Оглавление

Предисловие

Начнем с предисловия к книге алгоритмов практической арифметики.

— Иоанн Севильский (Ioannis Hispalensis),
Книга алгоритмов практической арифметики (около 1135 г.)

Должен ли я объяснять тебе, друг мой, как ты сможешь это понять?
Напиши об этом книгу.

— Генри Хоум, лорд Камес (Henry Home, Lord Kames) (1696–1782),
в письме сэру Гилберту Элиотту (Gilbert Eliott)

Человек всегда ошибается. Он строил множество планов и считал других людей своими помощниками, спорил с некоторыми или со всеми, много ошибался и кое-что сделал;
все это позволило продвинуться немного вперед, но человек всегда ошибается.
Оказалось, что получилось нечто новое, совсем не похожее на то, что он себе представлял.

— Ральф Уолдо Эмерсон (Ralph Waldo Emerson), Experience, эссе, второй сборник (1844 г.)

То, что я кратко описал выше, является содержанием книги, в которой реализация основной темы с включением подробностей, вероятно, стала бы невозможной; то, что я написал, является вторым или третьим черновым вариантом предварительной версии этой книги.

— Майкл Спивак (Michael Spivak), предисловие к первому изданию
книги «Дифференциальная геометрия», том 1 (1970 г.)

Об этой книге

Основой этой книги стали конспекты лекций, написанных мною для разнообразных курсов по алгоритмам в университете Иллинойса в Урбана-Шампейне (University of Illinois at Urbana-Champaign), где я преподаю примерно раз в год с января 1999 г. Из-за изменений в теоретической программе бакалавриата я существенно переработал основную редакцию этих конспектов лекций в 2016 г. Эта книга состоит из некоторого подмножества переработанных конспектов лекций в основном по материалам основного базового курса и главным образом отображает алгоритмическое содержимое новой университетской теоретической программы, обязательной для младших курсов.

Обязательный минимум

Для тех курсов по алгоритмам, которые я преподаю в университете Иллинойса, существуют два весьма важных предварительных требования к

обязательному образовательному минимуму: изучение курса дискретной математики и курса основных структур данных. Таким образом, эта книга, вероятнее всего, не подойдет для большинства студентов как начальный курс по структурам данных и алгоритмам. В частности, я предполагаю, что читатель как минимум хорошо знаком со специальными темами, такими как:

- **дискретная математика:** элементарная алгебра, логарифмические тождества, простейшая теория множеств, алгебра логики, логика первого порядка, множества, функции, эквивалентности (тождества), частичная упорядоченность, модульная арифметика (операции с остатками чисел по модулю), рекурсивные определения, деревья (как абстрактные объекты, а не структуры данных), графы (из вершин и ребер, а не графики функций);
- **методы доказательств:** прямой, косвенный, от противного (контрадикция), исчерпывающий (полный) анализ вариантов и индукция (особенно «строгая» и «структурная» индукция). В главе 0 используется индукция, и во всех случаях, когда в главе n–1 используется индукция, она используется и в главе n;
- **итеративные концепции программирования:** переменные, условные выражения, циклы, записи, косвенная адресация (адреса, указатели, ссылки), подпрограммы, рекурсия. Я не требую свободного владения каким-либо конкретным языком программирования, но предполагаю наличие у читателя практического опыта работы хотя бы с одним языком, который поддерживает косвенную адресацию и рекурсию;
- **основные (базовые) абстрактные типы данных:** скалярные значения, последовательности, векторы, множества, стеки, очереди, отображения/словари, упорядоченные отображения/словари, очереди с приоритетами;
- **основные (базовые) структуры данных:** массивы, связные списки (односвязные и двусвязные, линейные и кольцевые (циклические)), деревья двоичного поиска, как минимум одна форма сбалансированного дерева двоичного поиска (например, AVL-деревья, красно-черные деревья, декартовы деревья («дуча»), списки с пропусками или косые деревья), хеш-таблицы, двоичные кучи, а также, что наиболее важно, понимание различий между этим списком и предыдущим;
- **основные вычислительные задачи:** элементарная арифметика, сортировка, поиск, перечисление, обход вершин дерева (прямой (упорядоченный), центрированный (с порядковой выборкой), в обратном порядке (с отложенной выборкой), поиск в ширину и т. д.);
- **основные алгоритмы:** элементарные (арифметические) алгорифмы, последовательный поиск, двоичный (бинарный) поиск, сортировка (выбором, вставкой, слиянием, пирамидальная, быстрая, по-

разрядная (цифровая) и т. д.), поиск в ширину и в глубину в деревьях (как минимум в двоичных), а также, что наиболее важно, понимание различий между этим списком и предыдущим;

- **элементарный анализ алгоритмов:** асимптотическая нотация (o, O, Θ, Ω, ω), преобразование циклов в суммы и рекурсивных вызовов в рекуррентные соотношения, определение (вычисление) оценки простых сумм и рекуррентных (рекурсивных) соотношений;
- **математическая зрелость:** способность к абстракции, к формальным (особенно рекурсивным) определениям и к выполнению доказательств (особенно индуктивных); запись и прослеживание математических доказательств; распознавание и устранение синтаксических, смысловых (семантических) и/или логических элементов, не имеющих никакого смысла.

В этой книге кратко рассматриваются некоторые из перечисленных выше предварительно требуемых тем, когда этого требует контекст, но это в большей степени напоминание, нежели подробное введение. Для более глубокого изучения настоятельно рекомендуются следующие свободно распространяемые материалы:

- Margaret M. Fleck. «Building Blocks for Theoretical Computer Science». Версия 1.3 (January 2013) или более поздняя доступна здесь: **http://mfleck.cs.illinois.edu/building-blocks/**;
- Eric Lehman, F. Thomson Leighton, and Albert R. Meyer. «Mathematics for Computer Science». Версия июня 2018 г. доступна здесь: **https://courses.csail.mit.edu/6.042/spring18/**. (Настоятельно рекомендуется найти самую последнюю версию.);
- Pat Morin. «Open Data Structures». Редакция 0.1Gβ (январь 2016 г.) или более поздняя доступна здесь: **http://opendatastructures.org/**;
- Don Sheehy. «A Course in Data Structures and Object-Oriented Design». Версия февраля 2019 г. или более поздняя доступна здесь: **https://donsheehy.github.io/datastructures/**.

Дополнительные ссылки

Рекомендуется не ограничиваться одной из приведенных выше ссылок или каким-либо другим единственным источником. Авторы и читатели привносят собственные точки зрения и перспективные взгляды в любой учебно-просветительский материал. «Универсального преподавателя» для всех обучающихся подобрать невозможно, даже для очень умных студентов. Поиск «своего» автора, который наиболее эффективно вложит собственные знания и восприятие в ваш разум, требует определенных усилий, но эти усилия многократно окупаются в дальнейшем.

Приведенные ниже ссылки представляют собой особо ценные источники знаний и интуитивных представлений с примерами, упражнениями и развивающими заданиями, но этот список нельзя считать абсолютно полным.

- *Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman*. The Design and Analysis of Computer Algorithms. Addison-Wesley, 1974. (Я использовал эту книгу как студент в университете Райса – Rice University, потом как студент магистратуры в Калифорнийском университете в Ирвайне – UC Irvine.)
- *Boaz Barak*. Introduction to Theoretical Computer Science. Предварительная редакция книги, последняя редакция в июне 2019 г. (Это не книга по теории ИТ, доставшаяся вам от дедушки, это гораздо более качественный материал, а бесплатный доступ является дополнительным существенным преимуществом.)
- *Thomas Cormen, Charles Leiserson, Ron Rivest, and Cliff Stein*. Introduction to Algorithms, third edition. MIT Press/McGraw-Hill, 2009. (Я использовал первое издание этой книги, когда был ассистентом кафедры в университете Беркли – Berkeley.)
- *Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh V. Vazirani*. Algorithms. McGraw-Hill, 2006. (Вероятно, эта книга по содержанию ближе всего к моей, но она значительно менее подробна.)
- *Jeff Edmonds*. How to Think about Algorithms. Cambridge University Press, 2008.
- *Michael R. Garey and David S. Johnson*. Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman, 1979.
- *Michael T. Goodrich and Roberto Tamassia*. Algorithm Design: Foundations, Analysis, and Internet Examples. John Wiley & Sons, 2002.
- *Jon Kleinberg and Éva Tardos*. Algorithm Design. Addison-Wesley, 2005. Найдите эту книгу в библиотеке, если сможете.
- *Donald Knuth*. The Art of Computer Programming, volumes 1–4A. Addison-Wesley, 1997 and 2011. (Мои родители подарили мне первые три тома на Рождество, когда мне было 14 лет. Увы, до сих пор так и не удалось прочитать эти тома полностью.)
- *Udi Manber*. Introduction to Algorithms: A Creative Approach. Addison-Wesley, 1989. (Я использовал эту книгу, когда был ассистентом кафедры в университете Беркли – Berkeley.)
- *Ian Parberry*. Problems on Algorithms. Prentice-Hall, 1995 (не выпускалась в печатном виде). Книгу можно загрузить (**https://larc.unt.edu/ian/books/free/license.html**) после внесения небольшой суммы пожертвования. Соблюдайте соглашение о нераспространении.
- *Robert Sedgewick and Kevin Wayne*. Algorithms. Addison-Wesley, 2011.
- *Robert Endre Tarjan*. Data Structures and Network Algorithms. SIAM, 1983.
- Записи лекций по моему курсу алгоритмов в университете Беркли, особенно прочитанные Диком Карпом (Dick Karp) и Раймондом Зайделем (Raimund Seidel).

- Записи лекций, слайды, подготовительные материалы, экзаменационные материалы, видеолекции, отчеты об исследованиях, посты в блогах, вопросы и ответы на сайте StackExchange, подкасты и полнофункциональные массовые открытые онлайн-курсы, свободно распространяемые в веб-среде многочисленными учебными заведениями по всему миру.

Упражнения в этой книге

В конце каждой главы вниманию читателя предлагается несколько упражнений. большинство из которых я использовал как минимум один раз для домашних заданий, на семинарах и в лабораторных работах или на экзаменах. Упражнения не упорядочены по возрастанию сложности, но (в общем случае) сгруппированы по общему применяемому методу или по определенной теме. Некоторые задачи помечены символами, описанными ниже:

- символы красного сердечка (карточная масть «черви») ♥ обозначают особенно сложные задачи. Многие из этих задач предлагались студентам на экзаменах на получение степени доктора философии (Ph.D.) в университете Иллинойса. Некоторые (немногие) действительно сложные задачи обозначены увеличенным символом ♥;
- синие ромбики (карточная масть «бубны») ♦ обозначают задачи, для которых требуется изучение материала из следующих глав, но относящиеся к теме текущей главы. Но задачи, для которых требуется знание материала предыдущих глав, никак не помечены. В этой книге, как и в жизни, знания накапливаются постепенно;
- зеленые символы карточной масти «трефы» ♣ обозначают задачи, для которых требуется знание материала, не относящегося к тематике данной книги, например конечные автоматы, линейная алгебра, теория вероятностей или плоские графы. Но такие задачи встречаются редко;
- черные символы карточной масти «пики» ♠ обозначают задачи, требующие значительного объема рутинной работы и/или написания программного кода. Такие задачи также редко встречаются;
- оранжевые звездочки ★ означают, что вы едите «Лаки чармс» (Lucky Charms – детские сухие завтраки в виде глазированных фигурок-талисманов), произведенные до 1998 г., т. е. имеете дело с очень старыми продуктами. Брр.

Эти упражнения предоставляют возможность применить полученные знания на практике, а не являются «задачами ради самих задач». Цель каждого упражнения не решение конкретно сформулированной задачи, а развитие определенного набора навыков и умений или получение практического решения определенного типа задач. Отчасти по этой причине я не привожу решения упражнений, здесь решения не самое важное. Заметьте,

это не «учебное руководство», и если вы сами не можете решить задачу, то, вероятнее всего, вы не должны предлагать ее своим подопечным. Но при этом, возможно, вы сможете найти решения некоторых домашних заданий, которые опубликованы в этом семестре на веб-странице курса, который я читаю. Кроме того, что может помешать вам написать собственное учебное руководство.

Глава 0

Введение

Отсюда начинается алгоризм. Эта техника называется алгоризмом, используя который индийцы дважды наслаждаются пятью разными фигурами: 0. 9. 8. 7. 6. 5. 4. 3. 2. 1.
(Hinc incipit algorismus. Haec algorismus ars praesens dicitur in qua talibus indorum fruimur bis quinque figuris 0. 9. 8. 7. 6. 5. 4. 3. 2. 1.)

— Брат Александр де Вилья Деи (Friar Alexander de Villa Dei), Песнь об алгоризме (Carmen de Algorismo) (около 1220 г.)

Требуя от художника сознательного отношения к работе, Вы правы, но Вы смешиваете два понятия: решение вопроса и правильную постановку вопроса.

— А. П. Чехов, в письме А. С. Суворину (27 октября 1888 г.)

Чем больше мы приучаем себя к механическим действиям в малозначимых случаях, тем больше сил мы освобождаем для использования в более важных случаях.

— Анна Бракетт (Anna C. Brackett), The Technique of Rest (1892 г.)

И вот я в 2:30 ночи пишу о методе, несмотря на твердое убеждение, что мгновение, когда мужчина начинает говорить о методе, является доказательством того, что он только что лишился идей.

— Реймонд Чандлер (Raymond Chandler), в письме Эрлу Стенли Гарднеру (Erle Stanley Gardner) (5 мая 1939 г.)

Хорошему человеку не нужны правила.
Сегодня не тот день, чтобы выяснять, почему я имею так много.

— Доктор [Matt Smith], «Хороший человек идет на войну» (A Good Man Goes to War), сериал «Доктор Кто» (Doctor Who, 2011 г.)

0.1. Что такое алгоритм

Алгоритм – это явно заданная, точная, однозначная, механически исполняемая последовательность элементарных инструкций, обычно предназначенная для достижения конкретной цели. Например, ниже приведен алгоритм исполнения известной навязчивой песенки «99 бутылок пива на стене» (99 Bottles of Beer on the Wall) для произвольных значений, отличающихся от 99.

```
BottlesOfBeer(n):
  For i ← n down to 1
    Петь "i бутылок пива на стене, i бутылок пива,"
    Петь "Возьми одну, пусти по кругу, i – 1 бутылок пива на стене."

  Петь "Нет бутылок пива на стене, нет бутылок пива,"
  Петь "Сбегай в магазин, купи еще, n бутылок пива на стене."
```

Слово «алгоритм» происходит не от греческого корня *arithmos*, означающего «число», и *algos*, означающего «боль», как могли бы предположить алгоритмофобы-классики. Скорее, это искажение имени персидского ученого IX века Мухаммада ибн Мусы-аль-Хва̄ризми. Аль-Хва̄ризми, возможно, наиболее известен как автор трактата «Аль-Китāб аль-мухтасар ф āих āисāб аль-габр ва'л-мукāбала», от которого происходит современное слово «алгебра». В другом трактате аль-Хва̄ризми описал современную десятичную систему записи и манипулирования числами (в частности, использование маленького круга или сифра для обозначения недостающего количества), которая была разработана в Индии несколькими столетиями ранее. Методы, описанные в этом последнем трактате, с использованием либо письменных цифр, либо счетных камней, стали известны в английском языке как алгоризм или аугрим, а его цифры стали известны в английском языке как *ciphers* (шифры).

Несмотря на то что и позиционная нотация, и работы аль-Хва̄ризми были уже известны некоторым европейским ученым, «индийско-арабская» система была популяризирована в Европе средневековым итальянским математиком и торговцем Леонардо Пизанским, более известным как Фибоначчи. Во многом благодаря его книге «Liber Abaci»[1], написанной в 1202 г., запись чисел цифровыми символами стала заменять счетные таблицы (известные под названием «абак(а)» (*abacus*)) и «арифметику на пальцах»[2] как основные предпочитаемые методики вычислений[3] в Европе в XIII в. – но не потому, что запись десятичными цифрами была проще для обучения и использования, а потому, что такой способ обеспечивал ведение бухгалтерской книги. Цифры стали широко распространенными в Западной Европе только после появления наборного типографского шрифта,

[1] Несмотря на соблазн перевода названия Liber Abaci буквально как «Книга абака», более правильный перевод труда Фибоначчи – «Книга вычислений». И до и после Фибоначчи итальянское слово *abaco* использовалось для описания всего, что связано с вычислениями – устройств, методов, школ, книг и т. д. – во многом так же, как сегодня выражение *computer science* используется в английском языке (или «информационные технологии» в русском языке) или как китайское выражение «исследование операций» переводится буквально как «изучение использования счетных палочек».

[2] Счет с помощью десяти цифр-пальцев!

[3] Слово *calculate* (вычислять) произошло от латинского calculus, означающего «небольшой камешек», т. е. камешки на счетной таблице (доске), которые Джеффри Чосер (Chaucer, Geoffrey) называл *augrym stones*. В 440 г. до н. э. Геродот в своей «Истории» писал, что «греки пишут и считают (λογίζεσθαι ψήφοις, дословно: «считать камешками») слева направо, а египтяне делают это наоборот. Но они говорят, что их способ записи направлен вправо, а греческий способ записи направлен влево». (Странно, что Геродот ничего не говорит о том, с какого конца начинают разбивать яйцо египтяне.)

а действительно вездесущими – только при массовом производстве дешевой бумаги в самом начале XIX в.

В итоге слово «алгоризм» (*algorism*) превратилось в современный термин «алгоритм» (algorithm) по ложной («вульгарной») этимологии от греческого слова *arithmos* (и, возможно, от ранее упомянутого *algos*)[4]. Таким образом, до недавнего времени термин «алгоритм» относился исключительно к механическим (автоматически выполняемые) методам арифметики с использованием позиционной системы записи арабских цифр. Люди, обученные быстрому и надежному выполнению этих процедур, назывались алгоритмистами (*algorists*) или вычислителями – *computators*, упрощенно *computers*.

0.2. Умножение

Несмотря на то что как предмет в академии алгоритмы существуют лишь несколько десятилетий, они были вместе с нами с самого зарождения цивилизации. Описания пошаговых арифметических вычислений встречаются среди самых ранних образцов письменности, задолго до работ, представленных Фибоначчи и аль-Хорезми и даже до появления позиционной системы записи чисел, которую они внедряли.

Умножение методом решетки

Самый известный метод умножения больших чисел, по крайней мере для американских учащихся, – умножение методом решетки (lattice algorithm). Этот алгоритм был популяризирован Фибоначчи и описан в Liber Abaci. Фибоначчи узнал об этом алгоритме из арабских источников, в том числе из работ аль-Хорезми, который в свою очередь изучил его по индийским источникам, включая трактат VII в. «Brāhmasphuṭasiddhānta» Брахмагупты (Brahmagupta), который, возможно, обучался по китайским источникам. Старейшие сохранившиеся описания этого алгоритма встречаются в трактате «Математическая классика» Сунь Цзы, написанном в Китае между III и V вв., а также в комментариях Евтокия Аскалонского к трактату Архимеда «Об измерении круга», написанных около 500 г. н. э., но существуют свидетельства того, что этот алгоритм был известен намного раньше. Евтокий считает, что этот метод появился в последнем трактате Аполлона Пергийского (около 300 г. до н. э.) под названием «Okytokion» (Ὠκυτόκιον)[5]. Шумеры

[4] Некоторые средневековые источники утверждают, что греческая приставка *algo-* означает «искусство» или «введение». В других источниках сообщается, что алгоритмы были изобретены неким греческим философом, или королем Индии, или, возможно, королем Испании с именем Algus, или Algor, или Argus. Кое-кто, возможно, Данте Алигьери даже отождествлял изобретателя алгоритмов с мифологическим греческим кораблестроителем и одноименным аргонавтом (Арго). Совершенно неясно, претендовали ли все эти смехотворные утверждения на историческую точность или являлись лишь плодами богатой фантазии.

[5] Дословно: «лекарство, способствующее быстрым и легким родам». В папирусе из Александрийской библиотеки воспроизводились некоторые извлечения из «Okytokion» приблизительно за 200 лет до Евтокия, но его описание алгоритма умножения методом решетки (если Евтокий действительно приводил его) также утерян.

записывали схему умножения на глиняных табличках еще в 2600 г. до н. э., и можно предположить, что они, вероятно, использовали алгоритм умножения методом решетки[6].

Алгоритм умножения методом решетки предполагает, что исходные числа представлены как явным образом заданные строки цифр. Здесь я подразумеваю, что вычисления выполняются по основанию 10, но алгоритм легко обобщается для вычислений по любому другому основанию. Для упрощения нотации[7] входные данные состоят из пар элементов массивов $X[0..m-1]$ и $Y[0..n-1]$, представляющих числа:

$$x = \sum_{i=0}^{m-1} X[i] \cdot 10^i \quad \text{и} \quad y = \sum_{j=0}^{n-1} Y[j] \cdot 10^j.$$

Аналогичным образом выходные данные состоят из одного массива $Z[0..m+n-1]$, представляющего произведение:

$$z = x \cdot y = \sum_{k=0}^{m+n-1} Z[k] \cdot 10^k.$$

Этот алгоритм использует сложение и умножение одноразрядных чисел как простейшие операции. Сложение можно выполнить с использованием простого цикла `for`. На практике умножение одноразрядных чисел выполняется с помощью таблицы поиска (преобразования), вырезанной на глиняных табличках, нарисованной на деревянных или бамбуковых дощечках, записанной на бумаге, хранящейся в памяти, доступной только для чтения (ПЗУ), или в памяти вычислителя. Полный алгоритм решеточного умножения можно записать в виде обобщенной формулы:

$$x \cdot y = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} \left(X[i] \cdot Y[j] \cdot 10^{i+j} \right).$$

Различные варианты алгоритма решеточного умножения вычисляют частные произведения $X[i] \cdot Y[j] \cdot 10^{i+j}$ в разном порядке и используют разнообразные стратегии для вычисления их суммы. Например, в Liber Abaci

[6] Существует достаточно достоверное доказательство того, что древние шумеры выполняли точные вычисления с весьма большими числами, используя собственную шестидесятиричную позиционную систему записи чисел, но мне неизвестны какие-либо сохранившиеся записи о конкретных методах, применявшихся шумерами. В дополнение к стандартным таблицам умножения и обратных величин были найдены таблицы со списками квадратов целых чисел от 1 до 59, что привело некоторых историков математики к выводу о том, что вавилоняне выполняли умножение с использованием равенства, подобного $xy = ((x + y)^2 - x^2 - y^2)/2$. Но этот прием работает только при $x + y < 60$, а история умалчивает о том, как вавилоняне могли бы вычислить x^2 при $x \geqslant 60$.

[7] Но на грани разжигания исторического конфликта, как между Грецией и Египтом, или между Лилипутией и Блефуску, или между пользователями Mac и пользователями PC, или между людьми, считающими ноль натуральным числом, и тем, кто с этим не согласен.

Фибоначчи описывает вариант, в котором рассматриваются частные произведения *mn* в порядке возрастания величины, как показано в современном псевдокоде ниже.

```
FibonacciMultiply(X[0..m - 1], Y[0..n - 1]):
  hold ← 0
  for k ← 0 to n + m - 1
      for all i and j such that i + j = k
          hold ← hold + X[i] · Y[j]
      Z[k] ← hold mod 10
      hold ← ⌊hold/10⌋
  return Z[0..m + n - 1]
```

Алгоритм Фибоначчи часто выполняется с сохранением всех частных произведений в двумерной таблице (ее также называют «табло» (tableau), или «решетка» (grate), или «сетка» (lattice)) и последующим суммированием вдоль диагоналей с соответствующими переносами, как показано на рис. 0.1 справа. Ученики американских начальных школ обучаются умножению одного сомножителя (множимого) на каждую цифру второго сомножителя (множителя), записывая все произведения множителя на цифры множимого перед их суммированием, как показано на рис. 0.1 слева. Этот же метод был описан Евтокием, хотя он, как полагается, рассматривал цифры множителя слева направо, как показано на рис. 0.2. Оба эти варианта (и несколько других) описаны и проиллюстрированы в пошаговом виде в книге неизвестного автора «L'Arte dell'Abbaco» (1458 г.), также известной под названием «Treviso Arithmetic», первой печатной книги по математике на Западе.

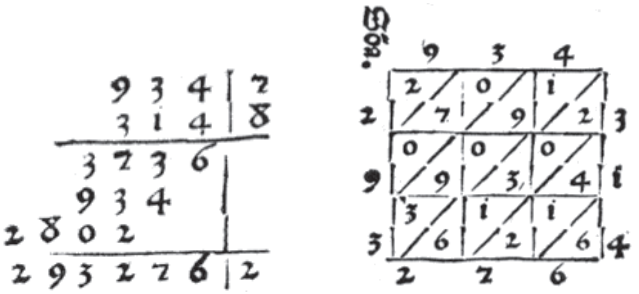

Рис. 0.1. Вычисление произведения 934 × 314 = 293276
с использованием умножения в столбик (с проверкой на ошибки методом, основанным на переходе к сравнениям, с «отбрасыванием» девяток) (слева) и решеточное умножение из книги «L'Arte dell'Abbaco» (1458 г.)
(Источник: Biblioteca nazionale Braidense (Milano);
http://atena.beic.it/webclient/DeliveryManager?pid=2953344)

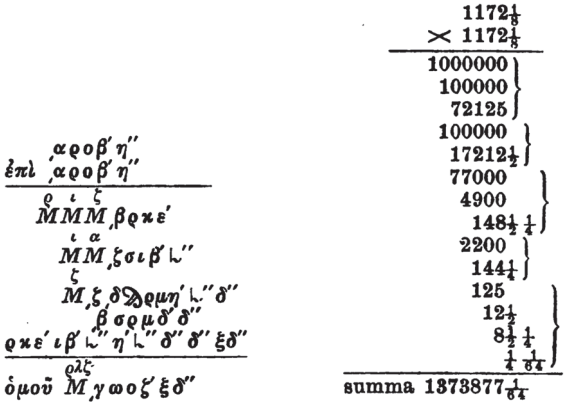

Рис. 0.2. Вычисление Евтокия в VI в. 1172 $^1/_8$ × 1172 $^1/_8$ = 1373877 $^1/_{64}$ в комментариях к трактату Архимеда «Об измерении круга», воспроизведенное в оригинале (слева) и преобразованное в современную форму записи (справа) Йоханом Хайбергом (Johan Heiberg) (1891 г.) (Источник: Глобальный архив Internet Archive; **https://archive.org/details/archimedisopera05eutogoog/page/n377**)

Все эти варианты алгоритма умножения методом решетки и другие схожие варианты, описанные Сунь Цзы, аль-Хорезми, Фибоначчи, в книге «L'Arte dell'Abbaco» и во многих других источниках, вычисляют произведение любого m-значного и любого n-значного числа за время $O(mn)$ – время выполнения каждого варианта определяется в основном количеством операций умножения отдельных цифр.

Удваивание и усреднение

Алгоритм умножения методом решетки не является самым старым алгоритмом умножения, для которого имеется прямое письменное подтверждение. Существует еще более древний и, вероятно, более простой алгоритм, не применяющий позиционную запись чисел, – его иногда называют русским или эфиопским крестьянским умножением или просто крестьянским умножением (peasant multiplication). Вариант этого алгоритма был скопирован в папирус Ринда (Rhind papyrus; или математический папирус Ахмеса) египетским писцом по имени Ахмес (Ahmes) около 1650 г. до н. э. из документа, который, по утверждению Ахмеса, был старше приблизительно на 350 лет[8]. Этому алгоритму продолжали обучать в начальных школах Восточной Европы даже в конце XX века, а кроме того, алгоритм широко использовался в самых первых цифровых компьютерах, в аппаратуре которых не было непосредственно реализовано целочисленное умножение.

[8] Версия этого алгоритма, действительно применявшаяся в Древнем Египте, не использовала усреднение или проверку на четность (сравнимость по модулю), но использовала сравнения. Чтобы избежать деления на два, алгоритм предварительно вычисляет две таблицы методом повторяющегося удваивания: одна таблица содержит все степени 2, не превышающие x, другая таблица содержит те же степени 2, умноженные на y. Затем выполняется поиск тех степеней 2, которые в сумме составляют число x, методом жадного вычитания, и соответствующие элементы в другой таблице суммируются для получения итогового произведения.

Алгоритм крестьянского умножения сводит сложную задачу умножения произвольных чисел к последовательности четырех более простых операций: (1) определения четности (четное или нечетное число), (2) сложения, (3) удваивания (числа) и (4) усреднения (взятие половины от числа с округлением в меньшую сторону или с недостатком).

```
PeasantMultiply(x, y):
  prod ← 0
  while x > 0
    if x is odd
      prod ← prod + y
    x ← ⌊x/2⌋
    y ← y + y
return prod
```

x	*y*		*prod*
			0
123	+ 456	=	456
61	+ 912	=	1368
30	~~1824~~		
15	+ 3648	=	5016
7	+ 7296	=	12312
3	+ 14592	=	26904
1	+ 29184	=	**56088**

Рис. 0.3. Умножение методом удваивания и усреднения

Правильность этого алгоритма доказывается методом индукции из следующего рекурсивного тождественного равенства, справедливого для всех неотрицательных целых чисел x и y:

$$x \cdot y = \begin{cases} 0 & \text{, если } x = 0; \\ \lfloor x/2 \rfloor \cdot (y + y) & \text{, если } x \text{ – четное число;} \\ \lfloor x/2 \rfloor \cdot (y + y) + y & \text{, если } x \text{ – нечетное число.} \end{cases}$$

Возможно, это рекуррентное тождество и есть алгоритм крестьянского умножения. Не позволяйте итеративному псевдокоду на рис. 0.3 обмануть вас – этот алгоритм рекурсивен по своей сущности.

Как было отмечено выше, алгоритм `PeasantMultiply` выполняет $O(\log x)$ операций проверки четности, сложения и усреднения, но можно улучшить его до $O(\log \min\{x, y\})$, если поменять местами аргументы при $x > y$. Предполагая, что числа представлены с использованием любой осмысленной позиционной нотации (двоичной, десятичной, вавилонской шестидесятеричной, египетской двенадцатеричной, римской, китайскими счетными палочками, позициями зернышек на абаке и т. д.), для каждой операции требуется как минимум $O(\log(xy)) = O(\log \max\{x, y\})$ операций умножения отдельных цифр, поэтому общее время выполнения алгоритма равно $O(\log \min\{x, y\} \cdot \log \max\{x, y\}) = O(\log x \cdot \log y)$.

Другими словами, этому алгоритму требуется время $O(mn)$ для умножения m-значного числа на n-значное число, с учетом постоянных сомножителей это то же самое время выполнения, что и для алгоритма умножения методом решетки. Этот алгоритм требует (при постоянном множителе) большего объема ручной работы, чем алгоритм решеточного умножения, но необходимые простейшие операции, вероятнее всего, гораздо проще выполняются человеком. В действительности эти два алгоритма равнозначны, если числа представлены в двоичном формате.

Циркуль и линейка

Греческие геометры-классики обозначали числа (или более точно – величины) отрезками прямых линий соответствующей длины, которыми они оперировали, используя два простых механических инструмента – циркуль и линейку, разновидности которых уже тогда широко применяли на практике землемеры, архитекторы и прочие ремесленники и мастеровые в течение многих столетий. Используя только эти два инструмента, ученые тех времен свели несколько сложных геометрических построений того времени к следующим простейшим операциям, начинающимся с одной или нескольких точно определенных исходных (начальных) точек:

- построению единственной прямой, проходящей через две различные точно определенные точки;
- построению единственной окружности с центром в одной точно определенной точке, проходящей через другую заданную точку;
- определению точки пересечения (если она существует) двух прямых;
- определению точек пересечения (если они существуют) прямой и окружности;
- определению точек пересечения (если они существуют) двух окружностей.

С определенной долей уверенности можно утверждать, что на практике греческие ученики, изучающие геометрию, выполняли построения на абаксе (abax (άβαξ)), поверхности, покрытой песком или плотным слоем пыли[9]. Несколькими веками ранее египетские землемеры выполняли многие из описанных выше построений, используя веревки для обозначения прямых и окружностей на земле[10]. Тем не менее Евклид и другие греческие геометры представляли построения с помощью циркуля и линейки как точные математические абстракции – точки как идеальные точки, прямые как идеальные прямые, окружности как идеальные окружности.

На рис. 0.4 показан алгоритм, описанный в «Началах» («Elements») Евклида около 2500 лет назад, для выполнения умножения или деления двух величин. Входные данные состоят из четырех различных точек A, B, C, D, а целью является построение точки Z, такой что $|AZ| = |AC||AD| / |AB|$. В частности, если определить $|AB|$ как единицу длины, то этот алгоритм вычисляет произведение $|AC|$ и $|AD|$.

Следует отметить, что Евклид сначала определяет новую элементарную операцию `RightAngle` с помощью написания «подпрограммы» (как назвали

[9] Изображаемые символы чисел от 1 до 9 были известны в Европе по меньшей мере за два века до написания Фибоначчи своего труда «Liber Abaci» как «гобар-числа» от арабского слова *ghubār*, означающего «пыль», что по существу представляло собой отсылку к индийскому практическому методу выполнения арифметических действий на поверхностях, покрытых песком. Греческое слово άβαξ является основой происхождения латинского слова *abacus*, которое изначально также означало поверхность, покрытую песком.

[10] Помните, что означает слово «геометрия»? Впоследствии Демокрит упоминал этих египетских землемеров в слегка ироничной манере, называя их *arpedonaptai* (άρπεδονάπται), что означает «вязатели веревок».

бы ее современные программисты). Корректность алгоритма доказывается следующим наблюдением: треугольники *ACE* и *AZF* являются подобными. Вторая и третья строки основного алгоритма неоднозначны, потому что прямая α пересекает любую окружность с центром *A* в двух различных точках, но алгоритм действительно корректен вне зависимости от того, какие точки пересечения выбраны для *E* и *F*.

```
⟨⟨Construct the line perpendicular to ℓ passing through P.⟩⟩
RightAngle(`, P):
Choose a point A ∈ ℓ
A, B ← Intersect(Circle(P, A), ℓ)
C, D ← Intersect(Circle(A, B), Circle(B, A))
return Line(C, D)
⟨⟨Construct a point Z such that |AZ| = |AC||AD|/|AB|.⟩⟩
MultiplyOrDivide(A, B, C, D):
α← RightAngle(Line(A, C), A)
E ← Intersect(Circle(A, B), α)
F ← Intersect(Circle(A, D), α)
β ← RightAngle(Line(E, C), F)
γ ← RightAngle(β, F)
return Intersect(γ, Line(A, C))
```

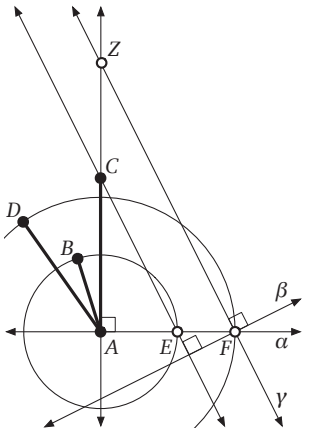

Рис. 0.4. Умножение с помощью циркуля и линейки

Алгоритм Евклида сводит задачу умножения двух величин (длин отрезков) к последовательности простейших операций, выполняемых с помощью циркуля и линейки. Эти операции сложно реализовать абсолютно точно в современных цифровых компьютерах, но алгоритм Евклида и не был предназначен для цифрового компьютера. Он был создан для платонова идеального геометра, вооруженного платоновым идеальным циркулем и платоновой идеальной линейкой, который способен точно выполнить каждую операцию за постоянное время согласно определению. В такой модели вычислений алгоритм `MultiplyOrDivide` выполняется за время $O(1)$.

0.3. Распределение мест в Конгрессе США

Ниже приведен еще один реальный пример практического алгоритма, весьма важного в сфере политики. Статья 1, раздел 2 Конституции США содержит следующее требование:

> «Места представителей и прямые налоги распределяются между отдельными штатами, которые могут быть включены в настоящий Союз, согласно численности их населения... Число представителей устанавливается из расчета один представитель не более чем от каждых тридцати тысяч жителей при том условии, что каждый штат будет иметь по крайней мере одного представителя...»

Поскольку в Палате представителей количество мест ограничено, точное пропорциональное представление требует либо представительства

одновременно от нескольких штатов, либо дробного числа представителей, но такие варианты недопустимы (незаконны и невозможны). В итоге в течение нескольких следующих десятилетий предлагалось множество разнообразных алгоритмов распределения, в которых использовалось «беспристрастное» округление идеального дробного решения. Алгоритм, практически применяемый в настоящее время, называется методом Хантингтона–Хилла (Huntington-Hill method) или методом равных пропорций (method of equal proportions), впервые предложенным статистиком Бюро переписи населения США Джозефом Хиллом (Joseph Hill) в 1911 г. и улучшенным математиком Гарвардского университета Эдвардом Хантингтоном (Edward Huntington) в 1920 г., затем принятым в качестве федерального закона (2 U.S.C. § 2a) в 1941 г. и оставшимся в силе после проверки в Верховном суде в 1992 г.[11].

Метод Хантингтона–Хилла распределяет представителей штатов поочередно по одному. Сначала на этапе предварительной обработки от каждого штата назначается один представитель. Затем на каждой итерации основного цикла следующий представитель назначается для штата с наивысшим приоритетом. Приоритет каждого штата определяется по формуле $P/\sqrt{r(r+1)}$), где P – численность населения штата, r – количество представителей, уже назначенных в этом штате.

Псевдокод этого алгоритма показан на рис 0.5. Входные данные состоят из массива $Pop[1..n]$, содержащего значения численности населения n штатов и целого числа R, равного общему количеству представителей, при этом предполагается, что $R \geqslant n$. (В настоящее время в США $n = 50$ и $R = 435$.) В выходном массиве $Rep[1..n]$ записано количество представителей от каждого штата.

Реализация алгоритма Хантингтона–Хилла использует очередь с приоритетами, поддерживающую операции `NewPriorityQueue`, `Insert` и `ExtractMax`. (Разумеется, в действующем законе ничего не сказано об очередях с приоритетами.) Вывод этого алгоритма, следовательно, его корректность никоим образом не зависит от того, как реализована эта очередь с приоритетами. Бюро переписи населения использует отсортированный массив, хранящийся в одном столбце электронной таблицы Excel, который вычисляется заново на каждой итерации. Вы изучали (по крайней мере, должны были изучать) более эффективную реализацию в начальном курсе по структурам данных.

[11] При аннулировании предыдущего решения Федерального окружного суда Верховный суд единогласно признал, что любой метод распределения, объективно принятый Конгрессом, является конституционным (Министерство торговли США против штата Монтана, юридический прецедент). Применяемый в настоящее время алгоритм распределения при избрании в Конгресс описан в чудовищных подробностях на веб-сайте Бюро переписи населения США **http://www.census.gov/topics/public-sector/congressional-apportionment.html**. Подробное описание истории решения этой задачи распределения можно найти здесь: **http://www.thirty-thousand.org/pages/Apportionment.htm**. Отчет Исследовательского управления Конгресса с описанием разнообразных методов распределения доступен здесь: **http://www.fas.org/sgp/crs/misc/R41382.pdf**.

```
ApportionCongress(Pop[1 .. n],R):
  PQ ← NewPriorityQueue
  ⟨⟨Give every state its first representative⟩⟩
  for s ← 1 to n
    Rep[s] ← 1
    Insert(PQ, s, Pop[i]/√2
  ⟨⟨Allocate the remaining n - R representatives⟩⟩
  for i ← 1 to n - R
    s ← ExtractMax(PQ)
    Rep[s] ← Rep[s] + 1
    priority ← Pop[s]/√Rep[s](Rep[s] + 1)
    Insert(PQ,s,priority)
  return Rep[1..n]
```

Рис. 0.5. Алгоритм распределения Хантингтона–Хилла

Подобные алгоритмы распределения используются при выборах в многопартийные парламенты по всему миру, где предполагается, что количество мест, выделяемых каждой партии, должно быть пропорциональным количеству голосов, полученных конкретной партией. Наиболее часто применяется метод Д'Ондта (D'Hondt)[12] и метод Сент-Лагю (Webster–Sainte-Laguë)[13], в которых используются соответственно приоритеты $P/(r + 1)$ и $P/(2r + 1)$ вместо выражения с квадратным корнем в методе Хантингтона–Хилла. Метод Хантингтона–Хилла является существенно единственным решением для Палаты представителей США во многом благодаря конституционному требованию, согласно которому каждый штат непременно должен иметь как минимум одного представителя.

0.4. Отрицательный пример

В качестве типичного примера последовательности инструкций, которая в действительности не является алгоритмом, рассмотрим «алгоритм Мартина»[14]:

```
BeAMillionaireAndNeverPayTaxes(): ⟨⟨БудьМиллионеромНоНикогдаНеПлатиНалоги():⟩⟩
  Получить миллион долларов.
  Если налоговый инспектор подходит к двери вашего дома и говорит:
  "Вы никогда не платили налоги!", -
    Скажите: "Я забыл".
```

[12] Разработан Томасом Джефферсоном (Thomas Jefferson) в 1792 г., использовался для распределения мест в Конгрессе США с 1792 по 1832 г., повторно открыт (разработан) бельгийским математиком Виктором Д'Ондтом (Victor D'Hondt) в 1878 г. и усовершенствован швейцарским физиком Эдуардом Хагенбахом-Бишоффом (Eduard Hagenbach-Bischoff) в 1888 г.

[13] Разработан Дэниэлом Уэбстером (Daniel Webster) в 1832 г., использовался для распределения мест в Конгрессе США с 1842 по 1911 г., повторно открыт (разработан) французским математиком Андре Сент-Лагю (André Sainte-Laguë) в 1910 г. и еще раз повторно разработан германским физиком Гансом Шеперсом (Hans Schepers) в 1980 г.

[14] Стив Мартин (Steve Martin), «You Can Be A Millionaire», телевизионное шоу Saturday Night Live, 21 января 1978 г. Также звучит в аудио-альбоме «Comedy Is Not Pretty», Warner Bros. Records, 1979 г.

Все весьма просто, за исключением самого первого шага – он сногсшибателен. Группа генеральных директоров-миллиардеров, венчурные инвесторы Кремниевой долины или дельцы, спекулирующие недвижимостью в Нью-Йорк-Сити, могут считать это алгоритмом, потому что для них самый первый шаг однозначен и банален[15], но для остальных бедных недотеп вроде нас с вами процедура Мартина является весьма неопределенной, чтобы рассматривать ее как настоящий алгоритм. С другой стороны, это превосходный пример редукции (сведе́ния) – в нем задача «перехода в миллионеры» и неуплаты налогов сводится к «более простой» задаче получения миллиона долларов. В этой книге операции редукции (сведе́ния) будут встречаться многократно. Сотни бизнесменов и политиков уже наглядно доказали: если вы знаете, как решить более простую задачу, то редукция подскажет, как решить более сложную.

Алгоритм Мартина, как и некоторые из приведенных выше примеров, не является тем типом алгоритмов, над которыми размышляют ученые-исследователи в области информационных технологий, поскольку он сформулирован с использованием терминологии операций, которые трудно выполнимы для компьютеров. Эта книга (почти) полностью сконцентрирована на алгоритмах, которые могут быть разумно реализованы на стандартном цифровом компьютере. Каждый шаг в этих алгоритмах либо поддерживается непосредственно языками программирования общего назначения (через арифметические операции, присваивания, циклы или рекурсию), либо представляет собой некоторое действие, способ выполнения которого вам уже известен (например, сортировка, двоичный поиск, проход по дереву или исполнение песни «*n* бутылок пива на стене»).

0.5. Описание алгоритмов

Профессиональные навыки, требуемые для эффективного проектирования и анализа алгоритмов, тесно связаны с навыками, требуемыми для эффективного описания алгоритмов. По крайней мере, в моих курсах полное описание любого алгоритма содержит четыре компонента:

- **что**: точная спецификация задачи, которую решает алгоритм;
- **как**: точное описание самого алгоритма;
- **почему**: доказательство того, что именно этот алгоритм решает поставленную задачу;
- **скорость (как быстро)**: анализ времени выполнения алгоритма.

Нет никакой необходимости в соблюдении указанного выше порядка разработки этих четырех компонентов (это даже нецелесообразно). Спецификации задач, описания алгоритмов, доказательства корректности и процедуры анализа времени выполнения обычно развиваются одновременно вместе с разработкой каждого компонента, предоставляющей информа-

[15] Что-то вроде надежно защищенного квантового блокчейна с алгоритмом глубокого обучения чему-то.

цию для разработки прочих компонентов. Например, может потребоваться поправка в описании задачи для поддержки более быстрого алгоритма или изменение алгоритма для обработки нестандартного (сложного) варианта при доказательстве корректности. Как бы то ни было, отдельное представление этих компонентов обычно наиболее понятно для читателя.

При написании любого материала важно ориентировать описания на соответствующую аудиторию. Я рекомендую создавать описания для опытных, но скептически настроенных программистов, которые не так умны, как вы. Подумайте о том, каким вы будете через шесть месяцев. При разработке любого нового алгоритма вы естественным образом выстраиваете множество интуитивных предположений о поставленной задаче и о том, как ваш алгоритм решает ее, поэтому такое неформальное обоснование управляется вашей интуицией. Но любой человек или объект, который будет в дальнейшем читать ваш алгоритм или код, написанный на основе этого алгоритма, не сможет воспользоваться вашей интуицией или опытом. Ни компилятор. Ни сами вы шесть месяцев спустя. Все, что окажется в наличии, – созданное вами описание.

Даже если вы никогда не объясняли свои алгоритмы кому-либо другому, важно при разработке держать в уме предполагаемую аудиторию. Попытка обмена информацией в понятной форме заставит вас мыслить более точно и ясно. Особенно описание для группы новичков, которые будут воспринимать слова в точности так, как они написаны, заставит вас тщательно проработать подробности независимо о того, насколько «очевидными» или «интуитивно понятными» могут показаться в этот момент разработанные вами концепции. Аналогично описание для скептически настроенной аудитории заставит вас проработать надежные аргументы для доказательства корректности и эффективности, а не полагаться лишь на интуицию или умственные способности[16].

Невозможно в достаточной степени выразить важность следующего положения: *ваша главная задача как создателя алгоритмов – объяснить другим людям, как и почему работают ваши алгоритмы*. Если вы не способны обменяться своими идеями и концепциями с другим человеком, то, вероятнее всего, ваши идеи как будто не существуют вовсе. Создание корректного и эффективно выполняемого кода является важной, но второстепенной целью. Убедить себя, своих профессоров, своих (потенциальных) работодателей, коллег или студентов в том, что вы действительно умны, – это в лучшем случае лишь третья, далеко не самая важная цель.

Определение конкретной задачи

Прежде чем начать разработку нового алгоритма, необходимо установить, для решения какой задачи предназначается этот алгоритм. Точно так же, прежде чем мы сможем приступить к описанию алгоритма, потребуется описание задачи, для решения которой предназначается этот алгоритм.

[16] В частности, я предполагаю, что читатель этой книги – скептически настроенный новичок.

Алгоритмические задачи часто формулируются с использованием обычного английского (русского или любого другого) языка с употреблением слов, обозначающих объекты реального мира. Но мы, как проектировщики алгоритмов, обязаны переформулировать эти задачи в терминах формальных, абстрактных, математических объектов, т. е. в форме чисел, массивов, списков, графов, деревьев и т. п., – таким образом, мы можем мыслить формально. Кроме того, мы обязательно должны определить, содержит ли формулировка задачи какие-либо скрытые неявные предположения (допущения), и зафиксировать эти предположения в явной форме. (Например, в песне «n бутылок пива на стене» подразумевается, что n всегда является неотрицательным целым числом[17].)

Возможно, потребуется редактирование спецификации в процессе разработки алгоритма. Например, для алгоритма может возникнуть необходимость в подробном представлении входных или итоговых выходных данных (результатов), которые остались не сформулированными в исходном неформальном описании задачи. Или алгоритм, возможно, в действительности решает более обобщенную задачу, нежели изначально предложенная. (Это часто встречающееся свойство рекурсивных алгоритмов.)

Спецификация должна включать ровно столько подробностей, сколько нужно для того, чтобы любой другой человек мог использовать алгоритм как черный ящик, не задумываясь о том, как и/или почему он работает. По существу, необходимо обязательно объяснить тип и смысл каждого входного параметра, а также точно описать, как итоговый вывод зависит от входных параметров. С другой стороны, эта спецификация должна умышленно скрывать все подробности, которые не требуются для применения алгоритма как черного ящика. Пусть то, что не имеет значения, будет надежно скрыто от посторонних глаз.

Например, оба алгоритма – решеточного умножения и удваивания и усреднения – решают одну и ту же задачу: взять два неотрицательных целых числа x и y, представленных в виде массивов цифр, вычислить их произведение $x \cdot y$, также представленное в виде массива цифр. Для того, кто использует эти алгоритмы, выбор того или иного варианта абсолютно не важен. С другой стороны, греческий алгоритм построений с помощью циркуля и линейки решает другую задачу, потому что входные и выходные значения представлены отрезками прямых, а не массивами цифр.

Описание алгоритма

Компьютерные программы являются точными представлениями алгоритмов, но алгоритмы – это не программы. Алгоритмы – это абстрактные автоматически выполняемые процедуры, которые можно реализовать на любом языке программирования, поддерживающем элементарные низкоуровневые операции. Идиосинкразические особенности синтаксиса вашего любимого языка программирования не имеют никакого значения, они

[17] Никогда не слышал, что кто-то поет «$\sqrt{2}$ бутылок пива на стене». Иногда приходилось слышать, как специалисты по теории множеств поют «$\aleph_0$ бутылок пива на стене», но по некоторой (вполне понятной) причине они никогда не могли допеть эту песню до конца.

лишь отвлекают ваше внимание (и внимание ваших читателей) от того, что в действительности происходит[18]. Качественное описание алгоритма ближе к тому, что необходимо писать в комментариях к реальной программе (как завершенному программному продукту), а не к ее исходному коду. Исходный код – плохое средство для повествования.

С другой стороны, описание в виде прозы на обычном английском (или любом другом языке) также не является удачным выбором. В алгоритмах содержится множество специфических структур – наиболее часто используются условные выражения, циклы, вызовы функций и рекурсия, – которые слишком легко «утопить» в неструктурированной прозе. Разговорный английский язык полон неоднозначностей и скрытых смыслов, но автор алгоритмов обязательно должен описывать их настолько однозначно и недвусмысленно, насколько это возможно. Обычная проза – плохой инструмент для соблюдения точности.

По моему мнению, самый точный способ представления алгоритма – использование сочетания псевдокода и структурированного английского языка. Псевдокод использует структуры формальных языков программирования и математические выражения для разделения алгоритма на элементарные шаги. Сами элементарные шаги могут быть записаны с применением математической нотации на правильном («чистом») английском языке или при объединении этих двух инструментов – выбирается самый понятный вариант. Правильно написанный псевдокод демонстрирует внутреннюю структуру алгоритма, но скрывает несущественные подробности реализации, делая алгоритм более простым для понимания, анализа, отладки и реализации.

Описание любого алгоритма должно включать все подробности, необходимые для полноценной спецификации алгоритма, для доказательства его корректности и для анализа времени его выполнения. В то же время следует исключить все подробности, которые не требуются для полноценной спецификации алгоритма, для доказательства его корректности и для анализа времени его выполнения. На уровне, более приближенном к практике, описание должно позволить опытному, но скептически настроенному программисту, который не читал эту книгу, быстро и правильно реализовать алгоритм на предпочитаемом языке программирования без необходимости понимания того, как работает этот алгоритм.

[18] Разумеется, это вопрос «религиозных убеждений». Лингвисты-догматики постоянно спорят о гипотезе Сепира–Уорфа (*Sapir-Whorf hypothesis*; гипотеза лингвистической относительности), которая полагает, что (в большей или меньшей степени) люди мыслят в категориях, определяемых их родным языком. В соответствии со строгой формулировкой этого принципа некоторые концепции одного языка просто недоступны для понимания носителям других языков, и не только из-за различий в техническом развитии – как бы вы перевели выражение «jump the shark» или «Fortnite streamer» на арамейский язык? – но и из-за неотъемлемых структурных различий между языками и культурами. Более скептическая точка зрения представлена Стивеном Пинкером (Steven Pinker) в книге «Язык как инстинкт» («The Language Instinct»). По общему признанию, эта идея обладает определенной силой в применении к различным парадигмам программирования. (Что такое *Y*-комбинатор? Как работают шаблоны? Что такое абстрактная фабрика?) К счастью, слишком незначительной, чтобы оказать какое-либо воздействие на материал этой книги. В качестве весьма убедительного контрпримера см. монографию Криса Окасаки (Chris Okasaki) «Purely Functional Data Structures», а также более поздние его работы.

Я не хочу утомлять вас перечислением правил, которые я соблюдаю при написании псевдокода, но обязан предостеречь от особо вредной привычки. *Никогда* не описывайте повторяющиеся операции неформально, как в этом (анти)примере: «Сделать [это] один раз, затем сделать [то же] во второй раз и так далее» или «Повторять этот процесс до тех пор, пока [некоторое условие]». Каждый, кому приходилось искать ответ на один из обескураживающих вопросов типа «Что происходит дальше в этой последовательности?», уже знает, что описание нескольких первых шагов алгоритма почти ничего не сообщает о том, что происходит на более поздних шагах. Если в алгоритме есть цикл, то запишите его как цикл и дайте явное описание происходящего на любой произвольной итерации. Если алгоритм рекурсивный, то запишите его в рекурсивной форме и точно опишите границы возможных вариантов и все происходящее в каждом варианте.

0.6. Анализ алгоритмов

Недостаточно просто записать алгоритм и сказать: «Узрите!». Необходимо также убедить аудиторию (да и нас самих) в том, что этот алгоритм действительно делает то, для чего он предназначен, и делает это эффективно.

Корректность

При некоторых параметрах настройки приложения вполне приемлемым для программ является корректное поведение в течение большей части времени при всех «разумных» входных данных. Но только не в этой книге – мы требуем, чтобы алгоритмы были корректными всегда, при всех возможных входных данных. Более того, мы обязаны доказать, что предлагаемые нами алгоритмы являются корректными, при этом недостаточно доверять лишь собственным инстинктивным (интуитивным) предположениям или протестировать только несколько вариантов. Иногда корректность действительно очевидна, особенно для алгоритмов, которые вы изучали в начальных курсах. С другой стороны, «очевидность» слишком часто становится синонимом понятия «ошибочность». Для большинства алгоритмов, рассматриваемых в этом курсе, требуется определенный объем работы для доказательства их корректности. В частности, при доказательстве корректности обычно используется метод индукции. Нам нравится индукция. Индукция – наш друг[19].

Разумеется, прежде чем мы сможем формально доказать, что предлагаемый алгоритм делает то, для чего он предназначен, сначала необходимо формально описать, что именно он должен делать.

Время выполнения

Наиболее часто используемым способом классификации различных алгоритмов для решения одной задачи является определение скорости их вы-

[19] Если индукция не является вашим другом, то вам трудно будет читать эту книгу.

полнения. В идеальном случае нужен самый быстрый алгоритм для любой конкретной задачи. При некоторых параметрах настройки приложения вполне приемлемым для программ является эффективное выполнение в течение большей части времени при всех «разумных» входных данных. Но только не в этой книге – мы требуем, чтобы алгоритмы всегда выполнялись эффективно, даже в наихудшем случае.

Но как измерить время выполнения алгоритма? В качестве конкретного примера: сколько времени занимает исполнение песни `BottlesOfBeer(n)`? Очевидно, что время исполнения является функцией от входного значения n, но оно также зависит от того, насколько быстро вы можете петь. Некоторым певцам может потребоваться десять секунд на исполнение куплета, другим – двадцать. Определенные методики и технологии еще больше расширяют возможности. Передача песни по телеграфу с использованием кода Морзе может потратить целую минуту на куплет. Загрузка в формате *mp3* из веба, вероятно, займет около десятой доли секунды на куплет. Для передачи содержимого *mp3*-файла в оперативную память компьютера, скорее всего, потребуется всего лишь несколько микросекунд на куплет.

Здесь самое важное то, как изменяется время исполнения песни при возрастании n. Исполнение `BottlesOfBeer(2n)` требует приблизительно в два раза больше времени, чем исполнение `BottlesOfBeer(n)`, независимо от используемой технологии. Это отображается в асимптотическом выражении времени исполнения $\Theta(n)$.

Можно измерять время, подсчитывая, сколько раз алгоритм выполняет определенную инструкцию или достигает контрольной точки в своем «коде». Например, можно заметить, что слово «beer» повторяется трижды в каждом куплете `BottlesOfBeer`, поэтому количество спетых слов «beer» – подходящий показатель общего времени пения. На этот вопрос можно дать точный ответ: в песне `BottlesOfBeer(n)` слово «beer» упоминается в точности $3n + 3$ раза.

Между прочим, существует множество песен с квадратичным временем исполнения. Вот, вероятно, одна из самых известных большинству людей, говорящих на английском языке:

```
NDaysOfChristmas(gifts[2..n]):
  for i ← 1 to n
    Sing "On the ith day of Christmas, my true love gave to me"
    for j ← i down to 2
      Sing "j gifts[j],"
    if i > 1
      Sing "and"
    Sing "a partridge in a pear tree."
```

Входными данными для `NDaysOfChristmas` является список из $n - 1$ подарков (gifts), представленный здесь как массив. Достаточно легко показать,

что время исполнения равно $\Theta(n^2)$, по существу, исполнитель упоминает название подарка $\sum_{i=1}^{n} i = n(n+1)/2$ раз (считая «partridge in the pear tree» – дословно: «куропатка на грушевой ветке»; идиом. «яблоки на сосне», «в лесу родилась елочка»). Кроме того, легко заметить, что во время первых n дней Рождества «моя единственная любовь» дарит в точности $\sum_{i=1}^{n}\sum_{j=1}^{n} j = n(n+1)(n+2)/6 = \Theta(n^3)$ подарков.

К другим песням с квадратичным временем исполнения относятся: «Old MacDonald Had a Farm» («У старого МакДональда была ферма»), «There Was an Old Lady Who Swallowed a Fly» («Я знаю старушку, которая проглотила муху»), «Hole in the Bottom of the Sea» («На дне моря есть дыра»), «Green Grow the Rushes O» («Растет зеленый тростник, О»), «The Rattlin' Bog» («Шумное болото»), «The Court Of King Caractacus» («Двор короля Каракта-ка»), «The Barley-Mow» («Стог ячменя»), «If I Were Not Upon the Stage» («Если бы я был не на сцене»), «Star Trekkin'» («Звездный путь»), «Ist das nicht ein Schnitzelbank?» («Это не резная скамейка?»)[20], «Il Pulcino Pio» («Цыпленок Пи»), «Minkurinn í hænsnakofanum» («Норка в курятнике»), «Echad Mi Yodea» («Кто знает Единого») и «Το κοκοράκι» («Маленький петушок»). Если нужно больше примеров, спросите у хорошо знакомого вам ребенка дошкольного возраста.

```
Alouette(lapart[1..n]):
  Chantez «Alouette, gentille alouette, alouette, je te plumerai.»
  pour tout i de 1 à n
    Chantez "Je te plumerai lapart[i]. Je te plumerai lapart[i]."
    pour tout j de i à 1 ⟨à rebours — вернуться назад⟩
      Chantez "Et lapart[j]! Et lapart[j]!"
    Chantez "Alouette! Alouette! Aaaaaa..."
    Chantez "...alouette, gentille allouette, alouette, je te plumerai."
```

Для некоторых песен время исполнения выглядит еще более странным образом. Относительно недавним примером является песня Гая Стила (Guy Steele) «The TELNET Song», которая в действительности требует времени $\Theta(2^n)$ для исполнения первых n куплетов, Стил рекомендовал $n = 4$. Наконец, существует несколько песен, которые не заканчиваются никогда[21].

За исключением «The TELNET Song» все перечисленные выше песни в основном вполне естественно записываются в виде небольшого набора вложенных циклов, поэтому время их исполнения (выполнения) можно вычислить с использованием вложенных операций суммирования. Время выполнения рекурсивного алгоритма проще выразить через рекуррентную формулу, как показано ниже:

[20] «Ja, das ist Otto von Schnitzelpusskrankengescheitmeyer!» – «Да, это Отто фон Шницельпусскранкэнгешайтмейер!»

[21] Они просто продолжаются и продолжаются бесконечно, мой друг.

$$x \cdot y = \begin{cases} 0 & \text{, если } x = 0; \\ \lfloor x/2 \rfloor \cdot (y + y) & \text{, если } x \text{ – четное число;} \\ \lfloor x/2 \rfloor \cdot (y + y) + y & \text{, если } x \text{ – нечетное число.} \end{cases}$$

Пусть $T(x, y)$ обозначает количество операций определения четности, сложения и усреднения, требуемых для вычисления $x \cdot y$. Эта функция соответствует рекурсивному неравенству $T(x, y) \leqslant T(\lfloor x/2 \rfloor, 2y) + 2$ в основном варианте $T(0, y) = 0$. Методики, описанные в следующей главе, полагают, что верхняя граница $T(x, y) = O(\log x)$.

Иногда время выполнения алгоритма зависит от конкретной реализации некоторой внутренней структуры данных в подпрограмме. Например, алгоритм распределения представителей Хантингтона–Хилла `ApportionCongress` выполняется за время $O(N + RI + (R - n)E)$, где N обозначает время выполнения подпрограммы `NewPriorityQueue`, I – время выполнения операции `Insert`, а E – время выполнения операции `ExtractMax`. При разумном предположении $R \geqslant 2n$ (в среднем каждый штат получает как минимум двух представителей) можно упростить это предельное время выполнения до $O(N + R(I + E))$. Точное время выполнения зависит от реализации внутренней очереди с приоритетами. Бюро переписи населения реализует очередь с приоритетами как неотсортированный массив, для которого $N = I = \Theta(1)$ и $E = \Theta(n)$, поэтому предлагаемая Бюро переписи населения реализация алгоритма `ApportionCongress` выполняется за время $O(Rn)$. Но если реализовать очередь с приоритетами как двоичную кучу (пирамиду) или как динамический упорядоченный массив (heap-ordered array), то получим $N = \Theta(1)$ и $I = E = O(\log n)$, так что весь алгоритм в целом выполняется за время $O(R \log n)$.

Иногда требуется оценка не только времени выполнения, но и других вычислительных ресурсов, таких как пространство (объем) памяти, количество бросков монеты, количество промахов кеша или страницы памяти, количество сообщений, передаваемых между процессами, или количество «подарков от моей единственной возлюбленной». Эти ресурсы можно проанализировать с использованием тех же методик, которые применяются для анализа времени выполнения. Например, решеточное умножение двух n-значных чисел требует $O(n^2)$ пространства (памяти), если записываются все промежуточные (частичные) произведения перед их сложением, но только лишь $O(n)$ пространства (объема памяти), если произведения суммируются динамически (без промежуточного запоминания).

Упражнения

0. Описать и проанализировать эффективный алгоритм, который определяет с учетом разрешенного размещения стандартных фигур на стандартной шахматной доске, какой игрок выигрывает партию из заданной начальной позиции, если оба игрока играют безошибочно. [*Подсказка*: существует простейшее однострочное решение.]

♥1. a) Определить (или написать) песню, которая требует времени $\Theta(n^3)$ для исполнения первых n куплетов.

b) Определить (или написать) песню, которая требует времени $\Theta(n \log n)$ для исполнения первых n куплетов.

c) Определить (или написать) песню, которая требует некоторого другого невероятного времени для исполнения первых n куплетов.

2. Внимательные читатели, возможно, заметили и посетовали, что анализ таких песен, как «n бутылок пива на стене» или «n дней Рождества» слишком упрощен, потому что большие числа определяют более длительное время исполнения, чем малые числа. В более общем смысле: поскольку существует только определенное количество слов заданной длины, более крупные наборы слов не обязательно содержат более длинные слова[22]. Можно более точно оценить время исполнения песни, подсчитывая число спетых слогов, а не количество слов.

   a) Сколько времени потребуется, чтобы спеть целое число n?

   b) Сколько времени потребуется, чтобы спеть «n бутылок пива на стене»?

   c) Сколько времени потребуется, чтобы спеть «n дней Рождества»?

   Как обычно, необходимо представить ответы в форме $O(f(n))$ для некоторой функции f.

3. Застольная песня с многочисленными повторами «The Barley Mow» («Стог ячменя») веками исполнялась на Британских островах. Существует множество вариаций этой песни. На рис. 0.6 показан текст одной из версий, традиционных для графств Девон (Девоншир) и Корнуолл. Здесь `vessel[i]` – это название сосуда, содержащего 2^i унций пива[23].

[22] Ja, das ist das Subatomarteilchenbeschleunigungsnaturmäßigkeitsuntersuchungsmaschine! – Да, это машина для исследования природного (естественного) ускорения субатомных частиц!

[23] В действительности в этой песне используется некоторое подмножество следующих сосудов: nipperkin (рюмашка), quarter-gill (четверть джила), half-a-gill (полджила), gill (джил), quarter-pint (четверть пинты), half-a-pint (полпинты), pint (пинта), quart (кварта), pottle (4 пинты, или полгаллона), gallon (галлон), half-anker (полбочонка), anker (бочонок), firkin (бочонок побольше), half-barrel/kilderkin (полбочки/килдеркин), barrel (бочка), hogshead (большая бочка, хогсхед), pipe/butt (очень большая (вертикальная) бочка), tun (огромная бочка-резервуар), well (колодец), river (река) и ocean (океан). С некоторыми исключениями (особенно в конце) каждый сосуд в этом списке вдвое больше по объему предыдущего. В ирландской и шотландской версиях этой песни немного другие тексты, и после галлона (gallon) они обычно переключаются на людей (barmaid (барменша), landlord (владелец (заведения), drayer и т. д.). Ранняя версия этой песни под названием «Give us once a drink» («Дай нам хоть раз выпить») впервые появилась в пьесе «Jack Drum's Entertainment, or the Comedie of Pasquill and Katherine» («Развлечения Джека Драма, или Комедия Пасквиля и Катерины»), написанной Джоном Марстоном (John Marston) около 1600 г. («Giue vs once a drinke for and the black bole. Sing gentle Butler bally moy!» (староангл.)). Существуют разногласия касательно того, написал ли Марстон эту «застольную голландскую песню» (high Dutch Song) специально для своей пьесы, является ли словосочетание «bally moy» мондегрином (дарвалдаем, т. е. неправильно услышанной фразой) от «barley mow» (собственно: «стог ячменя») и наоборот, или же это действительно та же самая песня. Эти споры лучше всего протекают за n бутылками пива.

```
BarleyMow(n):
  "Here's a health to the barley-mow, my brave boys,"
  "Here's a health to the barley-mow!"
  "We'll drink it out of the jolly brown bowl,"
  "Here's a health to the barley-mow!"
  "Here's a health to the barley-mow, my brave boys,"
  "Here's a health to the barley-mow!"
  for i ← 1 to n
    "We'll drink it out of the vessel[i], boys,"
    "Here's a health to the barley-mow!"
    for j ← i downto 1
      "The vessel[ j],"
    "And the jolly brown bowl!"
    "Here's a health to the barley-mow!"
    "Here's a health to the barley-mow, my brave boys,"
    "Here's a health to the barley-mow!"
```

Рис. 0.6. Исполнение песни «The Barley Mow» («Стог ячменя»)

a) Предположим, что каждое название сосуда `vessel[n]` является одним словом, и можно петь со скоростью четыре слова в секунду. Сколько времени потребуется на исполнение песни `BarleyMow(n)`? (Дать ответ в форме строгого асимптотического граничного значения.)

b) Если необходимо исполнять эту песню с произвольными большими значениями n, то придется придумать собственные названия сосудов. Чтобы избежать повторений, эти названия должны становиться все длиннее при увеличении n. Предположим, что в названии `vessel[n]` содержится $\Theta(\log n)$ слогов, и можно петь шесть слогов в секунду. Сколько времени в этом случае потребуется для исполнения песни `BarleyMow(n)`? (Дать ответ в форме строгого асимптотического граничного значения.)

c) Предположим, что при упоминании каждого названия сосуда действительно выпивается соответствующее количество пива: унция в каждой чаше «jolly brown bowl» и 2^i унций в каждом сосуде `vessel[i]`. Предполагая по условиям этой задачи, что вам не меньше 21 года, сколько унций пива вы выпьете при исполнении песни `BarleyMow(n)`? (Необходимо дать точный ответ, а не асимптотическое граничное значение.)

4. Напомню, что входными данными для алгоритма Хантингтона–Хилла `ApportionCongress` является массив $Pop[1..n]$, где $Pop[i]$ – численность населения i-го штата, и целое число R – общее количество представителей, которые должны быть распределены. Выходные

данные: массив *Rep*[1..*n*], где *Rep*[*i*] – число представителей, назначенных от *i*-го штата по этому алгоритму.

Алгоритм Хантингтона–Хилла иногда описывается способом, позволяющим полностью избежать использования очередей с приоритетами. Алгоритм самого верхнего уровня «угадывает» положительное действительное число *D*, называемое делителем, затем выполняет приведенную ниже подпрограмму для вычисления распределения. Переменная *q* – идеальная квота для представителей, назначаемых от штата, при заданном делителе *D*. Действительное число назначенных представителей всегда равно либо ⌈*q*⌉, либо ⌊*q*⌋.

```
HHGuess(Pop[1..n],R,D):
  reps ← 0
  for i ← 1 to n
    q ← Pop[i]/D
    if q · q < ⌈q⌉ · ⌊q⌋
      Rep[i] ← ⌊q⌋
    else
      Rep[i] ← ⌈q⌉
    reps ← reps + Rep[i]
  return reps
```

Существует три возможных варианта для конечного возвращаемого значения `reps`. Если `reps` < *R*, то требуемое количество представителей не назначено, что (по крайней мере, интуитивно) означает, что выбранный делитель *D* слишком мал. Если `reps` > *R*, то назначено слишком много представителей, что (по крайней мере, интуитивно) означает, что выбранный делитель *D* слишком велик. Наконец, если `reps` = *R*, то можно возвращать массив *Rep*[1..*n*] как окончательное распределение мест в Конгрессе. На практике можно вычислить корректное распределение (с результатом `reps` = *R*), вызывая подпрограмму `HHGuess` с небольшим количеством целочисленных делителей, близких к стандартному делителю *D* = *P*/*R*.

В предлагаемых ниже задачах предполагается, что $P = \sum_{i=1}^{n} Pop[i]$ обозначает общую численность населения во всех *n* штатах, а также что $n \leqslant R \leqslant P$.

a) Показать, что вызов `HHGuess` со стандартным делителем *D* = *P*/*R* не всегда позволяет получить корректное распределение.

b) Доказать, что если `HHGuess` возвращает одинаковое значение `reps` для двух различных делителей *D* и *D*′, то вычисляется и одинаковое распределение *Rep*[1..*n*] для обоих этих делителей.

c) Доказать, что если `HHGuess` возвращает верное значение *R*, то вычислено то же распределение *Rep*[1..*n*], что и при использовании ранее рассмотренного алгоритма `ApportionCongress`.

d) Доказать, что «верный» делитель D, возможно, не существует. То есть описать входные данные $Pop[1..n]$ и R, где $n \leqslant R \leqslant P$, так, что для каждого действительного числа $D > 0$ число представителей, распределенных подпрограммой `HHGuess`, не равно R. [*Подсказка*: что произойдет, если заменить знак < на знак $\leqslant$ в четвертой строке подпрограммы `HHGuess`?]

Глава 1

Рекурсия

Контроль над большими силами осуществляется по тому же принципу, что и контроль над несколькими людьми: это просто вопрос разделения их численности.

— Сунь Цзю, «Искусство войны» (около 400 г. н. э.)
(перевел на англ. яз. Лайонел Джайлз (Lionel Giles), 1910 г.)

Наша жизнь испорчена подробностями... Упрощайте, упрощайте.

— Генри Дэвид Торо (Henry David Thoreau), «Уолден, или Жизнь в лесу» (1854 г.)

И не спрашивайте меня, что такое Voom. Я никогда об этом не узнаю.
Но парень! Должен сказать тебе, что он ДЕЙСТВИТЕЛЬНО чистит снег!

— Доктор Сьюз (Dr. Seuss), Теодор Сьюз Гайсел (Theodor Seuss Geisel),
«Кот в шляпе возвращается» (1958 г.)

В первую очередь делайте тяжелую работу. Легкая работа позаботится о себе сама.

— Цитата приписывается Дейлу Карнеги (Dale Carnegie)

1.1. Сведе́ние

Сведе́ние, или редукция (reduction), – одна из методик, наиболее часто используемых в проектировании алгоритмов. Сведе́ние одной задачи X к другой задаче Y означает создание алгоритма для X, который использует алгоритм для Y как черный ящик или подпрограмму. Особенно важно то, что корректность итогового алгоритма для задачи X не может зависеть каким бы то ни было образом от того, как работает алгоритм для задачи Y. Единственное, что можно предполагать в этом случае, – черный ящик корректно решает задачу Y. Внутренние рабочие операции этого черного ящика просто нас не касаются, это задача решается кем-то другим. Часто наилучшим подходом является восприятие черного ящика как исключительно магического устройства.

Например, алгоритм «крестьянского умножения», описанный в предыдущей главе, сводит задачу умножения двух произвольных положительных целых чисел к трем более простым задачам: сложению, усреднению (взятию половины) и проверке четности. Алгоритм основан на абстрактном типе данных «положительное целое число», который поддерживает

три перечисленные операции, но корректность этого алгоритма умножения не зависит от точности представления данных (засечки на дощечке, глиняные таблички, вавилонская шестидесятеричная система, узелковое письмо кипу, счетные палочки, римские цифры, счет на пальцах, камешки на счетной таблице, числа гобар (хинди), двоичные числа, нега-позиционная двоичная система счисления, код Грея, равновесная троичная система счисления, фи-нарная система счисления (основанная на золотом сечении), четверично-мнимая система счисления…) или от точности реализации элементарных операций. Разумеется, время выполнения алгоритма умножения зависит от времени выполнения операций сложения, усреднения и определения четности, но это характеристика, не относящаяся к корректности. Наиболее важно то, что можно создать более эффективный алгоритм умножения с помощью простого перехода на более удобное представление данных (например, от засечек на дощечке к позиционной системе записи чисел).

Аналогичным образом алгоритм Хантингтона–Хилла сводит задачу распределения мест в Конгрессе к задаче сопровождения очереди с приоритетами, которая поддерживает операции `Insert` (вставка) и `ExtractMax` (извлечение максимального значения). Абстрактный тип данных «очередь с приоритетами» представляет собой черный ящик, а корректность алгоритма распределения не зависит от любой конкретной структуры данных очереди с приоритетами. Разумеется, время выполнения алгоритма распределения зависит от времени выполнения алгоритмов `Insert` и `ExtractMax`, но эти характеристики полностью отделены от характеристики корректности алгоритма. Достоинство сведе́ния состоит в том, что можно создать более эффективный алгоритм распределения, просто применив новую структуру данных очереди с приоритетами. Более того, проектировщику такой структуры данных не обязательно знать или беспокоиться о том, что она будет использоваться для распределения мест в Конгрессе.

При проектировании алгоритмов автор может не знать точно, как именно реализованы используемые им базовые структурные блоки или как создаваемые алгоритмы могут применяться в качестве базовых структурных блоков для решения более крупномасштабных задач. Такое незнание создает дискомфорт для многих начинающих алгоритмистов, но это неизбежно и чрезвычайно полезно. Даже если вы точно знаете, как работают используемые компоненты, часто весьма полезно сделать вид, что вам это неизвестно.

1.2. Упрощение и делегирование

Рекурсия (recursion) – это особенно мощный тип сведе́ния, который кратко можно описать следующим образом:

- если заданный частный случай задачи можно решить напрямую, то решить его напрямую;

- иначе свести его к одному или нескольким более простым частным случаям той же задачи.

Если ссылка на себя (рекурсивная ссылка) вас смущает, то, возможно, удобнее представить, что кто-то другой будет решать эти более простые задачи, как можно было бы представить это для других типов сведе́ния. Я предпочитаю называть этого «кого-то другого» Феей Рекурсией (Recursion Fairy). Ваша единственная обязанность – упростить исходную задачу или решить ее напрямую, если упрощение не нужно или невозможно, а Фея Рекурсия решит за вас все более простые подзадачи, применяя методы, «о которых вам не нужно знать», так что «не лезьте не в свое дело» (butt out)[1]. Читатели с хорошей математической подготовкой, вероятно, поняли, что под именем Феи Рекурсии скрывается более формальная сущность: индуктивное предположение (Induction hypothesis).

Существует одно умеренное техническое условие, которое необходимо соблюдать для обеспечения корректной работы любого рекурсивного метода: последовательность сведе́ний для получения все более простых частных случаев задачи не должна быть бесконечной. В конце концов рекурсивные сведе́ния непременно должны приводить к элементарному частному случаю задачи, который можно решить некоторым другим методом, иначе рекурсивный алгоритм будет бесконечно зациклен. Наиболее часто применяемым способом выполнения вышеописанного условия является сведе́ние к одному или нескольким более простым (более «мелким») частным случаям той же задачи. Например, если исходными входными данными является некий элемент с n атрибутами, то входными данными для каждого рекурсивного вызова должен быть элемент, в котором обязательно содержится строго меньше, чем n атрибутов. Разумеется, невозможно, чтобы элемент вообще не содержал атрибутов (не может быть отрицательным число атрибутов – это глупо и бессмысленно!), так что в этом случае мы непременно должны «перезаточить» элемент, используя некоторый другой метод.

Нам уже знаком один частный случай этого шаблона в алгоритме «крестьянского умножения», основанного непосредственно на следующем рекурсивном тождестве.

$$x \cdot y = \begin{cases} 0 & \text{, если } x = 0; \\ \lfloor x/2 \rfloor \cdot (y + y) & \text{, если } x \text{ – четное число;} \\ \lfloor x/2 \rfloor \cdot (y + y) + y & \text{, если } x \text{ – нечетное число.} \end{cases}$$

Это рекурсивное (рекуррентное) тождество можно записать в алгоритмической форме, как показано ниже:

[1] Когда я был студентом начальных курсов, то приписывал рекурсию «эльфам», а не Фее Рекурсии, в соответствии со сказкой братьев Гримм о старом сапожнике, который не закончил работу и лег спать, а когда проснулся, то обнаружил, что эльфы («Wichtelmänner») ночью сделали все за него. Кто-то более подкованный в энтеогенном плане, чем я, может распознать в этих «рекурсивных эльфах» (Rekursionswichtelmänner) «самотрансформирующихся механических (фрактальных) эльфов» Терренса Кемпа Маккенны (Terence Kemp McKenna).

```
PeasantMultiply(x,y):
  if x = 0
    return 0
  else
    x' ← ⌊x/2⌋
    y' ← y + y
    prod ← PeasantMultiply(x', y')      ⟨⟨Рекурсия!⟩⟩
    if x is odd
      prod ← prod + y
    return prod
```

Ленивый египетский писец мог выполнить этот алгоритм, вычисляя x' и y' и попросив более юного коллегу перемножить x' и y', а затем, вероятно, сложив у с результатом младшего писца. Задача юного писца проще, потому что $x' < x$, и постоянное уменьшение положительного целого числа в итоге приводит к 0. Старшего писца совершенно не интересует, как младший коллега в действительности вычисляет произведение $x' \cdot y'$ (и нам тоже нет до этого никакого дела).

1.3. Ханойские башни

Головоломка «Ханойская башня» (Tower of Hanoi) впервые была опубликована – как действительно существующая в реальном мире задача – французским преподавателем и специалистом по занимательной математике Эдуардом Люка (Édouard Lucas) в 1883 г.[2] под псевдонимом N. Claus (de Siam) (Н. Клаус (из Сиама)) (это анаграмма фразы «Lucas d'Amiens» (Люка из Амьена)). В следующем году Анри де Парвиль (Henry de Parville) описал эту головоломку в следующем удивительном рассказе[3]:

> «В большом храме города Бенареса[4] под куполом, накрывающим "центр всего мира" покоится медная плита, в которую вставлены три алмазные иглы длиной в локоть и толщиной в осиную талию. На одну из этих игл Бог при сотворении мира нанизал шестьдесят четыре диска из чистого золота – самый большой диск лежит в самом низу на медной плите, и каждый диск, лежащий выше, меньше предыдущего. Это "Башня Брамы". День и ночь священнослужители неустанно переносят диски с одной алмазной иглы на другую, руководствуясь навеки установленными и непреложными законами Брамы, по которым священнослужитель не должен двигать зараз более одного диска и всегда должен так переносить этот диск на иглу, чтобы под ним

[2] Позже Люка признался, что придумал эту головоломку в 1876 г.

[3] Для этой книги был использован перевод на английский язык, взятый из книги Уолтера Уильяма Роуз Болла (W. W. Rouse Ball) «Mathematical Recreations and Essays» («Математические эссе и развлечения»; есть перевод на русский язык) (1892 г.)

[4] Под «большим храмом города Бенареса» почти определенно подразумевается Храм Каши Вишванатх (Каши Вишванатх мандир) в священном городе Варанаси (штат Уттар-Прадеш, Индия), расположенный приблизительно на расстоянии 2400 км к северо-западу от Ханоя (Вьетнам), где предположительно проживает воображаемый Н. Клаус. По воле случая французская армия захватила Ханой в 1883 г., в том же году Люка опубликовал свою головоломку, в итоге Ханой был провозглашен столицей французского Индокитая.

не оказался диск, меньший его по размеру. А когда все шестьдесят четыре диска будут перенесены с той иглы, на которую Бог поместил их при сотворении мира, на одну из двух других игл, то башня, храм и сами священнослужители-брамины обратятся в прах, и грянет гром, и мир исчезнет».

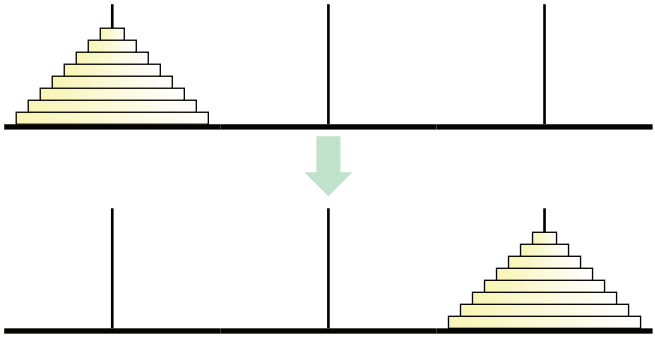

Рис. 1.1. Головоломка «Ханойская башня» (с 8 дисками)

Разумеется, как опытные специалисты в области информационных технологий, при чтении этого рассказа мы сразу же приходим к мысли о замене «жестко закодированной» константы 64 на переменную *n*. А поскольку большинство материальных экземпляров этой головоломки сделано из дерева, а не из алмазов и золота, я буду называть три возможных места для дисков стержнями, а не иглами. Как можно переместить башню из *n* дисков с одного стержня на другой, используя третий дополнительный стержень как временное место расположения дисков, при условии, что нельзя положить диск большего размера на диск меньшего размера?

Как подсказывает Н. Клаус (из Сиама) в своем наставлении, включенном в эту головоломку, секрет ее решения заключается в рекурсивном способе мышления. Вместо попыток решения всей головоломки сразу попробуйте сосредоточиться на перемещении только одного самого большого диска. Мы не можем переместить его первым, потому что этому мешают другие диски. Поэтому сначала необходимо переместить $n - 1$ этих дисков меньшего размера на вспомогательный стержень. После этого появляется возможность положить самый большой диск прямо на целевой стержень. Далее для завершения решения головоломки необходимо переместить $n - 1$ дисков меньшего размера со вспомогательного на целевой стержень.

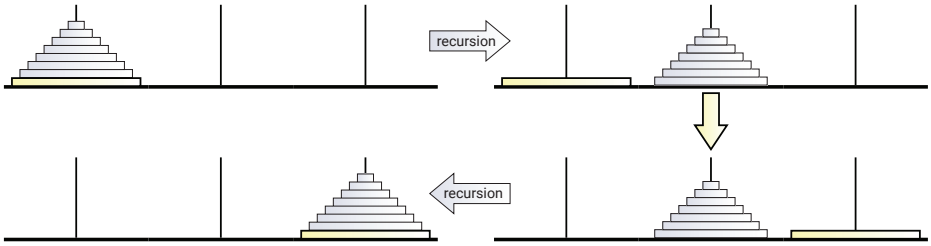

Рис. 1.2. Алгоритм решения головоломки «Ханойская башня» – сосредоточимся на самом нижнем диске, на остальные не обращаем внимания

Теперь, все, что нам нужно, – понять, как…

НЕТ! ПОСТОЙТЕ!

Вот и все. Решение найдено. Мы успешно свели задачу «Ханойская башня» с n дисками к двум частным случаям задачи «Ханойская башня» с $(n - 1)$ дисками, которую можно с ликованием передать Фее Рекурсии или в соответствии с легендой Люка поручить младшим монахам в храме. Наша часть работы завершена. Если бы мы не доверяли младшим монахам, то зачем было привлекать их к этой работе, так что позвольте им спокойно выполнять их обязанности.

Описанное здесь сведе́ние использует одно не слишком заметное, но чрезвычайно важное предположение: самый большой диск действительно существует. Показанный алгоритм работает для любого положительного числа дисков, но он отказывается работать при $n = 0$. Этот случай обязательно необходимо обработать с применением другого метода. К счастью, монахи в Бенаресе, будучи истинными буддистами, весьма компетентны в плане перемещения нуля дисков со стержня на стержень практически без затрат времени – они просто ничего не делают в этом случае.

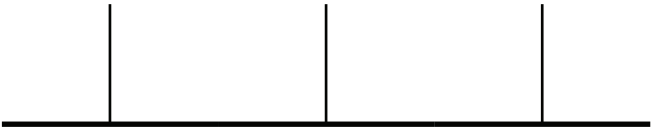

Рис. 1.3. Вырожденный частный случай алгоритма «Ханойская башня». «Ложки нет» (© к/ф «Матрица»)

Может возникнуть искушение задуматься о том, как перемещать все эти диски по стержням, или более обобщенно – что происходит, когда рекурсия разворачивается, – но в действительности не следует думать об этом. Для большинства рекурсивных алгоритмов (мысленное) развертывание процесса рекурсии не является необходимостью и не приносит никакой пользы. Единственная наша обязанность – свести частный случай поставленной задачи к одному или нескольким еще более простым частным случаям этой задачи или решить задачу напрямую, если такое сведе́ние невозможно. Предложенный здесь алгоритм решения «Ханойской башни» тривиально корректен при $n = 0$. При любом $n \geqslant 1$ Фея Рекурсия корректно перемещает $n - 1$ верхних дисков (более формально: по индуктивному предположению принимается, что наш рекурсивный алгоритм корректно перемещает $n - 1$ верхних дисков), таким образом, этот алгоритм корректен.

Рекурсивный алгоритм `Hanoi`, записанный на псевдокоде, показан на рис. 1.4. Алгоритм перемещает стопку из n дисков с исходного стержня-источника (`src`) на целевой стержень (`dst`), используя третий временный стержень (`tmp`) как место временного размещения дисков. Следует отметить, что алгоритм корректно ничего не делает при $n = 0$.

```
Hanoi(n, src, dst, tmp):
if n > 0
  Hanoi(n - 1, src, tmp, dst)     ⟨⟨Рекурсия!⟩⟩
  move disk n from src to dst
  Hanoi(n - 1, tmp, dst, src)     ⟨⟨Рекурсия!⟩⟩
```

Рис. 1.4. Рекурсивный алгоритм для решения задачи «Ханойская башня»

Пусть $T(n)$ обозначает количество ходов, требуемых для передвижения n дисков, – время работы нашего алгоритма. В вырожденном частном случае подразумевается, что $T(0) = 0$, а в более общем рекурсивном алгоритме предполагается, что $T(n) = 2T(n - 1) + 1$ при любом $n \geqslant 1$. Выписав первые несколько значений $T(n)$, мы можем легко догадаться, что $T(n) = 2^n - 1$; прямое индуктивное доказательство подтверждает правильность этого предположения. В частности, перемещение башни из 64 дисков требует $2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615$ отдельных ходов. Таким образом, даже при невероятной скорости выполнения одного хода в секунду монахам в храме Бенареса придется работать приблизительно 585 млрд. лет («plus de cinq milliards de siècles» – «более пяти миллиардов веков»), прежде чем башня, храм и сами монахи-брамины обратятся в прах и с раскатом грома мир исчезнет.

1.4. Сортировка слиянием

Сортировка слиянием (Mergesort) – один из самых первых алгоритмов, предназначенных для компьютеров общего назначения с возможностью хранения программ. Этот алгоритм был разработан Джоном фон Нейманом (John von Neumann) в 1945 г. и подробно описан в опубликованной совместно с Германом Голдстайном (Herman Goldstine) работе в 1947 г. как одна из первых нечисловых программ для компьютера EDVAC[5].

1. Разделить входной массив данных на два подмассива приблизительно равного размера.
2. Выполнить рекурсивно сортировку слиянием для каждого подмассива.
3. Выполнить слияние новых отсортированных подмассивов в один отсортированный массив.

[5] Голдстайн и фон Нейман в действительности описали нерекурсивный вариант, который теперь обычно называют восходящей сортировкой слиянием (bottom-up mergesoft). В те времена большие наборы данных сортировались машинами специального назначения, которые в основном производились компанией IBM. Для этих машин применялись перфокарты и использовались варианты двоичной поразрядной (цифровой) сортировки. Фон Нейман (успешно) доказал, что, поскольку компьютер EDVAC был способен выполнять сортировку быстрее, чем специализированные сортировочные машины IBM «без вмешательства человека и без потребности в дополнительном оборудовании», EDVAC представлял собой «универсальную (многоцелевую)» машину, поэтому необходимость в специализированных сортировочных машинах отпала.

Input:	S	O	R	T	I	N	G	E	X	A	M	P	L
Divide:	S	O	R	T	I	N	G	E	X	A	M	P	L
Recurse Left:	I	N	O	R	S	T	G	E	X	A	M	P	L
Recurse Right:	I	N	O	R	S	T	A	E	G	L	M	P	X
Merge:	A	E	G	I	L	M	N	O	P	R	S	T	X

Рис. 1.5. Пример сортировки слиянием

Первый шаг абсолютно очевиден – простое разделение массива на два по размеру, – а второй шаг можно передать (делегировать – *delegate*) Фее Рекурсии. Вся настоящая работа выполняется на завершающем шаге слияния. Полное описание этого алгоритма приведено на рис. 1.6. Для того чтобы рекурсивная структура была четко видна, я выделил шаг слияния в отдельную независимую подпрограмму. Алгоритм слияния также является рекурсивным – определение первого элемента выходного массива, затем рекурсивное слияние остальной части входных массивов.

```
MergeSort(A[1..n]):
if n > 1
  m ← ⌊n/2⌋
  MergeSort(A[1..m])          ⟨⟨Рекурсия!⟩⟩
  MergeSort(A[m + 1..n])      ⟨⟨Рекурсия!⟩⟩
  Merge(A[1..n], m)
```

```
Merge(A[1..n], m):
  i ← 1; j ← m + 1
  for k ← 1 to n
    if j > n
      B[k] ← A[i]; i ← i + 1
    else if i > m
      B[k] A[j]; j ← j + 1
    else if A[i] < A[j]
      B[k] ← A[i]; i ← i + 1
    else
      B[k] ← A[j]; j ← j + 1
  for k ← 1 to n
    A[k] ← B[k]
```

Рис. 1.6. Алгоритм сортировки слиянием

Корректность

Для доказательства корректности этого алгоритма дважды воспользуемся помощью нашего старого доброго друга – индукции: сначала для подпрограммы `Merge`, затем для алгоритма более высокого уровня `MergeSort`.

Лемма 1.1. Алгоритм `Merge` корректно выполняет слияние подмассивов $A[1..m]$ и $A[m + 1..n]$, предполагая, что эти подмассивы отсортированы при вводе.

Доказательство. Пусть $A[1..n]$ – любой произвольный массив, а m – любое произвольное целое число, такое что подмассивы $A[1..m]$ и $A[m + 1..n]$ отсортированы. Докажем, что для всех k от 0 до n $n - k - 1$ последних ите-

раций основного цикла корректно выполняют слияние $A[i..m]$ и $A[j..n]$ в массив $B[k..n]$. Это доказательство проводится методом (математической) индукции по $n - k + 1$, т. е. по числу оставшихся элементов, для которых требуется слияние.

Если $k > n$, то алгоритм корректно выполняет слияние двух пустых подмассивов, при этом абсолютно ничего не делая. (Это базовый (исходный) случай для индуктивного доказательства.) Иначе существуют четыре варианта, которые необходимо рассмотреть для k-й итерации основного цикла:

- если $j > n$, то подмассив $A[j..n]$ пуст,
  следовательно, $\min(A[i..m] \cup A[j..n]) = A[i]$;
- если $i > m$, то подмассив $A[i..m]$ пуст,
  следовательно, $\min(A[i..m] \cup A[j..n]) = A[j]$;
- иначе если $A[i] < A[j]$, то $\min(A[i..m] \cup A[j..n]) = A[i]$;
- иначе мы обязательно должны получить $A[i] \geqslant A[j]$
  и $\min(A[i..m] \cup A[j..n]) = A[i]$.

Во всех четырех случаях $B[k]$ правильно присваивается наименьший элемент $A[i..m] \cup A[j..n]$). В двух случаях во время присваивания $B[k] \leftarrow A[i]$ Фея Рекурсия корректно выполняет слияние… Извините, я имел в виду, что индуктивное предположение подразумевает, что $n - k$ последних итераций основного цикла правильно выполняет слияние $A[i + 1..m]$ и $A[j..n]$ в массив $B[k + 1..n]$. Аналогично в двух других случаях Фея Рекурсия также корректно выполняет слияние остальных элементов подмассивов. □

Теорема 1.2. Алгоритм `MergeSort` корректно сортирует любой входной массив $A[1..n]$.

Доказательство. Докажем эту теорему методом (математической) индукции по n. Если $n \leqslant 1$, то алгоритм ничего не делает, и это правильно. Иначе Фея Рекурсия корректно сортирует… Еще раз извините, я имел в виду, что индуктивное предположение подразумевает, что данный алгоритм корректно сортирует два меньших подмассива $A[1..m]$ и $A[m + 1..n]$, после чего эти два подмассива корректно объединяются подпрограммой `Merge` (слияние) в один отсортированный массив (по лемме 1.1). □

Анализ

Поскольку алгоритм `MergeSort` является рекурсивным, его время выполнения, естественно, записывается в виде рекуррентного выражения. Для выполнения подпрограммы `Merge`, очевидно, требуется время $O(n)$, потому что это простой цикл `for` с постоянным объемом работы на каждой итерации. Сразу же получаем следующее рекуррентное выражение для алгоритма `MergeSort`:

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n).$$

Как и в большинстве рекуррентных тождеств по принципу «разделяй и властвуй», можно безопасно отбросить минимальное и максимальное предельные значения (используя методику преобразований области (domain transformations), описанную ниже в этой главе), что дает упрощенное рекуррентное тождество $T(n) = 2T(n/2) + O(n)$. Случай «все уровни равнозначны» метода рекурсивного дерева (который также будет описан ниже в этой главе) позволяет немедленно получить аналитическое (в замкнутой форме) решение $T(n) = O(n \log n)$. Даже если вы (пока еще) не знакомы с рекурсивными деревьями, можете проверить правильность решения $T(n) = O(n \log n)$ методом индукции.

1.5. Быстрая сортировка

Быстрая сортировка (quicksort) – еще один рекурсивный алгоритм сортировки, разработанный Тони Хоаром (Tony Hoare) в 1959 г. и впервые опубликованный в 1961 г. В этом алгоритме основная часть работы заключается в разделении массива на подмассивы меньшего размера перед рекурсией, поэтому слияние отсортированных подмассивов является тривиальной задачей.

1. Выбрать опорный элемент (pivot) в исходном массиве.
2. Разделить исходный массив на три подмассива, содержащих элементы, меньшие опорного, сам опорный элемент и элементы, большие опорного.
3. Рекурсивно выполнить алгоритм быстрой сортировки для первого и последнего подмассива.

Input:	S	O	R	T	I	N	G	E	X	A	M	P	L
Choose a pivot:	S	O	R	T	I	N	G	E	X	A	M	P	L
Partition:	A	G	O	E	I	N	L	M	P	T	X	S	R
Recurse Left:	A	E	G	I	L	M	N	O	P	T	X	S	R
Recurse Right:	A	E	G	I	L	M	N	O	P	R	S	T	X

Рис. 1.7. Пример выполнения алгоритма быстрой сортировки

Более подробный псевдокод показан на рис. 1.8. В подпрограмме `Partition` входной параметр p является индексом опорного элемента в неотсортированном массиве. Эта подпрограмма разделяет массив на части и возвращает новый индекс опорного элемента. Существует много разнообразных эффективных алгоритмов разделения, авторство представленного здесь алгоритма приписывается Нико Ломуто (Nico Lomuto)[6]. Переменная `ℓ` является счетчиком элементов массива, меньших (`ℓess`), чем опорный элемент.

[6] Хоар предлагал более сложный «двунаправленный» алгоритм разделения, обладающий некоторыми практическими преимуществами, по сравнению с алгоритмом Ломуто. С другой стороны, алгоритм разделения Хоара – один из тех, в которых ошибки на единицу (ошибки неучтенной единицы – off-by-one errors) приводят к полному краху.

```
QuickSort(A[1..n]):
if n > 1
  Choose a pivot element A[p]
  r ← Partition(A, p)
  QuickSort(A[1..r - 1])    ⟨⟨Рекурсия!⟩⟩
  QuickSort(A[r + 1..n])    ⟨⟨Рекурсия!⟩⟩
```

```
Partition(A[1..n], p):
  swap A[p] ↔ A[n]
  ℓ ← 0       ⟨⟨#items < pivot⟩⟩
  for i ← 1 to n - 1
    if A[i] < A[n]
      ℓ ← ℓ + 1
      swap A[ℓ] ↔ A[i]
  swap A[n] ↔ A[ℓ + 1]
  return ℓ + 1
```

Рис. 1.8. Алгоритм быстрой сортировки Quicksort

Корректность

Как и для алгоритма сортировки слиянием, доказательство корректности алгоритма `QuickSort` требует двух отдельных доказательств методом индукции: одно для доказательства того, что подпрограмма `Partition` правильно разделяет массив, второе для доказательства того, что алгоритм `QuickSort` корректно выполняет сортировку при том, что подпрограмма `Partition` корректна. Для доказательства корректности подпрограммы `Partition` необходимо доказать инвариантность следующего цикла: в конце каждой итерации основного цикла каждый элемент в подмассиве $A[1..\ell]$ меньше $A[n]$ и ни один элемент в подмассиве $A[\ell + 1..n]$ не меньше $A[n]$. Остальные очевидные, но утомительные подробности этого доказательства предлагаются читателю в качестве упражнения для самостоятельного выполнения.

Анализ

Анализ алгоритма быстрой сортировки также похож на анализ алгоритма сортировки слиянием. Ясно, что подпрограмма `Partition` выполняется за время $O(n)$, потому что это простой цикл `for` с константным временем работы на каждой итерации. Для алгоритма `QuickSort` получаем рекуррентное выражение, зависящее от r – позиции (rank) выбранного опорного элемента:

$$T(n) = T(r - 1) + T(n - r) + O(n).$$

Если бы можно было каким-либо способом всегда выбирать опорный элемент так, чтобы он являлся элементом со средним значением (медианой) массива A, то получили бы позицию $r = \lceil n/2 \rceil$, и две подзадачи были бы близки по размеру друг к другу, насколько это возможно. При этом рекуррентное выражение приобрело бы следующий вид:

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

и в результате $T(n) = O(n \log n)$ при использовании либо метода рекурсивного дерева, либо еще более простого метода «О да, мы уже решили это рекуррентное выражение для алгоритма сортировки слиянием».

В сущности, как мы увидим немного позже в этой главе, в действительности можно разместить средний (по значению) элемент в неотсортированном массиве за линейное время, но этот алгоритм достаточно сложен, и скрытая постоянная в нотации $O(\cdot)$ достаточно велика для того, чтобы сделать итоговый алгоритм сортировки почти неприменимым. В практической деятельности большинство программистов применяет упрощенные способы, например выбор первого или последнего элемента массива. В этом случае позиция r может принимать любое значение между 1 и n, поэтому получаем:

$$T(n) = \max_{1 \leqslant r \leqslant n} (T(r-1) + T(n-r) + O(n)).$$

В наихудшем случае эти две подзадачи являются абсолютно несбалансированными как при $r = 1$, так и при $r = n$, и рекуррентное выражение приобретает следующий вид: $T(n) \leqslant T(n-1) + O(n)$.

Решением является $T(n) = O(n^2)$.

Другой общеизвестный эвристический алгоритм называется «среднее из трех (значений)» (median of three) – выбираются три элемента (обычно в начале, в середине и в конце массива), и среднее значение этих трех элементов принимается как опорный элемент. Хотя этот эвристический алгоритм на практике несколько более эффективен, чем простой выбор одного элемента, особенно если массив уже (почти) отсортирован, тем не менее мы можем получить не самое удачное значение $r = 2$ или $r = n - 1$ в наихудшем случае. При использовании эвристического алгоритма «среднее из трех» рекуррентное выражение принимает вид $T(n) \leqslant T(1) + T(n-2) + O(n)$, решением которого остается $T(n) = O(n^2)$.

На интуитивном уровне понятно, что опорный элемент «обычно» должен находиться где-то в середине массива, например в диапазоне между $n/10$ и $9n/10$. Это наблюдение позволяет предположить, что «в среднем» время выполнения алгоритма должно составлять $O(n \log n)$. Несмотря на то что это интуитивное предположение можно формализовать, большинство наиболее распространенных методов формализации принимает абсолютно нереалистичное предположение о том, что все перестановки в исходном массиве почти равнозначны. Данные в реальном мире могут быть случайными (случайно распределенными), но это совсем не та случайность, которую можно предсказать заранее, а кроме того, данные определенно не являются равномерно распределенными[7].

Кроме того, иногда по каким-то причинам принимается время выполнения «в наилучшем случае». Мы так делать не будем.

[7] С другой стороны, если индекс опорного элемента p единообразно выбирается случайным образом, то алгоритм `QuickSort` выполняется за время $O(n \log n)$ с высокой вероятностью для каждого возможного входного массива. Главное различие состоит в том, что эта случайность управляется самим алгоритмом, а не каким-то всемогущим злоумышленником, который передает входные данные после чтения нашего кода. К сожалению, анализ алгоритма быстрой сортировки со случайным выбором опорного элемента не относится к теме этой книги, но вы можете найти конспекты лекций по этой теме здесь: **http://algorithms.wtf/**.

данных представлена последовательность обхода его узлов в прямом порядке и упорядоченного обхода (обхода с порядковой выборкой – inorder).

(d) Описать и проанализировать рекурсивный алгоритм восстановления произвольного двоичного дерева поиска (binary search tree), если в качестве входных данных представлена только последовательность обхода его узлов в прямом порядке.

♥(e) Описать и проанализировать рекурсивный алгоритм восстановления произвольного двоичного дерева поиска, если в качестве входных данных представлена только последовательность обхода его узлов в прямом порядке за время $O(n)$.

В упражнениях (b)–(e) принимается, что все ключи (значения узлов) различны и что входные данные логически согласованы по крайней мере с одним двоичным деревом.

40. Предположим, что существует n точек, произвольно размещенных внутри двумерной прямоугольной области. *kd-дерево* (kd-tree)[21] рекурсивно подразделяет эти точки описанным ниже способом. Если внутри прямоугольной области нет точек, то задача решена. Иначе область делится вертикальной прямой на две прямоугольные области меньшего размера, при этом средняя точка внутри области (но не на ее границе) обеспечивает разделение точек настолько равномерно, насколько это возможно. Далее рекурсивно формируется kd-дерево для точек в каждой из двух меньших областей после поворота их на 90°. Таким образом, на каждом уровне рекурсии чередуется разделение по вертикали и по горизонтали. Окончательным результатом являются пустые прямоугольные области, называемые ячейками (cells).

(a) Сколько получается ячеек – определить их количество как функцию от n. Доказать правильность вашего ответа.

(b) Сколько в точности ячеек может пересечь горизонтальная прямая в наихудшем случае (как функция от n)? Доказать правильность вашего ответа. Допустим, что $n = 2^k - 1$ для некоторого целого числа k. [*Подсказка*: существует более одной функции f, такой что $f(16) = 4$.]

[21] Термин *kd-tree* (произносится [kay dee tree]) как аббревиатура происходит от выражения «k-dimensional tree» (*k*-мерное дерево), но в современном толковании его этимология игнорируется, отчасти потому что никто в здравом уме никогда не должен использовать букву *k* для обозначения размерности (dimension) вместо явно доминирующей в этом плане буквы *d*. Этимологическая согласованность потребовала бы назвать структуру данных в этой задаче «2d-tree» (или, возможно, «2-d tree»), но в настоящее время стандартным обозначением является «two-dimensional kd-tree» (двумерное kd-дерево). Смотрите также этимологию: B-дерево (B-tree) (возможно), альфа-форма (alpha shape), бета-остов (beta skeleton), эпсилон-сеть (epsilon net), р. Потомак (Potomac River), р. Миссисипи (Mississippi River), оз. Мичиган (Michigan Lake), оз. Тахо (Tahoe Lake), о-в Манхэттен (Manhattan Island), Смоляные ямы Ла Бреа (La Brea Tar Pits), пустыня Сахара (Sahara Desert), гора Килиманджаро (Mount Kilimanjaro), Южный Вьетнам (South Vietnam), Восточный Тимор (East Timor), галактика Млечный Путь (Milky Way Galaxy), г. Таунсвилл (City of Townsville) и беспилотные автомобили (self-driving automobiles).

(c) Предположим, что задано n точек, хранящихся в kd-дереве. Описать и проанализировать алгоритм, который подсчитывает количество точек над горизонтальной прямой (например, над штриховой линией на рис. 1.33) настолько быстро, насколько это возможно. [*Подсказка*: используйте упражнение (b).]

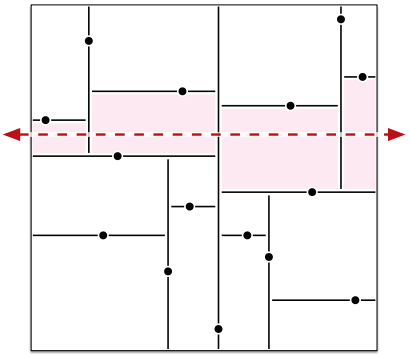

Рис. 1.33. Kd-дерево для 15 точек. Штриховая линия пересекает четыре закрашенные ячейки

(d) Описать и проанализировать эффективный алгоритм, который подсчитывает с учетом kd-дерева, содержащего n точек, количество точек, лежащих внутри прямоугольника R с горизонтальными и вертикальными сторонами. [*Подсказка*: используйте упражнение (c).]

♥41. Боб Ратенбур (Bob Ratenbur), студент начального курса по специальности CS 225, пытается написать код для выполнения обходов в прямом и обратном направлениях (с предварительной и с отложенной выборкой) и упорядоченного обхода (с прямой выборкой) двоичных деревьев. Боб вроде бы понимает основную идею алгоритмов обхода, но когда он пытается реализовать их практически, то неизбежно путается в рекурсивных вызовах. За пять минут до срока сдачи задания Боб в отчаянной спешке выдает исходный код со структурой, показанной ниже:

```
PreOrder(v):
  if v = Null
    return
  else
    print label(v)
    ████Order(left(v))
    ████Order(right(v))
```

```
PreOrder(v):
  if v = Null
    return
  else
    print label(v)
    ████Order(left(v))
    ████Order(right(v))
```

```
PostOrder(v):
  if v = Null
    return
  else
    ████Order(left(v))
    ████Order(right(v))
    print label(v)
```

В этом псевдокоде каждый блок символов ████ скрывает один из префиксов `Pre`, `In` или `Post`. Кроме того, каждый вызов перечислен-

ных ниже функций в коде, представленном Бобом, появляется ровно один раз:

```
PreOrder(left(v))          PreOrder(right(v))
InOreder(left(v))          InOrder(right(v))
PostOrder(left(v))         PostOrder(right(v))
```

Таким образом, существует в точности 36 возможных вариантов кода, написанного Бобом. К сожалению, Боб случайно удалил весь исходный код после успешной сдачи выполняемого файла, поэтому ни вам, ни ему неизвестно, какие именно функции вызывались в скрытых фрагментах кода.

Теперь предположим, что нам предоставлен вывод алгоритмов обхода Боба, выполненных для некоторого неизвестного двоичного дерева T. Выходные данные были предусмотрительно проанализированы и разделены на три массива $Pre[1..n]$, $In[1..n]$ и $Post[1..n]$. Можно предположить, что эти последовательности обхода согласованы ровно с одним двоичным деревом T, в частности, метки вершин этого неизвестного дерева T различны, и каждый внутренний узел в T имеет ровно двух потомков.

(a) Описать алгоритм восстановления неизвестного дерева T по заданным последовательностям обхода.

(b) Описать алгоритм, который либо восстанавливает исходный код Боба по заданным последовательностям обхода, либо верно сообщает о том, что эти последовательности обхода согласуются с несколькими группами алгоритмов.

Например, если заданы следующие входные данные:

```
Pre[1..n] = [H A E C B I F G D]
 In[1..n] = [A H D C E I F B G]
Post[1..n] = [A E I B F C D G H]
```

то ваш первый (a) алгоритм должен возвращать дерево, показанное на рис. 1.34.

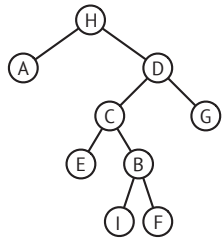

Рис. 1.34. Дерево T, восстановленное по заданным последовательностям обхода

Ваш второй (b) алгоритм должен восстановить следующий исходный код:

```
PreOrder(v):
  if v = Null
    return
  else
    print label(v)
    PreOrder(left(v))
    PostOrder(right(v))
```

```
InOrder(v):
  if v = Null
    return
  else
    PostOrder(left(v))
    print label(v)
    PreOrder(right(v))
```

```
PostOrder(v):
  if v = Null
    return
  else
    InOrder(left(v))
    InOrder(right(v))
    print label(v)
```

♥42. Пусть T – двоичное дерево, в узлах которого хранятся различные числовые значения. Напомню, что T является двоичным деревом поиска, если и только если (1) либо дерево T пустое, (2) либо дерево T соответствует перечисленным ниже рекурсивным условиям:

- левое поддерево дерева T является двоичным деревом поиска;
- все значения в левом поддереве меньше, чем значение в корне;
- правое поддерево дерева T является двоичным деревом поиска;
- все значения в правом поддереве меньше, чем значение в корне.

Рассмотрим следующую пару операций в двоичных деревьях:

- *поворот* (rotate) произвольного узла вверх[22];

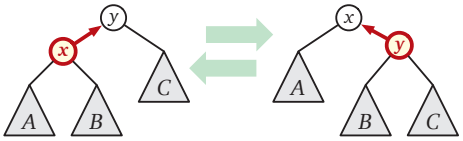

Рис. 1.35. Операция поворота узла в двоичном дереве

- *взаимная перестановка* (swap) левого и правого поддерева произвольного узла.

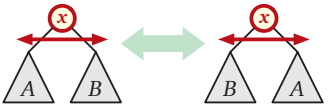

Рис. 1.36. Операция взаимной перестановки поддеревьев узла

В обеих операциях одно, все или ни одно из поддеревьев A, B, C могут быть пустыми.

(a) Описать алгоритм преобразования произвольного двоичного дерева с n узлами и с различными значениями узлов в двоичное

[22] Повороты (rotations) сохраняют упорядоченную (inorder) последовательность обхода узлов в двоичном дереве. Отчасти по этой причине повороты используются для сопровождения некоторых типов сбалансированных двоичных деревьев поиска, включая AVL-деревья, красно-черные деревья, косые деревья, деревья «козла отпущения» и дучи (treaps). Конспекты лекций по большинству этих структур данных см. здесь: **http://algorithms.wtf**.

дерево поиска, использующий не более $O(n^2)$ операций поворота и взаимной перестановки. На рис. 1.37 показана последовательность из восьми операций, преобразующая двоичное дерево с пятью узлами в двоичное дерево поиска.

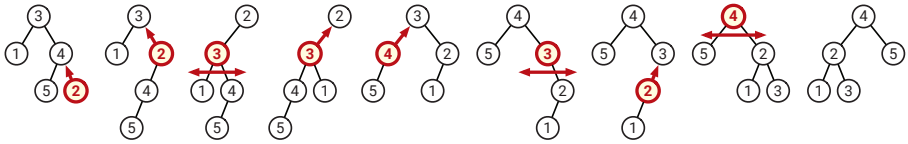

Рис. 1.37. «Сортировка» двоичного дерева: поворот узла 2, поворот узла 2, взаимная перестановка в узле 3, поворот узла 3, поворот узла 4, взаимная перестановка в узле 3, поворот узла 2, взаимная перестановка в узле 4

Этому алгоритму запрещено напрямую изменять указатели на предков или потомков, создавать новые узлы или удалять существующие узлы, единственный способ изменения дерева – выполнение операций поворота и взаимной перестановки.

С другой стороны, вы можете без ограничений выполнять любые вычисления при условии, что они не изменяют дерево, а время выполнения вашего алгоритма определяется числом выполняемых им операций поворота и взаимной перестановки.

♥(b) Описать алгоритм преобразования произвольного двоичного дерева с n узлами в двоичное дерево поиска, использующий не более $O(n \log n)$ операций поворота и взаимной перестановки.

(c) Доказать, что любое двоичное дерево поиска с n узлами можно преобразовать в любое другое двоичное дерево поиска с теми же значениями узлов, используя только лишь $O(n)$ операций поворота (но без операций взаимной перестановки).

♥(d) Нерешенная (открытая) задача: описать алгоритм преобразования произвольного двоичного дерева с n узлами в двоичное дерево поиска, использующий только лишь $O(n)$ операций поворота и взаимной перестановки, или доказать, что такой алгоритм невозможен. [*Подсказка*: я полагаю, что такой алгоритм невозможен.]