

УДК 519.6

ББК 22.19

Г 61

Головешкин В. А., Ульянов М. В. **Теория рекурсии для программистов.** — М.: ФИЗМАТЛИТ, 2006. — 296 с. — ISBN 5-9221-0721-6.

Книга является учебным пособием по теории рекурсии в аспекте ее применения в области программирования. В ней рассматриваются основы теории рекурсии и ее использование в области разработки и анализа рекурсивных алгоритмов. Приводятся основные сведения о рекурсивных последовательностях и функциях, даны примеры рекурсивных алгоритмов, разработанных на основе рекуррентных соотношений, метода декомпозиции и метода динамического программирования, излагаются методы разработки рекурсивных алгоритмов и их теоретического анализа, в том числе элементы теории ресурсной эффективности вычислительных алгоритмов. Детально изложены методы анализа рекурсивных алгоритмов, проиллюстрированные целым рядом примеров. Приложение содержит тексты программ, реализующих рекурсивные алгоритмы, рассмотренные в основном тексте книги, и результаты экспериментальных исследований. Учебное пособие ориентировано на специалистов в области информатики и анализа алгоритмов, разработчиков алгоритмического обеспечения и предназначено для студентов, аспирантов и преподавателей вузов, специализирующихся в области математической информатики, теории рекурсии, разработки, анализа и исследования рекурсивных алгоритмов.

© ФИЗМАТЛИТ, 2006

© В. А. Головешкин, М. В. Ульянов, 2006

Scan, OCR & DjVu - Kartfun [Kartfun@yandex.ru]

ISBN 5-9221-0721-6

ОГЛАВЛЕНИЕ

Предисловие	5
Введение	7
Глава 1. Введение в теорию рекурсии	11
§ 1. Основные понятия и определения	11
§ 2. Рекурсивно заданные последовательности и функции	23
§ 3. Классификация рекурсивно заданных последовательностей и функций	30
§ 4. Методы исследования и решения рекуррентных соотношений	36
Задачи и упражнения к главе 1	50
Глава 2. Рекурсивные алгоритмы и особенности их программных реализаций	51
§ 1. Рекурсивные алгоритмы	51
§ 2. Особенности программных реализаций рекурсивных алгоритмов	56
§ 3. Механизм обслуживания рекурсивного вызова	59
§ 4. Представление последовательности рекурсивных вызовов в виде дерева рекурсии	62
Задачи и упражнения к главе 2	65
Глава 3. Методы разработки рекурсивных алгоритмов	67
§ 1. Метод рекуррентных соотношений	68
§ 2. Метод декомпозиции	71
§ 3. Метод динамического программирования	74
Задачи и упражнения к главе 3	79
Глава 4. Элементы теории ресурсной эффективности вычислительных алгоритмов	82
§ 1. Терминология и обозначения в теории ресурсной эффективности вычислительных алгоритмов	83
§ 2. Функции ресурсной эффективности алгоритмов и их программных реализаций	87
§ 3. Классы открытых и закрытых задач и теоретическая нижняя граница временной сложности	97
§ 4. Классификации вычислительных алгоритмов по трудоемкости	101
§ 5. Информационная и размерностная чувствительность вычислительных алгоритмов	112

§ 6. Классификация вычислительных алгоритмов по дополнительной памяти	123
Глава 5. Специальные главы теории рекурсии	
§ 1. Основная теорема о рекуррентных соотношениях и некоторые особые случаи	129
§ 2. Производящие функции	133
§ 3. Методы исчисления конечных сумм	152
§ 4. Функция $\beta_1(n)$ и другие специальные функции	163
§ 5. Комбинаторные соотношения и их связь с рекурсивными алгоритмами	166
Задачи и упражнения к главе 5	170
Глава 6. Методы теоретического анализа ресурсной эффективности рекурсивных алгоритмов	
§ 1. Базовые операции процедурного языка высокого уровня и методика анализа основных алгоритмических конструкций	172
§ 2. Особенности анализа временной и емкостной эффективности рекурсивных алгоритмов	178
§ 3. Анализ трудоемкости методом подсчета вершин дерева рекурсии	181
§ 4. Анализ трудоемкости методом рекуррентных соотношений	185
§ 5. Способы повышения ресурсной эффективности рекурсивных алгоритмов	187
Задачи и упражнения к главе 6	190
Глава 7. Рекурсивные алгоритмы решения некоторых задач и их теоретический анализ	
§ 1. Алгоритм вычисления факториала	192
§ 2. Алгоритм вычисления чисел Фибоначчи	195
§ 3. Алгоритм вычисления квадратного корня	198
§ 4. Алгоритм быстрого возведения числа в целую степень	204
§ 5. Алгоритм Карацубы умножения длинных целых чисел	209
§ 6. Алгоритм фон Неймана сортировки массива чисел слиянием	219
§ 7. Генетический алгоритм эвристического поиска экстремума функций нескольких переменных	228
§ 8. Алгоритм Тарьяна поиска оставного дерева в графе	243
§ 9. Алгоритм Беллмана оптимальной одномерной упаковки	249
Задачи и упражнения к главе 7	262
Приложение А.Д. Брейман, Г.П. Рябов.	
Программные реализации рекурсивных алгоритмов и их экспериментальное исследование	266

ПРЕДИСЛОВИЕ

Когда в мае 2005 года издательство «Техносфера» любезно предложило нам написать дополнение к книге Р. Хаггарти «Дискретная математика для программистов», то, обсуждая оглавление будущего дополнения, мы единодушно включили в него главу, посвященную элементам теории рекурсии. В ходе работы над дополнением был подготовлен довольно обширный материал по применению рекурсии в программировании и оценке рекурсивных алгоритмов, и стало ясно, что у нас есть база для написания отдельной книги. Этот материал и стал одним из стимулов создания «Теории рекурсии для программистов» — книги, которую наш уважаемый читатель держит сейчас в руках.

Основной материал книги базируется на лекционных курсах «Дискретная математика» и «Разработка эффективных алгоритмов», читаемых авторами на протяжении ряда лет в Московском государственном университете приборостроения и информатики (МГУПИ), на материалах регулярного научного семинара «Анализ и разработка эффективных вычислительных алгоритмов» кафедры «Персональные компьютеры и сети» МГУПИ, а также на наших научных публикациях, посвященных исследованию и анализу вычислительных алгоритмов.

Наше понимание содержания «Теории рекурсии для программистов» как совокупности специального раздела современной дискретной математики — математической теории рекурсии, и основанных на ней методов анализа рекурсивных алгоритмов и элементов теории ресурсной эффективности нашло отражение в структуре данной книги. Мы начинаем с математического введения в теорию рекурсии, которому посвящена глава 1, включая частично рекурсивные и общерекурсивные функции, введенные С. Клини и А. Чёрчем, переходим к рекурсивным алгоритмам и методам их разработки в главах 2 и 3, и излагаем элементы теории ресурсной эффективности вычислительных алгоритмов в главе 4. Этот материал является основой для понимания более детального теоретического исследования рекурсии, которому посвящена глава 5, на основе которого в главе 6 излагаются методы теоретического анализа и способы совершенствования рекурсивных алгоритмов.

Мы ставили своей основной целью создание учебного пособия для студентов, преподавателей и программистов, поэтому на разнообразных примерах теоретического анализа рекурсивных алгоритмов, приведенных в главе 7, начиная с очень простых (мы надеемся, что подготовленный читатель нас здесь извинит), мы хотели показать, как эти методы

применимы в практике оценки ресурсной эффективности рекурсивных алгоритмов.

«Теория рекурсии для программистов» — это не руководство по написанию программ, реализующих рекурсивные алгоритмы в различных языках программирования — это изложение элементов теории рекурсии и методов оценки рекурсивных алгоритмов, ставящее своей задачей показать возможности, предоставляемые рекурсией в теории и практике программирования. В этом аспекте авторы уверены, что в современном море программных средств и систем, опирающихся на итерационные алгоритмы, реализованные в рамках процедурного объектно-ориентированного подхода, у рекурсивных алгоритмов есть свои освещённые солнцем заливы.

В процессе работы над книгой мы старались сохранить стиль и дух, присущий книгам издательства «ФИЗМАТЛИТ», целевой аудиторией которых являются, в первую очередь, студенты, аспиранты и преподаватели. Мы надеемся, предлагая эту книгу уважаемым читателям, на то, что наша аудитория, равно как и те, кто занимается реальной разработкой программных средств и систем, найдут в ней некоторые полезные и интересные для себя сведения и результаты.

Наши искренние благодарности — коллективу издательства «ФИЗМАТЛИТ», и лично Л.Ф. Соловейчику, за поддержку и творческое сотрудничество. Особые благодарности мы приносим авторам приложения — нашим коллегам: доцу кафедры «Персональные компьютеры и сети» МГУПИ А.Д. Брейману и аспиранту кафедры Г.П. Рябову, взявшим на себя труд провести серию экспериментов с программными реализациями рекурсивных алгоритмов с целью подтверждения полученных нами теоретических результатов.

В.А. Головешкин,

М.В. Ульянов

Москва, апрель 2006 г.

ВВЕДЕНИЕ

Парадигма рекурсивного программирования. В середине 1930-х гг. Стивен Клини, основываясь на исследованиях Курта Гёделя и Жака Эрбрана, вводит рекурсивные функции для уточнения общего понятия арифметического алгоритма. Предложение считать понятие вычислимой функции равнозначным понятию рекурсивной функции известно под названием тезиса Чёрча-Клини, сформулированного Алонзо Чёрчем и Стивеном Клини в 1936 году, — «любая интуитивно вычислимая функция является частично рекурсивной функцией». Начиная с этих работ, аппарат рекурсивных функций становится одной из фундаментальных основ современной теории алгоритмов. Возможность поддержки механизма рекурсивного вызова в языках высокого уровня, равно как и создание специальных языков, в которых рекурсивные вычисления являются базисом описания вычислительного процесса (Рефал, Лисп и т. д.), перевели рекурсию из области теоретического аппарата исследования общих свойств алгоритмов в область практического программирования. Идеи Р. Беллмана и его «основное функциональное уравнение», на наш взгляд, также немало способствовали популяризации рекурсии как парадигмы программирования. В математике и программировании рекурсия — это метод определения или выражения функции или процедуры посредством той же функции или процедуры. Рекурсия как парадигма программирования — это метод создания программ, использующий программные конструкции функций или процедур, которые вызывают сами себя либо непосредственно, либо косвенно. Рекурсия естественным образом встречается во всех областях машинной математики. Строгое описание идентификаторов и имен в языках программирования, записанное в нотации Бэкуса-Наура, содержит явные рекурсивные определения. На принципах рекурсии строятся формальные грамматики, описывающие сами языки программирования. Многие структуры данных, например, деревья, также допускают простое рекурсивное описание.

Те, кто впервые сталкивается с программированием, обычно очень настороженно относятся к рекурсии. Эта настороженность объясняется тем, что изначально механизм рекурсивного вызова не является очевидным, и возникает интуитивная опасность бесконечного цикла или аварийного завершения по обращению к адресу памяти за допустимыми границами. В противовес этому мнению мы считаем, что парадигма рекурсивного программирования предоставляет разработчикам в целом

ряде случаев простой, логичный и изящный инструмент создания программ, имеющий ряд преимуществ.

Преимущества рекурсивного подхода. Что реально может получить программист, используя рекурсивные функции и процедуры при разработке своих программ? Мы не пытаемся окружить рекурсию розовым облаком, но, тем не менее, в определенных случаях ее применение может принести значительную пользу. Мы хотим остановиться на нескольких аргументах в пользу рекурсии, а именно таких.

1. Существует достаточно обширный круг задач, которые имеют рекурсивную природу, а ряд методов разработки алгоритмов, в частности, те, о которых пойдет речь в настоящей книге, естественно порождают рекурсивные алгоритмы. Метод динамического программирования рекурсивен по своей сути; метод декомпозиции, предполагающий разбиение задачи на несколько аналогичных задач меньшей размерности, с последующим объединением решений, также порождает алгоритмы с очевидной рекурсивной структурой.

2. Целый ряд структур данных и многие объекты современных языков программирования рекурсивны по самой своей сути. Повторение целого в его части — основное свойство фрактальных объектов. Иерархия классов в объектно-ориентированном программировании, различные древовидные регулярные структуры данных имеют простое рекурсивное описание. Программы для работы с такими структурами выглядят намного более естественно в рекурсивной реализации.

3. Рекурсия в программировании является аналогом метода математической индукции в математике. На наш взгляд, оба метода обладают как внутренним изяществом и простотой, так и широкой областью применения.

4. Рекурсивно реализованные алгоритмы, при правильных на то основаниях, имеют лаконичную запись и менее трудоёмки при последующей отладке и модификации. Сегодня такой фактор, как временные затраты на разработку, отладку и модификацию программных средств также может быть использован в качестве аргумента в пользу рекурсии.

5. И, наконец, аргумент с теоретическим подтекстом — если аппарат частично- и общерекурсивных функций является одним из базисов современной теории алгоритмов, то почему сами алгоритмы не должны быть рекурсивными?

Контраргументы. Несмотря на перечисленные аргументы, рекурсия в настоящее время отнюдь не является обыденным средством разработки алгоритмов и программ для многих программистов, даже профессиональных. Обычно в качестве контраргументов выдвигаются следующие положения, в основе которых лежат интуитивные оценки ресурсной эффективности программных реализаций.

1. Рекурсивно реализованные алгоритмы обладают худшей времененной эффективностью. Да, конечно, обслуживание рекурсивных вы-

зовов влечет определенные накладные расходы, но при этом рекурсивные алгоритмы, разработанные, например, методом декомпозиции, имеют лучшие асимптотические оценки. Это означает, что, начиная с некоторой длины входа, достаточно часто соответствующей области практического использования, программная реализация рекурсивного алгоритма будет иметь лучшие временные показатели.

2. Емкостная эффективность рекурсивных реализаций хуже, чем итерационных. Да, рекурсивно реализованный алгоритм требует больше памяти за счет использования области программного стека, но эти затраты определяются глубиной рекурсии. Для большинства рекурсивных алгоритмов и практически значимых размерностей задач эта глубина, при современном объеме оперативной памяти компьютеров, не вносит существенного вклада в общий требуемый объем памяти.

Рекурсия в сравнении с итерацией. В соответствии с рядом тезисов теории алгоритмов (тезис Чёрча-Клини, тезис Чёрча Тьюринга) любой алгоритм, который можно реализовать в виде машины Тьюринга, может быть описан аппаратом рекурсивных функций. На практике этот результат означает, что рекурсия и итерация как способы разработки алгоритмов и программ эквивалентны. Рассмотрение этого вопроса с точки зрения теории ресурсной эффективности показывает, что, в зависимости от задачи и диапазона длин входов, более эффективной является или итерационная или рекурсивная реализация. В ряде случаев, особенно в программных реализациях алгоритмов, основанных на методе динамического программирования, эффективное решение состоит в построении комбинированного алгоритма, совмещающего рекурсию и итерацию.

Мы не можем предложить уважаемому читателю однозначных решений — если Вы придерживаетесь интуитивного подхода, то решение по выбору рекурсивного или итерационного подхода лежит в области Вашего понимания «рекурсивности» самого алгоритма или обрабатываемых им объектов, если Вы — сторонник математического обоснования выбора между рекурсией и итерацией — тогда в Вашем распоряжении современные методы анализа алгоритмов.

Обоснование выбора рационального алгоритма. Наша точка зрения состоит в том, что рекурсия — это естественный метод реализации алгоритмов для решения множества математических и вычислительных задач, и решение о выборе рекурсивной или итерационной реализации необходимо принимать на основе анализа их ресурсной эффективности. Современные математические методы анализа рекурсивных алгоритмов базируются как на методах решения рекуррентных соотношений, так и на методах асимптотической оценки рекурсивно заданных функций. Их применение позволяет получить теоретические оценки временной эффективности рекурсивных алгоритмов. Полученные результаты, например, для алгоритмов, основанных на методе декомпозиции, показывают целесообразность применения рекурсии для

решения целого ряда задач, по крайней мере, по критерию асимптотической оценки временной эффективности. Если мы при этом имеем в своем распоряжении методы детальной оценки ресурсной эффективности, то появляется возможность решать задачу рационального выбора вычислительных алгоритмов в реальном диапазоне длин входов. Вполне возможно, что такое решение будет принято в пользу рекурсивного, а не итерационного алгоритма.

Что такое рекурсия в математике и программировании, каким образом можно разрабатывать рекурсивные алгоритмы и как можно анализировать их ресурсную эффективность — на эти вопросы мы и пытаемся ответить в настоящей книге.

Г л а в а 1

ВВЕДЕНИЕ В ТЕОРИЮ РЕКУРСИИ

Введение. Мы начинаем изложение материала книги с введения в теорию рекурсии. Рассматривая ряд примеров, приводящих к рекурсивно заданным последовательностям и функциям, мы вводим основные понятия и определения общей теории рекурсии и указываем связь классической теории алгоритмов с понятием частично рекурсивной функции. Более детальное рассмотрение особенностей задания рекурсивных последовательностей и функций позволяет дать их классификацию, на основе которой мы приводим некоторые методы исследования и решения рекуррентных соотношений. Этот материал является базовым для изложения дальнейшего материала книги, и мы советуем читателям не только внимательно ознакомиться с содержанием этой главы, но и решить предлагаемые задачи и выполнить упражнения, приведенные в конце этой главы.

§ 1. Основные понятия и определения

Прежде чем переходить к основным понятиям и определениям общей теории рекурсии, рассмотрим некоторые примеры.

Пример 1.1. Эта задача, которая может показаться читателю немного странной, состоит в том, что, предположим, нам требуется решить на компьютере следующее уравнение: $0,7x = 2$, но наш компьютер таков, что он умеет складывать, вычитать, умножать и сравнивать, но не умеет делить. Предложим алгоритм, который позволит этому странному компьютеру решить данное уравнение (говоря более высоким стилем, мы покажем, что данная задача разрешима в предложенной модели вычислений).

Перепишем уравнение в виде $x = 0,3x + 2$, зададим значение $x_1 = 1$ и рассмотрим последовательность значений x_n , определенных соотношениями

$$\begin{cases} x_1 = 1; \\ x_{n+1} = 0,3x_n + 2. \end{cases} \quad (1.1.1)$$

Мы получили рекурсивно заданную последовательность, в которой следующее значение определяется на основе предыдущего. Изучим свойства этой последовательности. Обозначим $|x_{n+1} - x_n|$ через r_n ,

а поскольку по формуле (1.1.1) $x_{n+1} = 0,3x_n + 2$, и $x_{n+2} = 0,3x_{n+1} + 2$, то имеем

$$x_{n+2} - x_{n+1} = 0,3(x_{n+1} - x_n),$$

следовательно, $r_{n+1} = 0,3r_n$. Покажем, что последовательность x_n ограничена для всех значений n . Так как $x_1 = 1$, то $x_2 = 2,3$, тогда при $n > 2$ получаем

$$|x_n - x_1| = |(x_n - x_{n-1}) + (x_{n-1} - x_{n-2}) + \dots + (x_3 - x_2) + (x_2 - x_1)|.$$

Воспользовавшись свойством модуля, мы можем записать неравенство

$$|x_n - x_1| \leq |x_n - x_{n-1}| + |x_{n-1} - x_{n-2}| + \dots + |x_3 - x_2| + |x_2 - x_1|,$$

следовательно,

$$|x_n - x_1| \leq r_{n-1} + r_{n-2} + \dots + r_2 + r_1,$$

а поскольку $r_{n+1} = 0,3r_n$, то

$$|x_n - x_1| \leq 0,3^{n-2}r_1 + 0,3^{n-3}r_1 + \dots + 0,3r_1 + r_1.$$

Используя формулу суммы геометрической прогрессии, получаем

$$|x_n - x_1| \leq r_1 \frac{1 - 0,3^{n-1}}{1 - 0,3},$$

таким образом доказана ограниченность последовательности x_n .

Далее докажем, что данная последовательность имеет предел. Для этого обозначим через $d_{mn} = |x_n - x_m|$, и, для определенности, будем полагать, что $n > m$. Аналогично только что проделанным рассуждениям получаем следующее неравенство

$$d_{mn} \leq r_{n-1} + r_{n-2} + \dots + r_{m+1} + r_m,$$

следовательно,

$$d_{mn} \leq 0,3^{n-2}r_1 + 0,3^{n-3}r_1 + \dots + 0,3^m r_1 + 0,3^{m-1}r_1 =$$

$$= 0,3^{m-1}r_1 \frac{1 - 0,3^{n-m}}{1 - 0,3},$$

тогда для любого $\varepsilon > 0$ найдется число $n_0 = n_0(\varepsilon)$ такое, что при любых $n > n_0$, $m > n_0$ будет выполнено $d_{mn} < \varepsilon$. Из свойства полноты множества действительных чисел следует, что последовательность x_n имеет предел при n , стремящемся к бесконечности. Обозначим $\lim_{n \rightarrow \infty} x_n = z$ и перейдем к пределу в соотношении $x_{n+1} = 0,3x_n + 2$, имеем $\lim_{n \rightarrow \infty} x_{n+1} = \lim_{n \rightarrow \infty} (0,3x_n + 2)$, тогда получаем $z = 0,3z + 2$, следовательно, значение предела и является корнем уравнения. Таким образом, алгоритм, основанный на соотношении (1.1.1), позволяет решить уравнение с любой заданной точностью, не используя операцию деления. Отметим, что в соотношении (1.1.1) мы использовали определение следующего члена последовательности через его предыдущее значение.

Пример 1.2. Необходимо найти \sqrt{a} ($a > 0$), если разрешается использовать только четыре арифметические операции — сложение,

вычитание, умножение, деление и операцию сравнения. Значение \sqrt{a} является положительным корнем квадратного уравнения $x^2 = a$. Преобразуем это уравнение в уравнение $2x^2 = x^2 + a$, которое представим в виде

$$x = \frac{1}{2} \left(x + \frac{a}{x} \right).$$

Рассмотрим последовательность x_n , определяемую соотношениями:

$$\begin{cases} x_1 = 1; \\ x_{n+1} = \frac{1}{2} \left(x_n + \frac{a}{x_n} \right). \end{cases} \quad (1.1.2)$$

Очевидно, что данная последовательность принимает только положительные значения. Для того чтобы исследовать свойства данной последовательности, нам понадобится некоторая информация о свойствах функции

$$y(x) = \frac{1}{2} \left(x + \frac{a}{x} \right),$$

более точно — нас интересует ее производная

$$\frac{dy}{dx} = \frac{1}{2} \left(1 - \frac{a}{x^2} \right).$$

При положительных значениях аргумента производная равна нулю при $x = \sqrt{a}$. Функция $y(x)$ убывает на интервале $(0; \sqrt{a})$ и возрастает на интервале $(\sqrt{a}; \infty)$. Тогда для положительных значений аргумента точка $x = \sqrt{a}$ является точкой минимума, и, следовательно, при всех неотрицательных его значениях имеем

$$y(x) \geq \frac{1}{2} \left(\sqrt{a} + \frac{a}{\sqrt{a}} \right) = \sqrt{a}.$$

Значит, все члены последовательности, кроме, может быть, первого, больше, либо равны \sqrt{a} : $x_n \geq \sqrt{a}$ при $n > 1$. Далее покажем, что все члены последовательности, начиная со второго, не возрастают, т. е. $x_{n+1} \leq x_n$ при $n \geq 2$. Нетрудно заметить, что при $x \geq \sqrt{a}$ выполнено неравенство

$$\frac{1}{2} \left(x + \frac{a}{x} \right) \leq \frac{1}{2} (x + x) = x.$$

Следовательно, члены последовательности не возрастают, начиная со второго. Но из курса математического анализа известно, что не возрастающая ограниченная последовательность имеет предел. Обозначим $\lim_{n \rightarrow \infty} x_n = z$ и, переходя к пределу в соотношении

$$x_{n+1} = \frac{1}{2} \left(x_n + \frac{a}{x_n} \right),$$

получаем, что значение предела является положительным корнем уравнения

$$z = \frac{1}{2} \left(z + \frac{a}{z} \right),$$

который равен $z = \sqrt{a}$. Записанная последовательность, в которой опять следующее значение определено по некоторой формуле через предыдущее, может быть легко преобразована в алгоритм, позволяющий вычислять квадратные корни с нужной степенью точности — мы рассмотрим его подробно в главе 7.

Отметим, что изложенные методы основаны на более широком методе, применяемом в математике и известном под названием «*метод сжимающих отображений*». Сам математический метод опирался на идею построения рекуррентного соотношения, строгий смысл понятия «*рекуррентное соотношение*» мы поясним несколько позже, после рассмотрения нескольких примеров.

Пример 1.3. Требуется вычислить $\int_0^{\pi/2} \cos^n x dx$, где $n \geq 0$ — целое число. Будем рассматривать данный определенный интеграл как последовательность I_n .

Обозначим $I_n = \int_0^{\pi/2} \cos^n x dx$, нетрудно видеть, что

$$I_0 = \int_0^{\pi/2} \cos^0 x dx = \int_0^{\pi/2} dx = \pi/2, \quad I_1 = \int_0^{\pi/2} \cos x dx = \sin x \Big|_0^{\pi/2} = 1.$$

Теперь мы попытаемся найти способ вычисления данного интеграла в общем случае. Для этого воспользуемся известной формулой интегрирования по частям

$$\int_a^b u dv = uv \Big|_a^b - \int_a^b v du,$$

мы надеемся, что большинство читателей помнят эту формулу и не будут в обиде на нас за это напоминание, но на всякий случай мы все же ее приводим. Преобразуем искомый интеграл к виду

$$I_n = \int_0^{\pi/2} \cos^n x dx = \int_0^{\pi/2} \cos^{n-1} x \cdot \cos x dx$$

и обозначим $u = \cos^{n-1} x$, $dv = \cos x dx$, следовательно,

$$du = (\cos^{n-1} x)' dx = -(n-1) \cdot \cos^{n-2} x \cdot \sin x dx,$$

$$v = \int dv = \int \cos x dx = \sin x.$$

Применяя формулу интегрирования по частям, получаем

$$I_n = \int_0^{\pi/2} \cos^{n-1} x d \sin x = \cos^{n-1} x \cdot \sin x \Big|_0^{\frac{\pi}{2}} + (n-1) \int_0^{\pi/2} \cos^{n-2} x \cdot \sin^2 x dx,$$

поскольку $\sin 0 = 0$, $\cos(\pi/2) = 0$, $\sin^2 x = 1 - \cos^2 x$, то полученное выражение преобразуется к виду

$$\begin{aligned} I_n &= (n-1) \int_0^{\pi/2} \cos^{n-2} x \cdot (1 - \cos^2 x) dx = \\ &= (n-1) \int_0^{\pi/2} \cos^{n-2} x dx - (n-1) \int_0^{\pi/2} \cos^n x dx. \end{aligned}$$

Вспомним наши обозначения, так как $I_n = \int_0^{\pi/2} \cos^n x dx$, $I_{n-2} = \int_0^{\pi/2} \cos^{n-2} x dx$, то

$$I_n = (n-1)I_{n-2} - (n-1)I_n, \quad nI_n = (n-1)I_{n-2},$$

следовательно,

$$I_n = \frac{n-1}{n} I_{n-2}. \quad (1.1.3)$$

Зная значения I_0 и I_1 , с помощью формулы (1.1.3) мы можем вычислить за конечное число шагов значение I_n для любых n . Например, требуется вычислить

$$I_5 = \int_0^{\pi/2} \cos^5 x dx,$$

тогда, применяя (1.1.3), получаем

$$I_5 = \frac{4}{5} I_3 = \frac{4}{5} \cdot \frac{2}{3} I_1 = \frac{4}{5} \cdot \frac{2}{3} \cdot 1 = \frac{8}{15}.$$

Мы снова получили метод вычисления, в данном случае — последовательности значений определенного интеграла, в котором, зная несколько предыдущих элементов упорядоченного множества, мы вычисляем следующий.

Приведем еще один пример, когда по подобной схеме вычисляется не числовая последовательность, а функция.

Пример 1.4. Рассмотрим метод нахождения первообразной для функции

$$f(x) = \frac{1}{(x^2 + a^2)^n},$$

где n — целое положительное число. Будем рассматривать первообразную как функцию двух переменных — независимой переменной x и целой переменной n . Поскольку любая первообразная определена с точностью до произвольной постоянной, то произвольную постоянную мы будем опускать. Обозначим через

$$J_n(x) = \int \frac{dx}{(x^2 + a^2)^n}. \quad (1.1.4)$$

Нетрудно видеть, что

$$J_1(x) = \int \frac{dx}{(x^2 + a^2)} = \frac{1}{a} \operatorname{arctg} \frac{x}{a}.$$

Для вычисления первообразной при произвольном n преобразуем подынтегральное выражение. При этом будем использовать широко применяемые в математике приемы: *прибавление нуля и умножение на единицу* (при этом важно «удачно» записать ноль и единицу). Итак, имеем

$$\begin{aligned} J_n(x) &= \int \frac{dx}{(x^2 + a^2)^n} = \frac{1}{a^2} \int \frac{(x^2 + a^2 - x^2)dx}{(x^2 + a^2)^n} = \\ &= \frac{1}{a^2} \int \frac{(x^2 + a^2)dx}{(x^2 + a^2)^n} - \frac{1}{a^2} \int \frac{x^2 dx}{(x^2 + a^2)^n}. \end{aligned}$$

Это выражение может быть преобразовано к виду

$$J_n(x) = \frac{1}{a^2} \int \frac{dx}{(x^2 + a^2)^{n-1}} - \frac{1}{a^2} \int x \frac{dx}{(x^2 + a^2)^n}.$$

Первое слагаемое в правой части, согласно принятым обозначениям, равно

$$\frac{1}{a^2} \int \frac{dx}{(x^2 + a^2)^{n-1}} = \frac{1}{a^2} J_{n-1}(x).$$

Для вычисления второго слагаемого воспользуемся формулой интегрирования по частям, для чего обозначим

$$u = x, \quad dv = \frac{x dx}{(x^2 + a^2)^n}, \quad \text{тогда} \quad du = dx, \quad v = \int \frac{x dx}{(x^2 + a^2)^n}.$$

Для вычисления последнего интеграла сделаем замену переменной $z = x^2 + a^2$, тогда $x dx = \frac{1}{2} d(x^2 + a^2) = \frac{1}{2} dz$. Следовательно,

$$v = \int \frac{x dx}{(x^2 + a^2)^n} = \frac{1}{2} \int \frac{dz}{z^n} = -\frac{1}{2(n-1)z^{n-1}} = -\frac{1}{2(n-1)(x^2 + a^2)^{n-1}},$$

получаем

$$\begin{aligned} J_n(x) &= \frac{1}{a^2} J_{n-1}(x) - \\ &\quad - \frac{1}{a^2} \left[-\frac{x}{2(n-1)(x^2 + a^2)^{n-1}} + \frac{1}{2(n-1)} \int \frac{dx}{(x^2 + a^2)^{n-1}} \right]. \end{aligned}$$

Раскрывая скобки и приводя подобные, получаем формулу для вычисления (1.1.4):

$$J_n(x) = \frac{x}{2(n-1)a^2(x^2 + a^2)^{n-1}} + \frac{(2n-3)}{2(n-1)a^2} J_{n-1}(x). \quad (1.1.5)$$

Используя полученную формулу, вычислим, например, значение $\int \frac{dx}{(x^2 + 1)^3}$, при этом $a = 1$, $n = 3$, и на основании (1.1.5) имеем:

$$\begin{aligned} J_3(x) &= \int \frac{dx}{(x^2 + 1)^3} = \frac{x}{2 \cdot (3-1)(x^2 + 1)^{3-1}} + \frac{(2 \cdot 3 - 3)}{2(3-1)a^2} J_{3-1}(x) = \\ &= \frac{x}{4(x^2 + 1)^2} + \frac{3}{4} J_2(x) = \frac{x}{4(x^2 + 1)^2} + \frac{3}{4} \left[\frac{x}{2(x^2 + 1)} + \frac{1}{2} J_1(x) \right] = \\ &= \frac{x}{4(x^2 + 1)^2} + \frac{3}{8} \frac{x}{(x^2 + 1)} + \frac{3}{8} \operatorname{arctg} x. \end{aligned}$$

Пример 1.5. Приведем еще один пример — пример разработки алгоритма вычислений по подобной схеме, в которой параметром является размер массива исходных данных. Рассмотрим задачу вычисления определителя квадратной матрицы. Мы надеемся, что читатели знакомы с классическим курсом линейной алгебры и аналитической геометрии, но на всякий случай напомним некоторые основные моменты. Матрицей \mathbf{A} размера $m \times n$ называется прямоугольная таблица чисел, содержащая m строк и n столбцов, которую будем представлять в виде

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1j} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2j} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ a_{i1} & a_{i2} & \dots & a_{ij} & \dots & a_{in} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mj} & \dots & a_{mn} \end{pmatrix}.$$

Символом a_{ij} обозначается элемент, стоящий на пересечении строки с номером i и столбца с номером j . Если число строк равно числу столбцов, то матрица называется квадратной. Понятие определителя имеет смысл только для квадратной матрицы. Определитель — это некоторое число, которое ставится в соответствие квадратной матрице, оно обозначается символом Δ или $|\mathbf{A}|$. Ниже будет изложено правило его вычисления. Для матрицы \mathbf{A} размерности 1×1 — $\mathbf{A} = (a_{11})$ по определению $|\mathbf{A}| = a_{11}$. Минором M_{ij} матрицы \mathbf{A} называется определитель матрицы, получаемой из матрицы \mathbf{A} путем удаления строки с номером i и столбца с номером j . Например для матрицы

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 5 \\ 4 & 1 & 3 \\ 2 & 3 & 4 \end{pmatrix} \quad M_{21} = \begin{vmatrix} 2 & 5 \\ 3 & 4 \end{vmatrix}.$$

Алгебраическим дополнением элемента a_{ij} называется число, равное значению минора M_{ij} , если значение $i + j$ — четное число, и равное

значению минора с противоположным знаком $(-M_{ij})$ в противном случае. Алгебраическое дополнение будем обозначать символом A_{ij} . Таким образом $A_{ij} = (-1)^{i+j} M_{ij}$. Например для матрицы

$$A = \begin{pmatrix} 1 & 2 & 5 \\ 4 & 1 & 3 \\ 2 & 3 & 4 \end{pmatrix} \quad A_{21} = - \begin{vmatrix} 2 & 5 \\ 3 & 4 \end{vmatrix}.$$

Одно из свойств определителя формулируется следующим образом: *определитель равен сумме произведений элементов любой строки или столбца на их алгебраические дополнения*. Для построения алгоритма вычисления определителя квадратной матрицы для любой размерности мы это свойство сформулируем в следующем виде: *определитель равен сумме произведений элементов ПЕРВОГО столбца на их алгебраические дополнения*. Например, для матрицы второго порядка имеем

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - cb; \quad (1.1.6)$$

для матрицы третьего порядка

$$\begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix} = a_1(-1)^{1+1} \begin{vmatrix} b_2 & c_2 \\ b_3 & c_3 \end{vmatrix} + a_2(-1)^{2+1} \begin{vmatrix} b_1 & c_1 \\ b_3 & c_3 \end{vmatrix} + \\ + a_3(-1)^{3+1} \begin{vmatrix} b_1 & c_1 \\ b_2 & c_2 \end{vmatrix} = a_1 \begin{vmatrix} b_2 & c_2 \\ b_3 & c_3 \end{vmatrix} - a_2 \begin{vmatrix} b_1 & c_1 \\ b_3 & c_3 \end{vmatrix} + a_3 \begin{vmatrix} b_1 & c_1 \\ b_2 & c_2 \end{vmatrix}.$$

Отметим, что вычисление определителя матрицы третьего порядка свелось к вычислению трех определителей матрицы второго порядка. Точно по такой же схеме вычисление определителя матрицы порядка n сводится к вычислению n определителей матриц порядка $n - 1$. Например, для матрицы четвертого порядка

$$\Delta = \begin{vmatrix} 1 & 2 & 3 & 1 \\ 2 & 6 & 4 & 6 \\ 3 & 8 & 5 & 4 \\ 1 & 5 & 6 & 3 \end{vmatrix} = 1(-1)^{1+1} \begin{vmatrix} 6 & 4 & 6 \\ 8 & 5 & 4 \\ 5 & 6 & 3 \end{vmatrix} + 2(-1)^{2+1} \begin{vmatrix} 2 & 3 & 1 \\ 8 & 5 & 4 \\ 5 & 6 & 3 \end{vmatrix} + \\ + 3(-1)^{3+1} \begin{vmatrix} 2 & 3 & 1 \\ 6 & 4 & 6 \\ 5 & 6 & 3 \end{vmatrix} + 1(-1)^{4+1} \begin{vmatrix} 2 & 3 & 1 \\ 6 & 4 & 6 \\ 8 & 5 & 4 \end{vmatrix} = \\ = \left[6(-1)^{1+1} \begin{vmatrix} 5 & 4 \\ 6 & 3 \end{vmatrix} + 8(-1)^{2+1} \begin{vmatrix} 4 & 6 \\ 6 & 3 \end{vmatrix} + 5(-1)^{3+1} \begin{vmatrix} 4 & 6 \\ 5 & 4 \end{vmatrix} \right] - \\ - 2 \left[2(-1)^{1+1} \begin{vmatrix} 5 & 4 \\ 6 & 3 \end{vmatrix} + 8(-1)^{2+1} \begin{vmatrix} 3 & 1 \\ 6 & 3 \end{vmatrix} + 5(-1)^{3+1} \begin{vmatrix} 3 & 1 \\ 5 & 4 \end{vmatrix} \right] + \\ + 3 \left[2(-1)^{1+1} \begin{vmatrix} 4 & 6 \\ 6 & 3 \end{vmatrix} + 6(-1)^{2+1} \begin{vmatrix} 3 & 1 \\ 6 & 3 \end{vmatrix} + 5(-1)^{3+1} \begin{vmatrix} 3 & 1 \\ 4 & 6 \end{vmatrix} \right] -$$

$$\begin{aligned}
 & - \left[2(-1)^{1+1} \begin{vmatrix} 4 & 6 \\ 5 & 4 \end{vmatrix} + 6(-1)^{2+1} \begin{vmatrix} 3 & 1 \\ 5 & 4 \end{vmatrix} + 8(-1)^{3+1} \begin{vmatrix} 3 & 1 \\ 4 & 6 \end{vmatrix} \right] = \\
 & = [6(15 - 24) - 8(12 - 36) + 5(16 - 30)] - 2[2(15 - 24) - 8(9 - 6) + \\
 & + 5(12 - 5)] + 3[2(12 - 36) - 6(9 - 6) + 5(18 - 4)] - [2(16 - 30) - \\
 & - 6(12 - 5) + 8(18 - 4)] = [-54 + 192 - 70] - 2[-18 - 24 + 35] + \\
 & + 3[-48 - 18 + 70] - [-28 - 42 + 112] = 68 + 14 + 12 - 42 = 52.
 \end{aligned}$$

Мы специально подробно провели все промежуточные выкладки, чтобы отметить один из существенных недостатков предложенного алгоритма. Оценим необходимое количество операций для его выполнения. Вычисление определителя матрицы порядка n сводится к вычислению n определителей матриц порядка $n - 1$. Каждый из определителей матриц порядка $n - 1$ сводится к вычислению $n - 1$ определителя матриц порядка $n - 2$. Таким образом, вычисление определителя по подобной схеме требует количества операций порядка $n(n - 1)(n - 2) \times \dots \times 3 \cdot 2 \cdot 1 = n!$. При $n = 10$ имеем $10! = 3628800$, и с ростом размерности матрицы количество операций очень быстро возрастает. Отметим также, что предложенный алгоритм обладает и другими недостатками, связанными с ограниченностью разрядной сетки представления чисел, что может привести к значительной потере точности вычисления определителя для матриц большой размерности.

Можно привести другой метод вычисления определителя, основанной на той же схеме, но который свободен от некоторых указанных недостатков (алгоритм Гаусса). Для этого напомним еще два свойства определителя — при перестановке двух строк (или столбцов) определитель меняет знак на противоположный, и определитель не изменится, если к элементам одной строки или столбца прибавить соответствующие элементы другой строки или столбца, умноженные на некоторый множитель. Отметим, что если все элементы первого столбца кроме *самого первого* равны нулю, то вычисление определителя матрицы порядка n сводится к вычислению *одного определителя матрицы порядка $n - 1$* . Первое свойство позволяет на начальном этапе изменить матрицу так, что этот элемент будет не равен нулю. Вычислим теперь тот же определитель с применением этих свойств.

$$\Delta = \begin{vmatrix} 1 & 2 & 3 & 1 \\ 2 & 6 & 4 & 6 \\ 3 & 8 & 5 & 4 \\ 1 & 5 & 6 & 3 \end{vmatrix}.$$

Из второй строки вычтем первую, умноженную на 2; из третьей вычтем первую, умноженную на 3; из четвертой вычтем первую, тогда

$$\Delta = \begin{vmatrix} 1 & 2 & 3 & 1 \\ 2 - 2 \cdot 1 & 6 - 2 \cdot 2 & 4 - 2 \cdot 3 & 6 - 2 \cdot 1 \\ 3 - 3 \cdot 1 & 8 - 3 \cdot 2 & 5 - 3 \cdot 3 & 4 - 3 \cdot 1 \\ 1 - 1 & 5 - 2 & 6 - 3 & 3 - 1 \end{vmatrix} = \begin{vmatrix} 1 & 2 & 3 & 1 \\ 0 & 2 & -2 & 4 \\ 0 & 2 & -4 & 1 \\ 0 & 3 & 3 & 2 \end{vmatrix}.$$

Разложив определитель по первому столбцу, получаем

$$\Delta = \begin{vmatrix} 2 & -2 & 4 \\ 2 & -4 & 1 \\ 3 & 3 & 2 \end{vmatrix}.$$

Проведем аналогичные действия с полученным определителем. Из второй строки вычтем первую; из третьей строки вычтем первую, умноженную на множитель $\frac{3}{2}$.

$$\Delta = \begin{vmatrix} 2 & -2 & 4 \\ 2 - 2 & -4 - (-2) & 1 - 4 \\ 3 - \frac{3}{2} \cdot 2 & 3 - \frac{3}{2} \cdot (-2) & 2 - \frac{3}{2} \cdot 4 \end{vmatrix} = \begin{vmatrix} 2 & -2 & 4 \\ 0 & -2 & -3 \\ 0 & 6 & -4 \end{vmatrix}.$$

Разложим по первому столбцу, и получим окончательный результат

$$\Delta = 2 \cdot \begin{vmatrix} -2 & -3 \\ 6 & -4 \end{vmatrix} = 52.$$

Оценим количество операций, которое требует данный метод. Для того чтобы обнулить первый элемент в некоторой строке путем прибавления первой строки, умноженной на соответствующий множитель, для матрицы порядка n требуется n операций. Всего имеется n строк. Следовательно, для того чтобы свести вычисление определителя матрицы порядка n к вычислению *одного* определителя матрицы порядка $n - 1$, число требуемых операций имеет порядок n^2 . Следовательно, общее число операций, которое требуется для вычисления определителя, имеет порядок n^3 , а для больших значений n величина n^3 существенно меньше, чем $n!$. Особенностью предложенного алгоритма является тот факт, что его реализация для массива, хранящего квадратную матрицу некоторой заданной размерности, сводится путем некоторых преобразований к реализации того же самого алгоритма, но для массива меньшей размерности. Останов алгоритма происходит, когда размер матрицы равен 2×2 , и определитель может быть вычислен непосредственно по формуле (1.1.6).

Основные понятия и определения. Существует один общий момент, который объединяет изложенные в примерах математические методы решения рассмотренных задач. Так:

— в примерах 1.1–1.3 *следующие члены последовательности вычислялись по известным формулам с использованием предыдущих, ранее найденных членов исследуемой последовательности*;

— в примере 1.4 полученное соотношение позволяет *вычислять элементы функциональной последовательности по известным ее предыдущим элементам*;

— в примере 1.5 предложенный метод позволяет *сводить процедуру решения задачи для массива некоторой размерности к точно такой же процедуре, но для массива меньшей размерности*.

Все эти методы опираются на идею построения рекуррентного соотношения. Далее мы попытаемся пояснить смысл понятия рекурсии. Понятие *рекуррентный* восходит к латинскому *recurrentis* — возвращающийся. В математике *рекуррентные последовательности* (возвратные последовательности) — это последовательности, каждый следующий член которых, начиная с некоторого, выражается по заданному определенному правилу через предыдущие. *Рекурсивные функции* (восходит к латинскому *recursio* — возвращение) — это такие функции, значения которых для данного аргумента вычисляются с помощью значений для предшествующих аргументов. Рекурсивные функции являются частичными функциями, т. е. функциями, не обязательно всюду определенными. Чтобы подчеркнуть это обстоятельство, часто используют термин «частично рекурсивные функции». Под термином «рекурсия» подразумевается способ задания функции, при котором значения определяемой функции для любых значений аргумента выражаются по известным правилам через значения этой функции для *меньших* значений аргументов. Сам этот термин подразумевает, что значения аргументов упорядочены в некотором смысле и для каждой конкретной реализации процесса определения значения функции мы имеем *конечный* набор меньших значений аргумента. То есть фактически, в каждой конкретной реализации процесса определения значений рекурсивно заданной функции мы имеем дело с последовательностью ее значений. И в некотором смысле термины *рекурсивно заданная функция* и *рекурсивно заданная последовательность* эквивалентны, однако исторически сложилось так, что в некоторых случаях употребляется термин *последовательность*, а в некоторых — *рекурсивно заданная функция*.

Далее мы попытаемся дать более строгое математическое понятие термину «рекурсивно заданная функция». Числовые функции, значения которых можно установить с помощью некоторого известного алгоритма, называются *вычислимыми функциями*. Функция называется рекурсивной, если существует эффективная процедура для ее вычисления — т. е. существует определенный конечный набор известных математических операций, при выполнении которых мы получаем требуемое значение функции.

Впервые класс рекурсивных функций описал К. Гёдель — как класс всех числовых функций, определимых в некоторой формальной системе. А. Черч в 1936 году вывел тот же класс числовых функций, что и Гёдель, исходя из совершенно других предпосылок. Черчем была сформулирована гипотеза, известная под названием тезиса Черча: *класс рекурсивных функций тождественен с классом всюду определенных вычислимых функций*. Поскольку понятие вычислимой функции носит интуитивный характер, то доказать этот тезис не представляется возможным.

Рекурсивные функции в теории алгоритмов. Применение рекурсивных функций в теории алгоритмов основано на идее нумерации

слов в определенном алфавите. После нумерации входных и выходных слов любой алгоритм превращается в функцию $y = f(x)$, где аргументы x и значения функции y принимают лишь неотрицательные целые значения. Отметим, что указанная функция может быть определена не для всех значений аргумента, а лишь для тех, которые составляют ее область определения. Данные функции будем называть *арифметическими функциями*. Среди арифметических функций выделим класс *элементарных арифметических функций* в который входят

- функции, тождественно равные нулю — $O^n(x_1, x_2, \dots, x_n) = 0$;
- функция прибавления единицы — $S^1(x) = x + 1$;
- функция, выделяющая из системы натуральных чисел член с данным номером — $I_i^n(x_1, x_2, \dots, x_n) = x_i$ ($1 \leq i \leq n$; $n = 1, 2, \dots$).

Используя перечисленные элементарные функции и небольшое число конструктивных приемов, можно строить более сложные функции. В теории рекурсивных функций важное значение имеют следующие три операции — суперпозиции, примитивной рекурсии и минимизации.

Операция суперпозиции сопоставляет функции f от n переменных и функциям g_1, \dots, g_m от m переменных функцию h от m переменных, такую, что для любых натуральных чисел x_1, \dots, x_m имеем

$$h(x_1, \dots, x_m) \cong f(g_1(x_1, \dots, x_m), \dots, g_m(x_1, \dots, x_m)),$$

здесь и далее условное равенство \cong означает, что оба выражения, связываемые им, осмыслены одновременно, и в случае осмысленности имеют одно и то же значение.

Оператор примитивной рекурсии сопоставляет функции f от n переменных и функции g от $n+2$ переменных функцию h от $n+1$ переменных такую, что для любых натуральных чисел x_1, \dots, x_n, y имеем

$$\begin{aligned} h(x_1, \dots, x_n, 0) &\cong f(x_1, \dots, x_n), \\ h(x_1, \dots, x_n, y+1) &\cong g(x_1, \dots, x_n, y, h(x_1, \dots, x_n, y)). \end{aligned}$$

Оператор минимизации (наименьшего корня) сопоставляет функции f от $n+1$ переменных $f(x_1, \dots, x_n, y)$ функцию h от n переменных $h(x_1, \dots, x_n)$ такую, что для любых натуральных чисел x_1, \dots, x_n в качестве соответствующего значения $y = h(x_1, \dots, x_n)$ принимается наименьший неотрицательный корень уравнения $f(x_1, \dots, x_n, y) = 0$. В случае отсутствия целых неотрицательных корней данного уравнения функция $h(x_1, \dots, x_n)$ считается неопределенной при данном наборе переменных.

Важную роль в теории рекурсивных функций играют так называемые *примитивно рекурсивные функции* — такие рекурсивные функции, которые получаются из исходных функций в результате применения конечного числа одних лишь операторов подстановки и примитивной рекурсии. Арифметические функции, которые могут быть построены из элементарных арифметических функций с помощью операций суперпозиции, примитивной рекурсии и минимизации, называются ча-

стично рекурсивными функциями. Если такие функции оказываются определенными всюду, то они называются *общерекурсивными функциями*.

Понятие частично рекурсивной функции является одним из главных понятий в теории алгоритмов. Значение этого понятия определяется двумя моментами. Во-первых, каждая стандартно заданная частично рекурсивная функция вычислима путем определенной процедуры. Во-вторых, оказывается, что числовые функции, вычислимые известными на настоящее время точно очерченными алгоритмами, являются частично рекурсивными. Последнее замечание является подтверждением упомянутого выше тезиса Чёрча. В аспекте реального программирования это означает, что любой вычислительный алгоритм может быть реализован рекурсивно, т. е. задан в виде частично рекурсивной функции.

§ 2. Рекурсивно заданные последовательности и функции

В данном разделе мы попытаемся систематизировать некоторые наши знания, касающиеся рекурсивно заданных последовательностей и функций.

Рекурсивно заданные последовательности как функции целочисленного аргумента. Предположим, что требуется определить значения членов некоторой последовательности t_n — мы их будем рассматривать как значения функции $T(n)$, где аргумент n является целым положительным числом, при этом обычно предполагается, что $n \geq 0$ или $n \geq 1$. Кратко мы будем записывать $t_n = T(n)$. Рекурсивное задание последовательности t_n как последовательности значений функции $T(n)$ включает следующие два этапа.

Первый этап. Функция $T(n)$ задается непосредственно в виде числовых значений для некоторого конечного множества начальных значений аргумента n : $1 \leq n \leq m$.

Второй этап. Задается метод или формула, которые позволяют, зная все значения функции $T(n)$ при $n \leq k$ ($k \geq m$), вычислить ее значения при $n = k + 1$, т. е. найти $T(k + 1)$ — мы получаем рекуррентные соотношения, описывающие рекурсивно заданную функцию $T(n)$, равноположенную (равносильную) последовательности t_n . Полученная запись этих двух этапов, обычно объединенная фигурной скобкой

$$\begin{cases} T(1) = t_1, T(2) = t_2, \dots, T(m) = t_m; \\ T(n+1) = f(t_n, t_{n-1}, \dots, t_1), \quad n \geq m, \end{cases} \quad (1.2.1)$$

носит название *рекуррентного соотношения*. Таким образом, мы будем говорить, что *рекуррентное соотношение* (1.2.1) определяет *рекурсивно заданную функцию* $T(n)$, эквивалентную последовательности t_n .

Рассмотрим ряд примеров, и вначале вернемся к примерам предыдущего параграфа, в которых были описаны некоторые рекурсивно заданные последовательности.

Пример 1.6. Запишем рекурсивно заданную последовательность для решения уравнения $0,7x = 2$ (см. параграф 1.1) в виде рекурсивно заданной функции $T(n)$ — на основе (1.1.1) имеем *рекуррентное соотношение*

$$\begin{cases} T(1) = 1; \\ T(n+1) = 0,3T(n) + 2, \quad n \geq 1, \end{cases} \quad (1.2.2)$$

которое позволяет нам с нужной степенью точности вычислить корень уравнения $0,7x = 2$, не используя операцию деления.

Пример 1.7. Касаясь второго примера предыдущего параграфа — вычисления квадратного корня, стоит сказать, что в современных компьютерах представление действительных чисел сводится к представлению в виде ограниченной двоичной мантиссы и порядка — мы оперируем с рациональными числами, имеющими фиксированный делитель. Для получения с нужной точностью величин, не являющихся рациональными числами, приходится прибегать к таким математическим приемам, которые, почти всегда (а может быть и всегда) требуют построения рекуррентных соотношений. Последовательность, заданная формулой (1.1.2), может быть представлена в виде рекурсивно заданной функции $T(n)$, а именно *рекуррентное соотношение*

$$\begin{cases} T(1) = 1; \\ T(n+1) = \frac{1}{2} \left(T(n) + \frac{a}{T(n)} \right), \quad n \geq 1 \end{cases} \quad (1.2.3)$$

позволяет нам получить с нужной степенью точности значение $x = \sqrt{a}$ при положительных значениях числа a .

Пример 1.8. На основе формулы (1.1.3) (см. пример 1.3) для вычисления определенного интеграла $\int_0^{\pi/2} \cos^n x dx$ как функции аргумента n полученное *рекуррентное соотношение* может быть представлено в виде

$$\begin{cases} T(0) = \frac{\pi}{2}; T(1) = 1; \\ T(n+2) = \frac{n-1}{n} \cdot T(n), \quad n \geq 0. \end{cases} \quad (1.2.4)$$

Пример 1.9. Теперь приведем еще один пример, который показывает необходимость применения рекуррентных соотношений для программистов. Все, мы надеемся, знают определение $n!$ (n -факториал), как произведение всех положительных чисел от 1 до n . По определению $0! = 1$, далее: $1! = 1$, $2! = 1 \cdot 2$, $3! = 1 \cdot 2 \cdot 3 = 6$, и так далее $n! = 1 \cdot 2 \cdot \dots \cdot (n-1) \cdot n$. Однако современные компьютеры многоточие пока не воспринимают. Поэтому для вычисления последовательности $t_n = n!$

удобно использовать рекуррентное соотношение, задающее рекурсивно заданную функцию $T(n) = n!$ в виде

$$\begin{cases} T(0) = 1; \\ T(n+1) = (n+1) \cdot T(n), & n \geq 0. \end{cases}$$

Рекурсивно заданные функции. Во всех приведенных выше примерах мы имели дело с рекурсивно заданными последовательностями, как функциями, аргументом которых является номер члена последовательности. Касаясь рекурсивно заданных функций, нужно отметить один момент, поскольку это понятие допускает неоднозначное толкование. Под термином «рекурсивно заданная функция», как правило, понимается такая функция, значение которой для данного аргумента вычисляется с помощью значений для предшествующих аргументов. При каждой конкретной реализации процесса вычисления такой функции мы имеем дело с рекуррентно заданной последовательностью. В практике вычислений это связано с тем, что, имея дело с реальной функцией (для простоты будем говорить о функции одной переменной), мы не вычисляем ее в реальной задаче на всей числовой оси, а вычисляем для некоторой конечной последовательности значений аргументов. Поэтому в таком аспекте фраза «рекурсивно заданная функция» носит несколько условный характер.

Пример 1.10. Примером вычисления значений непрерывной функции для некоторой последовательности значений аргументов может служить известный метод Эйлера численного решения дифференциального уравнения. Мы заранее хотим отметить, что не будем говорить ни о точности этого метода, ни об условиях его применимости. Нам важна лишь принципиальная сторона этого метода с точки зрения рекурсивного задания функции. Рассмотрим дифференциальное уравнение первого порядка

$$\frac{dy}{dx} = f(x; y) \quad (1.2.5)$$

с начальным условием $y(x_0) = y_0$ (задача Коши). Требуется определить решение этого уравнения при $x > x_0$. Известно, что большинство таких уравнений не решается в аналитическом виде, и решение приходится искать численно. Зададимся некоторым, достаточно малым, шагом h разбиения числовой оси и построим последовательность точек x_n , где $x_n = x_0$ при $n = 0$ и $x_n = x_0 + nh$ при $n \geq 1$. Мы будем считать задачу определения решения выполненной, если сумеем определить значения неизвестной функции $y(x)$ не на всей числовой оси, а в последовательности точек x_n . Обозначим $y_n = y(x_n)$, тогда наша цель состоит в том, чтобы определить последовательность значений y_n . Вспомним определение производной:

$$\frac{dy}{dx} = \lim_{\Delta x \rightarrow 0} \frac{y(x + \Delta x) - y(x)}{\Delta x}.$$

Если считать шаг разбиения h достаточно малым, то мы можем записать *приближенное равенство*

$$\frac{dy}{dx} \Big|_{x=x_i} \approx \frac{y(x_{i+1}) - y(x_i)}{h}, \quad (1.2.6)$$

а согласно (1.2.5)

$$\frac{dy}{dx} \Big|_{x=x_i} = f(x_i; y_i), \quad (1.2.7)$$

следовательно, в соответствии с (1.2.6) и (1.2.7) имеем приближенное равенство

$$\frac{y_{i+1} - y_i}{h} \approx f(x_i; y_i),$$

которое преобразуется к виду

$$y_{i+1} \approx y_i + f(x_i; y_i) \cdot h. \quad (1.2.8)$$

Формула (1.2.8) позволяет записать рекуррентное соотношение для определения последовательности значений $y_n = y(x_n)$, т. е. функции $y(x)$ в точках $x_n = x_0 + nh$

$$\begin{cases} y(x_0) = y_0; \\ y(x_{n+1}) = y(x_n) + f(x_n; y(x_n)) \cdot h, \quad n \geq 0. \end{cases}$$

Отметим, что при каждой конкретной реализации этого метода мы будем иметь дело с рекурсивно заданной последовательностью значений, которые с некоторой степенью точности представляют неизвестную функцию $y(x)$. В заключение этого примера мы считаем необходимым напомнить, что данное учебное пособие посвящено теории рекурсии, а не численным методам решения дифференциальных уравнений. Поэтому мы не рекомендуем непосредственно применять метод Эйлера для решения конкретных дифференциальных уравнений без ознакомления со специальной литературой по численным методам. У непосредственного применения метода Эйлера есть достаточно много «подводных камней», в частности, это касается рационального выбора шага интегрирования для достижения требуемой точности результата.

Пример 1.11. Расширением нашего понимания рекурсивно заданной функции является более общий случай, когда речь идет о рекурсивно заданной последовательности функций. В этом случае предполагается, что задана функциональная последовательность $F_n(x)$ как функция двух переменных $F_n(x) = f(n; x)$, где первая переменная n принимает лишь целые значения (для определенности полагаем $n \geq 1$). При $1 \leq n \leq m$ члены функциональной последовательности являются известными функциями, и существует некоторое правило, которое позволяет для $n \geq m$ вычислить $F_{n+1}(x)$ по известным всем предшествующим членам функциональной последовательности.

Например, задачу, рассмотренную в примере 1.4 из предшествующего параграфа, мы можем переформулировать как задачу нахождения функции двух переменных, задающей первообразную вида

$$J(n; x) = \int \frac{dx}{(x^2 + a^2)^n},$$

где n — целое положительное число. Поскольку первообразные определены с точностью до произвольной постоянной, чтобы избежать неоднозначности, будем вести речь о первообразных, значения которых равны нулю в начале координат. Полученную формулу (см. (1.1.5)) для первообразной мы можем записать в виде функции, рекурсивно заданной по первому аргументу, и определенной рекуррентным соотношением

$$\begin{cases} J(1; x) = \frac{1}{a} \operatorname{arctg} \frac{x}{a}; \\ J(n; x) = \frac{x}{2(n-1)a^2(x^2 + a^2)^{n-1}} + \frac{(2n-3)}{2(n-1)a^2} J(n-1, x), \quad n \geq 2. \end{cases}$$

Отметим, что в данном примере для каждого конкретного значения переменной x мы имеем рекурсивно заданную числовую последовательность.

Пример 1.12. Приведем еще один пример, к которому в полной мере применим термин «рекурсивно заданная функция». Вспомним формулу вычисления биномиальных коэффициентов

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

Дадим этому соотношению более широкое толкование. Рассмотрим функцию двух переменных $C(n; k)$, область определения которой: n — целые неотрицательные числа, а k — целые числа. Положим, что

$$C(n; k) = C_n^k = \frac{n!}{k!(n-k)!}$$

при $0 \leq k \leq n$, и $C(n; k) = 0$ при $k < 0$ и $k > n$.

Заданную таким образом функцию можно легко определить как рекурсивно заданную функцию по переменной n следующим соотношением

$$\begin{cases} C(0; k) = 0, \quad k \neq 0, \quad C(0; 0) = 1; \\ C(n+1; k) = C(n; k) + C(n; k-1), \quad n \geq 0. \end{cases}$$

Нетрудно видеть, что определенная таким образом функция представляется хорошо известным треугольником Паскаля. Подробное исследование предлагаем самостоятельно провести читателям.

Пример 1.13. Рассмотрим известную в математике гамма-функцию Эйлера — $\Gamma(x)$. Она определяется соотношением

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt.$$

Будем ее рассматривать при значениях $x > 0$. Преобразуем выражение для $\Gamma(x)$, используя формулу интегрирования по частям

$$\begin{aligned}\Gamma(x) &= \int_0^\infty t^{x-1} e^{-t} dt = - \int_0^\infty t^{x-1} de^{-t} = \\ &= -t^{x-1} e^{-t} \Big|_0^\infty + (x-1) \int_0^\infty t^{x-2} e^{-t} dt = (x-1) \cdot \Gamma(x-1)\end{aligned}$$

при $x > 1$. Полученное соотношение позволяет по известным значениям функции $\Gamma(x)$ на полуинтервале $(0; 1]$, полученным с использованием численных методов, вычислять значения этой функции для аргумента $x > 1$ с помощью рекуррентного соотношения

$$\Gamma(x) = (x-1) \cdot \Gamma(x-1).$$

Пример 1.14. В заключение рассмотрим пример использования рекуррентных соотношений для решения линейных дифференциальных уравнений. Рассмотрим дифференциальное уравнение вида

$$\frac{d^2y}{dx^2} = -x^2 y$$

с условиями $y(0) = c$, $y'(0) = d$. Данное уравнение не решается аналитически в элементарных функциях. Попробуем найти его решение в виде степенного ряда, задающего искомую функцию $y(x)$

$$y(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_n x^n + \dots$$

Наша цель состоит в том, чтобы найти коэффициенты данного ряда. Из начальных условий следует, что $a_0 = c$, $a_1 = d$. Вычисляя вторую производную, получаем

$$\frac{d^2y}{dx^2} = 1 \cdot 2 \cdot a_2 + 2 \cdot 3 \cdot a_3 x + 3 \cdot 4 \cdot a_4 x^2 + \dots + n(n-1)a_n x^{n-2} + \dots$$

Подставляя результат в исходное уравнение, имеем

$$\begin{aligned}1 \cdot 2 \cdot a_2 + 2 \cdot 3 \cdot a_3 x + 3 \cdot 4 \cdot a_4 x^2 + \dots + n(n-1)a_n x^{n-2} + \dots &= \\ = -x^2 (a_0 + a_1 x + a_2 x^2 + \dots + a_n x^n + \dots),\end{aligned}$$

и, приравнивая коэффициенты при одинаковых степенях x , получаем

$$\begin{aligned} a_2 &= 0; \\ a_3 &= 0; \\ 3 \cdot 4 \cdot a_4 &= -a_0; \\ &\dots \\ n \cdot (n-1) \cdot a_n &= -a_{n-4}. \end{aligned}$$

Из этих соотношений получаем рекуррентное соотношение для определения значений рекурсивно заданной последовательности коэффициентов искомого степенного ряда

$$\begin{cases} a_0 = c; & a_1 = d; & a_2 = 0; & a_3 = 0; \\ a_n = -\frac{a_{n-4}}{n \cdot (n-1)}, & n \geq 4. \end{cases}$$

Мы надеемся, что наши читатели без труда смогут записать аналогичное соотношение для рекурсивно заданной функции $T(n) = a_n$. Полученные соотношения позволяют определить коэффициенты степенного ряда, задающего неизвестную функцию $y(x)$. Мы оставляем за пределами обсуждения вопрос определения области сходимости полученного ряда и, соответственно, вопрос о правомерности его почлененного дифференцирования. Эти вопросы изучаются в соответствующих разделах общего курса высшей математики.

Таким образом, термин «рекурсивно заданная функция» может рассматриваться в различных аспектах, а именно:

- рекурсивно заданные функции с целочисленным аргументом как аналог рекурсивно заданных последовательностей (примеры 1.6–1.9, 1.14);

- рекурсивно заданные функции с действительным аргументом, задающие значения непрерывной функции для счетной или конечной последовательности значений аргумента (примеры 1.10, 1.13);

- рекурсивно заданная последовательность функций как функция двух переменных, одна из которых принимает целочисленные значения и является аргументом рекурсии (примеры 1.11, 1.12).

Этим перечислением отнюдь не исчерпываются разновидности рекурсивно заданных функций, ниже мы познакомимся с дважды рекурсивными функциями (функция Аккермана), понятием косвенной рекурсии и т. д. Тем не менее, приведенные примеры уже показывают, что рекурсивно заданные функции, в различных аспектах их понимания, возникают в целом ряде задач, предполагающих использование компьютера для их решения — в «задачах компьютерной математики», мы считаем, что этот мало распространенный термин заслуживает намного более широкого использования.

§ 3. Классификация рекурсивно заданных последовательностей и функций

Классификация. При классификации рекурсивно заданных последовательностей используется следующая терминология. Последовательность x_1, x_2, \dots, x_n называется *последовательностью с частичной предысторией*, если следующий член последовательности зависит от некоторого фиксированного числа предыдущих членов. Она может быть выражена соотношением

$$x_n = f(x_{n-1}, x_{n-2}, \dots, x_{n-i}),$$

где $n > i$. Если x_n зависит от всех предыдущих членов, то говорят, что рекурсивно заданная последовательность имеет полную предысторию.

Важным классом рекурсивно заданных последовательностей являются последовательности, определяемые линейными однородными и линейными неоднородными рекуррентными соотношениями.

Линейным однородным рекуррентным соотношением порядка p называется соотношение вида

$$a_n = h_1(n) \cdot a_{n-1} + h_2(n) \cdot a_{n-2} + \dots + h_p(n) \cdot a_{n-p},$$

где $h_1(n), h_2(n), \dots, h_p(n)$ — заданные функции аргумента n , $h_p(n) \neq 0$. Если $h_1(n), h_2(n), \dots, h_p(n)$ не зависят от n , а являются постоянными, то такое соотношение называется *линейным однородным рекуррентным соотношением с постоянными коэффициентами порядка p* .

Линейным неоднородным рекуррентным соотношением порядка p называется соотношение вида

$$a_n = h_1(n) \cdot a_{n-1} + h_2(n) \cdot a_{n-2} + \dots + h_p(n) \cdot a_{n-p} + g(n),$$

где $h_1(n), h_2(n), \dots, h_p(n), g(n)$ — заданные функции аргумента n , $h_p(n) \neq 0$.

Примеры классификации. Приведем некоторые примеры отнесения рекурсивно заданных последовательностей и соответствующих рекуррентных соотношений к указанным классам. Полученное в примере 1.6 рекуррентное соотношение (1.2.2)

$$\begin{cases} T(1) = 1; \\ T(n+1) = 0,3T(n) + 2, & n \geq 1 \end{cases}$$

является линейным неоднородным рекуррентным соотношением порядка 1 с постоянными коэффициентами, а последовательность $t_n = T(n)$ — последовательностью с частичной предысторией. В примере 1.7 полученное рекуррентное соотношение (1.2.3)

$$\begin{cases} T(1) = 1; \\ T(n+1) = \frac{1}{2} \left[T(n) + \frac{a}{T(n)} \right], & n \geq 1 \end{cases}$$

является нелинейным рекуррентным соотношением с частичной предысторией. В примере 1.8 рекуррентное соотношение (1.2.4)

$$\begin{cases} T(0) = \frac{\pi}{2}; T(1) = 1; \\ T(n+2) = \frac{n-1}{n} T(n), \quad n \geq 0 \end{cases}$$

является линейным однородным рекуррентным отношением порядка 2 с переменными коэффициентами.

Приведем еще некоторые примеры математических задач, приводящих к необходимости использования рекуррентных соотношений.

Пример 1.15. Вывести формулу для вычисления $\int_0^x t^n e^{kt} dt$ как рекурсивно заданную функцию аргумента n . Будем рассматривать интеграл

$$\int_0^x t^n e^{kt} dt$$

как функцию двух переменных n и x

$$J(n; x) = \int_0^x t^n e^{kt} dt.$$

Для $n = 0$ получаем

$$J(0; x) = \int_0^x e^{kt} dt = \frac{1}{k} e^{kt} \Big|_0^x = \frac{1}{k} (e^{kx} - 1).$$

При $n > 0$, используя известную формулу интегрирования по частям, получаем

$$\begin{aligned} J(n; x) &= \int_0^x t^n e^{kt} dt = \frac{1}{k} \int_0^x t^n de^{kt} = \frac{1}{k} (t^n e^{kt}) \Big|_0^x - \frac{n}{k} \int_0^x t^{n-1} e^{kt} dt = \\ &= \frac{1}{k} x^n e^{kx} - \frac{n}{k} J(n-1; x), \end{aligned}$$

таким образом, для каждого фиксированного значения независимой переменной x мы получили рекуррентно заданную последовательность вида

$$\begin{cases} J(0; x) = \frac{1}{k} (e^{kx} - 1); \\ J(n; x) = -\frac{n}{k} J(n-1; x) + \frac{1}{k} x^n e^{kx}, \quad n > 0. \end{cases}$$

Полученное соотношение является линейным неоднородным рекуррентным соотношением порядка 1 с переменными коэффициентами.

Пример 1.16. В алфавите n символов. Сколько имеется слов в данном алфавите, которые содержат k символов и не содержат при этом одинаковых символов?

Обозначим количество таких слов через $A(n; k)$. Нетрудно видеть, что количество таких слов равно 0 при $n < k$. При $n \geq k$ количество слов может быть оценено следующим образом. Рассмотрим алфавит, содержащий $n - 1$ символ, и добавим еще один. Во-первых, сохранятся все слова содержащие k символов в алфавите из $n - 1$ символов — таких слов $A(n - 1; k)$. Во-вторых, образовать слово из k символов можно, если взять любое слово из старого алфавита, содержащее $k - 1$ символ и добавить на любое место новый символ — таких слов будет $k A(n - 1; k - 1)$. Таким образом,

$$A(n; k) = A(n - 1; k) + k A(n - 1; k - 1).$$

Следовательно, функциональная последовательность $A(n; k)$ для $n \geq 1$ и целых значений k может быть определена рекуррентным соотношением

$$\begin{cases} A(1; k) = 0, & k \neq 1, \quad k \neq 0; \\ A(1; 1) = 1, \quad A(1; 0) = 1; \\ A(n; k) = A(n - 1; k) + k A(n - 1; k - 1), & n > 1. \end{cases}$$

Данное соотношение является рекуррентным соотношением с частичной предысторией, но не для числовой последовательности, а для функциональной. Отметим, что определенное таким образом $A(n; k) = 0$ при $k < 0$ и при $k > n$. Для $0 \leq k \leq n$ значение $A(n; k)$ равно числу размещений по k элементов на множестве, содержащем n элементов

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

Отметим, что форма задания рекуррентной последовательности играет существенную роль при ее дальнейшем исследовании.

Пример 1.17. Рассмотрим рекурсивную последовательность x_1, x_2, \dots, x_n с полной предысторией, определяемую рекуррентным соотношением

$$\begin{cases} x_1 = 0; \quad x_2 = 1; \\ x_{n+1} = \sum_{k=1}^n x_k, \quad n \geq 2. \end{cases} \quad (1.3.1)$$

Исследуем внимательно эту последовательность при $n \geq 2$, для x_{n+1} имеем

$$x_{n+1} = x_1 + x_2 + \dots + x_n, \quad (1.3.2)$$

а для x_{n+2}

$$x_{n+2} = x_1 + x_2 + \dots + x_n + x_{n+1}. \quad (1.3.3)$$

Соотношение (1.3.3) может быть переписано в виде

$$x_{n+2} = (x_1 + x_2 + \dots + x_n) + x_{n+1} = 2x_{n+1},$$

следовательно, последовательность x_1, x_2, \dots, x_n может быть задана более простым рекуррентным соотношением, как последовательность с частичной предысторией

$$\begin{cases} x_1 = 0; & x_2 = 1; & x_3 = 1; \\ x_{n+1} = 2x_n, & n \geq 3. \end{cases} \quad (1.3.4)$$

Соотношение (1.3.4) очень напоминает геометрическую прогрессию и, наверное, не нужно долго объяснять преимущество такой записи по сравнению с (1.3.1).

Влияние вида рекуррентного соотношения на характеристики рекурсивного алгоритма. На следующем примере мы хотим показать, как в рамках одной задачи применяемые для ее решения линейное однородное рекуррентное соотношение и нелинейное рекуррентное соотношение влияют на характеристики алгоритмов, реализующих эти соотношения.

Пример 1.18. Задача о случайному блуждании частицы. Предположим, что частица может находиться в одном из m состояний S_1, S_2, \dots, S_m . За один шаг она может перейти из состояния S_i в состояние S_j с вероятностью $p_{ij}(1)$. Требуется определить вероятность $p_{ij}(n)$ — вероятность перехода из состояния S_i в состояние S_j за n шагов. Введем в рассмотрение матрицу переходов $P(n)$. Матрица переходов $P(n)$ — это квадратная матрица размерности $(m \times m)$, элементы которой в строке с номером i и столбце с номером j соответственно равны $p_{ij}(n)$, т. е.

$$P(n) = \begin{pmatrix} p_{11}(n) & p_{12}(n) & \dots & p_{1j}(n) & \dots & p_{1m}(n) \\ p_{21}(n) & p_{22}(n) & \dots & p_{2j}(n) & \dots & p_{2m}(n) \\ \dots & \dots & \dots & \dots & \dots & \dots \\ p_{i1}(n) & p_{i2}(n) & \dots & p_{ij}(n) & \dots & p_{im}(n) \\ \dots & \dots & \dots & \dots & \dots & \dots \\ p_{m1}(n) & p_{m2}(n) & \dots & p_{mj}(n) & \dots & p_{mm}(n) \end{pmatrix}.$$

Отметим, что матрица $P(1)$ нам известна. Для того чтобы получить выражение для матрицы $P(n)$ ($n > 1$), проведем следующие рассуждения. Рассмотрим некоторое промежуточное состояние, возникшее после k ($1 \leq k < n$) шагов перехода. Частица может находиться в любом состоянии S_l , где $1 \leq l \leq m$. Вероятность перехода в состояние S_l из состояния S_i за k шагов соответственно равна $p_{il}(k)$. Вероятность перехода частицы из состояния S_l в состояние S_j за оставшиеся $n - k$ шагов равна $p_{lj}(n - k)$. Согласно формуле полной вероятности

$$p_{ij}(n) = p_{i1}(k) \cdot p_{1j}(n - k) + p_{i2}(k) \cdot p_{2j}(n - k) + \dots + p_{im}(k) \cdot p_{mj}(n - k),$$

то есть

$$p_{ij}(n) = \sum_{l=1}^m p_{il}(k) p_{lj}(n - k).$$

Если мы вспомним операцию умножения матриц, то мы обнаружим следующее матричное соотношение

$$P(n) = P(k) \cdot P(n - k), \quad (1.3.5)$$

тогда

$$\begin{aligned} P(2) &= P(1) \cdot P(2 - 1) = [P(1)]^2, \\ P(3) &= P(1) \cdot P(3 - 1) = [P(1)]^3, \\ &\dots \\ P(n) &= P(1) \cdot P(n - 1) = [P(1)]^n. \end{aligned}$$

В результате мы получили, что матрица $P(n)$ при $n > 1$ и при известной матрице $P(1)$, удовлетворяет рекуррентному соотношению

$$P(n) = P(1) \cdot P(n - 1). \quad (1.3.6)$$

Это соотношение является линейным однородным рекуррентным соотношением. Соотношение (1.3.5) также можно рассматривать как рекуррентное соотношение, но уже нелинейное. Возникает вопрос — а какое из соотношений более удобно для использования? Ответ на этот вопрос можно получить на основе рассмотрения количества операций, задаваемых алгоритмами, которые построены по этим рекуррентным соотношениям.

Попробуем оценить количество операций, которое требуется выполнить, чтобы вычислить $P(n)$, если использовать соотношение (1.3.6). Будем считать, что при умножении матриц мы используем самый простой алгоритм, который требует количества операций порядка m^3 . Поскольку операцию умножения матриц мы выполняем $n - 1$ раз, то количество операций оценивается величиной порядка $(n - 1)m^3$. Оценим требуемый объем памяти. Для хранения в памяти матрицы размерности $(m \times m)$ требуется объем памяти порядка m^2 . В процессе использования данного алгоритма на каждом шаге требуются две матрицы, следовательно, суммарный объем требуемой памяти имеет порядок $2m^2$.

Но можно использовать и другой способ вычисления $P(n)$ с использованием соотношения (1.3.5). Представим число n в двоичной форме

$$n = a_k 2^k + a_{k-1} 2^{k-1} + \dots + a_i 2^i + \dots + a_1 2^1 + a_0 2^0.$$

Заметим, что числа a_i могут принимать только два значения — ноль или единица. Используя (1.3.5), получаем

$$\begin{aligned} P(n) &= P(a_k 2^k + a_{k-1} 2^{k-1} + \dots + a_i 2^i + \dots + a_1 2^1 + a_0 2^0) = \\ &= [P(2^k)]^{a_k} \cdot P(a_{k-1} 2^{k-1} + \dots + a_i 2^i + \dots + a_1 2^1 + a_0 2^0) = \\ &= [P(2^k)]^{a_k} \cdot [P(2^{k-1})]^{a_{k-1}} \cdot \dots \cdot [P(2^i)]^{a_i} \cdot \dots \cdot [P(2^1)]^{a_1} \cdot [P(1)]^{a_0}. \end{aligned} \quad (1.3.7)$$

Нетрудно заметить, что $P(2^k) = P(2^{k-1}) \cdot P(2^{k-1})$. Поскольку $k = \lceil \log_2 n \rceil$, то количество операций, необходимых для вычисления $P(2^k)$, оценивается величиной $\lceil \log_2 n \rceil m^3$. При вычислении с использованием соотношения (1.3.7) операция произведения матриц выполняется $\beta_1(n) - 1$ раз, где функция $\beta_1(n)$ определяется как число единиц в двоичном представлении числа n . Следовательно, суммарное количество операций оценивается величиной порядка $(\lceil \log_2 n \rceil + \beta_1(n) - 1)m^3$. При больших значениях n величина $(\lceil \log_2 n \rceil + \beta_1(n) - 1)m^3$ существенно меньше $(n - 1)m^3$. Значит, при больших значениях n , с точки зрения требуемого количества операций, второй вариант алгоритма является более предпочтительным. Но этот алгоритм требует большего объема памяти — порядка $(\beta_1(n) + 1)m^2$. С другой стороны, мы можем переписать соотношение (1.3.7), изменив порядок сомножителей на обратный, т. е. в виде

$$\begin{aligned} P(n) &= P(a_k 2^k + a_{k-1} 2^{k-1} + \dots + a_i 2^i + \dots + a_1 2^1 + a_0 2^0) = \\ &= [P(1)]^{a_0} \cdot [P(2^1)]^{a_1} \cdot \dots \cdot [P(2^i)]^{a_i} \cdot \dots \cdot [P(2^{k-1})]^{a_{k-1}} \cdot [P(2^k)]^{a_k}. \end{aligned} \quad (1.3.8)$$

Преимущество соотношения (1.3.8) по сравнению с (1.3.7) состоит в том, что вычисление степеней матрицы для одного фиксированного значения n требует объема памяти порядка $3m^2$, а количество операций при этом имеет прежний порядок — $(\lceil \log_2 n \rceil + \beta_1(n) - 1)m^3$. Данный пример показывает, что, с точки зрения требований конкретного алгоритма вычислений, сама форма его рекуррентного представления играет существенную роль.

Нелинейные рекуррентные соотношения. В заключение приведем еще два характерных типа нелинейных рекуррентных соотношений, которые встречается при решении конкретных задач. Рассмотрим следующую игру. Первый партнер задумывает одно из целых чисел от 1 до n . Второй партнер пытается угадать задуманное число. Называя некоторое число, второй партнер получает от первого один из трех ответов — «равно», «больше», «меньше». Требуется найти функцию $f(n)$, где $f(n)$ — среднее число вопросов при оптимальной стратегии второго игрока. Нетрудно заметить, что можно положить $f(0) = 0$, $f(1) = 1$. Тогда при значениях $n > 1$ из предположения об оптимальной стратегии второго игрока при выборе числа i , которое он называет, получаем следующее рекуррентное соотношение для определения $f(n)$:

$$f(n) = 1 + \min_{1 \leq i \leq n} \left\{ \frac{i-1}{n} f(i-1) + \frac{n-i}{n} f(n-i) \right\}.$$

Отметим, что при исследовании вопроса о трудоемкости алгоритмов приходится довольно часто иметь дело с еще одним типом нелинейных рекуррентных соотношений, а именно с соотношениями вида

$$T(n) = a \cdot T\left(\left[\frac{n}{b}\right]\right) + f(n),$$

где $[x]$ — обозначает целую часть выражения. Отметим, что довольно часто вместо целой части $[x]$ в выражении встречаются функции вида $\lfloor x \rfloor$ — «пол» или $\lceil x \rceil$ — «потолок». Исследованию асимптотического поведения функций, заданных подобными рекуррентными соотношениями, будет в дальнейшем посвящен специальный раздел.

§ 4. Методы исследования и решения рекуррентных соотношений

Цели исследования рекуррентных соотношений и асимптотические обозначения. Сформулируем некоторые цели, которые мы ставим перед собой, исследуя рекуррентные соотношения. Формулируя цели, мы будем иметь в виду рекурсивно заданную последовательность. Члены рекурсивно заданной последовательности будем обозначать функционально — через $T(n)$. Итак, пусть последовательность $T(n)$ задана рекуррентным соотношением

$$\begin{cases} T(1) = a_1, & T(2) = a_2, \dots, T(m) := a_m; \\ T(n+1) := g(a_n, a_{n-1}, \dots, a_{n-m}), & n > m. \end{cases}$$

Конечно, идеальной целью является нахождение функции $f(n)$ сравнительно простого вида, такой что $T(n) = f(n)$. Однако найти такую функцию удается достаточно редко, хотя ниже мы приведем некоторые методы нахождения точного выражения для членов рекуррентной последовательности. Чаще удается решить более скромную задачу — задачу нахождения оценок этих членов. Как правило, нас интересуют их оценки при достаточно больших значениях n . Прежде чем перейти к постановке задач, напомним некоторые основные обозначения. Вводя эти обозначения, мы заранее предполагаем, что речь идет о неотрицательных функциях, стремящихся, как правило, к бесконечности при стремлении к бесконечности аргумента n .

Обозначение o -малое. Если $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$, то пишем $f(n) = o(g(n))$ (читается эф от эн есть о малое от же от эн).

Обозначение O -большое. Если существует число $C > 0$, такое что $\frac{f(n)}{g(n)} < C$ для всех значений n , то пишем $f(n) = O(g(n))$ (читается эф от эн есть О большое от же от эн). Заметим, что всякое o -малое является одновременно и O -большим, но не наоборот.

Обозначение Θ . Если найдутся две неотрицательные константы $c_1 > 0$ и $c_2 > 0$, а так же число n_0 , такое что при всех $n > n_0$ выполнено $c_1 g(n) \leq f(n) \leq c_2 g(n)$, то пишем $f(n) = \Theta(g(n))$.

Если $f(n) = \Theta(g(n))$, то говорят, что $g(n)$ является асимптотически точной оценкой для $f(n)$.

Эквивалентность. Если $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 1$, то пишем $f(n) \sim g(n)$ (читается эф от эн эквивалентно же от эн). Заметим, что если $f(n) \sim g(n)$, то $f(n) = g(n) + o(g(n))$.

Получение подобных оценок и является основной задачей исследования рекуррентных соотношений. Если не удается найти точное выражение для значения последовательности $T(n) = f(n)$, то лучшим результатом является такая оценка $g(n)$ для члена последовательности, что $T(n) \sim g(n)$. Но такую оценку также удается найти достаточно редко. Неплохой оценкой считается $T(n) = \Theta(g(n))$, особенно если известны значения c_1 и c_2 , а не просто доказано, что они существуют. Оценка $T(n) = O(g(n))$ также позволяет делать некоторые выводы о характере поведения исследуемой последовательности.

Методы асимптотической оценки функций, заданных рекуррентными соотношениями. Один из приемов, который используется для построения подобных оценок — это построение такой функции $f(n)$, которая бы мажорировала $T(n)$ для всех значений n , т. е. при всех $n \geq 1$ должно выполняться $T(n) \leq f(n)$. Иногда для определения такой функции используется следующий прием. Задается вид функции $f(n)$ в предположении, что она зависит от некоторых параметров, а значения этих параметров уточняются в процессе доказательства. Проиллюстрируем это на примере.

Пример 1.19. Пусть

$$\begin{cases} T(1) = c_1; \\ T(n) := 2T\left(\left[\frac{n}{2}\right]\right) + c_2 n, \quad n > 1, \end{cases}$$

через $[x]$ мы обозначаем целую часть аргумента. Будем искать $f(n)$ в виде $f(n) = a n \log_2 n + b$, где a и b пока неизвестные параметры. Воспользуемся методом математической индукции. При $n = 1$ оценка работает, если положить $b \geq c_1$. В соответствии с методом математической индукции полагаем, что для всех $k < n$ выполняется неравенство $T(k) \leq a k \log_2 k + b$. Поскольку

$$T(n) = 2T\left(\left[\frac{n}{2}\right]\right) + c_2 n,$$

то, полагая $k := \left[\frac{n}{2}\right]$, получаем

$$\begin{aligned} T(n) &= 2T\left(\left[\frac{n}{2}\right]\right) + c_2 n \leq 2(a k \log_2 k + b) + c_2 n \leq \\ &\leq 2\left(a \frac{n}{2} \log_2 \frac{n}{2} + b\right) + c_2 n = \\ &= a n \log_2 n - a n + c_2 n + 2b \leq a n \log_2 n + b. \end{aligned}$$

Последнее неравенство получено в предположении, что $a \geq c_2 + b$, таким образом, приведенная оценка будет справедлива при условии $b \geq -c_1$ и $a \geq c_2 + b$. Можно положить $b = c_1$, $a = c_2 + c_1$. Тогда можно

сделать вывод, что при всех $n \geq 1$ имеет место оценка $T(n) \leq (c_1 + c_2)n \log_2 n + c_1$. Это означает, что $T(n) = O(n \log_2 n)$. Отметим, что при приведенном доказательстве мы *не можем утверждать*, что полученная оценка является точной асимптотической оценкой. Мы лишь доказали тот факт, что $T(n)$ растет не быстрее чем $f(n) = n \log_2 n$. И уж если быть совсем строгим, то полученная оценка должна быть записана в виде

$$T(n) \leq (c_1 + c_2)n^* \log_2 n^* + c_1,$$

где $n^* = n$, если n четное число, и $n^* = n + 1$ в противном случае.

Второй прием — это оценка рекуррентного отношения методом подстановки, который мы иллюстрируем на следующем примере.

Пример 1.20. Рассмотрим рекуррентное соотношение

$$T(n) = 3T\left(\left[\frac{n}{4}\right]\right) + n.$$

Выберем целое число m такое, что $4^{m-1} < n \leq 4^m$, тогда $T(n) \leq 3T(4^{m-1}) + 4^m$. В свою очередь, $T(4^{m-1}) = 3T(4^{m-2}) + 4^{m-1}$. Подставляя полученный результат в первое неравенство, получаем

$$T(n) \leq 3\left(3T(4^{m-2}) + 4^{m-1}\right) + 4^m = 3^2T(4^{m-2}) + 4^m\left(1 + \frac{3}{4}\right).$$

Поскольку $T(4^{m-2}) = 3 \cdot T(4^{m-3}) + 4^{m-2}$, то получаем

$$T(n) \leq 3^3T(4^{m-3}) + 4^m\left(1 + \frac{3}{4} + \left(\frac{3}{4}\right)^2\right).$$

Продолжая этот процесс, мы получим следующее неравенство:

$$T(n) \leq 3^mT(1) + 4^m\left(1 + \frac{3}{4} + \left(\frac{3}{4}\right)^2 + \dots + \left(\frac{3}{4}\right)^{m-1}\right).$$

Последняя сумма в круглых скобках является суммой геометрической прогрессии со знаменателем $q = 3/4$, и ее значение не превышает суммы аналогичной бесконечной геометрической прогрессии, которая равна 4. Тогда $T(n) \leq 3^mT(1) + 4^{m+1}$, но поскольку $m \leq \log_4 n + 1$, то

$$T(n) \leq 3^{\log_4 n + 1} \cdot T(1) + 16n,$$

а так как $3^{\log_4 n} = n^{\log_4 3} = o(n)$, то $T(n) = O(n)$.

Аналогия между линейными однородными рекуррентными соотношениями и линейными однородными дифференциальными уравнениями. Как уже отмечалось выше, явное выражение для соответствующего члена рекурсивно заданной последовательности удается найти довольно редко. Тем не менее, существует важный класс рекурсивно заданных последовательностей, для которых все-таки удается найти явное выражение для соответствующего члена последовательности. Рассмотрим линейные однородные рекуррентные отношения с постоянными коэффициентами с частичной предысторией.

Напомним, что линейным однородным рекуррентным отношением с постоянными коэффициентами порядка p называется соотношение вида

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \dots + c_p a_{n-p}, \quad \text{где } c_p \neq 0.$$

Решения данного рекуррентного отношения обладают свойством линейности. Свойство линейности означает следующее. Пусть последовательности b_1, b_2, \dots, b_n и c_1, c_2, \dots, c_n являются решениями данного рекуррентного отношения. Для краткости обозначим $B = \{b_1, b_2, \dots, b_n\}$, $C = \{c_1, c_2, \dots, c_n\}$, тогда последовательность $D = bB + cC$, (где b и c — произвольные постоянные величины), являющаяся линейной комбинацией последовательностей B и C , также будет решением рекуррентного отношения. Решение исследуемых рекуррентных отношений во многом аналогично решению линейных однородных дифференциальных уравнений с постоянными коэффициентами порядка p . Чтобы дать качественную картину, рассмотрим случай $p = 2$ и представим эту аналогию. Однородное линейное дифференциальное уравнение второго порядка

$$\frac{d^2y}{dx^2} + d_1 \frac{dy}{dx} + d_2 y = 0.$$

Линейное однородное рекуррентное соотношение с постоянными коэффициентами порядка 2 — $a_n = c_1 a_{n-1} + c_2 a_{n-2}$. Решение линейного однородного дифференциального уравнения ищем в виде $y = e^{\lambda x}$. Подставляя в уравнение, получаем

$$\lambda^2 e^{\lambda x} + d_1 \lambda e^{\lambda x} + d_2 e^{\lambda x} = 0; \quad \lambda^2 + d_1 \lambda + d_2 = 0.$$

Для определения значения λ получено уравнение $\lambda^2 + d_1 \lambda + d_2 = 0$, которое называется характеристическим уравнением. Решение линейного однородного рекуррентного соотношения с постоянными коэффициентами порядка 2 ищем в виде $a_n = r^n$. Подставляя в уравнение, получаем

$$r^n = c_1 r^{n-1} + c_2 r^{n-2}; \quad r^2 - c_1 r - c_2 = 0.$$

Для определения значения r получено уравнение $r^2 - c_1 r - c_2 = 0$, которое называется характеристическим уравнением. При решении характеристических уравнений возможны три случая.

Первый случай. Если характеристическое уравнение $\lambda^2 + d_1 \lambda + d_2 = 0$ имеет два различных действительных корня $\lambda = k_1, \lambda = k_2$, то общее решение линейного однородного дифференциального уравнения представляется в виде

$$y = A e^{k_1 x} + B e^{k_2 x},$$

где A и B — произвольные постоянные (этих обозначений мы будем придерживаться и для остальных случаев). Если характеристическое уравнение $r^2 - c_1 r - c_2 = 0$ имеет два различных действительных корня

$r = r_1$, $r = r_2$, то общее решение линейного однородного рекуррентного соотношения представляется в виде

$$a_n = Ar_1^n + Br_2^n.$$

Второй случай. Если характеристическое уравнение $\lambda^2 + d_1\lambda + d_2 = 0$ имеет два равных действительных корня $\lambda_1 = \lambda_2 = k$, то общее решение линейного однородного дифференциального уравнения представляется в виде

$$y = A e^{kx} + B x e^{kx}.$$

Если характеристическое уравнение $r^2 - c_1r - c_2 = 0$ имеет два равных действительных корня $r_1 = r_2 = a$, то общее решение линейного однородного рекуррентного соотношения представляется в виде

$$a_n = A a^n + B n a^n.$$

Поясним этот факт. Очевидно, что последовательность $a_n = a^n$ является решением рекуррентного соотношения, поскольку число $r = a$ является корнем характеристического уравнения. Рассмотрим последовательность $a_n = n a^n$. Подставим эту последовательность в рекуррентное соотношение $a_n = c_1 a_{n-1} + c_2 a_{n-2}$, имеем

$$na^n = c_1(n-1)a^{n-1} + c_2(n-2)a^{n-2}.$$

Преобразуем данное соотношение к виду

$$na^{n-2}(a^2 - c_1a - c_2) + a^{n-1}(c_1a + 2c_2) = 0.$$

Поскольку a является корнем характеристического уравнения, то $(a^2 - c_1a - c_2) = 0$, а поскольку a является корнем кратности 2 характеристического уравнения, то $(c_1a + 2c_2) = 0$. Таким образом, последовательность $a_n = n a^n$ действительно является решением рекуррентного соотношения.

Третий случай. Если характеристическое уравнение $\lambda^2 + d_1\lambda + d_2 = 0$ имеет два комплексно сопряженных корня $\lambda = \alpha \pm i\beta$, то общее решение линейного однородного дифференциального уравнения представляется в виде

$$y = A e^{\alpha x} \cos \beta x + B e^{\alpha x} \sin \beta x.$$

Если характеристическое уравнение $r^2 - c_1r - c_2 = 0$ имеет два комплексно сопряженных корня $r = \alpha \pm i\beta$, то общее решение линейного однородного рекуррентного соотношения представляется в виде

$$a_n = A(\alpha + i\beta)^n + B(\alpha - i\beta)^n.$$

Воспользуемся тем, что мы рассматриваем уравнение с действительными коэффициентами и перепишем полученную формулу в более удобном виде. Очевидно, что если некоторая комплексная последовательность является решением линейного однородного рекуррентного соотношения с действительными коэффициентами, то решениями такого рекуррентного соотношения будут являться действительная и мнимая

части последовательности. Комплексное число $\alpha \pm i\beta$ может быть записано в тригонометрической форме $\alpha + i\beta = \rho(\cos \varphi + i \sin \varphi)$, где модуль комплексного числа ρ равен $\rho = \sqrt{\alpha^2 + \beta^2}$, а аргумент φ определяется из условия

$$\cos \varphi = \frac{\alpha}{\rho}, \quad \sin \varphi = \frac{\beta}{\rho} \quad (0 \leq \varphi < 2\pi).$$

Согласно формуле Муавра $[\rho(\cos \varphi + i \sin \varphi)]^n = \rho^n (\cos n\varphi + i \sin n\varphi)$. Поскольку и действительная и мнимая части последовательности являются решениями рекуррентного соотношения, то последовательности $b_n = \rho^n \cos n\varphi$ и $c_n = \rho^n \sin n\varphi$ будут решениями рекуррентного соотношения. Тогда общее решение может быть записано в виде $a_n = A\rho^n \cos n\varphi + B\rho^n \sin n\varphi$, где A и B — произвольные постоянные.

Приведем примеры.

Пример 1.21. Найти решение для последовательности чисел Фибоначчи $\text{fb}(n)$, заданной соотношениями

$$\begin{cases} \text{fb}(1) = 1; \\ \text{fb}(2) = 1; \\ \text{fb}(n) = \text{fb}(n-1) + \text{fb}(n-2), \quad n > 2. \end{cases}$$

Характеристическое уравнение имеет вид $r^2 = r + 1$ или $r^2 - r - 1 = 0$. Решая это уравнение, получаем

$$r_{1,2} = \frac{1 \pm \sqrt{5}}{2},$$

тогда

$$\text{fb}(n) = c \left(\frac{1 + \sqrt{5}}{2} \right)^n + d \left(\frac{1 - \sqrt{5}}{2} \right)^n.$$

Определим значения постоянных c и d . Поскольку $\text{fb}(1) = 1$, то

$$c \left(\frac{1 + \sqrt{5}}{2} \right) + d \left(\frac{1 - \sqrt{5}}{2} \right) = 1,$$

а так как $\text{fb}(2) = 1$, то

$$c \left(\frac{1 + \sqrt{5}}{2} \right)^2 + d \left(\frac{1 - \sqrt{5}}{2} \right)^2 = 1.$$

Мы получили систему двух линейных уравнений с двумя неизвестными для определения c и d . Решая эту систему, получаем

$$c = \frac{1}{\sqrt{5}}, \quad d = -\frac{1}{\sqrt{5}},$$

таким образом

$$\text{fb}(n) = \frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1 - \sqrt{5}}{2} \right)^n.$$

Пример 1.22. Найти формулу для общего члена рекурсивно заданной последовательности (решить рекуррентное соотношение)

$$\begin{cases} a_1 = 7; \\ a_2 = 19; \\ a_n = 4a_{n-1} - 3a_{n-2}, \quad n > 2. \end{cases}$$

Характеристическое уравнение имеет вид $r^2 = 4r - 3$ или $r^2 - 4r + 3 = 0$. Решая уравнение, получаем $r_1 = 1$, $r_2 = 3$. Общее решение имеет вид $a_n = c + d \cdot 3^n$. Поскольку $a_1 = 7$, то $c + 3d = 7$, а поскольку $a_2 = 19$, то $c + 9d = 19$. Решая полученную систему уравнений, получаем $c = 1$, $d = 2$, тогда $a_n = 1 + 2 \cdot 3^n$.

Пример 1.23. Найти формулу для общего члена рекурсивно заданной последовательности (решить рекуррентное соотношение)

$$\begin{cases} a_1 = 4; \\ a_2 = 12; \\ a_n = 4a_{n-1} - 4a_{n-2}, \quad n > 2. \end{cases}$$

Характеристическое уравнение имеет вид $r^2 = 4r - 4$ или $r^2 - 4r + 4 = 0$. Решая уравнение, получаем $r_1 = r_2 = 2$. Поскольку характеристическое уравнение имеет два равных действительных корня, то общее решение имеет вид $a_n = c \cdot 2^n + dn \cdot 2^n$. Поскольку $a_1 = 4$, то $2c + 2d = 4$, а поскольку $a_2 = 12$, то $4c + 8d = 12$. Решая полученную систему уравнений, получаем $c = 1$, $d = 1$, тогда

$$a_n = 2^n + n \cdot 2^n = (n+1) \cdot 2^n.$$

Пример 1.24. Найти формулу для общего члена рекурсивно заданной последовательности (решить рекуррентное соотношение)

$$\begin{cases} a_1 = 1; \\ a_2 = -2; \\ a_n = 2a_{n-1} - 4a_{n-2}, \quad n > 2. \end{cases}$$

Характеристическое уравнение имеет вид $r^2 = 2r - 4$ или $r^2 - 2r + 4 = 0$. Решая уравнение, получаем $r_{1,2} = 1 \pm i\sqrt{3}$. Поскольку характеристическое уравнение имеет два комплексно сопряженных корня, то представим их в тригонометрической форме. Модуль комплексного числа $\rho = \sqrt{1+3} = 2$. Для определения аргумента имеем

$$\cos \varphi = \frac{1}{2}, \quad \sin \varphi = \frac{\sqrt{3}}{2}; \quad \varphi = \frac{\pi}{3},$$

тогда

$$a_n = c \cdot 2^n \cos\left(\frac{n\pi}{3}\right) + d \cdot 2^n \sin\left(\frac{n\pi}{3}\right).$$

Используя начальные значения функции, получаем систему уравнений

$$\begin{cases} c + \sqrt{3} \cdot d = 1; \\ -2c + 2\sqrt{3} \cdot d = -2. \end{cases}$$

Решая эту систему, получаем $c = 1$, $d = 0$, и окончательно

$$a_n = 2^n \cdot \cos\left(\frac{n\pi}{3}\right).$$

Обобщение на случай линейного однородного рекуррентного соотношения с постоянными коэффициентами порядка p . Для рекуррентного соотношения

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \dots + c_p a_{n-p}, \quad \text{где } c_p \neq 0,$$

характеристическое уравнение имеет вид

$$r^p = c_1 r^{p-1} + c_2 r^{p-2} + \dots + c_p,$$

или

$$r^p - c_1 r^{p-1} - c_2 r^{p-2} - \dots - c_p = 0.$$

Если действительное число $r = b$ является корнем характеристического уравнения кратности m , то оно порождает m решений вида

- 1) $a_n = b^n;$
- 2) $a_n = nb^n;$
- 3) $a_n = n^2b^n;$
-
- m) $a_n = n^{m-1}b^n.$

Если комплексно сопряженные числа $r = \rho(\cos \varphi \pm i \sin \varphi)$ являются корнями характеристического уравнения кратности m каждое, то они порождают $2m$ решений вида:

- 1) $a_n = \rho^n \cos n\varphi, a_n = \rho^n \sin n\varphi;$
- 2) $a_n = n\rho^n \cos n\varphi, a_n = n\rho^n \sin n\varphi;$
- 3) $a_n = n^2\rho^n \cos n\varphi, a_n = n^2\rho^n \sin n\varphi;$
-
- m) $a_n = n^{m-1}\rho^n \cos n\varphi, a_n = n^{m-1}\rho^n \sin n\varphi.$

Тогда общее решение может быть представлено в виде суммы указанных решений, умноженных на произвольные постоянные. Продемонстрируем это на примере.

Пример 1.25. Найти формулу для общего члена рекурсивно заданной последовательности (решить рекуррентное соотношение)

$$\begin{cases} a_1 = 3; \\ a_2 = 10; \\ a_3 = 19; \\ a_n = 4a_{n-1} - 5a_{n-2} + 2a_{n-3}, \quad n > 3. \end{cases}$$

Характеристическое уравнение имеет вид $r^3 = 4r^2 - 5r + 2$ или $r^3 - 4r^2 + 5r - 2 = 0$. Данное характеристическое уравнение имеет сле-

дующие корни $r_1 = r_2 = 1$, $r_3 = 2$. Поскольку $r = 1$ является корнем кратности 2, то этот корень порождает 2 решения: 1) $a_n = 1$; 2) $a_n = n$. Корень $r = 2$ порождает решение $a_n = 2^n$. Тогда общее решение рекуррентного соотношения имеет вид $a_n = b + cn + d \cdot 2^n$. Для определения значений b , c , d имеем систему уравнений

$$\begin{cases} b + c + 2d = 3; \\ b + 2c + 4d = 10; \\ b + 3c + 8d = 19. \end{cases}$$

Решая эту систему, получаем $b = 0$, $c = 1$, $d = 1$, тогда $a_n = n + 2^n$.

Метод решения линейных неоднородных рекуррентных соотношений с постоянными коэффициентами порядка p . Напомним, что так называются соотношения вида

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \dots + c_p a_{n-p} + g(n),$$

где $c_p \neq 0$, $g(n)$ — известная функция. Метод решения таких соотношений состоит в следующем. На первом этапе удаляется $g(n)$, (т. е. полагается равной нулю) и находится общее решение соответствующего однородного соотношения. Далее ищется какое-либо частное решение неоднородного соотношения. Тогда общее решение неоднородного соотношения представляется как сумма частного решения и общего решения соответствующего однородного соотношения. Поскольку существует универсальный метод решения однородных соотношений, то проблема решения неоднородного состоит в том, чтобы «угадать» любое частное решение неоднородного соотношения. Для некоторых частных видов $g(n)$ существует универсальный прием отыскания частных решений. Об этом приеме мы скажем ниже. А пока отметим один полезный факт. Если функция $g(n)$ представима в виде суммы двух слагаемых $g(n) = g_1(n) + g_2(n)$ и каким-то образом удалось найти частные решения соотношений

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \dots + c_p a_{n-p} + g_1(n),$$

и

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \dots + c_p a_{n-p} + g_2(n),$$

то сумма таких частных решений будет частным решением соотношения

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \dots + c_p a_{n-p} + g(n).$$

Касаясь вопроса отыскания частных решений, отметим следующее. Если функция $g(n)$ имеет вид $g(n) = P_m(n) \cdot b^n$, где $P_m(n)$ — многочлен степени m , то частное решение может быть найдено в виде $a_n = n^s \times Q_m(n) \cdot b^n$, где $Q_m(n)$ — многочлен той же степени, что и $P_m(n)$, s — кратность корня $r = b$ в характеристическом уравнении. Если $r = b$ не является корнем характеристического уравнения, то $s = 0$. Отметим, что сам многочлен $P_m(n)$ представим в виде $P_m(n) = P_m(n) \cdot 1^n$.

Рассмотрим пример.

Пример 1.26. Найти формулу для общего члена рекурсивно заданной последовательности (решить рекуррентное соотношение)

$$a_n = 4a_{n-1} - 3a_{n-2} + 5 \cdot 2^n + (5n + 4).$$

Соответствующее однородное соотношение исследовалось ранее, и были найдены корни характеристического уравнения $r_1 = 1$, $r_2 = 3$. Общее решение однородного соотношения имеет вид $a_n = c + d \cdot 3^n$, где c и d — произвольные постоянные. В данном примере функция $g(n) = 5 \cdot 2^n + (5n + 4)$. Она состоит из двух слагаемых: $g_1(n) = 5 \cdot 2^n$ и $g_2(n) = (5n + 4)$. Вначале найдем частное решение соотношения

$$a_n = 4a_{n-1} - 3a_{n-2} + 5 \cdot 2^n.$$

Имеем функцию $g_1(n)$ вида $g_1(n) = P_m(n) \cdot b^n$, где $P_m(n) = 5$ — многочлен нулевой степени, и, значит, $m = 0$; $b = 2$, и поскольку число $b = 2$ не является корнем характеристического уравнения, то можно положить его кратность $s = 0$. Тогда частное решение соотношения ищем в виде $a_n = n^0 \cdot Q_0(n) \cdot 2^n = A \cdot 2^n$, где A — неизвестная величина, которую мы определим из рекуррентного соотношения. Подставляя данное выражение в рекуррентное соотношение, получаем

$$A \cdot 2^n = 4A \cdot 2^{n-1} - 3A \cdot 2^{n-2} + 5 \cdot 2^n; \quad A = -20.$$

Следовательно, частное решение имеет вид $a_n = -20 \cdot 2^n$. Далее найдем частное решение соотношения

$$a_n = 4a_{n-1} - 3a_{n-2} + (5n + 4).$$

Имеем функцию $g_2(n)$ вида $g_2(n) = P_m(n) \cdot b^n$, где $P_m(n) = 5n + 4$ — многочлен первой степени, и, значит, $m = 1$; $b = 1$, и поскольку число $b = 1$ является корнем характеристического уравнения кратности 1, то можно положить $s = 1$. Тогда частное решение соотношения ищем в виде

$$a_n = n^1 \cdot Q_1(n) \cdot 1^n = n(An + B),$$

где A и B — неизвестные величины, которые мы определим из рекуррентного соотношения. Подставляя данное выражение в рекуррентное соотношение, получаем

$$n(An + B) = 4(n - 1)(A(n - 1) + B) - 3(n - 2)(A(n - 2) + B) + (5n + 4).$$

Приравнивая коэффициенты при одинаковых степенях n , получаем систему уравнений для определения A и B :

$$\begin{cases} 4A + 5 = 0; \\ -8A + 2B + 4 = 0. \end{cases}$$

Решая эту систему, получаем $A = -5/4$, $B = -7$. Тогда частное решение рекуррентного соотношения имеет вид

$$a_n = -20 \cdot 2^n - n \left(\frac{5}{4}n + 7 \right).$$

Общее решение соответственно равно

$$a_n = d \cdot 3^n - 20 \cdot 2^n - n \left(\frac{5}{4}n + 7 \right) + c,$$

где c и d — произвольные постоянные.

Исчисление конечных сумм с использованием линейных неоднородных рекуррентных соотношений. Довольно часто приходится иметь дело с вычислением конечных сумм вида

$$s_n = \sum_{k=1}^n a_k,$$

где $a_k = f(k)$ — заданная функция. Отметим, что последовательность конечных сумм может быть задана линейным неоднородным рекуррентным соотношением первого порядка в виде

$$\begin{cases} s_0 = 0; \\ s_n = s_{n-1} + f(n), \quad n > 0. \end{cases}$$

Проиллюстрируем этот подход на следующем примере.

Пример 1.27. Найти $s_n = \sum_{k=0}^n k$.

Данная сумма может быть задана рекуррентным соотношением вида

$$\begin{cases} s_0 = 0; \\ s_n = s_{n-1} + n, \quad n > 0. \end{cases}$$

Для соответствующего однородного рекуррентного соотношения имеем следующее характеристическое уравнение: $r = 1$. Далее найдем частное решение соотношения $s_n = s_{n-1} + n$. Имеем функцию $g(n)$ вида $g(n) = P_m(n) \cdot b^n$, где $P_m(n) = n$ — многочлен первой степени, и, значит, $m = 1$; $b = 1$, и поскольку число $b = 1$ является корнем характеристического уравнения кратности 1, то можно положить $s = 1$. Тогда частное решение соотношения ищем в виде

$$s_n = n^1 \cdot Q_1(n) \cdot 1^n = n(An + B),$$

где A и B — неизвестные величины, которые мы определим из рекуррентного соотношения. Подставляя данное выражение в рекуррентное соотношение, получаем

$$n(An + B) = (n - 1)(A(n - 1) + B) + n.$$

Приравнивая коэффициенты при одинаковых степенях n , получаем систему уравнений для определения A и B

$$\begin{cases} 2A = 1, \\ A - B = 0. \end{cases}$$

Решая эту систему, получаем $A = 1/2$, $B = 1/2$, тогда частное решение рекуррентного соотношения имеет вид

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

Общее решение соответственно равно

$$s_n = \frac{n(n+1)}{2} + c,$$

где c — произвольная постоянная. Используя условие $s_0 = 0$, получаем $c = 0$. Следовательно,

$$s_n = \sum_{k=0}^n k = \frac{n(n+1)}{2}.$$

Пример 1.28. Найти $s_n = \sum_{k=0}^n k^3$.

Данная сумма может быть задана рекуррентным соотношением вида

$$\begin{cases} s_0 = 0; \\ s_n = s_{n-1} + n^3, \quad n > 0. \end{cases}$$

Для соответствующего однородного рекуррентного соотношения имеем следующее характеристическое уравнение: $r = 1$. Далее найдем частное решение соотношения $s_n = s_{n-1} + n^3$. Имеем функцию $g(n)$ вида $g(n) = P_m(n) \cdot b^n$, где $P_m(n) = n^3$ — многочлен третьей степени, и, значит, $m = 3$; $b = 1$, и поскольку число $b = 1$ является корнем характеристического уравнения кратности 1, то можно положить $s = 1$. Тогда частное решение соотношения ищем в виде

$$s_n = n^1 \cdot Q_3(n) \cdot 1^n = n(An^3 + Bn^2 + Cn + D),$$

где A, B, C, D — неизвестные величины, которые мы определим из рекуррентного соотношения. Подставляя данное выражение в рекуррентное соотношение, получаем

$$\begin{aligned} n(An^3 + Bn^2 + Cn + D) &= (n-1)(A(n-1)^3 + B(n-1)^2 + \\ &\quad + C(n-1) + D) + n^3. \end{aligned}$$

Приравнивая коэффициенты при одинаковых степенях n , получаем систему уравнений для определения A, B, C, D .

$$\begin{cases} 4A = 1; \\ -6A + 3B = 0; \\ 4A - 3B + 2C = 0; \\ -A + B - C + D = 0. \end{cases}$$

Решая эту систему, получаем $A = 1/4$, $B = 1/2$, $C = 1/4$, $D = 0$. Тогда частное решение рекуррентного соотношения имеет вид

$$s_n = \frac{n(n^3 + 2n^2 + n)}{4} = \frac{n^2(n+1)^2}{4}.$$

Общее решение соответственно равно

$$s_n = \frac{n^2(n+1)^2}{4} + c,$$

где c — произвольная постоянная. Используя условие $s_0 = 0$, получаем $c = 0$. Следовательно,

$$s_n = \sum_{k=0}^n k^3 = \frac{n^2(n+1)^2}{4}.$$

В качестве «бесплатного» приложения из решения последних двух примеров мы получили любопытный факт:

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

Метод решения рекуррентных соотношений первого порядка с переменными коэффициентами. Рекуррентным соотношением первого порядка с переменными коэффициентами называется соотношение вида

$$a_n = b(n)a_{n-1} + c(n), \quad n \geq 1,$$

где a_0 предполагается известной величиной. Идея решения данного соотношения состоит во введении суммирующего множителя $F(n)$, равного

$$F(n) = \frac{1}{\prod_{k=1}^n b(k)}.$$

Умножим обе части соотношения на суммирующий множитель. Получим

$$\frac{a_n}{\prod_{k=1}^n b(k)} = \frac{a_{n-1}b(n)}{\prod_{k=1}^n b(k)} + \frac{c(n)}{\prod_{k=1}^n b(k)}.$$

Обозначим через $y_n = b(n+1)F(n+1)a_n$. Тогда полученное соотношение может быть переписано в виде

$$y_n = y_{n-1} + F(n)c(n).$$

Это соотношение решается простым суммированием

$$y_n = y_0 + \sum_{k=1}^n F(k)c(k),$$

тогда

$$a_n = \frac{a_0 + \sum_{k=1}^n F(k)c(k)}{b(n+1)F(n+1)}.$$

Поскольку вычисление произведений сводится к исчислению сумм

$$\prod_{k=1}^n b(k) = \exp \left(\ln \prod_{k=1}^n b(k) \right) = \exp \left(\sum_{k=1}^n \ln b(k) \right),$$

то и исследование таких рекуррентных соотношений сводится к вопросу исчисления сумм.

Пример 1.30. Найти решение рекуррентного соотношения

$$\begin{cases} x_0 = 0; \\ x_n = \frac{n+1}{n}x_{n-1} + 2, \quad n \geq 1. \end{cases}$$

Суммирующий множитель имеет вид

$$F(n) = \frac{1}{\prod_{k=1}^n \frac{k+1}{k}} = \frac{1}{n+1}.$$

Обозначим

$$y_n = b(n+1)F(n+1)x_n = \frac{n+2}{n+1} \cdot \frac{1}{n+2} \cdot x_n = \frac{1}{n+1} \cdot x_n.$$

Для определения y_n получаем рекуррентное соотношение

$$\begin{cases} y_0 = 0; \\ y_n = y_{n-1} + \frac{2}{n+1}. \end{cases}$$

Тогда

$$y_n = \sum_{k=1}^n \frac{2}{k+1},$$

следовательно,

$$x_n = 2(n+1) \cdot \sum_{k=1}^n \frac{1}{k+1} = 2(n+1)(H_{n+1} - 1),$$

где H_i — i -ое гармоническое число.

В этом разделе мы коснулись лишь широко известных методов исследования рекуррентных соотношений. Более «тонким» методам исследования посвящена специальная литература, с которой читатель, если у него возникнет интерес или необходимость, может ознакомиться. Такую информацию можно найти, например, в литературе, рекомендованной в конце данной главы.

Задачи и упражнения к главе 1

1.1. Записать рекурсивную последовательность для вычисления квадратного корня в виде рекурсивно заданной функции.

1.2. Показать, что рекурсивная последовательность

$$\begin{cases} a_1 = 1; \\ a_n = \frac{1}{3} \left(2a_{n-1} + \frac{8}{a_{n-1}^2} \right), \quad n > 1 \end{cases}$$

имеет предел при $n \rightarrow \infty$ и найти значение этого предела.

1.3. Найти решение рекуррентного соотношения

$$a_n = \frac{a_{n-1}}{3} + \frac{2}{3}n^2 + \frac{2}{3}n - \frac{1}{3}.$$

1.4. Найти решение рекуррентного соотношения

$$a_n = 2a_{n-1} - n + 2.$$

1.5. Найти решение рекуррентного соотношения

$$a_n = -5a_{n-1} - 6a_{n-2}.$$

1.6. Найти решение рекуррентного соотношения

$$a_n = -6a_{n-1} - 9a_{n-2}$$

1.7. Найти решение рекуррентного соотношения

$$a_n = -2a_{n-1} - 2a_{n-2}.$$

1.8. Составить рекуррентное соотношение для вычисления $\sqrt[n]{a}$, $a > 0$, $n \in \mathbb{Z}$.

Список литературы к главе 1

- 1.1. Грин Д., Кнут Д. Математические методы анализа алгоритмов. — М.: Мир, 1987. — 120 с.
- 1.2. Андерсон Дж. Дискретная математика и комбинаторика: Пер. с англ. — М.: Издательский дом «Вильямс». 2003. — 960 с.
- 1.3. Грэхем Р., Кнут Д., Паташник О. Конкретная математика. Основание информатики: Пер. с англ. — М.: Мир, 1998. — 703 с.
- 1.4. Фалевич Б.Я. Теория алгоритмов: Учебное пособие. — М.: Машиностроение, 2004. — 160 с.
- 1.5. Ноден П., Китте К. Алгебраическая алгоритмика (с упражнениями и решениями): Пер. с франц. — М.: Мир, 1999. — 720 с.
- 1.6. Хаггарти Р. Дискретная математика для программистов. — М.: Техносфера, 2005. — 400 с.

Глава 2

РЕКУРСИВНЫЕ АЛГОРИТМЫ И ОСОБЕННОСТИ ИХ ПРОГРАММНЫХ РЕАЛИЗАЦИЙ

Введение. Цель этой небольшой главы — показать, как рекурсивно заданные последовательности и функции, определенные рекуррентными соотношениями, могут быть реализованы в виде рекурсивных алгоритмов. Язык программирования, поддерживающий рекурсивные вызовы, должен обладать специальным механизмом, который обеспечивает правильное обращение программного модуля к самому себе, — мы приводим одну из возможных моделей такого механизма, достаточно близкую к реальным языкам процедурного программирования. Мы также вводим понятие дерева рекурсии как логической конструкции, отражающей последовательность рекурсивных вызовов, порожденных рекурсивным алгоритмом на определенном входе. Исследование таких деревьев рекурсии будет продолжено в главе 6, где на их основе мы опишем один из методов исследования и анализа рекурсивных алгоритмов.

§ 1. Рекурсивные алгоритмы

Общая схема. Имея рекуррентное соотношение, описывающее рекурсивно заданную последовательность или функцию, мы можем на этой основе построить вычислительную процедуру, позволяющую получить значение необходимого нам члена последовательности или функции для заданного аргумента. Такая процедура будет обращаться к самой себе для вычисления необходимых предыдущих значений вплоть до останова рекурсии — ситуации, когда начальный член или несколько первых членов последовательности могут быть вычислены непосредственно. Мы получаем рекурсивный алгоритм, непосредственно опирающийся на рекуррентное соотношение. Обращаясь к примеру вычисления определителя квадратной матрицы из параграфа 1.1, заметим, что процесс вычисления определителя может быть представлен как рекурсивный алгоритм, вызывающий сам себя с матрицами, имеющими на каждом шаге уменьшающуюся на единицу размерность, вплоть до возможности непосредственного вычисления определителя. Можно интерпретировать процесс вычисления определителя как процесс вычисления значения функции, аргументом которой является квадратная матрица, тогда у нас есть алгоритм, который позволяет свести вычисление этой функции для матрицы размерности n к вычислению функции для некоторой другой матрицы размерностью $n - 1$.

В дальнейшем, в целях единства изложения, мы будем считать, что рекуррентное соотношение задает рекурсивную функцию.

Итак, рекуррентное соотношение задает непосредственно вычисляемые значения функции для начальных значений аргумента и правило, по которому значение функции для некоторого аргумента может быть вычислено на основе предыдущих известных значений. Переходя от рекуррентного соотношения к рекурсивному алгоритму, мы должны построить вычислительную процедуру. Таким образом, рекурсивный алгоритм должен включать в себя по крайней мере два фрагмента — фрагмент непосредственного вычисления функции для начальных значений аргумента и фрагмент, содержащий рекурсивное обращение. Такая схема носит название схемы *прямой рекурсии*.

Мы будем придерживаться в дальнейшем предположения о том, что средой программной реализации является процедурный язык высокого уровня. Говоря высоким стилем, мы остаемся в императивной парадигме программирования. Поскольку программные реализации в приложении к этой книге выполнены на языке *Delphi*, то мы принимаем форму записи алгоритмов (псевдокод), близкую к конструкциям данного языка. Как это принято в псевдокоде, мы опускаем описание типов данных и операторных скобок, показывая структурные фрагменты сдвигом вправо, и надеемся, что такая форма записи алгоритмов будет понятна нашим читателям. Мы предполагаем также, что у нашего читателя есть определенные навыки программирования, но на одном моменте мы хотим остановиться подробнее. Обращаем внимание на то, что рекуррентное соотношение задает некоторую функцию целочисленного аргумента в рамках понимания термина «функция» в классическом математическом анализе — как однозначного отображения множества допустимых значений аргумента на множество значений функции. В отличие от этого, в контексте «алгоритм в виде процедурно реализованной рекурсивной функции $F(n)$ » термин «функция» используется в рамках его понимания в программировании, — т. е. как именованного фрагмента в языке записи алгоритмов или в языке программирования, возвращающего значение по своему имени [2.1].

Рассмотрим схему прямой рекурсии как рекурсивного алгоритма в виде процедурно реализованной рекурсивной функции $F(n)$.

Схема прямой рекурсии.

$F(n)$	(рекурсивная функция)
If ($n=n0$)	(проверка останова рекурсии)
then	
$F \leftarrow C$	(прямое вычисление F при $n=n0$)
else	
$F \leftarrow \dots F(m) \dots$	(рекурсивный вызов, $m < n$)
Return (F)	
End.	

Для начальных значений аргумента ($n=n_0$) значение функции вычисляется непосредственно ($F \leftarrow C$) и приводит к останову рекурсии, а для других значений функция $F(n)$ обращается сама к себе (вызов $F(m)$) с меньшим значением аргумента, порождая рекурсивный вызов. Более сложная конструкция, возникающая в некоторых задачах, приводит к схеме, когда процесс рекурсии организуется опосредованно, — некоторая функция вызывается вновь через цепочку вызовов других функций, например, две функции вызывают друг друга, — такая схема называется *косвенной рекурсией*. Рассмотрим, в качестве примера, вариант схемы косвенной рекурсии для двух взаимосвязанных алгоритмов, реализованных в виде процедурных косвенно рекурсивных функций $F(n)$ и $G(m)$.

Схема косвенной рекурсии для двух функций.

```

F(n)                               (косвенно рекурсивная функция)
  If (n=n0)                      (проверка останова рекурсии)
    then
      F ← C                         (прямое вычисление F при n=x)
    else
      F ← ...G(k)...               (рекурсивный вызов, k < n)
    Return(F)
End.

```

```

G(m)                               (косвенно рекурсивная функция)
  If (m=m0)                      (проверка останова рекурсии)
    then
      G ← C                         (прямое вычисление G при m=m0)
    else
      G ← ...F(j)...               (рекурсивный вызов, j < m)
    Return(G)
End.

```

В этом примере схемы косвенной рекурсии две функции вызывают друг друга, и мы можем определить наличие рекурсии, только исследуя тексты двух функций одновременно.

Еще одна схема, отличная от прямой рекурсии, возникает для функций нескольких переменных. В случае, когда не только сама функция, но и один из ее аргументов вычисляются рекурсивно, мы получаем *схему дважды рекурсивной функции*. Классический пример — функция Аккермана, которая задается следующим рекуррентным соотношением [2.2]:

$$A(m, n) = \begin{cases} n + 1, & m = 0; \\ A(m - 1, n), & n = 0; \\ A(m - 1, A(m, n - 1)). \end{cases}$$

Несмотря на достаточно простую форму самого рекуррентного соотношения, последовательность рекурсивных вызовов достаточно сложна. Чтобы понять, как «устроена» функция Аккермана, мы советуем на-

рисовать последовательность рекурсивных вызовов, а вместе с этим и вычислить значение этой функции при $m = 3$, $n = 5$ — $A(3, 5)$. Первый шаг рекурсии приведет к равенству

$$A(3, 5) = A(2, A(3, 4)),$$

и придется вычислить $A(3, 4)$, чтобы определить второй индекс.

Примеры рекурсивных алгоритмов. Продемонстрируем схему прямой рекурсии на примере вычисления значения определенного интеграла (см. пример 1.3 из параграфа 1.1). Полученную при исследовании определенного интеграла

$$\int_0^{\pi/2} \cos^n x \, dx$$

рекурсивно заданную последовательность будем интерпретировать как рекурсивно заданную функцию целочисленного аргумента

$$f(n) = I_n = \int_0^{\pi/2} \cos^n x \, dx,$$

что позволяет, опираясь на (1.1.3), записать рекуррентное соотношение

$$\begin{cases} f(0) = \pi/2; & f(1) = 1; \\ f(n) = \frac{n-1}{n} \cdot f(n-2), & n \geq 2. \end{cases} \quad (2.1.1)$$

Рассмотрим рекурсивный алгоритм, вычисляющий значение этого определенного интеграла, в виде процедурно реализованной рекурсивной функции **F(n)** (прямая рекурсия) с остановом при значениях $n = 0$ и $n = 1$. Заметим, что если мы хотим вычислить, например, значение этого определенного интеграла при $n = 10$, то в вызывающей программе мы пишем вызов **y ← F(10)**, а запись рекурсивного алгоритма, вычисляющего значение этого определенного интеграла, имеет вид

```

F(n)
1 If (n=0)                                (проверка останова рекурсии)
2   then
3     F ← 0,5*pi.                          (прямое вычисление F при n=0)
4   else
5     If (n=1)                                (проверка останова рекурсии)
6       then
7         F ← 1.                            (прямое вычисление F при n=1)
8       else
9         F ← ((n-1)/n)*F(n-2)          (рекурсивный вызов)
10 Return(F)
End.

```

Дадим некоторые комментарии. В строках 1 и 5 мы, непосредственно следуя (2.1.1), проверяем условие останова рекурсии. В строках 3 и 7 вычисляется значение функции при останове рекурсии, которое и возвращается через имя функции **F** в точку вызова. Стока 9 будет выполнена, если значение аргумента не равно ни нулю, ни единице — это рекурсивный вызов — функция обращается сама к себе с меньшим значением аргумента. Когда будет получен результат из рекурсивного вызова, то возвращенное по имени функции значение будет участвовать в вычислениях строки 9, после чего функция снова возвращает результат, полученный в строке 9, в точку своего вызова. Заметьте, что схема вычислений имеет следующий вид: функция **F(n)** разворачивает цепочку вызовов, не производя никаких вычислений, до тех пор, пока не будет получено значение, непосредственно вычисленное при останове рекурсии. Теперь, возвращая вычисленные значения в себя, функция **F(n)**, выполняя строку 9, в конце концов (т. к. значение *n* ограничено), вернет вычисленное значение в вызывающую программу.

Другой пример рекурсивного алгоритма, который мы приведем схематически — вычисление определителя матрицы размерности $n \times n$. Воспользуемся результатами, полученными в примере 1.5 параграфа 1.1. Для того чтобы вычислить определитель, нам необходимо обнулить все первые элементы строк, начиная со второй, путем вычисления их линейной комбинации с первой строкой матрицы. Это приводит к следующему алгоритму в схеме прямой рекурсии. Обращаем внимание на то, что аргументами функции являются сама квадратная матрица (в реальной программе, очевидно, ее адрес) и ее линейная размерность. Останов рекурсии происходит при $n = 2$, и в этом примере мы предполагаем, что значение аргумента при вызове функции из основной программы больше, либо равно двум.

```

D(A,n)
1 If (n=2)                                (проверка останова рекурсии)
2   then
3     D ← A[1,1]*A[2,2]–A[1,2]*A[2,1] (прямое вычисление D)
4   else
5     For i=2 to n                      (обнуление первых элементов строк)
6       вычислить коэффициент b для i-ой строки
7       вычесть из i-ой строки первую, умноженную на b
8     end For
9     z ← A[1,1]
10    сформировать новую матрицу A, размерностью n–1
11    D ← z*D(A,n–1)                   (рекурсивный вызов)
12 Return (D)
End.

```

Мы проверяем условие останова в строке 1 и вычисляем определитель непосредственно по формуле (1.1.6), если достигли условия останова. В противном случае мы, вычисляя линейную комбинацию

(строки 5–8), обнуляем первые элементы всех строк, кроме первой, запоминаем первый элемент первой строки (строка 9) и формируем новую матрицу меньшей размерности. Стока 11 представляет собой рекурсивный вызов с понижением размерности на единицу и скорректированной матрицей в качестве первого аргумента. Мы привели только *схему* алгоритма, при его детальной разработке необходимо учесть, что элемент $A[1,1]$ исходной или скорректированной матрицы может быть равен нулю.

При всей внешней привлекательности записи рекурсивных алгоритмов (мы имеем в виду, прежде всего, прямую аналогию с записью рекуррентного соотношения) остается непонятным механизм, который обеспечивает реализацию рекурсивного вызова — объяснению этого посвящен следующий параграф.

§ 2. Особенности программных реализаций рекурсивных алгоритмов

Процесс программной реализации рекурсивных алгоритмов обладает, по сравнению с итерационными алгоритмами, специфическими особенностями, а именно — необходимостью организации специальной структуры данных и обслуживания рекурсивных вызовов. При этом организация «правильной» структуры данных лежит на плечах программиста, в то время как механизм обслуживания рекурсивных вызовов обеспечивается языком программирования, точнее, компилятором этого языка, и поддерживается машинными командами компьютера.

Особенности разработки структур данных. Поскольку рекурсивный алгоритм задает во времени последовательность обращений к одному и тому же фрагменту программного кода, то применяемая структура данных должна обеспечить сохранность тех ячеек и массивов, которые будут задействованы алгоритмом после возврата из рекурсивного вызова. В рамках возможностей современных языков процедурного (императивного) программирования могут быть использованы разнообразные варианты [2.3, 2.4], из которых мы остановимся на трех основных.

1. *Глобальная структура данных.* В этом случае пользователь создает глобальную структуру, содержащую помимо описания ячеек и массивов для одного вызова, копии этих описаний для всей цепочки рекурсивных вызовов. Такая структура может представлять собой, например, многомерный массив данных, один из индексов которого соответствует уровню вложенности вызова. Эта структура формируется в вызывающей программе, и рекурсивной функции передается ее адрес, а слой структуры, используемый на данном рекурсивном вызове, определяется его глубиной. Более подробно мы продемонстрируем этот подход в главе 7 на ряде примеров реализации рекурсивных алгоритмов.

2. Локальная структура данных. При этом способе организации данных в теле рекурсивной функции или процедуры описывается необходимая структура данных, и создание копии этой структуры на рекурсивном вызове обеспечивается средствами языка программирования и компилятора. Несколько забегая вперед, отметим, что такой подход приводит, особенно при наличии массивов, к большим затратам памяти в области программного стека, размер которого, вообще говоря, ограничен. Этот способ приводит к тому, что общая структура данных, соответствующая текущей глубине рекурсии, — последовательности рекурсивных вызовов, будет создаваться механизмом обслуживания рекурсивного вызова непосредственно во время выполнения программного кода.

3. Динамическая структура данных. Третий подход к созданию необходимой структуры для программной реализации рекурсивного алгоритма состоит в том, что пользователь средствами языка программирования обращается к операционной системе для выделения необходимого в данный момент ресурса памяти. Этот подход, обладая определенной гибкостью, требует от пользователя аккуратного обращения с динамически выделяемой памятью, своевременного ее освобождения и организации правильной передачи аргументов при рекурсивных вызовах и правильного возвращения полученных результатов при рекурсивных возвратах. Отметим еще одну особенность этого подхода — в зависимости от текущего состояния фрагментации области динамической памяти, обслуживаемой средствами операционной системы, временные затраты на динамическое выделение памяти могут существенно влиять на общее время выполнения программной реализации. Очевидно, что в этом случае также затруднено теоретическое прогнозирование временных характеристик программ.

Модель программного стека. Рассмотрение механизма организации рекурсивных вызовов мы начнем с модели программного стека. Будем предполагать, что операционная система поддерживает специальную область оперативной памяти — программный стек, используя специальный регистр или ячейку памяти — указатель стека, хранящий адрес некоторой ячейки (слова) в этой области. Две специальные команды — «записать в стек» и «читать из стека» организуют работу с этой областью так, что логически мы считаем, что имеем дело со стеком как классической структурой, хотя указанные операции и изменяют адрес указателя стека. Например, команда «записать в стек» уменьшает текущий адрес и записывает по этому адресу содержимое своего операнда. Такой подход гарантирует, что любая операция со стеком не приводит к перезаписи со сдвигом его содержимого, и позволяет считать, что трудоемкость выполнения указанных команд не зависит от объема хранимой информации. Хотя наши команды неявно используют текущий указатель вершины программного стека, тем не менее, мы рассматриваем их как команды с одним операндом, указывающим элементарную ячейку хранения или регистр, содержимое которых либо

помещается в стек, либо считывается в них с вершины стека. Приведем пошаговое описание этих команд.

1. «Записать в стек» <операнд>. Команда уменьшает адрес указателя стека на длину операнда и помещает содержимое операнда, начиная с полученного адреса.

2. «Читать из стека» <операнд>. Команда считывает информацию, расположенную по указателю стека, в операнд и увеличивает указатель на длину операнда.

Механизм вызова процедуры с использованием программного стека. Прежде чем переходить к изучению механизма рекурсивного вызова, рассмотрим обычный в языках процедурного (императивного) программирования механизм, обеспечивающий передачу управления на некоторую процедуру и возврат управления в точку вызова. Напомним, что под процедурой понимается поименованный фрагмент программы, к которому можно обращаться по его имени с возможностью передачи ряда фактических параметров как по значению, так и по ссылке (адресу) для возврата полученных результатов в точку вызова [2.1]. В отличие от функции, процедура не возвращает значения по своему имени. Механизм должен обеспечить сохранение регистров процессора, т. к. при возврате в точку вызова мы должны восстановить их значения, кроме того, в процедуру должны быть некоторым образом переданы фактические параметры [2.1].

Для обеспечения этих действий мы и будем использовать описанный выше программный стек и обслуживающие его команды. Поскольку значения должны помещаться в программный стек в порядке, обратном их выборке, то вначале мы помещаем в стек адрес возврата в точку вызова, затем значения необходимых регистров и передаваемые в процедуру фактические параметры. Для заинтересованных читателей отметим, что в разных языках программирования, в зависимости от принятого соглашения, действия по сохранению регистров выполняются либо в момент вызова процедуры из основной программы, либо самой процедурой при получении управления. Схематично этот механизм проиллюстрирован на рис. 2.1 — мы считаем, что вызывающая программа сама сохраняет регистры, а вызываемая процедура восстанавливает их перед передачей управления (возвратом).

При вызове процедуры вызывающая программа помещает в стек адрес возврата, состояние необходимых регистров процессора, адреса возвращаемых значений и передаваемые параметры, пользуясь командой «записать в стек». После этого выполняется переход по адресу на вызываемую процедуру, которая извлекает переданные фактические параметры (командой «читать из стека»), выполняет необходимые вычисления и помещает результаты по указанным в стеке адресам. При завершении работы вызываемая процедура восстанавливает значения регистров, выталкивает из стека адрес возврата и осуществляет переход по этому адресу, возвращая управление в точку своего вызова (см. рис. 2.1).

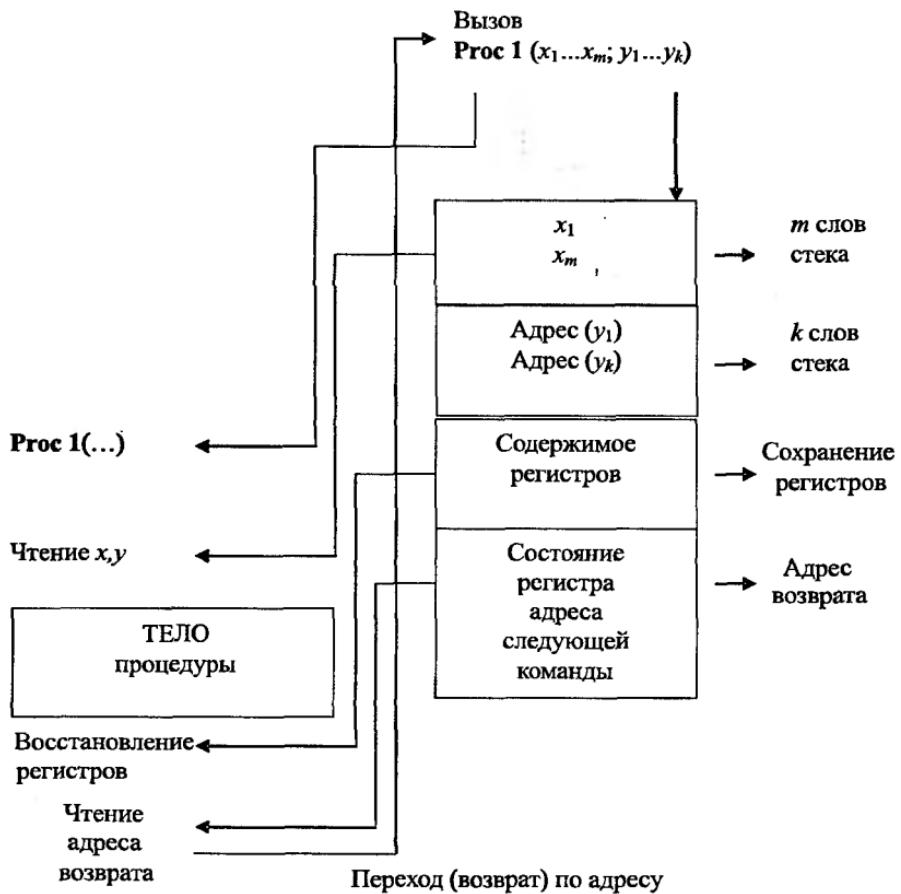


Рис. 2.1. Механизм вызова процедуры с использованием стека

§ 3. Механизм обслуживания рекурсивного вызова

Очевидно, что механизм обслуживания рекурсивного вызова базируется на механизме вызова процедуры. Дополнительные изменения, которые необходимо внести, касаются локальных ячеек рекурсивной функции или процедуры и передачи значения через имя функции. Поскольку мы рекурсивно вызываем тот же фрагмент программного кода, то состояние локальных ячеек рекурсивной функции должно быть также сохранено в программном стеке, равно как и значения регистров. При рекурсивном возврате мы должны обеспечить то же состояние регистров и локальных ячеек, которое было в момент рекурсивного вызова. Это приводит к необходимости использовать программный стек для хранения локальных ячеек и массивов рекурсивной функции. Что касается передачи результата функцией через свое имя, то мы будем



Рис. 2.2. Механизм рекурсивного вызова процедуры

считать, что имя функции также является локальной ячейкой, с той лишь разницей, что в момент возврата функция помещает в стек вычисленное значение, и восстановление локальных ячеек при возврате приведет к передаче этого значения в рекурсивный вызов. С учетом этого, механизм рекурсивного вызова процедуры может быть схематично представлен так, как это показано на рис. 2.2. Опишем этапы рекурсивного вызова и возврата более подробно, нумерация этапов соответствует рис. 2.2.

1. Непосредственно перед рекурсивным вызовом процедура последовательно помещает в программный стек адрес возврата, состояние регистров, содержимое своих локальных ячеек и массивов, и список передаваемых параметров рекурсивного вызова.

2. Выполняется рекурсивный вызов — процедура вызывает сама себя. Отметим, что с точки зрения процессора это всего лишь передача управления на другую машинную команду в программном коде. Этой командой является первая команда процедуры.

3. При получении управления процедура получает доступ к переданным параметрам через программный стек — они либо считываются из стека в регистры, либо процедура имеет прямой адресный доступ к области программного стека.

4. В предположении, что этот вызов является остановом рекурсии, процедура формирует результат в некоторой области оперативной памяти, адрес которой был ей передан при вызове, и восстанавливает состояние локальных ячеек и регистров процессора на момент ее вызова, используя информацию из стека и команду чтения.

5. После этапа 4 на верху стека остается адрес возврата, который и считывается для последующей передачи управления.

6. Выполняется команда перехода по адресу — процедура возвращает управление в тот программный модуль, откуда она была вызвана, но это возврат в тело этой же процедуры, и мы оказываемся в теле процедуры А на предыдущем уровне цепочки рекурсии.

Остановимся более подробно на этапе 4. Мы его описали в предположении останова рекурсии. Если это не так, то в теле процедуры формируется новый рекурсивный вызов, включающий сохранение локальных ячеек и регистров и вызов процедуры с новыми параметрами. Очевидно, что в этот момент информация о «новом» вызове сохраняется в стеке выше информации о «старом» вызове. Это приводит к тому, что хотя мы физически имеем один и тот же программный код, но предыстория вызовов сохраняется в стеке, и, следовательно, мы храним всю информацию о последовательности вызовов, на основе которой организуется цепочка возвратов после выполнения условия останова рекурсии.

Все вышесказанное остается в силе и для рекурсивной функции, за исключением того, что в области сохранения локальных ячеек и массивов будет выделена еще одна дополнительная ячейка, хранящая значение, вычисленное функцией.

Таким образом, механизм программного стека позволяет организовать цепочку рекурсивных вызовов процедур или функций в языке программирования высокого уровня. Хранение в стеке всей текущей информации об этих вызовах позволяет организовать обратную цепочку рекурсивных возвратов. Поэтому, если рассматривать выполнение рекурсивной функции во времени, то в момент останова рекурсии в стеке сохраняется вся информация о предыдущих рекурсивных вызовах, — и в этот момент мы используем наибольший объем памяти в области программного стека. Эта информация будет необходима нам в дальнейшем для исследования емкостной эффективности рекурсивных алгоритмов.

Отметим, что в настоящее время практически все языки программирования для персональных компьютеров, более корректно — обслуживающие их компиляторы, поддерживают механизм рекурсивного вызова, основанный на программном стеке [2.1, 2.5]. В свете вышесказанного этот механизм было бы более правильно назвать

механизмом рекурсивного вызова-возврата. Мы еще раз напоминаем уважаемым читателям, что изложенный здесь механизм является пусть и приближенной к действительности, но все же моделью механизма организации рекурсивного вызова. Компиляторы реальных языков программирования высокого уровня могут иметь собственные особенности организации такого механизма, и мы советуем ознакомиться с тем, как, например, этот механизм реализован в языках *Delphi* или *СИ* (см., например, [2.1, 2.5]).

§ 4. Представление последовательности рекурсивных вызовов в виде дерева рекурсии

Цель этого параграфа — ввести понятие «дерево рекурсии», которое мы будем в дальнейшем использовать как для представления рекурсивных алгоритмов, так и в целях их анализа. Мы уже знаем, что рекурсивный алгоритм порождает цепочку рекурсивных вызовов, которая обрывается при останове рекурсии. Рассмотрим более подробно этот процесс на примере рекурсивной функции, вычисляющей значение факториала для натурального аргумента. Рекуррентное соотношение, задающее функцию факториала, имеет вид:

$$\begin{cases} f(0) = 1; \\ f(1) = 1; \\ f(n) = n \cdot f(n-1), \quad n \geq 2. \end{cases}$$

В принятом нами псевдокоде рекурсивный алгоритм, соответствующий этому рекуррентному соотношению, может быть записан следующим образом:

```

F(n)
1  If (n=0 or n=1)      (проверка возможности прямого вычисления)
2    then
3      F ← 1            (останов рекурсии)
4    else
5      F ← n*F(n-1)    (рекурсивный вызов функции)
6  Return (F)
End.
```

Рассмотрим, как этот алгоритм порождает последовательность рекурсивных вызовов. Информация об этой последовательности хранится в программном стеке, механизм работы которого уже знаком читателям. В данном случае нас будет интересовать, как во времени организуется вычисление этой функции, например, для вызова **F(5)**. Поскольку вызов функции **F(n)** с аргументом **n=5** не приводит к останову рекурсии, то в строке 5 происходит рекурсивный вызов **F(4)** и так далее, до тех пор, пока не произойдет останов рекурсии при **n=1**. Полученная цепочка рекурсивных вызовов показана на рис. 2.3.

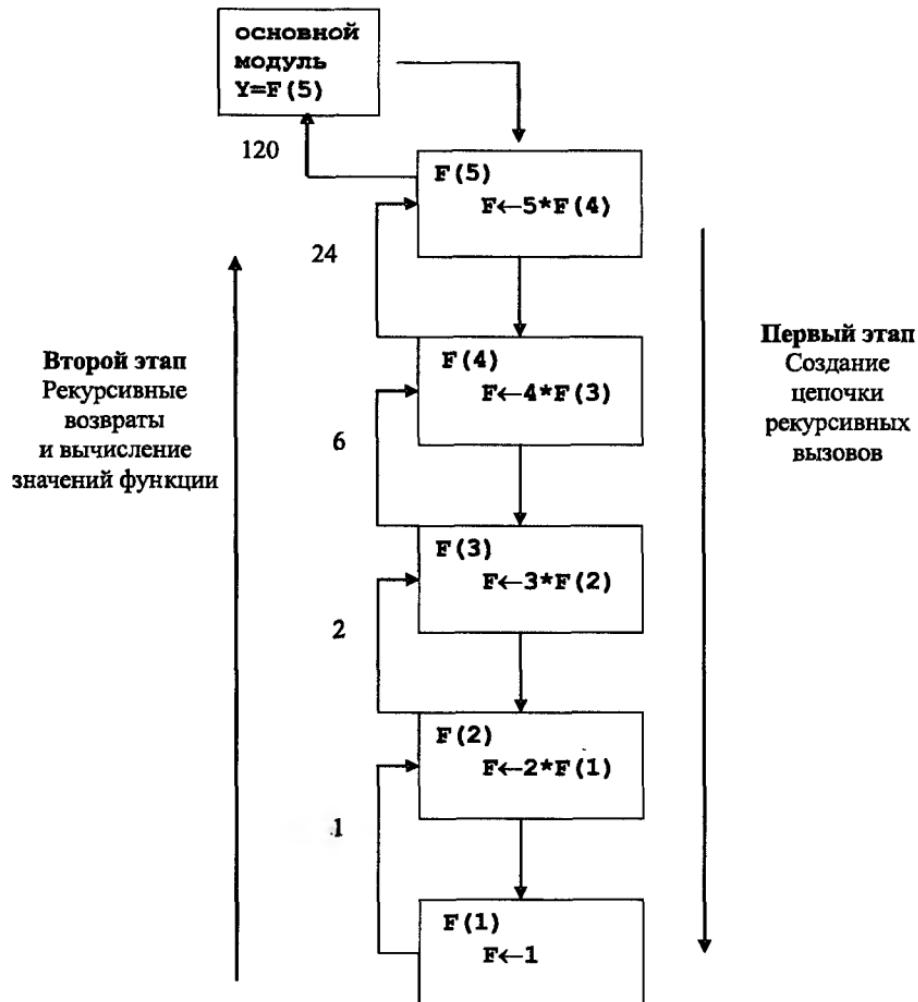


Рис. 2.3. Цепочка рекурсивных вызовов и возвратов факториала для значения $n = 5$

Вычисленное непосредственно значение $F(1)=1$ передается в точку вызова в функцию $F(2)$ и далее вверх, вплоть до вызова $F(5)$, который и возвращает вычисленное значение в основную (относительно этого вызова) программу.

Таким образом, мы получаем во времени два этапа — первый этап, на котором разворачивается цепочка рекурсивных вызовов, вплоть до останова рекурсии, и второй этап — этап возвратов по цепочке рекурсии, на котором и происходит собственно вычисление функции (см. рис. 2.3). Заметим, что на втором этапе цепочка рекурсии все время сокращается, и наибольшая глубина определяется точкой оста-

нова рекурсии. Если мы хотим показать только соподчинение рекурсивных вызовов, то рис. 2.3 может быть преобразован в унарное дерево, показанное на рис. 2.4, в вершинах которого мы указываем аргумент, с которым порожден следующий рекурсивный вызов.

Мы получаем, таким образом, графическую интерпретацию последовательности рекурсивных вызовов, порожденную данным алгоритмом. В дальнейшем будем понимать под *деревом рекурсии* граф, имеющий древовидную структуру, вершины которого отражают рекурсивную функцию или процедуру с данным аргументом, а дуги — непосредственно рекурсивные вызовы.

Отметим, что в литературе [2.6] под деревом рекурсии иногда понимается граф, каждая вершина которого представляет время, необходимое для выполнения отдельно взятой подзадачи, решаемой при одном из многочисленных рекурсивных вызовов функции. Такое дерево имеет вид, аналогичный нашему пониманию дерева рекурсии, разница состоит в том, что в нашем понимании вершина дерева представляет собой вызов рекурсивной функции с данным аргументом, а не время ее выполнения.

В терминах теории графов, дерево, представленное на рис. 2.4, является унарным деревом глубины пять [2.7]. Всегда ли дерево рекурсии, порожденное некоторым рекурсивным алгоритмом, является унарным? Ответ — нет. Рассмотрим, в качестве примера, алгоритм, вычисляющий числа Фибоначчи на основе рекуррентного соотношения из главы 1.

Fb(m)

```

1  If (m=1) or (m=2)          (проверка останова рекурсии)
2    then
3      Fb ← 1                (останов рекурсии)
4    else
5      Fb ← Fb(m-1)+Fb(m-2) (рекурсивные вызовы)
6  Return (Fb)
End.

```

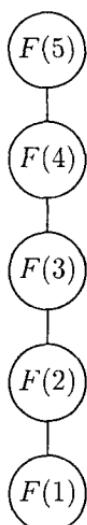


Рис. 2.4. Дерево рекурсии, порожденное алгоритмом вычисления факториала при $n = 5$

Строка 5 данного алгоритма порождает два рекурсивных вызова с различными аргументами. Возвращенные значения складываются, и полученная сумма возвращается как результат этого вызова. Каждый из рекурсивных вызовов, в свою очередь, порожда-

ет также два вызова, вплоть до останова рекурсии. Вид порожденного дерева рекурсии для вызова **Fb(4)** показан на рис. 2.5. В этом случае дерево рекурсии является бинарным. Более сложные деревья рекурсии мы продемонстрируем на примерах рекурсивных алгоритмов в главе 7.

Обращаем внимание уважаемых читателей на то, что дерево, показанное на рис. 2.5, отражает все рекурсивные вызовы, порожденные данным алгоритмом при вызове с аргументом 4. Если мы будем рассматривать процесс порождения этого дерева во времени, то обнаружим, что в фиксированный момент времени имеет место только одна ветвь этого дерева, поскольку второй рекурсивный вызов в строке 5 не будет выполнен до тех пор, пока предыдущий вызов не возвратит вычисленное значение. Порядок рекурсивных вызовов во времени показан нумерацией дуг на рис. 2.5. В терминологии теории графов это означает, что последовательность рекурсивных вызовов порождает дерево рекурсии левым обходом [2.7].

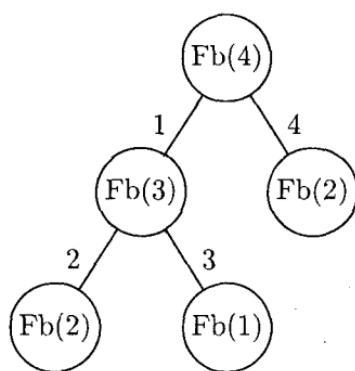


Рис. 2.5. Дерево рекурсии, порожденное алгоритмом вычисления чисел Фибоначчи при $n = 4$

Задачи и упражнения к главе 2

2.1. Разработайте и реализуйте рекурсивный алгоритм, вычисляющий сумму элементов массива. Итерационный алгоритм содержит, очевидно, конструкцию цикла. Что является аналогом цикла в рекурсивном алгоритме?

2.2. Реализуйте на языке программирования рекурсивный алгоритм вычисления функции Аккермана (см. параграф 2.1). Если в тело рекурсивной функции Вы вставите фрагмент, печатающий переданные в функцию аргументы, то сможете проследить порождающую последовательность рекурсивных вызовов. Какие особенности Вы можете при этом отметить?

2.3. Разработайте полностью и реализуйте алгоритм вычисления определителя квадратной матрицы, схема которого приведена в параграфе 2.1. Какой вариант организации данных, из предложенных в параграфе 2.2, Вы при этом выбрали, и почему?

2.4. Определите, какое соглашение о сохранении регистров и локальных ячеек используется в том языке программирования, который Вы используете. Для сравнения ознакомьтесь, как организован механизм рекурсивного вызова-возврата в языках *СИ* и *Delphi*.

2.5. Если Вы владеете Ассемблером и можете разобраться в машинном коде, полученном дизассемблированием EXE-файла для Вашей

программы вычисления определителя матрицы, то сможете ответить на вопросы: какие регистры сохраняются при рекурсивном вызове, и как организована передача параметров.

2.6. Используя программу, разработанную Вами в упражнении 2.2, нарисуйте дерево рекурсии, порождаемое при вычислении функции Аккермана $A(3, 2)$.

Список литературы к главе 2

- 2.1. Архангельский А.Я. Язык Pascal и основы программирования в Delphi. — М.:Бином, 2004. — 496 с.
- 2.2. Гудман С., Хидетиэми С. Введение в разработку и анализ алгоритмов. — М.: Мир, 1981. — 368 с.
- 2.3. Вирт Н. Алгоритмы и структуры данных: Пер. с англ. — 2-е изд., испр. — СПб.: Невский диалект, 2001. — 352 с.
- 2.4. Бакнел Дж.Н. Фундаментальные алгоритмы и структуры данных в Delphi. — СПб.:ДиаоСофтЮП, 2003. — 560 с.
- 2.5. Савич У. C++ во всей полноте. — СПб.: Питер, 2004. — 784 с.
- 2.6. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ, 2-е изд. — М.: Изд. дом «Вильямс», 2005. — 1296 с.
- 2.7. Хаггарти Р. Дискретная математика для программистов. — М.: Техносфера, 2005. — 400 с.

Глава 3

МЕТОДЫ РАЗРАБОТКИ РЕКУРСИВНЫХ АЛГОРИТМОВ

Введение. В этой главе мы даем краткое описание некоторых методов разработки рекурсивных алгоритмов. Прежде всего мы хотели бы отметить, что, по нашему глубокому убеждению, создание нового алгоритма — это искусство, базирующееся на математике, интуиции и «озарении». Недаром в названии алгоритма отражается фамилия его автора — алгоритм Тарьяна, алгоритм Дейкстры, алгоритм Беллмана-Форда, алгоритм Карацубы и т. д. Рассматривая множество реально применяемых алгоритмов, можно выделить группу алгоритмов, непосредственно реализующих математические методы, например, метод Рунгे-Кутта. В этом случае алгоритм является простым переложением на язык алгоритмических конструкций математически обоснованного численного метода решения задачи. Другую группу составляют алгоритмы, в основе которых лежит интеллект их авторов — алгоритмы Джарвиса и Грэхема для построения охватывающего контура, алгоритм Прима для поиска оставного дерева минимального веса и др. К третьей группе можно отнести алгоритмы, полученные на основе применения к решаемой задаче некоторых «универсальных» методов разработки алгоритмов.

Эти методы, о которых собственно и пойдет речь в настоящей главе, не являются методами в строго математическом понимании этого термина. Термин «метод» по отношению к разработке алгоритмов — это скорее дань исторической традиции, а не констатация гарантии быстрой и эффективной разработки. Под этим термином понимаются приемы или способы, пользуясь которыми, может быть, можно построить алгоритм решения задачи при условиях, что, во-первых, сам метод теоретически применим к этой задаче, и, во-вторых, что Вам удастся адаптировать этот метод к ее особенностям. Речь скорее идет о некоторых общих подходах, не содержащих конкретных рецептов, на базе которых, вероятно, могут быть разработаны новые алгоритмы. В этом случае конкретная разработка такого метода по отношению к реаль-

ной задаче требует специальных подходов и математических обоснований — мы снова констатируем интеллектуальный аспект в процессе разработки алгоритмов.

Применительно к рекурсивным алгоритмам такими «универсальными» и достаточно распространенными методами являются метод рекуррентных соотношений, метод декомпозиции и метод динамического программирования. Однако в каждом конкретном случае их применение не является автоматическим. Мы недаром поставили слово «универсальный» в кавычки — для каждого метода существуют некоторые границы его применимости, и чем более «универсально» сформулирован сам метод, тем интеллектуально сложнее разработка реального алгоритма на его основе. Чтобы убедиться в этом, попробуйте выполнить упражнения и задачи к этой главе, и мы надеемся, что Вы реально почувствуете разницу двух процессов — алгоритмирования, как интеллектуального процесса разработки алгоритма решения задачи, и программирования, как технического процесса записи уже готового алгоритма на некотором языке программирования.

§ 1. Метод рекуррентных соотношений

Идея метода рекуррентных соотношений чрезвычайно проста — мы, используя некоторые рассуждения, получаем рекуррентное соотношение, обеспечивающее решение нашей задачи, и на основе этого рекуррентного соотношения разрабатываем рекурсивный алгоритм. Отметим, что фраза «используя некоторые рассуждения» совершенно не определяет конкретный метод получения рекуррентного соотношения для определенной задачи. Мы можем лишь уточнить, что эти рассуждения должны отражать наше рекурсивное понимание структуры задачи, т. е. следовать схеме понижения аргумента или размерности. Решение для некоторой размерности задачи или для некоторого аргумента функции должно быть сформулировано на основе ее сведения к задачам меньшей размерности или функциям с меньшим значением аргумента. Условие останова рекурсии позволяет решить задачу при некоторых малых размерностях или вычислить функцию при начальных значениях аргумента. Разнообразные примеры применения этого метода к различным задачам уже были продемонстрированы в главе 1. Мы хотим привести еще один пример с комментариями к этапам разработки рекуррентного соотношения.

Пример 3.1. Рассмотрим задачу вычисления количества разбиений положительного целого числа. Разбиение положительного целого числа t — это его представление в виде суммы целых положительных чисел. Классической счетной задачей является определение функции, задающей количество разбиений числа t без учета порядка слагаемых

[3.1]. Приведем все разбиения числа $m = 6$ в порядке убывания первого слагаемого, прямой подсчет дает значение $P(6) = 11$.

6;

5 + 1;

4 + 2, 4 + 1 + 1;

3 + 3, 3 + 2 + 1, 3 + 1 + 1 + 1;

2 + 2 + 2, 2 + 2 + 1 + 1, 2 + 1 + 1 + 1 + 1;

1 + 1 + 1 + 1 + 1 + 1.

К сожалению, источник [3.1] не указывает автора идеи, предложившего следующий подход для рекурсивного вычисления количества разбиений числа m . Первым, и не совсем очевидным, шагом является выражение функции $P(m)$ через другую функцию $Q(m, n)$, которая определяется как число разбиений целого m со слагаемыми, не превышающими n . Если мы сможем вычислить $Q(m, n)$ для любых аргументов, то эта функция определяет $P(m)$, т. к. $P(m) = Q(m, m)$. На втором шаге, в соответствии с идеей метода, нам необходимо определить аргументы, с которыми функция $Q(m, n)$ может быть вычислена непосредственно, и определить, как $Q(m, n)$ задается через свои предыдущие значения. Этот шаг выполняется на основе «несложных рекурсивных рассуждений» — это цитата из [3.1] (заметим, что рекурсивные рассуждения требуют рекурсивного мышления). Эти рассуждения приводят к следующим пяти соотношениям:

1. $Q(m, 1) = 1$, поскольку существует только одно разбиение числа m с наибольшим слагаемым, равным единице, а именно $m = 1 + 1 + \dots + 1$.

2. $Q(1, n) = 1$, это очевидно, т. к. существует только одно разбиение числа 1, которое не зависит от величины наибольшего слагаемого n .

3. $Q(m, n) = Q(m, m)$, $m \leq n$. Совершенно ясно, что никакое разбиение числа m не может содержать каких-либо слагаемых n , больших, чем m .

4. $Q(m, m) = 1 + Q(m, m - 1)$. Есть только одно разбиение числа m со слагаемым, равным m , все другие разбиения m имеют наибольшее слагаемое $n \leq m - 1$.

5. $Q(m, n) = Q(m, n - 1) + Q(m - n, n)$. Это основное рассуждение рекурсии — любое разбиение числа m с наибольшим слагаемым, меньшим или равным n , или не содержит n в качестве слагаемого — в этом случае данное разбиение подсчитывается функцией $Q(m, n - 1)$, или содержит n , но при этом остальные слагаемые образуют разбиение числа $m - n$, и подсчитываются функцией $Q(m - n, n)$.

Полученные результаты позволяют записать следующее рекуррентное соотношение, определяющее рекурсивно заданную функцию

$Q(m, n)$:

$$\begin{cases} Q(m, 1) = 1; \\ Q(1, n) = 1; \\ Q(m, n) = Q(m, m), \quad m \leq n; \\ Q(m, m) = 1 + Q(m, m - 1), \quad n = m; \\ Q(m, n) = Q(m, n - 1) + Q(m - n, n). \end{cases} \quad (3.1.1)$$

На основе полученного рекуррентного соотношения может быть разработан рекурсивный алгоритм для вычисления количества разбиений числа m в виде процедурно реализованной рекурсивной функции $\mathbf{Q}(m, n)$, вызов которой с аргументами $\mathbf{Q}(m, m)$ при заданном значении m и решает поставленную задачу.

$\mathbf{Q}(m, n)$

```

If (m=1) or (n=1)           (проверка останова рекурсии)
  then
    Q ← 1                   (прямое вычисление Q)
  If (m < n)
    then
      Q ←  $\mathbf{Q}(m, m)$        (рекурсивный вызов, m < n).
  If (m=n)
    then
      Q ← 1+ $\mathbf{Q}(m, m-1)$    (рекурсивный вызов, m=n)
    else
      Q ←  $\mathbf{Q}(m, n-1)+\mathbf{Q}(m-n, n)$  (рекурсивный вызов, m > n)
  Return (Q)
End.
```

Заметим, что поскольку функция $\mathbf{Q}(m, n)$ имеет два аргумента, то в теле нашего алгоритма три различных рекурсивных вызова соответствуют ситуациям, когда аргументы равны, первый аргумент меньше и первый аргумент больше второго. Это не общее свойство рекурсивных функций двух аргументов, а отражение особенностей поведения функции $\mathbf{Q}(m, n)$, и, следовательно — решаемой задачи.

Мы не обсуждаем вопроса о том, насколько велики вычислительные затраты этого алгоритма, и существует ли другой алгоритм, может быть, не рекурсивный, который быстрее вычисляет $\mathbf{Q}(m, n)$. Более общий вопрос формулируется следующим образом — можно ли, зная, что алгоритм разработан методом рекуррентных соотношений, говорить о его вычислительной эффективности? Ответа в общем виде нет — в частных примерах мы можем говорить о том, что, например, для чисел Фибоначчи мы получим неоправданно большие вычислительные затраты, а при вычислении факториала — аналогичную итерационной реализации асимптотическую оценку.

Пример 3.2. Мы приводим этот пример с целью показать, как в примитивной модели вычислений рекурсивными алгоритмами могут

быть реализованы обычные арифметические операции. Наша модель вычислений оперирует с целыми неотрицательными числами и разрешает операции сравнения и нахождения предыдущего и последующего числа, опираясь на аксиомы Пеано. Приведем формулировку одной из аксиом Пеано [3.2]: «Для каждого натурального числа x имеется, и притом только одно, натуральное число, называемое его последующим и обозначаемое x' ». В целях удобства записи, будем обозначать операцию, задающую для целого неотрицательного числа n его последующее число через $n + 1$, а операцию, задающую для целого положительного числа n его предыдущее число, через $n - 1$. Наша задача состоит в том, чтобы разработать рекурсивный алгоритм для вычисления суммы двух чисел n, m в этой модели вычислений.

Условие останова рекурсии легко формулируется: $n + m = n, m = 0$. Выберем число m в качестве аргумента рекурсии — нам необходимо записать сумму при уменьшении аргумента на единицу с использованием разрешенных в нашей модели операций. Решение достаточно очевидно:

$$n + m = (n + 1) + (m - 1).$$

Проведенные рассуждения позволяют записать рекуррентное соотношение для рекурсивно заданной функции $S(n, m)$, значением которой является сумма ее аргументов:

$$\begin{cases} S(n, m) = n, & m = 0; \\ S(n, m) = S((n + 1), (m - 1)), & m > 0. \end{cases}$$

Надеемся, что читатели без труда разработают соответствующий рекурсивный алгоритм. Еще целый ряд интересных примеров разработки алгоритмов методом рекуррентных соотношений читатели смогут найти в [3.3], полезным будет также разработать рекурсивные алгоритмы самостоятельно, выполнив упражнения к этой главе.

§ 2. Метод декомпозиции

Чтобы перенести сотню кирпичей с одного места на другое, мы, скорее всего, будем переносить за один раз несколько кирпичей, а не всю сотню сразу. Таким образом, вместо того, чтобы решать задачу сразу, мы несколько раз решаем более простую задачу с понижением размерности — собственно говоря, эта интуитивно здравая идея и лежит в основе метода декомпозиции. Другое название этого метода — метод «разделяй и властвуй» [3.4]. Общая схема метода может быть описана как последовательность следующих шагов.

1. Шаг разделения задачи. На этом шаге выбирается способ разделения задачи на некоторое количество подзадач меньшей размерности.
2. Шаг решения полученных подзадач. Это рекурсивный шаг — мы рассматриваем каждую подзадачу как задачу определенной размерности, и разделяем ее на собственные подзадачи, выполняя снова шаг

1 (рекурсия) до тех пор, пока не получим такую размерность, при которой решение может быть найдено непосредственно.

3. Шаг останова рекурсии. На этом шаге выполняется непосредственное решение полученных подзадач для малых размерностей.

4. Шаг объединения решений. На этом шаге полученные решения подзадач меньшей размерности при возврате в точку рекурсивного вызова объединяются в решение задачи текущей размерности.

Эти основные шаги метода декомпозиции требуют некоторых комментариев. Во-первых, идея метода ничего не говорит о том, как эти шаги должны быть реализованы в конкретной задаче. Адаптация метода к конкретной задаче требует, как правило, серьезных размышлений. Во-вторых, условием применения метода является свойство аддитивности решений, полученных для подзадач меньшей размерности. Строго говоря, нам необходимо доказать, что, объединяя решения, мы получим правильный ответ, в особенности это касается задач оптимизации в постановке поиска глобального оптимума. Классическим примером является задача коммивояжера [3.5] — простое объединение двух оптимальных путей для графов половинной размерности вообще не является решением исходной задачи. В третьих, для одной и той же задачи могут быть предложены различные идеи ее разделения, и здесь необходимо оценить, к каким сложностям приводят каждая из этих идей на этапе объединения решений, который, как правило, является наиболее сложным.

Основное преимущество метода декомпозиции состоит в том, что он позволяет получить алгоритмы с хорошими асимптотическими оценками трудоемкости. Именно этот подход позволил А.Л. Кацаубе в 1962 г. получить алгоритм умножения длинных целых чисел с оценкой, лучше чем n^2 , а В. Штрассену в 1969 г. — алгоритм умножения квадратных матриц с оценкой, лучше чем n^3 [3.4]. Соответствующее обоснование асимптотических оценок трудоемкости алгоритмов, разработанных методом декомпозиции, читатель найдет в главе 6.

Приведем два примера применения метода декомпозиции.

Пример 3.3. Исторически одним из первых применений метода считается алгоритм сортировки слиянием, приписываемый Дж. фон Нейману — мы детально анализируем этот алгоритм в главе 7. Рассмотрим задачу сортировки массива чисел из n элементов и обсудим шаги метода декомпозиции применительно к этой задаче.

1. Шаг разделения задачи. Мы организуем деление на две подзадачи — массив разделяется на два массива, содержащих $\lfloor n/2 \rfloor$ и $\lceil n/2 \rceil$ элементов.

2. Шаг решения полученных подзадач. Если условие останова рекурсии не выполнено, то полученные массивы вновь разделяются на два массива — это шаг порождения дерева рекурсии.

3. Шаг останова рекурсии. В классическом изложении останов рекурсии происходит при длине массива, равной единице — массив из одного числа очевидно отсортирован.

4. Шаг объединения решений. На цепочке рекурсивных возвратов мы получаем два отсортированных массива, которые должны быть объединены в один. Этот шаг выполняется специальным алгоритмом слияния сортированных массивов, по которому получил название и сам алгоритм сортировки.

Эти шаги порождают асимптотически оптимальный алгоритм для задачи сортировки сравнениями со сложностью $\Theta(n \ln n)$. Из-за большого коэффициента при главном порядке область его рациональной применимости начинается с длин массива порядка 100, и более точно определяется особенностями реализации, в основном — выбранным алгоритмом слияния двух сортированных массивов.

Пример 3.4. Второй пример — задача умножения квадратных матриц. Для изложения идеи мы рассмотрим случай, когда линейный размер квадратной матрицы — n является степенью двойки. Шаги метода декомпозиции могут быть реализованы следующим образом.

1. Шаг разделения задачи. Мы организуем деление пополам линейного размера матрицы. Поскольку значение n является степенью двойки, то на любом шаге деления значение $n/2$ является целым числом. Таким образом, каждая из перемножаемых матриц делится на четыре подматрицы одинаковой размерности

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

где подматрицы результата задаются соотношениями

$$\begin{aligned} r &= a \times e + b \times g; \\ s &= a \times f + b \times h; \\ t &= c \times e + d \times g; \\ u &= c \times f + d \times h. \end{aligned} \tag{3.2.1}$$

Эти уравнения приводят к необходимости решения восьми подзадач размерности $n/2$. Операции умножения в (3.2.1) есть операции умножения матриц.

2. Шаг решения полученных подзадач. Если после разделения полученные матрицы размерностью $n/2 \times n/2$ не являются матрицами, состоящими из одного элемента, то полученные матрицы вновь разделяются на четыре подматрицы — это шаг порождения дерева рекурсии.

3. Шаг останова рекурсии. Если на текущем вызове алгоритма матрицы имеют размер 2×2 , то результат может быть вычислен непосредственно по формуле (3.2.1), обозначения в которой в данном случае интерпретируются как обозначения элементов матриц.

4. Шаг объединения решений. По цепочке рекурсивных возвратов мы получаем восемь результирующих матриц, имеющих размер $n/2 \times n/2$ относительно размера текущей матрицы. В соответствии с (3.2.1) нам необходимо выполнить сложение матриц и заполнение соответствующих позиций матрицы результата.

Элементарный анализ этих шагов позволяет получить рекуррентное соотношение, задающее суммарное количество выполненных алгоритмом операций умножения и сложения над числами — элементами матриц

$$S(n) = 8S(n/2) + \Theta(n^2).$$

Асимптотическая оценка решения этого рекуррентного соотношения $S(n) = \Theta(n^3)$, так что в таком подходе алгоритм, разработанный методом декомпозиции, оказывается по порядку таким же, как и обычный алгоритм умножения матриц. Тем не мене, именно этот метод позволил В. Штрассену получить алгоритм с асимптотически лучшей оценкой — $\Theta(n^{\log_2 7})$. Если Вы придумаете, как вместо восьми умножений матриц размером $n/2 \times n/2$ можно выполнить только семь, то Вы получите результат, аналогичный алгоритму Штрассена. Достаточно подробно метод Штрассена изложен, например, в [3.4].

С использованием метода декомпозиции в настоящее время получены эффективные алгоритмы решения целого ряда задач. Основная их особенность — «плата» за асимптотически лучшую функцию в оценке сложности большим коэффициентом при главном порядке. Тем самым, эти алгоритмы становятся рационально применимыми, начиная с относительно больших размерностей.

§ 3. Метод динамического программирования

Метод динамического программирования был предложен и обоснован Р. Беллманом в начале 1960-х годов [3.6]. Первоначально метод создавался в целях существенного сокращения перебора для решения целого ряда задач экономического характера, формулируемых в терминах задач целочисленного программирования [3.5]. Однако Р. Беллман и Р. Дрейфус в [3.6] показали, что он применим к достаточно широкому кругу задач, в том числе к задачам вариационного исчисления, поиску нулей функций и т. д. В общем виде метод ориентирован на поиск оптимума целевой функции или функционала в некоторой ограниченной области многомерного пространства. Предложенный Р. Беллманом метод не является универсальным, и условия его применения требуют, чтобы целевой функционал представлял собой аддитивную функцию, т. е.

$$f(\mathbf{x}) = \sum_{i=1}^n g_i(x_i), \mathbf{x} = (x_1, \dots, x_n). \quad (3.3.1)$$

Заметим, что требование аддитивности может быть ослаблено до требования сепарабельности. Приведем описание идеи этого метода, опираясь на оригинальное изложение и терминологию его автора в экономической интерпретации метода динамического программирования

[3.6]. В элементарной постановке задачи ограничения, задающие область поиска экстремума для целевого функционала, имеют вид

$$\sum_{i=1}^n x_i = C, \quad x_i \geq 0 \quad \forall i = \overline{1, n}.$$

Эти ограничения рассматриваются как ограничения на общий ресурс, который должен быть распределен по n инвестиционным процессам, приносящим некоторый доход. Значение дохода задается функциями $g_i(x_i)$, $i = \overline{1, n}$ в условиях целочисленности и неотрицательности значений x_i . Для решения задачи максимизации целевого функционала (3.3.1) — $f(\mathbf{x}) \rightarrow \max$, вместо рассмотрения одной задачи с данным количеством ресурса и фиксированным числом процессов рассматривается целое семейство задач, в которых число n принимает все возможные целые значения от 0 до n . Если исходная задача представляет собой статический процесс распределения ограниченного ресурса, то подход Р. Беллмана переводит ее в динамический процесс, требуя распределения ресурса последовательно по каждому процессу, что собственно и нашло отражение в названии метода — динамическое программирование [3.6].

Максимум целевого функционала $f(\mathbf{x})$ в указанной области зависит от количества процессов n , и от общего ограничения ресурса C . Эта зависимость в подходе динамического программирования записывается явно путем задания последовательности функций $\{f_i(c)\}$, $i = \overline{1, n}$, $0 \leq c \leq C$, следующим образом:

$$f_i(c) = \max_{x_1, \dots, x_i} f(x_1, \dots, x_i), \quad x_i \geq 0, \quad \sum_{j=1}^i x_j = c.$$

При этом функция $f_i(c)$ выражает оптимальный доход, получаемый от распределения ресурса c по i процессам. В двух частных случаях значения этой функции вычисляются элементарно. В разумном предположении, что доход каждого процесса от нулевого ресурса равен нулю — $g_i(0) = 0$, $\forall i = \overline{1, n}$, очевидно, что $f_i(0) = 0$, $\forall i = \overline{1, n}$. Также очевидно, что при значениях $c \geq 0$ функция $f_1(c) = g_1(c)$.

Для совместного распределения ресурса между несколькими процессами достаточно просто находятся рекуррентные соотношения, связывающие $f_m(c)$ и $f_{m-1}(c)$ для произвольных значений m и c . Если x_m — количество ресурса, назначенное для процесса с номером m , то остающееся количество — $(c - x_m)$ должно быть оптимально распределено для получения максимального дохода от оставшихся $m - 1$ процессов. Таким образом, при некотором значении x_m совокупный доход от распределения по m процессам составит

$$g_m(x_m) + f_{m-1}(c - x_m).$$

Очевидно, что оптимальным будет такой выбор значения x_m , который максимизирует эту функцию, что и приводит к следующему рекуррентному соотношению, которое называется основным функциональным уравнением метода динамического программирования [3.6]:

$$\begin{cases} f_1(c) = g_1(c); \\ f_m(c) = \max_{0 \leq x_m \leq c} [g_m(x_m) + f_{m-1}(c - x_m)], \quad \forall m = \overline{2, n}, \quad c \geq 0. \end{cases} \quad (3.3.2)$$

Таким образом, задача оптимизации в многомерном пространстве сводится к последовательности задач одномерной оптимизации, что существенно сокращает трудоемкость получения решения. Но условия применимости метода накладывают ограничения аддитивности (сепарабельности) на целевой функционал. Кроме того, система ограничений не должна быть слишком сложной и должна допускать построение рекуррентно связанных между собой функций Беллмана через возможно меньшее число аргументов, которые определяются «существенными» ограничениями задачи.

Рекуррентное соотношение (3.3.2) позволяет сразу получить рекурсивный алгоритм. Проблемы адаптации метода к конкретной задаче состоят, прежде всего, в доказательстве применимости самого метода и интерпретации постановки задачи в терминах динамического программирования, т. е. как задачи поиска экстремума аддитивной или сепарабельной целевой функции в замкнутой области многомерного пространства, заданной некоторой системой ограничений. Непосредственно этот подход мы иллюстрируем в главе 7 на примере задачи одномерной оптимальной упаковки.

Более широкая трактовка метода динамического программирования состоит в том, что функциональное уравнение Беллмана может быть построено и в случае, когда значение функции для больших значений аргументов может быть получено на основе минимаксного выбора из некоторого ряда ее значений для меньших значений аргументов или линейных комбинаций этих значений с другими функциями. Эту ситуацию мы хотим проиллюстрировать на одном из классических примеров применения метода динамического программирования — задачи вычисления редакционного расстояния.

Пример 3.5. Редакционное расстояние. Мы фиксируем некоторый алфавит символов, например, латинский, и рассматриваем строки над этим алфавитом. В целом ряде практически важных задач (проверка правописания, задачи поиска в текстовых базах данных, задачи исследования ДНК и т. д.) требуется измерить различие между строками, которое по аналогии с термином в метрических пространствах называется расстоянием. Существует несколько различных способов формализации понятия расстояния между строками, один из них, наиболее общий и простой, носит название редакционного расстояния. Термин и идея предложены В.И. Левенштейном [3.7] и достаточно часто редакционное расстояние называется расстоянием Левенштейна.

Идея основана на преобразовании одной строки в другую с помощью фиксированных операций редактирования — вставки символа в первую строку (I), удаления символа из первой строки (D), замены символа в первой строке (R) и «не операции» над правильным символом (M). Эти операции совместно с механизмом их реализации образуют специализированную модель вычислений. Последовательность этих операций — алгоритм преобразования строк — называется редакционным предписанием. Рассмотрим пример — мы хотим преобразовать строку $vintner$ в строку $writers$. Одна из возможных последовательностей операций редактирования (редакционное предписание) имеет вид

R	I	M	D	M	D	M	M	I
v		i	n	t	n	e	r	
w	r	i		t		e	r	s

По определению, редакционное расстояние между двумя строками есть минимальное число необходимых редакционных операций — вставок, удалений и замен, необходимых для преобразования первой строки во вторую. При этом совпадения символов не являются операциями, которые учитываются в редакционном расстоянии, поэтому, если строки совпадают, то их редакционное расстояние равно нулю для любой длины строк. В нашем примере мы выполняем 5 редакционных операций, но не гарантируем, что это значение является минимальным.

Следуя [3.8], введем следующие обозначения. Будем предполагать, что исходные строки символов S_1 и S_2 заданы посимвольно массивами $S_1[1 \dots n]$ и $S_2[1 \dots m]$. Введем в рассмотрение функцию $D(i, j)$, значением которой является редакционное расстояние между подстроками $S_1[1 \dots i]$ и $S_2[1 \dots j]$. Функция $D(i, j)$ определяет минимальное число редакционных операций для преобразования первых i символов строки S_1 в первые j символов строки S_2 . Таким образом, редакционное расстояние между строками S_1 и S_2 в точности равно $D(n, m)$.

Следуя подходу динамического программирования, мы должны определить, при каких аргументах функция $D(i, j)$ может быть вычислена непосредственно и как значение $D(n, m)$ выражается через минимаксный выбор ее значений для меньших аргументов. Заметим, что этот подход очевидно предполагает, что для вычисления $D(n, m)$ нам необходимо вычислить все предыдущие значения $D(i, j)$, $i = 0, n$, $j = \overline{0, m}$. Непосредственное вычисление функции $D(i, j)$ возможно в том случае, если одна из строк является пустой, тогда

$$D(i, 0) = i; \quad D(0, j) = j. \quad (3.3.3)$$

Действительно, для перевода строки из i символов в пустую строку, единственным способом является удаление этих символов — мы выполняем i операций удаления. Аналогично для преобразования нуля символов первой строки в j символов второй строки нам необходимо выполнить ровно j операций вставки. Основной шаг метода динами-

ческого программирования определяется следующей теоремой, доказательство которой мы приводим в соответствии с [3.8].

Теорема 3.1. Если значения i и j строго положительны, то

$$D(i, j) = \min \{D(i-1, j) + 1, D(i, j-1) + 1, D(i-1, j-1) + t(i, j)\}, \quad (3.3.4)$$

где

$$t(i, j) = \begin{cases} 1, & S_1[i] \neq S_2[j]; \\ 0, & S_1[i] = S_2[j]. \end{cases}$$

Доказательство. Рассмотрим редакционное предписание, преобразующее подстроку $S_1[1 \dots i]$ в $S_2[1 \dots j]$ за минимальное число редакционных операций. Нас интересует последняя операция этого предписания, которая может быть операцией I, D, R или M . Рассмотрим все возможные случаи.

Если это операция I , то последняя операция — это вставка символа $S_2[j]$ в конец преобразуемой первой строки. Предыдущие операции этого предписания должны обеспечить минимальное число операций для преобразования $S_1[1 \dots i]$ в $S_2[1 \dots j-1]$. По определению, это последнее преобразование требует $D(i, j-1)$ редакционных операций. Следовательно, если последняя редакционная операция — I , то $D(i, j) = D(i, j-1) + 1$.

Рассуждая аналогично для последней операции D в редакционном предписании, получим, что поскольку это операция удаления последнего символа из $S_1[1 \dots i]$, то предыдущие операции предписания должны оптимально преобразовывать $S_1[1 \dots i-1]$ в $S_2[1 \dots j]$. По определению, это последнее преобразование требует $D(i-1, j)$ редакционных операций. Следовательно, если последняя редакционная операция — D , то $D(i, j) = D(i-1, j) + 1$.

Если последняя операция предписания — R , то эта операция заменяет $S_1[i]$ на $S_2[j]$, а предыдущие операции предписания должны оптимально преобразовывать $S_1[1 \dots i-1]$ в $S_2[1 \dots j-1]$. В этом случае $D(i, j) = D(i-1, j-1) + 1$.

Если последняя операция предписания — M , то $S_1[i] = S_2[j]$, и поскольку эта операция не учитывается в редакционном расстоянии, в этом случае $D(i, j) = D(i-1, j-1)$.

Вводя в рассмотрение дополнительную функцию $t(i, j)$, мы можем объединить два последних случая в один: если последний символ предписания R или M , то $D(i, j) = D(i-1, j-1) + t(i, j)$. Этим исчерпываются все возможные случаи для последней операции в редакционном предписании.

Каким образом можно преобразовать $S_1[1 \dots i]$ в $S_2[1 \dots j]$? Можно преобразовать $S_1[1 \dots i]$ в $S_2[1 \dots j-1]$ за $D(i, j-1)$ операций и вставить символ $S_2[j]$ в конце, что дает $D(i, j-1) + 1$ редакционных операций. Можно преобразовать $S_1[1 \dots i-1]$ в $S_2[1 \dots j]$ за $D(i-1, j)$ операций и удалить $S_1[i]$ в конце, что дает в итоге $D(i-1, j) + 1$ редакционных операций. И, наконец, очевидно, как выполнить

такое преобразование за $D(i-1, j-1) + t(i, j)$ операций. Принцип оптимальности Беллмана гласит, что мы должны выбрать минимальное значение из этих трех вариантов, выше мы показали, что никаких других вариантов не существует. Отсюда следует, что $D(i, j)$ определяется формулой (3.3.4) и теорема доказана. Конец доказательства.

Таким образом, на основе доказанной теоремы мы получили функциональное уравнение метода динамического программирования для задачи вычисления редакционного расстояния в виде следующего рекуррентного соотношения:

$$\begin{cases} D(i, 0) = i, & j = 0; \\ D(0, j) = j, & i = 0; \\ D(i, j) = \min\{D(i-1, j)+1, D(i, j-1)+1, D(i-1, j-1)+t(i, j)\}, \\ & i > 0, j > 0. \end{cases} \quad (3.3.5)$$

На основе этого рекуррентного соотношения достаточно просто может быть разработан рекурсивный алгоритм, решающий задачу о вычислении редакционного расстояния между строками. Небольшие дополнения позволяют получить и само оптимальное редакционное предписание, подробности в [3.8].

Мы обращаем внимание читателей на то, что в большинстве случаев метод динамического программирования требует математического обоснования для функционального уравнения, равно как и математического обоснования корректности его применения для данной задачи. Исторически метод динамического программирования был одним из первых методов, который позволил получить за приемлемое время точное решение целого ряда переборных задач, сводящихся к задачам целочисленного программирования [3.5].

Задачи и упражнения к главе 3

3.1. Нарисуйте дерево рекурсии, порождаемое алгоритмом вычисления количества разбиений целого числа для вызова **Q(6,6)**. Соответствует ли порядок вычисления значения **Q(6,6)** порядку перечисления различных вариантов сумм, приведенных в числовом примере из параграфа 3.1?

3.2. Предположим, что Вы хотите оплатить почтовое отправление, стоимость которого составляет 96 копеек, а в Вашем распоряжении только почтовые марки достоинством 4, 6 и 10 копеек. Сколькими разными способами можно оплатить почтовое отправление? Введем понятие ограниченного разбиения целого числа в виде суммы некоторых фиксированных чисел. Пусть функция $P_{4,6,10}(m)$ задает количество ограниченных разбиений числа m только с использованием чисел 4, 6 и 10. Решением нашей задачи является значение $P_{4,6,10}(96)$. Обозначим для краткости $S(m) = P_{4,6,10}(m)$. Разработайте рекурсивный алгоритм для вычисления $S(m)$ методом рекуррентных соотношений. Одним из

возможных подходов к разработке рекуррентного соотношения, задающего $S(m)$, является обобщение рекуррентного соотношения для чисел Фибоначчи.

3.3. Рассмотрим последовательности длины n , состоящие из нулей и единиц. Мы можем интерпретировать их как двоичные числа, содержащие n бит. Пусть $G(n)$ — функция, значением которой является количество последовательностей длины n , не содержащих двух или более единиц подряд. Получите рекуррентное соотношение, задающее функцию $G(n)$. Какие аналогии с уже известными рекурсивно заданными функциями Вы можете провести?

3.4. Опираясь на алгоритм вычисления суммы двух целых чисел, полученный в примере 3.2, разработайте рекурсивный алгоритм для вычисления функции $M(n, m)$, значением которой является произведение ее аргументов — целых неотрицательных чисел.

3.5. Разработайте методом декомпозиции алгоритм построения выпуклого охватывающего контура для n точек, заданных своими координатами. Условие останова рекурсии может быть сформулировано, например, на основе следующего рассуждения: для трех точек, не лежащих на одной прямой, мы получаем треугольник, который и является выпуклым охватывающим контуром. Какие способы Вы можете предложить для реализации шага разделения задачи?

3.6. Предложите алгоритм, который умножает два комплексных числа

$$(a + ib)(c + id) = r + is.$$

Входом алгоритма являются числа a, b, c, d , а результатом — числа r, s . Для вычисления r, s , если мы будем следовать алгебраическому выражению, полученному после раскрытия скобок, требуется четыре умножения, однако можно предложить способ, который позволяет выполнить только три. Если Вы придумали, как это сделать, то Вы уже на том пути, который привел А.А. Кацаубу и В. Штассена к алгоритмам, названным их именами.

3.7. Алгоритм сортировки слиянием может быть разработан декомпозицией исходного массива на шаге разделения задачи не на два, а на три подмассива. Как Вы думаете, приведет ли такой подход к созданию более быстрого алгоритма? Если Вы обоснуете ответ экспериментально, то получите хорошие навыки разработки и исследования рекурсивных алгоритмов.

3.8. Нарисуйте дерево рекурсии, порождаемое алгоритмом вычисления редакционного расстояния, основанным на функциональном уравнении Беллмана, при вычислении значения $D(4, 4)$. Определите общее число вершин этого дерева. Попробуйте обобщить полученный результат и установить функциональную зависимость числа вершин дерева рекурсии $R(n, m)$ от длин обрабатываемых строк — n и m . Каков Ваш прогноз времени выполнения этого алгоритма при вычислении $D(20, 25)$?

3.9. Предложите простой итерационный алгоритм для вычисления значения редакционного расстояния, основанный на восходящем расчете, т. е. на последовательном вычислении значений $D(n, m)$ от меньших значений аргумента к большим. Такой табличный способ достаточно часто применяется для реализации рекуррентных соотношений, полученных методом динамического программирования. Каковы вычислительные затраты такого табличного алгоритма? Какой из двух алгоритмов вычисления редакционного расстояния Вы будете использовать на практике?

Список литературы к главе 3

- 3.1. Гудман С., Хидетиэми С. Введение в разработку и анализ алгоритмов. — М.: Мир, 1981. — 368 с.
- 3.2. Колмогоров А.Н., Драгалин А.Г. Математическая логика. — М.: Едиторал УРСС, 2005. — 240 с.
- 3.3. Баррон Д. Рекурсивные методы в программировании. — М.: Мир, 1974. — 79 с.
- 3.4. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ, 2-е изд.: Пер. с англ. — М.: Изд. дом «Вильямс», 2005. — 1296 с.
- 3.5. Хаггарти Р. Дискретная математика для программистов. — М.: Техносфера, 2005. — 400 с.
- 3.6. Беллман Р., Дрейфус Р. Прикладные задачи динамического программирования: Пер. с англ. — М.: Наука, 1965, — 457 с.
- 3.7. Левенштейн В.И. Двоичные коды с исправлением выпадений, вставок и замещений символов // Доклады АН СССР. 1965. Т. 163. С. 707-710.
- 3.8. Гасфилд Д. Строки, деревья и последовательности в алгоритмах: Информатика и вычислительная биология / Пер с англ. — СПб.: Невский диалект; БХВ-Петербург, 2003. — 654 с.

Глава 4

ЭЛЕМЕНТЫ ТЕОРИИ РЕСУРСНОЙ ЭФФЕКТИВНОСТИ ВЫЧИСЛИТЕЛЬНЫХ АЛГОРИТМОВ

Введение. В предыдущих главах, рассматривая различные рекурсивные алгоритмы, мы уже оперировали такими понятиями, как вычислительные затраты, сложность алгоритма и т. д., не давая их строгих определений. Цель настоящей главы — не только формализовать понятия и ввести терминологию, которая используется в области анализа и исследования эффективности вычислительных алгоритмов, но и описать некоторые классификации алгоритмов, которые мы будем впоследствии использовать. Мы приводим также описание методик оценки и выбора рациональных алгоритмов, которые могут быть использованы для обоснования решений при разработке алгоритмического обеспечения программных средств и систем, и, в частности, для обоснования выбора рекурсивных алгоритмов. Эта глава содержит краткое изложение теории ресурсной эффективности вычислительных алгоритмов, которая является областью научных исследований авторов этой книги.

В целом теория ресурсной эффективности вычислительных алгоритмов имеет своей целью создание научно-методической базы для решения вопросов сравнительного анализа и рационального выбора вычислительных алгоритмов в реальном диапазоне длин входов. Основная задача теории — это повышение ресурсной эффективности алгоритмического обеспечения программных средств и систем за счет разработки методов оценки и выбора рациональных алгоритмов решения вычислительных задач в заданных условиях применения. Для достижения этой цели и решения поставленных задач аппарат анализа ресурсной эффективности должен содержать ряд компонентов, теоретическое обоснование которых и составляет основы теории ресурсной эффективности вычислительных алгоритмов:

- систему обозначений, ориентированную на решение задач оценки и анализа ресурсной эффективности алгоритмов;
- способ оценки ресурсной эффективности алгоритмов на основе их ресурсных функций, учитывающих как ресурсные требования алгоритма, так и различные особенности области применения разрабатываемого программного продукта;
- методы получения ресурсных функций — функции объема памяти и функции трудоемкости для алгоритмов различных классов,

включая трудоемкости для среднего и худшего случаев как для процедурной, так и для рекурсивной реализации алгоритмов;

— методику анализа чувствительности алгоритмов к вариациям входных данных, позволяющую оценить устойчивость функции трудоемкости для среднего случая, а, следовательно, и временных оценок алгоритма для различных вариантов исходных данных при фиксированной длине входа;

— метод сравнительного анализа ресурсных функций алгоритмов с целью выбора рационального диапазона размерности множества входных данных или рационального алгоритма при известном диапазоне;

— метод выявления областей эквивалентной ресурсной эффективности алгоритмов, позволяющий указать предпочтения для выбора алгоритма в зависимости от диапазона размерности множества входных данных.

Изложению данного материала и посвящена настоящая глава.

§ 1. Терминология и обозначения в теории ресурсной эффективности вычислительных алгоритмов

Наиболее употребительными характеристиками ресурсной эффективности алгоритмов являются оценки временной и емкостной сложности, отражающие требуемые ресурсы процессора и оперативной памяти и/или внешней памяти. Терминологию в области анализа алгоритмов в настоящее время можно считать устоявшейся [4.1, 4.2], однако собственная система обозначений развита слабо и представлена, в основном, обозначениями асимптотического роста функций. Таким образом, возникает задача уточнения терминологии и введения системы соответствующих обозначений, ориентированных на анализ ресурсной эффективности вычислительных алгоритмов. Предлагаемые ниже определения и обозначения были введены одним из авторов в [4.3], этот материал также кратко изложен в [4.4].

Временная сложность и функция трудоемкости. Будем рассматривать в дальнейшем, придерживаясь терминологии, введенной Э. Постом, применимые к общей проблеме (задаче), правильные и финитные алгоритмы [4.5]. В качестве модели вычислений будем рассматривать абстрактную машину, включающую процессор, адресную оперативную память и набор элементарных операций, соотнесенных с процедурным языком программирования высокого уровня — модели вычислений такого типа носят название «машины с произвольным доступом к памяти» (RAM) [4.6].

Классический анализ вычислительных алгоритмов в рамках данной модели связан, прежде всего, с анализом их временной сложности. Его результатом является асимптотическая оценка количества задаваемых алгоритмом операций как функции длины входа, которая коррелирована с асимптотической оценкой времени выполнения программной реа-

лизации алгоритма. Однако асимптотические оценки указывают не более чем *порядок роста* функции, и результаты сравнения алгоритмов по этим оценкам будут справедливы только при очень больших длинах входов. Для сравнения алгоритмов в диапазоне реальных длин входов, определяемых областью применения программной системы, необходимо знание о точном количестве операций, задаваемых алгоритмом, т. е. о его функции трудоемкости.

Определение 4.1. Трудоёмкость алгоритма.

Пусть D_A есть множество допустимых конкретных проблем задачи, решаемой алгоритмом A , а его элемент $D \in D_A$ — конкретная проблема (вход алгоритма A) размерности n . Множество D есть конечное упорядоченное множество из n элементов d_i , представляющих собой слова фиксированной длины в алфавите $\{0, 1\}$:

$$D = \{d_i, i = \overline{1, n}\}, \quad |D| = n.$$

Под трудоёмкостью алгоритма A на входе D , будем понимать количество элементарных операций в принятой модели вычислений, задаваемых алгоритмом на этом входе. В дальнейшем будем обозначать через $f_A(D)$ функцию трудоёмкости алгоритма A для входа D .

Заметим, что значением функции трудоемкости для любого допустимого входа D является целое положительное число (в силу предположения о том, что алгоритм A является финитным 1-процессом по Посту).

При более детальном анализе ряда алгоритмов оказывается, что не всегда трудоемкость алгоритма на одном входе D длины n , где $n = |D|$, совпадает с его трудоемкостью на другом входе такой же длины. Рассмотрим допустимые входы алгоритма длины n — в общем случае существует подмножество (для большинства алгоритмов собственное) множества D_A , включающее все входы, имеющие размерность n , — обозначим его через D_n :

$$D_n = \{D \mid |D| = n\}.$$

Поскольку элементы d_i представляют собой слова фиксированной длины в алфавите $\{0, 1\}$, множество D_n является конечным — обозначим его мощность через M_{D_n} , т. е. $M_{D_n} = |D_n|$. Тогда алгоритм A , получая различные входы D из множества D_n , будет, возможно, задавать в каком-то случае наибольшее, а в каком-то случае наименьшее количество операций. Исключение составляют алгоритмы, для которых трудоемкость определяется только длиной входа. В связи с этим введем следующие обозначения, отметив, что соответствующая терминология является устоявшейся в области анализа алгоритмов.

Обозначим *худший случай* трудоемкости на всех входах фиксированной длины через $f_A^{\wedge}(n)$. Под худшим случаем мы понимаем наибольшее количество операций, задаваемых алгоритмом A на всех

входах размерности n :

$$\overline{f_A}(n) = \sum_{D \in D_n} p(D) f_A(D),$$

где $p(D)$ есть частотная встречаемость входа D для анализируемой области применения алгоритма. В случае, если все входы $D \in D_n$ равновероятны, то

$$\overline{f_A}(n) = \frac{1}{M_{D_n}} \sum_{D \in D_n} f_A(D).$$

Заметим, что функция $\bar{f}_A(n)$ есть вещественнозначная функция целочисленного аргумента. Если трудоемкость алгоритма зависит только от длины входа, то мы будем использовать обозначение $f_A(n)$.

На основе функции трудоемкости в худшем случае можно уточнить понятие сложности алгоритма. Заметим, что иногда сложность рассматривается и для среднего случая, но с соответствующей оговоркой.

Определение 4.2. Временная сложность алгоритма (сложность алгоритма).

Временная сложность алгоритма есть асимптотическая оценка в классах функций, определяемых обозначениями O или Θ , функции трудоемкости алгоритма для худшего случая — $f_A^{\wedge}(n) = O(g(n))$, или $f_A^{\wedge}(n) = \Theta(g(n))$, где $g(n)$ — функция, задающая класс O или Θ для $f_A^{\wedge}(n)$.

Заметим, что используемый для оценки функции $f_A^\wedge(n)$ асимптотический класс O (O большое), включает в себя средний и лучший случаи ($\overline{f_A}(n)$ и $f_A^V(n)$), т. к. запись $O(g(n))$ обозначает класс функций, имеющих скорость роста не более чем функция $g(n)$ с точностью до положительной константы [4.2], а из введенных обозначений следует, что: $f_A^V(n) \leq \overline{f_A}(n) \leq f_A^\wedge(n)$.

Емкостная сложность и функция объема памяти. Состояние памяти модели вычислений (для реального компьютера это будет оперативная память) определяется значениями, записанными в ячейках этой памяти. Тогда процессор модели вычислений, выполняя операции, заданные алгоритмом, переводит исходное состояние памяти модели

вычислений (исходные данные задачи — вход алгоритма) в конечное состояние — найденное алгоритмом решение задачи в терминах слов принятого алфавита. В ходе решения задачи может быть задействовано некоторое дополнительное количество ячеек памяти. Рассуждая по аналогии с временной сложностью и трудоемкостью, определим функцию объема памяти.

Определение 4.3. Функция объема памяти.

Под объемом памяти, требуемым алгоритмом A для входа, заданного множеством D , будем понимать максимальное количество ячеек памяти модели вычислений, задействованных в ходе выполнения алгоритма. Функцию объема памяти алгоритма для входа D будем обозначать через $V_A(D)$. Значение функции $V_A(D)$ есть целое положительное число.

Введем для функции объема памяти, по аналогии с трудоемкостью, обозначения для лучшего, худшего и среднего случая на различных входах размерности n : $V_A^\wedge(n)$, $V_A^\vee(n)$, $\bar{V}_A(n)$ соответственно. На этой основе мы можем определить емкостную сложность алгоритма.

Определение 4.4. Емкостная сложность алгоритма.

Емкостная сложность алгоритма есть асимптотическая оценка в классах функций, определяемых обозначениями O или Θ , функции объема памяти алгоритма для худшего случая — $V_A^\wedge(n) = O(h(n))$, или $V_A^\wedge(n) = \Theta(h(n))$, где $h(n)$ — функция, задающая класс O или Θ для $V_A^\wedge(n)$.

Ресурсная характеристика и ресурсная сложность алгоритма. Содержательно термин «ресурсная эффективность алгоритма» включает в себя как требуемый алгоритмом ресурс процессора, так и ресурс памяти, которые будут задействованы алгоритмом при решении задач размерности n . Поскольку в результате анализа алгоритма по этим ресурсам могут быть получены или функциональные зависимости от размерности, или их асимптотические оценки, то целесообразно ввести следующие определения и обозначения, имея в виду, что могут рассматриваться лучший, худший, или средний случай требуемых ресурсов при фиксированной размерности.

Определение 4.5. Ресурсная характеристика алгоритма.

Под ресурсной характеристикой алгоритма в худшем, среднем или лучшем случае будем понимать упорядоченную пару функций — соответствующих рассматриваемому случаю функции трудоемкости и функции объема памяти. Ресурсную характеристику алгоритма (**Resource characteristic**) будем обозначать через $\mathfrak{R}_h^*(A)$:

$$\mathfrak{R}_h^*(A) = \langle f_A^*(n), V_A^*(n) \rangle, \quad \text{где } * \in \{\wedge, \vee, \bar{}\}.$$

Определение 4.6. Ресурсная сложность алгоритма.

Под ресурсной сложностью алгоритма (**Resource complexity**) в худшем, среднем или лучшем случае будем понимать упорядоченную пару классов функций, заданных асимптотическими обозначениями O или

Θ — соответствующие рассматриваемому случаю временная сложность и емкостная сложность алгоритма, и обозначать ее через $\mathfrak{R}_c^*(A)$:

$$\mathfrak{R}_c^*(A) = \langle O(g(n)), O(h(n)) \rangle,$$

где

$$g(n): f_A^*(n) = O(g(n)), \quad h(n): V_A^*(n) = O(h(n)),$$

или

$$\mathfrak{R}_c^*(A) = \langle \Theta(g_1(n)), \Theta(h_1(n)) \rangle,$$

где:

$$\begin{aligned} g_1(n): f_A^*(n) &= \Theta(g_1(n)), \\ h_1(n): V_A^*(n) &= \Theta(h_1(n)), \end{aligned} \quad * \in \{\wedge, \vee, -\}.$$

Заметим, что в ряде случаев, особенно при сравнительном анализе ресурсной эффективности алгоритмов, более наглядным является переход в оценке емкостной сложности от общего объема памяти модели вычислений к объему дополнительной памяти, требуемой алгоритмом, т. к. объемы памяти для входа и результата одинаковы для всех сравниваемых алгоритмов. В этом случае можно сохранить введенные обозначения, но использовать их с соответствующей оговоркой. В качестве примера приведем обозначение ресурсной сложности для алгоритма сортировки вставками, для которого $f_A^\wedge(n) = \Theta(n^2)$ [4.2], а требуемая дополнительная память фиксирована и не зависит от длины сортируемого массива. Для этого алгоритма

$$\mathfrak{R}_c^\wedge(A) = \langle \Theta(n^2), \Theta(1) \rangle.$$

§ 2. Функции ресурсной эффективности алгоритмов и их программных реализаций

Особенности оценки и требования к алгоритмам в различных проблемных областях. При разработке алгоритмического обеспечения программных средств и систем, для решения задачи выбора алгоритмов, рациональных в данных условиях применения, возникает необходимость оценки качества алгоритма. Очевидно, эта задача является только частью одного из этапов разработки математического и программного обеспечения. Само программное обеспечение имеет широкий спектр собственных оценочных критериев, широко обсуждаемых в современной литературе. При этом сложность и разноплановый характер применения современных программных средств и систем обусловливают и комплексный подход к их оценке, учитывающий различные, порой противоречивые, критерии и требования. Это важная практическая и научная задача, которой посвящено много современных публикаций — укажем, например, на достаточно полное исследование вопросов качества программного обеспечения, содержащееся в монографии

графиях В.В. Липаева [4.7, 4.8]. Применение такого комплексного подхода целесообразно также и для оценки ресурсной эффективности алгоритмов и их программных реализаций. Это приводит к необходимости учета не только временной эффективности алгоритма, определяемой его функцией трудоемкости и средой реализации, но и ряда дополнительных оценок, отражающих требования алгоритма и реализующей его программы к другим ресурсам компьютера. Таким образом, мы вводим функцию ресурсной эффективности вычислительного алгоритма, учитывающую требования алгоритма к ресурсам компьютера. К таким ресурсам можно отнести ресурсы оперативной памяти, необходимые для хранения исходных данных, промежуточных и окончательных результатов, машинного кода программы, и ресурс стека. Ресурс процессора, требуемый алгоритмом, отражается непосредственно временной оценкой или, опосредованно, функцией трудоемкости. Отметим, что в терминах этих же оценок могут быть сформулированы и требования, предъявляемые к алгоритму со стороны разработчиков программной системы, учитывающие специфику его применения.

Очевидно, что в зависимости от области применения важность каждого ресурса будет изменяться, что приводит к необходимости введения весовых коэффициентов для компонентов функции ресурсной эффективности алгоритма. Такие весовые коэффициенты могут быть интерпретированы как удельные стоимости ресурсов. Рассмотрим наиболее характерные особенности оценки качества программного и, в основном, алгоритмического обеспечения для некоторых проблемных областей:

— научно-технические задачи большой сложности — программные системы, ориентированные на эту проблемную область, характеризуются большим удельным весом вычислений с плавающей точкой. При этом временные затраты вычислений преобладают над операциями ввода/вывода. Большие объемы исходных данных и промежуточных результатов приводят к необходимости компромисса между временной эффективностью, точностью результатов и объемами требуемой оперативной памяти;

— сетевое программное обеспечение и распределенные системы — одно из основных требований к таким программным системам — обеспечение минимально возможного времени передачи пакетов данных и их диспетчеризация в динамически изменяющихся условиях. Реализация этих требований приводит к необходимости быстрого и эффективного решения задачи адаптивной маршрутизации. Для этой сложной задачи продолжается поиск алгоритмов, эффективных как по времененным затратам на расчеты маршрутов, так и по полученным результатам (времени передачи пакетов данных);

— системы управления базами данных и знаний — эта область применения программного обеспечения характеризуется, прежде всего, значительными объемами обрабатываемой информации и созданием специальных структур данных, поддерживающих алгоритмы быстрого

поиска. Отметим, что на поиск информации в этих системах накладываются достаточно жесткие временные ограничения. В этой связи укажем на актуальную задачу, решение которой связано с эффективными алгоритмами многоключевого поиска — задачу разработки «поисковых машин» в современных Интернет-технологиях;

— диалоговые и мультимедийные системы — отличительной особенностью в оценке качества таких систем является время отклика на запрос, тем самым временные требования заставляют разработчиков уделять серьезное внимание эффективности как алгоритмов решения проблемных задач, так и алгоритмов мультимедийного общения;

— программное обеспечение бортовых компьютеров — это группа программных продуктов с наиболее жесткими требованиями и по времени выполнения, и по занимаемой оперативной памяти. Ряд дополнительных требований, связанных с точностью расчетов, надежностью программного обеспечения, только усиливает необходимость тщательного подхода к разработке алгоритмического обеспечения таких программных продуктов;

— программное обеспечение встраиваемых микропроцессорных систем, где, помимо временных требований, существенную роль при выборе алгоритмического обеспечения играют ресурсные ограничения по памяти. Дополнительно предъявляются также требования по надежности и устойчивости программного обеспечения. В рамках анализа алгоритмов в этой проблемной области необходимо также учитывать специфику машинных команд, особенно для микропроцессоров с *RISC* архитектурой.

Отметим, что требуемые ресурсы определяются как собственно самим алгоритмом, так и характеристиками исходных данных. Ресурс памяти может также зависеть и от особенностей поддерживаемых выбранным языком программирования типов и структур данных и тщательности программной реализации.

Компоненты функции ресурсной эффективности программной реализации алгоритма. Пусть D_A — конкретное множество исходных данных алгоритма (см. 4.1), тогда определим следующие компоненты в терминах требуемых алгоритмом ресурсов компьютера, возможно, зависящие от характеристик множества D_A или некоторых его элементов [4.3]:

— $V_{exe}(D_A)$ — ресурс оперативной памяти в области кода, требуемый под размещение машинного кода, реализующего данный алгоритм. Этот ресурс соотносим с объемом *EXE*-файла с учетом оверлейных структур и принципа организации управления программной системой. Заметим, что, как правило, короткие по объему реализующего программного кода алгоритмы имеют худшие временные оценки, чем более длинные, для одной и той же задачи («быстрые алгоритмы являются в большинстве случаев достаточно сложными» [4.1]). В основном для небольших программ объем области кода не зависит от D_A . В случае больших программных систем или комплексов модули управления

вычислительным процессом при определенных исходных данных могут подгружать в оперативную память дополнительные фрагменты кода. Такой подход характерен для оверлейных структур или программных систем со структурой, адаптивной к входным данным. Аналогичная ситуация может быть следствием выбора различных алгоритмов, рациональных в зависимости от характеристик входа. К получению такой адаптивной структуры программного обеспечения могут приводить результаты описываемого сравнительного анализа ресурсной эффективности алгоритмов;

— $V_{ram}(D_A)$ — ресурс дополнительной оперативной памяти в области данных, требуемый алгоритмом под временные ячейки, массивы и структуры. Ресурс памяти для входа и выхода алгоритма не учитывается в этой оценке, т. к. является неизменным для всех алгоритмов решения данной задачи. Обычно более быстрые алгоритмы решения некоторой задачи требуют большего объема дополнительной памяти [4.1, 4.9]. В качестве примера можно привести быстрый алгоритм сортировки методом индексов [4.2], требующий дополнительной памяти в объеме, равном значению максимального элемента исходного массива (алгоритм допускает только целочисленные входы). Такой дополнительный массив есть «плата» за быстродействие — при определенных условиях мы можем получить отсортированный массив за $\Theta(n)$ операций, где n — размерность входного массива;

— $V_{st}(D_A)$ — ресурс оперативной памяти в области стека, требуемый алгоритмом для организации вызова внутренних процедур и функций. Объем данного ресурса существенно зависит от того, в какой методике — итерационной или рекурсивной — реализован данный алгоритм. Требуемый объем памяти в области стека может быть критичен при рекурсивной реализации по отношению к размерности решаемой задачи, если дерево рекурсии достаточно «глубоко». Если алгоритм реализуется в объектно-ориентированной среде программирования, то требования к ресурсу стека могут быть значительны за счет длинных цепочек вызовов методов, связанных с наследованием в объектах;

— $T(D_A)$ — требуемый алгоритмом ресурс процессора — оценка времени выполнения данного алгоритма на данном компьютере. Эта оценка определяется функцией трудоемкости алгоритма в зависимости от характеристических особенностей множества исходных данных. Переход от функции трудоемкости к временной оценке связан с определением средневзвешенного времени $\bar{t}_{\text{оп}}$ выполнения обобщенной базовой операции в языке реализации алгоритма на данном процессоре и компьютере. Получение точной функции времени выполнения, учитывающей все особенности архитектуры компьютера, представляет собой достаточно сложную задачу; для оценки сверху можно воспользоваться функцией трудоемкости для худшего случая при данной размерности — $f_A^h(n)$. В большинстве случаев может быть использована функция трудоемкости в среднем — $\bar{f}_A(n)$. Средневзвешенное время $\bar{t}_{\text{оп}}$ может

быть получено экспериментальным путем в среде реализации алгоритма.

Функции ресурсной эффективности алгоритма и его программной реализации. Вводя в соответствии с принятым стоимостным подходом весовые коэффициенты, мы получаем функцию ресурсной эффективности алгоритма для входа D в виде

$$\Psi_A(D) = C_V \cdot V_A(D) + C_f \cdot f_A(D) \quad (4.2.1)$$

и функцию ресурсной эффективности программной реализации

$$\Psi_{AR}(D) = C_{exe} \cdot V_{exe}(D) + C_{ram} \cdot V_{ram}(D) + C_{st} \cdot V_{st}(D) + C_t \cdot T_A(D), \quad (4.2.2)$$

где конкретные значения коэффициентов $C_i, i \in \{exe, ram, st, t\}$ задают удельные стоимости ресурсов, определяемые условиями применения алгоритма и спецификой программной системы.

Выбор рационального алгоритма A_r может быть осуществлен при заданных значениях коэффициентов C_i по критерию минимума функции $\Psi_{AR}(D)$, рассчитанной для всех претендующих алгоритмов из множества A . Пусть множество претендующих алгоритмов A состоит из m элементов $A = \{A_i | i = 1, m\}$, тогда

$$A_r(D) : \quad \Psi_{AR_r}(D) = \min_{A_i} \{\Psi_{AR_i}(D)\}, \quad (4.2.3)$$

где минимум берется по всем претендующим алгоритмам из множества A , а запись $A_r(D)$ обозначает рациональный алгоритм для данного входа. Формально выполнив вычисления по формуле (4.2.3) для всех входов D из множества D_A , ограниченного особенностями применения данной задачи в разрабатываемой программной системе, мы можем определить те множества входов, при которых один из претендующих алгоритмов будет наиболее рациональным. Однако такой подход нецелесообразен из-за слишком большого количества элементов в D .

Более реалистический и практически приемлемый подход связан с использованием оценки в среднем или худшем случае для компонентов функции ресурсной эффективности, в зависимости от специфики требований. Обозначая соответствующие компоненты ресурсной функции как функции размерности с учетом обозначений из 4.1, получаем

$$\Psi_A^*(n) = C_V \cdot V_A^*(n) + C_f \cdot f_A^*(n), \quad (4.2.4)$$

где $* \in \{\wedge, \vee, -\}$ — обозначение худшего, лучшего и среднего случая ресурсной функции, и функцию ресурсной эффективности программной реализации в виде

$$\Psi_{AR}(n) = C_{exe} \cdot V_{exe}(n) + C_{ram} \cdot V_{ram}(n) + C_{st} \cdot V_{st}(n) + C_t \cdot T_A(n), \quad (4.2.5)$$

Для выбора рационального алгоритма определим функцию $R(n)$, значением которой является номер алгоритма r , для которого

$$R(n) = r, \quad r \in \{1, m\}; \quad r = \arg \min_{i=1,m} \{\Psi_{AR_i}(n)\}. \quad (4.2.6)$$

Тогда на исследуемом сегменте размерностей входа $[n_1, n_2]$ возможны следующие случаи:

— функция $R(n)$ принимает одинаковые значения на $[n_1, n_2]$,

$$R(n_i) = R(n_j) = k, \quad \forall n_i, n_j \in [n_1, n_2],$$

тем самым алгоритм с номером k является рациональным по ресурсной функции (4.2.5) на всем сегменте размерностей входа $[n_1, n_2]$;

— функция $R(n)$ принимает различные значения на $[n_1, n_2]$:

$$\exists n_i, n_j \in [n_1, n_2]: \quad R(n_i) \neq R(n_j),$$

тем самым несколько алгоритмов являются рациональными по ресурсной функции (4.2.5) для различных размерностей входа на сегменте $[n_1, n_2]$. В этом случае на сегменте $[n_1, n_2]$ можно выделить подсегменты, в которых значение $R(n_i) = \text{const}$, и реализовать адаптивный по размеру входа выбор рационального алгоритма.

Таким образом, на основе функции ресурсной эффективности можно выбрать алгоритм, имеющий минимальную ресурсную стоимость при данных весах на некотором подсегменте исследуемого сегмента размерностей. На основе функций ресурсной эффективности возможно более детальное исследование задачи выбора рационального алгоритма с точки зрения характеристик конкретного компьютера и внешних требований к алгоритму. Такой подход приводит к построению четырехмерного пространства аргументов, координатами которого являются значения ресурсных функций (объем памяти кода, дополнительных данных, стека и временная оценка). В таком пространстве внешние требования к алгоритму (более точно — к его программной реализации), заданные в виде ограничивающих сегментов, задают четырехмерную область W допустимых значений. Исследуемой программной реализации алгоритма на данном компьютере для каждого значения размерности задачи n в этом пространстве соответствует точка $X(n)$ с координатами

$$X(n) = (V_{exe}(n), V_{ram}(n), V_{st}(n), T(n)),$$

где координаты рассчитываются по среднему или худшему случаю для данной размерности. Очевидно, что от характеристик компьютера зависит только «координата» $T(n)$. Различным значениям n в этом пространстве соответствует «точечный график» — набор точек, каждая из которых связана со своим значением размерности. Интерес, очевидно, представляют такие значения размерности n , которым соответствуют точки $X(n)$, находящиеся на границе области W . При таком подходе появляется возможность исследования алгоритмов на «устойчивость»

по отношению к характеристикам множества исходных данных (в частности — размерности задачи) в заданной области требований. Например, возможно определение максимальной размерности решаемой задачи с учетом всех требуемых ресурсов. Другая возможность состоит в исследовании области W и характеристик компьютеров (опосредованно через $T(n)$) с целью выбора рационального компьютера для данного алгоритма решения задачи с учетом внешних требований.

Приведем графическую иллюстрацию указанного подхода, связанную с определением максимальной размерности задачи. Для перевода области W в двухмерное пространство введем обобщенный показатель ресурса памяти в среднем, как функцию размерности входа, в виде

$$V(n) = V_{exe}(n) + V_{ram}(n) + V_{st}(n).$$

Пусть V^* и T^* — ограничения со стороны разрабатываемой программной системы на максимально допустимый объем оперативной памяти и время выполнения. При условии, что минимальные значения равны нулю, прямоугольник

$$W = \{(0, 0), (T^*, V^*)\}$$

задает область W . Пусть также значения n_1, n_2, n_3, n_4 принадлежат области размерности задачи, причем $n_1 < n_2 < n_3 < n_4$. В указанных обозначениях на рис. 4.1 показано возможное определение верхней гра-

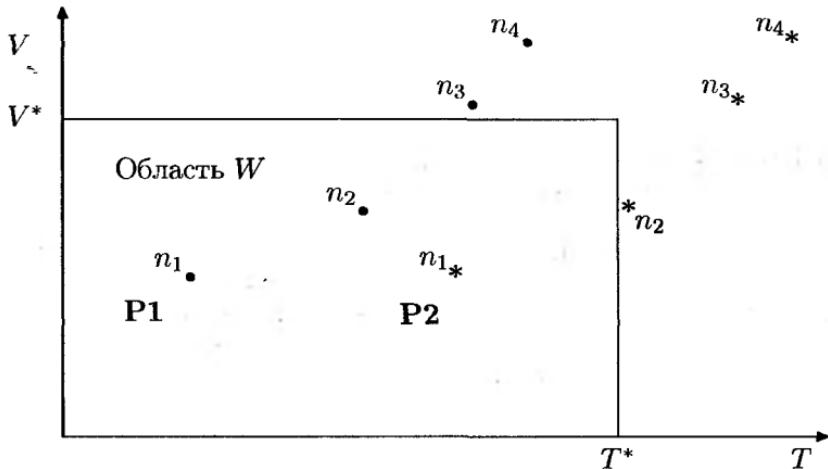


Рис. 4.1. Определение верхней границы размерности задачи с учетом ресурсных ограничений для различных компьютеров

ницы размерности задачи в области W . Точки и звездочки на рис. 4.1 соответствуют различным компьютерам — $P1$ и $P2$ соответственно, причем $P1$ является более производительным, чем $P2$.

С возрастанием размерности задачи, по крайней мере, не уменьшаются требуемая оперативная память и время выполнения программной

реализации алгоритма. На рис. 4.1 показано, что для размерности n_3 и реализации на компьютере $P1$ будет превышено пороговое значение (граница области W) по объему оперативной памяти. Для размерности n_2 и реализации на компьютере $P2$ будет превышена граница области W по времени выполнения.

Дополнительные компоненты функции ресурсной эффективности. Предложенная функция ресурсной эффективности вычислительных алгоритмов как функция размерности входа учитывает ресурсы компьютера, требуемые данным алгоритмом. Можно говорить о том, что это базовая ресурсная функция, компоненты которой могут быть модифицированы с учетом дополнительных требований к характеристикам алгоритма со стороны программной системы. Такая модификация может быть осуществлена введением дополнительных компонентов, учитывающих специальные требования к алгоритмическому обеспечению, или введением функциональных зависимостей для стоимостных коэффициентов с аргументом размерности входа в компонентах функции ресурсной эффективности. Дополнительные компоненты, вводимые в функцию ресурсной эффективности, могут быть связаны с необходимостью как отражения специфики проблемной области и учета особенностей задачи, так и учета специальных требований к алгоритмическому обеспечению, обусловленных техническим заданием на разработку данной программной системы. Укажем некоторые характерные случаи и возможные варианты построения дополнительных компонентов.

Учет требований точности. Требования обеспечения необходимой точности возникают, как правило, в задачах оптимизации, решаемых алгоритмами, использующими в своей основе аппарат численных методов. Другая ситуация связана с необходимостью получения решений для NP -полных задач. Поскольку точное решение для практически значимых размерностей не может быть пока получено за полиномиальное время [4.2], то одним из вариантов является использование специальных алгоритмов, обеспечивающих ϵ -полиномиальные приближения для NP -полных задач. Учет требуемой точности получаемого решения может быть осуществлен в рамках принятого стоимостного подхода следующими способами:

- первый способ основан на непосредственном учете точности в компонентах. Поскольку точность для оптимизационных алгоритмов (имеется в виду не оценка точности метода, лежащего в основе алгоритма, а точность, обеспечиваемая итерационной сходимостью) обеспечивается, как правило, процессом итерационной сходимости, то входное значение ϵ сильно влияет на трудоемкость. При этом функция трудоемкости, а следовательно, и время выполнения в среднем $T(n)$ зависят как от размерности, так и от точности: $T = T(n, \epsilon)$. Возможно, что и дополнительные затраты памяти также зависят от точности, если по каким-либо причинам в ходе итерационного процесса сходимости необходимо запоминать промежуточные результаты. Таким образом,

функция ресурсной эффективности программной реализации является функцией и размерности, и точности:

$$\Psi_{AR} = \Psi_{AR}(n, \varepsilon);$$

— второй способ состоит в построении базовой функции при фиксированном минимально приемлемом значении точности $\varepsilon = \varepsilon_0$. Ресурсные затраты на дальнейшее повышение точности результата могут быть вынесены в отдельный компонент функции ресурсной эффективности с собственным стоимостным весом:

$$E\left(n, \frac{\varepsilon_0}{\varepsilon}\right) = T_\varepsilon\left(n, \frac{\varepsilon_0}{\varepsilon}\right) + V_{\varepsilon, ram}\left(n, \frac{\varepsilon_0}{\varepsilon}\right),$$

где компонент T_ε учитывает в относительных единицах изменение временной эффективности при изменении точности, а компонент $V_{\varepsilon, ram}$ — изменение дополнительно требуемой памяти. Обозначая значение базовой функции при $\varepsilon = \varepsilon_0$ через $\Psi_{AR}(n, \varepsilon_0)$, получаем

$$\Psi_{AR}(n, \varepsilon) = \Psi_{AR}(n, \varepsilon_0) + E\left(n, \frac{\varepsilon_0}{\varepsilon}\right), \quad \varepsilon < \varepsilon_0.$$

Учет требований по временной устойчивости. В некоторых проблемных областях применения к программному обеспечению предъявляются специальные требования, связанные с особенностями функционирования аппаратных средств. Наиболее характерными примерами могут служить бортовые вычислительные комплексы и компьютеры, а также встраиваемые системы. Программное обеспечение таких систем должно не только удовлетворять жестким времененным ограничениям, но и обладать таким свойством, которое может быть названо временной устойчивостью по данным. Введенное понятие временной устойчивости по данным означает, что различные входы вычислительного алгоритма при фиксированной длине приводят к небольшим изменениям наблюдаемого времени выполнения. Этот показатель является важным, т. к. обеспечивает, наряду с исполнительной аппаратурой, расчетное время отклика системы на внешние воздействия. Учет требования временной устойчивости может быть выполнен на основе введения понятия информационной чувствительности алгоритма, отражающей изменение значений функции трудоемкости для различных входов фиксированной длины.

Количественная мера информационной чувствительности определяется в [4.10] на основе исследования алгоритма методами математической статистики. Обозначив, следуя [4.10], количественную меру информационной чувствительности алгоритма через $\delta_i(n)$ и связав с ней стоимостной коэффициент, получаем дополнительный компонент функции ресурсной эффективности в виде: $C_\delta \delta_i(n)$. Важность временной устойчивости и связанной с ней информационной чувствительности алгоритма может быть подчеркнута разработчиками путем выбора больших значений коэффициента C_δ .

Функциональные стоимостные коэффициенты. Ряд специальных требований к программному обеспечению может быть учтен в предлагаемой функции ресурсной эффективности не путем введения дополнительных компонентов, а путем изменения значений стоимостных коэффициентов при изменении размерности решаемой задачи. Такой путь эффективен при необходимости учета «пороговых» требований, например, по объему требуемой дополнительной памяти или времени выполнения. Поскольку функция ресурсной эффективности программной реализации алгоритма является функцией размерности входа, а пороговые требования формулируются в размерностях компонентов функции, то целесообразно рассматривать стоимостные коэффициенты как функции значений компонентов, например,

$$C_{ram} = C_{ram}(V_{ram}(n)). \quad (4.2.7)$$

Таким образом, стоимостные коэффициенты являются сложной функцией размерности задачи. Формализация пороговых требований коэффициентами функции ресурсной эффективности может быть выполнена с использованием функции Хевисайда — $H(t)$, которую будем использовать для произвольного аргумента x с обозначением $H(x)$:

$$H(x) = \int_{-\infty}^x \delta(x) dx,$$

где $\delta(x)$ — дельта функция. Рассмотрим общий случай, когда стоимостной коэффициент представляет собой ступенчатую функцию с несколькими пороговыми значениями. Пример компонента функции ресурсной эффективности, связанного с дополнительной памятью показан на рис. 4.2.

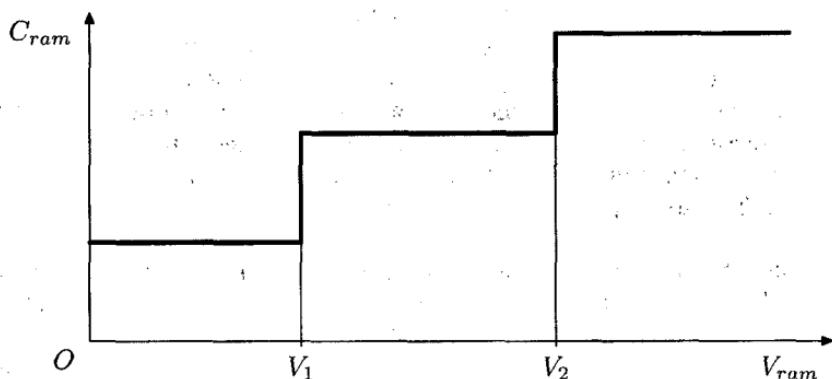


Рис. 4.2. Коэффициент C_{ram} , заданный ступенчатой функцией

Показанные на рис. 4.2 пороговые значения объема дополнительной памяти V_1 и V_2 могут соответствовать, например, значениям объемов

$L1$ и $L2$ кэш-памяти компьютера. Пусть k — количество уровней коэффициента C_{ram} , обозначим через $C_{ram,j}$ разность значений между j и $j - 1$ уровнями, полагая, что значение $C_{ram,0} = 0$, тогда коэффициенты могут быть заданы в виде

$$C_{ram}(n) = \sum_{j=1}^k C_{ram,j} H(V_{ram}(n) - V_{ram}(j-1)),$$

где $V_{ram}(j-1)$ — пороговые значения аргумента ступенчатой функции, причем $V_{ram,0} = 0$. Помимо ступенчатой функции, могут рассматриваться также варианты задания коэффициентов в виде кусочно-линейной или непрерывной функции.

§ 3. Классы открытых и закрытых задач и теоретическая нижняя граница временной сложности

В теоретическом аспекте анализа ресурсной эффективности представляет интерес классификация задач, основой которой является сравнение доказанной нижней границы временной сложности задачи и оценки наиболее эффективного из существующих алгоритмов ее решения. На основе такой классификации можно говорить о теоретической возможности улучшения временной эффективности алгоритма. В современной теории анализа алгоритмов известны доказательства нижних границ временной сложности для целого ряда задач [4.1, 4.11]. Результаты, полученные в области практической разработки и анализа алгоритмов, позволяют указать наиболее асимптотически эффективные на данный момент алгоритмы решения широкого круга задач. В связи с этим представляет интерес взаимное сравнение этих результатов и построение на этой основе классификации вычислительных задач, отражающей достичимость наиболее эффективным из известных в настоящее время алгоритмов доказанной нижней границы временной сложности задачи. Заметим, что определения классов P и EXP в теории алгоритмов [4.2, 4.12] также опираются на асимптотику временной сложности существующих алгоритмов решения задач в указанных классах.

Теоретическая нижняя граница временной сложности. Рассмотрим некоторую вычислительную задачу Z . Обозначим, используя терминологию Э. Поста [4.5], через D_Z множество конкретных допустимых проблем данной задачи Z . Введем также следующие обозначения: R_Z — множество правильных решений, Ver — верификационная функция задачи. Пусть $D \in D_Z$ — конкретная допустимая проблема данной задачи. Обозначим через A_Z полное множество всех алгоритмов (включаящее как известные, так и будущие алгоритмы), решающих задачу Z в понимании алгоритма как финитного 1-процесса по Посту, т. е.

любой алгоритм A из A_Z применим $\forall D \in D_Z$, заканчивается и дает правильный ответ

$$A_Z = \{ A | A : D \rightarrow R, D \in D_Z, R \in R_Z, Ver(D, R) = \text{True}, f_A(D) \neq \infty \quad \forall D \in D_Z \},$$

где $f_A(D)$ — значение функции трудоемкости алгоритма для входа D . В общем случае существует подмножество (для большинства задач собственное) множества D_Z , включающее все конкретные проблемы, имеющие размерность n , — обозначим его через D_{nz} :

$$D_{nz} = \{ D | D \in D_Z, |D| = n \}.$$

Для сравнения двух функций по асимптотике их роста будем использовать отношение \prec , введенное Поль-Дюбуа Раймоном [4.15]:

$$f(x) \prec g(x) \Leftrightarrow \lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = 0.$$

Введем понятие теоретически доказанной нижней границы временной сложности задачи с обозначением $f_{th\lim}(n)$ как Θ -оценки некоторой функции, аргументом которой является длина входа, такой, что теоретически невозможно предложить алгоритм, решающей данную задачу асимптотически быстрее, чем с оценкой $f_{th\lim}(n)$. Более корректно это понятие означает, что имеется строгое доказательство того, что любой алгоритм решения данной задачи имеет в худшем случае на входе длины n временную сложность не лучше, чем $f_{th\lim}(n)$, и данная функция представляет собой точную верхнюю грань множества функций, образующих асимптотическую иерархию (по отношению « \prec »), для которых такое доказательство возможно. Иначе говоря, для любой функции $g(n) \succ f_{th\lim}(n)$ указанное свойство не может быть доказано. Дадим формальное определение для $f_{th\lim}(n)$ [4.4]. Определим множество F_{\lim} , состоящее из функций f_{\lim} :

$$F_{\lim} = \{ f_{\lim} | \forall A \in A_Z, \forall n > 0, f_A^\wedge(n) = \Omega(f_{\lim}(n)) \},$$

где $f_A^\wedge(n)$ — функция трудоемкости для худшего случая из всех входов, имеющих размерность n , т. е. для любого $D \in D_{nz}$, тогда $f_{th\lim}(n) = \sup_{\prec} \{ F_{\lim} \}$.

Классическим примером задачи с теоретически доказанной нижней границей сложности является задача сортировки массива с использованием сравнений. Для этой задачи имеется строгое доказательство, основанное на рассмотрении бинарного дерева сравнений, того, что невозможно отсортировать массив, сравнивая элементы между собой, быстрее, чем за $\Theta(\log_2(n!))$ [4.13]. Применяя аппроксимацию для $n!$, имеем

$$\begin{aligned} f_{th\lim}(n) &= \Theta(\log_2(n!)) = \Theta(n \log_2(n) - n \log_2 e + 0,5 \log_2(2\pi n)) = \\ &= \Theta(n(\log_2(n) - \log_2 e) + 0,5 \log_2 n + 0,5 \log_2(2\pi)). \end{aligned}$$

Тривиальная нижняя граница сложности. Независимо от того, доказана или нет теоретическая нижняя граница временной сложности, мы можем для подавляющего большинства вычислительных задач указать нижнюю границу в виде Θ -оценки некоторой функции на основе рациональных рассуждений. Такая эвристическая оценка строится, как правило, на основе тривиальных предположений:

— оценка $\Theta(n)$, если для решения задачи необходимо как минимум обработать все исходные данные (задача поиска максимума в массиве, задача умножения матриц, задача коммивояжера);

— оценка $\Theta(1)$ для задач с необязательной обработкой всех исходных данных, например задача поиска в списке по ключу, при условии, что сам список является входом алгоритма.

Обозначим эту оценку через $f_{tr}(n)$ [4.4] и будем предполагать априорно, что оценка $f_{tr}(n)$ существует для задачи Z , тогда можно указать следующие возможные соотношения между оценками $f_{th\lim}(n)$ и $f_{tr}(n)$: оценка $f_{th\lim}(n)$ не доказана для задачи Z и для данной задачи принимается оценка $f_{tr}(n)$; оценка $f_{th\lim}(n)$ доказана, тогда либо: $f_{tr}(n) \prec f_{th\lim}(n)$, либо $f_{tr}(n) = f_{th\lim}(n)$.

Оценка сложности наилучшего известного алгоритма. Рассмотрим множество всех известных алгоритмов, решающих задачу Z , — множество A_R ; очевидно, что $A_R \subset A_Z$, отметим, что множество A_R конечно. Определим асимптотически лучший по сложности алгоритм, обозначив его через A_{min} [4.4]. Отметим, что, как это чрезвычайно часто бывает на практике, алгоритмы, имеющие лучшие асимптотические оценки, дают плохие по трудоемкости результаты на малых размерностях из-за значительных коэффициентов при главном порядке функции трудоемкости. В качестве примера можно привести «быстрые» алгоритмы умножения длинных двоичных целых чисел, предложенные Карацубой, Штрассеном и Шенхаге [4.2], реально эффективные начная с чисел, имеющих двоичное представление в несколько сотен и тысяч битов соответственно. Заметим также, что может существовать либо один, либо несколько алгоритмов с минимальной по всему множеству A_R асимптотической оценкой — в таком случае мы выбираем один из них в качестве алгоритма-представителя. Формально определим множество алгоритмов $A_M = \{A_m\}$ как подмножество известных алгоритмов A_R , обладающих минимальной асимптотической оценкой трудоемкости, и выделим во множестве A_M любой алгоритм:

$$A_M = \{A_m \mid \exists A \in A_R : f_A(n) \prec f_{A_m}(n)\}, \quad |A_M| \geq 1, \quad A_{min} \in A_M.$$

(*) обозначим трудоемкость этого алгоритма через $f_{A_{min}}(n)$.

Классификация задач по соотношению теоретической нижней границы временной сложности и сложности наиболее эффективного из существующих алгоритмов. Сравнивая между собой полученные оценки для некоторой задачи — $f_{th\lim}(n)$, если она существует, $f_{tr}(n)$ и $f_{A_{min}}(n)$, можно предложить следующую классификацию

задач, отражающую соотношение оценки задачи и наиболее эффективного из известных алгоритмов ее решения [4.4].

Класс задач THCL (Theoretical close). Это задачи, для которых $f_{th\lim}(n)$ существует и $f_{Amin}(n) = \Theta(f_{th\lim}(n))$. Можно говорить, что это класс теоретически (по временной сложности) закрытых задач, т. к. существует алгоритм, решающий данную задачу с асимптотической временной сложностью, равной теоретически доказанной нижней границе, что и отражено в названии этого класса. Разработка новых или модификация известных алгоритмов из этого класса может преследовать лишь цель улучшения коэффициента при главном порядке в функции трудоемкости, при этом

$$f_A(n) = \Omega(f_{th\lim}(n)) \quad \forall A \in A_Z \quad \text{и} \quad f_{Amin}(n) = \Theta(f_{th\lim}(n)).$$

Приведем несколько примеров задач этого класса: задача поиска максимума в массиве, где $f_{th\lim}(n) = \Theta(n)$, $f_{Amin}(n) = \Theta(n)$; задача сортировки массива с использованием сравнений, где $f_{th\lim}(n) = \Theta(n \log_2(n))$, $f_{Amin}(n) = \Theta(n \cdot \log_2(n))$ — алгоритм сортировки пирамидой [4.13]; задача умножения длинных двоичных целых чисел: $f_{th\lim}(n) = \Theta(n \ln(n) \cdot \ln \ln(n))$, $f_{Amin}(n) = \Theta(n \ln(n) \cdot \ln \ln(n))$ — алгоритм Штрассена – Шенхаге [4.2].

Класс задач PROP (Practical open). Это задачи, для которых $f_{th\lim}(n)$ существует и $f_{Amin}(n) > f_{th\lim}(n)$. Таким образом, для задач этого класса существует зазор между теоретической нижней границей сложности и оценкой наиболее эффективного из существующих сегодня алгоритмов ее решения. Предлагаемая аббревиатура *PROP* — «практически открытые» задачи — отражает тот факт, что для задач этого класса имеет место практическая проблема разработки алгоритма, обладающего доказанной теоретической нижней границей временной сложности. В случае разработки такого алгоритма данная задача будет переведена в класс *THCL*. Отметим, что достаточно часто на основе самого доказательства теоретической нижней границы временной сложности может быть построен алгоритм решения данной задачи, что объясняет небольшое количество задач в данном классе. Такие доказательства могут опираться на ресурсные требования, в частности, по объему дополнительной памяти, как например, при рассмотрении задачи информационного поиска с использованием хеш-адресации [4.1, 4.11], таким образом, связь временной сложности и доступной дополнительной памяти может быть предметом специального рассмотрения в аспекте расширения предлагаемой классификации.

Класс задач THOP (Theoretical open). Это задачи, для которых оценка $f_{th\lim}(n)$ не доказана, а $f_{Amin}(n) > f_{tr}(n)$. Для задач этого класса существует зазор между тривиальной нижней границей временной сложности и оценкой наиболее эффективного из существующих сегодня алгоритмов ее решения. Предлагаемая аббревиатура *THOP* — «теоретически открытые» задачи [4.4] — отражает тот факт, что для задач этого класса имеет место теоретическая проблема либо доказатель-

ства оценки $f_{lbm}(n)$, может быть даже равной $f_{tr}(n)$, переводящего задачу в класс *PROP*, либо доказательства глобальной оптимальности наиболее эффективного из существующих алгоритмов, переводящего задачу в класс *THCL*.

Отметим, что в теоретическом плане класс *THOP* достаточно широк. В этом классе находятся все задачи из класса *NPC* (*NP*-полные задачи), рассматриваемого в теории алгоритмов, для которых не известно алгоритмов, имеющих полиномиальную оценку, но на сегодня не доказано, что они не могут быть решены за полиномиальное время [4.2]. Под полиномиальным временем понимается зависимость времени решения задачи размерности n вида $O(n^k)$. В качестве другого непосредственного примера можно указать задачу умножения матриц, для которой $f_{Amin}(n) = O(n^{2.34})$ [4.2], а тривиальная оценка, отражающая необходимость просмотра всех элементов исходных матриц $f_{tr}(n) = \Theta(n^2)$, при этом теоретическая нижняя граница временной сложности для этой задачи пока не доказана.

§ 4. Классификации вычислительных алгоритмов по трудоемкости

Введение в классификацию алгоритмов. Важной составной частью любого научного исследования некоторой совокупности объектов являются классификации, выявляющие характерные общие свойства объектов и закладывающие теоретическую базу их дальнейших исследований. В рамках разработки математического и алгоритмического обеспечения компьютерных систем такой совокупностью объектов являются алгоритмы решения задач, на которых базируются разрабатываемые программные системы. Впечатляющий рост производительности современных компьютерных систем не снижает требований к алгоритмическому обеспечению, в том числе и по временной эффективности. Обеспечение временных требований связано, при выбранной аппаратной среде реализации, с выбором или разработкой математических методов и/или алгоритмов решения поставленных задач. Решение актуальной в практических целях задачи выбора рационального алгоритма может опираться и на специальные классификации вычислительных алгоритмов. Определенный вклад в более эффективное решение этой задачи могут внести классификации алгоритмов, отражающие как сложностные оценки алгоритмов, так и влияние на эти оценки характеристических особенностей множества исходных данных. Традиционная классификация теории алгоритмов связана с теорией сложности вычислений, в рамках которой получен целый ряд важных результатов, относящихся к классам задач, являющихся объектом исследований в этой теории [4.12]. К сожалению, объектами исследований классической теории алгоритмов являются вычислительные задачи, и принятые в этой теории классификации (классы задач) не могут

быть автоматически перенесены на алгоритмы решения этих задач. Это связано с тем, что теория сложности оперирует классами задач, а не классами алгоритмов, и большинство определений классов задач, за исключением классов P и EXP , не включает в себя явного указания сложностных оценок [4.2, 4.12].

Угловая мера асимптотического роста функций. В рамках теоретического исследования алгоритмов представляет интерес более детальное разграничение сложностных оценок функций трудоемкости, сохраняющее традиционное выделение полиномиальной и экспоненциальной сложности. Таким образом, речь идет о математической задаче разделения полиномов и экспонент в рамках единой меры, с выделением множества функций, разграничитывающих полиномы и экспоненты, и дополнительных множеств субполиномиальных и надэкспоненциальных функций. Один из возможных вариантов решения этой задачи предложен в [4.14]. Обозначим через p длину входа алгоритма, а через $f = f(n)$ — функцию сложности алгоритма. В рамках дальнейшего изложения будем считать, что аргумент x непрерывен, т. е. $f = f(x)$, а необходимые значения функции $f(x)$ вычисляются в целочисленных точках $x = n$. Для разграничения полиномов и экспонент предлагается использовать функцию степенного логарифма $g(x) = (\ln x)^{\ln x}$.

Утверждение 4.1. Функция степенного логарифма $g(x) = (\ln x)^{\ln x}$ является разграничитывающей для полиномов и экспонент.

Доказательство. Утверждение эквивалентно тому, что функция $g(x)$ удовлетворяет следующим двум соотношениям при $x \rightarrow \infty$:

$$\text{если } f(x) = x^k, \quad k > 0, \quad \text{то } f(x) = o(g(x)), \quad (4.4.1)$$

$$\text{если } f(x) = e^{\lambda x}, \quad \lambda > 0, \quad \text{то } g(x) = o(f(x)). \quad (4.4.2)$$

Для доказательства этих соотношений воспользуемся леммой о логарифмическом пределе: если $\lim_{x \rightarrow \infty} f(x) = \infty$, и $\lim_{x \rightarrow \infty} g(x) = \infty$, то если

$$\lim_{x \rightarrow \infty} \frac{\ln f(x)}{\ln g(x)} = 0, \quad \text{то} \quad \lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = 0, \quad \text{т. е.} \quad f(x) = o(g(x)).$$

На основании этой леммы покажем справедливость соотношения (4.4.1):

$$\lim_{x \rightarrow \infty} \frac{\ln(x^k)}{\ln((\ln x)^{\ln x})} = \lim_{x \rightarrow \infty} \frac{k \cdot \ln x}{\ln x \cdot \ln(\ln x)} = \lim_{x \rightarrow \infty} \frac{k}{\ln(\ln x)} = 0,$$

следовательно, $x^k = o((\ln x)^{\ln x})$ при $k > 0$, и соотношения (4.4.2):

$$\lim_{x \rightarrow \infty} \frac{\ln((\ln x)^{\ln x})}{\ln(e^{\lambda x})} = \lim_{x \rightarrow \infty} \frac{\ln x \cdot \ln(\ln x)}{\lambda x} = 0,$$

следовательно, $(\ln x)^{\ln x} = o(e^{\lambda x})$ при $\lambda > 0$. Конец доказательства.

Угловая мера асимптотического роста функций вводится в [4.14] следующим образом: пусть дана функция $f(x)$, монотонно возрастающая, и $\lim_{x \rightarrow \infty} f(x) = \infty$. Поставим ей в соответствие функцию

$$h(x) = \ln(f(x)) + \frac{\ln(f(x))}{\ln(f(x)) + \ln x} \cdot x. \quad (4.4.3)$$

Функция $h(x)$ обладает следующими свойствами, которые устанавливаются двумя леммами.

Лемма 4.1. Пусть $f(x) = e^{\lambda x}(1 + \gamma(x))$, где $\lambda > 0$, $\gamma(x) = o(1)$, $\gamma'(x) = o(1)$ при $x \rightarrow \infty$, тогда $\lim_{x \rightarrow \infty} h'(x) = \lambda + 1$.

Лемма 4.2. Пусть $f(x) = x^k(1 + \gamma(x))$, где $\gamma(x) = o(1)$, $\gamma'(x) = o(1)$, $x\gamma'(x) = O(1)$ при $x \rightarrow \infty$, тогда $\lim_{x \rightarrow \infty} h(x) = k/(k+1)$.

Равенство константе предела производной функции $h(x)$ как для полиномов, так и для экспонент позволяет доказать следующую лемму, вводящую преобразование координатной системы.

Лемма 4.3. Пусть дана функция $h(x)$, такая, что $\lim_{x \rightarrow \infty} h'(x) = C$, где $C > 0$. Рассмотрим образованную на основе функции $h(x)$ параметрически заданную функцию $z(s)$, определенную следующим образом:

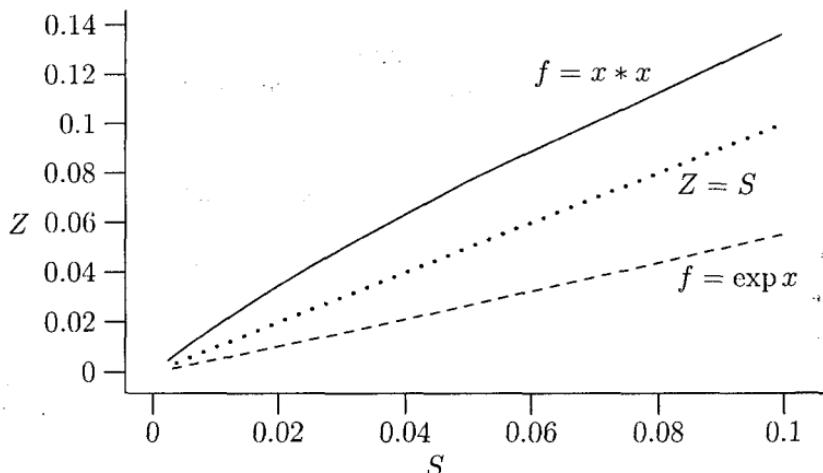
$$z(s) = \begin{cases} s - \arctg\left(\frac{1}{x}\right); \\ z = \arctg\left(\frac{1}{h(x)}\right); \end{cases} \quad \text{тогда} \quad \lim_{\substack{x \rightarrow \infty \\ (s \rightarrow 0)}} \frac{dz}{ds} = \frac{1}{C}. \quad (4.4.4)$$

Графически полученный в лемме 4.3 результат может быть интерпретирован следующим образом. В системе координат (z, s) полиномы и экспоненты отображаются в функции, имеющие в асимптотике при $x \rightarrow \infty$, $s \rightarrow 0$ разные углы наклона касательной в точке $(z = 0, s = 0)$, что и определило название угловой меры асимптотического роста функций. Пример функций $z(s)$, полученных по формуле (4.4.4) для $f(x) = x^2$ и $f(x) = e^x$, приведен на рис. 4.3.

Леммы 4.1, 4.2 и 4.3 служат основой следующей теоремы, доказанной в [4.14].

Теорема 4.1 (об угловой мере асимптотического роста функций). Пусть дана функция $f(x)$, монотонно возрастающая, и $\lim_{x \rightarrow \infty} f(x) = \infty$. Определим меру $\pi(f(x))$ асимптотического (на бесконечности) роста функции

$$f(x) : \pi(f(x)) = \pi - 2 \cdot \arctg(R), \quad \text{где} \quad R = \lim_{\substack{x \rightarrow \infty \\ (s \rightarrow 0)}} \frac{dz}{ds},$$

Рис. 4.3. Функция $z(s)$ для полинома $f(x) = x^2$ и экспоненты $f(x) = e^x$

где параметрически заданная функция $z(s)$ определена в виде (4.4.4), а функция $h(x)$ задана по функции $f(x)$ следующим образом:

$$h(x) = \ln(f(x)) + \frac{\ln(f(x))}{\ln(f(x)) + \ln x} \cdot x,$$

тогда если

- 1) $f(x) = e^{\lambda x}(1 + \gamma(x))$, где $\lambda > 0$, $\gamma(x) = o(1)$, $\gamma'(x) = o(1)$, при $x \rightarrow \infty$, то $\pi/2 < \pi(f(x)) < \pi$;
- 2) $f(x) = x^k(1 + \gamma(x))$, где $\gamma(x) = o(1)$, $\gamma'(x) = o(1)$, $x\gamma'(x) = O(1)$, при $x \rightarrow \infty$, то $0 < \pi(f(x)) < \pi/2$;
- 3) $f(x) = \ln x^{\ln x}$, то $\pi(f(x)) = \pi/2$.

Свойства угловой меры асимптотического роста функций. Предложенная угловая мера асимптотического роста функций обладает рядом свойств, которые позволяют использовать её для построения классификации алгоритмов по сложности функции трудоемкости. На базе введенной меры $\pi(f(x))$ определим следующие пять функциональных множеств в предположении, что $\lim_{x \rightarrow \infty} f(x) = \infty$:

- 1) Определим множество функций FZ : $FZ = \{f(x) | f(x) \prec x^k, \forall k > 0\}$. Для функции $f(x)$ из множества FZ значение R , определяемое по лемме 4.3, равно $+\infty$, и мера $\pi(f(x)) = \pi - 2 \operatorname{arctg}(R) = 0 \forall f(x) \in FZ$, в частности, $\pi(\ln(x)) = 0$.

- 2) Определим множество полиномов FP : $FP = \{f(x) | \exists k > 0 : f(x) = \Theta(x^k)\}$. Данное определение базируется на лемме 4.2, однако можно показать, что предложенная мера остается в силе и для более широкого класса функций вида $f(x) = \Theta(x^k) \cdot g(x)$, где $g(x) \in FZ$, тогда множество FP может быть определено следующим обра-

зом. Вначале определим множество функций F_k

$$F_k = \{ f(x) \mid x^{k-\varepsilon} \prec f(x) \prec x^{k+\varepsilon}, k > 0, \varepsilon \rightarrow +0, x \rightarrow +\infty \}$$

и на его основе определим множество обобщенных полиномов FP :

$$FP = \{ f(x) \mid \exists k > 0 : f(x) \in F_k \};$$

для функции $f(x)$ из FP значение $R := (k+1)/k$, $k > 0$, по леммам 4.2 и 4.3, и мера $\pi(f(x)) = \pi - 2 \operatorname{arctg}((k+1)/k)$, следовательно $0 < \pi(f(x)) < \pi/2$.

График меры для полиномов представлен на рис. 4.4.

3) Определим множество функций FL

$$FL = \{ f(x) \mid x^k \prec f(x) \prec e^{\lambda x}, \forall k > 0, \forall \lambda > 0 \}.$$

Для функции $f(x)$ из FL значение R , определяемое по лемме 4.3, равно 1, и мера $\pi(f(x)) = \pi - 2 \operatorname{arctg}(1) = \pi/2$, в частности, $\pi(\ln x^{\ln x}) = \pi/2$.

4) Определим множество экспонент FE : $FE = \{ f(x) \mid \exists \lambda > 0 : f(x) = \Theta(e^{\lambda x}) \}$. Данное определение базируется на лемме 4.1, однако можно показать, что предложенная мера остается в силе и для более широкого класса функций вида $f(x) = \Theta(e^{\lambda x}) \cdot g(x)$, где $g(x) \in \{FZ, FP, FL\}$, тогда множество FE может быть определено следующим образом. Определим множество функций F_λ

$$F_\lambda = \left\{ f(x) \mid e^{(\lambda-\varepsilon)x} \prec f(x) \prec e^{(\lambda+\varepsilon)x}, \lambda > 0, \varepsilon \rightarrow +0, x \rightarrow +\infty \right\}$$

и на его основе определим множество обобщенных экспонент FE :

$$FE = \{ f(x) \mid \exists \lambda > 0 : f(x) \in F_\lambda \}.$$

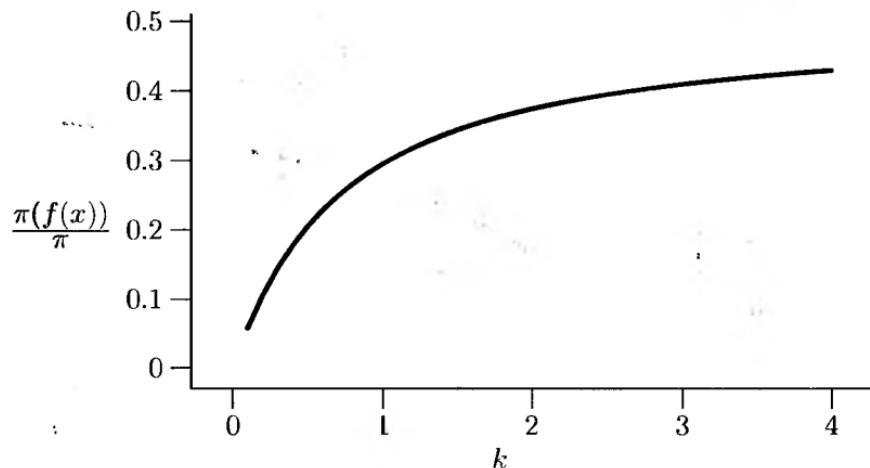


Рис. 4.4. График меры $\pi(f(x))$ для полиномов $f(x) = \Theta(x^k)$

Для функции $f(x)$ из FE значение $R = 1/(1 + \lambda)$, $\lambda > 0$, по леммам 4.1 и 4.3, и мера $\pi(f(x)) = \pi - 2 \operatorname{arctg}(1/(1 + \lambda))$, следовательно, $\pi/2 < \pi(f(x)) < \pi$. График меры для экспонент представлен на рис. 4.5.

5) Определим множество функций FF : $FF = \{f(x) \mid e^{\lambda x} \prec \prec f(x), \forall \lambda > 0\}$. Для функции $f(x)$ из FF значение R , определяемое по лемме 4.3, равно 0, и мера $\pi(f(x)) = \pi - 2 \operatorname{arctg}(R) = \pi$, в частности $\pi(e^x) = \pi$.

Укажем свойства, которыми обладает введенная угловая мера асимптотического роста функций — $\pi(f(x))$:

- мера принимает значение, равное $\pi/4$, для степени $k = \sqrt{2}/2$;
- мера $\pi(x^k)$ принимает значение, равное $3\pi/4$, при показателе $\lambda = \sqrt{2}$;
- мера $\pi(f(x))$ обладает интересным свойством:

$$\pi(x^{1/\lambda}) + \pi(e^{\lambda x}) = \pi, \quad \text{в частности, } \pi(x) + \pi(e^x) = \pi.$$

Классификация алгоритмов по сложности функции трудоемкости. Использование угловой меры асимптотического роста функций $\pi(f(x))$ позволяет предложить следующую *классификацию алгоритмов* по асимптотике роста функции трудоемкости (для среднего или худшего случаев). Сохраняя общепринятое обозначение n для размерности входа алгоритма A , обозначая через $f_A^*(n)$ функцию сложности и подразумевая формальный переход от n к x при вычислении $\pi(f(x))$, введем следующее теоретико-множественное определение классов [4.14].

1. Класс π_0 (пи нуль) — класс «быстрых алгоритмов» — это алгоритмы, для которых функции сложности принадлежат множеству

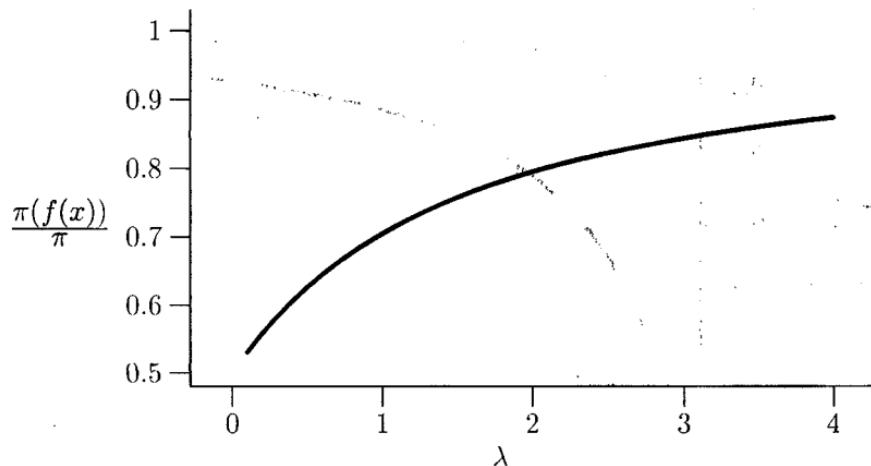


Рис. 4.5. График меры $\pi(f(x))$ для экспонент $f(x) = \Theta(e^{\lambda x})$

FZ и имеют меру ноль:

$$\pi 0 = \{ A \mid \pi(f_A^*(n)) = 0 \Leftrightarrow f_A^*(n) \in FZ \}.$$

Алгоритмы, принадлежащие этому классу, являются существенно быстрыми относительно длины входа; в основном это алгоритмы, имеющие полилогарифмическую или логарифмическую сложность. Так, например, к этому классу относится алгоритм бинарного поиска в массиве отсортированных ключей — асимптотическая оценка его трудоемкости — $O(\ln(n))$ [4.2], мера $\pi(\ln(n)) = 0$.

2. Класс πP — класс «рациональных (собственно полиномиальных) алгоритмов» — это алгоритмы, функции сложности которых принадлежат множеству FP :

$$\pi P = \{ A \mid 0 < \pi(f_A^*(n)) < \pi/2 = 0 \Leftrightarrow f_A^*(n) \in FP \}.$$

К этому классу относится большинство реально используемых алгоритмов, позволяющих решать вычислительные задачи за рациональное время; отметим, что этот класс обладает свойством естественной замкнутости. Введенный класс алгоритмов πP является подклассом алгоритмов, определяющих класс задач P в теории сложности вычислений.

3. Класс πL — класс «субэкспоненциальных алгоритмов» — это алгоритмы, функции сложности которых принадлежат множеству FL :

$$\pi L = \{ A \mid \pi(f_A^*(n)) = \pi/2 \Leftrightarrow f_A^*(n) \in FL \}.$$

Этот класс образуют алгоритмы с более чем полиномиальной, но менее чем экспоненциальной сложностью. Эти алгоритмы достаточно трудоёмки, соответствующие задачи, как правило, принадлежат сложностному классу NP , но для некоторых задач такие алгоритмы применяются на практике. Примером может служить алгоритм факторизации больших составных чисел методом обобщенного числового решета, применяемый для прямых атак на крипtosистему RSA . Если n есть количество битов числа, предъявляемого для факторизации, то эвристическая оценка сложности этого алгоритма, приведенная в [4.16], имеет вид

$$f_A(n) = \exp \left\{ O \left(n^{\frac{1}{3}} \cdot (\ln(n))^{\frac{2}{3}} \right) \right\}, \quad \text{и} \quad \pi(f_A(n)) = \pi/2.$$

Обозначение L в названии класса отражает тот факт, что функция степенного логарифма $g(x) = (\ln x)^{\ln x}$ является одной из функций, разграничивающих полиномы и экспоненты в силу теоремы 4.1.

4. Класс πE — класс «собственно экспоненциальных алгоритмов» — это алгоритмы, функции сложности которых принадлежат множеству FE :

$$\pi E = \{ A \mid \pi/2 < \pi(f_A^*(n)) < \pi \Leftrightarrow f_A^*(n) \in FE \}.$$

Это алгоритмы с экспоненциальной трудоемкостью, на сегодня практически применимые только для малой длины входа, возможности

реального использования таких алгоритмов связаны с практической реализацией квантовых компьютеров. Примерами алгоритмов этого класса являются переборные алгоритмы для точного решения NP -полных задач, таких, как задача о выполнимости схемы, задача о сумме, задача о клике и т. д. [4.2], имеющие асимптотические оценки трудоемкости (сложность) вида

$$O(2^n), O(n \cdot 2^n), O(n^2 \cdot 2^n).$$

5. Класс πF — класс «надэкспоненциальных алгоритмов» — это алгоритмы, функции сложности которых принадлежат множеству FF :

$$\pi E = \{ A \mid \pi(f_A^*(n)) = \pi \Leftrightarrow f_A^*(n) \in FF \}.$$

Это класс практически неприменимых алгоритмов, обладающих более чем экспоненциальной трудоемкостью факториального или показательно-степенного вида. Например, алгоритм решения задачи коммивояжера методом полного перебора имеет оценку $\Omega(n!)$, а поскольку $n! \sim \Gamma(n+1)$, где $\Gamma(x)$ — гамма-функция Эйлера, и $\ln(\Gamma(x+1)) \approx x \cdot \ln x - x$, то $n! \approx e^{n \cdot (\ln n - 1)}$, поскольку мера $\pi(e^{x(\ln x - 1)}) = \pi$, то этот алгоритм относится к классу πF . К этому же классу относится алгоритм полного перечисления всех остовых деревьев полного графа на n вершинах с асимптотической оценкой трудоемкости $\Omega(n^{n-2})$ [4.15]. Обозначение F в названии класса отражает принадлежность к этому классу алгоритмов с факториальной (Factorial) оценкой трудоемкости.

Классификации алгоритмов по степени влияния характеристических особенностей множества исходных данных на функцию трудоемкости. Поскольку значение функции трудоемкости может определяться не только размером входа n , но и другими характеристиками множества D (значениями и порядком расположения элементов), то выделим в функции $f_A(n, D)$ количественную и параметрическую составляющие, обозначив их через $f_n(n)$ и $g_p(D)$ соответственно [4.4], $f_A(n, D) = f_A(f_n(n), g_p(D))$. Для большинства алгоритмов функция $f_A(n, D)$ может быть представлена как композиция $f_n(n)$ и $g_p(D)$ либо в мультиплективной $f_A(n, D) = f_n(n)g_p^*(D)$, либо в аддитивной форме $f_A(n, D) = f_n(n) + g_p^+(D)$.

Укажем, что мультиплективная форма характерна для ряда алгоритмов, когда сильно параметрически зависимый внешний цикл, определяющий перебор вариантов, содержит внутренний цикл, проверяющий решение с полиномиальной зависимостью от размерности. Такая ситуация возникает, например, для ряда алгоритмов, решающих задачи из класса NPC [4.2]. В силу конечности множества D_n (см. 4.2) существует худший случай для функций $g_p^*(D)$ и $g_p^+(D)$. Обозначим соответствующие функции через $g^*(n)$ и $g^+(n)$:

$$g^*(n) = \max_{D \in D_n} \{ g_p^*(D) \}, \quad g^+(n) = \max_{D \in D_n} \{ g_p^+(D) \}.$$

Отметим, что функция $g^+(n)$ может быть выражена через функцию $g^*(n)$ следующим образом:

$$g^+(n) = g^*(n) f_n(n) - f_n(n).$$

В зависимости от влияния различных характеристик множества исходных данных на функцию трудоемкости алгоритма может быть предложена следующая классификация, имеющая практическое значение для анализа ресурсной эффективности алгоритмов [4.3, 4.4].

I. Класс N — класс количественно-зависимых по трудоемкости алгоритмов. Это алгоритмы, функция трудоемкости которых зависит только от размерности конкретного входа

$$g^+(n) = 0 \Rightarrow g^*(n) = 1 \Rightarrow f_A(n, D) = f_n(n).$$

Примерами алгоритмов с количественно-зависимой функцией трудоемкости могут служить алгоритмы для стандартных операций с массивами и матрицами — умножение матриц, умножение матрицы на вектор и т. д. Анализ таких алгоритмов, как правило, не вызывает затруднений. Заметим, что для алгоритмов класса N справедливо соотношение $f_A^\wedge(n) = f_A^\vee(n) = f_A(n)$.

II. Класс PR — класс параметрически-зависимых по трудоемкости алгоритмов. Это алгоритмы, трудоемкость которых определяется не размерностью входа (как правило, для этого класса размерность входа обычно фиксирована), а конкретными значениями всех или некоторых элементов из входного множества D

$$f_n(n) = \text{const} = c \Rightarrow f_A(n, D) = c g^*(n).$$

Примерами алгоритмов с параметрически-зависимой трудоемкостью являются алгоритмы вычисления стандартных функций с заданной точностью путем вычисления соответствующих степенных рядов. Очевидно, что такие алгоритмы, имея на входе два числовых значения — аргумент функции и точность, задают существенно зависящее от значений количество операций. Например, вычисление x^k последовательным умножением — $f_A(D) = f_A(x, k)$ или алгоритм вычисления $e^x = \sum (x^n/n!)$, с точностью до ε — $f_A(D) = f_A(x, \varepsilon)$.

III. Класс NPR — класс количественно-параметрических по трудоемкости алгоритмов. Это достаточно широкий класс алгоритмов, т. к. в большинстве практических случаев функция трудоемкости зависит как от количества данных на входе, так и от значений входных данных. В этом случае

$$f_n(n) \neq \text{const}, \quad g^*(n) \neq 1 \Rightarrow f_A^\wedge(n) \neq f_A^\vee(n),$$

$$f_A(n, D) = f_n(n) g^*(n), \quad \text{или} \quad f_A(n, D) = f_n(n) + g^+(n).$$

В качестве одного из общих примеров можно привести ряд алгоритмов численных методов, в которых параметрически-зависимый внешний

цикл по точности включает в себя количественно-зависимый фрагмент по размерности, порождая мультиплекативную форму для $f_A(n, D)$.

Подклассы класса PR . Достаточно часто алгоритмы класса PR обладают трудоемкостью, которая зависит только от фиксированного числа параметров множества исходных данных. Рассмотрим ситуацию, когда имеется только один параметр, определяющий трудоемкость. Таким образом, $f_A(D)$ зависит либо от количества бит в двоичном представлении этого параметра, либо от их значений. На основании этого рассуждения мы можем ввести дополнительные подклассы в классе PR , обозначив через m число значащих бит параметра. В этих предположениях и обозначениях $f_A(D) = f_A(m)$, и мы можем определить принадлежность функции $f_A(m)$ к одному из классов сложности. Тогда будем говорить, что алгоритм принадлежит подклассу $PlPR$ (полиномиальный подкласс), если $f_A(m) \in FP$, и подклассу $ExPR$ (экспоненциальный подкласс), если $f_A(m) \in FE$. Содержательно к подклассу $PlPR$ относятся алгоритмы, трудоемкость которых зависит от числа бит параметра, а к подклассу $ExPR$ — алгоритмы с зависимостью трудоемкости от числового значения этого параметра. Конкретные примеры таких алгоритмов мы приведем в главе 7.

Подклассы класса NPR . По отношению к количественно-параметрическим алгоритмам представляет интерес более детальная классификация, ставящая целью выяснение степени влияния количественной и параметрической составляющей на главный порядок функции $f_A(n, D) = f_n(n)g^*(n)$ и приводящая к выделению следующих подклассов в классе NPR [4.3]:

III.1. Подкласс $NPRL$ (*Low*) — подкласс алгоритмов, трудоемкость которых слабо зависит от параметрической составляющей

$$g^+(n) = O(f_n(n)) \Leftrightarrow g^*(n) = \Theta(1).$$

Таким образом, для алгоритмов этого подкласса параметрическая составляющая влияет не более чем на коэффициент главного порядка функции трудоемкости, который определяется количественной составляющей. В этом случае можно говорить о коэффициентно-параметрической функции трудоемкости. К этому подклассу относится, например, алгоритм поиска максимума в массиве, т. к. количество переприсваиваний максимума в худшем случае (массив отсортирован по возрастанию), определяющее $g^+(n) = \Theta(n)$, имеет равный порядок с оценкой внешнего цикла перебора n элементов.

III.2. Подкласс $NPRE$ (*Equivalent*) — подкласс алгоритмов, в трудоемкости которых составляющая $g^*(n)$ имеет порядок роста, не превышающий $f_n(n)$:

$$g^*(n) = \Theta(f_n(n)) \Leftrightarrow g^+(n) = \Theta(f_n^2(n)).$$

Таким образом, для алгоритмов этого подкласса параметрический компонент имеет сопоставимое влияние (в мультипликативной форме) с количественным компонентом на главный порядок функции трудоемкости. Для алгоритмов, относящихся к этому подклассу, можно говорить о квадратично-количественной функции трудоемкости. В этот подкласс входит, например, алгоритм сортировки массива методом пузырька, для которого количество перестановок элементов в худшем случае (обратно отсортированный массив) определяет $g^+(n) = \Theta(n^2)$, а $f_n(n) = \Theta(n)$.

III.3. Подкласс $NPRH$ (*High*) — подкласс алгоритмов, в трудоемкости которых составляющая $g^*(n)$ имеет асимптотический порядок роста выше $f_n(n)$

$$f_n(n) = o(g^*(n)).$$

Таким образом, для алгоритмов этого подкласса именно параметрический компонент определяет главный порядок функции трудоемкости. Можно говорить, что эти алгоритмы являются количественно-сильно-параметрическими по функции трудоемкости алгоритмами. В этот подкласс входят, например, уже упоминавшиеся алгоритмы точного решения NP -полных задач из класса πE . Для алгоритмов этого подкласса характерна, как правило, мультипликативная форма функции трудоемкости: $f_A(n, D) = f_n(n) g_p(D)$, что позволяет говорить о количественно-итерационно-параметрическом характере $f_A(n, D)$. Мультипликативная форма особенно ярко выражена в алгоритмах большинства итерационных вычислительных методов, в которых внешний цикл по точности порождает параметрическую составляющую, а трудоемкость тела цикла имеет количественную оценку.

Подклассы класса NPR по характеристикам особенностям множества исходных данных. Другая, не зависящая от предыдущей, классификация алгоритмов в классе NPR [4.3] предполагает выделение в функции $g_p(D)$ аддитивных компонентов, связанных со значениями элементов входа — $g_v(D)$ и их порядком — $g_s(D)$. Мы представляем функцию $g_p(D)$ в виде

$$g_p(D) = g_v(D) + g_s(D).$$

Выделим во множестве $D \in D_n$ подмножество однородных по существу задачи элементов D_e , состоящее из элементов $\{d_1, \dots, d_m\}$, $|D_e| = m$, и определим D_p как множество всех упорядоченных последовательностей из $\{d_1, \dots, d_m\}$, отметим, что в общем случае $|D_p| = m!$. Если $D_{e1}, D_{e2} \in D_p$, то соответствующие им множества $D_1, D_2 \in D_n$, тогда порядковая зависимость $g_s(D)$ для функции трудоемкости имеет место, если $f_A(n, D_1) \neq f_A(n, D_2)$ хотя бы для одной пары $D_1, D_2 \in D_n$. Зависимость трудоемкости от значений элементов входа предполагает, что

$$f_A(n, D) = f_A(n, p_1, \dots, p_m), \quad m \leq n, \quad p_i \in D, \quad i = \overline{1, m}.$$

Оценивая степень влияния $g_v(D)$, $g_s(D)$ на $g_p(D)$, можно определить следующие подклассы в классе *NPR*.

III.а. Подкласс *NPRS* (*Sequense*) — подкласс алгоритмов с количественной и порядково-зависимой функцией трудоемкости: $g_v(D) = o(g_s(D))$. В этом случае трудоемкость зависит от размерности входа и от порядка расположения однородных элементов; зависимость от значений не может быть полностью исключена, но она не является существенной. Мы можем говорить о количественно-порядковом характере функции трудоемкости. Примерами могут служить большинство алгоритмов сортировки сравнениями, алгоритмы поиска минимума и максимума в массиве.

III.б. Подкласс *NPRV* (*Value*) — подкласс алгоритмов с функцией трудоемкости, зависящей от длины входа и значений элементов в D : $g_s(D) = o(g_v(D))$. В этом случае трудоемкость зависит от размерности входа и от значений элементов входного множества; зависимость от порядка не является определяющей. В этот подкласс входит алгоритм решения задачи упаковки методом динамического программирования (табличный метод), для которого функция трудоемкости зависит как от количества типов грузов, так и от значений объемов грузов и упаковки. Порядок обработки типов грузов не является определяющим. Другой пример — алгоритм сортировки методом индексов [4.2], трудоемкость которого определяется количеством исходных чисел и значением максимального из них. При этом порядок чисел в массиве вообще не оказывает влияния на трудоемкость, за исключением фрагмента поиска максимума, лежащего в классе *NPRS*.

Отметим, что объединение этих подклассов не образует класс *NPR*:

$$NPR \dashv (NPRS \cup NPRV) \neq \emptyset.$$

Существуют алгоритмы, в которых и значения, и порядок расположения однородных элементов оказывают реальное и существенное влияние на функцию трудоемкости — например, таковыми являются итерационные алгоритмы решения систем линейных уравнений. Очевидно, что как перестановка значений в исходной матрице, так и изменение самих значений существенно меняют собственные числа матрицы, которые определяют сходимость итерационного процесса получения решения [4.17].

§ 5. Информационная и размерностная чувствительность вычислительных алгоритмов

В некоторых проблемных областях применения программных систем, например, для бортовых компьютерных систем, возникает требование временной устойчивости. Практически это означает, что для различных входов фиксированной длины время работы программы должно изменяться незначительно. Поэтому в целях анализа, исследования

и выбора алгоритмов представляет интерес учет влияния различных характеристик множества исходных данных на функцию трудоемкости. Отметим, что в классической теории сложности рассматривается только одна такая характеристика, а именно длина входа. В рамках исследования устойчивости временных характеристик алгоритмов, на основе введения понятия информационной чувствительности, представляет интерес и разработка соответствующих классификаций.

Информационная чувствительность алгоритмов по трудоемкости. Практически значимыми результатами анализа ресурсной эффективности некоторого алгоритма является получение таких сведений, которые могли бы дать возможность прогнозирования требуемых этим алгоритмом ресурсных затрат при решении задач из данной проблемной области. В аспекте ресурса процессора мы хотели бы прогнозировать трудоемкость алгоритма как для разных размерностей задачи, так и для различных входных данных. Идеальным результатом прогнозирования трудоемкости алгоритма можно считать получение точной функции $f_L(D)$. К сожалению, такая функция может быть реально получена только для количественно-зависимых алгоритмов, образующих класс N , и, может быть, для отдельных алгоритмов других классов, ввиду сложности формального описания влияния параметрической составляющей на трудоемкость. Для большинства алгоритмов класса NPR получение точной функции трудоемкости затруднительно даже для относительно простых алгоритмов, а использование оценки в среднем позволяет прогнозировать трудоемкость только при усреднении по большой выборке входов.

Более детальное рассмотрение задачи прогнозирования связано с изучением влияния характеристик множества исходных данных на функцию трудоемкости алгоритма. Традиционно влияние изменений параметров входа на выходную характеристику изучаемого объекта называется чувствительностью по входному параметру. Таким образом, можно говорить о чувствительности функции трудоемкости алгоритма к исходным данным. Поскольку трудоемкость алгоритма зависит как от количества элементов множества исходных данных, так и от параметров этого множества, то можно выделить различные аспекты чувствительности алгоритмов, соответствующие определения введены в [4.10].

Определение 4.7. Под размерностной чувствительностью алгоритма будем понимать влияние изменения размерности на значение функции трудоемкости.

Определение 4.8. Под информационной чувствительностью алгоритма будем понимать влияние различных входов фиксированной размерности на изменение значения функции трудоемкости алгоритма.

Количественная мера информационной чувствительности алгоритмов. Для иллюстрации понятия информационной чувствительности рассмотрим качественно вид функции трудоемкости для некоторого алгоритма, принадлежащего классу NPR , представленной на

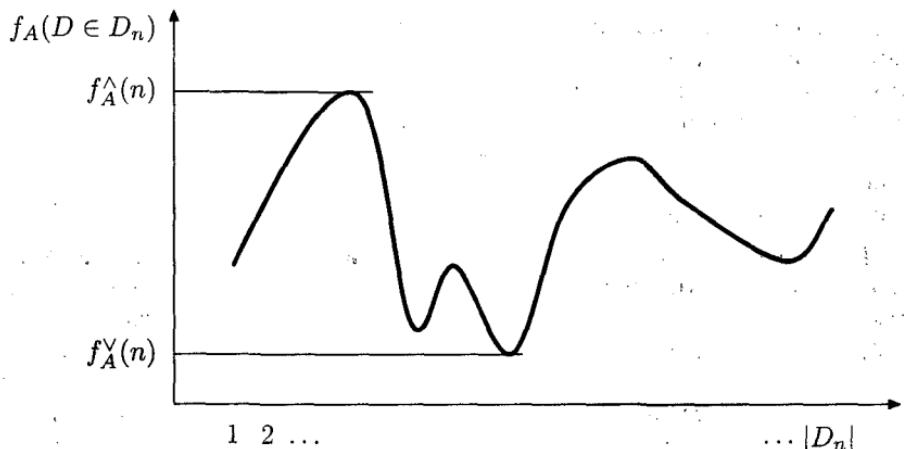


Рис. 4.6. График трудоемкости количественно-параметрического алгоритма для различных входов фиксированной размерности

рис. 4.6. Здесь по оси абсцисс отложены номера конкретных входов алгоритма с размерностью n , принадлежащих множеству D_n , всего таких входов $|D_n|$. Мощность множества D_n значительна, однако конечна, в предположении о конкретной реализации алгоритма, в силу ограниченности представления чисел в компьютере. В случае, если n — количество битов на входе, то максимально $|D_n| = 2^n$, если все битовые комбинации являются допустимыми для данного алгоритма. Реально мы имеем дело с целочисленной функцией целочисленного аргумента, поэтому график представляет собой набор точечных значений. Для наглядности на рис. 4.6 эти точечные значения соединены кривой. Отметим, что качественно вид графика очень сильно зависит от принятого способа нумерации элементов множества D_n . Для какого-то входа алгоритм выполняет максимальное количество операций, что соответствует худшему случаю для данной размерности $f_A^{\wedge}(n)$, аналогично в лучшем случае количество операций будет минимально — $f_A^{\vee}(n)$, и трудоемкость для любого входа будет заключена между этими крайними значениями.

Подсчитывая относительную по размерности множества D_n частотную встречаемость того или иного значения функции трудоемкости f_A , мы можем получить гистограмму относительных частот значений функции трудоемкости $P(f_A)$ как дискретной случайной величины. Возможный вид соответствующей сглаженной кривой показан на рис. 4.7.

При этом очевидно выполнено следующее условие

$$\sum P(f_A) = 1,$$

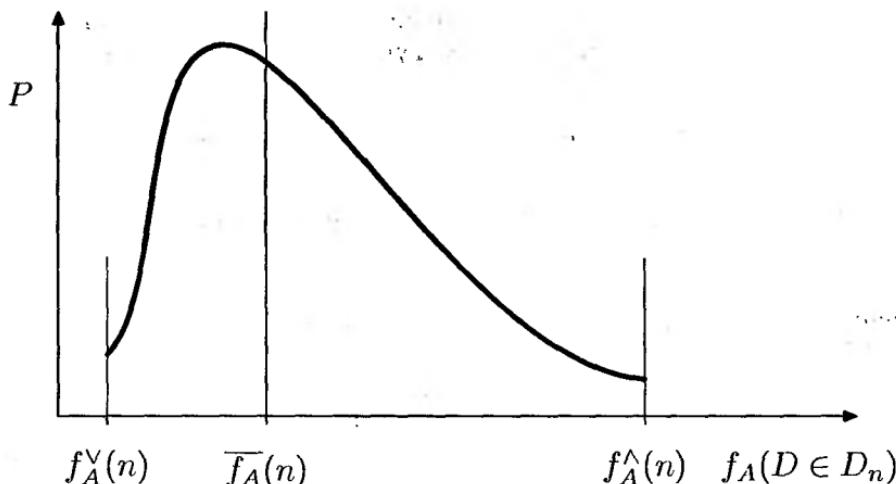


Рис. 4.7. График относительных частот значений функции трудоемкости для количественно-параметрического алгоритма

тогда среднее значение трудоемкости может быть определено как

$$\overline{f_A}(n) = \sum f_A \cdot P(f_A),$$

где обе суммы берутся по всем целочисленным значениям функции трудоемкости f_A на множестве D_n : $f_A^V \leq f_A \leq f_A^H$. Таким образом, мы получаем более детальную информацию о поведении алгоритма, рассматривая функцию трудоемкости как дискретную случайную величину, характеризуемую математическим ожиданием и дисперсией и ограниченную минимальным и максимальным значениями. В ряде случаев значения математического ожидания и дисперсии могут быть получены теоретически на основании совокупного анализа входов фиксированной длины, для некоторых алгоритмов такие оценки получены, например, Д. Кнутом [4.18].

Вводя понятие информационной чувствительности и в дальнейшем обозначая ее через $\delta_i(n)$, мы хотим установить количественную меру разброса или рассеяния значений функции трудоемкости относительно среднего значения для различных входов фиксированной длины с учетом граничных значений. В общем случае этот разброс будет зависеть от размерности входа, поэтому $\delta_i(n)$ вводится как функция от аргумента n . Мерой рассеяния случайной величины относительно математического ожидания является дисперсия D или среднеквадратическое отклонение σ . Верхняя граница σ_f для среднеквадратического отклонения ограниченной случайной величины, заданной функцией $f(x)$ определяется следующей теоремой [4.10].

Теорема 4.2. Непрерывная случайная величина, ограниченная по x сегментом $[a, b]$ и заданная на нем функцией распределения $f(x)$

$$\forall x : a \leq x \leq b : f(x) \geq 0, \quad \text{и} \quad \int_a^b f(x) dx = 1,$$

имеет максимальную дисперсию $D[f(x)]$, если функция $f(x)$ имеет вид

$$f(x) = q\delta(x - a) + p\delta(x - b),$$

причем $p = q = 1/2$, где $\delta(x)$ — дельта-функция, при этом $D[f(x)] \leq 1/4 \cdot (a - b)^2$.

Переходя к дискретным случайным величинам, мы можем на основании теоремы 4.2 говорить, что дисперсия $\sigma_{f_A}^2$ будет максимальна в случае, когда функция трудоемкости как дискретная случайная величина принимает равновероятно только минимальное и максимальное значение. В этом случае, как легко видеть, математическое ожидание и среднеквадратическое отклонение принимают следующие значения:

$$M[f_A(n)] = (f_A^\wedge(n) + f_A^\vee(n)) / 2, \quad \sigma_{f_A \max}(n) = (f_A^\wedge(n) - f_A^\vee(n)) / 2. \quad (4.5.1)$$

Корректное определение информационной чувствительности должно также учитывать размер интервала возможных значений функции трудоемкости. При одинаковом значении дисперсии более чувствительным должен быть алгоритм с большим интервалом возможных значений. Для этого будем использовать такое понятие математической статистики, как размах варьирования $R = (f_A^\wedge(n) - f_A^\vee(n))$ [4.19]. Отметим, что значение $\sigma_{f_A \max}(n)$ равно половине размаха варьирования в соответствии с определением R . В целях дальнейшего рассмотрения информационной чувствительности введем понятие нормированного (относительного) размаха варьирования функции трудоемкости $R_N(n)$ для входов длины n как отношение половины вариантного интервала к его середине:

$$R_N(n) = (f_A^\wedge(n) - f_A^\vee(n)) / (f_A^\wedge(n) + f_A^\vee(n)). \quad (4.5.2)$$

Поскольку все значения функции трудоемкости положительны и трудоемкость в худшем случае не меньше, чем в лучшем — $f_A^\wedge(n) \geq f_A^\vee(n)$, то $0 \leq R_N(n) \leq 1$.

Еще одной стандартной характеристикой вариационного ряда является генеральный коэффициент вариации V , определяемый как отношение генерального среднеквадратического отклонения к генеральному среднему значению [4.19]; для функции трудоемкости этот коэффициент имеет вид

$$V(n) = \sigma_{f_A(n)} / \bar{f}_A(n), \quad 0 \leq V(n) \leq 1, \quad (4.5.3)$$

где $\sigma_{f_A(n)}$ — среднеквадратическое отклонение функции трудоемкости, как дискретной случайной величины при фиксированной размерности

входа n . Обратим внимание на то, что в худшем случае для дисперсии функции трудоемкости генеральный коэффициент вариации совпадает с нормированным размахом варьирования в соответствии с (4.5.2) и (4.5.3). Поскольку $\sigma_{f_A(n)}$ может изменяться от 0 до $\sigma_{f_A \max}(n)$, то введем, следуя [4.10], количественную меру информационной чувствительности алгоритма в виде

$$\delta_i(n) = V(n) R_N(n), \quad 0 \leq \delta_i(n) \leq 1. \quad (4.5.4)$$

Тем самым информационная чувствительность алгоритма при фиксированной длине входа учитывает и среднеквадратическое отклонение функции трудоемкости, и размер интервала возможных значений в нормированных единицах.

Предложенное понятие и способ определения количественной меры информационной чувствительности алгоритмов могут быть использованы как дополнительный инструмент детального исследования алгоритмов. Например, количественная мера информационной чувствительности может быть применена для более обоснованного решения задачи выбора рациональных алгоритмов в рамках анализа ресурсной эффективности. В том случае, если к программной системе предъявляются временные требования с узкими границами, то рациональным является выбор алгоритма с малой количественной мерой информационной чувствительности. Такой подход приводит к постановке задачи классификации алгоритмов по введенной количественной мере информационной чувствительности.

Размерностная чувствительность алгоритмов по трудоемкости. Понятие информационной чувствительности отражает вероятностное варьирование трудоемкости алгоритма для различных входов фиксированной длины. Понятие размерностной чувствительности отражает изменение трудоемкости при изменении длины входа. Однако если мы будем рассматривать такое изменение для конкретных входов с различной длиной, то наблюдаемое изменение будет обусловлено как размерностной, так и информационной чувствительностью. Поэтому размерностную чувствительность есть смысл рассматривать для особых случаев функции трудоемкости при фиксированной длине входа, а именно для трудоемкости в лучшем, среднем и худшем случаях. Введем обозначение $f_A^*(n)$, где под символом * подразумевается одно из обозначений особых случаев трудоемкости

$$* \in \{\wedge, -, \vee\}. \quad (4.5.5)$$

Таким образом, мы вводим три количественных меры размерностной чувствительности алгоритма с обозначением $\delta_n^*(n)$, где символ * определен в соответствии с (4.5.5). Заметим, что поскольку соответствующие функции трудоемкости используются в определении информационной чувствительности, то возникает возможность ее определения для других размерностей на основе размерностной чувствительности

сти. Для формализации размерностной чувствительности необходимо рассмотреть задание функции трудоемкости как в явном функциональном виде, так и в виде рекурсивно заданной функции. Очевидным является требование совпадения определения количественной меры размерностной чувствительности вне зависимости от способа задания функции трудоемкости алгоритма. Для каждого из указанных способов задания функции трудоемкости будем считать, что мера размерностной чувствительности должна отражать изменение трудоемкости, заданной функциями $f_A^*(n)$, при увеличении размерности на единицу. Рассмотрим отдельно оба возможных способа задания функции трудоемкости [4.20].

Случай рекурсивного задания функции. Без потери общности можно считать, что функция $f_A^*(n)$ задана рекурсивно в виде

$$f_A^*(n+1) = h_A^*(n) f_A^*(n). \quad (4.5.6)$$

Поскольку функция $f_A^*(n)$ является, по крайней мере, неубывающей

$$f_A^*(n+1) \geq f_A^*(n), \quad \text{и} \quad \lim_{n \rightarrow \infty} f_A^*(n) = \infty, \quad \text{то} \quad h_A^*(n) \geq 1,$$

то представим функцию $h_A^*(n)$ в виде

$$h_A^*(n) = 1 + \delta_n^*(n), \quad \delta_n^*(n) \geq 0. \quad (4.5.7)$$

Подставляя (4.5.6) в (4.5.7) и выделяя $\delta_n^*(n)$, получим формулу для количественной меры размерностной чувствительности функции трудоемкости алгоритма для лучшего, среднего и худшего случаев исходных данных

$$\delta_n^*(n) = \frac{f_A^*(n+1) - f_A^*(n)}{f_A^*(n)}. \quad (4.5.8)$$

Таким образом, рекурсивно заданная функция $f_A^*(n)$ связана с мерой размерностной чувствительности соотношением

$$f_A^*(n+1) = f_A^*(n) + \delta_n^*(n) f_A^*(n). \quad (4.5.9)$$

Случай задания функции в явном виде. Следуя классическому определению чувствительности, необходимо рассмотреть производную функции $f_A^*(n)$ по размерности, однако для согласования с формулой (4.5.8) предлагается использовать логарифмическую производную

$$\delta_n^*(n) = \frac{d}{dn} \ln(f_A^*(n)) = \frac{(f_A^*(n))'}{f_A^*(n)}. \quad (4.5.10)$$

Если с учетом целочисленности аргумента функции перейти в формуле (4.5.10) от производной к конечной разности

$$(f_A^*(n))' \rightarrow f_A^*(n+1) - f_A^*(n),$$

то мы получим для $\delta_n^*(n)$ формулу (4.5.8), согласованную с определением для рекурсивного случая.

Размерностная чувствительность алгоритмов в подклассах класса NPR . Представляет интерес выяснение поведения меры размерностной чувствительности для различных подклассов алгоритмов в классе NPR как наиболее широком классе практически применяемых алгоритмов. Будем использовать обозначения, введенные в параграфе 4.4: $f_n(n)$ — количественный компонент функции трудоемкости; $g^+(n)$ — параметрический компонент в аддитивной форме; $g^*(n)$ — параметрический компонент в мультипликативной форме.

1. Подкласс $NPR - g^+(n) = O(f_n(n))$. В соответствии с определением понятия «о большое» будем считать, что $g^+(n) \leq c f_n(n)$, тогда для алгоритмов этого подкласса количественная мера размерностной чувствительности имеет вид

$$\delta_n^*(n) = \frac{f_n(n+1) + c f_n(n+1) - f_n(n) - c f_n(n)}{f_n(n) + c f_n(n)} = \frac{f_n(n+1) - f_n(n)}{f_n(n)}. \quad (4.5.11)$$

Таким образом, для алгоритмов класса NPR количественная мера размерностной чувствительности трудоемкости совпадает с мерой его количественного компонента.

2. Подкласс $NPRE - g^*(n) = \Theta(f_n(n))$. В соответствии с определением понятия «тета» будем считать, что $g^*(n) = c f_n(n)$, тогда для алгоритмов этого подкласса количественная мера размерностной чувствительности имеет вид

$$\begin{aligned} \delta_n^*(n) &= \frac{f_n(n+1) c f_n(n+1) - f_n(n) c f_n(n)}{f_n(n) c f_n(n)} = \\ &= \frac{f_n(n+1) - f_n(n)}{f_n(n)} \cdot \frac{f_n(n+1) + f_n(n)}{f_n(n)} = \delta_{f_n}(n) \cdot \left(1 + \frac{f_n(n+1)}{f_n(n)}\right), \end{aligned} \quad (4.5.12)$$

где через $\delta_{f_n}(n)$ обозначена мера размерностной чувствительности количественной компоненты функции трудоемкости. Заметим, что $f_n(n+1) \geq f_n(n)$, следовательно, $\delta_n^*(n) \geq 2\delta_{f_n}(n)$ в силу формулы (4.5.12). Таким образом, для алгоритмов класса $NPRE$ количественная мера размерностной чувствительности не менее чем в два раза превышает количественную меру размерностной чувствительности количественного компонента функции трудоемкости.

3. Подкласс $NPRH - f_n(n) = o(g^*(n))$. Используем запись компонентов функции трудоемкости через размерностную чувствительность в виде (4.5.9), тогда

$$f_n(n+1) = f_n(n) + \delta_{f_n}(n) f_n(n), \quad g^*(n+1) = g^*(n) + \delta_g^* g^*(n),$$

где через $\delta_g^*(n)$ обозначена мера размерностной чувствительности параметрического компонента функции трудоемкости. Тогда для алгорит-

мов этого подкласса количественная мера размерностной чувствительности имеет вид

$$\begin{aligned}\delta_n^*(n) &= \frac{(f_n(n) + \delta_{f_n}(n)f_n(n)) \cdot (g^*(n) + \delta_g^*(n)g^*(n)) - f_n(n)g^*(n)}{f_n(n)g^*(n)} = \\ &= \delta_{f_n}(n) + \delta_g^*(n) + \delta_{f_n}(n)\delta_g^*(n) = \delta_g^*(n) \cdot \left(1 + \delta_{f_n}(n) + \frac{\delta_{f_n}(n)}{\delta_g^*(n)}\right), \quad (4.5.13)\end{aligned}$$

в силу определения подкласса $\delta_g^*(n) \gg \delta_{f_n}(n)$, следовательно,

$$\delta_n^*(n) \approx \delta_g^*(n)(1 + \delta_{f_n}(n)).$$

Таким образом, для алгоритмов класса $NPRH$ количественная мера размерностной чувствительности определяется параметрическим и количественным компонентами функции трудоемкости. Полученные результаты позволяют подойти к определению подклассов алгоритмов в классе NPR с точки зрения количественной меры размерностной чувствительности.

Классификация алгоритмов по информационной чувствительности. В целях определения возможных границ для классификации алгоритмов по информационной чувствительности рассмотрим характерные случаи, когда функция трудоемкости как дискретная случайная величина соответствует различным законам распределения вероятностей [4.20]

— функция трудоемкости обладает максимальным среднеквадратичным отклонением, а размах варьирования стремится к единице — в этом случае

$$\sigma_{f_A}(n) = \sigma_{f_A max} = (f_A^\wedge(n) - f_A^\vee(n))/2, \quad \bar{f}_A(n) = (f_A^\wedge(n) + f_A^\vee(n))/2, \\ f_A^\vee(n) \ll f_A^\wedge(n), \quad \text{или} \quad f_A^\vee(n) = o(f_A^\wedge(n)),$$

что обеспечивает $R_N(n) \rightarrow 1$ при $n \rightarrow \infty$ в соответствии с (4.5.2), тогда $\delta_i(n) \rightarrow 1$ при $n \rightarrow \infty$;

— гистограмма относительных частот функции трудоемкости соответствует равномерному закону распределения, и $R_N(n) \rightarrow 1$ при $n \rightarrow \infty$, тогда, по [4.19]

$$\sigma_{f_A}(n) = (f_A^\wedge(n) - f_A^\vee(n))/\sqrt{12}, \quad \bar{f}_A(n) = (f_A^\wedge(n) + f_A^\vee(n))/2,$$

следовательно, $\delta_i(n) \rightarrow 2/\sqrt{12} \approx 0,57735$ при $n \rightarrow \infty$;

— гистограмма относительных частот функции трудоемкости соответствует нормальному закону распределения, в предположении, что $R_N(n) \rightarrow 1$ при $n \rightarrow \infty$, а размах варьирования составляет $6 \cdot \sigma_{f_A}(n)$, получаем

$$\sigma_{f_A}(n) = (f_A^\wedge(n) - f_A^\vee(n))/6, \quad \bar{f}_A(n) = (f_A^\wedge(n) + f_A^\vee(n))/2,$$

тогда $\delta_i(n) \rightarrow 2/6 \approx 0,33333$ при $n \rightarrow \infty$.

— гистограмма относительных частот функции трудоемкости соответствует нормальному закону распределения, $R_N(n) \rightarrow 1$ при $n \rightarrow \infty$, а среднеквадратичное отклонение составляет 2,5% (1/40) от размаха варьирования, т. е. интервал в $2\sigma_{f_A}(n)$ составляет 5% от этого размаха. В рамках таких допущений мы ожидаем отклонение наблюдаемых значений функции трудоемкости от среднего не более чем на 2,5% с вероятностью $p = 2\Phi(1) \approx 0,6826$ [4.19], в этом случае

$$\sigma_{f_A}(n) = (f_A^\wedge(n) - f_A^\vee(n)) / 40, \quad \bar{f}_A(n) = (f_A^\wedge(n) + f_A^\vee(n)) / 2,$$

тогда $\delta_i(n) \rightarrow 2/40 = 0,05$ при $n \rightarrow \infty$;

— функция трудоемкости обладает нулевым среднеквадратическим отклонением — трудоемкость для любых входов фиксированной длины одинакова, в этом случае и дисперсия, и размах варьирования равны нулю, тогда $\delta_i(n) = 0$.

На основе рассмотренных выше случаев введем следующую классификацию алгоритмов по информационной чувствительности при фиксированной длине входа [4.20].

Класс i1. Алгоритмы, не чувствительные к входным данным по функции трудоемкости: $\delta_i(n) = 0$. Этот тип совпадает с классом количественно-зависимых алгоритмов (класс N).

Класс i2. Алгоритмы, слабо чувствительные к входным данным по функции трудоемкости:

$$0 < \delta_i(n) < 0,05.$$

Класс i3. Алгоритмы, чувствительные к входным данным по функции трудоемкости:

$$0,05 \leq \delta_i(n) < 1/3.$$

Класс i4. Алгоритмы, сильно чувствительные к входным данным по функции трудоемкости:

$$1/3 \leq \delta_i(n) \leq 1,00.$$

Количественная мера информационной чувствительности $\delta_i(n)$ является комплексной, поэтому одинаковую чувствительность могут иметь алгоритмы, обладающие малым размахом варьирования при большой дисперсии, и алгоритмы с большим размахом варьирования, но с малой дисперсией. Такой подход является рациональным, т. к. в каждом из этих случаев разброс (по вероятности) ожидаемых относительных изменений функции трудоемкости будет примерно одинаков. Отметим также, что поскольку функция $\delta_i(n)$ зависит от размерности, то, возможно, один и тот же алгоритм при разных размерностях будет относиться к разным типам по информационной чувствительности.

Возможны два подхода к определению значения меры $\delta_i(n)$ — теоретический и экспериментальный. При теоретическом подходе необходимо получить как функции трудоемкости для лучшего, среднего и худшего случая, так и среднеквадратическое отклонение — $\sigma_{f_A}(n)$.

На основе этих результатов можно получить $\delta_i(n)$ в виде явной функции и, анализируя или вычисляя ее значения для интервала размерностей, характеризующих область применения, определить значения информационной чувствительности и принадлежность алгоритма к одному из введенных типов. Отметим особо, что необходимые для получения $\delta_i(n)$ теоретические зависимости, особенно $\sigma_{f_A}(n)$, должны учитывать особенности формирования входных данных алгоритма в конкретных условиях применения. Экспериментальный подход оперирует методами математической статистики и связан в первую очередь с получением статистического распределения выборки [4.19], на основании которого могут быть численно определены все необходимые для вычисления $\delta_i(n)$ значения. Тогда на основе серии испытаний при фиксированной размерности, используя такие показатели, как выборочное среднее, выборочная дисперсия и выборочный коэффициент вариации, можно прогнозировать ожидаемую трудоемкость на основе значений $\delta_i(n)$. Заметим, что, используя данный подход, мы получаем выборочную информационную чувствительность, так как оперируем с вариационным рядом, полученным по данным выборки. Отметим также, что выборка должна быть репрезентативна относительно множества исходных данных, соответствующих особенностям применения данного алгоритма в данной программной системе, которые и составляют в данном случае генеральную совокупность. При этом единичный эксперимент состоит в определении значения функции трудоемкости для программной реализации алгоритма при конкретном входе.

Классификация алгоритмов по размерностной чувствительности. Рассмотрим вначале несколько общих примеров определения размерностной чувствительности с целью последующей классификации алгоритмов.

Пример 4.1. Функция трудоемкости $f_A^*(n)$ имеет полиномиальный вид. Рассмотрим асимптотически главный компонент $f_A^*(n)$, представляющий собой степенную функцию $f_A^*(n) = cn^k$, $k > 0$, целое, $c > 0$. Тогда в силу определения $\delta_n^*(n)$

$$\delta_n^*(n) = \frac{c(n+1)^k - cn^k}{c \cdot n^k} = \frac{ck \cdot n^{k-1} + O(n^{k-2})}{c \cdot n^k} = \frac{k}{n} + O(n^{-2}).$$

Аналогичный результат для главного порядка $\delta_n^*(n)$ можно получить и для действительного значения k , если заменить приближенно $f_A^*(n+1)$ дифференциалом функции $f_A^*(n)$ при $\Delta n = 1 - f_A^*(n+1) \approx f_A^*(n) + kn^{k-1}\Delta n$. Таким образом, главный порядок размерностной чувствительности для степенных функций трудоемкости имеет вид k/n , где k — показатель степени, не зависит от коэффициента c и гиперболически уменьшается с ростом размерности. В частности, для $f_A^*(n) = cn$, $\delta_n^*(n) = 1/n$.

Пример 4.2. Функция трудоемкости имеет экспоненциальный вид:

$$f_A^*(n) = ce^{\lambda n}, \lambda > 0, c > 0.$$

По формуле (4.5.8)

$$\delta_n^*(n) = \frac{ce^{\lambda(n+1)} - ce^{\lambda n}}{ce^{\lambda n}} = \frac{ce^{\lambda n}e^\lambda - ce^{\lambda n}}{ce^{\lambda n}} = e^\lambda - 1 = \text{const} > 0.$$

Таким образом, размерностная чувствительность для экспоненциальных функций трудоемкости имеет вид $e^\lambda - 1$ вне зависимости от размерности входа алгоритма. На основе полученных результатов предлагается следующая классификация алгоритмов по количественной мере размерностной чувствительности к изменению размерности множества исходных данных по функции трудоемкости [4.20]:

Класс п1. Алгоритмы, мало чувствительные (не более чем линейная функция):

$$0 < \delta_n^* \leq n^{-1}, \quad \forall n > 1.$$

Класс п2. Алгоритмы, слабо чувствительные (не более чем степенная функция со степенью больше единицы):

$$\delta_n^* = k \cdot n^{-1}, \quad \forall n > 1, k > 1.$$

Класс п3. Алгоритмы, чувствительные (более чем степенная, и менее, чем показательная функция):

$$\delta_n^*(n) \succ n^{-1} \quad \text{и} \quad \lim_{n \rightarrow \infty} \delta_n^*(n) = 0,$$

где \succ — обозначение отношения асимптотической иерархии функций [4.15].

Класс п4. Алгоритмы, сильно чувствительные (не менее чем показательная функция):

$$\delta_n^*(n) = \text{const}, \quad \text{или} \quad \lim_{n \rightarrow \infty} \delta_n^*(n) = \infty.$$

Особо отметим, что у одного и того же алгоритма функции трудоемкости для лучшего, среднего и худшего случаев могут иметь не только разную количественную меру, но и обладать различным типом размерностной чувствительности.

§ 6. Классификация вычислительных алгоритмов по дополнительной памяти

Теоретическое исследование ресурсных характеристик вычислительных алгоритмов предполагает введение соответствующих классификаций, отражающих различные ресурсные требования алгоритма и его программной реализации. В этом параграфе мы рассматриваем классификацию вычислительных алгоритмов, в основу которой положен требуемый алгоритмом объем дополнительной памяти в принятой модели вычислений. Мы рассматриваем в данном случае только объем оперативной памяти, т. к. затраты памяти на внешних носителях требуют отдельного рассмотрения и выходят за рамки книги.

Пусть D — конкретный вход алгоритма A , и $|D| = n$, где n — длина или некоторая мера длины входа. Функция объема памяти $V_A(D)$ была введена в параграфе 4.1. Ресурсные требования алгоритма к объему памяти определяются памятью входа, выхода и дополнительной памятью, задействованной алгоритмом в ходе его выполнения. В соответствии с этим будем различать следующие компоненты функции объема памяти.

1. Память входа — $V_I(D)$. При этом по объему памяти входа будем различать две ситуации, связанные со способом получения алгоритмом входных данных:

а) алгоритмы «непотоковые по входу», или алгоритмы со статическим (предопределенным) входом, хранящие входные данные в памяти целиком, в этом случае $V_I(D) = V_I(n)$;

б) алгоритмы «потоковые по входу», или алгоритмы с потоковым входом, когда объекты входа поступают и обрабатываются алгоритмом поэлементно. Отметим, что поэлементное чтение из файла объектов входа в предопределенный массив мы рассматриваем как непотоковый вход. В этом случае для хранения текущего объекта требуется не более чем фиксированное число ячеек оперативной памяти, и $V_I(D) = \Theta(1)$.

2. Память выхода — $V_R(D)$. Аналогично будем различать:

а) алгоритмы со «статическим выходом», хранящие результат в памяти целиком, при этом отметим, что для хранения результата может использоваться как специальный блок памяти, так и память входа. В последнем случае $V_R(D) = 0$, и обычно для таких алгоритмов используют термин «результат по месту», например алгоритмы сортировки по месту [4.2];

б) алгоритмы с «потоковым выходом». Результаты, получаемые алгоритмом, поэлементно, в процессе их получения, выдаются на некоторое устройство вывода, в этом случае нам необходима память не более чем для одного элемента результата, и $V_R(D) = \Theta(1)$.

3. Дополнительная память — $V_t(D)$. Это дополнительная память модели вычислений, задействованная алгоритмом для получения результата.

Поскольку ресурсные компоненты $V_I(D)$ и $V_R(D)$ во многом определяются особенностями задачи, а для различных алгоритмов ее решения, обладающих статическим входом и выходом, фактически совпадают, то именно компонент $V_t(D)$ различает алгоритмы между собой по ресурсным затратам памяти в рамках статического или потокового типа. Очевидно, что для современных компьютеров ограничение решаемых задач по ресурсу оперативной памяти не столь существенно, как ограничения по временной эффективности. Тем не менее, известная дилемма «память–время» приводит к тому, что попытка улучшения временных характеристик требует ощутимых затрат памяти, даже при современных её объемах. Актуальные размерности современных сложных задач также приводят к тому, что ресурс памяти становится значимым в комплексной оценке ресурсной эффективности алгоритмов.

ма. Однако при сравнении алгоритмов, относящихся к статическому или потоковому типу основную роль играет именно дополнительная память, затраты которой обусловлены именно спецификой данного алгоритма. В связи с этим мы предлагаем следующую классификацию алгоритмов, основанную на оценке дополнительной памяти.

I. Класс V0 — алгоритмы с нулевой дополнительной памятью:

$$\forall n > 0, \forall D \in D_n V_t(D) = 0.$$

Алгоритмы этого класса либо вообще не требуют дополнительных ячеек памяти, либо используют ресурсы памяти входа и/или памяти выхода в качестве необходимых дополнительных ячеек по мере обработки элементов входа или по мере заполнения памяти результата. Очевидно, что это наиболее рациональные алгоритмы по критерию емкостной эффективности.

II. Класс VC — алгоритмы с фиксированной дополнительной памятью:

$$\forall n > 0, \forall D \in D_n V_t(D) = \text{const} \neq 0.$$

Алгоритмы класса *VC* используют постоянное, не зависящее от длины и особенностей входа и длины выхода, число дополнительных ячеек оперативной памяти. В реальных алгоритмах это, как правило, ячейки для хранения счетчиков циклов, промежуточных результатов вычислений и указателей на структуры.

III. Класс VL — алгоритмы, дополнительная память которых линейно зависит от длины входа. Введем в рассмотрение функцию дополнительной памяти в худшем случае для всех допустимых входов с мерой n , обозначив ее через $V_t^\wedge(n)$:

$$V_t^\wedge(n) = \max_{D \in D_n} \{V_t(D)\}.$$

В этих обозначениях алгоритм принадлежит к типу с линейной дополнительной памятью, если:

$$V_t^\wedge(n) = \Theta(n),$$

где обозначение Θ есть стандартное обозначение в асимптотическом анализе функций. Содержательно такие затраты означают необходимость хранения копии входного массива, или же алгоритм статического типа требует затрат дополнительной памяти порядка длины входа.

IV. Класс VQ — алгоритмы, дополнительная память которых квадратично зависит от длины входа. Для этого типа алгоритмов объем дополнительной памяти в худшем случае для входов размерности n пропорционален по порядку квадрату меры размерности:

$$V_t^\wedge(n) = \Theta(n^2).$$

Примером может служить стандартный алгоритм умножения длинных целых чисел, заданных побитно массивами длиной n . Алгоритм умно-

жения «в столбик» требует для двух исходных массивов длины n и массива результата длины $2n$ дополнительного массива размерностью $2n \times n = \Theta(n^2)$.

V. Класс VP — алгоритмы, дополнительная память которых имеет полиномиальную надквадратичную зависимость. Это алгоритмы, требующие ресурса дополнительной памяти по порядку большего, чем квадрат меры длины входа, но полиномиально зависящего от этой меры:

$$\exists k > 2 : V_t^\wedge(n) = \Theta(n^k).$$

К этому классу относятся, например, рекурсивные алгоритмы, порождающее дерево рекурсии с оценкой глубины $\Theta(n^k)$, $k > 1$, и требующие хранения в каждой вершине дерева дополнительного массива, имеющего длину порядка $\Theta(n)$.

VI. Класс VE — алгоритмы, дополнительная память которых экспоненциально зависит от длины входа. Формально это тип алгоритмов, требующих, в худшем случае, ресурса дополнительной памяти, по порядку экспоненциально зависящего от меры длины входа:

$$\exists \lambda > 0 : V_t^\wedge(n) = \Theta(e^{\lambda n}).$$

Реально это алгоритмы, использующие дополнительные массивы или более сложные структуры данных, размер которых определяется значениями элементов входа. Характерным примером может служить алгоритм сортировки методом индексов или табличные алгоритмы, реализующие метод динамического программирования. Например, табличный алгоритм, решающий задачу одномерной упаковки для грузов 100 типов в объем 10000 будет использовать два массива, содержащих 10^6 элементов каждый [4.21].

С теоретической точки зрения определенный интерес может представлять также и другой подход, в основе которого лежит нормированная мера, равная отношению ресурса дополнительной памяти к общей:

$$\delta_V(D) = \frac{V_t(D)}{V_A(D)}, \quad \forall n \forall D \in D_n \quad 0 \leq \delta_V(D) \leq 1,$$

но предлагаемая типизация по асимптотической оценке абсолютных затрат дополнительной памяти представляется более целесообразной с точки зрения разработчиков алгоритмического обеспечения программных средств и систем.

Заключение по главе 4. В заключение мы хотим кратко остановиться на проблеме выбора рационального алгоритма. Рациональный выбор того или иного вычислительного алгоритма для решения некоторой задачи требует проведения исследования претендующих алгоритмов не только в целях получения асимптотических оценок функции трудоемкости, т. е. оценок сложности алгоритмов. Проблема в том, что для целого ряда вычислительных задач асимптотически более эффективный алгоритм имеет значительный коэффициент при глав-

ном порядке функции трудоемкости, и тем самым не всегда является эффективным на всем заданном диапазоне размерности входа. Реальные значения этого диапазона известны разработчикам программной системы, т. е. определяются особенностями использования данного алгоритма. Для выбора рационального алгоритма по критерию ресурсной эффективности необходимо детальное исследование претендующих алгоритмов в границах реального диапазона применения. При этом вполне вероятно, что на разных сегментах этого диапазона разные алгоритмы могут быть выбраны как наиболее рациональные. Таким образом, возможно адаптивное по размерности входа управление выбором алгоритма решения данной задачи в проектируемой программной системе. Решение по рациональному выбору лежит в области сравнительного анализа вычислительных алгоритмов по ресурсным функциям и связано с выполнением следующих этапов:

- детальный анализ ресурсной эффективности сравниваемых алгоритмов, т. е. получение функции трудоемкости и функции объема памяти в явном виде;
- сравнительный анализ ресурсных функций с целью выбора рационального алгоритма при реальных ограничениях на длину входа.

Более подробную информацию по этой проблематике заинтересованный читатель найдет в [4.3, 4.4], а пример анализа и рекомендации по рациональному применению алгоритмов — в [4.21].

Поскольку данная глава посвящена различным классификациям вычислительных алгоритмов, мы не приводим задач и упражнений, но проиллюстрируем некоторые классификации на конкретных алгоритмах в главе 7.

Список литературы к главе 4

- 4.1. Ахо А., Хопкрофт Дж., Ульман Дж. Структуры данных и алгоритмы: Пер. с англ.: — М.: Изд. дом «Вильямс», 2001. — 384 с.
- 4.2. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. — М.: МЦНМО, 1999. — 960 с.
- 4.3. Ульянов М.В. Классификация и методы сравнительного анализа вычислительных алгоритмов. Научное издание. — М.: Изд-во физико-математической литературы, 2004. — 212 с.
- 4.4. Ульянов М.В. Дополнение к книге Дж. Макконелла Основы современных алгоритмов. — М.: Изд-во Техносфера, 2004. С. 303–366.
- 4.5. Успенский В.А. Машина Поста. — М.: Наука, 1979. — 96 с.
- 4.6. Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов: Пер. с англ.: — М.: Мир, 1979. — 546 с.
- 4.7. Липаев В.В. Методы обеспечения качества крупномасштабных программных средств. — М.: СИНТЕГ, 2003. — 520 с. (Серия «Управление качеством»).

- 4.8. Липаев В.В. Обеспечение качества программных средств. — М.: СИНТЕГ. 2001. — 384 с.
- 4.9. Вирт Н. Алгоритмы и структуры данных: Пер. с англ. — 2-е изд., испр. — СПб.: Невский диалект, 2001. — 352 с.
- 4.10. Ульянов М.В., Головешкин В.А. Информационная чувствительность функции трудоемкости алгоритмов к входным данным // Новые информационные технологии: Сборник трудов VII Всероссийской научно-технической конференции (Москва, 24-25 марта 2004). / Под общ. ред. А.П. Хныкина — М.: МГАПИ, 2004. С. 19–26.
- 4.11. Гасанов Э.Э., Кудрявцев В.Б. Теория хранения и поиска информации. — М.: Физматлит, 2002. — 288 с.
- 4.12. Хопкрофт Дж., Мотовані Р., Ульман Дж. Введение в теорию автоматов, языков и вычислений, 2-е изд.: Пер. с англ. — М.: Изд. дом «Вильямс», 2002. — 528 с.
- 4.13. Гудман С., Хидетниemi С. Введение в разработку и анализ алгоритмов. — М.: Мир, 1981. — 368 с.
- 4.14. Головешкин В.А., Ульянов М.В. Метод классификации вычислительных алгоритмов по сложности на основе угловой меры асимптотического роста функций // Вычислительные технологии. 2006. Т. 11, № 1. С. 52–62.
- 4.15. Грэхем Р., Кнут Д., Паташник О. Конкретная математика. Основание информатики: Пер. с англ. — М.: Мир, 1998. — 703 с.
- 4.16. Чмора А.Л. Современная прикладная криптография. — М.: Гелиос АРВ, 2001. — 256 с.
- 4.17. Бахвалов Н.С., Жидков Н.Н., Кобельков Г.М. Численные методы. — М.: Лаборатория Базовых Знаний, 2001 г. — 632 с.
- 4.18. Кнут Д.Э. Искусство программирования, том 1. Основные алгоритмы, 3-е изд.: Пер. с англ. — М.: Изд. дом «Вильямс», 2002. — 720 с.
- 4.19. Гмурман В.Е. Теория вероятностей и математическая статистика: Учеб. пособие для вузов —9-е изд., стер. — М.: Высш. шк., 2003. — 479 с.
- 4.20. Ульянов М.В. Исследование и классификация вычислительных алгоритмов на основе чувствительности функции трудоемкости // Системы управления и информационные технологии. 2004. № 4 (16). С. 97–104.
- 4.21. Ульянов М.В., Гурин Ф.Е., Исаков А.С., Бударагин В.Е. Сравнительный анализ табличного и рекурсивного алгоритмов точного решения задачи одномерной упаковки // Exponenta Pro Математика в приложениях. 2004. №2(6). С. 64–70.

Глава 5

СПЕЦИАЛЬНЫЕ ГЛАВЫ ТЕОРИИ РЕКУРСИИ

Введение. Эта глава посвящена нетрадиционным методам решения рекуррентных соотношений и некоторым специальным вопросам, возникающим при их исследовании. Мы приводим основную теорему о рекуррентных соотношениях и примеры ее использования, показываем, как с помощью аппарата производящих функций и свертки последовательностей можно получать решения ряда рекуррентных соотношений. Рассматриваются также методы исчисления и оценки конечных сумм, часто применяемые в процессе анализа и исследования алгоритмов, в частности рекурсивных. Мы даем сведения о некоторых специальных функциях, возникающих при исследовании рекурсивных алгоритмов, примеры применения которых читатель найдет в главе 7. Рекурсивный характер некоторых алгоритмов приводит также к необходимости исследования комбинаторных соотношений, которые мы также кратко рассматриваем в данной главе.

§ 1. Основная теорема о рекуррентных соотношениях и некоторые особые случаи

Следующая теорема, доказанная в 1980 г. Дж. Бентли, Д. Хакен и Дж. Саксом [5.1] является достаточно мощным средством асимптотической оценки функциональных рекуррентных соотношений определенного вида. Такого рода рекурсивно заданные функции получаются при оценке вычислительной сложности рекурсивных алгоритмов, основанных на методе декомпозиции задачи.

Теорема (J.L. Bentley, D. Haken, J.B. Saxe, 1980).

Пусть $a \geq 1$ и $b > 1$ — константы, $f(n)$ — известная функция, $T(n)$ определено при неотрицательных значениях n формулой

$$T(n) = aT\left(\left[\frac{n}{b}\right]\right) + f(n),$$

тогда:

- 1) если $f(n) = O(n^{\log_b a - \varepsilon})$ для некоторого $\varepsilon > 0$, то $T(n) = \Theta(n^{\log_b a})$;
- 2) если $f(n) = \Theta(n^{\log_b a})$, то $T(n) = \Theta(n^{\log_b a} \log_b n)$;
- 3) если найдутся такие $c > 0$ и $\varepsilon > 0$, что при достаточно больших n выполнено $f(n) > cn^{\log_b a + \varepsilon}$ и найдется положительная константа $d < 1$

такая, что при достаточно больших n выполнено $a f\left(\frac{n}{b}\right) \leq d f(n)$, то $T(n) = \Theta(f(n))$.

Рассмотрим примеры асимптотической оценки функциональных рекуррентных соотношений с использованием данной теоремы.

Пример 5.1. Получить асимптотическую оценку функции:

$$T(n) = 8T\left(\left[\frac{n}{4}\right]\right) + n.$$

В этом случае $a = 8$, $b = 4$, $f(n) = n$, а $n^{\log_b a} = n^{\log_4 8} = \Theta\left(n^{\frac{3}{2}}\right)$. Поскольку $f(n) = O\left(n^{\frac{3}{2}-\varepsilon}\right)$ для $\varepsilon = \frac{1}{2}$, то, применяя первое утверждение теоремы, делаем вывод, что $T(n) = \Theta\left(n^{\frac{3}{2}}\right)$.

Пример 5.2. Получить асимптотическую оценку функции

$$T(n) = T\left(\left[\frac{3n}{4}\right]\right) + 5.$$

Здесь $a = 1$, $b = 4/3$, $f(n) = 5$, а $n^{\log_b a} = n^{\log_{4/3} 1} = n^0 = \Theta(1)$. Воспользуемся вторым утверждением теоремы. Поскольку $f(n) = 5 = \Theta(n^{\log_b a}) = \Theta(1)$, то получаем $T(n) = \Theta\left(\log_{4/3} n\right)$.

Пример 5.3. Получить асимптотическую оценку функции

$$T(n) = 2T\left(\left[\frac{n}{4}\right]\right) + n \log_4 n.$$

в этом случае $a = 2$, $b = 4$, $f(n) = n \log_4 n$, а $n^{\log_b a} = n^{\log_4 2} = n^{0.5} = \Theta(\sqrt{n})$ и, например, при больших n и $\varepsilon = 0,5$ $f(n) = n \log_4 n \geq n^{0.5+\varepsilon}$. Остается проверить последнее условие третьего утверждения теоремы. Для достаточно больших значений n имеем

$$2f\left(\frac{n}{4}\right) = 2 \frac{n}{4} \log_4 \frac{n}{4} \leq \frac{1}{2} n \log_4 n = df(n), \quad \text{где } d = \frac{1}{2} < 1,$$

тогда $T(n) = \Theta(n \log_4 n)$.

Пример 5.4. Вычислить a^n , где n — целое неотрицательное число. Последовательность a^n можно рассматривать как рекурсивно заданную последовательность вида

$$\begin{cases} T(0) = 1, \\ T(n) = aT(n-1) \quad \text{при } n \geq 1. \end{cases}$$

В случае подобного задания количество операций для вычисления a^n имеет порядок n . Но рекурсивная последовательность для a^n может быть задана и другим способом, а именно:

$$\begin{cases} T(0) = 1, \quad T(1) = a; \\ T(n) = T\left(\left[\frac{n}{2}\right]\right) T\left(\left[\frac{n}{2}\right]\right), \quad n \geq 2. \end{cases}$$

Напомним, что $\left\lfloor \frac{n}{2} \right\rfloor$ и $\left\lceil \frac{n}{2} \right\rceil$ обозначают «пол» и «потолок» аргумента соответственно. Зададимся вопросом: скольких операций требует такой алгоритм? Пусть $F(n)$ обозначает количество операций, требуемых для вычисления $T(n)$. Нетрудно заметить, что количество операций, требуемых для вычисления $T\left(\left\lceil \frac{n}{2} \right\rceil\right)$, на константу отличается от количества операций, необходимых для вычисления $T\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$. Мы не будем конкретизировать значения этой константы, а обозначим ее через c . Следовательно, для оценки $F(n)$ мы имеем соотношение

$$F(n) = F\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + c.$$

Это соотношение соответствует второму случаю основной теоремы. Применяя ее, получаем

$$F(n) = \Theta(\log_2 n).$$

Пример 5.5. В качестве еще одного примера рассмотрим известный метод половинного деления отрезка, применяемый для решения уравнений. Требуется определить с заданной точностью h корень уравнения $f(x) = 0$, где $f(x)$ — непрерывная функция.

Предполагаем, что из каких-то соображений известно, что корень лежит на отрезке $[a; b]$. Кроме того, известно, что на этом отрезке лежит только один корень данного уравнения. Следовательно, на концах этого отрезка значения функции имеют разные знаки. Для определенности будем считать, что $f(a) < 0$, а $f(b) > 0$. Мы будем считать также, что корень определен с нужной степенью точности h , если мы определим такой отрезок $[c; d]$, что $f(c) < 0$, $f(d) > 0$ и при этом $d - c \leq h$. Рассмотрим следующий алгоритм.

Обозначим $a_1 = a$, $b_1 = b$. Поскольку $f(a_1) < 0$, $f(b_1) > 0$, и непрерывная на отрезке функция принимает все промежуточные значения, то мы можем утверждать, что неизвестный корень x уравнения $f(x) = 0$ лежит на отрезке $[a_1; b_1]$. Идея алгоритма собственно и состоит в построении последовательности отрезков $[a_i; b_i]$, такой что $f(a_i) < 0$, $f(b_i) > 0$, и при этом $b_{i+1} - a_{i+1} < b_i - a_i$, т. е. на каждом шаге отрезок, на котором лежит корень уравнения, уменьшается. Возьмем значение c в середине отрезка $c = \frac{a_i + b_i}{2}$, и вычислим значение $f(c)$. Возможны два случая $f(c) < 0$ и $f(c) > 0$. (Случай $f(c) = 0$ ввиду его слишком малой вероятности для реальных уравнений не рассматриваем). В случае $f(c) < 0$ полагаем $a_{i+1} = c$, $b_{i+1} = b_i$. В противном случае, при $f(c) > 0$ полагаем $a_{i+1} = a_i$, $b_{i+1} = c$, и вычисления повторяются. Условием останова является $b_{i+1} - a_{i+1} < h$.

Рекурсивный характер данного алгоритма очевиден, но может возникнуть вопрос, что в этом алгоритме играет роль параметра n . Параметр n , несмотря на то, что он явно не проявляется, тем не менее,

присутствует в данном алгоритме. Роль параметра n играет то, сколько раз значение требуемой точности h «укладывается» на отрезке:

$$n = \left[\frac{b-a}{h} \right].$$

Для оценки количества необходимых операций $F(n)$ мы снова получаем соотношение вида

$$F(n) = F\left(\left[\frac{n}{2}\right]\right) + c,$$

где значение c определяется количеством операций, требуемых для вычисления функции и соответствующих операции сравнения и присваивания новых значений концам отрезка. Для значения $F(n)$ с использованием основной теоремы получаем следующую асимптотическую оценку

$$F(n) = \Theta(\log_2 n).$$

В заключение данного раздела приведем метод оценки членов рекурсивной последовательности, которая определяется рекуррентным соотношением вида [5.2]

$$a_{n+1} = a_n^2 + g_n, \quad (5.1.1)$$

где g_n — медленно возрастающая функция, которая может зависеть от предыдущих членов последовательности. Более строгие требования к ней будут сформулированы позднее. Мы заранее будем предполагать, что все члены рекурсивной последовательности положительны. Данное соотношение является нелинейным. Логарифмируя (5.1.1), мы получаем соотношение (5.1.2), которое является «почти линейным». Смысл понятия «почти линейное» прояснится позднее. Итак:

$$b_{n+1} = 2b_n + \beta_n, \quad (5.1.2)$$

где

$$b_n = \ln a_n, \quad \beta_n = \ln \left(1 + \frac{g_n}{a_n^2}\right)$$

Согласно (5.1.2) и (5.1.1) получаем

$$b_n = 2^n \left(b_0 + \frac{\beta_0}{2} + \frac{\beta_1}{2^2} + \dots + \frac{\beta_{n-1}}{2^n} \right).$$

Введем величину

$$B_n = 2^n b_0 + \sum_{i=0}^{\infty} 2^{n-i-1} \beta_i,$$

предполагая, что соответствующий ряд сходится, определим остаток

$$r_n = B_n - b_n = \sum_{i=n}^{\infty} 2^{n-i-1} \beta_i.$$

Подобная операция имеет смысл только тогда, когда полученный ряд сходится достаточно быстро. Второе допущение, которое раскрывает смысл понятия, что g_n — медленно возрастающая функция, состоит в предположении, что, начиная с некоторого номера n_0 , последовательность $|\beta_n|$ не возрастает, т. е. $|\beta_{n+1}| \leq |\beta_n|$, при $n > n_0$. Тогда ряд для B_n сходится, а остаток r_n не превосходит $|\beta_n|$, следовательно, решение исходного рекуррентного соотношения может быть представлено в виде

$$a_n = e^{B_n - r_n} = k^{2^n} e^{-r_n}, \quad (5.1.3)$$

где

$$k = a_0 \exp \left(\sum_{i=0}^{\infty} 2^{-i-1} \beta_i \right). \quad (5.1.4)$$

Полученное выражение (5.1.3) не является в прямом смысле решением рекуррентного соотношения. Оно лишь показывает, что существует константа k , которая характеризует полученное асимптотическое поведение членов последовательности. Определение точного значения этой константы является самостоятельной задачей для каждого конкретного соотношения, и задачей далеко не простой.

§ 2. Производящие функции

Довольно часто объектом исследования является некоторая счетная последовательность чисел

$$(a_0, a_1, \dots, a_n, \dots).$$

Мы будем говорить о бесконечных последовательностях чисел. При этом мы не ограничиваем общности, поскольку любую конечную последовательность можно рассматривать как бесконечную, считая, что все ее члены, начиная с некоторого номера, равны нулю. Использование производящих функций ставит своей целью рассматривать последовательность не как набор чисел, а как единое целое. Указанной выше последовательности ставится в соответствие степенной ряд

$$\sum_{k=0}^{\infty} a_k z^k,$$

где z рассматривается как комплексное число. Сумма этого ряда, которую мы будем обозначать через $G(z)$, и называется производящей функцией для данной последовательности. Таким образом, производящая функция $G(z)$ для последовательности $(a_0, a_1, \dots, a_n, \dots)$ равна

$$G(z) = \sum_{k=0}^{\infty} a_k z^k.$$

Одно преимущество производящей функции уже видно. Вместо бесконечного набора чисел мы будем иметь дело с одной функцией. Конечно, это достоинство будет действительно таковым, если сама функция будет иметь не очень сложный вид. Но возникает резонный вопрос, — каким образом, зная функцию $G(z)$, восстановить последовательность $(a_0, a_1, \dots, a_n, \dots)$? Без ответа на такой вопрос все дальнейшие рассуждения теряют всякий смысл. Мы не будем вдаваться в тонкости теории функций комплексного переменного. При не слишком жёстких ограничениях степенной ряд для функции $G(z)$ абсолютно сходится в некотором круге на комплексной плоскости с центром в точке $z = 0$, а функция $G(z)$ является аналитической в этом круге. Тогда члены последовательности $(a_0, a_1, \dots, a_n, \dots)$ являются коэффициентами ряда Тейлора для этой функции в точке $z = 0$. Для вычисления коэффициентов ряда Тейлора известна формула

$$a_n = \frac{1}{n!} D^n(G(z)) \Big|_{z=0},$$

где $D^n(G(z))|_{z=0}$ есть производная порядка n функции $G(z)$ в точке $z = 0$.

Сейчас мы установим связь между некоторыми конкретными последовательностями и соответствующими производящими функциями. Пусть

$$G(z) = (az + b)^n.$$

Мы надеемся, что читатель помнит простейшие правила вычисления производных. Получаем:

$$D((az + b)^n) = n(az + b)^{n-1}a;$$

$$D^2((az + b)^n) = n(n - 1)(az + b)^{n-2}a^2;$$

.....

$$D^k((az + b)^n) = n(n - 1) \dots (n - k + 1)(az + b)^{n-k}a^k, \quad \text{при } k \leq n;$$

$$D^k((az + b)^n) = 0, \quad \text{при } k > n.$$

Так как для членов соответствующей последовательности имеет место формула

$$a_k = \frac{1}{k!} D^k(G(z)) \Big|_{z=0},$$

то имеем $a_0 = b^n$, $a_1 = nab^{n-1}$, $a_2 = \frac{1}{2!}n(n - 1)a^2b^{n-2}$, и в общем случае

$$a_k = \frac{1}{k!}n(n - 1) \dots (n - k + 1)a^k b^{n-k}, \quad k \leq n,$$

и все $a_k = 0$ при $k > n$. Заметим, что

$$\frac{1}{k!} n(n-1) \dots (n-k+1) = \frac{1}{k!} \cdot \frac{n(n-1) \dots 2 \cdot 1}{(n-k)(n-k-1) \dots 2 \cdot 1} = \\ = \frac{n!}{k!(n-k)!} = C_n^k,$$

где C_n^k — биномиальные коэффициенты или, что то же самое, число сочетаний без повторений из n по k . Таким образом, данной производящей функции соответствует последовательность $(a_0, a_1, \dots, a_n, \dots)$, где $a_k = C_n^k a^k b^{n-k}$ при $k \leq n$; $a_k = 0$ при $k > n$. Рассмотрим частные случаи этой формулы.

Пусть $G(z) = (z - b)^n$, тогда $a_k = C_n^k (-1)^{n-k} b^{n-k}$ при $k \leq n$; $a_k = 0$ при $k > n$.

И очень важный частный случай, когда $G(z) = (1 + z)^n$. В этом случае $a_k = C_n^k$ при $k \leq n$; $a_k = 0$ при $k > n$. То есть функция $G(z) = (1 + z)^n$ является производящей функцией для числа сочетаний без повторений из n по k .

Теперь рассмотрим функцию

$$G(z) = \frac{1}{(az + b)^{m+1}}, \quad \text{где } m \geq 0,$$

$$D^k(g(z)) = D^k \left(\frac{1}{(az + b)^{m+1}} \right) = (-1)^k \frac{(m+1)(m+2) \dots (m+k)}{(az + b)^{m+1+k}} a^k,$$

тогда

$$a_k = \frac{1}{k!} (-1)^k \frac{(m+1)(m+2) \dots (m+k)}{b^{m+1+k}} a^k = \\ = (-1)^k \frac{a^k}{b^{m+1+k}} \cdot \frac{1}{k!} \cdot \frac{1 \cdot 2 \cdot \dots \cdot (m+k)}{1 \cdot 2 \cdot \dots \cdot m}$$

или в окончательном виде

$$a_k = (-1)^k \frac{a^k}{b^{m+1+k}} C(m+k, m).$$

Рассмотрим частные случаи полученной формулы. Пусть

$$G(z) = \frac{1}{(z - b)^{m+1}},$$

тогда

$$a_k = (-1)^{m+1} \frac{1}{b^{m+1+k}} C_{m+k}^m. \quad (5.2.1)$$

Пусть $G(z) = \frac{1}{(1+z)^{m+1}}$, тогда $a_k = (-1)^k \cdot C_{m+k}^m$.

Пусть $G(z) = \frac{1}{(1-z)^m}$, тогда $a_k = C_{m+k}^m$.

Отдельно рассмотрим случай для $m = 0$, при этом:

$$G(z) = \frac{1}{1+z},$$

тогда $a_k = (-1)^k$, т. е. последовательность состоит из единиц с чередующимися знаками. Наконец, для функции

$$G(z) = \frac{1}{1-z}$$

имеем $a_k = 1$, т. е. все члены данной последовательности равны единице. Найдем соответствующую последовательность для производящей функции вида

$$G(z) = \frac{1 - z^{n+1}}{1 - z}.$$

Мы надеемся, что читатель еще не забыл формулу суммы геометрической прогрессии, и он вспомнит, что данная функция есть не что иное, как сумма вида

$$G(z) = 1 + z + z^2 + \dots + z^n,$$

поэтому последовательность, соответствующая данной производящей функции, имеет вид: $a_k = 1$ при $k \leq n$; $a_k = 0$ при $k > n$. Этот вывод можно было получить другим способом, если обратить внимание на одно очень важное свойство производящих функций.

Пусть производящей функции $G(z)$ соответствует последовательность $(a_0, a_1, \dots, a_n, \dots)$. Тогда функции $G_1(z) = z^n G(z)$ ($n > 0$) соответствует последовательность $(b_0, b_1, \dots, b_n, \dots)$ и между этими последовательностями существует следующая связь: $b_k = 0$ при $k < n$; $b_k = a_{k-n}$ при $k \geq n$. То есть при умножении производящей функции на z^n соответствующая последовательность сдвигается на n шагов вправо. С учетом указанного замечания полученный результат для

$$G(z) = \frac{1 - z^{n+1}}{1 - z}$$

становится очевидным.

Отметим также тот факт, что производящие функции обладают свойством линейности по отношению к порождающим их последовательностям. Линейность понимается в следующем плане. Пусть имеются две последовательности $(a_0, a_1, \dots, a_n, \dots)$ и $(b_0, b_1, \dots, b_n, \dots)$, производящие функции для которых соответственно равны $G_1(z)$ и $G_2(z)$. Тогда для последовательности

$$(ca_0 + db_0, ca_1 + db_1, \dots, ca_n + db_n, \dots)$$

производящая функция $G(z)$ равна

$$G(z) = cG_1(z) + dG_2(z).$$

В дальнейшем мы рассмотрим методы восстановления последовательности, когда производящая функция является рациональной дробью.

Напомним некоторые факты, касающиеся многочленов в комплексной области. Многочленом степени n (будем обозначать его $P_n(z)$) называется выражение вида

$$P_n(z) = a_n z^n + a_{n-1} z^{n-1} + \dots + a_2 z^2 + a_1 z + a_0,$$

где a_0, a_1, \dots, a_n в общем случае являются комплексными числами, $a_n \neq 0$. Корнем многочлена называется число c такое, что $P_n(c) = 0$. Если число c является корнем многочлена, то многочлен может быть представлен в виде $P_n(z) = (z - c)P_{n-1}(z)$, где $P_{n-1}(z)$ — многочлен степени $(n - 1)$. Число c является корнем многочлена кратности r , если многочлен может быть представлен в виде $P_n(z) = (z - c)^r P_{n-r}(z)$, где $P_{n-r}(c) \neq 0$. Многочлен степени n имеет ровно n корней с учетом их кратности. Последний факт означает, что если многочлен

$$P_n(z) = a_n z^n + a_{n-1} z^{n-1} + \dots + a_2 z^2 + a_1 z + a_0$$

имеет k различных корней c_1, c_2, \dots, c_k кратности r_1, r_2, \dots, r_k соответственно, при этом $r_1 + r_2 + \dots + r_k = n$, то его можно представить в виде

$$P_n(z) = a_n (z - c_1)^{r_1} (z - c_2)^{r_2} \dots (z - c_k)^{r_k}.$$

Напомним, что рациональной дробью $R(z)$ называется выражение вида

$$R(z) = \frac{P_n(z)}{P_m(z)},$$

где $P_n(z)$ и $P_m(z)$ — многочлены соответствующих степеней. Рациональная дробь называется правильной дробью, если степень многочлена в знаменателе строго больше степени многочлена в числителе. В противном случае дробь называется неправильной. Если $R(z)$ является неправильной дробью, то она может быть представлена в виде $R(z) = P_{n-m}(z) + R_1(z)$, где $P_{n-m}(z)$ — многочлен степени $n - m$, а $R_1(z)$ — правильная рациональная дробь.

Элементарной (простейшей) рациональной дробью называется выражение вида

$$\frac{A}{(z - a)^m}.$$

Любая правильная рациональная дробь может быть разложена в сумму элементарных дробей. Этот факт позволяет представить схему нахождения соответствующих последовательностей в случае, когда производящая функция является рациональной дробью. На первом этапе для неправильной дроби ее представляют в виде суммы многочлена и правильной дроби. Проблема нахождения соответствующей последовательности для многочлена решается просто. Правильная дробь раскладывается в сумму элементарных дробей. Способ определения соответствующей последовательности для элементарной дроби мы изложили ранее. Схема представления правильной дроби в виде суммы

элементарных дробей выглядит следующим образом. Пусть имеется правильная дробь вида

$$R(z) = \frac{P_n(z)}{P_m(z)}, \quad (m > n),$$

при этом многочлен в знаменателе

$$P_m(z) = b_m z^m + b_{m-1} z^{m-1} + \dots + b_2 z^2 + b_1 z + b_0$$

имеет k различных корней c_1, c_2, \dots, c_k кратности r_1, r_2, \dots, r_k соответственно и, значит, его можно представить в виде

$$P_m(z) = b_m (z - c_1)^{r_1} (z - c_2)^{r_2} \dots (z - c_k)^{r_k}.$$

Тогда рациональная дробь $R(z)$ может быть представлена в следующем виде

$$\begin{aligned} R(z) &= \frac{P_n(z)}{b_m (z - c_1)^{r_1} (z - c_2)^{r_2} \dots (z - c_k)^{r_k}} = \\ &= \frac{A_{1,r_1}}{(z - c_1)^{r_1}} + \frac{A_{1,r_1-1}}{(z - c_1)^{r_1-1}} + \dots + \frac{A_{1,1}}{z - c_1} + \\ &+ \frac{A_{2,r_2}}{(z - c_2)^{r_2}} + \frac{A_{2,r_2-1}}{(z - c_2)^{r_2-1}} + \dots + \frac{A_{2,1}}{z - c_2} + \dots + \\ &+ \frac{A_{k,r_k}}{(z - c_k)^{r_k}} + \frac{A_{k,r_k-1}}{(z - c_k)^{r_k-1}} + \dots + \frac{A_{k,1}}{z - c_k}. \end{aligned}$$

Способ определения коэффициентов мы покажем на примерах.

Пример 5.6. Найти последовательность $(a_0, a_1, \dots, a_n, \dots)$, для которой производящая функция равна

$$G(z) = \frac{4z^2 - z - 9}{(z + 2)(z + 1)(z - 1)}.$$

Выражение для производящей функции является правильной рациональной дробью. Представление этой дроби в виде суммы простейших дробей ищем в виде

$$\frac{4z^2 - z - 9}{(z + 2)(z + 1)(z - 1)} = \frac{A}{z + 2} + \frac{B}{z + 1} + \frac{C}{z - 1}.$$

Осталось определить неизвестные пока коэффициенты A, B, C . Приведем правую часть к общему знаменателю, который равен знаменателю в левой части выражения и приравняем числители в правой и левой частях. Получим

$$4z^2 - z - 9 = A(z + 1)(z - 1) + B(z + 2)(z - 1) + C(z + 2)(z + 1). \quad (5.2.2)$$

В соотношении (5.2.2) приравниваем коэффициенты при одинаковых степенях z и получаем систему уравнений для определения A, B, C :

$$\begin{cases} A + B + C = 4; \\ B + 3C = -1; \\ -A - 2B + 2C = -9. \end{cases}$$

Решая эту систему уравнений, находим коэффициенты. Отметим, что в данном случае нет необходимости решать систему уравнений, поскольку неизвестные коэффициенты можно определить более простым путем. Равенство (5.2.2) рассматриваем как равенство многочленов, верное для любых значений z . Подставим в левую и правую части этого равенства $z = -2$. Получим $9 = 3A$, и $A = 3$. Подставляя $z = -1$, имеем $-4 = -2B$, откуда $B = 2$. Подстановка $z = 1$ дает $-6 = 6C$, и $C = -1$. Следовательно,

$$\frac{4z^2 - z - 9}{(z+2)(z+1)(z-1)} = \frac{3}{z+2} + \frac{2}{z+1} - \frac{1}{z-1}.$$

Для элементарных дробей $\frac{1}{z+2}, \frac{1}{z+1}, \frac{1}{z-1}$, согласно (5.2.1), последовательности $(b_0, b_1, \dots, b_n, \dots)$, имеют вид

$$b_n = \frac{(-1)^n}{2^{n+1}}; \quad b_n = (-1)^n \quad \text{и} \quad b_n = -1$$

соответственно. Надеемся, что читатель без особых усилий доведет решение задачи до конца.

Пример 5.7. Найти последовательность $(a_0, a_1, \dots, a_n, \dots)$, для которой производящая функция равна

$$G(z) = \frac{4z - 8}{(z-3)(z-1)^2}.$$

Поскольку дробь правильная, то представим ее в виде суммы элементарных дробей вида

$$\frac{4z - 8}{(z-3)(z-1)^2} = \frac{A}{z-3} + \frac{B}{(z-1)^2} + \frac{C}{z-1}.$$

Приводя к общему знаменателю и приравнивая числители в левой и правой частях, получаем

$$4z - 8 = A(z-1)^2 + B(z-3) + C(z-3)(z-1). \quad (5.2.3)$$

Приравниваем коэффициенты при одинаковых степенях z и получаем систему уравнений для определения A, B, C :

$$\begin{cases} A + C = 0; \\ -2A + B - 4C = 4; \\ A - 3B + 3C = -8. \end{cases}$$

Решая эту систему уравнений, находим коэффициенты. Отметим, что в данном случае нет необходимости полностью решать систему уравнений, поскольку часть неизвестных коэффициентов можно определить более простым путем. Равенство (5.2.3) рассматриваем как равенство многочленов, верное для любых значений z . Подставим в левую и правую части этого равенства $z = 3$. Получим $4 = 4A$, и $A = 1$. Подставим в левую и правую части этого равенства $z = 1$, получим $-4 = -2B$, что дает $B = 2$. Тогда из первого уравнения находим $C = -1$, и

$$\frac{4z - 8}{(z - 3)(z - 1)^2} = \frac{1}{z - 3} + \frac{2}{(z - 1)^2} - \frac{1}{z - 1}.$$

Для элементарных дробей $\frac{1}{z - 3}$, $\frac{1}{(z - 1)^2}$, $\frac{1}{z - 1}$, согласно (5.2.1), последовательности $(b_0, b_1, \dots, b_n, \dots)$ имеют вид:

$$b_n = -\frac{1}{3^{n+1}}, \quad b_n = C_{n+1}^1 = n + 1, \quad b_n = -1$$

соответственно. Получение окончательного ответа мы предоставим читателю.

Пример 5.8. Найти последовательность $(a_0, a_1, \dots, a_n, \dots)$, для которой производящая функция равна

$$G(z) = \frac{4}{z^2 - 2z + 2}.$$

Представим $G(z)$ в виде

$$G(z) = \frac{4}{(z - (1+i))(z - (1-i))}.$$

Разложение на элементарные дроби ищем в виде

$$\frac{4}{(z - (1+i))(z - (1-i))} = \frac{A}{z - (1+i)} + \frac{B}{z - (1-i)}.$$

Аналогичным образом, определяя неизвестные коэффициенты, получаем $A = -2i$, $B = 2i$, тогда

$$\frac{4}{(z - (1+i))(z - (1-i))} = \frac{-2i}{z - (1+i)} + \frac{2i}{z - (1-i)}.$$

Для элементарных дробей

$$\frac{1}{z - (1+i)}, \quad \frac{1}{z - (1-i)},$$

согласно (5.2.1), последовательности $(b_0, b_1, \dots, b_n, \dots)$ и $(c_0, c_1, \dots, c_n, \dots)$ имеют вид:

$$b_n = -\frac{1}{(1+i)^{n+1}}, \quad c_n = -\frac{1}{(1-i)^{n+1}}.$$

Тогда

$$\begin{aligned} a_n &= -2ib_n + 2ic_n = 2i \left(\frac{1}{(1+i)^{n+1}} - \frac{1}{(1-i)^{n+1}} \right) = \\ &= \frac{i}{2^n} ((1-i)^{n+1} - (1+i)^{n+1}). \end{aligned}$$

Заметим, что

$$1+i = \sqrt{2} \left(\cos \frac{\pi}{4} + i \sin \frac{\pi}{4} \right), \quad 1-i = \sqrt{2} \left(\cos \left(-\frac{\pi}{4} \right) + i \sin \left(-\frac{\pi}{4} \right) \right).$$

Тогда

$$\begin{aligned} (1+i)^{n+1} &= 2^{\frac{n+1}{2}} \left(\cos(n+1) \frac{\pi}{4} + i \sin(n+1) \frac{\pi}{4} \right), \\ (1-i)^{n+1} &= 2^{\frac{n+1}{2}} \left(\cos(n+1) \left(-\frac{\pi}{4} \right) + i \sin(n+1) \left(-\frac{\pi}{4} \right) \right). \end{aligned}$$

следовательно,

$$a_n = \frac{i}{2^n} 2^{\frac{n+1}{2}} \left(-2i \sin(n+1) \left(\frac{\pi}{4} \right) \right) = 2^{\frac{3-n}{2}} \sin(n+1) \left(\frac{\pi}{4} \right).$$

Свертки последовательностей. Производящие функции обладают еще одним интересным свойством. Пусть даны две последовательности $(a_0, a_1, \dots, a_n, \dots)$ и $(b_0, b_1, \dots, b_n, \dots)$, производящие функции для которых соответственно равны $G_1(z)$ и $G_2(z)$. Тогда сверткой данных последовательностей называется последовательность $(c_0, c_1, \dots, c_n, \dots)$ такая, что

$$c_n = \sum_{k=0}^n a_k b_{n-k},$$

а производящая функция $G(z)$ для этой последовательности равна

$$G(z) = G_1(z)G_2(z).$$

Отметим интересный факт. Если последовательность $(b_0, b_1, \dots, b_n, \dots)$ такова, что $b_n = 1$ для всех значений n , то свертка представляет последовательность частичных сумм последовательности $(a_0, a_1, \dots, a_n, \dots)$, т. е.

$$c_n = \sum_{k=0}^n a_k.$$

Поскольку для последовательности, состоящей из единиц, производящая функция равна $G_2(z) = 1/(1-z)$, то производящая функция $G(z)$ для последовательности частичных сумм равна

$$G(z) = \frac{G_1(z)}{1-z}.$$

Используем этот факт для вычисления сумм.

Пример 5.9. Вычислить сумму

$$s_n = \sum_{k=0}^n k^2.$$

Рассмотрим последовательность $a_n = n^2$. Найдем производящую функцию

$$G_1(z) = \sum_{n=0}^{\infty} n^2 z^n.$$

Имеем

$$\begin{aligned} G_1(z) &= \sum_{n=0}^{\infty} n^2 z^n = z \sum_{n=1}^{\infty} n^2 z^{n-1} = z \frac{d}{dz} \left(\sum_{n=1}^{\infty} n z^n \right) = \\ &= z \frac{d}{dz} \left(z \sum_{n=1}^{\infty} n z^{n-1} \right) = z \frac{d}{dz} \left(z \frac{d}{dz} \left(\sum_{n=1}^{\infty} z^n \right) \right) = z \frac{d}{dz} \left(z \frac{d}{dz} \left(\frac{z}{1-z} \right) \right) = \\ &= z \frac{d}{dz} \left(\frac{z}{(1-z)^2} \right) = \frac{z+z^2}{(1-z)^3}. \end{aligned}$$

Тогда для последовательности $s_n = \sum_{k=0}^n k^2$ производящая функция имеет вид

$$G(z) = \frac{1}{1-z} G_1(z) = \frac{z+z^2}{(1-z)^4}.$$

Разложим это выражение на элементарные дроби:

$$G(z) = \frac{A}{(z-1)^4} + \frac{B}{(z-1)^3} + \frac{C}{(z-1)^2} + \frac{D}{z-1},$$

получаем

$$z+z^2 = A(z-1) + B(z-1)^2 + C(z-1)^3 + D(z-1)^4.$$

Система уравнений для определения неизвестных коэффициентов имеет вид

$$\begin{cases} D = 0; \\ C - 3D = 1; \\ B - 2C + 3D = 1; \\ A - B + C - D = 0. \end{cases}$$

решая эту систему, получаем $D = 0$, $C = 1$, $B = 3$, $A = 2$, и, следовательно,

$$G(z) = \frac{2}{(z-1)^4} + \frac{3}{(z-1)^3} + \frac{1}{(z-1)^2}.$$

Используя соотношение (5.2.1), получаем

$$s_n = \sum_{k=0}^n k^2 = 2C_{3+n}^3 - 3C_{2+n}^2 + C_{1+n}^1 = \frac{n(n+1)(2n+1)}{6}.$$

Использование аппарата производящих функций для решения некоторых задач комбинаторики. Вначале уточним наше понимание биномиальных коэффициентов: C_n^k — число сочетаний по k из n . В уточненном понимании мы будем использовать для них обозначение $C(n, k)$. Будем считать, что $n \geq 0$, тогда

$$C(n, k) = C_n^k = \frac{n!}{k!(n-k)!}:$$

при $0 \leq k \leq n$ это есть число сочетаний по k из n ; $C(n, k) = 0$ при $k < 0$ и $k > n$. Ранее мы показали, что производящая функция $(1+z)^n$ соответствует последовательности $(a_0, a_1, \dots, a_n, \dots)$, где $a_k = C(n, k)$. Отметим, что при $k > n$ все члены последовательности равны нулю. Рассмотрим два положительных целых числа m и r таких, что $m+r = n$. Производящей функции $G_1(z) = (1+z)^m$ соответствует последовательность $(b_0, b_1, \dots, b_n, \dots)$, где $b_k = C(m, k)$. Производящей функции $G_2(z) = (1+z)^r$ соответствует последовательность $(c_0, c_1, \dots, c_n, \dots)$, где $c_k = C(r, k)$.

Произведению этих функций соответствует функция

$$G(z) = G_1(z)G_2(z) = (1+z)^m(1+z)^r = (1+z)^{m+r} = (1+z)^n.$$

Этой производящей функции соответствует последовательность $(a_0, a_1, \dots, a_n, \dots)$, где $a_k = C(n, k)$. С другой стороны, последовательность $(a_0, a_1, \dots, a_n, \dots)$ является сверткой последовательностей $(b_0, b_1, \dots, b_n, \dots)$ и $(c_0, c_1, \dots, c_n, \dots)$, следовательно, для любого k имеем

$$a_k = \sum_{l=0}^k b_l c_{k-l}.$$

Следовательно, имеем равенство

$$C(n, k) = \sum_{l=0}^k C(m, l)C(r, k-l).$$

Так как $n = r+m$, то это равенство можно записать в виде

$$C(r+m, k) = \sum_{l=0}^k C(m, l)C(r, k-l).$$

Это соотношение называется правилом свертки Вандермонда.

Рассмотрим еще один пример. Производящей функции $G_1(z) = (1+z)^n$ соответствует последовательность $(b_0, b_1, \dots, b_n, \dots)$, где $b_k =$

$= C(n, k)$. Производящей функции $G_2(z) = (1 - z)^n$ соответствует последовательность $(c_0, c_1, \dots, c_n, \dots)$, где $c_k = (-1)^k C(n, k)$. Произведению этих функций соответствует функция

$$G(z) = G_1(z)G_2(z) = (1 + z)^n(1 - z)^n = (1 - z^2)^n.$$

Этой производящей функции соответствует последовательность $(a_0, a_1, \dots, a_n, \dots)$, где $a_{2m} = (-1)^m C(n, m)$, $a_{2m+1} = 0$. Поскольку

$$a_k = \sum_{l=0}^k b_l c_{k-l},$$

то получаем следующее свойство биномиальных коэффициентов:

$$a_{2m} = \sum_{l=0}^{2m} b_l c_{2m-l} = \sum_{l=0}^{2m} C(n, l)(-1)^{2m-l} C(n, 2m-l),$$

которое можно представить в виде

$$(-1)^m C(n, m) = \sum_{l=0}^{2m} (-1)^l C(n, l) C(n, 2m-l).$$

Теперь рассмотрим примеры применения производящих функций для решения комбинаторных задач.

Пример 5.10. Сколько существует способов выдать сумму 12 копеек, используя монеты достоинством в 1 и 5 копеек?

Число способов выдать сумму в k копеек однокопеечными монетами образует последовательность $(b_0, b_1, \dots, b_n, \dots)$, где $b_k = 1$. Производящая функция этой последовательности $G_1(z)$ имеет вид

$$G_1(z) = 1 + z + z^2 + \dots + z^k + \dots = \frac{1}{1-z}.$$

Для монет достоинством в 5 копеек соответствующая последовательность $(c_0, c_1, \dots, c_n, \dots)$, где $c_k = 1$, если k кратно 5, и $c_k = 0$ в противном случае. Производящая функция этой последовательности $G_2(z)$ имеет вид:

$$G_2(z) = 1 + z^5 + z^{10} + \dots + z^{5k} + \dots = \frac{1}{1-z^5}.$$

Тогда последовательность числа способов для сумм, которые можно выдать однокопеечными и пятикопеечными монетами, является сверткой данных последовательностей и, значит, ее производящая функция $G(z)$ равна

$$G(z) = G_1(z)G_2(z) = \frac{1}{1-z} \cdot \frac{1}{1-z^5}.$$

Тогда число способов, которыми можно выдать сумму в n копеек, равно коэффициенту при z^n . Найдем этот коэффициент для $n = 12$. Для нахождения этого коэффициента воспользуемся представлением

функций в виде ряда. Нетрудно заметить, что z^{12} можно представить тремя способами:

$$z^{12} = z^{12} \cdot 1, \quad z^{12} = z^7 z^5, \quad z^{12} = z^2 z^{10},$$

следовательно, существуют три способа: 12 монет по 1 копейке, 7 монет по 1 копейке и 1 монета в 5 копеек, 2 монеты по 1 копейке и 2 монеты по 5 копеек.

До этого момента мы не использовали аппарат производящих функций для решения других задач. Покажем на примерах, как можно использовать этот аппарат для решения рекуррентных соотношений.

Пример 5.11. Решить рекуррентное соотношение

$$\begin{cases} a_0 = 0; \\ a_n = \frac{1}{2}a_{n-1} + \frac{1}{2^n}, \quad n \geq 1. \end{cases}$$

Найдем a_n с использованием производящей функции

$$G(z) = \sum_{n=0}^{\infty} a_n z^n.$$

Для этого обе части рекуррентного соотношения

$$a_n = \frac{1}{2}a_{n-1} + \frac{1}{2^n}$$

умножим на z^n и просуммируем при $1 \leq n \leq \infty$, получаем

$$\sum_{n=1}^{\infty} a_n z^n = \frac{1}{2} \sum_{n=1}^{\infty} a_{n-1} z^n + \sum_{n=1}^{\infty} \frac{1}{2^n} z^n.$$

Поскольку $a_0 = 0$, то

$$\sum_{n=1}^{\infty} a_n z^n = \sum_{n=0}^{\infty} a_n z^n = G(z).$$

Преобразуем $\sum_{n=1}^{\infty} a_{n-1} z^n$, имеем

$$\sum_{n=1}^{\infty} a_{n-1} z^n = \sum_{n=0}^{\infty} a_n z^{n+1} = z \sum_{n=0}^{\infty} a_n z^n = zG(z),$$

$$\sum_{n=1}^{\infty} \frac{1}{2^n} z^n = \sum_{n=1}^{\infty} \left(\frac{z}{2}\right)^n = \frac{z}{2} \cdot \frac{1}{1 - \frac{z}{2}} = -\frac{z}{z-2}$$

как сумма геометрической прогрессии. Тогда для определения $G(z)$ имеем

$$G(z) = \frac{1}{2}zG(z) - \frac{z}{z-2}.$$

Следовательно,

$$G(z) = \frac{2z}{(z-2)^2}.$$

Для элементарной дроби $\frac{1}{(z-2)^2}$, согласно (5.2.1), последовательность $(b_0, b_1, \dots, b_n, \dots)$ имеет вид

$$b_n = \frac{1}{2^{n+2}} C_{n+1}^1 = \frac{n+1}{2^{n+2}}.$$

Поскольку умножение на z сдвигает последовательность на единицу вправо, то мы получаем при $n \geq 1$

$$a_n = 2b_{n-1} = \frac{n}{2^n}.$$

Отметим, что поскольку $a_0 = 0$, то эта формула верна и при $n = 0$.

Следовательно, при $n \geq 0$

$$a_n = \frac{n}{2^n}.$$

Пример 5.12. Решить рекуррентное соотношение

$$\begin{cases} a_0 = 0; & a_1 = 1; \\ a_n = \frac{3}{4}a_{n-1} - \frac{1}{8}a_{n-2} + \frac{9}{32}n + \frac{3}{8}, & n \geq 2. \end{cases}$$

Введем производящую функцию

$$G(z) = \sum_{n=0}^{\infty} a_n z^n.$$

Обе части рекуррентного соотношения умножим на z^n и просуммируем при $2 \leq n < \infty$.

$$\sum_{n=2}^{\infty} a_n z^n = \frac{3}{4} \sum_{n=2}^{\infty} a_{n-1} z^n - \frac{1}{8} \sum_{n=2}^{\infty} a_{n-2} z^n + \frac{9}{32} \sum_{n=2}^{\infty} n z^n + \frac{3}{8} \sum_{n=2}^{\infty} z^n.$$

Заметим, что

$$\sum_{n=2}^{\infty} a_n z^n = \sum_{n=0}^{\infty} a_n z^n - a_0 - a_1 z = G(z) - z;$$

$$\sum_{n=2}^{\infty} a_{n-1} z^n = \sum_{n=1}^{\infty} a_n z^{n+1} = \sum_{n=0}^{\infty} a_n z^{n+1} - a_0 z = zG(z);$$

$$\sum_{n=2}^{\infty} a_{n-2} z^n = \sum_{n=0}^{\infty} a_n z^{n+2} = z^2 G(z);$$

$$\sum_{n=2}^{\infty} nz^n = z \sum_{n=2}^{\infty} nz^{n-1} = z \frac{d}{dz} \left(\sum_{n=2}^{\infty} z^n \right) = z \frac{d}{dz} \left(\frac{z^2}{1-z} \right) = \frac{-z^3 + 2z^2}{(1-z)^2};$$

$$\sum_{n=2}^{\infty} z^n = \frac{z^2}{1-z}.$$

Для определения функции $G(z)$ имеем уравнение

$$G(z) - z = \frac{3}{4}zG(z) - \frac{1}{8}z^2G(z) - \frac{9(z^3 - 2z^2)}{32(1-z)^2} + \frac{3z^2}{8(1-z)},$$

тогда

$$G(z) = \frac{11z^3 - 34z^2 + 32z}{4(z-1)^2(z^2 - 6z + 8)} = \frac{11z^3 - 34z^2 + 32z}{4(z-1)^2(z-2)(z-4)}.$$

Разложим это выражение на элементарные дроби:

$$\frac{11z^3 - 34z^2 + 32z}{(z-1)^2(z-2)(z-4)} = \frac{A}{(z-1)^2} + \frac{B}{z-1} + \frac{C}{z-2} + \frac{D}{z-4}.$$

Определяем коэффициенты:

$$11z^3 - 34z^2 + 32z = A(z-2)(z-4) + B(z-1)(z-2)(z-4) + C(z-1)^2(z-4) + D(z-1)^2(z-2).$$

Приравнивая коэффициенты при одинаковых степенях z , получаем

$$\begin{cases} B + C + D = 11; \\ A - 7B - 6C - 4D = -34; \\ -6A + 15B + 9C + 5D = 32; \\ 8A - 12B - 4C - 2D = 0. \end{cases}$$

Часть коэффициентов определим независимым путем. Полагая последовательно $z = 1$, $z = 2$, и $z = 4$, получаем $3A = 9$, и $A = 3$; $-2C = 16$, и $C = -8$; $18D = 288$, следовательно $D = 16$, тогда $B = 3$, и, окончательно

$$G(z) = \frac{3}{4(z-1)^2} + \frac{3}{4(z-1)} - \frac{2}{z-2} + \frac{4}{z-4}.$$

Используя (5.2.1), получаем

$$a_n = \frac{3}{4}(n+1) - \frac{3}{4} + \left(\frac{1}{2}\right)^n - \left(\frac{1}{4}\right)^n = \frac{3}{4}n + \frac{1}{2^n} - \frac{1}{4^n}.$$

Пример 5.13. Числа Фибоначчи могут быть заданы как рекурсивная последовательность вида

$$\begin{cases} a_0 = 0, & a_1 = 1; \\ a_n = a_{n-1} + a_{n-2}, & n \geq 2. \end{cases}$$

Введем производящую функцию

$$G(z) = \sum_{n=0}^{\infty} a_n z^n$$

Обе части рекуррентного соотношения умножим на z^n и просуммируем при $2 \leq n < \infty$:

$$\sum_{n=2}^{\infty} a_n z^n = \sum_{n=2}^{\infty} a_{n-1} z^n + \sum_{n=2}^{\infty} a_{n-2} z^n,$$

имеем

$$\sum_{n=2}^{\infty} a_n z^n = \sum_{n=0}^{\infty} a_n z^n - a_0 - a_1 z = G(z) - z,$$

$$\sum_{n=2}^{\infty} a_{n-1} z^n = \sum_{n=1}^{\infty} a_n z^{n+1} = \sum_{n=0}^{\infty} a_n z^{n+1} - a_0 z = z \sum_{n=0}^{\infty} a_n z^n = zG(z),$$

$$\sum_{n=2}^{\infty} a_{n-2} z^n = \sum_{n=0}^{\infty} a_n z^{n+2} = z^2 G(z).$$

Получаем

$$G(z) - z = zG(z) + z^2 G(z),$$

$$\begin{aligned} G(z) &= \frac{-z}{z^2 + z - 1} = \frac{-z}{\left(z - \frac{-1+\sqrt{5}}{2}\right)\left(z - \frac{-1-\sqrt{5}}{2}\right)} = \\ &= \frac{1}{\sqrt{5}} \left(-\frac{\frac{-1+\sqrt{5}}{2}}{z - \frac{-1+\sqrt{5}}{2}} + \frac{\frac{-1-\sqrt{5}}{2}}{z - \frac{-1-\sqrt{5}}{2}} \right), \end{aligned}$$

тогда, используя умножение на сопряженное выражение, имеем

$$\begin{aligned} a_n &= \frac{1}{\sqrt{5}} \left(\frac{2^n}{(-1 + \sqrt{5})^n} - \frac{2^n}{(-1 - \sqrt{5})^n} \right) = \\ &= \frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1 - \sqrt{5}}{2} \right)^n. \end{aligned}$$

Пример 5.14. До данного момента с помощью аппарата производящих функций мы исследовали линейные рекуррентные соотношения с постоянными коэффициентами. Покажем, что в отдельных случаях производящие функции позволяют исследовать рекуррентные

соотношения с переменными коэффициентами. Задана рекуррентная последовательность вида

$$\begin{cases} a_0 = 1; \\ a_n = \frac{p}{n} a_{n-1}, \quad n \geq 1. \end{cases}$$

Введем производящую функцию

$$G(z) = \sum_{n=0}^{\infty} a_n z^n.$$

Обе части рекуррентного соотношения умножим на z^n и просуммируем при $1 \leq n < \infty$:

$$\sum_{n=1}^{\infty} a_n z^n = \sum_{n=1}^{\infty} \frac{p}{n} a_{n-1} z^n.$$

Имеем

$$\sum_{n=1}^{\infty} a_n z^n = \sum_{n=0}^{\infty} a_n z^n - a_0 = G(z) - 1,$$

$$\sum_{n=1}^{\infty} \frac{p}{n} a_{n-1} z^n = p \sum_{n=1}^{\infty} \int_0^z a_{n-1} t^{n-1} dt = p \sum_{n=0}^{\infty} \int_0^z a_n t^n dt = p \int_0^z G(t) dt.$$

Для определения $G(z)$ получаем уравнение

$$G(z) - 1 = p \int_0^z G(t) dt,$$

которое дифференцированием сводится к линейному однородному дифференциальному уравнению первого порядка

$$\frac{dG}{dz} = pG$$

с начальным условием $G(0) = 1$. Решая данное дифференциальное уравнение, получаем $G = e^{pz}$. Нетрудно заметить, что

$$a_n = \left. \frac{1}{n!} D^n(G(z)) \right|_{z=0} = \left. \frac{1}{n!} D^n(e^{pz}) \right|_{z=0} = \frac{p^n}{n!}.$$

Пример 5.15. Задана рекуррентная последовательность вида

$$\begin{cases} a_0 = 0, \quad a_1 = p; \\ a_n = \frac{p^2}{n(n-1)} a_{n-2}, \quad n \geq 2. \end{cases}$$

Введем производящую функцию

$$G(z) = \sum_{n=0}^{\infty} a_n z^n.$$

Обе части рекуррентного соотношения умножим на z^n и просуммируем при $2 \leq n < \infty$:

$$\sum_{n=2}^{\infty} a_n z^n = \sum_{n=2}^{\infty} \frac{p^2}{n(n-1)} a_{n-2} z^n.$$

Имеем

$$\sum_{n=2}^{\infty} a_n z^n = \sum_{n=0}^{\infty} a_n z^n - a_0 - a_1 z = G(z) - pz,$$

$$\begin{aligned} \sum_{n=2}^{\infty} \frac{p^2}{n(n-1)} a_{n-2} z^n &= p^2 \sum_{n=0}^{\infty} \frac{p^2}{(n+2)(n+1)} a_n z^{n+2} = \\ &= p^2 \sum_{n=0}^{\infty} \int_0^z \frac{a_n}{n+1} t^{n+1} dt = p^2 \sum_{n=0}^{\infty} \int_0^z \left(\int_0^t a_n u^n du \right) dt = p^2 \int_0^z \left(\int_0^u G(u) du \right) dt \end{aligned}$$

Для определения $G(z)$ получаем уравнение

$$G(z) - pz = p^2 \int_0^z \left(\int_0^u G(u) du \right) dt.$$

Дифференцируя это уравнение два раза по z , получаем дифференциальное уравнение

$$\frac{d^2 G}{dz^2} = p^2 G$$

с начальными условиями

$$G(0) = 0, \quad \left. \frac{dG}{dz} \right|_{z=0} = p.$$

Полученное уравнение является линейным однородным дифференциальным уравнением второго порядка с постоянными коэффициентами. Его общее решение имеет вид

$$G(z) = Ae^{pz} + Be^{-pz},$$

где A и B — произвольные постоянные. Используя начальные условия, определяем значения произвольных постоянных. Выражение для производящей функции имеет вид

$$G(z) = \frac{1}{2}e^{pz} - \frac{1}{2}e^{-pz} = \operatorname{sh}(pz),$$

где $\operatorname{sh}(x) = (e^x - e^{-x})/2$ — гиперболический синус. Дифференцируя полученное выражение, получаем значения членов рекуррентной последовательности

$$a_n = \frac{p^n}{2n!} (1 - (-1)^n).$$

Отметим, что все четные члены последовательности равны нулю.

Пример 5.16. Задана рекуррентная последовательность вида

$$\begin{cases} a_0 = 1; \\ a_n = \frac{n-1+r}{n} a_{n-1}, \quad n \geq 1. \end{cases}$$

Введем производящую функцию

$$G(z) = \sum_{n=0}^{\infty} a_n z^n.$$

Обе части рекуррентного отношения умножим на z^n и просуммируем при $1 \leq n < \infty$, в итоге получаем:

$$\sum_{n=1}^{\infty} a_n z^n = \sum_{n=1}^{\infty} \frac{n-1+r}{n} a_{n-1} z^n.$$

Имеем

$$\sum_{n=1}^{\infty} a_n z^n = \sum_{n=0}^{\infty} a_n z^n - a_0 = G(z) - 1,$$

$$\begin{aligned} \sum_{n=1}^{\infty} \frac{n-1+r}{n} a_{n-1} z^n &= \sum_{n=1}^{\infty} a_{n-1} z^n + (r-1) \sum_{n=1}^{\infty} \int_0^z a_{n-1} t^{n-1} dt = \\ &= z \sum_{n=0}^{\infty} a_n z^n + (r-1) \sum_{n=0}^{\infty} \int_0^z a_n t^n dt = zG(z) + (r-1) \int_0^z G(t) dt. \end{aligned}$$

Для определения $G(z)$ получаем уравнение

$$G(z) - 1 = zG(z) + (r-1) \int_0^z G(t) dt,$$

которое дифференцированием сводится к линейному однородному дифференциальному уравнению первого порядка

$$(1-z) \frac{dG}{dz} = rG$$

с начальным условием $G(0) = 1$. Решая данное дифференциальное уравнение методом разделения переменных,

$$\frac{dG}{G} = \frac{r}{1-z} dz;$$

$$\int_1^G \frac{dt}{t} = \int_0^z \frac{r dt}{1-t}; \quad \ln G = -r \ln(1-z),$$

получаем

$$G(z) = \frac{1}{(1-z)^r}.$$

Значения членов последовательности предлагаем читателям найти самостоятельно.

§ 3. Методы исчисления конечных сумм

Понятия и обозначения. Понятие суммы часто встречается в математике, практике программирования и анализе алгоритмов, в том числе рекурсивных. Мы опишем кратко некоторые основные методы, которые используются при их исследовании и оценке, но вначале напомним некоторые основные понятия и обозначения. Речь будем вести о конечных суммах общего вида:

$$a_1 + a_2 + \dots + a_n,$$

где a_k — слагаемое суммы с номером k — есть некоторым образом определенное число. Каждый элемент суммы a_k называется ее членом. При этом предполагается, что он определяется по некоторому достаточно очевидному правилу, например, вычисляется как некоторая заданная функция аргумента k , т. е. $a_k = f(k)$. В качестве примера приведем следующие суммы:

$$1 + 4 + \dots + n^2,$$

$$1 + \frac{1}{4} + \dots + \frac{1}{n^2},$$

для первой суммы $f(k) = k^2$, а для второй — $f(k) = \frac{1}{k^2}$.

Запись суммы с использованием многоточия распространена в литературе, она наглядна, поскольку позволяет при наличии минимального воображения мысленно дополнить недостающие слагаемые, однако обладает некоторыми недостатками. Во-первых, эта запись громоздка, а во-вторых, не всегда допускает однозначное толкование. Поэтому при

более строгом подходе используются другие обозначения, общепринятыми являются следующие:

$$\sum_{k=1}^n a_k, \quad \text{или} \quad \sum_{1 \leq k \leq n} a_k. \quad (5.3.1)$$

Данное обозначение суммы введено в математику Жозефом Фурье в 1820 году и называется сигма-обозначением. Отметим, что когда мы пишем сумму вида (5.3.1), то предполагаются два момента. Первый — существует сравнительно простое правило вычисления членов суммы a_k по известному номеру k . Второй — нас интересует достаточно большое множество возможных значений n . Например, если нам требуется только найти сумму $1 + 2 + 3 + 4 + 5$, то нет смысла писать $1 + 2 + \dots + 5$ или тем более $\sum_{k=1}^5 k$, а необходимо сразу написать $1 + 2 + 3 + 4 + 5 = 15$.

Как мы уже отмечали, каждую конечную сумму можно рассматривать как рекурсивно заданную последовательность. Поэтому все методы, применяемые к исследованию рекурсивно заданных последовательностей, могут быть перенесены на исследование конечных сумм. С другой стороны, при исследовании конечных сумм существуют и свои собственные методы, а само исследование асимптотики рекурсивно заданных последовательностей приводит к необходимости получения оценок некоторых сумм.

Прежде чем перейти к дальнейшему материалу, приведем две «классические» суммы — суммы арифметической и геометрической прогрессий. Сумма арифметической прогрессии — это сумма вида

$$\sum_{k=1}^n (a + (k - 1) \cdot d) = na + \frac{(n - 1)n}{2}d,$$

а сумма геометрической прогрессии — это сумма вида

$$\sum_{k=1}^n aq^{k-1} = a \frac{q^n - 1}{q - 1}.$$

Сами эти формулы мы получим ниже.

Цели, которые ставятся при исследовании конечных сумм, во многом аналогичны целям, возникающим при исследовании рекурсивно заданных последовательностей. Эти цели сформулированы ранее и нет необходимости их повторять.

Пример 5.17. Исследовать сумму вида $S(n) = \sum_{k=1}^n k^3$.

Получим оценку O -большое для суммы $S(n) = \sum_{k=1}^n k^3$. Очевидно, что для всех $1 \leq k \leq n$ справедливо $k^3 \leq n^3$, тогда получаем

$$S(n) = \sum_{k=1}^n k^3 \leq \sum_{k=1}^n n^3 = n^4.$$

Этот результат позволяет сделать вывод, что

$$S(n) = \sum_{k=1}^n k^3 = O(n^4).$$

Используя метод математической индукции, попытаемся доказать, что функция $g(n) = n^4$ является асимптотически точной оценкой для суммы

$$S(n) = \sum_{k=1}^n k^3, \quad \text{т. е.} \quad S(n) = \Theta(n^4).$$

Суть метода состоит в том, что мы «угадываем» решение, проверяем его правильность при $n = 1$, и, считая, что результат верен при $n > 1$, доказываем его для $n + 1$. Для этого, в соответствии с определением оценки Θ , требуется найти такое значение константы c_1 и такое значение n_0 , что при всех $n > n_0$ выполнено

$$S(n) = \sum_{k=1}^n k^3 \geq c_1 n^4. \quad (5.3.2)$$

Не будем конкретизировать значение c_1 . Сам метод позволит дать оценку этому числу. Пусть для некоторого значения n выполнено условие (5.3.2). Требуется так подобрать c_1 , чтобы для всех $n > n_0$ было выполнено неравенство

$$S(n+1) = \sum_{k=1}^{n+1} k^3 \geq c_1 (n+1)^4. \quad (5.3.3)$$

Проведем следующую цепочку рассуждений. Имеем

$$\begin{aligned} S(n+1) &= \sum_{k=1}^{n+1} k^3 = \sum_{k=1}^n k^3 + (n+1)^3 = \\ &= S(n) + (n+1)^3 \geq c_1 n^4 + (n+1)^3. \end{aligned}$$

Если нам удастся подобрать такое значение c_1 , что

$$c_1 n^4 + (n+1)^3 \geq c_1 (n+1)^4 \quad (5.3.4)$$

при всех значениях n , то заведомо будет выполнено неравенство (5.3.3), которое нам и требуется доказать. Тогда для определения c_1 имеем неравенство (5.3.4), преобразуя которое, получаем

$$(n+1)^3 \geq c_1 \left((n+1)^4 - n^4 \right), \quad c_1 \leq \frac{(n+1)^3}{4n^3 + 6n^2 + 4n + 1}.$$

Заметим, что для всех $n \geq 1$ выполнено

$$\frac{(n+1)^3}{4n^3 + 6n^2 + 4n + 1} > \frac{n^3}{4n^3 + 6n^2 + 4n^2 + n^3} = \frac{1}{15},$$

тогда выберем $c_1 = \frac{1}{15}$, очевидно, что $S(1) = 1^3 \geq \frac{1}{15} 1^4$, следовательно, мы можем утверждать, что при всех $n \geq 1$ выполнено

$$S(n) = \sum_{k=1}^n k^3 \geq \frac{1}{15} n^4.$$

Ранее мы показали, что при всех $n \geq 1$ выполнено $S(n) \leq n^4$, следовательно, при значениях $c_1 = 1/15$, $c_2 = 1$ получаем $c_1 n^4 \leq S(n) \leq c_2 n^4$, что влечет $S(n) = \Theta(n^4)$, и функция $g(n) = n^4$ является асимптотически точной оценкой для функции $S(n)$.

Теперь попытаемся найти эквивалентную оценку исследуемой суммы, используя метод сравнения с определенным интегралом. Заметим, что при $k \geq 1$ выполнены очевидные неравенства

$$\int_{k-1}^k x^3 dx < k^3 < \int_k^{k+1} x^3 dx,$$

тогда

$$S(n) = \sum_{k=1}^n k^3 > \int_0^n x^3 dx = \frac{1}{4} n^4,$$

но другой стороны

$$S(n) = \sum_{k=1}^n k^3 < \int_1^{n+1} x^3 dx = \frac{1}{4} \left((n+1)^4 - 1 \right).$$

Покажем, что функция $g(n) = \frac{1}{4} n^4$ эквивалентна $S(n)$ при больших n . Имеем

$$1 \leq \frac{S(n)}{(1/4) n^4} \leq \frac{(1/4) \left((n+1)^4 - 1 \right)}{(1/4) n^4},$$

$$\lim_{n \rightarrow \infty} \frac{(1/4) \left((n+1)^4 - 1 \right)}{(1/4) n^4} = \lim_{n \rightarrow \infty} \frac{(1+1/n)^4 - 1/n^4}{1} = 1.$$

Из полученного результата следует, что

$$\lim_{n \rightarrow \infty} \left(\frac{S(n)}{(1/4) n^4} \right) = 1,$$

и, следовательно,

$$S(n) = \sum_{k=1}^n k^3 \sim \frac{1}{4} n^4.$$

И, наконец, найдем точное значение исследуемой суммы. Для этого воспользуемся методом математической индукции и следующими соображениями. Ранее мы показали уже один способ отыскания этой суммы, используя методы решения линейных рекуррентных отношений. Теперь используем другой прием. В силу полученных выше оценок попытаемся искать значение суммы в виде многочлена четвертой степени от n , т. е. в виде

$$S(n) = An^4 + Bn^3 + Cn^2 + Dn + E. \quad (5.3.5)$$

Поскольку $S(1) = 1$ — базис индукции, то имеем

$$A + B + C + D + E = 1.$$

Пусть $S(n)$ имеет вид (5.3.5) — предположение индукции, тогда

$$S(n+1) = A(n+1)^4 + B(n+1)^3 + C(n+1)^2 + D(n+1) + E,$$

с другой стороны,

$$S(n+1) = S(n) + (n+1)^3. \quad (5.3.6)$$

Тогда, подставляя в (5.3.6) значение $S(n)$ из формулы (5.3.5), имеем равенство

$$\begin{aligned} An^4 + Bn^3 + Cn^2 + Dn + E + (n+1)^3 &= \\ &= A(n+1)^4 + B(n+1)^3 + C(n+1)^2 + D(n+1) + E, \end{aligned}$$

которое должно выполняться для всех значений n . Приравнивая коэффициенты при одинаковых степенях n , получаем систему уравнений для определения коэффициентов A, B, C, D :

$$\begin{cases} 4A = 1; \\ 6A + 3B = 3; \\ 4A + 3B + 2C = 3; \\ A + B + C + D = 1. \end{cases}$$

Решая эту систему, получаем $A = \frac{1}{4}$, $B = \frac{1}{2}$, $C = \frac{1}{2}$, $D = 0$. Зная коэффициенты A, B, C, D , из уравнения

$$A + B + C + D + E = 1$$

находим, что $E = 0$, и, подставляя значения коэффициентов в (5.3.5), имеем

$$S(n) = \sum_{k=1}^n \frac{1}{k^3} = \frac{1}{4}n^4 + \frac{1}{2}n^3 + \frac{1}{4}n^2 = \frac{n^2(n+1)^2}{4}.$$

Теперь напомним некоторые основные свойства конечных сумм. Первое — это то, что конечные суммы допускают любую перестановку слагаемых. Второй важный факт — это свойство линейности конечной суммы. Математически это записывается следующим образом:

$$\sum_{k=1}^n (ca_k + db_k) = c \sum_{k=1}^n a_k + d \sum_{k=1}^n b_k.$$

Приведем пример вычисления значения суммы с использованием указанных свойств. Отметим, что в математике активно используются три приема доказательств: 1) тождественное преобразование выражений; 2) прибавление выражения, равного нулю; 3) умножение на выражение, равное единице. Эти приемы мы будем в дальнейшем активно использовать.

Пример 5.18. Найти функциональную зависимость $S(n)$ для суммы

$$S(n) = \sum_{k=1}^n \frac{1}{k(k+2)}.$$

Нетрудно вычислить первые два значения $S(n)$

$$S(1) = \frac{1}{3}; \quad S(2) = \frac{1}{3} + \frac{1}{8} = \frac{11}{24}.$$

Перепишем выражение для этой суммы несколько иначе и воспользуемся свойством линейности, тогда получаем

$$S(n) = \sum_{k=1}^n \left(\frac{1}{2k} - \frac{1}{2(k+2)} \right) = \frac{1}{2} \sum_{k=1}^n \frac{1}{k} - \frac{1}{2} \sum_{k=1}^n \frac{1}{k+2}.$$

Введем обозначения для сумм в правой части

$$U(n) = \sum_{k=1}^n \frac{1}{k}, \quad V(n) = \sum_{k=1}^n \frac{1}{k+2},$$

и исследуем вторую сумму — $V(n)$. Для ее исследования удобно прибавить ноль, записанный в следующем виде: $0 = \frac{1}{1} + \frac{1}{2} - \frac{1}{1} - \frac{1}{2}$, тогда имеем

$$\begin{aligned} V(n) &= \sum_{k=1}^n \frac{1}{k+2} = \frac{1}{1} + \frac{1}{2} + \sum_{k=1}^n \frac{1}{k+2} - \frac{1}{1} - \frac{1}{2} = \\ &= \sum_{k=1}^{n+2} \frac{1}{k} - \frac{1}{1} - \frac{1}{2} = \sum_{k=1}^n \frac{1}{k} + \frac{1}{n+1} + \frac{1}{n+2} - \frac{1}{1} - \frac{1}{2}, \end{aligned}$$

следовательно,

$$V(n) = U(n) + \frac{1}{n+1} + \frac{1}{n+2} - \frac{1}{1} - \frac{1}{2},$$

тогда

$$S(n) = \frac{1}{2}(U(n) - V(n)) = \frac{1}{2}\left(U(n) - U(n) - \frac{1}{n+1} - \frac{1}{n+2} + \frac{1}{1} + \frac{1}{2}\right).$$

Получаем, что при $n \geq 3$

$$S(n) = \frac{3}{4} - \frac{1}{2(n+1)} - \frac{1}{2(n+2)}.$$

Пример 5.19. В качестве еще одного примера вычислим сумму геометрической прогрессии. Преобразуем ее к виду

$$\begin{aligned} S(n) &= \sum_{k=1}^n aq^{k-1} = a + \sum_{k=2}^n aq^{k-1} = a + q \sum_{k=2}^n aq^{k-2} = \\ &= a + \left(q \sum_{k=2}^n aq^{k-2} + aq^n\right) - aq^n \end{aligned}$$

(мы прибавили ноль, записанный в виде $0 = aq^n - aq^n$), далее получаем

$$\begin{aligned} S(n) &= a + q \left(\sum_{k=2}^n aq^{k-2} + aq^{n-1} \right) - aq^n = \\ &= a + q \left(\sum_{k=2}^{n+1} aq^{k-2} \right) - aq^n = a(1 - q^n) + q \left(\sum_{k=1}^n aq^{k-1} \right). \end{aligned}$$

Так как

$$\sum_{k=1}^n aq^{k-1} = S(n),$$

то для определения суммы $S(n)$ мы получили уравнение

$$S(n) = a(1 - q^n) + qS(n),$$

решая которое, получаем искомый результат:

$$S(n) = \sum_{k=1}^n aq^{k-1} = a \frac{q^n - 1}{q - 1}.$$

Пример 5.20. Доказать методом математической индукции формулу суммы арифметической прогрессии

$$\sum_{k=1}^n (a + (k-1)d) = na + \frac{(n-1)n}{2}d.$$

Обозначим через

$$S(n) = \sum_{k=1}^n (a + (k-1)d).$$

При $n = 1$ справедливость данной формулы очевидна. Предположим, что она верна для некоторого значения n , тогда для значения $(n+1)$ имеем

$$\begin{aligned} S(n+1) &= \sum_{k=1}^{n+1} (a + (k-1)d) = \sum_{k=1}^n (a + (k-1)d) + a + nd = \\ &= S(n) + a + nd. \end{aligned}$$

Поскольку

$$S(n) = na + \frac{(n-1)n}{2}d,$$

то

$$S(n+1) = na + \frac{(n-1)n}{2}d + a + nd = (n+1)a + \frac{n(n+1)}{2}d,$$

что и требовалось доказать.

Пример 5.21. Вычислить сумму вида

$$S(n) = \sum_{k=1}^n k3^k. \quad (5.3.7)$$

Использование операции дифференцирования непрерывных функций иногда также оказывается полезным для вычисления сумм. Покажем это на предложенном примере. Вначале «усложним» задачу. Вместо данной суммы рассмотрим функцию

$$f(x) = \sum_{k=1}^n kx^k,$$

тогда интересующая нас сумма есть просто значение указанной выше функции в точке $x = 3$, т. е. $\sum_{k=1}^n k3^k = f(3)$.

Проведем некоторые преобразования, которые в конечном итоге позволяют найти данную сумму.

$$f(x) = x \sum_{k=1}^n kx^{k-1} = x \frac{d}{dx} \left(\sum_{k=1}^n x^k \right).$$

Сумма в скобках является суммой геометрической прогрессии и может быть легко вычислена

$$\sum_{k=1}^n x^k = \sum_{k=1}^n x \cdot x^{k-1} = x \frac{x^n - 1}{x - 1} = \frac{x^{n+1} - x}{x - 1}.$$

Вычисляя производную, имеем

$$\frac{d}{dx} \left(\sum_{k=1}^n x^k \right) = \frac{nx^{n+1} - (n+1)x^n + 1}{(x-1)^2}.$$

Таким образом,

$$f(x) = \sum_{k=1}^n kx^k = x \frac{nx^{n+1} - (n+1)x^n + 1}{(x-1)^2}, \quad (5.3.8)$$

и, следовательно, подставляя значение $x = 3$, окончательно получаем

$$\sum_{k=1}^n k3^k = f(3) = \frac{3}{4} (n3^{n+1} - (n+1)3^n + 1).$$

Отметим, что полученный результат позволяет найти точные значения сумм вида (5.3.7) с разными основаниями степени. Так, например, сумма

$$S(n) = \sum_{k=1}^n k2^k = f(2),$$

которая встречается при анализе трудоемкости некоторых алгоритмов, может быть без труда получена на основе формулы (5.3.8).

Пример 5.22. Получить асимптотическую оценку для суммы

$$U(n) = \sum_{k=1}^n \frac{k}{k+1} 2^{k-1}.$$

Для построения асимптотических оценок сумм бывает полезен метод почленного сравнения. Отметим также, что при построении асимптотических оценок мы можем пренебречь любым конечным числом слагаемых. Метод почленных сравнений основан на следующем факте. Пусть имеются две конечные суммы

$$U(n) = \sum_{k=1}^n a_k, \quad V(n) = \sum_{k=1}^n b_k,$$

при этом для всех значений k выполнено условие $a_k \leq b_k$. Тогда можно утверждать, что $U(n) \leq V(n)$. Этот факт удобен для построения асимптотических оценок. При этом для сравнения с заданной суммой выбирается такая сумма, значение которой или ее асимптотическая оценка известны.

Для решения данного примера выберем для сравнения геометрическую прогрессию

$$V(n) = \sum_{k=1}^n 2^{k-1},$$

сумма которой известна и равна

$$V(n) = 2^n - 1.$$

Поскольку $\frac{k}{k+1} < 1$, то $U(n) \leq V(n) = 2^n - 1$, и можно утверждать, что

$$U(n) = O(2^n).$$

Пример 5.23. Исследовать сумму гармонического ряда с целью получения асимптотической оценки. Гармонический ряд задается формулой

$$S(n) = \sum_{k=1}^n \frac{1}{k}.$$

Метод сравнения с определенным интегралом — еще один полезный прием построения асимптотических оценок. Этот прием удобно применять для исследования сумм, слагаемые которых имеют вид $a_k = f(k)$, и при этом функция действительной переменной $y = f(x)$ неотрицательна и монотонна для положительных значений переменной x .

Если функция $y = f(x)$ не убывает, то нетрудно видеть, что

$$a_k \leq \int_k^{k+1} f(x) dx \quad \text{и} \quad a_k \geq \int_{k-1}^k f(x) dx,$$

тогда для значения суммы имеем следующие оценки:

$$S(n) \leq \int_1^{n+1} f(x) dx \quad \text{и} \quad S(n) \geq a_1 + \int_1^n f(x) dx.$$

Если функция $y = f(x)$ не возрастает, то нетрудно видеть, что

$$a_k \geq \int_k^{k+1} f(x) dx \quad \text{и} \quad a_k \leq \int_{k-1}^k f(x) dx,$$

тогда для функции $S(n)$ имеем

$$S(n) \geq \int_1^{n+1} f(x) dx \quad \text{и} \quad S(n) \leq a_1 + \int_1^n f(x) dx.$$

В данном примере для сравнения рассмотрим функцию $f(x) = \frac{1}{x}$, которая монотонно убывает при положительных значениях аргумента.

Имеем

$$\int_1^{n+1} \frac{1}{x} dx \leq S(n) \leq 1 + \int_1^n \frac{1}{x} dx,$$

следовательно,

$$\ln(n+1) \leq S(n) \leq 1 + \ln(n).$$

Из данного неравенства легко показать, что $S(n) \sim \ln(n)$. Отметим, что можно доказать и более сильное утверждение, а именно, что

$$S(n) = \ln(n) + O(1).$$

Пример 5.24. Применить метод сравнения с определенным интегралом для исследования суммы

$$S(n) = \sum_{k=1}^n \ln k.$$

Поскольку функция $y = \ln x$ монотонно возрастает при положительных значениях аргумента, то имеют место следующие оценки для данной суммы:

$$\int_1^n \ln x dx < \sum_{k=1}^n \ln k < \int_1^{n+1} \ln x dx.$$

Поскольку $\int \ln x dx = x \ln x - x$, то получаем следующую оценку суммы

$$n \ln n - n < \sum_{k=1}^n \ln k < (n+1) \ln(n+1) - (n+1).$$

Полученная оценка позволяет сделать вывод, что

$$\sum_{k=1}^n \ln k \sim n \ln n.$$

В качестве приложения к полученному соотношению получим оценку значения $n!$. Поскольку

$$n! = e^{\ln(n!)} = \exp \left\{ \sum_{k=1}^n \ln k \right\}, \quad \text{то} \quad \frac{n^n}{e^n} < n! < \frac{(n+1)^{n+1}}{e^{n+1}}.$$

Отметим, что в литературе существуют и более точные оценки значения $n!$.

Пример 5.25. Доказать, что

$$\sum_{k=1}^n \frac{1}{k^2} = O(1).$$

Для этого требуется доказать, что найдется значение c такое, что

$$\sum_{k=1}^n \frac{1}{k^2} \leq c$$

для всех значений n . Рассмотрим функцию $f(x) = \frac{1}{x^2}$, которая монотонно убывает при положительных значениях аргумента. Тогда

$$\sum_{k=1}^n \frac{1}{k^2} \leq 1 + \int_1^n \frac{1}{x^2} dx = 1 - \frac{1}{n} + 1 = 2 - \frac{1}{n} \leq 2,$$

что и требовалось доказать.

Мы рассмотрели некоторые методы оценки и исчисления конечных сумм, которые могут быть полезны для исследования и анализа рекурсивных алгоритмов. Отметим, что дальнейшим обобщением исчисления конечных сумм является рассмотрение бесконечных сумм вида

$$\sum_{k=1}^{\infty} a_k,$$

которые называются рядами, являющимися предметом специальных разделов классического курса математического анализа.

§ 4. Функция $\beta_1(n)$ и другие специальные функции

Некоторые функции довольно часто встречаются при исследовании вопросов трудоемкости алгоритмов. Первая, которую мы рассмотрим — функция $\beta_1(n)$, означающая количество единиц в двоичном представлении числа n . Рекурсивное задание такой функции не представляет существенных трудностей:

$$\begin{cases} \beta_1(0) = 0; & \beta_1(2n) = \beta_1(n); \\ & \beta_1(2n+1) = \beta_1(2n) + 1. \end{cases}$$

Но, с другой стороны, ее аналитическое представление вряд ли возможно какими-либо простыми соотношениями. Отметим, что функция $\beta_1(n)$ не является монотонно возрастающей функцией, поскольку $\beta_1(n) = 1$ для всех $n = 2^k$, а для $n = 2^k - 1$ значение функции $\beta_1(n) = k$. Точечный график функции $\beta_1(n)$ для начальных значений n представлен на рис. 5.1.

Следующая функция — $\beta_0(n)$, значением которой является количество нулей в двоичном представлении числа n . Рекурсивное задание такой функции имеет вид

$$\begin{cases} \beta_0(0) = 1; & \beta_0(2n) = \beta_0(n) + 1; \\ & \beta_0(2n+1) = \beta_0(2n). \end{cases}$$

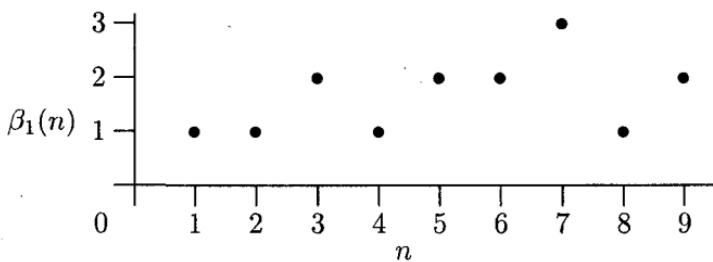


Рис. 5.1. Значения функции $\beta_1(n)$ для значений $n = 1, 2, \dots, 9$

Отметим, что функция $\beta_0(n)$ также не является монотонно возрастающей функцией, так как $\beta_0(n) = 0$ для всех $n = 2^k - 1$.

Следующая функция — $\beta(n)$, которая определена для целого положительного n , и $\beta(n)$ есть количество значащих бит в двоичном представлении целого числа n . Отметим, что функция $\beta(n)$ может быть задана аналитически в виде

$$\beta(n) = [\log_2(n)] + 1,$$

где $[z]$ — целая часть z , $n > 0$.

Укажем некоторые соотношения между этими функциями. В силу определения $\beta_1(n)$ справедливо неравенство

$$1 \leq \beta_1(n) \leq \beta(n),$$

а для любого $n > 0$ справедливо соотношение

$$\beta(n) = \beta_0(n) + \beta_1(n).$$

В целях анализа рекурсивных алгоритмов представляет также интерес определение среднего значения функции $\beta_1(n)$ для $n = 0, 1, N$, где $N = 2^k - 1$ (т. е. если двоичное представление числа n занимает не более k двоичных разрядов), обозначим его через $\beta_s(N)$, тогда

$$\beta_s(N) = \frac{1}{N+1} \sum_{m=0}^N \beta_1(m).$$

Поскольку количество чисел, имеющих l единиц в k разрядах, равно количеству сочетаний из l по k , то

$$\sum_{m=0}^N \beta_1(m) = \sum_{l=1}^k l C_k^l = \sum_{l=1}^k l \frac{k}{l} C_{k-1}^{l-1} = k \sum_{l=0}^{k-1} C_{k-1}^l = k 2^{k-1}.$$

Поскольку $N = 2^k - 1$, получаем

$$\beta_s(N) = \frac{1}{N+1} \sum_{m=0}^N \beta_1(m) = \frac{k 2^{k-1}}{2^k - 1 + 1} = \frac{k}{2} = \frac{\log_2(N+1)}{2} = \frac{\beta(N)}{2}.$$

Следующая функция, которая достаточно часто используется при анализе различных алгоритмов — это сумма гармонического ряда, которая обычно обозначается как H_n , т. е.

$$H_n = \sum_{k=1}^n \frac{1}{k}.$$

Отметим, что при больших значениях n имеет место асимптотическое представление

$$H_n = \ln n + \gamma + \frac{1}{2n} + O\left(\frac{1}{n^2}\right),$$

где $\gamma = 0,5772\dots$ — постоянная Эйлера.

При анализе комбинаторных соотношений возникают биномиальные коэффициенты. Хорошо известная формула вычисления биномиальных коэффициентов

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

обобщается на случай не обязательно целых значений $n = r$ выражением

$$C(r; 0) = 0, \quad C(r; k) = \frac{r(r-1)(r-2)\dots(r-k+1)}{k!}, \quad k > 1.$$

Последовательность $C(r; k)$ может быть задана рекуррентным соотношением

$$\begin{cases} C(r; 0) = 1, \\ C(r; k+1) = C(r; k) \frac{r-k}{k+1}, \quad k \geq 0. \end{cases}$$

Производящая функция $G(z)$, соответствующая данной последовательности, имеет вид $G(z) = (1+z)^r$.

Еще один вид специальных чисел, иногда возникающих при анализе алгоритмов — это числа Стирлинга первого и второго рода. Числа Стирлинга первого рода $S_n^{(k)}$ представляют собой коэффициенты разложения факториального многочлена степени n вида

$$x(x-1)(x-2)\dots(x-n+1)$$

по степеням x , т. е. факториальный многочлен представляется в виде суммы

$$x(x-1)(x-2)\dots(x-n+1) = \sum_{k=0}^n S_n^{(k)} x^k.$$

Если числа Стирлинга первого рода доопределить соотношением $S_k^{(n)} = 0$ при $k > n$ и $k \leq 0$, то они могут быть заданы рекуррентным

соотношением

$$\begin{cases} S_1^{(1)} = 1; & S_1^{(k)} = 0, \quad k \neq 1; \\ S_{n+1}^{(k)} = S_n^{(k-1)} - nS_n^{(k)}. \end{cases}$$

Числа Стирлинга второго рода $\sigma_n^{(k)}$ представляют собой коэффициенты разложения многочлена x^n по факториальным многочленам

$$x(x-1)(x-2) \dots (x-k+1),$$

то есть имеет место представление вида

$$x^n = \sum_{k=0}^n \sigma_n^{(k)} x(x-1)(x-2) \dots (x-k+1).$$

Если числа Стирлинга второго рода доопределить соотношением $\sigma_k^{(n)} = 0$ при $k > n$ и $k \leq 0$, то они могут быть заданы рекуррентным соотношением

$$\begin{cases} \sigma_1^{(1)} = 1; & \sigma_1^{(k)} = 0, \quad k \neq 1; \\ \sigma_{n+1}^{(k)} = \sigma_n^{(k-1)} + k\sigma_n^{(k)}. \end{cases}$$

§ 5. Комбинаторные соотношения и их связь с рекурсивными алгоритмами

Ряд известных комбинаторных тождеств и соотношений используется при анализе рекурсивных алгоритмов (см. главу 7). Мы демонстрируем эти тождества и соотношения на примерах.

Пример 5.26. Начнем с простейшего комбинаторного тождества для биномиальных коэффициентов

$$C_{n+1}^k = C_n^k + C_n^{k-1}.$$

Пока мы ограничиваемся целыми значениями n и k . В дальнейшем мы будем использовать расширенную трактовку биномиальных коэффициентов, поэтому будем их обозначать $C(n; k)$, где

$$C(n; k) = C_n^k = \frac{n!}{k!(n-k)!}$$

при $0 \leq k \leq n$ и $C(n; k) = 0$ в противном случае. Изначальная трактовка $C(n; k)$ — это число подмножеств, содержащих k элементов, на множестве, состоящем из n элементов. В этой трактовке очевидно, что $C(1; 0) = 1$, $C(1; 1) = 1$ и $C(n; k) = 0$ для остальных значений k . Предположим, что для некоторого значения n мы знаем все значения $C(n; k)$.

Попытаемся теперь определить $C(n+1; k)$. Будем рассматривать «старое» множество, содержащее n элементов, и «новое», в котором добавлен один элемент. Тогда можно определить $C(n+1; k)$ — как число подмножеств, содержащих k элементов на «новом» множестве.

Во-первых, остаются все подмножества, содержащие k элементов «старого» множества, и их число равно $C(n; k)$. Во-вторых, можно образовать подмножества, содержащие k элементов, если взять подмножества «старого» множества, содержащие $k - 1$ элемент и к ним добавить «новый» элемент. Число таких подмножеств равно $C(n; k - 1)$. Таким образом, для определения $C(n; k)$ получаем рекуррентное соотношение

$$C(n + 1; k) = C(n; k) + C(n; k - 1)$$

при $n \geq 1$.

Пример 5.27. В условиях предыдущего примера будем считать, что мы сумели определить все значения $C(i; k)$ для всех k и для всех значений $i < n$, где $n \geq 2$. Задача состоит в том, чтобы определить значение $C(n; k)$.

Поступим следующим образом. Разобьем произвольным образом данное исходное множество на два множества. Причем первое множество будет содержать m элементов, а второе $n - m$ элементов, где $1 \leq m < n$. Тогда подмножество, содержащее k элементов на исходном множестве, можно образовать так. Возьмем подмножество, содержащее j элементов в первом множестве, их число равно $C(m; j)$, и объединим его с подмножеством, содержащим $k - j$ элементов второго множества — их число равно $C(n - m; k - j)$. Следовательно, число таких подмножеств равно $C(m; j)C(n - m; k - j)$. Суммируя по j , получаем рекуррентное соотношение для определения $C(n; k)$:

$$C(n; k) = \sum_{j=0}^k C(m; j)C(n - m; k - j).$$

Это соотношение эквивалентно правилу свертки Вандермонда, полученному ранее с помощью аппарата теории производящих функций.

Пример 5.28. Рассмотрим следующую схему движения. В единицу времени частица сдвигается либо на единицу вверх, либо на единицу вправо. То есть в каждый момент времени положение точки можно охарактеризовать парой координат $(p; q)$, где p и q — целые неотрицательные числа. При этом имеются ограничения на их значения $p \leq m$, $q \leq n$. Тогда за промежуток времени $m + n$, считая, что в начальный момент $p = 0$, $q = 0$, частица придет в положение $p = m$, $q = n$. Всего существует C_{n+m}^m различных траекторий движения. Попробуем подсчитать количество траекторий другим способом. Будем различать траектории по первому выходу на верхний уровень в точку $(i; n)$, где $0 \leq i \leq m$. Для $0 \leq i \leq m$ количество таких траекторий равно C_{n-1+i}^i . Тогда получаем следующее равенство:

$$\sum_{i=0}^m C_{n-1+i}^i = C_{n+m}^m.$$

Заменяя n на $n + 1$, получаем известное биномиальное тождество

$$\sum_{i=0}^m C_{n+i}^i = C_{n+m+1}^m.$$

Пример 5.29. Известно биномиальное тождество

$$\sum_{k=0}^n C(k; m) = C(n+1; m+1).$$

Покажем простой способ получения этого тождества с использованием аппарата производящих функций. Рассмотрим производящую функцию

$$G(z) = \sum_{k=0}^n (1+z)^k.$$

Выражение $\sum_{k=0}^n C(k; m)$ является членом с номером m последовательности, соответствующей данной функции, поскольку коэффициенты $C(k; m)$ являются множителями в каждом слагаемом при z^m . Поскольку рассматриваемая сумма является суммой геометрической прогрессии, то получаем

$$G(z) = \frac{(1+z)^{n+1} - 1}{z}.$$

Рассмотрим $F(z) = zG(z) = (1+z)^{n+1} - 1$. Умножение на множитель z сдвигает соответствующую последовательность на единицу вправо. Коэффициент при z^{m+1} легко вычисляется и равен $C(n+1; m+1)$, следовательно

$$\sum_{k=0}^n C(k; m) = C(n+1; m+1).$$

Пример 5.30. Числа Каталана. Числа Каталана возникают в следующей задаче. Рассматривается пространство $2n$ -мерных векторов $(x_1; x_2; \dots; x_{2n})$, координаты которых могут принимать значения $x_i = \pm 1$. Нетрудно проверить, что общее число таких векторов равно 2^{2n} . Среди данных векторов выделяется отдельный класс, удовлетворяющий условиям

$$\sum_{i=1}^k x_i \geq 0 \quad \text{для всех } 1 \leq k \leq 2n-1, \quad \sum_{i=1}^{2n} x_i = 0.$$

Количество таких векторов, которое обозначается C_n , называется числом Каталана. По определению положим $C_0 = 0$. Нетрудно проверить, что $C_1 = 1$, $C_2 = 2$.

Данной задаче можно дать и другую интерпретацию. Рассмотрим схему движения частицы, предложенную в примере 5.28. Положим $m = n$ и введем дополнительное ограничение $q \leq p$. Тогда количество

таких траекторий и будет равно числу Каталана C_n . Попытаемся получить рекуррентное соотношение для определения чисел Каталана. Введем вспомогательную величину D_n как число векторов, удовлетворяющих более сильным условиям

$$\sum_{i=1}^k x_i > 0 \quad \text{для всех } 1 \leq k \leq 2n-1, \quad \sum_{i=1}^{2n} x_i = 0.$$

Рассмотрим вектор $(x_1; x_2; \dots; x_{2n})$, удовлетворяющий условиям

$$\sum_{i=1}^{2s-1} x_i > 0 \quad \text{для всех } s, \text{ таких что } 1 \leq s \leq m, \text{ и } \sum_{i=1}^{2m} x_i = 0.$$

Число таких векторов при $m = n$ равно D_n , а для $m < n$ равно $D_m C_{n-m}$. Суммируя по всем значениям, получаем

$$C_n = D_n + \sum_{m=1}^{n-1} D_m C_{n-m}.$$

У любого вектора $(x_1; x_2; \dots; x_{2n})$, удовлетворяющего условиям

$$\sum_{i=1}^k x_i > 0 \quad \text{для всех } 1 \leq k \leq 2n-1, \quad \sum_{i=1}^{2n} x_i = 0$$

первая координата обязательно равна $+1$, а последняя -1 . Если отбросить первую и последнюю координату, то мы обнаружим, что оставшийся вектор удовлетворяет тем условиям, которые были положены при определении числа Каталана для пространства размерности $2(n-1)$. Число таких векторов равно C_{n-1} . Следовательно, $D_n = C_{n-1}$. Тогда имеем

$$\begin{aligned} C_n &= D_n + \sum_{m=1}^{n-1} D_m C_{n-m} = C_{n-1} + \sum_{m=1}^{n-1} C_{m-1} C_{n-m} = \\ &= C_{n-1} C_0 + \sum_{m=1}^{n-1} C_{m-1} C_{n-m} = \sum_{m=1}^n C_{m-1} C_{n-m}. \end{aligned}$$

Мы получили рекуррентное соотношение для определения чисел Каталана, которое запишем в виде

$$C_n = \sum_{m=0}^{n-1} C_m C_{n-m-1}.$$

Рассмотрим производящую функцию

$$G(z) = \sum_{n=0}^{\infty} C_n z^n.$$

Из рекуррентного соотношения получаем

$$\sum_{n=1}^{\infty} C_n z^n = \sum_{n=1}^{\infty} \left(\sum_{m=0}^{n-1} C_m C_{n-m-1} \right) z^n,$$

заменив в правой части $n - 1$ на n , имеем

$$\sum_{n=1}^{\infty} C_n z^n = z \sum_{n=0}^{\infty} \left(\sum_{m=0}^n C_m C_{n-m} \right) \cdot z^n.$$

Последнее соотношение преобразуется к виду

$$G(z) - 1 = z G^2(z),$$

решая которое, получаем

$$G(z) = \frac{1}{2z} \left(1 - (1 - 4z)^{1/2} \right).$$

Разлагая $G(z)$ в ряд с использованием формулы для биномиальных коэффициентов и выделяя коэффициент при z^n , имеем

$$C_n = \frac{1}{2} (-1)^n C\left(\frac{1}{2}; n+1\right) \cdot 4^n.$$

После некоторых преобразований (предлагаем их самостоятельно провести читателям) получаем

$$C_n = \frac{1}{n+1} C_{2n}^n.$$

Задачи и упражнения к главе 5

5.1. Найти последовательности, соответствующие данным производящим функциям $G(z)$:

а) $G(z) = \frac{1}{4-z^2};$

б) $G(z) = \frac{1}{9+z^2};$

в) $G(z) = \frac{z}{z^2-6z+8};$

г) $G(z) = \frac{1}{(z-1)^2(2z+1)}.$

5.2. С помощью производящих функций решить следующие рекуррентные соотношения:

а) $\begin{cases} a_0 = 2; \\ a_n = 3a_{n-1} - 2n + 3, & n > 0. \end{cases}$

б) $\begin{cases} a_0 = 2; & a_1 = 6; \\ a_n = 6a_{n-1} - 8a_{n-2} + 3n^2 - 23n + 36, & n > 1. \end{cases}$

5.3. Имеются числа 1, 2, 3, 4, 5. Сколькими способами, суммируя эти числа, можно получить 6, если каждое слагаемое можно использовать не более одного раза? Порядок суммирования не играет роли.

- 5.4.** Найдите сумму $\sum_{k=1}^n (2n - 2k)$.
- 5.5.** Найдите сумму $\sum_{k=1}^n \frac{1}{k(k+1)}$.
- 5.6.** Найдите точную асимптотическую оценку для суммы $\sum_{k=1}^n \sqrt{k}$.
- 5.7.** Найдите точную асимптотическую оценку для суммы $\sum_{k=1}^n \frac{1}{\sqrt{k}}$.
- 5.8.** Покажите, что $\sum_{k=1}^n \frac{1}{\sqrt{k^3}} = O(1)$.

Список литературы к главе 5

- 5.1. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. — М.: МЦНМО, 1999. — 960 с.
- 5.2. Грин Д., Кнут Д. Математические методы анализа алгоритмов. — М.: Мир, 1987. — 120 с.
- 5.3. Андерсон Дж. Дискретная математика и комбинаторика: Пер. с англ. — М.: Изд. дом «Вильямс». 2003. — 960 с.
- 5.4. Грэхем Р., Кнут Д., Паташник О. Конкретная математика. Основание информатики: Пер. с англ. — М.: Мир, 1998. — 703 с.
- 5.5. Хаггарти Р. Дискретная математика для программистов. — М.: Техносфера, 2005. — 400 с.
- 5.6. Ландо С.К. Лекции о производящих функциях. — 2-е изд., испр. — М.: МЦНМО, 2004. — 144 с.
- 5.7. Сачков В.Н. Введение в комбинаторные методы дискретной математики. — М.: Наука. Главная редакция физико-математической литературы, 1982. — 384 с.

Глава 6

МЕТОДЫ ТЕОРЕТИЧЕСКОГО АНАЛИЗА РЕСУРСНОЙ ЭФФЕКТИВНОСТИ РЕКУРСИВНЫХ АЛГОРИТМОВ

Введение. Формально первым этапом теоретического анализа алгоритмов является введение модели вычислений, в базовых операциях которой определяется функция трудоемкости. Предполагая, что средой реализации рекурсивных алгоритмов является процедурный язык высокого уровня, мы вводим такие базовые операции, предварительно определяя объектный и алгоритмический базисы. Активное использование рекурсивными алгоритмами области программного стека и многократные рекурсивные обращения, порождающие дерево рекурсии, обуславливают ряд специфических отличий анализа временной и емкостной эффективности рекурсивных алгоритмов в сравнении с методами анализа итерационных алгоритмов. Мы описываем в этой главе в общем виде два специальных метода анализа — метод рекуррентных соотношений и метод подсчета вершин дерева рекурсии, примеры применения которых приводятся в главе 7. Мы кратко останавливаемся также на некоторых подходах к повышению ресурсной эффективности рекурсивных алгоритмов, основанных на исследовании деревьев рекурсии.

§ 1. Базовые операции процедурного языка высокого уровня и методика анализа основных алгоритмических конструкций

Объектный и алгоритмический базисы. Использование в качестве модели вычислений классических алгоритмических формализмов, таких как машина Тьюринга, машина Поста, нормальные алгоритмы (алгорифмы) Маркова или схема алгоритмов Янова — сопряжено с трудностями преодоления «семантического разрыва» между этими моделями и языком реализации алгоритма. Наиболее близкой к реальному компьютеру является модель вычислений в виде машины с произвольным доступом к памяти, для которой, тем не менее, необходим также формальный переход к процедурному языку высокого уровня. Такой переход может быть основан на введении понятий объектного и алгоритмического базисов.

По отношению к процессу разработки алгоритма решения некоторой задачи мы можем говорить о существовании следующих трех компонентов, определяющих базис модели вычислений [6.1]:

— множество объектов Σ , над которыми производятся действия алгоритма, включающее множество исходных объектов D_A , множество промежуточных объектов D_P и множество результатов D_R :

$$\Sigma = D_A \cup D_P \cup D_R;$$

— устройство, производящее действия над объектами (элементами множества Σ), — процессор R — механизм реализации;

— конечное множество элементарных операций процессора R над элементами множества Σ — множество C , отражающее изначально заданную способность процессора R выполнять операции над объектами в модели вычислений.

Все эти три компонента по отношению к процессу разработки алгоритма решения задачи являются априорными, что влечет за собой следующее определение объектного базиса [6.1]: *объектным базисом* B_R будем называть тройку, состоящую из множества объектов, процессора и множества выполняемых им элементарных операций:

$$B_R = \{ \Sigma, R, C \}.$$

Таким образом, объектный базис ассоциирован с выбранной моделью вычислений в части, определяющей трудоемкость алгоритма. Собственно процесс разработки алгоритма, являясь очевидно интеллектуальным, состоит в определении конечной последовательности элементарных операций процессора R , приводящих за конечное время к решению поставленной задачи. Необходимость записи алгоритма приводит к введению двух специальных систем обозначений, а именно:

— системы обозначений для элементов множества элементарных операций процессора R , заданной множеством S , для элементов которого установлено некоторое соответствие с элементами множества C или его ограниченным подмножеством. Заметим, что для одной и той же элементарной операции из C можно предложить несколько разных обозначений. Например, операция «сложить» может быть обозначена как «+» или «add». Одно из таких обозначений выбирается как предпочтительное. Принципиально возможно установление соответствия некоторого обозначения и конечной последовательности операций объектного базиса;

— системы обозначений для записи последовательности элементарных операций, более корректно — для записи основных алгоритмических конструкций (следование, ветвление, цикл) с использованием операций управления из C , заданной множеством T . Отметим, что в реальной алгоритмической практике используются различные системы обозначения последовательности действий.

Поскольку выбор систем обозначений операций и их последовательности не влияет на объектный базис, но определяет запись алгоритма, предлагается следующее определение алгоритмического базиса [6.1]: *алгоритмическим базисом* B_A для заданного объектного базиса B_P будем называть двойку, включающую в себя конечное множество обо-

значений элементарных операций S и конечное множество обозначений алгоритмических конструкций T :

$$B_A = \{ S, T \}.$$

Отметим, что поскольку алгоритмический базис есть только система записи, не влияющая на существование (идею) алгоритма, то различные алгоритмические базисы в этом смысле эквивалентны. Выбор того или иного алгоритмического базиса определяется в основном требованиями наглядности или удобочитаемости записи алгоритма.

По отношению к функции трудоемкости понятие эквивалентности базисов нуждается в уточнении. Это связано с количеством операций принятой модели вычислений (объектного базиса), которому соответствует выбранное обозначение в алгоритмическом базисе. В этом смысле мы вправе потребовать, чтобы обозначение в алгоритмическом базисе приводило не более чем к $O(1)$ операций в модели вычислений, не зависящих от количества и значений входных данных. Будем говорить, что алгоритмический и объектный базисы по функции трудоемкости $O(1)$ — *эквивалентны*, если

$$\begin{cases} \forall s \in S : s \mapsto c' \subset C^k, k : k \geq 1, k = O(1); \\ \forall t \in T : t \mapsto c' \subset C^k, k : k \geq 1, k = O(1). \end{cases}$$

Введенная $O(1)$ — эквивалентность гарантирует, что если элементарные операции модели вычислений выполняются за единичное время, то и операции алгоритмического базиса будут выполняться за единичное время с точностью до постоянного множителя. При этом алгоритм решения некоторой задачи в объектном базисе $\{ \Sigma, R, C \}$ представляет собой записанную в алгоритмическом базисе $\{ S, T \}$ конечную последовательность элементарных операций процессора R , приводящую к решению поставленной задачи за конечное время.

Базовые операции в процедурном языке высокого уровня. Переход от алгоритмического базиса машины с произвольным доступом к памяти к алгоритмическому базису процедурного языка высокого уровня связан с тем, что классический процедурный (императивный) подход остается в настоящее время наиболее широко используемым при программной реализации алгоритмов. Практически значимые реализации вычислительных алгоритмов включают в себя обычно следующие программные фрагменты:

- диалоговый или файловый ввод исходных данных;
- проверка исходных данных на допустимость;
- собственно решение поставленной задачи;
- представление (вывод) полученных результатов.

В рамках сравнительного анализа ресурсной эффективности вычислительных алгоритмов мы можем считать, что фрагменты программной реализации, обслуживающие непосредственный алгоритм решения задачи, а именно фрагменты ввода, проверки и вывода, являются общими

или эквивалентными для разных алгоритмов решения данной задачи. Такой подход приводит к необходимости сравнительного анализа только алгоритмов непосредственного решения задачи. При этом предполагается размещение исходных данных и результатов в «оперативной» памяти, включая в это понятие как собственно оперативную память, так и кэш-память, регистры и буферы реального процессора. Таким образом, множество элементарных операций, учитываемых в функции трудоемкости, не включает операции ввода/вывода данных на внешние носители.

Для алгоритмического базиса формального процедурного языка высокого уровня такими базовыми операциями, коррелированными с основными операциями языков процедурного программирования, будем считать следующие [6.2]:

- простое присваивание: $a \leftarrow b$;
- одномерная индексация: $a[i]$: (адрес $(a) + i \cdot (\text{длина элемента})$);
- арифметические операции: $\{ *, /, -, + \}$;
- операции сравнения: $a \{ <, >, =, \leq, \geq \} b$;
- логические операции: $(l1) \{ \text{or, and, not} \} (l2)$;
- операции взятия содержимого по адресу (штрих-операция) и адресации в сложных типах данных (*name1.name2*);
- операции обслуживания программного стека.

По отношению к введенному набору базовых операций для алгоритмического базиса формального процедурного языка высокого уровня необходимо сделать несколько замечаний:

— опираясь на идеи структурного программирования, из набора элементарных операций исключается команда перехода, поскольку ее можно считать связанный с операцией сравнения в конструкции ветвления или цикла по условию. Такое исключение оправдано запретом использования оператора перехода на метку в идеологии структурного программирования;

— операции доступа к простым именованным ячейкам памяти считаются связанными с операциями, операнды которых они хранят;

— операции индексации элементов массивов и сложной адресации в типах данных вынесены в отдельные элементарные операции с целью возможного согласования временных оценок программных реализаций;

— конструкции циклов не рассматриваются, т. к. могут быть сведены к указанному набору элементарных операций.

С учетом этих замечаний можно говорить, что введенный алгоритмический базис является по трудоемкости $O(1)$ — эквивалентным машине с произвольным доступом к памяти как выбранной модели вычислений. Таким образом, полученная в данном алгоритмическом базисе функция трудоемкости алгоритма будет отличаться от значения функции нумерации в машине с произвольным доступом к памяти не более чем на постоянный множитель.

Методика анализа трудоемкости основных алгоритмических конструкций. Основными алгоритмическими конструкциями в процес-

дурном подходе являются конструкции следования, ветвления и цикла. Получение функции трудоемкости некоторого алгоритма основано на учитываемых базовых операциях процедурного языка высокого уровня и методике анализа основных алгоритмических конструкций. Отметим, что для получения функций трудоемкости для лучшего, среднего и худшего случаев при фиксированной размерности задачи, если алгоритм не принадлежит классу N , особенности анализа алгоритмических конструкций, трудоемкость которых зависит от данных («ветвление» и «цикл по условию») будут различны. С учетом введенных базовых операций методика анализа трудоемкости основных алгоритмических конструкций в процедурной реализации алгоритмов для общего случая сводится к следующим положениям [6.1].

Конструкция «Следование». Трудоемкость конструкции есть сумма трудоемкости блоков, следующих друг за другом

$$f_{\text{следование}} = f_1 + \dots + f_k, \quad (6.1.1)$$

где k — количество блоков в конструкции «Следование». Отметим, что трудоемкость самой конструкции не зависит от данных, при этом очевидно, что блоки, связанные конструкцией «Следование», могут обладать трудоемкостью, сильно зависящей от данных.

Конструкция «Ветвление». Запишем общий вид конструкции «Ветвление», характерный для процедурных языков высокого уровня:

If (l)
then

блок с трудоемкостью f_{then} , выполняемый с вероятностью p

else

блок с трудоемкостью f_{else} , выполняемый с вероятностью $(1-p)$

В этой записи (l) обозначает логическое выражение, состоящее из логических переменных и/или арифметических, символьных или других по типу сравнений с трудоемкостью вычисления f_l . Вероятности перехода на соответствующие блоки могут меняться как в зависимости от данных, так и в зависимости от параметров внешних охватывающих циклов и/или других условий. Достаточно трудно дать какие-либо общие рекомендации для получения p вне зависимости от конкретного алгоритма или особенностей входа. Общая трудоемкость конструкции «Ветвление» в среднем случае требует анализа для получения вероятности p выполнения переходов на блоки «**then**» и «**else**». При известном значении вероятности p трудоемкость конструкции определяется как

$$f_{\text{ветвление}} = f_l + f_{\text{then}}p + f_{\text{else}}(1-p), \quad (6.1.2)$$

где f_l — трудоемкость вычисления условия (l). В общем случае вероятность перехода p есть функция исходных данных и связанных с ними промежуточных результатов $p = p(D)$. Очевидно, что для анализа худшего случая может быть выбран тот блок ветвления, который имеет

большую трудоемкость, а для лучшего случая — блок с меньшей трудоемкостью.

Конструкция «Цикл по счетчику». Стандартная запись конструкции и ее эквивалент, записанный в виде цикла с нижним условием, имеют вид:

For $i \leftarrow 1$ **to** n
 тело цикла
end For

$i \leftarrow 1$
тело цикла
 $i \leftarrow i + 1$
until $i \leq n$

После сведения конструкции к введенным базовым операциям ее трудоемкость определяется следующим образом:

$$f_{\text{цикл}} = 1 + 3n + n f_{\text{тело цикла}}. \quad (6.1.3)$$

Формула (6.1.3) отражает инициализацию счетчика ($i \leftarrow 1$) и три операции, собственно организующие цикл ($i \leftarrow i + 1$, $\text{until } i \leq n$). Анализ вложенных циклов по счетчику с независимыми индексами не составляет труда и сводится к погружению трудоемкости цикла в трудоемкость тела охватывающего его цикла. Для k вложенных зависимых циклов трудоемкость определяется в виде вложенных сумм с зависимыми индексами, методика вычисления которых подробно обсуждается, например, в [6.3].

Конструкция «Цикл по условию». Конкретная реализация цикла по условию (цикл с верхним или нижним условием) не меняет методику оценки его трудоемкости. На каждом проходе выполняется оценка условия и, может быть, изменение каких-либо переменных, влияющих на значение этого условия. Для худшего случая могут быть использованы верхние граничные оценки. Общие рекомендации по определению суммарного количества проходов цикла крайне затруднительны из-за сложных зависимостей от исходных данных. Так, например, для задачи решения системы линейных уравнений итерационными методами количество итераций по точности (сходимость) определяется собственными числами исходной матрицы, трудоемкость вычисления которых сопоставима по трудоемкости с получением самого решения [6.4].

На основе методики анализа основных алгоритмических конструкций и введенных базовых операций процедурного языка высокого уровня может быть сформулирован общий метод получения функции трудоемкости, который включает в себя следующие этапы.

1. Декомпозиция алгоритма — выделение структурных частей алгоритма в виде базовых алгоритмических конструкций, связанных следованием, вплоть до построчной детализации.

2. Построчный анализ трудоемкости по базовым операциям процедурного языка высокого уровня. При этом возможны два варианта такого построчного анализа — совокупный анализ, когда учитывается общее количество базовых операций, и пооперационный анализ, при котором можно получить функции трудоемкости для каждой из базовых операций.

3. Обратную композицию функции трудоемкости на основе методики анализа основных алгоритмических конструкций и известных методов анализа алгоритмов — вероятностного анализа, амортизационного анализа, методов оценки вычислительной сложности алгоритмов, с учетом типа получаемой функции трудоемкости — для лучшего, худшего или среднего случая.

В простейшем случае композиция функции трудоемкости — это аддитивное объединение трудоемкости структурных частей алгоритма, связанных конструкцией следования. Ниже мы покажем, что для рекурсивных алгоритмов такая композиция требует анализа порожденной цепочки рекурсивных вызовов.

§ 2. Особенности анализа временной и емкостной эффективности рекурсивных алгоритмов

Основной особенностью анализа ресурсной эффективности рекурсивных алгоритмов является необходимость учета дополнительных затрат памяти и трудоемкости, связанных с механизмом организации рекурсии. Получение ресурсных функций вычислительных алгоритмов в рекурсивной реализации базируется как на методах оценки вычислительной сложности собственно тела рекурсивного алгоритма, так и на способах учета ресурсных затрат на организацию рекурсии и детальном анализе цепочки рекурсивных вызовов и возвратов — порожденного дерева рекурсии. Трудоемкость рекурсивных реализаций алгоритмов, очевидно, связана как с количеством операций, выполняемых при одном вызове рекурсивной функции, так и с количеством таких вызовов. Должны быть учтены также и затраты на возврат вычисленных значений и передачу управления в точку вызова. Эти операции должны быть включены в функцию трудоемкости рекурсивно реализованного алгоритма. Можно заметить, что некоторая ветвь дерева рекурсивных вызовов обрывается при достижении такого значения передаваемого параметра, при котором функция может быть вычислена непосредственно. Таким образом, рекурсия в этом смысле эквивалентна конструкции цикла, в котором каждый проход цикла есть выполнение рекурсивной функции с заданным параметром.

Анализ трудоемкости вызова рекурсивной функции. Механизм вызова функции или процедуры в языке высокого уровня существенно зависит от архитектуры компьютера и операционной системы. В рамках архитектуры и операционных систем IBM PC совместимых компьютеров этот механизм реализован через программный стек. Как передаваемые в процедуру или функцию фактические параметры, так и возвращаемые из них значения помещаются в программный стек специальными командами процессора — мы считаем эти операции базовыми в нашей модели вычислений (см. параграф 6.1). Модель

механизма вызова-возврата, использующая программный стек, была подробно описана в главе 2.

Для подсчета трудоемкости вызова в базовых операциях обслуживания стека напомним, что при вызове процедуры или функции в стек помещается адрес возврата, состояние необходимых регистров процессора, состояние локальных ячеек вызывающей процедуры или функции, адреса возвращаемых значений и передаваемые параметры. После этого выполняется переход по адресу на вызываемую процедуру, которая извлекает переданные фактические параметры, выполняет вычисления и помещает результаты по указанным в стеке адресам. При завершении работы вызываемая процедура восстанавливает регистры, локальные ячейки, выталкивает из стека адрес возврата и осуществляется переход по этому адресу.

Для анализа трудоемкости механизма вызова-возврата для рекурсивной процедуры введем следующие обозначения:

p — количество передаваемых фактических параметров,

r — количество сохраняемых в стеке регистров,

k — количество возвращаемых по адресной ссылке значений,

l — количество локальных ячеек процедуры.

Поскольку каждый объект в некоторый момент помещается в стек и в какой-то момент выталкивается из него, то трудоемкость рекурсивной процедуры на один вызов-возврат $f_R(1) = 2(p + r + k + l + 1)$ в базовых операциях составит:

$$f_R(1) = 2(p + k + r + l + 1). \quad (6.2.1)$$

Дополнительная единица в формуле (6.2.1) учитывает операции с адресом возврата. Для рекурсивной функции мы должны дополнительно учесть еще одну локальную ячейку, через которую передается значение функции. Мы обозначим этот параметр через f , считая, что его значение равно единице, тогда трудоемкость на один вызов-возврат рекурсивной функции может быть определена следующим образом:

$$f_R(1) = 2(p + k + r + f + l + 1), \quad f = 1. \quad (6.2.2)$$

Анализ трудоемкости рекурсивных алгоритмов в части совокупной трудоемкости самого рекурсивного вызова-возврата можно выполнять разными способами в зависимости от того, как формируется итоговая сумма базовых операций — либо отдельно по цепочкам рекурсивных вызовов и возвратов, либо совокупно по вершинам дерева рекурсии. Формулы (6.2.1) и (6.2.2) отражают второй способ, заметим также, что если положить $f = 0$ для рекурсивной процедуры в формуле (6.2.2), то она сводится к (6.2.1).

Учет особенностей рекурсивной реализации в функциях ресурсной эффективности программных реализаций алгоритмов. Суммарные ресурсы, требуемые рекурсивной реализацией алгоритма, могут быть разделены на ресурсы, собственно связанные с решением задачи, и ресурсы, необходимые для организации рекурсии. При срав-

нительном анализе итерационной и рекурсивной реализации можно отметить, что особенность последней состоит как в наличии дополнительных операций, организующих рекурсию, так и дополнительных затрат оперативной памяти в области программного стека для хранения информации о цепочке рекурсивных вызовов. При этом отметим, что эти затраты в ряде случаев оправданы, например, рекурсивные алгоритмы, использующие метод декомпозиции, позволяют получить асимптотически более эффективные алгоритмы для целого ряда задач [6.5, 6.6, 6.7]. Эта эффективность достигается в частности и за счет большего объема памяти в области программного стека, что должно быть обязательно учтено в комплексном критерии оценки качества алгоритмов.

Оценка требуемой памяти в стеке может быть получена следующим образом: поскольку рекурсивные вызовы обрабатываются последовательно, то во временной динамике в области стека хранится не фрагмент дерева рекурсии, а только текущая цепочка рекурсивных вызовов — унарный фрагмент этого дерева. Из этого следует, что требуемый объем памяти в области программного стека определяется не общим количеством вершин дерева рекурсии, а максимальной глубиной его листьев. Очевидно, что вид и глубина рекурсивного дерева определяются как особенностями самого алгоритма, так и характеристиками множества исходных данных. Обозначив через $H_R(D)$ максимальную глубину рекурсивного дерева, порожденного данным алгоритмом на данном входе D , можно оценить требуемый объем программного стека, опираясь на реализацию механизма вызова функции. Предполагая, в худшем случае, что параметры, передаваемые через стек, сохраняются в нем, максимальный объем памяти в области стека может быть определен на основе (6.2.2) следующим образом:

$$V_{st}(D) = H_R(D)(p + r + k + f + l + 1)l_w, \quad (6.2.3)$$

где l_w — длина слова в байтах.

Отметим также, что, в отличие от объема памяти в области программного стека, требуемого для организации рекурсии, который зависит от максимальной глубины рекурсивного дерева, количество операций со стеком на один вызов-возврат, задаваемое формулой (6.2.2), должно быть учтено в функции трудоемкости для всех вызовов. Таким образом, получение функции трудоемкости в рекурсивной реализации требует определения общего количества вершин рекурсивного дерева. Если структура дерева такова, что оно является не только глубоким, но и «широким», то совокупные затраты на организацию рекурсии могут быть значительны. Трудоемкость тела рекурсивной функции или процедуры может быть получена на основе методов анализа итерационных алгоритмов (см., например, [6.1, 6.2]). Однако общая трудоемкость определяется числом порожденных вершин дерева рекурсии, кроме того, мы должны учесть, что в общем случае трудоемкость в разных

вершинах может различаться и зависеть от данных или параметров рекурсии.

§ 3. Анализ трудоемкости методом подсчета вершин дерева рекурсии

При теоретическом построении ресурсных функций рекурсивного алгоритма необходимо учесть ряд ресурсных затрат и особенностей рекурсивной реализации, а именно:

— ресурсные затраты на обслуживание рекурсивных вызовов-возвратов, передачу параметров и возврат значений рекурсивных функций (ресурсные затраты обслуживания рекурсии);

— специфику фрагмента останова рекурсии, приводящую к необходимости отдельного учета ресурсных затрат в листьях дерева рекурсии.

Учет этих особенностей при теоретическом анализе рекурсивных алгоритмов приводит к необходимости получить функциональные зависимости общего количества вершин дерева рекурсии и количества его внутренних вершин и листьев от характеристик множества входных данных. Если мы можем определить ресурсные затраты в каждой вершине дерева, то, суммируя, мы получим ресурсную функцию алгоритма в целом. Такой подход мы будем называть в дальнейшем получением ресурсных функций для рекурсивных реализаций алгоритмов *методом подсчета вершин дерева рекурсии* [6.2].

Анализ дерева рекурсии. Первым этапом метода является исследование дерева рекурсии. В предположении, что n — длина входа алгоритма для классов $N, PR; m_1, \dots, m_k$ — значение параметров входа для классов PR, NPR ($k \leq n$), а D — конкретный вход алгоритма, введем следующие обозначения для характеристик дерева рекурсии, порожденного рекурсивным алгоритмом:

$R(D)$ — общее количество вершин дерева рекурсии для входа D ;

$R_V(D)$ — количество внутренних вершин дерева для входа D ;

$R_L(D)$ — количество листьев дерева рекурсии для входа D ;

$H_R(D)$ — максимальная глубина дерева рекурсии (максимальное по всем листьям дерева количество вершин в пути от корня дерева до листа), тогда очевидно, что справедливы соотношения

$$R(D) = R_V(D) + R_L(D), \quad H_R(D) \leq R_V(D) + 1.$$

Рассмотрим более подробно особенности функциональной зависимости общего числа вершин в дереве рекурсии для алгоритмов, принадлежащих различным трудоемкостным классам по характеристическим особенностям множества исходных данных. В соответствии с определением классов (см. главу 2) имеем:

1) класс N : $R(D) = R(n) \quad \forall D \in D_n$;

2) класс PR : $R(D) = R(m_1, \dots, m_k)$;

3) класс NPR : $R(D) = R(n, m_1, \dots, m_k)$.

Таким образом, основная задача при использовании этого метода состоит в теоретическом построении функций $R_V(D)$, $R_L(D)$ и $H_R(D)$ как функций характеристик множества входных данных в зависимости от принадлежности алгоритма к одному из основных классов. Дополнительный интерес, в смысле анализа дерева рекурсии, представляет информация об отношении количества листьев к общему количеству вершин рекурсивного дерева. Это характеристика относительной «ширины» нижнего уровня (уровня листьев) в дереве рекурсии

$$B_L(D) = \frac{R_L(D)}{R(D)}, \quad 0 < B_L(D) < 1. \quad (6.3.1)$$

Значение $B_L(D)$ будет минимально для цепочки (унарного дерева) и будет возрастать при увеличении числа вершин, порожденных во внутренних вершинах дерева рекурсии, т. е. с ростом «арности» дерева. Напомним, что дерево, каждая вершина которого порождает две новых вершины, называется полным бинарным деревом, а порождение трех вершин приводит к тернарному дереву.

Рассмотрим более подробно, как на основе функций $R_V(D)$ и $R_L(D)$ возможно теоретическое построение ресурсных функций рекурсивных алгоритмов.

Получение функции трудоемкости методом подсчета вершин дерева рекурсии. Для рекурсивных алгоритмов трудоемкость решения конкретной задачи включает в себя не только трудоемкость непосредственной обработки данных в теле рекурсивной функции, но и затраты на организацию рекурсии. Более точно, трудоемкость $f_A(D)$ алгоритма A на конкретном входе D определяется трудоемкостью обслуживания дерева рекурсии, зависящей от общего количества его вершин, и трудоемкостью продуктивных вычислений, выполненных во всех вершинах дерева рекурсии. В связи с этим обозначим через

$f_R(D)$ — трудоемкость порождения и обслуживания дерева рекурсии,

$f_C(D)$ — трудоемкость продуктивных вычислений алгоритма, и с учетом введенных обозначений получаем

$$f_A(D) = f_R(D) + f_C(D). \quad (6.3.2)$$

Трудоемкость обслуживания дерева рекурсии может быть вычислена достаточно просто, а именно, если функция $R(D)$ известна, и на обслуживание одного рекурсивного вызова затрачивается фиксированное количество базовых операций (см. 6.2) $f_R(1)$, определяемых, например, по формуле (6.2.2), то

$$f_R(D) = R(D) f_R(1). \quad (6.3.3)$$

При подсчете трудоемкости продуктивных вычислений необходимо учесть, что для листьев рекурсивного дерева алгоритм будет выполнять непосредственное вычисление значений, и эта трудоемкость отлична от

трудоемкости во внутренних вершинах, следовательно, трудоемкость алгоритма при останове рекурсии должна быть учтена отдельно. В связи с этим обозначим через

$f_{CV}(D)$ — трудоемкость продуктивных вычислений во внутренних вершинах,

$f_{CL}(D)$ — трудоемкость продуктивных вычислений в листьях дерева рекурсии, это приводит к определению $f_C(D)$ в виде

$$f_C(D) = f_{CV}(D) + f_{CL}(D). \quad (6.3.4)$$

Обозначим через $f_{CL}(1)$ трудоемкость алгоритма при останове рекурсии. Заметим, что, как правило, значение $f_{CL}(1)$ может быть сравнительно легко получено, т. к. выражается, как правило, фиксированным числом базовых операций. Зная количество листьев рекурсивного дерева, можно определить $f_{CL}(D)$:

$$f_{CL}(D) = R_L(D) f_{CL}(1). \quad (6.3.5)$$

Во внутренних вершинах дерева рекурсии выполняются некоторые действия, связанные с подготовкой параметров следующих рекурсивных вызовов и обработкой возвращаемых результатов. Трудоемкость такой обработки может зависеть как от обрабатываемых в этой вершине данных, так и от положения вершины в дереве рекурсии. С целью учета этой зависимости введем нумерацию внутренних вершин, начиная с корня, по уровням дерева. Заметим, что число уровней внутренних вершин в дереве на единицу меньше глубины рекурсии $H_R(D)$. Пусть m есть номер уровня, $m = \overline{1, H_R(D) - 1}$, а k — номер вершины на уровне, $k = \overline{1, K(m)}$, где $K(m)$ — количество внутренних вершин на уровне m , заметим, что неполное дерево на уровне k может содержать как внутренние вершины, так и листья. С учетом такой нумерации обозначим вершины дерева через

$$v_{mk}, \quad m = \overline{1, H_R(D) - 1}; \quad k = \overline{1, K(m)},$$

при этом очевидно, что

$$\sum_{m=1}^{H_R(D)-1} \sum_{k=1}^{K(m)} 1 = R_V(D).$$

Обозначим трудоемкость продуктивных вычислений в вершине v_{mk} через $f_{CV}(v_{mk})$, тогда формула для трудоемкости продуктивных вычислений во внутренних вершинах дерева рекурсии имеет вид

$$f_{CV}(D) = \sum_{m=1}^{H_R(D)-1} \sum_{k=1}^{K(m)} f_{CV}(v_{mk}). \quad (6.3.6)$$

Заметим, что в случае, когда значения функции $f_{CV}(v_{mk})$ не зависят от номера вершины дерева рекурсии, т. е. трудоемкость продуктивных вычислений в вершинах не зависит от данных, то, обозначая

трудоемкость продуктивных вычислений для любой внутренней вершины дерева через $f_{CV}(v)$, имеем

$$f_{CV}(D) = R_V(D) f_{CV}(v). \quad (6.3.7)$$

Подставляя полученные результаты в формулу (6.3.2), окончательно получаем формулу для определения трудоемкости рекурсивного алгоритма на основе метода подсчета вершин дерева рекурсии в общем случае

$$f_A(D) = R(D) f_R(1) + R_L(D) f_{CL}(1) + \sum_{m=1}^{H_R(D)-1} \sum_{k=1}^{K(m)} f_{CV}(v_{mk}), \quad (6.3.8)$$

и в частном случае, когда трудоемкость продуктивных вычислений для любой внутренней вершины дерева рекурсии одинакова:

$$f_A(D) = R(D) f_R(1) + R_L(D) f_{CL}(1) + R_V(D) f_{CV}(v). \quad (6.3.9)$$

По аналогии с характеристикой $B_L(D)$ можно рассмотреть дополнительную характеристику трудоемкости рекурсивного алгоритма — долю операций обслуживания дерева рекурсии. Обозначая ее через $F_R(D)$, имеем

$$F_R(D) = \frac{f_R(D)}{f_A(D)}, \quad 0 < F_R(D) < 1. \quad (6.3.10)$$

Значение $F_R(D)$ показывает, насколько трудоемкость обслуживания дерева рекурсии значима в общей трудоемкости рекурсивного алгоритма.

Сформулируем этапы анализа трудоемкости рекурсивного алгоритма методом подсчета вершин дерева рекурсии.

1. Анализ порождаемого данным алгоритмом дерева рекурсии с целью получения теоретических зависимостей для характеристик дерева — $R(n, m_1, \dots, m_k)$, $R_L(n, m_1, \dots, m_k)$, $R_V(n, m_1, \dots, m_k)$, $H_R(n, m_1, \dots, m_k)$ как функций длины входа (n) и/или характеристических особенностей множества входных данных.

2. Определение трудоемкости обслуживания рекурсии на один вызов-возврат $f_R(1)$, например, по формуле (6.2.2).

3. Определение трудоемкости алгоритма при останове рекурсии $f_{CL}(1)$. Если останов происходит при нескольких значениях аргумента, и трудоемкость вычисления в разных листьях различна, то необходим более детальный подсчет, но уже с учетом типов листьев в дереве рекурсии.

4. Исследование трудоемкости продуктивных вычислений во внутренних вершинах дерева рекурсии — получение функции $f_{CV}(v_{mk})$.

5. Получение функции трудоемкости рекурсивного алгоритма по формулам (6.3.8) или (6.3.9) в зависимости от поведения функции $f_{CV}(v_{mk})$.

Некоторые замечания по поводу этого метода касаются особенностей анализа дерева рекурсии и функции $f_{CV}(v_{mk})$. Если анализируемое дерево является регулярным, то выполнение первого пункта метода не представляет особых трудностей. Однако для алгоритмов с сильной параметрической зависимостью структура дерева плохо поддается формализации, и одним из подходов к анализу таких деревьев является введение обобщающего параметра, приводящего к оценке характеристик дерева в среднем случае. Другая серьезная проблема анализа — получение функции $f_{CV}(v_{mk})$ в случае, если трудоемкость в вершине сильно зависит от данных. Ряд примеров анализа рекурсивных алгоритмов показывает, что для прямого определения $f_{CV}(v_{mk})$ приходится прибегать к достаточно тонким методам или специальной параметризации задачи. Одно из альтернативных решений состоит в том, чтобы, используя идеи амортизационного анализа [6.7], попытаться вместо аналитического задания функции $f_{CV}(v_{mk})$ получить ее сумму либо по уровням дерева, либо по всем внутренним вершинам. Примеры анализа трудоемкости рекурсивных алгоритмов с использованием этого метода и некоторых его модификаций мы приведем в главе 7.

§ 4. Анализ трудоемкости методом рекуррентных соотношений

Еще один специальный метод анализа трудоемкости рекурсивных алгоритмов основан на очевидном предположении о том, что если сам алгоритм имеет рекурсивную структуру, то и функция его трудоемкости может быть описана аппаратом теории рекурсии. Идея метода состоит в получении рекуррентного соотношения, основанного на результатах анализа алгоритма и задающего функцию его трудоемкости, и решения этого соотношения с помощью известных методов, например тех, которые мы изложили в главах 1 и 5. Мы не можем предложить универсальной формулы для этого метода, поскольку в каждом конкретном случае рекуррентное соотношение для функции трудоемкости отражает специфику рекурсивного алгоритма. Определенные обобщения могут быть сделаны для рекурсивных алгоритмов, разработанных по методу декомпозиции, и в ряде других частных случаев.

Рассмотрим вначале этот метод в приложении к анализу трудоемкости рекурсивных алгоритмов, построенных на основе метода декомпозиции. Напомним, что идея метода состоит в разделении задачи на части меньшей размерности, получении решений для выделенных частей и объединении решений при возврате из рекурсивных вызовов. Если в алгоритме при решении задачи размерности n происходит такое ее разделение, которое приводит к необходимости решения a подзадач размерности n/b (b является делителем n), то функция трудоемкости имеет вид

$$f_A(n) = a f_A(n/b) + d(n) + U(n), \quad (6.4.1)$$

где $d(n)$ — трудоемкость алгоритма разделения задачи на подзадачи, а $U(n)$ — трудоемкость дополнительного алгоритма, объединяющего полученные решения. При этом для некоторой малой размерности задачи, т. е. при $n = n_0$, возможно ее прямое (не рекурсивное) решение. Обозначив трудоемкость получения этого прямого решения через $f_A(n_0)$, мы получаем общую форму рекуррентного соотношения для функции трудоемкости алгоритмов, разработанных методом декомпозиции задачи:

$$\begin{cases} f_A(n_0), & n = n_0; \\ f_A(n) = a f_A(n/b) + d(n) + U(n), & n > n_0. \end{cases} \quad (6.4.2)$$

Дополнительное уточнение касается трудоемкости организации рекурсии, которая должна быть учтена на каждом рекурсивном вызове. Эта трудоемкость может быть получена по формуле (6.2.1), и, сохранив обозначение $f_R(1)$, получаем

$$\begin{cases} f_A(n_0) + f_R(1), & n = n_0; \\ f_A(n) = a f_A(n/b) + d(n) + U(n) + f_R(1), & n > n_0. \end{cases} \quad (6.4.3)$$

Еще один нюанс, серьёзно усложняющий получение решения, состоит в том, что размерность решаемой задачи должна быть целой. Поэтому в общем случае вместо n/b в качестве аргумента функции в (6.4.3) должна фигурировать целая часть частного с округлением вниз или вверх, т. е. $\lfloor n/b \rfloor$ или $\lceil n/b \rceil$.

Рассмотрим в качестве примера известный алгоритм сортировки слиянием [6.7]. На каждом рекурсивном вызове переданный массив делится пополам, что дает оценку для функции $d(n) = \Theta(1) = c_1$, далее рекурсивно вызывается сортировка полученных массивов половинной длины до тех пор, пока длина массива не станет равной единице. Массив единичной длины очевидно сортирован, и наши затраты состоят только в распознавании длины массива — мы затрачиваем на это c_2 базовых операций. Возвращенные отсортированные массивы объединяются с трудоемкостью $a n + b$, где коэффициенты определяются на основе анализа алгоритма слияния. Организация рекурсии на один вызов требует фиксированного числа операций $f_R(1) = c_3$, и, используя (6.4.3), мы получаем рекуррентное соотношение для функции трудоемкости алгоритма сортировки слиянием:

$$\begin{cases} c_2 + c_3, & n = 1; \\ f_A(n) = f_A(\lfloor n/b \rfloor) + f_A(\lceil n/b \rceil) + c_1 + a n + b + c_3, & n > 1. \end{cases}$$

Детальный анализ этого алгоритма мы проведем в главе 7, где и получим явную функцию его трудоемкости.

Рассмотрим еще одно обобщение для рекурсивных алгоритмов, пониждающих размерность задачи на единицу при каждой рекурсии. При этом трудоемкость фрагмента алгоритма, сводящего задачу размерности n к задаче размерности $n - 1$ будем обозначать через $f_{CV}(n)$ — в терминологии деревьев рекурсии это есть трудоемкость обработки

во внутренней вершине. Как и в предыдущем примере, будем предполагать, что останов рекурсии происходит при размерности $n = n_0$, достаточно часто при $n_0 = 1$. Сохраняя обозначения, введенные в 6.3 — $f_{CL}(1)$ для трудоемкости останова рекурсии ($f_{CL}(1) = f_A(n_0)$) и $f_R(1)$ для трудоемкости обслуживания рекурсии, получаем рекуррентное соотношение для $f_A(n)$:

$$\begin{cases} f_A(n_0) = f_R(1) + f_{CL}(1), & n = n_0; \\ f_A(n) = f_A(n-1) + f_R(1) + f_{CV}(n), & n > n_0. \end{cases} \quad (6.4.4)$$

Приведем пример использования (6.4.4) для построения рекуррентного соотношения, задающего функцию трудоемкости алгоритма вычисления определителя квадратной матрицы (см. параграф 2.1). Останов рекурсии происходит при $n = 2$ и требует фиксированного числа базовых операций — обозначим это число через c_1 . Трудоемкость организации рекурсии на один вызов также требует фиксированного числа операций $f_R(1) = c_2$. Трудоемкость сведения задачи вычисления определителя матрицы размерности n к задаче размерности $n - 1$ определяется числом элементов матрицы и может быть представлена в виде

$$f_{CV}(n) = a n^2 + b n + d,$$

где коэффициенты определяются путем анализа трудоемкости фрагмента сведения матрицы по методике из параграфа 6.1. Подставляя полученные результаты в (6.4.4), получаем рекуррентное соотношение, задающее функцию трудоемкости данного алгоритма:

$$\begin{cases} f_A(2) = c_1 + c_2; \\ f_A(n) = f_A(n-1) + a n^2 + b n + d + c_2, & n > 2. \end{cases}$$

Решение может быть получено методами, изложенными в главе 1, и мы оставляем его получение в качестве упражнения для наших читателей.

Ниже, в главе 7, мы продемонстрируем применение этого метода для анализа трудоемкости рекурсивных алгоритмов. Сравнивая оба изложенных метода анализа рекурсивных алгоритмов, можно сказать, что метод подсчета вершин дерева рекурсии позволяет получить более детальную информацию о ресурсных затратах, в частности выделить общие затраты на организацию рекурсии и трудоемкость в листьях рекурсивного дерева.

§ 5. Способы повышения ресурсной эффективности рекурсивных алгоритмов

Мы возвращаемся к вопросу, который обсуждали кратко еще во введении к этой книге — когда и при каких условиях рационально применять рекурсивные алгоритмы? Теперь мы можем поставить этот вопрос более корректно — какова ресурсная эффективность рекурсивного алгоритма в сравнении с ресурсной эффективностью другого

итерационного алгоритма решения данной задачи? Рассматривая полную функцию ресурсной эффективности, т. е. ее временную и емкостную составляющие, мы, в самом общем случае, можем сказать, что рекурсивный алгоритм требует значительно больших затрат памяти в области программного стека и дополнительных базовых операций на обслуживание механизма рекурсии. В обозначениях, введенных в параграфах 6.2 и 6.3, эти затраты задаются функцией $f_R(D)$, которая пропорциональна общему числу вершин дерева рекурсии $R(D)$, и функцией $V_{st}(D)$, значение которой зависит от глубины дерева рекурсии $H_R(D)$, где D — конкретный вход алгоритма. Мы также специально ввели характеристику удельного веса обслуживания рекурсии $F_R(D)$.

Улучшение ресурсной эффективности решения некоторой задачи с использованием рекурсивного алгоритма обуславливается самим методом его разработки. Уже известный читателям метод декомпозиции позволяет в целом ряде случаев достичь границы теоретического нижнего предела трудоемкости (в асимптотическом понимании). Примерами могут служить алгоритм сортировки слиянием и алгоритм быстрого преобразования Фурье [6.7]. Значительные улучшения асимптотических оценок трудоемкости позволяют получить как алгоритм Карацубы при умножении длинных целых чисел [6.8], так и алгоритм Штрассена при умножении квадратных матриц [6.7]. Естественно, получаемые этим методом рекурсивные алгоритмы, как правило, асимптотически лучше прямых итерационных методов, но за счет больших коэффициентов у компонента главного порядка функции трудоемкости рациональны в применении только для больших размерностей. Для метода динамического программирования (ДП) сравнение ресурсной эффективности рекурсивного и итерационного (этот реализация в методе ДП носит название табличного алгоритма) подхода не столь очевидна из-за сложной параметрической зависимости трудоемкости от особенностей входа, и требует специального анализа (см., например, [6.9]). Непосредственная рекурсивная реализация функционального уравнения Беллмана для решаемой задачи порождает, в большинстве случаев, не только глубокое, но и широкое (за счет цикла поиска максимума) дерево рекурсии, что влечет, очевидно, рост затрат на его обслуживание.

Если мы хотим, оставаясь в рамках рекурсивного подхода, улучшить ресурсные характеристики данного алгоритма, то поскольку и трудоемкость и емкостная эффективность непосредственно зависят от характеристик дерева рекурсии, то основной путь — это сокращение числа вершин дерева. В рамках конкретной задачи могут быть предложены специальные подходы, однако рекомендации общего вида сводятся всего к нескольким способам, два из которых мы и хотим кратко изложить.

Динамическое сохранение результатов. Этот способ состоит в том, что в динамике порождения дерева рекурсии при рекурсивных возвратах мы сохраняем вычисленные значения, предполагая, что в по-

следующих цепочках рекурсии эти значения могут быть использованы для останова в некоторой ветви, если результат для данного аргумента был уже ранее получен [6.7]. Посмотрите внимательно на дерево рекурсии, порождаемое алгоритмом вычисления чисел Фибоначчи — чем ниже уровень рекурсии, тем чаще встречаются вызовы с одинаковым аргументом. Если мы заведем специальный массив, в котором будем хранить уже вычисленные значения, то общее дерево рекурсии будет значительно сокращено. Увы, этот способ не всегда приводит к желаемому результату. Если дерево рекурсии широкое и содержит много вершин с редко повторяющимися аргументами вызова, то мы получим значительное увеличение требуемой дополнительной памяти при незначительном сокращении трудоемкости. Вообще емкостные затраты этого способа могут быть значительны, если мы используем прямые индексные массивы. Один из путей улучшения емкостной эффективности — создание хэш-таблиц и динамическое выделение памяти. Рекомендации по применению этого подхода должны основываться на тщательном анализе порожденного дерева в аспекте вероятности повторения рекурсивного вызова с данными аргументами и обязательной оценкой емкостной эффективности предлагаемой структуры данных для динамического хранения результатов.

Предвычисления в листьях. Для широкого дерева рекурсии, например тернарного, количество листьев в общем количестве вершин дерева превышает половину. Если мы можем так организовать рекурсивный процесс вычислений, что останов рекурсии будет происходить на один или несколько уровней выше, то трудоемкость такого алгоритма будет, скорее всего, меньше, чем при полном дереве рекурсии. Классический пример — алгоритм сортировки слиянием. Если останов рекурсии происходит при длине массива, равной двум или трём, то мы как минимум вдвое сокращаем число вершин дерева рекурсии, затрачивая некоторое количество операций на сортировку коротких массивов в листьях дерева. Нахождение оптимальной границы длины для останова рекурсии на основе анализа трудоемкости такого комбинированного алгоритма является хорошей исследовательской задачей для студентов. Проблемы этого способа начинаются там, где мы заранее не знаем аргументов, с которыми будут выполнены конкретные остановы рекурсии в листьях. Типичный пример — задача одномерной оптимальной упаковки, которую мы будем рассматривать в главе 7. Мы можем заранее вычислить все возможные результаты для аргументов останова рекурсии, но в этом случае полученный массив может иметь достаточно большую размерность. Мы снова сталкиваемся с классической дилеммой — сокращение трудоемкости за счет увеличения объема дополнительной памяти. Анализ ресурсной эффективности комбинированного алгоритма с целью определения уровня рекурсии, до которого рационально проводить предвычисления в листьях, также является интересной исследовательской задачей. Заметим, что такой способ часто

используется для алгоритмов, основанных на методе динамического программирования.

Ряд полезных сведений о подходах к совершенствованию рекурсивных алгоритмов читатели могут найти также в [6.6] и [6.7].

Задачи и упражнения к главе 6

6.1. Мы считаем, что все локально описанные ячейки и массивы должны быть сохранены в программном стеке при рекурсивном вызове. Как изменится формула (6.2.1), если одномерный массив из n элементов описан в некоторой процедуре как локальный?

6.2. Какое дерево рекурсии является более предпочтительным в смысле ресурсной эффективности рекурсивного алгоритма — унарное дерево, содержащее n вершин на n уровнях, или $(n - 1)$ -арное дерево, содержащее корень и $n - 1$ вершину на втором уровне?

6.3. Будем рассматривать полные m -арные деревья. Получите формулу зависимости относительной ширины уровня листьев m -арного дерева глубиной n , т. е. получите функцию $B_L(n, m)$ в явном виде, опираясь на ее определение — формулу (6.3.1).

6.4. Как Вы считаете, доля операций обслуживания дерева рекурсии, задаваемая формулой (6.3.10), будет расти или падать, если мы, зафиксировав общее количество вершин, будем менять ширину дерева, т. е., число его уровней?

6.5. Мы получили в параграфе 6.4 два рекуррентных соотношения, задающих, с точностью до коэффициентов, функции трудоемкости для алгоритма сортировки слиянием и алгоритма вычисления определителя квадратной матрицы. Используя основную теорему о рекуррентных соотношениях, изложенную в главе 5, получите асимптотические оценки их трудоемкости.

6.6. Разработайте алгоритм сортировки двух и трех чисел, и на их основе постройте рекурсивный алгоритм сортировки слиянием с остакновом при длине массива, равной двум или трем. Одной из задач анализа рекурсивного алгоритма является анализ порожденного дерева рекурсии. Нарисуйте несколько деревьев, порожденных Вашим алгоритмом для произвольных длин исходного массива. Попробуйте получить теоретические функциональные зависимости характеристик такого дерева от длины исходного массива.

Список литературы к главе 6

- 6.1. Ульянов М.В. Классификация и методы сравнительного анализа вычислительных алгоритмов. Научное издание. — М.: Изд-во физико-математической литературы, 2004. — 212 с.
- 6.2. Ульянов М.В. Дополнение к книге Дж. Макконелла Основы современных алгоритмов. — М.: Изд-во Техносфера, 2004. С. 303–366.
- 6.3. Грэхем Р., Кнут Д., Паташник О. Конкретная математика. Основания информатики: Пер. с англ. — М.: Мир, 1998. — 703 с.

- 6.4. *Бахвалов Н.С., Жидков Н.П., Кобельков Г.М.* Численные методы. — М.: Лаборатория Базовых Знаний, 2001 г. — 632 с.
- 6.5. *Axo A., Хопкрофт Дж., Ульман Дж.* Построение и анализ вычислительных алгоритмов: Пер. с англ.: — М.: Мир, 1979. — 546 с.
- 6.6. *Axo A., Хопкрофт Дж., Ульман Дж.* Структуры данных и алгоритмы: Пер. с англ.: — М.: Изд. дом «Вильямс», 2001. — 384 с.
- 6.7. *Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К.* Алгоритмы: построение и анализ, 2-е издание : Пер. с англ. — М.: Изд. дом «Вильямс», 2005. — 1296 с.
- 6.8. <http://www.ccas.ru>
- 6.9. *Ульянов М.В., Гурин Ф.Е., Исаков А.С., Бударагин В.Е.* Сравнительный анализ табличного и рекурсивного алгоритмов точного решения задачи одномерной упаковки // Exponenta Pro Математика в приложениях. 2004. №2(6). С. 64–70.

Глава 7

РЕКУРСИВНЫЕ АЛГОРИТМЫ РЕШЕНИЯ НЕКОТОРЫХ ЗАДАЧ И ИХ ТЕОРЕТИЧЕСКИЙ АНАЛИЗ

Введение. В предыдущих главах были описаны некоторые подходы и методы анализа и получения ресурсных функций рекурсивных алгоритмов. Цель настоящей главы — показать, как они могут быть применены для исследования рекурсивных алгоритмов, относящихся к различным трудоемкостным и емкостным классам. Приведенные ниже примеры рекурсивных алгоритмов не ставят своей целью продемонстрировать наиболее ресурсно-эффективные реализации. Мы ставим несколько иную задачу — продемонстрировать возможности различных методов анализа рекурсивных алгоритмов, и на этой основе сформулировать некоторые пути модификации этих алгоритмов для улучшения их ресурсных характеристик.

Мы начнем с очевидных и подробных примеров, чтобы на их основе продемонстрировать особенности применения различных методов для получения ресурсных функций рекурсивных алгоритмов. Переходя далее к более сложным алгоритмам, разработанным на основе методов декомпозиции и динамического программирования и требующим специальных подходов к их анализу, мы хотим обсудить возможности повышения временной эффективности рекурсивных алгоритмов. Одновременно на этих примерах мы хотим показать, как методы решения рекуррентных соотношений применимы к решению задачи анализа трудоемкости рекурсивных алгоритмов.

§ 1. Алгоритм вычисления факториала

Рекурсивный алгоритм. Рекуррентное соотношение, определяющее рекурсивно заданную функцию факториала от целочисленного аргумента m , имеет вид

$$\begin{cases} f(0) = 1; & f(1) = 1; \\ f(m) = mf(m-1), & m > 1. \end{cases} \quad (7.1.1)$$

Рассмотрим рекурсивный алгоритм, вычисляющий значение факториала, в виде процедурно реализованной рекурсивной функции $F(m)$ с остановом при значении $m = 1$ (справа указано количество базовых операций в строке).

```

F(m)
  If (m=1)           (проверка останова рекурсии)    1
    then
      F ← 1          (останов рекурсии)                1
    else
      F ← m * F(m-1) (рекурсивный вызов)            3
  Return (F)
End.

```

Сложностной класс алгоритма. Алгоритм относится к классу PR , поскольку, имея на входе только одно числовое значение, задает различное количество базовых операций, зависящее от значения параметра m . Подход к определению его принадлежности к подклассу класса PR состоит в том, что мы рассматриваем действительную длину входа алгоритма — количество значащих бит $n = \beta(m)$, предназначенных для двоичного представления числа m . В этом случае, как легко видеть, $2^{n-1} \leq m \leq 2^n - 1$, и трудоемкость алгоритма определяется не количеством n бит числа m , а его значением: $f_A(n) = \Theta(2^n)$, и, следовательно, алгоритм принадлежит подклассу $ExPR$.

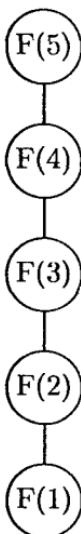


Рис. 7.1. Дерево рекурсии, порожденное алгоритмом вычисления факториала при $m = 4$

алгоритма со значением m порождает дерево рекурсии со следующими характеристиками:

$$\begin{aligned} R(m) &= m, & R_V(m) &= m - 1, & R_L(m) &= 1, \\ H_R(m) &= m, & B_L(m) &= \frac{1}{m}. \end{aligned} \quad (7.1.2)$$

Временная эффективность — функция трудоемкости. В соответствии с методом подсчета вершин дерева рекурсии определим вначале трудоемкость функции $F(m)$ на один вызов/возврат $f_R(1)$. Напомним результат, полученный в главе 6 — трудоемкость механизма вызова/возврата функции или процедуры в базовых операциях обслуживания программного стека составляет

$$f_R(1) = 2(p + r + f + l + 1). \quad (7.1.3)$$

Поскольку функция $F(m)$ передает один параметр ($p = 1$), мы предполагаем, что в стеке сохраняются значения четырех регистров ($r = 4$), одно значение возвращается через имя функции ($f = 1$), и функция $F(m)$ не имеет локальных переменных ($l = 0$), то на основании формулы (7.1.3) имеем (в базовых операциях)

$$f_R(1) = 2(1 + 4 + 1 + 0 + 1) = 14,$$

и, следовательно, с учетом (7.1.2)

$$f_R(m) = R(m) f_R(1) = m \cdot 14.$$

Трудоемкость останова рекурсии включает в себя одно сравнение и однотипное присваивание, таким образом, $f_{CL}(1) = 2$, следовательно:

$$f_{CL}(m) = R_L(m) f_{CL}(1) = 1 \cdot 2 = 2,$$

во всех внутренних вершинах дерева (фрагмент рекурсивного вызова) выполняется одинаковое количество операций $f_{CV}(v) = 4$, и

$$f_{CV}(m) = R_V(m) f_{CV}(v) = (m - 1) \cdot 4 = 4m - 4.$$

Объединяя все компоненты, получаем

$$f_A(m) = 14m + 4m - 4 + 2 = 18m - 2, \quad (7.1.4)$$

и доля операций обслуживания дерева рекурсии составляет

$$F_R(m) = \frac{f_R(m)}{f_A(m)} = \frac{14m}{18m - 2} \approx \frac{7}{9}, \quad m \rightarrow \infty. \quad (7.1.5)$$

Полученные результаты говорят о том, что более 77% трудоемкости тратится на организацию рекурсии, и можно говорить о крайней неэффективности рекурсивной реализации (что, впрочем, было и так очевидно — тем не менее, мы получили количественную оценку). Заметим, что трудоемкость процедурной реализации в принятой методике счета базовых операций составляет $f_A(m) = 5m + 5$, что в три с лишним раза меньше.

Емкостная эффективность — функция объема памяти. Для получения оценки емкостной эффективности нам нужна функция $H_R(m)$, определяющая наибольшую глубину дерева рекурсии. Напомним, что мы оцениваем объем оперативной памяти в условных ячейках, размер которых достаточен для хранения используемых простых переменных. Поскольку, в соответствии с описанным в главе 2 механизмом рекурсивного вызова, функция перед передачей управления сохраняет в программном стеке значения параметров, локальных переменных и регистров общего назначения в точке своего вызова, а также адрес возврата, то наибольший объем памяти в области стека будет задействован в конце самой глубокой ветви дерева рекурсии и составит

$$V_{st}(m) = H_R(m)(p + r + f + l + 1).$$

Поскольку данный алгоритм не требует локальных переменных и не использует глобальных ячеек и массивов, то

$$V(m) = V_{st}(m) = H_R(m)(1 + 4 + 1 + 0 + 1) = 7H_R(m) = 7m. \quad (7.1.6)$$

На основе (7.1.4) и (7.1.6) мы можем определить ресурсную сложность алгоритма

$$\mathfrak{R}_c(m) = \langle \Theta(m), \Theta(m) \rangle.$$

Анализ трудоемкости алгоритма методом рекуррентных соотношений. Воспользуемся уже известными значениями $f_{CL}(1) = 2$, $f_R(1) = 14$ и $f_{CV}(1) = 4$ для получения функции трудоемкости методом рекуррентных соотношений. На основании общей формулы этого метода для случая фиксированных значений трудоемкости в вершинах дерева имеем

$$\begin{cases} f_A(1) = f_R(1) + f_{CL}(1); \\ f_A(m) = f_A(m-1) + f_R(1) + f_{CV}(v), \quad m > 1. \end{cases}$$

Подставляя числовые значения, получаем рекуррентное соотношение для определения функции трудоемкости $f_A(m)$:

$$\begin{cases} f_A(1) = 16; \\ f_A(m) = f_A(m-1) + 18, \quad m > 1, \end{cases}$$

и решение в замкнутой форме является очевидным

$$f_A(m) = 18(m-1) + 16 = 18m - 2.$$

Отметим, что, хотя мы и получили результат, аналогичный (7.1.4), метод подсчета вершин дерева рекурсии позволяет получить более детальную информацию о рекурсивном алгоритме.

§ 2. Алгоритм вычисления чисел Фибоначчи

Рекурсивный алгоритм. Рекуррентное соотношение, определяющее рекурсивно заданную функцию, значениями которой являются числа Фибоначчи (для аргумента m), имеет вид

$$\begin{cases} fb(1) = 1; \quad fb(2) = 1; \\ fb(m) = fb(m-1) + fb(m-2), \quad m \geq 3. \end{cases} \quad (7.2.1)$$

Рассмотрим рекурсивный алгоритм, вычисляющий числа Фибоначчи, в виде рекурсивной функции **Fb(m)** с остановом при значении $m = 1$ или $m = 2$.

Fb(m)

If (m=1) or (m=2)	(проверка останова рекурсии)	3
then		
Fb \leftarrow 1	(останов рекурсии)	1
else		
Fb \leftarrow Fb(m-1)+Fb(m-2)	(рекурсия)	4

Return (Fb)

End.

Его принадлежность к сложностному классу мы определим после анализа трудоемкости алгоритма.

Анализ алгоритма методом подсчета вершин рекурсивного дерева. Дерево рекурсии, порождаемое данным алгоритмом, представляет собой несимметричное бинарное дерево, вид которого для $m = 5$ приведен на рисунке 7.2. Поскольку вычисляемое число Фибоначчи получается данным алгоритмом путем нарастающего суммирования единичных значений в листьях, то, очевидно, что количество листьев дерева равно значению числа Фибоначчи — $fb(m)$. Замкнутая форма для вычисления $fb(m)$ имеет вид (см. главу 1)

$$fb(m) = \frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2} \right)^m - \frac{1}{\sqrt{5}} \left(\frac{1 - \sqrt{5}}{2} \right)^m.$$

Для определения количества внутренних вершин в дереве рекурсии заметим, что $R_V(1) = 0$, $R_V(2) = 0$, и увеличение значения m на единицу порождает объединение двух деревьев с добавлением корневой вершины (см. рис. 7.2).

Эти рассуждения позволяют записать рекуррентное соотношение для вычисления $R_V(m)$:

$$\begin{cases} R_V(1) = 0; & R_V(2) = 0; \\ R_V(m) = R_V(m-1) + R_V(m-2) + 1, & m \geq 3. \end{cases}$$

Это линейное неоднородное рекуррентное соотношение с постоянными коэффициентами и функцией $g(m) = 1$. Используя метод решения из главы 1, получаем

$$R_V(m) = fb(m) - 1.$$

Поскольку в силу (7.2.1) рекурсия останавливается при $m = 1$ или $m = 2$, то вызов данного алгоритма со значением m порождает дерево

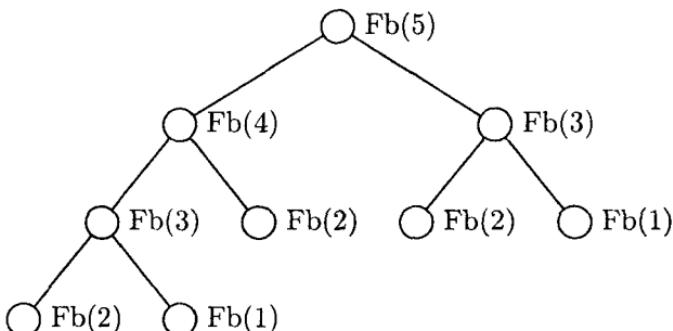


Рис. 7.2. Дерево рекурсии, порожденное алгоритмом вычисления чисел Фибоначчи при $m = 5$

рекурсии со следующими характеристиками при $m \geq 3$:

$$\begin{aligned} R(m) &= 2 \text{fb}(m) - 1, & R_V(m) &= \text{fb}(m) - 1, & R_L(m) &= \text{fb}(m), \\ H_R(m) &= m - 1, & B_L(m) &\approx \frac{1}{2}, & m &\rightarrow \infty \end{aligned} \quad (7.2.2)$$

Временная эффективность — функция трудоемкости. В соответствии с методом подсчета вершин дерева рекурсии определим вначале трудоемкость $f_R(1)$ на один вызов/возврат функции **Fb(m)**. Поскольку функция **Fb(m)** передает один параметр ($p = 1$), мы предполагаем, что в стеке сохраняются значения четырех регистров ($r = 4$), одно значение возвращается через имя функции ($f = 1$) и функция **Fb(m)** не имеет локальных переменных ($l = 0$), то имеем

$$f_R(1) = 2(1 + 4 + 1 + 0 + 1) = 14,$$

и, следовательно, с учетом (7.2.2)

$$f_R(m) = R(m) f_R(1) = (2 \text{fb}(m) - 1) \cdot 14 = 28 \text{fb}(m) - 14.$$

Трудоемкость останова рекурсии включает в себя два сравнения, одну логическую операцию и одно присваивание, таким образом $f_{CL}(1) = 4$, следовательно:

$$f_{CL}(m) = R_L(m) f_{CL}(1) = \text{fb}(m) \cdot 4,$$

во всех внутренних вершинах дерева (фрагмент рекурсивного вызова) выполняется одинаковое количество операций $f_{CV}(v) = 3 + 4 = 7$, и

$$f_{CV}(m) = R_V(m) f_{CV}(v) = (\text{fb}(m) - 1) \cdot 7 = 7 \text{fb}(m) - 7.$$

Объединяя все компоненты, получаем

$$f_A(m) = 39 \text{fb}(m) - 21, \quad (7.2.3)$$

и доля операций обслуживания дерева рекурсии составляет

$$F_R(m) = \frac{f_R(m)}{f_A(m)} = \frac{28 \text{fb}(m) - 14}{39 \text{fb}(m) - 21} \approx \frac{28}{39}, \quad m \rightarrow \infty. \quad (7.2.4)$$

Емкостная эффективность — функция объема памяти. Получение оценки емкостной эффективности для данного алгоритма аналогично оценке алгоритма вычисления факториала и определяется функцией $H_R(m)$ — наибольшей глубиной дерева рекурсии. Поскольку данный алгоритм также не требует локальных переменных и не использует глобальных ячеек, то требуемая память составит

$$V(m) = V_{st}(m) = H_R(m)(1 + 4 + 1 + 0 + 1) = 7H_R(m) = 7m - 7, \quad (7.2.5)$$

и ресурсная сложность алгоритма с учетом (7.2.3) имеет вид

$$\mathfrak{R}_c(m) = \langle \Theta(\text{fb}(m)), \Theta(m) \rangle.$$

Сложностной класс алгоритма. Алгоритм, очевидно, относится к классу PR , поскольку, имея на входе только одно числовое значение — m , задает различное количество базовых операций, зависящее от значения этого параметра. Заметим, что поскольку числа Фибоначчи растут экспоненциально, то в силу формулы (7.2.3) анализируемый алгоритм принадлежит подклассу $ExPR$, т. е. трудоемкость является экспоненциальной по m в силу формулы для общего вида чисел Фибоначчи и дважды экспоненциальной по количеству битов в двоичном представлении числа m .

Анализ трудоемкости алгоритма методом рекуррентных соотношений. Воспользуемся уже известными значениями $f_{CL}(1) = 4$, $f_R(1) = 14$ и $f_{CV}(v) = 7$ для получения функции трудоемкости методом рекуррентных соотношений. На основании общей формулы этого метода для случая фиксированных значений имеем

$$\begin{cases} f_A(1) = f_R(1) + f_{CL}(1); \\ f_A(2) = f_R(1) + f_{CL}(1); \\ f_A(m) = f_A(m-1) + f_A(m-2) + f_R(1) + f_{CV}(1), \quad m \geq 3. \end{cases}$$

Подставляя числовые значения, получаем рекуррентное соотношение для определения функции трудоемкости — $f_A(m)$

$$\begin{cases} f_A(1) = 18; \quad f_A(2) = 18; \\ f_A(m) = f_A(m-1) + f_A(m-2) + 21, \quad m \geq 3, \end{cases}$$

решение в замкнутой форме будем искать в виде $f_A(m) = a \text{fb}(m) + b$, и, подставляя в рекуррентное соотношение, имеем

$$a \text{fb}(m) + b = a \text{fb}(m-1) + b + a \text{fb}(m-2) + b + 21,$$

но $\text{fb}(m) = \text{fb}(m-1) + \text{fb}(m-2)$, следовательно, $b = -21$, подставляя в условие останова рекурсии, получаем $a \text{fb}(1) - 21 = 18$, откуда $a = 39$, и, окончательно

$$f_A(m) = 39 \text{fb}(m) - 21,$$

где $\text{fb}(m)$ — m -ое число Фибоначчи.

§ 3. Алгоритм вычисления квадратного корня

Математическое обоснование рекурсивного алгоритма. Рекуррентное соотношение, определяющее рекурсивную последовательность с начальным значением $x_1 = 1$, пределом которой при $n \rightarrow \infty$ является значение \sqrt{a} , мы уже рассматривали в главе 1. Оно имеет вид

$$x_{n+1} = \frac{1}{2} \left(x_n + \frac{a}{x_n} \right). \quad (7.3.1)$$

Однако в реальных компьютерных вычислениях нам необходимо получить это значение с некоторой точностью ε , которая зависит от нужд

применения и ограничена точностью представления чисел с плавающей точкой в используемом компьютере. Как правило, и это согласуется с форматами данных языков программирования, значение $\epsilon = 10^{-20}$ является вполне достаточным. Таким образом, возникает задача оценки количества членов последовательности (7.3.1), позволяющих достичь заданной точности. В общем случае значение x_n , доставляющее заданную точность, равно как и значение его номера n , зависит как от требуемой точности ϵ , так и от значения числа a , т. е. $n = n(a, \epsilon)$. Попробуем оценить, как абсолютная ошибка для следующего члена последовательности (7.3.1) зависит от ошибки предыдущего члена. Введем следующие обозначения:

$$\epsilon_{n+1} = x_{n+1} - \sqrt{a}, \quad \epsilon_n = x_n - \sqrt{a},$$

и после подстановки в (7.3.1) и несложных преобразований получим

$$\epsilon_{n+1} = \frac{1}{2} \left(\frac{\epsilon_n^2}{\sqrt{a} + \epsilon_n} \right). \quad (7.3.2)$$

В целях дальнейшего анализа и, как это станет ясно ниже, для повышения временной эффективности алгоритма, целесообразно так организовать процесс вычислений, чтобы зависимость $n(a, \epsilon)$ была аддитивной по a и ϵ . Тем самым мы хотим разделить процесс вычисления квадратного корня на две фазы, трудоемкость каждой из которых определялась бы только одним значением — a или ϵ , что приводит к представлению $n(a, \epsilon)$ в виде

$$n(a, \epsilon) = h(a) + g(\epsilon). \quad (7.3.3)$$

Основная идея выделения таких фаз заключается в следующем — на первой фазе мы последовательным делением уменьшаем (при $a \geq 1$) значение a до некоторой величины a' , выполняя $h(a)$ операций деления, а на второй фазе получаем значение квадратного корня из a' , вычисляя $g(\epsilon)$ значений последовательности (7.3.1). После этого остается только скорректировать полученный результат с учетом предшествовавших операций деления.

Рассмотрим вначале вторую фазу. Отметим, что если в формуле (7.3.2) значения a ограничены сверху, то при условиях, что $\sqrt{a} \geq 1$ и $\forall n |\epsilon_n| < \sqrt{a}$, формула (7.3.2) приобретает вид рекуррентного соотношения, зависящего только от начального значения ϵ_1 . Этот путь является конструктивным, поскольку он позволяет выбрать такой сегмент значений a , при котором значение ϵ_1 удовлетворяет условию $|\epsilon_1| \leq 0,5$, что обеспечивает быструю квадратичную сходимость значений ϵ_n к нулю. Выбор границы $|\epsilon_1| \leq 0,5$ обусловлен как значением коэффициента в формуле (7.3.2), так и принятой двоичной системой счисления. Определим сегмент значений a , при котором $|\epsilon_1| \leq 0,5$. Для этого вначале ограничим область допустимых значений a : $a \geq 1$. Если мы фиксируем начальное приближение корня $x_1 = 1,5$, то в силу определения ϵ_1 и условия $|\epsilon_1| \leq 0,5$, значение корня лежит в пределах

$1 \leq \sqrt{a} \leq 2$, и, следовательно, значения a ограничены сегментом $1 \leq a \leq 4$. При этом, поскольку $1 \leq \sqrt{a} \leq 2$ и $\forall n > 2 \quad 0 < \varepsilon_n < 0,5$, то $\sqrt{a} + \varepsilon_n > 1$, и имеет место соотношение

$$\varepsilon_{n+1} = \frac{1}{2} \left(\frac{\varepsilon_n^2}{\sqrt{a} + \varepsilon_n} \right) < \frac{1}{2} \varepsilon_n^2.$$

Тем самым, если мы, переходя к абсолютным значениям ε_n для оценки ошибки между истинным значением \sqrt{a} и x_n , выбираем значения a , максимизирующие $\varepsilon_1 - \varepsilon_1 = 0,5$, то в результате получаем следующее рекуррентное соотношение:

$$\begin{cases} \varepsilon_1 = \frac{1}{2}; \\ \varepsilon_{n+1} = \frac{1}{2} \varepsilon_n^2, \quad n > 1. \end{cases} \quad (7.3.4)$$

Рекуррентное соотношение (7.3.4) может быть решено методом подстановки

$$\varepsilon_2 = \frac{1}{2} \varepsilon_1^2, \quad \varepsilon_3 = \frac{1}{2} \left(\frac{1}{2} \varepsilon_1^2 \right)^2 = \frac{1}{8} \varepsilon_1^4, \quad \varepsilon_4 = \frac{1}{2} \left(\frac{1}{8} \varepsilon_1^4 \right)^2 = \frac{1}{128} \varepsilon_1^8,$$

что позволяет получить общий вид для ε_n :

$$\varepsilon_n = \frac{1}{2^{2^{n-1}-1}} \varepsilon_1^{2^{n-1}} = 2 \left(\frac{\varepsilon_1}{2} \right)^{2^{n-1}},$$

а поскольку начальное значение $\varepsilon_1 = 0,5$, то окончательная формула имеет вид

$$\varepsilon_n = 2 \left(\frac{1}{4} \right)^{2^{n-1}}. \quad (7.3.5)$$

Если мы задаем ε — точность вычислений (абсолютную ошибку), то необходимое число членов последовательности (7.3.1) — $n = g(\varepsilon)$ определяется из уравнения $\varepsilon = \varepsilon_n$ на основе (7.3.5). Дважды логарифмируя это уравнение, получаем:

$$g(\varepsilon) = \log_2 (1 - \log_2 \varepsilon). \quad (7.3.6)$$

Некоторые значения функции $g(\varepsilon)$ приведены в таблице 7.1. Поскольку значение $g(\varepsilon)$ должно быть целым, то нам необходимо вычислить $n = \lceil g(\varepsilon) \rceil$ членов последовательности. Напомним, что мы получили оценку сверху, а вычислительные эксперименты показывают, что при $1 \leq a \leq 4$ и начальном значении $x_1 = 1,5$ уже шестой член последовательности (7.3.1) дает точность порядка 10^{-25} , тем самым для большинства применений $g(\varepsilon) = 6$.

Таблица 7.1

ε	$g(\varepsilon)$
1,00E-10	5,09673754
1,00E-15	5,66757771
1,00E-20	6,07550187
1,00E-25	6,39314506
1,00E-30	6,65331578

На второй фазе мы используем значение $a' : 1 \leq a' \leq 4$ как результат, полученный последовательными операциями деления на первой фазе. Теперь, когда сегмент для a' определен, становится ясно, что делителем в первой фазе является число 4, и вычисления первой фазы могут быть описаны в виде следующего рекуррентного соотношения, задающего последовательность y_k :

$$\begin{cases} y_1 = a; \\ y_{k+1} = y_k/4, \quad y_k > 4. \end{cases} \quad (7.3.7)$$

Для некоторого значения $k = m$, при котором $y_m \leq 4$, рекурсия останавливается и выполняется вторая фаза с $a' = y_m$. По полученному значению $\sqrt{a'}$ искомый корень может быть восстановлен по формуле

$$\sqrt{a} = 2^{m-1} \sqrt{a'}. \quad (7.3.8)$$

Особо отметим, что практически формула (7.3.8) может быть реализована путем сложения порядка числа $\sqrt{a'}$ с числом $m - 1$.

Определим количество членов последовательности (7.3.7), необходимых для получения $a' : 1 \leq a' \leq 4$. С этой целью рассмотрим рекурсивно заданную целую функцию $h(a)$, значением которой является необходимое количество операций деления на 4 для получения a' . Рекуррентное соотношение, задающее $h(a)$, имеет вид:

$$\begin{cases} h(a) = 0, \quad 1 \leq a \leq 4; \\ h(a) = h(a/4) + 1, \quad a > 4. \end{cases} \quad (7.3.9)$$

Использование метода подстановки позволяет получить неравенство, на основе которого функция $h(a)$ может быть определена в явном виде:

$$1 \leq \frac{a}{4^{h(a)}} \leq 4,$$

откуда, логарифмируя и учитывая целочисленность функции $h(a)$, получаем

$$h(a) = \lceil \log_4 a \rceil - 1. \quad (7.3.10)$$

Таким образом, мы имеем математическое обоснование для разработки двухфазного рекурсивного алгоритма вычисления квадратного корня.

Рекурсивный алгоритм. Рассмотрим двухфазный рекурсивный алгоритм, вычисляющий значение квадратного корня, в виде двух процедурно реализованных рекурсивных функций.

Функция **Sqrt(a)** — реализация первой фазы алгоритма при $a > 1$.

Sqrt(a)

If (a<4)	(проверка останова рекурсии)	1
then		
Sqrt ← Sq(a,6)	(останов рекурсии — вызов второй фазы)	1
else		
Sqrt ← 2 * Sqrt(a/4)	(рекурсивный вызов)	3

Return (Sqrt)
End.

Мы рассмотрели для определенности случай, когда $a > 1$, если же значение $a < 1$, то первая фаза сводится к умножению на 4 при рекурсивном вызове и последующему делению на 2 при рекурсивном возврате.

Функция **Sq(a,n)** — реализация второй фазы алгоритма.

Sq(a,n)

If (n=1)	(проверка останова рекурсии)	1
then		
Sq $\leftarrow 1,5$	(останов рекурсии)	1
else		
y $\leftarrow \text{Sq}(a,n-1)$	(рекурсивный вызов)	2
Sq $\leftarrow 0,5 * (y+a/y)$	(результат)	4

Return (Sq)

End.

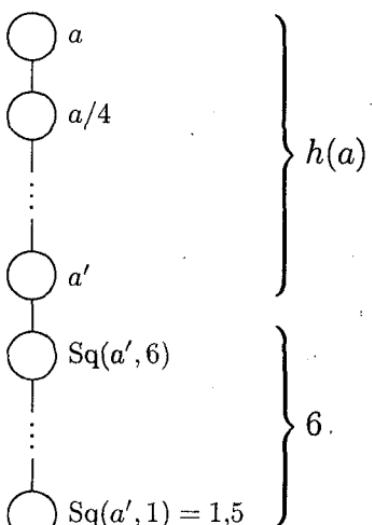
Сложностной класс алгоритма. Алгоритм относится к классу *PR*, поскольку, имея на входе только одно числовое значение a (мы зафиксировали $\varepsilon = 10^{-20}$ и $g(\varepsilon) = 6$), задает различное количество базовых операций, зависящее от значения параметра a . Обращаем внимание на особенность, связанную с определением его принадлежности к подклассу класса *PR* — если мы рассматриваем обычное двоичное представление числа a , то в соответствии с формулой (7.3.10) для $h(a)$ он относится к подклассу *PlPR*.

Однако для реального представления действительных чисел в виде мантиссы и порядка функция $h(a)$ линейна по количеству бит порядка и, следовательно, алгоритм относится к подклассу *ExPR*.

Анализ алгоритма методом подсчета вершин дерева рекурсии Поскольку необходимое число членов двух последовательностей, задающих дерево рекурсии, мы уже определили выше, то полное дерево, порожданное данным алгоритмом — это унарное дерево с

Рис. 7.3. Дерево рекурсии, порожденное алгоритмами **Sqrt** и **Sq** при вычислении квадратного корня из a

$h(a) + 7$ вершинами, т. к. число вершин в первой фазе на единицу больше $h(a)$, и с одним листом (см. рис. 7.3).



Характеристики дерева рекурсии при вызове алгоритма со значением a имеют вид:

$$\begin{aligned} R(a) &= h(a) + 7, \quad R_V(a) = h(a) + 6, \quad R_L(a) = 1, \\ H_R(a) &= h(a) + 7, \quad B_L(a) = \frac{1}{h(a) + 7}. \end{aligned} \quad (7.3.11)$$

Временная эффективность — функция трудоемкости. При анализе временной эффективности мы будем рассматривать только первую фазу алгоритма — функцию **Sqrt(a)**, считая, что трудоемкость второй фазы — вычисление **Sq(a,6)**, является трудоемкостью останова рекурсии в первой фазе алгоритма. Определим трудоемкость второй фазы, при вычислении которой мы вначале определим трудоемкость обслуживания рекурсии. Поскольку функция **Sq(a,n)** передает два параметра ($p = 2$), мы предполагаем, что в стеке сохраняются значения четырех регистров ($r = 4$), одно значение возвращается через имя функции ($f = 1$) и функция **Sq(a,n)** имеет одну локальную ячейку y ($l = 1$), то на основании общей формулы имеем $f_{RSq}(1) = 2(2 + 4 + 1 + 1 + 1) = 18$. Общая трудоемкость второй фазы, которая включает в себя 5 рекурсивных вызовов и один останов, по уже освоенной читателем методике (мы надеемся на это), с учетом трудоемкости обслуживания рекурсии, одной операции присваивания и сравнения во фрагменте останова рекурсии для функции **Sqrt(a)**, составляет

$$f_{CL}(a) = 6f_{RSq}(1) + 5(1 + 2 + 4) + 1 \cdot 2 + 2 = 147.$$

Трудоемкость функции **Sqrt(a)** — первой фазы нашего алгоритма — на обслуживание рекурсии (один передаваемый параметр без локальных переменных — $f_R(1) = 14$) составляет

$$f_R(a) = (h(a) + 1)f_R(1) = \lceil \log_4 a \rceil \cdot 14,$$

во всех внутренних вершинах дерева рекурсии для функции **Sqrt(a)** (фрагмент рекурсивного вызова) выполняется одинаковое количество операций $f_{CV}(v) = 1 + 3 = 4$, и

$$f_{CV}(a) = h(a)f_{CV}(v) = (\lceil \log_4 a \rceil - 1) \cdot 4.$$

Объединяя все компоненты, получаем трудоемкость двухфазного алгоритма

$$f_A(a) = 18\lceil \log_4 a \rceil + 143, \quad (7.3.12)$$

и доля операций обслуживания дерева рекурсии по обеим фазам составляет

$$F_R(a) = \frac{f_R(a)}{f_A(a)} = \frac{14\lceil \log_4 a \rceil + 6 \cdot 18}{18\lceil \log_4 a \rceil + 143} \approx \frac{7}{9}, \quad m \rightarrow \infty. \quad (7.3.13)$$

Отметим, что в силу формулы (7.3.2) при больших значениях a несколько первых членов рекурсивной последовательности (7.3.1) будут иметь только вдвое меньшую абсолютную ошибку. Это означает, что описанный двухфазный алгоритм обладает в силу (7.3.12) лучшей

временной эффективностью по сравнению с непосредственной реализацией рекуррентного соотношения (7.3.1), требующей 6 продуктивных операций на каждом рекурсивном вызове.

Емкостная эффективность — функция объема памяти. При оценке объема памяти, требуемого данным двухфазным алгоритмом, обратим внимание, что во второй фазе помимо сохранения регистров в области стека туда же будет помещаться текущее значение локальной переменной в точке вызова — $l_2 = 1$. Это приводит к отдельному учету глубины дерева рекурсии для первой и второй фаз, и соответствующая формула имеет вид

$$V_{st}(m) = (h(a) + 1)(p + r + f + l_1 + 1) + 6(p + r + f + l_2 + 1).$$

и, следовательно, при отсутствии глобальных ячеек и массивов

$$\begin{aligned} V(m) = V_{st}(m) &= (h(a) + 1)(1 + 4 + 1 + 0 + 1) + \\ &+ 6(2 + 4 + 1 + 1 + 1) = 7 \lceil \log_4 a \rceil + 54. \end{aligned}$$

Таким образом, ресурсная сложность алгоритма имеет вид

$$\mathfrak{R}_c(a) = \langle \Theta(\lceil \log_4 a \rceil), \Theta(\lceil \log_4 a \rceil) \rangle.$$

Мы оставляем получение функции трудоемкости методом рекуррентных соотношений для этого алгоритма в качестве упражнения для заинтересованных читателей.

§ 4. Алгоритм быстрого возведения числа в целую степень

Математическое обоснование рекурсивного алгоритма. Задача о быстром возведении числа в целую степень, т. е. вычисление значения $y = x^m$ для целого значения m , лежит в основе алгоритмического обеспечения многих криптосистем, например RSA [7.3]. Отметим, что в этом аспекте применения значения m достаточно велики, что делает неприемлемым обычный алгоритм последовательного умножения, а вычисления производятся с целым значением x в мультиплексивной группе вычетов Z_n^* . В этом случае используется быстрый алгоритм возведения в степень методом последовательного возведения в квадрат [7.1], детальный анализ которого представляет определенный интерес.

Идея быстрого алгоритма возведения в степень состоит в использовании двоичного представления числа m и вычисления четных степеней x путем повторного возведения в квадрат [7.1]. Пусть, например, $m = 11$, тогда вычисления идут по схеме:

$$x^{11} = x \cdot x^5 \cdot x^5, \quad x^5 = x \cdot x^2 \cdot x^2, \quad x^2 = x \cdot x, \quad x = x \cdot 1,$$

а рекуррентное соотношение, определяющее степень m числа x , имеет вид:

$$x^m = \begin{cases} 1, & m = 0; \\ (x^{m/2})^2, & m = 2k; \\ x(x^{(m-1)/2})^2, & m = 2k + 1. \end{cases}$$

Это позволяет записать рекуррентное соотношение, определяющее рекурсивно заданную функцию $f(x, m) = x^m$:

$$f(x, m) = \begin{cases} 1, & m = 0; \\ (f(x, m/2))^2, & m = 2k; \\ x(f(x, (m-1)/2))^2, & m = 2k + 1. \end{cases} \quad (7.4.1)$$

Рекурсивный алгоритм. Используя соотношение (7.4.1), построим рекурсивный алгоритм, вычисляющий значение степени, в виде процедурно реализованной рекурсивной функции $F(x, m)$ с остановом при значении $m = 0$ (справа, как обычно, указано количество базовых операций в строке). Особо отметим, что для нечетных чисел стандартная операция целочисленного деления отбрасывает остаток, т. е.

$$m \text{ div } 2 = (m - 1) \text{ div } 2 = k, \quad m = 2k + 1, \quad k \geq 0,$$

что позволяет избежать вычитания единицы из m для нечетных чисел.

F(x, m)

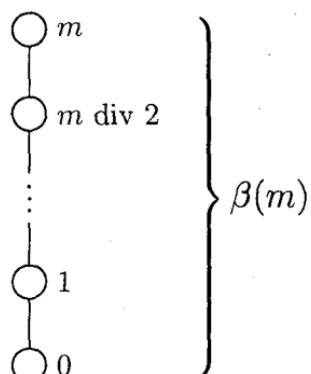
```

If (m = 0)                                (проверка останова рекурсии)    1
  then
    F ← 1                                    (останов рекурсии)            1
  else
    If (m mod 2)=0                          (проверка четности)          2
      then
        z ← F(x, m div 2)                  (рекурсия)                   2
        F ← z * z                         (квадрат результата)          2
      else
        z ← F(x, m div 2)                  (рекурсия)                   2
        F ← x * z * z                    (умноженный на x квадрат результата) 3
    end If
end If
Return (F)
End.

```

Сложностной класс алгоритма. Алгоритм относится к классу PR , поскольку, имея на входе два числовых значения — x и m ,

задает различное количество базовых операций, зависящее от значения параметра m . Отметим, что поскольку операция умножения является базовой, значение x не влияет на трудоемкость алгоритма. Определим его принадлежность к подклассу класса PR . Вначале напомним читателям, что в главе 5 были введены специальные функции, которые мы будем использовать в целях анализа этого алгоритма: $\beta(m)$, значением которой является общее количество разрядов в двоичном представлении числа m , и функции $\beta_1(m)$ и $\beta_0(m)$, задающие, соответственно, количество единиц и нулей в этом представлении.



Рассмотрим действительную длину входа алгоритма — количество значащих бит n , предназначенных для двоичного представления числа m , $n = \beta(m)$. Поскольку каждый рекурсивный вызов уменьшает на единицу количество бит степени (при делении на два), то трудоемкость алгоритма определяется количеством бит n на входе, $f_A(n) = \Theta(n) = \Theta(\lfloor \log_2 m \rfloor)$, и, следовательно, алгоритм принадлежит подклассу $PIPR$.

Анализ алгоритма методом подсчета вершин рекурсивного дерева. Дерево рекурсии, порожденное данным алгоритмом, представляет собой унарное дерево — цепочку с одним листом, и количеством внутренних вершин равным $\beta(m)$. Вид дерева рекурсии показан на рисунке 7.4. Таким образом,

Рис. 7.4. Дерево рекурсии, порожденное алгоритмом возведения в степень для показателя m

вызов данного алгоритма со значением m порождает дерево рекурсии со следующими характеристиками:

$$R(m) = \beta(m) + 1 = \lfloor \log_2 m \rfloor + 2, \quad R_V(m) = \beta(m) = \beta_1(m) + \beta_0(m),$$

$$R_L(m) = 1, \quad H_R(m) = \beta(m) + 1, \quad B_L(m) = \frac{1}{\lfloor \log_2 m \rfloor + 2}. \quad (7.4.2)$$

Временная эффективность — функция трудоемкости. В соответствии с методом подсчета вершин дерева рекурсии определим вначале трудоемкость функции $F(x, m)$ на один вызов/возврат — $f_R(1)$. Поскольку функция $F(x, m)$ передает два параметра ($p = 2$), мы предполагаем, что в стеке сохраняются значения четырех регистров ($r = 4$), одно значение возвращается через имя функции ($f = 1$) и функция $F(x, m)$ имеет одну локальную переменную z ($l = 1$), то

$$f_R(1) = 2(2 + 4 + 1 + 1 + 1) = 18,$$

и, следовательно, с учетом (7.4.2)

$$f_R(m) = R(m) f_R(1) = (\beta(m) + 1) \cdot 18.$$

Трудоемкость останова рекурсии включает в себя одно сравнение и одноточечное присваивание, таким образом $f_{CL}(1) = 2$, и

$$f_{CL}(m) = R_L(m) f_{CL}(1) = 1 \cdot 2 = 2,$$

во внутренних вершинах дерева (фрагмент рекурсивного вызова) выполняется $f_{CV}(v) = 7$ или $f_{CV}(v) = 8$ операций, в зависимости от того, четным или нечетным является текущий аргумент функции (младший бит равен 0 или 1), следовательно,

$$f_{CV}(m) = 8\beta_1(m) + 7\beta_0(m).$$

Объединяя все компоненты, на основании общей формулы метода, получаем

$$f_A(m) = 18\beta(m) + 8\beta_1(m) + 7\beta_0(m) + 20, \quad (7.4.3)$$

и доля операций обслуживания дерева рекурсии составляет

$$F_R(m) = \frac{f_R(m)}{f_A(m)} = \frac{18\beta(m) + 18}{18\beta(m) + 8\beta_1(m) + 7\beta_0(m) + 20} \leq \frac{9}{13}, \\ m \rightarrow \infty. \quad (7.4.4)$$

Определим трудоемкость алгоритма для лучшего и худшего случаев. При этом мы должны рассматривать фиксированную битовую длину входа алгоритма, т.е. те значения m , при которых функция $\beta(m) = \text{const}$. Обозначим $\beta(m) = n$ и рассмотрим значения m , которые доставляют минимальное и максимальное значение функции $\beta_1(m)$ при $\beta(m) = n$. Точки экстремума функции $\beta_1(m)$ соответствуют значениям аргумента $m = 2^{n-1}$ и $m = 2^n - 1$. В случае, если $m = 2^{n-1}$ (только старший бит двоичного представления числа m равен единице), $\beta_1(m) = 1$, то

$$f_A^V(m : \beta(m) = n) = 18n + 81 + 7(n-1) + 20 = 25n + 21,$$

а в случае, если $m = 2^n - 1$ (все биты числа m равны единице), $\beta_1(n) = n$, то

$$f_A^H(m : \beta(m) = n) = 18n + 8n + 20 = 26n + 20. \quad (7.4.5)$$

Заметим, что для данного алгоритма функция трудоемкости не является монотонно возрастающей. Рассмотрим $m = 2^n - 1$ и $m + 1 = 2^n$, в этом случае

$$\beta(m) = n, \quad \beta(m+1) = n+1, \quad \beta_1(m) = n, \quad \beta_1(m+1) = 1,$$

и, следовательно,

$$f_A(m) = 26n + 20, \quad f_A(m+1) = 25(n+1) + 21 = 25n + 46,$$

что влечет $f_A(m) > f_A(m+1)$ при $m \geq 2^{27}$.

Если показатель степени заранее не известен, то можно получить среднюю оценку в предположении, что представление числа m занимает n значащих двоичных разрядов, т.е.

$$2^{n-1} \leq m < 2^n, \quad n-1 \leq \log_2(m) < n, \quad \lfloor \log_2(m) \rfloor := n-1,$$

тогда, используя результат, полученный в главе 5 для $\beta_s(m)$, имеем

$$\begin{aligned}\bar{f}_A(m) &= 18\beta(m) + 8\beta_s(m) + 7(\beta(m) - \beta_s(m)) + 20 = \\ &= 25\beta(m) + \beta_s(m) + 20 = 25,5\beta(m) + 20,5,\end{aligned}\quad (7.4.6)$$

и переходя в (7.4.6) от $\beta(m)$ к n , получаем

$$\bar{f}_A(m : \beta(m) = n) = 25,5n + 20,5. \quad (7.4.7)$$

Таким образом, трудоемкость быстрого алгоритма возвведения в степень в среднем линейно зависит от количества бит в двоичном представлении показателя степени. Введение специальных функций $\beta_1(m)$ и $\beta(m)$ позволяет получить точное значение трудоемкости анализируемого алгоритма как функции показателя степени.

Емкостная эффективность — функция объема памяти. Для получения оценки емкостной эффективности используем функцию $H_R(m)$. Данный алгоритм использует одну локальную ячейку для хранения значения z , и, в соответствии с принятой методикой, наибольший объем памяти в области стека составит

$$V_{st}(m) = H_R(m)(2 + 4 + 1 + 1 + 1) = 9H_R(m),$$

подставляя $H_R(m)$ из (7.4.2), получаем емкостную эффективность алгоритма

$$V(m) = V_{st}(m) = 9H(m) = 9\lfloor\log_2 m\rfloor + 18, \quad (7.4.8)$$

что позволяет, на основе формул (7.4.5) и (7.4.8), определить ресурсную сложность алгоритма в худшем случае:

$$\mathfrak{R}_c^{\wedge}(m) = \langle \Theta(\log_2 m), \Theta(\log_2 m) \rangle.$$

Анализ трудоемкости алгоритма методом рекуррентных соотношений. Получим трудоемкость алгоритма в среднем. Для этого используем уже известные значения $f_{CL}(1) = 2$, $f_R(1) = 18$ и среднее значение $\bar{f}_{CV}(v) = 7,5$. На основании общей формулы этого метода для случая фиксированных значений имеем

$$\begin{cases} f_A(0) = f_R(1) + f_{CL}(1); \\ f_A(m) = f_A\left(\left\lfloor \frac{m}{2} \right\rfloor\right) + f_R(1) + \bar{f}_{CV}(v), \quad m \geq 1. \end{cases}$$

Подставляя числовые значения, получаем рекуррентное соотношение для определения функции трудоемкости в среднем — $\bar{f}_A(m)$:

$$\begin{cases} \bar{f}_A(0) = 20; \\ \bar{f}_A(m) = \bar{f}_A\left(\left\lfloor \frac{m}{2} \right\rfloor\right) + 25,5, \quad m \geq 1, \end{cases} \quad (7.4.9)$$

решение в замкнутой форме может быть получено решением рекуррентного соотношения (7.4.9) методами, изложенными в главе 5, и, переходя к функции $\beta(m)$, получаем

$$\bar{f}_A(m : \beta(m) = n) = 25,5\beta(m) + 20,$$

что практически согласуется с (7.4.7).

§ 5. Алгоритм Карацубы умножения длинных целых чисел

Математическое обоснование алгоритма. Метод декомпозиции позволяет в целом ряде случаев, получить достаточно эффективные по асимптотической оценке рекурсивные алгоритмы. Для задачи умножения длинных целых чисел использование этого метода позволило А.А. Карацубе [7.2] впервые получить в 1962 г. алгоритм умножения двух n битовых чисел, имеющий асимптотическую оценку, лучшую, чем $\Theta(n^2)$. Изложим основные идеи этого алгоритма. Пусть a и b — два n -битовых числа в обычном двоичном представлении, т. е. старший бит расположен слева. Количество разрядов n таково, что $n = 2^k$ — это обеспечивает деление n пополам без остатка на всех рекурсивных вызовах. Обозначим через nh значение $n/2$. Поскольку метод декомпозиции предполагает разделение задачи, в данном случае на две равные части на любом уровне рекурсии, то представим числа a и b в виде

$$a = 2^{nh}a_2 + a_1, \quad b = 2^{nh}b_2 + b_1,$$

где a_1, a_2, b_1, b_2 — числа, имеющие длину в $nh = n/2$ бит (разрядов), причем a_2, b_2 — старшие nh разрядов чисел a и b . Произведение ab может быть записано в следующем виде (и это есть основная идея, предложенная А.А. Карацубой):

$$ab = 2^n a_2 b_2 + 2^{nh} ((a_1 + a_2)(b_1 + b_2) - (a_1 b_1 + a_2 b_2)) + a_1 b_1. \quad (7.5.1)$$

Произведения в (7.5.1) могут быть вычислены рекурсивно, но мы должны обеспечить умножение чисел, имеющих длину nh , это очевидно для чисел a_1, a_2, b_1, b_2 — они получены делением двух n -битовых чисел пополам. Однако числа $a_1 + a_2$ и $b_1 + b_2$ могут иметь $nh + 1$ двоичных разрядов. В этом случае мы можем представить их в виде суммы чисел длиной nh — a_3, b_3 , и однобитовых чисел — a_4, b_4 :

$$a_1 + a_2 = 2a_3 + a_4, \quad b_1 + b_2 = 2b_3 + b_4,$$

тогда

$$(a_1 + a_2)(b_1 + b_2) = 4a_3b_3 + 2a_4b_3 + 2a_3b_4 + a_4b_4, \quad (7.5.2)$$

где a_3b_3 есть произведение чисел длиной nh . Слагаемое a_4b_4 в (7.5.2) имеет длину один бит, а остальные слагаемые представляют собой произведения nh -битового и однобитового чисел, вычисление которых также не требует рекурсивных вызовов. Таким образом, мы свели

задачу умножения двух n -битовых чисел к умножению трех пар чисел, имеющих ровно $nh = n/2$ разрядов.

Вычислительная сложность алгоритма. Найдем вычислительную сложность алгоритма Карацубы, т. е. асимптотическую оценку функции трудоемкости, на основе следующих рассуждений. Очевидно, что пять сложений, одно вычитание и операции сдвига на n и $n/2$ разрядов, заданные формулой (7.5.1), требуют не более чем $\Theta(n)$ базовых операций, равно как и операции умножения на однобитовые числа, сложения и сдвиги на четыре и два разряда в формуле (7.5.2). В данном случае мы предполагаем, что числа настолько велики, что они хранятся в массивах, каждый элемент которого содержит один бит числа. Мы останавливаем рекурсию при значении $n = 1$, когда произведение ab вычисляется элементарно и требует не более чем фиксированного числа базовых операций, которое мы обозначим через C . Поскольку мы рекурсивно перемножаем три пары чисел половинной длины, то приведенные выше рассуждения позволяют записать рекуррентное соотношение для функции трудоемкости исследуемого алгоритма:

$$\begin{cases} f_A(1) = C; \\ f_A(n) = 3f_A(n/2) + \Theta(n). \end{cases} \quad (7.5.3)$$

Используя основную теорему о рекуррентных соотношениях (Бентли, Хакен, Сакс), мы немедленно получаем оценку

$$f_A(n) = \Theta\left(n^{\log_2 3}\right), \quad n^{\log_2 3} \approx n^{1.5849}, \quad (7.5.4)$$

что асимптотически лучше умножения «в столбик» с оценкой $\Theta(n^2)$.

В случае если $n \neq 2^k$, мы определяем k из неравенства $2^{k-1} < n \leq 2^k$, т. е. $k = \lceil \log_2 n \rceil$, и дополняем числа a и b до k разрядов нулями слева, получая следующую асимптотическую оценку:

$$n^{\log_2 3} = 3^{\log_2 n} \Rightarrow f_A(n) = \Theta\left(3^{\lceil \log_2 n \rceil}\right).$$

Существует возможность улучшения полученной асимптотической оценки этого алгоритма, связанная с разбиением чисел a и b на большее, чем два, количество слагаемых. Однако такой подход приводит к увеличению коэффициента, скрываемого Θ оценкой, и рассмотрение этих модификаций выходит за рамки данной книги. Укажем для любознательных читателей, что асимптотически лучший (с доказанной нижней границей задачи) алгоритм умножения длинных целых чисел, основанный на использовании быстрого преобразования Фурье, принадлежит Штрассену и Шенгаге [7.3] и имеет оценку $f_A(n) = \Theta(n \ln n \ln \ln n)$.

Сложение и вычитание длинных битовых чисел. Реализация алгоритма Карацубы предполагает наличие дополнительных алгоритмов для сложения и вычитания двух двоичных чисел, заданных поразрядно битовыми массивами. Рассмотрим вариант алгоритма сложения, ис-

пользующий специальную ячейку для хранения переноса в следующий разряд. В этой же ячейке мы формируем текущую сумму. Массивы **P** и **Q** содержат исходные l -разрядные двоичные числа, а результат формируется в $l + 1$ ячейках массива **S**.

Sum (l, P, Q, S)

(P, Q — исходные числа — массивы длиной l)

(S — массив результата длины l+1)

$sm \leftarrow 0$ (начальный перенос в следующий разряд) 1

For $i \leftarrow 1$ **downto** 1 $1+3 * 1$

$sm \leftarrow P[i] + Q[i] + sm$ $5 * 1$

$S[i] \leftarrow sm \bmod 2$ $3 * 1$

$sm \leftarrow sm \bmod 2$ $2 * 1$

end For

$S[0] \leftarrow sm$ 2

End.

Получим трудоемкость алгоритма **Sum** относительно длины исходных массивов l . Трудоемкость не зависит от значений бит, хранящихся в ячейках массивов, и на основании количества базовых операций в строках имеем

$$f_{sum}(l) = 1 + 1 + l(3 + 5 + 3 + 2) + 2 = 13l + 4. \quad (7.5.5)$$

Алгоритм вычитания может быть построен на основании сведения операции вычитания к сложению на основании следующих рассуждений. Пусть числа p и q имеют длину l бит, тогда очевидно, что сумма числа q и его инвертированного значения \bar{q} равна $2^l - 1$, и разность $p - q$ может быть записана в виде

$$p - q = p - (2^l - 1 - \bar{q}) = p + \bar{q} + 1 - 2^l. \quad (7.5.6)$$

Мы предполагаем, что $p \geq q$, что очевидно из (7.5.1). Тогда, поскольку инверсия выполняется побитно, а операция вычитания числа 2^l , имеющего длину $l + 1$, сводится к обнулению старшего $(l + 1)$ -го разряда суммы, то, алгоритм вычитания на основе (7.5.6) может быть записан следующим образом.

Sub (l, P, Q, S)

(P, Q — исходные числа — массивы длиной l)

(S — массив результата длины l+1, результат в $S[1..l]$)

$sm \leftarrow 1$ (начальный перенос в следующий разряд) 1

For $i \leftarrow 1$ **downto** 1 $1+3 * 1$

$sm \leftarrow P[i] + (1 - Q[i]) + sm$ $6 * 1$

$S[i] \leftarrow sm \bmod 2$ $3 * 1$

$sm \leftarrow sm \bmod 2$ $2 * 1$

end For

$S[0] \leftarrow 0$ 2

End.

Обратите внимание на то, что цикл не затрагивает старшего $(l + 1)$ -го разряда, и мы просто его обнуляем в последней строке. Получим

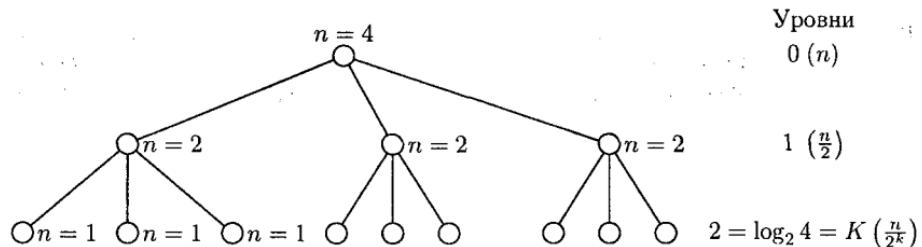


Рис. 7.5. Полное тернарное дерево рекурсии, порожденное алгоритмом Карацу́бы при $n = 4$

трудоемкость алгоритма **Sub** относительно длины исходных массивов l :

$$f_{sub}(l) = 1 + 1 + l(3 + 6 + 3 + 2) + 2 = 14l + 4. \quad (7.5.7)$$

Копирование и заполнение массивов. Еще два дополнительных алгоритма, которые нам будут необходимы — это копирование массива и обнуление массива. Ниже мы приводим запись этих алгоритмов с целью получения их функций трудоемкости.

Move (P, Q, l)

(Копирование l элементов из массива P в массив Q)

For $i \leftarrow 1$ to l

$Q[i] \leftarrow P[i]$

end For

End.

$1+3 * l$

$3 * l$

Трудоемкость алгоритма **Move** составляет

$$f_{move}(l) = 1 + l(3 + 3) = 6l + 1. \quad (7.5.8)$$

Fill (P, l, s)

(Заполнение l элементов массива P числом s)

For $i \leftarrow 1$ to l

$P[i] \leftarrow s$

end For

End.

$1+3 * l$

$2 * l$

Трудоемкость алгоритма **Fill** составляет

$$f_{fill}(l) = 1 + l(3 + 2) = 5l + 1. \quad (7.5.9)$$

Структура данных. Исследуемый рекурсивный алгоритм порождает полное тернарное дерево (см. рис. 7.5), содержащее $k + 1$ уровней, где $k = \log_2 n$, n — длина сомножителей.

Поскольку операнды и результаты передаются из уровня в уровень, и первый рекурсивный вызов порождает цепочку рекурсий, вплоть до чисел единичной длины, то мы должны обеспечить структуру данных, имеющую $k + 1$ уровней. Выбор между структурой, оптимальной по емкостной эффективности, и структурой, обеспечивающей простоту понимания и изложения материала, мы решаем в пользу последней.

На каждом уровне должно храниться произведение, сомножители, три результата умножения чисел половинной длины и две суммы — $a_1 + a_2$, $b_3 + b_4$. Мы предлагаем структуру данных для этого алгоритма в виде массива записей **D[1..k+1].name[0..M]**, где первый индекс есть номер рекурсивного уровня, **name** — одно из имен восьми массивов данного уровня, а **M** — длина результата **M=2*p**. Эта структура данных показана на рисунке 7.6.

		n	n+1	M
AB	результат умножения	A*B в ячейках 1..M		
A	первый операнд A	a_2		a_1
B	второй операнд B	b_2		b_1
A1B1			$a_1 * b_1$	
A2B2			$a_2 * b_2$	
A3B3			$a_3 * a_2 b_3$	
A1pA2	$(a_1+a_2)=2*a_3+a_4$		a_3	a_4
B1pB2	$(b_1+b_2)=2*b_3+b_4$		b_3	b_4

Рис. 7.6. Структура данных для алгоритма Карацубы

Исходные числа размещаются в **D[1].A[n+1..M]** и **D[1].B[n+1..M]**, а результат их умножения возвращается в **D[1].AB[1..M]**. Очередное упражнение, которое мы предлагаем читателям, состоит в разработке структуры данных, оптимальной по емкостной эффективности. Заметим, что идея динамического выделения и освобождения памяти при каждом рекурсивном вызове неконструктивна из-за резко возрастающего времени выполнения программной реализации за счет обращения к модулям операционной системы.

Алгоритм умножения длинных целых чисел. Рекурсивный алгоритм — процедура **KMul** получает на вход два длинных целых числа, размещенных в компонентах **A** и **B** структуры **D**, представленных в виде двух массивов длиной **N**. Элементы этих массивов — биты чисел **a** и **b**, которые расположены, начиная с правой границы компонента, имеющего индекс **M**. Таким образом, числа **a** и **b** размещены как обычные двоичные числа — крайний левый значащий бит является старшим. Уровень рекурсии, передаваемый как параметр процедуры, необходим для правильной индексации в выбранной структуре данных.

KMul(N, L)

(Умножение длинных целых:

(N — длина чисел, L — уровень рекурсии)

(Значение M — количество разрядов окончательного результата)

If N = 1 (проверка останова рекурсии) 1

then

D[L].AB[M] \leftarrow D[L].A[M] * D[L].B[M]

else

Nh \leftarrow N div 2 (половина длины) 2

11

2

$\text{NextL} \leftarrow L+1$	(номер следующего уровня)	2
$i1 \leftarrow M \cdot Nh + 1$	(индекс правой половины)	3
$i2 \leftarrow M \cdot N + 1$	(индекс левой половины)	3
	(произведение правых половин — $A1B1 = A1 \cdot B1$)	
Move(D[L].A[i1], D[NextL].A[i1], Nh)		6
Move(D[L].B[i1], D[NextL].B[i1], Nh)		6
KMul(Nh, NextL)	(1-ое рекурсивное умножение) (возврат результата)	6
Move(D[NextL].AB[i2], D[L].A1B1[i2], N)		6
	(произведение левых половин — $A2B2 = A2 \cdot B2$)	
Move(D[L].A[i2], D[NextL].A[i1], Nh)		6
Move(D[L].B[i2], D[NextL].B[i1], Nh)		6
KMul(Nh, NextL)	(2-ое рекурсивное умножение) (возврат результата)	6
Move(D[NextL].AB[i2], D[L].A2B2[i2], N)		6
	(вычисление суммы $A1pA2 = A1 + A2$)	
KSum(Nh, D[L].A[i1-1], D[L].A[i2-1], D[L].A1pA2[i1-1])		12
	(вычисление суммы $B1pB2 = B1 + B2$)	
KSum(Nh, D[L].B[i1-1], D[L].B[i2-1], D[L].B1pB2[i1-1])		12
	(вычисление произведения $A3B3 = A3 \cdot B3$)	
$A4 \leftarrow D[L].A1pA2[M]$	($a4$ — младший бит суммы)	4
$D[L].A1pA2[M] \leftarrow 0$	($A3 \cdot 2$)	4
$B4 \leftarrow D[L].B1pB2[M]$	($b4$ — младший бит суммы)	4
$D[L].B1pB2[M] \leftarrow 0$	($B3 \cdot 2$)	4
	(произведение $A3B3 = A3 \cdot B3$)	
Move(D[L].A1pA2[i1-1], D[NextL].A[i1], Nh)		7
	6(n/2)+1 [Move] + 16 [Call]	
Move(D[L].B1pB2[i1-1], D[NextL].B[i1], Nh)		7
	6(n/2)+1 [Move] + 16 [Call]	
KMul(Nh, NextL)	(3-ое рекурсивное умножение) (возврат результата с умножением на 4 — влево на два бита)	
Move(D[NextL].AB[i2], D[L].A3B3[i2-2], N)		7
	6(n)+1 [Move] + 16 [Call]	
	(обнуление младших битов справа от произведения)	
$D[L].A3B3[M] \leftarrow 0$		4
$D[L].A3B3[M-1] \leftarrow 0$		5
If $A4=1$		1
then		
	(обнулить старшие разряды в $B1pB2$ — выравнивание длин)	
Fill(D[L].B1pB2[i2-3], Nh+2, 0)		5
	5(2+n/2)+1 [Fc] + 16 [Call]	
	(сложение: $A3B3 \leftarrow A3B3 + 2 \cdot B3$)	
KSum(N+2, D[L].A3B3[i2-3], D[L].B1pB2[i2-3], D[L].A3B3[i2-3])		13

```

    13(2+n)+4 [Ksum] + 18 [Call]
end If                                         1
If B4=1
  then
    (обнулить старшие разряды в A1pA2 — выравнивание длин) 5
      Fill(D[L].A1pA2[i2-3],Nh+2,0)
    (прибавить A4B4 к A3)                                5
      D[L].A1pA2[M] ← A4
    (сложение: A3B3 ← A3B3+2 * A3)                      4
      KSum(N+2,D[L].A3B3[i2-3],D[L].A1pA2[i2-3],D[L].A3B3[i2-3]) 13
      13(2+n)+4 [Ksum] + 18 [Call]
    (Вычитание: A3B3 ← A3B3 - A1B1)
  end If
  D[L].A1B1[i2-1] ← 0                                 5
  D[L].A1B1[i2-2] ← 0                                 5
  KSub(N+2,D[L].A3B3[i2-3],D[L].A1B1[i2-3],D[L].A3B3[i2-3]) 13
  14(2+n)+4 [Ksub] + 18 [Call]
  (Вычитание: A3B3 ← A3B3-A2B2)
  D[L].A2B2[i2-1] ← 0                                 5
  D[L].A2B2[i2-2] ← 0                                 5
  KSub(N+2,D[L].A3B3[i2-3],D[L].A2B2[i2-3],D[L].A3B3[i2-3]) 13
  14(2+n)+4 [Ksub] + 18 [Call]
  (формирование результата)
  (в старшую половину компоненты A1B1 занести A2B2)
  Move(D[L].A2B2[i2],D[L].A1B1[M-2 * N+1],N)         9
  6(n)+1 [Move] + 16 [Call]
  (предварительно обнуляем результат — компонент AB)
  Fill(D[L].AB[M-2 * N],2 * N+1,0)                     7
  5(2 * n+1)+1 [Fc] + 16 [Call]
  (в середину компоненты AB занести A3B3)
  Move(D[L].A3B3[i2-1],D[L].AB[M-N-Nh],N+1)          10
  6(n+1)+1 [Move] + 16 [Call]
  (окончательный результат AB←AB+A1B1; AB←A1B1+A2B2+A3B3)
  KSum(2*N,D[L].AB[M-2*N],D[L].A1B1[M-2*N],D[L].AB[M-2*N]) 16
  13(2 * n)+4 [Ksum] + 18 [Call]
end If
End.

```

Обращаем внимание на трудоемкости, которые мы указали в строках после вызова внутренних процедур — значение n является текущей длиной обрабатываемого фрагмента и сокращается вдвое на каждом рекурсивном вызове. Обозначение [Call] относится к трудоемкости вызова/возврата процедуры, и зависит, что очевидно, от числа передаваемых параметров. Трудоемкости, указанные непосредственно в строке вызова процедур, отражают операции, включая индексацию в массиве и структуре, необходимые для вычисления передаваемых параметров. Заметьте, что трудоемкость алгоритма может быть улучшена за счет замены каждого вызова вспомогательных процедур на соответствующее тело — это приведет к увеличению длины текста алгоритма и потери

ясности, но любые улучшения ресурсной эффективности должны чем-то оплачиваться.

Анализ дерева рекурсии. Выполним подсчет вершин в дереве, порождаемым данным алгоритмом. Для этого алгоритма мы рассмотрим только случай, когда разрядность чисел a и b одинакова и равна степени двойки: $n = 2^k$, $k = \log_2 n$. Алгоритм, получая на входе два массива из n элементов (двоичные разряды исходных чисел), делит его ровно пополам при каждом вызове, и это деление продолжается рекурсивно вплоть до единичной длины. Поскольку каждое половинное деление приводит к трем рекурсивным вызовам, то мы имеем дерево рекурсий, представляющее собой полное тернарное дерево, содержащее $k + 1$ уровней; такое дерево для $n = 4$ показано на рисунке 7.5. Определим характеристики полного тернарного дерева глубины $k + 1$, используя рекурсивное задание функции $R(n)$. Формула для общего количества вершин может быть записана в виде следующего рекуррентного соотношения, при условии, что значение $k = \log_2 n$ есть целое число:

$$\begin{cases} R(1) = 1; \\ R(n) = 3R(n/2) + 1. \end{cases}$$

Нумеруя уровни дерева от 0 до k , можно определить $R(n)$, используя формулу для суммы геометрической прогрессии

$$R(n) = 1 + 3 + 9 + \dots + 3^k = \frac{3^{k+1} - 1}{3 - 1} = \frac{3}{2} \cdot 3^k - \frac{1}{2}, \quad k = \log_2 n.$$

В этом дереве на уровне с номером k расположено 3^k вершин, что доставляет решение для числа листьев дерева

$$R_L(n) = 3^k = 3^{\log_2 n} = n^{\log_2 3}.$$

Количество внутренних вершин, порождающих рекурсию, может быть определено как разность $R_V(n) = R(n) - R_L(n)$. В результате вызов данного алгоритма для умножения двух n -разрядных двоичных чисел порождает дерево рекурсии со следующими характеристиками:

$$\begin{aligned} R(n) &= \frac{3}{2}n^{\log_2 3} - \frac{1}{2}, & R_V(n) &= \frac{1}{2}n^{\log_2 3} - \frac{1}{2}, & R_L(n) &= n^{\log_2 3}, \\ H_R(n) &= 1 + \log_2 n, & B_L(n) &= \frac{2n^{\log_2 3}}{3n^{\log_2 3} - 1} \approx \frac{2}{3}. \end{aligned} \quad (7.5.10)$$

Анализ трудоемкости методом рекуррентных соотношений. Покажем, как можно получить функцию трудоемкости данного алгоритма, опираясь на результаты анализа дерева рекурсии и решения рекуррентного соотношения. Метод подсчета вершин дерева рекурсии включает три компонента для $f_A(n)$: $f_R(n)$, $f_{CL}(n)$ и $f_{CV}(n)$, которые мы и хотим получить в виде функций с неизвестными коэффициентами на основе результатов анализа дерева рекурсии. Трудоемкость обслу-

живания рекурсии определяется общим числом вершин и значением $f_R(1)$, которое мы обозначим через r :

$$f_R(n) = R(n) f_R(1) = r \left(\frac{3}{2} n^{\log_2 3} - \frac{1}{2} \right). \quad (7.5.11)$$

Обозначая $f_{CL}(1) = c_1$, получим общую трудоемкость в листьях дерева

$$f_{CL}(n) = R_L(n) f_{CL}(1) = c_1 \left(n^{\log_2 3} \right). \quad (7.5.12)$$

Функцию трудоемкости во внутренних вершинах $f_{CV}(n)$ мы получим, заменяя в рекуррентном соотношении (7.5.3) функцию $\Theta(n)$ на линейную функцию с неизвестными коэффициентами:

$$f_{CV}(n) = 3f_{CV}(n/2) + cn + d. \quad (7.5.13)$$

Общий вид $f_{CV}(n)$ получим суммированием функции $cn + d$ по всем вызовам во внутренних вершинах дерева рекурсии, т. е. по всем уровням от 0 до $k - 1$. Поскольку число n есть полная степень двойки, то последовательные деления на два дают нулевые остатки, трудоемкость во всех внутренних вершинах одного уровня одинакова:

$$\begin{aligned} f_{CV}(n) &= 1(cn + d) + 3\left(c\frac{n}{2} + d\right) + \dots + 3^{k-1}\left(c\frac{n}{2^{k-1}} + d\right) = \\ &= R_V(n)d + cn\left(1 + \frac{3}{2} + \dots + \left(\frac{3}{2}\right)^{k-1}\right) = R_V(n)d + 2cn\left(\frac{3^k}{2^k} - 1\right), \end{aligned}$$

но т. к. $2^k = 2^{\log_2 n} = n$, с учетом формулы (7.5.10) для $R_V(n)$ получаем

$$f_{CV}(n) = \left(2c + \frac{d}{2}\right)n^{\log_2 3} - 2cn - \frac{d}{2}. \quad (7.5.14)$$

Объединяя все компоненты, заданные формулами (7.5.11), (7.5.12) и (7.5.13), окончательно получаем общую формулу трудоемкости для алгоритма Карацубы:

$$\bar{f}_A(n) = n^{\log_2 3} \left(2c + \frac{d}{2} + c_1 + \frac{3}{2}r \right) - 2cn - \left(\frac{d+r}{2} \right). \quad (7.5.15)$$

Неизвестные коэффициенты в (7.5.15) мы можем получить на основе анализа текста алгоритма. Определим значение $r = f_R(1)$. Поскольку процедура **Kmul(N,L)** передает два параметра ($p = 2$) и имеет четыре локальных переменных, то

$$r = f_R(1) = 2(2 + 4 + 0 + 4 + 1) = 22,$$

трудоемкость останова рекурсии — сравнение и 11 операций, таким образом,

$$c_1 = f_{CL}(1) = 12.$$

Трудоемкость в среднем для внутренней вершины дерева определим относительно корня на основе анализа трудоемкости в строках текста

алгоритма. Мы предполагаем по вероятности, что $p(a_4 = 1) = 1/2$, и равноположено $p(b_4 = 1) = 1/2$. Суммируя количество базовых операций в строках фрагмента порождения рекурсии, получаем

$$\bar{f}_{CV}(n) = 140,5n + 683.$$

Подставляя полученные коэффициенты в (7.5.15), получаем окончательный вид функции трудоемкости в среднем для алгоритма умножения длинных целых чисел в случае, когда исходные числа имеют длину $n = 2^k$:

$$\bar{f}_A(n) = 667,5n^{\log_2 3} - 281n - 352,5, \quad (7.5.16)$$

и доля операций обслуживания дерева рекурсии составляет

$$F_R(n) = \frac{f_R(n)}{f_A(n)} = \frac{33n^{\log_2 3} - 11}{667,5n^{\log_2 3} - 281n - 352,5} \approx 0,0494, \quad n \rightarrow \infty. \quad (7.5.17)$$

Отметим, что для данного алгоритма доля обслуживания рекурсии для реальных размерностей вполне приемлема и не превышает 5%. Лучший и худший случай трудоемкости данного алгоритма могут быть легко оценены исходя из того, что на каждом рекурсивном вызове алгоритма во внутренних вершинах дерева младшие биты сумм $a_1 + a_2$, $b_1 + b_2 - a_4$, b_4 всегда равны или нулю (лучший случай) или единице (худший случай). Несложные выкладки показывают, что

$$\begin{aligned} f_A^\vee(n) &= 589n^{\log_2 3} - 250n - 305,0, \\ f_A^\wedge(n) &= 744,5n^{\log_2 3} - 312n - 398,5. \end{aligned} \quad (7.5.18)$$

Сложностной класс алгоритма. Полученные формулы для лучшего и худшего случаев трудоемкости свидетельствуют, что параметрический компонент влияет только на коэффициент при главном порядке, и, следовательно, алгоритм принадлежит к классу $NPRL$ и обладает хорошей временной устойчивостью. Сложностной класс алгоритма легко определяется на основе (7.5.18) — мы имеем полиномиальную, с показателем $\log_2 3$, зависимость трудоемкости от длины входа, что позволяет отнести алгоритм к классу πP .

Решение по рациональному выбору между алгоритмом Карацубы и простым умножением в столбик может быть принято на основе их детального анализа и коррекции коэффициентов с учетом экспериментального времени выполнения базовой операции. Данные ряда публикаций показывают, что алгоритм Карацубы эффективнее, начиная с длины сомножителей порядка 1500–2000 битов [7.4]. Отметим, что асимптотически оптимальный алгоритм Штрассена–Шенхаге реально эффективен, начиная с еще больших длин входа.

Емкостная эффективность — функция объема памяти. Оценим требуемый объем памяти в области стека, используя функцию $H_R(n)$. Формула оценки памяти в области программного стека известна:

$$V_{st}(n) = H_R(n)(p + r + f + l + 1),$$

данный алгоритм передает два параметра и использует четыре локальных переменных, что приводит к

$$V_{st}(n) = H_R(n)(2 + 4 + 0 + 4 + 1) = 11 \log_2 n + 11,$$

объем памяти используемой структуры данных может быть легко получен — фактически мы имеем трехмерный массив, количество двумерных слоев которого равно глубине рекурсии, таким образом,

$$V_{ram}(n) = 8(1 + 2n)(1 + \log_2 n) = 16n \log_2 n + 16n + 8 \log_2 n + 8,$$

$$V(n) = 16n \log_2 n + 27n + 8 \log_2 n + 19. \quad (7.5.19)$$

На основе (7.5.16) и (7.5.19) определяется ресурсная сложность алгоритма

$$\mathfrak{R}_c(n) = \left\langle \Theta\left(n^{\log_2 3}\right), \Theta(n \log_2 n) \right\rangle.$$

§ 6. Алгоритм фон Неймана сортировки массива чисел слиянием

Разработка рекурсивного алгоритма. Покажем, как метод декомпозиции может быть применен для решения задачи сортировки, исторически эта идея приписывается Дж. фон Нейману (информация из [7.3]). Напомним, что метод декомпозиции предписывает разделение задачи на части, решение подзадач и объединение решений. В применении к задаче сортировки этот метод приводит к разделению входного массива на две части — для четной длины они будут одинаковые, а для нечетной длины одна из частей массива будет на единицу больше. Затем мы рекурсивно вызываем алгоритм для сортировки полученных частей и объединяем возвращенные из двух рекурсивных вызовов отсортированные фрагменты массива в один сортированный массив. В классической реализации останов рекурсии происходит при единичной длине массива [7.3]. Существует целый ряд различных модификаций этого алгоритма, связанных с более ранним остановом рекурсии на основе сортировки фрагментов небольшой длины алгоритмом, более эффективным в этом диапазоне размерности. Рассмотрение этих модификаций выходит за рамки данной книги, мы рекомендуем заинтересованным читателям обратиться, например, к [7.4, 7.5].

Слияние отсортированных массивов. Собственно продуктивной частью данного алгоритма является слияние двух отсортированных массивов в один. Идея этого алгоритма состоит в том, что мы устанавливаем два указателя на первые элементы массивов, и, сравнивая элементы, заданные этими указателями, заносим в результатирующий

массив меньшее значение (при сортировке по возрастанию). Указатель перемещенного элемента сдвигается на единицу вправо, и цикл повторяется. Есть различные способы останова указателя на границе массива, мы выбрали для реализации способ «концевых заглушек» — в конец двух массивов записывается заранее известно большее значение, гарантирующее, что основной цикл выберет при слиянии все числа двух массивов, за исключением заглушек. Отметим, что массивы для слияния являются фрагментами основного сортируемого массива, в связи с чем нам необходимо переместить их вначале в два вспомогательных массива, т. к. результат должен быть помещен обратно в основной массив. Альтернативой является более трудоемкий алгоритм, реализующий слияние по месту, т. е. в основном массиве, и не требующий дополнительной памяти [7.3]. Заметим, что проблема выбора между этими алгоритмами является иллюстрацией классической дилеммы выбора между времененной и емкостной эффективностью.

Рассмотрим алгоритм слияния (**Merge**) отсортированных фрагментов массива **A**, расположенных в позициях между **r** и **r**, и **r+1** и **q**. Алгоритм использует дополнительные массивы **B_r** и **B_q**, в конце которых, с целью остановки указателей, помещаются заглушки. Вначале выполняется копирование отсортированных частей в **B_r** и **B_q**, а затем объединенный массив формируется непосредственно в массиве **A** между индексами **r** и **q**. Для удобства анализа в записи алгоритма справа указано количество базовых операций в данной строке.

Merge (A, p, r, q)

(формирование заглушки)

max $\leftarrow A[r]$	2
If Max < A[q]	2
then	
max $\leftarrow A[q]$	2
kp $\leftarrow r-p+1$	3
p1 $\leftarrow p-1$	2
For i $\leftarrow 1$ to kp	1+kp * 3
Bp[i] $\leftarrow A[p1+i]$	4
Bp[kp+1] $\leftarrow \text{max}$	3
kq $\leftarrow q-r$	2
For i $\leftarrow 1$ to kq	1+kq * 3
Bq[i] $\leftarrow A[r+i]$	4
Bq[kq+1] $\leftarrow \text{max}$	3
pp $\leftarrow 1$	1
pq $\leftarrow 1$	1
For i $\leftarrow p$ to q	1+m * 3
If Bp[pp] < Bq[pq]	3
then	
A[i] $\leftarrow Bp[pp]$	3
(kопирование в массивы Bp, Bq)	
(zаглушка)	
(слияние частей)	
(инициализация указателей)	

```

    pp ← pp + 1
else
    A[i] ← Bq[pq]
    pq ← pq + 1
end If

```

End.

Найдем трудоемкость данного алгоритма для объединенного массива, имеющего длину m . Пусть $m = kp + kq = q - p + 1$ есть длина объединенного массива. На основании указанного в строках количества операций и предположения о том, что строка $\max \leftarrow A[q]$ выполняется в среднем для половины обращений, можно получить трудоемкость алгоритма слияния отсортированных массивов как функцию длины массива результата:

$$\begin{aligned} \bar{f}_{\text{merge}}(m) &= 2 + 2 + 1 + 3 + 2 + 1 + 3kp + 4kp + 3 + 2 + 1 + 3kq + \\ &\quad + 4kq + 3 + 1 + 1 + 1 + 3m + m(3+5) = \\ &= 11m + 7(kp + kq) + 23 = 18m + 23. \quad (7.6.1) \end{aligned}$$

Заметим, что трудоемкость алгоритма слияния отсортированных массивов практически не зависит от данных, этот парадокс объясняется тем, что блоки в конструкции **if** $Bp[pp] < Bq[pq]$ содержат одинаковое количество операций, и, следовательно, вероятности выбора блоков **then** и **else**, очевидно зависящие от данных, не влияют на трудоемкость конструкции ветвления в целом. Разница между худшим и лучшим случаями равна двум операциям — строка $\max \leftarrow A[q]$ либо выполняется, либо обходится. В теории алгоритм принадлежит классу NPR , подклассу $NPRL$, но практически его можно считать алгоритмом класса N .

Алгоритм сортировки слиянием. Рекурсивный алгоритм **MSort** получает на вход массив **A**, и два индекса **r** и **q**, указывающие на ту часть массива, которая будет сортироваться при данном вызове. Запись этого алгоритма в виде рекурсивной процедуры на языке высокого уровня имеет вид

MSort(A, r, q)

If $p \neq q$	(проверка на останов рекурсии)	1
then		
$r \leftarrow (p+q) \text{ div } 2$	(середина массива)	3
Merge_Sort (A, p, r)	(сортировка левой части массива)	
Merge_Sort (A, r+1, q)	(сортировка правой части массива)	
Merge (A, p, r, q)	(слияние отсортированных частей)	

Return(A)**End.**

Первоначальный вызов данной процедуры для сортировки всего массива **A**, содержащего n элементов — **MSort(A, 1, n)**. Принадлежность этого алгоритма к одному из трудоемкостных классов мы определим после его анализа.

Анализ алгоритма методом подсчета вершин дерева рекурсий.

В соответствии с методом детального анализа дерева рекурсий выполним подсчет вершин в дереве рекурсивных вызовов алгоритма сортировки слиянием. Вначале рассмотрим случай, когда длина сортируемого массива есть степень двойки.

Случай 1. Длина входа $n = 2^k$, $k = \log_2 n$. Алгоритм, получая на входе массив из n элементов, делит его ровно пополам при первом вызове, и это деление продолжается рекурсивно вплоть до единичных элементов массива. В этом случае мы имеем дерево рекурсии, представляющее собой полное бинарное дерево, содержащее k уровней и n листьев; такое дерево для $n = 4$ показано на рисунке 7.7.

Определим характеристики полного бинарного дерева глубины k , содержащего n листьев. Общее количество вершин $R(n)$ может быть определено с использованием формулы для суммы геометрической прогрессии

$$R(n) = 1 + 2 + \dots + 2^k = 2 \cdot 2^k - 1 = 2n - 1.$$

Поскольку полное бинарное дерево содержит $n = 2^k$ листьев, то $R_L(n) = n$, количество внутренних вершин, порождающих рекурсию, равно $R_V(n) = n - 1$. Тем самым вызов данного алгоритма для сортировки массива длины n порождает дерево рекурсии со следующими характеристиками:

$$\begin{aligned} R(n) &= 2n - 1, & R_V(n) &= n - 1, & R_L(n) &= n, \\ H_R(n) &= 1 + \log_2 n, & B_L(n) &= \frac{n}{2n - 1}. \end{aligned} \quad (7.6.2)$$

Временная эффективность — функция трудоемкости. В соответствии с методом подсчета вершин дерева рекурсий определим вначале трудоемкость процедуры **MSort(A,p,q)** на один вызов/возврат $f_R(1)$. Поскольку процедура **MSort(A,p,q)** передает три параметра ($p = 3$), в стеке сохраняются значения четырех регистров ($r = 4$), ни одно

Уровни $-k$ Высота $H_R(n)$

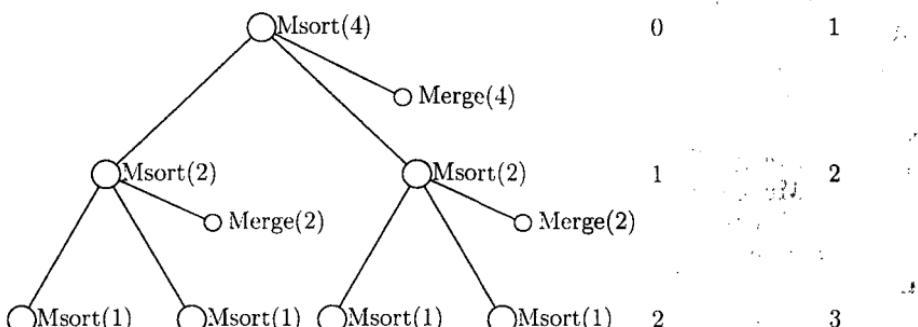


Рис. 7.7. Полное бинарное дерево рекурсии, порожденное алгоритмом сортировки слиянием при $n = 2^2 = 4$

значение не возвращается через имя процедуры ($f = 0$), — обращаем на это внимание, и процедура **MSort(A,p,q)** имеет одну локальную переменную r ($l = 1$), то в результате имеем

$$f_R(1) = 2(3 + 4 + 0 + 1 + 1) = 18,$$

и, следовательно, с учетом (7.6.2)

$$f_R(n) = R(n) f_R(1) = (2n - 1) 18 = 36n - 18. \quad (7.6.3)$$

Трудоемкость останова рекурсии включает в себя одно сравнение, таким образом $f_{CL}(1) = 1$, следовательно,

$$f_{CL}(n) = R_L(n) f_{CL}(1) = n \cdot 1 = n. \quad (7.6.4)$$

Во всех внутренних вершинах дерева (фрагмент рекурсивного вызова) трудоемкость включает в себя подготовку двух рекурсивных вызовов, вызов и возврат из процедуры **Merge(A,p,r,q)** — $f_{Msort}(v)$, и трудоемкость выполнения процедуры **Merge(A,p,r,q)** — $f_{Merge}(v)$. В связи с этим представим трудоемкость во внутренней вершине $f_{CV}(v)$ и суммарную трудоемкость внутренних вершин $f_{CV}(n)$ в виде следующих сумм:

$$f_{CV}(v) = f_{Msort}(v) + f_{Merge}(v),$$

$$f_{CV}(n) = f_{CV\ Msort}(n) + f_{CV\ Merge}(n).$$

Вычислим значение $f_{Msort}(v)$ — мы выполняем сравнение, вычисление середины длины, прибавление единицы (**r+1**) и, сохраняя значение переменной r ($l = 1$), передаем управление на процедуру **Merge(A,p,r,q)**, имеющую четыре параметра, и получаем управление обратно, таким образом

$$f_{Msort}(v) = 1 + 3 + 1 + 2(4 + 4 + 0 + 1 + 1) = 25,$$

и сумма $f_{Msort}(v)$ по всем внутренним вершинам составляет

$$f_{CV\ Msort}(n) = R_V(n) f_{Msort}(v) = (n - 1) \cdot 25 = 25n - 25. \quad (7.6.5)$$

Для анализируемого алгоритма сортировки функция $g(v_j)$, фигурирующая в формуле для метода подсчета вершин, в данном случае есть трудоемкость слияния в вершине v_j : $g(v_j) = f_{Merge}(v_j)$. Для рассматриваемого случая на фиксированном уровне рекурсивного дерева слиянию подвергаются массивы одинаковой длины. Это значительно упрощает вычисление суммы

$$\sum_{j=1}^{R_V(n)} g(v_j),$$

поскольку одинаковые слагаемые, связанные с одним уровнем, могут быть объединены. Поскольку трудоемкость алгоритма слияния для массива длины m составляет в соответствии с формулой (7.6.1) $18m + 23$ и алгоритм вызывается $R_V(n) = n - 1$ раз с разными длинами

объединяемых фрагментов массива на каждом уровне дерева, то значения $g(v_j)$ имеют вид

$$\begin{cases} g(v_1) = 18n + 23; \\ g(v_2) = g(v_3) = 18(n/2) + 23; \\ g(v_4) = g(v_5) = g(v_6) = g(v_7) = 18(n/4) + 23; \\ \dots \end{cases}$$

Суммируя $g(v_j)$ по всем внутренним вершинам дерева, имеем

$$\begin{aligned} f_{CV\ Merge}(n) &= \sum_{j=1}^{R_V(n)} g(v_j) = \\ &= 23R_V(n) + 18n + 2 \cdot 18(n/2) + 4 \cdot 18(n/4) + \dots, \end{aligned}$$

учитывая, что таким образом обрабатываются все уровни дерева, кроме последнего, который не содержит внутренних вершин, т. е. k уровней рекурсивного дерева с номерами от 0 до $k - 1$ (см. рис. 7.6), получаем

$$f_{CV\ Merge}(n) = 18nk + 23(n - 1) = 18n \log_2 n + 23n - 23. \quad (7.6.6)$$

Учитывая все компоненты (см. формулы (7.6.3)–(7.6.6)), получаем окончательный вид функции трудоемкости для алгоритма сортировки слиянием в случае $n = 2^k$:

$$\begin{aligned} \bar{f}_A(n) &= f_R(n) + f_{CL}(n) + f_{CV\ Msort}(n) + f_{CV\ Merge}(n) = \\ &= 36n - 18 + n + 25n - 25 + 18n \log_2 n + 23n - 23 = \\ &= 18n \log_2 n + 85n - 66, \quad (7.6.7) \end{aligned}$$

и доля операций обслуживания дерева рекурсии составляет

$$F_R(n) = \frac{f_R(n)}{\bar{f}_A(n)} = \frac{36n - 18}{18n \log_2 n + 85n - 66}. \quad (7.6.8)$$

Значения $F_R(n)$ медленно убывают с ростом длины массива, так при $n = 1024$ $F_R(n) \approx 0,135$, а при $n = 1048576$ $F_R(n) \approx 0,08$, но для данного алгоритма доля обслуживания рекурсии для реальных размерностей уже находится в пределах 8–15%.

Мы получили функцию трудоемкости в среднем (7.6.7), очевидно согласующуюся с результатом оценки главного порядка $\Theta(n \log_2 n)$, полученным по теореме Бентли, Хакен, Сакса. На основе этого результата мы можем говорить, что алгоритм относится к классу πP , т. е. является полиномиальным по длине входа. Заметим, что асимптотически он лучше, чем простые квадратичные алгоритмы сортировки, а область его рационального применения по отношению к другим алгоритмам сортировки можно получить на основе методики сравнительного анализа, изложенной в [7.9].

Лучший и худший случай трудоемкости данного алгоритма могут быть легко оценены исходя из того, что каждый вызов процедуры

слияния при вычислении заглушки либо выполняет, либо пропускает 2 операции. Поскольку процедура слияния выполняется для каждой внутренней вершины дерева, то имеем

$$\begin{aligned} f_A^V(n) &= \bar{f}_A(n) - 1 \cdot R_V(n) = 18n \log_2 n + 84n - 67. \\ f_A^L(n) &= \bar{f}_A(n) + 1 \cdot R_V(n) = 18n \log_2 n + 86n - 65. \end{aligned}$$

Полученные результаты позволяют говорить об очень слабой зависимости трудоемкости от данных, алгоритм принадлежит к классу *NPR*L и обладает очень хорошей временной устойчивостью.

Емкостная эффективность — функция объема памяти. Оценим требуемый объем памяти в области стека, используя функцию $H_R(n)$. Формула оценки памяти стека известна:

$$V_{st}(n) = H_R(n) (p + r + f + l + 1).$$

Данный алгоритм требует в процедуре слияния двух дополнительных массивов длиной $1 + n/2$, передает три параметра, использует одну локальную переменную и семь в процедуре слияния, таким образом, требуемая память составляет

$$\begin{aligned} V(n) &= 2(1 + n/2) + 8 + V_{st}(n) = \\ &= n + 10 + H_R(n)(3 + 4 + 0 + 1 + 1) = n + 9 \log_2 n + 19, \quad (7.6.9) \end{aligned}$$

а его ресурсная сложность в худшем случае имеет вид

$$\mathfrak{R}_c^V(A) = \langle \Theta(n \log_2 n), \Theta(n) \rangle.$$

Случай 2. Длина входа $2^{k-1} < n < 2^k$, $k = [\log_2 n]$. Алгоритм, получая на входе массив из n элементов, делит его на две равные части при четном n , и части, отличающиеся друг от друга на единицу, при нечетном значении n . Такое деление продолжается рекурсивно вплоть до единичных элементов массива. В этом случае мы имеем дерево рекурсии, представляющее собой бинарное дерево, содержащее k уровней, из которых только $k - 1$ уровней являются полными; такое дерево для значения $n = 11$ показано на рисунке 7.8.

Определим характеристики этого бинарного дерева. Для этого, пользуясь методом математической индукции, докажем, что формулы

$$R(n) = 2n - 1, \quad R_V(n) = n - 1, \quad R_L(n) = n, \quad (7.6.10)$$

остаются верными и для случая неполного дерева. Базис индукции — если $n = 2$, то дерево содержит одну внутреннюю вершину и два листа — $R(2) = 3$, $R_V(2) = 1$, $R_L(2) = 2$, что согласуется с (7.6.10). Пусть формулы (7.6.10) верны для n , докажем, что они верны и для $n + 1$. Нетрудно заметить, что увеличение n на единицу приводит к замене одного листа на внутреннюю вершину с двумя листьями, таким образом, $R_V(n + 1) = n - 1 + 1 = n$, а $R_L(n + 1) = n - 1 + 2 = n + 1$, что и доказывает правильность формулы (7.6.10) для произвольного значения n . Поскольку высота дерева на единицу больше

номера последнего уровня (напомним, что уровни нумеруются, начиная с нуля), то

$$H_R(n) = 1 + k = 1 + \lceil \log_2 n \rceil, \quad B_L(n) = \frac{n}{2n - 1}. \quad (7.6.11)$$

Поскольку формулы для характеристик дерева рекурсии не изменились, то связанные с ними формулы для компонентов трудоемкости (7.6.3)–(7.6.5) также остаются в силе. Изменения в рассматриваемом случае касаются только трудоемкости слияния во внутренних вершинах

$$f_{CV\ Merge}(n) = \sum_{j=1}^{R_V(n)} g(v_j).$$

Введем нумерацию внутренних вершин по уровням дерева и обозначим через $n_j^{(k)}$ длину фрагмента массива, обрабатываемого в j -ой внутренней вершине k -го уровня, а через $v_j^{(k)}$ — саму эту вершину (см. рис. 7.8). С учетом того, что $g(v_j^{(k)}) = 18n_j^{(k)} + 23$, искомая сумма может быть записана следующим образом:

$$f_{CV\ Merge}(n) = \sum_{j=1}^{R_V(n)} (18n_j^{(k)} + 23) = 23R_V(n) + \sum_{j=1}^{R_V(n)} 18n_j^{(k)}. \quad (7.6.12)$$

Для нас представляет интерес последняя сумма в (7.6.12). Мы уже отмечали, что дерево рекурсии в рассматриваемом случае является полным, за исключением последнего уровня. Обозначим через m номер последнего полного уровня — $m = \lceil \log_2 n \rceil$, тогда гарантированно предыдущий уровень с номером $m - 1$ не содержит листьев,

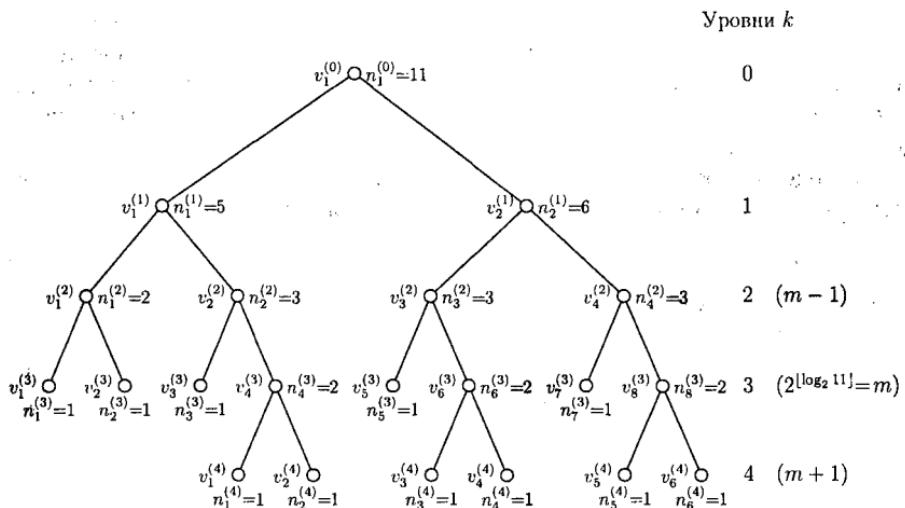


Рис. 7.8. Дерево рекурсии, порожденное алгоритмом сортировки слиянием при $n = 11$

а последний $(m + 1)$ -й уровень содержит только листья. На уровне m могут находиться как листья, так и внутренние вершины (см. рис. 7.8). Нас интересует количество внутренних вершин на уровне m , заметим, что поскольку дерево является бинарным, а на уровне $m + 1$ находятся только листья, то $n_j^{(m)} = 2$. Поскольку увеличение n на единицу приводит к созданию новой внутренней вершины, а для полного бинарного дерева все вершины последнего уровня являются листьями, то на уровне m будет находиться $n - 2^m$ внутренних вершин. Эти рассуждения позволяют записать последнюю сумму в (7.6.12) в виде двух сумм: суммы по всем $m - 1$ уровням, содержащим только внутренние вершины, и суммы по внутренним вершинам уровня m :

$$\sum_{j=1}^{R_V(n)} 18n_j^{(k)} = \sum_{k=0}^{m-1} \sum_{j=1}^{2^k} 18n_j^{(k)} + \sum_{j=1}^{n-2^m} 18n_j^{(m)}.$$

Поскольку сумма объединяемых фрагментов массива на полных уровнях рекурсивного дерева всегда равна полной длине массива n (надеемся, что это утверждение не вызовет сомнений у наших читателей), а $n_j^{(m)} = 2$, то результат имеет вид

$$\sum_{j=1}^{R_V(n)} 18n_j^{(k)} = 18nm + 18(n - 2^m) \cdot 2,$$

и в итоге мы получаем

$$f_{CV\ Merge}(n) = 23R_V(n) + 18nm + 18(n - 2^m) \cdot 2,$$

поскольку $m = \lfloor \log_2 n \rfloor$, а $R_V(n) = n - 1$, то окончательно

$$f_{CV\ Merge}(n) = 18n \lfloor \log_2 n \rfloor + 59n - 36 \cdot 2^{\lfloor \log_2 n \rfloor} - 23.$$

Учитывая компоненты трудоемкости, задаваемые формулами (7.6.3)–(7.6.5), получим трудоемкость алгоритма сортировки слиянием в среднем для случая $n \neq 2^k$:

$$\begin{aligned} \bar{f}_A(n) &= f_R(n) + f_{CL}(n) + f_{CV\ Msor}(n) + f_{CV\ Merge}(n) = \\ &= 36n - 18 + n + 25n - 25 + f_{CV\ Merge}(n) = \\ &= 18n \lfloor \log_2 n \rfloor + 121n - 36 \cdot 2^{\lfloor \log_2 n \rfloor} - 66. \end{aligned} \quad (7.6.13)$$

Как и в случае 1, разница между полученной трудоемкостью в среднем, и худшим и лучшем случаями равна $n - 1$.

Для данного алгоритма мы не будем приводить детального решения для получения трудоемкости методом рекуррентных соотношений, отметим только, что полученная формула (7.6.13) является решением соотношения

$$\begin{cases} f_A(1) = a; \\ f_A(n) = f_A(\lfloor n/2 \rfloor) + f_A(\lceil n/2 \rceil) + bn + c \end{cases}$$

с конкретными коэффициентами a, b, c , определение значений которых в соответствии с методом рекуррентных соотношений мы оставляем читателю, и напоминаем, что общий случай решения рекуррентных соотношений такого типа был рассмотрен в главе 5.

§ 7. Генетический алгоритм эвристического поиска экстремума функции нескольких переменных

Введение в генетические алгоритмы. Практическая необходимость решения целого ряда NP -полных задач в оптимизационной постановке и задач поиска экстремума целевых функций при проектировании и исследовании сложных систем привела разработчиков алгоритмического обеспечения к использованию биологических механизмов для эвристического поиска наилучших решений. В настоящее время результаты, полученные в рамках научного направления, которое можно назвать «природные вычисления» [7.6], позволяют разрабатывать алгоритмы, достаточно эффективные как по качеству получаемых решений, так и по временным оценкам их получения. Это направление объединяет такие разделы, как генетические алгоритмы, эволюционное программирование, нейросетевые вычисления, клеточные автоматы и ДНК-вычисления [7.6], муравьиные алгоритмы [7.7]. Исследователи обращаются к природным механизмам, которые миллионы лет обеспечивают адаптацию биоценозов к окружающей среде. Одним из таких механизмов, имеющих фундаментальный характер, является механизм наследственности. Его использование для решения задач оптимизации привело к появлению генетических алгоритмов. В живой природе особи в биоценозе конкурируют друг с другом за различные ресурсы. Те особи, которые более приспособлены к окружающим условиям, будут иметь больше шансов на создание потомства. Слабо приспособленные либо не произведут потомства, либо их потомство будет очень немногочисленным. Это означает, что гены от более приспособленных особей будут распространяться в последующих поколениях. Комбинация хороших характеристик от различных родителей иногда может приводить к появлению потомка, приспособленность которого в данном биоценозе больше, чем приспособленность его родителей. Таким образом, популяция в целом развивается, все лучше и лучше приспособливаясь к среде обитания. Алгоритм решения задач оптимизации, основанный на идеях наследственности в биологических популяциях, был впервые предложен Джоном Холландом (1975 г.). Он получил название репродуктивного плана Холланда и широко использовался как базовый алгоритм в эволюционных вычислениях. Дальнейшее развитие, как, собственно, и свое название — генетические алгоритмы, эти идеи получили в работах Гольдберга и Де Йонга [7.6, 7.8]. Цель генетического алгоритма при решении задачи оптимизации состоит в том, чтобы найти лучшее возможное, но не гарантированно оптимальное решение. Для реализации генетического алгоритма необходимо выбрать подхо-

дящую структуру данных для представления решений. В постановке задачи поиска экстремума экземпляр этой структуры должен содержать информацию о некоторой точке в пространстве решений.

Терминология и структура данных. Структура данных генетического алгоритма состоит из набора *хромосом*. Хромосома, как правило, представляет собой битовую строку, так что термин строки часто заменяет понятие «хромосома». Вообще говоря, хромосомы генетических алгоритмов не ограничены только бинарным представлением, известны другие реализации, построенные на векторах вещественных чисел [7.8]. Несмотря на то, что для многих реальных задач, видимо, больше подходят строки переменной длины, в настоящее время структуры фиксированной длины наиболее распространены и изучены. Для иллюстрации идеи ограничимся битовыми строками. В этом случае каждая хромосома (строка) представляет собой последовательное объединение ряда подкомпонентов, которые называются *генами*. Гены расположены в различных позициях или *локусах* хромосомы и принимают значения, называемые *аллелями* — это биологическая терминология. В представлении хромосомы бинарной строкой ген является битом этой строки, локус есть позиция бита в строке, а аллель — это значение гена, 0 или 1. Биологический термин *генотип* относится к полной генетической модели особи и соответствует структуре в генетическом алгоритме. Термин *фенотип* относится к внешним наблюдаемым признакам и соответствует вектору в пространстве параметров задачи. В генетике под *мутацией* понимается преобразование хромосомы, случайно изменяющее один или несколько генов. Наиболее распространенный вид мутаций — случайное изменение только одного из генов хромосомы. Термин *кроссинговер* обозначает порождение из двух хромосом двух новых путем обмена генами. В литературе по генетическим алгоритмам употребляются также термины *скрецивание*, *кроссовер* или *рекомбинация*. В простейшем случае кроссинговер в генетическом алгоритме реализуется так же, как и в биологии. При скрецивании хромосомы разрезаются в случайной точке и обмениваются частями между собой. Например, если хромосомы (11, 12, 13, 14) и (0, 0, 0, 0) разрезать между вторым и третьим генами и обменять их части, то получатся следующие потомки — (0, 0, 13, 14) и (11, 12, 0, 0).

Проблемы и особенности применения. Основная проблема, связанная с применением генетических алгоритмов, обусловлена их эвристическим характером. Говоря более строго, какова вероятность достижения популяцией глобального оптимума в заданной области при данных настройках алгоритма? В настоящее время на этот вопрос не существует строгого ответа и теоретически обоснованных оценок. Имеются предположения, что генетический алгоритм может стать эффективной процедурой для поиска оптимального решения задачи при условиях, если пространство поиска достаточно велико, и предполагается, что целевая функция не является гладкой и унимодальной в области поиска; а также в случаях, если задача не требует нахождения глобального

оптимума, поиск которого влечет значительные вычислительные затраты, а состоит в необходимости достаточно быстро найти приемлемое «хорошее» решение — условие, которое довольно часто встречается в реальных задачах.

Если целевая функция обладает свойствами гладкости и унимодальности, то любой градиентный метод, такой, например, как метод наискорейшего спуска, будет более эффективен. Генетический алгоритм является в определенном смысле универсальным методом, т. е. он явно не учитывает специфику задачи или должен быть каким-то образом специально на нее настроен. Поэтому, если мы имеем некоторую дополнительную информацию о целевой функции и пространстве поиска (как, например, для хорошо известной задачи коммивояжера), то методы поиска, использующие эвристики, определяемые задачей, часто превосходят любой универсальный метод. С другой стороны, при достаточно сложном рельефе целевой функции градиентные методы с единственным решением могут останавливаться в локальном экстремуме. Наличие у генетических алгоритмов целой «популяции» решений, совместно с вероятностным механизмом мутации, позволяют предполагать меньшую вероятность блуждания в окрестности локального оптимума и большую эффективность работы на многоэкстремальном ландшафте.

Сегодня генетические алгоритмы успешно применяются для получения эвристических решений классических NP -трудных задач, задач оптимизации в пространствах с большим количеством измерений, ряда экономических задач оптимального характера [7.6, 7.8, 7.9]. Практика их применения показывает, что в ряде случаев решения, получаемые генетическими алгоритмами, достаточно близки к оптимальным или иногда даже совпадают с ними [7.8].

Базовый генетический алгоритм поиска оптимальных решений. Приведем простой иллюстративный пример [7.8] — задачу максимизации некоторой функции двух переменных $f(x_1, x_2)$, при ограничениях: $0 \leq x_1 \leq 1$ и $0 \leq x_2 \leq 1$. Обычно, методика кодирования вещественных переменных x_1 и x_2 состоит в преобразовании их в двоичные целочисленные строки определенной длины, достаточной для того, чтобы обеспечить желаемую точность. Предположим, что 10-разрядное кодирование достаточно для x_1 и x_2 . Установить соответствие между генотипом и фенотипом можно, разделив соответствующее двоичное целое число на $2^{10} - 1$. Например, 0000000000 соответствует 0, а 1111111111 соответствует $1023/1023$ или 1. Оптимизируемая структура данных есть 20-битовая строка, представляющая собой конкатенацию (объединение) кодировок для x_1 и x_2 . Пусть переменная x_1 размещается в крайних левых 10 битах строки, тогда как x_2 размещается в правой части генотипа особи. Таким образом, генотип представляет собой одну точку в 20-мерном целочисленном пространстве (вершину единичного гиперкуба), точки которого и исследуются генетическим

алгоритмом. Для этой задачи фенотип будет представлять собой точку x в двумерном пространстве аргументов — $x = (x_1, x_2)$.

Чтобы решить задачу оптимизации, нужно задать некоторую меру качества для каждой структуры в пространстве поиска. Для этой цели используется функция приспособленности. При максимизации целевая функция часто сама выступает в качестве функции приспособленности, для задач минимизации целевая функция инвертируется и смещается в область положительных значений. Рассмотрим фазы работы базового генетического алгоритма [7.9]. Вначале случайным образом генерируется начальная популяция (набор хромосом). Работа алгоритма представляет собой итерационный процесс, который продолжается до тех пор, пока не будет смоделировано заданное число поколений или выполнен некоторый критерий останова. В каждом поколении реализуется пропорциональный отбор приспособленности, одноточечная рекомбинация и вероятностная мутация. Пропорциональный отбор реализуется путем назначения каждой особи (хромосоме) i , $i = 1, n$ вероятности $P(i)$, равной отношению ее приспособленности к суммарной приспособленности популяции (по целевой функции):

$$P(i) = \frac{f(i)}{S}, \quad S = \sum_{i=1}^n f(i).$$

Затем происходит отбор (с замещением) всех n особей для дальнейшей генетической обработки, согласно убыванию величины $P(i)$. Простейший пропорциональный отбор реализуется с помощью рулетки → случайного пропорционального выбора (Гольдберг, 1989 г.). При этом «колесо» рулетки содержит по одному сектору для каждого члена популяции, а длина i -го сектора пропорциональна значению $P(i)$. При таком отборе члены популяции, обладающие более высокой приспособленностью, будут чаще выбираться по вероятности. После этого выбранные n особей подвергаются рекомбинации с заданной вероятностью P_c , при этом n хромосом случайным образом разбиваются на $n/2$ пар. Для каждой пары с вероятностью P_c может быть выполнена рекомбинация. Если рекомбинация происходит, то полученные потомки заменяют собой родителей. Одноточечная рекомбинация работает следующим образом. Случайным образом выбирается точка разрыва, т. е. участок между соседними битами в строке. Обе родительские структуры разрываются на два сегмента по этому участку. Затем соответствующие сегменты различных родителей склеиваются и получаются два генотипа потомков. После стадии рекомбинации выполняется фаза мутации. В каждой строке, которая подвергается мутации, каждый бит инвертируется с вероятностью P_m . Популяция, полученная после мутации, записывается поверх старой и на этом завершается цикл одного поколения в генетическом алгоритме.

Полученное новое поколение обладает (по вероятности) более высокой приспособленностью, наследованной от «хороших» представителей предыдущего поколения. Таким образом, из поколения в поколение, хо-

ющие характеристики распространяются по всей популяции. Скрещивание наиболее приспособленных особей приводит к тому, что исследуются наиболее перспективные участки пространства поиска. В результате популяция будет сходиться к локально оптимальному решению задачи, а иногда, благодаря мутации, может быть, и к глобальному оптимуму. Базовый генетический алгоритм может быть представлен в виде последовательности следующих шагов:

1. Создать начальную популяцию

2. Цикл по поколениям, пока не выполнено условие останова

(цикл жизни одного поколения)

- 2.1. Оценить приспособленность каждой особи
- 2.2. Выполнить отбор по приспособленности
- 2.3. Случайным образом разбить популяцию на две группы пар
- 2.4. Выполнить фазу вероятностной рекомбинации (кроссинговера)
для пар популяции и заменить родителей
- 2.5. Выполнить фазу вероятностной мутации
- 2.6. Оценить приспособленность новой популяции и вычислить
условие останова
- 2.7. Объявить потомков новым поколением

3. Конец цикла по поколениям

Практическая реализация данного алгоритма требует ответа, по крайней мере, на два вопроса — какова начальная численность популяции, и как может быть сформулировано условие останова? Второй вопрос требует тщательного анализа, т. к. «слабое» условие быстро приведет к ближайшему локальному экстремуму, а «сильное» — к рождению значительного числа поколений, вплоть до цикла с непредсказуемым временем выполнения.

Варианты генетических алгоритмов. Очевидно, что тонкая настройка базового генетического алгоритма может быть выполнена путем изменения значений вероятностей рекомбинации и мутации, существует много исследований и предложений в данной области, (см., например, [7.6, 7.8, 7.10]). В настоящее времялагаются разнообразные модификации генетических алгоритмов в части методов отбора по приспособленности, рекомбинации и мутации [7.8, 7.10]. Приведем несколько примеров. Метод турнирного отбора (Бриндел, 1981 г.; Гольдберг и Деб, 1991 г. [7.8]) реализуется в виде n турниров для выборки n особей. Каждый турнир состоит в выборе k элементов из популяции и отбора лучшей особи среди них. Элитные методы отбора (Де Йонг, 1975 г.) гарантируют, что при отборе обязательно будут выживать лучший или лучшие члены популяции. Наиболее распространена процедура обязательного сохранения только одной лучшей особи. Двухточечная рекомбинация (Гольдберг, 1989 г.) и равномерная рекомбинация (Сисверда, 1989 г.) являются вполне достойными альтернативами одноточечному оператору. При двухточечной рекомбинации выбираются две точки разрыва, и родительские хромосомы обмениваются сегментом, который находится между двумя этими точками. Равномерная рекомбинация предполагает, что каждый бит

первого родителя наследуется первым потомком с заданной вероятностью; в противном случае этот бит передается второму потомку. Механизмы мутаций могут быть так же заимствованы из молекулярной биологии, например, обмен концевых участков хромосомы (механизм транслокации), обмен смежных сегментов (транспозиция) [7.10]. По мнению автора, интерес представляет также механизм инверсии, т. е. перестановки генов в хромосоме, управляющим параметром при этом может выступать инверсионное расстояние — минимальное количество единичных инверсий генов, преобразующих исходную хромосому в мутированную [7.10].

Математическое описание алгоритма. При разработке рекурсивного генетического алгоритма будем опираться на изложенные выше принципы одноточечной рекомбинации (кросинговера) со случайно выбранной точкой разрыва и одноточечной мутации со случайным выбором локуса. Под хромосомой будем понимать вектор вещественных чисел, тем самым, предлагаемый алгоритм является генетическим алгоритмом без битового кодирования. Будем считать для определенности, что задача поиска экстремума функции формулируется как задача поиска минимума вещественноизначной функции нескольких переменных в ограниченной области. Формально: пусть в некоторой области G m -мерного евклидова пространства E^m задана функция $g(\mathbf{x})$, $\mathbf{x} \in E^m$, $\mathbf{x} = (x_1, \dots, x_m)$ со значениями в $R - g(\mathbf{x}) : G \subseteq E^m \rightarrow R$. Будем считать, что возможные значения аргумента \mathbf{x} ограничены в E^m единичным кубом с началом в нуле. Таким образом, область G , в которой ищется минимум функции $g(\mathbf{x})$, задана следующим образом:

$$G = \{ \mathbf{x} \mid \mathbf{x} \in E^m, \forall i = \overline{1, m} \quad 0 \leq x_i \leq 1 \}.$$

Введем в рассмотрение следующие функции — Rnd , значением которой является случайное вещественное число из полуинтервала $[0, 1]$ и $\text{Rnd}(k)$, значением которой является случайное целое число из сегмента $[1, k]$, при этом числа, генерируемые этими функциями, имеют равномерное распределение. На их основе определим при $1 \leq k \leq m$ функцию $\zeta(k, m)$ со значениями в E^m следующим образом: вначале зададим множество

$$J = \{ j_i \mid j_i = \text{Rnd}(m), j_i \neq j_l, i \neq l \}, \quad |J| = k,$$

которое будем рассматривать как множество случайных несовпадающих индексов координат точки, тогда точка $\mathbf{x} = \zeta(k, m)$ в пространстве E^m имеет координаты: $\mathbf{x} = (x_1, \dots, x_i, \dots, x_m)$, определяемые в соответствии с формулой

$$x_i = \begin{cases} 0, & i \notin J \\ \text{Rnd}, & i \in J \end{cases}.$$

Содержательно функция $\zeta(k, m)$ генерирует точку в единичном кубе пространства E^m с началом в нуле, у которой k случайно выбранных координат являются случайными вещественными числами из полу-

интервала $[0, 1)$, а остальные равны нулю. Отметим, что точка $\mathbf{x} = \zeta(m, m)$, полученная обращением к функции ζ с первым параметром равным m , есть точка единичного куба, все координаты которой являются случайными вещественными числами. Определим следующие операции над точками \mathbf{x} пространства E^m , которые являются генетическими преобразованиями этих точек. При этом точка, имеющая m координат, рассматривается как хромосома, имеющая m локусов, гены в которых являются вещественными числами. Операция одноточечной мутации:

$$M(\mathbf{x}) : \mathbf{x} \rightarrow \tilde{\mathbf{x}}, \quad \text{где} \quad \tilde{\mathbf{x}} = (\mathbf{x} + \zeta(1, m)) \bmod 1,$$

мы используем обозначение $y \bmod 1$ для дробной части вещественного числа при покомпонентном сложении координат точки \mathbf{x} и точки $\zeta(1, m)$. Очевидно, что k -точечная мутация определяется аналогично с использованием функции $\zeta(k, m)$, определяющей как номера локусов, подвергаемых мутации, так и случайные значения, добавляемые (по $\bmod 1$) к соответствующим генам (координатам точки \mathbf{x}). Операция левого (C_L) кроссинговера:

$$C_L(\mathbf{x}_1, \mathbf{x}_2) : (\mathbf{x}_1, \mathbf{x}_2) \rightarrow \mathbf{x}_L,$$

при этом

$$\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_L \in E^m,$$

$$\mathbf{x}_L : k = \text{Rnd}(m - 1), \forall i = \overline{1, k} \ x_{Li} = x_{2i}, \forall i = \overline{k + 1, m} \ x_{Li} = x_{1i},$$

и операция правого (C_R) кроссинговера:

$$C_R(\mathbf{x}_1, \mathbf{x}_2) : (\mathbf{x}_1, \mathbf{x}_2) \rightarrow \mathbf{x}_R,$$

при этом

$$\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_R \in E^m,$$

$$\mathbf{x}_R : k = \text{Rnd}(m - 1), \forall i = \overline{1, k} \ x_{Ri} = x_{1i}, \forall i = \overline{k + 1, m} \ x_{Ri} = x_{2i}.$$

Таким образом, две новые хромосомы получаются перекрещиванием (кроссинговером) двух исходных с разрезом в случайной точке (индексе координаты). Вполне возможно, что, унаследовав разные «хорошие» половины хромосом родителей, эти потомки окажутся более «приспособленными».

Общая идея предлагаемого алгоритма состоит в том, что, используя рекурсию для порождения бинарного дерева популяции исследуемых точек, на шагах рекурсивного возврата по известным точкам $\mathbf{x}_1, \mathbf{x}_2$ с использованием операций мутации и кроссинговера строятся 6 дополнительных точек

$$\mathbf{x}_L, \mathbf{x}_R, M(\mathbf{x}_1), M(\mathbf{x}_2), M(\mathbf{x}_L), M(\mathbf{x}_R),$$

и вверх по дереву рекурсии возвращается та точка x^* , в которой функция g имеет минимальное значение:

$$x^* = \arg \min \left\{ \begin{array}{l} g(x_1), g(x_2), g(x_L), g(x_R), \\ g(M(x_1)), g(M(x_2)), g(M(x_L)), g(M(x_R)) \end{array} \right\}. \quad (7.7.1)$$

Начальные особи популяции, т. е. точки в листьях бинарного дерева рекурсии, могут быть получены с использованием функции $\zeta(m, m)$.

Схема генетического алгоритма. На основе сформулированных выше предложений рекурсивный генетический алгоритм поиска экстремума (минимума) функции $g(x)$ может быть построен следующим образом — рекурсивная функция $F(n, x)$ возвращает в F значение экстремума (минимума) функции $g(x)$, определяемое рекурсивно по всем n уровням дерева популяции, а в x — координаты точки, в которой функция $g(x)$ имеет минимальное из всех исследованных точек значение, т. е. координаты точки эвристического минимума, найденного с использованием формулы (7.7.1). Значение n — число уровней бинарного дерева популяции задается пользователем. Это значение определяет глубину дерева рекурсии, листьями которого являются точки $x \in E^m$ со случайными координатами $x = \zeta(m, m)$, принадлежащие единичному кубу с началом в нуле. Приведем схему данного алгоритма в виде рекурсивной функции:

F(n,x)

(Рекурсивный генетический алгоритм поиска минимума функции,)
(n — глубина дерева рекурсии, x — эвристическое решение,)
($E=g(x)$ — значение функции в точке x)

If $n=1$

then

$x=Rnd$ (Точка со случайными координатами в листе дерева)

else

$gx1=F(n-1,x)$ (Рекурсивный вызов — левое поддерево)

$x1=x$ (Наилучшая точка левого поддерева)

$gx2=F(n-1,x)$ (Рекурсивный вызов — правое поддерево)

$x2=x$ (Наилучшая точка правого поддерева)

$xL=CL(x1,x2)$ (Левый кроссинговер $x1,x2$)

$xR=CR(x1,x2)$ (Правый кроссинговер $x1,x2$)

$Mx1=M(x1)$ (Одноточечные мутации возвращенных точек)

$Mx2=M(x2)$

$MxL=M(xL)$ (Мутации точек, полученных кроссинговером)

$MxR=M(xR)$

$x=\arg \min(g(x1), g(x2), g(xL), g(xR),$

$g(Mx1), g(Mx2), g(MxL), g(MxR))$

(Оптимальная точка x — та, в которой значение $g(x)$)

является наименьшим из восьми исследованных точек)

end If

$F=g(x)$ (Значение функции в точке x)

(наилучшей для этой вершины рекурсивного дерева)

Return(F,x)

(Возврат в точку вызова координат точки x и значения F)

End.

Рассмотрим вызов этой функции с $n = 3$. Вызов $F(3, x)$ порождает два вызова $F(2, x)$, которые, в свою очередь, порождают по два вызова $F(1, x)$. Обращение к рекурсивной функции $F(n, x)$ с первым параметром, равным единице, приводит к генерации точки со случайными координатами. Возврат двух точек из обращений к $F(1, x)$ в тело $F(2, x)$ приводит к выполнению операций кроссинговера и мутаций — на основе двух полученных точек порождается дополнительно еще шесть. Наилучшая точка, в смысле минимума целевой функции, возвращается в тело вызова $F(3, x)$ по левой ветви дерева рекурсии. Аналогично формируется результат, возвращаемый $F(2, x)$ по правой ветви дерева (второе обращение). В теле функции $F(3, x)$ две наилучшие точки, полученные двумя вызовами $F(2, x)$, подвергаются кроссинговеру и мутации, и функция возвращает координаты точки, в которой исследуемая функция имеет наименьшее значение.

Отметим, что различные вызовы данного алгоритма приведут, возможно, к получению различных результатов — алгоритм не гарантирует глобального оптимума, а генерируемые при каждом вызове случайные совокупности начальных точек будут различны. Аналогия с популяцией выглядит следующим образом — цепочка рекурсивных вызовов порождает регулярную структуру популяций, в которой уровень листьев является начальной популяцией, а каждая следующая образуется из лучших особей предыдущей популяции и новых особей, полученных путем кроссинговера и мутаций. Цепочка рекурсивных возвратов реализует отбор лучших особей предыдущих популяций, при этом в каждой внутренней вершине дерева выбирается одна лучшая особь.

Возможные модификации алгоритма. Этот рекурсивный алгоритм допускает простые модификации, среди которых, помимо очевидных модификаций генетических операций, можно выделить следующие:

- построение рекурсивного дерева популяции в виде n -арного дерева, что может быть реализовано путем добавления строк рекурсивного вызова;
- выбор в каждой вершине, за исключением корневой, нескольких лучших особей (точек) из популяции, порожденной данной вершиной, что реализуется передачей вверх по дереву рекурсии координат нескольких точек в окрестности локального экстремума (в смысле точек данной вершины дерева популяции);
- использование направленной генерации начальных точек популяции в листьях рекурсивного дерева, учитывающей определенным образом специфику локализаций экстремума в конкретной задаче.

Вычислительная сложность алгоритма. Найдем вычислительную сложность, т. е. асимптотическую оценку функции трудоемкости предложенного алгоритма. Очевидно, что эта оценка определяется общим количеством точек (особей популяций), исследуемых алгоритмом и трудоемкостью их обработки. Для каждой точки алгоритм вычисляет

значение функции $g(\mathbf{x})$ и выполняет одну из генетических операций, включая случайное порождение. Введем обозначения:

$P(n)$ — количество точек, порожденных во всех n уровнях дерева;

$f_g(m)$ — трудоемкость вычисления функции $g(\mathbf{x})$;

$f_{gen}(m)$ — трудоемкость генетических операций.

В результате получаем

$$f_A(n, m) = \Theta(P(n)(f_g(m) + f_{gen}(m))), \quad (7.7.2)$$

где $f_A(n, m)$ — трудоемкость исследуемого алгоритма. Отметим, что в данном случае трудоемкость обслуживания дерева рекурсии не определяет главный порядок в оценке трудоемкости вычислений, и поэтому не учитывается, равно как и вычисление \mathbf{x} по формуле (7.7.1), требующее фиксированного числа операций. Фиксируем значение размерности пространства m , замечая, что все генетические операции не более чем однократно обрабатывают все координаты точки \mathbf{x} , и, считая, что трудоемкость функции Rnd составляет $\Theta(1)$, что справедливо для линейных конгруэнтных генераторов [7.1], получаем оценку

$$f_{gen}(m) = \Theta(m). \quad (7.7.3)$$

Для определения количества точек, исследуемых алгоритмом, введем в рассмотрение рекурсивно заданную функцию $P(n)$. Поскольку в листе дерева порождается только одна точка, само дерево является бинарным и в каждой внутренней вершине исследуется дополнительно 6 точек, имеем линейное неоднородное рекуррентное соотношение первого порядка

$$P(n) = \begin{cases} 1, & n = 1; \\ 2P(n-1) + 6, & n \geq 2. \end{cases} \quad (7.7.4)$$

В соответствии с теорией, изложенной в главе 1, будем искать решение этого рекуррентного соотношения в виде $P(n) = a \cdot 2^n + b$. Подставляя это решение в (7.7.4), имеем $a \cdot 2^n + b = 2(a \cdot 2^{n-1} + b) + 6$, откуда $b = -6$, а начальное значение $P(1) = 1$ позволяет определить a

из уравнения $a \cdot 2^1 - 6 = 1$, откуда

$$a = 7/2, \Rightarrow P(n) = 7 \cdot 2^{n-1} - 6. \quad (7.7.5)$$

Количество точек, исследуемых алгоритмом, в зависимости от параметра n , приведено в таблице 7.2. Отметим, что этот параметр задается пользователем и определяет общее число точек, исследуемых алгоритмом в заданной области.

Подставляя (7.7.3) и (7.7.5) в (7.7.2), окончательно получаем искомую оценку трудоемкости (сложность) рекурсивного генетического алгоритма

$$\begin{aligned} f_A(n, m) &= \Theta((7 \cdot 2^{n-1} - 6)(f_g(m) + \Theta(m))) = \\ &= \Theta(2^n(f_g(m) + m)), \end{aligned} \quad (7.7.6)$$

n	$P(n)$
18	917 498
19	1 835 002
20	3 670 010
21	7 340 026
22	14 680 058

таким образом, трудоемкость данного алгоритма не зависит от особенностей локализации экстремумов исследуемой функции в области определения, и, следовательно, предлагаемый алгоритм обладает свойством временной устойчивости. Отличительной особенностью данного алгоритма является зависимость его трудоемкости только от размерности пространства, в котором задана исследуемая функция, и количества уровней (поколений) в рекурсивном дереве популяции, заданных пользователем.

Сложностной класс алгоритма. Алгоритм относится к классу PR , поскольку, имея на входе только одно числовое значение, задает различное количество базовых операций, которое при фиксированной размерности пространства m зависит только от значения параметра n , заданного пользователем. Трудоемкость алгоритма определяется, в соответствии с (7.7.6), значением $n = f_A(n) = \Theta(2^n)$, а не количеством бит в его представлении, и, следовательно, алгоритм принадлежит подклассу $ExPR$. Если мы рассматриваем битовую длину входа алгоритма $m : 2^{m-1} \leq n < 2^m$, то в этом аспекте он принадлежит сложностному классу πF , т. к. является дважды экспоненциальным по m .

Структура данных. Исследуемый рекурсивный алгоритм порождает полное бинарное дерево, содержащее n уровней, где n — задаваемый пользователем параметр. Значение n определяет также и количество точек, исследуемых алгоритмом — их будет $P(n)$. Но мы уже знаем, что дерево рекурсии порождается левым обходом, и, следовательно, в каждый момент времени на некотором уровне дерева будет активна только одна вершина. Мы предлагаем следующую структуру данных для этого алгоритма, основанную на том, что алгоритм ищет экстремум функции m переменных, т. е. точка x есть точка m -мерного пространства, и в каждой внутренней вершине дерева обрабатывается 8 точек. Структура данных представляет собой трехмерную матрицу $Vx[1..n, 1..8, 0..m]$, первый индекс которой соответствует текущему уровню дерева рекурсии, второй — номеру исследуемой точки ($i = 1..8$) в данной вершине дерева, а третий пробегает значения координат точки $x_i = (x_{i1}, \dots, x_{im})$, и по нулевому значению третьего индекса мы храним значение целевой функции $g(x_i)$. Обращаем внимание наших читателей на то, что в конкретной реализации значение m является константой. Эта структура данных показана на рисунке 7.9.

Используемая структура не является оптимальной по затратам памяти, мы используем ее, прежде всего, для простоты понимания алго-

	0	1	M
x_1	$g(x_1)$	x_1	x_m
x_2	$g(x_2)$		
x_3	$g(x_3)$		
x_4	$g(x_4)$		
x_5	$g(x_5)$		
x_6	$g(x_6)$		
x_7	$g(x_7)$		
x_8	$g(x_8)$		

Рис. 7.9. Структура данных для рекурсивного генетического алгоритма поиска экстремума

ритма. Насколько можно сократить используемую память, и как может быть построена оптимальная структура хранения данных для этого алгоритма? — эти вопросы мы оставляем в качестве упражнений для наших читателей.

Генератор случайных чисел. В целях анализа трудоемкости исследуемого алгоритма необходимо знать трудоемкость генерации псевдослучайного числа. Мы приводим наиболее распространенный генератор, использующий линейный конгруэнтный метод [7.1], без указания конкретных настроек коэффициентов. Заметим, что качество получаемой псевдослучайной последовательности существенно определяется значениями a, b, N и запускающим значением z_0 . Функция возвращает псевдослучайное число в полуинтервале $[0, 1)$.

Rnd $(z — \text{глобальная переменная})$

$z \leftarrow (z * a + b) \bmod N$

4

$Rnd \leftarrow z/N$

2

End.

Трудоемкость этого алгоритма (как функции, возвращающей значение по имени), в базовых операциях принятой нами модели вычислений, на один вызов/возврат составляет

$$f_{Rnd} = 2(0 + 4 + 1 + 0 + 1) + 4 + 2 = 18.$$

Рекурсивный алгоритм. Рассмотрим рекурсивный алгоритм, реализующий идеи генетического поиска минимума многомерной функции, в виде рекурсивной процедуры **F(n)**. Выше, в общем случае, мы рассматривали рекурсивную функцию, однако предложенная структура данных позволяет хранить значения целевой функции, и реализация в виде процедуры представляется нам более целесообразной. Останов рекурсии в этой процедуре происходит при значении $n = 1$ и сводится к генерации случайной точки в m -мерном пространстве. Некоторых пояснений требует расположение результатов — процедура на уровне k возвращает значение целевой функции в **Vx[k,1,0]**, координаты найденной оптимальной точки в **Vx[k,1,1..m]**, — это первая строка двумерной матрицы уровня k , а окончательный результат возвращается в **Vx[n,1,0]** и **Vx[n,1,1..m]**.

F(n)

If ($n=1$)

1

then

For $i \leftarrow 1$ **to** m

$1+3m$

Vx[n,1,i] ← Rnd

$(4+18)m$

Vx[n,1,0] ← g(n,1)

$4+f(g)$

else

F(n-1)

1

(1-ый рекурсивный вызов-результат в **Vx[n,1,0..m]**)

For $i \leftarrow 0$ **to** m

$1+3(m+1)$

Vx[n,1,i] ← Vx[n-1,1,i]

$8(m+1)$

F(n-1)

1

(2-ой рекурсивный вызов-результат в $Vx[n,2,0..m]$)

For $i \leftarrow 0$ **to** m $1+3(m+1)$
 $Vx[n,2,i] \leftarrow Vx[n-1,1,i]$ $8(m+1)$
 (кроссинговер координат полученных точек)

$k \leftarrow 2+Rnd(m-2)$ $3+18$
 (k — точка разрыва)

For $i \leftarrow 1$ **to** k $1+3k$
 $Vx[n,3,i] \leftarrow Vx[n,2,i]$ $7k$
 $Vx[n,4,i] \leftarrow Vx[n,1,i]$ $7k$

For $i \leftarrow k+1$ **to** m $2+3(m-k)$
 $Vx[n,3,i] \leftarrow Vx[n,1,i]$ $7(m-k)$
 $Vx[n,4,i] \leftarrow Vx[n,2,i]$ $7(m-k)$
 (копирование для последующей мутации)

For $j \leftarrow 1$ **to** 4 $1+3 * 4$
For $i \leftarrow 1$ **to** m $(1+3m) * 4$
 $Vx[n,j+4,i] \leftarrow Vx[n,j,i]$ $8 * 4m$
 (мутации хромосом 5-8)

For $j \leftarrow 5$ **to** 8 $1+3 * 4$
 $k \leftarrow 1+Rnd(m)$ $(2+18) * 4$
 $Vx[n,j,k] \leftarrow Vx[n,j,k]+Rnd$ $(8+18) * 4$
If $Vx[n,j,k] > 1$ $4 * 4$
 then
 $Vx[n,j,k] \leftarrow Vx[n,j,k]-1$ $8 * 4 * 0,5$
 (расчет значений функции для точек 3-8)

For $j \leftarrow 3$ **to** 8 $1+3 * 6$
 $Vx[n,j,0] \leftarrow g(n,j)$ $(4+f(G)) * 6$
 (поиск минимума)

$k \leftarrow 1$ 1
 $Gmin \leftarrow Vx[n,1,0]$ 4

For $j \leftarrow 2$ **to** 8 $1+3 * 7$
If $Vx[n,j,0] < Gmin$ $4 * 7$
 then
 $k \leftarrow j$ 1
 $Gmin \leftarrow Vx[n,j,0]$ 4
 (формирование результата)

For $i \leftarrow 0$ **to** m $1+3 * (m+1)$
 $Vx[n,1,i] \leftarrow Vx[n,k,i]$ $7 * (m+1)$

end If

End.

Анализ трудоемкости алгоритма методом рекуррентных соотношений. Для данного алгоритма мы получим функцию трудоемкости методом рекуррентных соотношений и надеемся, что читатели смогут получить аналогичный результат методом подсчета вершин дерева рекурсии, что мы и оставляем в качестве упражнения. Тем не менее, характеристики дерева рекурсии представляют для нас интерес. Данный алгоритм с параметром n порождает, очевидно, полное бинарное дерево рекурсии с n уровнями (см. рис. 7.10), которое имеет следующие

характеристики:

$$\begin{aligned} R(n) &= 2^n - 1, \quad R_V(n) = 2^{n-1} - 1, \quad R_L(n) := 2^{n-1}, \\ H_R(n) &= n, \quad B_L(n) = \frac{2^{n-1}}{2^n - 1}. \end{aligned} \quad (7.7.7)$$

Общая трудоемкость данного алгоритма включает в себя и затраты на вычисление целевой функции, однако, поскольку эти затраты определяются только в конкретном применении, мы их учтем в общем виде в результирующей формуле. Заметим, что при конкретном применении размерность пространства фиксирована, и, следовательно, мы можем использовать общую формулу метода рекуррентных соотношений для случая фиксированных значений трудоемкости, поскольку $f_{CV}(v)$ не зависит от n . Для исследуемого алгоритма общая формула имеет вид

$$\begin{cases} f_A(1) = f_R(1) + f_{CL}(1); \\ f_A(n) = 2f_A(n-1) + f_R(1) + f_{CV}(v), \quad n > 1. \end{cases} \quad (7.7.8)$$

Определим трудоемкость процедуры **F(p)** на один вызов/возврат $f_R(1)$. Поскольку передается один параметр ($p = 1$), в стеке сохраняются значения четырех регистров ($r = 4$), процедура имеет 4 локальных переменных (**i,j,k,Gmin**), и, следовательно, $l = 4$, то

$$f_R(1) = 2(1 + 4 + 0 + 4 + 1) = 20.$$

Трудоемкость останова рекурсии $f_{CL}(1)$ включает в себя одно сравнение, генерацию m случайных координат точки \mathbf{x} , и вычисление целевой функции, в итоге

$$f_{CL}(1) = 1 + 1 + 3m + m(4 + 18) + 4 = 25m + 6.$$

Трудоемкость во всех внутренних вершинах дерева не зависит от номера уровня, и, чтобы не утруждать читателей расписыванием компонентов суммы, мы, ссылаясь на указанное в строках процедуры

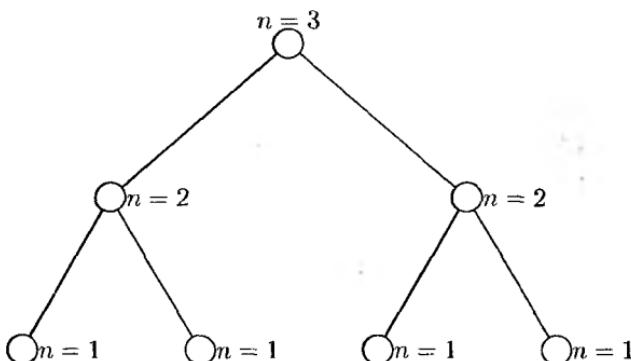


Рис. 7.10. Полное бинарное дерево рекурсии, порожденное рекурсивным генетическим алгоритмом при $n = 3$

F(n) число базовых операций, приводим окончательный результат для $f_{CV}(v)$ в среднем:

$$\bar{f}_{CV}(v) = 93m + 401 + 5H(8) \approx 93m + 414,59.$$

Слагаемое $5H(8)$ отражает трудоемкость фрагмента присваивания при поиске минимума целевой функции, поскольку в среднем общее количество переприсваиваний определяется значением гармонического числа от количества элементов массива [7.3] — $H(k)$. Мы ищем минимум из 8 чисел, значение $H(8) \approx 2,718$, а трудоемкость фрагмента равна 5 операциям.

Решение для рекуррентного соотношения

$$\begin{cases} f_A(1) = g; \\ f_A(n) = 2f_A(n-1) + h, & n > 1, \end{cases}$$

где g, h — константы, не зависящие от аргумента n , может быть получено методами, изложенными в главе 1:

$$f_A(n) = \frac{g+h}{2} \cdot 2^n - h,$$

так как в соответствии с (7.7.8)

$$\begin{aligned} g &= 20 + 25m + 6 = 25m + 26, \\ h &= 20 + 93m + 414,59 = 93m + 434,59, \end{aligned}$$

то получаем функцию трудоемкости данного алгоритма в среднем

$$\bar{f}_A(n, m) = 59m \cdot 2^n + 230,295 \cdot 2^n - 93m - 434,59. \quad (7.7.9)$$

Поскольку для каждой исследуемой точки, а всего их будет $P(n)$, алгоритм вычисляет значение целевой функции, то, обозначая трудоемкость вычисления $g(x)$ через $f_g(m)$, можно получить полную трудоемкость данного алгоритма в среднем:

$$\bar{f}_{A\Sigma}(n, m) = \bar{f}_A(n, m) + P(n)f_g(m),$$

где $P(n)$ определяется по формуле (7.7.5).

Емкостная эффективность — функция объема памяти. Для получения оценки емкостной эффективности используем функцию $H_R(m)$, определяющую наибольшую глубину дерева рекурсии. Общая формула для оценки памяти стека имеет вид

$$V_{st}(n) = H_R(n)(p + r + f + l + 1).$$

Поскольку данный алгоритм требует 4 локальных переменных, то

$$V_{st}(n) = H_R(n)(1 + 4 + 0 + 4 + 1) = 10H_R(n) = 10n. \quad (7.7.10)$$

Объем глобальной трехмерной матрицы определяется элементарно

$$V_s(n, m) = n \cdot 8(m+1) = 8mn + 8n,$$

таким образом, общие затраты памяти с учетом (7.7.10) составят

$$V(n, m) = 8mn + 18n,$$

а ресурсная сложность алгоритма имеет вид

$$\mathfrak{R}_c(n, m) = \langle \Theta(2^n(m + f_g(m))), \Theta(mn) \rangle.$$

§ 8. Алгоритм Тарьяна поиска оставного дерева в графе

Идея рекурсивного алгоритма. При решении целого ряда задач, моделируемых с использованием графов, необходимо найти оставное дерево графа G . Наиболее характерный пример — проверка связности графа. Другая характерная подзадача, возникающая во многих алгоритмах обработки графов — проверка всех ребер хотя бы по одному разу. Простое решение этих задач, основанное на рекурсии, было предложено Р. Тарьяном в 1972 г. Алгоритм получил название «поиск в глубину» (DFS) за счет того, что он пытается построить наиболее глубокое оставное дерево графа, в отличие от другого подхода, строящего наиболее широкое оставное дерево. За подробностями и нюансами мы рекомендуем читателям обратиться к оригинальной статье Тарьяна [7.11] и многочисленным изложениям этого алгоритма, например, в [7.3, 7.12, 7.13].

Идея алгоритма достаточно проста — она состоит в том, чтобы исследовать все ребра исходного графа, помечая при этом те вершины, которые уже обработаны. Поясним это более подробно, основываясь на представлении неориентированного графа матрицей смежности. Вызов данного алгоритма с некоторой вершиной в качестве начальной приводит к пометке этой вершины и ее последующей обработке. Относительно текущей в данный момент вершины графа алгоритм, прежде всего, проверяет, какие вершины являются смежными с текущей, обрабатывая тем самым все, инцидентные данной вершине, ребра. Для каждой смежной вершины проверяется, была ли она ранее помечена, и если нет, то соответствующее ребро включается в оставное дерево, и происходит рекурсивный вызов алгоритма с найденной смежной вершиной в качестве параметра. Именно этот вызов и послужил поводом для названия алгоритма Тарьяна — «поиск в глубину». Мы еще не обработали все вершины, смежные с данной, а вызываем алгоритм рекурсивно для первой непомеченной смежной вершины. Для полного графа на n вершинах (все вершины полного графа являются смежными) алгоритм Тарьяна построит остов, представляющий собой унарное дерево глубины n , в то время как алгоритм поиска в ширину — $(n - 1)$ -арное дерево глубиной 2. Эта ситуация проиллюстрирована на рисунке 7.11.

Схема алгоритма. Реализация изложенной выше идеи достаточно проста — нам, помимо исходной матрицы смежности, необходим мас-

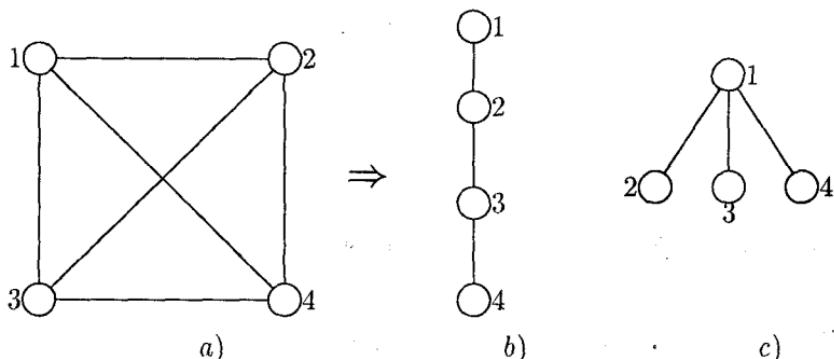


Рис. 7.11. Полный граф на четырех вершинах (а) и его оставные деревья, полученные алгоритмом поиска в глубину (б) и в ширину (с)

сив пометок вершин и структура, в которой будет формироваться результат — оставное дерево исходного графа. Приведем схематическую запись данного алгоритма в виде рекурсивной процедуры **DFS(V)**. Мы предполагаем, что изначально структура **T**, хранящая оставное дерево, пуста, массив пометок обнулен, а граф **G** представлен матрицей смежности **G[1..n,1..n]**.

DFS(V)

```

Visit[V] ← 1      (пометка вершины)
For W ← 1 to n    (цикл по строке матрицы смежности)
  If (G[V,W]=1)   (проверка смежности вершины W)
    then
      If (Visit[W]=0)
        (проверка непомеченной вершины)
        then
          T ← T + Edge(V,W)
          (добавление ребра к T)
          DFS(W)
          (рекурсивный вызов для проверки вершины W)
        end If
      end If
    end For
End.

```

Обращаем внимание читателей на то, что мы впервые в этом алгоритме сталкиваемся с рекурсивным вызовом, расположенным внутри цикла, что создает определенные трудности при его анализе.

Доказательство правильности алгоритма DFS. Построение доказательства правильности этого алгоритма может быть основано на следующих замечаниях. Мы приводим эти замечания, имея ввиду, что граф **G** является связным. Вообще говоря, алгоритм Тарьяна строит некоторый граф **T**, добавляя к нему некоторые ребра из графа **G**. Мы только предполагаем, что результат работы алгоритма представляет собой оставное дерево графа **G**. Алгоритм добавляет очередное ребро

к формируемому графу T , тогда и только тогда, если одна из вершин этого ребра еще не просмотрена. Это замечание гарантирует, что формируемый алгоритмом граф T не может содержать циклов. Обратим внимание на то, что первая вершина будет помечена этим алгоритмом сразу, на первом вызове. Добавление ребра в T происходит для каждой непомеченной вершины, а последующий рекурсивный вызов алгоритма гарантирует пометку этой вершины. На основании этого можно сделать вывод, что ни одно ребро не будет добавлено в T дважды. Если каждая из $n - 1$ вершин графа G , отличных от начальной вершины, действительно проверена, то алгоритм добавляет в T ровно $n - 1$ ребер. Осталось указать, что, по определению, оставной подграф графа G , не имеющий циклов и содержащий $n - 1$ ребер, является оставным деревом графа G .

Вычислительная сложность алгоритма. Найдем вычислительную сложность алгоритма Тарьяна, используя следующие рассуждения. Вне зависимости от структуры графа, при условии его связности, общее количество вызовов алгоритма будет равно количеству вершин графа — n , из которых $n - 1$ вызовов будут рекурсивными. Это объясняется тем фактом, что алгоритм будет работать до тех пор, пока останется хотя бы одна непомеченная вершина, и для каждой непомеченной вершины будет выполнен рекурсивный вызов. Заметим, что в данном случае глубина рекурсии определяется структурой графа и может меняться от 2 до n . Тем не менее, на каждом вызове мы выполняем цикл поиска смежных вершин по строке матрицы смежности с очевидной оценкой в $\Theta(n)$ базовых операций. Поскольку мы имеем n вызовов, то оценка трудоемкости имеет вид

$$f_A(n) = n \Theta(n) = \Theta(n^2). \quad (7.8.1)$$

Сравнение If (**Visit[W]=0**) будет выполнено дважды для всех ребер (v, w) исходного графа в первый раз, когда текущей вершиной алгоритма будет первая вершина ребра v , а во второй раз, когда текущей будет вершина w . Поскольку в полном графе количество ребер имеет оценку $\Theta(n^2)$, то мы не ухудшаем (7.8.1), тем более, мы не ухудшаем эту оценку и во фрагменте добавления ребра в оставное дерево, имеющего, очевидно, оценку $\Theta(n)$ по всем вызовам. Таким образом, при задании графа матрицей смежности вычислительная сложность алгоритма Тарьяна составляет $\Theta(n^2)$, т. е. он является квадратичным по числу вершин графа. Отметим, что в данном случае для анализа вычислительной сложности мы использовали подход, при котором рассматривается совокупное поведение алгоритма на всех вызовах, а не его трудоемкость для конкретной вершины. Такой подход носит название амортизационного анализа и достаточно подробно описан в [7.3].

Структура данных. Исследуемый рекурсивный алгоритм требует трех глобальных массивов — двумерного массива **G[1..n, 1..n]**, хра-

нящего матрицу стоимости, одномерного массива **Visit[1..n]**, содержащего пометки вершин, и структуры, хранящей оставное дерево — мы будем использовать для этого массив, состоящий из двух столбцов — **T[1..n,1..2]**. Очевидно, что массив пометок вершин должен быть обнулен при запуске алгоритма, также как и счетчик ребер оставшегося дерева. Конечно, для неориентированного графа матрица стоимости является симметричной и содержит нулевые элементы на главной диагонали. Можно организовать хранение верхней диагонали в виде одномерного массива с пересчетом индексов, но в этом случае, что очевидно, экономия порядка $n^2/2$ ячеек памяти приведет к увеличению трудоемкости алгоритма. Выбор в пользу того или иного решения связан с конкретными ресурсными требованиями, и ресурсная эффективность каждого варианта может быть в данном случае легко оценена.

Рекурсивный алгоритм. Рассмотрим рекурсивный алгоритм Тарьяна, находящий оставное дерево графа в виде рекурсивной процедуры **DFS(V)** (справа указано количество базовых операций в строке). Вызов из основной программы имеет вид **DFS(1)**.

DFS(V)

($G[1..n,1..n]$ — матрица смежности)

($Visit[1..n]$ — массив пометок вершин, 1 — вершина помечена)

($T[1..n,1..2]$ — найденное оставное дерево, m — счетчик ребер)

$Visit[V] \leftarrow 1$ 2

For $W \leftarrow 1$ **to** N $1+3 * n$

If ($G[V,W] = 1$) 3

then

If ($Visit[W] = 0$) 2 * n

then

$M \leftarrow M+1$ 2

$T[M,1] \leftarrow V$ 3

$T[M,2] \leftarrow W$ 3

DFS(W)

end If

end If

end For

End.

Сложностной класс алгоритма. Если мы считаем, что длина входа алгоритма — это количество вершин n , то число задаваемых алгоритмом операций зависит не только от количества вершин, но и от количества ребер исходного графа, которое очевидно может варьироваться от 0 для графа без ребер до $n(n - 1)/2$ у полного графа. Поэтому в общем случае, т. е. без ограничений на матрицу стоимости, в лучшем случае алгоритм останавливается на первом вызове, выполняя $\Theta(n)$ операций. Заметим, что для этого достаточно, чтобы первая вершина графа была изолирована. Понятно также, что в случае несвязного графа алгоритм находит оставное дерево того его компонента, который включает первую вершину. В худшем случае — для полного графа мы имеем оценку $\Theta(n^2)$, равную как и для всех связных графов. Таким

образом, алгоритм относится к сложностному классу πP , а по характеристическим особенностям в общем случае — к подклассу $NPRH$, а в случае связных графов — к подклассу $NPRL$. Далее, анализируя алгоритм Тарьяна, мы будем предполагать, что матрица смежности на входе алгоритма описывает связный граф.

Анализ алгоритма методом подсчета вершин дерева рекурсии. При анализе дерева рекурсии, порождаемого данным алгоритмом, мы впервые сталкиваемся с ситуацией, когда это дерево сильно зависит от данных. Знание о том, что мы ищем оставное дерево для графа на n вершинах, оказывается недостаточным для того, чтобы построить дерево рекурсии. Мы надеемся, что внимательный читатель уже понял, что это дерево просто совпадает с оставным деревом графа, которое является результатом работы алгоритма. Выход из этой ситуации связан с использованием метода амортизационного анализа — мы рассматриваем поведение алгоритма на всех вызовах, а не в конкретной вершине дерева рекурсии. Для связного графа оставное дерево содержит $n - 1$ ребер, и алгоритм помечает все n вершин графа — отсюда следует, что

$$R(n) = n. \quad (7.8.2)$$

Количество листьев определяется оставным деревом, и, в смысле анализа трудоемкости, для нас важным является то, что в листьях дерева рекурсии все равно выполняется цикл просмотра строки матрицы смежности, но без добавления ребра в оставное дерево, и, следовательно, без рекурсивного вызова.

Временная эффективность — функция трудоемкости. Итак, наша цель — получить трудоемкость алгоритма Тарьяна, опираясь на знание общего количества вершин и идеи амортизационного анализа. Определим трудоемкость процедуры **DFS(V)** на один вызов/возврат $f_R(1)$. Мы передаем один параметр, не возвращаем значений, они хранятся в глобальных массивах, и процедура имеет одну локальную переменную, следовательно,

$$f_R(1) = 2(1 + 4 + 0 + 1 + 1) = 14, \quad f_R(n) = R(n)f_R(1) = 14n. \quad (7.8.3)$$

Опираясь на метод амортизационного анализа, найдем трудоемкость в вершинах дерева в предположении, что количество ребер графа заранее известно и равно m . Трудоемкость во всех вершинах дерева рекурсии представим как сумму трудоемкости цикла $f_{For}(n)$, проверки пометок $f_{if}(m)$ и добавления ребра в оставное дерево $f_T(n)$ по всем вызовам алгоритма

$$f_A(n, m) = f_R(n) + f_{For}(n) + f_{if}(m) + f_T(n). \quad (7.8.4)$$

На каждом вызове всегда выполняется цикл и проверка смежности, что позволяет получить функцию $f_{For}(n)$:

$$f_{For}(n) = R(n)(2 + 1 + 3n + 3n) = n(6n + 3) = 6n^2 + 3n.$$

Мы уже знаем, что алгоритм проверяет каждое ребро дважды и за-

трачивает две операции на проверку метки вершины, что позволяет определить функцию $f_{if}(m)$

$$f_{if}(m) = 2 \cdot 2m = 4m,$$

за все время работы для связного графа в оствное дерево будет добавлено $n - 1$ ребер, что позволяет определить $f_T(n)$:

$$f_T(n) = (n - 1)(2 + 3 + 3) = 8n - 8.$$

Объединяя все полученные компоненты, на основе (7.8.4) получаем

$$f_A(n, m) = 6n^2 + 25n + 4m - 8. \quad (7.8.5)$$

Рассмотрим лучший и худший случаи трудоемкости для связного графа. Минимальный связный граф на n вершинах содержит $m = n - 1$ ребер, что приводит к

$$f_A^V(n) = 6n^2 + 29n - 12, \quad (7.8.6)$$

в худшем случае для полного графа $m = n(n - 1)/2$, и

$$f_A^U(n) = 8n^2 + 23n - 8. \quad (7.8.7)$$

Отметим, что для этого алгоритма доля операций обслуживания дерева рекурсии падает с ростом числа вершин графа и составляет для худшего случая

$$F_R(n) = \frac{f_R(n)}{f_A(n)} = \frac{14n}{8n^2 + 23n - 8} \approx \frac{7}{4n}, \quad n \gg 1. \quad (7.8.8)$$

что позволяет говорить об эффективности его применения.

Емкостная эффективность — функция объема памяти. Для получения оценки емкостной эффективности нам нужна функция $H_R(n)$, определяющая наибольшую глубину дерева рекурсии. Для этого алгоритма мы впервые сталкиваемся с ситуацией, когда глубина дерева зависит от данных, и для связного графа может варьироваться от 2 до n , в зависимости от вида оствного дерева (см. рис. 7.11). Поэтому получим оценку памяти стека в худшем случае

$$V_{st}^{\wedge}(n) = H_R(n)(1 + 4 + 0 + 1 + 1) = 7n.$$

Объем памяти, занимаемой матрицей смежности, массивом пометок и структурой дерева может быть легко получен, и общие затраты с учетом стека составляют

$$V_A^{\wedge}(n) = n^2 + 10n - 2, \quad (7.8.9)$$

заметим, что оценка главного порядка функции емкостной эффективности справедлива и в лучшем случае, поскольку определяется объемом матрицы стоимости, и на основе (7.8.7) и (7.8.9) мы можем определить ресурсную сложность алгоритма

$$\mathfrak{R}_c(n) = \left\langle \Theta(n^2), \Theta(n^2) \right\rangle,$$

которая справедлива для любых связных графов на n вершинах.

§ 9. Алгоритм Беллмана оптимальной одномерной упаковки

Введение. Начиная с конца 50-х годов XX века, когда Р. Беллман предложил и обосновал основные идеи метода динамического программирования, задача оптимальной по стоимости одномерной упаковки привлекает внимание программистов и ученых, занимающихся разработкой и анализом алгоритмов. Повышенный интерес к этой задаче, равно как и к ее многочисленным модификациям, связан с разнообразными практическими областями применения. Заметим в этой связи, что метод динамического программирования хотя и позволил существенно сократить полный перебор вариантов, но реализующие этот метод алгоритмы не являются полиномиальными относительно длины входа. В настоящее время рост производительности и, прежде всего, доступной оперативной памяти современных компьютеров обеспечивает приемлемое время решения для ряда практических задач, сводящихся к одномерной оптимальной по стоимости упаковке, с применением точных классических алгоритмов, реализующих метод динамического программирования.

Изложение применения метода динамического программирования для решения задачи одномерной оптимальной по стоимости упаковки основано на классической книге Р. Беллмана и С. Дрейфуса [7.14] и статье [7.15], посвященной анализу трудоемкости классических алгоритмов, реализующих идеи Р. Беллмана. Мы хотим не только проиллюстрировать метод динамического программирования, но получить оценку вычислительной сложности и функцию трудоемкости рекурсивного алгоритма, используя методы дискретной математики и теории рекурсии, изложенные в главе 5.

Содержательная постановка задачи упаковки. Представим себе, что у нас есть рюкзак с прямоугольным дном и нерастяжимыми стенками определенной высоты. У нас есть также несколько групп коробок, с таким же дном, как у рюкзака. В любой группе количество коробок достаточно для упаковки всего рюкзака, каждая коробка в группе одинакова по высоте и имеет определенную стоимость. Наша задача состоит в том, чтобы упаковать рюкзак так, чтобы он закрывался, и сумма стоимостей упакованных коробок была бы наибольшей. Хотя содержательно мы имеем дело с объемом, но в реальности мы рассматриваем только высоту рюкзака и высоты коробок — в этом смысле задача является одномерной. Поскольку у нас нет ограничений на состав коробок в рюкзаке, то интуитивное решение состоит в том, чтобы выбрать коробки из группы, обладающей максимальной удельной (на единицу высоты) стоимостью. Но, к сожалению, мы не можем разрезать коробки по высоте — задача является целочисленной, и интуитивное решение не всегда оптимально. Например, из двух групп коробок с высотами 5 и 7 и стоимостями 10 и 18, в рюкзак высотой 10 лучше положить две коробки из первой группы, чем одну

из второй, хотя удельная стоимость коробок второй группы больше, чем в первой. Другая идея состоит в том, что можно рассмотреть все возможные варианты упаковки рюкзака и выбрать наилучший, но если рюкзак очень высокий, и у нас много разных групп коробок, то такой полный перебор может потребовать значительного времени. Ниже мы получим оценку сложности алгоритма, реализующего полный перебор вариантов.

Эта задача, известная, как задача об упаковке рюкзака, более корректно — задача оптимальной по стоимости одномерной упаковки, имеет разнообразные практические применения, для которых сегодня актуальным является получение именно точных решений. К такой постановке сводится задача одномерного раскroя материала, формирования оптимального пакета акций на фиксированную сумму. На основе полученных оптимальных решений при значительном количестве групп коробок можно даже построить достаточно надежную криптосистему. Эта задача относится к группе задач целочисленного программирования, которые формулируются как задачи поиска экстремума функций нескольких переменных, аргументами которой являются координаты точек ограниченного подмножества целочисленного пространства. В связи с этим мы позволим себе напомнить читателю некоторые сведения о целочисленных пространствах.

Понятие t -мерного евклидова целочисленного пространства. При изложении задач целочисленного программирования удобно использовать геометрическую терминологию, обобщающую наше представление о трехмерном пространстве. Поскольку в рассматриваемых задачах решение представляется в виде набора целых, как правило, неотрицательных чисел, то нас будет интересовать многомерное целочисленное пространство, которое является подмножеством классического t -мерного евклидова пространства [7.16]. Будем называть t -мерным координатным пространством множество упорядоченных совокупностей (x_1, x_2, \dots, x_m) , состоящих из t вещественных чисел x_1, x_2, \dots, x_m , и обозначать это пространство символом A^m . Каждую такую упорядоченную совокупность будем называть *точкой* t -мерного координатного пространства, обозначая ее через \mathbf{x} , а числа x_1, x_2, \dots, x_m будем называть *координатами точки* \mathbf{x} . Координатное пространство A^m называется t -мерным евклидовым пространством, если между любыми двумя точками \mathbf{x} и \mathbf{y} в t -мерного пространства A^m определено расстояние, обозначаемое символом $\rho(\mathbf{x}, \mathbf{y})$ и выражающееся соотношением

$$\rho(\mathbf{x}, \mathbf{y}) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \dots + (x_n - y_n)^2}. \quad (7.9.1)$$

Общеупотребительным для t -мерного евклидового пространства является обозначение E^m . Пространство, в котором указано правило, ставящее в соответствие любым двум элементам \mathbf{x} и \mathbf{y} вещественное число, называемое расстоянием между этими элементами — $\rho(\mathbf{x}, \mathbf{y})$, и удовлетворяющее следующим аксиомам:

- 1) $\rho(x, y) = \rho(y, x);$
- 2) $\rho(x, y) \leq \rho(x, z) + \rho(z, y);$
- 3) $\rho(x, y) \geq 0, \quad \rho(x, y) = 0 \Leftrightarrow x = y,$

называется *метрическим пространством*. Легко проверить, что расстояние, введенное формулой (7.9.1), удовлетворяет этим аксиомам, и, следовательно, пространство E^m является метрическим пространством. Подробные сведения по общей топологии и аксиоматике линейных, нормированных и метрических пространств содержатся, например, в [7.16, 7.17]. Пространство E^m содержит точки, координаты которых являются вещественными числами. В целях формализации задач целочисленного программирования нам необходимо такое подмножество пространства E^m , которое содержит только точки с целочисленными координатами. Будем называть это подмножество *m-мерным евклидовым целочисленным пространством* [7.18] и обозначать его через E_z^m :

$$E_z^m = \{x \mid x \in E^m, x = (x_1, x_2, \dots, x_m), \quad x_i \in Z \quad \forall i = \overline{1, m}\},$$

где Z — множество целых чисел.

Ограничения, возникающее в задачах целочисленного программирования, приводят к тому, что мы рассматриваем не всё множество E_z^m , а некоторую ограниченную совокупность его точек — в общем случае, некоторый многогранник с вершинами, имеющими целочисленные координаты. В связи с этим введем понятие *m-мерного координатного куба*. Множество Y всех точек y из E_z^m , координаты (y_1, y_2, \dots, y_m) которых удовлетворяют неравенствам

$$y_1 - x_1 \leq k, \quad y_2 - x_2 \leq k, \quad \dots, \quad y_m - x_m \leq k, \quad y_i \geq x_i \quad (7.9.2)$$

$$\forall i = \overline{1, m}, \quad k \in N,$$

будем называть *m-мерным координатным кубом* с ребром k с началом в точке $x = (x_1, x_2, \dots, x_m)$, $x \in E_z^m$, и обозначать его через $Cub_z^m(x, k)$ [7.18].

Поскольку многие задачи целочисленного программирования приводят к перебору точек *m-мерного координатного куба* с началом в точке $0 = (0, \dots, 0)$, т.е. куба $Cub_z^m(0, k)$, то представляется целесообразным ввести специальное обозначение: $k_z^m = Cub_z^m(0, k)$ [7.18]. Таким образом, например, 1_z^m — это *m-мерный координатный куб* с началом в нуле, все точки которого имеют целочисленные координаты — 0 или 1. Сколько целочисленных точек содержит *m-мерный координатный куб*? Ответ на этот вопрос является важным, т. к. позволяет указать верхнюю оценку границы размерности задачи целочисленного программирования, если мы пытаемся решить ее путем полного перебора точек. Поскольку в силу условия (7.9.2) каждая координата точки куба может принимать не более чем $k + 1$ целочисленных значений, и мы имеем m независимых координат для точки в пространстве E_z^m , то приходим к выводу, что $|Cub_z^m(x, k)| = (k + 1)^m$ вне зависимости от начальной точки куба. В частности, даже куб 1_z^m содержит 2^m точек.

Математическая постановка задачи. Рассмотрим общую постановку задачи одномерной оптимальной по стоимости упаковки. Пусть задано множество типов грузов

$$Y = \{y_i\}, \quad y_i = \{v_i, c_i\}, \quad i = \overline{1, n},$$

где каждый элемент y_i , соотнесенный с типом груза, обладает целочисленным линейным размером v_i , или «объемом» в общепринятых терминах задачи упаковки, и ценовой характеристикой c_i , которая содержательно отражает практически значимые предпочтения для загрузки объектов данного типа. Также целочисленным значением задан основной объем упаковки V . В классической постановке элементы y_i называются типами грузов. Для описания количества загружаемых в объем V элементов y_i введем в рассмотрение следующий характеристический вектор:

$$\mathbf{x} = \{x_i\}, \quad x = \overline{1, n},$$

где x_i — неотрицательное целое, т. е. $\mathbf{x} \in E_z^n$, $x_i \geq 0$. Значение компонента вектора $x_i = k$ соответствует загрузке k элементов типа y_i в объем V . Таким образом, описание некоторой упаковки грузов представляет собой целочисленную точку в n -мерном пространстве E_z^n . Среди всех возможных упаковок объема V грузами из Y должна существовать, по крайней мере, одна, максимизирующая суммарную стоимость, что приводит к следующей постановке задачи упаковки как задачи линейного целочисленного программирования.

Максимизировать линейный функционал:

$$P_n(\mathbf{x}) = \sum_{i=1}^n C_i(x_i) = \sum_{i=1}^n x_i c_i \rightarrow \max, \quad \sum_{i=1}^n x_i v_i \leq V. \quad (7.9.3)$$

Содержательно ограничение в (7.9.3) означает, что суммарный объем, занимаемый грузами всех типов в количествах, указанных характеристическим вектором \mathbf{x} , не должен превышать общего объема упаковки.

Вычислительная сложность полного перебора. Поскольку число возможных решений задачи упаковки, в силу (7.9.3), ограничено сверху, то мы можем гарантировать, что существует координатный куб, в который вписан многогранник, формируемый этими ограничениями. В этой связи очевидным методом решения является полный перебор всех возможных вариантов — точек целочисленного пространства. Заметим, что в англоязычной литературе этот метод называется British museum technique — техника Британского музея. При всей простоте метода нас останавливает проблема размерности — как только размерность целочисленного пространства становится большой — перебор требует хотя и конечного, но астрономического времени. Попробуем получить некоторые оценки для рассматриваемой задачи одномерной упаковки. Из всех n типов грузов по крайней мере один имеет минимальный объем. Обозначим через m максимальное количество грузов этого типа, размещающихся в объеме V , тогда весь объем перебора

ограничен кубом m_z^n , который содержит $(m+1)^n$ точек. Если задача состоит в упаковке 20 типов грузов ($n = 20$), и груз каждого типа размещается в объеме V не более 9 раз, то мы имеем задачу перебора 10^{20} точек в 20-мерном пространстве. Попробуйте оценить требуемое время в предположении, что анализ одной точки занимает на Вашем компьютере не более 1000 тактов.

Более точную оценку можно получить, если рассматривать прямоугольный параллелепипед в пространстве E_z^n , размеры сторон которого определяются типами грузов. Это приводит к верхней оценке количества точек перебора в виде

$$\prod_{i=1}^n (\lfloor V/v_i \rfloor + 1) \leq (m+1)^n, \quad m = \max_{i=1,n} \{V/v_i\}.$$

При значительном разбросе значений v_i эта оценка значительно лучше, чем $(m+1)^n$, но все равно неприемлема для практически значимых постановок задачи упаковки. Этот неутешительный результат и заставляет искать эффективные алгоритмы решения задачи одномерной оптимальной упаковки, существенно сокращающие перебор. Отметим в этой связи, что рассматриваемая задача упаковки является *NP*-трудной, и для нее в общей постановке отсутствуют сегодня полиномиальные по n точные алгоритмы решения.

Функциональное уравнение Беллмана для задачи упаковки. Опираясь на терминологию метода динамического программирования (см. главу 3), будем считать, что в рассматриваемой задаче распределяемым ресурсом является объем упаковки V , а функции дохода линейны: $g_i(x_i) = c_i x_i$. Наша задача — максимизировать доход (стоимость упаковки), заданный линейным функционалом $P_n(x)$ (см. 7.9.3), путем распределения ограниченного ресурса объема упаковки между грузами указанных типов. В этих условиях мы можем непосредственно использовать идею метода динамического программирования. С учетом обозначений, введенных в математической постановке задачи упаковки, основное функциональное уравнение Беллмана имеет вид [7.14]:

$$\left\{ \begin{array}{l} f_0(v) = 0; \\ f_m(v) = \max_{x_m} \{x_m c_m + f_{m-1}(v - x_m v_m)\}, \\ m = \overline{1, n}, \quad v = \overline{0, V}, \quad x_m = 0, 1, \dots, \left\lfloor \frac{v}{v_m} \right\rfloor. \end{array} \right. \quad (7.9.4)$$

Таким образом, метод предполагает последовательное решение одномерных задач целочисленной оптимизации с использованием информации об оптимальной упаковке объема v предыдущими типами грузов. Решением поставленной задачи является значение $f_n(V)$. Поскольку значения $f_1(v)$ могут быть элементарно вычислены на основе (7.9.4), то в дальнейшем мы будем рассматривать следующее основное функциональное уравнение для задачи одномерной оптимальной упаковки

ковки, записанное в виде рекуррентного соотношения, определяющего рекурсивно заданную функцию $f_m(v)$:

$$\begin{cases} f_1(v) = \left\lfloor \frac{v}{v_k} \right\rfloor \cdot c_1, & m = 1; \\ f_m(v) = \max_{x_m} \{x_m c_m + f_{m-1}(v - x_m v_m)\}, \\ m \geq 2, \quad x_m = 0, 1, \dots, \left\lfloor \frac{v}{v_m} \right\rfloor. \end{cases} \quad (7.9.5)$$

Схема алгоритма. Рекурсивная реализация основного функционального уравнения Беллмана (7.9.5) для задачи одномерной упаковки (7.9.3) достаточно проста. Рекурсивный алгоритм, напрямую реализующий рекуррентные соотношения (7.9.5), останавливается при значении $m = 1$, когда значение функции $f_1(v)$ может быть вычислено непосредственно. При $m \geq 2$ рекурсия выполняется для вычисления оптимальной стоимости упаковки текущего объема v , грузами типа t совместно с грузами предыдущих $m - 1$ типов в том же или меньшем объеме. Ниже мы приводим схему данного алгоритма в виде рекурсивной функции $F(V, x)$, не детализируя структуру данных для хранения вектора упаковки.

F(V, x)

(V — текущий объем упаковки)

(x — номер текущего типа груза)

($\text{boxV}[1..n], \text{boxC}[1..n]$ — массивы исходных данных)

If $x=1$

then (Останов рекурсии — прямое вычисление при $x=1$)

$k \leftarrow V \text{ div } \text{boxV}[1]$

$F \leftarrow k * \text{boxC}[1]$

 (оптимальная стоимость в V для 1-го типа)

else (Рекурсивные вызовы для определения $\max F$)

$\text{Max} \leftarrow 0;$

$k \leftarrow V \text{ div } \text{boxV}[x]$

 For $i \leftarrow 0$ to k

 (цикл поиска максимума)

$\text{Cost} \leftarrow i * \text{boxC}[x] + F((V - \text{boxV}[x] * i), x-1)$

 If $\text{Cost} > \text{Max}$

 then

$\text{Max} \leftarrow \text{Cost}$

$\text{Optx} \leftarrow i$

 end For

$F \leftarrow \text{Max}$ (функция возвращает оптимальную стоимость)

 (количество грузов типа x в объеме V равно Optx)

end If

Return (F)

End.

Пример дерева рекурсии. Рассмотрим пример решения задачи одномерной упаковки с использованием данного алгоритма — нас интересует порожденное дерево рекурсии. Мы решаем задачу с тремя типами грузов, при общем объеме упаковки $V = 10$. Информация об объемах v_i и стоимостях c_i для трех типов грузов приведена в табл. 7.3.

Таблица 7.3

Описание типов грузов

i	v_i	c_i
1	2	3
2	3	5
3	4	7

Решением поставленной задачи будет значение функции Беллмана

$f_3(10)$, при этом алгоритм порождает дерево рекурсии, показанное на рис. 7.12.

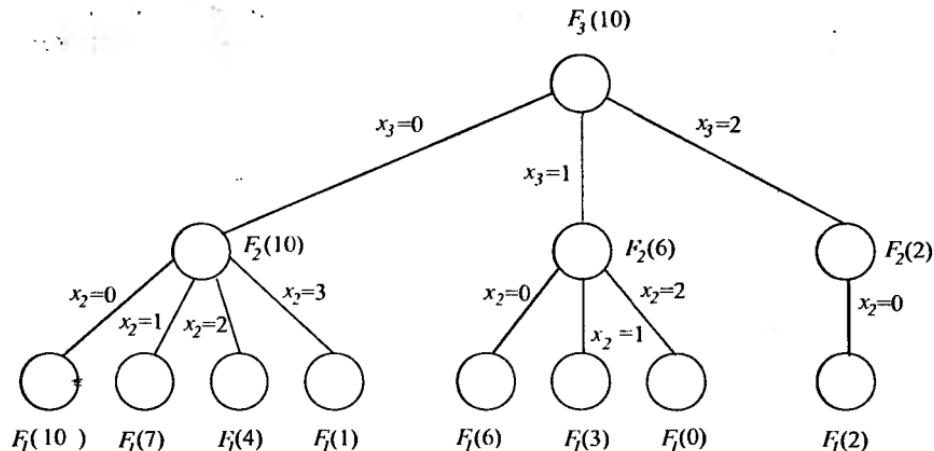


Рис. 7.12. Дерево рекурсии, порождаемое алгоритмом упаковки

Параметризация задачи. Исследуемый рекурсивный алгоритм упаковки является количественно параметрическим. Напомним, что в этом случае число элементарных операций, задаваемых алгоритмом, зависит не только от количества данных на входе, но и от их значений. Из (7.9.5) очевидно, что оценка вычислительной сложности будет зависеть как от значения n , так и от значений параметров V, v_1, \dots, v_n , учет которых существенно затрудняет анализ. С целью упрощения анализа рекурсивного алгоритма мы вводим параметр

$$k = \frac{V}{\bar{v}}, \quad \bar{v} = \frac{1}{n} \sum_{i=1}^n v_i,$$

показывающий, сколько грузов среднего объема размещается в объеме упаковки V . Очевидно, что в реальности количество любых грузов, размещенных в объеме V , является целым числом, но для оценки вычислительной сложности в среднем необходимо учитывать, что параметр k может быть и действительным (вещественным) числом.

Структура данных. Исследуемый рекурсивный алгоритм требует двух глобальных одномерных массивов, хранящих описание стоимости и объема типов грузов — **boxC[1..n]** и **boxV[1..n]**. Для хранения конечного и промежуточных результатов мы будем использовать двумерный массив **XArr[1..n,0..n]**, первый индекс которого соответствует номеру типа груза, по второму индексу в **XArr[x,1..n]** хранится оптимальный вектор, а в **XArr[x,0]** — значение оптимальной упаковки. В этой же ячейке — **XArr[x,0]** мы будем хранить текущий максимум. Таким образом, при вызове алгоритма для типа груза с номером **x** результаты непосредственных рекурсивных вызовов располагаются в строке **XArr[x-1,0..n]**, а результат формируется в **XArr[x,0..n]**.

Рекурсивный алгоритм. Рассмотрим рекурсивный алгоритм, находящий оптимальную одномерную упаковку, в виде рекурсивной процедуры **F(V,x)** (справа указано количество базовых операций в строке).

F(V, x)

(**V** — текущий объем упаковки)
 (**x** — номер текущего типа груза)
 (**boxV[1..n]**, **boxC[1..n]** — массивы исходных данных)
 (**XArr[1..n,0..n]** — массив результатов)

```

If x=1
  then          (Останов рекурсии)           1
    k ← V div boxV[1]                      3
    XArr[1,0] ← k * boxC[1]                 5
    (формирование вектора при x=1)
  For j ← 2 to n                           1+(n-1) * 3
    XArr[1,j] ← 0                          3 * (n-1)
    XArr[1,1] ← k                         3
  else          (Рекурсия для определения max F)   1+(n+1) * 3
    (Обнуление оптимального вектора)
    For j ← 0 to n                         3 * (n+1)
      XArr[x,j] ← 0                       3
      k ← V div boxV[x]
      (цикл поиска максимума)
    For i ← 0 to k                         1+(k+1) * 3
      F((V-boxV[x] * i),x-1)               4 * k
      Cost ← i * boxC[x]+XArr[x-1,0]        7 * k
      If Cost >= XArr[x,0]                  3 * k
        then
          (копирование вектора от x-1)
          For j ← 1 to n                   1+n * 3
            XArr[x,j] ← XArr[x-1,j]         6 * n
            XArr[x,x] ← i                  3
            XArr[x,0] ← Cost                3
        end For
      end If
    End.
  
```

Анализ алгоритма методом подсчета вершин дерева рекурсии.

Дерево рекурсии, порождаемое данным алгоритмом, является параметрически зависимым — количество порожденных вершин определяется переменной цикла, и, следовательно, параметром k , при этом на каждом рекурсивном вызове меняется еще и текущий объем упаковки. Наша задача — попытаться получить явную формулу для количества вершин дерева рекурсий в условиях введенной параметризации. Обозначим через $R(n, k)$ функцию, задающую общее количество вершин дерева в зависимости от значений n и k . Рассмотрим структуру дерева рекурсии при некоторых значениях n и k . Фрагменты дерева для значений $k = 1$ и $k = 2$ и соответствующие значения функции $R(n, k)$ показаны на рис. 7.13 и 7.14.

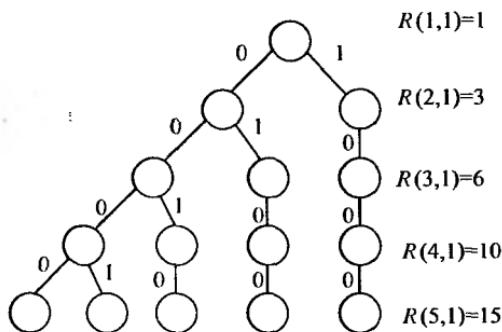


Рис. 7.13. Фрагмент дерева рекурсий при $k = 1$

Для того чтобы найти закономерность для значений $R(n, k)$, построим треугольник Паскаля, первые восемь строк которого показаны на рис. 7.15.

Теперь видно, что значения $R(n, k)$ — это элементы треугольника Паскаля — биномиальные коэффициенты. Напомним читателям, что в главе 5 описаны методы решения рекуррентных соотношений, приводящих к биномиальным коэффициентам, в частности, обращаем

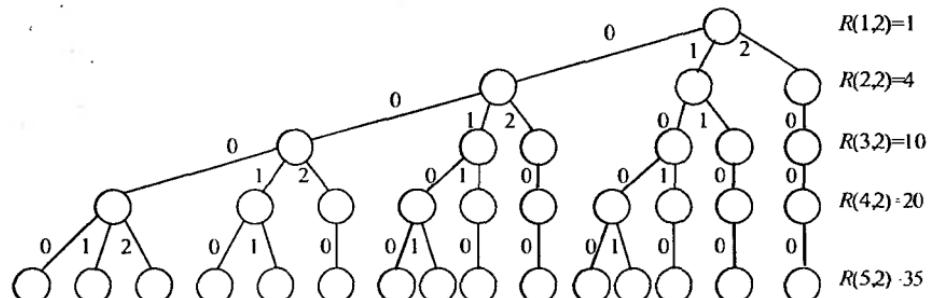


Рис. 7.14. Фрагмент дерева рекурсий при $k = 2$

$m = 0$					1			
$m = 1$					1	1		
$m = 2$				1	2	1		
$m = 3$			1	3	3	1		
$m = 4$		1	4	6	4	1		
$m = 5$	1	5	10	10	5	1		
$m = 6$	1	6	15	20	15	6	1	
$m = 7$	1	7	21	35	35	21	7	1

Рис. 7.15. Треугольник Паскаля — биномиальные коэффициенты

внимание на результаты исследования «биномиального» рекурсивного дерева, на основе которых можно теоретически доказать формулу для общего числа вершин данного дерева. Устанавливая соответствие между номерами строк треугольника и значениями n и k , окончательно получаем

$$R(n, k) = C_{n+k}^{n-1}. \quad (7.9.6)$$

Проводя аналогичные рассуждения для количества внутренних вершин и листьев порожденного дерева (заметьте, например, что $R_V(5, 2) = R(4, 2)$), получаем, с учетом введенной параметризации, аналогичные формулы для $R_V(n, k)$ и $R_L(n, k)$. Поскольку все эти значения являются биномиальными коэффициентами, мы, в целях дальнейшего удобства анализа, приведем также коэффициенты для их сведения к $R(n, k)$

$$\begin{aligned} R_L(n, k) &= C_{n+k-1}^{n-1} = R(n, k) \frac{k+1}{n+k}, \\ R_V(n, k) &= C_{n+k}^{n-2} = R(n, k) \frac{n-1}{n+k}. \end{aligned} \quad (7.9.7)$$

Однако, в отличие от алгоритма Тарьяна, любая ветвь рекурсии требует рассмотрения всех типов грузов, вне зависимости от параметра k , что позволяет легко определить глубину дерева, которая равна n . Таким образом, на основе (7.9.6) и (7.9.7) мы можем получить все формулы для дерева рекурсии, порождаемого алгоритмом Беллмана в рамках принятой параметризации:

$$\begin{aligned} R(n, k) &= C_{n+k}^{n-1}, \quad R_L(n, k) = C_{n+k-1}^{n-1}, \quad R_V(n, k) = C_{n+k}^{n-2}, \\ H_R(n) &= n, \quad B_L(n, k) = \frac{k+1}{n+k}. \end{aligned} \quad (7.9.8)$$

Временная эффективность — функция трудоемкости. Трудоемкость данного алгоритма мы найдем, учитывая предложенную параметризацию, т. е. $f_A = f_A(n, k)$. В соответствии с методом подсчета

вершин дерева рекурсии определим вначале трудоемкость процедуры на один вызов/возврат $f_R(1)$. Поскольку процедура $\mathbf{F}(\mathbf{V}, \mathbf{x})$ передает два параметра ($p = 2$), в стеке сохраняются значения четырех регистров ($r = 4$) и процедура имеет четыре локальных переменных ($l = 4$) (напомним, что результаты возвращаются через глобальный массив), то

$$f_R(1) = 2(2 + 4 + 0 + 4 + 1) = 22,$$

и, следовательно,

$$f_R(n, k) = 22R(n, k) = 22C_{n+k}^{n-1}. \quad (7.9.9)$$

Трудоемкость останова рекурсии включает в себя одно сравнение для входа в блок останова и $6n + 6$ операций формирования вектора, таким образом, $f_{CL}(1) = 6n + 7$, следовательно,

$$f_{CL}(n, k) = R_L(n, k) f_{CL}(1) = (6n + 7) C_{n+k-1}^{n-1}.$$

В целях удобства получения общей формулы введем обозначения для двух коэффициентов, показывающих долю внутренних вершин и листьев в общем количестве вершин дерева рекурсии:

$$\begin{aligned} R_L(n, k) &= \alpha_L R(n, k), \quad \alpha_L = \frac{k+1}{n+k}, \\ R_V(n, k) &= \alpha_V R(n, k), \quad \alpha_V = \frac{n-1}{n+k}, \end{aligned} \quad (7.9.10)$$

заметим, что $\alpha_L + \alpha_V = 1$, таким образом,

$$f_{CL}(n, k) = (6n + 7) \alpha_L R(n, k). \quad (7.9.11)$$

Трудоемкость во внутренних вершинах дерева представим в виде суммы трудоемкости обнуления вектора и трудоемкости цикла

$$f_{CV}(n, k) = f_{CVvec}(n, k) + f_{CVfor}(n, k),$$

при этом в трудоемкость обнуления вектора включим операцию инициализации цикла поиска максимума, и, суммируя операции, получаем

$$f_{CVvec}(n, k) = R_V(n, k) f_{CVvec}(v) = (6n + 12) \alpha_V R(n, k). \quad (7.9.12)$$

Наибольший интерес представляет трудоемкость цикла поиска максимума. Заметим, что 17 операций, выполняемых в первых трех строках цикла, включая три операции на его обслуживание, выполняются для каждого рекурсивного обращения к процедуре, включая вызовы листьев, за исключением одного основного вызова, что определяет одно слагаемое трудоемкости цикла — $17(R(n, k) - 1)$.

Трудоемкость копирования оптимального вектора внутри цикла поиска максимума составляет $9n + 7$ операций. Однако количество переприсваиваний в блоке **then** определяется данными, даже при фиксированном параметре k . Мы снова обращаемся к методу амортизационного анализа, т. к. можем указать общее количество выполнений этого блока по всем рекурсивным вызовам, в то время как анализ в отдельно взятой вершине затруднен. В лучшем случае, а именно когда для любого

объема оптимальной является упаковка первого типа груза, этот блок будет выполнен однократно в каждой внутренней вершине дерева. Худшим случаем, когда блок **then** будет выполняться каждый раз на проходе цикла, является ситуация, при которой упаковка каждого следующего по номеру типа груза является предпочтительной. Это означает, что возврат из каждого рекурсивного вызова, включая листья, будет приводить к выполнению этого блока, что совокупно равно общему числу вершин. Анализ в среднем связан с суммированием гармонических чисел, умноженных на биномиальные коэффициенты, и выходит за рамки данной книги, однако обычное усреднение дает также вполне приемлемые результаты. Для учета этих случаев мы вводим специальный коэффициент относительно общего количества вершин дерева, значения которого определяются приведенными выше рассуждениями:

$$\alpha_f^{\wedge} = 1, \quad \alpha_f^{\vee} = \alpha_V = \frac{n-1}{n+k}, \quad \bar{\alpha}_f = \frac{1+\alpha_V}{2} = 1 - \frac{k+1}{2(n+k)}. \quad (7.9.13)$$

Теперь мы можем записать общую формулу для лучшего, худшего и среднего случаев $f_{CVfor}(n, k)$, используя введенный коэффициент:

$$f_{CVfor}(n, k) = 17(R(n, k) - 1) + (9n + 7)\alpha_f R(n, k). \quad (7.9.14)$$

Объединяя все компоненты в соответствии с методом подсчета вершин дерева рекурсии (7.9.9)–(7.9.14), и учитывая, что $\alpha_L + \alpha_V = 1$, окончательно получаем формулу трудоемкости алгоритма Беллмана в предположении, что все типы грузов обладают одинаковым значением параметра k , а различные случаи трудоемкости различаются выбором коэффициента α_f в соответствии с (7.9.13):

$$f_A(n, k) = (6n + 9n\alpha_f)R(n, k) + \\ + (46 + 5\alpha_V + 7\alpha_f)R(n, k) - 17. \quad (7.9.15)$$

Доля операций обслуживания дерева рекурсии составляет

$$F_R(n, k) = \frac{f_R(n, k)}{f_A(n, k)} \approx \frac{22}{6n + 9n\alpha_f + 46 + 5\alpha_V + 7\alpha_f}, \\ F_R(n, k) \rightarrow 0, \quad n \rightarrow \infty.$$

Для реальных входов значение параметра k не обязательно одинаково для всех типов грузов и является действительным числом. В этом случае, как показывают проведенные исследования [7.15], мы можем перейти от классической формулы для количества сочетаний к формуле, использующей гамма-функцию Эйлера $\Gamma(s)$, как вещественному аналогу факториала

$$R(n, k) = \frac{\Gamma(n+k+1)}{\Gamma(n)\Gamma(k+2)}.$$

Емкостная эффективность — функция объема памяти. Для получения оценки емкостной эффективности в области памяти стека нам нужна функция $H_R(n)$, поскольку

$$V_{st}(n) = H_R(n)(p + r + f + l + 1),$$

а алгоритм Беллмана требует четыре локальные ячейки, то

$$V_{st}(n) = H_R(n)(2 + 4 + 0 + 4 + 1) = 11H_R(n) = 11n,$$

глобальная память представлена массивами стоимостей и объемов типов грузов и двумерным массивом **XArr[1..n,0..n]** и не зависит от параметра k . Объем глобальной памяти вычисляется элементарно, и

$$V(n) = V_{ram}(n) + V_{st}(n) = n^2 + 3n + 11n = n^2 + 14n.$$

На основе полученных результатов мы можем определить ресурсную сложность алгоритма при введенной параметризации задачи

$$\mathfrak{R}_c(n, k) = \left\langle \Theta(nC_{n+k}^{n-1}), \Theta(n^2) \right\rangle,$$

при фиксированном значении n ресурсная сложность алгоритма является наихудшей при $k = n - 2$, в силу свойств биномиальных коэффициентов.

Сложностной класс алгоритма. Полученные результаты говорят о том, что трудоемкость алгоритма Беллмана очень сильно зависит от данных — параметр k определяет главный порядок функции трудоемкости, и алгоритм принадлежит подклассу $NPRH$. При фиксированном количестве типов грузов дерево рекурсии может быть как унарным (при $k = 0$), так и достаточно широким. Наихудшая ситуация возникает если $k = n - 2$ — мы имеем максимально возможный при данном значении n биномиальный коэффициент, имеющий экспоненциальную оценку. В связи с этим, в теории алгоритм принадлежит (по оценке худшего случая при разных значениях параметра k) к сложностному классу πE .

Некоторые рекомендации по его рациональному применению можно дать на основе следующих рассуждений. В силу свойств биномиальных коэффициентов

$$C_{n+k}^{n-1} = C_{n+k}^{k+1}, \quad C_{n+k}^{k+1} = \Theta((n+k)^{k+1}), \quad k \ll n,$$

поэтому при небольших и фиксированных значениях k мы будем иметь полиномиальную по n трудоемкость. Малые k содержательно означают, что размеры типов грузов всего в несколько раз меньше объема упаковки, и алгоритм порождает не очень широкое дерево рекурсии с приемлемой трудоемкостью. Более интересные и содержательные результаты можно получить на основе сравнительного анализа рекурсивного и табличного алгоритмов решения задачи одномерной упаковки. Такой детальный сравнительный анализ, особенно при получении временных характеристик программных реализаций алгоритмов, дает

возможность обоснования их рационального выбора. Предварительные результаты такого анализа опубликованы в [7.15].

Задачи и упражнения к главе 7

7.1. Напишите итерационный алгоритм вычисления факториала и проведите анализ его трудоемкости. Совпадает ли полученный Вами результат с функцией трудоемкости, приведенной в параграфе 7.1 — $f_A(m) = 5m + 5$?

7.2. Разработайте итерационный алгоритм вычисления чисел Фибоначчи со сложностью $\Theta(m)$ и получите его теоретическую функцию трудоемкости. Если не хранить все предыдущие числа в массиве, — каково минимальное количество необходимых дополнительных ячеек памяти?

7.3. Модифицируйте алгоритм вычисления квадратного корня так, что бы он работал при всех значениях $a \geq 0$.

7.4. Получите функцию трудоемкости двухфазного алгоритма вычисления квадратного корня методом рекуррентных соотношений, считая, что трудоемкость второй фазы является трудоемкостью останова первой.

7.5. Напишите программу, которая непосредственно вычисляет значение квадратного корня, используя рекуррентное соотношение (7.3.1). Расставьте счетчик базовых операций и проведите вычислительные эксперименты при разных значениях a с целью определения трудоемкости при фиксированной точности. Начиная с каких значений a предложенный в параграфе 7.3 двухфазный алгоритм будет обладать лучшей трудоемкостью?

7.6. Трудоемкость алгоритма быстрого возведения в степень не зависит от значения x — умножение есть базовая операция, но время выполнения машинного алгоритма умножения зависит от значений бит числа. Определите, насколько различаются времена выполнения программной реализации для различных x ? Для ответа на этот вопрос Вам будет необходим доступ к тактовому счетчику Вашего компьютера.

7.7. Является ли алгоритм, описанный в 7.4, асимптотически оптимальным, и в каких случаях? На сколько мы можем уменьшить количество умножений, необходимых для возведения числа в данную степень? Это достаточно серьезный вопрос, и мы советуем, если Вы не обосновали ответ самостоятельно, обратиться к специальной литературе.

7.8. Разработайте алгоритм умножения «в столбик» двух длинных двоичных чисел, заданных массивом бит. Вы можете использовать дополнительный алгоритм сложения из параграфа 7.5. Получите функцию трудоемкости этого алгоритма, и, пользуясь методикой сравнительного анализа, определите рациональный диапазон его применения относительно алгоритма Карацубы.

7.9. Если Вы внимательно посмотрите на трудоемкости строк алгоритма Карацубы, то наверняка заметите, что «программирование в про-

цедурах» приводят к лишним вычислительным затратам. Попробуйте заменить вызовы процедур сложения, вычитания и пересылки на их непосредственные тексты. Будьте внимательны с индексами массивов. Если у Вас хватит терпения проанализировать полученный алгоритм, то Вы ответите на вопрос, — на сколько Вам удалось улучшить коэффициенты функции трудоемкости, и чем Вам пришлось за это заплатить?

7.10. Предложенная в этой главе структура данных для алгоритма Карацубы, очевидно, не является оптимальной. Попробуйте предложить структуру, обладающую хорошей емкостной эффективностью. Приведет ли такая структура к росту трудоемкости, или Вам удастся сохранить коэффициенты $f_A(n)$, а может быть, даже и улучшить их?

7.11. Разработайте алгоритм слияния двух отсортированных массивов, работающий «по месту», и требующий не более чем фиксированного числа дополнительных ячеек памяти. Это хорошая алгоритмическая задача — известный авторам алгоритм достаточно «хитрый», и выполняет ряд неочевидных пересылок, чтобы освободить ячейку и не потерять сортированный порядок. Как Вы считаете, оправдано ли увеличение трудоемкости алгоритма полученной экономией двух дополнительных массивов? Какой из алгоритмов слияния Вы будете использовать на практике?

7.12. Проанализируйте любой известный Вам «квадратичный» алгоритм сортировки, например, алгоритм сортировки вставками. На основе полученной функции трудоемкости и трудоемкости алгоритма сортировки слиянием определите рациональный диапазон их применения. Определите, насколько различаются времена выполнения обобщенных базовых операций у этих алгоритмов? Для ответа на этот вопрос необходим вычислительный эксперимент.

7.13. Разработайте более эффективную, чем это предложено в параграфе 7.7, структуру данных для рекурсивного генетического алгоритма поиска экстремума. На сколько Вам удалось уменьшить коэффициент у главного порядка функции объема памяти?

7.14. Попробуйте получить функцию трудоемкости рекурсивного генетического алгоритма методом подсчета вершин дерева рекурсии. Сравните полученный Вами результат с формулой (7.7.9).

7.15. Определите, какие ситуации с данными приводят к лучшему и худшему случаям трудоемкости рекурсивного генетического алгоритма. На основании этих рассуждений Вы можете получить формулы трудоемкости для этих случаев и тем самым явно установить характеристики временной устойчивости данного алгоритма при фиксированном значении n . Естественно, в этой задаче мы предполагаем, что трудоемкость вычисления целевой функции не зависит от значений аргумента.

7.16. Предложите модифицированный алгоритм Тарьяна, который находит все связные компоненты данного графа, при очевидном условии, что исходный граф не является связным, и состоит, по крайней мере, из двух компонентов.

7.17. Предложите другую структуру хранения информации о графе, которая позволила бы улучшить временную эффективность алгоритма Тарьяна. Какими особенностями должен обладать граф, чтобы Ваша структура давала ощущимый эффект? Какую структуру Вы предложили бы для сильно связных графов, количество ребер которых близко к полному?

7.18. Мы можем строить оптимальную упаковку «снизу вверх», начиная с оптимальной упаковки грузами первого типа (табличный алгоритм Беллмана). Предложите комбинированный алгоритм, использующий табличный и рекурсивный метод, позволяющий сократить дерево рекурсии и обладающий оптимальной трудоемкостью — это, на наш взгляд, серьезная исследовательская задача.

Список литературы к главе 7

- 7.1. Ноден П., Кимме К. Алгебраическая алгоритмика: Пер. с франц. — М.: Мир, 1999. — 720 с.
- 7.2. <http://www.ccas.ru>
- 7.3. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ, 2-е изд.: Пер. с англ. — М.: Изд. дом «Вильямс», 2005. — 1296 с.
- 7.4. Ахо А., Хопкрофт Дж., Ульман Дж. Структуры данных и алгоритмы: Пер. с англ.: — М.: Изд. дом «Вильямс», 2001. — 384 с.
- 7.5. Кнут Д.Э. Искусство программирования, том 3. Сортировка и поиск, 2-е изд.: Пер. с англ.: Уч. пос. — М.: Изд. дом «Вильямс», 2000. — 832 с.
- 7.6. Рутковский Л., Пилинский М., Рутковская Д. Нейронные сети, генетические алгоритмы и нечеткие системы. — М.: Горячая линия-Телеком, 2004 г. — 452 с.
- 7.7. Штоловба С.Д. Муравьиные алгоритмы // Exponenta Pro Математика в приложениях. 2003. №4. С.70–75.
- 7.8. Гладков Л.А., Курейчик В.В., Курейчик В.М. Генетические алгоритмы / Под ред. В.М. Курейчика. — 2-е изд., испр. и доп. — М.: ФИЗМАТЛИТ, 2006. — 320 с.
- 7.9. Макконелл Дж. Основы современных алгоритмов. 2-е доп. изд. — М.: Техносфера, 2004. — 368 с.
- 7.10. Гасфилд Д. Строки, деревья и последовательности в алгоритмах: Информатика и вычислительная биология / Пер. с англ. И.В. Романовского. — СПб.: Невский диалект; БХВ-Петербург, 2003 г. — 654 с.
- 7.11. Tarjan R.E. Efficiency of a good but not linear graph algorithms // Siam journal on Computing, 1972, 1(2), pp. 215–225.
- 7.12. Гудман С., Хидетниеми С. Введение в разработку и анализ алгоритмов. — М.: Мир, 1981. — 368 с.
- 7.13. Евстигнеев В.А., Касьянов В.Н. Теория графов: алгоритмы обработки деревьев. — Новосибирск: Наука, 1994.
- 7.14. Беллман Р., Дрейфус Р. Прикладные задачи динамического программирования: Пер. с англ. — М.: Наука, 1965, — 457 с.
- 7.15. Ульянов М.В., Гурин Ф.Е., Исаков А.С., Бударгин В.Е. Сравнительный анализ табличного и рекурсивного алгоритмов точного решения задачи

- одномерной упаковки // Exponenta Pro Математика в приложениях. 2004. №2(6). С. 64–70.
- 7.16. Ильин В.А., Садовничий В.А., Сенцов Бл.Х. Математический анализ. — М.: Наука. Главная редакция физико-математической литературы, 1979. — 720 с.
- 7.17. Юдин Д.Б., Горяшко А.П., Немировский А.С. Математические методы оптимизации устройств и алгоритмов АСУ / Под ред. Ю.В. Асафьева, В.А. Шабалина. — М.: Радио и связь. 1982. — 288 с.
- 7.18. Хаггарти Р. Дискретная математика для программистов. — М.: Техносфера, 2005. — 400 с.

Приложение

ПРОГРАММНЫЕ РЕАЛИЗАЦИИ РЕКУРСИВНЫХ АЛГОРИТМОВ И ИХ ЭКСПЕРИМЕНТАЛЬНОЕ ИССЛЕДОВАНИЕ

(*А.Д. Брейман, Г.П. Рябов*)

1. Цели и задачи экспериментального исследования рекурсивных алгоритмов

В основе сравнительного анализа алгоритмов лежат теоретические и экспериментальные оценки требуемой памяти и количества базовых операций, задаваемых алгоритмом в выбранной модели вычислений — функции трудоемкости для лучшего, среднего и худшего случаев при фиксированной размерности. Для рекурсивных алгоритмов дополнительными характеристиками являются функции оценки числа вершин в дереве рекурсии. Теоретические функции обычно базируются на определенных предположениях о входах алгоритма и, вообще говоря, требуют экспериментальных подтверждений. В связи с этим экспериментальное исследование программных реализаций рекурсивных алгоритмов, как составная часть анализа их ресурсной эффективности, связано с необходимостью решения следующих задач:

- экспериментальное подтверждение теоретически полученной функции трудоемкости, как правило, для трудоемкости в среднем, с учетом принадлежности алгоритма к одному из классов по влиянию характеристических особенностей исходных данных на функцию трудоемкости алгоритма;
- экспериментальное подтверждение теоретически полученных функций для характеристик дерева рекурсии на исследуемом диапазоне размерности входа.

2. Методика проведения экспериментов и обработка результатов

Получение достоверных результатов для этих задач сопряжено с организацией следующих этапов экспериментального исследования:

- модификация исходного текста программной реализации алгоритма для определения характеристик дерева рекурсии и получения количества выполненных базовых операций при данном входе, связанная с расстановкой счетчиков базовых операций и счетчиков порожденных вершин дерева рекурсии;

— организация генерации входов алгоритма, обеспечивающей презентативность выборки, т. е. входов, соответствующих особенностям применения исследуемого алгоритма в данной программной системе, значения функции трудоемкости алгоритма для этого множества входов и составляют в данном случае генеральную совокупность;

— планирование эксперимента, состоящее в определении необходимого (минимального) объема выборки для фиксированной размерности задачи, определение исследуемого сегмента и шага изменения размерности;

— собственно проведение вычислительного эксперимента, в ходе которого могут быть получены как экспериментальные значения функции трудоемкости, так и экспериментальные времена выполнения;

— обработка результатов эксперимента и проверка гипотезы о соответствии теоретической функции трудоемкости в среднем и экспериментально полученных выборочных средних значений.

Для всех экспериментов с программными реализациями рекурсивных алгоритмов переменные счетчиков были назначены следующим образом: для подсчета $R_V(D)$ используется переменная **rv**, для $R_L(D)$ — **rl**, для $f_R(D)$ — **fr**, для $f_{CL}(D)$ — **fcl**, для $f_{CV}(D)$ — **fcv**, для $f_A(D)$ — **fa**. Перед обращением к функции Power все счетчики обнуляются. Продемонстрируем расстановку счетчиков вершин дерева рекурсии и счетчиков базовых операций на примере программы возведения числа в целую степень (см. П.3.3.3).

```

var rv, rl, fr, fcl, fcv, fa: Integer;
function Power( x: Extended; m: Integer ):Extended;
begin
    fa:=fa+18; // количество операций
    fr:=fr+18; // трудоемкость организации рекурсии
    if m=0 then begin
        rl:=rl+1; // количество листьев дерева
        Power := 1;
        fa:=fa+2; // количество операций
        fcl:=fcl+2; // трудоемкость останова рекурсии
    end
    else begin
        if m mod 2=0 then begin
            rv:=rv+1; // количество внутр. вершин дерева
            z := Power(x,m Div 2);
            f := z*z;
            fa:=fa+7; // количество операций
            fcv:=fcv+7; // трудоемкость порождения рекурсии
        end
        else begin
            rv:=rv+1; // количество внутр. вершин дерева
            z := Power(x, m Div 2);
            Power := z*z*x;
        end
    end

```

```

    fa := fa + 8; // количество операций
    fcv := fc v + 8; // трудоемкость порождения рекурсии
end;
end;
end;

```

3. Программные реализации рекурсивных алгоритмов

Программные реализации алгоритмов были выполнены на языке Delphi, компилировались и запускались в среде Borland Delphi 7.

3.1. Классический пример — задача вычисления факториала

Для решения многих комбинаторных задач нужно уметь вычислять количество всех возможных различных перестановок n элементов — факториал $n!$ целого числа n . Один элемент можно разместить единственным способом, $1! = 1$. Если элементов больше одного, любую их перестановку можно получить, последовательно выбирая элементы для заполнения n мест. В качестве первого элемента можно выбрать любой элемент из n , в качестве второго — любой элемент из $n - 1$ оставшихся, и т. д. Тогда число перестановок n элементов равно произведению n на число перестановок $n - 1$ оставшихся элементов: $n! = n(n - 1)!$.

3.1.1. Программная реализация рекурсивного алгоритма

```

// функция Factorial вычисляет n!
function Factorial( n: Extended ): Extended;
begin
  if n = 1 then
    Factorial := 1
  else
    Factorial := n * Factorial(n - 1);
end;

```

3.1.2. Результаты экспериментальных исследований

На основании экспериментов, проведенных с программной реализацией рекурсивного алгоритма вычисления факториала, были получены следующие результаты по анализу дерева рекурсии и трудоемкости программной реализации. Мы приводим их в сравнении с теоретическими значениями, рассчитанными по формулам (7.1.2)–(7.1.4), в таблицах П.1 и П.2 соответственно.

Мы не приводим значения абсолютных и относительных ошибок, поскольку в данном случае экспериментальные результаты полностью совпадают с теоретическими значениями.

3.2. Задача о кроликах — вычисление чисел Фибоначчи

Знаменитый итальянский математик Леонардо из Пизы, известный больше по своему прозвищу Фибоначчи, в сочинении «Книга об абаке»

Таблица П.1
Результаты анализа дерева рекурсии

<i>m</i>	<i>R(m)</i>		<i>R_V(m)</i>		<i>R_L(m)</i>		<i>H_R(m)</i>	
	эксп.	теор.	эксп.	теор.	эксп.	теор.	эксп.	теор.
1	1	1	0	0	1	1	1	1
5	5	5	4	4	1	1	5	5
10	10	10	9	9	1	1	10	10
15	15	15	14	14	1	1	15	15

Таблица П.2
Результаты анализа трудоемкости алгоритма

<i>m</i>	<i>f_R(m)</i>		<i>f_{CL}(m)</i>		<i>f_{CV}(m)</i>		<i>f_A(m)</i>	
	эксп.	теор.	эксп.	теор.	эксп.	теор.	эксп.	теор.
1	14	14	2	2	0	0	16	16
5	70	70	2	2	16	16	88	88
10	140	140	2	2	36	36	178	178
15	210	210	2	2	56	56	268	268

(дошедший до наших времен вариант датируется 1228 годом) приводит следующую задачу: «Некто поместил пару кроликов в некоем месте, огороженном со всех сторон стеной, чтобы узнать, сколько пар кроликов родится при этом в течение года, если природа кроликов такова, что через месяц пара кроликов производит на свет другую пару, а рожают крольчики со второго месяца после своего рождения». Далее Фибоначчи приводит помесячный расчет числа пар кроликов: на первом месяце имеется одна пара, на втором месяце они производят еще одну (всего $1 + 1 = 2$), на третьем месяце первая пара производят еще одну (всего $2 + 1 = 3$), на четвертом месяце уже две пары производят потомство (всего $3 + 2 = 5$) и т.д. Поскольку кролики начинают размножаться со второго месяца, на каждом шаге прирост составляет ровно столько пар кроликов, сколько пар кроликов было два месяца назад. Прирост же этот добавляется к тому количеству кроликов, которое было в предыдущем месяце. Если выписать получающиеся числа подряд, то окажется, что каждое следующее число, начиная с третьего, равно сумме двух предыдущих: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377 и т. д. Эта последовательность называется числами Фибоначчи.

3.2.1. Программная реализация рекурсивного алгоритма

```
// функция Fibonacci вычисляет n-е число Фибоначчи
function Fibonacci( n: Extended ): Extended;
begin
  if n < 3 then
    Fibonacci := 1
  else
    Fibonacci := Fibonacci(n-1) + Fibonacci(n-2);
end;
```

```

else
    Fibonacci := Fibonacci(n-2) + Fibonacci(n-1);
end;

```

3.2.2. Результаты экспериментальных исследований

Эксперименты, проведенные с программной реализацией рекурсивного алгоритма вычисления чисел Фибоначчи, позволили получить следующие результаты по анализу дерева рекурсии и трудоемкости программной реализации. Эти результаты приводятся, в сравнении с теоретическими значениями, рассчитанными по формулам (7.2.2)–(7.2.3), в таблицах П.3 и П.4 соответственно.

Таблица П.3
Результаты анализа дерева рекурсии

m	$R(m)$		$R_V(m)$		$R_L(m)$		$H_R(m)$	
	эксп.	теор.	эксп.	теор.	эксп.	теор.	эксп.	теор.
1	1	1	0	0	1	1	1	0
3	3	3	1	1	2	2	2	2
10	109	109	54	54	55	55	9	9
14	753	753	376	376	377	377	13	13
18	5 167	5 167	2 583	2 583	2 584	2 584	17	17

Таблица П.4
Результаты анализа трудоемкости алгоритма

m	$f_R(m)$		$f_{CL}(m)$		$f_{CV}(m)$		$f_A(m)$	
	эксп.	теор.	эксп.	теор.	эксп.	теор.	эксп.	теор.
1	14	14	4	4	0	0	18	18
3	42	42	8	8	7	7	57	57
10	1 526	1 526	220	220	378	378	2 124	2 124
14	10 542	10 542	1 508	1 508	2 632	2 632	14 682	14 682
18	72 338	72 338	10 336	10 336	18 081	18 081	100 755	100 755

Аналогично с таблицами П.1 и П.2 значения ошибок не приводятся, т. к. и в данном случае результаты экспериментов полностью совпадают с теоретическими значениями.

3.3. Стандартные математические функции — вычисление квадратного корня

Нахождение корней квадратных уравнений — одна из распространенных подзадач, встречающихся в процессе решения многих проблем. Для вычисления корней квадратного уравнения, помимо основных

арифметических операций, требуется операция вычисления квадратного корня.

3.3.1. Программная реализация рекурсивного алгоритма

```
// функция Sq вычисляет квадратный корень из a<4
// за n шагов рекурсии
function Sq( a: Extended; n: Integer ): Extended;
begin
  if n=1 then
    Sq := 1.5
  else begin
    y := Sq(a, n-1);
    Sq := 0.5*(y+a/y);
  end;
end;
// функция Sqrt2 вычисляет квадратный корень из произвольного a,
// рекурсивно вызывая себя с аргументом a/4, пока не выполнится a<4,
// затем вызывает Sq с глубиной 6
function Sqrt2( a: Extended ): Extended;
begin
  if a<4 then
    Sqrt2 := Sq(a,6)
  else begin
    Sqrt2 := 2*Sqrt2(a/4);
  end;
end;
```

3.3.2. Результаты экспериментальных исследований

С программной реализацией рекурсивного алгоритма вычисления квадратного корня были проведены эксперименты, в результате которых были получены значения характеристик для дерева рекурсии и трудоемкости программной реализации, приведенные в соответствующих таблицах. Полученные исследуемым алгоритмом значения квадратных корней не приводятся, т. к. полностью совпадают со значениями стандартной функции языка Delphi. Результаты в таблице П.5 приводятся в сравнении с теоретическими значениями, рассчитанными по формулам (7.3.11), т. е. для полного дерева рекурсии, включая фрагмент дерева, порождаемый функцией **Sq**.

При теоретическом анализе данного алгоритма трудоемкость останова рекурсии в первой фазе алгоритма — функции **Sqrt2(a)**, была принята равной полной трудоемкости второй фазы алгоритма — функции **Sq(a,6)**. Поэтому в таблице П.6 для компонента $f_{CL}(a)$ приводится, полученное в параграфе 7.3 теоретическое значение общей трудоемкости второй фазы при $n = 6$ — $f_{CL}(a) = 145$, а для $f_R(a)$ и $f_{CV}(a)$ — соответствующие трудоемкости только для функции **Sqrt2(a)**. Мы также не приводим значения ошибок, т. к. для данного алгоритма результаты, полученные в ходе экспериментов с программной реализацией, полностью совпадают с теоретическими значениями.

Таблица П.5

Результаты анализа дерева рекурсии

a	$R(a)$		$R_V(a)$		$R_L(a)$		$H_R(a)$	
	эксп.	теор.	эксп.	теор.	эксп.	теор.	эксп.	теор.
2,0	7	7	6	6	1	1	7	7
7,5	8	8	7	7	1	1	8	8
146,8	10	10	9	9	1	1	10	10
512,0	11	11	10	10	1	1	11	11
1 000,0	11	11	10	10	1	1	11	11
10 000,0	13	13	12	12	1	1	13	13
1 000 000,0	16	16	15	15	1	1	16	16

Таблица П.6

Результаты анализа трудоемкости алгоритма

a	$f_R(a)$		$f_{CL}(a)$		$f_{CV}(a)$		$f_A(a)$	
	эксп.	теор.	эксп.	теор.	эксп.	теор.	эксп.	теор.
2,0	14	14	147	147	0	0	161	161
7,5	28	28	147	147	4	4	179	179
146,8	56	56	147	147	12	12	215	215
512,0	70	70	147	147	16	16	233	233
1 000,0	70	70	147	147	16	16	233	233
10 000,0	98	98	147	147	24	24	269	269
1 000 000,0	140	140	147	147	36	36	323	323

3.4. Программное обеспечение современных криптосистем — введение числа в целую степень

Система асимметричного шифрования RSA разработана в 1977 году исследователями из Массачусетского Технологического Института Рональдом Ривестом, Ади Шамиром и Леонардом Адлеманом. Как шифрование, так и дешифрование сообщения выполняется по одному и тому же алгоритму: сообщение (последовательность битов, рассматриваемая, как большое целое число) возводится в целую степень и берется остаток от деления результата на другое целое число (модуль). Ключ при этом состоит из двух чисел: показателя степени и модуля. Алгоритм RSA позволяет подобрать такое значение модуля и пару показателей степени, что сообщение, зашифрованное одним ключом, расшифровывается другим. Соответственно, один из ключей может быть сделан открытым, публично доступным. Все, что будет зашифровано открытым ключом, сможет расшифровать только владелец второго (секретного) ключа из пары. Второе применение системы асиммет-

ричного шифрования – цифровая подпись, подтверждение авторства сообщения. Если зашифрованное сообщение удалось расшифровать открытым ключом, значит, оно было зашифровано парным секретным ключом, а сделать это мог только его владелец. Надежность системы асимметричного шифрования зависит от того, насколько сложно, зная открытый ключ, подобрать парный ему секретный. Криптостойкость алгоритма RSA опирается на то, что в настоящее время неизвестен достаточно быстрый алгоритм для разложения больших чисел на простые множители. Таким образом, основой многих современных процедур шифрования и дешифрования является возвведение чисел в целую степень.

3.4.1. Программная реализация рекурсивного алгоритма

```
// функция Power возводит x в степень m
function Power( x: Extended; m: Integer ): Extended;
var z: Extended;
begin
  if m=0 then
    Power := 1
  else begin
    if m mod 2=0 then begin
      z := Power(x,m Div 2);
      Power := z*z;
    end
    else begin
      z := Power(x,m Div 2);
      Power := z*z*x;
    end;
  end;
end;
```

3.4.2. Результаты экспериментальных исследований

Эксперименты, проведенные с программной реализацией рекурсивного алгоритма возведения числа в целую степень позволили получить значения характеристик для дерева рекурсии и трудоемкости программной реализации, которые мы приводим в таблицах П.7, П.8 и П.9. Поскольку значение числа x не влияет на трудоемкость данного алгоритма, то оно было зафиксировано во всех экспериментах значением $x = 1,000001$. Результаты экспериментов по анализу дерева рекурсии в таблице П.7 приводятся в сравнении с теоретическими значениями, рассчитанными по формуле (7.4.2). При теоретическом анализе данного алгоритма были получены две формулы для трудоемкости – точная (7.4.3), использующая функции $\beta_0(m)$ и $\beta_1(m)$, и формула для средней трудоемкости (7.4.6) на основе функции $\beta(m)$. В связи с этим в таблице П.8 приводятся, в сравнении с экспериментальными результатами, теоретические значения трудоемкости, полученные на основе точной формулы (в скобках указано двоичное представление числа m). В таблице П.9 – среднее значение трудоемкости, полученное

на основе усреднения результатов при фиксированном значении n , соответствующий диапазон для m приведен в скобках.

Таблица П.7
Результаты анализа дерева рекурсии

m	$R(m)$		$R_V(m)$		$R_L(m)$		$H_R(m)$	
	эксп.	теор.	эксп.	теор.	эксп.	теор.	эксп.	теор.
1	2	2	1	1	1	1	2	2
2	3	3	2	2	1	1	3	3
1023	11	11	10	10	1	1	11	11
1024	12	12	11	11	1	1	12	12
1327	12	12	11	11	1	1	12	12
100 000	18	18	17	17	1	1	18	18
1 000 000	21	21	20	20	1	1	21	21

Таблица П.8
Результаты анализа трудоемкости алгоритма

m	$f_R(m)$		$f_{CL}(m)$		$f_{CV}(m)$		$f_A(m)$	
	эксп.	теор.	эксп.	теор.	эксп.	теор.	эксп.	теор.
1 (1)	36	36	2	2	8	8	46	46
2 (10)	54	54	2	2	15	15	71	71
1023 (111111111)	198	198	2	2	80	80	280	280
1024 (100 000 0000)	216	216	2	2	78	78	296	296
1327 (101 0010 1111)	216	216	2	2	84	84	302	302
100 000 (1 1000 0110 1010 0000)	324	324	2	2	125	125	451	451
1 000 000 (1111 0100 0010 0100 0000)	378	378	2	2	147	147	527	527

3.5. Программное обеспечение современных криптосистем — умножение длинных целых чисел

Рассмотрим чуть подробнее процесс генерации ключей в системе RSA, описанной в предыдущем пункте. На первом шаге выбираются

Таблица П.9

Результаты анализа трудоемкости алгоритма в среднем

n	$\bar{f}_A(n)$		Δf	
	эксп.	теор.	абс.	отн.
6 ($32 \leq m \leq 63$)	173,5	173,5	0	0
8 ($128 \leq m \leq 255$)	224,5	224,5	0	0
10 ($512 \leq m \leq 1023$)	275,5	275,5	0	0

два больших простых числа p и q . Эти числа, помимо простоты, должны иметь длину в сотни бит (в настоящее время достаточно надежным выбором считаются числа от 512 бит), а числа $(p - 1)$ и $(q - 1)$ — иметь хотя бы один большой простой множитель. Для проверки простоты числа известен целый ряд быстрых алгоритмов, в частности, алгоритм Миллера-Рабина. На втором шаге вычисляются произведение простых чисел $N = pq$, которое будет служить модулем, и функция Эйлера $\varphi(N) = (p - 1)(q - 1)$. Затем выбирается число e , взаимно простое с $\varphi(N)$ и подбирается число d , такое, что $ed = 1 \pmod{\varphi(N)}$. Для вычисления d обычно используется расширенный алгоритм Евклида. Открытый ключ составляет пара (e, N) , а секретный ключ — пара (d, N) . Шифрование сообщения x выполняется вычислением $y = x^e \pmod{N}$, а дешифрование сообщения y — вычислением $x = y^d \pmod{N}$. Итак, для генерации ключей в системе RSA нужно уметь генерировать большие простые числа и быстро перемножать их. Один из быстрых алгоритмов умножения длинных целых чисел был предложен А.А. Карацубой в 1962 году.

3.5.1. Программная реализация рекурсивного алгоритма

const $M = 1024; //$ длина произведения в битах $Mh = M \text{ div } 2; //$ длина множителя в битах $LogMh = 10; //$ глубина дерева рекурсии**type** $TNum - \text{тип массива, в котором хранится длинное число}$ $//$ один бит в каждом байте массива, от старшего к младшему, $//$ младший бит — в последней ячейке $TNum = array[0..M] of Byte;$ $PNum - \text{тип указателя на массив, в котором хранится длинное число}$ $PNum = ^TNum;$ **var** $D - \text{массив записей промежуточных результатов}$ $//$ вычислений для всей текущей ветви дерева рекурсии $D: array[1..LogMh] of record$ $AB, A, B, A1B1, A2B2, A3B3, A1pA2, B1pB2: TNum;$ **end;** $//$ Процедура KSum складывает два длинных числа P^* и Q^* $//$ Результат помещается в S^*

```

// Длина аргументов в битах - Nbit
procedure KSum( Nbit: Word; P,Q,S: PNum );
var sm: word; // перенос
    i: integer;
begin
    sm := 0;
    for i := Nbit downto 1 do begin
        sm := P^ [ i ] + Q^ [ i ] + sm; // плюс перенос из sm
        S^ [ i ] := sm mod 2;
        sm := sm div 2;
    end;
    S^ [ 0 ] := sm;
end;
// Процедура KSub вычитает длинное число Q^ из P^
// Результат помещается в S^
// Длина аргументов в битах - Nbit
procedure KSub( Nbit: Word; P,Q,S: PNum );
var sm: word;
    i: integer;
begin
    sm := 1;
    for i := Nbit downto 1 do begin
        sm := P^ [ i ] + (1-Q^ [ i ]) + sm;
        S^ [ i ] := sm mod 2;
        sm := sm div 2;
    end;
    S^ [ 0 ] := 0;
end;
// Процедура KMul перемножает длинные числа D[L].A и D[L].B
// Результат помещается в D[L].AB
// Длина аргументов в битах - N
// Текущий уровень рекурсии - L
// Промежуточные результаты хранятся в D[L]
procedure KMul( N, L: Word );
var Nh, NextL, i1, i2, A4, B4: word;
begin
    if N = 1 then
        D [ L ].AB [ M ] := D [ L ].A [ M ] * D [ L ].B [ M ]
    else begin
        Nh := N div 2;
        NextL := L+1;
        i1 := M-Nh+1;
        i2 := M-N+1;
    // A1B1 = A1*B1
    Move (D [ L ].A [ i1 ], D [ NextL ].A [ i1 ], Nh);
    Move (D [ L ].B [ i1 ], D [ NextL ].B [ i1 ], Nh);

```

```

KMul(Nh,L+1);
Move(D[NextL].AB[i2],D[L].A1B1[i2],N);
// A2B2 = A2*B2
Move(D[L].A[i2],D[NextL].A[i1],Nh);
Move(D[L].B[i2],D[NextL].B[i1],Nh);
KMul(Nh,L+1);
Move(D[NextL].AB[i2],D[L].A2B2[i2],N);
// A1pA2 = A1+A2
KSum(Nh,@D[L].A[i1-1],@D[L].A[i2-1],
      @D[L].A1pA2[i1-1]);
// B1pB2 = B1+B2
KSum(Nh,@D[L].B[i1-1],@D[L].B[i2-1],
      @D[L].B1pB2[i1-1]);
// A3B3 = A3*B3
A4 := D[L].A1pA2[M];
D[L].A1pA2[M] := 0;
B4 := D[L].B1pB2[M];
D[L].B1pB2[M] := 0;
Move(D[L].A1pA2[i1-1],D[NextL].A[i1],Nh);
Move(D[L].B1pB2[i1-1],D[NextL].B[i1],Nh);
KMul(Nh,L+1);
Move(D[NextL].AB[i2],D[L].A3B3[i2-2],N);
// i2-2 - для *4
D[L].A3B3[M] := 0;
→ D[L].A3B3[M-1] := 0;
if A4=1 then begin
  FillChar(D[L].B1pB2[i2-3],Nh+2,0);
// A3B3 = A3B3+2*B3
  KSum(N+2,@D[L].A3B3[i2-3],
        @D[L].B1pB2[i2-3],@D[L].A3B3[i2-3]);
end;
if B4=1 then begin
  FillChar(D[L].A1pA2[i2-3],Nh+2,0);
  D[L].A1pA2[M] := A4; // A3B3+=A4B4
// A3B3 = A3B3+2*A3
  KSum(N+2,@D[L].A3B3[i2-3],@D[L].A1pA2[i2-3],
        @D[L].A3B3[i2-3]);
end;
// A3B3 = A3B3-A1B1
D[L].A1B1[i2-1] := 0;
D[L].A1B1[i2-2] := 0;
KSub(N+2,@D[L].A3B3[i2-3],@D[L].A1B1[i2-3],
      @D[L].A3B3[i2-3]);
// A3B3 = A3B3-A2B2
D[L].A2B2[i2-1] := 0;

```

```

D[L].A2B2[i2-2] := 0;
KSub(N+2,@D[L].A3B3[i2-3],@D[L].A2B2[i2-3],
      @D[L].A3B3[i2-3]);
// A2B2 => старшая половина A1B1
Move(D[L].A2B2[i2],D[L].A1B1[M-2*N+1],N);
// A3B3 => середина AB
FillChar(D[L].AB[M-2*N],2*N+1,0);
Move(D[L].A3B3[i2-1],D[L].AB[M-N-Nh],N+1);
// AB = AB+A1B1
KSum(2*N,@D[L].AB[M-2*N],@D[L].A1B1[M-2*N],
      @D[L].AB[M-2*N]);
end;
end;
// Для перемножения длинных целых чисел A и B (длиной до Mh битов),
// их нужно поместить в D[1].A и D[1].B
// и вызвать процедуру KMul(Mh,1)
// результат (длиной до M битов) взять в D[1].AB

```

3.5.2. Результаты экспериментальных исследований

Мы приводим в таблицах П.10, П.11 результаты, полученные в ходе экспериментов с программной реализацией рекурсивного алгоритма умножения длинных целых чисел. Результаты вычислительных экспериментов по анализу дерева рекурсии в таблице П.10 даны в сравнении с теоретическими значениями, рассчитанными в соответствии с формулами (7.5.8).

Таблица П.10

Результаты анализа дерева рекурсии

n	$k = \log_2 n$	$R(n)$		$R_V(n)$		$R_L(n)$		$H_R(n)$	
		эксп.	теор.	эксп.	теор.	эксп.	теор.	эксп.	теор.
32	5	364	364	121	121	243	243	6	6
64	6	1 093	1 093	364	364	729	729	7	7
128	7	3 280	3 280	1 093	1 093	2 187	2 187	8	8
256	8	9 841	9 841	3 280	3 280	6 561	6 561	9	9
512	9	29 524	29 524	9 841	9 841	19 683	19 683	10	10

Теоретический анализ трудоемкости данного алгоритма был проведен для среднего случая, в соответствии с этим мы приводим в таблице П.11 экспериментальные результаты, усредненные по $N_e = 10\,000$ экспериментам. В каждом таком эксперименте исходные массивы длины n , соответствующие умножаемым числам, заполнялись значениями 0 или 1 с использованием генератора случайных чисел. Трудоемкость одного вычислительного эксперимента определялась на основе счетчика базовых операций, а среднее значение вычислялось усреднением по N_e экспериментам. Теоретические значения трудоемкости рассчитаны

по формуле (7.5.16). В последнем столбце мы приводим абсолютную и относительную погрешности для теоретической трудоемкости относительно экспериментальных результатов.

Таблица П.11

Результаты анализа трудоемкости алгоритма в среднем

n	$\bar{f}_A(n)$		погрешность Δf	
	эксперимент	теория	абсолютная	относительная
32	156 779,70	152 858,00	- 3 921,70	2,57%
64	480 141,90	468 271,00	- 11 870,90	2,54%
128	1 459 317,30	1 423 502,00	- 35 815,30	2,52%
256	4 414 525,80	4 307 179,00	- 107 346,80	2,49%
512	13 316 996,00	12 994 178,00	- 322 818,00	2,48%

3.6. Алгоритмическое обеспечение общего назначения — сортировка массива чисел слиянием

Алгоритм сортировки слиянием (Дж. фон Нейман, 1945 г.) делит массив на две части, рекурсивно сортирует каждую часть, а затем сливает их воедино. Для слияния двух частей используется дополнительная буферная память. На каждой итерации цикла слияния в буфер переносится наименьший из двух элементов, стоящих в начале частей. Сортировка слиянием хорошо приспособлена к обработке внешних файлов, чей размер превышает размер доступной памяти, а также потоков данных, которые нужно сортировать по мере их поступления. В этом случае каждая поступающая порция данных сортируется, а затем сливается с отсортированным массивом данных, накопленных к этому времени. Многопутевая сортировка слиянием позволяет эффективно использовать возможности кластерных и параллельных систем.

3.6.1. Программная реализация рекурсивного алгоритма

```

const maxN = 2500; // длина массива
var
// A - сортируемый массив
A: array [1..maxN] of Extended;
// Bp, Bq - служебные массивы
Bp, Bq: array [1..(maxN div 2)+2] of Extended;
// процедура Merge выполняет слияние двух подмассивов
// A[p..r] и A[r+1..q] в подмассив A[p..q]
procedure Merge(p, r, q: Integer);
var AMax: Extended; // текущий максимум
    i, kp, kq, pl, pp, pq: Integer;
begin
    AMax := A[r];
    for i := r+1 to q do
        if A[i] > AMax then
            AMax := A[i];
    for i := r+1 to q do
        if A[i] <= AMax then
            Bp[kp] := A[i];
            kp := kp + 1;
        else
            Bq[kq] := A[i];
            kq := kq + 1;
    for i := p to r do
        A[i] := Bp[i];
    for i := r+1 to q do
        A[i] := Bq[i];

```

```

if AMax < A[q] then
    AMax := A[q];
// перенос A[p..r] в Bp[1..r-p+1]
kp := r-p+1;
p1 := p-1;
for i := 1 to kp do
    Bp[i] := A[p1+i];
// устанавливаем ограничитель в конце Bp
Bp[kp+1] := AMax;
// перенос A[r+1..q] в Bq[1..q-r]
kq := q-r;
for i := 1 to kq do
    Bq[i] := A[r+i];
// устанавливаем ограничитель в конце Bq
Bq[kq+1] := AMax;
// выбираем максимальные элементы из начала массивов Bp и Bq
// и переносим их в A
pp := 1;
pq := 1;
for i := p to q do
    if Bp[pp] < Bq[pq] then begin
        A[i] := Bp[pp];
        pp := pp+1;
    end
    else begin
        A[i] := Bq[pq];
        pq := pq+1;
    end;
end;
end;
// процедура Merge_Sort выполняет сортировку слиянием
// подмассива A[p..q]
procedure Merge_Sort(p, q: Integer);
var r: Integer;
begin
    if p <> q then begin
        r := (p+q) div 2; // точка разделения массива A
        Merge_Sort(p,r);
        Merge_Sort(r+1,q);
        Merge(p,r,q);
    end;
end;
// Для сортировки массива длиной N его нужно поместить в A[1..N]
// и вызвать процедуру Merge_Sort(1,N)
// Результат будет размещен на том же месте (A[1..N])

```

3.6.2. Результаты экспериментальных исследований

Мы приводим в таблицах П.12, П.13 и П.14 результаты экспериментальных исследований для программной реализации рекурсивного алгоритма сортировки слиянием по характеристикам дерева рекурсии и трудоемкости данного алгоритма. Результаты экспериментов по анализу дерева рекурсии в таблице П.12 приводятся в сравнении с теоретическими значениями, которые рассчитаны в соответствии с формулами (7.6.2). Напомним, что в теории формулы для количества вершин дерева справедливы как в случае $n = 2^k$, так и в случае $n \neq 2^k$ — мы явно демонстрируем это в таблице. Значения $H_R(n)$ рассчитаны для первого случая по формуле (7.6.2), для второго — по (7.6.11).

Таблица П.12
Результаты анализа дерева рекурсии

n	$R(n)$		$R_V(n)$		$R_L(n)$		$H_R(n)$	
	эксп.	теор.	эксп.	теор.	эксп.	теор.	эксп.	теор.
8	15	15	7	7	8	8	4	4
256	511	511	255	255	256	256	9	9
1024	2 047	2 047	1 023	1 023	1 024	1 024	11	11
2048	4 095	4 095	2 047	2 047	2 048	2 048	12	12
93	185	185	92	92	93	93	8	8
827	1 653	1 653	826	826	827	827	11	11
1324	2 647	2 647	1 323	1 323	1 324	1 324	12	12
1778	3 555	3 555	1 777	1 777	1 778	1 778	12	12

Теоретический анализ трудоемкости данного алгоритма был проведен для двух случаев, в соответствии с этим мы приводим экспериментальные результаты в двух таблицах — П.13 для случая 1 ($n = 2^k$, формула для трудоемкости (7.6.7)), и П.14 — для случая 2 ($n \neq 2^k$, формула для трудоемкости (7.6.13)). Отметим, что формула (7.6.13) при $n = 2^k$ совпадает с формулой (7.6.7). В теории расхождение реальной трудоемкости от среднего значения Δf не превышает $n - 1$, экспериментальные значения Δf приведены в последнем столбце таблицы. Целью данных экспериментов была проверка максимального расхождения теоретической функции трудоемкости в среднем и полученных экспериментальных значений. В связи с этим в таблицах П.13 и П.14 мы приводим экспериментальную трудоемкость, имеющую наибольшее расхождение с теоретической формулой по 1000 экспериментов со случайными массивами данной размерности.

Таблица П.13

Результаты анализа трудоемкости алгоритма — случай 1

n	$f_A(n)$		$\Delta f_{\max}(n)$	
	эксп.	теор.	эксп.	теор.
8	1 041	1 046	5	7
256	58 503	58 558	55	255
1024	271 225	271 294	69	1023
2048	579 447	579 518	71	2047

Таблица П.14

Результаты анализа трудоемкости алгоритма — случай 2

n	$f_A(n)$		$\Delta f_{\max}(n)$	
	эксп.	теор.	эксп.	теор.
93	18 897	18 927	30	92
827	215 413	215 543	130	826
1324	361 363	361 594	231	1323
1778	498 005	498 248	243	1777

3.7. Современные методы оптимизации — генетический поиск экстремума функции нескольких переменных

Генетические алгоритмы — это группа методов поиска решений в многомерном пространстве. Решение должно быть представлено в виде вектора (хромосомы), в котором каждый элемент (ген) принимает значения из соответствующей области определения. Функция приспособленности сопоставляет каждому возможному решению его оценку — числовое значение. На каждом шаге алгоритма оценивается приспособленность популяции, после чего часть хромосом подвергается мутации (в хромосоме случайно изменяются отдельные гены) и кроссинговеру (пара хромосом обменивается своими частями). Если приспособленность новой популяции улучшилась, она становится новым поколением, иначе — нет. Традиционно генетические алгоритмы реализуются в виде цикла, однако с такой реализацией связана трудность априорной оценки количества шагов, требуемых для нахождения решения приемлемой точности.

Рекурсивный генетический алгоритм заменяет циклическое повторение развертыванием дерева рекурсии, в листьях которого находятся случайные хромосомы, а по дереву вверх передаются лучшие из них. В каждом промежуточном узле при этом сравниваются восемь хромосом: две, пришедшие от потомков, две, полученные путем их кроссинговера, и четыре, полученные мутацией. Результатом работы алгоритма является хромосома, доставляющая наилучшее значение функции приспособленности.

3.7.1. Программная реализация рекурсивного алгоритма

const

```
maxN = 30; // максимальная глубина рекурсии
M = 12; // размерность вектора
```

var

```
// Vx - массив геномов для текущей ветви дерева рекурсии
// первый индекс - текущая глубина рекурсии
// второй индекс - номер хромосомы
// третий индекс - значение функции (0) и координаты вектора(1..M)
Vx: array [1..maxN, 1..8, 0..M] of Extended;
```

```
// функция G, минимум которой ищется алгоритмом
// параметр n - текущая глубина рекурсии
```

```
// параметр j - номер хромосомы
```

```
// значение функции G для хромосомы Vx[n,j,1..M]
```

```
// записывается в Vx[n,j,0] и возвращается функцией
```

```
function G( n, j: Integer ): Extended;
```

```
var i: Integer;
```

```
    S, Z: Extended;
```

```
begin
```

```
    S := 0;
```

```
    for i := 1 to M do begin
```

```
        Z := Vx[n, j, i] - 0.3;
```

```
        S := S + Z * Z;
```

```
    end;
```

```
    Vx[n, j, 0] := S;
```

```
    Result := S;
```

```
end;
```

```
// функция F ищет минимум функции G на глубине n дерева рекурсии
```

```
// результат помещается в Vx[n,1,0] (значение функции G)
```

```
// Vx[n,1,1..M] - хромосома, доставляющая минимум функции G
```

```
procedure F( n: Integer );
```

```
var i, j, k: Integer;
```

```
    GMin: Extended;
```

```
begin
```

```
    if n = 1 then begin
```

```
        // на уровне 1 генерируем случайную хромосому
```

```
        for i := 1 to M do
```

```
            Vx[n, 1, i] := Random;
```

```
            Vx[n, 1, 0] := G(n, 1); // значение целевой функции
```

```
    end
```

```
    else begin
```

```
        // рекурсия к одному ребенку, результат переносим в Vx[n,1,*]
```

```

F(n-1);
for i := 0 to M do
  Vx[n,1,i] := Vx[n-1,1,i];
// рекурсия ко второму ребенку, результат переносим в Vx[n,2,*]
F(n-1);
for i := 0 to M do
  Vx[n,2,i] := Vx[n-1,1,i];
// кроссинговеры Vx[n,3] и Vx[n,4] - из Vx[n,1] и Vx[n,2]
// k - точка разрыва; кроссируются гены с 1 по k включительно
  k := 2+Random(M-2);
// 2<=k<=m-1, Random(m-2) возвращает 0..m-3
  for i := 1 to k do begin
    Vx[n,3,i] := Vx[n,2,i];
    Vx[n,4,i] := Vx[n,1,i];
  end;
  for i := k+1 to M do begin
    Vx[n,3,i] := Vx[n,1,i];
    Vx[n,4,i] := Vx[n,2,i];
  end;
// Vx[n,5..8] копируются из Vx[n,1..4]
  for j := 1 to 4 do
    for i := 1 to M do
      Vx[n,j+4,i] := Vx[n,j,i];
// мутация случайных генов в Vx[n,5..8]
  for j := 5 to 8 do begin
    k := 1+Random(m); // 1<=k<=m
    Vx[n,j,k] := Vx[n,j,k]+Random;
    if Vx[n,j,k] >= 1 then
      Vx[n,j,k] := Vx[n,j,k]-1;
  end;
// расчет значений функции G для хромосом 3..8
  for j := 3 to 8 do
    Vx[n,j,0]:=G(n,j); // Vx[n,j,0] := G(Vx[n,j,1..M])
// выбор наилучшей из восьми хромосом
// k - номер наилучшей хромосомы
  k := 1;
  GMin := Vx[n,1,0];
  for j := 2 to 8 do begin
    if Vx[n,j,0] < GMin then begin
      k := j;
      GMin := Vx[n,j,0];
    end;
  end;
// перенос наилучшей хромосомы в Vx[n,1,*]
  for i := 0 to M do
    Vx[n,1,i] := Vx[n,k,i];

```

```

end;
end;
// Для нахождения минимума функции G вызывается F(1)
// Результат - в Vx[1,1,*]

```

3.7.2. Результаты экспериментальных исследований

В таблицах П.15 и П.16 показаны результаты проведенных экспериментальных исследований с программной реализацией рекурсивного генетического алгоритма поиска минимума многомерной функции. Поскольку для каждой точки алгоритм однократно вычисляет значение целевой функции, то количество точек определялось счетчиком обращений к функции $g(x)$. Результаты по анализу дерева рекурсии и количеству порожденных точек приведены в таблице П.15 в сравнении с теоретическими значениями, рассчитанными по формулам (7.7.5) и (7.7.7).

Т а б л и ц а П.15
Результаты анализа дерева рекурсии

<i>n</i>	<i>R(n)</i>		<i>R_V(n)</i>		<i>H_R(n)</i>		<i>P(n)</i>	
	эксп.	теор.	эксп.	теор.	эксп.	теор.	эксп.	теор.
14	16 383	16 383	8 192	8 192	14	14	57 338	57 338
16	65 535	65 535	32 768	32 768	16	16	229 370	229 370
18	*262 143	262 143	131 072	131 072	18	18	917 498	917 498
20	1 048 575	1 048 575	524 288	524 288	20	20	3 670 010	3 670 010

Теоретический анализ трудоемкости данного алгоритма был проведен для трудоемкости в среднем (см. формулу (7.7.8)). Поскольку фрагмент поиска минимума из 8 чисел вызывается в каждой внутренней вершине дерева рекурсии, то при однократном выполнении программной реализации мы имеем усреднение по $R_V(n)$ обращениям к этому фрагменту, что при тестовых значениях n является вполне достаточным. Тестовая функция была определена в 12-мерном пространстве, т. е. значение $m = 12$, в соответствии с этим и была рассчитана теоретическая трудоемкость. Мы не ставили явной целью исследование предложенного генетического алгоритма на качество получаемых решений, тем не менее, данные, полученные в экспериментах, показали вполне приемлемые результаты. Внушительные значения трудоемкости на самом деле становятся вполне допустимыми при переходе к временным оценкам. Среднее время на одну базовую операцию для процессора Р-IV имеет порядок 5 тактов для смеси операций индексации и обработки целых и действительных чисел, что приводит при $n = 20$, $m = 12$ и тактовой частоте в 3 ГГц к оценке времени выполнения порядка 1,5 сек.

Таблица П.16

Результаты анализа трудоемкости алгоритма в среднем при $m = 12$

n	$\bar{f}_A(n)$		погрешность Δf	
	эксперимент	теория	абсолютная	относительная
14	15 351 887,08	15 371 468,8	-19 581,76	0,1274%
16	61 412 829,05	61 490 527,1	-77 698,08	0,1264%
18	245 655 061,92	245 966 760,2	-311 698,35	0,1267%
20	982 624 823,46	983 871 692,8	-1 246 869,38	0,1267%

3.8. Классические алгоритмы на графах — нахождение оствового дерева

Графы позволяют естественным образом описывать множество задач — от поиска путей между городами до синтаксического разбора текстов. Так, например, локальные компьютерные сети в настоящее время обычно строят на коммутаторах, работающих по алгоритму прозрачного моста. В таких сетях нельзя допускать появления циклов, иначе возможно лавинообразное размножение каждого пакета. С другой стороны, для повышения надежности сети часто желательно иметь резервные или дублирующие каналы связи между коммутаторами, что приводит к появлению циклов. Современные коммутаторы решают это противоречие с помощью протокола построения оствового дерева, т.е. такого ациклического связного подграфа, который содержит все узлы исходного графа. Все каналы, не вошедшие в оствовое дерево, не используются. Оствовое дерево регулярно перестраивается, что позволяет исключать из него неработоспособные каналы и задействовать резервные. Протокол построения оствового дерева STP выполняется всеми коммутаторами, причем ни один из них не обладает полной информацией о топологии сети.

Мы рассматриваем более простую задачу нахождения оствового дерева при известной топологии сети (например, при известной матрице смежности графа), решаемую алгоритмом поиска в глубину, предложенным Тарьянном.

3.8.1. Программная реализация рекурсивного алгоритма

```
const
```

```
Nvertex = 100; // максимальное количество вершин графа
```

```
var
```

```
// G - матрица смежности
```

```
G: array [1..Nvertex, 1..Nvertex] of Byte;
```

```
// Visit - вектор посещений узлов
```

```
Visit: array [1..Nvertex] of Byte;
```

```
// T - оствовое дерево
```

```
// i-е ребро оства соединяет вершины T[i,1] и T[i,2]
```

```
T: array [1..100, 1..2] of Word;
```

```
N: Word; // количество вершин графа
```

```

M: Word; // количество ребер в остове
// процедура поиска в глубину от вершины V
procedure DepthFirstSearch( V: Word );
var W: word;
begin
// отмечаем посещение вершины V
Visit[V] := 1;
// заходим во все смежные непосещенные вершины W
for W := 1 to N do begin
  if (G[V,W] = 1) then begin
    if (Visit[W] = 0) then begin
// добавляем ребро (V,W) в остов
    M := M+1;
    T[M,1] := V;
    T[M,2] := W;
    DepthFirstSearch(W);
  end;
end;
end;
end;
// Для построения остовного дерева установить N=количество вершин,
// занести матрицу смежности в G, обнулить вектор Visit и массив T,
// вызвать DepthFirstSearch(1)
// Результат (список ребер остовного дерева) - в массиве T

```

3.8.2. Результаты экспериментальных исследований

Результаты экспериментов с программной реализацией алгоритма Тарьяна приведены в таблицах П.17 и П.18. Поскольку для этого

алгоритма количество листьев и внутренних вершин не определялось в теории из-за сильной параметрической зависимости, то в таблице П.17 приведены значения только для общего числа вершин дерева рекурсии в сравнении с теоретическими значениями, рассчитанными по формуле (7.8.2). В эксперименте генерировались случайные матрицы стоимости на основе стандартного равномерного генератора, что обеспечивало в среднем $(n - 1)/2$ ребер у каждой вершины и очевидную связность графа.

Точная формула теоретической трудоемкости данного алгоритма была получена при

условии, что количество ребер графа известно. В связи с этим была проведена серия экспериментов, в которой для каждого фиксированного числа вершин генерировались случайные графы, содержащие заранее определенное количество ребер. Учитывались только те результаты, в которых сгенерированный граф оказался связным. Количество

n	$R(n)$	
	эксп.	теор.
10	10	10
25	25	25
50	50	50
100	100	100

Таблица П.18

Результаты анализа трудоемкости алгоритма

Число вершин n	Число ребер m	$f_A(n, m)$	
		эксперимент	теория
10	14	898	898
10	18	914	914
10	23	934	934
10	27	950	950
25	90	4 727	4 727
25	120	4 847	4 847
25	150	4 967	4 967
25	180	5 087	5 087
50	368	17 714	17 714
50	490	18 202	18 202
50	613	18 694	18 694
50	735	19 182	19 182
100	1485	68 432	68 432
100	1980	70 412	70 412
100	2475	72 392	72 392
100	2970	74 372	74 372

ребер задавалось равным ближайшему целому к 30%, 40%, 50%, 60% от количества ребер соответствующего полного неориентированного графа на n вершинах — $n(n - 1)/2$. Полученные результаты приведены в таблице П.18 в столбце экспериментальных данных, теоретические значения были рассчитаны по формуле (7.8.5).

3.9. Оптимальные экономические решения — пакет акций с максимальной доходностью — одномерная оптимальная упаковка

Задача одномерной оптимальной упаковки описывает ситуацию, когда имеется ограниченный объем, который можно заполнить грузами разных типов (каждому типу соответствует объем и стоимость груза) в разных комбинациях. Целью является определение набора грузов, помещающихся в заданный объем, совокупная стоимость которых максимальна.

Например, имеется фиксированное количество денег, на которые можно купить акции различных предприятий. Пусть все акции одного

предприятия имеют одинаковую стоимость и фиксированную заранее известную доходность. Акции разных предприятий могут иметь разную стоимость и доходность. Тогда нужно определить, сколько акций каких предприятий нужно купить, чтобы в последующем совокупный доход с них был как можно больше.

Задача одномерной упаковки решается с помощью метода динамического программирования Беллмана.

3.9.1. Программная реализация рекурсивного алгоритма

```

const
  Nmax = 100; // максимальное количество типов грузов
var
  boxC: array [1..Nmax] of Word; // стоимости грузов
  boxV: array [1..Nmax] of Word; // объемы грузов
  // оптимальные стоимости
  XArr: array [1..Nmax, 0..Nmax] of Word;
  N: Word; // количество типов грузов
// процедура F получает оптимальную упаковку
// X типов грузов в объеме V
// результат: стоимость - в XArr[X,0]
// количество грузов i-го типа - в XArr[X,i]
procedure F( V, X: Word );
var Cost, k, i, j: Word;
begin
  if X=1 then begin
    // Останов рекурсии - прямое вычисление при x=1
    k := V div boxV[1];
    XArr[1,0] := k*boxC[1];
  // функция возвращает оптимальную стоимость
    for j:=2 to N do
      XArr[1,j] := 0; // все Xj=0 кроме X1=k
    XArr[1,1] := k;
  end
  else begin // Рекурсивные вызовы для определения max F
    for j:=0 to N do
      XArr[X,j] := 0; // начальный max: функция=0, все Xj=0
    k := V div boxV[X];
  // перебираем все возможные количества груза типа X
    for i:=0 to k do begin
    // рекурсивно размещаем X-1 тип грузов в оставшемся пространстве
      F((V-boxV[x]*i),X-1);
      Cost := i*boxC[X]+XArr[X-1,0];
      if Cost >= XArr[X,0] then begin
    // копирование максимума от X-1
      for j:=1 to N do
        XArr[X,j] := XArr[X-1,j];
    
```

```

    XArr[X,X] := i;
    XArr[X,0] := Cost;
  end;
end;
end;
end;

```

3.9.2. Результаты экспериментальных исследований

В таблицах П.19 и П.20 мы приводим результаты экспериментов с программной реализацией алгоритма Беллмана. Поскольку для этого алгоритма характеристики дерева рекурсии зависят не только от коли-

Таблица П.19

Результаты анализа дерева рекурсии

n	k	$R(n, k)$		$R_V(n, k)$		$R_L(n, k)$		$H_R(n)$	
		эксп.	теор.	эксп.	теор.	эксп.	теор.	эксп.	теор.
8	3	330	330	210	210	120	120	8	8
8	4	792	792	462	462	330	330	8	8
8	5	1 716	1 716	924	924	792	792	8	8
10	3	715	715	495	495	220	220	10	10
10	4	2 002	2 002	1 287	1 287	715	715	10	10
10	5	5 005	5 005	3 003	3 003	2 002	2 002	10	10
12	3	1 365	1 365	1 001	1 001	364	364	12	12
12	4	4 368	4 368	3 003	3 003	1 365	1 365	12	12
12	5	12 376	12 376	8 008	8 008	4 368	4 368	12	12

чества типов грузов, но и от параметра k , то в таблице П.19 приведены экспериментальные характеристики дерева рекурсии для различных значений k при фиксированном n в сравнении с теоретическими значениями, рассчитанными по формулам (7.9.8). В эксперименте генерировались случайные входные массивы на основе стандартного равномерного генератора языка программирования *Delphi*. Значение объема упаковки было фиксировано и равно $V = 100$, а объемы типов грузов выбирались случайно из диапазона, гарантирующего фиксированное и одинаковое для всех типов грузов значение k , следующим образом:

$$k = 3 : \quad 26 \leq v_i \leq 33, \quad k = 4 : \quad 21 \leq v_i \leq 25, \quad k = 5 : \quad 17 \leq v_i \leq 20.$$

Поскольку стоимости типов грузов не влияют на характеристики дерева рекурсии, то они, для определенности, выбирались случайно из диапазона от 50 до 100.

При анализе трудоемкости проверялась формула для среднего значения. В эксперименте подсчитывалось количество базовых операций для конкретного входа, обладающего заданной параметризацией. Для каждой пары n, k было проведено 10000 экспериментов, их усреднен-

Таблица П.20

Результаты анализа трудоемкости алгоритма

n	k	$\bar{f}_A(n, k)$		
		эксперимент	теория	$\Delta f\%$
8	3	51 916,40	53 383	2,75%
8	4	123 024,80	126 274	2,57%
8	5	263 581,40	270 187	2,44%
10	3	133 124,40	136 933	2,78%
10	4	367 103,30	378 147	2,92%
10	5	906 313,50	933 916	2,96%
12	3	293 371,20	302 103	2,89%
12	4	926 885,60	954 255	2,87%
12	5	2 587 106,90	2 672 471	3,19%

ные результаты приведены в таблице П.20 в столбце экспериментальных данных, теоретические значения были рассчитаны по формуле (7.9.15) при соответствующем выборе коэффициента α_f для средней трудоемкости.

4. Обсуждение результатов и выводы

4.1. Характеристики дерева рекурсии

Экспериментальные значения характеристик дерева рекурсии, полученные при исследовании программных реализаций всех алгоритмы, точно совпали с теоретическими оценками, приведенными авторами книги в главе 7.

4.2. Функции трудоемкости

Для программных реализаций алгоритмов вычисления факториала, нахождения чисел Фибоначчи, вычисления квадратного корня, нахождения остовного дерева экспериментальные значения функций трудоемкости точно совпали с теоретическими оценками.

Максимальная относительная погрешность теоретической оценки трудоемкости сортировки слиянием составила по экспериментальным данным для массивов длины кратной степени двойки ($n = 2^k$) 0,012%, а для массивов прочих длин ($n \neq 2^k$) 0,05% при том, что теоретически максимальная относительная погрешность может достигать соответственно 0,35% и 0,36%.

Для остальных алгоритмов исследовалась только средняя трудоемкость.

4.3. Средняя трудоемкость

Для программной реализации алгоритма возвведения в целую степень экспериментальные значения средней трудоемкости точно совпали с теоретическими оценками.

Относительная погрешность теоретической оценки средней трудоемкости программной реализации алгоритма умножения длинных чисел лежит в пределах 2,48–2,57%. Наличие этой погрешности связано с тем, что теоретические оценки основываются на предположении о равномерном распределении нулей и единиц не только в исходных перемножаемых числах, но и в перемножаемых фрагментах этих чисел на всех уровнях рекурсии. Экспериментальные результаты показывают, что это предположение не вполне оправдано.

Относительная погрешность теоретической оценки средней трудоемкости программной реализации алгоритма генетического поиска экстремума функции многих переменных лежит в пределах 0,1264–0,1274%. Наличие этой погрешности можно объяснить тем, что когда наилучшая хромосома лежит достаточно близко к экстремуму, только другая хромосома или два кроссинговера могут улучшить ее результат, а четыре мутации дают заведомо худшие значения. В результате фактически выбор осуществляется не из восьми, а лишь из четырех вариантов, и экспериментальные средние меньше теоретических.

Относительная погрешность теоретической оценки средней трудоемкости программной реализации алгоритма одномерной оптимальной упаковки составляет 2,44…3,19%. В параграфе 7.9 указано, что реальная зависимость среднего числа переприсваиваний в цикле поиска максимума, как функция от количества типов грузов и параметра, носит более сложный характер, чем та, которая была использована авторами в формулах (7.9.13) и (7.9.14). Полученные экспериментально относительные погрешности можно считать приемлемыми для прогнозирования времени выполнения алгоритма.

4.4. Выводы

Проведенные экспериментальные исследования программных реализаций рекурсивных алгоритмов подтверждают полученные в главе 7 теоретические оценки их трудоемкости и характеристики порождаемых деревьев рекурсии.

Учебное издание

*ГОЛОВЕШКИН Василий Адамович
УЛЬЯНОВ Михаил Васильевич*

ТЕОРИЯ РЕКУРСИИ ДЛЯ ПРОГРАММИСТОВ

Редактор *В.В. Панюхин*
Оригинал-макет: *В.В. Дядичев*
Оформление переплета: *А.Ю. Алексина*

Подписано в печать 30.05.06. Формат 60×90/16. Бумага офсетная.
Печать офсетная. Усл. печ. л. 18,5. Уч.-изд. л. 19,86. Тираж 1500 экз.
Заказ № 3693

Издательская фирма «Физико-математическая литература»
МАИК «Наука/Интерperiодика»
117997, Москва, ул. Профсоюзная, 90
E-mail: fizmat@maik.ru, fmlsale@mail.ru;
<http://www.fml.ru>

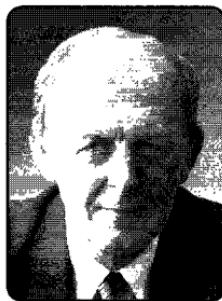
Отпечатано с готовых диапозитивов
в ППП «Типография «Наука»
121099, г. Москва, Шубинский пер., 6

ISBN 5-9221-0721-6



9 785922 107211

МАТЕМАТИКА. ПРИКЛАДНАЯ МАТЕМАТИКА



В.А. ГОЛОВЕШКИН



М.В. УЛЬЯНОВ

Доктор технических наук, специалист по механике деформируемого твердого тела.

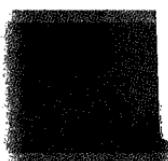
Доктор технических наук, область научных интересов – анализ и разработка эффективных вычислительных алгоритмов.

Теория рекурсии – это, наверное, скучна. Ничего подобного – это интересный полезный математический аппарат для разработки эффективных вычислительных алгоритмов.

Библио Глобус

Москва Мясницкая 6
<http://www.biblio-globus.ru>

Тел 928-35-67
924-46-80
781-19-00



Головешкин Теория рекурсии для
Цена: 213.00 978592210721