с элемента, следующего после заданного. Конкатенацию можно выполнить за ограниченное число шагов, добавив в представление списка дополнительный указатель. Этот указатель содержит индекс последнего элемента списка и позволяет не просматривать весь список в поиска его последнего элемента. Разделение списка можно выполнить за ограниченное время, если известен индекс элемента, непосредственно предшествующего месту разделения.

Списки можно обходить в обоих направлениях, если добавить еще один массив с именем PREVIOUS. Значение PREVIOUS[$I$] — это адрес ячейки, в которой находится элемент списка, который стоит непосредственно перед элементом из ячейки $I$. Список такого рода называется *двусвязным*. Для вставки и удаления элемента двусвязного списка не обязательно знать адрес предыдущего элемента.

Часто работа со списками ограничивается очень небольшим числом операций. Например, иногда элементы добавляются или удаляются только в конце списка. Иначе говоря, элементы вставляются и удаляются по принципу “последний вошел — первый вышел”. В этом случае список называют *стеком* или *магазином*.

Часто стек реализуется в виде отдельного массива. Например, список

Элемент 1, Элемент 2, Элемент 3

можно было бы хранить в массиве NAME, как показано на рис. 2.4. Переменная TOP является указателем на последний элемент, добавленный в стек. Чтобы затолкнуть (PUSH) новый элемент в стек, значение TOP увеличивают на единицу, а затем помещают новый элемент в ячейку NAME[TOP]. (Поскольку массив NAME имеет конечную длину $l$, перед вставкой нового элемента следует проверить условие TOP $< l - 1$.) Чтобы вытолкнуть (POP) элемент из вершины стека, надо просто уменьшить значение TOP на единицу. Заметим, что не обязательно физически стирать элемент, удаляемый из стека. Чтобы узнать, пуст ли стек, достаточно проверить условие TOP $< 0$. Очевидно, что время выполнения операций PUSH и POP, а также проверка пустоты стека не зависят от количества элементов в стеке.

Другой специальный вид списка — *очередь*, т.е. список, в который элементы всегда добавляются с одного конца (FRONT), а удаляются с другого (REAR). Как

и стек, очередь можно реализовать в виде массива (рис. 2.5), где продемонстрирована очередь, содержащая список из элементов *P*, *Q*, *R*, *S*, *T*. Два указателя обозначают ячейки, соответствующие голове (front) и хвосту (rear) очереди в текущий момент. Чтобы вставить (`ENQUEUE`) новый элемент в очередь, как и в случае стека, выполняют операцию `FRONT = FRONT + 1` и помещают новый элемент в ячейку `NAME[FRONT]`. Чтобы удалить (`DEQUEUE`) элемент из очереди, выполняют операцию `REAR = REAR + 1`. Заметим, что этот способ доступа к элементам основан на принципе "первый вошел — первый вышел".


**Рис. 2.4.** Реализация стека

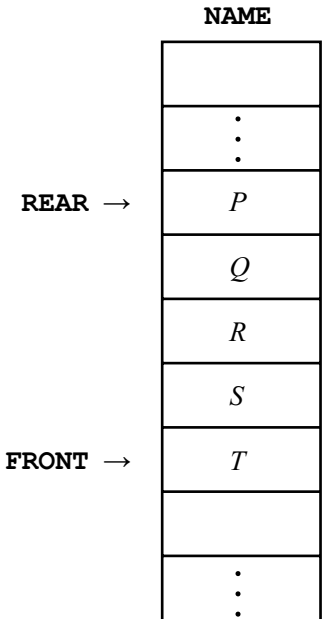



**Рис. 2.5.** Реализация очереди в виде массива

Поскольку массив `NAME` имеет конечную длину, скажем, *l*, то указатели `FRONT` и `REAR` рано или поздно достигнут его конца. Если длина списка, представленного этой очередью, никогда не превосходит *l*, то можно интерпретировать выражение `NAME[0]` как элемент, следующий за элементом `NAME[`*l* − 1`]`.

Элементы, организованные в виде списка, сами могут быть сложными структурами. Например, при работе со списком массивов на самом деле добавляются и удаляются не массивы, а указатели на них, поскольку каждое добавление или удаление массива потребовало бы времени, пропорционального его размеру. Таким образом, сложную структуру можно добавить или удалить за фиксированное время, не зависящее от ее размера.

## 2.2. Представления множеств

Обычно списки применяются для представления множеств. В этом случае объем памяти, необходимый для представления множества, пропорционален числу его элементов. Количество времени, требуемое для выполнения операции над множествами, зависит от ее природы. Например, предположим, что $A$ и $B$ — два множества. Операция $A \cap B$ требует времени, по крайней мере, пропорционального сумме размеров этих множеств, поскольку списки, представляющие множества $A$ и $B$, надо просмотреть хотя бы один раз.[3]

Аналогично для выполнения операции $A \cup B$ требуется количество времени, пропорциональное сумме размеров множеств, поскольку надо найти элементы, входящие в оба множества, и удалить один экземпляр каждого такого элемента. Если же множества $A$ и $B$ не пересекаются, найти $A \cup B$ можно за время, не зависящее от размера $A$ и $B$, выполнив конкатенацию списков, представляющих множества $A$ и $B$. Задача объединения двух непересекающихся множеств усложняется, если необходимо быстро определить, входит ли заданный элемент в заданное множество. Этот вопрос подробно обсуждается в разделах 4.6 и 4.7.

Альтернативный способ представления множества — вектор битов. Пусть $U$ — универсальное множество (т.е. множество всех множеств), состоящее из $n$ элементов. Упорядочим его линейно. Подмножество $S \subseteq U$ представляется в виде вектора $v_S$ из $n$ битов, в котором $i$-й бит равен единице тогда и только тогда, когда $i$-й элемент множества $U$ принадлежит множеству $S$. Будем называть $v_S$ *характеристическим вектором* множества $S$.

Преимущество представления множества в виде вектора битов заключается в том, что оно позволяет определять принадлежность $i$-го элемента множества $U$ заданному множеству за время, не зависящее от размера заданного множества. Более того, основные операции над множествами, такие как объединение и пересечение, можно осуществить как операции $\vee$ и $\wedge$ над векторами битов.

Если мы не хотим считать операции над векторами битов элементарными (выполняемыми за единицу времени), то можно с таким же успехом вместо характеристического вектора определить массив $A$, для которого $A[i] = 1$ тогда и только тогда, когда $i$-й элемент множества $U$ принадлежит $S$. При таком представлении также нетрудно определить, принадлежит ли данный элемент данному множеству. Недостаток этого представления заключается в том, что для выполнения операций объединения и пересечения требуется количество времени, пропорциональное $||U||$,[4] а не размерам рассматриваемых множеств. Аналогично объем памяти, необходимой для хранения множества $S$, пропорционален $||U||$, а не $||S||$.

---

[3] Если оба списка упорядочены, то найти их пересечение можно с помощью линейного алгоритма.

[4] Здесь $||X||$ обозначает количество элементов (размер, или мощность) множества $X$.

## 2.3. Графы

Введем математическое понятие графа и рассмотрим структуры данных, которые обычно применяются для его представления.

***Определение.*** *Граф* $G = (V, E)$ состоит из конечного непустого множества *вершин* $V$ и множества *ребер* $E$. Если ребра представлены в виде упорядоченных пар вершин $(v, w)$, то граф называется *ориентированным*. Вершина $v$ называется *началом*, а вершина $w$ — *концом* ребра $(v, w)$. Если ребра представляют собой неупорядоченные пары (множества) различных вершин, также обозначаемые $(v, w)$, то граф называют *неориентированным*.[5]

Если в ориентированном графе $G = (V, E)$ пара $(v, w)$ принадлежит множеству ребер $E$, то вершина $w$ называется *смежной* с вершиной $v$. Говорят, что ребро $(v, w)$ *ведет из* $v$ *в* $w$. Количество вершин, смежных с вершиной $v$, называется *полустепенью исхода* вершины $v$.

Если в неориентированном графе $G = (V, E)$ пара $(v, w)$ принадлежит множеству ребер $E$, то считается, что $(v, w) = (w, v)$, так что $(w, v)$ — то же самое ребро. Вершина $w$ называется *смежной* с вершиной $v$, если $(v, w)$ (следовательно, и $(w, v)$) принадлежит множеству $E$. *Степень* вершины — это количество вершин, смежных с ней.

*Путем* в ориентированном или неориентированном графе называют последовательность ребер вида $(v_1, v_2)$, $(v_2, v_3)$, ..., $(v_{n-1}, v_n)$. Говорят, что этот путь *ведет из* $v_1$ *в* $v_n$, а его *длина* равна $n - 1$. Часто такой путь представляют в виде последовательности $v_1, v_2, ..., v_n$ вершин, лежащих на нем. В вырожденном случае одна вершина обозначает путь нулевой длины, ведущий из этой вершины в нее же. Путь называется *простым*, если все ребра и все вершины на этом пути, кроме, возможно, первой и последней, отличаются одна от другой. *Цикл* — это простой путь длины не менее единицы, который начинается и заканчивается в одной и той же вершине. Заметим, что в неориентированном графе длина цикла должна быть не менее трех.

Существует несколько представлений графа $G = (V, E)$. Один из них — *матрица смежности*, т.е. матрица $A$ размером $||V|| \times ||V||$, состоящая из нулей и единиц, в которой $A[i, j] = 1$ тогда и только тогда, когда существует ребро, ведущее из вершины $i$ в вершину $j$. Представление в виде матрицы смежности удобно для тех алгоритмов на графах, в которых часто требуется знать, существует ли в графе заданное ребро, поскольку время, необходимое для проверки наличия ребра, фиксировано и не зависит от $||V||$ и $||E||$. Основной недостаток матрицы смежности заключается в том, что она занимает память объема $||V||^2$ даже тогда, когда граф содержит только $O(||V||)$ ребер. Даже простая инициализация матрицы смежности требует времени порядка $O(||V||^2)$, что препятствует созданию алгоритмов сложности $O(||V||)$ для работы

[5] Заметим, что в ориентированном графе может существовать ребро $(a, a)$, а в неориентированном — нет.

с графами, содержащими $O(\|V\|)$ ребер. Несмотря на то что существуют методы для преодоления этой трудности (см. упражнение 2.12), почти неизбежно возникают другие проблемы, которые приводят к тому, что алгоритмы сложности $O(\|V\|)$, основанные на работе с матрицей смежности, являются редкостью.

Интересной альтернативой является представление строк и/или столбцов матрицы смежности в виде векторов битов. Такое представление может значительно повысить эффективность алгоритмов на графах.

Еще одно возможное представление графа основано на списках. *Списком смежности* для вершины $v$ называется список всех вершин $w$, смежных с вершиной $v$. Граф можно представить с помощью $\|V\|$ списков смежности, по одному для каждой вершины.

**Пример 2.2.** На рис. 2.6, *а*, изображен ориентированный граф, содержащий четыре вершины, а на рис. 2.6, *б*, — его матрица смежности. На рис. 2.6, *в*, показаны четыре списка смежности, по одному для каждой вершины. Например, в графе есть ребра, ведущие из вершины 1 в вершины 2 и 4, поэтому список смежности для вершины 1 содержит элементы 2 и 4, связанные так, как показано на рис. 2.1.

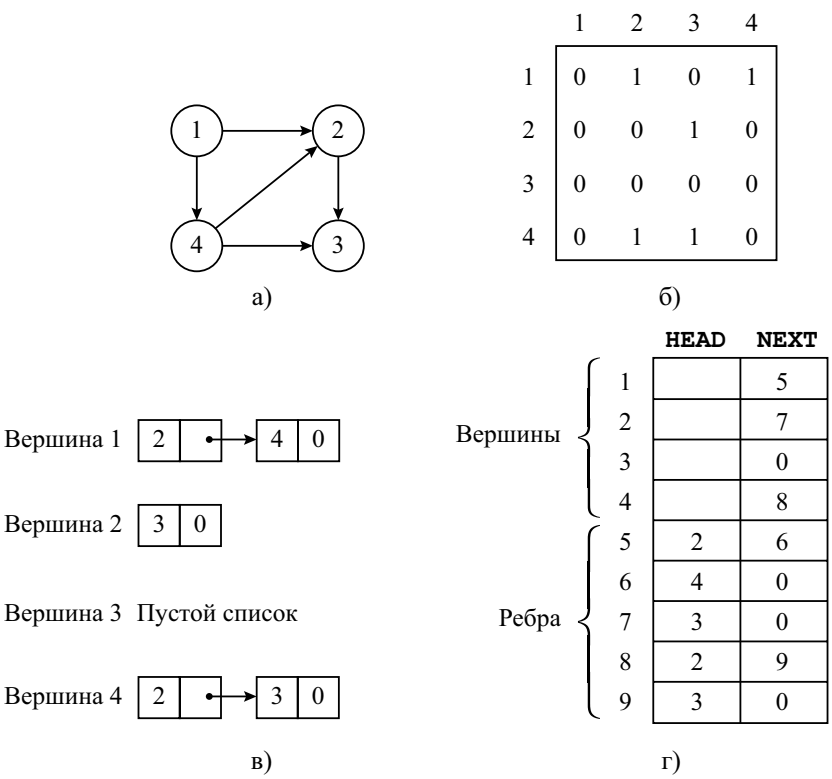



**Рис. 2.6.** Ориентированный граф и его представления: (*а*) ориентированный граф; (*б*) матрица смежности; (*в*) списки смежности; (*г*) табличное представление списков смежности

Табличное представление списков смежности приведено на рис. 2.6, *г*. Каждая из первых четырех ячеек в массиве NEXT содержит указатель на первую вершину списка смежности, причем указатель NEXT[*i*] ссылается на первую вершину списка смежности для вершины i. Заметим, что NEXT[3]=0, поскольку список смежности для вершины 3 пуст. Остальные элементы массива NEXT представляют ребра графа. Массив HEAD содержит вершины из списков смежности. Таким образом, список смежности вершины 1 начинается в ячейке 5, поскольку NEXT[1]=5, HEAD[5]=2; это показывает, что существует ребро (1, 2). Равенства NEXT[5]=6 и HEAD[6]=4 означают, что существует ребро (1, 4), а равенство NEXT[6]=0 означает, что больше нет ребер, начинающихся в вершине 1. □

Заметим, что представление графа в виде списков смежности требует объема памяти порядка $||V|| + ||E||$. Представлением с помощью списков смежности часто пользуются, когда $||E|| \ll ||V||^2$.

Если граф является неориентированным, то каждое ребро $(v, w)$ представляется дважды: один раз в списке смежности для вершины $v$ и один раз в списке смежности для вершины $w$. В этом случае можно добавить новый массив с именем LINK, чтобы согласовать оба экземпляра неориентированного ребра. Таким образом, если $i$ — ячейка, соответствующая вершине $w$ в списке смежности для вершины $v$, то LINK[*i*] — ячейка, соответствующая вершине $v$ в списке смежности для вершины $w$.

Чтобы эффективно удалять ребра из неориентированного графа, списки смежности можно связать дважды (как описано в разделе 2.1). Это бывает необходимо потому, что даже если всегда удалять ребро $(v, w)$, стоящее первым в списке смежности вершины $v$, то все равно может оказаться, что ребро, ведущее в обратном направлении, стоит в середине списка смежности вершины $w$. Чтобы быстро удалить ребро $(v, w)$ из списка смежности для вершины $w$, надо уметь быстро находить ячейку, содержащую предыдущее ребро в этом списке смежности.

## 2.4. Деревья

Теперь введем очень важный вид ориентированных графов — деревья — и рассмотрим структуры данных, подходящие для их представления.

***Определение.*** Ориентированный граф без циклов называется *ориентированным ациклическим графом*. *Ориентированное дерево* (иногда его называют *корневым деревом*) — это ориентированный ациклический граф, удовлетворяющий следующим условиям.

1. Существует только одна вершина, называемая *корнем*, в которую не входит ни одно ребро.

2. В каждую вершину, кроме корня, входит только одно ребро.
3. Из корня в каждую вершину ведет путь (который, как легко показать, единственный).

Ориентированный граф, состоящий из нескольких деревьев, называется *лесом*. Леса и деревья представляют собой частные случаи ориентированных ациклических графов, которые встречаются настолько часто, что для описания их свойств разработана специальная терминология.

***Определение.*** Пусть $F = (V, E)$ — граф, являющийся лесом. Если $(v, w)$ принадлежит множеству $E$, то $v$ называется *отцом* вершины $w$, а $w$ — *сыном* вершины $v$. Если существует путь из $v$ в $w$, то $v$ называется *предком* вершины $w$, а $w$ — *потомком* вершины $v$. Более того, если $v \neq w$, то $v$ называется *прямым предком* вершины $w$, а $w$ — *прямым потомком* вершины $v$. Вершина без прямых потомков называется *листом*. Вершина $v$ и ее потомки в совокупности образуют *поддерево* леса $F$, и вершина $v$ называется *корнем* этого поддерева.

*Глубина вершины v* в дереве — это длина пути из корня в вершину $v$. *Высота вершины v* в дереве — это длина самого длинного пути из вершины $v$ в какой-нибудь лист. *Высотой дерева* называется высота его корня. *Уровень вершины v* в дереве равен разности высоты дерева и глубины вершины $v$. Например, на рис. 2.7, *а*, вершина 3 имеет глубину 2, высоту 0 и уровень 1.

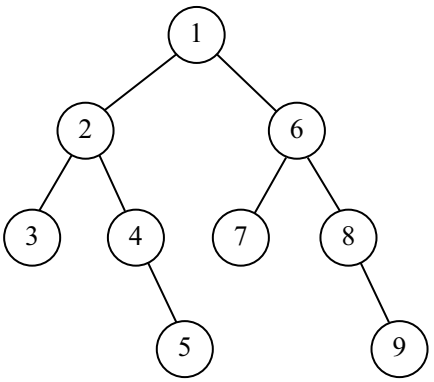
*Упорядоченным деревом* называется дерево, в котором множество сыновей каждой вершины упорядочено. При изображении упорядоченного дерева будем считать, что множество сыновей каждой вершины упорядочено слева направо. *Бинарным деревом* называется такое упорядоченное дерево, что

1) каждый сын произвольной вершины идентифицируется либо как *левый сын*, либо как *правый*;
2) каждая вершина имеет не более одного левого сына и не более одного правого.

Поддерево $T_l$ (если оно существует), корнем которого является левый сын вершины $v$, называется *левым поддеревом* вершины $v$. Аналогично поддерево $T_r$ (если оно существует), корнем которого является правый сын вершины $v$, называется *правым поддеревом* вершины $v$. Все вершины в поддереве $T_l$ расположены левее всех вершин в поддереве $T_r$.

Бинарное дерево обычно представляют в виде двух массивов `LEFTSON` и `RIGHTSON`. Пусть вершины бинарного дерева занумерованы целыми числами от 1 до $n$. В этом случае условие `LEFTSON[`$i$`]` $= j$ выполняется тогда и только тогда, когда вершина с номером $j$ является левым сыном вершины с номером $i$.

Если у вершины $i$ нет левого сына, то `LEFTSON[i]` = 0. Условия для элемента `RIGHTSON[i]` определяются аналогично.




а)

| | LEFTSON | RIGHTSON |
|---|---|---|
| 1 | 2 | 6 |
| 2 | 3 | 4 |
| 3 | 0 | 0 |
| 4 | 0 | 5 |
| 5 | 0 | 0 |
| 6 | 7 | 8 |
| 7 | 0 | 0 |
| 8 | 0 | 9 |
| 9 | 0 | 0 |

б)

**Рис. 2.7.** Бинарное дерево и его представление

**Пример 2.3.** Бинарное дерево и его представление изображены на рис. 2.7, *а* и *б*. □

***Определение.*** Бинарное дерево называется *полным*, если для некоторого целого числа $k$ каждая вершина глубины, меньшей $k$, имеет как левого, так и правого сына, и каждая вершина глубины $k$ является листом. Полное бинарное дерево высоты $k$ имеет ровно $2^{k+1} - 1$ вершин.

Полное бинарное дерево высоты $k$ часто представляют с помощью одного массива. В первой ячейке этого массива находится корень. Левый сын вершины в ячейке $i$ расположен в ячейке $2i$, а его правый сын — в ячейке $2i + 1$. Отец вершины, находящейся в положении $i > 1$, расположен в ячейке $\lfloor i / 2 \rfloor$.

Многие алгоритмы, использующие деревья, часто *обходят* дерево (посещают каждую его вершину) в определенном порядке. Известно несколько систематических способов обхода деревьев Мы рассмотрим три широко распространенных способа обхода: прямой, обратный и симметричный.

***Определение.*** Пусть $T$ — дерево с корнем $r$ и сыновьями $v_1, v_2, ..., v_k$, $k \geq 0$. При $k = 0$ это дерево состоит из единственной вершины $r$.

*Прямой обход* дерева $T$ определяется следующей рекурсией:

**1)** посетить корень $r$;

**2)** выполнить прямой обход поддеревьев с корнями $v_1, v_2, ..., v_k$ в указанной последовательности.

*Обратный обход* дерева $T$ определяется следующей рекурсией:

**1)** выполнить обратный обход поддеревьев с корнями $v_1, v_2, ..., v_k$ в указанной последовательности;

**2)** посетить корень $r$.

*Симметричный обход* бинарного дерева определяется следующей рекурсией:

**1)** выполнить симметричный обход левого поддерева корня (если оно существует);

**2)** посетить корень;

**3)** выполнить симметричный обход правого поддерева корня (если оно существует).

**Пример 2.4.** На рис. 2.8 изображено бинарное дерево, вершины которого пронумерованы в соответствии с прямым (рис. 2.8, *а*), обратным (рис. 2.8, *б*) и симметричным обходом (рис. 2.8, *в*). □

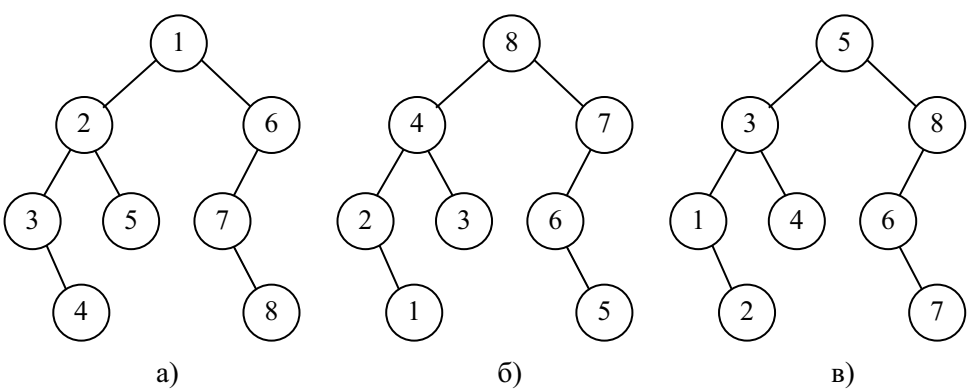



**Рис. 2.8.** Обход вершин дерева: (*а*) прямой, (*б*) обратный, (*в*) симметричный

Если при обходе дерева его вершинам были присвоены номера, то на вершины удобно ссылаться по этим номерам. Тогда $v$ будет обозначать вершину, которой был присвоен номер $v$. Если все вершины занумерованы в порядке обхода, то рассмотренные нумерации обладают рядом интересных свойств.

При нумерации в прямом обходе все вершины поддерева с корнем $r$ имеют номера, не меньшие $r$. Точнее, если $D$ — множество потомков вершины $r$, то $v$ будет номером некоторой вершины из множества $D$ тогда и только тогда, когда $r \le v < r + ||D_r||$. Поставив в соответствие каждой вершине $v$ ее номер в прямом обходе и количество ее потомков, легко определить, является ли вершина $w$ потомком вершины $v$. Присвоив вершинам номера в прямом обходе и вычислив количество потомков каждой вершины, на вопрос, является ли вершина $w$ потомком вершины $v$, можно ответить за фиксированное время, не зависящее от размера дерева. Номера, соответствующие обратному обходу, обладают аналогичным свойством.

При нумерации вершин бинарного дерева в симметричном обходе номера вершин в левом поддереве с корнем $v$ меньше $v$, а в правом поддереве — больше $v$. Таким образом, чтобы найти вершину с номером $w$, надо сравнить $w$ с номером корня $r$. Если $w = r$, то искомая вершина найдена. Если $w < r$, следует повторить этот процесс для левого поддерева, если $w > r$ — для правого поддерева. В конце концов, вершина с номером $w$ будет найдена. Такие свойства обхода нам понадобятся в следующих главах.

Дадим теперь еще одно, последнее определение, касающееся деревьев.

---

***Определение.*** *Неориентированным деревом* называется неориентированный ациклический *связный* граф (т.е. неориентированный ациклический граф, любые две вершины которого соединены путем). *Корневое* неориентированное дерево — это неориентированное дерево, в котором одна вершина выделена в качестве корня.

---

Ориентированное дерево можно превратить в корневое неориентированное, просто сделав все его ребра неориентированными. Мы будем использовать одни и те же термины и обозначения как для неориентированных, так и ориентированных корневых деревьев. Основное математическое различие между ними заключается в том, что все пути в ориентированном дереве ведут от предков к потомкам, а в корневом неориентированном дереве пути могут вести в обоих направлениях.

## 2.5. Рекурсия

Процедура, которая прямо или косвенно вызывает сама себя, называется *рекурсивной*. Применение рекурсии часто позволяет создавать более четкие и лаконичные описания алгоритмов. В настоящем разделе приводится пример рекурсивного алгоритма и кратко описана реализация рекурсии на машинах RAM.

Рассмотрим определение симметричного обхода бинарного дерева, данное в разделе 2.4. При создании алгоритма, который присваивает вершинам номера в соответствии с симметричным обходом, хорошо было бы отразить в нем опреде-

ление симметричного обхода. Один из таких алгоритмов приведен ниже. Заметим, что он рекурсивно обращается к себе для нумерации поддерева.

**Алгоритм 2.1.** Нумерация вершин бинарного дерева в соответствии с порядком симметричного обхода.

*Вход*. Бинарное дерево, представленное массивами LEFTSON и RIGHTSON.

*Выход*. Массив NUMBER, такой, что NUMBER[*i*] — номер вершины *i* в порядке симметричного обхода.

*Метод*. Кроме массивов LEFTSON, RIGHTSON и NUMBER, алгоритм использует глобальную переменную COUNT, значением которой является номер очередной вершины в соответствии с симметричным порядком обхода. Начальное значение переменной COUNT равно единице. Параметр VERTEX вначале равен корню. Процедура, изображенная на рис. 2.9, применяется рекурсивно.

```
    procedure INORDER(VERTEX);
    begin
1.      if LEFTSON[VERTEX] ≠ 0 then
            INORDER(LEFTSON[VERTEX]);
2.      NUMBER[VERTEX] ← COUNT;
3.      COUNT ← COUNT + 1;
4.      if RIGHTSON[VERTEX] ≠ 0 then
            INORDER(RIGHTSON[VERTEX]);
    end
```

**Рис. 2.9.** Рекурсивная процедура симметричного обхода

Сам алгоритм имеет следующий вид.

```
    begin
        COUNT ← 1;
        INORDER(ROOT)
    end  □
```

Рекурсия имеет несколько преимуществ, главное из которых — программы становятся понятнее. Если бы приведенный выше алгоритм не был записан в рекурсивном виде, пришлось бы создать явный механизм для обхода дерева. Двигаться вниз по дереву нетрудно, но, чтобы обеспечить возможность вернуться к предку, необходимо помнить предков в стеке, а инструкции для работы со стеком усложнили бы алгоритм. Его нерекурсивная версия выглядела бы примерно следующим образом.

**Алгоритм 2.2.** Нерекурсивная версия алгоритма 2.1.

*Вход*. Как у алгоритма 2.1.

*Выход*. Как у алгоритма 2.1.

*Метод*. При прохождении дерева в стеке запоминаются все вершины, которые еще не были пронумерованы и которые лежат на пути из корня в текущую вершину. При переходе из вершины $v$ к ее левому сыну вершина $v$ запоминается в стеке. После обнаружения левого поддерева вершины $v$ она нумеруется и выталкивается из стека. Затем нумеруется правое поддерево вершины $v$.

При переходе из вершины $v$ к ее правому сыну она не заталкивается в стек, поскольку после нумерации правого поддерева мы планируем возвращаться в вершину $v$. Вместо этого мы хотим вернуться к тому предку вершины $v$, который еще не был пронумерован (т.е. к ближайшему предку $w$ вершины $v$, такому, что $v$ лежит в левом поддереве вершины $w$). Этот алгоритм приведен на рис. 2.10. □

```
        begin
            COUNT ← 1;
            VERTEX ← ROOT;
            STACK ← пуст;
left:       while LEFTSON[VERTEX] ≠ 0 do
                begin
                    затолкнуть VERTEX в STACK
                    VERTEX ← LEFTSON[VERTEX]
                end;
center:     NUMBER[VERTEX] ← COUNT;
            COUNT ← COUNT + 1;
            if RIGHTSON[VERTEX] ≠ 0 then
                begin
                    VERTEX ← RIGHTSON[VERTEX];
                    goto left;
                end;
            if STACK не пуст then
                begin
                    VERTEX ← элемент вершины стека;
                    вытолкнуть STACK;
                    goto center;
                end;
         end
```

**Рис. 2.10.** Нерекурсивный алгоритм симметричного обхода

Корректность рекурсивного варианта нетрудно доказать индукцией по числу вершин в бинарном дереве. Корректность нерекурсивного варианта также можно доказать, но в этом случае предположение индукции не столь интуитивно понятно и возникают дополнительные трудности, связанные со стеком и правильным об-

ходом бинарного дерева. С другой стороны, платой за рекурсию может оказаться увеличение временной и пространственной сложностей.

Возникает естественный вопрос: как перевести рекурсивные алгоритмы в инструкции RAM? Согласно теореме 1.2, достаточно рассмотреть построение RASP-программы, так как такие программы можно моделировать с помощью машин RAM с замедлением не более чем в постоянное число раз. Рассмотрим довольно простой способ реализации рекурсии. Он пригоден для всех программ, о которых идет речь в этой книге, но не является универсальным.

В основе реализации рекурсивной процедуры лежит стек, где хранятся данные, участвующие во всех вызовах процедуры, при которых она еще не завершила свою работу. Иными словами, в стеке находятся все локальные данные. Стек разделен на фреймы, представляющие собой блоки последовательных ячеек (регистров). Каждый вызов процедуры использует фрейм стека, длина которого зависит от вызываемой процедуры.

Допустим, сейчас выполняется процедура *A*. Стек выглядит, как показано на рис. 2.11. Если процедура *A* вызывает процедуру *B*, происходит следующее.

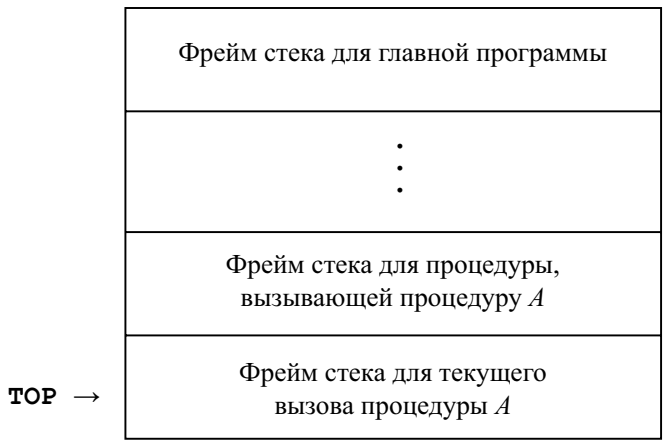



**Рис. 2.11.** Стек для рекурсивного вызова процедур

1. В вершину стека помещается фрейм стека нужного размера. В него входят следующие параметры, в порядке, известном процедуре *B*.
   - **а)** Указатели фактических параметров текущего вызова процедуры.[6]
   - **б)** Пустое место для локальных переменных, используемых в процедуры *B*.

[6]Если фактическим параметром является выражение, то его значение вычисляется во фрейме стека процедуры *A*, а указатель помещается во фрейм стека процедуры *B*. Если фактическим параметром является структура (например, массив), то достаточно передать указатель на ее первое слово.

**в)** Адрес инструкции RASP в процедуре *A*, которую следует выполнить после того, как данный вызов процедуры *B* закончит работу (*адрес возврата*).[7] Если *B* — функция, возвращающая некоторое значение, то во фрейм стека процедуры *B* также помещается указатель ячейки во фрейме стека для процедуры *A*, в которую следует поместить это значение (*адрес значения*).

**2.** Управление переходит к первой инструкции процедуры *B*. Адрес значения любого параметра или локального идентификатора, принадлежащего процедуре *B*, определяется с помощью индексации во фрейме стека процедуры *B*.

**3.** Когда процедура *B* заканчивает работу, управление передается процедуре *A* с помощью следующей последовательности шагов.

**а)** Из вершины стека извлекается адрес возврата.

**б)** Если *B* — функция, то значение, заданное выражением из оператора `return`, запоминается в ячейке, заданной адресом значения в стеке.

**в)** Фрейм стека процедуры *B* выталкивается из стека (в вершине стека помещается фрейм процедуры *A*).

**г)** Выполнение процедуры *A* возобновляется с ячейки, указанной в адресе возврата.

---

**Пример 2.5.** Рассмотрим процедуру `INORDER` из алгоритма 2.1. Когда она вызывает себя с фактическим параметром `LEFTSON[VERTEX]`, в ее стеке запоминается адрес нового значения параметра `VERTEX` вместе с адресом возврата, указывающим, что выполнение программы продолжается со строки 2. В результате во всех инструкциях процедуры, в которых встречается переменная `VERTEX`, она заменяется на `LEFTSON[VERTEX]`.

Описанную выше реализацию в определенном смысле моделирует нерекурсивный алгоритм 2.2. Однако там окончание выполнения вызова процедуры `INORDER` с фактическим параметром `RIGHTSON[VERTEX]` завершает выполнение самой вызывающей процедуры. По этой причине нам не обязательно хранить адрес возврата и переменную `VERTEX` в стеке, если фактическим параметром является `RIGHTSON[VERTEX]`. □

Время, требуемое для вызова процедуры, пропорционально времени, которое затрачивается на вычисление значений фактических параметров и запоминание указателей их значений в стеке. Время возврата, конечно, не превосходит этого времени.

При подсчете времени, затрачиваемого несколькими рекурсивными процедурами, обычно легче всего оценить затраты, связанные с вызовом процедуры, кото-

---

[7] Мы используем модель RAS именно из-за этих переходов к адресам возврата, которые в модели RAM доставляют неудобства (впрочем, вполне преодолимые).

рая выполняет этот вызов. Тогда разность между временем, затрачиваемым на вызов каждой процедуры, и временем, затрачиваемым теми процедурами, которые она вызывает, можно оценить сверху как функцию от размера входных данных. Суммируя эти оценки по всем вызовам процедур, получаем верхнюю границу общего затраченного времени.

Для подсчета времени работы рекурсивного алгоритма применяются рекуррентные уравнения. С $i$-й процедурой связывается функция $T_i(n)$, обозначающая время выполнения $i$-й процедуры как функцию некоторого параметра $n$ рассматриваемых входных данных. Обычно рекуррентное уравнение для $T_i(n)$ можно записать относительно времени выполнения процедур, вызываемых процедурой $i$. Затем решается полученная система рекуррентных уравнений. Часто в алгоритме используется только одна процедура, и $T(n)$ зависит лишь от значений $T(m)$ для конечного числа аргументов $m$, меньших $n$. В следующем разделе мы изучим решения некоторых часто встречающихся систем рекуррентных уравнений.

Напомним, что здесь, как в других разделах, весь анализ сложности основан на равномерной функции стоимости. Если брать логарифмическую функцию стоимости, то на анализе временной сложности может сказаться длина стека, используемого для реализации процедур с рекурсией.

## 2.6. Метод “Разделяй и властвуй”

Задачи часто разделяют на части, находят их решения и затем из них получают решение всей задачи. Этот подход, особенно в его рекурсивном варианте, часто приводит к эффективному решению задач, подзадачи которых представляют собой их меньшие версии. Проиллюстрируем этот метод на двух примерах и проанализируем получающиеся рекуррентные уравнения.

Рассмотрим задачу о нахождении наибольшего и наименьшего элементов множества $S$, содержащего $n$ элементов. Для простоты будем считать, что $n$ — это степень двойки. Очевидный путь поиска наибольшего и наименьшего элементов состоит в том, чтобы искать их по отдельности. Например, следующая процедура находит наибольший элемент множества $S$, выполнив $n$ 1 сравнений его элементов.

```
begin
    MAX ← произвольный элемент из S;
    for все другие элементы x из S do
        if x > MAX then MAX ← x
end
```

Аналогично можно найти наименьший из остальных $n - 1$ элементов, произведя $n - 2$ сравнений. В итоге для нахождения наибольшего и наименьшего элементов при $n \geq 2$ потребуется $2n - 3$ сравнений.

Применяя метод “разделяй и властвуй“, мы могли бы разбить множество $S$ на два подмножества — $S_1$ и $S_2$, каждое из которых содержит $n/2$ элементов. Тогда

описанный выше алгоритм нашел бы наибольший и наименьший элементы в каждой из двух половин с помощью рекурсии. Наибольший и наименьший элементы множества $S$ можно было бы определить, произведя еще два сравнения — наибольших и наименьших элементов в $S_1$ и $S_2$ соответственно. Сформулируем этот алгоритм более точно.

**Алгоритм 2.3.** Нахождение наибольшего и наименьшего элементов множества

*Вход*. Множество $S$ из $n$ элементов, где $n$ — степень двойки и $n \geq 2$.
*Выход*. Наибольший и наименьший элементы множества $S$.
*Метод*. К множеству $S$ применяется рекурсивная процедура `MAXMIN`. Она имеет один аргумент,[8] представляющий собой множество $S$, такое, что $||S|| = 2^k$ для некоторого $k \geq 1$, и возвращает пару $(a, b)$, где $a$ — наибольший и $b$ — наименьший элементы множества $S$. Процедура `MAXMIN` приведена на рис. 2.12. □

Заметим, что сравнения элементов множества $S$ происходят только на шаге 3, где сравниваются два элемента множества $S$, и на шаге 7, где сравниваются `max1` с `max2` и `min1` с `min2`. Пусть $T(n)$ — количество сравнений элементов множества $S$, которые надо выполнить в процедуре `MAXMIN`, чтобы найти наибольший и наименьший элементы множества, состоящего из $n$ элементов. Ясно, что $T(2) = 1$. Если $n > 2$, то $T(n)$ равно сумме общего количества сравнений, выполненных в двух вызовах процедуры `MAXMIN` (строки 5 и 6), работающей на множествах размера $n/2$, и двух сравнений в строке 7.

```
    procedure MAXMIN(S):

1.  if ||S|| = 2 then
        begin
2.          пусть S ={a, b};
3.          return (MAX(a, b), MIN(a, b))
        end
    else
        begin
4.          разбить S на два равных подмножества S1 и S2;
5.          (max1, min1) ← MAXMIN(S1);
6.          (max2, min2) ← MAXMIN(S2);
7.          return (MAX(max1, max2), MIN(min1 , min2))
        end
```

**Рис. 2.12.** Процедура для нахождения `MAX` и `MIN`

[8] Поскольку здесь подсчитываются только сравнения, способ перебора аргументов не имеет значения. Однако если множество $S$ представлено массивом, можно организовать эффективный вызов процедуры `MAXMIN`, установив указатели на первый и последний элементы подмножества `S`, состоящего из последовательных элементов этого массива.

Таким образом, при $n = 2$,

$$T(n) = \begin{cases} 1, & \text{при } n = 2; \\ 2T\dfrac{n}{2} + 2, & \text{при } n > 2. \end{cases} \tag{2.2}$$

Решением рекуррентных уравнений (2.2) является функция $T(n) = \dfrac{3}{2}n - 2$. Нетрудно убедиться, что эта функция удовлетворяет уравнению (2.2) при $n = 2$, и если она удовлетворяет (2.2) при $n = m$, где $m \geq 2$, то

$$T(2m) = 2\left(\frac{3m}{2} - 2\right) + 2 = \frac{3}{2}(2m) - 2,$$

т.е. она является решением уравнения (2.2) при $n = 2m$. Таким образом, индукцией по $n$ доказано, что $T(n) = \dfrac{3}{2}n - 2$ является решением уравнения (2.2), если $n$ — степень двойки.

Можно показать, что для одновременного нахождения наибольшего и наименьшего элементов множества, состоящего из $n$ элементов, необходимо выполнить не менее $\dfrac{3}{2}n - 2$ сравнений его элементов. Следовательно, алгоритм 2.3 оптимален в смысле количества сравнений элементов множества $S$, если $n$ — степень двойки.

В предыдущем примере метод "разделяй и властвуй" позволил уменьшить количество сравнений лишь в фиксированное число раз. В следующем примере с помощью этого метода мы уменьшим порядок роста сложности алгоритма.

Рассмотрим умножение двух $n$-разрядных двоичных чисел. Традиционный метод требует выполнить $O(n^2)$ битовых операций. В методе, изложенном ниже, достаточно выполнить порядка $n^{\log 3}$, т.е. примерно $n^{1,59}$ битовых операций.[9]

Пусть $x$ и $y$ — два $n$-разрядных двоичных числа. Снова предположим для простоты, что $n$ — степень двойки. Разобьем $x$ и $y$ на две равные части, как показано на рис. 2.13. Если рассматривать каждую из этих частей как $(n/2)$-разрядное число, то можно представить произведение чисел $x$ и $y$ в следующем виде:

$$\begin{aligned} xy &= (a2^{n/2} + b)(c2^{n/2} + d) = \\ &= ac2^n + (ad + bc)2^{n/2} + bd. \end{aligned} \tag{2.3}$$

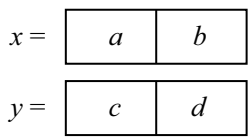



**Рис. 2.13.** Разбиение битовых строк

[9]Напомним, что по умолчанию все логарифмы в книге являются двоичными.

Равенство (2.3) позволяет вычислить произведение $x$ и $y$ с помощью четырех умножений ($n$/2)-разрядных чисел и нескольких сложений и сдвигов (умножений на степень двойки). Произведение $z$ чисел $x$ и $y$ также можно вычислить по следующей программе:

```
begin
    u ← (a + b) + (c + d);
    v ← a * c;
    w ← b * d;
    z ← v * 2^n + (u - v - w) * 2^(n/2) + w
end
```
(2.4)

На время забудем тот факт, что из-за переноса разряда числа $a + b$ и $c + d$ могут иметь $n/2 + 1$ разрядов, и предположим, что они состоят лишь из $n/2$ разрядов. Описанная выше схема для умножения двух $n$-разрядных чисел требует только трех умножений ($n$/2)-разрядных чисел и нескольких сложений и сдвигов. Для вычисления произведений чисел $u$, $v$ и $w$ эту программу можно применять рекурсивно. Сложения и сдвиги занимают $O_B(n)$ времени. Следовательно, временн*а*я сложность умножения двух $n$-разрядных чисел ограничена сверху функцией

$$T(n) = \begin{cases} k & \text{при } n = 1, \\ 3T(n/2) + kn & \text{при } n > 1, \end{cases} \tag{2.5}$$

где $k$ — константа, соответствующая операциям сложения и сдвига в выражениях, входящих в программу (2.4). Решение рекуррентных уравнений (2.5) ограничено сверху функцией

$$3kn^{\log 3} \approx 3kn^{1,59}.$$

На самом деле можно показать, что в формуле (2.5)

$$T(n) = 3kn^{\log 3} - 2kn.$$

Доказательство проведем индукцией по $n$, где $n$ — степень двойки. База рекурсии, т.е. случай $n = 1$, тривиальна. Если функция $T(n) = 3kn^{\log 3} - 2kn$ удовлетворяет равенству (2.5) при $n = m$, то по предположению индукции

$$\begin{aligned} T(2m) &= 3T(m) + 2km = \\ &= 3\left(3km^{\log 3} - 2km\right) + 2km = \\ &= 3k(2m)^{\log 3} - 2k(2m). \end{aligned}$$

Отсюда следует, что $T(n) \le 3kn^{\log 3}$. Заметим, что попытка использовать в индукции $3kn^{\log 3}$ вместо $3kn^{\log 3} - 2kn$ не проходит.

Для завершения алгоритма умножения мы должны учесть, что числа $a + b$ и $c + d$, вообще говоря, имеют $n/2 + 1$ разрядов, и поэтому произведение $(a + b) \times (c + d)$ нельзя вычислить непосредственным рекурсивным применением нашего алгоритма к задаче размера $n/2$. Вместо этого нужно записать $a + b$ в виде $a_1 2^{n/2} + b_1$, где $a_1$ — ведущий бит суммы $a + b$, а $b_1$ — остальные биты. Аналогично запишем $c + d$ в виде $c_1 2^{n/2} + d_1$. Тогда произведение $(a + b)(c + d)$ можно представить в виде

$$a_1 c_1 2^n + (a_1 d_1 + b_1 c_1) 2^{n/2} + b_1 d_1. \tag{2.6}$$

Слагаемое $b_1 d_1$ вычисляется с помощью рекурсивного применения нашего алгоритма умножения к задаче размера $n/2$. Остальные умножения в выражении (2.6) можно выполнить за время порядка $O_B(n)$, поскольку они содержат в качестве одного из аргументов либо единственный бит $a_1$ или $c_1$, либо степень двойки.

---

**Пример 2.6.** Этот асимптотически быстрый алгоритм умножения целых чисел можно применять не только к двоичным, но и к десятичным числам. Проиллюстрируем это следующими вычислениями:

$$\begin{array}{lrr} x = 3141 & a = 31 & c = 59 \\ y = 5927 & b = \underline{41} & d = \underline{27} \\ & a + b = 72 & c + d = 86 \end{array}$$

$$\begin{aligned} u &= (a+b)(c+d) = 72 \cdot 86 = 6192, \\ v &= ac = 31 \cdot 59 = 1829, \\ w &= bd = 41 \cdot 27 = 1107, \\ xy &= 18\ 290\ 000 + (6192 - 1829 - 1107) \cdot 100 + 1107 = \\ &= 18\ 616\ 707. \quad \square \end{aligned}$$

Заметим, что в алгоритме, основанном на программе (2.4), одно умножение было заменено тремя сложениями и вычитанием (по сравнению с (2.3)). Асимптотическая эффективность, такого алгоритма интуитивно объясняется тем, что умножение выполнить труднее, чем сложение, и для достаточно больших $n$ любое фиксированное количество $n$-разрядных сложений требует меньше времени, чем $n$-разрядное умножение, независимо от того, какой из (известных) алгоритмов используется. На первый взгляд, уменьшение числа $(n/2)$-разрядных произведений с четырех до трех может уменьшить общее время в лучшем случае на 25%. Однако соотношения (2.4) применяются рекурсивно для вычисления $(n/2)$-разрядных, $(n/4)$-разрядных и т.д. произведений. Эти 25-процентные уменьшения накапливаются и дают в результате асимптотическое снижение временно*й* сложности.

Временн*ая* сложность процедуры определяется количеством и размером подзадач и в меньшей степени операциями, необходимыми для разбиения данной за-

дачи на подзадачи. Поскольку рекуррентные уравнения вида (2.2) и (2.5) часто возникают при анализе рекурсивных алгоритмов типа “разделяй и властвуй”, рассмотрим решение таких уравнений в общем виде.

---

**Теорема 2.1.** Пусть $a$, $b$ и $c$ — неотрицательные константы. Решение рекуррентных уравнений

$$T(n)=\begin{cases} b & \text{при } n=1,\\ aT(n/c)+bn & \text{при } n>1, \end{cases}$$

где $n$ — степень $c$, имеет вид

$$T(n)=\begin{cases} O(n), & \text{если } a<c,\\ O(n\log n), & \text{если } a=c,\\ O\left(n^{\log_c a}\right), & \text{если } a>c. \end{cases}$$

*Доказательство.* Если $n$ — степень числа $c$, то

$$T(n)=bn\sum_{i=0}^{\log_c n} r^i, \text{ где } r=a/c.$$

Если $a < c$, то ряд $\sum_{i=0}^{\infty} r^i$ сходится и $T(n)$ имеет порядок $O(n)$. Если $a = c$, то каждый член суммы равен единице, а всего в ней $O(\log n)$ членов. Поэтому $T(n)=O(n\log n)$. Наконец, если $a > c$, то

$$bn\sum_{i=0}^{\log_c n} r^i = bn\frac{r^{1+\log_c n}-1}{r-1}.$$

Это выражение имеет порядок $O\left(a^{\log_c n}\right)$, что эквивалентно $O\left(n^{\log_c a}\right)$. □

Из теоремы 2.1 вытекает, что разбиение задачи размера $n$ (за линейное время) на две подзадачи размера $n/2$ позволяет создать алгоритм, имеющий сложность $O(n\log n)$. Если бы подзадач было 3, 4 или 8, то получился бы алгоритм сложности порядка $n^{\log 3}$, $n^2$ и $n^3$ соответственно. С другой стороны, разбиение задачи на четыре подзадачи размера $n/4$ приводит к алгоритму, имеющему сложность порядка $O(n\log n)$, а разбиение на 9 и 16 подзадач приводит к алгоритму, имеющему порядок сложности $n^{\log 3}$ и $n^2$ соответственно. Следовательно, асимптотически более быстрый алгоритм умножения целых чисел можно было бы получить, если бы удалось так разбить исходные целые числа на четыре части, чтобы суметь выразить исходное умножение через восемь или меньшее количество умножений чисел меньшей длины. Другой тип рекуррентных соотношений возникает в слу-

чае, когда сложность разбиения задачи не пропорциональна ее размеру. Некоторые типы рекуррентных соотношений описаны в упражнениях.

Если $n$ не является степенью числа $c$, то обычно можно вложить задачу размера $n$ в задачу размера $n'$, где $n'$ — наименьшая степень числа $c$, бо́льшая или равная $n$. Таким образом, асимптотические порядки роста, указанные в теореме 2.1, сохраняются для любого $n$. На практике часто можно разработать рекурсивные алгоритмы, разбивающие задачи произвольного размера на $c$ равных частей, где $c$ велико, насколько возможно. Эти алгоритмы, как правило, эффективнее (на постоянный множитель) тех, которые получаются путем представления размера входа в виде ближайшей сверху степени числа $c$.

## 2.7. Балансировка

В обоих наших примерах применения метода "разделяй и властвуй" задача разбивалась на подзадачи одинаковых размеров. Это не было случайностью. Поддержание равновесия — основной принцип разработки хорошего алгоритма. Для иллюстрации этого принципа рассмотрим пример сортировки и сравним эффекты от разбиения задачи на подзадачи одинаковых и неодинаковых размеров. Из этого примера не следует, что метод "разделяй и властвуй" — единственный, в котором полезна балансировка. В главе 4 приводится несколько примеров, в которых балансировки размеров поддеревьев или весов двух операций приводят к эффективным алгоритмам.

Рассмотрим задачу упорядочения целых чисел в порядке неубывания. По-видимому, проще всего было бы найти наименьший элемент, просмотрев всю последовательность и поменяв местами наименьший элемент с первым. Этот процесс применяется к оставшимся $n - 1$ элементам, и в результате наименьший элемент оказывается на втором месте. Применение этого процесса к остальным $n - 2$, $n - 3$, ..., 2 элементам упорядочивает всю последовательность в порядке неубывания.

Этот алгоритм приводит к рекуррентным уравнениям

$$T(n) = \begin{cases} 0, & \text{если } n = 1, \\ T(n-1) + n - 1, & \text{если } n > 1, \end{cases} \tag{2.7}$$

которые задают количество сравнений упорядочиваемых элементов. Решением уравнений (2.7) является функция $T(n) = n(n-1)/2$, имеющая порядок $O(n^2)$.

Этот алгоритм можно считать рекурсивным применением метода "разделяй и властвуй" с разбиением задачи на неравные части, но при больших $n$ он не эффективен. Для разработки асимптотически эффективного алгоритма сортировки необходимо обеспечить балансировку. Вместо того чтобы разбивать задачу размера $n$ на две подзадачи, одна из которых имеет размер 1, а другая — $n - 1$, необхо-

димо разбить ее на две подзадачи с размерами примерно по $n/2$. Эту задачу решает *сортировка слиянием*.

Рассмотрим последовательность целых чисел $x_1, x_2, ..., x_n$. Снова предположим для простоты, что $n$ — степень двойки. Один из способов упорядочить эту последовательность — разделить ее на две подпоследовательности, $x_1, x_2, ..., x_{n/2}$ и $x_{n/2+1}, ..., x_n$, упорядочить каждую из них, а затем выполнить их слияние. Под слиянием здесь понимается объединение двух уже упорядоченных последовательностей в одну упорядоченную последовательность.

**Алгоритм 2.4.** Сортировка слиянием

*Вход*. Последовательность чисел $x_1, x_2, ..., x_n$, где $n$ — степень двойки.

*Выход*. Последовательность $y_1, y_2, ..., y_n$, являющаяся перестановкой входной последовательности и удовлетворяющая неравенствам $y_1 \le y_2 \le ... \le y_n$.

*Метод*. Применим процедуру `MERGE(S, T)`, входом которой служат две упорядоченные последовательности $S$ и $T$, а выходом — последовательность элементов из $S$ и $T$, упорядоченных в порядке неубывания. Поскольку последовательности $S$ и $T$ упорядочены, процедура `MERGE` требует не больше сравнений, чем сумма длин $S$ и $T$ без единицы. Эта повторяющаяся процедура выбирает больший из наибольших элементов, остающихся в последовательностях $S$ и $T$, а затем удаляет этот элемент. Если наибольшие элементы в обеих последовательностях совпадают, то предпочтение отдается последовательности $S$.

Кроме того, в этом алгоритме применяется процедура `SORT(i, j)` (рис. 2.14), сортирующая подпоследовательность $x_1, x_{i+1}, ..., x_j$ в предположении, что она имеет длину $2^k$ при некотором $k \ge 0$.

Для сортировки последовательности $x_1, x_2, ..., x_n$ выполняется вызов процедуры `SORT(1, n)`. □

```
procedure SORT(i, j):
if i = j then return x_i
else
    begin
       m ← (i + j - 1)/2;
       return MERGESORT(SORT(i, m), SORT(m + 1, j))
    end
```

**Рис. 2.14.** Сортировка слиянием

Подсчет количества сравнений в алгоритме 2.4 приводит к рекуррентным уравнениям

$$T(n) = \begin{cases} 0, & \text{если } n = 1, \\ T(n-1) + n - 1, & \text{если } n > 1, \end{cases}$$

решением которых по теореме 2.1 является функция $T(n) = O(n \log n)$. При больших $n$ сбалансированность размеров подзадач обеспечивает значительное преимущество. Аналогичный анализ показывает, что общее время работы процедуры `SORT`, помимо операций сравнения, также имеет порядок $O(n \log n)$.

## 2.8. Динамическое программирование

Рекурсия полезна, если разбиение задачи на подзадачи не требует значительных усилий, а суммарный размер подзадач невелик. Из теоремы 2.1 следует, что если сумма размеров подзадач равна $an$ для некоторой константы $a > 1$, то рекурсивный алгоритм, скорее всего, имеет полиномиальную временную сложность. Однако если очевидное разбиение задачи размера $n$ сводит ее к $n$ задачам размера $n - 1$, то рекурсивный алгоритм, скорее всего, имеет экспоненциальную сложность. В этом случае более эффективные алгоритмы часто можно получить с помощью табличного метода, который называется *динамическим программированием*.

По существу, динамическое программирование вычисляет решение всех подзадач. Вычисление идет от меньших подзадач к бо́льшим, и ответы запоминаются в таблице. Преимущество этого метода состоит в том, что решения подзадач сохраняются и никогда не вычисляются заново. Этот метод легко понять на простом примере.

Рассмотрим вычисление произведения $n$ матриц

$$M = M_1 \times M_2 \times \ldots \times M_n,$$

где $M_i$ — матрица с $r_{i-1}$ строками и $r_i$ столбцами. Порядок перемножения этих матриц может существенно повлиять на общее количество операций, необходимых для вычисления матрицы $M$, независимо от алгоритма умножения матриц.

---

**Пример 2.7.** Предположим, что умножение матрицы размерностью $p \times q$ на матрицу размерностью $q \times r$ требует $pqr$ операций, как в обычном алгоритме. Рассмотрим произведение

$$M = \underset{[10\times 20]}{M_1} \times \underset{[20\times 50]}{M_2} \times \underset{[50\times 1]}{M_3} \times \underset{[1\times 100]}{M_4},$$

где размеры каждой матрицы $M_i$ указаны в скобках. Если вычислять матрицу $M$ в порядке

$$M_1 \times (M_2 \times (M_3 \times M_4)),$$

то потребуется 125000 операций, тогда как для вычисления $M$ в порядке

$$(M_1 \times (M_2 \times M_3)) \times M_4,$$

понадобятся лишь 2200 операций. □

Процесс перебора всех вариантов порядка вычисления рассматриваемого произведения $n$ матриц с целью минимизировать количество операций имеет экспоненциальную сложность (см. упражнение 2.31), что при больших $n$ непрактично. Однако динамическое программирование приводит к алгоритму, имеющему сложность $O(n^3)$. Пусть $m_{ij}$ — минимальная сложность вычисления $M_i \times M_{i+1} \times ... \times M_j$ при $1 \le i \le j \le n$. Очевидно, что

$$m_{ij} = \begin{cases} 0, & \text{если } i = j, \\ \underset{i \le k \le j}{\mathtt{MIN}} \left( m_{ik} + m_{k+1,j} + r_{i-1} r_k r_j \right), & \text{если } j > i, \end{cases} \tag{2.9}$$

где $m_{ik}$ — минимальная сложность вычисления произведения матриц $M' = M_i \times \times M_{i+1} \times ... \times M_k$, второй член, $m_{k+1,j}$, представляет собой минимальную сложность вычисления матрицы

$$M'' = M_{k+1} \times M_{k+2} \times ... \times M_j,$$

а третье слагаемое — это сложность перемножения матриц $M'$ и $M''$. Обратите внимание на то, что $M'$ — матрица размерностью $r_{i-1} \times r_k$, а $M''$ — матрица размерностью $r_k \times r_j$. Формула (2.9) утверждает, что $m_{ij}$, $(j > i)$ — наименьшая из сумм этих трех членов по всем возможным значениям $k$, лежащим между $i$ и $j - 1$.

При динамическом программировании величины $m_{ij}$ вычисляются в порядке возрастания разностей нижних индексов. Вычисления начинаются с $m_{ii}$ для всех $i$, затем $m_{i,i+1}$ для всех $i$, затем $m_{i,i+2}$ и т.д. Таким образом, когда мы приступим к вычислению $m_{ij}$, члены $m_{ik}$ и $m_{k+1,j}$ в формуле (2.9) будут уже вычислены. Это следует из того, что при $i \le k < j$ разность $j - i$ должна быть больше $k - i$, а также и $j - (k + 1)$. Соответствующий алгоритм приведен ниже.

---

**Алгоритм 2.5.** Алгоритм динамического программирования для вычисления порядка, минимизирующего сложность перемножения $n$ матриц $M = M_1 \times M_2 \times ... \times M_n$.

*Вход*. $r_0, r_1, ..., r_n$, где $r_{i-1}$ и $r_i$ — размерности матрицы $M_i$.

*Выход*. Минимальная сложность умножения матриц $M_i$ при условии, что для умножения матрицы размерностью $p \times q$ на матрицу размерностью $q \times r$ требуется $pqr$ операций.

*Метод*. Алгоритм, приведенный на рис. 2.15. □

```
    begin
1.      for i ← 1 until n do m_ii ← 0:
2.      for l ← 1 until n - 1 do
3.          for i ← 1 until n - l do
                begin
4.                  j ← i + l;
5.                  m_ij ← MIN_{i≤k<j} (m_ik + m_{k+1,j} + r_{i-1} * r_k * r_j)
                end;
6.      write m_1n
    end
```

**Рис. 2.15.** Алгоритм динамического программирования для вычисления порядка при перемножении матриц

**Пример 2.8.** Если применить этот алгоритм к цепочке из четырех матриц (2.8), где $r_0$, ..., $r_4$ равны соответственно 10, 20, 50, 1, 100, то в результате вычислений получатся значения $m_{ij}$, приведенные на рис. 2.16. Таким образом, минимальное количество операций, необходимых для вычисления этого произведения, равно 2200. Порядок, в котором можно выполнить эти умножения, легко определить, приписав каждой ячейке таблицы то значение $k$, на котором достигается минимум в формуле (2.9). □

| | | | |
|---|---|---|---|
| $m_{11} = 0$ | $m_{22} = 0$ | $m_{33} = 0$ | $m_{44} = 0$ |
| $m_{12} = 10{,}000$ | $m_{23} = 1000$ | $m_{34} = 5000$ | |
| $m_{13} = 1200$ | $m_{24} = 3000$ | | |
| $m_{14} = 2200$ | | | |

**Рис. 2.16.** Сложность вычисления произведений $M_i \times M_{i+1} \times ... \times M_j$

## 2.9. Заключение

В этой главе рассмотрен ряд основных методов, которые используются при разработке эффективных алгоритмов. Мы увидели, как структуры высокого уровня, такие как списки, очереди и стеки, избавляют разработчиков алгоритмов от утомительной работы, например, связанной с указателями, позволяя сосредоточиться на общей структуре самого алгоритма. Кроме того, было показано, как мощные методы рекурсии и динамического программирования часто приводят к элегантным и естественным алгоритмам. Мы также ознакомились с некоторыми общими методами, такими как “разделяй и властвуй“ и балансировка.

Конечно, эти методы — не единственно доступные, но являются одними из важнейших. Далее в этой книге мы рассмотрим и другие методы. Они будут относиться к различным задачам — от выбора подходящего представления до определения разумного порядка выполнения операций. Возможно, важнейшим качеством

хорошего разработчика алгоритмов должно быть постоянное чувство неудовлетворенности. Разработчик должен исследовать задачу с разных точек зрения, пока не убедится, что создал алгоритм, наиболее подходящий для его целей.

## Упражнения

**2.1.** Выберите реализацию для дважды связанного списка. Напишите алгоритм на псевдо-Алголе для вставки и удаления элемента. Убедитесь в работоспособности своей программы на примерах, когда требуется удалить первый и (или) последний элементы и когда список пуст.

**2.2.** Напишите алгоритм для изменения порядка следования элементов в списке на противоположный. Докажите, что он работает правильно.

**2.3.** Напишите алгоритмы, реализующие операции `PUSH`, `POP`, `ENQUEUE` и `DEQUEUE`, упомянутые в разделе 2.1. Не забывайте проверять, достиг ли указатель конца массива, зарезервированного для стека или очереди.

**2.4.** Напишите условия для проверки пустоты очереди. Допустим, массив `NAME` из раздела 2.1 имеет размер $k$. Сколько элементов можно хранить в соответствующей очереди? Создайте рисунки, иллюстрирующие эту очередь и типичные положения указателей `FRONT` и `REAR` в случае, когда очередь (*а*) пустая, (*б*) содержит один элемент и (*в*) полная.

**2.5.** Напишите алгоритм для удаления первого ребра $(v, w)$ из списка смежности для вершины $v$ в неориентированном графе. Считайте, что списки смежности дважды связаны и что массив `LINK` помещает вершину $v$ в список смежности вершины $w$, как описано в разделе 2.3.

**2.6.** Напишите алгоритм для построения списков смежности для неориентированного графа. Каждое ребро $(v, w)$ необходимо представить дважды: в списках смежности для $v$ и для $w$. Оба экземпляра каждого ребра должны связываться между собой так, что если один из них удаляется, то другой также можно легко удалить.

***2.7.** (*Топологическая сортировка*). Пусть $G = (V, E)$ — ориентированный ациклический граф. Напишите алгоритм, присваивающий целые числа вершинам графа $G$ по следующему правилу: если из вершины с номером $i$ в вершину с номером $j$ идет ориентированное ребро, то $i < j$. (*Подсказка*: ациклический граф должен иметь вершину, в которую не входит ни одно ребро. Почему? Одно из решений этой задачи можно получить так: находим вершину, в которую не входят ребра, приписываем ей наименьший номер и удаляем из графа вместе со всеми ребрами, выходящими из нее. Повторяем этот процесс для графа, полученного в результате такого удаления, присваиваем

следующий наименьший номер и т.д. Для того чтобы обеспечить эффективность этого алгоритма, т.е. чтобы он имел сложность $O(||E|| + ||V||)$, необходимо, чтобы в поисках вершины, в которую не входит ни одно ребро, не приходилось просматривать каждый новый граф.

***2.8.** Пусть $G = (V, E)$ — ориентированный ациклический граф с двумя выделенными вершинами — начальной и целевой. Напишите алгоритм для нахождения такого множества путей из начальной вершины в целевую, чтобы

**а)** ни одна вершина, кроме начальной и целевой, не лежала на двух путях;

**б)** в этом множество нельзя было добавить ни одного нового пути, не нарушив условия 1.

Заметим, что этим условиям удовлетворяет много множеств. Не обязательно находить множество с наибольшим числом путей, достаточно какого-нибудь множества, удовлетворяющего приведенным выше условиям. Ваш алгоритм должен иметь временную сложность $O(||E|| + ||V||)$.

***2.9.** (*Задача об устойчивом паросочетании*). Пусть $B$ — множество из $n$ юношей, а $G$ — множество из $n$ девушек. Каждый юноша оценивает девушек числами от 1 до $n$, и каждая девушка оценивает юношей числами от 1 до $n$. *Паросочетанием* называется взаимно однозначное соответствие между юношами и девушками. Паросочетание устойчиво, если для любых двух юношей $b_1$ и $b_2$ и соответствующих им в этом паросочетании девушек $g_1$ и $g_2$ выполняются следующие два условия:

**а)** либо $b_1$ оценивает $g_1$ выше, чем $g_2$, либо $g_2$ оценивает $b_2$ выше, чем $b_1$;

**б)** либо $b_2$ оценивает $g_2$ выше, чем $g_1$, либо $g_1$ оценивает $b_1$ выше, чем $b_2$.

Докажите, что устойчивое паросочетание всегда существует. Напишите алгоритм для нахождения одного из таких паросочетаний.

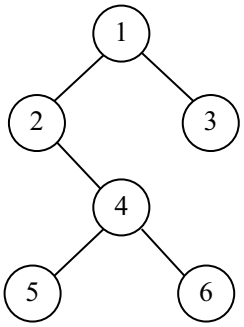
**2.10.** Рассмотрим бинарное дерево, вершинам которого приписаны имена. Напишите алгоритм, печатающий эти имена в (*а*) прямом порядке, (*б*) обратном и (*в*) симметричном.

***2.11.** Напишите алгоритм для вычисления значений арифметических выражений, содержащих операции + и × и представленных в (*а*) префиксной польской записи, (*б*) инфиксной записи и (*в*) постфиксной польской записи.

***2.12.** Разработайте метод, который позволял бы приравнивать к нулю те элементы матрицы, которые выбираются первый раз: таким способом устраняется работа по инициализации исходной записи матрицы смежности, занимающая время $O(||V||^2)$. (*Подсказка*: предусмотрите в каждом инициализируемом элементе матрицы указатель на обратный указатель в стеке. Всякий раз, когда выбирается какой-то элемент, проверяйте, что он имеет неслу-

чайное значение, т.е. что его указатель указывает на активную часть стека, а соответствующий обратный указатель — на выбираемый элемент.)

**2.13.** Рассмотрите работу алгоритмов 2.1 и 2.2 на бинарном дереве рис. 2.17.




| | LEFTSON | RIGHTSON |
|---|---|---|
| 1 | 2 | 3 |
| 2 | 0 | 4 |
| 3 | 0 | 0 |
| 4 | 5 | 6 |
| 5 | 0 | 0 |
| 6 | 0 | 0 |

**Рис. 2.17.** Бинарное дерево

***2.14.** Докажите, что алгоритм 2.2 корректен.

***2.15.** (*Ханойские башни*). Имеются три стержня, *A*, *B* и *C*, и *n* дисков разных размеров. Вначале все диски нанизаны на стержень *A* в порядке убывания размеров, так что наибольший диск находится внизу. Цель задачи — переместить диски со стержня *A* на стержень *B*, чтобы никакой больший диск ни разу не оказался над меньшим диском. За один шаг можно перемещать только один диск. Стержень *C* можно использовать для временного хранения дисков. Напишите рекурсивный алгоритм решения этой задачи. Каково время работы вашего алгоритма в терминах числа перемещений дисков?

****2.16.** Решите упражнение 2.15 с помощью алгоритма без рекурсии. Какой из алгоритмов легче понять и корректность какого из них проще доказать?

****2.17.** Докажите, что для решения упражнения 2.15 необходимо и достаточно $2^{n-1}$ перемещений.

**2.18.** Напишите алгоритм генерирования всех перестановок целых чисел от 1 до *n*. (*Подсказка*: множество перестановок целых чисел от 1 до *n* можно получить из множества перестановок целых чисел от 1 до $n - 1$, вставляя *n* во все возможные позиции в каждой перестановке.)

**2.19.** Напишите алгоритм нахождения высоты бинарного дерева, представленного на рис. 2.7, *б*.

**2.20.** Напишите алгоритм подсчета числа потомков каждой вершины дерева.

****2.21.** Рассмотрим двухпозиционный переключатель с двумя входами и двумя выходами, показанный на рис. 2.18. В одной позиции входы 1 и 2 соединяются

соответственно с выходами 1 и 2, в другой — с выходами 2 и 1. Используя эти переключатели, создайте сеть с $n$ входами и $n$ выходами, на которой можно получить любую из возможных перестановок входов. В этой сети должно быть не более $O(n \log n)$ переключателей.

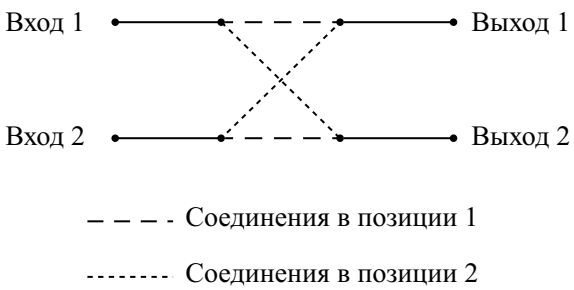



**Рис. 2.18.** Двухпозиционный переключатель

**2.22.** Напишите программу RASP, которая выполняла бы следующую процедуру вычисления программы, вычисляющей $\binom{n}{m}$ — число сочетаний по $m$ из $n$:

```
procedure COMB(n, m):
if m = 0 or n = m then return 1
else return (COMB(n - 1, m) + COMB(n - 1, m - 1)
```

Текущие значения $n$ и $m$, а также адреса возвращаемого значения и значений, передаваемых при вызове, можно хранить в стеке.

***2.23.** Иногда задачу размера $n$ удобно разбить на $\sqrt{n}$ подзадач размера $\sqrt{n}$. В результате получается рекуррентное уравнение вида

$$T\left(\frac{n^2}{2^r}\right) = nT(n) + bn^2,$$

где $r$ — целое число, $r \geq 1$. Покажите, что решение этого рекуррентного уравнения имеет порядок $O\left(n(\log n)^r \log\log n\right)$.

**2.24.** Вычислите следующие суммы:

**а)** $\sum_{i=1}^{n} i$; **б)** $\sum_{i=1}^{n} a^i$; **в)** $\sum_{i=1}^{n} i a^i$;

**г)** $\sum_{i=1}^{k} 2^{k-i} i^2$; **д)** $\sum_{i=1}^{n} \binom{n}{i}$; **е)** $\sum_{i=1}^{n} i \binom{n}{i}$.

***2.25.** Решите следующие рекуррентные уравнения при $T(1) = 1$:

**а)** $T(n) = aT(n - 1) + bn$,

**б)** $T(n) = aT(n/2) + bn \log n$,

**в)** $T(n) = aT(n - 1) + bn^c$,

**г)** $T(n) = aT(n/2) + bn^c$.

***2.26.** Модифицируйте алгоритм 2.3 для нахождения наибольшего и наименьшего элементов множества, разрешив рекурсию до уровня $\|S\| = 1$. Какова асимптотическая скорость роста числа сравнений?

***2.27.** Покажите, что для одновременного нахождения наибольшего и наименьшего элементов $n$-элементного множества необходимо и достаточно $\left\lceil \frac{3}{2}n - 2 \right\rceil$ сравнений.

***2.28.** Модифицируйте алгоритм для умножения целых чисел с помощью разбиения каждого целого числа на (*а*) три и (*б*) четыре части. Какова сложность каждого из этих алгоритмов?

***2.29.** Пусть $A$ — массив размера $n$, состоящий из положительных и отрицательных целых чисел, причем $A[1] < A[2] < ... < A[n]$. Напишите алгоритм нахождения числа $i$, для которого $A[i] = i$ (если такое число существует). Какова временн*ая* сложность вашего алгоритма? Докажите, что $O_C(\log n)$ — наилучший возможный порядок.

****2.30.** Для числа $n$, не являющегося степенью двойки, из алгоритма 2.4 можно получить корректный алгоритм сортировки слиянием, если заменить оператор $m \leftarrow (i + j - 1)/2$ на рис. 2.14 оператором $m \leftarrow \lfloor (i + j)/2 \rfloor$. Пусть $T(n)$ — число сравнений, необходимых для сортировки $n$ элементов этим методом.

**а)** Покажите, что

$$T(1) = 0,$$
$$T(n) = T\lfloor n/2 \rfloor + T\lceil n/2 \rceil + n - 1.$$

**б)** Покажите, что решением этого рекуррентного уравнения является функция

$$T(n) = n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1.$$

***2.31.** Покажите, что решением рекуррентного уравнения

$$X(1)=1,$$
$$X(n)=\sum_{i=1}^{n-1}X(i)X(n-i),\ \text{при } n>1$$

является функция

$$X(n+1)=\frac{1}{n+1}\binom{2n}{n}.$$

где $X(n)$— это число способов правильной расстановки скобок в цепочке из $n$ символов. Числа $X(n)$ называются числами Каталана. Покажите, что $X(n)\geq 2^{n-2}$.

**2.32.** Модифицируйте алгоритм 2.5 так, чтобы он выдавал порядок перемножения матриц, при котором количество умножений их элементов становится минимальным.

**2.33.** Напишите эффективный алгоритм для определения порядка, в котором следует вычислять произведение матриц $M_1 \times M_1 \times ... \times M_n$, чтобы минимизировать число умножений элементов в случае, когда каждая матрица $M_i$ имеет один из размеров $1 \times 1$, $1 \times d$, $d \times 1$ или $d \times d$ при некотором фиксированном $d$.

---

***Определение.*** *Контекстно-свободной грамматикой* $G$ в нормальной форме Хомского называется четверка $(N, \Sigma, P, S)$, где (1) $N$ — конечное множество *нетерминальных символов*; (2) $\Sigma$ — конечное множество *терминальных символов*; (3) $P$ — конечное множество пар, называемых *правилами вывода* и имеющими вид $A \to BC$ или $A \to a$ ($A$, $B$, $C$ — символы из $N$ и $a$ — символ из множества $\Sigma$); и (4) $S$ — выделенный символ из $N$. Мы будем писать $\alpha A\gamma \Rightarrow \alpha\beta\gamma$, если $\alpha$, $\beta$, $\gamma$ — строки, состоящие из терминальных и нетерминальных символов, и пара $A \to \beta$ принадлежит $P$. *Языком* $L(G)$, *порождаемым грамматикой* $G$, называется множество строк терминальных символов $\left\{w \middle| S \overset{*}{\Rightarrow} w\right\}$, где $\overset{*}{\Rightarrow}$ обозначает рефлексивное и транзитивное замыкание отношения $\Rightarrow$.

---

***2.34.** Напишите алгоритм сложности $O(n^3)$, который определял бы принадлежность данной строки $w = a_1a_2...a_n$ языку $L(G)$, где $G = (N, \Sigma, P, S)$ — контекстно-свободная грамматика в нормальной форме Хомского. (*Подсказка*. Пусть $m_{ij} = \left\{A \middle| A \in N \text{ и } A \overset{*}{\Rightarrow} a_i a_{i+1} ... a_j\right\}$. Слово $w$ принадлежит языку $L(G)$ тогда и только тогда, когда $S \in m_{1n}$. Для вычисления $m_{ij}$ примените динамическое программирование.)

***2.35.** Пусть $x$ и $y$ — строки символов из некоторого алфавита. Рассмотрите операции удаления символа из строки $x$, вставки символа в строку $x$ и замены символа в строке $x$ другим символом. Опишите алгоритм нахождения минимального числа таких операций, необходимых для преобразования строки $x$ в строку $y$.

## Библиографические замечания

Дополнительную информацию о структурах данных и их реализации можно найти в книгах Knuth [1968] и Stone [1972]. Книга Pratt [1975] содержит описание реализации рекурсии в языках, подобных Алголу. Теория графов изложена в книгах Berge [1958] и Harary [1969]. Knuth [1968] приводит информацию о деревьях и алгоритмах их обхода. Дальнейшие сведения об алгоритмах обхода деревьев можно найти у Burkhard [1973].

Оптимальность алгоритма 2.3 (для нахождения максимума и минимума) показал Pohl [1072]. Алгоритм сложности $O(n^{1,59})$ для целых чисел (см. раздел 2.6) приведен в работе Karatsuba and Ofman [1962]. Winograd [1973] рассматривает аналогичные методы ускорения с более общей точки зрения.

Понятие динамического программирования развивал Bellman [1957]. Алгоритм 2.5 представляет собой известное приложение этого понятия, опубликованное Godbole [1973], а также Muraoka и Kuck [1973]. Приложением динамического программирования к распознаванию принадлежности строки к контекстно-свободному языку (см. упражнение 2.34) независимо друг от друга занимались J. Cocke and Kasami [1965], а также Younger [1967]. Решение упражнения 2.35 приведено в работе Wagner and Fisher [1974].

Дальнейшую информацию о решении рекуррентных уравнений см. в книгах Liu [1968] или Sloane [1973].