М. В. Ульянов

РЕСУРСНО-ЭФФЕКТИВНЫЕ КОМПЬЮТЕРНЫЕ АЛГОРИТМЫ. РАЗРАБОТКА И АНАЛИЗ

УЧЕБНОЕ ПОСОБИЕ

МОСКВА
НАУКА
ФИЗМАТЛИТ
2007

О Г Л А В Л Е Н И Е

Предисловие
Введение

РАЗДЕЛ I. АЛГОРИТМЫ И МОДЕЛИ ВЫЧИСЛЕНИЙ

ГЛАВА 1. Элементы теории алгоритмов

1.1. Понятие и определения алгоритма
1.2. Требования к алгоритмам и их свойства
1.3. Временная и емкостная сложность алгоритма
1.4. Основные сложностные классы задач

ГЛАВА 2. Модели вычислений и алгоритмические базисы

2.1. Введение в модели вычислений
2.2. Машина с произвольным доступом к памяти
2.3. Объектный и алгоритмический базисы
2.4. Модель вычислений для языка процедурного программирования с поддержкой механизма рекурсивного вызова

РАЗДЕЛ II. ТЕОРИЯ РЕСУРСНОЙ ЭФФЕКТИВНОСТИ КОМПЬЮТЕРНЫХ АЛГОРИТМОВ

ГЛАВА 3. Введение в теорию ресурсной эффективности компьютерных алгоритмов

3.1. Терминология и обозначения в теории ресурсной эффективности и асимптотическом анализе функций
3.2. Функции ресурсной эффективности компьютерных алгоритмов и их программных реализаций
3.3. Способы построения комплексных критериев оценки качества алгоритмов
3.4. Временные оценки и прогнозирование времён выполнения программных реализаций на основе функции трудоемкости

ГЛАВА 4. Классификационные признаки и специальные классификации компьютерных алгоритмов и задач

4.1. Классы открытых и закрытых задач и теоретическая нижняя граница временной сложности
4.2. Классификация компьютерных алгоритмов на основе угловой меры асимптотического роста функций
4.3. Классификация компьютерных алгоритмов и задач по влиянию на трудоемкость особенностей входов
4.4. Классификация компьютерных алгоритмов по информационной и размерностной чувствительности
4.5. Классификация компьютерных алгоритмов по требованиям к дополнительной памяти
4.6. Классы статических и потоковых алгоритмов и их особенности

ГЛАВА 5. Методика сравнительного анализа и рационального выбора компьютерных алгоритмов

5.1. Угловая мера близости ресурсных оценок
5.2. Методика сравнительного анализа алгоритмов по ресурсным функциям
5.3. Области эквивалентной ресурсной эффективности компьютерных алгоритмов

РАЗДЕЛ III. МЕТОДЫ РАЗРАБОТКИ ЭФФЕКТИВНЫХ АЛГОРИТМОВ

ГЛАВА 6. Универсальные методы разработки алгоритмов

6.1. Метод декомпозиции задачи
6.2. Метод рекуррентных соотношений
6.3. Метод динамического программирования

ГЛАВА 7. Методы разработки алгоритмов решения оптимизационных задач

7.1. Постановка и примеры задач целочисленного программирования
7.2. Методы разработки точных алгоритмов решения целочисленных задач
7.3. Эволюционные вычисления и генетические алгоритмы

РАЗДЕЛ IV. МЕТОДЫ АНАЛИЗА ТРУДОЕМКОСТИ КОМПЬЮТЕРНЫХ АЛГОРИТМОВ

ГЛАВА 8. Анализ итерационных алгоритмов

8.1. Методика анализа основных алгоритмических конструкций
8.2. Анализ трудоемкости количественно-зависимых алгоритмов
8.3. Метод вероятностного анализа количественно-параметрических алгоритмов
8.4. Метод амортизационного анализа

ГЛАВА 9. Анализ рекурсивных алгоритмов

9.1. Анализ вычислительной сложности рекурсивных алгоритмов, основанных на методе декомпозиции
9.2. Метод подсчета вершин дерева рекурсии
9.3. Метод рекуррентных соотношений

РАЗДЕЛ V. РАЗРАБОТКА И ВЫБОР ЭФФЕКТИВНЫХ АЛГОРИТМОВ НА ОСНОВЕ РЕСУРСНОГО АНАЛИЗА

ГЛАВА 10. Примеры эффективных алгоритмов для задач с фиксированной длиной входа

10.1. Обмен содержимого ячеек
10.2. Умножение комплексных чисел
10.3 Поиск максимума из трех чисел
10.4. Сортировка трех чисел по месту
10.5. Возведение числа в целую степень
10.6. Извлечение квадратного корня

ГЛАВА 11. Ресурсно-эффективные алгоритмические решения для некоторых задач с переменной длиной входа

11.1. Поиск минимума и максимума в массиве
11.2. Организация счетчика в массиве
11.3. Определение областей применения для алгоритмов сортировки вставками и сортировки методом индексов
11.4. Выбор алгоритма умножения длинных целых чисел на основе анализа временной эффективности

11.5. Выбор рациональных алгоритмов поиска по ключу на основе анализа их информационной чувствительности

ГЛАВА 12. Ресурсно-эффективные комбинированные алгоритмические решения

12.1. Комбинированный рекурсивно-итерационный алгоритм сортировки

12.2. Эффективный по трудоемкости комбинированный алгоритм решения классической задачи одномерной упаковки

12.3. Модификация алгоритма метода ветвей и границ для задачи коммивояжера на основе рационального использования доступного объема памяти

12.4. Рациональные ресурсно-адаптивные алгоритмические решения по компоненту формирования глобальной матрицы в системе конечно-элементного анализа

12.5. Рациональная организация аналитической базы данных с использованием алгоритмов решения задачи упаковки с динамической внутренней границей объема

П Р Е Д И С Л О В И Е

Разработка и анализ компьютерных алгоритмов — новая дисциплина, возникшая на стыке дискретной математики, программирования и классической теории алгоритмов, играющая важную роль в современных компьютерных технологиях. Для большинства практически значимых задач, решаемых сегодня с использованием компьютеров, существуют разнообразные алгоритмы их решения. Этот факт приводит к тому, что разработчики программных средств сталкиваются с проблемой выбора наиболее рационального алгоритма решения поставленной задачи. Изложить некоторые подходы к решению задачи выбора на основе теории ресурсной эффективности и продемонстрировать их на ряде модельных примеров и составляло основную цель автора при написании данного учебного пособия.

Основной материал книги базируется на лекционных курсах «Математическая логика и теория алгоритмов» и «Разработка эффективных алгоритмов», читаемых автором, на протяжении ряда лет, в Московском государственном университете приборостроения и информатики (МГУПИ), а также на научных публикациях, посвященных исследованию и анализу вычислительных алгоритмов.

Задачи анализа компьютерных алгоритмов не могут быть грамотно сформулированы без понимания основ теории алгоритмов. В связи с этим изложение материала начинается с введения в теорию алгоритмов и модели вычислений в разделе 1. Далее в разделе 2 достаточно подробно излагается теория ресурсной эффективности компьютерных алгоритмов. Раздел 3 учебного пособия посвящен краткому изложению общих методов разработки эффективных алгоритмов. В разделе 4 рассматриваются методы теоретического анализа трудоемкости компьютерных алгоритмов в рамках принятой модели вычислений. Весь этот материал иллюстрируется целым рядом примеров разработки и выбора эффективных алгоритмов на основе их ресурсного анализа в трех главах раздела 5.

Целевой аудиторией данной книги автор видит, прежде всего, студентов, аспирантов и преподавателей, и, предлагая эту книгу уважаемым читателям, автор

надеется на то, что они найдут в ней ряд полезных и интересных для себя сведений и результатов.

Автор благодарит коллектив издательства «ФИЗМАТЛИТ» за внимание и творческое сотрудничество. Особые благодарности автора — моим коллегам — коллективам кафедр «Персональные компьютеры и сети» МГУПИ и «Прикладная математика и моделирование систем» МГУП, студентам МГУПИ за проведенные экспериментальные исследования, а также всем тем, кто меня поддерживал и помогал при написании этой книги.

М. В. Ульянов,

Москва, март 2007 г.

ВВЕДЕНИЕ

АЛГОРИТМИЧЕСКОЕ ОБЕСПЕЧЕНИЕ ПРОГРАММНЫХ СРЕДСТВ И СИСТЕМ: ТРЕБОВАНИЯ И КРИТЕРИИ ОЦЕНКИ

Практически значимыми и актуальными в современных наукоемких технологиях становятся сегодня сложные задачи большой размерности и вычислительной сложности, программные системы обработки потоков задач и обслуживания потоков запросов со значительными объемами обрабатываемых данных, эффективные программные средства для бортовых вычислительных систем, работающих в режиме реального времени в условиях ограниченных вычислительных ресурсов. Примерами могут служить программные системы, использующие методы конечно-элементного анализа для задач расчета деформаций и тепловых полей в сложных объектах, программы моделирования сложных систем, программное обеспечение информационных, телекоммуникационных систем и компьютерных сетей, в том числе информационно-поисковые системы Интернета и др. Актуальность этой тематики отражена и в приоритетных направлениях развития науки России — в перечне критических технологий РФ.

К современным программным средствам и системам, предназначенным для решения указанного круга задач, предъявляется ряд достаточно жестких требований по ресурсной эффективности, при этом такие характеристики их качества, как временная эффективность и ресурсоемкость являются одними из определяющих. Решение этих приоритетных, и ряда других практически актуальных задач не может опираться только на возрастающие мощности современных компьютеров. Поскольку эффективность выбранных алгоритмических решений во многом влияет на характеристики программных систем, то, как один подходов в современных наукоемких компьютерных технологиях рассматривается путь совершенствования их алгоритмического обеспечения. При этом многообразие практических ситуаций, связанных с необходимостью разработки алгоритмического обеспечения, улучшающего ресурсные характеристики программных средств и систем, обу-

словливает актуальность разработки методов и методик исследования и сравнительного анализа ресурсной эффективности компьютерных алгоритмов.

Теоретическое исследование алгоритмов, как этап разработки компьютерного алгоритмического обеспечения, приводит к необходимости использования методов теории алгоритмов, асимптотического анализа сложности и оценки качества как самих программных средств и систем. так и их алгоритмического обеспечения. Несмотря на интенсивные исследования в области классической теории алгоритмов и асимптотического анализа сложности, нерешенными остаются некоторые вопросы сравнительного анализа ресурсной эффективности алгоритмов в реальном диапазоне длин входов, где результаты асимптотического анализа не всегда могут быть использованы.

Показатели качества программных средств и систем. В настоящее время программные средства и системы рассматриваются как новые, специфические рыночные продукты, для которых такие показатели, как качество, информационная безопасность и надежность функционирования, во многом определяют их экономическую эффективность, возможность реализации и применения. Теории и методы управления качеством программных продуктов рассматриваются сегодня как новое направление в управлении качеством продукции [В.1, В.2], разрабатываются специальные методы оценки программных систем. По определению терминологического стандарта ИСО 8402 под качеством понимается «совокупность характеристик объекта, относящихся к его способности удовлетворять установленные и предполагаемые потребности» [В.1].

Для конкретных программных систем предпочтения по критериям качества формируются на этапе их проектирования и определяются требованиями технического задания, отражающего функциональное назначение и специфику области их применения. Программы для современных компьютеров, рассматриваемые как объекты проектирования и разработки, могут быть охарактеризованы следующими обобщенными показателями [В.1, В.3], которые существенно влияют на решения по их алгоритмическому обеспечению:

— техническим назначением программ и проблемно-ориентированной областью применения. Уточнение этого обобщенного показателя в техническом за-

дании приводит к формулировке конкретных ограничений, например по временной и/или емкостной эффективности программных реализаций;

— типом решаемых функциональных задач. Этот показатель определяет в смысле алгоритмического обеспечения множество существующих алгоритмов решения этих задач, при этом выбор и/или разработка рациональных алгоритмов является одной из важных задач при создании эффективных программ;

— объемом и сложностью совокупности программ и их информационного обеспечения в разрабатываемой программной системе;

— требуемыми значениями характеристик качества функционирования программ и величиной допустимого риска.

В настоящее время существует целый ряд стандартов в области характеристик качества программных систем, достаточно полно изложенных в [В.1]. Из совокупности общих характеристик качества можно выделить следующую группу характеристик, на которые оказывают существенное влияние решения, принятые на этапе разработки алгоритмического обеспечения:

— временная эффективность — способность программы выполнять заданные действия в интервале времени, отвечающем заданным в техническом задании требованиям;

— используемость ресурсов или ресурсоемкость — минимально необходимые вычислительные ресурсы при эксплуатации программных систем. С точки зрения рациональных алгоритмических решений речь идет, в первую очередь, о ресурсах оперативной и внешней памяти;

— анализируемость — возможность прогнозирования временной эффективности и ресурсоемкости программных систем, во многом определяемой приятыми на этапе разработки математического обеспечения алгоритмическими решениями;

— изменяемость или модифицируемость — обеспечение простоты внесения необходимых изменений и доработок в программу в процессе эксплуатации, определяемая не только качеством программирования, но и «читаемостью» используемых алгоритмов;

— стабильность — устойчивая работоспособность программной системы в области входных данных, определяемой спецификой применения, обеспечиваемая не только тщательностью программирования «особых» ситуаций с данными, но и алгоритмическими решениями;

— тестируемость — полнота проверки возможных маршрутов выполнения программы в ограничениях решаемой задачи, задаваемых проблемной областью применения.

Наряду со стандартными, в настоящее время рассматривается и ряд специальных подходов к оценке характеристик качества программного обеспечения. В [В.4] предлагается модель идеализированного программного цикла, которая строится на основе длины, объема, уровней алгоритмических языков и понятия интеллектуальной емкости алгоритмов. На этой основе проводится оценка корреляции между алгоритмами и их реализациями на алгоритмических языках. Поскольку такие измеримые свойства реализации алгоритма, как количество операторов и операндов и частота их встречаемости в тексте программной реализации, зависят от алгоритмического языка, то в [В.4] вводится параметр интеллектуальной емкости, являющийся постоянным для любой из реализаций алгоритма. Этот параметр отражает количество независимых операндов на входе алгоритма и количество выходных параметров и представляет собой оценку сложности текста алгоритма.

Рассматривается также подход к оценке качества программ, основанный на предложенной в [В.5] модели аспектов программирования. На этой основе строится упорядоченная структура элементарных технических показателей качества программ. Из традиционных метрик сложности программ, как наиболее интересные и осмысленные, выбираются мера Холстеда и цикломатическое число. К сожалению, приводимые оценки свойств аспектов программирования (полнота, избыточность и согласованность) и условия формирования оценок не учитывают особенностей алгоритмических решений, принимаемых разработчиками программных систем. Однако, как отмечается в [В.5], хотя показателей качества программ чрезвычайно много, они в настоящее время не упорядочены и их недостаточно, так как многие практически важные аспекты, понятные специалисту, не отражены в этих показателях. Отметим также, что в состав существующих пока-

зателей качества программных систем явно не входят непосредственные оценки качества алгоритмического обеспечения.

Особенности оценки и требования к алгоритмическому обеспечению в различных проблемных областях. Создание алгоритмического обеспечения является важным этапом разработки программных систем. Конечным результатом процесса разработки алгоритмического обеспечения является совокупность алгоритмов, которые лежат в основе последующей программной или программно-аппаратной реализации. Эффективность использования ресурсов компьютера, хотя отчасти и определяется выбором среды реализации, языка и тщательностью программирования, но в основном зависит от разработанных и/или выбранных алгоритмов. Как правило, в качестве основных критериев оценки алгоритма используется временная и емкостная сложность, т. е. оценки требований алгоритма к ресурсам процессора и оперативной памяти.

Необходимость получения оценок или границ для объема памяти или времени выполнения, равно как и прочих ресурсов компьютера, является основной практически значимой причиной теоретического и экспериментального анализа алгоритмов. Результаты такого анализа важны при проектировании программных систем, чтобы исключить превышения ограничений технического задания по памяти или временной эффективности, а также выявить узкие по ресурсоемкости места алгоритмического обеспечения. В рамках проведения такого анализа желательно иметь количественный критерий для сравнения разных алгоритмов решения одой задачи, равно как и механизм выявления более рациональных по ресурсной эффективности алгоритмов.

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 монографиях В. В. Липаева [В.1, В.2]. 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. К таким ресурсам можно отнести ресурсы оперативной памяти, необходимые для хранения исходных данных, промежуточных и окончательных результатов, машинного кода программы, и ресурс стека. 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 действительными числами. 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 диспетчеризации 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 временным затратам на расчеты маршрутов, так и по полученным результатам, т. е. по среднему 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 предъявляются 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 множества исходных данных, определяемыми

областью применения. Ресурс памяти может также зависеть и от особенностей поддерживаемых выбранным языком программирования типов и структур данных и тщательности программной реализации.

Учет ресурсной эффективности на этапах разработки алгоритмического обеспечения. Более детальное представление о месте этапа анализа и исследования ресурсной эффективности компьютерных алгоритмов в разработке математического обеспечения программных средств и систем может быть получено на основе рассмотрения стандартных этапов решения вычислительных задач:

— постановка задачи;

— выбор математической модели задачи;

— детальная разработка структур данных;

— разработка или выбор алгоритма на основе математической модели и выбранных структур данных;

— анализ алгоритма, включая анализ его ресурсной эффективности;

— модификация алгоритма;

— программирование в выбранной среде реализации;

— тестирование и отладка программного обеспечения.

Рассмотрим более подробно основные проблемы, возникающие на этих этапах, которые связаны с разработкой и выбором алгоритма решения задачи:

Выбор математической модели задачи. Выбор той или иной математической модели в значительной степени влияет на последующие алгоритмические решения и, следовательно, является одним из важных этапов общей процедуры решения задачи [В.6].

В качестве примера существенного влияния модели на алгоритмические решения рассмотрим задачу построения разреза объемной детали. При использовании дискретной трехмерной модели деталь представляется трехмерной $(0,1)$ — матрицей принадлежности элементарного дискрета объема прямоугольного параллелепипеда элементарному дискрету данной детали. В этой модели разрез детали плоскостью, параллельной одной из осей, уже содержится в самой модели. Реализация данной модели требует значительного объема оперативной и внешней

памяти, но обеспечивает хорошие временные характеристики для рассматриваемой задачи.

При использовании реберно-граневой модели деталь представляется в виде совокупности точек, ребер и граней, заданных соответствующими массивами. При решении задачи разреза на основе этой модели требуется нахождение точек пересечения секущей плоскости с каждым ребром и построение контура разреза по полученным точкам. Использование данной модели обеспечивает компактное хранение, но требует, по сравнению с дискретной трехмерной моделью, существенно большего количества вычислений для получения результата. Отметим, что такое соотношение между ресурсом оперативной памяти, используемым алгоритмом, и временными оценками характерно для целого ряда задач [В.7]. Достаточно часто для повышения временной эффективности программной реализации приходится выбирать алгоритм, требующий значительных дополнительных объемов оперативной памяти, примером может служить алгоритм сортировки методом индексов.

Детальная разработка структур данных. На основе выбранной модели производится формализация исходных данных, промежуточных и конечных результатов в терминах выбранных структур, которые отражают принятую математическую модель задачи. В ряде случаев особое внимание обращается на внутренние структуры представления промежуточных результатов, например, структура пирамиды в одноименном алгоритме быстрой сортировки. Важность использования эффективных структур данных, в частности абстрактных типов данных и специальных структур хранения, обсуждается в целом ряде работ [В.7, В.8]. Ряд эффективных структур данных, например, красно-черные деревья, позволяют для некоторых задач получить алгоритмы, работающие с оценками трудоемкости, равными или близкими к теоретической нижней границе сложности задачи. Как правило, совместно с эффективными структурами данных разрабатываются и специальные процедуры и функции для выполнения стандартных операций на этих структурах.

Разработка или выбор алгоритма решения задачи с учетом математической модели и структур данных. Этот этап, в общем случае, включает в себя разработку общего алгоритма по крупным функциональным блокам и разработку или вы-

бор частных алгоритмов для выделенных блоков. При этом, как правило, общий алгоритм решения существенно опирается на выбранную математическую модель. Частные алгоритмы для подзадач разрабатываются на принятых внутренних структурах данных. Отметим, что как для общего алгоритма решения, так и для алгоритмов частных подзадач, разработчики сталкиваются с необходимостью наиболее рационального выбора из «веера» существующих алгоритмов. Решение о необходимости разработки новых или модификации существующих алгоритмов принимается на основе результатов проведенного анализа. Дополнительные условия, приводящие к необходимости разработки или модификации алгоритмов при создании математического обеспечения программных систем, могут быть связаны не только с временными характеристиками, но и с требованиями по точности, особенно при решении оптимизационных задач.

Анализ алгоритма. Укажем основные составляющие этого этапа и рассмотрим более подробно их содержание:

Проверка правильности алгоритма. В этой части анализа с применением различных методов верификации [В.9] проверяется правильность работы предложенного общего алгоритма и частных алгоритмов функциональных блоков. При этом в первую очередь требуется тщательная проверка для особых ситуаций с исходными данными. Результаты проверки правильности, как правило, приводят к изменениям алгоритма, связанным с отсечением особых ситуаций с данными и обеспечением устойчивой работы алгоритма для допустимой области исходных значений. Другая цель такой проверки — выявление подмножества множества входов алгоритма, в котором наблюдается неустойчивое поведение результатов (в основном по точности), связанное как с самими математическими методами, так и с накоплением ошибок в ходе циклических вычислений. Заметим, что необходимость повышения точности выполнения арифметических операций заставляет разработчиков использовать специальные методы представления чисел, например, системы счисления в остатках [В.10].

Теоретический и экспериментальный анализ ресурсной эффективности алгоритма. Рассматриваемый алгоритм решения задачи должен быть проверен на

временные и ресурсные ограничения. Такого рода исследования предполагаю полный анализ алгоритма, включающий в себя следующие этапы:

— получение оценки объема требуемой оперативной и внешней памяти, в том числе и памяти под промежуточные результаты, как функции длины входа алгоритма. Отметим, что в теории алгоритмов специально рассматривается класс $PSPACE$, как класс задач с полиномиальной оценкой затрат памяти относительно длины входа [В.11];

— получение оценки сложности алгоритма как асимптотической оценки функции его трудоемкости; методы получения таких оценок достаточно хорошо представлены в литературе;

— теоретическое получение функции трудоемкости алгоритма для лучшего, худшего и среднего случая на множестве исходных данных. Аргументом функции является длина входа, а значением — количество базовых операций в выбранной модели вычислений;

— экспериментальный анализ трудоемкости алгоритма — получение экспериментальных значений счетчика обобщенных базовых операций для различных входов и построение экспериментальной оценки трудоемкости для среднего случая или прогноз трудоемкости, например на основе применения аппарата марковских случайных процессов;

— коррекция теоретической функции трудоемкости на основе данных экспериментального анализа при ситуации с трудностями теоретического обоснования вероятностного поведения алгоритма в среднем случае;

— получение временных характеристик функционирования алгоритма для заданной аппаратной конфигурации компьютера и программной среды реализации;

— определение допустимых границ применимости алгоритма по характеристикам исходных данных в рамках заданных временных ограничений;

— построение совокупной экономической оценки эффективности алгоритма в параметрах «ресурсы памяти — время выполнения» или «ресурсы памяти и процессора — точность».

Модификация алгоритма. Если предложенные алгоритмические решения не удовлетворяют критериям по времени, ресурсам памяти или точности вычислений, то выполняется, если это возможно, модификация алгоритма с последующим возвратом к этапу анализа. Такая модификация может включать в себя как поиск и выбор более производительных алгоритмов для функциональных блоков, так и частичные модификации уже принятых алгоритмов, в особенности «узких мест» по трудоемкости, выявленных на этапе анализа. При невыполнении поставленных в техническом задании ресурсных ограничений могут потребоваться более широкие исследования, связанные с поиском или разработкой алгоритма, более эффективного в данной системе критериев оценки. В любом случае для принятия обоснованного решения необходим аппарат сравнительного анализа ресурсной эффективности компьютерных алгоритмов.

Заключение. В настоящее время известно достаточно большое количество разнообразных по своим характеристикам компьютерных алгоритмов решения актуальных практических задач. В связи с этим, при разработке алгоритмического обеспечения возникает практическая необходимость исследования, оценки и сравнительного анализа ресурсной эффективности различных алгоритмов, предназначенных для решения некоторой задачи в области того множества входов, характеристики которого определяются спецификой области применения. Результаты такого сравнительного анализа позволяют обосновать в принятой системе количественных оценок выбор рациональных ресурсно-эффективных алгоритмов и сформулировать рекомендации по их совершенствованию. Изложению элементов теории ресурсной эффективности и ряда примеров разработки и рационального выбора компьютерных алгоритмов и посвящена эта книга.

Список литературы к введению.

[В.1] Липаев В. В. Методы обеспечения качества крупномасштабных программных средств. — М.: СИНТЕГ, 2003. — 520 с. (Серия «Управление качеством»).

[В.2] Липаев В. В. Обеспечение качества программных средств. — М.: СИНТЕГ. 2001. — 384 с.

[В.3] Оценка качества программных средств. Общие положения. ГОСТ 28195-89.

[В.4] Жигарев А. Ю., Синицин С. В. Оценка характеристик качества программного обеспечения // Современные технологии в задачах управления, автоматики и обработки информации: Сборник трудов XI Международного научно-технического семинара. — М.: МГАПИ, 2002. С. 352–353.

[В.5] Маркова Н. А. Качество программы и его измерения // Системы и средства информатики. Вып. 12. — М.: Наука, 2002. С. 170–188.

[В.6] Советов Б. Я., Яковлев С. А. Моделирование систем: Учеб. для вузов — 3-е изд., перераб. и доп. — М.: Высш. шк., 2001. — 343 с.

[В.7] Ахо А., Хопкрофт Дж., Ульман Дж. Структуры данных и алгоритмы: Пер. с англ.: — М.: Издательский дом «Вильямс», 2001. — 384 с.

[В.8] Вирт Н. Алгоритмы и структуры данных: Пер. с англ. –2-е изд., испр. — СПб.: Невский диалект, 2001. — 352 с.

[В.9] Котов В. Е., Сабельфельд В. К. Теория схем программ. — М.: Наука, 1991. — 231 с.

[В.10] Жуков О. Д. Методы контроля ошибок для компьютерных модулярных вычислений // Информационные технологии. 2003. № 2. С. 33–40.

[В.11] Хопкрофт Дж., Мотовани Р., Ульман Дж. Ведение в теорию автоматов, языков и вычислений, 2-е изд.: Пер. с англ. — М.: Издательский дом «Вильямс», 2002. — 528 с.

Р А З Д Е Л I

А Л Г О Р И Т М Ы И М О Д Е Л И В Ы Ч И С Л Е Н И Й

Цель настоящего раздела — кратко познакомить читателя с основными понятиями и положениями теории алгоритмов — науки, без знания фундаментальных результатов которой невозможно говорить о ресурсно-эффективных компьютерных алгоритмах. Автор считает также необходимым кратко изложить материал о моделях вычислений, поскольку именно на их основе вводятся и исследуются ресурсные оценки алгоритмов, позволяющие сравнивать алгоритмы решения одной и той же задачи между собой и, в конце концов, выбирать наиболее рациональные при данных условиях применения.

Г Л А В А 1 .

Э Л Е М Е Н Т Ы Т Е О Р И И А Л Г О Р И Т М О В

Введение

Задача точного определения понятия алгоритма, разработка формальных моделей описания алгоритмов, формулировка их основных свойств и требований, составляют предмет исследований в теории алгоритмов. На основе формализации понятия алгоритма возможно сравнение алгоритмов по их эффективности, проверка их эквивалентности, определение областей применимости.

Разработанные в 1930-х годах формальные модели вычислений (Пост, Тьюринг, Черч) [1.1, 1.2], равно как и предложенные в 1950-х годах модели и описания Колмогорова и Маркова, оказались эквивалентными в том смысле, что любой класс проблем, разрешимых в одной модели, разрешим и в другой. Все эти определения алгоритма являются равнозначными и определяют один и тот же класс алгоритмически вычислимых функций, что отражают известные тезисы Поста, Черча и Тьюринга [1.3]. Важным этапом в развитии теории алгоритмов стали формулировка и доказательство алгоритмически неразрешимых проблем. В 1950-

е годы существенный вклад в теорию алгоритмов внесли работы Колмогорова и Маркова. К 1960 – 70-м годам оформились следующие направления в теории алгоритмов:

— классическая теория алгоритмов — создание и исследование различных формальных моделей вычислений, исследование алгоритмов в формализме машины Тьюринга, доказательство алгоритмической неразрешимости и т. д.

— теория сложности вычислений — формулировка задач в терминах формальных языков, понятие задачи разрешения, введение сложностных классов, формулировка проблемы $P = NP$, открытие класса NP-полных задач и его исследование;

— теория асимптотического анализа алгоритмов — исследование сложности и трудоёмкости алгоритмов в асимптотических оценках, формулировка критериев оценки, методы получения асимптотических оценок, в частности для класса рекурсивных алгоритмов, асимптотический анализ времени выполнения;

— практический анализ вычислительных алгоритмов — получение практически значимых оценок, разработка критериев качества алгоритмов и методов выбора рациональных алгоритмов; (основополагающей работой в этом направлении, очевидно, следует считать фундаментальный труд Д. Кнута «Искусство программирования для ЭВМ» [1.4]).

В настоящее время теория алгоритмов образует теоретический фундамент вычислительных наук, а полученные на ее основе рекомендации получают всё большее распространение в области программирования. Цель настоящей главы — на уровне введения познакомить читателей с основными положениями теории алгоритмов, включая алгоритмически неразрешимые проблемы и некоторыми классами задач, рассматриваемых в теории сложности вычислений.

1.1 Понятие и определения алгоритма

Происхождение термина «алгоритм». Термин «алгоритм» обязан своим происхождением великому ученому средневекового Востока — Муххамаду ибн Муса ал-Хорезми (Магомет, сын Моисея, из Хорезма. ~783-850). В латинских переводах арифметического трактата ал-Хорезми его имя транскрибировалось как algorismi. Именно эта транскрипция явилась основой термина «алгоритм» — сна-

чала для обозначения цифровых вычислений в арифметике, а затем для обозначения произвольных процессов, в которых искомые величины решаемых задач находятся последовательно на основе исходных данных по определенным правилам и инструкциям.

Интуитивное понятие алгоритма. Во всех сферах своей деятельности, и в частности в сфере обработки информации, человек сталкивается с различными способами или методами решения разнообразных задач. Они определяют порядок выполнения некоторых действий для получения желаемого результата — мы можем трактовать это как первоначальное или интуитивное определение алгоритма. Таким образом, можно нестрого определить алгоритм как однозначно трактуемую процедуру решения задачи, т. е. как задание порядка выполнения действий для получения желаемого результата. Дополнительное требование о выполнении всей последовательности действий, задаваемых алгоритмом, за конечное время для всех допустимых входных данных приводят к следующему неформальному (интуитивному) определению алгоритма:

Алгоритм — это заданное на некотором языке конечное предписание, задающее конечную последовательность выполнимых и точно определенных элементарных операций для решения задачи, общее для класса возможных исходных данных.

Несмотря на усилия ученых, сегодня отсутствует одно исчерпывающе строгое словесное определение понятия «алгоритм». Из разнообразных вариантов таких словесных определений наиболее удачные, по мнению автора, принадлежат российским ученым А.Н. Колмогорову [1.5] и А.А. Маркову [1.6]:

Определение 1.1 (А.Н. Колмогоров)

Алгоритм — это всякая система вычислений, выполняемых по строго определенным правилам, которая после какого-либо числа шагов заведомо приводит к решению поставленной задачи.

Определение 1.2 (А.А. Марков)

Алгоритм — это точное предписание, определяющее вычислительный процесс, идущий от варьируемых исходных данных к искомому результату.

К началу XX века такие разделы математики, как алгебра и теория чисел, дали много ярких примеров алгоритмов, соответствующих приведенным выше определениям. Среди них укажем алгоритм Евклида нахождения наибольшего общего делителя двух натуральных чисел или двух целочисленных многочленов, алгоритм Гаусса решения системы линейных уравнений над полем, алгоритм нахождения рациональных корней многочленов одного переменного с рациональными коэффициентами, алгоритм Штурма для определения числа действительных корней многочлена с действительными коэффициентами на некотором сегменте, алгоритм разложения многочлена одного переменного над конечным полем на неприводимые множители.

Однако в начале XX века был сформулирован ряд проблем, возможность положительного решения которых некоторым алгоритмом представлялась маловероятной. Решение таких проблем потребовало привлечения новых мощных логических средств. Ведь одно дело доказать существование разрешающего алгоритма — это можно сделать, используя интуитивное понятие алгоритма. Другая, значительно более сложная задача — доказать отсутствие алгоритма, решающего данную проблему. Для решения этой задачи необходим математически строгий подход к определению понятия алгоритма.

Математически строгие подходы к определению алгоритма. Задача математически строгого определения понятия алгоритма была решена в 30-х годах XX века в работах Чёрча, Клини, Поста, Тьюринга в двух формах: на основе понятия рекурсивной функции и на основе описания алгоритмического процесса. Рекурсивная функция — это функция, для которой существует алгоритм вычисления ее значений для произвольного значения аргумента на основе известных предыдущих значений. Класс рекурсивных функций был определен строго как конкретный класс функций в некоторой формальной системе. Был сформулирован тезис, который называется «тезис Чёрча», утверждающий, что данный класс функций совпадает с множеством функций, для которых имеется алгоритм вычисления их значений по значению аргументов. Другой подход заключался в том, что алгоритм определяется как дискретный процесс, осуществимый на абстрактной машине, называемой «машиной Тьюринга». Был сформулирован тезис — «те-

зис Тьюринга», утверждающий, что любой алгоритм может быть реализован на машине Тьюринга. Оба данных подхода, а также другие подходы (Маркова, Поста) привели к определению одного и того же класса алгоритмически вычислимых функций и подтвердили целесообразность использования тезиса Чёрча или тезиса Тьюринга для решения алгоритмических проблем. Поскольку понятие рекурсивной функции является математически строгим, то можно доказать, что решающая некоторую задачу функция не является рекурсивной, что эквивалентно отсутствию для данной задачи разрешающего ее алгоритма. Аналогично, отсутствие разрешающей машины Тьюринга для некоторой задачи равносильно отсутствию для нее разрешающего алгоритма. Указанные результаты составляют основу так называемой дескриптивной теории алгоритмов, основным содержанием которой является классификация задач по признаку алгоритмической разрешимости, то есть получение высказываний типа «Задача Z алгоритмически разрешима» или «Задача Z алгоритмически неразрешима».

Формализация понятия алгоритма. Неудобства словесных определений связаны с проблемой однозначной трактовки терминов. В таких определениях должен быть, хотя бы неявно, указан исполнитель действий или предписаний. Алгоритм вычисления производной для полинома фиксированной степени вполне ясен тем, кто знаком с основами математического анализа, но для прочих он может оказаться совершенно непонятным. Это рассуждение заставляет нас указать так же вычислительные возможности исполнителя, а именно уточнить какие операции для него являются «элементарными». Другие трудности связаны с тем, что если даже алгоритм заведомо существует, то его очень трудно описать в некоторой заранее заданной форме. Классический пример такой ситуации — алгоритм завязывания шнурков на ботинках «в бантик». Вы сможете дать только словесное описание этого алгоритма без использования иллюстраций?

В связи с этим формально строгие определения понятия алгоритма связаны с введением специальных математических конструкций — формальных алгоритмических систем или моделей вычислений, каковыми являются машина Поста, машина Тьюринга, рекурсивно-вычислимые функции Черча, и постулированием тезиса об эквивалентности такого формализма и понятия «алгоритм». Несмотря

на принципиально разные строгие подходы, используемые в теории алгоритмов, интересным результатом является формулировка гипотез об эквивалентности этих формальных определений в смысле их равномощности. Более подробную информацию о классической теории алгоритмов читатель найдет, например. в [1.3], в рамках данной книги автор ограничивается элементарным изложением машин Поста и Тьюринга и введением в проблематику теории алгоритмов.

Машина Поста. Одной из фундаментальных статей, лежащей в основе современной теории алгоритмов, является статья Эмиля Поста (Emil Post), «Финитные комбинаторные процессы, формулировка 1», опубликованная в 1936 году в сентябрьском номере «Журнала символической логики» [1.1]. Пост рассматривает общую проблему, состоящую из множества конкретных проблем, при этом решением общей проблемы является такое решение, которое доставляет ответ для каждой конкретной проблемы. Например, решение уравнения $3x+9=0$ — это одна из конкретных проблем, а решение уравнения $ax+b=0$ — это общая проблема, тем самым алгоритм должен быть универсальным, т. е. должен быть соотнесен с общей проблемой. Отметим, что сам термин «алгоритм» не используется Э. Постом, его заменяет в статье понятие набора инструкций.

Основные понятия алгоритмического формализма Поста — это пространство символов, в котором задаётся конкретная проблема и получается ответ, и набор инструкций, т. е. операций в пространстве символов, задающих как сами операции, так и порядок их выполнения. Постовское пространство символов представляет собой бесконечную ленту ящиков (ячеек), каждый из которых может быть или помечен или не помечен, что схематично показано на рисунке 1.1.

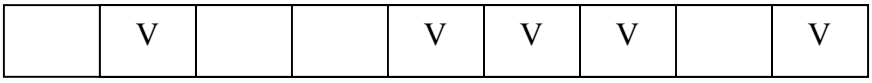


Рис. 1.1. Лента ячеек в машине Поста

Конкретная проблема задается «внешней силой» (термин Поста) пометкой конечного числа ящиков, при этом, очевидно, что любая конфигурация начинается и заканчивается помеченным ящиком. После применения к конкретной про-

блеме некоторого набора инструкций решение представляется также в виде набора помеченных и непомеченных ящиков, распознаваемое той же внешней силой. Пост предложил набор инструкций или элементарно выполнимых операций, которые выполняет «работник», отметим в этой связи, что в 1936 году не было еще ни одной работающей электронной вычислительной машины. Сегодня мы бы сказали, что этот набор инструкций является минимальным набором битовых операций элементарного процессора:

1. Пометить ящик, если он пуст;

2. Стереть метку, если она есть;

3. Переместиться влево на 1 ящик;

4. Переместиться вправо на 1 ящик;

5. Определить помечен ящик или нет, и по результату перейти на одну из двух указанных инструкций;

6. Остановиться.

Отметим, что формулировка первых двух инструкций включает в себя защиту от неправильных ситуаций с данными. Программа представляет собой нумерованную последовательность инструкций, причем переходы производятся на указанные в ней номера других инструкций. Программа, или набор инструкций в терминах Э. Поста, является одной и той же для всех конкретных проблем, и поэтому соотнесена с общей проблемой — таким образом, Пост формулирует требование универсальности алгоритма.

Далее Э. Пост вводит следующие понятия:

— набор инструкций *применим* к общей проблеме, если для каждой конкретной проблемы не возникает коллизий в инструкциях 1 и 2, т. е. никогда программа не стирает метку в пустом ящике и не устанавливает метку в помеченном ящике;

— набор инструкций *заканчивается*, если после конечного количества инструкций выполняется инструкция 6;

— набор инструкций задаёт *финитный* 1-*процесс*, если набор применим и заканчивается для каждой конкретной проблемы;

— финитный 1-процесс для общей проблемы есть 1-*решение*, если ответ для каждой конкретной проблемы правильный, что определяется внешней силой, задающей конкретную проблему.

По Э. Посту проблема задаётся внешней силой путем пометки конечного количества ящиков ленты. В более поздних работах по машине Поста [1.7] принято считать, что машина работает в единичной системе счисления (0=V; 1=VV; 2=VVV), т. е. ноль представляется одним помеченным ящиком, а целое положительное число — помеченными ящиками в количестве на единицу больше его значения. Поскольку множество конкретных проблем, составляющих общую проблему, является счетным, то можно установить взаимно однозначное соответствие (биективное отображение) между множеством положительных целых чисел N и множеством конкретных проблем. Общая проблема называется по Посту 1-заданой, если существует такой финитный 1-процесс, что, будучи применим к $n \in N$ в качестве исходной конфигурации ящиков, он задает n-ую конкретную проблему в виде набора помеченных ящиков в пространстве символов.

Если общая проблема 1-задана и 1-разрешима, то, соединяя наборы инструкций по заданию проблемы и ее решению, мы получаем ответ по номеру проблемы — это и есть *формулировка* 1 в терминах статьи Э. Поста.

Э. Пост завершает свою статью следующей фразой [1.7]: «Автор ожидает, что его формулировка окажется логически эквивалентной рекурсивности в смысле Гёделя — Черча. Цель формулировки, однако, в том, чтобы предложить систему не только определенной логической силы, но и психологической достоверности. В этом последнем смысле подлежат рассмотрению всё более и более широкие формулировки. С другой стороны, нашей целью будет показать, что все они логически сводимы к формулировке 1. В настоящий момент мы выдвигаем это умозаключение в качестве *рабочей гипотезы*. … Успех вышеизложенной программы заключался бы для нас в превращении этой гипотезы не столько в определение или аксиому, сколько в закон природы».

Таким образом, гипотеза Поста состоит в том, что любые более широкие формулировки в смысле алфавита символов ленты, набора инструкций, представ-

ления и интерпретации конкретных проблем сводимы к формулировке 1 (см. рис 1.2).

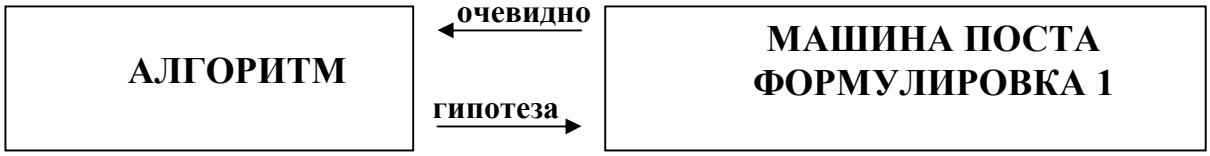


Рис. 1.2 Графическая интерпретация гипотезы Поста.

Следовательно, если гипотеза верна, то любые другие формальные определения, задающие некоторый класс алгоритмов, эквивалентны классу алгоритмов, заданных формулировкой 1, «обоснование этой гипотезы происходит сегодня не путем строго математического доказательства, а на пути эксперимента — действительно, всякий раз, когда нам указывают алгоритм, его можно перевести в форму программы машины Поста, приводящей к тому же результату» [1.7].

Машина Тьюринга. Алан Тьюринг (Alan Turing) в 1936 году опубликовал в трудах Лондонского математического общества статью «О вычислимых числах в приложении к проблеме разрешения», которая наравне с работами Поста и Черча лежит в основе современной теории алгоритмов [1.2]. Предыстория создания этой работы связана с формулировкой Давидом Гильбертом на Международном математическом конгрессе в Париже в 1900 году нерешенных математических проблем. Одной из них была задача доказательства непротиворечивости системы аксиом обычной арифметики, которую Гильберт в дальнейшем уточнил как «проблему разрешимости» — нахождение общего метода, для определения выполнимости данного высказывания на языке формальной логики. Статья Тьюринга как раз и давала ответ на эту проблему — вторая проблема Гильберта оказалась неразрешимой. Но значение статьи Тьюринга выходило далеко за рамки той задачи, по поводу которой она была написана.

Приведем характеристику этой работы, принадлежащую Джону Хопкрофту [1.8]: «Работая над проблемой Гильберта, Тьюрингу пришлось дать четкое определение самого понятия метода. Отталкиваясь от интуитивного представления о методе как о некоем алгоритме, т. е. процедуре, которая может быть выполнена механически, без творческого вмешательства, он показал, как эту идею можно во-

плотить в виде подробной модели вычислительного процесса. Полученная модель вычислений, в которой каждый алгоритм разбивался на последовательность простых, элементарных шагов, и была логической конструкцией, названной впоследствии машиной Тьюринга».

Машина Тьюринга является расширением модели конечного автомата, расширением, включающим потенциально бесконечную память с возможностью перехода (движения) от обозреваемой в данный момент ячейки к ее левому или правому соседу [1.9]. Формально машина Тьюринга может быть описана следующим образом — пусть заданы:

— конечное множество состояний Q, в которых может находиться машина Тьюринга;

— конечное множество символов ленты Γ;

— функция δ (функция переходов или программа), которая задается отображением пары из декартова произведения $Q \times \Gamma$ (машина находится в состоянии q_i и обозревает символ γ_i) в тройку декартова произведения $Q \times \Gamma \times \{L, R\}$ (машина переходит в состояние q_j, заменяет символ γ_i на символ γ_j и передвигается влево или вправо на один символ ленты):

$$Q \times \Gamma \to Q \times \Gamma \times \{L, R\};$$

— существует выделенный пустой символ $e \in \Gamma$;

— подмножество Σ — входной алфавит, $\Sigma \subset \Gamma$, определяется как подмножество входных символов ленты, причем символ $e \in \Gamma - \Sigma$;

— одно из состояний $q_0 \in Q$ является начальным состоянием машины.

Решаемая проблема задается путем записи на ленту конечного количества символов s_i, образующих слово ω в алфавите Σ, что схематично показано на рисунке 1.3. После задания проблемы машина переводится в начальное состояние, и головка устанавливается у самого левого непустого символа, после чего в соответствии с указанной функцией переходов машина начинает заменять обозреваемые символы, передвигать головку вправо или влево и переходить в другие состояния. Останов машины происходит в том случае, если для пары (q_i, s_i) функция перехода не определена.

e	s_1	s_2	s_3	s_4	.....	s_n	e

Рис 1.3 Задание решаемой проблемы на машине Тьюринга

А. Тьюринг высказал предположение, что любой алгоритм в интуитивном смысле этого слова может быть представлен эквивалентной машиной в предложенной им модели вычислений. Это предположение известно как тезис Черча-Тьюринга. Каждый компьютер может моделировать машину Тьюринга, для этого достаточно моделировать операции перезаписи ячеек, сравнения и перехода к другой соседней ячейке с учетом изменения состояния машины. Таким образом, компьютер может моделировать алгоритмы в машине Тьюринга, и из этого тезиса следует, что все компьютеры, независимо от мощности, архитектуры и других особенностей, эквивалентны с точки зрения принципиальной возможности решения алгоритмически разрешимых задач.

Алгоритмически неразрешимые проблемы. За время своего существования человечество придумало множество алгоритмов для решения разнообразных практических и научных проблем. Зададимся вопросом — а существуют ли какие-нибудь проблемы, для которых невозможно придумать алгоритмы их решения? Утверждение о существовании алгоритмически неразрешимых проблем является весьма сильным — мы констатируем, что мы не только сейчас не знаем соответствующего алгоритма, но мы не можем принципиально никогда его найти.

Успехи математики к концу XIX века привели к формированию мнения, которое выразил Д. Гильберт — «в математике не может быть неразрешимых проблем», в связи с этим формулировка Гильбертом нерешенных проблем на конгрессе в Париже в 1900 году была руководством к действию, констатацией отсутствия решений лишь на данный момент. Первой фундаментальной теоретической работой, связанной с доказательством алгоритмической неразрешимости, была работа К. Гёделя — его теорема о неполноте символических логик. Это строго формулированная математическая проблема, для которой не существует решающего ее алгоритма. Усилиями различных исследователей список алгоритмически неразрешимых проблем был значительно расширен. Сегодня при доказательстве

алгоритмической неразрешимости некоторой проблемы принято сводить ее к ставшей классической задаче — «проблеме останова» машины Тьюринга.

В данном направлении получен ряд фундаментальных результатов. Среди них отметим отрицательное решение П. С. Новиковым в 1952 году классической проблемы тождества для конечно определенных групп, сформулированной Деном еще в 1912 году, и доказательство в 1970 году Ю. В. Мятиясевичем алгоритмической неразрешимости 10-й проблемы Гильберта. Десятая проблема Гильберта, формулируется следующим образом: «10. Задача о разрешимости диофантова уравнения. Пусть задано диофантово уравнение с произвольными неизвестными и целыми рациональными числовыми коэффициентами. Указать способ, при помощи которого возможно после конечного числа операций установить, разрешимо ли это уравнение в целых рациональных числах».

В настоящее время основой доказательства алгоритмической неразрешимости является следующая теорема о алгоритмической неразрешимости проблемы останова, доказательство которой читатель может найти, например, в [1.9].

Теорема. Не существует алгоритма (машины Тьюринга), позволяющего по описанию произвольного алгоритма и его исходных данных, при этом и алгоритм и данные заданы символами на ленте машины Тьюринга, определить, останавливается ли этот алгоритм на этих данных или работает бесконечно.

Таким образом, фундаментально, алгоритмическая неразрешимость связана с бесконечностью задаваемых алгоритмом действий, иными словами с невозможностью предсказать, что для любых исходных данных решение будет получено за конечное количество шагов.

Приведем несколько примеров алгоритмически неразрешимых проблем.

Пример 1.1. Распределение девяток в записи числа π [1.7].

Определим функцию $f(n)=i$, где n — количество девяток подряд в десятичной записи числа π, а i — номер самой левой после запятой девятки из n девяток подряд. Поскольку $\pi \approx 3{,}141592\ldots$, то $f(1)=5$. Задача состоит в вычислении функции $f(n)$ для произвольно заданного значения n.

Поскольку число π является иррациональным и трансцендентным, то мы не знаем никакой информации о распределении девяток, равно как и любых других цифр, в десятичной записи числа π. Вычисление $f(n)$ связано с вычислением последующих цифр в разложении числа π, до тех пор, пока мы не обнаружим n девяток подряд. Однако у нас нет общего метода вычисления $f(n)$, поэтому для некоторых n вычисления могут продолжаться бесконечно — мы даже не знаем в принципе (по природе числа π) существует ли решение для всех значений n.

Пример 1.2. Вычисление совершенных чисел [1.10].

Совершенные числа – это числа, которые равны сумме своих делителей, например: $28 = 1 + 2 + 4 + 7 + 14$. Определим функцию $S(n)$, задающую n-ое по счёту совершенное число и поставим задачу вычисления $S(n)$ по произвольно заданному значению n. Нет общего метода вычисления совершенных чисел, мы даже не знаем о том, является ли множество совершенных чисел конечным или счетным, поэтому наш алгоритм должен перебирать все числа подряд, проверяя их на совершенность. Отсутствие общего метода решения не позволяет ответить на вопрос об останове алгоритма. Если мы проверили m чисел при поиске n-ого совершенного числа — означает ли это, что его вообще не существует?

Пример 1.3. Поиск последнего помеченного ящика в машине Поста [1.7].

Пусть на ленте машины Поста заданы наборы помеченных ящиков (кортежи) произвольной длины с произвольными расстояниями между кортежами и головка находится у самого левого помеченного ящика. Задача состоит в установке головки на самый правый помеченный ящик последнего кортежа. Попытка построения алгоритма, решающего эту задачу приводит к необходимости ответа на вопрос — когда после обнаружения конца кортежа мы сдвинулись вправо по пустым ящикам на m позиций и не обнаружили начало следующего кортежа — больше на ленте кортежей нет или они есть где-то правее? Информационная неопределенность задачи связана с отсутствием информации либо о количестве кортежей на ленте, либо о максимальном расстоянии между кортежами — при наличии такой информации, т. е. при разрешении информационной неопределенности задача становится алгоритмически разрешимой.

Пример 1.4. Проблема эквивалентности алгоритмов [1.9].

По двум произвольным заданным алгоритмам, например, по двум машинам Тьюринга, определить, будут ли они выдавать одинаковые выходные результаты при любых входных данных.

Пример 1.5. Проблема тотальности [1.9].

По записи произвольно заданного алгоритма определить, будет ли он останавливаться на всех возможных наборах исходных данных. Другая формулировка этой задачи выглядит следующим образом: является ли частичный алгоритм всюду определённым алгоритмом?

Пример 1.6. Проблема соответствий Поста над алфавитом Σ [1.10].

В качестве более подробного примера алгоритмически неразрешимой задачи рассмотрим проблему соответствий, формулировка которой принадлежит Посту [5]. Мы выделили эту задачу, поскольку на первый взгляд она выглядит достаточно «алгоритмизиуемой», но, тем не менее, она сводима к проблеме останова и является алгоритмически неразрешимой.

Постановка задачи:

Пусть дан алфавит $\Sigma : |\Sigma| \geq 2$ (для алфавита, состоящего из одного символа, задача имеет решение) и дано конечное множество пар из $\Sigma^+ \times \Sigma^+$, т. е. пары непустых цепочек произвольного языка над алфавитом $\Sigma : (x_1, y_1), \ldots, (x_m, y_m)$. Проблема соответствий Поста состоит в том, что необходимо выяснить существует ли конечная последовательность этих пар, не обязательно различных, такая что цепочка, составленная из левых подцепочек, совпадает с последовательностью правых подцепочек — такая последовательность называется решающей.

В качестве примера рассмотрим алфавит $\Sigma = \{a, b\}$, и две последовательности входных цепочек.

1. Входные цепочки: $(abbb, b), (a, aab), (ba, b)$.

Решающая последовательность для этой задачи имеет вид:

$(a, aab), (a, aab), (ba, b), (abbb, b)$, так как: $a\ a\ ba\ abbb = aab\ aab\ b\ b$.

2. Входные цепочки: $(ab, aba), (aba, baa), (baa, aa)$.

Данная задача вообще не имеет решения, так как нельзя начинать с пары (aba, baa) или (baa, aa), поскольку не совпадают начальные символы подцепочек,

но если начинать с цепочки (ab, aba), то в последующем не будет совпадать общее количество символов «a», т.к. в других двух парах количество символов «a» одинаково.

В общем случае мы можем построить частичный алгоритм, основанный на идее упорядоченной генерации возможных последовательностей цепочек, отметим, что мы имеем счетное множество таких последовательностей, с проверкой выполнения условий задачи. Если последовательность является решающей, то мы получаем результативный ответ за конечное количество шагов. Поскольку общий метод определения отсутствия решающей последовательности не может быть указан, т. к. задача сводима к проблеме «останова» и, следовательно, является алгоритмически неразрешимой, то при отсутствии решающей последовательности алгоритм порождает бесконечный цикл генерации и проверок.

В теории алгоритмов такого рода проблемы, для которых может быть предложен частичный алгоритм их решения, частичный в том смысле, что он возможно, но не обязательно, за конечное количество шагов находит решение проблемы, называются *частично разрешимыми проблемами.* Например, проблема останова является частично разрешимой проблемой, а проблемы эквивалентности и тотальности не являются таковыми.

Наряду с понятием частично разрешимой проблемы в теории алгоритмов вводятся понятия частичного и полного алгоритмов. Пусть D_Z — область (множество) исходных данных задачи Z, а R — множество возможных результатов, тогда мы можем говорить, что алгоритм осуществляет отображение $D_Z \to R$. Поскольку такое отображение может быть не полным, то алгоритм называется частичным алгоритмом, если мы получаем результат только для некоторых исходных данных $d \in D_Z$ и полным алгоритмом, если алгоритм получает правильный результат для всех $d \in D_Z$.

Применение результатов теории алгоритмов. В технику термин «алгоритм» пришел вместе с кибернетикой. Применение ЭВМ послужило стимулом развития теории алгоритмов и изучения алгоритмических моделей, изучения самих алгоритмов с целью их сравнения по рабочим характеристикам. Возникло важное направление в теории алгоритмов — сложность алгоритмов и вычисле-

ний. Начала складываться так называемая метрическая теория алгоритмов, основным содержанием которой является классификация задач по сложности. Сами алгоритмы стали объектом точного исследования, как и те объекты, для работы с которыми они предназначены. На основе результатов теории алгоритмов были разработаны быстрые алгоритмы умножения целых чисел, многочленов, матриц, решения линейных систем уравнений, которые требуют значительно меньше ресурсов, чем традиционные математические алгоритмы.

Полученные в теории алгоритмов результаты находят сегодня достаточно широкое практическое применение, в рамках которого можно выделить два следующих аспекта:

Теоретический аспект: при исследовании некоторой задачи результаты теории алгоритмов позволяют ответить на вопрос — является ли эта задача в принципе алгоритмически разрешимой. Для алгоритмически неразрешимых задач возможно их сведение к задаче останова машины Тьюринга. В случае алгоритмической разрешимости задачи следующим важным теоретическим вопросом является вопрос о принадлежности этой задачи к классу NP-полных задач. При утвердительном ответе можно говорить о существенных временных затратах для получения точного решения этой задачи для больших размерностей исходных данных, иными словами — об отсутствии быстрого точного алгоритма ее решения.

Практический аспект: методы и методики теории алгоритмов, в основном разделов асимптотического и практического анализа, позволяют осуществить:

— получение временных оценок решения сложных задач;

— получение достоверных оценок невозможности решения некоторой задачи за определенное время, такого рода обратные по временной эффективности задачи оказываются сегодня также востребованы, например, для криптографических методов;

— разработку и совершенствование эффективных алгоритмов решения практически значимых задач в области обработки информации.

Обобщая исследования в различных разделах теории алгоритмов можно выделить следующие основные направления развития, характерные для современной теории алгоритмов:

— формализация понятия «алгоритм» и исследование формальных алгоритмических систем (моделей вычислений);

— доказательство алгоритмической неразрешимости задач;

— формальное доказательство правильности и эквивалентности алгоритмов;

— классификации задач, определение и исследование сложностных классов;

— доказательство теоретических нижних оценок сложности задач;

— получение методов разработки эффективных алгоритмов;

— асимптотический анализ сложности итерационных алгоритмов;

— исследование и анализ рекурсивных алгоритмов;

— получение явных функций трудоемкости алгоритмов;

— разработка классификаций алгоритмов;

— исследование емкостной сложности задач и алгоритмов;

— разработка критериев оценки алгоритмов и методов их сравнительного анализа.

1.2 Требования к алгоритмам и их свойства

Отметим, что различные определения алгоритма, в явной или неявной форме, постулируют следующий ряд общих требований к алгоритмам и их свойств.

Основные требования к алгоритмам. Приведенные в параграфе 1.1. определения позволяют выделить следующие основные требования к алгоритмам [1.11]:

— правильность, алгоритм должен получать правильное решение по отношению к поставленной задаче. Алгоритм считается правильным, если при любых допустимых входных данных он заканчивает работу и выдает результат, удовлетворяющий требованиям задачи. Особо отметим, что доказательство правильности алгоритмов составляет предмет специального раздела в теории алгоритмов;

— универсальность или массовость, алгоритм должен быть единым для всех допустимых исходных данных, т. е. должен описывать некоторый универсальный процесс, применимый для различных входных данных из некоторого множества. Совокупность этих входных данных соотнесена с общей проблемой, решаемой алгоритмом;

— конечность записи, запись алгоритма должна содержать конечное количество элементарно выполнимых операций, называемых также предписаниями или действиями;

— дискретность, процесс решения задачи задается алгоритмом в виде последовательности отдельных операций, следующих друг за другом;

— конечность количества действий, алгоритм должен выполнять конечное количество операций при решении любой допустимой задачи;

— элементарность операций, каждое действие является или предполагается настолько простым, что оно не допускает возможности неоднозначного толкования;

— определенность, каждая операция однозначно определена и после выполнения каждого действия однозначно определяется, какое действие будет выполнено следующим;

— результативность, в момент останова алгоритма известно, что является результатом и как он, будучи записан словами алфавита в памяти, интерпретируется в проблемную область задачи;

— однозначность, алгоритм считается однозначным, если при применении его к одним и тем же входным данным он дает один и тот же выходной результат.

Основные свойства алгоритмов. Укажем следующие основные свойства алгоритма [1.11], уточняющие его определения:

1. Наличие данных, обрабатываемых алгоритмом. Каждый алгоритм работает с данными — это входные данные, промежуточные данные и результаты алгоритма. Для уточнения понятия данных фиксируется некоторый конечный алфавит символов, и указываются правила построения объектов множества данных алгоритма — слов данного алфавита. Формальным аппаратом описания таких правил являются формальные грамматики, порождающие индуктивное построение объектов. Наиболее типичными алгоритмическими объектами являются слова конечной длины в конечных алфавитах. Естественной мерой таких объектов является объем — количество символов в слове (длина). При реализации на компьютере такими объектами являются целые и действительные числа, логические и сим-

вольные переменные. На их основе могут быть определены более сложные структурированные объекты: массивы, строки, структуры.

2. Наличие памяти для размещения данных. Алгоритм использует память для размещения промежуточных и выходных данных. Вход алгоритма формируется в этой же памяти внешней силой (термин Э. Поста [1.7]). Обычно такая память в формальных системах считается однородной и дискретной — память состоит из однородных ячеек, возможно, адресуемых, причем каждая ячейка может содержать или символ алфавита, или слово ограниченной длины. В ЭВМ память состоит из адресуемых ячеек (байтов), причем единицы объема данных и памяти согласованы так, что в прикладных алгоритмических моделях объем данных можно измерять числом ячеек, в которых эти данные размещены.

3. Априорность операций. Алгоритм есть последовательность элементарных шагов или операций, которые известны или заданы до формулировки алгоритма (априорно). Исходное множество различных операций в любой формальной системе (модели вычислений) — конечно. Реально под элементарными шагами можно подразумевать машинные команды, входящие в набор команд компьютера. При записи алгоритмов на языках высокого уровня в качестве базовых операций могут выступать основные операторы языка.

4. Детерминированность шагов. Выбранная форма записи однозначно должна определять последовательность шагов алгоритма. Это означает, что после каждого шага указывается, какой шаг выполняется далее, либо фиксируется завершение работы алгоритма.

5. Сходимость алгоритма. Это свойство алгоритма результативно останавливаться после конечного количества шагов, зависящего от входных данных в памяти, с выдачей результата, правильного для определенного множества входных данных.

6. Существование механизма реализации. Алгоритм предполагает механизм реализации, который порождает пошаговый процесс вычислений для исходных данных в памяти. Это порождение происходит в механизме реализации на основе предъявленного описания алгоритма.

Имеются также некоторые свойства неформального понятия алгоритма, относительно которых не достигнуто окончательного соглашения. Эти свойства могут быть сформулированы в виде следующих вопросов.

7. Следует ли фиксировать конечную границу для размера входных данных?

8. Следует ли фиксировать конечную границу для числа заданных алгоритмом элементарных шагов?

9. Следует ли фиксировать конечную границу для размера памяти?

На все эти вопросы далее принимается ответ «НЕТ», поскольку у физически существующих ЭВМ соответствующие размеры ресурсов ограничены, хотя возможны и другие варианты ответов. Однако теория, изучающая алгоритмические вычисления, осуществимые в принципе, не должна считаться с такого рода ограничениями, поскольку они преодолимы, например, вообще говоря, любой фиксированный размер памяти всегда можно увеличить на одну ячейку.

Таким образом, уточнение понятия алгоритма связано с уточнением алфавита данных и формы их представления, памяти и размещения в ней данных, элементарных шагов алгоритма и механизма реализации алгоритма. Однако эти понятия сами нуждаются в уточнении. Ясно, что их словесные определения потребуют введения новых понятий, для которых в свою очередь, снова потребуются уточнения и т. д. Поэтому в теории алгоритмов принят другой подход, основанный на конкретной алгоритмической модели, в которой все сформулированные требования выполняются очевидным образом. При этом используемые алгоритмические модели универсальны, то есть моделируют любые другие разумные алгоритмические модели, что позволяет снять возможное возражение против такого подхода: не приводит ли жесткая фиксация алгоритмической модели к потере общности формализации алгоритма? В связи с этим все рассматриваемые в теории алгоритмов модели (модели вычислений) отождествляются с формальным понятием алгоритма.

1.3 Временная и ёмкостная сложность алгоритма

Дальнейшее исследование алгоритмов связано с оценками их ресурсоемкости или ресурсной эффективности. Выполнение требования конечности количества действий еще не говорит о том, насколько долго мы будем ожидать результата. В этом смысле было бы целесообразно попытаться получить зависимость ресурс-

ных оценок алгоритма, как функцию некоторого аргумента. Поскольку интуитивно различные конкретные проблемы, скорее всего, будут решаться данным алгоритмом за большее время при большем количестве элементов на входе, то в качестве такого аргумента можно выбрать длину входа алгоритма. В классической теории под длиной входа понимается число символов ленты машины Тьюринга. Однако в рамках практического анализа алгоритмов под длиной входа обычно понимается или количество объектов, обрабатываемых алгоритмом (для большинства случаев — это количество чисел или символов) или некоторая мера количества этих объектов.

Наиболее употребительными оценками ресурсной эффективности алгоритмов являются:

— оценка требуемого ресурса процессора — временная сложность;

— оценка общего объема требуемой памяти — емкостная сложность.

Такие оценки обычно вводятся на основании следующих рассуждений [1.12]. Пусть алгоритм A предназначен для решения общей задачи $\widetilde{Z}$, состоящей из ряда конкретных задач Z. При исследовании ресурсной эффективности алгоритм A рассматривается как способ задания последовательности операций, преобразующих исходные данные конкретной задачи Z в требуемый результат. Используемая в алгоритме память для размещения данных при этом проходит ряд промежуточных состояний от состояния задания входных данных через промежуточные результаты к выходным данным. Переход из одного состояния памяти в другое есть результат выполнения одной элементарной операции. Количество этих операций есть суммарная трудоемкость, затраченная алгоритмом A для решения конкретной задачи из Z. Эта трудоемкость коррелированна с временной эффективностью программной реализации алгоритма, что приводит к термину временной сложности решения задачи Z алгоритмом A.

Обозначим трудоемкость решения конкретной задачи Z алгоритмом A через $f_A(Z)$. Трудоемкость алгоритма можно определить, проведя вычислительный эксперимент. Однако алгоритм A предназначен для решения общей задачи, состоящей из множества конкретных задач, а трудоемкость $f_A(Z)$ определяется для каждой конкретной задачи индивидуально. Для введения обобщенного параметра

множество решаемых алгоритмом задач разбивают на классы [1.13], внутри которых трудоемкость алгоритма для конкретных задач класса сопоставима. Классы задач обычно определяются одним или несколькими целочисленными параметрами — как правило, размерностями, характеризующими используемую алгоритмом память для размещения структур данных.

Обозначая размерность задачи (длину входа алгоритма) через n, где n — целое число, можно ввести верхнюю границу для количества операций, задаваемых алгоритмом A для решения любой задачи Z из класса размерности n [1.12], — верхнюю границу временной сложности:

$$F_A(n) = \max \{ f_A(Z) | \text{ задача } Z \text{ имеет размерность } n \}.$$

Верхнюю оценку временной сложности алгоритма можно не только найти экспериментально, но и, в большинстве случаев, рассчитать теоретически для произвольного значения n. Поведение функции $F_A(n)$, выраженное в асимптотических оценках, называется асимптотической временной сложностью. Асимптотическая сложность алгоритма является одним из критериев сравнения выбранного алгоритма с техническим заданием, поскольку определяет порядок размерность задач, которые можно решить этим алгоритмом в рамках заданных ограничений.

Задача, связанная с оценкой трудоемкости алгоритма, — это задача поиска наиболее рационального по времени алгоритма решения некоторой задачи Z для размерностей, соответствующих условиям применения алгоритма в разрабатываемой программной системе. Сложность ее решения связана с тем, что совершенствование вычислительной техники ведет к изменению системы команд процессоров и операторов языков программирования, в ряде случаев возможна параллельная или конвейерная обработка данных, поэтому почти никогда нельзя дать гарантию, что кроме известных алгоритмов есть другие, возможно, более эффективные.

Оценка емкостной сложности алгоритма A вводится аналогично. Если ограничиться памятью механизма реализации (для реального компьютера это будет в основном оперативная память без учета внешней памяти и ввода/вывода), то его состояние определяется перечнем значений, записанных в ячейках этой памяти. Тогда логично работу программной реализации алгоритма интерпретировать как

процесс перевода исходного (начального) состояния механизма реализации, представляющего собой исходные данные задачи, в конечное состояние, представляющее собой найденное алгоритмом решение конкретной задачи в терминах слов принятого алфавита.

Пусть m — количество элементарных шагов алгоритма для конкретной задачи Z, тогда определим емкостную сложность задачи в виде

$$V_A(Z) = \max\{ V_A(i) \mid i = \overline{0, m} \},$$

где $V_A(i)$ — количество задействованных алгоритмом ячеек по состоянию после i-го элементарного шага, $V_A(0) = n$ — размерность исходных данных конкретной задачи. Тогда верхняя граница емкостной сложности алгоритма A для размерности n может быть определена в следующем виде

$$V_A(n) = \max\{ V_A(Z) \mid \text{задача } Z \text{ имеет размерность } n \}.$$

Отметим, что верхние оценки емкостной сложности $V_A(n)$ для большинства алгоритмов могут быть получены с незначительными трудозатратами, в том случае, если память, требуемая алгоритмом, не зависит от значений информационных объектов на входе алгоритма.

Отметим, что введенные понятия временной и емкостной сложности, необходимые для изложения основных сложностных классов, рассматриваемых в теории алгоритмов, будут уточнены в главе 3 на основе функции трудоемкости и функции объема памяти.

1.4 Основные сложностные классы задач

В середине 1960-х — начале 1970-х годов в связи с разработкой теории сложностных классов [1.14, 1.15, 1.16] был дан мощный толчок исследованиям, как в области классов вычислительных задач, так и в области анализа сложности вычислительных алгоритмов. Возникло важное направление в теории алгоритмов — теория сложности алгоритмов и вычислений. Одной из первых фундаментальных работ (и остающейся таковой по праву) в области математического анализа сложности алгоритмов является известная книга Д. Кнута [1.4]. Алгоритмы стали объектом точного исследования, и с выделением задач получения верхних и нижних оценок сложности алгоритмов стали развиваться методы их исследования.

В области получения верхних оценок получено много ярких результатов для конкретных задач. Укажем на эффективные алгоритмы умножения длинных целых чисел, решения систем линейных уравнений, умножения числовых и булевых матриц. Для установления нижних оценок необходимо доказать, что никакой алгоритм не имеет сложности меньшей, чем некоторая граница. Получение такого рода результатов связано с точным определением алгоритмической модели, и такие результаты получены для ограниченного ряда задач.

Теория сложности вычислений продолжает развиваться, причем как в направлении исследования и развития собственно сложностных классов задач (рассматриваются классы по вычислительной сложности, памяти и ряд специальных классов) [1.8], так и в области асимптотического анализа сложности вычислительных алгоритмов. Основной задачей теории сложности вычислений при анализе того или иного алгоритма решения задачи является получение асимптотических верхних оценок временной и емкостной сложности алгоритмов и оценок в среднем. Такие оценки позволяют определить качественные предпочтения в использовании того или иного алгоритма для больших значений размерности исходных данных.

Теория сложности оперирует со специальными обозначениями в асимптотическом анализе функций, которые автор хотел бы напомнить уважаемым читателям.

Обозначения в асимптотическом анализе функций. При анализе сложности алгоритмов, равно как и в теории ресурсной эффективности, используются принятые в математике асимптотические обозначения, позволяющие показать главный порядок функции, маскируя при этом конкретные коэффициенты и аддитивные компоненты неглавного порядка.

Поскольку число объектов на входе алгоритма, количество учитываемых операций и объем требуемой памяти суть положительные числа асимптотические обозначения будут введены в предположении, что функции $f(n)$ и $g(n)$ есть функции положительного целочисленного аргумента $n \geq 1$, имеющие положительные значения.

Обозначение 1.1. *Оценка* Θ (*тета*)

Функция $f(n)=\Theta(g(n))$, если:

$$\exists\, c_1>0, c_2>0, n_0>0 : \forall n>n_0 \quad c_1 g(n)\le f(n)\le c_2 g(n). \qquad (1.4.1)$$

При этом обычно говорят, что при этом функция $g(n)$ является асимптотически точной оценкой функции $f(n)$, т. к. в силу (1.4.1) функция $f(n)$ отличается от функции $g(n)$ на положительный ограниченный множитель при всех значениях аргумента $n>n_0$. Запись $f(n)=\Theta(1)$ означает, что функция $f(n)$ или равна константе, не равной нулю, или ограничена двумя положительными константами при любых значениях аргумента $n>n_0$. Более корректно обозначение $\Theta(g(n))$ есть обозначение класса функций, каждая из которых удовлетворяет условию (1.4.1).

Обозначение 1.2. *Оценка* O (*О большое*)

В отличие от оценки Θ, оценка O требует только, что бы функция $f(n)$ не превышала значения функции $g(n)$ при $n>n_0$, с точностью до положительного постоянного множителя, а именно: $f(n)=\mathrm{O}(g(n))$, если:

$$\exists\, c>0, n_0>0 : \forall n>n_0 \quad 0\le f(n)\le c g(n). \qquad (1.4.2)$$

Как и в предыдущем определении, запись $\mathrm{O}(g(n))$ обозначает класс функций, таких, что все они растут не быстрее, чем функция $g(n)$ с точностью до положительного постоянного множителя в силу (1.4.2), поэтому иногда говорят, что функция $g(n)$ мажорирует функцию $f(n)$. Например, для всех функций:

$$f_1(n)=12,\ f_2(n)=5n+23,\ f_3(n)=n\ln n,\ f_4(n)=7n^2+12n-34$$

будет справедлива оценка $\mathrm{O}\left(n^2\right)$. Однако, указывая оценку O есть смысл указывать наиболее «близкую» мажорирующую функцию, поскольку, например, для функции $f(n)=12n^3$ справедлива оценка $\mathrm{O}\left(2^n\right)$, однако практически она будет мало пригодна.

Обозначение 1.3. *Оценка* Ω (*омега*)

В отличие от оценки O, оценка Ω является оценкой снизу — т. е. определяет класс функций, которые растут не медленнее, чем функция $g(n)$ с точностью до положительного постоянного множителя: $f(n)=\Omega(g(n))$, если:

$$\exists\, c > 0, n_0 > 0 : \forall n > n_0 \quad 0 \le cg(n) \le f(n). \qquad (1.4.3)$$

Например, запись $\Omega(n \ln n)$ обозначает класс функций, которые растут не медленнее, чем $g(n) = n \ln n$, в этот класс попадают, например, все полиномы со степенью большей единицы.

Асимптотическое обозначение O восходит к учебнику Бахмана по теории простых чисел, обозначения Θ, Ω введены Д. Кнутом [1.17]. Отметим, что не всегда любая пара функций связана одним из указанных асимптотических обозначений, например следующие функции

$$f(n) = n^{1+\sin(n)},\ g(n) = n$$

не связаны каким либо асимптотическим обозначением.

Класс P — класс задач с полиномиальной сложностью или класс полиномиально-решаемых задач. В начале 1960-х годов, в связи с началом широкого использования вычислительной техники для решения практических задач, возник вопрос о границах практической применимости данного алгоритма решения некоторой задачи в смысле ограничений на ее размерность. Какие задачи могут быть решены на ЭВМ за реальное время? Теоретический ответ на этот вопрос был обоснован в работах Кобмена (Alan Cobham, 1964), и Эдмнодса (Jack Edmonds, 1965), где был независимо введен сложностной класс задач P [1.18].

Задача называется полиномиальной, т. е. относится к классу P, если существует константа k и алгоритм, решающий эту задачу за время $O(n^k)$, где n есть длина входа алгоритма в битах. Отметим, что класс задач P определяется через существование полиномиального по времени алгоритма ее решения, при этом неявно предполагается худший случай по времени для всех различных входов длины n. Интуитивно задачи класса P — это задачи, решаемые за реальное время. Отметим следующие преимущества задач из этого класса:

— для большинства реальных задач из класса P константа k меньше или равна 6, что позволяет прогнозировать реальные времена решения задач для практически значимых размерностей входов;

— задачи класса P обладают инвариантностью (в смысле полиномиального времени) в рамках достаточно широкого класса моделей вычислений;

— класс P обладает свойством естественной замкнутости, т. к. сумма или произведение полиномов также есть полином.

Таким образом, класс P есть класс задач, уточняющий интуитивное определение «практически разрешимой задачи» для входов больших размерностей.

Класс NP — класс задач с полиномиально проверяемым решением или класс полиномиально-проверяемых задач. Представим себе, что некоторый алгоритм получает решение некоторой задачи. Мы вправе задать вопрос — соответствует ли полученный ответ поставленной задаче, и насколько быстро мы можем проверить его правильность?

Рассмотрим в качестве примера задачу о сумме. Дано n целых чисел, содержащихся в массиве $A = \{a_1, \ldots, a_n\}$ и целое число V. Задача состоит в том, чтобы найти массив $X = \{x_1, \ldots, x_n\}, x_i \in \{0,1\}$, такой, что

$$\sum_{k=1}^{n} a_k x_k = V. \qquad (1.4.4)$$

Содержательная постановка задачи выглядит следующим образом: может ли быть представлено число V в виде суммы каких либо чисел из массива A. Если какой-то алгоритм выдает результат — массив X, то проверка правильности этого результата может быть выполнена с полиномиальной сложностью, поскольку проверка справедливости формулы (1.4.4) требует не более $\Theta(n)$ базовых операций.

Попытаемся описать процесс проверки решения определенной задачи, полученного некоторым алгоритмом, более формально. Каждому конкретному множеству исходных данных $D, |D| = n$ поставим в соответствие сертификат S, такой, что $|S| = O(n^l)$, где l — некоторая константа, и алгоритм $A_S = A_S(D, S)$, такой, что он выдает «1», если проверяемое решение правильно, и «0», если решение неверно. Тогда задача принадлежит сложностному классу NP, если существует константа m, и алгоритм A_S имеет временную сложность не более чем $O(n^m)$ [1.18]. Содержательно задача относится к классу NP, если ее решение, полученное некоторым алгоритмом, может быть проверено с полиномиальной временной сложностью. Иначе говоря, задача относится к классу NP, если алгоритм,

проверяющий решение этой задачи относится к классу P. Класс NP был впервые введен в работах Эдмондса.

Проблема $P = NP$**.** После введения в теорию алгоритмов понятий сложностных классов, Эдмондсом (Edmonds, 1965) была сформулирована основная проблема теории сложности — $P = NP$?, и высказана гипотеза о несовпадении этих классов. Словесно проблему можно сформулировать следующим образом: можно ли все задачи, решение которых проверяется с полиномиальной сложностью, решить также за полиномиальное время? Очевидно, что любая задача, принадлежащая классу P, принадлежит и классу NP, т. к. она может быть полиномиально проверена, при этом задача проверки решения может состоять просто в повторном решении задачи.

На сегодня отсутствуют теоретические доказательства как совпадения классов P и NP, так и их несовпадения. В настоящее время предположение состоит в том, что класс P является собственным подмножеством класса NP, т. е. множество задач $NP \setminus P$ не пусто, как это показано на рисунке 1.4. Это предположение опирается на существование еще одного класса, а именно класса NPC или класса NP-полных задач.

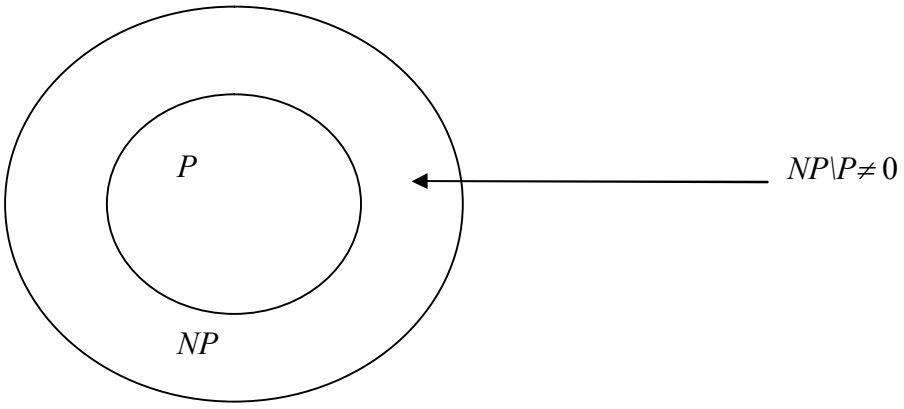


Рис. 1.4. Соотношение классов P и NP.

Класс NPC **или класс** NP**-полных задач**. Понятие NP-полноты было введено независимо Куком (Stephen Cook, 1971) и Левиным (1973) [1.19] и основывается на понятии сводимости одной задачи к другой. Сводимость задач может быть представлена следующим образом: если мы имеем задачу $Z1$ и решающий эту задачу алгоритм, выдающий правильный ответ для всех допустимых входов,

и, предположим, что для задачи $Z2$ алгоритм решения неизвестен, то если мы можем переформулировать (свести) допустимые входы задачи $Z2$ в терминах задачи $Z1$, то мы решаем задачу $Z2$ с помощью алгоритма решения задачи $Z1$.

Более формально процесс сведения может быть описан следующим образом: пусть задача $Z1$ задана множеством допустимых входов D_{Z1}, а задача $Z2$ — множеством D_{Z2}, и существует функция f_s, реализованная некоторым алгоритмом, сводящая конкретную постановку $d_{Z2} \in D_{Z2}$ задачи $Z2$ к конкретной постановке $d_{Z1} \in D_{Z1}$ задачи $Z1$, т. е. если

$$\forall d_{Z2} \in D_{Z2} \quad f_s(d_{Z2} \in D_{Z2}) = d_{Z1} \in D_{Z1},$$

то задача $Z2$ сводима к задаче $Z1$.

Если при этом временная сложность алгоритма, реализующего функцию f_s есть $O(n^k)$, где n — длина входа, равная числу бит d_{Z2}, а k — некоторая константа, т. е. алгоритм сведения принадлежит классу P, то говорят, что задача $Z2$ полиномиально сводится к задаче $Z1$ [2]. В теории сложности вычислений принято говорить, что задача задается некоторым языком, тогда если задача $Z1$ задана языком $L1$, а задача $Z2$ — языком $L2$, то полиномиальная сводимость языков, задающих задачи, обозначается следующим образом

$$L2 \leq_p L1.$$

Определение класса NPC ($NPcomplete$) или класса NP-полных задач требует выполнения следующих двух условий:

— во-первых, задача должна принадлежать классу NP ($L \in NP$),

— во-вторых, к этой задаче должны полиномиально сводиться *все* задачи из класса NP ($\forall Lx \in NP \quad Lx \leq_p L$), что схематично представлено на рисунке 1.5.

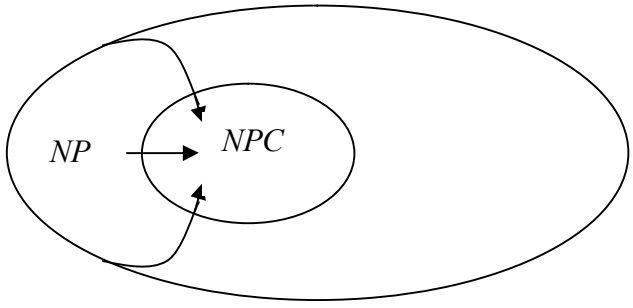


Рис. 1.5. Сводимость задач и класс NPC

Для класса NPC доказана следующая теорема: Если существует задача, принадлежащая классу NPC, для которой существует полиномиальный алгоритм решения, то класс P совпадает с классом NP, т. е. $P = NP$ [1.18]. Схема доказательства состоит в сведении с полиномиальной трудоемкостью любой задачи из класса NP к данной задаче из класса NPC и решении этой задачи за полиномиальное время (по условию теоремы).

В настоящее время доказано существование сотен NP-полных задач [1.20], но ни для одной из них пока не удалось найти полиномиального алгоритма решения. Отметим, что для многих из них предложены приближенные полиномиальные алгоритмы [1.18]. Сегодня ученые предполагают, что соотношение классов, имеет вид, показанный на рис. 1.6., а именно $P \neq NP$, то есть $NP \setminus P \neq 0$, и ни одна задача из класса NPC не может быть решена, по крайней мере, сегодня, с полиномиальной сложностью.

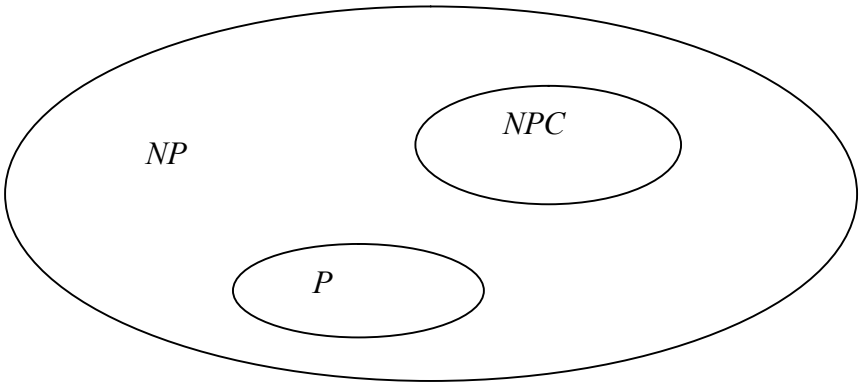


Рис. 1.6. Соотношение классов P, NP, NPC

Исторически первое доказательство NP-полноты было предложено Куком для задачи о выполнимости схемы (теорема Кука). Метод сведения за полиномиальное время был предложен Карпом (Richard Karp) и использован им для доказательства NP-полноты целого ряда задач. Описание многих NP-полных задач содержится в классической книге Гэри и Джонсона [1.20], отметим, что задача о сумме также является NP-полной задачей. Описание некоторых других классов задач, рассматриваемых в теории сложности, уважаемый читатель может найти, например, в [1.8].

Список литературы к главе 1.

[1.1] Post E. L. Finite combinatory process — formulation 1 // J. Symbolic Logic (1936) 1, pp. 103–105.

[1.2] Turing A. M. On Computable Numbers, whiz an Application to the Entsheidungsproblem // Proc. London Math. Soc. (1937) 42, Ser 2, pp. 230–235.

[1.3] Фалевич Б. Я. Теория алгоритмов: Учебное пособие. — М.: Машиностроение, 2004. —160 с.

[1.4] Кнут Д. Э. Искусство программирования, том 1. Основные алгоритмы, 3-е изд.: Пер. с англ. — М.: Издательский дом «Вильямс», 2002. — 720 с.

[1.5] Колмогоров А. Н., Успенский В. А. К определению алгоритма // Успехи математических наук. 1958 . Т. 13. № 4. С. 3–28.

[1.6] Марков А. А. Теория алгоритмов. // Труды математического института АН СССР им. В. А. Стеклова. — М.,1954. Т. 42.

[1.7] Успенский В. А. Машина Поста. — М.: Наука, 1979. — 96 с.

[1.8] Хопкрофт Дж., Мотовани Р., Ульман Дж. Ведение в теорию автоматов, языков и вычислений, 2-е изд.: Пер. с англ. — М.: Издательский дом «Вильямс», 2002. — 528 с.

[1.9] Карпов Ю. Г. Теория автоматов — СПб.: Питер, 2002. — 224 с.

[1.10] Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. В 2 томах. Т. 1. Синтаксический анализ: Пер. с англ.: — М.: Мир, 1978. — 612 с.

[1.11] Носов В. А. Основы теории алгоритмов и анализа их сложности (http: // intsys.msk.ru).

[1.12] Бондаренко В. А. О сложности дискретных задач (http: //edu.yar.ru/russian /pedbank/sor-pro/bondarenko).

[1.13] Макконелл Дж. Основы современных алгоритмов. 2-е дополненное издание. — М.: Техносфера, 2004. — 368 с.

[1.14] Cobham A. The intrinsic computational difficulty of functions // Proc. Congress for Logic, Mathematics, and the Philosophy of Science.–North Holland, Amsterdam, 1964. pp. 24–30.

[1.15] Cook S. C. The complexity of theorem–proving procedures // Third ACM Symposium on Theory of Computing., — ACM, New York, 1971. pp. 151–158.

[1.16] Karp R. M. Reducibility among combinatorial problems // Complexity of Computer Computations / R. E. Miller, ed.,–Plenum Press, New York, 1972. pp. 85–104.

[1.17] Knuth D. Big Omicron and Big Omega and Big Theta. SIGACT News 8(2), (1976), pp. 18–24.

[1.18] Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ, 2-ое издание: Пер. с англ. — М.: Издательский дом «Вильямс», 2005. — 1296 с.

[В.12] Левин Л. А. Универсальные проблемы упорядочения // Проблемы передачи информации. 1973. — Т. 9. № 3. С. 115–117.

[1.19] Гэри М., Джонсон Д. Вычислительные машины и труднорешаемые задачи. — М.: Мир, 1982. — 416 с.

ГЛАВА 2.

МОДЕЛИ ВЫЧИСЛЕНИЙ И АЛГОРИТМИЧЕСКИЕ БАЗИСЫ

Введение

Понятие модели вычислений является важным как с точки зрения теории алгоритмов, так и с точки зрения теории ресурсной эффективности. Основные показатели ресурсной эффективности алгоритмов — функция трудоемкости и функция объема памяти могут быть конкретизированы только в терминах выбранной модели вычислений. Материал данной главы содержит необходимые сведения о моделях вычислений и описание той модели, на базе которой в дальнейшем будут исследоваться ресурсные характеристики компьютерных алгоритмов.

2.1 Введение в модели вычислений

Понятие модели вычислений. Рассматриваемые в теории алгоритмов модели вычислений существенно опираются на свойства алгоритмов, которые были сформулированы в параграфе 1.2. Формализация этих свойств и приводит к понятию модели вычислений и ее основных составляющих. Рассмотрим более подробно свойства алгоритмов, определяющие компоненты модели вычислений:

— свойство наличия данных, обрабатываемых алгоритмом, может быть более формально описано как существование некоторого множества объектов, над которыми выполняются операции, задаваемые алгоритмом. Как правило, эти объ-

екты рассматриваются как символы некоторого конечного алфавита, которые могут быть кодированы словами в алфавите $\{0,1\}$;

— свойство наличия памяти для размещения объектов приводит к формальному введению носителя некоторой алгебры, сигнатура которой представляет собой набор операций доступа к этим объектам. Такая алгебра часто называется информационной алгеброй модели вычислений;

— свойство априорности операций приводит к постулированию некоторого конечного набора операций, в дальнейшем называемых базовыми, которые однозначно определены и известны до формулировки алгоритма. Как правило, эти базовые операции включают в себя и операции сигнатуры информационной алгебры для доступа к объектам носителя;

— свойство существования механизма реализации предполагает, что существует некоторый абстрактный механизм, который выполняет базовые операции над объектами носителя и операции сигнатуры информационной алгебры. Кроме того, предполагается, что механизм реализации распознает запись алгоритма, задающего последовательность базовых операций, и тем самым предполагается наличие еще одной группы базовых операций, которые путем управления процессом выбора операций в записи алгоритма определяют последовательность их выполнения.

Таким образом, модель вычислений представляет собой совокупность информационной алгебры, механизма реализации и априорного набора базовых операций, в терминах которых и формулируется алгоритм решения задачи.

Основные алгоритмические формализмы — машина Поста, машина Тьюринга, нормальные алгорифмы Маркова, могут быть представлены в виде модели вычислений, содержащей вышеприведенные компоненты.

Приведем пример формального описания модели вычислений, основанного на теоретико-множественном аппарате, с целью показать один из возможных вариантов построения информационной алгебры. Данный пример основан на результатах, опубликованных в статье [2.1].

Теоретико-множественное описание информационной алгебры. Выделение операций с памятью, и связанных с ними операций адресации, в отдельные

операции модели вычислений приводит к необходимости формального описания памяти и операций с объектами хранения.

Наша задача — унифицировать операции адресации и доступа, как унарные операции сигнатуры алгебры, которую далее будем называть информационной, над носителем, теоретико-множественная описание которого является одной из возможных моделей памяти с адресным доступом. Следуя классическим алгоритмическим формализмам Поста и Тьюринга, будем считать, что носитель информационной алгебры есть счетное множество.

Теоретико-множественное описание информационной алгебры I_A начнем с введения конечного алфавита символов — Σ, элементы которого являются информационными объектами хранения в модели памяти с адресным доступом

$$\Sigma = \{ s_1, \ldots s_{n-1}, e \},$$

причем выделенный символ $e = s_n$ является пустым символом. Каждая ячейка памяти имеет собственный абстрактный адрес — элемент из счетного множества адресов A, которое вполне упорядочено по некоторому отношению следования $\prec$, так, что

$$\forall\, k \in Z \quad a_k \prec a_{k+1} : \mathrm{A} = (\cdots a_{k-1}, a_k, a_{k+1} \ldots),$$

где Z — множество целых чисел, при этом множество A не содержит повторяющихся элементов. Если мы предполагаем, что ячейки памяти нумерованы, то множество A совпадает с множеством целых чисел. Множество информационных элементов I введем как собственное подмножество декартова произведения

$$\Sigma \times \mathrm{A} \ — \ I \subset \Sigma \times \mathrm{A},$$

при этом информационный элемент $Ie \in I$ представляет собой упорядоченную пару

$$Ie \in \Sigma \times \mathrm{A}; \quad Ie_k = (s \in \Sigma, a_k \in \mathrm{A}) = (s, a_k), k \in Z,$$

мы используем здесь известное в литературе обозначение упорядоченной пары элементов a, b в виде (a, b), равносильное классическому определению — $\{ a, \{a, b\} \}$. При этом каждая ячейка памяти заполнена, по крайней мере, пустым символом по всему множеству абстрактных адресов, и имеет собственный уникальный адрес

$$\forall a \in \mathrm{A}\ \exists s \in \Sigma : (s, a) \in I,$$

$$\forall a \in \mathrm{A} \quad (s_i, a) \in I \,\&\, (s_j, a) \in I \Rightarrow i = j.$$

Введем два одноэлементных множества

$$\zeta_s = \{ s \in \Sigma \},\ \zeta_a = \{ a \in \mathrm{A} \},\ |\zeta_s| = 1,\ |\zeta_a| = 1,$$

рассмотрим упорядоченное множество $\zeta = \zeta_s \times \zeta_a = (s, a)$ и введем носитель информационной алгебры J в виде

$$J = I \times \zeta; \quad Je \in J: \quad \forall k \in Z \quad Je_k = (Ie_k, \zeta) = ((s, a_k), (s, a)).$$

Введем сигнатуру информационной алгебры в виде конечного множества операций Z над носителем J:

$$Z = \{ S(s_i), A(a_i), N(a_l), P(a_l), R(a_l), W(a_l) \},$$

где полагая, что в текущем состоянии носителя

$$\zeta_s = \{ s_m \}, \zeta_a = \{ a_l \}, \text{ т. е. } \zeta = \zeta_s \times \zeta_a = (s_m, a_l),$$

операция $S(s_i)$ есть операция задания символа

$$S(s_i): \quad \forall k \in Z \quad ((s, a_k), (s_m, a_l)) \to ((s, a_k), (s_i, a_l)),$$

операция $A(a_i)$ — операция задания произвольного адреса

$$A(a_i): \quad \forall k \in Z \quad ((s, a_k), (s_m, a_l)) \to ((s, a_k), (s_m, a_i)),$$

операция $N(a_l)$ — операция задания следующего адреса

$$N(a_l): \quad \forall k \in Z \quad ((s, a_k), (s_m, a_l)) \to ((s, a_k), (s_m, a_{l+1})),$$

операция $P(a_l)$ — операция задания предыдущего адреса

$$P(a_l): \quad \forall k \in Z \quad ((s, a_k), (s_m, a_l)) \to ((s, a_k), (s_m, a_{l-1})),$$

операция $R(a_l)$ — операция чтения по адресу

$$R(a_l): \quad \forall k \in Z \quad \begin{cases} ((s, a_k), (s_m, a_l)) \to ((s, a_k), (s_m, a_l)), k \neq l \\ ((s_j, a_k), (s_m, a_l)) \to ((s_j, a_k), (s_j, a_l)), k = l' \end{cases}$$

операция $W(a_l)$ — операция записи по адресу

$$W(a_l): \quad \forall k \in Z \quad \begin{cases} ((s, a_k), (s_m, a_l)) \to ((s, a_k), (s_m, a_l)), k \neq l \\ ((s_j, a_k), (s_m, a_l)) \to ((s_m, a_k), (s_m, a_l)), k = l \cdot \end{cases}$$

Поскольку множество $\zeta_s = \{ s_i \in \Sigma \}$ и элемент этого множества — s_i являются различными объектами с точки зрения теории множеств, введем дополнительно еще одну операцию, определенную на множествах ζ_s, ζ_a со значениями из

множеств Σ, A соответственно. Результатом этой операции является элемент, содержащийся в одноэлементном множестве. Следуя [2.2] будем называть ее штрих операцией, и обозначать как ζ'_s, ζ'_a:

$$\begin{cases} \zeta'_s = s_i \in \Sigma, \quad s_i : \zeta_s = \{ s_i \} \\ \zeta'_a = a_i \in \mathrm{A}, \quad a_i : \zeta_a = \{ a_i \} \end{cases}.$$

С использованием штрих операции обращение, например, к операции сигнатуры $R(\zeta'_a)$ при условии, что $\zeta_a = \{ a_l \}$ есть обращение к операции R с элементом a_l — $R(\zeta'_a) \equiv R(a_l)$. Особо отметим, что в предположении о том, что $\zeta_a = \{ a_l \}$ операции сигнатуры $N(a_l), P(a_l), R(a_l), W(a_l)$ есть операции, аргументом которых является ζ'_a.

Таким образом, информационная алгебра $I_A = \langle J, \mathrm{Z} \rangle$ является моделью памяти с адресным доступом, сигнатура которой состоит формально из унарных операций. Операции сигнатуры позволяют явно разделить операции адресации и доступа в целях детализации функции трудоемкости.

Модель вычислений с предусловием выполнения базовых операций. На основе сигнатуры информационной алгебры введем модель вычислений M_C и, используя принятые в дискретной математике обозначения [2.3], представим формально модель вычислений M_C в виде двойки

$$M_C = \langle I_A, P \rangle,$$

где $I_A = \langle J, \mathrm{Z} \rangle$ — информационная алгебра модели вычислений; P — абстрактный процессор. Абстрактный процессор представим в виде тройки

$$P = \langle R, C, \Omega \rangle,$$

где R — механизм реализации в виде абстрактной модели, представляющей собой, например, композицию операционного и управляющего автоматов [2.2]; C — множество базовых операций модели вычислений, выполняемых механизмом реализации

$$C = C_I \cup C_D \cup C_P \cup \{ \mathrm{stop} \},$$

где C_I — операции механизма реализации с информационным носителем, включающие в себя операции сигнатуры Z информационной алгебры I_A и штрих опе-

рацию над одноэлементным множеством; C_D — операции сравнения и обработки информационных элементов; C_P — операции управления последовательностью выполнения операций базовой модели вычислений — условные и безусловные переходы на операции базовой модели. Команда stop приводит к останову механизма реализации.

Мы остаемся в рамках классического подхода к способу записи алгоритма в операциях модели вычислений — базовые операции выполняются последовательно, пока не будет выполнена команда перехода на некоторую метку в записи исходного алгоритма. Для описания предусловий выполнения базовых операций введем в рассмотрение множество $\Omega = \{ \varphi, \psi \}$, состоящее из двух одноэлементных множеств φ, ψ, элементами которых являются числа 0 или 1. По сути, множество Ω представляет собой множество управляющих ячеек механизма реализации, используемых в предусловии выполнения операций. Любая операция выполняется механизмом реализации, если указанное в предусловии одноэлементное множество содержит число 1. Таким образом, общий формат базовой операции имеет вид

(элемент из Ω, базовая операция из C).

Распространяя штрих операцию на одноэлементные множества φ, ψ, мы можем описать общий алгоритм работы механизма реализации следующим образом: если $\varphi' = 1$ или $\psi' = 1$ то, указанная далее в записи операция выполняется механизмом реализации. Очевидно, что операции сравнения должны быть введены таким образом, что бы они изменяли элемент, содержащийся во множествах φ, ψ. В связи с этим для записи операций сравнения содержимого одноэлементных множеств можно использовать нотацию, предложенную К. Айверсоном [2.4], а именно для двух элементов a, b

$$[\, a = b \,] = \begin{cases} 0, & a \neq b \\ 1, & a = b \end{cases}.$$

С учетом предусловия выполнения команд и записи сравнений в нотации Айверсона запись операции сравнения в модели вычислений имеет вид

$$\left(\psi,\ \varphi = \left\{ [\, \zeta'_s \ \mathrm{Cp} \ s_j \,] \right\} \right), \quad \mathrm{Cp} = \{ =, \neq, <, \leq, >, \geq \},$$

при этом сравнение будет выполнено, если $\psi' = 1$, и если, например, Ср = "=", то если одноэлементное множество ζ_s, входящее в информационную алгебру I_A, содержит символ s_j, то после выполнения этой операции сравнения $\varphi = \{1\}$.

Пример. В качестве примера приведем запись некоторых команд машины Поста в терминах рассматриваемой модели вычислений. Алфавит символов машины Поста может быть представлен в виде $\Sigma = \{\vee, e\}$, где символ $\vee$ обозначает помеченную ячейку. Будем предполагать, что инициализация абстрактного процессора детализированной модели вычислений выполнена занесением 1 во множество ψ. Для описания команд перехода будем использовать запись @число, предполагая, что операции модели вычислений имеют числовые метки.

1. *Переместиться влево на 1 ячейку*. Считаем, что одноэлементное множество ζ_a содержит адрес обозреваемой ячейки — элемента информационной алгебры, тогда запись команды имеет вид

$$(\psi,\ P(\zeta'_a)).$$

2. *Определить, помечена ячейка или нет, и перейти на указанную инструкцию*. Приведем пример для команды: Если ячейка помечена, то перейти на команду 17, иначе на команду 14 (номера команд в записи инструкций машины Поста):

$(\psi,\ R(\zeta'_a))$; чтение — элемент из ячейки занесен в ζ_s

$(\psi,\ \varphi = \{[\zeta'_s = \vee]\})$; сравнение метки

$(\varphi,\ @17)$; переход на базовую операцию с меткой 17

$(\psi,\ @14)$.

3. *Стереть метку, если она есть*.

$(\psi,\ R(\zeta'_a))$; чтение — элемент из ячейки занесен в ζ_s

$(\psi,\ \varphi = \{[\zeta'_s = e]\})$; сравнение метки

$(\varphi,\ \text{stop})$; останов по недопустимой команде

$(\psi,\ S(e))$; задание пустого символа для записи

$(\psi,\ W(\zeta'_a))$. запись в обозреваемую ячейку

Отметим, что введенная модель вычислений может быть использована в целях детального теоретического анализа алгоритмов с разделением функции тру-

доемкости по базовым операциям, включая выделение операций доступа к информационным объектам.

Специальные модели вычислений для оценки сложности алгоритмов. В области анализа и исследования компьютерных алгоритмов рассматриваются разнообразные модели вычислений, ориентированные на решение задач оценки сложности алгоритмов.

Алгебраические деревья вычислений. Одной из таких моделей, используемых для исследования сложности алгоритмов, является алгебраическое дерево вычислений, предложенное Бен-Ором [2.5]. Алгебраическое дерево вычислений (АДВ) строится как бинарное дерево D на множестве переменных, принимающих значения на множестве действительных чисел. Такое бинарное дерево размечается следующим образом [2.6]:

— каждой вершине v, имеющей в точности одного сына (простая вершина), приписывается операция вида $f_v = f_{v1}\ op\ f_{v2}$, или $f_v = f_{v1}\ op\ c$, или $f_v = \sqrt{f_{v1}}$, где v_1, v_2 — предки вершины v в дереве D, или переменные, op — арифметическая операция, а c — константа.

— каждой вершине v, у которой есть в точности два сына (вершина ветвления), приписывается операция сравнения вида $f_{v1} > 0, f_{v1} < 0$ $f_{v1} = 0$, где v_1 — предок вершины v в дереве D, или переменная.

— каждому листу дерева приписывается одно из значений «ДА» или «НЕТ».

Считается, что алгебраическое дерево вычислений D решает задачу Z, если возвращаемый ответ верен для любого запроса $x \in X$. Сложность дерева D, обозначаемая через $C(D)$, определяется как максимум величины $cost(x, D)$, взятый по всем значениям x, где $cost(x, D)$ есть число вершин, проходимых путем $P(x)$ в дереве. В этом случае сложность задачи Z, обозначаемая как $C(Z)$, есть минимум $C(D)$ по всем алгебраическим деревьям вычислений D, решающих задачу Z.

Одной из разновидностей алгебраического дерева вычислений является дерево решений порядка d. Это дерево, каждой вершине которого соответствует

сравнение вида $f(x) > 0, f(x) < 0, f(x) = 0$, где вычисляемая функция $f(x)$ имеет полиномиальную сложность относительно входа x с показателем степени не более d. Для случая $d = 1$ мы получаем линейное дерево решений, с использованием которого получены доказательства нижних оценок сложности многих задач [2.7]. Ряд результатов по временной сложности алгебраических деревьев получен М. Ю. Мошковым [2.8].

Информационные графы. Специальная и мощная по своим возможностям модель, ориентированная на анализ алгоритмов и задач информационного поиска — информационный граф, — предложена Гасановым и Кудрявцевым [2.6]. Структурно информационный граф вводится в [2.6] следующим образом:

Пусть дана пара $\Re = \langle F, G \rangle$, называемая базовым множеством, где F — есть множество одноместных предикатов, заданных на множестве запросов X, а G — есть множество одноместных переключателей, заданных на множестве X. Задано так же множество записей Y.

Пусть дана ориентированная многополюсная сеть, один из выделенных полюсов которой называется корнем, а все остальные — листьями. Некоторые выделенные вершины сети называются точками переключения.

Для каждой вершины β сети полустепень исхода вершины обозначается через ψ_β (обозначения в соответствии с [2.6]). Каждой точке переключения β сопоставляется некоторый символ из G; такое соответствие называется нагрузкой точек переключения.

Далее для каждой точки переключения исходящим из нее ребрам ставится во взаимно однозначное соответствие число из множества $\{1, 2, ..., \psi_\beta\}$. Такие ребра называются переключательными, а остальные ребра называются предикатными. Каждому предикатному ребру сопоставляется некоторый символ из F — это есть нагрузка предикатных ребер. Каждому листу сети сопоставляется некоторая запись из множества Y, при этом такое сопоставление называется нагрузкой листьев. Полученная нагруженная сеть называется информационным графом над базовым множеством $\Re = \langle F, G \rangle$. На основе введенной модели вычислений в

[35] получен ряд теоретических результатов по оценкам сложности задач информационного поиска.

Информацию о других моделях вычислений заинтересованный читатель может найти, например, в [2.9].

2.3 Машина с произвольным доступом к памяти

Модель вычислений для машины с произвольным доступом к памяти (RAM) является естественной попыткой приблизить формализм алгоритма к среде его реализации. В этой модели механизм реализации приближен к реальному компьютеру за счет включения возможности хранения чисел в ячейках памяти и введения операции доступа по адресу [2.10]. Формализм машины с произвольным доступом был введен ее авторами с целью моделирования реальных вычислительных машин и получения оценок сложности вычислений. Отметим, что Д. Кнут в [2.11] рассматривает и использует еще более приближенную к реальному компьютеру модель вычислений — машину MIX.

Структура машины с произвольным доступом к памяти. Модель вычислений машины с произвольным доступом к памяти включает в себя следующие структурные компоненты:

— данные, обрабатываемые машиной, являются словами фиксированной длины в алфавите $\{0,1\}$, интерпретируемые как произвольные числа;

— память под размещение данных является памятью с произвольным доступом. Она состоит из ячеек, каждая из которых может хранить слово данных и обладает адресом, по которому может быть осуществлено обращение к этой ячейке. Ячейка с номером 0 называется сумматором;

— базовые операции — множество базовых операций машины с произвольным доступом к памяти состоит из операций с памятью, включая операцию адресации, арифметических операций, сравнений и переходов на другие операции. Все базовые операции являются операциями с одним операндом, для арифметических операций вторым операндом является ячейка с номером 0;

— форма записи — в качестве формы записи будем предполагать (т. к. впоследствии не будем явно использовать) форму, аналогичную машине MIX [2.11],

т. е. близкую к языку Ассемблера. Операции записываются в порядке их выполнения построчно, команды переходов указывают номер или имя команды. Таким образом, программа (запись алгоритма) представляет собой последовательность пронумерованных операций;

— механизм реализации модели — абстрактный процессор, выполняющий априорный набор базовых операций на основании записи алгоритма, выполненной в принятой форме записи.

Операции в машине с произвольным доступом к памяти. Базовыми операциями в машине с произвольным доступом к памяти, выполняемыми за единицу времени, являются:

— запись в ячейку памяти по адресу;

— чтение из ячейки памяти по адресу;

— арифметические операции с числами;

— сравнения чисел с последующим переходом на другие команды;

— безусловный переход на другую команду;

— останов вычислений.

Типы операндов. В этой модели вычислений рассматриваются операнды следующих трех типов:

— непосредственный операнд, т. е. числовое значение, которое указано в записи операции;

— прямой адрес, т. е. в записи операции указывается адрес ячейки памяти, которая содержит операнд;

— косвенный адрес, т. е. в записи операции указывается адрес ячейки, которая содержит адрес необходимого операнда.

Более подробную информацию о этой модели читатель может найти, например, в [2.9].

Как наиболее близкая к записи алгоритма на процедурном языке программирования высокого уровня, эта модель вычислений с определенными модификациями будет применяться в дальнейшем для определения ресурсных характеристик алгоритмов.

В дальнейшем в книге под термином *компьютерный алгоритм* понимается реализуемый на ЭВМ алгоритм, соответствующий модели вычислений машины с произвольным доступом к памяти (RAM модели вычислений), под *ресурсной эффективностью алгоритма* — его временная эффективность и ресурсоемкость, определяющие соответствующие показатели качества программных средств (ГОСТ 28195).

2.3 Объектный и алгоритмический базисы

Использование в качестве модели вычислений классических алгоритмических формализмов — таких как машина Тьюринга, машина Поста, нормальных алгоритмов (алгорифмов) Маркова — сопряжено с трудностями преодоления «семантического разрыва» между этими моделями и языком реализации алгоритма. Наиболее близкой к реальному компьютеру является рассмотренная выше 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 предложен автором в [2.12] на основе введения понятий объектного и алгоритмического базисов.

Определение объектного и алгоритмического базисов. По отношению к процессу разработки алгоритма решения некоторой задачи мы можем говорить о существовании следующих трех компонент, введённых в параграфе 2.1, которые определяют базис модели вычислений:

— множество объектов Σ, над которыми производятся действия алгоритма, включающее множество исходных объектов — D_A, множество промежуточных объектов — D_P, и множество результатов — D_R:

$$\Sigma = D_A \bigcup D_P \bigcup D_R;$$

— устройство, производящее действия над объектами (элементами множества Σ), — процессор R — механизм реализации;

— конечное множество базовых 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:

Определение 2.1. *Объектным базисом* (B_R) будем в соответствии с [2.12] называть тройку, состоящую из множества объектов, процессора и множества выполняемых им базовых операций:

$$B_R = \{ \Sigma, R, C \}.$$

Таким образом, объектный базис ассоциирован с выбранной моделью вычислений в части, определяющей трудоемкость и ёмкостную сложность алгоритма.

Собственно процесс разработки алгоритма, являясь очевидно интеллектуальным процессом, состоит в определении конечной последовательности базовых 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 базовых 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 базовой 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 базовых 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, то определение алгоритмического базиса вводится следующим образом [2.12]:

Определение 2.2. *Алгоритмическим базисом* (B_A) для заданного объектного базиса B_P будем называть двойку, включающую в себя конечное множество обозначений базовых 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 существо (идею) алгоритма, то различные алгоритмические базисы одного и того же объектного базиса 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 принятой модели вычислений.

Определение 2.3. Будем говорить, что алгоритмический и объектный базисы по функции трудоемкости $O(1)$ — *эквивалентны*, если

$$\begin{cases} \forall\, s \in S : s = C' \subset C^k, k : k \geq 1,\ k = O(1); \\ \forall\, t \in T :\ t = C' \subset C^k, k : k \geq 1,\ k = O(1). \end{cases}$$

Введенная $O(1)$ — эквивалентность гарантирует, что если базовые 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 базовых операций принятой модели вычислений, выполняемых механизмом реализации, приводящую к решению поставленной задачи за конечное время.

2.4 Модель вычислений для языка процедурного программирования с поддержкой механизма рекурсивного вызова

Описание модели вычислений. В целях удобства анализа компьютерных алгоритмов, запись которых приближена к языку процедурного программирования высокого уровня, необходимо ввести соответствующую модель вычислений. Переход от модели вычислений машины с произвольным доступом к памяти к модели вычислений процедурного языка высокого уровня связан с тем, что классический процедурный подход остается в настоящее время наиболее широко используемым при программной реализации алгоритмов.

Эта модель вводится как некоторое расширение машины с произвольным доступом к памяти в части набора базовых операций, и фактически является новым алгоритмическим базисом для объектного базиса классической модели — машины с произвольным доступом к памяти. Дополнительно модель включает в себя программный стек и операции его обслуживания в целях учета ресурсных затрат рекурсивных алгоритмов.

В дальнейшем будем считать, что объектами хранения в информационном носителе являются битовые слова фиксированной длины, обращение к которым производится по их адресу в символической записи. Механизм реализации, аналогичный классическому последовательному процессору фон-Неймановской архитектуры предполагает расположение программы и данных в информационном носителе. При этом считается, что базовыми операциями такого процессора являются операции, коррелированные с основными операторами процедурного языка высокого уровня.

Базовые операции механизма реализации. Для вводимой модели вычислений процедурного языка высокого уровня базовыми операциями будем считать следующие:

— простое присваивание: $a \leftarrow b$;

— одномерная индексация: $a[i]$: (адрес $(a) + i \cdot$ длина элемента);

— арифметические операции: $\{ *, /, -, + \}$;

— операции сравнения: $a \{ <, >, =, \leq, \geq \} b$;

— логические операции: $(l1)\{or, and, not\}(l2)$;

— операции взятия содержимого по адресу (штрих-операция) и адресации в сложных типах данных $(name1.name2)$.

По отношению к введенному набору базовых операций необходимо сделать несколько замечаний:

— опираясь на идеи структурного программирования, из набора базовых операций исключается команда перехода, поскольку ее можно считать связанной 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 для получения операндов включаются в результативные операции обработки этих операндов;

— 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 базовых операций.

С учетом этих замечаний можно говорить, что введенный алгоритмический базис выбранной модели вычислений является по трудоемкости $O(1)$ — эквивалентным машине с произвольным доступом к памяти.

Модель программного стека. Анализ рекурсивных алгоритмов предполагает учет трудоемкости обслуживания вызова процедуры ил функции. Поскольку реально такие вызовы обслуживаются через программный стек, то формальное рассмотрение механизма организации рекурсивных вызовов предполагает введение модели программного стека. Будем предполагать далее, что модель вычислений поддерживает специальную область памяти — программный стек, используя специальную ячейку памяти — указатель стека, хранящий адрес некоторой ячейки в этой области.

Две специальные операции — «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, операция «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 операций не зависит от объема хранимой информации.

Хотя наши операции неявно используют текущий указатель вершины программного стека, тем не менее, мы рассматриваем их как операции с одним операндом, который указывает элементарную ячейку хранения, содержимое которой либо помещается в стек, либо считывается в них с вершины стека. Приведем пошаговое описание этих операций.

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

2. «Читать из стека» <операнд>. Операция 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. 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.

Схематично этот механизм проиллюстрирован на рисунке 2.1 — мы считаем, что вызывающая программа сама сохраняет регистры, а вызываемая процедура восстанавливает их перед передачей управления (возвратом).

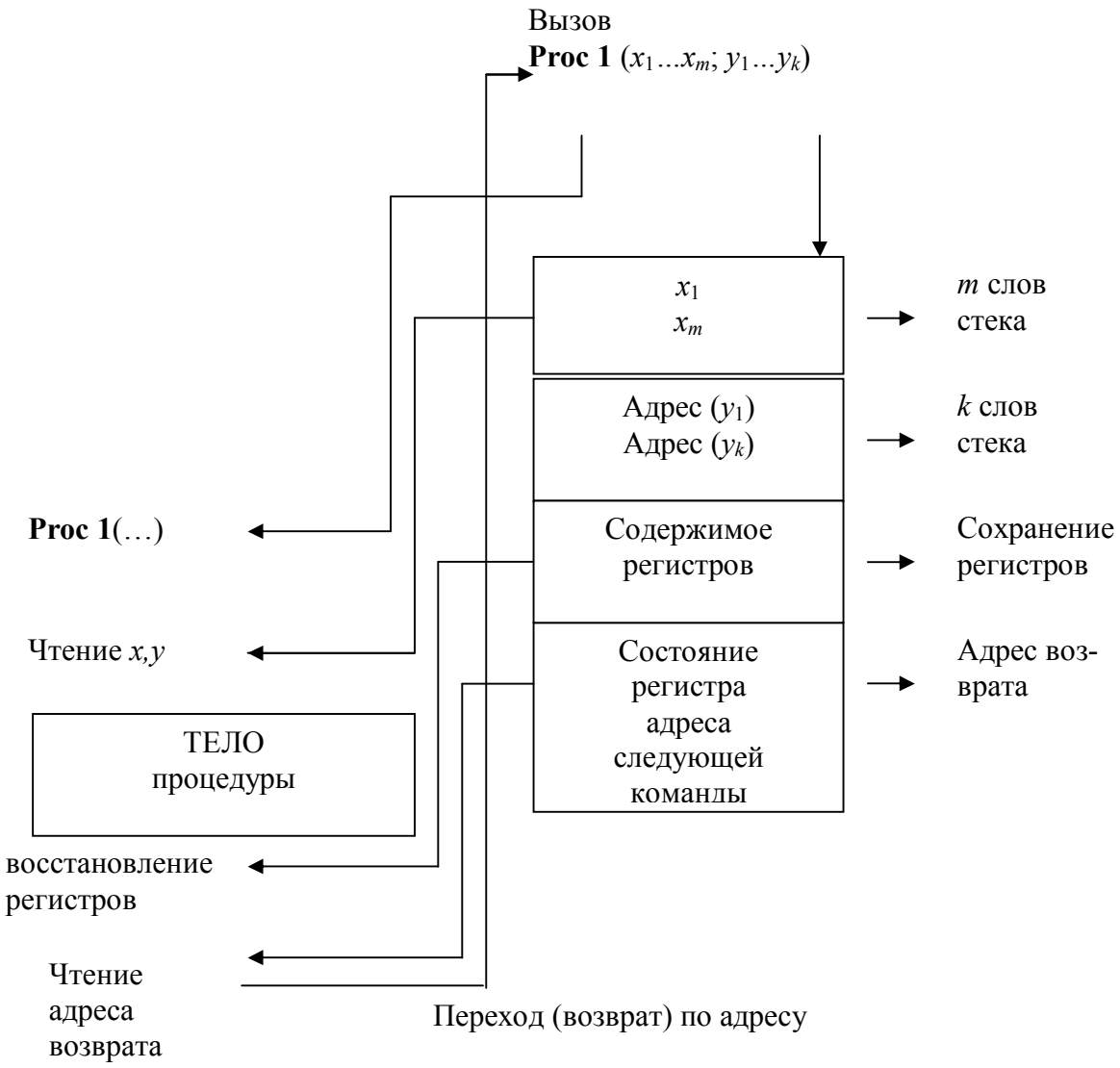


Рисунок 2.1. Механизм вызова процедуры с использованием стека.

Для обеспечения этих действий мы и будем использовать описанный выше программный стек и обслуживающие его операции. 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.

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).

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 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 рисунке 2.2. Опишем этапы рекурсивного вызова и возврата более подробно (нумерация этапов соответствует рисунку 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.

Рисунок 2.2. Механизм рекурсивного вызова процедуры.

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 A 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. В свете вышесказанного этот механизм было бы более правильно назвать механизмом рекурсивного вызова-возврата.

Еще раз напомним уважаемым читателям, что изложенный здесь механизм является, пусть и приближенной к действительности, но все же моделью механизма организации рекурсивного вызова в описываемой модели вычислений.

Компиляторы языков программирования высокого уровня могут иметь собственные особенности организации такого механизма, и читателям есть смысл ознакомиться с тем, как, например, этот механизм реализован в языках *Delphi* или *СИ*.

Введенная модель вычислений будет использоваться при дальнейшем изложении материала книги, и именно на её основе будут исследоваться алгоритмы решения задач, составляющих основной иллюстративный материал по разработке ресурсно-эффективных алгоритмов.

Список литературы к главе 2.

[2.1] Михайлов Б. М., Головешкин В. А., Ульянов М. В. Модель вычислений с информационной алгеброй и предусловием выполнения элементарных операций // Вестник МГАПИ. Серия: Технические и естественные науки. 2006. №3. С. 114–123.

[2.2] Глушков В. М., Цейтлин Г. Е., Ющенко Е. Л. Алгебра. Языки. Программирование. — Киев: Наукова думка, 1978. — 318 с.

[2.3] Белоусов А. И., Ткачев С. Б. Дискретная математика: Учеб. для вузов / Под ред. В. С. Зарубина, А. П. Крищенко. — М.: Изд-во МГТУ им. Н.Э. Баумана, 2001. — 744 с.

[2.4] Грэхем Р., Кнут Д., Паташник О. Конкретная математика. Основание информатики: Пер. с англ. — М.: Мир, 1998. — 703 с.

[2.5] Ben-Or M. Lower bounds for algebraic computation trees // Proc. 15th ACM Annu. Symp. Theory Comput. Apr. 1983. pp. 80–86.

[2.6] Гасанов Э. Э., Кудрявцев В. Б. Теория хранения и поиска информации. — М.: Физматлит, 2002. — 288 с.

[2.7] Steele L. M., Yao A. C. Lower bounds for algebraic decision trees // J. Algorith. 1982., 3. P. 1–8.

[2.8] Мошков М. Ю. О глубине деревьев решений // Доклады РАН. 1998. т. 358, №1. С. 26–32.

[2.9] Алексеев В. Е., Таланов В. А. Графы и алгоритмы. Структуры данных. Модели вычислений: Учебник. — М.: Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2006. —320 с.

[2.10] Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов: Пер. с англ.: — М.: Мир, 1979. — 546 с.

[2.11] Кнут Д. Э. Искусство программирования, том 1. Основные алгоритмы, 3-е изд.: Пер. с англ. — М.: Издательский дом «Вильямс», 2002. — 720 с.

[2.12] Ульянов М. В. Классификация и методы сравнительного анализа вычислительных алгоритмов. Научное издание. — М.: Издательство физико-математической литературы, 2004. — 212 с.

РАЗДЕЛ II

ТЕОРИЯ РЕСУРСНОЙ ЭФФЕКТИВНОСТИ ВЫЧИСЛИТЕЛЬНЫХ АЛГОРИТМОВ

Цель настоящего раздела — не только формализовать понятия и ввести терминологию, которая используется в области анализа и исследования эффективности вычислительных алгоритмов, но и описать некоторые классификации алгоритмов, которые будут впоследствии использоваться. В разделе приводятся также методики оценки и выбора рациональных алгоритмов, которые могут быть использованы для обоснования решений при разработке алгоритмического обеспечения программных средств и систем. В целом раздел содержит краткое изложение теории ресурсной эффективности вычислительных алгоритмов, которая является областью научных исследований автора этой книги.

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 теории ресурсной эффективности вычислительных алгоритмов включает в себя следующие компоненты:

— 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 посвящен настоящий раздел.

Г Л А В А 3 .

В В Е Д Е Н И Е В Т Е О Р И Ю Р Е С У Р С Н О Й Э Ф Ф Е К Т И В Н О С Т И В Ы Ч И С Л И Т Е Л Ь Н Ы Х А Л Г О Р И Т М О В

Введение

Материал этой главы начинается с изложения основных понятий и определений, используемых в теории ресурсной эффективности, которые широко будут использоваться в дальнейшем. Понятия функции трудоемкости и функции объема памяти являются в этом отношении определяющими. На их основе рассматриваются варианты построения комплексного критерия оценки качества алгоритмов. В последнем параграфе показывается, как на основе функции трудоемкости и ряда экспериментальных данных можно прогнозировать времена выполнения программных реализаций компьютерных алгоритмов.

3.1 Терминология и обозначения в теории ресурсной эффективности

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 [3.1, 3.2], однако собственная система обозначений развита слабо, и представлена, в основном, обозначениями асимптотического роста функций, или, в ряде случаев, отсутствует. 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 автором в [3.3], этот материал также изложен в [3.4].

Временная сложность и функция трудоемкости. Классический анализ вычислительных алгоритмов связан, прежде всего, с анализом их временной сложности. Его результатом является асимптотическая оценка количества задаваемых алгоритмом операций, как функции длины входа, которая коррелированна с асимптотической оценкой времени выполнения программной реализации 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 3.1. *Трудоёмкость алгоритма*.

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

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

Под трудоёмкостью алгоритма A на входе D, будем понимать количество базовых 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 положительной число (в силу предположения о том, что алгоритм A является финитным 1-процессом по Посту [3.5]).

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 понимается наибольшее количество операций, задаваемых алгоритмом A на всех входах размерности n, т. е. на всех входах $D \in D_n$:

$$f_A^{\wedge}(n) = \max_{D \in D_n} \{ f_A(D) \},$$

по аналогии через $f_A^{\vee}(n)$ будем обозначать *лучший случай*, как наименьшее количество операций, задаваемых алгоритмом A на входах размерности n:

$$f_A^{\vee}(n) = \min_{D \in D_n} \{ f_A(D) \}.$$

Трудоемкость алгоритма в среднем будем обозначать через $\overline{f_A}(n)$ — это *средний случай* трудоемкости, т. е. среднее количество операций, задаваемых алгоритмом A на входах размерности n:

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

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 $\overline{f_A}(n)$ 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 3.2. *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 $f_A^{\vee}(n)$), т. к. читателю уже известно, что запись $O(g(n))$ обозначает класс функций, имеющих скорость роста *не более чем* функция $g(n)$ с точностью до положительной константы, а из введенных обозначений следует, что: $f_A^{\vee}(n) \le \overline{f_A}(n) \le f_A^{\wedge}(n)$.

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 механизм реализации модели вычислений, выполняя операции, заданные алгоритмом, перево-

дит 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, имеем:

Определение 3.3. *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: $V_A^{\wedge}(n)$, $V_A^{\vee}(n)$, $\overline{V_A}(n)$. На этой основе мы можем определить емкостную сложность алгоритма.

Определение 3.4. *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 как лучший, худший, так и средний случай требуемых ресурсов при фиксированной размерности:

Определение 3.5. *Ресурсная характеристика алгоритма*.

Под ресурсной характеристикой алгоритма в худшем, среднем или лучшем случае, будем понимать упорядоченную пару функций — соответствующие рассматриваемому случаю функция трудоемкости и функция объема памяти. Ресурсную характеристику алгоритма (**R**esource c**h**aracteristic) будем обозначать через $\Re_h^*(A)$:

$$\Re_h^*(A) = \left\langle f_A^*(n), V_A^*(n) \right\rangle, \text{ где } * \in \left\{ ^{\wedge}, ^{\vee}, ^{-} \right\}.$$

Определение 3.6. *Ресурсная сложность алгоритма*.

Под ресурсной сложностью алгоритма (**R**esource **c**omplexity) 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 $\Re_c^*(A)$**:**

$$\Re_c^*(A) = \langle O(g(n)), O(h(n)) \rangle,$$

где

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

или

$$\Re_c^*(A) = \langle \Theta(g1(n)), \Theta(h1(n)) \rangle,$$

где:

$$g1(n):\ f_A^*(n) = \Theta(g1(n)),\ h1(n):\ V_A^*(n) = \Theta(h1(n)),\ * \in \left\{ ^{\wedge}, ^{\vee}, ^{-} \right\}.$$

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 $f_A^{\wedge}(n) = \Theta\left(n^2\right)$ [3.2], а требуемая дополнительная память фиксирована, и не зависит от длины сортируемого массива. Для этого алгоритма $\Re_c^{\wedge}(A) = \left\langle \Theta\left(n^2\right), \Theta(1) \right\rangle$.

3.2. Функции ресурсной эффективности компьютерных алгоритмов и их программных реализаций

Компоненты функции ресурсной эффективности программной реализации алгоритма. Пусть D_A — конкретное множество исходных данных алгоритма (см. § 3.1). Определим в терминах требуемых алгоритмом ресурсов компьютера следующие компоненты, значения которых возможно зависят от характеристик множества D_A или некоторых его элементов [3.3]:

— $V_{exe}(D_A)$ — 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, для одной и той же задачи, короткие по объему реализующего программного кода алгоритмы имеют худшие временные оценки, чем более длинные («быстрые алгоритмы являются в большинстве случаев достаточно сложными» [3.1]). 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 [3.1, 3.6]. В качестве примера можно привести быстрый алгоритм сортировки методом индексов [3.2], требующий дополнительной памяти в объеме, рав-

ном 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 каком подходе, итерационном или рекурсивном, 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 $\bar{t}_{on}$ 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 — $f_A^{\wedge}(n)$. В большинстве случаев может быть использована функция трудоемкости в среднем — $\overline{f}_A(n)$. Средневзвешенное время $\bar{t}_{on}$ может быть получено экспериментальным путем в среде реализации алгоритма. Подробно этот вопрос рассматривается в § 3.4.

Функции ресурсной эффективности алгоритма и его программной реализации. Одним из возможных подходов к построению ресурсных функций является стоимостной подход, приводящий к введению весовых коэффициентов. При-

меняя его в аддитивной форме, получаем функцию ресурсной эффективности алгоритма для входа D в виде:

$$\Psi_A(D) = C_V \cdot V_A(D) + C_f \cdot f_A(D), \qquad (3.2.1)$$

и функцию ресурсной эффективности программной реализации

$$\Psi_{AR}(D) = C_{exe} \cdot V_{exe}(D) + C_{ram} \cdot V_{ram}(D) + C_{st} \cdot V_{st}(D) + C_t \cdot T_A(D), \qquad (3.2.2)$$

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 $A = \{ A_i \mid i = \overline{1, m} \}$, тогда

$$A_r(D) : \Psi_{AR_r}(D) = \min_{A_i} \{ \Psi_{AR_i}(D) \}, \qquad (3.2.3)$$

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 (3.2.3) 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 реальный 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 § 3.1, получаем

$$\Psi^*_A(n) = C_V \cdot V^*_A(n) + C_f \cdot f^*_A(n), \qquad (3.2.4)$$

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

$$\Psi_{AR}(n) = C_{exe} \cdot V_{exe}(n) + C_{ram} \cdot V_{ram}(n) + C_{st} \cdot V_{st}(n) + C_t \cdot T_A(n), \qquad (3.2.5)$$

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

$$R(n) = r,\ r \in \{1, m\};\ r = \arg \min_{i=\overline{1,m}} \{\Psi_{AR_i}(n)\}. \qquad (3.2.6)$$

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

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

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

тем самым алгоритм с номером k является рациональным по ресурсной функции (3.2.5) на всем сегменте размерностей входа $[n_1, n_2]$;

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

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

тем самым несколько алгоритмов являются рациональными по ресурсной функции (3.2.5) 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 $R(n_i) = const$, 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

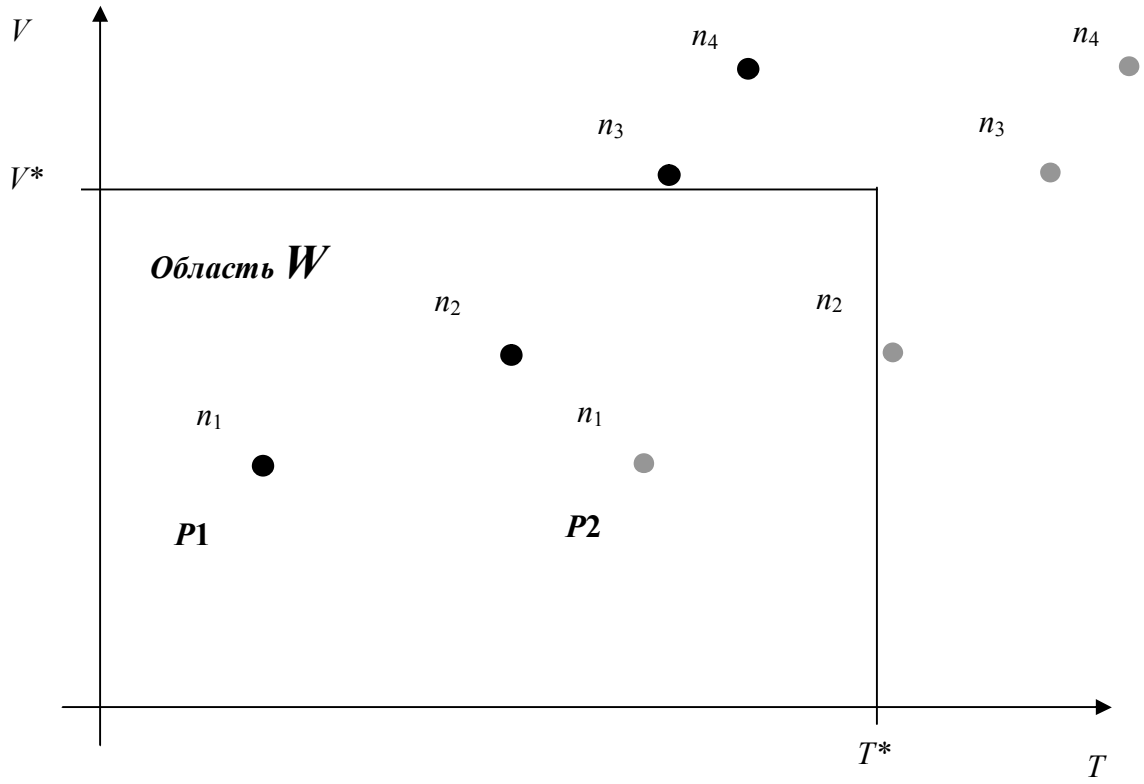
$$V(n) = V_{exe}(n) + V_{ram}(n) + V_{st}(n).$$

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

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

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 рисунке 3.1 показано возможное определение верхней границы размерности задачи в области W. Черные и серые точки на рисунке 3.1 соответствуют различным компьютерам — $P1$ и $P2$ соответственно, причем $P1$ является более производительным, чем $P2$.

Рисунок 3.1. Определение верхней границы размерности задачи с учетом



ресурсных ограничений для различных компьютеров.

Очевидно, что с возрастанием размерности задачи требуемая оперативная память и время выполнения программной реализации алгоритма, по крайней мере, не уменьшаются. На рисунке 3.1 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.

3.3. Способы построения комплексных критериев оценки качества алгоритмов

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 компонент, 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 [3.2], 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. Точность для оптимизационных алгоритмов, понимаемая не как оценка точности метода, лежащего в основе алгоритма, а как точность, обеспечиваемая процессом итерационной сходимости, задается входным значением ε, которое 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 — $T = T(n, \varepsilon)$. Возможно, что и дополнительные затраты памяти также зависят от точности, если по каким-либо причинам в ходе итерационного процесса 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

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

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

Количественная мера информационной чувствительности определяется в [3.4] на основе исследования алгоритма методами математической статистики. Более подробно понятие информационной чувствительности будет рассмотрено в

главе 4. Обозначив, следуя [3.4], количественную меру информационной чувствительности алгоритма через $\delta_i(n)$ и связав с ней соответствующий стоимостной коэффициент, получаем дополнительный компонент функции ресурсной эффективности в виде: $C_\delta \cdot \delta_i(n)$. 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,

$$C_{ram} = C_{ram}(V_{ram}(n)).$$

Таким образом, стоимостные коэффициенты являются сложной функцией от 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 рисунке 3.2.

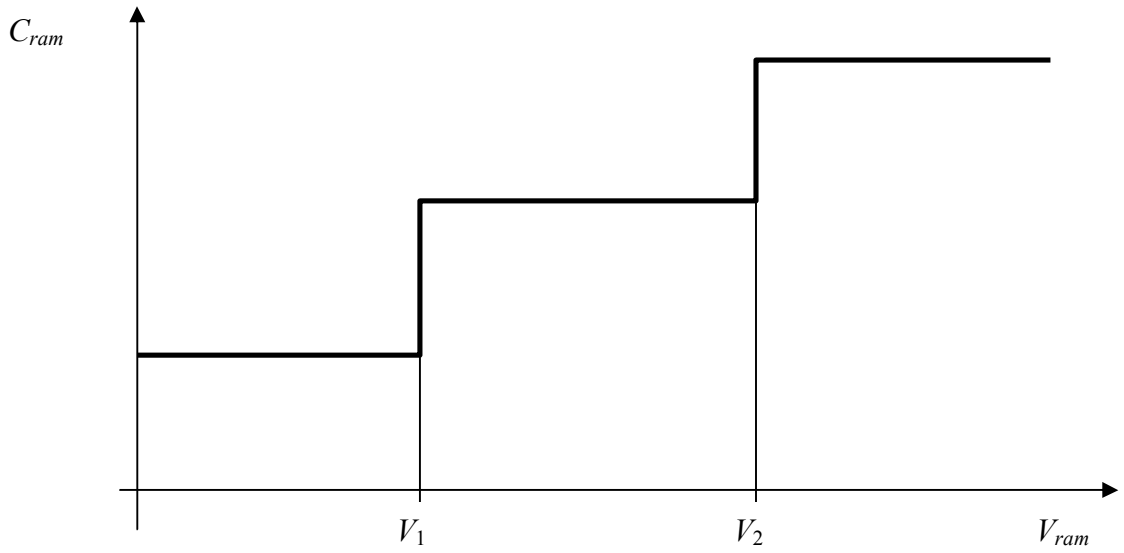


Рисунок 3.2. Коэффициент C_{ram}, заданный ступенчатой функцией.

Показанные на рисунке 3.2 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. В предположении, что значение $C_{ram\,0}=0$, коэффициенты могут быть заданы в виде

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

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.

3.4 Временные оценки и прогнозирование времён выполнения программных реализаций на основе функции трудоемкости

Проблемы теоретического построения временных оценок. Результаты сравнительного анализа алгоритмов по времени выполнения их программных реализаций не всегда совпадают с результатами анализа по функции трудоемкости, из-за различной частотной встречаемости операций и различий во времени их выполнения, связанных с особенностями компилятора, операционной системы и архитектуры компьютера. Простое экспериментальное решение задачи построения временных характеристик — эксперимент с программной реализацией и последующая экстраполяция полученных значений. Этот подход не обладает достаточной точностью из-за отсутствия информации о свойствах реализуемого алгоритма, что приводит к необходимости учета функции трудоемкости при решении задачи прогнозирования. С целью получения более достоверных результатов необходим переход от функции трудоемкости к функции времени выполнения программной реализации алгоритма. При этом в соответствии с определением временной эффективности (см. ISO 14756 [3.7]) будем рассматривать средние и/или худшие случаи для фиксированной размерности задачи:

$$T_A(n) = T_A(f_A(n), r_1, r_2, \ldots, r_k), \qquad (3.4.1)$$

где: $T_A(n)$ — функция временной эффективности, задающая время выполнения программной реализации алгоритма в среднем или худшем случае; $f_A(n)$ — функция трудоемкости алгоритма в среднем или худшем случае; $r_i, i = \overline{1,k}; k \le n$ — некоторые параметры, учитывающие особенности среды реализации.

На пути теоретического построения функции $T_A(n)$ мы сталкиваемся с рядом факторов, связанных с особенностями среды реализации (язык программирования, компилятор, операционная система, компьютер), учет которых вызывает существенные трудности. Укажем наиболее значимые из них:

— частичное несоответствие структур и конструкций языка программирования, выбранного для реализации алгоритма, особенно в части структур и типов данных, и реальной системы команд процессора и поддерживаемых им типов

данных — проблема, известная в области архитектур процессоров под названием семантического разрыва [3.8];

— наличие архитектурных особенностей компьютера, и, прежде всего, процессора, существенно влияющих на наблюдаемое время выполнения программы, таких как стековая обработка, конвейер команд и конвейер данных, наличие нескольких уровней быстрых буферов памяти (кеш-память), аппаратные и программные средства предвыборки команд и данных, и т. д. [3.9];

— различие во временах выполнения различных машинных команд, обусловленное различной сложностью внутренних алгоритмов реализации аппаратных вычислений. Отметим, что эта проблема частично снимается для современных RISC процессоров, в которых большинство машинных команд выполняется за фиксированное количество тактов [3.9];

— различие времени реального выполнения однородных команд в зависимости от типов данных, причем это различия могут достигать порядка для коротких целых в сравнении с длинными действительными типами [3.9];

— различие во времени выполнения одной команды, в зависимости от значений операндов. Такие различия могут быть усреднены, но если особенности формирования входных данных алгоритма устойчивы, то такие различия так же должны учитываться;

— неоднозначность компиляции исходного текста, обусловленная как спецификой выбранного компилятора, например, в части поддержки им различных методов оптимизации кода, так и особенностями его настройки;

— дополнительные временные задержки, обусловленные выбранной средой реализации, такие как время оверлейной подкачки фрагментов программы, задержки наблюдаемого времени, вносимые операционной системой, например задержки, связанные с квантованием времени задач и т. д.

Перечисленные факторы существенно затрудняют теоретическое построение функции времени выполнения, тем не менее, попытки различного подхода к их учету привели к появлению разнообразных методов построения временных оценок.

Метод типовых задач. Это один из первых методов, представляющий собой попытку получить универсальный подход к получению временных оценок, учитывающий только тип решаемой задачи. Идея метода состоит в том, что в рамках фиксированного типа задач, например задач, связанных с научно-техническими расчетами, среднее время на одну обобщенную операцию будет устойчиво из-за использования одинаковых типов данных и близкой частотной встречаемости операций. Метод предполагает проведение совокупного анализа исследуемого алгоритма по трудоемкости и переход к временной оценке его программной реализации на основе принадлежности решаемой задачи к одному из следующих типов: задачи научно-технического характера с преобладанием операций с операндами действительного типа; задачи дискретной математики с преобладанием операций с операндами целого типа; задачи баз данных с преобладанием операций с операндами строкового типа.

Далее на основе анализа множества реальных программ для решения соответствующих типов задач определяется частотная встречаемость операций. Полученные результаты лежат в основе создания соответствующих тестовых программ, порождающих заданную частотную встречаемость операций для заданных типов данных. На основе экспериментов с этими тестовыми программами и определяется среднее время на обобщенную операцию в данном типе задач — $\bar{t}_{\text{тип задачи}}$, далее оценивается общее время работы программной реализации алгоритма для данного компьютера и данной среды реализации в виде:

$$\overline{T_A}(n) = \overline{f_A}(n) \cdot \bar{t}_{\text{тип задачи}}.$$

Преимущество метода состоит в том, что значение $\bar{t}_{\text{тип задачи}}$ определяется однократно для данного языка, компилятора, ОС и компьютера, и затем используется для получения временных оценок программных реализаций алгоритмов на основе принадлежности решаемой задачи к одному из указанных типов. Такие оценки, очевидно, не обладают большой точностью, но могут быть достаточно просто получены для любой программной реализации исполняемой на данном компьютере.

Метод пооперационного анализа. Идея метода пооперационного анализа состоит в представлении функции трудоемкости алгоритма в виде суммы трудоёмкостей по базовым операциям и определения среднего времени выполнения каждой из них в выбранной среде реализации. Метод пооперационного анализа включает в себя следующие этапы:

— получение пооперационных функций трудоемкости — $\overline{f_{A\ \text{оп}\,i}}(n)$ для каждой из используемых алгоритмом базовых операций с учетом типов данных. Получение таких пооперационных функций требует определенных затрат, однако, в некоторых случаях такой подход позволяет получить значимые результаты по выбору рациональных алгоритмов;

— экспериментальное определение среднего времени выполнения данной базовой операции — $\bar{t}_{\text{оп}\,A\,i}$ на конкретной вычислительной машине в среде выбранного языка программирования и операционной системы с помощью специальной тестовой программы;

— ожидаемое время выполнения программной реализации рассчитывается как сумма произведений пооперационной трудоемкости на средние времена операций:

$$\overline{T}_A(n) = \sum_{i=1}^{k} \bar{t}_{\text{оп}\,A\,i} \cdot \overline{f_{A\ \text{оп}\,i}}(n), \qquad (3.4.2)$$

где: k — количество базовых операций; $\overline{f_{A\ \text{оп}\,i}}(n)$ — функция трудоемкости алгоритма по операции $i, i = \overline{1,k}$; $\bar{t}_{\text{оп}\,A\,i}$ — экспериментальное среднее время выполнения операции i.

Ошибки прогноза в данном методе, несмотря на его достаточную детальность, связаны с тем, что реальный поток операций, порождаемых алгоритмом, не всегда совпадает с потоком в экспериментальной программе, определяющей среднее время выполнения базовой операции. Такие расхождения могут обуславливаться, например, конвейерной архитектурой процессора, наличием блока предвыборки команд, внутренними алгоритмами управления кэш-памятью и другими особенностями архитектуры. Однако в ряде случаев только применение метода пооперационного анализа позволяет выявить тонкие аспекты и особенности

рационального применения того или иного алгоритма решения задачи. Применение этого метода для анализа алгоритма умножения комплексных чисел будет проиллюстрировано в главе 10.

Экспериментальный метод получения временных оценок на основе функции трудоемкости. На основе функции трудоемкости алгоритма и ряда экспериментов с его программной реализацией возможно получение временных оценок на основе среднего времени выполнения обобщенной базовой операции. Метод детально разработан автором [3.10], и представляет собой совокупность следующих этапов:

— совокупный анализ трудоемкости алгоритма без разделения на операции; на этом этапе определяется $\overline{f_A}(n)$ в обобщенных базовых операциях принятой модели вычислений;

— проведение вычислительных экспериментов с программной реализацией алгоритма; на этом этапе для каждого из выбранных значений размерности задачи $n_i, i=\overline{1,m}$ проводится определенное количество экспериментов $n_э, j=\overline{1,n_э}$ с разными исходными данными, в ходе которых определяются времена выполнения — $t_{э\,j}(n_i)$, на основе которых рассчитывается среднее экспериментальное время выполнения:

$$\overline{t_э}(n_i)=\left(\frac{1}{n_э}\right)\cdot\sum_{j=1}^{n_э}t_{э\,j}(n_i). \qquad (3.4.3)$$

— расчет среднего времени; в рамках этого этапа на основе известной функции трудоемкости алгоритма в среднем $\overline{f_A}(n_i)$ рассчитывается среднее время на обобщенную базовую операцию, порождаемое данным алгоритмом, компилятором, операционной средой и компьютером для данной размерности — $\bar{t}_{оп\,A}(n_i)$, и по всему эксперименту в целом — $\bar{t}_{оп\,A\,э}$:

$$\bar{t}_{оп\,A}(n_i)=\frac{\overline{t_э}(n_i)}{\overline{f_A}(n_i)},\ \bar{t}_{оп\,A\,э}=\frac{1}{m}\sum_{i=1}^{m}\bar{t}_{оп\,A}(n_i), \qquad (3.4.4)$$

где m — количество тестируемых значений размерности задачи.

— прогноз временной эффективности; на этом этапе в предположении об устойчивости среднего времени выполнения обобщенной базовой операции, по-

лученные результаты могут быть интерполированы или экстраполированы на другие значения размерности задачи:

$$\overline{T_A}(n) = \bar{t}_{\text{оп } A\,\text{э}} \cdot \overline{f_A}(n), \quad n \notin n_i, i = \overline{1,m}. \tag{3.4.5}$$

В качестве примера приведем экспериментальные данные, полученные для алгоритма сортировки методом прямого включения (реализация: язык Паскаль, Windows 98, ASUS COUPLE-VM, PIII-633 Мгц). Тестовые варианты были получены стандартным генератором псевдослучайных чисел с равномерным распределением. Для каждого фиксированного значения размерности массива выполнялось $n_{\text{э}} = 10^6$ экспериментов. Полученные результаты приведены в таблице 3.1. Здесь: t_{sum} — время в секундах по всем экспериментам; $t_{\text{э}}(n_i)$ — среднее время в секундах на сортировку одного массива; $\bar{t}_{\text{оп } A}(n_i)$ — среднее время на обобщенную базовую операцию в наносекундах.

Таблица 3.1

i	n_i	$\overline{f_A}(n)$	t_{sum}	$t_{\text{э}}(n_i)$	$\bar{t}_{\text{оп } A}(n_i)$
Экспериментальные результаты для определения $\bar{t}_{\text{оп } A\,\text{э}}$					
1	50	6 815	17,51	0,00001751	2,5705
2	60	9 680	24,45	0,00002445	2,5266
3	70	13 045	32,52	0,00003252	2,4935
4	80	16 910	41,80	0,00004180	2,4723
5	90	21 275	52,30	0,00005230	2,4586
6	100	26 140	63,88	0,00006388	2,4440
7	200	102 290	244,90	0,00024490	2,3942
				$\bar{\bar{t}}_{\text{оп } A\,\text{э}}$	2,4800
Экспериментальные и прогнозируемые результаты для ***n***=300, 400					
			Прогноз	Эксперимент	Ошибка
8	300	228 440	0,00056652	0,00054330	4,274%
9	400	404 590	0,00100337	0,00095690	4,856%

Точность прогноза может быть повышена за счет выяснения зависимости $\bar{t}_{\text{оп}\,A}(n_i)$ от размерности задачи, экспериментальные данные (таблица 3.1.) показывают, что такая зависимость существует.

Анализ влияния размерности задачи на среднее время выполнения обобщенной базовой операции. Рассмотрим, следуя [3.10], теоретический аспект поставленной задачи — выяснение природы и функционального вида зависимости $\bar{t}_{\text{оп}\,A}(n)$. Может быть предложено следующее качественное объяснение наблюдаемой в эксперименте зависимости $\bar{t}_{\text{оп}\,A\,\text{э}}$ от размерности, учитывающее тот факт, что для больших значений n среднее время стремится к константе. Для большинства алгоритмов фрагмент, обуславливающий в оценке по среднему главный порядок функции трудоемкости, обладает несколько иной частотной встречаемостью операций, чем фрагменты, порождающие асимптотически более слабые порядки в $\overline{f_A}(n)$. Это, в свою очередь, приводит к тому, что в области «малых» размерностей совокупное среднее время на операцию будет отличаться от среднего времени в области «больших» размерностей. Поскольку практически каждый алгоритм содержит конструкции следования, то функция трудоемкости носит аддитивный характер и, следовательно, может быть представлена в виде:

$$\overline{f_A}(n)=\sum_{i=0}^{k}\overline{f_i}(n),\quad \overline{f_0}(n)=const, \tag{3.4.6}$$

где k — количество функциональных аддитивных компонент функции трудоемкости. Компоненты этой суммы $\overline{f_i}(n)$ могут быть упорядочены по асимптотической иерархии:

$$\overline{f_i}(n)\succ\overline{f_{i-1}}(n),\quad \forall\, i=\overline{1,k}\,, \tag{3.4.7}$$

при этом, очевидно, что $\overline{f_k}(n)$ есть компонент главного порядка функции трудоемкости алгоритма, определяющий его сложность в предположении об асимптотической иерархии $\overline{f_i}(n)$:

$$\Theta\left(\overline{f_A}(n)\right)=\Theta\left(\overline{f_k}(n)\right).$$

Поведение $\overline{f_A}(n)$ в области малых размерностей определяется не только $\overline{f_k}(n)$, но и более «слабыми» асимптотическими компонентами, что приводит к необходимо-

сти учета их влияния на значения $\overline{f_A}(n)$, и, следовательно, на временные оценки. При условии, что в рамках области применения $(n_{\min} \le n \le n_{\max})$ выполняется соотношение:

$$\overline{f_{k-2}}(n_{\min}) << \overline{f_{k-1}}(n_{\min}),$$

в целях практического анализа можно ограничиться рассмотрением основных компонент функции $\overline{f_A}(n)$, а именно $\overline{f_k}(n)$ и $\overline{f_{k-1}}(n)$. Пусть $\overline{t_i}$ — есть среднее время на обобщенную базовую операцию для i-ой компоненты функции трудоемкости, тогда среднее время на обобщенную базовую операцию для алгоритма в целом имеет вид:

$$\bar{t}_{\text{оп } A}(n) = \sum_{i=0}^{k} \overline{t_i} \cdot \overline{f_i}(n) \Big/ \sum_{i=0}^{k} \overline{f_i}(n), \ \overline{f_A}(n) = \sum_{i=0}^{k} \overline{f_i}(n). \qquad (3.4.8)$$

Таким образом, мы получаем средневзвешенную оценку, в которой компоненты $\overline{t_i}$ имеют веса, равные слагаемым в функции трудоемкости (при фиксированном значении n). Рассмотрение только двух компонент — k-ой и $(k-1)$-ой приводит к:

$$\bar{t}_{\text{оп } A}(n) \approx \frac{\left(\overline{t_k} \cdot \overline{f_k}(n) + \overline{t_{k-1}} \cdot \overline{f_{k-1}}(n)\right)}{\left(\overline{f_k}(n) + \overline{f_{k-1}}(n)\right)},$$

и, вынося $\overline{t_k}$ за скобки, имеем:

$$\bar{t}_{\text{оп } A}(n) \approx \overline{t_k} \cdot \left\{ \frac{\overline{f_k}(n)}{\overline{f_k}(n) + \overline{f_{k-1}}(n)} + \frac{\overline{t_{k-1}}}{\overline{t_k}} \cdot \frac{\overline{f_k}(n)}{\overline{f_k}(n) + \overline{f_{k-1}}(n)} \right\}. \qquad (3.4.9)$$

В силу предположения об асимптотической иерархии компонент функции трудоемкости полученная формула теоретически объясняет зависимость среднего времени на обобщенную базовую операцию от размерности задачи. Более того, из (3.4.8) и условия (3.4.7) следует, что:

$$\lim_{n \to \infty} \bar{t}_{\text{оп } A}(n) = \overline{t_k},$$

и, следовательно, в области больших размерностей $\bar{t}_{\text{оп } A}(n)$ слабо зависит от значения n.

Очевидно, что конкретный вид зависимости $\bar{t}_{\text{оп } A}(n)$ от размерности задачи определяется конкретными компонентами функции трудоемкости алгоритма. Будем предполагать далее, что функция трудоемкости алгоритма известна, по край-

ней мере, с точностью до коэффициентов у двух асимптотически главных слагаемых функциональной суммы (3.4.6). Рассмотрим характерный пример для алгоритма с полиномиальной сложностью:

Пусть функция трудоемкости алгоритма имеет вид:

$$\overline{f_A}(n) = a_2 \cdot n^2 + a_1 \cdot n + a_0,\ k = 2.$$

Будем считать, что в рамках области применения алгоритма значение размерности таково, что $a_1 \cdot n_{\min} >> a_0$, и, следовательно, влияние a_0 можно не учитывать. С учетом введенных обозначений для средних времен операций в компонентах — $\overline{t_i}$ запишем оценку временной эффективности программной реализации:

$$\overline{t_A}(n) = \overline{t_2} \cdot a_2 \cdot n^2 + \overline{t_1} \cdot a_1 \cdot n,$$

и введем дополнительно следующие обозначения:

$$\alpha = \frac{\overline{t_1}}{\overline{t_2}},\quad \overline{t_1} = \alpha \cdot \overline{t_2},\ \beta = \frac{a_1}{a_2},\quad a_1 = \beta \cdot a_2,$$

тогда:

$$\overline{t_A}(n) = \overline{t_2} \cdot a_2 \cdot n^2 + \overline{t_2} \cdot \alpha \cdot \beta \cdot a_1 \cdot n = \overline{t_2} \cdot a_2 \cdot n \cdot (n + \alpha \cdot \beta),$$

$$\overline{f_A}(n) = a_2 \cdot n^2 + a_1 \cdot n = a_2 \cdot n^2 + \beta \cdot a_2 \cdot n = a_2 \cdot n \cdot (n + \beta).$$

Функция среднего времени на обобщенную базовую операцию в соответствии с формулами (3.4.8), (3.4.9) и введенными обозначениями имеет вид:

$$\bar{t}_{\text{оп } A}(n) = \bar{t}_2 \cdot \frac{n + \alpha\beta}{n + \beta}, \qquad (3.4.10)$$

очевидно, что: $\lim\limits_{n \to \infty} \bar{t}_{\text{оп } A}(n) = \bar{t}_2$. Для выяснения характера поведения $\bar{t}_{\text{оп } A}(n)$ выполним следующие преобразования:

$$\bar{t}_{\text{оп } A}(n) = \bar{t}_2 \cdot \frac{n + \alpha\beta + \beta - \beta}{n + \beta} = \bar{t}_2 \cdot \left(1 + \frac{\alpha \cdot \beta - \beta}{n + \beta}\right) = \bar{t}_2 \cdot \left(1 + \frac{\beta \cdot (\alpha - 1)}{n + \beta}\right),$$

тогда, если значение $\alpha > 1$, т. е. среднее время на операцию в линейной компоненте больше, чем в квадратичной, то функция $\bar{t}_{\text{оп } A}(n)$ будет уменьшаться с ростом n, асимптотически стремясь к $\bar{t}_2$ сверху. Если $\alpha < 1$, то функция $\bar{t}_{\text{оп } A}(n)$ будет увеличиваться с ростом n, асимптотически стремясь к $\bar{t}_2$ снизу, как это показано на рисунке 3.3.

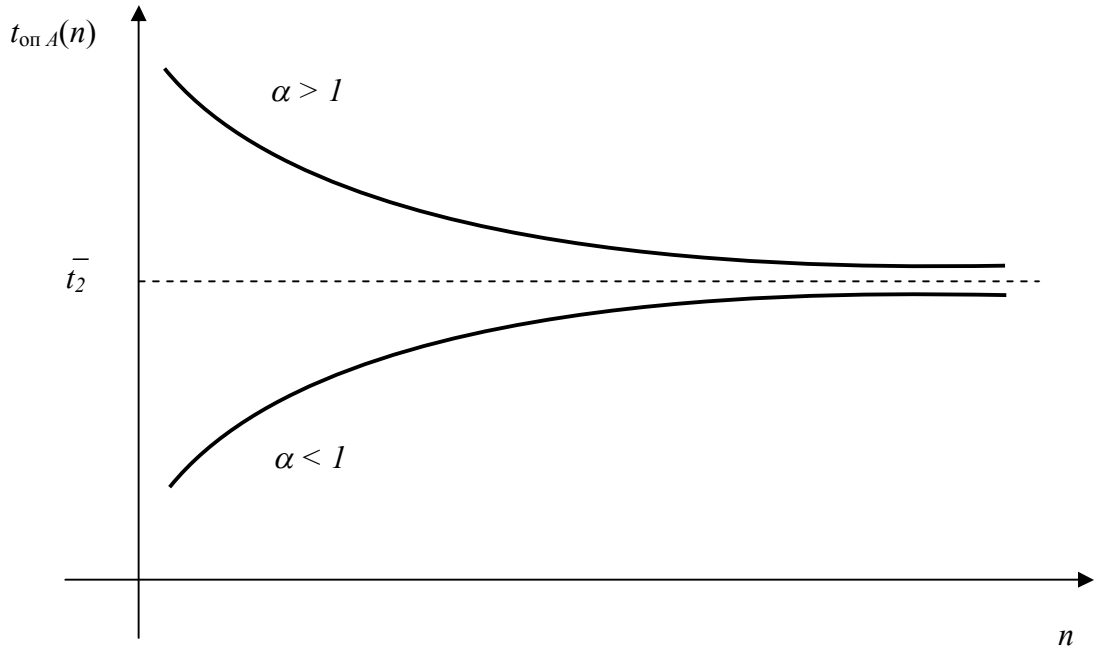


Рис. 3..3. Вид функции $\bar{t}_{\text{оп }A}(n)$ в зависимости от значения α.

Для построения реального прогноза временной эффективности алгоритма необходимо определить числовые значения $\bar{t}_2$ и α на основе экспериментальных данных. Заметим, что значение β известно из функции трудоемкости алгоритма, полученной в результате анализа, а экспериментальный ряд значений размерностей и средних времен — $\{n_i\}, \{\bar{t}_{\text{оп }A}(n_i)\}, i = \overline{1,m}$ уже получен при определении среднего времени выполнения обобщенной базовой операции.

Могут быть предложены следующие два варианта линеаризации функции $\bar{t}_{\text{оп }A}(n)$, использующие линейный регрессионный анализ [3.11] для получения значений $\bar{t}_2$ и α на основе экспериментальных данных:

Вариант 1. Преобразуем функцию $\bar{t}_{\text{оп }A}(n)$ к виду:

$$\bar{t}_{\text{оп }A}(n) = \bar{t}_2 + \bar{t}_2 \cdot \beta \cdot (\alpha - 1) \cdot \frac{1}{n+\beta}.$$

Для линеаризации введем замену переменных:

$$x = \frac{1}{n+\beta}, \quad y = \bar{t}_{\text{оп }A}(n),$$

эту замену будем называть линейно-гиперболической. Отметим, что значение β известно из функции трудоемкости, и, следовательно, значения x могут быть вычислены априорно для экспериментальных значений размерности задачи n_i. Обо-

значая через b значение $\bar{t}_2$, а через k значение $\bar{t}_2 \cdot \beta \cdot (\alpha - 1)$ окончательно получаем $y = k \cdot x + b$. Используя метод наименьших квадратов можно построить линейную регрессию по экспериментальному ряду значений, и определить оптимальные значения k^* и b^*. После чего принимаем $\bar{t}_2 = b^*$, а отношение средних времен α вычисляется исходя из введенного обозначения для k по формуле:

$$\alpha^* = 1 + \frac{k^*}{b^* \cdot \beta}.$$

Вариант 2. В этом варианте линеаризации будем использовать представление $\bar{t}_{\text{оп } A}(n)$ в виде (3.4.10)

$$\bar{t}_{\text{оп } A}(n) = \bar{t}_2 \cdot \frac{n + \alpha\beta}{n + \beta},$$

тогда, логарифмируя, имеем

$$\ln\left(\bar{t}_{\text{оп } A}(n)\right) = \ln\left(\bar{t}_2\right) + \ln(n + \alpha \cdot \beta) - \ln(n + \beta),$$

преобразуя выражения

$$n + \alpha \cdot \beta = n \cdot \left(1 + \frac{\alpha \cdot \beta}{n}\right), \quad n + \beta = n \cdot \left(1 + \frac{\beta}{n}\right),$$

получаем

$$\ln\left(\bar{t}_{\text{оп } A}(n)\right) = \ln\left(\bar{t}_2\right) + \ln(n) + \ln\left(1 + \frac{\alpha \cdot \beta}{n}\right) - \ln(n) - \ln\left(1 + \frac{\beta}{n}\right).$$

В предположении, что область реальных размерностей такова, что значение $\frac{1}{n^2}$ достаточно мало, и используя асимптотическое представление для $\ln(1 + x)$, при $x \to 0$: $\ln(1 + x) \approx x + O\left(x^2\right)$ окончательно получаем:

$$\ln\left(\bar{t}_{\text{оп } A}(n)\right) = \ln\left(\bar{t}_2\right) + \frac{\alpha \cdot \beta}{n} - \frac{\beta}{n} = \ln\left(\bar{t}_2\right) + \frac{\beta}{n} \cdot (\alpha - 1).$$

После введения замен:

$$x = \frac{\beta}{n}, \quad y = \ln\left(\bar{t}_{\text{оп } A}(n)\right),$$

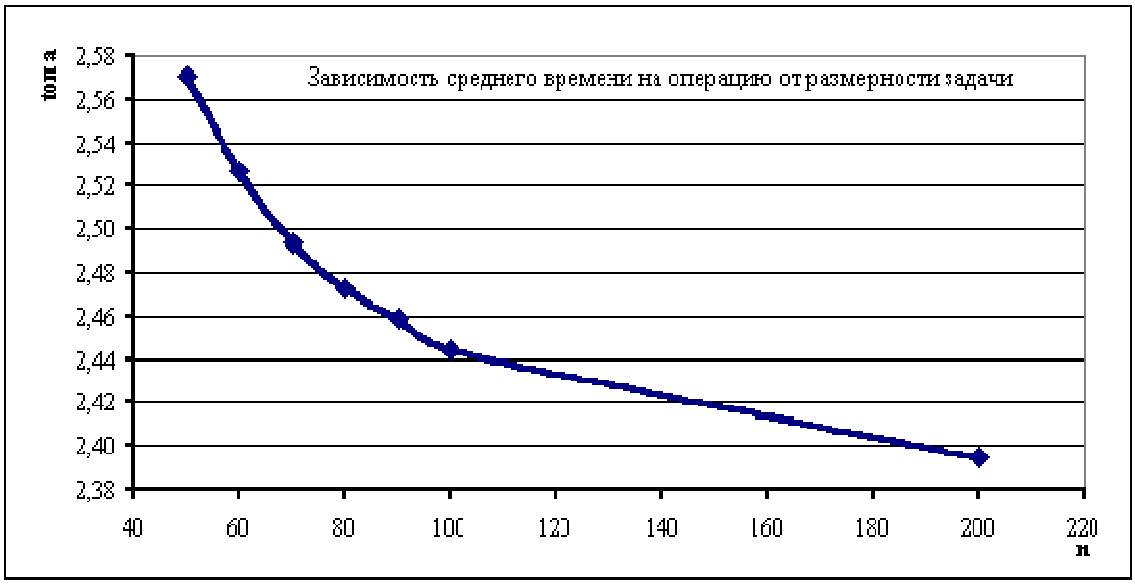
эту замену будем называть логарифмически-гиперболической, имеем линейную форму:

$$y = k \cdot x + b, \quad \text{где} \quad b = \ln\left(\bar{t}_2\right), \quad k = (\alpha - 1).$$

Получая, аналогично варианту 1 оптимальные по минимуму суммарного квадратичного отклонения значения k^* и b^*, можно определить $\bar{t}_2 = e^{b^*}$ и $\alpha = k^* + 1$. Отметим, что данный вариант линеаризации использует предположение о малости значения $\frac{1}{n^2}$ в области реальных размерностей задачи.

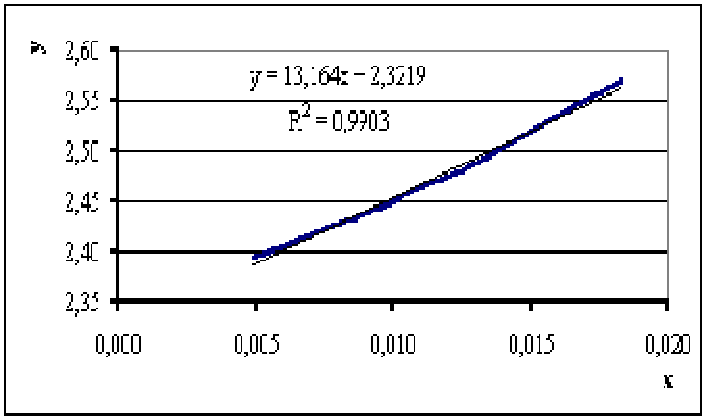
Изложенный подход может быть обобщен на случай, когда аддитивная функция трудоемкости представляет собой полином k - ой степени с известными коэффициентами, и на случай, когда некоторые аддитивные компоненты функции трудоемкости имеют логарифмические или полилогарифмические множители (подробнее см. в [3.10]).

Пример прогноза времени выполнения. Приведем пример применения описанного метода для прогноза времени выполнения программной реализации алгоритма сортировки вставками. На рисунке 3.4 приведены результаты анализа экспериментальных данных для определения коэффициентов функции $\bar{t}_{\text{оп }A}(n)$, вид которой определяется формулой (3.4.10) для двух предложенных вариантов линеаризации. В таблице 3.2 приведены результаты и прогнозы времени выполнения для программной реализации алгоритма сортировки вставками, которые получены на основе функции трудоемкости и вычисленных коэффициентов зависимости среднего времени обобщенной операции от размерности задачи. Приведенные данные свидетельствуют о значительном повышении точности прогноза по сравнению с использованием среднего времени обобщенной базовой операции.



Вариант 1 Y=K*X + B

K	13,164342
B	2,321930
t_2	2,321930
alfa	2,232515



Вариант 2 Y=K*X + B

K	1,034124
B	0,847156
t_2	2,333001
alfa	2,034124

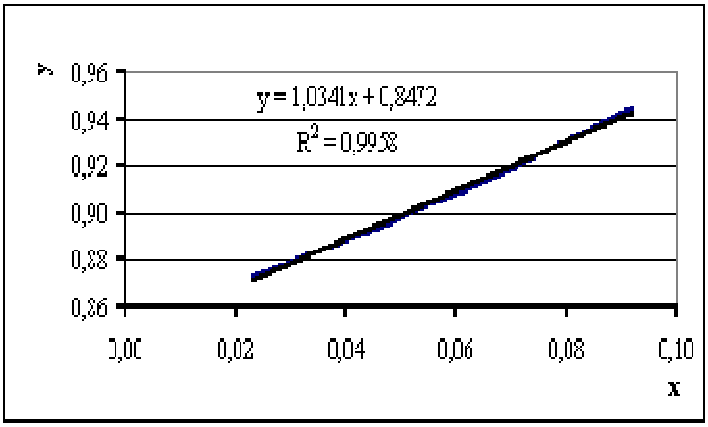


Рис. 3.4. Определение коэффициентов функции $\bar{t}_{\text{оп } A}(n)$ для программной реализации алгоритма сортировки вставками для двух вариантов линеаризации.

Таблица 3.2

i	n_i	$\overline{f_A}(n)$	t_{sum}	$t_э(n_i)$	$\bar{t}_{оп\,A}(n_i)$
Экспериментальные результаты					
1	50	6 812	17,51	0,00001751	2,5705
2	60	9 677	24,45	0,00002445	2,5266
3	70	13 042	32,52	0,00003252	2,4935
4	80	16 907	41,80	0,00004180	2,4723
5	90	21 272	52,30	0,00005230	2,4586
6	100	26 137	63,88	0,00006388	2,4440
7	200	102 287	244,90	0,00024490	2,3942
Прогнозируемые результаты — линеаризация по варианту 1					
	β	4,60		t_2	2,321930
	α	2,232515			
			Прогноз	Эксперимент	Ошибка
	300	228 437	0,00054029	0,00054330	-0,554%
	400	404 587	0,00095259	0,00095690	-0,451%
Прогнозируемые результаты — линеаризация по варианту 2					
	β	4,60		t_2	2,333001
	α	2,034124			
			Прогноз	Эксперимент	Ошибка
	300	228 437	0,00054127	0,00054330	-0,374%
	400	404 587	0,00095500	0,00095690	-0,199%

На основании результатов, приведенных в таблице 3.2, можно говорить, что учёт зависимости среднего времени на обобщенную базовую операцию от размерности задачи позволил для данного алгоритма более чем на порядок уменьшить ошибку прогноза времени выполнения.

Список литературы к главе 3.

[3.1] Ахо А., Хопкрофт Дж., Ульман Дж. Структуры данных и алгоритмы: Пер. с англ.: — М.: Издательский дом «Вильямс», 2001. — 384 с.

[3.2] Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ, 2-ое издание: Пер. с англ. — М.: Издательский дом «Вильямс», 2005. — 1296 с.

[3.3] Ульянов М. В. Классификация и методы сравнительного анализа вычислительных алгоритмов. Научное издание. — М.: Издательство физико-математической литературы, 2004. — 212 с.

[3.4] Головешкин В.А., Ульянов М. В. Теория рекурсии для программистов. М.: Издательство «Наука ФИЗМАТЛИТ», 2006. — 296 с.

[3.5] Успенский В. А. Машина Поста. — М.: Наука, 1979. — 96 с.

[3.6] Вирт Н. Алгоритмы и структуры данных: Пер. с англ. –2-е изд., испр. — СПб.: Невский диалект, 2001. — 352 с.

[3.7] Сборник действующих международных стандартов ИСО серии 9000. Т-1, 2, 3. — М.: ВНИИКИ. 1998.

[3.8] Амамия М., Танака Ю. Архитектура ЭВМ и искусственный интеллект: Пер. с японск. — М.: Мир, 1993. — 400 с.

[3.9] Столлингс В. Структурная организация и архитектура компьютерных систем, 5-е изд.: Пер. с англ. — М.: Издательский дом «Вильямс», 2002. — 896 с.

[3.10] Ульянов М. В. Метод прогнозирования временных оценок программных реализаций алгоритмов на основе функции трудоемкости // Информационные технологии. 2004. № 5. С. 54–62.

[3.11] Советов Б. Я., Яковлев С. А. Моделирование систем: Учеб. для вузов — 3-е изд., перераб. и доп. — М.: Высш. шк., 2001. — 343 с.

ГЛАВА 4.

КЛАССИФИКАЦИОННЫЕ ПРИЗНАКИ И СПЕЦИАЛЬНЫЕ КЛАССИФИКАЦИИ ВЫЧИСЛИТЕЛЬНЫХ АЛГОРИТМОВ И ЗАДАЧ

Введение

Важной составной частью научного исследования некоторой совокупности объектов являются классификации, выявляющие характерные общие свойства объектов и закладывающие теоретическую базу их дальнейших исследований. В рамках создания математического и алгоритмического обеспечения программных средств и систем такой совокупностью объектов являются алгоритмы решения базовых задач, которые лежат в основе всей последующей программной разработки. Впечатляющий рост производительности современных компьютерных систем не снижает требований к алгоритмическому обеспечению, в том числе и по обеспечению временной эффективности программных реализаций. При этом обеспечение указанных в техническом задании временных требований связано, при выбранной аппаратной среде реализации, с выбором или разработкой математических методов и/или алгоритмов решения поставленных задач. Рассматривая набор известных алгоритмов решения некоторой задачи, разработчики сталкиваются с проблемой выбора алгоритма, рационального в данных условия применения. Необходимость решения этой актуальной проблемы приводит к использованию разнообразных подходов, методов и методик, и, в частности, может опираться на специальные классификации алгоритмов и задач.

Определенный вклад в более эффективное решение этой проблемы могут внести классификации алгоритмов, отражающие как сложностные оценки алгоритмов, так и влияние на эти оценки характеристических особенностей множества исходных данных, порожденных решаемой задачей. 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 [4.1]. 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 [4.1, 4.2]. Могут рассматриваться и другие специальные классификации, учитывающие определенные требования к программным средствам и системам. Например, требование временной устойчивости может быть учтено в классификации алгоритмов по информационной чувствительности.

Помимо классификаций алгоритмов, могут рассматриваться и специальные классификации задач, ориентированные на возможность дальнейшего совершенствования ресурсных характеристик алгоритмов. Рассмотрению различных классификаций алгоритмов и задач, которые могут быть использованы при решении проблемы выбора рациональных алгоритмов и посвящена настоящая глава.

4.1 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 [4.3, 4.4]. 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, достигает или не достигает наиболее эффективный из известных в настоящее время алгоритмов, доказанной нижней границы временной сложности задачи. Заметим, что определения классов P и EXP в тео-

рии алгоритмов [4.1, 4.2], также опираются на асимптотику временной сложности существующих алгоритмов решения задач в указанных классах.

Теоретическая нижняя граница 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 (включающее 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

$$A_Z = \{\, A \mid A : D \to R, D \in D_Z, R \in R_Z, Ver(D,R) = True, f_A(D) \neq \infty\ \forall D \in D_Z \},$$

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}

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

Для сравнения двух функций по асимптотике их роста будем использовать отношение $\prec$, введенное Поль-Дюбуа Раймоном [4.6]:

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

Введем содержательно 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 $f_{thlim}(n)$. 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 $f_{thlim}(n)$, 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 $g(n) \succ f_{thlim}(n)$ указанное свойство не может быть доказано. Дадим формальное определение для $f_{thlim}(n)$ [4.7]. Определим множество F_{lim}, состоящее из функций f_{lim}:

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

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 $f_{thlim}(n) = \sup\limits_{\prec} \{F_{lim}\}$.

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 $\Theta(\log_2(n!))$ [4.8]. Применяя аппроксимацию Стирлинга для $n!$, имеем

$$f_{thlim}(n) = \Theta(\log_2(n!)) = \Theta(n \cdot \log_2(n) - n \cdot \log_2 e + 0{,}5 \cdot \log_2(2\pi n)) =$$
$$= \Theta(n \cdot (\log_2(n) - \log_2 e) + 0{,}5 \cdot \log_2 n + 0{,}5 \cdot \log_2(2\pi)).$$

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 $f_{tr}(n)$ [4.7], и будем предполагать априорно, что оценка $f_{tr}(n)$ существует для задачи Z, тогда можно указать следующие возможные соотношения между оценками $f_{thlim}(n)$ и $f_{tr}(n)$ — оценка $f_{thlim}(n)$ не доказа-

на для задачи Z и для данной задачи принимается оценка $f_{tr}(n)$; оценка $f_{thlim}(n)$ доказана, тогда либо: $f_{tr}(n) \prec f_{thlim}(n)$, либо $f_{tr}(n) = f_{thlim}(n)$.

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} [4.7]. 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 начиная 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:

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

Обозначим трудоемкость этого алгоритма через $f_{Amin}(n)$.

Классификация задач по соотношению теоретической нижней границы временной сложности задачи и сложности наиболее эффективного из существующих алгоритмов. Сравнивая между собой полученные оценки для некоторой задачи — $f_{thlim}(n)$, если она существует, $f_{tr}(n)$ и $f_{Amin}(n)$, можно предложить следующую классификацию задач, отражающую соотношение оценки задачи и наиболее эффективного из известных алгоритмов ее решения [4.7]:

Класс задач $THCL(Theoretical\ close)$. Это задачи, для которых $f_{thlim}(n)$ существует и $f_{Amin}(n) = \Theta(f_{thlim}(n))$. 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 лишь преследовать цель улучшения коэффициента при главном порядке в функции трудоемкости, при этом

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

Приведем несколько примеров задач этого класса: задача поиска максимума в массиве: $f_{thlim}(n) = \Theta(n), f_{Amin}(n) = \Theta(n)$; задача сортировки массива с использованием сравнений: $f_{thlim}(n) = \Theta(n \cdot \log_2(n)), \; f_{Amin}(n) = \Theta(n \cdot \log_2(n))$ — алгоритм сортировки пирамидой [4.8]; задача умножения длинных двоичных целых чисел: $f_{thlim}(n) = \Theta(n \cdot \ln(n) \cdot \ln\ln(n)), \; f_{Amin}(n) = \Theta(n \cdot \ln(n) \cdot \ln\ln(n))$ — алгоритм Штрассена – Шенхаге [4.2].

Класс задач PROP (*Practical open*). Это задачи, для которых $f_{thlim}(n)$ существует и $f_{Amin}(n) \succ f_{thlim}(n)$. Таким образом, для задач из 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 определяет 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 [4.3, 4.4], 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 $f_{thlim}(n)$ не доказана, а $f_{Amin}(n) \succ f_{tr}(n)$. Для задачи из 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 [4.7] — отражает тот факт, что для задач этого класса имеет место теоретическая проблема либо доказательства оценки $f_{thlim}(n)$, 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]. В качестве другого непосредственного примера можно указать задачу умножения матриц, для которой $f_{Amin}(n) = O\left(n^{2,34}\right)$ [4.2], а тривиальная оценка, отражающая необходимость просмотра всех элементов исходных матриц $f_{tr}(n) = \Theta\left(n^2\right)$, при этом теоретическая нижняя граница временной сложности для этой задачи пока не доказана.

4.2 Классификация компьютерных алгоритмов на основе угловой меры асимптотического роста функций

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, разграничивающих полиномы и экспоненты, и дополнительных множеств субполиномиальных и надэкспоненциальных функций. Один из возможных вариантов решения этой задачи предложен в [4.9]. Обозначим через n длину входа алгоритма, а через $f = f(n)$ — функцию сложности алгоритма. В рамках дальнейшего изложения будем считать, что аргумент x непрерывен, то есть $f = f(x)$, а необходимые значения функции $f(x)$ вычис-

ляются 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 разграничивающей для полиномов и экспонент.

Доказательство.

Утверждение эквивалентно тому, что функция $g(x)$ удовлетворяет следующим двум соотношениям при $x \to \infty$,

$$\text{если } f(x) = x^k,\ k > 0,\ \text{то } f(x) = o(g(x)), \tag{4.2.1}$$

$$\text{если } f(x) = e^{\lambda x},\ \lambda > 0,\ \text{то } g(x) = o(f(x)). \tag{4.2.2}$$

Для доказательства этих соотношений воспользуемся леммой о логарифмическом пределе: если $\lim\limits_{x\to\infty} f(x) = \infty$, и $\lim\limits_{x\to\infty} g(x) = \infty$, то если

$$\lim_{x\to\infty} \frac{\ln f(x)}{\ln g(x)} = 0, \text{ то } \lim_{x\to\infty} \frac{f(x)}{g(x)} = 0, \text{ т. е. } f(x) = o(g(x)).$$

На основании этой леммы покажем справедливость соотношения (4.2.1):

$$\lim_{x\to\infty} \frac{\ln (x^k)}{\ln\left((\ln x)^{\ln x}\right)} = \lim_{x\to\infty} \frac{k \cdot \ln x}{\ln x \cdot \ln (\ln x)} = \lim_{x\to\infty} \frac{k}{\ln (\ln x)} = 0,$$

следовательно, $x^k = o\left((\ln x)^{\ln x}\right)$ при $k > 0$, и соотношения (4.2.2):

$$\lim_{x\to\infty} \frac{\ln \left((\ln x)^{\ln x}\right)}{\ln\left(e^{\lambda x}\right)} = \lim_{x\to\infty} \frac{\ln x \cdot \ln (\ln x)}{\lambda x} = 0,$$

следовательно, $(\ln x)^{\ln x} = o\left(e^{\lambda x}\right)$ при $\lambda > 0$.

Конец доказательства.

Угловая мера асимптотического роста функций вводится в [4.9] следующим образом: пусть дана функция $f = f(x)$, монотонно возрастающая, и $\lim\limits_{x\to\infty} f(x) = \infty$. Поставим ей в соответствие функцию

$$h(x) = \ln(f(x)) + \frac{\ln(f(x))}{\ln(f(x)) + \ln x} \cdot x. \tag{4.2.3}$$

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 $x\to\infty$, **тогда** $\lim\limits_{x\to\infty} h'(x) = \lambda+1$.

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 $x\to\infty$, **тогда** $\lim\limits_{x\to\infty} h(x) = \dfrac{k}{k+1}$.

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 $\lim\limits_{x\to\infty} h'(x)=C$, где $C>0$, рассмотрим образованную на основе функции $h(x)$ параметрически заданную функцию $z(s)$, определенную следующим образом:

$$z(s)=\begin{cases} s=arctg\left(\dfrac{1}{x}\right); \\ z=arctg\left(\dfrac{1}{h(x)}\right); \end{cases}, \text{ тогда } \lim_{\substack{x\to\infty \\ (s\to 0)}} \frac{dz}{ds}=\frac{1}{C}. \qquad (4.2.4)$$

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 $x\to\infty, s\to 0$ 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 (4.2.4) для $f(x)=x^2$ и $f(x)=e^x$ приведен на рисунке 4.1.

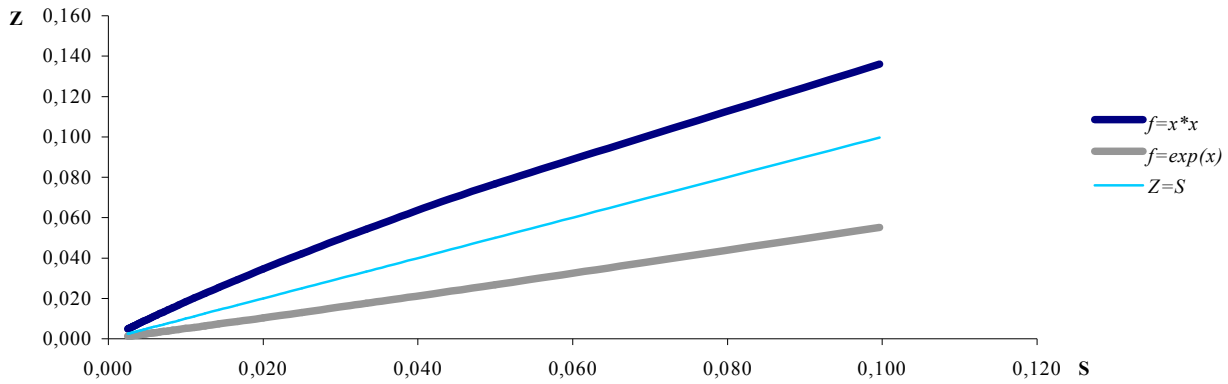


Рисунок 4.1. Функция $z(s)$ для полинома $f(x)=x^2$ и экспоненты $f(x)=e^x$.

Леммы 4.1, 4.2 и 4.3 служат основой следующей теоремы, доказанной в [4.9].

Теорема 4.1. (об угловой мере асимптотического роста функций). Пусть дана функция $f = f(x)$, монотонно возрастающая, и $\lim\limits_{x\to\infty} f(x) = \infty$. Определим меру $\pi(f(x))$ асимптотического (на бесконечности) роста функции

$$f(x):\ \pi(f(x)) = \pi - 2\cdot \mathrm{arctg}(R),\ \text{где}\ R = \lim_{\substack{x\to\infty \\ (s\to 0)}} \frac{dz}{ds},$$

где параметрически заданная функция $z(s)$ определена в виде (4.2.4), 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 $x \to \infty$,

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 $x \to \infty$,

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 $\lim\limits_{x\to\infty} f(x) = \infty$:

1) Определим множество функций FZ: $FZ = \left\{ f(x) \mid f(x) \prec x^k, \forall k > 0 \right\}$. Для функции $f(x)$ из множества FZ значение R, определяемое по лемме 4.3, равно $+\infty$, и мера $\pi(f(x)) = \pi - 2\cdot\mathrm{arctg}(R) = 0\ \ \forall\ f(x) \in FZ$, в частности $\pi(\ln(x)) = 0$.

2) Определим множество полиномов FP: $FP = \left\{ f(x) \mid \exists\, k > 0 : f(x) = \Theta\left(x^k\right) \right\}$. Данное определение базируется на лемме 4.2, однако можно показать, что предложенная мера остается в силе и для более широкого класса функций вида $f(x) = \Theta\left(x^k\right)\cdot g(x)$, где $g(x) \in FZ$, тогда множество

FP может быть определено следующим образом — вначале определим множество функций F_k:

$$F_k = \left\{ f(x) \mid x^{k-\varepsilon} \prec f(x) \prec x^{k+\varepsilon}, k > 0, \varepsilon \to +0, x \to +\infty \right\},$$

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 $\pi(f(x)) = \pi - 2 \cdot \mathrm{arctg}((k+1)/k)$, следовательно $0 < \pi(f(x)) < \pi/2$. График меры для полиномов представлен на рисунке 4.2.

3) Определим множество функций FL:

$$FL = \left\{ f(x) \mid x^k \prec f(x) \prec e^{\lambda x}, \forall\, k > 0, \forall\, \lambda > 0 \right\}.$$

Для функции $f(x)$ из FL значение R, определяемое по лемме 4.3, равно 1, и мера $\pi(f(x)) = \pi - 2 \cdot \mathrm{arctg}(1) = \pi/2$, в частности $\pi(\ln x^{\ln x}) = \pi/2$.

4) Определим множество экспонент FE: $FE = \left\{ f(x) \mid \exists \lambda > 0 : f(x) = \Theta\left(e^{\lambda x}\right) \right\}$. Данное определение базируется на лемме 4.1, однако можно показать, что предложенная мера остается в силе и для более широкого класса функций вида $f(x) = \Theta\left(e^{\lambda x}\right) \cdot g(x)$, где $g(x) \in \{FZ, FP, FL\}$, тогда множество FE может быть определено следующим образом — определим множество функций F_λ:

$$F_\lambda = \left\{ f(x) \mid e^{(\lambda-\varepsilon)\cdot x} \prec f(x) \prec e^{(\lambda+\varepsilon)\cdot x}, \lambda > 0, \varepsilon \to +0, x \to +\infty \right\},$$

и на его основе определим множество обобщенных экспонент FE:

$$FE = \{ f(x) \mid \exists \lambda > 0 : f(x) \in F_\lambda \}.$$

Для функции $f(x)$ из FE значение $R = 1/(1+\lambda), \lambda > 0$, по леммам 4.1 и 4.3, и мера $\pi(f(x)) = \pi - 2 \cdot \mathrm{arctg}(1/(1+\lambda))$, следовательно $\pi/2 < \pi(f(x)) < \pi$. График меры для экспонент представлен на рисунке 4.3.

5) Определим множество функций FF: $FF = \left\{ f(x) \mid e^{\lambda x} \prec f(x), \forall\, \lambda > 0 \right\}$. Для функции $f(x)$ из FF значение R, определяемое по лемме 4.3, равно 0, и мера $\pi(f(x)) = \pi - 2 \cdot \mathrm{arctg}(R) = \pi$, в частности $\pi\left(x^x\right) = \pi$.

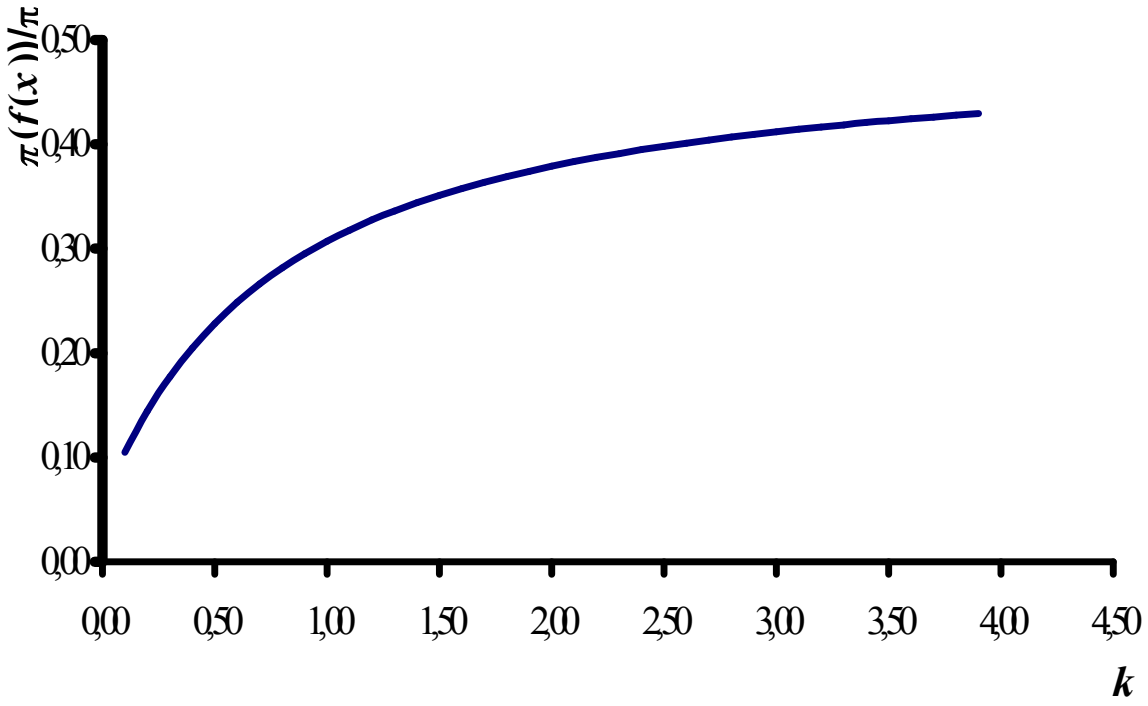


Рисунок. 4.2. График меры $\pi(f(x))$ для полиномов $f(x)=\Theta\left(x^k\right)$.

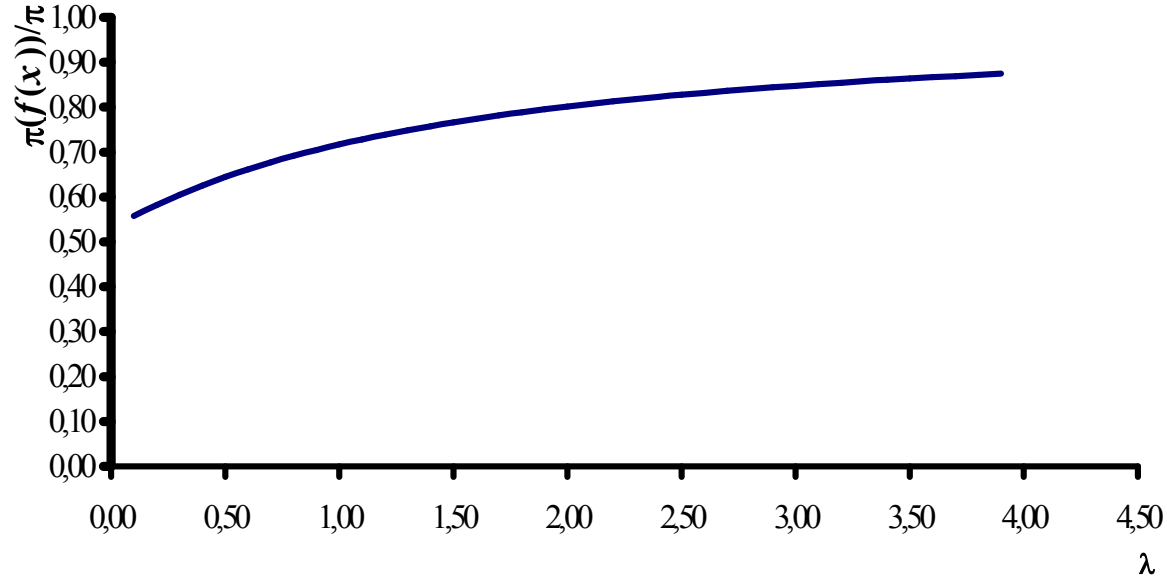


Рисунок 4.3. График меры $\pi(f(x))$ для экспонент $f(x)=\Theta\left(e^{\lambda x}\right)$.

Укажем следующий ряд свойств, которыми обладает введенная угловая мера асимптотического роста функций — $\pi(f(x))$:

— мера $\pi\left(x^k\right)$ принимает значение, равное $\pi/4$, для степени $k=\sqrt{2}/2$;

— мера $\pi\left(e^{\lambda x}\right)$ принимает значение, равное $3\pi/4$, при показателе $\lambda=\sqrt{2}$;

— мера $\pi(f(x))$ обладает следующим интересным свойством

$$\pi\left(x^{1/\lambda}\right)+\pi\left(e^{\lambda x}\right)=\pi, \text{ в частности } \pi(x)+\pi\left(e^{x}\right)=\pi.$$

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 [4.9]:

1. Класс $\pi 0$ (пи нуль) — класс «быстрых алгоритмов» — это алгоритмы, для которых функции сложности принадлежат множеству FZ и имеют меру ноль:

$$\pi 0=\left\{ A \mid \pi\left(f_A^*(n)\right)=0 \Leftrightarrow f_A^*(n)\in FZ \right\}.$$

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:

$$\pi P=\left\{ A \mid 0<\pi\left(f_A^*(n)\right)<\pi/2=0 \Leftrightarrow f_A^*(n)\in FP \right\}.$$

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:

$$\pi L=\left\{ A \mid \pi\left(f_A^*(n)\right)=\pi/2 \Leftrightarrow f_A^*(n)\in FL \right\}.$$

Этот класс образуют алгоритмы с более чем полиномиальной, но менее чем экспоненциальной сложностью. Эти алгоритмы достаточно трудоёмки, соответст-

вующие 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 криптосистему RSA. Если n есть количество битов числа, предъявляемого для факторизации, то эвристическая оценка сложности этого алгоритма, приведенная в [4.10], имеет вид

$$f_A(n) = e^{O\left(n^{\frac{1}{3}\cdot(\ln(n))^{\frac{2}{3}}}\right)}, \text{ и } \pi(f_A(n)) = \pi/2.$$

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:

$$\pi E = \left\{ A \mid \pi/2 < \pi\left(f_A^*(n)\right) < \pi \Leftrightarrow f_A^*(n) \in FE \right\}.$$

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

$$O\left(2^n\right), O\left(n \cdot 2^n\right), O\left(n^2 \cdot 2^n\right).$$

5. Класс πF — класс «надэкспоненциальных алгоритмов» — это алгоритмы, функции сложности которых принадлежат множеству FF:

$$\pi E = \left\{ A \mid \pi\left(f_A^*(n)\right) = \pi \Leftrightarrow f_A^*(n) \in FF \right\}.$$

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 $n! = \Gamma(n+1)$, 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 $\pi\left(e^{x(\ln x - 1)}\right) = \pi$, то

этот алгоритм относится к классу πF. К этому же классу относится алгоритм полного перечисления всех остовных деревьев полного графа на n вершинах с асимптотической оценкой трудоемкости $\Omega\left(n^{n-2}\right)$ [4.6]. Обозначение F в названии класса отражает принадлежность к этому классу алгоритмов с факториальной (Factorial) оценкой трудоемкости.

4.3. Классификация компьютерных алгоритмов и задач по влиянию на трудоемкость особенностей входов

Введение. На ресурсные характеристики алгоритма, под которыми мы будем в дальнейшем понимать трудоемкость в базовых операциях (основная характеристика) и требуемый объем памяти (емкостная эффективность), оказывают влияние различные характеристики множества исходных данных — совокупности допустимых входов алгоритма. Это длина входа, конкретные значения некоторых элементов входа, их порядок и т.д. [4.11]. Тем не менее, основной принятой характеристикой входа алгоритма является его длина. Цель настоящего параграфа — обобщить результаты по классификации алгоритмов на основе типизации задач по порождаемой длине входа.

Будем использовать далее терминологию и обозначения, введенные в параграфе 3.1. Поскольку значение функции трудоемкости $f_A(D)$ может определяться не только размером входа n, но и другими характеристиками множества D, например значениями и порядком расположения элементов, то выделим в функции $f_A(D)$ количественную и параметрическую составляющую, обозначив их через $f_n(n)$ и $g_p(D)$ (обозначения предложены в [4.11]), тогда

$$f_A(D) = f_A\left(f_n(n), g_p(D)\right).$$

Для большинства алгоритмов функция $f_A(D)$ может быть представлена как композиция функций $f_n(n)$ и $g_p(D)$ либо в мультипликативной, либо в аддитивной форме

$$f_A(n, D) = f_n(n) \cdot g_p^*(D), \text{ или } f_A(n, D) = f_n(n) + g_p^+(D).$$

Укажем, что мультипликативная форма характерна для ряда алгоритмов, когда сильно параметрически зависимый внешний цикл, определяющий перебор вари-

антов решения задачи, содержит внутренний цикл, проверяющий решение с количественно-зависимой от размерности трудоемкостью. Такая ситуация возникает, например, для ряда алгоритмов, решающих задачи из класса NPC [4.2]. В силу конечности множества D_n существует худший случай для функций $g_p^*(D)$, и $g_p^+(D)$. Обозначим соответствующие функции через $g^*(n)$ и $g^+(n)$:

$$g^*(n) = \max_{D \in D_n} \{ g_p^*(D) \}, \quad g^+(n) = \max_{D \in D_n} \{ g_p^+(D) \}.$$

Используя эти понятия и обозначения, введем типизацию задач и алгоритмов по длине входа и уточняющие ее классификации.

I. Типизация задач и решающих их алгоритмов по длине входа

Конкретные входы, порожденные различными задачами, обладают различными характеристическими особенностями. Однако мы можем выделить группу задач, всегда порождающих входы фиксированной длины. Например, задача извлечения квадратно корня с заданной точностью всегда порождает вход, состоящий из двух чисел. В отличие от этой группы, другие задачи порождают входы переменной длины — задачи сортировки или умножения матриц являются характерными примерами задач этой группы. Таким образом, алгоритмы, решающие задачи из разных групп, имеют на входе или множество с заранее фиксированным числом элементов, или множество с переменным числом элементов, что приводит к следующей типизации задач, и, соответственно, решающих их алгоритмов.

A. Тип L_c — алгоритмы с постоянной (фиксированной) длиной входа. Для этого типа алгоритмов и соответствующих им задач

$$\exists\, n = const : \forall D \in D_A \; |D| = n .$$

Примерами для этого типа могут служить алгоритмы вычисления значений стандартных функций, вычисления наибольшего общего делителя двух чисел и т.д.

B. Тип L_n — алгоритмы с переменной длиной входа. Для этого типа алгоритмов и соответствующих им задач

$$\exists D_1, D_2 \in D_A : |D_1| \neq |D_2| .$$

Примерами являются алгоритмы выполнения операций с матрицами и векторами, алгоритмы обработки строк символов, алгоритмы сортировка и т.д.

II. Классификация алгоритмов по трудоемкости в типе L_c

Рассмотрим ситуацию, когда на функцию трудоемкости алгоритма не оказывают влияние никакие другие характеристические особенности входа, за исключением его длины. Но поскольку мы рассматриваем алгоритмы, принадлежащие к типу L_c, то оказывается, что в силу предположения, эти алгоритмы должны иметь фиксированную трудоемкость. Таким образом, мы вводим в типе L_c класс алгоритмов с постоянной трудоемкостью. В противном случае трудоемкость зависит от некоторых дополнительных характеристических особенностей конкретного входа, как правило, от значений элементов множества D, и мы относим эти алгоритмы к классу алгоритмов, параметрически-зависимых по трудоемкости. Более формально:

A. Класс C — алгоритмы с постоянной (фиксированной) трудоемкостью:

$$\exists n = const, \exists\, k = const : \forall D \in D_A \; |D| = n \;, f_A(D) = k \;.$$

Тривиальный пример — алгоритм сложения двух чисел. Более реальный пример — вычисление прогиба балки по ее известным характеристикам и значению нагрузки.

B. Класс PR — алгоритмы с трудоемкостью, параметрически зависящей от значений некоторых элементов множества D.

$$\exists n = const, \exists\, k = const : \forall D \in D_A \; |D| = n \; f_n(n) = const = c, \Rightarrow f_A(D) = c \cdot g(D).$$

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 этих значений количество базовых операций. Например, для алгоритма вычисления x^k последовательным умножением — $f_A(D) = f_A(k)$, а для алгоритма вычисления экспоненты по формуле $e^x = \sum \left(x^n / n! \right)$, с точностью до ε — $f_A(D) = f_A(x, \varepsilon)$.

III. Подклассы класса PR в типе L_c

Достаточно часто алгоритмы класса PR обладают трудоемкостью, которая зависит только от фиксированного числа параметров множества исходных дан-

ных. 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 значений этих бит. Это рассуждение может быть обобщено на несколько параметров во множестве D, влияющих на трудоемкость алгоритма. 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 — полиномиальному или экспоненциальному, получая тем самым два подкласса а классе PR.

A. Подкласс $PR\,pl$ — подкласс алгоритмов с полиномиальной параметрически-зависимой трудоемкостью

$$\exists\, k = const : f_A(m) = \mathrm{O}\left(m^k\right).$$

Содержательно к подклассу $PR\,pl$ относятся алгоритмы, трудоемкость которых зависит только от числа бит параметров. Примером является алгоритм возведения в целую степень методом повторного возведения в квадрат [4.12].

B. Подкласс $PR\exp$ — подкласс алгоритмов с экспоненциальной параметрически-зависимой трудоемкостью

$$\exists\, k = const : f_A(m) = \mathrm{O}\left(e^m\right).$$

Содержательно к подклассу $PR\exp$ относятся алгоритмы, трудоемкость которых зависит от числового значения параметров. Примером является алгоритм возведения в целую степень методом последовательного умножения [4.12].

IV. Классификации алгоритмов по трудоемкости в типе L_n

Для задач и соответствующих алгоритмов, относящихся к типу L_n возможно несколько вариантов зависимости трудоемкости алгоритма от характеристических особенностей множества D. Напомним, что в типе L_n мы рассматриваем совокупность подмножеств входов, обладающих разной длиной n — D_n. Формально могут быть выделены следующие классы:

A. Класс C — алгоритмы с постоянной (фиксированной) трудоемкостью:

$$\exists\, k = const : \forall n : D_n \in D_A \ \forall D \in D_n \quad f_A(D) = k\,.$$

Эта, на первый взгляд, парадоксальная ситуация, тем не менее, реальна. Рассмотрим задачу поиска максимального элемента в уже отсортированном массиве, содержащем n элементов. Очевидно, что вне зависимости от значения n результат получается за фиксированное число базовых операций модели вычислений. Еще один пример — поиск по ключу в массиве длиной n с использованием минимальной совершенной функции хеширования [4.2], что обеспечивает поиск по любому ключу за $\Theta(1)$ базовых операций.

B. Класс PR — класс параметрически-зависимых по трудоемкости алгоритмов. Это алгоритмы, трудоемкость которых определяется не размерностью входа, а конкретными значениями всех или некоторых элементов из входного множества D или иными характеристическими особенностями входа, например порядком расположения элементов.

$$\forall D_n \in D_A \ \forall D \in D_n \ f_n(n) = const = c, \Rightarrow f_A(n, D) = c \cdot g^*(n)\,.$$

C. Класс N — класс количественно-зависимых по трудоемкости алгоритмов. Это алгоритмы, функция трудоемкости которых зависит только от размерности конкретного входа, при этом

$$g^+(n) = 0 \Rightarrow g^*(n) = 1 \Rightarrow f_A(D) = f_n(n)\,.$$

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.

D. 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

$$f_n(n) \neq const,\ \ g^*(n) \neq const,$$

$$f_A(D) = f_n(n) \cdot g^*(n), \text{ или } f_A(D) = f_n(n) + g^+(n)\,.$$

В качестве одного из общих примеров можно привести алгоритмы, реализующие ряд численных методов, в которых параметрически-зависимый внешний цикл по

точности включает в себя количественно-зависимый фрагмент по размерности, порождая мультипликативную форму для $f_A(D)$.

V. Подклассы класса PR **в типе** L_n

Для алгоритмов с переменной длиной входа параметрическая зависимость трудоемкости может определяться как заранее фиксированным числом параметров, так и переменным числом параметров, вплоть до значений всех элементов входа. Это наблюдение приводит в к выделению следующих подклассов класса PR в типе L_n:

A. Класс $PR\ fix$ **— алгоритмы с трудоемкостью, параметрически зависящей от фиксированного числа элементов множества** D**:**

$$f_n(n) = const = c, \Rightarrow f_A(D) = c \cdot g(D),\ g(D) = g(p_1, \ldots, p_m),$$

где m не зависит от размерности.

Это случай, когда только заранее фиксированное число параметров входа алгоритма определяют функцию трудоемкости. Пример — случайный выбор k элементов из массива длиной n. В ряде случаев эта ситуация свидетельствует о нерационально организованном входном массиве.

B. Класс $PR\ float$ **— алгоритмы с трудоемкостью, параметрически зависящей от переменного числа значений элементов множества** D**:**

$$f_n(n) = const = c, \Rightarrow f_A(D) = c \cdot g(D),\ g(D) = g(p_1, \ldots, p_m),$$

где значение m может зависеть как от размерности, так и особенностей конкретного входа, но в общем случае не является фиксированным. В качестве примера рассмотрим задачу определения числа элементов входного массива, сумма значений которых превышает заданное число. Суммирование ведется в порядке возрастания индексов исходного массива. Очевидно, что условие выхода из цикла суммирования есть превышение текущей суммой значения заданного числа. Таким образом, трудоемкость определяется значениями некоторой части элементов исходного массива, а в худшем случае трудоемкости — всех его элементов.

VI. Подклассы класса NPR **по степени влияния на трудоемкость параметрического компонента**

Для количественно-параметрических алгоритмов может быть предложена более детальная классификация, ставящая целью выяснение степени влияния количественной и параметрической составляющей на главный порядок функции $f_A(D) = f_n(n) \cdot g^*(n)$, и приводящая к выделению следующих подклассов в классе NPR [4.11]:

A. Подкласс $NPRL$ (Low) **— слабо-параметрический подкласс класса** NPR**:**

$$g^+(n) = O(f_n(n)) \Leftrightarrow g^*(n) = \Theta(1).$$

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 об алгоритмах, трудоемкость которых слабо зависит от параметрической составляющей. К этому подклассу относится, например, алгоритм поиска максимума в массиве, т. к. количество переприсваиваний максимума в худшем случае, когда массив отсортирован по возрастанию, определяющее $g^+(n) = \Theta(n)$, имеет равный порядок с оценкой внешнего цикла перебора n элементов.

B. Подкласс $NPRE$ $(Equivalent)$ **— средне-параметрический подкласс класса** NPR**.** Это подкласс алгоритмов, у которых в функции трудоемкости составляющая $g^*(n)$ имеет порядок роста, не превышающий $f_n(n)$:

$$g^*(n) = \Theta(f_n(n)) \Leftrightarrow g^+(n) = \Theta\left(f_n{}^2(n)\right).$$

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. а это — обратно отсортированный массив, определяет $g^+(n) = \Theta\left(n^2\right)$, а $f_n(n) = \Theta(n)$.

C. Подкласс $NPRH$ $(High)$ **— сильно-параметрической подкласс класса** NPR**.** Это подкласс алгоритмов, в трудоемкости которых составляющая $g^*(n)$ имеет асимптотический порядок роста выше $f_n(n)$: $f_n(n) = o\left(g^*(n)\right)$. Алгоритмы

этого подкласса отличаются тем, что именно параметрический компонент определяет главный порядок функции трудоемкости. В этот подкласс входят, например, все алгоритмы точного решения NP-полных задач. Для алгоритмов этого подкласса характерна, как правило, мультипликативная форма функции трудоемкости, что позволяет говорить об итерационно-параметрическом характере $f_A(D)$. 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.

VII. Подклассы класса NPR по характеристическим особенностям множества исходных данных.

Другая, и не зависимая от предыдущей, классификация алгоритмов в классе NPR [4.11] предполагает выделение в функции $g_p(D)$ аддитивных компонент, связанных со значениями элементов входа — $g_v(D)$, и их порядком — $g_s(D)$. Таким образом, функция $g_p(D)$ представляется 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 $\{d_1,\ldots,d_m\}, |D_e| = m$, и определим D_p как множество всех упорядоченных последовательностей из $\{d_1,\ldots,d_m\}$, 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

$$f_A(n, D) = f_A(n, p_1, \ldots p_m),\ m \le n,\ p_i \in D,\ i = \overline{1,m}.$$

Оценивая степень влияния $g_v(D), g_s(D)$ на $g_p(D)$, можно определить следующие подклассы в классе NPR:

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. Можно говорить о количественно-порядковом характере функции трудоемкости. Примерами могут служить большинство алгоритмов сортировки сравнениями, алгоритмы поиска минимума и максимума в массиве.

В. 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:

$$NPR - (NPRS \cup NPRV) \neq \varnothing.$$

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 [4.13].

В целом описанная типизация и классификация алгоритмов может быть представлена иерархическим деревом классов, показанном на рис 4.4.

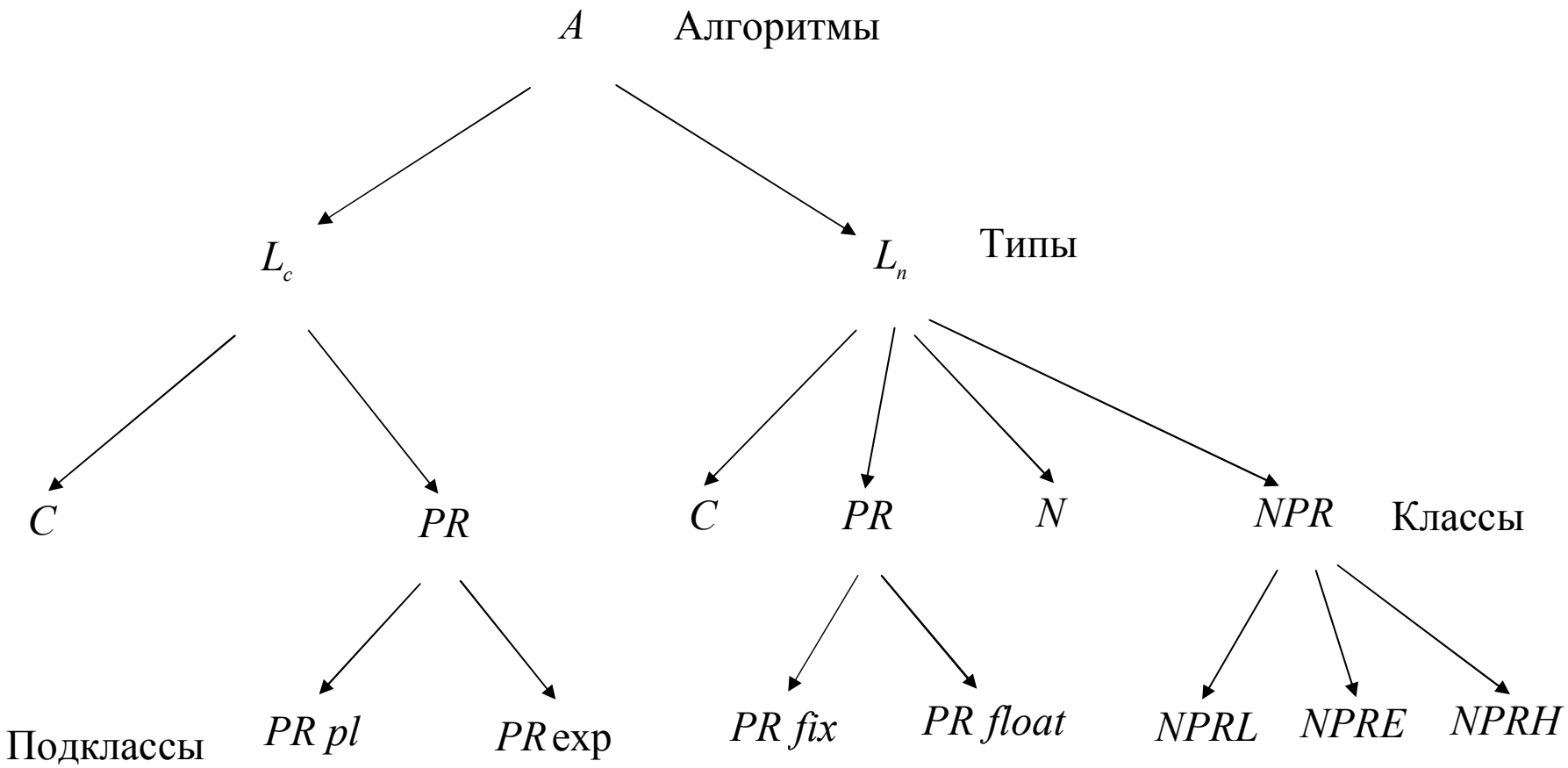


Рисунок 4.4. Иерархия типов и классов алгоритмов

4.4 Классификация компьютерных алгоритмов по информационной и размерностной чувствительности

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 $f_A(D)$. 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 [4.14].

Определение 4.1. Под *размерностной чувствительностью алгоритма* будем понимать влияние изменения размерности на значения функции трудоемкости.

Определение 4.2. Под *информационной чувствительностью алгоритма* будем понимать влияние различных входов фиксированной размерности на изменение значений 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 рисунке 4.5. 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 рисунке 4.5 показана огибающая линия этих точечных значений. 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.

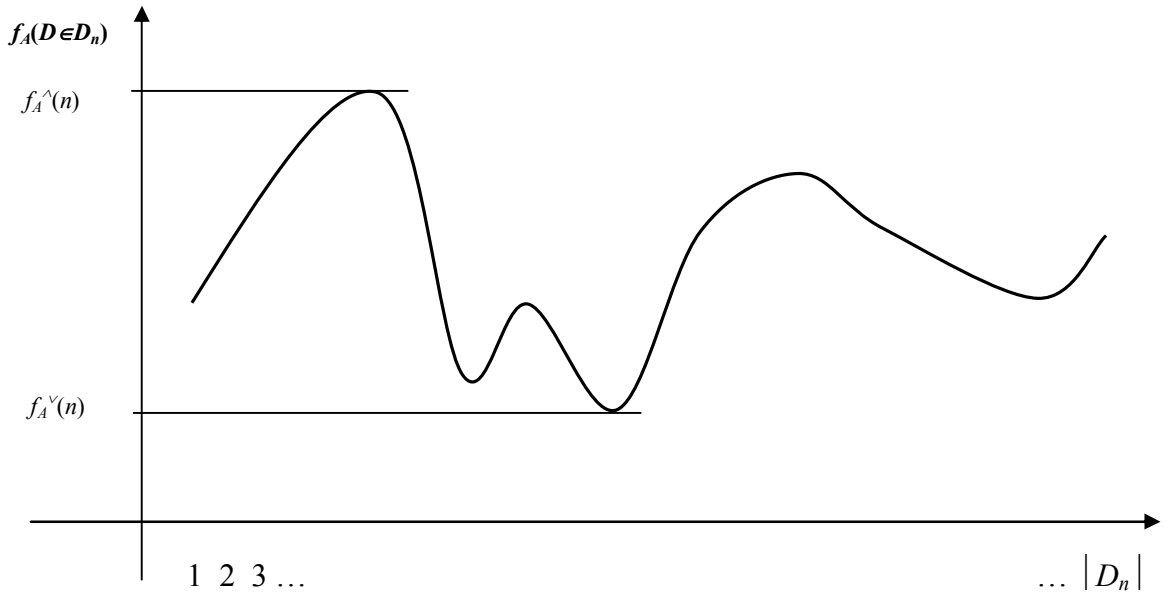


Рисунок 4.5. Огибающая точечного графика трудоемкости количественно-параметрического алгоритма для различных входов фиксированной размерности.

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 огибающей такой гистограммы показан на рисунке. 4.6.

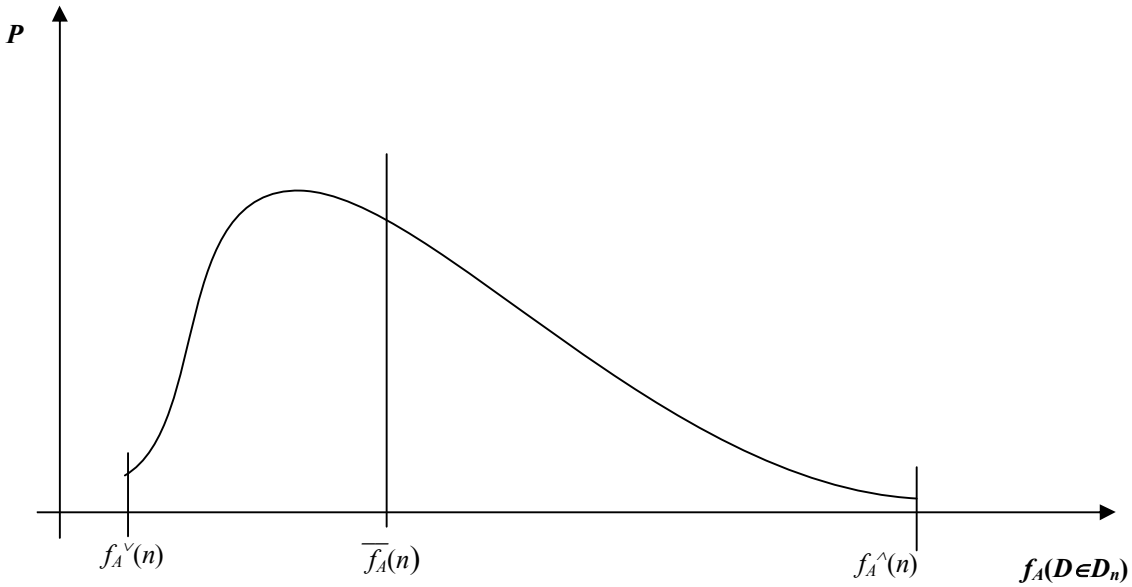


Рисунок 4.6. Огибающая гистограммы относительных частот значений функции
трудоемкости для количественно-параметрического алгоритма.

При этом очевидно выполнено следующее условие

$$\sum P(f_A) = 1,$$

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 $D_n : f_A^{\vee} \le f_A \le f_A^{\wedge}$. Таким образом, мы получаем более детальную информацию о поведении алгоритма, рассматривая функцию трудоемкости как дискретную случайную величину, характеризующуюся 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 [4.15].

Вводя понятие информационной чувствительности, и в дальнейшем обозначая ее через $\delta_i(n)$, имеет смысл 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. Стандартно мерой рассеяния случайной величины относительно математического ожидания является дисперсия — D или среднеквадратическое отклонение — σ. Верхняя граница для среднеквадратического отклонения ограниченной случайной величины, заданной функций $f(x)$ — σ_f определяется следующей теоремой [4.14].

Теорема 4.2. Непрерывная случайная величина, ограниченная по x сегментом $[a,b]$, и заданная на нем функцией распределения $f(x)$:

$$\forall\, x : a \le x \le b : f(x) \ge 0, \text{ и } \int_a^b f(x)\, dx = 1,$$

имеет максимальную дисперсию $D[f(x)]$, если функция $f(x)$ имеет вид

$$f(x) = q \cdot \delta(x-a) + p \cdot \delta(x-b),$$

причем $p = q = 1/2$, где $\delta(x)$ — дельта функция, при этом $D[f(x)] \le 1/4 \cdot (a-b)^2$.

Переходя к дискретным случайным величинам, мы можем на основании теоремы 4.2 говорить, что дисперсия σ_{f_A} 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

$$M[f_A(n)] = \left(f_A^{\wedge}(n) + f_A^{\vee}(n)\right)/2, \quad \sigma_{f_A max}(n) = \left(f_A^{\wedge}(n) - f_A^{\vee}(n)\right)/2. \qquad (4.4.1)$$

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 $R = \left(f_A^{\wedge}(n) - f_A^{\vee}(n)\right)$ [4.16]. 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 для входов длины n — $R_N(n)$ как отношение половины вариантного интервала к его середине:

$$R_N(n) = \left(f_A^{\wedge}(n) - f_A^{\vee}(n)\right) / \left(f_A^{\wedge}(n) + f_A^{\vee}(n)\right). \qquad (4.4.2)$$

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 [4.16]; для функции трудоемкости V, как функция длины входа, имеет вид

$$V(n) = \sigma_{f_A(n)} / \overline{f_A}(n), \quad 0 \leq V(n) \leq 1, \qquad (4.4.3)$$

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 (4.4.2) и (4.4.3). Поскольку $\sigma_{f_A(n)}$ может изменяться от 0

до $\sigma_{f_A max}(n)$, то введем, следуя [4.14], количественную меру информационной чувствительности алгоритма в виде

$$\delta_i(n) = V(n) \cdot R_N(n), \quad 0 \le \delta_i(n) \le 1. \tag{4.4.4}$$

Тем самым информационная чувствительность алгоритма при фиксированной длине входа учитывает и среднеквадратическое отклонение функции трудоемкости, и размер интервала возможных значений в нормированных единицах.

Предложенные понятие и определение 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

$$* \in \{ \wedge, \bar{}, \vee \}. \tag{4.4.5}$$

Таким образом, мы вводим три количественных меры размерностной чувствительности алгоритма с обозначением $\delta_n^*(n)$, где символ * определен в соответст-

вии с (4.4.5). Заметим, что поскольку соответствующие функции трудоемкости используются в определении информационной чувствительности, то возникает возможность ее определения для других размерностей на основе размерностной чувствительности. 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 [4.17].

Случай рекурсивного задания функции. Без потери общности можно считать, что функция $f_A^*(n)$ задана рекурсивно в виде

$$f_A^*(n+1) = h_A^*(n) \cdot f_A^*(n). \tag{4.4.6}$$

Поскольку функция $f_A^*(n)$ является, по крайней мере, неубывающей:

$$f_A^*(n+1) \geq f_A^*(n), \text{ и } \lim_{n\to\infty} f_A^*(n) = \infty, \text{ то } \quad h_A^*(n) \geq 1,$$

то представим функцию $h_A^*(n)$ в виде

$$h_A^*(n) = 1 + \delta_n^*(n), \quad \delta_n^*(n) \geq 0. \tag{4.4.7}$$

Подставляя (4.4.6) в (4.4.7) и выделяя $\delta_n^*(n)$, получим формулу для количественной меры размерностной чувствительности функции трудоемкости алгоритма для лучшего, среднего и худшего случаев исходных данных

$$\delta_n^*(n) = \frac{f_a{}^*(n+1) - f_a{}^*(n)}{f_a{}^*(n)}. \tag{4.4.8}$$

Таким образом, рекурсивно заданная функция $f_A^*(n)$ связана с мерой размерностной чувствительности соотношением

$$f_A^*(n+1) = f_A^*(n) + \delta_n^*(n) \cdot f_A^*(n). \tag{4.4.9}$$

Случай задания функции в явном виде. Следуя классическому определению чувствительности, необходимо рассмотреть производную функции $f_A^*(n)$ по

размерности, однако для согласования с формулой (4.4.8) предлагается использовать логарифмическую производную:

$$\delta_n^*(n) = \frac{d}{dn}\ln\left(f_A^*(n)\right) = \frac{(f_A^*(n))'}{f_A^*(n)}. \qquad (4.4.10)$$

Если с учетом целочисленности аргумента функции перейти в формуле (4.4.10) от производной к конечной разности

$$\left(f_A^*(n)\right)' \to f_A^*(n+1) - f_A^*(n),$$

то мы получим для $\delta_n^*(n)$ формулу (4.4.8), 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 4.3: $f_n(n)$ — количественный компонент функции трудоемкости; $g^+(n)$ — параметрический компонент в аддитивной форме; $g^*(n)$ — параметрический компонент в мультипликативной форме.

1. Подкласс $NPRL - g^+(n) = O(f_n(n))$. В соответствии с определением понятия «о большое» будем считать, что $g^+(n) \le c \cdot f_n(n)$, тогда для алгоритмов этого подкласса количественная мера размерностной чувствительности имеет вид

$$\delta_n^*(n) = \frac{f_n(n+1) + c \cdot f_n(n+1) - f_n(n) - c \cdot f_n(n)}{f_n(n) + c \cdot f_n(n)} = \frac{f_n(n+1) - f_n(n)}{f_n(n)}. \qquad (4.4.11)$$

Таким образом, для алгоритмов класса $NPRL$ количественная мера размерностной чувствительности трудоемкости совпадает с мерой ее количественной компоненты.

2. Подкласс $NPRE - g^*(n) = \Theta(f_n(n))$. В соответствии с определением понятия «тета» будем считать, что $g^*(n) = c \cdot f_n(n)$, тогда для алгоритмов этого подкласса количественная мера размерностной чувствительности имеет вид

$$\delta_n^*(n) = \frac{f_n(n+1) \cdot c\ f_n(n+1) - f_n(n) \cdot c\ f_n(n)}{f_n(n) \cdot c\ f_n(n)} =$$

$$= \frac{f_n(n+1) - f_n(n)}{f_n(n)} \cdot \frac{f_n(n+1) + f_n(n)}{f_n(n)} = \delta_{f_n}(n) \cdot \left(1 + \frac{f_n(n+1)}{f_n(n)}\right), \qquad (4.4.12)$$

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, $\delta_n^*(n) \geq 2 \cdot \delta_{f_n}(n)$ в силу формулы (4.4.12). Таким образом, для алгоритмов класса *NPRE* количественная мера размерностной чувствительности не менее чем в два раза превышает количественную меру размерностной чувствительности количественной компоненты функции трудоемкости.

3. Подкласс $NPRH - f_n(n) = o\left(g^*(n)\right)$. Используем запись компонент функции трудоемкости через размерностную чувствительность в виде (4.4.9), тогда

$$f_n(n+1) = f_n(n) + \delta_{f_n}(n) \cdot f_n(n), \quad g^*(n+1) = g^*(n) + \delta_g^* \cdot g^*(n),$$

где через $\delta_g^*(n)$ обозначена мера размерностной чувствительности параметрической компоненты функции трудоемкости. Тогда для алгоритмов этого подкласса количественная мера размерностной чувствительности имеет вид

$$\delta_n^*(n) = \frac{\left(f_n(n) + \delta_{f_n}(n) \cdot f_n(n)\right) \cdot \left(g^*(n) + \delta_g^*(n) \cdot g^*(n)\right) - f_n(n) \cdot g^*(n)}{f_n(n) \cdot g^*(n)} =$$

$$= \delta_{f_n}(n) + \delta_g^*(n) + \delta_{f_n}(n) \cdot \delta_g^*(n) = \delta_g^*(n) \cdot \left(1 + \delta_{f_n}(n) + \frac{\delta_{f_n}(n)}{\delta_g^*(n)}\right), \qquad (4.4.13)$$

в силу определения подкласса $\delta_g^*(n) >> \delta_{f_n}(n)$, следовательно,

$$\delta_n^*(n) \approx \delta_g^*(n) \cdot \left(1 + \delta_{f_n}(n)\right).$$

Таким образом, для алгоритмов класса *NPRH* количественная мера размерностной чувствительности определяется параметрической и количественной компонентами функции трудоемкости. Полученные результаты позволяют подойти к определению подклассов алгоритмов в классе *NPR* и 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 [4.16]:

— функция трудоемкости обладает максимальным среднеквадратичным отклонением, а размах варьирования стремится к единице — в этом случае

$$\sigma_{f_A}(n) = \sigma_{f_A max} = \left(f_A^{\wedge}(n) - f_A^{\vee}(n)\right)/2, \quad \overline{f}_A(n) = \left(f_A^{\wedge}(n) + f_A^{\vee}(n)\right)/2,$$

$$f_A^{\vee}(n) << f_A^{\wedge}(n), \text{ или } f_A^{\vee}(n) = o\left(f_A^{\wedge}(n)\right),$$

что обеспечивает $R_N(n) \to 1$ при $n \to \infty$ в соответствии с (4.4.2), тогда $\delta_i(n) \to 1$ при $n \to \infty$;

— гистограмма относительных частот функции трудоемкости соответствует равномерному закону распределения, и $R_N(n) \to 1$ при $n \to \infty$, тогда, по [4.16]:

$$\sigma_{f_A}(n) = \left(f_A^{\wedge}(n) - f_A^{\vee}(n)\right)/\sqrt{12}, \quad \overline{f}_A(n) = \left(f_A^{\wedge}(n) + f_A^{\vee}(n)\right)/2,$$

следовательно $\delta_i(n) \to 2/\sqrt{12} \approx 0.57735$ при $n \to \infty$;

— гистограмма относительных частот функции трудоемкости соответствует нормальному закону распределения, в предположении, что $R_N(n) \to 1$ при $n \to \infty$, а размах варьирования составляет $6 \cdot \sigma_{f_A}(n)$, получаем

$$\sigma_{f_A}(n) = \left(f_A^{\wedge}(n) - f_A^{\vee}(n)\right)/6, \quad \overline{f}_A(n) = \left(f_A^{\wedge}(n) + f_A^{\vee}(n)\right)/2,$$

тогда $\delta_i(n) \to 2/6 \approx 0{,}33333$ при $n \to \infty$.

— гистограмма относительных частот функции трудоемкости соответствует нормальному закону распределения, $R_N(n) \to 1$ при $n \to \infty$, а среднеквадратичное отклонение составляет $2{,}5\%\,(1/40)$ от размаха варьирования, т. е. интервал в $2 \cdot \sigma_{f_A}(n)$ 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 $p = 2 \cdot \Phi(1) \approx 0{,}6826$ [4.16], в этом случае

$$\sigma_{f_A}(n) = \left(f_A^{\wedge}(n) - f_A^{\vee}(n)\right)/40, \quad \overline{f}_A(n) = \left(f_A^{\wedge}(n) + f_A^{\vee}(n)\right)/2,$$

тогда $\delta_i(n) \to 2/40 = 0{,}05$ при $n \to \infty$.

— 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 [4.17].

Класс i1. Алгоритмы, не чувствительные к входным данным по функции трудоемкости при фиксированной длине входа $\delta_i(n) = 0$. Этот тип совпадает с классом количественно-зависимых алгоритмов (класс N).

Класс i2. Алгоритмы, слабо чувствительные к входным данным по функции трудоемкости при фиксированной длине входа:

$$0 < \delta_i(n) < 0{,}05.$$

Класс i3. Алгоритмы, чувствительные к входным данным по функции трудоемкости при фиксированной длине входа:

$$0{,}05 \le \delta_i(n) < 1/3.$$

Класс i4. Алгоритмы, сильно чувствительные к входным данным по функции трудоемкости при фиксированной длине входа:

$$1/3 \le \delta_i(n) \le 1{,}00.$$

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 теоретическое 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 [4.16], 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 $f_A^*(n) = c \cdot n^k$, $k > 0$, целое, $c > 0$. Тогда в силу определения $\delta_n^*(n)$

$$\delta_n^*(n) = \frac{c \cdot (n+1)^k - c \cdot n^k}{c \cdot n^k} = \frac{c \cdot k \cdot n^{k-1} + O(n^{k-2})}{c \cdot n^k} = \frac{k}{n} + O(n^{-2}).$$

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 $\Delta n = 1$ — $f_A^*(n+1) \approx f_A^*(n) + k \cdot n^{k-1} \cdot \Delta n$. Таким образом, главный порядок размерностной чувствительности для степенных функций трудоемкости имеет вид k/n, где k — показатель степени, не зависит от коэффици-

ента c и гиперболически уменьшается с ростом размерности. В частности, для $f_A^*(n) = c \cdot n, \delta_n^*(n) = 1/n$.

Пример 4.2. Функция трудоемкости имеет экспоненциальный вид.

$$f_A^*(n) = c \cdot e^{\lambda n}, \lambda > 0, c > 0.$$

По определению (4.4.8)

$$\delta_n^*(n) = \frac{c \cdot e^{\lambda(n+1)} - c \cdot e^{\lambda n}}{c \cdot e^{\lambda n}} = \frac{c \cdot e^{\lambda n} \cdot e^{\lambda} - c \cdot e^{\lambda n}}{c \cdot e^{\lambda n}} = e^{\lambda} - 1 = const > 0,$$

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 [4.17]:

Класс n1. Алгоритмы, мало чувствительные (не более чем линейная функция) к изменению размерности множества исходных данных по функции трудоемкости:

$$0 < \delta_n^* \le n^{-1}, \quad \forall n > 1.$$

Класс n2. Алгоритмы, слабо чувствительные (не более чем степенная функция со степенью больше единицы) к изменению размерности множества исходных данных по функции трудоемкости:

$$\delta_n^* = k \cdot n^{-1}, \quad \forall n > 1, k > 1.$$

Класс n3. Алгоритмы, чувствительные (более чем степенная, и менее, чем показательная функция) к множества исходных данных по функции трудоемкости:

$$\delta_n^*(n) \succ n^{-1}, \text{ и } \lim_{n \to \infty} \delta_n^*(n) = 0,$$

где $\succ$ — обозначение отношения асимптотической иерархии функций.

Класс n4. Алгоритмы, сильно чувствительные (не менее чем показательная функция) к изменению размерности по функции трудоемкости:

$$\delta_n^*(n) = const, \text{ или } \lim_{n \to \infty} \delta_n^*(n) = \infty.$$

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.

4.5 Классификация вычислительных алгоритмов по требованиям к дополнительной памяти

Теоретическое исследование ресурсных характеристик компьютерных алгоритмов предполагает введение соответствующих классификаций, отражающих различные ресурсные требования алгоритма и его программной реализации. В этом параграфе рассматривается классификация компьютерных алгоритмов, в основу которой положен требуемый алгоритмом объем дополнительной памяти в принятой модели вычислений. В данном случае рассматривается только объем оперативной памяти, т. к. затраты памяти на внешних носителях требую 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 3.1. 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 объектов входа из файла 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 приводит к ощутимым затратам памяти, даже при современных объемах. Актуальные размерности современных сложных задач также приводят к тому, что ресурс памяти становится значимым в комплексной оценке ресурсной эффективности алгоритма. Однако при сравнении алгоритмов, относящихся или к статическому или потоковому типу основную роль играет именно дополнительная память, затраты которой обусловлены именно спецификой данного алгоритма. В связи с этим предлагается следующая классификация алгоритмов, основанная 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 за-

полнения памяти результата. Очевидно, что это наиболее рациональные алгоритмы по критерию емкостной эффективности.

II. Класс VC **—** алгоритмы с фиксированной дополнительной памятью.

$$\forall n > 0, \forall D \in D_n \; V_t(D) = const \neq 0 .$$

Алгоритмы класса VC используют постоянное, и 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 класса функций в асимптотическом анализе. 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:

$$V_t^{\wedge}(n) = \Theta\left(n^2\right).$$

Примером может служить стандартный алгоритм умножения длинных целых чисел, заданных побитно массивами длиной n. Алгоритм умножения «в столбик» требует для двух исходных массивов длины n и массива результата длины $2 \cdot n$ дополнительного массива размерностью $2 \cdot n \times n = \Theta\left(n^2\right)$.

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:

$$\exists k > 2 : V_t\hat{}(n) = \Theta\left(n^k\right).$$

К этому классу относятся, например, рекурсивные алгоритмы, порождающее дерево рекурсии с оценкой глубины $\Theta\left(n^k\right), k > 1$, 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:

$$\exists \lambda > 0 : V_t\hat{}(n) = \Theta\left(e^{\lambda n}\right).$$

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 [4.18].

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:

$$\delta_V(D) = \frac{V_t(D)}{V_A(D)}, \quad \forall n \, \forall D \in D_n \quad 0 \le \delta_V(D) \le 1,$$

но предлагаемая типизация по асимптотической оценке абсолютных затрат дополнительной памяти представляется автору более целесообразной с точки зрения использования при разработке алгоритмического обеспечения программных средств и систем.

Список литературы к главе 4

[4.1] Хопкрофт Дж., Мотовани Р., Ульман Дж. Ведение в теорию автоматов, языков и вычислений, 2-е изд.: Пер. с англ. — М.: Издательский дом «Вильямс», 2002. — 528 с.

[4.2] Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ, 2-ое издание: Пер. с англ. — М.: Издательский дом «Вильямс», 2005. — 1296 с.

[4.3] Ахо А., Хопкрофт Дж., Ульман Дж. Структуры данных и алгоритмы: Пер. с англ.: — М.: Издательский дом «Вильямс», 2001. — 384 с.

[4.4] Гасанов Э. Э., Кудрявцев В. Б. Теория хранения и поиска информации. — М.: Физматлит, 2002. — 288 с.

[4.5] Успенский В. А. Машина Поста. — М.: Наука, 1979. — 96 с.

[4.6] Грэхем Р., Кнут Д., Паташник О. Конкретная математика. Основание информатики: Пер. с англ. — М.: Мир, 1998. — 703 с.

[4.7] Ульянов М. В. Дополнение к книге Дж. Макконелла Основы современных алгоритмов. — М.: Издательство Техносфера, 2004. С. 303–366.

[4.8] Гудман С., Хидетниеми С. Ведение в разработку и анализ алгоритмов. — М.: Мир, 1981. — 368 с.

[4.9] 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.

[4.10] Чмора А. Л. Современная прикладная криптография. — М.: Гелиос АРВ, 2001. — 256 с.

[4.11] Ульянов М. В. Классификация и методы сравнительного анализа вычислительных алгоритмов. Научное издание. — М.: Издательство физико-математической литературы, 2004. — 212 с.

[4.12] Головешкин В.А., Ульянов М. В. Теория рекурсии для программистов. М.: Издательство «Наука ФИЗМАТЛИТ», 2006. — 296 с.

[4.13] Бахвалов Н. С., Жидков Н. П., Кобельков Г. М. Численные методы. — М.: Лаборатория Базовых Знаний, 2001 г. — 632 с.

[4.14] 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.

[4.15] Кнут Д. Э. Искусство программирования, том 1. Основные алгоритмы, 3-е изд.: Пер. с англ. — М.: Издательский дом «Вильямс», 2002. — 720 с.

[4.16] Гмурман В. Е. Теория вероятностей и математическая статистика: Учеб. пособие для вузов –9-е изд., стер. — М.: Высш. шк., 2003. — 479 с.

[4.17] Ульянов М. В. Исследование и классификация вычислительных алгоритмов на основе чувствительности функции трудоемкости // Системы управления и информационные технологии. 2004. № 4 (16). С. 97–104.

[4.18] 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.

МЕТОДИКА СРАВНИТЕЛЬНОГО АНАЛИЗА И РАЦИОНАЛЬНОГО ВЫБОРА КОМПЬЮТЕРНЫХ АЛГОРИТМОВ

Введение

Рациональный выбор того или иного компьютерного 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 состоит в том, что для целого ряда вычислительных задач асимптотически более эффективный алгоритм имеет значительный коэффициент при главном порядке функции трудоемкости, и тем самым не всегда является эффективным на всем исследуемом диапазоне размерности входа. Реальные значения этого диапазона известны разработчикам программной системы на основе анализа области применения, т. е. определяются особенностями использования данного алгоритма в разрабатываемой программной системе. Такой подход может быть обобщен и на функцию ресурсной эффективности алгоритма.

Для выбора рационального алгоритма, в смысле ресурсной эффективности, необходимо детальное исследование претендующих алгоритмов в границах реального диапазона применения. При этом вполне вероятно, что на разных интервалах этого диапазона разные алгоритмы могут быть выбраны как наиболее рациональные. Таким образом, возможно адаптивное по размерности входа управление выбором алгоритма решения данной вычислительной задачи в проектируемой программной системе. Причина такого выбора обусловлена тем, что не всегда алгоритм, имеющий асимптотически оптимальную сложность (в смысле оценок O или Θ), имеет лучшие значения функции трудоемкости в целочисленных точках значений размерности реального множества исходных данных из-за существенного различия констант, скрывающихся за O или Θ асимптотическими оценками различных претендующих алгоритмов. Более того, обычно асимптотически лучшая производительность достигается за счет большего требуемого объема до-

полнительной памяти, и ухудшения коэффициента не только при первом, но и при втором аддитивном компоненте функции трудоемкости. В качестве примера сравним функции трудоемкости в среднем случае для двух алгоритмов сортировки [5.1]: $A1$ — сортировка вставками, $A2$ — сортировка слиянием:

$$\overline{f}_{A1}(n) = 2{,}5 \cdot n^2 + 11{,}5 \cdot n - 10; \quad \overline{f}_{A2}(n) = 18 \cdot n \cdot \log_2 n + 85 \cdot n - 66,$$

отметим, что в этом случае коэффициенты, как у главного порядка функции трудоемкости, так и у вторых компонент различаются более чем в 7 раз.

Описанию одного из возможных подходов к исследованию ресурсных функций компьютерных алгоритмов на практически значимом интервале размерностей входов и посвящена настоящая глава.

5.1. Угловая мера близости ресурсных оценок

В целях сравнительного анализа ресурсных функций компьютерных алгоритмов должна быть введена некоторая мера расхождения значений этих функций при заданном значении размерности n, которую в дальнейшем будем обозначать через $\pi(f(n), g(n))$. Сформулируем требования, которым должна удовлетворять мера расхождения значений функций $f(n)$ и $g(n)$ при фиксированном значении n:

— антисимметричность, т. е.

$$\pi(f(n), g(n)) = -\pi(g(n), f(n)),$$

отметим, что из требования антисимметричности следует равенство нулю значения меры при равных аргументах, т. е.

$$\pi(f(n), g(n)) = 0, \text{ если } f(n) = g(n).$$

Требование антисимметричности является очевидным в предположении, что функция меры должна быть положительна при большем первом аргументе;

— возможность угловой интерпретации значения меры $\pi(f(n), g(n))$; это требование связано с соблюдением единой концепции угловых мер, по аналогии с уже введенной угловой мерой асимптотического роста функций — $\pi(f(n))$. Заметим, что данное требование влечет за собой также требование ограниченности значений меры при любых значениях аргументов

$$|\pi(f(n), g(n))| \le const \quad \forall n > 0.$$

Может быть предложено несколько различных функций меры, удовлетворяющих данным требованиям. Для использования в анализе ресурсных функций на конечном интервале предлагается функция меры расхождения значений двух функций в точке $n = n_1$ в виде [5.2]:

$$\pi(a, b) = \operatorname{arctg}\frac{a}{b} - \operatorname{arctg}\frac{b}{a}, \text{ где } a = f(n_1), b = g(n_1). \qquad (5.1.1)$$

Значение введенной меры может быть интерпретировано как «угловое расхождение» значений аргументов, соответствующих длинам катетов прямоугольного треугольника относительно значения $\pi/4$. Поскольку значения ресурсных функций положительны для любого n и нигде не обращаются в ноль, то при таком определении меры значения $\pi(f(n), g(n))$ ограничены между

$$-\pi/2 < \pi(f(n), g(n)) < \pi/2.$$

Введенная мера обладает свойством антисимметричности и обращается в ноль при равенстве значений аргументов; поскольку значения меры ограничены, то выполнены все предъявляемые к мере $\pi(f(n), g(n))$ требования. Пороговое значение φ, при котором расхождение значений ресурсных функций является значимым может быть интерпретировано как максимальное значение «угла расхождения» значений этих ресурсных функций, при котором алгоритмы могут быть признаны равно применимыми на исследуемом интервале размерностей входа.

Заметим, что угловая мера $\pi(f(n), g(n))$ есть мера расхождения значений двух аргументов меры при фиксированном значении n. В целях анализа ресурсных функций сравниваемых алгоритмов можно рассматривать функцию $z = \pi(x, y)$, аргументами которой являются значения сравниваемых ресурсных функций.

5.2. Методика сравнительного анализа алгоритмов по ресурсным функциям

Принятие решения по рациональному выбору лежит в области сравнительного анализа компьютерных алгоритмов по ресурсным функциям и связано с выполнением следующих этапов [5.2]:

— детальный анализ ресурсной эффективности сравниваемых алгоритмов, т. е. получение функции трудоемкости и функции объема памяти в явном виде. Отметим, что асимптотической оценки сложности недостаточно для сравнительного анализа по функции трудоемкости;

— сравнительный анализ ресурсных функций претендующих алгоритмов с целью выбора рационального алгоритма решения данной задачи при реальных ограничениях на размерность множества исходных данных.

Реализация второго этапа — сравнительного анализа ресурсных функций — предполагает определение тех характеристик реального множества исходных данных задачи (D_A), при которых предпочтение может быть отдано одному из анализируемых алгоритмов. В ходе дальнейшего изложения будем считать, что таким характеристическим значением является n — размерность множества исходных данных.

Для класса количественно-зависимых алгоритмов (класс N) трудоемкость алгоритма (точное значение) полностью определяется размерностью множества исходных данных $f_A(D \in D_n) = f_A(n)$. Для подклассов количественно-параметрических алгоритмов со слабым и средним параметрическим влиянием (подклассы $NPRL$, $NPRE$) в качестве анализируемой функции трудоемкости может быть взято среднее значение — $\overline{f_A}(n)$, при условии малой информационной чувствительности. Для алгоритмов из количественно-параметрического подкласса $NPRH$ в качестве анализируемой функции трудоемкости может быть взято либо среднее значение — $\overline{f}_A(n)$, либо функция трудоемкости в худшем случае.

Сравнительный анализ ресурсных функций предполагает определение тех практически значимых интервалов аргумента n, при которых данный алгоритм может быть выбран в качестве рационального. В связи с этим предлагаемая ниже методика получила название методики анализа ресурсных функций на конечном интервале, по аналогии с асимптотическим анализом — определением сложности алгоритмов.

Исходными данными для анализа ресурсных функций на конечном интервале являются известные ресурсные функции (трудоемкости и объема памяти) ал-

горитмов, а результатами — предпочтительные интервалы характеристических значений множества исходных данных задачи для рационального применения того или иного алгоритма. Реально (в смысле дополнительных критериев целесообразности) выбор данного алгоритма, как предпочтительного, может быть произведен даже тогда, когда его ресурсная функция, например, трудоемкость на данном интервале несколько хуже, но не более чем на порог φ, по сравнению с трудоемкостью других алгоритмов [5.3]. Также возможно, что несколько алгоритмов могут быть использованы на этом интервале эквивалентно, с расхождением ресурсных функций не более чем на порог φ.

Такой подход может быть также обусловлен тем, что мы учитываем информационную чувствительность алгоритмов, в предположении, что можем ожидать наиболее вероятные значения трудоемкости в рамках порогового φ коридора. Это приводит к необходимости введения специальных обозначений в аппарате анализа ресурсных функций на конечном интервале, которые будут рассмотрены на примере функций трудоемкости.

Обозначения в анализе ресурсных функций на конечном интервале. Пусть функции трудоемкости двух алгоритмов заданы в явном виде функциями $f(n)$ и $g(n)$ соответственно — это может быть пара точных функций трудоемкости либо пара функций, описывающих поведение трудоемкости алгоритма для среднего или худшего случаев. Пусть задан также интервал (a,b) значений n, порог φ допустимого расхождения значений функций на заданном интервале и некоторая мера $\pi(f(n), g(n))$ расхождения значений функций при заданном значении n, тогда в соответствии с [5.3] будем говорить, что:

Обозначения 5.1

$$f(n) = \Delta_\varphi(g(n)), \text{ если } \quad \min\{\pi(f(n), g(n))\} \geq \varphi \qquad \forall n \in (a,b); \qquad (5.2.1)$$

$$f(n) = \Theta_\varphi(g(n)), \text{ если } \quad \max|\{\pi(f(n), g(n))\}| < \varphi \quad \forall n \in (a,b); \qquad (5.2.2)$$

$$f(n) = o_\varphi(g(n)), \text{ если } \quad \max\{\pi(f(n), g(n))\} \leq -\varphi \qquad \forall n \in (a,b). \qquad (5.2.3)$$

Таким образом, функция $f(n)$ есть $\Delta_\varphi(g(n))$ (дельта большое фи) на интервале (a,b) с порогом φ, если значение меры $\pi(f(n), g(n))$ во всех точках данного ин-

тервала больше или равно пороговому значению. Определение для Θ_φ (тета фи) предполагает, что значения функций на интервале (a, b) различаются менее чем на порог φ в рамках принятой меры. Функция $f(n)$ есть $o_\varphi(g(n))$ (о малое фи) на интервале (a, b) с порогом φ, если значение меры $\pi(f(n), g(n))$ во всех точках данного интервала меньше или равно пороговому значению, взятому с отрицательным знаком. Обозначение o_φ является симметрично противоположным обозначению Δ_φ — из того, что $f(n) = o_\varphi(g(n))$, следует в силу (5.2.1) и (5.2.3), что $g(n) = \Delta_\varphi(f(n))$.

Методика анализа ресурсных функций на конечном интервале. С учетом введенных обозначений может быть предложена следующая методика анализа ресурсных функций алгоритмов на конечном интервале [5.3]:

— получение в явном виде ресурсных функций анализируемых алгоритмов $f(n)$ и $g(n)$ для среднего или худшего случая, либо точных функций для алгоритмов класса N;

— определение интервала или сегмента изменения размерности множества исходных данных (n_1, n_2) на основе особенностей применения данного алгоритма в условиях конкретной программной системы;

— разбиение интервала (n_1, n_2) на подинтервалы, в каждой целочисленной точке которых явно выполняется одно из следующих соотношений для принятой в интервальном анализе меры $\pi(f(n), g(n))$:

— если $\varphi - \pi(f(n), g(n)) \le 0$, то $f(n) = \Delta_\varphi(g(n))$; предпочтительным на данном подинтервале является алгоритм, имеющий функцию трудоемкости $g(n)$;

— если $|\pi(f(n), g(n))| - \varphi < 0$, то $f(n) = \Theta_\varphi(g(n))$; оба алгоритма с точностью до порога φ могут быть равно применимы на этом подинтервале;

— если $\pi(f(n), g(n)) + \varphi \le 0$, то $f(n) = o_\varphi(g(n))$; предпочтительным на данном подинтервале является алгоритм, имеющий функцию трудоемкости $f(n)$.

На основе разбиения интервала размерности входа (n_1, n_2) на подинтрвалы и принимается решение о выборе рационального по ресурсной функции алгоритма для проектируемой программной системы.

Может быть предложена следующая графическая интерпретация введенной угловой меры, показанная на рисунке 5.1. Стрелки на рисунке обозначают те области угловой меры, в которых выполняются указанные соотношения между значениями ресурсных функций, например, для функций трудоемкости. Само значение меры $\pi(f(n), g(n))$ представляет собой центральный угол, отдельно на данном рисунке обозначены границы $(+\varphi, -\varphi)$ для тета фи области.

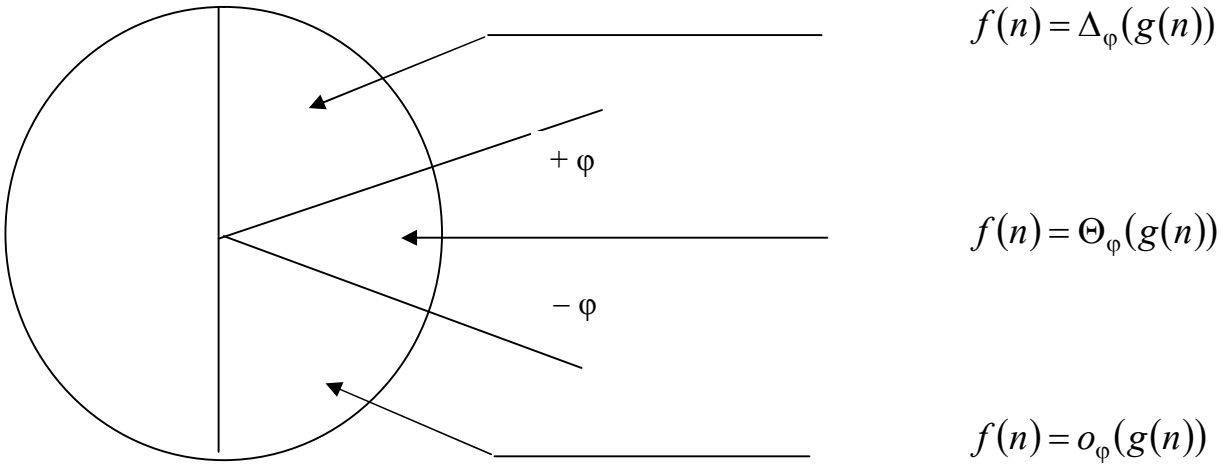


Рис. 5.1 Области угловых значений меры, с указанными соотношениями между функциями трудоемкости алгоритмов.

Пример сравнительного анализа алгоритмов по функции трудоемкости. В качестве примера на рисунке 5.2 приведены графики функций трудоемкости $f(n)$ и $g(n)$ для среднего случая следующих двух алгоритмов сортировки:

— алгоритма сортировки методом поиска максимума и минимума с трудоемкостью в среднем [5.3]: $f(n) = 1{,}75 \cdot n^2 + 2 \cdot n \cdot \ln n + 15 \cdot n + 9$;

— рекурсивного алгоритма сортировки слиянием с трудоемкостью в среднем [5.1] $g(n) = 18 \cdot n \cdot \log_2 n + 85 \cdot n - 66$.

Для указанных алгоритмов был выбран сегмент значений размерности массива $n_1 = 71, n_2 = 99$, таким образом, чтобы при пороговом значении $\varphi = \pi/32$

в нем содержалась бы область, в которой выполняется соотношение $f(n) = \Theta_{\varphi}(g(n))$.

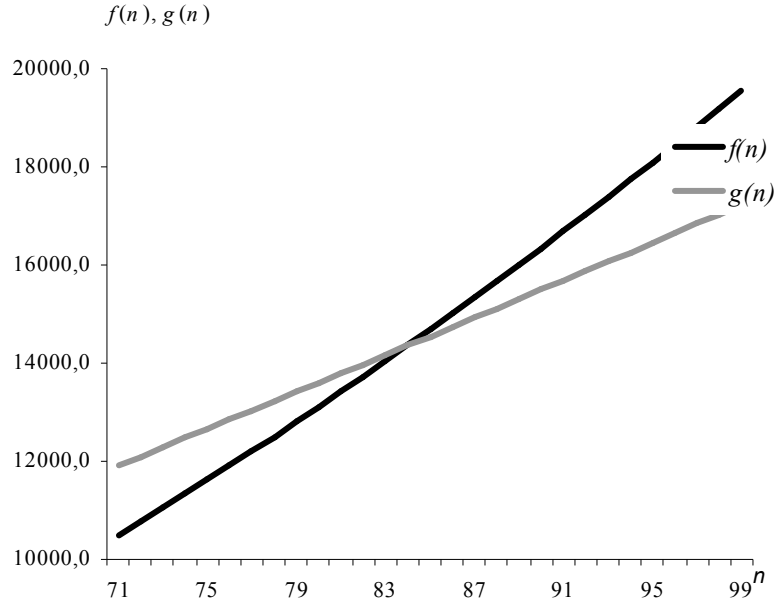


Рис. 5.2 Графики функций трудоемкости двух алгоритмов сортировки

В таблице 5.1 приведены значения функций трудоемкости алгоритма сортировки методом поиска минимума и максимума — $f(n)$ и алгоритма сортировки слиянием — $g(n)$, а также значения функции меры $\pi(f(n), g(n))$ [5.3]. В последнем столбце таблицы жирным шрифтом выделены те значения меры, которые не превышают порога, значение которого было выбрано равным $\varphi = \pi/32 \approx 0{,}0981$.

Таблица 5.1

n	$f(n)$	$g(n)$	$\pi(f(n), g(n))$
71	10501,051	11908,705	-0,1255
72	10776,840	12095,559	-0,1152
73	11056,157	12282,664	-0,1050
74	11339,002	12470,015	**-0,0949**
75	11625,373	12657,609	**-0,0850**
76	11915,271	12845,443	**-0,0751**
77	12208,696	13033,514	**-0,0653**
78	12505,647	13221,819	**-0,0557**
79	12806,123	13410,355	**-0,0461**
80	13110,124	13599,118	**-0,0366**
81	13417,651	13788,107	**-0,0272**
82	13728,702	13977,318	**-0,0179**
83	14043,278	14166,748	**-0,0088**
84	14361,377	14356,395	**0,0003**
85	14683,001	14546,256	**0,0094**

86	15008,148	14736,330	**0,0183**
87	15336,818	14926,612	**0,0271**
88	15669,011	15117,102	**0,0359**
89	16004,727	15307,795	**0,0445**
90	16343,966	15498,692	**0,0531**
91	16686,726	15689,788	**0,0616**
92	17033,009	15881,082	**0,0700**
93	17382,814	16072,572	**0,0783**
94	17736,139	16264,255	**0,0865**
95	18092,987	16456,129	**0,0947**
96	18453,355	16648,194	0,1028
97	18817,244	16840,445	0,1108
98	19184,654	17032,883	0,1187
99	19555,584	17225,504	0,1265

На основе проведенного анализа можно сделать вывод о том, что при выбранном значении $\varphi = \pi/32$ асимптотически более медленный квадратичный алгоритм сортировки методом поиска максимума и минимума является предпочтительным для сортировки массивов размерностью до 73. В интервале размерностей от 74 до 95 оба алгоритма равно применимы с точностью до выбранного значения φ в рамках применяемой меры. При количестве элементов массива более 95, рекурсивный алгоритм сортировки слиянием является более эффективным по функции трудоемкости. Таким образом, если в рамках разрабатываемой программной системы возникает задача сортировки коротких массивов (до 73 элементов) и в качестве вариантов рассматриваются указанные два алгоритма, то при условии предпочтения по трудоемкости рациональный выбор состоит в квадратичном алгоритме сортировки.

5.3. Области эквивалентной ресурсной эффективности компьютерных алгоритмов

Понятие области эквивалентной ресурсной эффективности алгоритмов. Аппарат анализа ресурсных функций на конечном интервале является важной составной частью сравнительного анализа ресурсной эффективности вычислительных алгоритмов. На его основе возможно более детальное исследование ресурсных функций сравниваемых алгоритмов, с целью выявления областей их эквивалентной эффективности. Задачами этого детального исследования является выяснение возможного количества и условий существования областей эквивалентной

ресурсной эффективности алгоритмов на исследуемом интервале размерностей. Под областями эквивалентной ресурсной эффективности будем понимать такие интервалы размерностей исходных данных задачи, в которых значения ресурсных функций сравниваемых алгоритмов различаются не более чем на заданный порог, в рамках принятой меры.

Обозначим через n длину входа алгоритма, а через $f_A(n)$ и $g_A(n)$ ресурсные функции сравниваемых алгоритмов. Более корректно под $f_A(n)$ и $g_A(n)$ понимаются или точные ресурсные функции для алгоритмов класса N, или ресурсные функции в среднем или худшем случае для алгоритмов из других классов. В рамках дальнейшего изложения будем считать, что аргумент x непрерывен, т. е. $f_A = f_A(x), g_A = g_A(x)$, а практически необходимые значения функций вычисляются в целочисленных точках $x = n$. Отметим, что поскольку функции $f_A(x), g_A(x)$ являются ресурсными функциями алгоритмов, то будем предполагать, что для них выполнены следующие условия [5.3], которые в рамках дальнейшего изложения будем обозначать как условия (5.3.1):

— $f_A(x) > 0, g_A(x) > 0$ при $x > 1$;

— функции $f_A(x), g_A(x)$ непрерывны и дважды дифференцируемы на любом сегменте $[n1, n2], n_2 > n_1 > 0$;

— производные функций $f_A(x), g_A(x)$ неотрицательны: $f'_A(x) \geq 0, g'_A(x) \geq 0$, т. е. содержательно ресурсные функции или монотонно возрастают, или, по крайней мере, являются неубывающими функциями размерности входа.

В соответствии с введенными определениями (см. параграф 5.2.) область, в которой значения ресурсных функций на интервале (a, b) различаются менее чем на порог φ в рамках принятой меры, обозначается как Θ_φ (тета фи), с учетом этого введем следующее определение [5.3]:

Определение 5.1. Будем говорить, в предположении о переходе к непрерывности аргумента (от n к x), что при заданном пороге φ на интервале (a, b) по отношению к функциям $f_A(x)$ и $g_A(x)$ *существует* Θ_φ*-область*, т. е. $f_A(x) = \Theta_\varphi(g_A(x))$, если

$$\max\left|\{\pi(f_A(x), g_A(x))\}\right| < \varphi \quad \forall x \in (a, b). \tag{5.3.2}$$

Произвольная точка x_0, принадлежащая исследуемому интервалу размерностей, принадлежит некоторой Θ_φ-области, если при заданном значении φ

$$\left|\{\pi(f_A(x_0), g_A(x_0))\}\right| < \varphi.$$

Ширина Θ_φ областей и их количество в исследуемом сегменте размерностей $[n_1, n_2]$ определяются как самими функциями $f_A(x)$ и $g_A(x)$ и принятой мерой $\pi(f_A(x), g_A(x))$, так и значением порога φ, определяющим порог равно применимости алгоритмов в разрабатываемой программной системе. Наличие или отсутствие на исследуемом сегменте $[n_1, n_2]$ Θ_φ-области определяется, очевидно, «близостью» значений функций $f_A(x)$ и $g_A(x)$ на этом интервале в рамках принятой меры.

Исследование областей эквивалентной ресурсной эффективности. Для дальнейшего исследования Θ_φ-областей введем в рассмотрение следующие функции:

$$u(x) = f_A(x) - g_A(x);$$

$$u_1(x) = |f_A(x) - g_A(x)|;$$

$$d(\varphi, x) = \begin{cases} 1, & |\pi(f_A(x), g_A(x))| < \varphi; \\ 0, & |\pi(f_A(x), g_A(x))| \geq \varphi; \end{cases}$$

$$h(x, \varepsilon) = u(x - \varepsilon) \cdot u(x + \varepsilon), \varepsilon > 0.$$

Поскольку функция $\pi(f_A(x), g_A(x))$ непрерывна в силу (5.3.1), то на заданном сегменте $[n_1, n_2]$ по второй теореме Вейерштрасса [5.4] найдется такая точка ζ, что значение меры $\pi(f_A(\zeta), g_A(\zeta))$ в этой точке равно точной нижней грани значений $\pi(f_A(x), g_A(x))$ на сегменте $[n_1, n_2]$

$$\exists \zeta : \pi(f_A(\zeta), g_A(\zeta)) = \min\{\pi(f_A(x), g_A(x))\}, x \in [n_1, n_2], \zeta \in [n_1, n_2].$$

Тогда, если $d(\varphi, \zeta) = 1$, то при заданном пороге φ на сегменте $[n_1, n_2]$ существует по крайней мере одна Θ_φ-область. Если $d(\varphi, \zeta) = 0$, то областей эквивалентной ресурсной эффективности нет, и один из алгоритмов на всем сегменте предпочтительнее другого с порогом φ. Предпочтительность определяется в зависимости от того, выполняется или нет одно из следующих равенств:

$$f(n)=\Delta_{\varphi}(g(n)) \text{ или } f(n)=o_{\varphi}(g(n)) \quad \forall n\in[n_1,n_2].$$

В случае если $d(\varphi,\zeta)=1$, то практически значимым является вопрос о том, как будет вести себя выявленная Θ_{φ} область при уменьшении значения порога. При уменьшении φ возможны два следующих случая:

— Θ_{φ} область сохраняется при любых сколь угодно малых значениях φ;

— Θ_{φ} область перестает существовать, начиная с некоторого значения φ.

В соответствии с этими рассуждениями введем следующие определения.

Определение 5.2. Соответствующую точке ζ область будем называть φ-*независимой* Θ_{φ} *областью*, если при $\varphi\to 0$

$$\exists\,\varepsilon>0,\delta>0: d(\varphi,x)=1, \forall x\in[\zeta-\varepsilon,\zeta+\delta].$$

Поскольку Θ_{φ} область зависит от вида функций $f_A(x)$ и $g_A(x)$ и меры, и не обязательно является симметричной относительно точки ζ, то значения ε и δ указывают длину левой и правой частей этой области, что показано на рисунке 5.3.

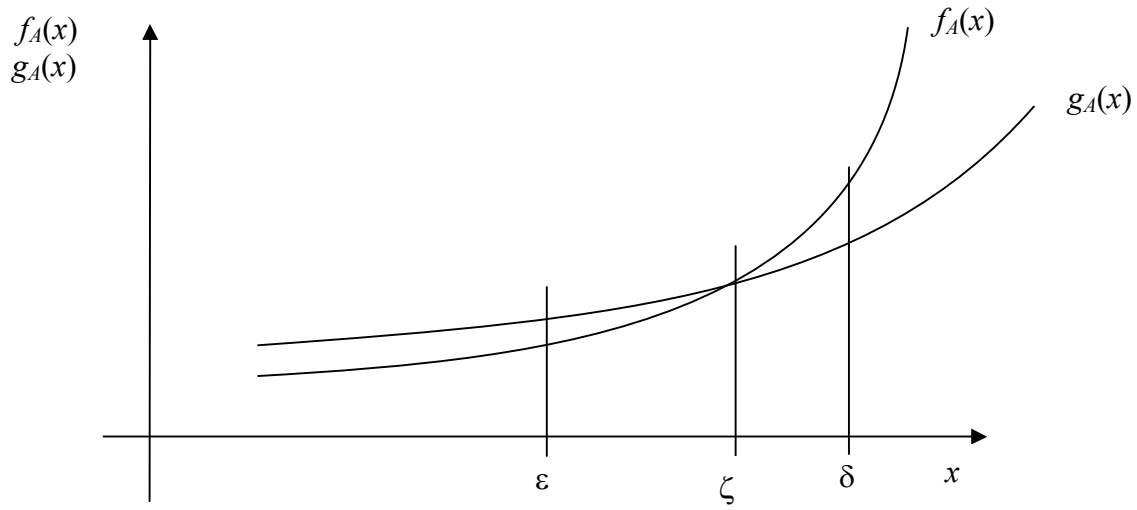


Рис. 5.3 Несимметричная Θ_{φ} область.

Определение 5.3

Соответствующую точке ζ область будем называть φ-*зависимой* Θ_{φ} *областью*, если

$$\exists\,\varphi_0:\forall\,\varphi<\varphi_0\ \exists\,\varepsilon>0,\delta>0: d(\varphi,x)=1, x\in[\,\zeta-\varepsilon,\zeta+\delta\,] \text{ и}$$

$$\forall\,\varphi\ge\varphi_0 \quad d(\varphi,\zeta)=0.$$

Утверждение 5.1

На сегменте $[\,n_1, n_2\,]$ φ-независимая Θ_φ область для функций $f_A(x)$, $g_A(x)$, удовлетворяющих условиям (5.3.1), существует в точке ζ, в которой $u_1(\zeta) = \min\{\,u_1(x)\,\}, x \in [\,n_1, n_2\,]$, тогда и только тогда, когда $u_1(\zeta) = 0$.

Доказательство

Пусть $u1(\zeta) \neq 0$, тогда вычислим значение меры $\pi(f_A(x), g_A(x))$ в точке ζ, выполнив следующие преобразования и считая для определенности, что $f_A(\zeta) > g_A(\zeta)$:

$$\pi(f_A(\zeta), g_A(\zeta)) = \operatorname{arctg}\frac{f_A(\zeta)}{g_A(\zeta)} - \operatorname{arctg}\frac{g_A(\zeta)}{f_A(\zeta)} = 2 \cdot \operatorname{arctg}\frac{f_A(\zeta)}{g_A(\zeta)} - \frac{\pi}{2} =$$

$$= 2 \cdot \operatorname{arctg}\left(1 + \frac{u_1(\zeta)}{g_A(\zeta)}\right) - \frac{\pi}{2}. \qquad (5.3.3)$$

Обозначим значение $\pi(f_A(\zeta), g_A(\zeta))$ через φ_0. Поскольку $\operatorname{arctg}(1) = \pi/4$, и если по предположению $u_1(\zeta) \neq 0$, то $u_1(\zeta) > 0$ по определению $u_1(x)$, тогда в силу (5.3.3) $\varphi_0 > 0$.

По определению φ-независимой Θ_φ области, функция $d(\varphi, \zeta) = 1$ при $\varphi \to 0$, но при нашем предположении о том, что $u_1(\zeta) \neq 0$, функция $d(\varphi, \zeta) = 0$, при $0 < \varphi < \varphi_0$, что и доказывает утверждение.

Выполнение условия $u_1(\zeta) = 0$ равносильно тому, что функции $f_A(x)$, $g_A(x)$ либо пересекаются в точке ζ, либо касаются в ней. Касание или пересечение может быть определено по знаку функции $h(\zeta, \varepsilon)$.

Если при $\varepsilon \to 0$ значение $h(\zeta, \varepsilon) > 0$, то функции касаются в точке ζ, если при $\varepsilon \to 0$ значение $h(\zeta, \varepsilon) < 0$, то пересекаются, обеспечивая в каждом случае φ-независимую Θ_φ область, как это показано на рисунке 5.4.

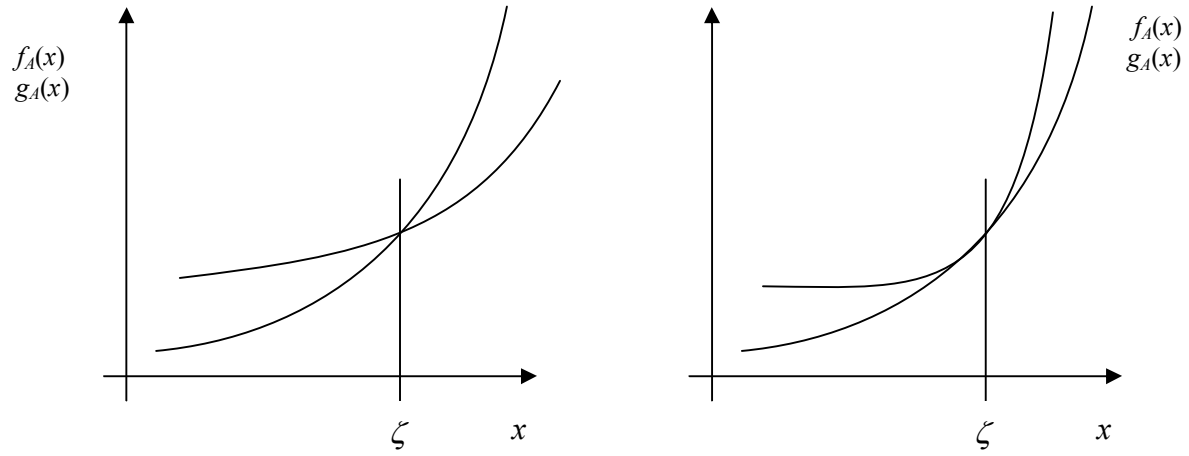


Рис. 5.4 Варианты φ-независимых Θ_φ областей.

Если функции $f_A(x),\ g_A(x)$ касаются в точке ζ, то сдвиг графика одной из функций по оси Y (изменение одной из функции на константу) приводит к образованию двух точек пересечения, и следовательно, к возможности существования двух φ-независимых Θ_φ-областей.

Заметим, что при увеличении порога φ две эти области могут быть объединены в одну. Детальное исследование возможного количества φ-независимых областей эквивалентной ресурсной эффективности алгоритмов связано со следующим утверждением [5.3].

Утверждение 5.2

Если функции $f_A(x),\ g_A(x)$ непрерывны и дважды дифференцируемы на сегменте $[a,b]$, и вторая производная функции $u(x)=f_A(x)-g_A(x)$ не обращается в нуль на этом сегменте — $u''(x)\neq 0, x\in[a,b]$, то функции $f_A(x),\ g_A(x)$ имеют на сегменте $[a,b]$ не более двух точек пересечения или одной точки касания.

Доказательство.

По теореме Ролля [5.4], если на сегменте $[a,b]$ функция $f(x)$ непрерывна и дифференцируема и $f(a)=f(b)$, то внутри сегмента найдется точка ζ, такая, что $f(\zeta)=0$. Предположим, что функция $u(x)$ трижды пересекает ось X на сегменте $[a,b]$ в точках x_1, x_2, x_3 (см. рисунок 5.5). Выделим внутри сегмента $[a,b]$ сегмент $[x_1,x_2]$, тогда по теореме Ролля $\exists\zeta_1\in[x_1,x_2]: u'(\zeta_1)=0$, на сегменте $[x_1,x_2]$ $\exists\zeta_2\in[x_1,x_2]: u'(\zeta_2)=0$. Применим теорему Ролля к функции $u'(x)$ на

сегменте $[\zeta_1, \zeta_2]$ — $\exists \zeta_3 \in [\zeta_1, \zeta_2] : u''(\zeta_3) = 0$, что противоречит условиям утверждения.

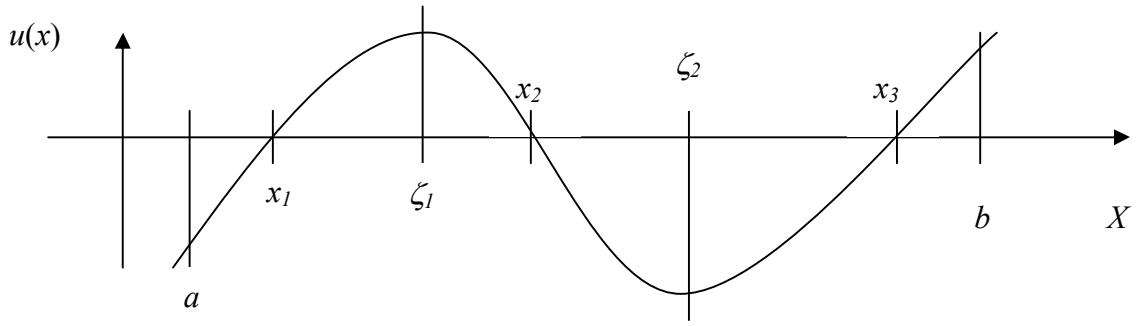


Рис. 5.5 Функция $u(x)$ с тремя точками пересечения оси X.

По аналогии можно провести рассуждения с двумя точками касания, которые будут являться точками локального экстремума функции $u(x)$.

Между этими точками по теореме Ролля должна находиться еще одна точка, в которой производная функции $u(x)$ обращается в ноль. Таким образом, на сегменте $[a, b]$ будет три точки, в которых $u'(x) = 0$, и, следовательно, две точки, в которых $u''(x) = 0$. Аналогично можно показать, что условие $u''(x) \neq 0$ несовместимо с одной точкой пересечения и одной точкой касания функцией $u(x)$ оси X на сегменте $[a, b]$, что полностью доказывает утверждение.

Практическое применение результата состоит в том, что для некоторых ресурсных функций алгоритмов на исследуемом интервале размерностей возможно существование двух φ-независимых Θ_φ-областей при определенном значении порога φ, что необходимо учитывать при проведении сравнительного анализа алгоритмов. Одна из возможных ситуаций разбиения исследуемого интервала на области эквивалентной ресурсной эффективности и области предпочтения приведена на рисунке 5.6.

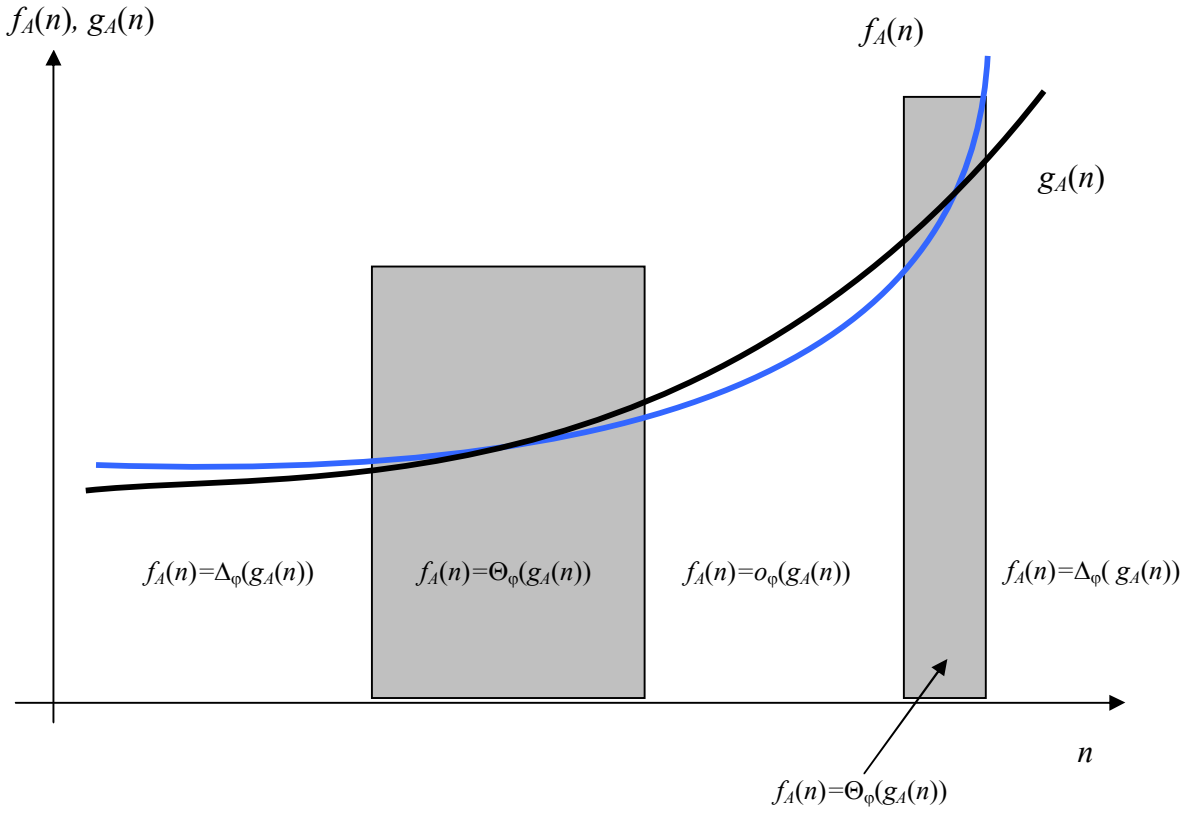


Рис. 5.6 Возможные области эквивалентной ресурсной эффективности алгоритмов (выделены серым цветом).

Если функции ресурсной эффективности были получены в ходе анализа алгоритмов в явном функциональном виде, то возможно предварительное исследование взаимного поведения этих функций, с целью выявления возможных φ-независимых Θ_φ областей около точек пересечения. Ширина этих областей будет определяться пороговым значением, заданным разработчиками программной системы. Если данное исследование проводится с ресурсными функциями в среднем, то возможен учет компонентов информационной чувствительности, например среднеквадратического отклонения. В этом случае в качестве области эквивалентной ресурсной эффективности можно принять область, образованную пересечением двух «коридоров» ресурсных функций исследуемых алгоритмов с σ отклонениями [5.3].

Список литературы к главе 5.

[5.1] Головешкин В.А., Ульянов М. В. Теория рекурсии для программистов. М.: Издательство «Наука ФИЗМАТЛИТ», 2006. — 296 с.

[5.2] Ульянов М. В. Классификация и методы сравнительного анализа вычислительных алгоритмов. Научное издание. — М.: Издательство физико-математической литературы, 2004. — 212 с.

[5.3] Ульянов М. В. Определение областей рационального применения алгоритмов на основе исследования функций трудоемкости // Вестник Рязанской государственной радиотехнической академии. 2004. № 15. С. 32–39.

[5.4] Ильин В. А., Садовничий В. А., Сендов Бл. Х. Математический анализ. — М.: Наука. Главная редакция физико-математической литературы, 1979. — 720 с.

РАЗДЕЛ III

МЕТОДЫ РАЗРАБОТКИ ЭФФЕКТИВНЫХ АЛГОРИТМОВ

В этой главе даётся краткое описание ряда методов разработки алгоритмов, некоторые из которых, например, метод декомпозиции, позволяют получить алгоритмы, достаточно эффективные в смысле асимптотических оценок трудоемкости. Прежде всего, хотелось бы отметить, что 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 известные математические методы, например, метод Рунге-Кутты. 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 остовного 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 настоящем разделе, 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, равно как и показателей качества получаемых алгоритмов. Под этим термином понимаются приемы или способы, пользуясь которыми может быть можно построить алгоритм решения задачи при условии, что, во-первых, сам метод теоретически применим к этой задаче, и, во-вторых, при условии, что Вам удастся адаптировать этот метод к ее особенностям. Речь скорее идет о некоторых общих подходах, не

содержащих рецептов конкретного применения, на базе которых, вероятно, могут быть разработаны новые алгоритмы. В этом случае конкретное применение такого метода по отношению к реальной задаче требует специальных подходов и математических обоснований — и снова приходится констатировать интеллектуальный аспект в процессе разработки алгоритмов.

Г Л А В А 6.

У Н И В Е Р С А Л Ь Н Ы Е М Е Т О Д Ы Р А З Р А Б О Т К И А Л Г О Р И Т М О В

Введение

«Универсальные» методы разработки алгоритмов — это методы, которые с одной стороны не накладывают никаких серьезных ограничений на возможности их применения, а с другой стороны — не содержат никаких конкретных рецептов по отношению к определенной задаче. Их конкретная «доводка» — дело рук и ума разработчика. В этой главе будут изложены три «универсальных» и достаточно распространенных метода, приводящие к построению рекурсивных алгоритмов — метод рекуррентных соотношений, метод декомпозиции и метод динамического программирования, который хотя и ориентирован на решение задач целочисленной оптимизации, может быть использован и для ряда других задач. Слово «универсальный» недаром поставлено 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. Если Вы реально занимались разработкой алгоритмов, то Вы, наверное, ощущаете разницу двух процессов — алгоритмирования, как интеллектуального процесса разработки алгоритма решения задачи, и программирования, как технического процесса записи уже готового алгоритма на некотором языке программирования.

6.1. Метод декомпозиции задачи

Чтобы перенести сотню кирпичей с одного места на другое, Вы, скорее всего, будете переносить за один раз несколько кирпичей, чем всю сотню сразу. Таким образом, вместо того чтобы решать задачу сразу, несколько раз решается более простая задача с понижением размерности — собственно говоря, эта интуитивно здравая идея и лежит в основе метода декомпозиции. Другое название этого метода — метод «разделяй и властвуй» [6.1]. 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 — каждая подзадача рассматривается как задача определенной размерности, которая разделяется на собственные подзадачи, путем повторного выполнения шага 1, что приводит к рекурсии, до тех пор, пока не будет получена такая размерность, при которой решение может быть получено 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 [6.2] — 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 приводит каждая из

этих идей на этапе объединения решений, который, как правило, является наиболее сложным при реализации этого метода.

Основное преимущество метода декомпозиции состоит в том, что он позволяет получить алгоритмы с хорошими асимптотическими оценками трудоемкости. Именно этот подход позволил А.А. Карацубе в 1962 г. получить алгоритм умножения длинных целых чисел с оценкой, лучше, чем n^2, а В. Штрассену — алгоритм умножения квадратных матриц с оценкой, лучше, чем n^3 [6.1]. Соответствующее обоснование асимптотических оценок трудоемкости алгоритмов, разработанных методом декомпозиции, читатель найдет в главе 9.

Приведем два примера применения метода декомпозиции.

Пример 6.1. Исторически одним из первых применений метода считается алгоритм сортировки слиянием, приписываемый Дж. фон Нейману — этот алгоритм будет подробно рассмотрен и модифицирован в разделе V. Рассмотрим задачу сортировки массива чисел, состоящего из n элементов, и обсудим шаги метода декомпозиции, применительно к этой задаче.

1. Шаг разделения задачи. Организуется разделение текущей задачи сортировки на две подзадачи — сортируемый массив из n элементов разделяется на два массива, которые содержат $\lfloor n/2 \rfloor$ 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 равном 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 $\Theta(n \cdot \ln n)$. 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 6.2. 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 уравнениями

$$\begin{aligned} r &= a \times e + b \times g; \\ s &= a \times f + b \times h; \\ t &= c \times e + d \times g; \\ u &= c \times f + d \times h. \end{aligned} \qquad (6.1.1)$$

Эти уравнения приводят к необходимости решения восьми подзадач размерностью $n/2$. Операции умножения в (6.1.1) 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 (6.1.1), 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 (6.1.1) 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

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

Асимптотическая оценка решения этого рекуррентного соотношения $S(n) = \Theta\left(n^3\right)$ может быть получена методом, изложенным в главе 9, 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 — $\Theta\left(n^{\log_2 7}\right)$. 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 [6.1].

В целом с использованием метода декомпозиции в настоящее время получены эффективные алгоритмы решения целого ряда задач. Основной их особенностью является то, что «платой» за асимптотически лучшую функцию в оценке сложности является существенно больший, чем у других алгоритмов, коэффициент при главном порядке. Тем самым эти алгоритмы становятся рационально применимыми, начиная с относительно больших длин входов.

6.2. Метод рекуррентных соотношений

Идея метода рекуррентных соотношений чрезвычайно проста — на основе некоторых рассуждений, строится рекуррентное соотношение, решение которого доставляет решение нашей задачи, и на основе этого рекуррентного соотношения разрабатывается 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. Приведем примеры использования этого метода с комментариями к этапам разработки рекуррентного соотношения.

Пример 6.3. Рассмотрим задачу вычисления количества разбиений положительного целого числа. Разбиение положительного целого числа m — это его представление в виде суммы целых положительных чисел. Классической счетной задачей является определение функции, задающей количество разбиений числа m без учета порядка слагаемых [6.3]. Приведем все разбиения числа $m = 6$ в порядке убывания первого слагаемого, прямой подсчет дает значение.

$6;$

$5+1;$

$4+2, \quad 4+1+1;$

$3+3, \quad 3+2+1, \quad 3+1+1+1;$

$2+2+2, \quad 2+2+1+1, \quad 2+1+1+1+1;$

$1+1+1+1+1+1.$

К сожалению, источник [6.3] 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 [6.3] (заметим, что рекурсивные рассуждения требуют рекурсивного мышления). Эти рассуждения приводят к следующим пяти уравнениям:

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

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

3. $Q(m,n)=Q(m,m), m \le n$. 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 $n \le m-1$.

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 пять уравнений определяют рекурсивно заданную функцию $Q(m,n)$ следующим рекуррентным соотношением

$$\begin{cases} Q(m,1)=1; \\ Q(1,n)=1; \\ Q(m,n)=Q(m,m), m \le n; \\ Q(m,m)=1+Q(m,m-1), n=m; \\ Q(m,n)=Q(m,n-1)+Q(m-n,n). \end{cases} \qquad (6.2.1)$$

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

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

Заметим, что поскольку функция `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 `Q(m,n)`, и, следовательно, — решаемой задачи.

Вопрос о том, насколько велики вычислительные затраты этого алгоритма пока не обсуждается. Более интересный вопрос состоит в том, существует ли другой алгоритм, может быть не рекурсивный, который вычисляет значение функции `Q(m,n)` быстрее? 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 можно говорить о том, что, например, для чисел Фибоначчи получаются неоправданно большие вычислительные затраты, а при вычислении факториала — аналогичная итерационной реализации асимптотическая оценка.

Пример 6.4. Этот пример приводится 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 их аксиом Пеано [6.4] «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, что бы 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

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

Надеемся, что читатели без труда разработают соответствующий рекурсивный алгоритм. Еще целый ряд интересных примеров разработки алгоритмов методом рекуррентных соотношений читатели смогут найти в [6.5].

6.3. Метод динамического программирования

Метод динамического программирования был предложен и обоснован Р. Беллманом в начале 1960-х годов [6.6]. Первоначально метод создавался в целях существенного сокращения перебора для решения целого ряда задач экономического характера, формулируемых в терминах задач целочисленного программирования [6.2]. Однако Р. Беллман и Р. Дрейфус в [6.6] 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 полностью универсальным, поскольку условия его применения требуют, чтобы целевой функционал представлял собой аддитивную функцию, т. е.

$$f(\mathbf{x}) = \sum_{i=1}^{n} g_i(x_i), \mathbf{x} = (x_1, \ldots, x_n), \qquad (6.3.1)$$

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 [6.6]. 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 некото-

рый 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 (6.3.1) — $f(\mathbf{x})\to\max$, 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 [6.6].

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 $\{\, f_i(c)\,\},\ i=\overline{1,n}\,,\ 0\le c\le C$, следующим образом

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

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 $c\ge 0$ 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 остающихся $m-1$ 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 [6.6]

$$\begin{cases} f_1(c) = g_1(c); \\ f_m(c) = \max\limits_{0 \le x_m \le c} [g_m(x_m) + f_{m-1}(c - x_m)], \forall m = \overline{2, n}, \quad c \ge 0. \end{cases} \tag{6.3.2}$$

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 допускать построение рекуррентно связанных между собой функций Беллмана через возможно меньшее число аргументов, которые определяются «существенными» ограничениями задачи.

Рекуррентное соотношение (6.3.2) 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.

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 аргументов или линейных комбинаций этих значений с другими функциями. Эту ситуацию автор хочет проиллюстрировать на одном из классических примеров применения метода динамического программирования — на задаче вычисления редакционного расстояния.

Пример 6.5. 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, одни из них, наиболее общий и простой, носит название редакционного расстояния. Термин и идея предложены В.И. Левенштейном [6.7] 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 определяется как 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 [6.8], введем следующие обозначения. Будем предполагать, что исходные строки символов S_1 и S_2 заданы посимвольно массивами $S_1[1\ldots n]$ и $S_2[1\ldots m]$. Введем в рассмотрение функцию $D(i, j)$, значением которой является редакционное расстояние между подстроками $S_1[1\ldots i]$ и $S_2[1\ldots j]$. 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 $D(i, j), i = \overline{0, n}, j = \overline{0, m}$. Непосредственное вычисление функции $D(i, j)$ возможно в том случае, если одна из строк является пустой, тогда

$$\begin{aligned} D(i, 0) &= i; \\ D(0, j) &= j. \end{aligned} \tag{6.3.3}$$

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 динамического программирования определяется следующей теоремой, доказательство которой приводится в соответствии с [6.8].

Теорема. Если значения i и j строго положительны, то

$$D(i, j) = \min\{\, D(i-1, j)+1, \quad D(i, j-1)+1, \quad D(i-1, j-1)+t(i, j)\,\}, \tag{6.3.4}$$

где

$$t(i, j) = 1, S_1[i] \neq S_2[j]; \quad t(i, j) = 0, S_1[i] = S_2[j].$$

Доказательство. Рассмотрим редакционное предписание, преобразующее подстроку $S_1[1\ldots i]$ в $S_2[1\ldots j]$ 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 $S_1[1\ldots i]$ в

$S_2[1\ldots j-1]$. 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 $S_1[1\ldots i]$, то предыдущие операции предписания должны оптимально преобразовывать $S_1[1\ldots i-1]$ в $S_2[1\ldots j]$. 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 $S_1[1\ldots i-1]$ в $S_2[1\ldots j-1]$. 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 $S_1[1\ldots i]$ в $S_2[1\ldots j]$ — можно преобразовать $S_1[1\ldots i]$ в $S_2[1\ldots j-1]$ 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 $S_1[1\ldots i-1]$ в $S_2[1\ldots j]$ 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 (6.3.4) 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

$$\begin{cases} D(i,0) = i\,, j = 0; \\ D(0,\,j) = j\,, i = 0; \\ D(i,\,j) = \min\{\, D(i-1,\,j)+1,\;\; D(i,\,j-1)+1,\;\; D(i-1,\,j-1)+t(i,\,j)\,\}, i > 0, j > 0. \end{cases} \qquad (6.3.5)$$

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 читатель найдет в [6.8].

Следует обратить 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, но не полиномиальное время, точное решение целого ряда переборных задач, сводящихся к задачам целочисленного программирования [6.2]. Отметим также, что для каждой конкретной задачи метод должен быть детально разработан, т. е. сформулировано основное функциональное уравнение метода динамического программирования применительно к данной задаче. Непосредственная реализация основного функционального уравнения приводит к созданию рекурсивного алгоритма, а переход к итерационной реализации требует использования специальных структур хранения промежуточных результатов, например, таблиц. Его применение к решению задачи оптимальной по стоимости одномерной упаковки будет подробно описано в разделе V.

Задачи и упражнения к главе 6

6.1. 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 рекур-

рентное соотношение, задающее функцию $G(n)$. Какие аналогии с уже известными рекурсивно заданными функциями Вы можете провести?

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

6.3. Разработайте методом декомпозиции алгоритм построения выпуклого охватывающего контура для n точек, заданных своими координатами. Условие останова рекурсии может быть сформулировано, например, на основе следующих рассуждений — 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?

6.4. 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.

6.5. 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 $D(20,25)$?

Список литературы к главе 6

Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ, 2-ое издание : Пер. с англ. — М.: Издательский дом «Вильямс», 2005. — 1296 с.

Хаггарти Р. Дискретная математика для программистов. — М.: Техносфера, 2005. — 400с.

Гудман С., Хидетниеми С. Ведение в разработку и анализ алгоритмов. — М.: Мир, 1981. — 368 с.

Колмогоров А.Н., Драгалин А.Г. Математическая логика. — М.: Едиторал УРСС. 2005. —240 с.

Баррон Д. Рекурсивные методы в программировании. — М.: Мир, 1974. – 79 с.

Беллман Р., Дрейфус Р. Прикладные задачи динамического программирования: Пер. с англ. — М.: Наука, 1965, — 457 с

Левенштейн В.И. Двоичные коды с исправлением выпадений, вставок и замещений символов // Доклады АН СССР. 1965. Т. 163. С. 707-710.

Гасфилд Д. Строки, деревья и последовательности в алгоритмах: Информатика и вычислительная биология / Пер с англ. — СПб.: Невский диалект; БХВ–Петербург, 2003. — 654 с.

Г Л А В А 7 .

М Е Т О Д Ы Р А З Р А Б О Т К И А Л Г О Р И Т М О В Р Е Ш Е Н И Я О П Т И М И З А Ц И О Н Н Ы Х З А Д А Ч

Введение

Достаточно широким классом задач, необходимость решения которых часто возникает на практике, является класс оптимизационных задач. В рамках этого класса особый интерес в смысле разработки алгоритмического обеспечения представляют задачи, связанные с поиском экстремальных значений целевой функции на ограниченном множестве целочисленных точек многомерного пространства. Такого рода задачи часто называют задачами перебора, более корректно — это задачи целочисленного программирования.

Прежде, чем перейти к описанию постановки задачи целочисленного программирования и изложению методов ее решения, автор хотел бы сделать замечание о том, что он принимает расширенное толкование термина «задача целочисленного программирования». Под этим термином автор понимает задачу нахождения экстремума любой вычислимой вещественнозначной функции, определенной в некоторой области m-мерного евклидова целочисленного пространства при ограничениях (условиях) общего вида, задающих такое сужение области определения функции, которое заключено в некоторый конечный m-мерный целочисленный координатный куб.

Для этих задач в последние 50 лет предложен целый ряд методов разработки алгоритмов, позволяющий получить как их точные, так и эвристические решения, ознакомительному изложению которых и посвящен материал данной главы.

7.1. Постановка и примеры задач целочисленного программирования

Понятие *m*-мерного евклидова целочисленного пространства. При изложении задач целочисленного программирования удобно использовать геометрическую терминологию, обобщающую наше представление о трехмерном пространстве. Поскольку в рассматриваемых задачах результатом является набор целых, как правило, неотрицательных чисел, то нас будет интересовать многомер-

ное целочисленное пространство. Опишем вначале классическое m-мерное евклидово пространство [7.1].

Будем назвать *m-мерным координатным пространством* множество упорядоченных совокупностей $(x_1, x_2, \cdots, x_m)$, состоящих из m вещественных чисел $x_1, x_2, \cdots, x_m$, и обозначать это пространство символом A^m. Каждую такую упорядоченную совокупность будем называть *точкой* m-мерного координатного пространства, обозначая ее через $\mathbf{x}$, а числа $x_1, x_2, \cdots, x_m$ будем называть *координатами точки* $\mathbf{x}$. Если рассматривать m-мерное координатное пространство как множество *векторов* $\mathbf{x}$, ввести понятие суммы векторов как вектор с покоординатной суммой и определить произведение вектора на вещественное число как покоординатное произведение, то мы получим m-мерное линейное пространство. Координатное пространство A^m называется *m-мерным евклидовым пространством*, если между любыми двумя точками $\mathbf{x}, \mathbf{y}$ m-мерного пространства A^m определено расстояние, обозначаемое символом $\rho(\mathbf{x}, \mathbf{y})$ и выражающееся соотношением

$$\rho(\mathbf{x}, \mathbf{y}) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \cdots + (x_n - y_n)^2} \qquad (7.1.1)$$

Общеупотребительным для m-мерного евклидового пространства является обозначение E^m. Пространство, в котором указано правило, ставящее в соответствие любым двум элементам $\mathbf{x}, \mathbf{y}$ вещественное число, называемое расстоянием между этими элементами — $\rho(\mathbf{x}, \mathbf{y})$, которое удовлетворяет следующим трем аксиомам:

1) $\rho(\mathbf{x}, \mathbf{y}) = \rho(\mathbf{y}, \mathbf{x})$;

2) $\rho(\mathbf{x}, \mathbf{y}) \le \rho(\mathbf{x}, \mathbf{z}) + \rho(\mathbf{z}, \mathbf{x})$;

3) $\rho(\mathbf{x}, \mathbf{y}) \ge 0, \quad \rho(\mathbf{x}, \mathbf{y}) = 0 \Leftrightarrow \mathbf{x} = \mathbf{y}$,

называется *метрическим пространством*. Легко проверить, что расстояние, введенное формулой (7.1.1) 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 [7.1] и [7.2].

Но пространство E^m содержит точки, координаты которых являются вещественными числами. В целях формализации задач целочисленного программиро-

вания нам необходимо подмножество пространства E^m, содержащее только те точки из E^m, которые имеют целочисленные координаты. Будем называть это подмножество *m-мерным евклидовым целочисленным пространством*, и обозначать его через E_z^m:

$$E_z^m = \left\{ \mathbf{x} \mid \mathbf{x} \in E^m, \mathbf{x} = (x_1, x_2, \cdots, x_m), \quad x_i \in Z \ \ \forall i = \overline{1,m} \right\},$$

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 $\mathbf{y}$ из E_z^m, координаты $(y_1, y_2, \cdots, y_m)$ которых удовлетворяют неравенствам

$$y_1 - x_1 \le k, y_2 - x_2 \le k, \cdots, y_m - x_m \le k, \quad y_i \ge x_i \ \forall i = \overline{1,m}, \qquad (7.1.2)$$

будем называть *m-мерным координатным кубом* с ребром k с началом в точке $\mathbf{x} = (x_1, x_2, \cdots, x_m)$, $\mathbf{x} \in E_z^m$, и обозначать его через $Cub_z^m(\mathbf{x}, k)$. Поскольку многие задачи целочисленного программирования приводят к перебору точек m-мерного координатного куба с началом в точке $\mathbf{0} = (0, \cdots, 0)$, т. е. куба $Cub_z^m(\mathbf{0}, k)$, то представляется целесообразным ввести специальное обозначение: $K_z^m = Cub_z^m(\mathbf{0}, k)$. Таким образом, например, 1_z^m — это m-мерный координатный куб с началом в нуле, все точки которого имеют целочисленные координаты — 0 или 1.

Аналогично, множество всех точек Y из E_z^m, координаты $(y_1, y_2, \cdots, y_m)$ которых удовлетворяют равенству $\rho(\mathbf{x}, \mathbf{y}) = r$, будем называть *$m$-мерной сферой* радиуса r с центром в точке $\mathbf{x} = (x_1, x_2, \cdots, x_m)$, $\mathbf{x} \in E_z^m$, и обозначать ее через $S_z^m(\mathbf{x}, r)$.

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 (7.1.2) 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 $\left|Cub_z^m(\mathbf{x},k)\right| = (k+1)^m$ вне зависимости от начальной точки куба. В частности, даже куб 1_z^m содержит 2^m точек. Этот неутешительный результат и заставляет искать эффективные алгоритмы решения задач целочисленного программирования, существенно сокращающие перебор.

Общая постановка задачи целочисленного программирования. Пусть задано подмножество G в целочисленном пространстве E_z^m. На точках $\mathbf{x}$ подмножества G определена вещественнозначная функция $f(\mathbf{x}): G \to R$, где R — множество действительных чисел. Эту функцию будем в дальнейшем, следуя [7.2], называть *целевым функционалом задачи*. В литературе встречаются также равно положенные термины — показатель качества решения задачи, функция потерь, критерий эффективности и т. д. Пусть заданы также ограничения (неравенства) на значения вещественнозначных функций $g_i(\mathbf{x}): G \to R, i = \overline{1,k}$ в виде

$$g_i(\mathbf{x}) \le a_i \text{ , } a_i \in R, i = \overline{1,k} \text{ ,}$$

тогда общая постановка задачи целочисленного программирования имеет вид

$$f(\mathbf{x}) \to \min \text{ , при } g_i(\mathbf{x}) \le a_i \text{ , } a_i \in R, i = \overline{1,k} \text{ , } \mathbf{x} \in G. \qquad (7.1.3)$$

В задаче требуется найти такой вектор $\mathbf{x}$, удовлетворяющий ограничениям — неравенствам, и принадлежащий подмножеству G, который доставляет минимум целевому функционалу $f(\mathbf{x})$. Решить задачу (2.3) точно — значит найти какой-нибудь, или все векторы $\mathbf{x}$, удовлетворяющие (7.1.3), или доказать, что задача несовместна.

Типизация задач целочисленного программирования. Алгоритмы и методы решения задач вида (7.1.3) существенно зависят от того, что представляет из себя область G, какой вид имеет целевой функционал и какова структура функций ограничений. По структуре целевого функционала, области определения и функциям ограничения имеет место следующая типизация задач целочисленного программирования:

— задачи линейного целочисленного программирования, в которых $f(\mathbf{x})$ и $g_i(\mathbf{x})$ являются линейными функционалами;

— задачи *нелинейного* целочисленного программирования, в которых хотя бы один из функционалов $f(\mathbf{x})$ или $g_i(\mathbf{x})$ является нелинейным, достаточно часто возникающие в различных прикладных областях.

Из нелинейных задач наиболее полно разработан класс задач выпуклого программирования, в которых множество G является выпуклой областью, а целевой функционал и функции ограничений — выпуклые функции. В этом классе выделяются также задачи квадратичного выпуклого целочисленного программирования. Напомним, что множество точек в пространстве E^m называется *выпуклым*, если вместе с любой парой своих точек, оно содержит и весь соединяющий их отрезок. Непрерывная функция $f(x)$, определенная на выпуклом множестве G называется выпуклой вниз, если для всех $x, y \in G$ справедливо неравенство

$$\forall \alpha, 0 \le \alpha \le 1, \quad f(\alpha x + (1-\alpha)y) \le \alpha f(x) + (1-\alpha) f(y).$$

Примеры задач целочисленного программирования. Ниже приводится ряд примеров постановок задач целочисленного программирования, которые имеют важное практическое значение.

Задача оптимального резервирования. Пусть некоторая система состоит из n основных элементов, относящихся к n разным типам. Каждый элемент может быть резервирован в целях повышения надежности функционирования всей системы в целом. Пусть значение x_i есть число элементов i-ого типа, включенных в систему, $x_i \ge 1, i = \overline{1,n}$. Один из этих элементов находится в штатном режиме функционирования, остальные находятся в резерве. Мы предполагаем, что заданы функции $f_i(x_i)$, дающие значение показателя надежности системы по элементам типа i, в зависимости от их количества. Если известны вероятности отказа элемента типа i — p_i, то функция $f_i(x_i)$ может иметь, например, следующий вид

$$f_i(x_i) = 1 - p_i^{x_i}.$$

Мы хотим максимизировать целевой функционал, отражающий показатель надежности системы в целом, при условии, что существует ограничение на сум-

марные затраты. В предположении, что показатель надежности системы есть произведение показателей надежности по типам элементов, а значение c — есть заданное ограничение на затраты, задача оптимального резервирования есть задача определения числа резервных элементов каждого типа, максимизирующих нелинейный целевой функционал при заданных ограничениях

$$\prod_{i=1}^{n} f_i(x_i) \to \max, \quad \sum_{i=1}^{n} a_i \cdot x_i \le c, \quad \mathbf{x} = (x_1, \cdots x_n) \in E_z^n, x_i \ge 1.$$

Отметим, что особенностью данной задачи является сепарабельность целевого функционала и ограничений — изменение показателя надежности и показателя затрат при изменении значений по одному типу элементов не зависит от значений остальных переменных. Иными словами изменение значения x_i не влияет на другие значения $x_j, j \ne i$. Задачи с сепарабельными функционалами и ограничениями могут быть решены методом динамического программирования.

Задача оптимального снабжения. Нам необходимо организовать рациональное снабжение предприятия, занимающегося сборкой, комплектующими деталями в контейнерах, которые изготавливаются и доставляются из других городов. Мы предполагаем, что в снабжении предприятия участвуют n городов. Пусть x_i— есть количество контейнеров, поставляемых из i-ого города, а c_i — суммарная стоимость производства и доставки одного контейнера деталей из города i, $i = \overline{1,n}$. Тогда стоимость всех привезенных контейнеров задается линейной функцией

$$f(\mathbf{x}) = \sum_{i=1}^{n} c_i \cdot x_i .$$

Потребность предприятия в комплектующих деталях определяется величиной b (контейнеров), что приводит к первому ограничению

$$\sum_{i=1}^{n} x_i = b .$$

Производство деталей в городе i ограничено величиной b_i, а обратные перевозки исключены, что приводит к очевидным ограничениям. В этих предположениях постановка задачи имеет вид

$$f(\mathbf{x}) \to \min, \quad \sum_{i=1}^{n} x_i = b, \quad 0 \le x_i \le b_i, \quad i = \overline{1,n}, \quad \mathbf{x} \in E_z^n .$$

В классической непрерывной постановке ($\mathbf{x} \in E^n$) — это просто решаемая задача линейного программирования, мы обращаем ее в задачу целочисленного линейного программирования путем помещения товаров в контейнеры, которые являются минимальными единицами перевозки. Заметим, что область перебора — это n-мерный координатный параллелепипед с началом в нуле, и ребрами длины b_i.

Оптимизация многопроцессорного расписания. Приведем еще одну постановку задачи, более близкую и понятную программистам. Суть задачи сводится к составлению оптимального расписания работы многопроцессорной системы.

Пусть система состоит из m неоднородных процессоров, по которым необходимо распределить n заданий, причем $n > m$. Если такое распределение есть, то каждый процессор последовательно выполняет указанные ему задания. Мы заранее знаем время выполнения каждого задания на каждом процессоре, или имеем его оценку. Пусть t_{ij} — есть время выполнения задания i на процессоре j. Наша цель — минимизировать суммарное время решения всего набора заданий.

Рассмотрим вектор x в пространстве E_z^n, и будем интерпретировать значение компоненты x_i этого вектора как номер процессора, на котором должно выполняться задание с номером i. Этот вектор и будет являться «расписанием» работы процессоров. Поскольку имеется m процессоров, и каждое задание должно быть назначено на какой-то процессор, то возникает очевидная система ограничений

$$1 \le x_i \le m, \quad i = \overline{1,n} .$$

Таким образом, областью G для этой задачи является координатный куб с началом в точке $\mathbf{1} = (1,1,\cdots,1)$ в n-мерном целочисленном пространстве с ребром m — $Cub_z^n(\mathbf{1}, m)$.

Поскольку весь набор заданий считается выполненным, когда последний (по времени) процессор завершит выполнение последнего задания, то целевой функционал может быть записан в виде

$$f(\mathbf{x}) = \max_{j=1,m}\{T_j\} \to \min,$$

где T_j — время работы процессора j, определяемое как

$$T_j = \sum_{i=1}^{n} [x_i = j] \cdot t_{ij},$$

где под знаком суммы использована нотация Айверсона — $[k = l] = 1$, если $k = l$, иначе — 0.

7.2. Методы разработки точных алгоритмов решения целочисленных задач

Первые работы по целочисленному программированию были опубликованы в конце1950-х – начале 1960-х годов. За прошедшее время было разработано достаточно много различных алгоритмов и методов, обеспечивающих как точное, так и приближенное решение целочисленных задач.

Известно, что точное решение не может быть получено за полиномиальное время для тех задач целочисленного программирования, которые относятся к классу NPC. Но, оказывается, что построение приближенных методов (с гарантированной погрешностью) для многих классов дискретных задач является столь же трудоемкой проблемой, что и поиск точного решения [7.2]. Тем не менее, для ряда задач такие эффективные приближенные алгоритмы существуют. В частности, к таким относится уже известные читателю задачи о составлении расписания многопроцессорной системы, и об оптимальном резервировании элементов.

Отметим, что в последнее десятилетие широко разрабатываются новые интересные алгоритмические методы, основанные, в частности, на биологических аналогиях — имеются в виду генетические и муравьиные алгоритмы, успешно применяемые для решения задач целочисленного программирования. Кратко эти подходы будут изложены в параграфе 7.3.

При достаточном разнообразии, точные алгоритмы решения могут быть описаны несколькими общими методами, универсального и ограниченного применения. Большинство универсальных алгоритмов основано на идеях метода отсечения и метода ветвей и границ. Алгоритмы ограниченного применения являются, в основном, модификациями метода динамического программирования.

Ниже мы излагаем основные принципы этих методов, которые далее проиллюстрируем на конкретных задачах.

Метод полного перебора. Поскольку число возможных решений задачи целочисленного программирования ограничено сверху — мы можем гарантировать, что существует координатный куб, в который вписан многогранник, формируемый ограничениями задачи, то очевидным методом является полный перебор всех возможных вариантов — точек целочисленного пространства. В англоязычной литературе этот метод называется British museum technique — техника британского музея.

Мы пытаемся решить задачи перебора методом перебора, но нас останавливает проблема размерности — как только размерность целочисленного пространства становится большой — перебор требует, хотя и конечного, но астрономического времени. Попробуйте, например, оценить, при каких значениях n ваш компьютер решит задачу перебора в кубе 1_z^n не более чем за один час.

Метод отсечений. Идея метода принадлежит Г. Данцигу и Р. Гомори [7.2]. Изначально метод был предложен для задач линейного целочисленного программирования. Суть идеи заключается в том, чтобы вначале пренебречь требованием целочисленности и попытаться решить соответствующую непрерывную задачу линейного программирования. Если оптимальное решение задачи непрерывного линейного программирования удовлетворяет требованию целочисленности, то тогда оно является одновременно и решением целочисленной задачи.

Если же на первом шаге получено нецелочисленное решение, то к исходным ограничениям добавляется новое линейное неравенство, которое формируется таким образом, что бы полученное нецелочисленное решение не удовлетворяло новому неравенству, а любое целочисленное решение заведомо удовлетворяло ему. Описанная процедура повторяется с новой задачей линейного программирования, система ограничений которой содержит на одно неравенство больше. Полученное решение вновь проверяется на целочисленность, и, при необходимости дополняется новым неравенством. Через некоторое число итераций, будет либо найдено целочисленное решение, либо очередная система неравенств окажется несовместной. Очевидно, что число итераций существенно зависит от способа

формирования дополнительных ограничений. В общем виде метод может быть представлен следующим образом

```
Procedure Метод отсечений
Begin
   1. Решить непрерывную задачу линейного программирования
      с исходной системой ограничений
   While (пока не найдено целочисленное решение или
          система ограничений несовместна)
      begin
         2.1. Сформировать новое ограничение, отсекающее
              полученное нецелочисленное решение
         2.2. Решить непрерывную задачу линейного
              программирования с новой системой ограничений
   end while
End.
```

В терминах геометрии многомерного пространства добавление каждого нового линейного неравенства означает построение гиперплоскости, отсекающей от многогранного множества, сформированного исходными ограничениями задачи линейного программирования оптимальную, но нецелочисленную, вершину и сохраняющей все целочисленные точки области определения задачи. Эта аналогия и дала название методам, основанных на идеях Г. Данцига — «методы отсечений».

Метод ветвей и границ. Основная идея метода была предложена в работах А. Лэнда и А. Дойга по линейному целочисленному программированию в начале 1960-х годов. Метод стал широко известен благодаря детально разработанному и эффективному алгоритму Дж. Литла, К. Мерти, Д. Суини и К. Кэрол [7.3] для точного решения задачи коммивояжера, которая является NP-полной (класс NPC).

Основная идея метода состоит в попытке существенного сокращения перебора, за счет анализа подмножеств общего множества точек многогранника, подлежащих перебору. Наиболее важным этапом метода является выявление подмножеств точек, гарантированно не содержащих оптимального решения.

Метод основан на последовательном разбиении исходного множества точек на подмножества — этот процесс называется ветвлением, и вычислении оценок целевого функционала задачи (нижних при решении задачи на минимум) на выделенных подмножествах — этот процесс называется вычислением границ. Таким образом, строится поисковое дерево решений, и основная идея метода состоит в отсечении тех ветвей этого дерева, которые заведомо не содержат оптимального

решения. Проблема применения метода к конкретной задаче состоит в выборе стратегии ветвления и принципов построения оценок.

В качестве простейшего способа вычисления оценок предлагается следующий подход — решить непрерывную задачу при ограничениях, соответствующих рассматриваемому подмножеству точек. Обычно стратегии ветвления и оценивания существенно опираются на специфику задачи. При удачно выбранных стратегиях сокращение перебора оказывается настолько существенно, что позволяет решить задачи практически значимых размерностей за приемлемое время, именно такими свойствами обладает указанный выше алгоритм решения задачи коммивояжера. Подробно метод ветвей и границ в приложении к задаче коммивояжера будет изложен в главе 12.

7.4. Эволюционные вычисления и генетические алгоритмы

Введение в эволюционные вычисления. Практическая необходимость решения ряда NP-полных задач в оптимизационной постановке при проектировании и исследовании сложных систем привела разработчиков алгоритмического обеспечения к использованию биологических механизмов поиска наилучших решений. В настоящее время эффективные алгоритмы разрабатываются в рамках научного направления, которое можно назвать «эволюционные вычисления» [7.4], объединяющего такие разделы, как генетические алгоритмы, эволюционное программирование, нейросетевые вычисления [7.5], клеточные автоматы и ДНК-вычисления [7.4], муравьиные алгоритмы. Исследователи обращаются к природным механизмам, которые миллионы лет обеспечивают адаптацию биоценозов к окружающей среде. На основе этих механизмов могут быть разработаны эффективные эвристические алгоритмы решения задач оптимизации. Практика их применения показывает, что в целом ряде случаев, получаемы результаты, в смысле близости к глобальному оптимуму, оказываются лучше, чем при других эвристических подходах. Отметим также, что трудоемкость этих алгоритмов является вполне приемлемой, и позволяет решать задачи большой размерности.

Генетические алгоритмы. 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, такие, как пища или вода. Кроме того, особи одного вида в популяции конкурируют между собой, например, за привлечение брачного партнера. 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 высоко приспособленных особей будут распространяться в последующих поколениях. Комбинация хороших характеристик от различных родителей иногда может приводить к появлению потомка, приспособленность которого даже больше, чем приспособленность его родителей. Таким образом, вид в целом развивается, лучше и лучше приспосабливаясь 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 эти идеи, как собственно и свое название — генетические алгоритмы, получили в работах Гольдберга и Де Йонга [7.6]. 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 подходящую структуру данных для представления решений. В постановке задачи поиска оптимума, экземпляр этой структуры должен содержать информацию о некоторой точке в пространстве решений.

Структура данных генетического алгоритма состоит из набора хромосом. Хромосома, как правило, представляет собой битовую строку, так что термин строка часто заменяет понятие «хромосома». Вообще говоря, хромосомы генетических алгоритмов не ограничены только бинарным представлением. Известны другие реализации, построенные на векторах вещественных чисел [7.6]. 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 только структурам, которые являются битовыми строками. 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 также употребляется термин кроссовер, скрещивание или рекомбинация. В простейшем случае кроссинговер в генетическом алгоритме реализуется так же, как и в биологии. При скрещивании 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 (11, 12, 0, 0) и (0, 0, 13, 14).

Основные структуры и фазы генетического алгоритма. Приведем простой иллюстративный пример [7.7] — задачу максимизации некоторой функции двух переменных $f(x_1, x_2)$, при ограничениях: $0 < x_1 < 1, 0 < x_2 < 1$. Обычно, методика кодирования реальных 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-и 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 $0/1023$ или 0, тогда как 1111111111 соответст-

вует 1023/1023 или 1. Оптимизируемая структура данных есть 20-ти битовая строка, представляющая собой конкатенацию (объединение) кодировок x_1 и x_2. Пусть переменная x_1 размещается в крайних левых 10-и битах строки, тогда как x_2 размещается в правой части генотипа особи. Таким образом, генотип представляет собой точку в 20-ти мерном целочисленном пространстве (вершину единичного гиперкуба), которая исследуется генетическим алгоритмом. Для этой задачи фенотип будет представлять собой точку в двумерном пространстве параметров (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 простого генетического алгоритма [7.7]. В начале 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 вероятности $P(i)$, равной отношению ее приспособленности к суммарной приспособленности популяции (по целевой функции):

$$P(i) = \frac{f(i)}{\sum_{i=1}^{n} f(i)}.$$

Затем происходит отбор (с замещением) всех n особей для дальнейшей генетической обработки, согласно убыванию величины $P(i)$. Простейший пропорциональный отбор реализуется с помощью рулетки (Гольдберг, 1989 г.). «Колесо» рулетки содержит по одному сектору для каждого члена популяции, а размер i-ого сектора пропорционален соответствующей величине $P(i)$. При таком отборе члены популяции, обладающие более высокой приспособленностью, будут выбираться чаще по вероятности.

После отбора 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 вероятность P_c может быть выполнена рекомбинация. Если рекомбинация происходит, то полученные потомки заменяют собой родителей. Одноточечная рекомбинация работает следующим образом. Случайным образом выбирается одна из точек 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, хорошие 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, может быть, благодаря мутации, и к глобальному оптимуму. Таким образом, основные шаги генетического алгоритма могут быть сформулированы следующим образом:

```
1. Создать начальную популяцию
2. Цикл по поколениям, пока не выполнено условие останова
   3. Оценить приспособленность каждой особи
   4. Выполнить отбор по приспособленности
   5. Случайным образом разбить популяцию на две группы пар
   6. Выполнить фазу вероятностной рекомбинации для пар популяции
      и заменить родителей
   7. Выполнить фазу вероятностной мутации
   8. Оценить приспособленность новой популяции и
      вычислить условие останова
   9. Объявить потомков новым поколением
10. Конец цикла по поколениям
```

Модификации генетического алгоритма. Очевидно, что тонкая настройка базового генетического алгоритма может быть выполнена путем изменения зна-

чений вероятностей рекомбинации и мутации, существует много исследований и предложений в данной области, более подробно этот вопрос освящен в [7.5] и [7.8]. В настоящее время предлагаются разнообразные модификации генетических алгоритмов в части методов отбора по приспособленности, рекомбинации и мутации [7.5, 7.6, 7.9]. Приведем несколько примеров [7.7].

Метод турнирного отбора (Бриндел, 1981 г.; Гольдберг и Деб, 1991 г.) 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, если она не прошла через процесс отбора, рекомбинации и мутации. Этот метод может быть внедрен практически в любой стандартный метод отбора. 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) [7.4]. По мнению автора интерес представляет 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 [7.4].

Применение генетических алгоритмов. Основная проблема, связанная с применением генетических алгоритмов — это их эвристический характер. Говоря более строго, какова вероятность достижения популяцией глобального оптимума в заданной области при данных настройках алгоритма? В настоящее время не существует строгого ответа и теоретически обоснованных оценок. Имеются предполо-

жения, что генетический алгоритм может стать эффективной процедурой поиска оптимального решения, если:

— пространство поиска достаточно велико, и предполагается, что целевая функция не является гладкой и унимодальной в области поиска, т.е. не содержит один гладкий экстремум;

— задача не требует нахождения глобального оптимума, необходимо достаточно быстро найти приемлемое «хорошее» решение, что довольно часто встречается в реальных задачах.

Если целевая функция обладает свойствами гладкости и унимодальности, то любой градиентный метод, такой 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 на нее каким-то образом специально 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 функции приспособленности градиентные методы с единственным решением могут останавливаться в локальном решении. Наличие у генетических алгоритмов целой «популяции» решений, совместно с вероятностным механизмом мутации, позволяют предполагать меньшую вероятность нахождения локального оптимума и большую эффективность работы на многоэкстремальном ландшафте.

Сегодня генетические алгоритмы успешно при меняются как для решения классических NP-полных задач, задач оптимизации в пространствах с большим количеством измерений, ряда экономических задач оптимального характера, например, задач распределения инвестиций. Много полезной и доступно изложенной информации по генетическим алгоритмам читатель может найти в книге [7.6].

Муравьиные алгоритмы. Муравьиные алгоритмы представляют собой новый перспективный метод решения задач оптимизации, в основе которого лежит моделирование поведения колонии муравьев. Колония представляет собой систему с очень простыми правилами автономного поведения особей. Однако, не смот-

ря на примитивность поведения каждого отдельного муравья, поведение всей колонии оказывается достаточно разумным. Эти принципы проверены временем — удачная адаптация к окружающему миру на протяжении миллионов лет означает, что природа выработала очень удачный механизм поведения. Исследования в этой области начались в середине 90-х годов XX века, автором идеи является Марко Дориго из Университета Брюсселя, Бельгия [7.10, 7.11, 7.12].

Биологические принципы поведения муравьиной колонии. Муравьи относятся к социальным насекомым, образующим коллективы. В биологии коллектив муравьев называется колонией. Число муравьев в колонии может достигать нескольких миллионов, на сегодня известны суперколонии муравьев (Formica lugubrus), протянувшиеся на сотни километров. Одним из подтверждений оптимальности поведения колоний является тот факт, что сеть гнезд суперколоний близка к минимальному остовному дереву графа их муравейников [7.13].

Основу поведения муравьиной колонии составляет самоорганизация, обеспечивающая достижения общих целей колонии на основе низкоуровневого взаимодействия. Колония не имеет централизованного управления, и ее особенностями является обмен локальной информацией только между отдельными особями (прямой обмен — пища, визуальные и химические контакты) и наличие непрямого обмена, который и используется в муравьиных алгоритмах.

Непрямой обмен — *стигмержи* (stigmergy), представляет собой разнесенное во времени взаимодействие, при котором одна особь изменяет некоторую область окружающей среды, а другие используют эту информацию позже, в момент, когда они в нее попадают. Биологи установили, что такое отложенное взаимодействие происходит через специальное химическое вещество — *феромон* (pheromone), секрет специальных желез, откладываемый при перемещении муравья. Концентрация феромона на тропе определяет предпочтительность движения по ней. Адаптивность поведения реализуется испарением феромона, который в природе воспринимается муравьями в течение нескольких суток. Мы можем провести некоторую аналогию между распределением феромона в окружающем колонию пространстве, и «глобальной» памятью муравейника, носящей динамический характер [7.13].

Идея муравьиного алгоритма. Идея муравьиного алгоритма — моделирование поведения муравьев, связанное с их способностью быстро находить кратчайший путь от муравейника к источнику пищи и адаптироваться к изменяющимся условиям, находя новый кратчайший путь. При своем движении муравей метит свой путь феромом, и эта информация используется другими муравьями для выбора пути. Это элементарное правило поведения и определяет способность муравьев находить новый путь, если старый оказывается недоступным. Дойдя до преграды, муравьи с равной вероятностью будут обходить ее справа и слева. То же самое будет происходить и на обратной стороне преграды. Однако, те муравьи, которые случайно выберут кратчайший путь, будут быстрее его проходить, и за несколько передвижений он будет более обогащен феромоном. Поскольку движение муравьев определяется концентрацией феромона, то следующие будут предпочитать именно этот путь, продолжая обогащать его феромоном, до тех пор, пока этот путь по какой-либо причине не станет доступен. Очевидная положительная обратная связь быстро приведет к тому, что кратчайший путь станет единственным маршрутом движения большинства муравьев. Моделирование испарения феромона — отрицательной обратной связи, гарантирует нам, что найденное локально оптимальное решение не будет единственным — муравьи будут искать и другие пути. Если мы моделируем процесс такого поведения на некотором графе, ребра которого представляют собой возможные пути перемещения муравьев, в течение определенного времени, то наиболее обогащенный феромоном путь по ребрам этого графа и будет являться решением задачи, полученным с помощью муравьиного алгоритма. Рассмотрим конкретный пример.

Формализация задачи коммивояжера в терминах подхода муравьиных алгоритмов. Задача формулируется как задача поиска минимального по стоимости замкнутого маршрута по всем вершинам без повторений на полном взвешенном графе с n вершинами. Содержательно вершины графа являются городами, которые должен посетить коммивояжер, а веса ребер отражают расстояния (длины) или стоимости проезда. Эта задача является NP-трудной, и точный переборный алгоритм ее решения имеет факториальную сложность. Приводимое здесь описание

муравьиного алгоритма для задачи коммивояжера является кратким изложением статьи С. Д. Штовбы [7.13].

Моделирование поведения муравьев связано с распределением феромона на тропе — ребре графа в задаче коммивояжера. При этом вероятность включения ребра в маршрут отдельного муравья пропорциональна количеству феромона на этом ребре, а количество откладываемого феромона пропорционально длине маршрута. Чем короче маршрут, тем больше феромона будет отложено на его ребрах, следовательно, большее количество муравьев будет включать его в синтез собственных маршрутов. Моделирование такого подхода, использующего только положительную обратную связь, приводит к преждевременной сходимости — большинство муравьев двигается по локально оптимальному маршруту. Избежать этого можно, моделируя отрицательную обратную связь в виде испарения феромона. При этом если феромон испаряется быстро, то это приводит к потере памяти колонии и забыванию хороших решений, с дугой стороны, большое время испарения может привести к получению устойчивого локально оптимального решения. Теперь, с учетом особенностей задачи коммивояжера, мы можем описать локальные правила поведения муравьев при выборе пути.

— муравьи имеют собственную «память». Поскольку каждый город может быть посещен только один раз, у каждого муравья сеть список уже посещенных городов — список запретов. Обозначим через $J_{i,k}$ список городов, которые необходимо посетить муравью k, находящемуся в городе i;

— муравьи обладают «зрением» — видимость есть эвристическое желание посетить город j, если муравей находится в городе i. Будем считать, что видимость обратно пропорциональна расстоянию между городами i и j — D_{ij}

$$\eta_{ij} = 1/D_{ij}.$$

— муравьи обладают «обонянием» — они могут улавливать след феромона, подтверждающий желание посетить город j из города i, на основании опыта других муравьев. Количество феромона на ребре (i, j) в момент времени t обозначим через $\tau_{ij}(t)$.

На этом основании мы можем сформулировать вероятностно-пропорциональное правило [7.13], определяющее вероятность перехода k-ого муравья из города i в город j:

$$\begin{cases} P_{i\,j,k}(t) = \dfrac{[\tau_{ij}(t)]^{\alpha} \cdot [\eta_{ij}]^{\beta}}{\sum\limits_{l \in J_{i,k}} [\tau_{il}(t)]^{\alpha} \cdot [\eta_{il}]^{\beta}}, \quad j \in J_{i,k}; \\ P_{i\,j,k}(t) = 0, \quad j \notin J_{i,k}, \end{cases}, \tag{7.3.1}$$

где α, β — параметры, задающие веса следа феромона, при $\alpha = 0$ алгоритм вырождается до жадного алгоритма (будет выбран ближайший город). Заметим, что выбор города является вероятностным, правило (7.3.1) лишь определяет ширину зоны города j; в общую зону всех городов $J_{i,k}$ бросается случайное число, которое и определяет выбор муравья. Правило (7.3.1) не изменяется в ходе алгоритма, но у двух разных муравьев значение вероятности перехода будут отличаться, т. к. они имеют разный список разрешенных городов.

Пройдя ребро (i, j), муравей откладывает на нем некоторое количество феромона, которое должно быть связано с оптимальностью сделанного выбора. Пусть $T_k(t)$ есть маршрут, пройденный муравьем k к моменту времени t, а $L_k(t)$ — длина этого маршрута. Пусть также Q — параметр, имеющий значение порядка длины оптимального пути. Тогда откладываемое количество феромона может быть задано в виде

$$\Delta\tau_{ij,k}(t) = \begin{cases} \dfrac{Q}{L_k(t)}, & (i,j) \in T_k(t); \\ 0, & (i,j) \notin T_k(t). \end{cases}$$

Правила внешней среды определяют, в первую очередь, испарение феромона. Пусть $p \in [0,1]$ есть коэффициент испарения, тогда правило испарения имеет вид

$$\tau_{ij}(t+1) = (1-p) \cdot \tau_{ij}(t) + \Delta\tau_{ij}(t); \quad \Delta\tau_{ij}(t) = \sum_{k=1}^{m} \Delta\tau_{ij,k}(t), \tag{7.3.2}$$

где m — количество муравьев в колонии.

В начале алгоритма количество феромона на ребрах принимается равным небольшому положительному числу. Общее количество муравьев остается постоянным и равным количеству городов, каждый муравей начинает маршрут из сво-

его города. Дополнительная модификация алгоритма может состоять во введении так называемых «элитных» муравьев, которые усиливают ребра наилучшего маршрута, найденного с начала работы алгоритма. Обозначим через T^* наилучший текущий маршрут, через L^* — его длину. Тогда если в колонии есть e элитных муравьев, ребра маршрута получат дополнительное количество феромона

$$\Delta\tau_e = e \cdot Q / L^* . \qquad (7.3.3)$$

Муравьиный алгоритм для задачи коммивояжера

```
1. Ввод матрицы расстояний D.
2. Инициализация параметров алгоритма – α, β, e, Q.
3. Инициализация ребер – присвоение видимости η_ij
   и начальной концентрации феромона.
4. Размещение муравьев в случайно выбранные города без совпадений.
5. Выбор начального кратчайшего маршрута и определение L*
6. Цикл по времени жизни колонии t=1,t_max.
   7. Цикл по всем муравьям k=1,m
      8. Построить маршрут T_k(t)по правилу (7.3.1)
         и рассчитать длину L_k(t).
   9. конец цикла по муравьям.
   10. Проверка всех L_k(t) на лучшее решение по сравнению с L*.
   11. Если да, то обновить L* и T*.
   12. Цикл по всем ребрам графа.
      13. Обновить следы феромона на ребре
          по правилам (7.3.2) и (7.3.3).
   14. конец цикла по ребрам.
15. конец цикла по времени.
16. Вывести кратчайший маршрут T* и его длину L*.
```

Сложность данного алгоритма определяется непосредственно из приведенного выше текста — $\Theta\left(t_{\max} \cdot \max\left(m, n^2\right)\right)$, таким образом, сложность зависит от времени жизни колонии, количества городов и количества муравьев в колонии.

Области применения и возможные модификации. Поскольку в основе муравьиного алгоритма лежит моделирование передвижения муравьев по некоторым путям, то такой подход может стать эффективным способом поиска рациональных решений для задач оптимизации, допускающих графовую интерпретацию. Ряд экспериментов показывает, что эффективность муравьиных алгоритмов растет с ростом размерности решаемых задач оптимизации. Хорошие результаты получаются для нестационарных систем с изменяемыми во времени параметрами, например, для расчетов телекоммуникационных и компьютерных сетей [7.13]. В [7.14] описано применение муравьиного алгоритма для разработки оптимальной

структуры съемочных сетей *GPS*, в рамках создания высокоточных геодезических и съемочных технологий. В настоящее время на основе применения муравьиных алгоритмов получены хорошие результаты для таких сложных оптимизационных задач как задача коммивояжера, транспортная задача, задача календарного планирования, задача раскраски графа, квадратичной задачи о назначениях, задачи оптимизации сетевых графиков и ряда других [7.13].

Качество получаемых решений во многом зависит от настроечных параметров в вероятностно-пропорциональном правиле выбора пути на основе текущего количества феромона и параметров правил откладывания и испарения феромона. Возможно, что динамическая адаптационная настройка этих параметров может способствовать получению лучших решений. Немаловажную роль играет и начальное распределение феромона, а также выбор условно оптимального решения на шаге инициализации. В [7.13] отмечается, что перспективными путями улучшения муравьиных алгоритмов является адаптация параметров с использованием базы нечетких правил и их гибридизация, например, с генетическими алгоритмами. Как вариант, такая гибридизация может состоять в обмене, через определенные промежутки времени, текущими наилучшими решениями.

Много полезной информации по муравьиным алгоритмам читатель может найти на специальном англоязычном сайте по этому направлению [7.14].

Список литературы к главе 7

[7.1] Ильин В. А., Садовничий В. А., Сендов Бл. Х. Математический анализ. — М.: Наука. Главная редакция физико-математической литературы, 1979. — 720 с.

[7.2] Шевченко В. Н. Качественные вопросы целочисленного программирования, — М: Физматлит, 1995. — 192 с.

[7.3] Little R. D. C., Murty K. G., Sweeney D. W., Karel C. An Algorithm for the Traveling Salesman Problem. Oper. Res., 11: 979–989 (1963) (IM).

[7.4] 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.

[7.5] Рутковский Л., Пилиньский М., Рутковская Д. Нейронные сети, генетические алгоритмы и нечеткие системы. — М.: Горячая линия-Телеком, 2004 г. — 452 с.

[7.6] Гладков Л.А, Курейчик В.В., Курейчик В.М. Генетические алгоритмы / Под ред. В.М. Курейчика. — 2-ое изд., испр. и доп. — М.: ФИЗМАТЛИТ, 2006. — 320 с.

[7.7] http://www.krf.bsu.by/ELib/Genetic/GenAlg_2/index.htm.

[7.8] http://www.neuroproject.ru.

[7.9] http://www.aic.nrl.navy.mil/galist/.

[7.10] Bonavear E., Dorigo M. Swarm Intelligence: from Natural to Artificial Systems. — Oxford University Press, 1999, — 307 p.

[7.11] Corne D., Dorigo M., Glover F. New Ideas in Optimization. — McGrav-Hill, 1999.

[7.12] http://iridia.ulb.ac.be/dorigo/ACO/ACO.html.

[7.13] Штовба С.Д. Муравьиные алгоритмы // Exponenta Pro Математика в приложениях. 2003. №4. С.70–75.

[7.14] http://www.swarm.org.

РАЗДЕЛ IV

МЕТОДЫ АНАЛИЗА ТРУДОЕМКОСТИ КОМПЬЮТЕРНЫХ АЛГОРИТМОВ

Основная цель данного раздела — познакомить читателей с некоторыми методами анализа компьютерных алгоритмов, позволяющих получить точные или асимптотические оценки трудоемкости. Несмотря на более чем сорокалетнюю историю этой проблематики, универсальные методы анализа либо отсутствуют, либо формулируются в столь общем виде, что их конкретное применение требует серьёзной адаптации. Тем не менее, как для итерационной, так и для рекурсивной реализаций алгоритмов существует ряд методов, позволяющих либо получить точную функцию трудоемкости, либо вычислительную сложность алгоритма. Наиболее полную информацию о ресурсных требованиях алгоритма в зависимости от длины входа можно получить на основе ресурсных функций — трудоемкости алгоритма и функции объема памяти, на основе которых возможно решение задачи выбора рациональных алгоритмов в области реальных длин входов путем их сравнительного анализа на конечном интервале.

Очевидно, что принадлежность алгоритма к одному из классов по влиянию на трудоемкость характеристических особенностей множества исходных данных (см. глава 4) существенно влияет на трудности, связанные с его анализом. Отметим в этой связи, что, например, для алгоритмов с сильной параметрической зависимостью трудоемкости, оценка в среднем должна опираться на априорную или фактическую информацию о частотной встречаемости входов. Также в этом случае размах варьирования и рассмотрение функции трудоемкости как дискретной ограниченной случайной величины могут давать более значимые результаты в аспекте сравнительного анализа, чем непосредственная оценка в среднем.

Методы в данном разделе излагаются в общем виде, с небольшими иллюстративными примерами. Эти методы базируются на известных подходах к анализу компьютерных алгоритмов, прежде всего методах асимптотического анализа. Однако для получения функции трудоемкости, в отличие от ее асимптотической

оценки — вычислительной сложности, необходимо дополнительно учитывать принятую модель вычислений, постулирующую базовые операции процедурного языка высокого уровня. Дополнительно для рекурсивных алгоритмов приводится метод оценки объема памяти в области программного стека, поскольку в ряде случаев такие затраты могут быть существенны по отношению к общему объему дополнительной памяти, требуемому алгоритмом.

Применение этих методов к анализу алгоритмов читатель может найти в разделе V, где будут рассмотрены и проанализированы ресурсно-эффективные алгоритмы решения целого ряда задач.

Г Л А В А 8.

А Н А Л И З И Т Е Р А Ц И О Н Н Ы Х А Л Г О Р И Т М О В

Введение

Итерационная, или как ее ещё часто называют — процедурная реализация алгоритмов является до сих пор одним из основных подходов к разработке программных средств и систем. С середины 1960-х годов, после введения Эдмондсом в теорию алгоритмов класса полиномиально разрешимых задач, которое стимулировало значительный рост интереса к проблематике вычислительной сложности, накоплен значительный опыт анализа и исследования итерационных алгоритмов. В этой главе приводятся только основные и самые распространенные методы, широко используемые на практике.

8.1. Методика анализа основных алгоритмических конструкций

Предварительные замечания. Практически значимые реализации компьютерных 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 анализа алгоритмов по трудоемкости мы можем считать, что, обслуживающие фрагменты программной реализации (ввод, проверка и вывод), являются общими или эквивалентными для разных алгоритмов решения данной задачи. Такой подход приводит к необходимости анализа только непосредственного алгоритма 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 буфера реального процессора. Таким образом, множество операций, учитываемых в функции трудоемкости, не включает операции ввода/вывода данных на внешние носители.

По отношению к набору базовых операций принятой модели вычислений (см. главу 2), коррелированному с операторами процедурного языка высокого уровня необходимо также сделать несколько замечаний:

— опираясь на идеи структурного программирования, из набора базовых операций исключается команда перехода, поскольку ее можно считать связанной 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 базовыми операциями, операндами которых они являются;

— конструкции циклов не рассматриваются, т. к. могут быть сведены к указанному набору базовых операций.

Методика анализа основных алгоритмических конструкций. Несмотря на наличие разнообразных подходов к созданию компьютерных программ и различных технологий программирования, классический процедурный (итерационный) подход остается определяющим при программной реализации алгоритмов. Основными алгоритмическими конструкциями в этом подходе являются конструкции следования, ветвления и цикла. Для получения функции трудоемкости некоторого алгоритма необходима методика анализа трудоемкости основных алгоритмических конструкций. С учетом введенных базовых операций такая методика сводится к следующим положениям:

Конструкция «Следование». Трудоемкость конструкции есть сумма трудоёмкостей блоков, следующих друг за другом

$$f_{\text{следование}} = f_1 + \ldots + f_k,$$

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». Конструкция «Ветвление» может быть без потери общности представлена в следующем виде:

```
If  ( l )
   then
      блок с трудоемкостью f_then, выполняемый с вероятностью p;
   else
      блок с трудоемкостью f_else, выполняемый с вероятностью (1- p)
end If
```

В этой конструкции (l) обозначает логическое выражение, состоящее из логических переменных и/или арифметических, символьных или других по типу сравнений с ограничением $\Theta(1)$ на трудоемкость его вычисления. 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 значений p вне зависимости от конкретного алгоритма или особенностей входа.

Общая трудоемкость конструкции «Ветвление» для построения функции трудоемкости в среднем случае требует специального анализа для получения вероятности p выполнения переходов на блоки «**then**» и «**else**». При известном значении этой вероятности трудоемкость конструкции определяется как

$$f_{\text{ветвление}} = f_l + f_{then} \cdot p + f_{else} \cdot (1 - p),$$

где f_l — трудоемкость вычисления условия (l). Отметим, что 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». Конструкция «Цикл по счетчику» может быть следующим образом сведена к базовым операциям принятой модели вычислений:

```
For i ← 1 to n                    ⇔    i ←1
     тело цикла                        Repeat
end For                                     тело цикла
                                  i ← i+1
                                  until i ≤ n
```

После сведения конструкции к введенным базовым операциям ее трудоемкость определяется следующим образом

$$f_{\text{цикл}} = 1 + 3 \cdot n + n \cdot f_{\text{тело цикла}}.$$

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, исчисление которых заинтересованный читатель может найти, например, в [8.1].

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. Общие рекомендации по определению суммарного количества проходов цикла крайне затруднительны из-за сложных зависимостей от исходных данных. Для худшего случая могут быть использованы верхние граничные оценки. 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.

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, зависящих от данных («ветвление» и «цикл по условию») будут различны.

Несмотря на достаточно простые подходы к анализу основных алгоритмических конструкций, получение функций трудоемкости алгоритмов остается достаточно сложной задачей, требующей применения специальных подходов и методов, а иногда и введения специальных функций. Тем не менее, для количественно-зависимых алгоритмов из класса N можно получить точное значение функции трудоемкости.

Основные трудности анализа алгоритма в среднем случае связаны как с выяснением значений трудоемкости для конкретных входов, так и с определением вероятности их появления. В связи с этим задача вычисления трудоемкости в среднем для алгоритмов класса NPR, особенно при сильной параметрической зависимости является достаточно трудоемкой. Для получения оценок вычислительной сложности алгоритмов в среднем случае в литературе предлагаются различные методы, как общего, так и частного характера, например метод вероятностного анализа [8.2], амортизационный анализ [8.3], метод классов входных данных [8.4]. Эти методы могут быть использованы и для получения функции трудоемкости алгоритмов в среднем случае.

8.2. Анализ трудоемкости количественно-зависимых алгоритмов

В смысле получения детальной функции трудоемкости алгоритмы класса N являются наиболее удобными объектами анализа — трудоемкость алгоритма зависит только от меры длины входа. Если рассматривать общую ситуацию анализа итерационного алгоритма, то может быть предложен некоторый обобщающий метод, который опирается на введенную в главе 2 модель вычислений и её базовые операции, коррелированные с процедурным языком высокого уровня, методику анализа основных алгоритмических конструкций (см. 8.1) и известные методы анализа алгоритмов. Описания различных методов анализа алгоритмов читатель может найти также в целом ряде литературных источников [8.2, 8.3, 8.5, 8.6], посвященных анализу сложности алгоритмов.

Метод включает в себя следующие этапы:

1. Рекурсивную декомпозицию 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 их 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.

Основные трудности, связанные с получением функции трудоемкости, относятся к алгоритмам класса NPR, т. к. для случаев, когда цикл по условию является сильно зависимым от данных (для алгоритмов с большой информационной чувствительностью), получение функции трудоемкости в среднем требует детального анализа, введения дополнительной параметризации задачи или предположений о границах диапазонов изменения исходных данных [8.7].

В качестве примеров, иллюстрирующих построение ресурсных функций, рассмотрим ряд алгоритмов, относящихся к классу N. Основной целью является демонстрация методики анализа алгоритмических конструкций с учетом введенных базовых операций модели вычислений на основе известных методов анализа алгоритмов.

Пример 8.1. Алгоритм решения задачи суммирования элементов квадратной матрицы. Запись алгоритма имеет вид:

```
SumM (A, n; Sum)
   Sum ← 0                                    1
   For i ← 1 to n                           1+3n
      For j ← 1 to n                        1+3n
         Sum ← Sum + A[i,j]                   4
      end For j
   end For i
Return (Sum)
End.
```

Этот простейший алгоритм, очевидно, выполняет одинаковое количество базовых операций при фиксированном значении n. Таким образом, он является количественно-зависимым и относится к классу N. Внутренний цикл не зависит от внешнего, что позволяет непосредственно применить формулу для трудоемкости конструкции «Цикл по счетчику». Обратим внимание на то, что в строке вычисления нового значения **Sum** двойная индексация в обращении к элементу **A[i,j]** считается за 2 базовые операции. Для данного алгоритма вычисления суммы, с учетом трудоемкостей, приведенных в строках, получаем

$$f_A(n) = 1 + 1 + n(3 + 1 + n(3 + 4)) = 7n^2 + 4n + 2 = \Theta\left(n^2\right).$$

Заметим, что под n здесь понимается линейная размерность матрицы, т. е. мера длины входа, в то время как на вход алгоритма в действительности подается n^2 значений. Такой подход не согласуется с классической теорией алгоритмов, в которой сложность алгоритма определяется как функция непосредственной длины входа, но достаточно часто используется в литературе по практическому применению методов анализа алгоритмов [8.3].

Входом алгоритма является массив, содержащий n^2 элементов, и алгоритм требует три ячейки для счетчиков циклов и хранения суммы, таким образом, функция объема памяти этого алгоритма:

$$V_A(n) = n^2 + 3\,.$$

Ресурсная характеристика алгоритма определяется полученными выше ресурсными функциями, а его ресурсная сложность имеет вид

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

Пример 8.2. Алгоритм для фрагмента статистической обработки данных, вычисляющего вектор ковариаций.

Алгоритм предназначен для вычисления вектора значений ковариаций массива X, содержащего n элементов, и строк матрицы Y размерностью $n \times n$ элементов. Фрагмент опирается на формулы для вычисления значения ковариации, приведенные в [8.8]. Средние выборочные значения $Mx, My[i], i = \overline{1,n}$, для матрицы Y предполагаются заранее вычисленными. Запись алгоритма имеет следующий вид:

```
Cov_calc (X, Y ,Mx, My, n; Cov)
   For i ← 1 to n                              1+3n
      s← 0                                       1
      For j ← 1 to n                           1+3n
         s ← s +(X[i] − Mx) *(Y[i,j] − My[i])   9
      end For j
   Cov[i] ← s                                    2
   end For i
Return (Cov)
End.
```

Данный алгоритм относится также к классу N, поэтому при фиксированной размерности исходных данных трудоемкости в лучшем, среднем и худшем случаях совпадают. Применяя сведение конструкции цикла по счетчику к базовым операциям, имеем

$$f_A(n) = 1 + n(3 + 1 + 1 + 2 + n(3 + 9)) = 12n^2 + 7n + 1 = \Theta\left(n^2\right).$$

Входом алгоритма являются два одномерных массива — **X, Y**, и двумерный массив **MY**, выход алгоритма — массив **Cov**, и алгоритм требует четыре ячейки — **i, j, Mx, s**, таким образом, функция объема памяти этого алгоритма

$$V_A(n) = n^2 + 3 \cdot n + 4,$$

а ресурсная сложность имеет вид

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

Пример 8.3. Алгоритм умножения двух квадратных матриц. Запись этого классического алгоритма имеет вид:

```
MultM (A, B, n; C)
   For i ← 1 to n                              1+3n
      For j ← 1 to n                           1+3n
         Sum ← 0                                 1
         For k ← 1 to n                        1+3n
            Sum ← Sum + A[i,k]*B[k,j]            7
         end For k
         C[i,j] ← Sum                            3
```

```
      end For j
   end For i
Return (C)
End.
```

Этот алгоритм является количественно-зависимым и относится к классу N. Все циклы не зависит друг от друга, что позволяет непосредственно применить формулу для трудоемкости конструкции «Цикл по счетчику», в результате, с учетом трудоемкости в строках, получаем

$$f_A(n) = 1 + n(3 + 1 + n(3 + 1 + 1 + 3 + n(3 + 7))) = 10n^3 + 8n^2 + 4n + 1 = \Theta\left(n^3\right).$$

Здесь под n также понимается линейная размерность, — подход общепринятый при анализе алгоритмов умножения квадратных матриц.

Входом алгоритма являются два массива, содержащие по n^2 элементов, такой же массив хранит вычисленный результат и алгоритм требует четыре ячейки для счетчиков циклов и хранения суммы, таким образом, функция объема памяти имеет вид:

$$V_A(n) = 3n^2 + 4.$$

Ресурсная сложность алгоритма задается формулой

$$\mathfrak{R}_c(A) = \left\langle \Theta\left(n^3\right), \Theta\left(n^2\right) \right\rangle.$$

Пример 8.4. Алгоритм вычисления матрицы расстояний между n точками.

Алгоритм предназначен для вычисления матрицы квадратов расстояний между всеми заданными точками. Координаты точек хранятся в массивах X и Y. Результатом является симметричная квадратная матрица S, размерностью $n \times n$ элементов, с нулевыми элементами на главной диагонали. Запись алгоритма имеет следующий вид:

```
Lcalc (X, Y, n; S)
   For i ← 1 to n                          1+3n
      S[i,i]← 0                              3
      For j ← i+1 to n                     2+3n
         dx ← (X[i]—X[j])                    4
         dy ← (Y[i]—Y[j])                    4
         S[i,j] ← dx*dx + dy*dy              6
         S[j,i] ← S[i,j]                     5
      end For j
   end For i
Return (S)
End.
```

Данный алгоритм относится также к классу N, поэтому при фиксированной размерности исходных данных трудоемкости в лучшем, среднем и худшем случаях совпадают. Но в этом алгоритме внутренний цикл зависит от внешнего, поэтому для получения функции трудоемкости нам необходимо просуммировать количество проходов внутреннего цикла $(n-i)$ при изменении внешнего (i), в результате получаем

$$\sum_{i=1}^{n}(n-i)=n^2-\frac{n(n+1)}{2}=\frac{n^2-n}{2}.$$

На основе полученного результата, применяя сведение конструкции цикла по счетчику к базовым операциям, получаем трудоемкость этого алгоритма

$$f_A(n)=1+n(3+3+2)+\frac{n^2-n}{2}(3+4+4+6+5)=11n^2-3n+1=\Theta\left(n^2\right).$$

Входом алгоритма являются два одномерных массива — **`X, Y`**, выход алгоритма — двумерный массив **`S`**, алгоритм требует четыре дополнительные ячейки — **`i, j, dx, dy`**, таким образом, функция объема памяти этого алгоритма

$$V_A(n)=n^2+2\cdot n+4,$$

а ресурсная сложность имеет вид

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

8.3. Метод вероятностного анализа количественно-параметрических алгоритмов

Класс количественно-параметрических алгоритмов (NPR) является достаточно обширным классом алгоритмов, которые имеют различную трудоемкость на разных входах фиксированной длины. В целях их детального анализа необходимо исследовать лучший, худший и средний случай трудоемкости. Для большинства практических применений в целях сравнительного анализа алгоритмов функция трудоемкости в среднем является наиболее значимой. Основные трудности анализа алгоритма в среднем случае связаны как с выяснением значений трудоемкости для конкретных входов, так и с определением вероятности их появления. В связи с этим задача вычисления $\overline{f_A}(n)$ для алгоритмов класса NPR является в целом ряде случаев довольно непростой.

Для получения сложностных оценок алгоритмов в среднем случае в литературе предлагаются различные методы, как общего, так и частного характера, например метод вероятностного анализа по классам входных данных [8.4], амортизационный анализ [8.3]. Рассматриваются также подходы, основанные на операторных схемах, подходы, связанные с использованием конечных цепей Маркова [8.9] и уравнений Крихгофа [8.6].

Рассмотрим один из методов анализа, в основе которого лежит определение различных классов входных данных, на которые следует разбивать возможные входы алгоритма фиксированной длины [8.4]. Метод предполагает проведение следующих этапов анализа:

— на первом этапе выполняется разбиение множества D_n на классы входных данных. Необходимо разбить различные входы из D_n на классы, в зависимости от поведения трудоемкости алгоритма на каждом входе. Если классы выбраны правильно, то на всех множествах входных данных одного класса алгоритм задает одинаковое количество базовых операций, а на множествах из другого класса это количество операций, скорее всего, будет другим. Такое разбиение позволяет уменьшить количество рассматриваемых возможных входов;

— на втором этапе анализа определяется вероятность появления входа, принадлежащего каждому классу. Распределение вероятностей по классам или определяется из общих соображений, например из предположения о равновероятном распределении значений чисел в массиве или их порядка, или отражает особенности частотного появления входов, обусловленные спецификой применения проектируемой программной системы;

— на третьем этапе определяется количество базовых операций, задаваемых алгоритмом, для данных из каждого входного класса при условии, что оно будет одинаковым для всех входов данного класса.

Определенные ограничения, накладываемые данным методом, состоят в том, что количество операций на всех входных данных, принадлежащих одному классу, должно быть одинаковым. Тем самым предполагается, что в рамках входных данных внутри каждого класса алгоритм является количественно-зависимым, а параметрическая зависимость функции трудоемкости наблюдается только при

переходе к входам другого класса, что усложняет в общем случае задачу разбиения на классы [8.4].

Трудоемкость алгоритма в среднем по данному методу определяется по следующей формуле:

$$\overline{f_A}(n) = \sum_{i=1}^{m} p_i \cdot f_i(n), \quad \sum_{i=1}^{m} p_i = 1,$$

где n — количество элементов во множестве $D \in D_n$, т. е. длина конкретного входа алгоритма; m — количество классов входных данных; p_i — вероятность того, что входные данные принадлежат классу i; $f_i(n)$ — количество базовых операций, задаваемых алгоритмом при обработке входов из класса с номером i — «частная» трудоемкость алгоритма для данного класса.

Преимущество этого метода состоит в том, что, вместо анализа трудоемкости для каждого входа фиксированной длины и определения вероятности его появления, выполняется анализ трудоемкости для классов входных данных, что приводит к значительному упрощению анализа.

Проиллюстрируем этот подход на примере алгоритма, находящего максимальный элемент в массиве из n чисел.

Пример 8.5. Поиск максимального элемента в массиве.

```
Max1(A, N; Max)
   Max ← A[1]
   For i ← 2 to N
      If Max < A[i]
         then
            Max ← A[i]
      end If
   end For
Return (Max)
End
```

Очевидно, что если список упорядочен в порядке убывания, то перед началом цикла будет сделано 1 присваивание, а в теле цикла дальнейших присваиваний не будет. Если список упорядочен по возрастанию, то всего будет сделано n присваиваний (одно — до цикла, и $(n-1)$ — в цикле).

При анализе трудоемкости алгоритма в среднем должны быть рассмотрены *различные* множества входных значений, поскольку, если мы ограничимся одним множеством, оно может оказаться тем самым, на котором решение самое быстрое

(или самое медленное), т. е. мы будем иметь лучший или худший случай трудоемкости. В этом случае необходимо разбить различные входные множества на классы в зависимости от поведения алгоритма на каждом множестве. Такое разбиение позволяет уменьшить количество рассматриваемых возможностей.

Пусть, например, наш список состоит из 10 несовпадающих чисел. Число различных расстановок 10 чисел в списке равно 10!=3 628 800. Применим к списку из 10 чисел вышеприведенный алгоритм поиска максимального элемента. Имеется 362 880 входных множеств, у которых первое число является наибольшим; их все можно поместить в первый класс (1 присваивание и 9 сравнений).

Если наибольшее по величине число стоит на втором месте, то алгоритм сделает 2 присваивания и 9 сравнений. Таких множеств тоже 362 880. Их можно отнести к другому классу.

Таким образом, мы разбиваем все входные множества на классы по числу сделанных присваиваний. Обратите внимание на то, что прямая аналогия не является правильной. Если максимальное число расположено на третьем месте, то число присваиваний не обязательно будет равно трем! Нет необходимости выписывать или описывать детально все множества, помещенные в каждый класс. Нужно знать лишь количество классов и объем работы алгоритма на каждом классе входных данных. Если классы выбраны правильно, то на всех множествах входных данных одного класса алгоритм производит одинаковое количество операций; а на множествах из другого класса это количество операций, скорее всего, будет другим. Реальные примеры применения этого метода будут приведены в главе 10, например, он будет использован для анализа алгоритма сортировки трех чисел по месту.

Обобщая идею метода классов входных данных, мы приходим к методу вероятностного анализа, основное положение которого может быть сформулировано следующим образом [8.2] — располагаем ли мы какой либо информацией или разумными предположениями, на основе которых можно определить вероятность того, что исследуемый алгоритм совершает некоторое определенное действие. Если ответ положителен, то на его основе можно получить функцию трудоемко-

сти данного алгоритма в среднем. Именно такой подход позволит нам далее получить среднюю трудоемкость алгоритма сортировки вставками.

8.4. Метод амортизационного анализа

Общие положения. Предположим, что у нас есть некоторая структура данных, например, стек, или более сложная структура — связанный список или динамическая таблица. Над этой структурой определены операции, обслуживающие добавление, удаление и пополнение данных в структуре. Тем самым в терминах объектно-ориентированного программирования мы имеем объект — структуру данных и обслуживающие эту структуру методы.

Очевидно, что различные операции со структурой имеют различную трудоемкость. Оценка в худшем случае для длительной последовательности операций может быть легко получена — необходимо умножить число операций в последовательности на трудоемкость наихудшей (в смысле числа базовых операций) операции. Но, в ряде случаев, можно получить более точную оценку, близкую к средней, опираясь на исследование всей длительной последовательности различных операций. Такой подход носит название «амортизационного анализа», по аналогии с изучением износа оборудования за длительный срок эксплуатации. Сам термин «амортизационный анализ» введен Слейтором и Тарьяном [8.3].

Основная идея метода, имеющего целый ряд разновидностей, состоит в том. что каждой операции присваивается некоторая учетная стоимость, отражающая трудоемкость операции. При этом не обязательно, что эта стоимость должна быть пропорциональна реальной трудоемкости. Правило присвоения учетных стоимостей должно лишь обеспечивать выполнение следующего условия: суммарная трудоемкость выполнения некоторой последовательности операций над структурой данных не больше суммы учетных стоимостей операций. Заметим, что в рамках этого правила возможны различные способы назначения учетных стоимостей исследуемым операциям.

В рамках амортизационного анализа рассматривается несколько методов. В методе группировки оценивается стоимость n операций в худшем случае, если удается установить, что эта стоимость не превосходит $c(n)$, то можно считать, что

учетная стоимость любой операции, независимо от ее реальной трудоемкости, равна $c(n)/n$. В методе предоплаты разным операциям могут быть присвоены различные учетные стоимости, у некоторых из них эти стоимости объявляются выше реальной трудоемкости. При выполнении такой операции остается резерв этой учетной стоимости, который считается хранящимся в определенном месте структуры данных. Этот резерв используется в дальнейшем для доплаты за операции, учетная стоимость которых ниже их реальной трудоемкости. Еще один метод амортизационного анализа — метод потенциалов использует специальную функцию, описывающую текущее состояние структуры данных, которая и называется «потенциалом». Для заинтересованных читателей укажем, что метод потенциалов достаточно хорошо изложен, например, в [8.3]. Применение метода для анализа операций со стеком будет приведено ниже, другие примеры читатель найдет в разделе V, в частности метод группировки будет использован для анализа эффективного алгоритма организации двоичного счетчика.

Модельная структура данных. В качестве примера применения двух методов амортизационного анализа — метода группировки и метода предоплаты к анализу последовательности операций над структурой данных будем использовать структуру данных стека, который обслуживается следующими операциями:

— **`Push(x)`** добавляет элемент из ячейки **`x`** в стек, проталкивая все ранее хранящиеся элементы стека вниз;

— **`Pop(x)`** извлекает элемент из стека, проталкивая все ранее хранящиеся элементы стека вверх, и записывает его в ячейку **`x`**;

— **`Multipop(x, k)`** удаляет **`k`** элементов из стека, при этом, если в стеке храниться менее **`k`** элементов, то удаляются все элементы стека.

При организации стека в оперативной памяти, оперируя с указателем на вершину стека, первые две операции — **`Push(x)`** и **`Pop(x)`** требуют не более чем $\Theta(1)$ базовых операций, следовательно, выполнение n простых операций со стеком требует не более, чем $\Theta(n)$ базовых операций.

Трудоемкость операции **`Multipop(x, k)`** очевидно зависит от состояния стека, и если мы обозначим через m число элементов, хранящихся в данный момент времени в стеке, то ее трудоемкость может быть оценена как $\Theta(\min(m,k))$.

Наша задача состоит в оценке учетной стоимости этих операций методом группировки и методом предоплаты.

Метод группировки. Будем исходить из предположения, что изначально стек пуст, и мы рассматриваем последовательность из n операций со стеком. В этом случае общее число реально выполняемых операций **Pop(x)**, включая те, которые инициируются операцией **Multipop(x, k)**, не превосходит общего числа операций **Push(x)** — мы можем извлечь из стека только те данные, которые были туда помещены. Поскольку мы рассматриваем последовательность из n операций со стеком, то общее число операций **Push(x)** также не превосходит n. Таким образом, стоимость любой последовательности из n операций **Push(x)**, **Pop(x)** и **Multipop(x, k)**, примененных к пустому стеку, есть $\Theta(n)$. На основании этого утверждения метод группировки позволяет сделать вывод о том, что учетная стоимость любой из рассматриваемых операций, в частности и операции **Multipop(x, k)**, есть $\Theta(n)/n = \Theta(1)$.

Метод предоплаты. Метод предполагает назначение операциям со структурой данных некоторых учетных стоимостей в соответствии с определенными условиями, а именно — сумма учетных стоимостей должна покрывать реальную трудоемкость исследуемой последовательности операций. Эти учетные стоимости могут иметь как постоянные значения, так и быть описаны некоторыми функциональными зависимостями.

Для рассматриваемой структуры данных присвоим операциям следующие учетные стоимости:

Push(x) — 2

Pop(x) — 0

Multipop(x, k) — 0

Покажем, что выбранные учетные стоимости позволяют покрыть реальную трудоемкость операций со стеком. Первоначально стек пуст, и при добавлении элемента в стек, мы платим единицу учетной стоимости — реальную цену этой операции. При этом еще одна единица учетной стоимости остается прикрепленной к элементу, помещенному в стек — это есть предоплата за удаление этого элемента из стека. Таким образом, все операции **Pop(x)** и **Multipop(x, k)** оказываются

оплаченными, именно поэтому присвоенные им учетные стоимости равны нулю. При рассмотрении последовательности из n операций **`Push(x)`**, **`Pop(x)`** и **`Multi-pop(x, k)`** в худшем случае сумма учетных стоимостей не превышает $2n$, и, следовательно, учетная стоимость на одну операцию есть $\Theta(1)$.

Задачи и упражнения к главе 8

8.1. Языки программирования поддерживают такую дополнительную алгоритмическую конструкцию как «Выбор» (Case). Сведите эту конструкцию к введенным базовым операциям, используя конструкцию «Ветвление».

8.2. В одном из примеров параграфа 8.2 была получена функция трудоемкости для алгоритма вычисления квадратов расстояний между точками — пример 8.4. Если мы будем вычислять расстояния, то необходимо вызывать функцию вычисления квадратного корня. Сформулируйте свои предложения по поводу анализа трудоемкости такого алгоритма.

8.3. Если учитывать только дополнительную память, которая требуется алгоритмами, приведенными в примерах параграфа 8.2 — как измениться функция объема памяти и ресурсная сложность этих алгоритмов?

8.4. Для примера 8.5 из параграфа 8.3 определите, для ситуации, когда максимальный элемент расположен на третьем месте в массиве, сколько выходных массивов приводят к двум присваиваниям максимума, а сколько — к трем?

8.5. Метод амортизационного анализа был проиллюстрирован в параграфе 8.4 на примере стека. Проведите аналогичные рассуждения в случае, если исследуемой структурой данных является очередь.

Список литературы к главе 8

[8.1.] Грэхем Р., Кнут Д., Паташник О. Конкретная математика. Основание информатики: Пер. с англ. — М.: Мир, 1998. — 703 с.

[8.2.] Гудман С., Хидетниеми С. Ведение в разработку и анализ алгоритмов. — М.: Мир, 1981. — 368 с.

[8.3.] Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ, 2-ое изд.: Пер. с англ. — М.: Издательский дом «Вильямс», 2005. — 1296 с.

[8.4.] Макконелл Дж. Основы современных алгоритмов. 2-е дополненное издание. — М.: Техносфера, 2004. — 368 с.

[8.5.] Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов: Пер. с англ.: — М.: Мир, 1979. — 546 с.

[8.6.] Кнут Д. Э. Искусство программирования, том 1. Основные алгоритмы, 3-е изд.: Пер. с англ. — М.: Издательский дом «Вильямс», 2002. — 720 с.

[8.7.] Ульянов М. В. Классификация и методы сравнительного анализа вычислительных алгоритмов. Научное издание. — М.: Издательство физико-математической литературы, 2004. — 212 с.

[8.8.] Тюрин Ю. Н., Макаров А. А. Анализ данных на компьютере / Под ред. В.Э. Фигурнова. — 3-е изд., перераб. и доп. — М.: ИНФРА, 2003. — 544 с.

[8.9.] Кемени Дж., Снелл Дж. Конечные цепи Маркова: Пер. с англ. М.: Наука, 1970.

Г Л А В А 9 .

А Н А Л И З Р Е К У Р С И В Н Ы Х А Л Г О Р И Т М О В

Введение

В отличие от итерационных алгоритмов, рекурсивные алгоритмы обладают рядом особенностей, которые необходимо учитывать при их анализе. Основной особенностью анализа ресурсной эффективности рекурсивных алгоритмов является необходимость учета дополнительных затрат памяти и трудоемкости, связанных с механизмом организации рекурсии. Получение ресурсных функций компьютерных 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 цепочек рекурсивных вызовов и возвратов, образующих порожденное данным алгоритмом и входом дерево 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. Все эти операции должны быть также включены в функцию трудоемкости рекурсивно реализованного алгоритма.

В функции объема памяти так же необходимо учитывать затраты на организацию рекурсии. Можно заметить, что некоторая ветвь дерева рекурсивных вызовов обрывается при достижении такого значения передаваемого параметра, при котором функция может быть вычислена непосредственно. Следовательно, рекурсия в этом смысле эквивалентна конструкции цикла, в котором каждый проход есть выполнение рекурсивной функции с заданным параметром. Таким образом, дополнительные затраты памяти на организацию рекурсии пропорциональны максимальной глубине дерева рекурсии.

Для оценки и детального анализа ресурсной эффективности рекурсивных алгоритмов применяется ряд специальных методов, основные из которых и излагаются в этой главе.

9.1. Анализ вычислительной сложности рекурсивных алгоритмов, основанных на методе декомпозиции

Метод декомпозиции, изложенный в главе 6, приводит непосредственно к рекурсивному алгоритму решения задачи с понижением размерности на каждом шаге рекурсии. Анализ трудоемкости таких алгоритмов приводит к необходимости решения функциональных рекуррентных соотношений определенного вида. Однако, если наша задача состоит в том, чтобы получить только асимптотическую оценку трудоемкости — вычислительную сложность, то в нашем распоряжении следующая теорема, доказанная в 1980 г. Дж. Бентли, Д. Хакен и Дж Саксом [9.1]. Полученный авторами результат является достаточно мощным средством асимптотической оценки рекурсивно заданных функций определенного вида, и применим к анализу трудоемкости рекурсивных алгоритмов, основанных на методе декомпозиции.

Теорема. (J. L. Bentley, Dorothea 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 $f(n) = O\left(n^{\log_b a - \varepsilon}\right)$ для некоторого $\varepsilon > 0$, то $T(n) = \Theta\left(n^{\log_b a}\right)$;

2) если $f(n) = \Theta\left(n^{\log_b a}\right)$, то $T(n) = \Theta\left(n^{\log_b a} \log_b n\right)$;

3) если найдутся $c > 0$ и $\varepsilon > 0$, такие, что при достаточно больших n выполнено условие

$$f(n) > cn^{\log_b a + \varepsilon}$$

и найдется положительная константа $d < 1$ такая, что при достаточно больших n выполнено

$$af\left(\frac{n}{b}\right) \leq df(n),$$

то

$$T(n) = \Theta\left(f(n)\right).$$

Рассмотрим примеры асимптотической оценки функциональных рекуррентных соотношений с использованием данной теоремы. Во всех нижеследующих примерах константа c считается строго положительной.

Пример 9.1. Предположим, что метод декомпозиции при разработке алгоритма решения некоторой задачи приводит к разделению задачи на четыре части и возникновению восьми подзадач меньшей размерности, причем деление и объединение решений имеют линейную трудоемкость. В этом случае функция $T(n)$, очевидно, имеет вид

$$T(n) = 8T\left(\left[\frac{n}{4}\right]\right) + cn.$$

В обозначениях теоремы $a = 8$, $b = 4$, $f(n) = cn$, 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 9.2. Предположим, что метод декомпозиции приводит к такому разбиению задачи, что функция $T(n)$ имеет вид

$$T(n) = T\left(\left[\frac{3n}{4}\right]\right) + 5.$$

Для получения асимптотической оценки функции $T(n)$ выпишем коэффициенты в обозначениях теоремы — $a = 1$, $b = 4/3$, $f(n) = 5$, а $n^{\log_b a} = n^{\log_{4/3} 1} = n^0 = \Theta(1)$. На этом основании воспользуемся вторым случаем теоремы. Поскольку

$$f(n) = 5 = \Theta\left(n^{\log_b a}\right) = \Theta(1),$$

то получаем, что $T(n) = \Theta(\log_{4/3} n)$.

Пример 9.3. Пусть метод декомпозиции при разработке алгоритма решения задачи приводит к разделению задачи на четыре части и возникновению двух подзадач меньшей размерности, а деление и объединение решений имеет следующую трудоемкость

$$f(n) = cn \log_4 n.$$

Для этого примера функция $T(n)$ может быть записана в виде

$$T(n) = 2T\left(\left[\frac{n}{4}\right]\right) + cn\log_4 n.$$

Имеем (в обозначениях теоремы) $a = 2$, $b = 4$, $f(n) = cn\log_4 n$, а $n^{\log_b a} = n^{\log_4 2} = n^{0,5} = \Theta\left(\sqrt{n}\right)$ и, например, при больших n и $\varepsilon = 0{,}5$

$$f(n) = cn\log_4 n \geq n^{0,5+\varepsilon}.$$

Это третий случай теоремы и остается проверить последнее условие для третьего случая. Для достаточно больших значений n имеем

$$2f\left(\frac{n}{4}\right) = 2\frac{n}{4}\log_4\frac{n}{4} \leq \frac{1}{2}n\log_4 n = df(n), \text{ где } d = \frac{1}{2} < 1,$$

и, тогда $T(n) = \Theta(n\log_4 n)$.

Пример 9.4. 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 $f(a) < 0$, $f(b) > 0$. Мы будем считать, что корень определен с нужной степенью точности, если мы определим такой отрезок $[c;d]$, что $f(c) < 0$, $f(d) > 0$ и при этом $d - c \leq h$.

Рассмотрим следующий алгоритм [9.2] для определения корня уравнения $f(x) = 0$. Обозначим $a_1 = a$, $b_1 = b$. Мы можем утверждать, поскольку $f(a_1) < 0$, $f(b_1) > 0$, а функция непрерывная на отрезке принимает все промежуточные значения, что неизвестный корень x уравнения $f(x) = 0$ лежит на отрезке $[a_1;b_1]$. Идея алгоритма состоит в построении такой последовательности отрезков $[a_i;b_i]$, что $f(a_i) < 0$, $f(b_i) > 0$, и при этом $b_{i+1} - a_{i+1} < b_i - a_i$, то есть на каждом шаге алгоритма длина отрезка, на котором лежит корень уравнения, уменьшается. Возь-

мем значение c в середине отрезка $c=\frac{a_i+b_i}{2}$ и вычислим значение $f(c)$. Возможны два случая $f(c)<0$ или $f(c)>0$. (Случай $f(c)=0$ в виду 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, когда $f(c)>0$, полагаем $a_{i+1}=a_i$, $b_{i+1}=c$. Далее вычисления повторяются. Условием останова алгоритма является выполнение условия $b_{i+1}-a_{i+1}<h$.

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 асимптотической оценки количества необходимых базовых операций этого алгоритма получаем соотношение вида

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

где значение c определяется количеством операций, требуемых для вычисления функции и соответствующих операции сравнения и присваивания новых значений концам текущего отрезка. Для значения $T(n)$ с использованием основной теоремы получаем следующую асимптотическую оценку

$$T(n)=\Theta(\log_2 n).$$

В дальнейшем, при изложении материала раздела V, эта теорема будет применяться для получения асимптотических оценок трудоемкости эффективных алгоритмов и ресурсно-эффективных алгоритмических решений.

9.2. Метод подсчета вершин дерева рекурсии

Прежде чем излагать метод подсчета вершин дерева рекурсии для получения ресурсных функций рекурсивных алгоритмов, необходимо оценить ресурс-

ные затраты на организацию рекурсии, в основе которых лежит трудоемкость вызова рекурсивной функции.

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 [9.3]. Как передаваемые в процедуру или функцию фактические параметры, так и возвращаемые из них значения, помещаются в программный стек специальными командами процессора — эти операции считаются базовыми в нашей модели вычислений.

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 осуществляет переход по этому адресу. Для анализа трудоемкости механизма вызова-возврата для рекурсивной процедуры введем следующие обозначения

p — количество передаваемых фактических параметров,

k — количество возвращаемых по адресной ссылке значений,

r — количество сохраняемых в стеке регистров,

l — количество локальных ячеек процедуры.

Обозначим трудоемкость обслуживания рекурсивной процедуры на один вызов-возврат через $f_R(1)$. Поскольку каждый объект в некоторый момент помещается в стек, и в какой-то момент выталкивается из него, то трудоемкость обслуживания рекурсивной процедуры в базовых операциях составит:

$$f_R(1) = 2 \cdot (p + k + r + l + 1). \qquad (9.2.1)$$

Дополнительная единица в формуле (9.2.1) 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:

$$f_R(1) = 2 \cdot (p + k + r + f + l + 1),\ f = 1. \qquad (9.2.2)$$

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 (9.2.1) и (9.2.2) отражают второй способ, заметим также, что если положить $f = 0$ для рекурсивной процедуры в формуле (9.2.2), то она сводится к (9.2.1).

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 сравнительном 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 [9.4, 9.5]. 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, порождаемого 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, то максимальный объем памяти в области стека может быть определен на основе (9.2.2) следующим образом

$$V_{st}(D) = H_R(D) \cdot (p + k + r + f + l + 1) \cdot l_w, \qquad (9.2.3)$$

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 (9.2.2), 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, [9.1, 9.4, 9.5]). Однако общая трудоемкость определяется числом порожденных вершин дерева рекурсии, кроме того, необходимо учесть, что в общем случае трудоемкость в разных вершинах может различаться, и определяться как данными, так и параметрами рекурсии.

Анализ 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 рекур-

сивных функций (ресурсные затраты обслуживания рекурсии) и ресурсные затраты в листьях дерева рекурсии, учитывающие специфику фрагмента останова.

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 — это и есть основная идея метода подсчета вершин дерева рекурсии для анализа трудоемкости рекурсивных алгоритмов [9.6].

Первым этапом метода является исследование дерева рекурсии. В предположении, что n — длина входа алгоритма для классов N, NPR; $m_1, \cdots m_k$ — значение параметров входа для классов PR, NPR $(k \le n)$, 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

$$R(D) = R_V(D) + R_L(D),\ H_R(D) \le R_V(D) + 1.$$

Рассмотрим более подробно особенности функциональной зависимости общего числа вершин в дереве рекурсии для алгоритмов, принадлежащих различным классам по характеристическим особенностям множества исходных данных. В соответствие с определениями классов (см. гл. 4) имеем

1. Класс N: $\quad R(D) = R(n) \quad \forall D \in D_n$;

2. Класс PR: $\quad R(D) = R(\ m_1, \cdots m_k)$;

3. Класс NPR: $\quad R(D) = R(\ n,\ m_1, \cdots m_k)$.

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 от

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. Эта характеристика относительной «ширины» нижнего уровня (уровня листьев) в дереве рекурсии

$$B_L(D) = \frac{R_L(D)}{R(D)}, \quad 0 < B_L(D) < 1. \tag{9.2.4}$$

Значение $B_L(D)$ будет минимально для цепочки (унарного дерева), и будет возрастать при увеличении числа вершин, порожденных во внутренних вершинах дерева рекурсии.

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 алгоритма A на конкретном входе D — $f_A(D)$ 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

$$f_A(D) = f_R(D) + f_C(D). \tag{9.2.5}$$

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 — $f_R(1)$, определяемых, например, по формуле (9.2.2), то

$$f_R(D) = R(D) \cdot f_R(1). \tag{9.2.6}$$

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 вер-

шинах, следовательно, трудоемкость алгоритма при останове рекурсии должна быть учтена отдельно. В связи с этим обозначим через

$f_{CV}(D)$ — трудоемкость вычислений во внутренних вершинах,

$f_{CL}(D)$ — трудоемкость продуктивных вычислений в листьях дерева рекурсии, это приводит к определению $f_C(D)$ в виде

$$f_C(D) = f_{CV}(D) + f_{CL}(D). \tag{9.2.7}$$

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 фиксированным числом базовых операций. Зная количество листьев рекурсивного дерева, можно определить $f_{CL}(D)$

$$f_{CL}(D) = R_L(D) \cdot f_{CL}(1). \tag{9.2.8}$$

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

$$v_{mk}, m = \overline{1, H_R(D)-1}; k = \overline{1, K(m)},$$

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

$$f_{CV}(D) = \sum_{m=1}^{H_R(D)-1} \sum_{k=1}^{K(m)} f_{CV}(v_{mk}). \qquad (9.2.9)$$

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

$$f_{CV}(D) = R_V(D) \cdot f_{CV}(v). \qquad (9.2.10)$$

Подставляя полученные результаты в формулу (9.2.5), окончательно получаем формулу для определения трудоемкости рекурсивного алгоритма на основе метода подсчета вершин дерева рекурсии в общем случае

$$f_A(D) = R(D) \cdot f_R(1) + R_L(D) \cdot f_{CL}(1) + \sum_{m=1}^{H_R(D)-1} \sum_{k=1}^{K(m)} f_{CV}(v_{mk}), \qquad (9.2.11)$$

и в частном случае, когда трудоемкость продуктивных вычислений для любой внутренней вершины дерева рекурсии одинакова

$$f_A(D) = R(D) \cdot f_R(1) + R_L(D) \cdot f_{CL}(1) + R_V(D) \cdot f_{CV}(v). \qquad (9.2.12)$$

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

$$F_R(D) = \frac{f_R(D)}{f_A(D)}, \quad 0 < F_R(D) < 1. \qquad (9.2.13)$$

Значение $F_R(D)$ показывает насколько трудоемкость обслуживания дерева рекурсии значима в общей трудоемкости рекурсивного алгоритма.

В заключении 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— $R(n, m_1, \cdots m_k)$, $R_L(n, m_1, \cdots m_k)$, $R_V(n, m_1, \cdots m_k)$, $H_R(n, m_1, \cdots m_k)$, как функций от длины входа (n) и/или характеристических особенностей множества входных данных.

2. Определение трудоемкости обслуживания рекурсии на одни вызов-возврат — $f_R(1)$, например, по формуле (9.2.2).

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 (9.2.11) или (9.2.12) 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, что бы используя идеи амортизационного анализа [9.1], попытаться вместо аналитического задания функции $f_{CV}(v_{mk})$ получить ее сумму либо по уровням дерева, либо по всем внутренним вершинам. Некоторые примеры анализа трудоемкости рекурсивных алгоритмов с использованием этого метода и некоторых его модификаций будут приведены в разделе V.

9.3. Метод рекуррентных соотношений

Этот 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 трудо-

емкости, и решения этого соотношения с помощью известных методов, например тех, которые изложены в [9.6]. Для этого метода не может быть предложена универсальная формула, 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 размерностью n, происходит такое ее разделение, которое приводит к необходимости решения a подзадач размерностью n/b (b является делителем n), то функция трудоемкости имеет вид:

$$f_A(n) = a \cdot f_A(n/b) + d(n) + U(n), \tag{9.3.1}$$

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 $f_A(n_0)$, получаем общую форму рекуррентного соотношения для функции трудоемкости алгоритмов, разработанных методом декомпозиции

$$\begin{cases} f_A(n_0), & n = n_0; \\ f_A(n) = a \cdot f_A(n/b) + d(n) + U(n), & n > n_0. \end{cases} \tag{9.3.2}$$

Дополнительное уточнение касается трудоемкости организации рекурсии, которая должна быть учтена на каждом рекурсивном вызове. Эта трудоемкость может быть получена по формуле (9.2.1), и, сохраняя обозначение $f_R(1)$, получаем

$$\begin{cases} f_A(n_0) + f_R(1), & n = n_0; \\ f_A(n) = a \cdot f_A(n/b) + d(n) + U(n) + f_R(1), & n > n_0. \end{cases} \tag{9.3.3}$$

Еще один нюанс, серьёзно усложняющий получение решения, состоит в том, что размерность решаемой задачи должна быть целой. Поэтому, в общем случае, вме-

сто n/b в качестве аргумента функции в (9.3.2) 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 [9.1]. 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 $a \cdot n + b$, где коэффициенты определяются на основе анализа алгоритма слияния. Организация рекурсии на один вызов требует фиксированного числа операций $f_R(1) = c_3$, и, используя (9.3.3), получаем рекуррентное соотношение для функции трудоемкости алгоритма сортировки слиянием

$$\begin{cases} c_2 + c_3, \quad n = 1; \\ f_A(n) = f_A(\lfloor n/b \rfloor) + f_A(\lceil n/b \rceil) + c_1 + a \cdot n + b + c_3, \, n > 1. \end{cases}$$

Рассмотрим еще одно обобщение для рекурсивных алгоритмов, понижающих 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 параграфе 9.2, — $f_{CL}(1)$ для трудоемкости останова рекурсии (при этом $f_{CL}(1) = f_A(n_0)$) и $f_R(1)$ для трудоемкости обслуживания рекурсии на один вызов/возврат, получаем

$$\begin{cases} f_A(n_0) = f_R(1) + f_{CL}(1), \, n = n_0; \\ f_A(n) = f_A(n-1) + f_R(1) + f_{CV}(n), \, n > n_0. \end{cases} \tag{9.3.4}$$

Приведем пример использования (9.3.4) для построения рекуррентного соотношения, задающего функцию трудоемкости алгоритма вычисления определителя квадратной матрицы. Останов рекурсии происходит при $n = 2$, и требует

фиксированного числа базовых операций — обозначим это число через c_1. Трудоемкость организации рекурсии на один вызов так же требует фиксированного числа операций $f_R(1) = c_2$. Трудоемкость сведения задачи вычисления определителя матрицы размерностью n к задаче размерностью $n-1$ определяется числом элементов матрицы, и может быть представлена в виде [9.6]

$$f_{CV}(n) = a \cdot n^2 + b \cdot n + d\,,$$

где коэффициенты могут быть определены путем анализа трудоемкости фрагмента сведения матрицы с использованием одного их методов, изложенных в главе 8. Подставляя полученные результаты в (9.3.4) мы получаем рекуррентное соотношение, задающее функцию трудоемкости данного алгоритма

$$\begin{cases} f_A(2) = c_1 + c_2; \\ f_A(n) = f_A(n-1) + a \cdot n^2 + b \cdot n + d + c_2, \; n > 2. \end{cases}$$

Решение может быть получено методами, изложенными, например, в [9.6].

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. С другой стороны, метод рекуррентных соотношений требует только построения самого соотношения и решения его известными методами, и если детализация трудоемкости не требуется, то является, очевидно, более предпочтительным.

Задачи и упражнения к главе 9

9.1. При анализе трудоемкости вызова функции считается, что все локально описанные ячейки и массивы должны быть сохранены в программном стеке при рекурсивном вызове. Как измениться формула (9.2.1), если одномерный массив из n элементов описан в некоторой процедуре как локальный?

9.2. В смысле ресурсной эффективности рекурсивного алгоритма, какое дерево рекурсии является более предпочтительным — унарное дерево, содержащее n вершин на n уровнях, или $n-1$-арное дерево, содержащее корень и $n-1$ вершину на втором уровне?

9.3. 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 (9.2.4).

9.4. Как Вы считаете — доля операций обслуживания дерева рекурсии, задаваемая формулой (9.2.13), будет расти или падать, если, зафиксировав общее количество вершин, изменять ширину дерева, и, следовательно, число его уровней?

Список литературы к главе 9

[9.1] Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ, 2-ое изд.: Пер. с англ. — М.: Издательский дом «Вильямс», 2005. — 1296 с.

[9.2] Бахвалов Н. С., Жидков Н. П., Кобельков Г. М. Численные методы. — М.: Лаборатория Базовых Знаний, 2001 г. — 632 с.

[9.3] Архангельский А.Я. Язык Pascal и основы программирования в Delphi. — М.:Бином, 2004. — 496с.

[9.4] Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов: Пер. с англ.: — М.: Мир, 1979. — 546 с.

[9.5] Ахо А., Хопкрофт Дж., Ульман Дж. Структуры данных и алгоритмы: Пер. с англ.: — М.: Издательский дом «Вильямс», 2001. — 384 с.

[9.6] Головешкин В. А., Ульянов М. В. Теория рекурсии для программистов. — М.: Издательство «Наука ФИЗМАТЛИТ», 2006. — 296 с.

РАЗДЕЛ V

РАЗРАБОТКА И ВЫБОР ЭФФЕКТИВНЫХ АЛГОРИТМОВ НА ОСНОВЕ РЕСУРСНОГО АНАЛИЗА

Материал этого раздела содержит целый ряд иллюстративных примеров ресурсно-эффективных компьютерных алгоритмов, начиная с алгоритмических решений для достаточно простых задач с фиксированной длиной входа. Далее излагаются некоторые ресурсно-эффективные алгоритмические решения для задач с переменной длиной входа, включая обоснование рекомендаций по рациональному выбору алгоритмов. Изложение ресурсно-эффективных комбинированных алгоритмических решений в последней главе этого раздела призвано продемонстрировать применение теоретического аппарата к решению ряда практически значимых задач.

ГЛАВА 10.

ПРИМЕРЫ ЭФФЕКТИВНЫХ АЛГОРИТМОВ ДЛЯ ЗАДАЧ С ФИКСИРОВАННОЙ ДЛИНОЙ ВХОДА

Введение

Достаточно простые задачи с фиксированной длиной входа, используемые в этой главе как «модельные» примеры, позволяют, тем не менее, продемонстрировать возможность разработки «веера» алгоритмов их решения, обладающих различными ресурсными характеристиками. Для всех алгоритмов этой главы приводится детальный анализ трудоемкости и емкостной эффективности, на основе которого, в практике программирования, можно принимать обоснованные решения по их выбору.

10.1. Обмен содержимого ячеек

Рассмотрим очень простую задачу, два разных алгоритма решения которой представляют собой характерную пару в аспекте изучения ресурсной эффективности. Это задача обмена содержимого двух ячеек, часто используемая как фрагмент многих других алгоритмов, например, алгоритмов сортировки. Первый алгоритм, назовем его **Swap_A1**, использует для обмена одну дополнительную ячейку памяти:

```
Swap_A1 (a, b)
   t ← b                                   1
   b ← a                                   1
   a ← t                                   1
Return (a, b)
End.
```

Справа в каждой строке приведено число базовых операций, задаваемых данной строкой в записи алгоритма. Такая информация будет приводиться и далее для всех обсуждаемых алгоритмов. Любой вход D для этой задачи всегда состоит из двух ячеек, а трудоемкость (без трудоемкости вызова/возврата процедуры) и функция объема дополнительной памяти алгоритма **Swap_A1** имеют вид:

$$f_{A1}(D)=3,\ V_{A1}(D)=1. \qquad (10.1.1)$$

Алгоритмически задача становится более интересной, если критерий емкостной эффективности является значимым. Можно ли предложить алгоритм обмена, не использующий дополнительную ячейку? Алгоритм **Swap_A2** дает положительный ответ на этот вопрос, а его запись имеет вид:

```
Swap_A2 (a, b)
   b ← a+b                                 2
   a ← b-a                                 2
   b ← b-a                                 2
Return (a, b)
End.
```

Функции трудоемкости и объема дополнительной памяти для алгоритма **Swap_A2** имеют вид:

$$f_{A2}(D)=6,\ V_{A2}(D)=0. \qquad (10.1.2)$$

Необходимо отметить, что алгоритм **Swap_A2** накладывает некоторые ограничения на значения в ячейках, связанные с вычислением суммы, но в программной реализации эти ограничения можно снять, используя вместо операций сложения и вычитания операцию сложения по модулю два.

Автор надеется, что читателям очевидна принадлежность алгоритмов **Swap_A1** и **Swap_A2** к классу C в типе L_c, а вот по объему дополнительной памяти они относятся к классам VC и $V0$ соответственно.

Из (10.1.1) и (10.1.2) следует, что отсутствие дополнительной ячейки памяти «стоило» нам трех дополнительных операций. Рассмотрение этих двух алгоритмов с точки зрения критерия ресурсной эффективности приводит к постановке следующего вопроса — оправдано ли такое увеличение трудоемкости экономией одной ячейки памяти? В общем случае ответ не очевиден, до тех пор, пока неизвестны веса ресурсных характеристик в комплексном критерии оценки качества алгоритма. С другой стороны понятно, что эти веса отражают особенности проблемной области, для которой разрабатываются программные средства, и, в конце концов, определяются техническим заданием на разработку. Рассмотренная ситуация очень характерна в области разработки эффективных алгоритмов, поскольку для целого ряда задач возможна разработка таких алгоритмов, которые за счет значительного увеличения объема дополнительной памяти позволяют улучшить временную эффективность. Очевидно, что в этом случае задача выбора рационального алгоритма определяется соотношением требуемых и имеющихся в наличии вычислительных ресурсов.

10.2. Умножение комплексных чисел

Рассмотрим задачу умножения двух комплексных чисел, действительные и мнимые части которых заданы действительными числами. Произведение вычисляется по известной формуле

$$(a+i\,b)(c+i\,d)=(ac-bd)+i\,(ad+bc)=g+i\,h,$$

$$g=(ac-bd),\ h=(ad+bc). \qquad (10.2.1)$$

Нашей задачей является обоснование, путем сравнительного пооперационного анализа, рационального выбора одного из двух алгоритмов, выполняющих вычисление значений g и h за четыре и за три операции умножения. Непосредственное использование формулы (10.2.1) приводит к алгоритму **MultC_A1**, вычисляющему произведение двух комплексных чисел за четыре операции умножения:

```
MultC_A1 (a, b, c, d; g, h)
   g ← a*c - b*d                                4
   h ← a*d + b*c                                4
Return (g, h)
End.
```

Элементарным подсчетом определятся, что для любого входа D, состоящего для этой задачи всегда из четырех чисел, пооперационная трудоемкость алгоритма **MultC_A1** составляет:

$$f_{A1(*)}(D)=4,\ f_{A1(\pm)}(D)=2,\ f_{A1(\leftarrow)}(D)=2, \qquad (10.2.2)$$

где в нижнем индексе в скобках указаны операции из набора базовых операций принятой модели вычислений (см. § 2.5), а суммарная трудоемкость $f_{A1}(D)=8$.

Небольшие алгебраические преобразования, аналогичные подходу, использованному А. А. Карацубой при разработке рекурсивного алгоритма умножения длинных целых чисел [10.1], позволяют получить другой алгоритм, выполняющий при вычислении значений g и h только три операции умножения. Запись алгоритма **MultC_A2** имеет вид:

```
MultC_A2 (a, b, c, d; g, h)
   z1←c*(a + b)                                 3
   z2←b*(d + c)                                 3
   z3←a*(d - c)                                 3
   g←z1 - z2                                    2
   h←z1 + z3                                    2
Return (g, h)
End.
```

Аналогично определятся, что для любого входа D пооперационная трудоемкость алгоритма **MultC_A2** составляет:

$$f_{A2(*)}(D)=3,\ f_{A2(\pm)}(D)=5,\ f_{A2(\leftarrow)}(D)=5, \qquad (10.2.3)$$

а суммарная трудоемкость $f_{A2}(D)=13$.

Алгоритмы **MultC_A1** и **MultC_A2** имеют фиксированную длину входа, и задаваемое ими число базовых операций не зависит от числовых значений, таким образом, оба алгоритма принадлежат к типу L_c, и классу C в этом типе.

По суммарной трудоемкости алгоритм **MultC_A2** уступает алгоритму **MultC_A1** более чем в полтора раза, однако в реальных процессорах операция умножения требует большего времени в среднем, чем операция сложения. В этом предположении, используя результаты пооперационного анализа, можно ответить

на вопрос, при каких условиях (по соотношению времен выполнения базовых операций) программная реализация алгоритма **MultC_A2** будет предпочтительнее реализации алгоритма **MultC_A1**? Введем с этой целью коэффициенты q и r, устанавливающие соотношения между средними временами выполнения механизмом реализации операции сложения, и операций умножения и присваивания. Используя коэффициенты q и r можно привести временные оценки программных реализаций алгоритмов к времени выполнения операции сложения/вычитания — $t_{(\pm)}$:

$$t_{(*)} = q\,t_{(\pm)},\ q>1,\ t_{(\leftarrow)} = r\,t_{(\pm)},\ r<1,$$

где $t_{(*)}$ — время умножения, а $t_{(\leftarrow)}$ — время операции присваивания. Тогда приведенные к $t_{(\pm)}$ временные оценки программных реализаций алгоритмов на основе (10.2.2) и (10.2.3) имеют вид

$$t_{A1} = 4q\,t_{(\pm)} + 2t_{(\pm)} + 2r\,t_{(\pm)} = t_{(\pm)}(4q+2+2r),$$

$$t_{A2} = 3qt_{(\pm)} + 5t_{(\pm)} + 5rt_{(\pm)} = t_{(\pm)}(3q+5+5r).$$

Равенство времен будет достигнуто при условии

$$4q+2+2r = 3q+5+5r,$$

откуда

$$q = 3+3r,$$

и, следовательно, при $q>3+3r$ программная реализация алгоритма **A2** будет работать более эффективно. Таким образом, если среда программной реализации алгоритмов (тип данных, язык программирования, обслуживающий его компилятор и компьютер) такова, что порождаемое этой средой время выполнения операции умножения, например, для действительных чисел, более чем втрое превышает время их сложения, то в предположении, что значение $r<<1$, (вполне реальное соотношение для языков программирования высокого уровня), более предпочтительным для реализации является выбор алгоритма **MultC_A2**. По поводу времен выполнения операций необходимо сделать следующее замечание — поскольку в зависимости от машинного алгоритма реализации операций умножения и сложения/вычитания и особенностей их конвейеризации время выполнения этих операций для различных операндов может изменяться, то для выбора рационального

алгоритма необходимо использовать экспериментально определенное среднее время выполнения операций.

Этот простой пример показывает, что принятие решения о рациональном выборе иногда требует применения «тонких» методов исследования — в данном случае — пооперационного анализа. Конечно, выигрыш во времени пренебрежимо мал, если мы умножаем только два комплексных числа, однако, если этот алгоритм является частью сложной вычислительной задачи с комплексными числами, требующей достаточно большого количества умножений, то выигрыш может быть ощутимым. Оценка такого сокращения на одно умножение комплексных чисел следует из только что проведенного анализа

$$\Delta t = (q - 3 - 3r)t_{(\pm)},\ q > 3 + 3r\ .$$

Очевидно, что метод пооперационного анализа является более трудоемким, однако его использование позволяет получить более детальные сведения об исследуемом алгоритме. Заметим, что метод опирается на предположение о том, что времена выполнения отдельных базовых операций известны. Поскольку общий способ экспериментального получения этих времен не учитывает реальные операционные потоки, которые порождаются исследуемыми алгоритмами, то такие особенности архитектуры процессора, как наличие стека и конвейера в этом подходе не могут быть полностью учтены.

10.3. Поиск максимального из трех чисел

Рассмотрим задачу поиска максимального значения из трех чисел, поступающих на вход алгоритма. Для этой задачи также существует несколько алгоритмов решения, различающихся ресурсной эффективностью. Выбор рационального алгоритма будем осуществлять по критерию минимума трудоемкости в среднем случае. Выбор этого критерия приводит к необходимости применения метода вероятностного анализа для исследования трудоемкости алгоритмов решения этой задачи.

Первое рассматриваемое алгоритмическое решение — использовать стандартный алгоритм поиска максимума в массиве и развернуть цикл сравнений. Запись алгоритма имеет вид:

```
MaxABC_A1(a, b, c; max)
   max ← a                                   1
   If b > max                                1
      then
         max ← b                          1*pb
   If c > max                                1
      then
         max ← c                          1*pc
Return (max)
End
```

Метод вероятностного анализа предполагает определение вероятностей p_b, p_c при условии, что известен закон распределения, которому подчиняются входные данные. При допущении о равно вероятности всех входов из D_A получаем, что вероятность того, что второе значение (b) больше первого (a) — p_b равна $1/2$, а вероятность того, что максимальным является третье число (c) — p_c равна $1/3$. Таким образом, с учетом трудоемкости в строках алгоритма, получаем трудоемкость алгоритма **MaxABC_A1** в среднем

$$\overline{f_{A1}}(D) = 1+1+1\cdot\frac{1}{2}+1+1\cdot\frac{1}{3} = 3\frac{5}{6}. \qquad (10.3.1)$$

Читатели без труда могут получить трудоемкость этого алгоритма для лучшего и худшего случаев, что позволит убедиться в том, что этот алгоритм принадлежит к типу L_c и классу PR, параметрическая зависимость трудоемкости определяется числовыми значениями и порядком расположения чисел на входе алгоритма. Нерациональные действия этого алгоритма для некоторых входов очевидны — если третье число максимально, то нет необходимости выполнять две лишних операции присваивания. Новая идея заключается в том, чтобы вначале на основе сравнений определить максимальное значение, а затем выполнить присваивание. Реализация этой идеи приводит к алгоритму **MaxABC_A2**, запись которого имеет вид:

```
MaxABC_A2(a, b, c; max)
   If a > b                                  1
      then
         If a > c                       1*pab
            then
               max ← a                1*pab*pac
            else
               max ← c                1*pab*pca
         end If
      else
         If b > c                       1*pba
```

```
            then
               max ← b                    1*p_ba*p_bc
            else
               max ← c                    1*p_ba*p_cb
         end If
   end If
Return (max)
End
```

В строках справа через p_{ab} обозначена вероятность того, что значение a больше, чем значение b, а через p_{ba} — обратная вероятность. Остальные обозначения введены по аналогии. Трудоемкость в среднем алгоритма **MaxABC_A2** может быть получена, если будут определены вероятности переходов на блоки **then** и **else** в соответствующих конструкциях ветвления.

Исходя из предположения о равно вероятности всех входов из D_A можно показать, что вероятность того, что число a больше, чем число b равна $1/2$. Результат аналогичен и для других возможных пар, таким образом

$$p_{ab} = p_{ba} = p_{ac} = p_{ca} = p_{bc} = p_{cb} = \frac{1}{2},$$

что позволяет вычислить методом вероятностного анализа трудоемкость алгоритма **MaxABC_A2** в среднем

$$\overline{f_{A2}}(D) = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{2} + \frac{1}{4} + \frac{1}{4} = 3. \qquad (10.3.2)$$

Более внимательный взгляд на алгоритм **MaxABC_A2** позволят получить более значимый результат, а именно: для любых входов из D_A этот алгоритм всегда выполняет три базовых операции. Таким образом, алгоритм принадлежит к типу L_c и классу C. Обратим внимание на интересный факт — два разных алгоритма решения одной задачи принадлежат к различным классам.

Рассматривая общий критерий ресурсной эффективности алгоритмов, попробуем понять, что есть «плата» за снижение трудоемкости в среднем? Алгоритм **MaxABC_A2** содержит три операции сравнения и четыре операции присваивания, а алгоритм **MaxABC_A1** — два сравнения и три присваивания, таким образом, более длинный текст записи алгоритма **MaxABC_A2** порождает более длинный программный код. Платой за улучшение трудоемкости является увеличение затрат памяти в области программного кода.

Может ли быть улучшен результат, задаваемый формулой (10.3.2)? Следующие рассуждения показывают, что мы получили алгоритм, оптимальный по трудоемкости. Поскольку максимальное значение должно быть помещено в отдельную ячейку памяти, то любой алгоритм будет содержать как минимум одну операцию присваивания, а для выбора одного из трех значений необходимо, как минимум, два бинарных сравнения, но алгоритм **`MaxABC_A2`** как раз и выполняет для любого входа два сравнения и одно присваивание.

С точки зрения требуемого объема дополнительной памяти оба алгоритма одинаковы

$$V_{A1}(D) = V_{A2}(D) = 0,$$

и относятся к классу $V0$.

Этот, пока еще «модельный», пример, тем не менее, показывает, что различные компоненты общего критерия ресурсной эффективности находятся в «обратно пропорциональной» зависимости — улучшение одного критерия оценки качества алгоритма, как правило, оплачивается некоторым ухудшением другого — такого рода ситуации будут нам встречаться достаточно часто.

10.4. Сортировка трех чисел по месту

Рассмотрим следующую, на первый взгляд так же простую задачу. Исходными данными являются три произвольных числа, реально — адреса этих чисел. Необходимо отсортировать эти числа по возрастанию и возвратить результат «по месту», т. е. результат должен быть расположен в тех же ячейках, что и исходные числа. Может быть предложено несколько различных алгоритмов решения с различными ресурсными характеристиками. Цель этого небольшого исследования — получить алгоритм с оптимальной трудоемкостью в среднем случае.

Рассмотрим вначале алгоритм, решающий данную задачу с помощью сравнений пар чисел и соответствующих перестановок. Фактически это реализация метода «пузырька» для трех чисел. Запись алгоритма **`SortABC_A1`**, реализующего этот метод, выглядит следующим образом:

```
SortABC_A1(a, b, c)
   If a>b                                   1
      then (обмен a и b)                    3
         t ← a
         a ← b
         b ← t
   If b>c                                   1
      then (обмен b и c)                    3
         t ← b
         b ← c
         c ← t
   If a>b                                   1
      then (обмен a и b)                    3
         t ← a
         a ← b
         b ← t
Return (a, b, c)
End
```

Для анализа трудоемкости этого алгоритма воспользуемся методом классов входных данных. Действительно, множество всех допустимых входов достаточно велико, и определяется всеми возможными тройками чисел, представляемых в выбранном формате данных. Даже для целых чисел без знака длиной два байта

$$|D_A| = \left(2^{16}\right)^3 = 2^{48},$$

но трудоемкость алгоритма зависит не от значений чисел, а от соотношений между ними в смысле арифметических сравнений. Введем условное обозначение «1» для минимального значения, «3» для максимального, и «2» для промежуточного. Очевидно, что все множество входов разбивается при этом на $3! = 6$ классов входных данных, и трудоемкость алгоритма для любого входа из каждого класса одинакова. Введем следующие обозначения классов входных данных (см. табл. 10.1), которые соответствуют шести различным соотношениям по арифметическому сравнению между тремя числами.

Таблица 10.1

	a	b	c
D_1	1	2	3
D_2	1	3	2
D_3	2	1	3
D_4	2	3	1
D_5	3	1	2
D_6	3	2	1

Трудоемкости для различных классов могут быть получены на основе анализа выполнения тех ветвей алгоритма, которые задействованы при обработке входов, соответствующих данному классу. В результате получаем:

$$f_{A1}(D_1)=3,\ f_{A1}(D_2)=6,\ f_{A1}(D_3)=6,$$
$$f_{A1}(D_4)=9,\ f_{A1}(D_5)=9,\ f_{A1}(D_6)=12, \qquad (10.4.1)$$

таким образом, лучшим случаем является любой вход из класса D_1, при этом $f^{\vee}_{A1}(D)=3$, а худшим — любой вход из класса D_6, для которого $f^{\wedge}_{A1}(D)=12$. Трудоемкость в среднем может быть получена, если задана вероятность появления входов, относящихся к различным классам. В предположении, что эти вероятности одинаковы, с учетом (10.4.1) получаем

$$\overline{f_{A1}}(D)=\frac{1}{6}\cdot(3+6+6+9+9+12)=7\frac{1}{2}. \qquad (10.4.2)$$

Полученные результаты позволяют сделать вывод о том, что алгоритм **`SortABC_A1`** относится к типу L_C и классу PR в этом типе, при этом параметрическая зависимость определяется взаимными соотношениями между числами по операции арифметического сравнения. Одна дополнительная ячейка, требуемая алгоритмом **`SortABC_A1`**, обуславливает его принадлежность к классу VC по дополнительной памяти.

Разработка оптимального по трудоемкости алгоритма должна базироваться на очевидном предположении, что вход из каждого класса обрабатывается за минимально возможное число операций. Таким образом, необходимо используя сравнения, определить принадлежность входа к одному из шести классов, и построить оптимальный фрагмент обмена ячеек, сортирующий данный вход.

Основная идея состоит в использовании кольцевого сдвига для входов из классов D_4 и D_5, что позволяет выполнить сортировку за 4 операции (без трудоемкости определения класса). Заметим, что такой подход приводит только к одному обмену содержимого ячеек для класса D_6, в котором алгоритм **`SortABC_A1`** выполнял три обмена. Запись алгоритма **`SortABC_A2`**, реализующего эти идеи, имеет вид

```
SortABC_A2(a, b, c)
   If a>b
      then
         If b>c
            then (класс «321» - обмен 3 и 1 местами f=2+3=5)
               t ← a
               a ← c
               c ← t
            else
               If a>c
                  then (класс «312» - кольцевой обмен f=3+4=7)
                     t ← a
                     a ← b
                     b ← c
                     c ← t
                  else (класс «213» - обмен 2 и 1 местами f=3+3=6)
                     t ← a
                     a ← b
                     b ← t
               end If
         end If
      else
         If b>c
            then
               If a>c
                  then (класс «231» - кольцевой обмен f=3+4=7)
                     t ← c
                     c ← b
                     b ← a
                     a ← t
                  else (класс «132» - обмен 3 и 2 f=3+3=6)
                     t ← b
                     b ← c
                     c ← t
               end If
            else (класс «123» - без обменов f=2+0=2)
         end If
   end If
Return (a, b, c)
End
```

В приведенных внутри текста алгоритма комментариях в записи типа `f=3+4=7` первое слагаемое есть трудоемкость идентификации класса входных данных, а второе — трудоемкость собственно сортировки. Трудоемкость алгоритма **SortABC_A2** для различных классов определяется на основании анализа текста алгоритма (см. комментарии), и составляет

$$f_{A2}(D_1)=2,\ f_{A2}(D_2)=6,\ f_{A2}(D_3)=6,$$
$$f_{A2}(D_4)=7,\ f_{A2}(D_5)=7,\ f_{A2}(D_6)=5, \qquad (10.4.3)$$

таким образом, лучшим случаем для алгоритма **SortABC_A2** является любой вход из класса D_1, при этом $f^{\vee}_{A2}(D)=2$, а худшим — любые входы из классов D_4 и D_5, для которых $f^{\wedge}_{A2}(D)=7$. Трудоемкость в среднем, в предположении о равной вероятности входов из любого класса, рассчитывается на основе (10.4.3), и составляет

$$\overline{f_{A2}}(D)=\frac{1}{6}\cdot(2+6+6+7+7+5)=5\frac{1}{2}. \qquad (10.4.4)$$

Полученный результат не означает, что при каждом запуске данного алгоритма экспериментально определенная трудоемкость будет определяться формулой (10.4.4). Боле того, эта трудоемкость вообще никогда не будет равна указанному значению, поскольку функция трудоемкости для конкретного входа есть целое положительное число. Формула (10.4.4) есть теоретически полученное математическое ожидание функции трудоемкости этого алгоритма, как дискретной ограниченной случайной величины. Этот результат был получен методом вероятностного анализа классов входных данных, и при его экспериментальном подтверждении на основе выборочных средних значений трудоемкости не надо забывать, что он получен в предположении о равной вероятности появления входов, принадлежащих любому из шести исследованных классов.

Полученные результаты позволяют отнести алгоритм **SortABC_A2** к типу L_C и классу PR для этого типа, и так же, как и алгоритм **SortABC_A1** он относится к классу VC по дополнительной памяти. Кроме того отметим, что алгоритм **SortABC_A2** обладает меньшим размахом варьирования значений трудоемкости.

С точки зрения теории ресурсной эффективности возникает резонный вопрос — за счет чего в алгоритме **SortABC_A2** удалось снизить трудоемкость в среднем? Внимательный читатель наверняка уже знает ответ — более длинный текст записи **SortABC_A2** порождает более длинный программный код. Таким образом, и в этом случае, как и в предыдущем примере, платой за снижение трудоемкости является увеличение затрат памяти в области программного кода.

Еще один вопрос, оставшийся пока без ответа — это доказательство оптимальности алгоритма **SortABC_A2** по трудоемкости. Может ли быть предложен иной алгоритм, имеющий трудоемкость в среднем, лучшую, чем $5\frac{1}{2}$ базовых опе-

раций? Обоснуем отрицательный ответ на этот вопрос, при условии, что принятая модель вычислений и введенные базовые операции остаются неизменными. Можно рассуждать следующим образом — определение принадлежности входа к одному из шести классов невозможно сделать за два сравнения, это следует из рассмотрения бинарного дерева сравнений. Полное бинарное дерево глубиной три содержит восемь листьев, а дерево глубиной два — только четыре, поэтому бинарное дерево с шестью листьями имеет глубину три, более корректно

$$h_{\min} = \lceil \log_2 6 \rceil = 3.$$

Наличие только шести листьев позволяет не строить полное бинарное дерево глубиной три, однако при этом не более чем два класса могут быть идентифицированы только за два сравнения. Но именно такое бинарное дерево сравнений и реализует алгоритм **`SortABC_A2`**. В принятой модели вычислений обмен содержимого ячеек требует минимум трех операций, а кольцевой сдвиг содержимого трех ячеек не может быть реализован быстрее, чем за четыре присваивания. Таким образом, алгоритм выполняет минимальное число базовых операций для любого входа из каждого класса входных данных, и, следовательно, он оптимален по трудоемкости для любого входа. Сказанное относится к общему случаю задачи сортировки трех чисел — т. е. к случаю, когда эти числа различны. Частные случаи, когда какие-либо два или все три числа равны между собой, предоставляются читателям для самостоятельного рассмотрения.

Отметим, что в чистом виде эта задача, конечно, не представляет интереса для программистов, но она может быть использована как составная часть при построении ресурсно-эффективных комбинированных алгоритмических решений. Например, алгоритм **`SortABC_A2`** совместно с алгоритмом сортировки двух чисел может быть использован как компонент комбинированного алгоритма сортировки для останова рекурсии в модифицированном алгоритме сортировки слиянием.

10.5. Возведение числа в целую степень

Еще одна задача, принадлежащая типу L_c, алгоритмы решения которой имеют фиксированную длину входа, — задача о возведении числа в целую степень, т. е. вычисление значения $y = x^m$ для целого неотрицательного значения m.

Рассмотрим примитивное решение этой задачи, которое использует последовательное умножение. Запись алгоритма **Pow_A1** выглядит следующим образом

```
Pow_A1 (x, m; y)
   y ← 1                                    1
   For i ← 1 to m                         1+3m
      y ← y*x                              2m
Return (y)
End.
```

Поскольку операция умножения является базовой, то значение x не влияет на трудоемкость (хотя, вполне возможно, влияет на время выполнения программной реализации). На основании построчного анализа записи алгоритма элементарно определяется его трудоемкость

$$f_{A1}(D) = f_{A1}(x,m) = f_{A1}(m) = 5m + 2, \qquad (10.5.1)$$

и принадлежность к классу PR в типе L_c. Если рассматривать число значащих бит — n в двоичном представлении числа m, то очевидно, что

$$f_{A1}(m) = f_{A1}(n) = \Theta\left(2^n\right),$$

и, следовательно, алгоритм **Pow_A1** принадлежит подклассу $PR\exp$, т.е. параметрическая зависимость функции трудоемкости является экспоненциальной по числу бит параметра. Одна дополнительная ячейка, требуемая счетчиком цикла, обуславливает принадлежность алгоритма **Pow_A1** к классу VC по дополнительной памяти.

Вполне возможно, что для небольших значений m алгоритм **Pow_A1** вполне применим, однако в области алгоритмического обеспечения ряда криптосистем, базирующихся на результатах теории чисел, например RSA [10.1] требуются алгоритмы, решающие эту задачу за приемлемое время для очень больших значений показателя. Так, например, в криптосистеме RSA рекомендуемое значение $m \approx 2^{768}$ [10.1]. В этом случае используется быстрый алгоритм возведения в степень методом последовательного возведения в квадрат [10.2], детальный анализ которого представляет определенный интерес. Идея состоит в рассмотрении двоичного представления числа m и вычисления четных степеней x путем повторного возведения в квадрат [10.2]. Пусть, например, $m = 11$, тогда вычисления идут по схеме

$$x^{11} = x \cdot x^5 \cdot x^5,\ x^5 = x \cdot x^2 \cdot x^2, x^2 = x \cdot x, x = x \cdot 1,$$

а рекуррентное соотношение, определяющее степень m числа x имеет вид

$$x^m = \begin{cases} 1, & m = 0; \\ \left(x^{m/2}\right)^2, & m = 2k; \\ x \cdot \left(x^{(m-1)/2}\right)^2, & m = 2k+1. \end{cases} \qquad (10.5.2)$$

На основе (10.5.2) можно записать рекуррентное соотношение, определяющее рекурсивно заданную функцию $f(x,m) = x^m$:

$$f(x,m) = \begin{cases} 1, & m = 0; \\ (f(x,m/2))^2, & m = 2k; \\ x \cdot (f(x,(m-1)/2))^2, & m = 2k+1. \end{cases} \qquad (10.5.3)$$

Соотношение (10.5.3) позволяет разработать рекурсивный алгоритм в виде процедурно реализованной рекурсивной функции **Pow_A2(x,m)** (справа, как обычно, указано количество базовых операций в строке). Поскольку для нечетных чисел стандартная операция целочисленного деления отбрасывает остаток, т. е.

$$m \text{ div } 2 = (m-1) \text{ div } 2 = k, m = 2k+1, k \geq 0,$$

то можно избежать вычитания единицы из m для нечетных чисел.

```
Pow_A2(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  (m – четно)
               z ← Pow_A2(x, m div 2)          2
               Pow_A2 ← z*z                    2
               (квадрат результата)
            else  (m – нечетно)
               z ← Pow_A2(x, m div 2)          2
               Pow_A2 ← x*z*z                  3
               (умноженный на x квадрат результата)
         end if
   end if
Return (Pow_A2)
End.
```

Данный рекурсивный алгоритм также относится к классу PR. Определим его принадлежность к подклассу класса PR. В целях анализа этого алгоритма будем использовать специальные функции, — $\beta(m)$, значением которой является общее количество разрядов в двоичном представлении числа m, и функции $\beta_1(m)$

и $\beta_0(m)$, задающие количество единиц и нулей в этом представлении [10.3]. Рассмотрим действительную длину входа алгоритма — количество значащих бит n, в двоичном представлении числа m, $n = \beta(m)$. Поскольку каждый рекурсивный вызов при делении на два уменьшает на единицу количество бит степени, то трудоемкость алгоритма определяется количеством бит на входе

$$f_{A2}(n) = \Theta(n) = \Theta(\lfloor \log_2 m \rfloor),$$

и, следовательно, алгоритм принадлежит подклассу $PR\ pl$.

Трудоемкость этого алгоритма может быть получена методом подсчета вершин рекурсивного дерева. Метод изложен в главе 9, ряд примеров применения метода читатель может найти в [10.3]. Дерево рекурсии, порождаемое данным алгоритмом, представляет собой унарное дерево — цепочку с одним листом, и количеством внутренних вершин равным $\beta(m)$. Таким образом, вызов данного алгоритма со значением m порождает дерево рекурсии со следующими характеристиками

$$R(m) = \beta(m) + 1 = \lfloor \log_2 m \rfloor + 2, \quad R_V(m) = \beta(m) = \beta_1(m) + \beta_0(m),$$
$$R_L(m) = 1, \quad H_R(m) = \beta(m) + 1,. \qquad (10.5.4)$$

где $R(m)$ — общее число вершин, $R_V(m)$ — число внутренних вершин, $R_L(m)$ — число листьев порожденного дерева рекурсии, а $H_R(m)$ — его глубина. 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

$$f_R(1) = 2 \cdot (2 + 4 + 1 + 1 + 1) = 18,$$

и, следовательно, с учетом (10.5.4) затраты на обслуживание рекурсии составляют

$$f_R(m) = R(m) \cdot f_R(1) = (\beta(m) + 1) \cdot 18.$$

Трудоемкость останова рекурсии включает в себя одно сравнение и одно присваивание, таким образом $f_{CL}(1) = 2$, и трудоемкость в листьях равна

$$f_{CL}(m) = R_L(m) \cdot f_{CL}(1) = 1 \cdot 2 = 2.$$

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, трудоемкость во внутренних вершинах равна

$$f_{CV}(m)=8\cdot\beta_1(m)+7\cdot\beta_0(m).$$

Объединяя все компоненты, получаем

$$f_{A2}(m)=18\cdot\beta(m)+8\cdot\beta_1(m)+7\cdot\beta_0(m)+20. \qquad (10.5.5)$$

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 $\beta(m)=const$. Обозначим $\beta(m)=n$, и рассмотрим значения m, которые доставляют минимальное и максимальное значение функции $\beta_1(m)$ при $\beta(m)=n$. Точкам экстремума функции $\beta_1(m)$ соответствуют значения 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), $\beta_1(m)=1$, и

$$f_{A2}^{\vee}(m:\beta(m)=n)=18\cdot n+8\cdot 1+7\cdot(n-1)+20=25\cdot n+21, \qquad (10.5.6)$$

а в случае если $m=2^n-1$ (все биты числа m равны единице), $\beta_1(n)=n$, тогда

$$f_{A2}^{\wedge}(m:\beta(m)=n)=18\cdot n+8\cdot n+20=26\cdot n+20. \qquad (10.5.7)$$

Заметим, что для данного алгоритма функция трудоемкости не является монотонно возрастающей. Рассмотрим $m=2^n-1$, тогда $m+1=2^n$, в этом случае

$$\beta(m)=n,\ \beta(m+1)=n+1,\ \beta_1(m)=n,\ \beta_1(m+1)=1,$$

и, следовательно

$$f_{A2}(m)=26\cdot n+20,\quad f_{A2}(m+1)=25\cdot(m+1)+21=25\cdot n+46,$$

что влечет $f_{A2}(m)>f_{A2}(m+1)$ при $m\geq 2^{27}$.

Если показатель степени заранее не известен, то трудоемкость в среднем, в предположении, что представление числа m занимает n значащих двоичных разрядов, задается формулой [10.3]

$$\overline{f}_{A2}(m:\beta(m)=n)=25{,}5\cdot n+20{,}5. \qquad (10.5.8)$$

Таким образом, трудоемкость быстрого алгоритма возведения в степень линейно зависит от количества бит в двоичном представлении показателя степени.

Для получения оценки емкостной эффективности используем функцию $H_R(m)$. Данный алгоритм использует одну локальную ячейку для хранения значения z, и, в соответствии с методикой, описанной в [10.3], наибольший объем памяти в области стека составит

$$V_{st}(m) = H_R(m) \cdot (2+4+1+1+1) = 9 \cdot H_R(m),$$

подставляя $H_R(m)$ из (10.5.4) получаем емкостную эффективность алгоритма

$$V_{A2}(m) = V_{st}(m) = 9 \cdot H(m) = 9 \cdot \lfloor \log_2 m \rfloor + 18, \qquad (10.5.9)$$

что позволяет, определить ресурсную сложность алгоритма в худшем случае

$$\mathfrak{R}_c^{\wedge}(m) = \langle \Theta(\log_2 m), \Theta(\log_2 m) \rangle.$$

Понятно, что доля операций обслуживания рекурсии в алгоритме **Pow_A2** значительна, и, следовательно, коэффициент при главном порядке в функции трудоемкости может быть улучшен при переходе от рекурсивной реализации к процедурной. При этом повторным возведением в квадрат вычисляются значения степеней числа x, соответствующие двоичному представлению показателя. Пусть, например, $n = 11$, тогда $x^{11} = x^8 \cdot x^2 \cdot x^1$, причем вычисления идут по схеме: $x^2 = x \cdot x, x^4 = x^2 \cdot x^2, x^8 = x^4 \cdot x^4$. Алгоритмическая реализация этой идеи требует последовательного выделения битов числа m, возведения значения x в квадрат и умножения y на те двоичные степени x, для которых в двоичном разложении m присутствует единица. Мы получаем алгоритм **Pow_A3**, запись которого имеет вид

```
Pow_A3(x, m; y)
   z ← x                                          1
   y ← 1                                          1
   Repeat
      If (m mod 2) = 1                          2*β(m)
         then («1» в текущем разряде)
            y ← y*z                            2*β1(m)
               (умножение на текущую степень x)
      end If
      z ← z*z                                   2*β(m)
      (повторное возведение в квадрат)
      m ← m div 2                               2*β(m)
      (переход к следующему разряду)
   Until m = 0                                  1*β(m)
Return (y)
End.
```

Получим точную функцию трудоемкости данного алгоритма, используя введенные выше обозначения для специальных функций:

$$f_{A3}(n) = 2 + \beta(m) \cdot (2 + 2 + 2 + 1) + \beta_1(m) \cdot (2) = 7 \cdot \beta(m) + 2 \cdot \beta_1(m) + 2 . \quad (10.5.10)$$

Количество проходов цикла определяется количеством битов в двоичном представлении числа m, которое определяется функцией $\beta(m)$, а количество повторений операции формирования результата (`y` $\leftarrow$ `y*z`) — количеством единиц в этом представлении, которое определяется функцией $\beta_1(m)$. Аналогично анализу рекурсивного алгоритма в случае если $m = 2^k$, то

$$\beta_1(m) = 1 \text{ и } f_{A3}(m) = 7 \cdot \beta(m) + 4 ,$$

а если $m = 2^k - 1$, то

$$\beta_1(m) = \beta(m) \text{ и } f_{A3}(m) = 9 \cdot \beta(m) + 2 .$$

Формула для средней трудоемкости получается с использованием результата о среднем числе единиц в m битовых числах [10.3]

$$\overline{f}_{A3}(m) = 7 \cdot \beta(m) + 2 \cdot \beta_s(m) + 2 = 8 \cdot \log_2(m) + 2. \quad (10.5.11)$$

Поскольку алгоритм **`Pow_A3`** требует одну дополнительную ячейку памяти, то его ресурсная сложность имеет вид

$$\overline{\Re}_c(m) = \langle \Theta(\log_2(m)), \Theta(1) \rangle .$$

Сравнительный анализ трех исследованных алгоритмов по функции трудоемкости показывает, что **`Pow_A3`** является наилучшим для всех показателей степени, а при выборе между **`Pow_A1`** и **`Pow_A2`** последний предпочтительнее при значениях $m \geq 29$.

10.6 Извлечение квадратного корня

Для вычисления квадратного корня с любой точностью можно воспользоваться известным в математике рекуррентным соотношением, определяющим бесконечную рекурсивную последовательность с начальным значением $x_1 = 1$, пределом которой при $n \to \infty$ является значение $\sqrt{a}$. Это соотношение имеет вид [10.3]

$$x_{n+1} = \frac{1}{2} \cdot \left(x_n + \frac{a}{x_n} \right). \qquad (10.6.1)$$

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 $\varepsilon = 10^{-20}$ является вполне достаточным.

Будем, для определенности рассматривать ситуацию, когда значение $a \geq 1$, тогда если проанализировать разности между предыдущим и следующим членами последовательности, определяемой формулой (10.6.1), то все они, кроме первой будут положительны. Это позволяет при записи алгоритма не вычислять значение абсолютной величины разности, если цикл по точности будет начинаться с вычисления третьего элемента последовательности. Таким образом, алгоритм **Sqrt_A1**, непосредственно реализующий вычисление квадратного корня по формуле (10.6.1) имеет вид

```
Sqrt_A1 (a, eps)
   x1 ← 1.0                                  1
   x2 ← 0.5*(x1+a/x1)                        4
   Repeat
      x1 ← x2                                1
      x2 ← 0.5*(x1+a/x1)                     4
   Until (x1-x2) <= eps                      2
Return (x2)
End.
```

Для получения функции трудоёмкости этого, очевидно параметричечски-зависимого алгоритма, необходимо оценить число членов последовательности (10.6.1), 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. $n = n(a, \varepsilon)$.

Фиксируя значение $\varepsilon = 10^{-20}$, попробуем получить аппроксимацию этой зависимости на основе экспериментальных данных. Полученные значения приведены в таблице 10.2.

Таблица 10.2

a	$n = n\left(a,10^{-20}\right)$
1,00E+02	8
1,00E+03	10
1,00E+04	11
1,00E+05	13
1,00E+06	15
1,00E+07	16
1,00E+08	18
1,00E+09	20

Анализ этих результатов позволяет предположить логарифмическую зависимость вида

$$n\left(a,10^{-20}\right) = n'(a) = b\log_2 a + c,$$

которая хорошо подтверждается трендом со значениями $b = 0{,}5002$ и $c = 4{,}7123$, таким образом можно приближенно считать, что

$$n'(a) \approx \left\lceil \log_2\left(26{,}231\sqrt{a}\right)\right\rceil,$$

и эта зависимость подтверждается последующими экспериментами при дальнейшем увеличении значений a. Так, например, для $a = 10^{50}$ экспериментальное значение номера последовательности равно 88, а $\log_2\left(26{,}231\sqrt{10^{50}}\right) \approx 87{,}7614$.

Полученная аппроксимация позволяет получить функцию трудоемкости алгоритма **`Sqrt_A1`**, напомним, что второй член последовательности вычисляется вне цикла по точности

$$f_{A1}(a) = 5 + 7(n(a) - 1) \approx 3{,}5\lceil \log_2 a \rceil + 31. \qquad (10.6.2)$$

На основе (10.6.2) можно сделать вывод о том, что этот алгоритм принадлежит к типу L_c и классу PR, параметрическая зависимость трудоемкости определяется значением числа a, более точно — числом бит двоичного представления числа a. Это позволяет отнести данный алгоритм к подклассу $PR\ pl$ в классе PR.

Разработка алгоритма, имеющего лучшую временную эффективность связана с такой организацией процесса вычислений, что бы зависимость $n(a,\varepsilon)$ была бы 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. В этом случае возможно представление функции $n(a,\varepsilon)$ в виде

$$n(a,\varepsilon) = h(a) + g(\varepsilon). \tag{10.6.3}$$

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 $m(a)$ операций деления, а на второй фазе получаем значение квадратного корня из a', вычисляя $g(\varepsilon)$ значений последовательности (10.6.1). После этого остается только скорректировать полученный результат с учетом предшествовавших операций деления.

Рассмотрим вначале вторую фазу. Одной из конструктивных идей является начальная аппроксимация значения квадратного корня на некотором сегменте с помощью наилучшего равномерного приближения по Чебышеву [10.4]. Поскольку коррекция полученного результата — фактически третья фаза алгоритма производится умножением, то, учитывая возможные потери точности, мы должны производить умножение на число, равное степени двойки, т. е. корректировать только порядок числа. Эти рассуждения приводят к выбору величины a', равной, например, 64, и корректирующему умножению на 8. Таким образом, возникает задача построения линейной аппроксимации значения квадратного корня на сегменте $[1, 64]$ с использованием метода наилучшего равномерного приближения. Введем следующие обозначения

$$f(x) = \sqrt{x}, \quad y = a_0 + a_1 x, \quad a = 1, \quad b = 64,$$

тогда в рамках данной постановки теорема Чебышева [10.4] устанавливает, что

$$\begin{cases} \sqrt{a} - (a_0 + a_1 a) = \alpha L \\ \sqrt{d} - (a_0 + a_1 d) = -\alpha L \\ \sqrt{b} - (a_0 + a_1 b) = \alpha L \end{cases} \tag{10.6.4}$$

где точки a, b, d есть точки чебышевского альтернанса, а значение d определяется из условия $f'(d) - a_1 = 0$. Неизвестные коэффициенты a_0, a_1 могут быть получены следующим образом — вычитая третье уравнение в (10.6.4.) из первого определяем значение a_1:

$$a_1 = \frac{8-1}{64-1} = \frac{1}{9}, \tag{10.6.5}$$

из условия $f'(d) - a_1 = 0$ получаем значение $d = 20{,}25$, а, складывая первое и второе уравнение системы (10.6.4), имеем

$$1 + 4\frac{1}{2} = 2a_0 + \frac{1}{9}\left(1 + 20\frac{1}{4}\right),$$

откуда получаем значение a_0

$$a_0 = \frac{113}{72}.$$

Теперь можно вычислить значение L — максимальное расхождение построенной аппроксимации и функции квадратного корня в точках чебышевского альтернанса, а именно $L = 0{,}6805(5)$. Непосредственные вычисления показывают, что для данного максимального расхождения и достижения точности $\varepsilon = 10^{-20}$ достаточно всего пяти итераций по формуле (10.6.1) с начальным приближением

$$x_1 = \frac{1}{9}a' + \frac{113}{72}, \tag{10.6.6}$$

где значение $a' : 1 \le a' \le 64$. Значение a' представляет собой результат, полученный последовательными операциями деления на первой фазе. Теперь, когда сегмент для a' определен, становится ясно, что делителем в первой фазе является число 64. Определим количество шагов $m(a)$, необходимых для получения $a' : 1 \le a' \le 64$ по заданному значению a.

$$m(a) = \lceil \log_{64} a \rceil - 1 = \left\lceil \frac{\log_2 a}{6} \right\rceil - 1. \tag{10.6.7}$$

Таким образом, мы имеем математическое обоснование для разработки двухфазного эффективного алгоритма вычисления квадратного корня — на первой фазе исходный аргумент последовательным делением приводится к сегменту [1,64], затем по формуле (10.6.6) вычисляется начальное приближение, после чего за пять итераций вычисляется значение квадратного корня с точностью $\varepsilon = 10^{-20}$, которое затем корректируется путем умножения на $8m(a)$. Запись алгоритма, назовем его **`Sqrt_A2`**, имеет вид

```
Sqrt_A2 (a)
   m ← 1                                  1
   While (a => 64)                        1
      a ← a/64                            2
      m ← m*8                             2
   end While
   x ← (1/9)*a+(113/72)                   5
   For j ← 1 to 5
      x ← 0.5*(x+a/x)                     4
   end For
   x ← x*m                                2
Return (x)
End.
```

На основе указанного числа базовых операций в строках алгоритма можно получить функцию трудоемкости алгоритма **Sqrt_A2**

$$f_{A2}(a) = 1 + 5m(a) + 1 + 5 + 1 + 5(3+4) + 2 = 5m(a) + 45,$$

откуда, с учетом формулы (10.6.7), получаем

$$f_{A2}(a) = 5\left\lceil \frac{\log_2 a}{6} \right\rceil + 40. \qquad (10.6.8)$$

Очевидно, что алгоритм **Sqrt_A2** обладает лучшей временной эффективностью, например, для значения $a = 10^{28}$ он совершает в три раза меньше базовых операций. При значениях $1 \le a \le 10$ трудоёмкости двух алгоритмов совпадают, что позволяет говорить об эффективности применения алгоритма **Sqrt_A2** на всем реальном сегменте входов. Так же, как и алгоритм **Sqrt_A1**, алгоритм **Sqrt_A2** принадлежит к типу L_c, классу PR, и к подклассу $PR\ pl$ в данном классе. Поскольку оба алгоритма требуют фиксированного числа дополнительных ячеек памяти, то их принадлежность к классу VC не должна вызывать сомнений у читателей.

Задачи и упражнения к главе 10

10.1. Определите, каковы реальные времена выполнения программных реализаций двух алгоритмов умножения комплексных чисел, обсуждаемых в параграфе 10.2, используя, например, возможность обращения к тактовому счетчику.

10.2. Алгоритм с лучшей трудоемкостью в теории не всегда обязательно обладает таким же свойством в программной реализации. Определите, действительно ли более эффективный в теории алгоритм поиска максимума из трех чисел,

приведенный в параграфе 10.3, обладает лучшим временем выполнения на Вашем компьютере?

10.3. Предположим, что в модели вычислений операция обмена содержимого двух ячеек является базовой. Модифицируйте, в рамках этого предположения, тексты алгоритмов сортировки трех чисел (параграф 10.4) и получите функции их трудоемкости. Какой из двух алгоритмов оказался при этом в теории лучше? Объясните полученный результат.

10.4. Пусть модель вычислений содержит операции для работы с битами (сравнение, сдвиг и т. д.). Как можно улучшить на этой основе быстрый алгоритм возведения в степень из параграфа 10.5?

10.5. При разработке эффективного алгоритма вычисления квадратного корня можно делить исходное число на 256, восстанавливая результат умножением на 16. Постройте функцию наилучшего равномерного приближения по Чебышеву для сегмента [1,256]. Определите необходимое число итераций для достижения точности 10^{-20} в точках чебышевского альтернанса и получите функцию трудоемкости полученного нового алгоритма. Определите сегменты размерности с предпочтительным выбором старого или нового алгоритма на основе сравнения функций их трудоемкости.

Список литературы к главе 10

[10.1] Молдовян А. А. и др. Криптография: скоростные шифры. — СПб.: БХВ-Петербург, 2002. — 496 с.

[10.2] Ноден П., Китте К. Алгебраическая алгоритмика: Пер. с франц. — М.: Мир, 1999. — 720 с.

[10.3] Головешкин В. А., Ульянов М. В. Теория рекурсии для программистов. — М.: Издательство «Наука ФИЗМАТЛИТ», 2006. — 296 с.

[10.4] Численные методы / Н. С. Бахвалов, Н. П. Жидков, Г. М. Кобельков. — 4-ое изд. — М.: БИНОМ. Лаборатория знаний, 2006. — 636 с.

ГЛАВА 11.

РЕСУРСНО-ЭФФЕКТИВНЫЕ АЛГОРИТМИЧЕСКИЕ РЕШЕНИЯ ДЛЯ НЕКОТОРЫХ ЗАДАЧ С ПЕРЕМЕННОЙ ДЛИНОЙ ВХОДА

Введение

Примеры алгоритмических решений этой главы призваны продемонстрировать как некоторые методы разработки и анализа компьютерных алгоритмов, так и их применение для обоснования рационального выбора. Еще один важный момент, на который автор хотел бы обратить внимание уважаемых читателей — это учет особенностей задач и дополнительных требований технического задания при выборе рациональных алгоритмов. Очевидно, что предположение о равновероятности всех допустимых входов алгоритма при эксплуатации программного средства не всегда является обоснованным. Если мы знаем, что, как правило, на вход алгоритма поступают данные, обладающие определенными и устойчивыми особенностями, то на этой основе могут быть сформулированы рекомендации о предпочтительном использовании того или иного алгоритма. Такие требования технического задания как временная устойчивость или ограничение на максимальное время выполнения программной реализации могут также существенно влиять на решение задачи рационального выбора — соответствующие примеры читатель найдет в материале данной главы.

11.1. Поиск минимума и максимума в массиве

Наша задача — найти минимальный и максимальный элемент в массиве чисел, речь идет как о значениях соответствующих элементах массива, так и об их индексах. Эта задача достаточно часто встречается как вспомогательная в целом ряде задач статистической обработки данных, сортировки, при работе с матрицами и т. д. Вначале рассмотрим задачу поиска только максимального элемента и его индекса в массиве. Запись стандартного алгоритма, решающего эту задачу, в

принятом алгоритмическом базисе имеет вид (справа в строке, как и в главе 10, приведено задаваемое этой строкой число базовых операций):

```
Max_A0 (S, n; Max, iMax)
   Max ← S[1]                                  2
   iMax=1                                      1
   For i ← 2 to n                          1+3(n-1)
      If Max < S[i]                          2(n-1)
         then
            Max ← S[i]                         2m
            iMax=i                             1m
   end For
Return (Max, iMax)
End.
```

Входом данного алгоритма является массив из n элементов — соответствующая задача имеет тип L_n, а трудоемкость алгоритма зависит как от количества чисел в массиве, так и от порядка их расположения — алгоритм является количественно-параметрическим (класс NPR, подкласс $NPRS$). Полный анализ для фиксированной размерности множества исходных данных предполагает получение функций трудоемкости для худшего, лучшего и среднего случаев. Для данного алгоритма фрагментом, определяющим параметрическую зависимость трудоемкости, является блок **then**, содержащий строки `Max ← S[i]` и `iMax=i`. Обозначим общее количество выполнений этого блока, задаваемое алгоритмом **Max_A0**, через m, заметим, что сам блок содержит три операции.

Худший случай. Значение m будет максимально, а именно $m = n-1$, когда на каждом проходе цикла выполняется переприсваивание текущего максимума — это ситуация когда элементы исходного массива различны и отсортированы по возрастанию. В этом случае трудоемкость алгоритма равна:

$$f_A^{\wedge}(n) = 2+1+1+(n-1)(3+2)+(n-1)(3) = 8n-4 = \Theta(n). \qquad (11.1.1)$$

Лучший случай. Значение m будет минимально, и равно нулю ($m = 0$), в том случае, если первый элемент массива является максимальным. Трудоемкость алгоритма в этом случае равна:

$$f_A^{\vee}(n) = 2+1+1+(n-1)(3+2) = 5n-1 = \Theta(n) \qquad (11.1.2)$$

Трудоемкость в среднем. Для анализа трудоемкости этого алгоритма в среднем случае необходимо обосновать значение вероятности, с которой происходит переприсваивание максимума в строке `Max ← S[i]`. Это обоснование мо-

жет опираться на следующие рассуждения, основанные на равновероятном распределении положения текущего максимума. Алгоритм поиска максимума последовательно перебирает элементы массива, сравнивая текущий элемент массива с текущим значением максимума. На очередном шаге, когда просматривается k-ый элемент массива, переприсваивание максимума произойдет, если в подмассиве из первых k элементов максимальным элементом является последний. В случае если элементы исходного массива подчиняются равномерному распределению, вероятность того, что максимальный из k элементов расположен в определенной, например, последней позиции, равна $1/k$. Обозначим среднее значение m через $\overline{m}$. Тогда в массиве, содержащем n элементов, значение $\overline{m}$, а именно оно нас и интересует, определяется формулой:

$$\overline{m} = \sum_{k=2}^{n} \frac{1}{k} = H_n - 1 \approx \ln n + \gamma - 1, \gamma \approx 0{,}57,$$

где H_n — n-ое гармоническое число, а γ — постоянная Эйлера [11.1]. Суммирование начинается с двойки, поскольку первое присваивание максимума уже сделано до входа в цикл. Асимптотическое поведение H_n при $n \to \infty$ имеет вид [11.1]

$$H_n = \ln n + \gamma - \frac{1}{2n} - \frac{1}{12n^2} + O\left(n^{-3}\right).$$

Таким образом, если исходный массив состоит из различных чисел, а генеральная совокупность представляет собой $n!$ различных перестановок чисел массива, при условии, что появление каждой такой перестановки на входе алгоритма равновероятно, то значение $\overline{m}$ — математического ожидания количества операций присваивания максимума в основном цикле алгоритма для массива из n элементов, равно $H_n - 1$ (более подробный анализ, основанный на аппарате производящих функций см. в [11.2]), а включая первое присваивание — H_n, тогда:

$$\overline{f_A}(n) = 1 + (n-1)(3+2) + 3(\ln(n) + \gamma) = 5n + 3\ln(n) - 4 + 3\gamma = \Theta(n). \quad (11.1.3)$$

Заметим, что для данного алгоритма значение средней трудоемкости значительно ближе к лучшему случаю, т. е. смещено влево относительно середины границ колебаний трудоемкости при фиксированной размерности задачи.

Переходя к задаче поиска минимального и максимального элементов и их индексов, можно предложить ее решение в виде простой модификации алгоритма **Max_A0**. Запись этого алгоритма имеет вид:

```
MinMax_A1 (S, n; Max, Min, iMax, iMin)
   Max ← S[1]                                  2
   iMax=1                                      1
   Min ← S[1]                                  2
   iMin=1                                      1
   For i ← 2 to n                          1+3(n-1)
      If Max < S[i]                          2(n-1)
         then
            Max ← S[i]                          2m
            iMax=i                              1m
      If Min > S[i]                          2(n-1)
         then
            Min ← S[i]                          2m
            iMin=i                              1m
   end For
Return (Max, Min, iMax, iMin)
End.
```

Трудоемкость этого алгоритма для худшего и лучшего случаев читатели могут легко получить самостоятельно, а трудоемкость в среднем определяется на основе уже известных рассуждений о вероятности появления текущего максимума в текущем элементе массива, которые, очевидно, справедливы и для текущего минимума. Таким образом,

$$\begin{aligned}\overline{f_{A1}}(n) &= 1+(n-1)(3+2+2)+3(\ln(n)+\gamma)+3(\ln(n)+\gamma)= \\ &= 7n+6\ln(n)-6+6\gamma=\Theta(n).\end{aligned} \qquad (11.1.4)$$

Заметим, что в этом алгоритме выполняется два сравнения для каждого элемента массива, таким образом, два элемента обрабатываются за четыре сравнения.

Можно ли разработать алгоритм, обладающий меньшей трудоемкостью? Более корректно мы должны ответить на два вопроса:

1. Существует ли алгоритм, обладающий лучшей, чем $\Theta(n)$, асимптотической оценкой трудоемкости, или алгоритм **MinMax_A1** обладает оценкой, совпадающей с теоретическим нижним пределом?

2. Существует ли алгоритм, функция трудоемкости которого имеет меньший коэффициент у компонента главного порядка, т. е. возможно ли уменьшить коэффициент 7 в формуле (11.1.4)?

Для данной задачи ответ на первый вопрос отрицательный — улучшения в смысле асимптотической оценки невозможны, т. к. для определения максимального или минимального элемента необходимо просмотреть все элементы массива, и, следовательно, оценка $\Theta(n)$ есть теоретический нижний предел трудоемкости для данной задачи. Но коэффициент в формуле (11.1.4) оказывается можно улучшить, при этом, очевидно, что речь идет об улучшениях в рамках определения трудоемкости в базовых операциях принятой модели вычислений.

Следующий алгоритм решает задачу нахождения минимума и максимума, затрачивая всего три сравнения на два элемента массива. Идея состоит в том, что вначале два последовательных элемента массива сравниваются между собой, а затем меньший из них проверяется на минимум, а больший — на максимум. Небольшие трудности, возникающие из-за четного или нечетного числа элементов, могут быть достаточно просто преодолены — для массива четной длины цикл необходимо начинать с единицы, а для нечётной длины — с двойки. Запись алгоритма, реализующего эту идею, имеет вид:

```
MinMax_A2 (S, n; Max, Min, iMax, iMin)
   Max ← S[1]                                   2
   iMax=1                                       1
   Min ← Max                                    1
   iMin=1                                       1
   i=1+(n mod 2)                                2
   Repeat
      si ← S[i]                              2(n/2)
      sj ← S[i+1]                            3(n/2)
      If Si < Sj                             1(n/2)
         then
            If Max < Sj                  1(1/2)(n/2)
               then
                  Max ← Sj                        1m1
                  iMax=i+1                        2m1
            If Min > Si                  1(1/2)(n/2)
               then
                  Min ← Si                        1m3
                  iMin=i                          1m3
         else
            If Max < Si                  1(1/2)(n/2)
               then
                  Max ← Si                        1m2
                  iMax=i                          1m2
            If Min > Sj                  1(1/2)(n/2)
               then
                  Min ← Sj                        1m4
```

```
                iMin=i+1                          2m4
      end If
      i ← i+2                                  2(n/2)
   Until (i>n)                                 1(n/2)
Return (Max, Min, iMax, iMin)
End.
```

При анализе трудоемкости в среднем будем предполагать, что первый элемент в каждой паре равновероятно больше или меньше второго элемента пары. Это предположение отражено в счетчиках выполнений строк, приведенных в записи алгоритма (множитель $1/2$). Поскольку происходит перебор пар элементов, то количество проходов цикла равно половине длины массива, а инициализация значения счетчика цикла (i) обеспечивает всегда целое число исследуемых пар. При этом для массива четной длины обрабатывается $n/2$ пар, а для нечетной — $(n-1)/2$. Заметим также, что при поиске минимального и максимального элементов исходный массив разбивается на две половины, и, следовательно, сам поиск выполняется этим алгоритмом в одной половине элементов массива для минимума, и в другой половине — для максимума, таким образом, для массива нечетной длины

$$m_1 + m_2 = m_3 + m_4 = H_{n/2} - 1,$$

а для четной длины

$$m_1 + m_2 = m_3 + m_4 = H_{n/2},$$

Эти рассуждения позволяют получить функцию трудоемкости этого алгоритма в среднем (с учетом предположения, что при равномерном распределении средние значения равны — $\overline{m_1} = \overline{m_2} = \overline{m_3} = \overline{m_4}$):

$$\overline{f_{A2}}(n) \approx 2+1+1+1+2+\left(\frac{n}{2}\right)(2+3+1+2+2+1)+$$
$$+\left(\ln\left(\frac{n}{2}\right)+\gamma-1\right)\left(2\cdot\frac{1}{2}+3\cdot\frac{1}{2}\right)+\left(\ln\left(\frac{n}{2}\right)+\gamma-1\right)\left(2\cdot\frac{1}{2}+3\cdot\frac{1}{2}\right) = \quad (11.1.5)$$
$$= 5{,}5n + 5\ln n - 5\ln 2 + 5\gamma + 2 \approx 5{,}5n + 5\ln n + 1{,}38 = \Theta(n).$$

Приближенное равенство в (11.1.5) связано с тем, что формула не учитывает небольшие расхождения порядка $\Theta(1)$ для четных и нечетных длин массива. Автор оставляет получение функций $\overline{f_{A2}}(2n)$ и $\overline{f_{A2}}(2n+1)$ в качестве полезного упражнения для заинтересованных читателей.

Сравнивая (11.1.3) и (11.1.5) можно сказать, что предложенный алгоритм ищет максимум и минимум с практически такой же трудоемкостью, как алгоритм поиска максимума. Обращаясь к комплексной оценке ресурсной эффективности, можно заметить, что в данном случае «платой» за улучшение трудоемкости является увеличение объема памяти, занимаемого программой (длина кода).

Тем не менее, для всех рассмотренных алгоритмов ресурсная сложность одинакова, и имеет вид

$$\overline{\mathfrak{R}}_c(A) = \langle \Theta(n), \Theta(n) \rangle,$$

где второй компонент отражает суммарные затраты памяти, определяемые очевидно объемом хранения исходного массива. Дополнительные затраты памяти у всех алгоритмов составляют $\Theta(1)$.

11.2. Организация двоичного счетчика в массиве

В ряде задач, относящихся к области компьютерной математики, необходимо иметь точный двоичный счетчик с количеством разрядов, превосходящим размер стандартных типов простых данных. Отметим, что эта задача возникает и при экспериментальном исследовании трудоемкости алгоритмов, если нас интересует точное значение $f_A(D)$. Вообще арифметика длинных целых чисел — это отдельный раздел в области компьютерных алгоритмов с достаточно интересными решениями, и в этой книге читатель найдет ещё одну задачу из этого раздела, — а именно задачу умножения длинных целых чисел.

Рассмотрим следующую постановку задачи организации двоичного счетчика: определен массив целых чисел C, элементы которого рассматриваются как биты двоичного числа без знака в обычной записи, т. е. старший бит расположен слева. Длина массива n, равная разрядности числа, полагается заведомо большей, чем двоичный логарифм максимального ожидаемого значения счетчика. Иначе говоря, теоретически возможное переполнение не рассматривается, или же анализируется вне алгоритма увеличения счетчика, например, путем проверки на единицу «резервного» нулевого элемента массива. Изначально массив счетчика обнулён, и каждое обращение к процедуре счетчика увеличивает его значение на единицу. В рамках данной постановки, задача состоит в разработке такого алго-

ритма, который обладает минимальной трудоемкостью в среднем. Сама задача, очевидно, принадлежит к типу L_n.

Первое рассматриваемое решение состоит в сложении числа, содержащегося в массиве C, с единицей путем обычного двоичного сложения с переносом. Алгоритм **Count_A1**, реализующий эту идею, имеет следующую запись:

```
Count_A1 (C, n)
   (начальный перенос в следующий разряд - 1)
   p ← 1                                    1
   For i ← n downto 1                     1+3*n
      sm ← C[i]+p                          3*n
      C[i] ← sm mod 2                      3*n
      p ← sm div 2                         2*n
   end For
   C[0] ← p                                2
Return (C)
End.
```

Данный алгоритм по трудоемкости является, очевидно, количественно-зависимым (алгоритм относится к классу N), а его функция трудоемкости может быть легко получена на основе анализа операций в строках:

$$f_{A1}(n) = 11n + 4 = \Theta(n). \qquad (11.2.1)$$

Недостатки этого алгоритма достаточно очевидны — как только текущее значение переноса в следующий разряд станет равным нулю, алгоритм выполняет бессмысленное прибавление нуля к оставшимся слева разрядам. Таким образом, возникает новая идея — необходимо остановить вычисления при обнаружении нулевого переноса. Реализация этой идеи приводит к алгоритму **Count_A2**, который, как можно надеется, будет обладать лучшей временной эффективностью.

```
Count_A2 (C, n)
   (начальный перенос в следующий разряд - 1)
   p ← 1                                    1
   i ← n                                    1
   Repeat
      sm ← C[i] + p                        3*k
      C[i] ← sm mod 2                      3*k
      p ← sm div 2                         2*k
      i ← i - 1                            2*k
   Until p=0                               1*k
Return (C)
End.
```

Значение k есть количество повторений цикла, равное числу реально обрабатываемых бит счетчика. Это значение является вполне конкретным, если извест-

но, каково текущее состояние массива C. Предположение о невозможности переполнения счетчика есть гарантия того, что индекс массива не станет отрицательным. Трудоемкость алгоритма **Count_A2** зависит от состояния массива, и достаточно просто определить, что лучшим случаем трудоемкости является ситуация, когда счетчик хранит четное число, т. е. когда `C[n]=0`. В худшем случае счетчик хранит число $2^n - 1$, и все элементы массива, кроме элемента с нулевым индексом, равны единице. Таким образом, алгоритм является количественно-параметрическим и относится к классу NPR. Автор надеется, что читателям не составит труда получить самостоятельно функции трудоемкости для лучшего и худшего случаев.

И с практической, и с теоретической точек зрения, представляет интерес функция трудоемкости этого алгоритма в среднем, для получения которой придется воспользоваться методом амортизационного анализа (см. главу 5). Напомним, что метод предполагает наблюдение за трудоемкостью алгоритма в серии характерных вызовов, которая в данном случае состоит в заполнении предварительно обнуленного счетчика. Таким образом, наша цель состоит в получении информации о значениях k — числа проходов цикла в зависимости от хранимого счетчиком значения на протяжении 2^n вызовов, при счете от 0 до $2^n - 1$, а также частотной встречаемости этих значений. Рассмотрим вначале поведение k для нескольких первых обращений, и попытаемся определить закономерность — соответствующие значения приведены в таблице 11.1, где в скобках приведены десятичные значения счетчика.

Таблица 11.1

Состояние счетчика до вызова	Состояние счетчика после вызова	Значение k
0000 (0)	0001 (1)	1
0001 (1)	0010 (2)	2
0010 (2)	0011 (3)	1
0011 (3)	0100 (4)	3
0100 (4)	0101 (5)	1
0101 (5)	0110 (6)	2
0110 (6)	0111 (7)	1
0111 (7)	1000 (8)	4
1000 (8)	1001 (9)	1
1001 (9)	1010 (10)	2
1010 (10)	1011 (11)	1
1011 (11)	1100 (12)	3

Для всех четных чисел значение $k=1$ с частотной встречаемостью $1/2$; $k=2$ в ситуации, когда последние два элемента массива содержат значения «...01», а её частотная встречаемость равна $1/4$; $k=3$ (последние три элемента массива содержат значения «...011») с частотной встречаемостью $1/8$, и т. д. Эти рассуждения позволяют записать общее выражение для $\bar{k}(n)$ — значения k в среднем за 2^n вызовов алгоритма, при счете от 0 до 2^n-1, как функции от n:

$$\bar{k}(n)=1\cdot\frac{1}{2}+2\cdot\frac{1}{4}+3\cdot\frac{1}{8}+\cdots=\sum_{k=1}^{n+1}k\left(\frac{1}{2}\right)^k. \qquad (11.2.2)$$

Сумма (11.2.2) может быть легко вычислена на основе применения теоремы о дифференцировании абсолютно сходящихся рядов [11.3] к сумме геометрической прогрессии, общее решение при $0<x<1$, имеет вид

$$\sum_{k=1}^{\infty}k\,x^k=\frac{x}{(1-x)^2},$$

и для $x=1/2$ получаем

$$\sum_{k=1}^{\infty}k\left(\frac{1}{2}\right)^k=2. \qquad (11.2.3)$$

Ряд (11.2.2) достаточно быстро сходится, уже при $n=10$ значение $\bar{k}(10)=1{,}993652$ и для практических расчетов можно считать, что $\bar{k}(n)\approx 2,\ n\geq 10$. Таким образом, как это ни странно, среднее число проходов цикла асимптотически стремится к двойке, и практически не зависит от длины массива счетчика при $n\geq 10$. Это дает возможность получить трудоемкость алгоритма **`Count_A2`** в среднем

$$\overline{f_{A2}}(n)=2+11\cdot\bar{k}(n)=2+11\cdot 2=24. \qquad (11.2.4)$$

Мы получили асимптотически оптимальный результат — практически можно считать, что трудоемкость в среднем, при $n\geq 10$, не зависит от длины массива счетчика. За что еще можно бороться? Оказывается можно уменьшить значение среднего числа базовых операций. Если внимательно проанализировать таблицу 11.1, то можно заметить, что прибавление единицы приводит к обнулению всех единиц справа в записи текущего числа счетчика, до первого нуля, который заменяется единицей. Алгоритм **`Count_A3`** реализует эту идею.

```
Count_A3 (C, n)
   i ← n                                    1
   While (C[i]=1)                    2+2*(k-1)
      C[i] ← 0                         2*(k-1)
      i ← i - 1                        2*(k-1)
   end While
   C[i] ← 1                                 2
Return (C)
End.
```

Для четных значений в счетчике алгоритм **Count_A3** выполняет только проверку и сразу устанавливает младший (правый) элемент в единицу. Если же справа есть единицы, то они обнуляются до обнаружения первого нуля слева от них, который последней строкой устанавливается в единицу, что равноположено сложению текущего числа в счетчике с единицей. Поэтому тело цикла выполняется $k-1$ раз, под k понимается определённое выше число элементов массива C, изменяемых алгоритмом двоичного счетчика. На основе этих рассуждений и (11.2.3) получаем трудоемкость алгоритма **Count_A3** в среднем

$$\overline{f_{A3}}(n) = 5 + 6 \cdot \left(\overline{k}(n) - 1\right) = 5 + 6 = 11. \qquad (11.2.5)$$

Таким образом, алгоритм **Count_A3** затрачивает в среднем 11 базовых операций на увеличение счетчика и является оптимальным в принятой модели вычислений для рассматриваемой постановки задачи. Отметим, что более чем двукратное улучшение трудоемкости достигнуто и при сокращении дополнительно требуемых ячеек памяти — с трех до одной! Еще одно замечание касается того, что при разработке алгоритма **Count_A3** не использовался какой то «метод» — это результат «интеллектуального» подхода к разработке эффективных алгоритмов.

В заключение отметим, что для алгоритмов **Count_A2** и **Count_A3** ресурсная сложность в среднем имеет вид

$$\overline{\mathfrak{R}}_c(A) = \langle \Theta(1), \Theta(n) \rangle,$$

где второй компонент определяется, очевидно, объемом массива счетчика. В случае, если рассматриваются только затраты дополнительной памяти, то

$$\overline{\mathfrak{R}}_c(A) = \langle \Theta(1), \Theta(1) \rangle.$$

С точки зрения информационной чувствительности последовательность вызовов алгоритма **Count_A3** при счете от 0 до $2^n - 1$ задаёт такую частотную встречае-

мость значений трудоемкости, что частота лучшего случая равна $1/2$, и каждое следующее значение трудоемкости имеет вдвое меньшую частотную встречаемость. Это собственно и объясняет то, что трудоемкость в среднем не зависит от длины массива счетчика. При фиксированном значении n, худший случай трудоемкости будет наблюдаться всего один раз, а его частотная встречаемость равна 2^{-n}. Обратим внимание читателей на еще одну интересную особенность алгоритма **Count_A3** — количественно-параметрический алгоритм, обладающий в худшем случае оценкой $\Theta(n)$, имеет трудоемкость с оценкой $\Theta(1)$ в среднем.

В таблице 11.2 приведены экспериментальные результаты, подтверждающие полученные теоретические формулы трудоемкости. В заголовке таблицы **Fth** — теоретическое значение, **Fe** — экспериментальный результат.

Таблица 11.2

n	Fth A1	Fe A1	Fth A2	Fe A2	Fth A3	Fe A3
10	114,00	114,00	24,00	23,892	11,00	10,941
12	136,00	136,00	24,00	23,967	11,00	10,982
14	158,00	158,00	24,00	23,990	11,00	10,994
16	180,00	180,00	24,00	23,997	11,00	10,998
18	202,00	202,00	24,00	23,999	11,00	10,999
20	224,00	224,00	24,00	23,999	11,00	10,999

11.3. Определение областей рационального применения для алгоритмов сортировки вставками и сортировки методом индексов

Достаточно часто при анализе алгоритмов для получения функции трудоёмкости или других ресурсных функций приходится, помимо основного аргумента — длины входа, вводить дополнительные параметры, отражающие особенности задачи или данного алгоритма. В этом случае границы областей рационального применения алгоритмов зависят уже от нескольких аргументов и представляют собой некоторые кривые в пространстве аргументов ресурсных функций. Рассматриваемый пример определения областей рационального применения для алгоритмов сортировки вставками и сортировки методом индексов демонстрирует описанный выше подход к анализу.

Алгоритм сортировки массива методом прямого включения. Это достаточно хорошо известный алгоритм, использующий метод последовательного включения нового элемента в уже отсортированную часть массива. В литературе

по алгоритмам его часто называют также алгоритмом сортировки вставками. Изначально рассматривается массив, состоящий из одного элемента, к которому последовательно добавляются новые элементы из еще не отсортированной части исходного массива. При этом одновременно с поиском места для вставки нового элемента уже отсортированная часть массива справа сдвигается на одну позицию, освобождая место для вставки этого элемента. Таким образом, алгоритм работает «по месту», т. е. сортирует массив, не требуя дополнительного массива для размещения результата. Запись этого алгоритма в принятом алгоритмическом базисе имеет следующий вид:

```
Sort_Ins_A1 (A, n)
   For i ← 2 to n                          1+3(n-1)
      key ← A[i]                             2(n-1)
      j ← i-1                                2(n-1)
      While (j>0) and (A[j]>key)             4(k+1)
         A[j+1] ← A[j]                           4k
         j ← j - 1                               2k
      end While
      A[j+1] ← key                           3(n-1)
   end For
Return (A)
End.
```

Входом данного алгоритма является массив из n элементов — соответствующая задача имеет тип L_n, а трудоемкость алгоритма зависит как от количества чисел в массиве, так и от порядка их расположения — алгоритм является количественно-параметрическим (класс NPR, подкласс $NPRS$). Значение k в стоках, связанных с циклом **While**, обозначает неизвестное число проходов данного цикла. Нельзя заранее сказать, сколько проходов выполнит цикл **While** для некоторого элемента массива, но, тем не менее, мы можем получить результат, рассматривая обработку всего массива в целом, т. е. используя амортизационный подход. Для анализа трудоемкости в среднем определим, следуя [11.4], среднее общее количество сравнений, используя метод вероятностного анализа. Для этого заметим, что при вставке очередного элемента на i-ом шаге, когда $i-1$ элементов уже отсортированы, существует i позиций для вставки нового элемента. В предположении о равновероятном попадании нового элемента в любую позицию алгоритм выполняет в среднем $i/2$ сравнений и перемещений для его обработки, что в сум-

ме по всему внешнему циклу дает амортизационную оценку суммарного числа проходов цикла **`While`** при сортировке всего массива

$$\sum k = \sum_{i=1}^{n-1} \frac{i}{2} = \frac{n^2}{4} - \frac{n}{4}.$$

Используя трудоёмкости, указанные в строках записи алгоритма и методику анализа алгоритмических конструкций с учетом того, что на последнем проходе цикла **`While`** выполняется только сравнение, окончательно получаем функцию трудоемкости этого алгоритма в среднем

$$\overline{f}_{A1}(n) = 1 + (n-1)\cdot(3+2+2+4+3) + (4+4+2)\cdot\left(\frac{n^2}{4} - \frac{n}{4}\right) = \\ = 2{,}5n^2 + 11{,}5\cdot n - 13. \qquad (11.3.1)$$

Таким образом, данный алгоритм относится к группе квадратичных по трудоемкости алгоритмов сортировки. Отметим, что поскольку значение коэффициента при главном порядке мало, то можно прогнозировать рациональность этого алгоритма при малых длинах входа.

Приведенные в таблице 11.3 экспериментальные данные, полученные усреднением по 10000 массивам для каждой размерности, подтверждают полученный теоретический результат (**`Fth`** — теоретическое значение по формуле (11.3.1), **`Fe`** — экспериментальные результаты, **`d%`** — относительная ошибка).

Таблица 11.3

n	Fth	Fe	d%
4	73,00	73,09	0,1233%
8	239,00	239,20	0,0837%
16	811,00	810,92	-0,0099%
32	2 915,00	2 917,63	0,0902%
64	10 963,00	10 962,09	-0,0083%
128	42 419,00	42 419,06	0,0001%
256	166 771,00	166 782,82	0,0071%
512	661 235,00	661 249,14	0,0021%

Для данного алгоритма ресурсная сложность определяется главным порядком функции трудоемкости и имеет вид

$$\overline{\Re}_c(A) = \left\langle \Theta\left(n^2\right), \Theta(n) \right\rangle,$$

где второй компонент отражает суммарные затраты памяти, определяемые объемом хранения исходного массива. Дополнительные затраты памяти алгоритма составляют $\Theta(1)$.

Алгоритм сортировки массива методом индексов. Этот оригинальный метод работает не для всех возможных массивов. Его применение предполагает, что исходные числа являются целями и положительными. Основная идея алгоритма заключается в использовании дополнительного массива, который изначально заполнен нулями, и количество элементов в котором равно максимальному значению в исходном массиве. В предположении, что в исходном массиве нет повторяющихся элементов, можно использовать значение элемента из исходного массива в качестве индекса дополнительного массива. Таким образом, если некоторый элемент исходного массива равен 17, то 17-ый элемент дополнительного массива устанавливается в единицу. Остается только убрать «лишние» нули, и мы получим исходный массив, отсортированный по возрастанию. Более детальное рассмотрение этой идеи позволяет уменьшить затраты дополнительной памяти. На самом деле, если минимальный элемент исходного массива равен 1000, то, очевидно, что первые 1000 элементов дополнительного массива не будут использоваться. Таким образом, размер дополнительного массива может быть равен размаху варьирования сортируемых чисел. Реализация такого алгоритма включает в себя следующие шаги:

— определение максимального и минимального значений в исходном массиве и размаха варьирования;

— обнуление дополнительного массива;

— заполнение дополнительного массива единицами методом индексов;

— получение сортированного массива путем сборки единиц в дополнительном массиве и сдвига ненулевых индексов на размах варьирования.

Запись этого алгоритма в принятом алгоритмическом базисе имеет вид:

```
Sort_Ind_A2 (A, n)
   Max ← A[1]                                  2
   Min ← A[1]                                  2
   For i ← 2 to n                          1+3(n-1)
      If Max < A[i]                          2(n-1)
         then
```

```
         Max ← A[i]                          2m
   If Min > A[i]                          2(n-1)
      then
         Min ← A[i]                          2m
end For
d ← Min-1                                     2
L ← Max-Min+1                                 2
For i ← 1 to L                             1+3L
   B[i] ← 0                                  2L
end For
For i ← 1 to n                             1+3n
   B[A[i]-d] ← 1                             4n
end For
i ← 1                                         1
For j ← 1 to L                             1+3L
   If B[j]=1                                 2L
      then
         A[i]=j+d                            3n
         i ← i+1                             2n
end For

Return (A)
End.
```

Отметим, что трудоемкость этого алгоритма существенно зависит от размаха варьирования L, который и будет использован как дополнительный аргумент функции трудоемкости. Трудоемкость первого этапа алгоритма — поиска минимума и максимума уже была получена в параграфе 11.1, и мы будем использовать результат для среднего случая. С учетом введенной параметризации функция трудоемкости может быть легко получена на основе информации о числе базовых операций, указанных в строках записи алгоритма.

$$\begin{aligned}\overline{f_{A2}}(n,L) &= 7n+4\ln(n)-6+4\gamma+5+1+5L+1+7n+1+1+5L+5n = \\ &= 10L+19n+4\ln(n)+3+4\gamma.\end{aligned} \qquad (11.3.2)$$

Отметим, что при большом размахе варьирования объем дополнительной памяти, требуемой данным алгоритмом, может быть значительным. Интересно отметить также, что при значениях $n \approx L$ данный алгоритм за счет дополнительный затрат памяти сортирует массив за линейное время.

Для данного алгоритма ресурсная сложность имеет вид

$$\overline{\mathfrak{R}}_c(A) = \langle \Theta(n+L), \Theta(n+L) \rangle,$$

где второй компонент отражает суммарные затраты памяти, определяемые объемом исходного и дополнительного массивов. Дополнительные затраты памяти алгоритма составляют $\Theta(L)$.

Определение областей рационального применения алгоритмов. При определении областей рационального применения рассмотренных алгоритмов будем считать, что исходные размерности таковы, что компоненты порядка $\Theta(1), \Theta(\ln n)$ не являются существенными. В этом случае, приравнивая трудоемкости, получаем уравнение

$$2{,}5n^2 + 11{,}5n = 10L + 19n,$$

решая которое относительно L, получаем

$$L(n) = \frac{1}{4}n^2 - \frac{3}{4}n. \qquad (11.3.3)$$

Таким образом, в координатах аргументов функции трудоемкости — L, n мы получили уравнение параболы, разграничивающей области рационального применения рассматриваемых алгоритмов, что проиллюстрировано на рисунке 11.1.

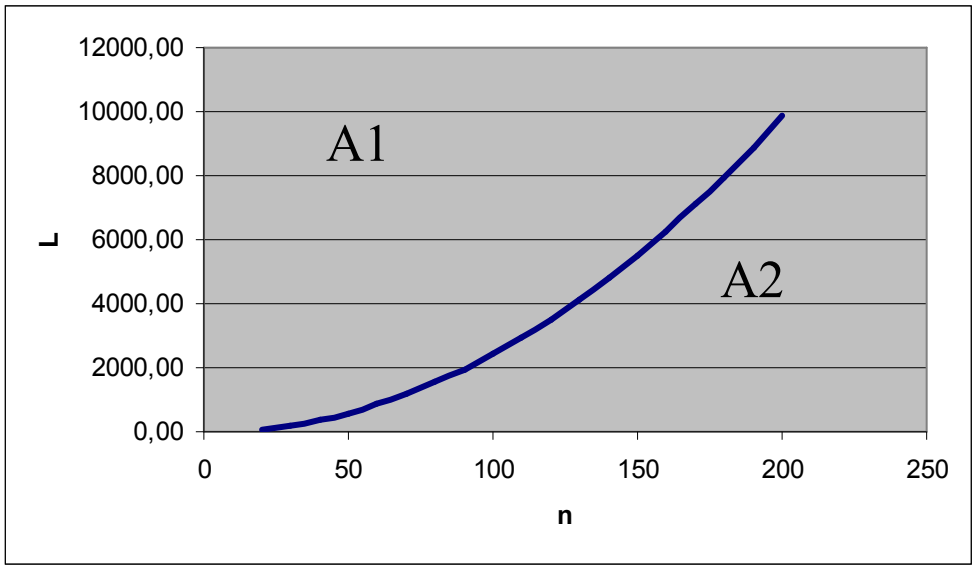


Рисунок 11.1. Разграничение областей рационального применения алгоритмов сортировки
методом индексов и методом прямого включения

Для конкретных случаев выбора можно вычислить пороговое значение L по формуле (11.3.3) и сравнить с реальным размахом варьирования значений исходного массива. Например, при $n = 100$ пороговое значение L равно 2425, т. е. если размах варьирования в исходном массиве меньше чем 2425, то более эффектив-

ным по трудоемкости будет алгоритм сортировка индексами (очевидно в рамках условий его применения), при больших размахах варьирования — алгоритм сортировки методом индексов.

11.4. Выбор алгоритма умножения длинных целых чисел на основе анализа временной эффективности

Современная криптография и задача умножения длинных целых чисел. В настоящее время реализация целого ряда криптосистем приводит к необходимости выполнения арифметических операций над длинными целями числами. В качестве примера рассмотрим систему асимметричного шифрования RSA, разработанную в 1977 году исследователями из Массачусетского Технологического Института Рональдом Ривестом, Ади Шамиром и Леонардом Адлеманом [11.5]. Как шифрование, так и дешифрование сообщения в RSA 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 асимметричного 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 существенно опирается на то, что в настоящее время неизвестен достаточно быстрый алгоритм для разложения больших чисел на простые множители. Рассмотрим более подробно процесс генерации ключей в системе RSA. На первом шаге выбираются два больших простых числа p и q. Эти числа, помимо простоты, должны иметь длину в сотни бит (в настоящее время достаточ-

но надежным выбором считаются числа от 512 бит), а числа $(p-1)$ и $(q-1)$ — иметь хотя бы один большой простой множитель. На втором шаге вычисляется 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 $ed = 1 \bmod \varphi(n)$. 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 $y = x^e \bmod n$, а дешифрование сообщения y — вычислением $x = y^d \bmod n$ [11.5].

Итак, для генерации ключей в системе RSA нужно уметь генерировать большие простые числа и быстро перемножать их. Шифрование и дешифрование сообщений производится быстрым алгоритмом возведения в степень методом повторного возведения в квадрат, что опять приводит к многократному решению задаче умножения длинных целых чисел.

Алгоритм умножения в столбик (A1). Будем далее предполагать, что два длинных целых двоичных числа — сомножители, равно как и результат умножения представляют собой битовые массивы в обычном кодировании, т. е. старший бит числа расположен слева и хранится в первом элементе массива. Первый рассматриваемый алгоритм реализует обычное умножение битовых чисел, сводящееся к сложению со сдвигом. Если очередной бит второго числа равен единице, то первое число добавляется к результату с соответствующим сдвигом, эту очевидную идею и реализует алгоритм **Mult_A1**. Массивы **A** и **B** содержат исходные n разрядные двоичные числа, а результат формируется в $2n$ ячейках массива **C**. Запись алгоритма **Mult_A1** имеет вид.

```
Mult_A1(n, A, B, C)
(A, B - исходные числа - массивы длиной n)
(C - массив результата длины 2n)
(D - вспомогательный массив для суммирования)
   n2 ← n*2                                    2
   n1 ← n-1                                    2
   For i ← 1 to n2                         1+3*2*n
      C[i] ← 0                               2*2*n
   end for
   dc ← n2 (начальное смещение суммирования)  1
```

```
   For i ← 0 to n1                              1+3*n
         (цикл по битам массива B)
      jb ← n-i                                   2*n
      If B[jb] = 1                               2*n
         then  (суммирование: C=C+A)
            p ← 0                                1*n
            jc ← dc                              1*n
            For ja ← n downto 1             1*n+3*n*n
               s ← C[jc]+A[ja]+p                5*n*n
               D[jc] ← s mod 2                  3*n*n
               p ← s div 2                      2*n*n
               jc ← jc-1                        2*n*n
            end for
            D[jc] ← p                            2*n
            (копирование в исходный массив)
            For i ← 1 to n2                1*n+3*2*n*n
               C[i] ← D[i]                     3*2*n*n
            end for
      end if
      dc ← dc-1                                  2*n
   end for
End.
```

Получим трудоемкость этого алгоритма в среднем относительно длины исходных массивов — n, в предположении о равной вероятности появления нулей и единиц в массиве **B**. Трудоемкость фрагмента сложения не зависит от значений бит, хранящихся в ячейках массивов, и на основании количества базовых операций, указанных в строках записи алгоритма, имеем

$$\overline{f}_{A!}(n) = 7 + 10n + 9n + \frac{1}{2}\left(6n + 15n^2 + 12n^2\right) = 13\frac{1}{2}n^2 + 22n + 7. \qquad (11.4.1)$$

Квадратичная сложность этого алгоритма очевидна, и мы будем сравнивать его с более эффективным (по асимптотике трудоемкости) алгоритмом умножения, предложенным А. А. Карацубой.

Математическое обоснование алгоритма Карацубы (А2). Читателям уже известно, что метод декомпозиции позволяет, в целом ряде случаев, получить достаточно эффективные в смысле вычислительной сложности рекурсивные алгоритмы. Для задачи умножения длинных целых чисел использование этого метода позволило А. А. Карацубе [11.6] впервые, в 1962 г., получить алгоритм умножения двух n битовых чисел, имеющий асимптотическую оценку, лучшую, чем $\Theta\left(n^2\right)$. Изложим основные идеи этого алгоритма. Пусть a, b — два n битовых числа в обычном двоичном представлении, т. е. старший бит расположен сле-

ва. Пусть число разрядов n таково, что $n = 2^k$ — это обеспечивает целое деление n пополам без остатка на всех рекурсивных вызовах. Обозначим через nh значение $n/2$. Поскольку метод декомпозиции предполагает разделение задачи, в данном случае всегда на две равные части на любом уровне рекурсии, то представим числа a, b в виде

$$a = 2^{nh} \cdot a_2 + a_1, \quad b = 2^{nh} \cdot b_2 + b_1,$$

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, b. Произведение $a \cdot b$ может быть записано в следующем виде, и это есть основная идея, предложенная А. А. Карауцбой [11.6]

$$a \cdot b = 2^n \cdot a_2 \cdot b_2 + 2^{nh} \cdot ((a_1 + a_2) \cdot (b_1 + b_2) - (a_1 \cdot b_1 + a_2 \cdot b_2)) + a_1 \cdot b_1. \quad (11.4.2)$$

Произведения в (11.4.2) 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 их можно представить в виде суммы чисел длиной nh — a_3, b_3, и однобитовых чисел — a_4, b_4

$$a_1 + a_2 = 2 \cdot a_3 + a_4, \quad b_1 + b_2 = 2 \cdot b_3 + b_4,$$

тогда

$$(a_1 + a_2) \cdot (b_1 + b_2) = 4 \cdot a_3 \cdot b_3 + 2 \cdot a_4 \cdot b_3 + 2 \cdot a_3 \cdot b_4 + a_4 \cdot b_4, \quad (11.4.3)$$

где $a_3 \cdot b_3$ есть произведение чисел длиной nh. Слагаемое $a_4 \cdot b_4$ в (11.4.3) 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 А2. Получим 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 (11.4.2), 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

(11.4.3). В данном случае предполагается, что числа настолько велики, что они хранятся в массивах, каждый элемент которого содержит один бит числа. Рекурсия останавливается при значении $n=1$, когда произведение $a \cdot b$ вычисляется элементарно, и требует не более чем фиксированного числа базовых операций, которое мы обозначим через C. Поскольку алгоритм рекурсивно перемножает три пары чисел половинной длины, то приведенные выше рассуждения позволяют записать рекуррентное соотношение для функции трудоемкости исследуемого алгоритма

$$\begin{cases} f_A(1) = C; \\ f_A(n) = 3 \cdot f_A(n/2) + \Theta(n). \end{cases}$$

Используя основную теорему о рекуррентных соотношениях (Бентли, Хакен, Сакс), мы немедленно получаем оценку

$$f_A(n) = \Theta\left(n^{\log_2 3}\right),\ n^{\log_2 3} \approx n^{1,5849}, \qquad (11.4.4)$$

что асимптотически лучше умножения «в столбик» с оценкой $\Theta\left(n^2\right)$. Для любознательных читателей укажем, что асимптотически лучший (с доказанной нижней границей задачи) алгоритм умножения длинных целых чисел, принадлежит Штрассену и Шенгаге [11.7] и имеет оценку $f_A(n) = \Theta(n \cdot \ln n \cdot \ln \ln n)$.

Трудоемкость алгоритма А2. Трудоемкость алгоритма Карацубы, реализованного на основе вспомогательных процедур, получена в [11.8] на основе метода рекуррентных соотношений, и задается следующей формулой

$$\overline{f}_A(n) = 667{,}5 \cdot n^{\log_2 3} - 281 \cdot n - 352{,}5, \qquad (11.4.5)$$

что, очевидно, согласуется с асимптотической оценкой. Трудоемкость этого алгоритма, в смысле значения коэффициента при главном порядке, может быть улучшена за счет замены каждого вызова вспомогательных процедур на соответствующее тело процедуры. Такая модификация приводит к увеличению длины текста алгоритма и потери ясности, но любые улучшения ресурсной эффективности должны чем-то оплачиваться. Теоретические расчеты, выполненные автором, показывают, что функция трудоемкости такого модифицированного алгоритма составляет

$$\overline{f}_{A2}(n) = 322{,}5 \cdot n^{\log_2 3} - 281 \cdot n - 46, \qquad (11.4.6)$$

Решение по рациональному выбору между алгоритмом Карацубы (А2) и алгоритмом, реализующим умножение в столбик, может быть принято на основе детального анализа их функций трудоемкости. Соответствующие значения приведены в таблице 11.4, трудоёмкость алгоритма $A1$ вычислена по формуле (11.4.1), алгоритма $A2$ — по (11.4.6), значение $\Delta = \overline{f}_{A2}(n) - \overline{f}_{A1}(n)$.

Таблица 11.4

n	$\overline{f}_{A1}(n)$	$\overline{f}_{A2}(n)$	Δ
2000	54 044 007,0	54 460 220,5	416 213,5
2005	54 314 454,5	54 676 995,4	362 540,9
2010	54 585 577,0	54 894 088,8	308 511,8
2015	54 857 374,5	55 111 500,5	254 126,0
2020	55 129 847,0	55 329 229,9	199 382,9
2025	55 402 994,5	55 547 276,9	144 282,4
2030	55 676 817,0	55 765 641,1	88 824,1
2035	55 951 314,5	55 984 322,1	33 007,6
2040	56 226 487,0	56 203 319,7	**- 23 167,3**
2045	56 502 334,5	56 422 633,6	**- 79 700,9**
2050	56 778 857,0	56 642 263,3	**- 136 593,7**

На основании этих данных можно рекомендовать алгоритм $A2$, как более эффективный по трудоемкости, только начиная с длин массивов более 2040. Заметим, что эта рекомендация относится к обсуждаемой модификации алгоритма Карацубы, описанного в [11.8]. Данные ряда публикаций показывают, что алгоритм Карацубы эффективнее, начиная с длины сомножителей порядка 1500-2000 битов [11.9], что согласуется с полученным выше результатом.

Еще одно замечание касается перехода от трудоемкости к реальному времени выполнения, и, поскольку операции обслуживания программного стека являются достаточно быстрыми, то следует ожидать определенного понижения границы рационального выбора для алгоритма Карацубы по критерию временной эффективности. Отметим, что асимптотически оптимальный алгоритм умножения длинных целых чисел — алгоритм Штрассена-Шенхаге, реально эффективен, начиная с еще больших длин входа. Возможные дополнительные улучшения рассматриваемых алгоритмов обсуждаются в упражнениях к этой главе.

11.5. Выбор рациональных алгоритмов поиска по ключу на основе анализа их информационной чувствительности

В отличие от предыдущих, данный параграф посвящен изложению результатов реальных исследований, связанных с выбором рациональных алгоритмов поиска по ключу по критерию информационной чувствительности на основе экспериментально определенных тактовых времен выполнения программных реализаций.

Содержательная постановка задачи. Излагаемые ниже результаты основаны на статье [11.10], в которой рассматривается задача выбора рационального алгоритма для поиска в индексных структурах баз данных оперативной памяти (БДОП). Разработка СУБД, поддерживающих БДОП, ведется уже более 20 лет, однако широкое их использование стало возможно лишь в последние годы с повышением доступности и ростом объема модулей памяти. Имеется оценка группы ведущих исследователей, согласно которой в обозримом будущем все массивы данных, за исключением наиболее крупных, будут храниться в БДОП [4].

При разработке методов доступа к данным для БДОП важно минимизировать процессорное время и объем основной памяти, при этом для повышения эффективности поиска в реляционных базах данных (РБД) обычно используют дополнительные структуры данных — индексы двух основных разновидностей: деревья поиска и хэш-массивы [11.11]. На стратегию выделения памяти в БДОП влияет необходимость обеспечения надежности путем хранения вторичных копий данных на дисках. Поэтому память в БДОП выделяется страницами (блоками), размер которых согласован с размером кластера файловой системы и обычно составляет от 2 до 64 килобайт. Оценим количество элементов на странице. Минимальный размер элемента — 4 байта (размер указателя), соответственно, максимальное количество элементов на странице — 16384 (для кластера в 64 Кб). Нижняя оценка количества элементов может быть получена в предположении об использовании суррогатных ключей при составных индексах, содержащих не более четырех компонентов. В этом случае количество элементов на странице составляет 4096, и, учитывая использование части объема страницы под хранение служеб-

ной информации, получаем оценку числа элементов на странице в интервале от 4000 до 16000.

Таким образом, задача состоит в выборе такого алгоритма поиска по ключу, который был бы рационален для массива ключей, длина которого совпадает с размером страницы памяти, а число элементов может изменяться от 4000 до 16000.

Описание исследуемых алгоритмов. В связи с указанными выше особенностями поиска в БДОП и оговоренным сегментом длин входов, в качестве претендующих рассматриваются следующие основные алгоритмы: алгоритм бинарного поиска; алгоритм интерполяционного поиска и алгоритм поиска с использованием хеш-функции.

Стандартный алгоритм бинарного поиска достаточно полно описан в [11.12], и, как известно, имеет следующую оценку временной эффективности в среднем — $\Theta(\ln(n))$, где n — длина массива ключей.

Алгоритм интерполяционного поиска [11.12] в предположении о равномерном распределении ключей имеет оценку в среднем $\Theta(\ln\ln(n))$, однако выполняет большее количество операций на один шаг, что связано с расчетом позиции для следующего обращения.

Для алгоритмов хеширования временные оценки определяются как самой хеш-функцией, так и отношением количества ключей к выделенному объему памяти, определяющим среднюю длину списка коллизий [11.7]. В реализации использовалась достаточно хорошо зарекомендовавшая себя хеш-функция вида [11.7]:

$$h(K) = \left(K^2 + K + 1\right) \bmod M, \qquad (11.5.1)$$

где M — длина основного хеш-массива ключей, а K — численное значение ключа. Для алгоритмов хеширования временная эффективность сильно зависит от выделенного объема памяти, в то время как алгоритмы бинарного и интерполяционного поиска работают непосредственно с отсортированным массивом ключей.

Для того чтобы уравнять оценки емкостной эффективности можно использовать минимальную совершенную функцию хеширования [11.7], но временная оценка ее реализации очевидно неприемлема в рамках рассматриваемой поста-

новки задачи. Поэтому при выборе длины основного хеш-массива ключей будем руководствоваться ограничением на значение β — отношения среднего ожидаемого количества промахов хеш-функции в основном массиве ключей к длине этого массива. Формула для β может быть определена на основе следующих рассуждений: в предположении о равномерном распределении значений хеш-функции ключей вероятность занятия элемента основного массива равна $1/M$, после заполнения основного массива n ключами с построением списка коллизий, по вероятности имеем:

$$\beta = \left(1 - \frac{1}{M}\right)^n, \tag{11.5.2}$$

откуда логарифмируя и используя приближение: $\ln(1 \pm x) = \pm x$ для малых x, а затем, возводя в степень, окончательно получаем:

$$\beta \approx e^{-\frac{n}{M}}. \tag{11.5.3}$$

Если

$$n = 2 \cdot M, \text{ то } \beta \approx 0{,}135335,$$

что является вполне приемлемым в смысле уравнивания оценок емкостной эффективности алгоритма поиска на основе хеш-функции и других рассматриваемых алгоритмов.

Показатель среднего такового времени и методика проведения экспериментов. Сравнение временной эффективности алгоритмов проводилось по показателю среднего тактового времени на одну операцию поиска. Значение $t_{\text{такт}}(key)$, где key — конкретный ключ поиска, определялось на основе обращения к регистру тактового счетчика до, и после обращения к программной реализации алгоритма поиска. Для получения значимых средних времен при фиксированной длине массива ключей количество экспериментов — Ne рассчитывалось на основе методики, предложенной в [11.13], при этом для достоверности $\gamma = 0{,}95$ значение Ne определяется по формуле:

$$Ne = 3{,}8416 \cdot \frac{V^2}{\varepsilon^2}, \tag{11.5.4}$$

где V — коэффициент вариации, а ε — относительная ошибка между выборочным и генеральным средними. Поскольку для заданного диапазона длин входов и среды реализации $\bar{t}_{\text{такт}}(n)$ не превышает 1000, то значение ε было выбрано равным 0,001, что обеспечивает три значащие цифры $\bar{t}_{\text{такт}}(n)$. Для полученных выборочных средних и дисперсий, значение Ne, вычисленное по (11.5.4) лежало в диапазоне от 38000 до 56000 в зависимости от исследуемого алгоритма и длины массива ключей. Значение $\bar{t}_{\text{такт}}(n)$ определялось как выборочное среднее по Ne экспериментам на входах длины n.

Для каждого исследуемого значения размерности $4000 \le n \le 16000$ с шагом $\Delta n = 250$ стандартным равномерным генератором формировался массив исходных ключей в виде действительных 10-байтовых чисел, на котором исследовались претендующие алгоритмы. Особенности рассматриваемой постановки задачи поиска в БДОП учитывались значением показателя частотной встречаемости обращения по поиску отсутствующего ключа, которое было выбрано равным 50%. В каждом из Ne экспериментов для фиксированной размерности ключ поиска генерировался с равной частотной встречаемостью либо как случайное обращение к номеру ключа в исходном массиве, либо как новая случайная генерация ключа, что для рассматриваемого сегмента длин массива ключей практически обеспечивало генерацию отсутствующего ключа.

Программная реализация алгоритмов поиска была выполнена в среде программирования Borland Delphi 7.0 под ОС Microsoft Windows XP Professional, экспериментальное исследование проводилось на компьютере со следующей конфигурацией:

— процессор Intel Pentium III с тактовой частотой 800 МГц;

— размер кэш-памяти второго уровня, работающей на частоте 800 МГц — 256 Кбайт;

— размер оперативной памяти — 512 Мбайт,

— частота шины памяти — 133МГц.

Экспериментальная рациональная граница прямого перебора. Для алгоритмов бинарного и интерполяционного поиска хорошо известны рекомендации по использованию метода прямого перебора на коротких длинах массива ключей

[11.7]. Поскольку реальная граница длины массива, при которой рационально переходить на прямой перебор, зависит от среды реализации, то предварительной задачей исследования алгоритмов поиска для целей БДОП было ее экспериментальное определение. Алгоритмы бинарного и интерполяционного поиска сравнивались на сегменте длин массива ключей от 5до 30 с алгоритмом прямого перебора по показателю временной эффективности. Логарифмические тренды полученных точечных значений среднего тактового времени алгоритмов поиска для исследуемого сегмента размерностей приведены на рисунке 11.2.

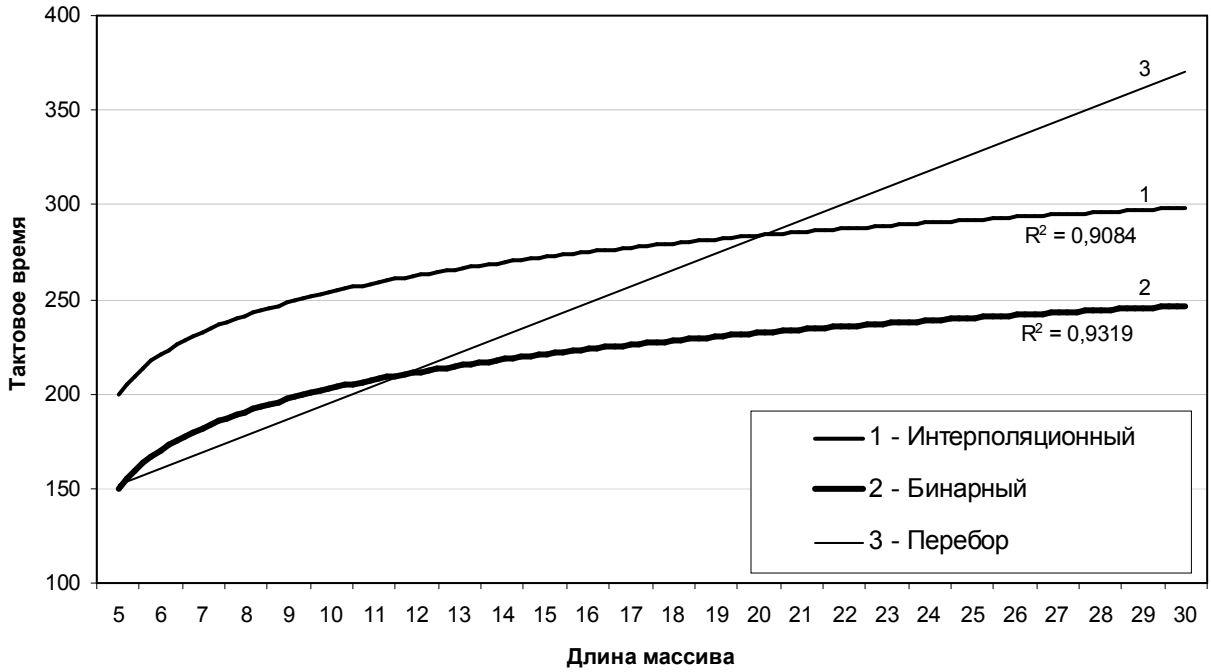


Рисунок 11.2. Зависимость среднего тактового времени от длины массива ключей для программных реализаций алгоритмов интерполяционного поиска, бинарного поиска и поиска методом прямого перебора.

На основе полученной информации были реализованы комбинированные алгоритмы бинарного и интерполяционного поиска с пороговой длиной переключения на прямой перебор равной 11 и 20 соответственно.

Предварительный выбор алгоритмов. По описанной выше методике проведения экспериментов для исследуемого сегмента размерностей массива ключей [4000, 16000] было проведено исследование алгоритма хеширования, алгоритмов бинарного поиска, интерполяционного поиска и их модификаций, полученных путем комбинирования с прямым перебором. Логарифмические тренды получен-

ных точечных значений для среднего тактового времени приведены на рисунке 11.3. В результате для дальнейшего детального исследования были выбраны комбинированный алгоритм интерполяционного поиска и алгоритм хеширования.

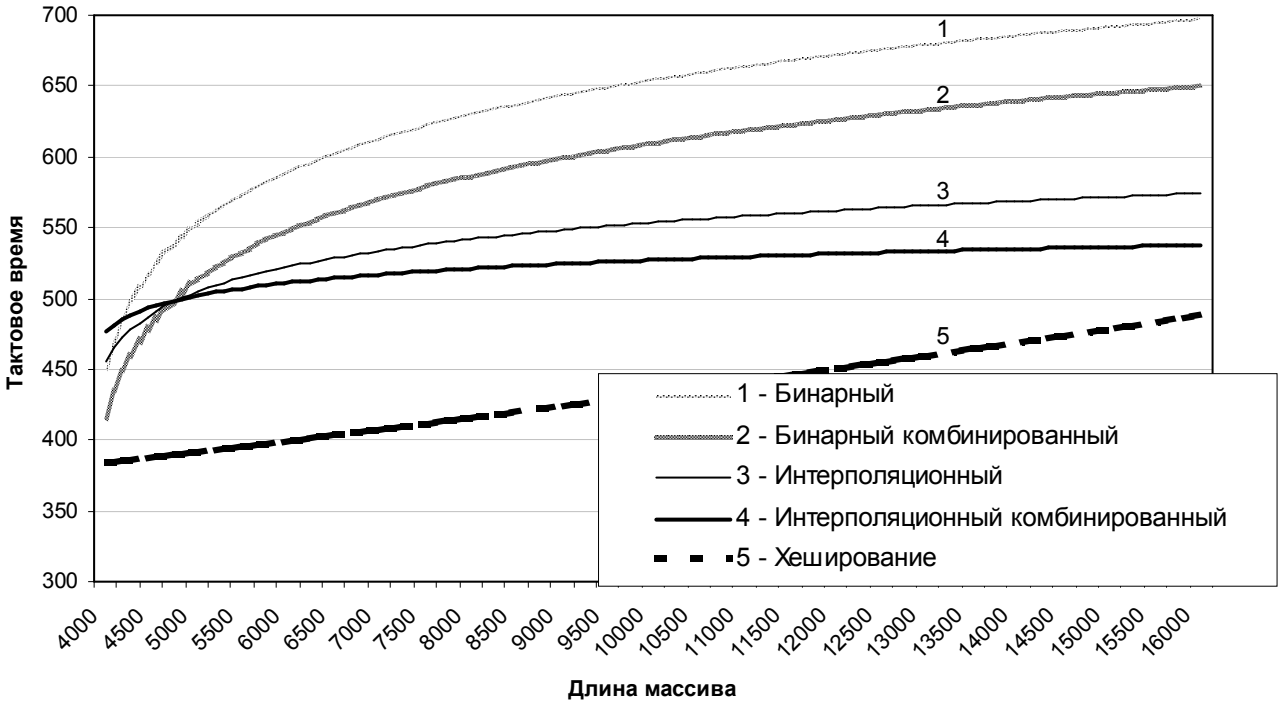


Рисунок 11.3. Зависимость среднего тактового времени от длины массива ключей для программных реализаций алгоритмов интерполяционного поиска, бинарного поиска, их модификаций, комбинированных с прямым перебором, и алгоритма поиска методом хеширования.

Особенности реализации алгоритмов хеширования и комбинированного интерполяционного поиска. Поскольку экспериментальные исследования выполнялись с программными реализациями алгоритмов, то в этом параграфе приводятся тексты программ, а не записи алгоритмов в принятом алгоритмическом базисе. В алгоритме комбинированного интерполяционного поиска шаги интерполяции выполняются до тех пор, пока длина текущего интервала поиска не становится меньше 20, что было обосновано на этапе предварительного исследования. Каждый такой шаг начинается со сравнения искомого ключа с крайними значениями интервала, затем вычисляется индекс ключа, делящего интервал пропорционально искомому ключу. Если ключ с вычисленным индексом совпадает с искомым — поиск завершен, если он меньше искомого — поиск продолжается в правом подинтервале, иначе — в левом подинтервале.

```
Function FindInterpolated( AVal: Extended ): Longint;
var
   i, iFrom, iTo: Longint;
Begin
   Result := 0;
   iFrom := 0;
   iTo := DataLen; //максимальный индекс в массиве Data
   // интерполяция до порога SLevel=20
   While (iTo-iFrom) > SLevel do
      begin
         // проверка на попадание в интервал
         If (AVal < Data^[iFrom]) or (AVal > Data^[iTo])
            then
               exit;
         // вычисление индекса
         i := iFrom +
                     Round(((iTo-iFrom)*(AVal-Data^[iFrom]))/
                     (Data^[iTo]-Data^[iFrom]));
         If Data^[i] < AVal
            then
               iFrom := i+1 // переход к правому подинтервалу
            else
               If Data^[i] > AVal
               then
                  iTo := i-1 // переход к левому подинтервалу
               else
                  begin
                     Result := i; // ключ найден
                     exit;
                  end;
      end;
   Repeat  // после порога - перебор
      If (Data^[iFrom]=AVal)
         then
            begin
               Result := iFrom;
               break;
            end;
      iFrom := iFrom+1;
   until iFrom > iTo;
End;
```

В алгоритме хеширования после вычисления хеш-функции выполняется проход по выбранной цепочке коллизий, пока либо не найдется искомый ключ, либо не закончится цепочка.

```
// вычисление хеш-функции
Function Hash(AVal: Extended): Longint;
begin
   AVal := AVal*HashLen; // HashLen - длина хеш-массива\
   Result := Round(AVal*AVal+AVal+1) mod HashLen;
end;

Function FindHash(AVal: Extended): Longint;
var
```

```
   v: Longint;
   p: PHashRec;
Begin
   Result := 0;
   v := Hash(AVal); // вычисление хеш-функции
   P := @HashData^[v]; // P - ссылка на элемент хеш-массива
   Repeat
      If (P^.V = AVal)
         then
            begin
               Result := v; // ключ найден
               break;
            end;
      P := P^.Next; // переход к следующему ключу в цепочке
   until P = nil; // пока цепочка не закончится
End;
```

Исследование алгоритмов по частотному распределению тактового времени поиска. Будем рассматривать в качестве входа множество D, представимое в виде массива объектов в котором производится поиск, имеющего размерность n. Множество, содержащее все допустимые входы алгоритма размерности n, обозначим через D_n. Тогда общее количество возможных входов с заданной размерностью представляется некоторым числом N, таким что:

$$N : D_n = \left\{ D_j \,|\, |D_j| = n,\ j = \overline{1, N} \right\}, |D_n| = N, \tag{11.5.5}$$

причем заведомо $N >> 1$ при $n > 0$. Для каждого конкретного входа D_j из множества D_n имеется функция $t_{\text{такт}}(D_j)$, задающая число тактов, выполненных процессором на входе D_j. Отметим, что значение функции $t_{\text{такт}}(D_j)$ есть целое положительное число. Значение $t_{\text{такт}}(D_j)$ ограничено между:

$$t^{\vee}_{\text{такт}}(n) \le t_{\text{такт}}(D_j) \le t^{\wedge}_{\text{такт}}(n), \tag{11.5.6}$$

где $t^{\vee}_{\text{такт}}(n)$ — лучший случай, $t^{\wedge}_{\text{такт}}(n)$ — худший случай, по всем входам из множества D_n. Введем в рассмотрение функцию:

$$T_n^{\text{такт}}(m) = \frac{X(m)}{N},\ \forall m \in \left[t^{\vee}_{\text{такт}}(n), t^{\wedge}_{\text{такт}}(n) \right], \tag{11.5.7}$$

где функция $X(m)$ задает количество элементов во множестве D_n, для которых $t_{\text{такт}}(D_j) = m$, т. е. количество появлений значения m, как тактового времени выполнения программной реализации алгоритма на входах из множества D_n, а N есть количество элементов во множестве D_n.

Фактически функция $T_n^{\text{такт}}(m)$ является частотным аналогом закона распределения дискретной ограниченной случайной величины T_n, где T_n — количество тактов, выполняемых процессором при реализации алгоритма поиска на входах длины n. Функция $T_n^{\text{такт}}(m)$ является полной характеристикой программной реализации алгоритма с точки зрения тактового показателя его временной эффективности как дискретной ограниченной случайной величины в заданной среде реализации. Очевидно, что в эксперименте мы получаем выборочные значения для $T_n^{\text{такт}}(m)$.

Исследование алгоритмов на основе функции $T_n^{\text{такт}}(m)$ предполагает определение выборочного размаха варьирования и исследования вида функции $T_n^{\text{такт}}(m)$, которое и было проведено для выбранных алгоритмов поиска. Совмещенный график огибающих для границ значений $t_{\text{такт}}(n)$ приведен на рисунке 11.4, частотные гистограммы выборочных значений $T_n^{\text{такт}}(m)$ для исследуемых алгоритмов приведены на рисунках 11.5 и11.6. Гистограммы построены на основании обработки результатов $16 \cdot 10^6$ экспериментов для каждого значения n.

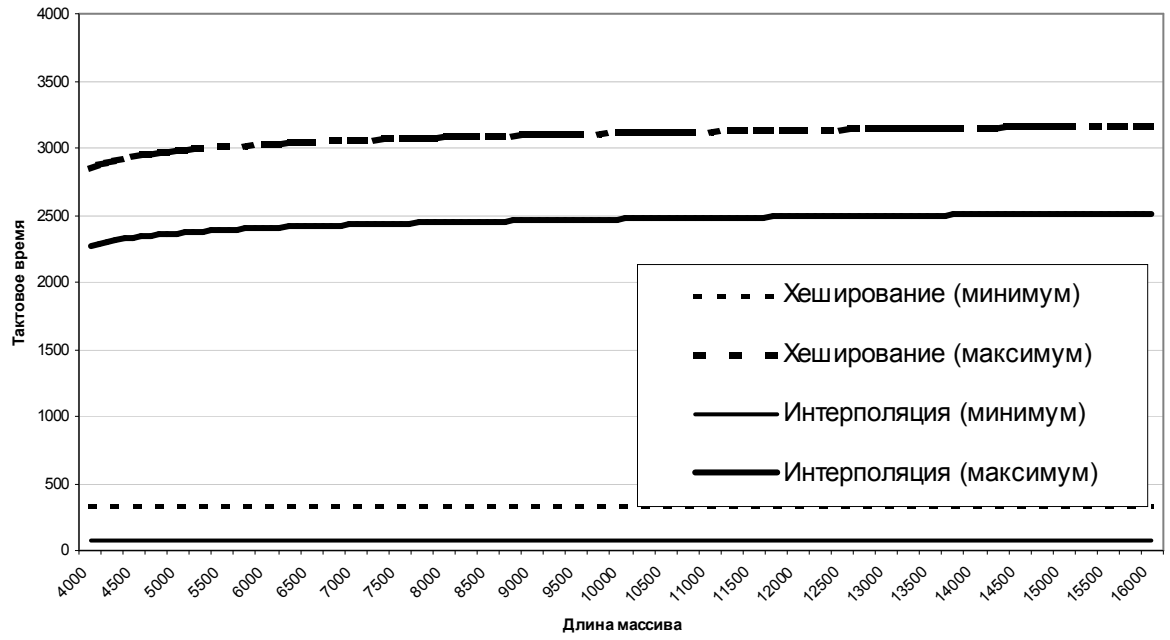


Рисунок 11.4. Зависимость минимального и максимального тактового времени от длины массива ключей для программных реализаций алгоритмов комбинированного интерполяционного поиска и поиска методом хеширования.

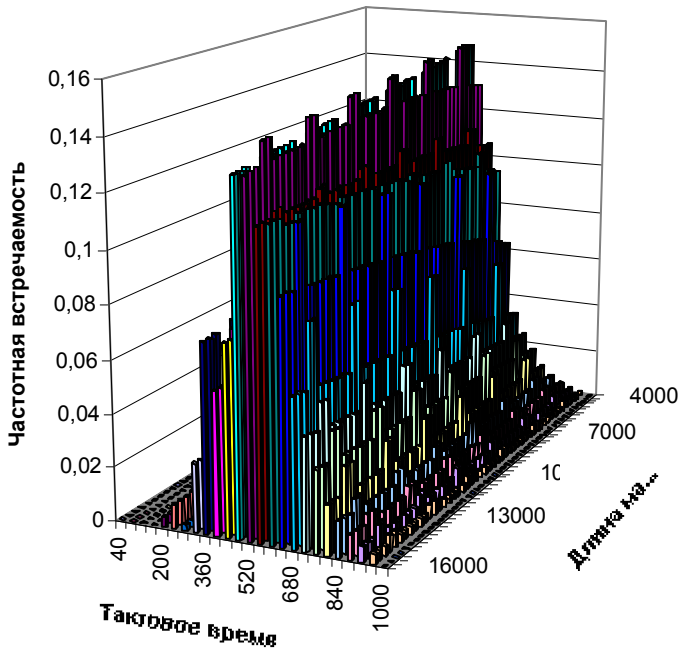


Рисунок 11.5. Частотная гистограмма значений тактового времени выполнения для программной реализации алгоритма комбинированного интерполяционного поиска.

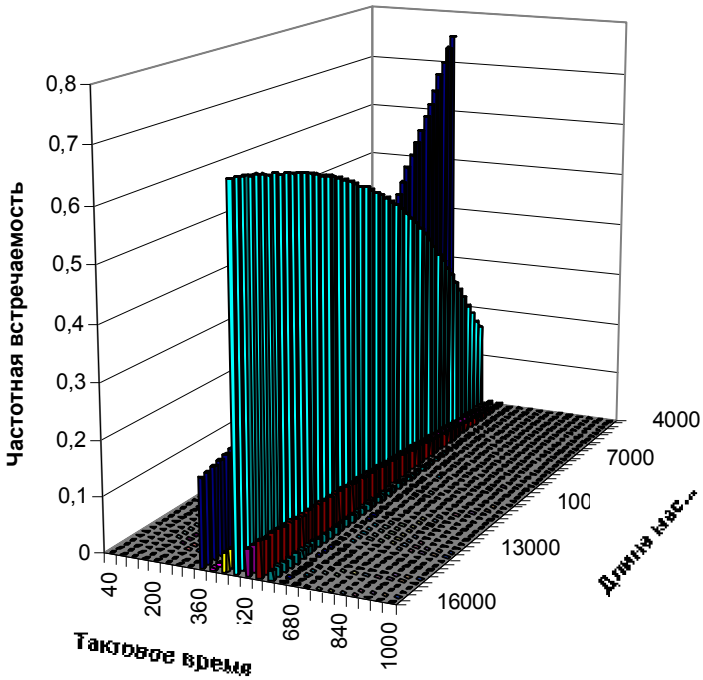


Рисунок 11.6. Частотная гистограмма значений тактового времени выполнения для программной реализации алгоритма поиска методом хеширования.

Сравнивая полученные результаты, следует отметить, что хотя алгоритм хеширования показывает лучшее среднее тактовое время, он имеет больший размах варьирования за счет попадания ключа поиска в конец длинных цепочек коллизий. Гистограммы на рисунках 11.5 и 11.6 показывают, что вид функций распределения для $T_n^{\text{такт}}(m)$ исследуемых алгоритмов различен, что приводит к различной временной устойчивости тактового времени поиска у этих алгоритмов. Заметим также, что при ограничении на максимальное время поиска в 2500 тактов, алгоритм интерполяционного поиска является по тренду рациональным на всем сегменте длин массива ключей.

Исследование алгоритмов поиска по информационной чувствительности тактового времени поиска. Использование характеристики информационной чувствительности позволяет принимать рациональные решения по выбору алгоритмов в ситуации, когда техническое задание накладывает ограничения на диапазон временной эффективности программной реализации алгоритма, т. е. на временную устойчивость. В этом случае менее информационно чувствительный алгоритм будет являться, при прочих равных, более предпочтительным. Количественная мера информационной чувствительности алгоритма A по определяется по формуле (см. глава 4):

$$\delta_i(n) = V_A(n) \cdot Q_A(n), \quad 0 \le \delta_i(n) \le 1, \tag{11.5.8}$$

где $V_A(n)$ — генеральный коэффициент вариации, который для тактового времени выполнения программной реализации алгоритма имеет вид:

$$V_A(n) = \sigma_{\text{такт}}(n) / \bar{t}_{\text{такт}}(n), \quad 0 \le V(n) \le 1, \tag{11.5.9}$$

а $Q_A(n)$ есть нормированный (относительный) размах варьирования ресурсной функции для входов длины n, определяемый как отношение половины вариантного интервала к его середине, и в данном случае определяется как:

$$Q_A(n) = \left(t^{\wedge}_{\text{такт}}(n) - t^{\vee}_{\text{такт}}(n)\right) / \left(t^{\wedge}_{\text{такт}}(n) + t^{\vee}_{\text{такт}}(n)\right). \tag{11.5.10}$$

Полученные значения $\delta_i(n)$ для исследуемых алгоритмов приведены в виде огибающих точечных значений на рисунке 11.7. На основе полученных экспериментальных данных и информационной чувствительности можно говорить о том,

что с учетом требования временной устойчивости алгоритм хеширования является более предпочтительным.

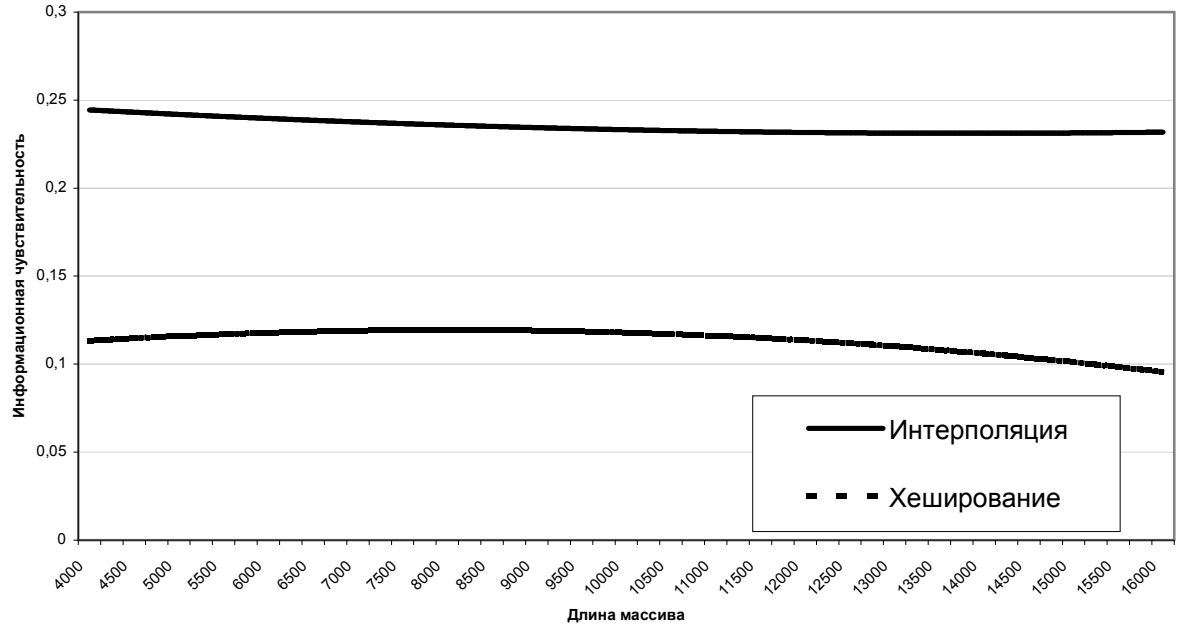


Рисунок 11.7. Зависимость информационной чувствительности от длины массива ключей для программных реализаций алгоритмов комбинированного интерполяционного поиска и поиска методом хеширования.

Выводы. Таким образом, на основе анализа частотного распределения тактового времени поиска, поведения размаха варьирования тактового времени, и информационной чувствительности претендующих алгоритмов на конечном сегменте длин ключей, могут быть сформулированы следующие рекомендации:

— при оценке качества алгоритмов по показателю среднего тактового времени рациональным на всем исследованном сегменте является алгоритм поиска на основе хеш-функции;

— при оценке качества алгоритмов по значению показателя информационной чувствительности предпочтение так же может быть отдано алгоритму поиска методом хеширования;

— при оценке качества алгоритмов по показателю тактового времени в худшем и лучшем случаях рациональным на всем исследованном сегменте является алгоритм комбинированного интерполяционного поиска.

Задачи и упражнения к главе 11

11.1. Разработайте алгоритм поиска не только первого, но и второго максимального и минимального элементов в массиве. За основу может быть взят алгоритм из параграфа 11.1. Может ли быть в этом случае применена идея сравнения последовательных пар элементов?

11.2. Более интересная задача, требующая серьезных алгоритмических размышлений — поиск k-ого максимума в массиве с линейной оценкой трудоемкости. Попробуйте самостоятельно разработать такой алгоритм, не обращаясь к специальной литературе.

11.3. Рассмотрим задачу организации обратного двоичного счетчика — при каждом обращении к нему происходит вычитание единицы из числа, хранящегося в битовом массиве. Разработайте алгоритм организации такого счетчика с не зависящей от длины битового массива трудоемкостью в среднем.

11.4. Модифицируйте алгоритм сортировки методом индексов таким образом, чтобы он работал и в случае, если в массиве есть повторяющиеся элементы.

11.5. Будет ли приведенный в параграфе 11.3 алгоритм сортировки работать правильно, если исходный массив содержит отрицательные числа?

11.6. Сравнение по трудоемкости не совсем справедливо для программных реализаций алгоритмов. Постройте, на основе экспериментальных исследований, функции времени выполнения для сравниваемых в параграфе 11.3 алгоритмов сортировки. Определите, на сколько изменились коэффициенты параболы, разграничивающей области рационального использования этих алгоритмов?

11.7. Если использовать реализованную на компьютере операцию умножения целых двухбайтовых чисел, то можно значительно улучшить трудоемкость алгоритма умножения длинных целых чисел. Разработайте соответствующую структуру данных и алгоритм умножения. Проанализируйте трудоемкость полученного Вами алгоритма. Насколько Вам удалось улучшить коэффициент у главного порядка функции трудоемкости, по сравнению с алгоритмом из параграфа 11.4, использующим структуру битового хранения длинного целого числа?

11.8. Приведите несколько примеров реальных ситуаций использования программных средств, в которых наиболее значимым критерием оценки качества

функционирования программной реализации алгоритма является выполнение ограничения на максимально допустимое время при фиксированной длине входа.

Список литературы к главе 11

[11.1] Грин Д., Кнут Д. Математические методы анализа алгоритмов. — М.: Мир, 1987. — 120 с.

[11.2] Кнут Д. Э. Искусство программирования, том 1. Основные алгоритмы, 3-е изд.: Пер. с англ. — М.: Издательский дом «Вильямс», 2002. — 720 с.

[11.3] Хаггарти Р. Дискретная математика для программистов. — М.: Техносфера, 2005. — 400с.

[11.4] Гудман С., Хидетниеми С. Ведение в разработку и анализ алгоритмов. — М.: Мир, 1981. — 368 с.

[11.5] Молдовян А. А. и др. Криптография: скоростные шифры. — СПб.: БХВ-Петербург, 2002. — 496 с.

[11.6] http:/www.ccas.ru

[11.7] Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. – М.: МЦНМО, 1999. – 960 с., 263 ил.

[11.8] Головешкин В.А., Ульянов М. В. Теория рекурсии для программистов. М.: Издательство «Наука ФИЗМАТЛИТ», 2006. — 296 с.

[11.9] Ахо А., Хопкрофт Дж., Ульман Дж. Структуры данных и алгоритмы: Пер. с англ.: — М.: Издательский дом «Вильямс», 2001. — 384 с.

[11.10] Брейман А. Д., Ульянов М. В. Выбор рациональных алгоритмов поиска по ключу для баз данных, размещаемых в оперативной памяти, на основе анализа их информационной чувствительности // Информационные технологии. 2006. № 1. С. 50–56.

[11.11] Weikum G., Wossen G. Transactional information systems. Theory, algorithms, and the practice of concurrency control and recovery. – Academic Press, 2002. – 852pp.

[11.12] Кнут Д.Э. Искусство программирования. Т.3. Сортировка и поиск. – М.:Вильямс, 2003. – 400с.

[11.13] Ульянов М. В. Классификация и методы сравнительного анализа вычислительных алгоритмов. Научное издание. — М.: Издательство физико-математической литературы, 2004. — 212 с.

Г Л А В А 1 2 .

РЕСУРСНО-ЭФФЕКТИВНЫЕ КОМБИНИРОВАННЫЕ АЛГОРИТМИЧЕСКИЕ РЕШЕНИЯ

Введение

Использование аппарата теории ресурсной эффективности компьютерных алгоритмов, в частности методов получения ресурсных функций в процедурной и рекурсивной реализации и методики их сравнительного анализа на конечном интервале, позволяет решать задачу выбора рациональных алгоритмов в области реальных длин входов при разработке программных средств и систем. Такой выбор может носить не только характер рекомендаций, но и позволяет построить комбинированные ресурсно-эффективные алгоритмы, в которых выбор того или иного алгоритма осуществляется на основе информации о длине входа и характеристических особенностях множества исходных данных. Цель настоящей главы — проиллюстрировать применение аппарата теории ресурсной эффективности компьютерных алгоритмов и соответствующих подходов для решения прикладных задач по выбору рациональных алгоритмов и построению комбинированных ресурсно-эффективных алгоритмических решений, как для модельных задач, так и для некоторых реальных программных систем, а именно — для системы конечно-элементного анализа, реализующей метод конечных элементов для решения обратных задач термоупругости, и для аналитического компонента индивидуальных информационных систем в части алгоритмов рациональной организации данных.

12.1 Комбинированный рекурсивно-итерационный алгоритм сортировки

Введение. Применение метода декомпозиции позволяет получить алгоритмы, обладающие хорошей временной эффективностью при больших длинах входов. Как правило, при этом платой за улучшение асимптотической оценки функции трудоемкости является большой коэффициент при главном порядке. Для целого ряда рекурсивных алгоритмов возможно построение комбинированной схемы, обладающей лучшей трудоемкостью за счет более раннего останова рекурсии и решения полученных подзадач в листьях дерева рекурсии другим алгоритмом,

обладающим лучшей трудоемкостью в полученном сегменте размерностей. Такой подход приводит к повышению трудоемкости вычислений в листьях дерева рекурсии, одновременно сокращая общее число его вершин, что в свою очередь приводит к необходимости решения задачи выбора оптимальных параметров подзадач для останова рекурсии. В данном параграфе этот подход применяется для улучшения трудоемкости классического алгоритма сортировки слиянием, путем его комбинирования с алгоритмом сортировки вставками.

Разработка рекурсивного алгоритма. Метод декомпозиции может быть применен для решения задачи сортировки, исторически эта идея приписывается Дж. фон Нейману. 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 алгоритм рекурсивно вызывается для сортировки полученных частей, и возвращенные отсортированные фрагменты массива объединяются при возврате из двух рекурсивных вызовов в один сортированный массив. В классической реализации останов рекурсии происходит при единичной длине массива.

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 результирующий 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 — в данном случае выбран способ «концевых заглушек» — в конец двух массивов записывается заведомо 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, необходимо переместить их вначале в два вспомогательных массива, т. к. результат должен быть помещен обратно в основной массив. Альтерна-

тивой является более трудоемкий алгоритм, реализующий слияние по месту, т. е. в основном массиве, и не требующий дополнительной памяти. Заметим, что проблема выбора между этими алгоритмами является иллюстрацией классической дилеммы выбора между временной и емкостной эффективностью.

Рассмотрим алгоритм слияния (**Merge**) отсортированных фрагментов массива **A**, расположенных в позициях между **p** и **r**, и **r+1** и **q**. Алгоритм использует дополнительные массивы **Bp** и **Bq**, в конец которых, с целью остановки указателей, помещаются заглушки. Вначале выполняется копирование отсортированных частей в **Bp** и **Bq**, а затем объединенный массив формируется непосредственно в массиве **A** между индексами **p** и **q**. Для удобства анализа в записи алгоритма справа указано количество базовых операций в данной строке.

```
Merge (A, p, r, q)
(формирование заглушки)
   max ← A[r]                                  2
   If Max < A[q]                               2
      then
         max ← A[q]                            2
   end If
   (копирование в массивы Bp, Bq)
   kp ← r—p+1                                  3
   p1 ← p—1                                    2
   For i ← 1 to kp                          1+3kp
      Bp[i] ← A[p1+i]                         4kp
   end For
   Bp[kp+1] ← max     (заглушка)               3
   kq ← q—r                                    2
   For i ← 1 to kq                          1+3kq
      Bq[i] ←A[r+i]                           4kq
   end For
   Bq[kq+1] ← max    (заглушка)               3kq

   (слияние частей)
   pp ← 1                                      1
   pq ← 1         (инициализация указателей)   1
   For i ← p to q                           1+3m
      If Bp[pp] < Bq[pq]                       3m
         then
            A[i] ← Bp[pp]                      3k
            pp ← pp +1                         2k
         else
            A[i] ← Bq[pq]                      3l
            pq ← pq +1                         2l
      end If
   end For
```

```
End.
```

Получим, следуя [12.1], 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 ← A[q]` выполняется в среднем для половины обращений, можно получить трудоемкость алгоритма слияния отсортированных массивов как функцию длины массива результата:

$$\overline{f}_{merge}(m) = 2+2+1+3+2+1+3\cdot kp+4\cdot kp+3+2+1+3\cdot kq+4\cdot kq+3+1+1+$$
$$+1+3\cdot m+m\cdot(3+5)=11\cdot m+7\cdot(kp+kq)+23=18\cdot m+23. \qquad (12.1.1)$$

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 ← 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_A1** получает на вход массив **A**, и два индекса **p** 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_A1 (A, p ,q)
   If p≠q  (проверка на останов рекурсии)      1
      then
         r←(p+q) div 2   (середина массива)    3
         MSort_A1 (A, p, r)   (сортировка левой части массива)
         MSort_A1 (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_A1(A,1,n)**. Принадлежность этого алгоритма к одному из трудоемкостных классов мы определим после его анализа.

Анализ трудоемкости алгоритма методом подсчета вершин дерева рекурсии. В соответствии с методом детального анализа дерева рекурсий выполним

подсчет вершин в дереве рекурсивных вызовов алгоритма сортировки слиянием. Рассмотрим случай, когда длина сортируемого массива есть степень двойки — $n = 2^k, k = \log_2 n$. В этом случае 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. Получаемое при этом дерево рекурсии представляет собой полное бинарное дерево, содержащее k уровней и n листьев. Дальнейший анализ этого алгоритма излагается в соответствии с [12.1].

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

$$R(n) = 1 + 2 + \cdots + 2^k = 2 \cdot 2^k - 1 = 2 \cdot n - 1.$$

Поскольку полное бинарное дерево содержит $n = 2^k$ листьев, то число листьев $R_L(n) = n$, а число 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

$$R(n) = 2 \cdot n - 1,\ R_V(n) = n - 1,\ R_L(n) = n,\ H_R(n) = 1 + \log_2 n,\ B_L(n) = \frac{n}{2 \cdot n - 1}. \quad (12.1.2)$$

В соответствии с методом подсчета вершин дерева рекурсии определим вначале трудоемкость процедуры **`MSort_A1(A,p,q)`** на один вызов/возврат — $f_R(1)$. Поскольку процедура **`MSort_A1(A,p,q)`** передает три параметра ($p = 3$), в стеке сохраняются значения четырех регистров ($r = 4$), ни одно значение не возвращается через имя процедуры ($f = 0$), — обращаем на это внимание, и процедура **`MSort_A1(A,p,q)`** имеет одну локальную переменную **`r`** ($l = 1$), то в результате имеем

$$f_R(1) = 2 \cdot (3 + 4 + 0 + 1 + 1) = 18,$$

и, следовательно, с учетом (12.1.2)

$$f_R(n) = R(n) \cdot f_R(1) = (2 \cdot n - 1) \cdot 18 = 36 \cdot n - 18. \quad (12.1.3)$$

Трудоемкость останова рекурсии включает в себя одно сравнение, таким образом $f_{CL}(1) = 1$, следовательно

$$f_{CL}(n) = R_L(n) \cdot f_{CL}(1) = n \cdot 1 = n. \qquad (12.1.4)$$

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 $f_{Msort}(v)$ — выполняется сравнение, вычисление середины длины, прибавление единицы (**`r+1`**), сохранение значения переменной **`r`** ($l = 1$), передача управления на процедуру **`Merge(A,p,r,q)`**, имеющую четыре параметра, и возврат управления обратно, таким образом

$$f_{Msort}(v) = 1 + 3 + 1 + 2 \cdot (4 + 4 + 0 + 1 + 1) = 25,$$

и сумма $f_{Msort}(v)$ по всем внутренним вершинам составляет

$$f_{CV\ Msort}(n) = R_V(n) \cdot f_{Msort}(v) = (n-1) \cdot 25 = 25 \cdot n - 25 \qquad (12.1.5)$$

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 (12.1.1) $18 \cdot m + 23$ 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

$$\begin{cases} g(v_1) = 18 \cdot n + 23; \\ g(v_2) = g(v_3) = 18 \cdot (n/2) + 23; \\ g(v_4) = g(v_5) = g(v_6) = g(v_7) = 18 \cdot (n/4) + 23; \\ \dots \end{cases}$$

Суммируя $g(v_j)$ по всем внутренним вершинам дерева, имеем

$$f_{CV\ Merge}(n) = \sum_{j=1}^{R_V(n)} g(v_j) =$$

$$= 23 \cdot R_V(n) + 18 \cdot n + 2 \cdot 18 \cdot (n/2) + 4 \cdot 18 \cdot (n/4) + \dots,$$

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 $k-1$, получаем

$$f_{CV\ Merge}(n) = 18 \cdot n \cdot k + 23 \cdot (n-1) = 18 \cdot n \cdot \log_2 n + 23 \cdot n - 23. \qquad (12.1.6)$$

Учитывая все компоненты (см. формулы 12.1.3 – 12.1.6), получаем окончательный вид функции трудоемкости для алгоритма сортировки слиянием в случае $n = 2^k$

$$\overline{f}_A(n) = f_R(n) + f_{CL}(n) + f_{CV\ Msort}(n) + f_{CV\ Merge}(n) =$$

$$= 36 \cdot n - 18 + n + 25 \cdot n - 25 + 18 \cdot n \cdot \log_2 n + 23 \cdot n - 23 =$$

$$= 18 \cdot n \cdot \log_2 n + 85 \cdot n - 66. \qquad (12.1.7)$$

и доля операций обслуживания дерева рекурсии составляет

$$F_R(n) = \frac{f_R(n)}{f_A(n)} = \frac{36 \cdot n - 18}{18 \cdot n \cdot \log_2 n + 85 \cdot n - 66}. \qquad (12.1.8)$$

Значения $F_R(n)$ медленно убывают с ростом длины массива, так при $n = 1024$ $F_R(n) \approx 0{,}135$, а при $n = 1048576$ $F_R(n) \approx 0{,}08$, и доля трудоемкости обслуживания рекурсии для реальных размерностей уже находится в пределах 8-15%.

Полученная функция трудоемкости в среднем (12.1.7), очевидно согласуется с результатом оценки главного порядка $\Theta(n \cdot \log_2 n)$, полученным по теореме Бентли, Хакен, Сакса. На основе этого результата можно говорить, что алгоритм относится к классу πP, т. е. является полиномиальным по длине входа. Заметим, что асимптотически он лучше, чем простые квадратичные алгоритмы сортировки.

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

$$f_A^{\vee}(n) = \overline{f}_A(n) - 1 \cdot R_V(n) = 18 \cdot n \cdot \log_2 n + 84 \cdot n - 67.$$

$$f_A^{\wedge}(n) = \overline{f}_A(n) + 1 \cdot R_V(n) = 18 \cdot n \cdot \log_2 n + 86 \cdot n - 65.$$

Полученные результаты позволяют говорить об очень слабой зависимости трудоемкости от данных, алгоритм принадлежит к классу $NPRL$, и обладает очень хорошей временной устойчивостью.

Комбинированный алгоритм сортировки. Рассмотрим вариант останова рекурсии в алгоритме **MSort_A1** при некоторой длине массива, большей единицы. Полученные подмассивы в листьях дерева рекурсии будем сортировать методом прямого включения, используя алгоритм **Sort_Ins_A2** (см. параграф 11.3). Поскольку функция трудоемкости этого алгоритма имеет вид (см. формулу 11.3.1):

$$\overline{f}_{A2}(n) = 2{,}5 \cdot n^2 + 11{,}5 \cdot n - 13 = \Theta(n^2),$$

то за счет малого коэффициента при главном порядке на малых длинах массивов он является более рациональным, чем алгоритм сортировки слиянием. В предположении, что ячейка с именем **k** хранит значение оптимального порога переключения на сортировку вставками, запись такого комбинированного алгоритма имеет следующий вид:

```
MSort_A3 (A, p ,q)
   l=q-p+1
   If l>k  (проверка на останов рекурсии)
      then
         r←(p+q) div 2          (середина массива)
         MSort_A3 (A, p, r)     (сортировка левой части массива)
         MSort_A3 (A, r+1, q)   (сортировка правой части массива)
         Merge(A, p, r, q)      (слияние отсортированных частей)
      else
         (сортировка вставками для коротких длин)
         Sort_Ins_A2(A, p, q)
Return (A)
End.
```

Определение рационального порога переключения. Поскольку наша цель состоит в минимизации трудоемкости комбинированного алгоритма, то в первую очередь необходимо получить его функцию трудоемкости, как функцию от исходной длины массива и значения порога переключения. Сразу оговоримся,

что будет получена некоторая оценка порога переключения, поскольку делается допущение о полном бинарном дереве рекурсии и допущение о непрерывности аргументов функции трудоемкости с целью последующего дифференцирования.

Пусть k есть значение порога переключения, и в силу допущений предполагается, что весь исходный массив длины n разбивается на $\frac{n}{k}$ подмассивов длиной k каждый, которые и сортируются алгоритмом прямого включения в листьях дерева рекурсии. На основе формул трудоемкости двух исходных алгоритмов получаем

$$\overline{f}_{A3}(n,k) = \overline{f}_{A1}(n,k) + \overline{f}_{A2}(n,k) =$$

$$= (18\,n \log_2 \frac{n}{k} + 85\,\frac{n}{k} - 66) +$$

$$+ \frac{n}{k}(2{,}5k^2 + 11{,}5k - 13),$$

что после несложных преобразований приводит к

$$\overline{f}_{A3}(n,k) = (18\,n \log_2 n - 18\,n \log_2 k + 85\,\frac{n}{k} - 66) +$$

$$+ (2{,}5nk + 11{,}5n - 13\,\frac{n}{k}). \qquad (12.1.9)$$

Оптимальное значение длины массива для переключения на сортировку методом прямого включения можно получить, приравняв нулю частную производную полученной функции трудоёмкости

$$\frac{\partial \overline{f}_{A3}(n,k)}{\partial k} = 2{,}5n - \frac{72n}{k^2} - \frac{18n}{k \cdot \ln 2} = 0,$$

что приводит к следующему квадратному уравнению

$$2{,}5k^2 - 25{,}96k - 72 = 0,$$

положительный корень которого равен 12,66. Поскольку значение k является целым числом, то рациональный порог переключения равен 12 или 13.

Хотя мы получили оценочный результат, тем не менее, проведенные эксперименты подтверждают, что оптимальный порог переключения для массивов длиной $n > 1000$ действительно составляет 12 или 13 в зависимости от конкретной длины массива.

Подстановка значения $k = 12$ в формулу (12.1.9) позволяет получить функцию трудоемкости комбинированного алгоритма

$$\overline{f}_{A3}(n) = 18\, n \log_2 n - 17\, n - 66\ . \qquad (12.1.10)$$

Сравнивая трудоемкости исходного и комбинированного алгоритмов (12.1.7) и (12.1.10) можно оценить полученное сокращение трудоемкости

$$\overline{\Delta f}_A(n) = 102\ n\ .$$

Экспериментальные результаты показывают, что на сегменте длин исходных массивов [100, 3000] комбинированный алгоритм обладает почти что вдвое лучшей трудоемкостью по сравнению с исходным алгоритмом сортировки слиянием, а именно отношение трудоёмкостей изменяется от 2,0 для длины 100 до 1,54 для длины 3000.

12.2 Эффективный по трудоемкости комбинированный алгоритм решения классической задачи одномерной упаковки

Введение. Изложение в этом параграфе применения метода динамического программирования для решения задачи одномерной оптимальной по стоимости упаковки основано на классической книге Р. Беллмана и С. Дрейфуса [12.2]. Два алгоритма, реализующие этот метод — табличный и рекурсивный имеют различные ресурсные характеристики, что позволяет предложить комбинированный алгоритм решения задачи упаковки, в котором рациональный порог переключения алгоритмов зависит от особенностей входных данных.

Содержательная постановка задачи упаковки. Представим себе, что у нас есть рюкзак с прямоугольным дном и нерастяжимыми стенками определенной высоты. У нас есть так же 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, что бы 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 интуитив-

ное 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 будет получена оценка 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 раскроя материала, формирования оптимального пакета акций на фиксированную сумму. На основе полученных оптимальных решений при значительном количестве групп коробок можно даже построить достаточно надежную криптосистему.

Математическая постановка задачи. Рассмотрим общую постановку задачи одномерной оптимальной по стоимости упаковки. Пусть задано множество типов грузов

$$Y = \{ y_i \}, y_i = \{ v_i, c_i \}, i = \overline{1,n},$$

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. Так же 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:

$$P_n(\mathbf{x}) = \sum_{i=1}^{n} C_i(x_i) = \sum_{i=1}^{n} x_i \cdot c_i \to \max, \quad \sum_{i=1}^{n} x_i \cdot v_i \leq V. \qquad (12.2.1)$$

Содержательно ограничение в (12.2.1) 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 (12.2.1), 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 один, по крайней мере, 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-ти 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

$$\prod_{i=1}^{n}(\lfloor V/v_i \rfloor + 1) \le (m+1)^n, \quad m = \max_{i=1,n}\{ V/v_i \}.$$

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, будем считать, что в рассматриваемой задаче распределяемым ресурсом является объем упаковки V, а функции дохода линейны — $g_i(x_i) = c_i \cdot x_i$. Наша задача — максимизировать доход (стоимость упаковки), заданный линейным функционалом $P_n(\mathbf{x})$ (см. 12.2.1), 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

$$\begin{cases} f_0(v) = 0; \\ f_m(v) = \max\limits_{x_m} \{ x_m \cdot c_m + f_{m-1}(v - x_m \cdot v_m) \}, \; m = \overline{1,n}, \; v = \overline{0,V}, \; x_m = 0,1,\ldots,\left\lfloor \dfrac{v}{v_m} \right\rfloor. \end{cases} \quad (12.2.2)$$

Таким образом, метод предполагает последовательное решение одномерных задач целочисленной оптимизации с использованием информации об оптимальной упаковке объема v предыдущими типами грузов. Решением поставленной задачи яв-

ляется значение $f_n(V)$. Поскольку значения $f_1(v)$ могут быть элементарно вычислены на основе (12.2.2), то в дальнейшем будет рассматриваться следующее основное функциональное уравнение для задачи одномерной оптимальной упаковки, записанное в виде рекуррентного соотношения, определяющего рекурсивно заданную функцию $f_m(v)$ [12.1]

$$\begin{cases} f_1(v) = \left\lfloor \dfrac{v}{v_k} \right\rfloor \cdot c_1 \,, m = 1; \\ f_m(v) = \max\limits_{x_m} \{ x_m \cdot c_m + f_{m-1}(v - x_m \cdot v_m) \}, m \geq 2, x_m = 0,1,\ldots, \left\lfloor \dfrac{v}{v_m} \right\rfloor . \end{cases} \tag{12.2.3}$$

Схема алгоритма. Рекурсивная реализация основного функционального уравнения Беллмана (12.2.3) для задачи одномерной упаковки (12.2.1) достаточно проста. Рекурсивный алгоритм, напрямую реализующий рекуррентные соотношения (12.2.3), останавливается при значении $m = 1$, когда значение функции $f_1(v)$ может быть вычислено непосредственно. Рекурсия при $m \geq 2$, выполняется для вычисления оптимальной стоимости упаковки текущего объема v грузами типа m совместно с грузами предыдущих $m-1$ типов в том же или меньшем объеме. Ниже приводится схема данного алгоритма в виде рекурсивной функции **A1(V,x)** в которой пока не детализируется структура данных для хранения вектора упаковки.

```
A1(V, x)
   (V - текущий объем упаковки)
   (x - номер текущего типа груза)
   (boxV[1..n], boxC[1..n] - массивы исходных данных)
   If x=1
      then  (Останов рекурсии - прямое вычисление при x=1)
         k ← V div boxV[1]
         F ← k*boxC[1] (оптимальная стоимость в V для 1-ого типа)
      else  (Рекурсивные вызовы для определения max F)
         Max ← 0;
         k ← V div boxV[x]
         For i ← 0 to k (цикл поиска максимума)
            Cost ← i*boxC[x]+A1((V - boxV[x]*i),x-1)
            If Cost > Max
               then
                  Max ← Cost
                  Optx ← i
         end (for i)
         A1 ← Max (функция возвращает оптимальную стоимость)
                  (Количество грузов типа x в объеме V равно Optx)
```

```
   end (If)
Return (A1)
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 таблице 12.1.

Таблица 12.1. 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 рисунке 12.1

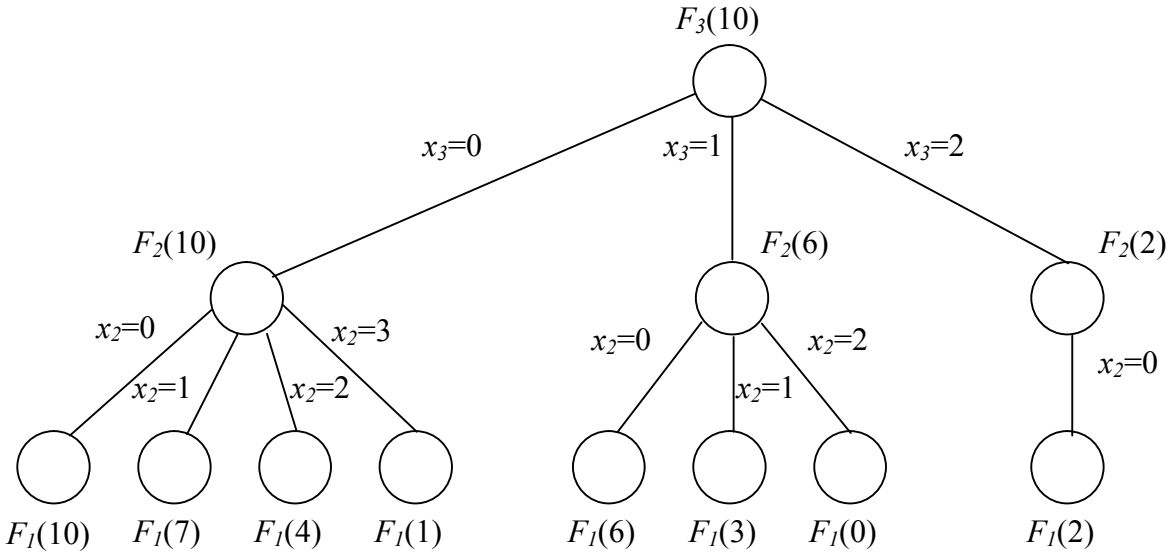


Рисунок 12.1 Дерево рекурсии, порождаемое алгоритмом упаковки.

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 (12.2.3) очевидно, что оценка вычислительной сложности будет зависеть как от значения n, так и от значений параметров $V, v_1, \cdots, v_n$, учет которых существенно затрудняет анализ. С целью упрощения анализа рекурсивного алгоритма, следуя [12.1] введем параметр

$$k = \frac{V}{\bar{v}},\ \bar{v} = \frac{1}{n} \cdot \sum_{i=1}^{n} v_i,$$

характеризующий, 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, а 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 **A1(V,x)** (справа указано количество базовых операций в строке).

```
A1(V, x)
   Рекурсивный алгоритм решения задачи упаковки
   (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           1
      then  (Останов рекурсии)
         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)
            (Обнуление оптимального вектора)
         For j ← 0 to n                    1+(n+1)*3
            XArr[x,j] ← 0                    3*(n+1)
```

```
        k ← V div boxV[x]                          3
           (цикл поиска максимума)
        For i ← 0 to k                      1+(k+1)*3
           A1((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 i)
   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 рисунках 12.2 и 12.3.

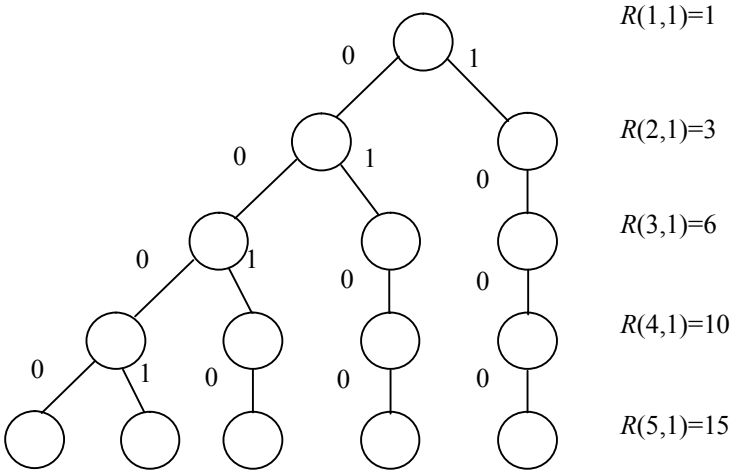


Рисунок 12.2. Фрагмент дерева рекурсии при $k=1$.

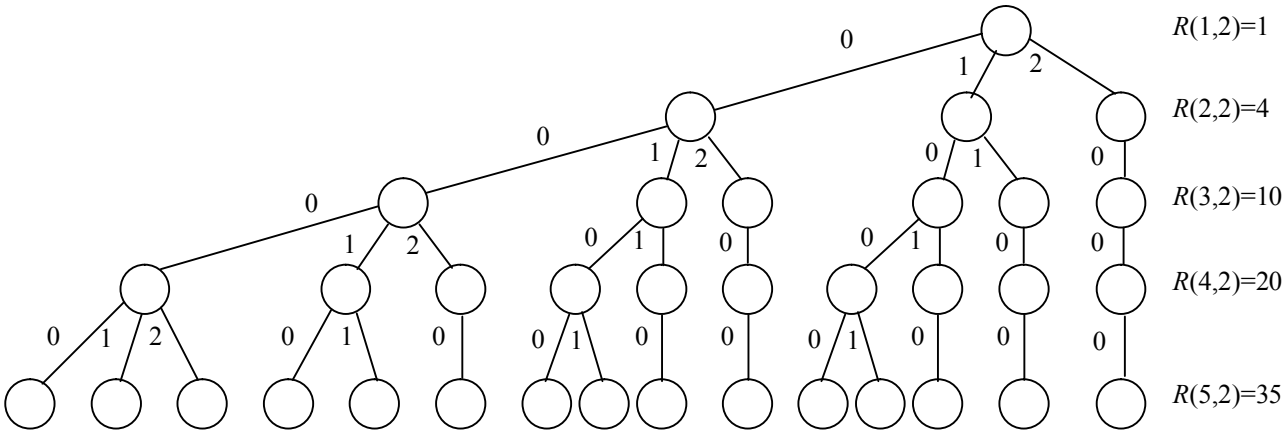


Рисунок 12.3. Фрагмент дерева рекурсии при $k = 2$.

Для того чтобы найти закономерность для значений $R(n,k)$ построим треугольник Паскаля, первые семь строк которого показаны на рисунке 12.4.

***m*=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

Рисунок 12.4. Треугольник Паскаля — биномиальные коэффициенты.

Теперь видно, что значения $R(n,k)$ это элементы треугольника Паскаля — биномиальные коэффициенты. Устанавливая соответствие между номерами строк треугольника и значениями n и k, окончательно получаем

$$R(n,k) = C_{n+k}^{n-1}. \qquad (12.2.4)$$

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, в целях удобства дальнейшего анализа, приведем также коэффициенты для их сведения к $R(n,k)$

$$R_L(n,k) = C_{n+k-1}^{n-1} = R(n,k)\cdot\frac{k+1}{n+k},\ R_V(n,k) = C_{n+k}^{n-2} = R(n,k)\cdot\frac{n-1}{n+k} \qquad (12.2.5)$$

Однако, 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 (12.2.4) и (12.2.5) можно получить все формулы для характеристик дерева рекурсии, порождаемого алгоритмом Беллмана в рамках принятой параметризации

$$R(n,k)=C_{n+k}^{n-1},\ \ R_L(n,k)=C_{n+k-1}^{n-1},\ \ R_V(n,k)=C_{n+k}^{n-2},$$

$$H_R(n)=n,\ \ B_L(n,k)=\frac{k+1}{n+k} \qquad (12.2.6)$$

Временная эффективность — функция трудоемкости. Трудоемкость данного алгоритма получим, 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 **`A1(V,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

$$f_R(1)=2\cdot(2+4+0+4+1)=22,$$

и, следовательно,

$$f_R(n,k)=22\cdot R(n,k)=22\cdot C_{n+k}^{n-1}. \qquad (12.2.7)$$

Трудоемкость останова рекурсии включает в себя одно сравнение для входа в блок останова, и $6\cdot n+6$ операций формирования вектора, таким образом, $f_{CL}(1)=6\cdot n+7$, следовательно

$$f_{CL}(n,k)=R_L(n,k)\cdot f_{CL}(1)=(6\cdot n+7)\cdot C_{n+k-1}^{n-1}.$$

В целях удобства получения общей формулы введем обозначения для двух коэффициентов, показывающих долю внутренних вершин и листьев в общем количестве вершин дерева рекурсии

$$R_L(n,k)=\alpha_L\cdot R(n,k),\ \alpha_L=\frac{k+1}{n+k},\ R_V(n,k)=\alpha_V\cdot R(n,k),\ \alpha_V=\frac{n-1}{n+k} \qquad (12.2.8)$$

заметим, что $\alpha_L+\alpha_V=1$, таким образом,

$$f_{CL}(n,k)=(6\cdot n+7)\cdot\alpha_L\cdot R(n,k). \qquad (12.2.9)$$

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

$$f_{CVvec}(n,k) = R_V(n,k) \cdot f_{CVvec}(v) = (6 \cdot n + 12) \cdot \alpha_V \cdot R(n,k). \qquad (12.2.10)$$

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 — $17 \cdot (R(n,k) - 1)$.

Трудоемкость копирования оптимального вектора внутри цикла поиска максимума составляет $9 \cdot n + 7$ операций. Однако количество переприсваиваний в блоке **`then`** определяется данными, даже при фиксированном параметре k. Снова будет использован метод амортизационного анализа, т. к. можно указать общее количество выполнения 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 упаковка первого типа груза является оптимальной, 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 введем специальный коэффициент относительно общего количества вершин дерева, значения которого определяются приведенными выше рассуждениями

$$\alpha_f^{\wedge} = 1,\ \alpha_f^{\vee} = \alpha_V = \frac{n-1}{n+k},\ \bar{\alpha}_f = \frac{1+\alpha_V}{2} = 1 - \frac{k+1}{2 \cdot (n+k)}. \qquad (12.2.11)$$

Теперь можно записать общую формулу для лучшего, худшего и среднего случаев $f_{CVfor}(n,k)$, используя введенный коэффициент

$$f_{CVfor}(n,k)=17\cdot(R(n,k)-1)+(9\cdot n+7)\cdot\alpha_f\cdot R(n,k). \qquad (12.2.12)$$

Объединяя все компоненты в соответствии с методом подсчета вершин дерева рекурсии (12.2.7 — 12.2.12), 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 (12.2.11)

$$f_{A1}(n,k)=\left(6\cdot n+9\cdot n\cdot\alpha_f\right)\cdot R(n,k)+\left(46+5\cdot\alpha_V+7\cdot\alpha_f\right)\cdot R(n,k)-17. \qquad (12.2.13)$$

и доля операций обслуживания дерева рекурсии составляет

$$F_R(n,k)=\frac{f_R(n,k)}{f_A(n,k)}\approx\frac{22}{6\cdot n+9\cdot n\cdot\alpha_f+46+5\cdot\alpha_V+7\cdot\alpha_f},\ F_R(n,k)\to 0,\ n\to\infty.$$

Для реальных входов значение параметра k не обязательно одинаково для всех типов грузов, и его значение является действительным числом. В этом случае, как показывают проведенные исследования [12.3], мы можем перейти от классической формулы для количества сочетаний, к формуле, использующей гамма функцию Эйлера — $\Gamma(s)$, как вещественному аналогу факториала

$$R(n,k)=\frac{\Gamma(n+k+1)}{\Gamma(n)\cdot\Gamma(k+2)}.$$

Емкостная эффективность — функция объема памяти. Для получения оценки емкостной эффективности в области памяти стека нам нужна функция $H_R(n)$, поскольку

$$V_{st}(m)=H_R(m)\cdot(p+r+f+l+1),$$

а алгоритм Беллмана требует четыре локальные ячейки, то

$$V_{st}(n)=H_R(n)\cdot(2+4++0+4+1)=11\cdot H_R(n)=11\cdot n,$$

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

$$V(n)=V_{ram}(n)+V_{st}(n)=n^2+3\cdot n+11\cdot n=n^2+14\cdot n.$$

На основе полученных результатов мы можем определить ресурсную сложность алгоритма при введенной параметризации задачи

$$\Re_c(n,k) = \left\langle \Theta\left(n \cdot C_{n+k}^{n-1}\right), \Theta\left(n^2\right)\right\rangle,$$

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

$$C_{n+k}^{n-1} = C_{n+k}^{k+1}, \quad C_{n+k}^{k+1} = \Theta\left((n+k)^{k+1}\right), k << n,$$

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, подробнее см в [12.3].

Табличный алгоритм решения задачи упаковки. Рассмотрим другой вариант точного решения этой задачи методом динамического программирования — табличный алгоритм, позволяющий получить набор оптимальных решений не только для заданного объема V, но и для всех промежуточных дискретных значений этого объема. Обозначим вектор оптимальной упаковки для объема v эле-

ментами y_j, $i=\overline{1,m}$, $m \le n$, через $\mathbf{x}_v^m$, а через $f_m(v)$ стоимость оптимальной упаковки в этом объеме. В силу принципа оптимальности [12.2]

$$f_m(v) = \max_{x_m} P_m(\mathbf{x}),$$

а порядок предъявления элементов y_i не является существенным. Решение табличным алгоритмом осуществляется на основе функционального уравнения метода динамического программирования. Результатом решения задачи является набор оптимальных значений целевой функции $f_n(v)$ и соответствующих векторов оптимальной упаковки $\mathbf{x}_v^n$ для всех объемов v от 0 до V исходными элементами (грузами различных типов) из множества Y. Отметим, что поскольку табличный способ позволяет получить оптимальные решения для всех промежуточных объемов упаковки, то эту информацию можно использовать, например, для исследования чувствительности целевой функции $f_n(v)$ по изменению объема упаковки.

Рассмотрим пример решения задачи одномерной упаковки с использованием табличного алгоритма для исходных данных из таблицы 12.1. Табличная реализация основного функционального уравнения Беллмана приводит к последовательному рассмотрению типов грузов, для каждого из которых мы получаем оптимальную упаковку для всех дискретов объема от 1 до 10, приведенную в таблице 12.2. Заметим, что рекурсивный вызов в функциональном уравнении (12.2.3) в табличном алгоритме есть не что иное, как обращение к предыдущей таблице оптимальной упаковки для определенного значения объема.

Таблица 12.2. Таблица оптимальной упаковки.

v	x_1	$f_1(v)$
1	0	0
2	1	3
3	1	3
4	2	6
5	2	6
6	3	9
7	3	9
8	4	12
9	4	12
10	5	15

$\Rightarrow$

v	x_1	x_2	$f_2(v)$
1	0	0	0
2	1	0	3
3	0	1	5
4	2	0	6
5	1	1	8
6	0	2	10
7	2	1	11
8	1	2	13
9	0	3	15
10	2	2	16

$\Rightarrow$

v	x_1	x_2	x_3	$f_3(v)$
1	0	0	0	0
2	1	0	0	3
3	0	1	0	5
4	0	0	1	7
5	1	1	0	8
6	0	2	0	10
7	0	1	1	12
8	0	0	2	14
9	0	3	0	15
10	?	?	?	?

Обратите внимание на то, как изменяется состав грузов, оптимально заполняющих объем, при изменении объема упаковки на единицу — такая чувствительность оптимального решения характерна для целого ряда задач целочисленного программирования.

Запись табличного алгоритма решения задачи упаковки в принятом алгоритмическом базисе имеет следующий вид.

```
A2(boxC, boxV, n, V; f,x)
Табличный алгоритм для задачи упаковкиИнициализация
   (boxV[1..n] - объемы типов грузов)
   (boxC[1..n] – стоимости типов грузов)
   (n - количество типов грузов)
   (V - объем упаковки)
   (f[0..V], f1[0..V] – функция Беллмана – оптимальная стоимость
    упаковки для текущего и предыдущего типов грузов)
   (x[0..V, 1..n],x1[0..V, 1..n] – массивы векторов оптимальной
    упаковки для текущего и предыдущего типов грузов)
Формирование таблицы для грузов типа 1
   for v ← 0 to V (по всем дискретам объема)
      begin
         x[v, 1] ← v div boxV[1] (число грузов типа 1 в объеме i)
         x1[v, 1] ← x[v, 1]
         f[v] ← x[v, 1] * boxC[1]
         f1[v] ← f[v]
      end
Основная часть алгоритма
for i ← 2 to N (цикл по типам грузов)
   begin
      Vi ← boxV[i]
      Ci ← boxC[i]
      for v ← 0 to V (цикл по дискретам объема упаковки)
         begin
            maxKol ← 0;
            maxC ← f[v]
            z ← v/Vi (z - максимальное количество груза
                      типа i в объеме v)
            for k ← 0 to z (цикл нахождения максимума f)
               begin
                  C ← Ci*k+f1[v-k*Vi]
                  if maxC < C
                     then
                        maxC = C
                        maxKol = k
               end (конец цикла по k)
            (сохранение оптимального решения
             и формирование оптимального вектора упаковки)
            f[v] ← maxC
            jr ← v-maxKol*Vi
            for j=1 to i-1
```

```
                begin
                   x[v, j] = x1[v,jr]
                end
             x[v, i]=maxKol
          end (конец цикла по дискретам объема)
   Переход к новому типу груза
      (копирование массива векторов оптимальной упаковки
       и массива оптимальных стоимостей
       упаковок всех дискретов объема грузами i типов)
      for v=0 to V
         for j=1 to i
            x1[v,j] ← x[v, j]
         end (end for j)
         f1[v] ← f[v]
      end (end for v)
   end (конец цикла по типам грузов)
End.
```

Оценка вычислительной сложности табличного алгоритма. Оценим вычислительную сложность в среднем для основной части алгоритма в условиях принятой параметризации. Мы считаем, что все грузы имеют одинаковый объем, равный среднему значению, и, следовательно, не более k грузов каждого типа может быть размещено в объеме V.

Заметим, что количество операций в цикле нахождения максимума целевого функционала имеет порядок $\Theta(1)$, равно как и количество операций, задаваемых собственно циклом по дискрету объема. Цикл по нахождению максимума целевого функционала зависит от цикла по дискретам объема, и их совместное рассмотрение приводит к следующим результатам, которые сведены в таблицу 12.3.

Таблица 12.3. Анализ трудоемкости циклов в табличном алгоритме упаковки

Значения счетчика цикла по дискретам объема	Количество проходов цикла вычисления максимума
от 0 до $\bar{v}-1$	0
от $\bar{v}$ до $2\cdot\bar{v}-1$	1
…	…
от $(k-1)\cdot\bar{v}$ до $k\cdot\bar{v}-1$	$k-1$
при $k\cdot\bar{v}=V$	k

Таким образом, совокупное количество проходов внешнего (по v) и внутреннего (по z) циклов равно

$$V\cdot\Theta(1)+\left(1\cdot\bar{v}+2\cdot\bar{v}+\cdots+(k-1)\cdot\bar{v}+k\right)\cdot\Theta(1)=$$

$$=\Theta(1)\cdot\left(V+k+\bar{v}\cdot\sum_{i=1}^{k-1}i\right)=\Theta(1)\cdot\left(V+k+\bar{v}\cdot\frac{k\cdot(k-1)}{2}\right). \qquad (12.2.14)$$

Поскольку в силу введенной параметризации $k=V/\bar{v}$, то подстановка в (12.2.14) дает следующую оценку для двух внутренних циклов

$$\left(\frac{V^2}{2\bar{v}}+\frac{V}{2}+\frac{V}{\bar{v}}\right)\cdot\Theta(1),$$

и учитывая, что цикл по типам грузов выполняется $n-1$ раз, окончательно получаем вычислительную сложность основной части табличного алгоритма (без учета трудоемкости фрагмента копирования при переходе к следующему типу груза) в виде

$$\Theta\left(\frac{nV^2}{2\cdot\bar{v}}\right)=\Theta\left(\frac{nVk}{2}\right).$$

Функция трудоемкости табличного алгоритма. Для табличного алгоритма в строках его записи не приведены значения числа базовых операций, которые читатель может без труда определить самостоятельно. С учетом проведенного анализа вычислительной сложности и очевидной вложенности других циклов табличного алгоритма можно получить функцию его трудоемкости. Проверка результата, полученного автором, может стать полезным упражнением для читателей. Результат имеет следующий вид.

$$f_{A2}(n,V,k)=8n^2V+6nVk+8n^2+45nV+12nk-6Vk+32n-31V-12k-16. \qquad (12.2.15)$$

Отметим, что в отличие от рекурсивного алгоритма, трудоемкость табличного алгоритма зависит также и от значения общего объема упаковки.

Сравнительный анализ и идея построения комбинированного алгоритма. Что лучше использовать — табличный или рекурсивный алгоритм? Если основным критерием выбора рационального алгоритма является его трудоемкость, то при заданных значениях n,V,k необходимо вычислить значения по формулам (12.2.13) и (12.2.15) и сравнить их. Но, при определенных условиях, зависящих от исходных данных, возможно построение комбинированного алгоритма, основной идеей которого является предвычисление оптимальных упаковок для некоторого

числа грузов табличным алгоритмом и дальнейшее решение задачи рекурсивным алгоритмом с сокращенным числом типов грузов. Какова рациональная граница (по числу типов грузов) для применения табличного алгоритма? Ответ на этот вопрос можно получить на основе следующих рассуждений.

Предположим, что табличным алгоритмом находится оптимальное решение для упаковки m типами грузов, при этом для остальных $n-m$ типов задача решается рекурсивным алгоритмом. Поскольку исходные данные известны, то на основе формул (12.2.13) и (12.2.15) можно построить функцию $g(m)$, значением которой является совокупная трудоемкость комбинированного алгоритма, при этом предполагается, что значения n,V,k известны и фиксированы

$$g(m)=f_{A1}(n-m,k)+f_{A2}(m,V,k).$$

Тогда оптимальное значение m^* может быть найдено как

$$m^*=\arg\min_{0\le m\le n} g(m). \qquad (12.2.16)$$

Поскольку значение m является целым числом, то m^* может быть найдено простым вычислением значений функции $g(m)$, при этом возможны три следующих случая, возникновение которых существенно определяется значениями исходных данных — n,V,k:

1. Случай 1 — $m^*=0$ или $m^*=1$. В этом случае рекурсивный алгоритм, будучи применен к исходной задаче упаковки, обеспечивает наилучшую трудоемкость, и комбинации с табличным алгоритмом не требуется.

2. Случай 2 — $m^*=n$. В этом случае табличный алгоритм, будучи применен к исходной задаче упаковки, обеспечивает наилучшую трудоемкость, и комбинации с рекурсивным алгоритмом не требуется.

3. Случай 3 — $2\le m^*\le n-1$. В этом случае рациональным является применение комбинированного алгоритма с порогом переключения равным m^*.

Совокупно программная реализация будет содержать табличный, рекурсивный и комбинированный алгоритмы и некоторый управляющий фрагмент, который по значениям параметров входа — n,V,k решаемой задачи определяет значение m^* по формуле (12.2.16) и, на этой основе, наиболее рациональны алгоритм для текущего входа.

12.3 Модификация алгоритма метода ветвей и границ для задачи коммивояжера на основе рационального использования доступного объема памяти

Введение. Повышение временной эффективности некоторых алгоритмов может быть достигнуто за счет использования дополнительного объема памяти. Такая ситуация возникает, в частности, и в случае, когда разработчики алгоритмического обеспечения сталкиваются с дилеммой — повторные вычисления или хранение промежуточных результатов. Хорошим примером для модификации с целью улучшения трудоемкости за счет эффективного использования дополнительной памяти является классический алгоритм Дж. Литла, К. Мерти, Д. Суини и К. Кэрол для точного решения задачи коммивояжера методом ветвей и границ [12.4], который и лежит в основе материала данного параграфа.

Содержательная постановка задачи. Коммивояжер — это сотрудник торговой организации, задача которого состоит в том, чтобы посетить ряд городов с целью рекламы и продажи товаров. Предполагается, что коммивояжер живет в стране с развитой транспортной сетью, и между каждой парой городов существует собственное транспортное сообщение. Будем называть туром порядок посещения всех городов, и потребуем, чтобы каждый город был посещен только один раз — это так называемая задача коммивояжера без возвратов. Стоимости проезда между городами известны, причем, в общем случае, стоимость проезда туда и обратно различны — это несимметричная постановка задачи коммивояжера.

Собственно сама задача состоит в том, чтобы найти такой тур коммивояжера, который имел бы минимальную стоимость. Очевидно, что число туров — конечно, и мы можем решить задачу прямым перебором. Оценим объем такого перебора в задаче с n городами. Поскольку тур — это замкнутый цикл, то в качестве начального пункта можно выбрать любой город. Отправляясь из него, у нас есть $n-1$ возможных вариантов, и мы выбираем один их них. В следующем городе таких вариантов будет $n-2$, т. к. возврат обратно запрещен. Поскольку выбор является независимым, то полное множество будет содержать $(n-1)!$ туров, и переборный алгоритм решения становится для реальных размерностей задачи совершенно неприемлемым по времени.

Постановка в терминах теории графов. Если ассоциировать города с вершинами графа, а пути сообщения и стоимости проезда с нагруженными ребрами, то получается полный ориентированный асимметричный граф без собственных петель на n вершинах. Граф может быть описан матрицей стоимости C, значение каждого элемента которой — c_{ij} равно стоимости (измеряемой реально в единицах времени, денег или расстояния) прямого проезда из города i в город j. Задача коммивояжера называется симметричной, если $c_{ij} = c_{ji}\ \forall i \neq j,\ i, j = \overline{1,n}$, т. е. если стоимость проезда между каждыми двумя городами не зависит от направления, и несимметричной, если это не так. Отсутствие собственных петель может быть обозначено как $c_{ii} = \infty\ \forall i = \overline{1,n}$ (подумайте о том, как Вы будете реализовывать это в реальной программе?).

Задача в терминах теории графов формулируется как задача нахождения покрывающего (полного) цикла наименьшей стоимости, который называется туром, на полном ориентированном графе, заданном несимметричной (в общем случае) матрицей стоимостей C.

Постановка в терминах целочисленного программирования. Задача коммивояжера, как задача целочисленного программирования, допускает несколько разных формулировок. Рассмотрим два следующих подхода.

1. Постановка в пространстве $E_z^{n^2-n}$.

Для формулировки задачи коммивояжера, как целочисленной задачи, определим вначале размерность целочисленного пространства. Поскольку наша цель — выбор некоторых ребер полного графа, составляющих тур, из всего множества ребер, то предлагается рассматривать компоненты вектора $\mathbf{x}$, как указатели на выбираемые ребра. Полный ориентированный граф на n вершинах содержит $n \cdot (n-1) = n^2 - n$ ребер — это и есть размерность нашего целочисленного пространства. Компоненты вектора $\mathbf{x}$ принимают значения 0 или 1, указывая на выбираемые ребра и, следовательно, $\mathbf{x} \in 1_z^{n^2-n}$. Поскольку тур содержит ровно n ребер, то вектор $\mathbf{x}$ содержит ровно n компонент, равных единице. Таким образом, множество точек перебора — это некоторое подмножество точек положительной полу-

сферы ($x_i \geq 0$) с радиусом равным $\sqrt{n}$, и с центром в нуле. В принятых обозначениях (см. гл. 7) это — $S_z^{n^2-n}\left(\mathbf{0}, \sqrt{n}\right)$. Заметим, что не все точки этой сферы с положительными целыми координатами являются возможными решениями задачи коммивояжера, поскольку ребра, выбранные по единичным компонентам вектора $\mathbf{x}$, должны составлять тур. Для построения целевого функционала и ограничений необходимо ввести два отображения. Обозначим далее через N — конечное множество целых чисел

$$N = \left\{ i \mid i = \overline{1, n} \right\},$$

через M — множество

$$M = \left\{ i \mid i = \overline{1, n^2 - n} \right\},$$

и через R^+ — множество положительных вещественных чисел. Определим взаимно однозначное отображение

$$M \underset{v}{\rightarrow} N \times N,\ v(i) = (k, l),$$

которое каждому номеру ребра i ориентированного графа ставит в соответствие номера пар инцидентных ребру вершин — (k, l), и отображение

$$M \underset{c}{\rightarrow} R^+,\ c(i) = c_{v(i)} = c_{kl},$$

которое задает веса (стоимости) для последовательно пронумерованных ребер. Поскольку задача состоит в минимизации общей стоимости объезда городов, то постановка задачи имеет вид

$$f(\mathbf{x}) = \sum_{i=1}^{n^2-n} c(i) \cdot x_i,\ f(\mathbf{x}) \rightarrow \min \text{ , при}$$

$$\begin{cases} x_i \in \{ 0,1 \},\ \sum\limits_{i=1}^{n^2-n} x_i = n,\ \mathbf{x} \in S_z^{n^2-n}\left(\mathbf{0}, \sqrt{n}\right); \\ n \text{ ребер, соответсвующих вектору } \mathbf{x} \text{ образуют тур.,} \end{cases}$$

математическая формулировка последнего условия может быть полезным упражнением для читателей.

2. Постановка в пространстве E_z^{n-1}.

Предыдущая постановка, несмотря на кажущуюся простоту структуры вектора $\mathbf{x}$, имеет серьёзный недостаток — среди всех рассматриваемых векторов

только очень немногие образуют тур. Хотелось бы, чтобы все точки некоторого множества в пространстве E_z^{n-1} гарантированно представляли тур. Если это так, то опадает необходимость проверки условия тура для точек этого множества, кроме того, по очевидным соображениям, хотелось бы уменьшить размерность пространства. Новая постановка задачи опирается на понятие перестановки на множестве целых чисел. Будем далее обозначать через $\pi(k,l), l \geq k$ множество всех перестановок целых чисел $k, k+1, \cdots, l$, очевидно, что таких перестановок $(l-k+1)!$. Рассмотрим множество перестановок $\pi(2,n)$ — оно содержит

$$|\pi(2,n)| = (n-2+1)! = (n-1)!$$

различных перестановок — ровно столько имеется различных туров коммивояжера в задаче с n городами. Поскольку тур — это полный цикл по всем вершинам, то начальная вершина тура может быть выбрана произвольно. Фиксируем вершину с номером один, и считаем, что некоторая перестановка из множества $\pi(2,n)$ задает порядок обхода вершин, начиная с первой, и после последней вершины, заданной этой перестановкой, мы снова возвращаемся в начало тура. Таким образом, компоненты нашего вектора $\mathbf{x}$ в пространстве E_z^{n-1} — это числа от двух до n, а сам вектор ассоциирован с некоторой перестановкой из $\pi(2,n)$, и мы можем записать

$$x_i \in \{\, 2,n \,\}, i = \overline{1,n-1}, x_i \neq x_j, \text{при } i \neq j, \quad \mathbf{x} \in \pi(2,n). \tag{12.3.1}$$

Множество векторов $\mathbf{x}$, координаты которых совпадают с перестановками из $\pi(2,n)$, обозначим через $\pi_z^{n-1}(2,n)$. Где расположены точки различных векторов $\mathbf{x}$ в пространстве E_z^{n-1}, удовлетворяющие (12.3.1)? Заметим, что сумма квадратов компонент различных векторов $\mathbf{x}$ одинакова — это разные перестановки чисел от двух до n. Таким образом, точки различных векторов $\mathbf{x}$ являются точками положительной полусферы в E_z^{n-1} с центром в нуле и радиусом

$$r = \sqrt{\sum_{i=2}^{n} i^2} = \sqrt{\frac{n \cdot (n+1) \cdot (2n+1)}{6} - 1}, \quad \mathbf{x} \in S_z^{n-1}(\mathbf{0}, r).$$

Осталось сформулировать, как задаются стоимости ребер — воспользуемся уже введенными в первой постанове задачи множествами, и определим отображение

$$N \times N \underset{c}{\to} R^+,\ c(i,j) = c_{ij},\ c(i,i) = \infty,$$

которое каждой упорядоченной паре вершин графа ставит в соответствие стоимость ребра, инцидентного этой паре. В построенном формализме задача коммивояжера в пространстве E_z^{n-1} имеет следующую постановку

$$f(\mathbf{x}) = c(1, x_1) + c(x_{n-1}, 1) + \sum_{i=1}^{n-2} c(x_i, x_{i+1}),\quad f(\mathbf{x}) \to \min,\ \text{при}$$

$$\mathbf{x} = (x_1, \cdots, x_n) \in \pi_z^{n-1}(2, n).$$

Рассматриваемый далее метод ветвей и границ работает, хотя и не явно, именно с этой постановкой задачи коммивояжера.

Реализация идей метода ветвей и границ для задачи коммивояжера. В соответствии с общей идеей метода ветвей и границ все множество допустимых решений должно последовательно разделяться на подмножества с целью сокращения перебора — это процедура ветвления. С каждым таким подмножеством должна быть связана оценка (нижняя граница при поиске минимума), обеспечивающая отсечение тех подмножеств, которые заведомо не содержат оптимального решения — это процедура построения границ. Таким образом, метод приводит к исследованию древовидной модели пространства решений.

В рассматриваемой задаче таким исходным множеством является множество всех туров коммивояжера, и минимизируется целевой функционал, задающий стоимость тура. Излагаемые ниже идеи авторов алгоритма — это своего рода классика метода ветвей и границ. Для построения алгоритма необходимо описать две основные процедуры — ветвление и построение границ. Начнем рассмотрение с процедуры ветвления. Построение поискового дерева решений начинается с корня, который будет соответствовать множеству «всех возможных туров», т. е. эта вершина дерева представляет множество R всех $(n-1)!$ возможных туров в задаче с n городами. Ветви, выходящие из корня, определяются выбором одного ребра, скажем ребра (k,l). Идея авторов алгоритма состоит в том, чтобы разделить текущее множество туров на два множества: одно — которое, весьма вероятно, содержит оптимальный тур, и другое — которое, вероятно, этого тура не содержит. Для этого выбирается ребро (k,l), которое, вполне вероятно, входит в оп-

тимальный тур, и множество R разделяется на два множества $\{k,l\}$ и $\{\overline{k,l}\}$. Во множество $\{k,l\}$ входят все туры из R, содержащие ребро (k,l), т. е. проходящие через него, а во множество $\{\overline{k,l}\}$ — туры, не содержащие это ребро. Заметим, что идея авторов алгоритма очень перспективна — если так организовать процесс ветвления, что на каждом шаге выбирается «правильное» ребро, то весь процесс будет завершен за n шагов.

Проиллюстрируем сказанное на конкретном примере. В таблице 12.4 приведена матрица стоимостей для несимметричной задачи коммивояжера с пятью городами (пример из [12.5]).

Табл. 12.4 Исходная матрица стоимостей

	1	2	3	4	5
1	∞	25	40	31	27
2	5	∞	17	30	25
3	19	15	∞	6	1
4	9	50	24	∞	6
5	22	8	7	10	∞

В качестве претендентов на ребро ветвления можно рассмотреть ребра, имеющие небольшие стоимости. Предположим, что производится ветвление на ребре $(3,5)$, имеющем наименьшую стоимость во всей матрице. Ниже будет рассмотрен другой алгоритм выбора ребра ветвления. Корень и первый уровень поискового дерева решений будут тогда такими, как показано на рисунке 12.5. Заметим, что каждый тур содержится только в одном из множеств уровня 1. Если бы как-то можно было сделать вывод о том, что множество $\{\overline{3,5}\}$ не содержит оптимального тура, то нужно было бы исследовать далее только множество $\{3,5\}$ — в этой идее, собственно говоря, и заключается суть метода ветвей и границ.

Продолжая процедуру ветвления, разделяем множество $\{3,5\}$ так же, как и множество R. Следующее по дешевизне ребро в матрице — это ребро $(2,1)$ со стоимостью 5. Поэтому можно разделить множество $\{3,5\}$ на подмножество ту-

ров, включающих ребро $(2,1)$, и подмножество туров, не включающих это ребро. Полученное поисковое дерево решений показано на рисунке 12.5.

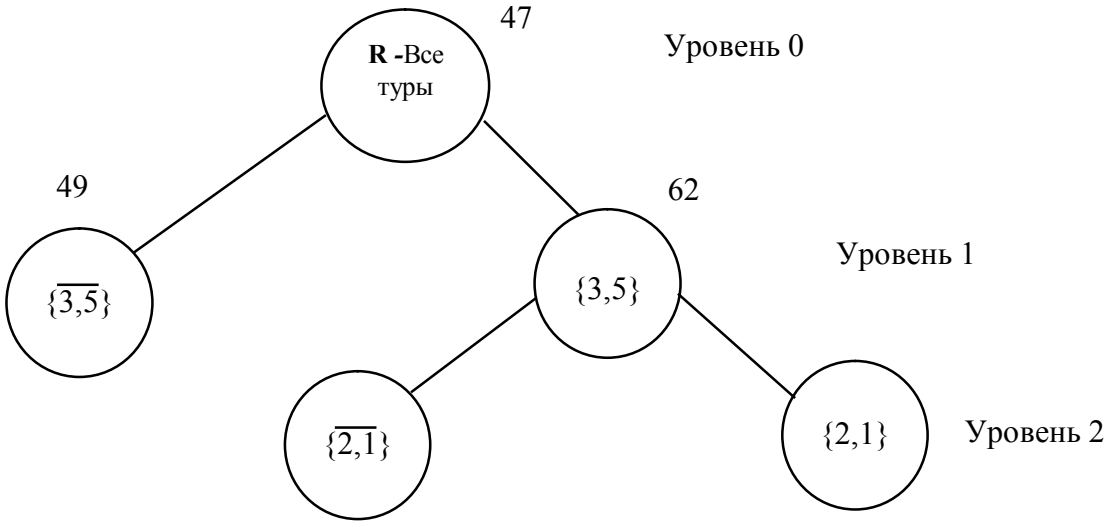


Рисунок 12.5. Фрагмент поискового дерева решений

Путь от корня к любой вершине дерева выделяет определенные ребра, которые должны, или не должны быть включены во множество, представленное данной вершиной. Например, левая вершина уровня 2 на рисунке 12.5 представляет множество всех туров, проходящих через ребро $(3,5)$ и не проходящих через ребро $(2,1)$. Таким образом, процедура ветвления состоит в разделении текущего подмножества туров на два подмножества путем выбора некоторого ребра.

Теперь расскажем об идее авторов алгоритма, связанной с процедурой вычисления границ. С каждой вершиной дерева связывается нижняя граница стоимости любого тура из множества, представленного данной вершиной. Вычисление нижних границ — основной фактор, дающий возможность сокращения перебора в любом алгоритме, реализующем метод ветвей и границ. Очевидно, что задача состоит в получении как можно более точных нижних границ. Причина этого следующая. Предположим, что уже получен конкретный полный тур со стоимостью Ct. Если нижняя граница, связанная с множеством туров, представленных некоторой вершиной поискового дерева, больше, чем Ct, то до конца процесса поиска не нужно рассматривать эту и все следующие за ней вершины, т. к. если мы гарантируем нижнюю границу, то все содержащиеся в этой вершине туры имеют стоимость большую, чем уже найденный тур. В реализации это

приводит к усечению поискового дерева решений путем отбрасывания всех листьев поискового дерева, имеющих стоимость большую, чем Ct.

Основной шаг при вычислении нижних границ известен как процедура *приведения матрицы стоимостей.* Эта процедура основана на следующих двух соображениях:

1. В терминах матрицы стоимостей каждый полный тур содержит только один элемент (ребро и соответствующую стоимость) из каждого столбца и каждой строки матрицы. Заметим, что обратное утверждение не всегда верно — множество, содержащее один и только один элемент из каждой строки и из каждого столбца, не обязательно представляет тур.

2. Если вычесть константу h из каждого элемента какой-то строки или столбца матрицы стоимостей, то стоимость любого тура при новой матрице будет ровно на h меньше. Поскольку любой тур должен содержать ребро из данной строки или данного столбца, стоимость всех туров уменьшается на h. Это вычитание называется приведением строки (или столбца) матрицы стоимостей. Пусть t — оптимальный тур при матрице стоимостей C. Стоимость тура t может быть определена как

$$z(t) = \sum_{(i,j)\in t} c_{ij} .$$

Если матрица C' получается из матрицы C приведением строки (или столбца), то тур t должен оставаться оптимальным туром и в матрице C', при этом стоимости туров связаны соотношением $z(t) = h + z'(t)$, где $z'(t)$ — стоимость тура t в матрице C'. Под процедурой приведения всей матрицы стоимостей понимается следующее: последовательно просматриваются все строки и значение наименьшего элемента h_i каждой строки вычитается из каждого элемента этой строки. Затем аналогичные действия выполняются для каждого столбца. Если для некоторого столбца или строки значение $h_i = 0$, то рассматриваемый столбец или строка уже приведены, и происходит переход к следующему столбцу или строке матрицы стоимостей.

Полученную в результате матрицу стоимостей назовем *приведенной из* C. В таблице 12.5 показано приведение исходной матрицы стоимостей. Значения h_i

даны в конце каждой строки и столбца (строки и столбцы последовательно пронумерованы).

Таблица 12.5 Приведенная матрица стоимостей

	1	2	3	4	5	
1	∞	25	40	31	27	h=25
2	0	∞	17	30	25	h_2=5
3	19	15	∞	6	1	h_3=1
4	9	50	24	∞	6	h_4=6
5	22	8	7	10	∞	h_5=7
	h_6=0	h_7=0	h_8=0	h_9=3	h_{10}=0	

Общее приведение составляет $h = 25+5+1+6+7+3 = 47$, следовательно, нижняя граница стоимости любого тура из множества R равна 47, т. е.

$$z(t) = h + z'(t) \geq h = 47,$$

так как $z'(t) \geq 0$ для любого тура t при приведенной матрице C'. Эта граница и указана около корня дерева на рисунке 12.5. Заметим, что если ребра, имеющие нулевую стоимость в приведенной матрице, составляют тур, то алгоритм сразу получает оптимальное решение, но, к сожалению, это почти что всегда не так.

Следующий вопрос — это вопрос о вычислении границ для некорневых вершин поискового дерева решений. Авторы алгоритма предлагают следующие процедуры для получения этих оценок. По определению ребро $(3,5)$ содержится в каждом туре множества $\{3,5\}$. Этот факт препятствует выбору ребра $(5,3)$, так как ребра $(3,5)$ и $(5,3)$ образуют цикл, а это не допускается ни для какого тура. Поэтому ребро $(5,3)$ можно исключить из рассмотрения, положив значение $c_{53} = \infty$. Строку 3 и столбец 5 также можно исключить из дальнейшего рассмотрения по отношению к множеству туров $\{3,5\}$, потому что уже есть ребро из вершины 3 в вершину 5. Часть приведенной матрицы стоимостей, из таблицы 12.5, которая будет необходима для дальнейшего поиска на множестве туров $\{3,5\}$, показана в таблице 12.6. Она может быть приведена к матрице стоимостей, показанной в таблице 12.7 со значением $h = 15$. Теперь нижняя граница для любого тура из множества $\{3,5\}$ равна 47+15=62.

Таблица 12.6. Матрица стоимостей для множества $\{3,5\}$.

	1	2	3	4
1	∞	0	15	3
2	0	∞	12	22
4	3	44	18	∞
5	5	1	∞	0

Таблица 12.7. Приведенная матрица стоимостей для множества $\{3,5\}$.

	1	2	3	4	
1	∞	0	3	3	0
2	0	∞	0	22	0
4	0	41	3	∞	h=3
5	15	1	∞	0	0
	0	0	h=12	0	

Нижняя граница для множества $\{\overline{3,5}\}$ получается несколько иным способом. Ребро $(3,5)$ не может находиться в этом множестве, поэтому полагаем $c_{35}=\infty$ в матрице, приведенной в таблице 12.5. В любой тур из множества $\{\overline{3,5}\}$ будет входить какое-то ребро из вершины 3 и какое-то ребро к вершине 5. Самое дешевое ребро из вершины 3, исключая старое значение $(3,5)$, имеет стоимость 2, а самое дешевое ребро к вершине 5 имеет стоимость 0. Следовательно, нижняя граница любого тура во множестве $\{\overline{3,5}\}$ равна 47+2+0=49.

При приведении матрицы стоимости для множества $\{3,5\}$ нам удалось сократить ее размер. Очевидно, что это сокращение будет происходить, каждый раз для множества $\{k,l\}$, так что при этом значительно сокращаются вычислительные затраты. Еще один вывод состоит в том, что если можно найти тур из множества $\{\overline{3,5}\}$ со стоимостью, меньшей или равной 62, то тогда вершина $\{3,5\}$ поискового дерева решений может быть отброшена, и в этом случае будем говорить, что вершина $\{3,5\}$ в дереве отработана. Тогда следующей целью может быть ветвление из вершины $\{\overline{3,5}\}$ в надежде найти тур со стоимостью в пределах $49 \le Ct \le 62$.

Алгоритм метода ветвей и границ для задачи коммивояжера. Теперь, когда ясно как, в общих чертах, выглядят процедуры ветвления и построения границ, можно предложить следующую схему алгоритма метода ветвей и границ (МВГ) для задачи коммивояжера, в которой приняты следующие обозначения. Пусть X — текущая вершина поискового дерева, а (k,l) — ребро по которому происходит ветвление, обозначим вершины, непосредственно следующие за X, через Y и $\overline{Y}$. Множество Y есть подмножество туров из X, проходящих через ребро (k,l), а множество $\overline{Y}$ — подмножество туров из X, не проходящих через ребро (k,l). Вычисленные нижние границы для множеств Y и $\overline{Y}$ обозначим через $w(Y)$ и $w(\overline{Y})$ соответственно. Самый дешевый тур, известный алгоритму в данный момент обозначим через z_0, причем, в момент инициализации, $z_0 = \infty$.

```
A1(C, n) Схема алгоритма МВГ для задачи коммивояжера
   (n - размерность, C - матрица стоимостей.)
   1. Инициализация.
   2. Приведение матрицы стоимостей C.
   3. Установка корня поискового дерева решений X=R
      приведение исходной матрицы - вычисление w(X).
   While (w(X) < z0)
      begin
         4. Выбор ребра ветвления(k,l).
         5. Процесс ветвления. Создание вершины Y¯
            и вычисление w(Y¯).
         6. Процесс ветвления. Создание вершины Y
            и вычисление w(Y).
         If (размер матрицы стоимостей в вершине Y = 2)
            then
               begin
                  7. Проведение исчерпывающей оценки для вершины Y
                  If (w(Y) < z0)
                     then
                                          begin
                                                z0 =
w(Y)(запоминаем тур)
                                            end
               end
         8. Выбор следующей вершины поискового дерева
            решений, и установка X
         9. Вычисление фрагмента матрицы C,
            соответствующего выбранной вершине X,
            на основании пути от корня поискового
            дерева решений до текущей вершины.
      end (while w(X) < z0)
Оптимальное решение со стоимостью z0 найдено
End.
```

Особенности реализации этапов алгоритма. В приведенной укрупненной схеме данного алгоритма остались нераскрытыми некоторые важные детали, которые необходимо обсудить.

Этап 1. Установление начальных значений переменных, или инициализация, не представляет труда для программиста. Единственный вопрос — это выбор структуры данных для хранения поискового дерева решений, при этом необходимо иметь в виду, что количество вершин поискового дерева может быть значительно. Это замечание касается способа выделения памяти под структуру дерева.

Этап 2. Приведение исходной матрицы стоимостей — это непосредственная реализация описанной ранее процедуры, детали которой оставляются в качестве упражнения для читателей.

Этап 3. Инициализация корня поискового дерева. Основной вопрос в том, будет ли вместе с вершиной дерева храниться и матрица стоимостей. Альтернативой является перевычисление матрицы стоимостей для текущей вершины на основе исходной. Это классический выбор между производительностью и требуемым объемом памяти. Вопрос хранения матриц для вершин поискового дерева будет более подробно обсуждаться в рамках модификации этого алгоритма. Классический алгоритм предполагает перевычисление матрицы стоимостей для вновь устанавливаемой текущей вершины поискового дерева решений.

Этап 4. Как уже обсуждалось, выбор следующего ребра ветвления (k,l) определяет множества Y и $\overline{Y}$, непосредственно следующие за текущим множеством решений X. Ребро (k,l) для обеспечения сокращения перебора, нужно выбирать так, чтобы попытаться получить большую по величине нижнюю границу на множестве $\overline{Y}$, что облегчит проведение оценки для множества Y. Таким образом, мы пытаемся заставить алгоритм выбирать на каждом шаге множество Y, пытаясь решить задачу всего за n шагов. Как применить эти идеи к выбору конкретного ребра ветвления (k,l)? В приведенной матрице стоимостей C', связанной с вершиной X, каждая строка и столбец имеют хотя бы по одному нулевому элементу (если это не так, то матрица C' не полностью приведена). Можно предположить, что ребра, соответствующие этим нулевым стоимостям, будут с большей вероятностью входить в оптимальный тур, чем ребра с большими стоимостями. Поэтому в каче-

стве ребра ветвления можно выбрать одно из них, при этом хотелось бы, чтобы у множества $\overline{Y} = \{\overline{k,l}\}$ была как можно большая нижняя граница. Пусть ребро (k,l) имеет $c_{kl} = 0$, и, обращаясь к процедуре вычисления нижней границы, мы видим, что для множества $\overline{Y}$ эта граница задается в виде

$$w(\overline{Y}) = w(X) + \min_{i,i\neq k} c_{kl} + \min_{j,j\neq l} c_{kl},$$

следовательно, из всех ребер (k,l), у которых $c_{kl} = 0$ в текущей матрице C' выбирается то, которое дает наибольшее значение для нижней границы — $w(\overline{Y})$. Более подробный алгоритм выбора ребра ветвления на этапе 4 имеет вид

```
Выбор ребра ветвления
Begin
   (Пусть S — множество ребер (i, j), таких,
    что c[i,j] = 0 в текущей матрице стоимостей C.
    Положим   Dij = min (стоимости в строке i, исключая c[i,j])
                    +
                    min (стоимость в столбце j, исключая c[i,j])
   )
      1. Вычисляем Dij для всех (i, j) ∈ S
      2. Выбираем следующее ребро ветвления (k, l) из условия
         Dkl = max (Dij) для всех (i, j) ∈ S
End
```

Этап 5. На этом этапе реализуется ветвление, и в поисковое дерево добавляется вершина $\overline{Y}$, следующая за X. Нижняя граница для множества $\overline{Y}$ вычисляется следующим образом

$$w(\overline{Y}) = w(X) + D_{kl},$$

где D_{kl} уже вычислено на этапе 4.

Этап 6. Вершине Y, следующей за X, соответствует подмножество туров из множества X, содержащих то ребро (k,l), которое выбрано на этапе 4. При формировании матрицы стоимости, соответствующей вершине Y необходимо учитывать, что наша задача — нахождение тура, и все ребра, которые могут привести к более коротким циклам, должны быть запрещены. Детальное рассмотрение приводит к следующему алгоритму.

```
Формирование матрицы для вершины Y
   begin
      1. Из матрицы C для вершины X исключаем
         строку k и столбец l.
      If (ребро (k, l) не изолировано от других ребер,
          включенных в тур по пути от корня дерева до вершины X)
```

```
    then
       begin
          2. Находим начальный - p и конечный - q город пути
          3. C[p, q]=∞
       end
    4. Выполняем процедуру приведения полученной матрицы
       стоимостей. h = сумма констант приведения.
    5. Вычисляем w(Y) = w(X)+h
end
```

Этап 7. В конце концов, процесс ветвления приводит нас к множествам, содержащим так мало туров, что можно рассмотреть каждый из них и провести оценку для этой вершины без дальнейшего ветвления. Каждое ребро, про которое известно, что оно содержится во всех турах из множества Y, сокращает размер матрицы стоимостей на одну строку и один столбец. Если в исходной задаче было n городов, а текущая матрица имеет размер 2×2, то уже $n-2$ ребер содержатся во множестве Y, поэтому в Y содержится самое большее два тура — можно провести исчерпывающую оценку и получить конкретный тур коммивояжера.

Этап 8. Теперь нужно выбрать следующую вершину X, от которой необходимо проводить ветвление. Этот выбор довольно очевиден. Мы выбираем ту вершину, которая имеет в данный момент наименьшую нижнюю границу, и из которой в данный момент не выходят ветви, т. е. — это лист поискового дерева решений с минимальной нижней границей.

Этап 9. Поскольку новая вершина поискового дерева X выбрана в качестве текущей, то задача этого этапа — получить матрицу стоимостей, соответствующую вершине X. Если мы храним матрицы стоимостей вместе с вершинами поискового дерева, то матрица уже есть. В противном случае нам необходимо найти путь от корня до этой вершины и последовательно корректировать исходную матрицу стоимостей.

В заключение обсуждения этапов приведем фрагмент дерево решений для рассматриваемого примера, построенного на основе приведенных алгоритмов — рисунок 12.6.

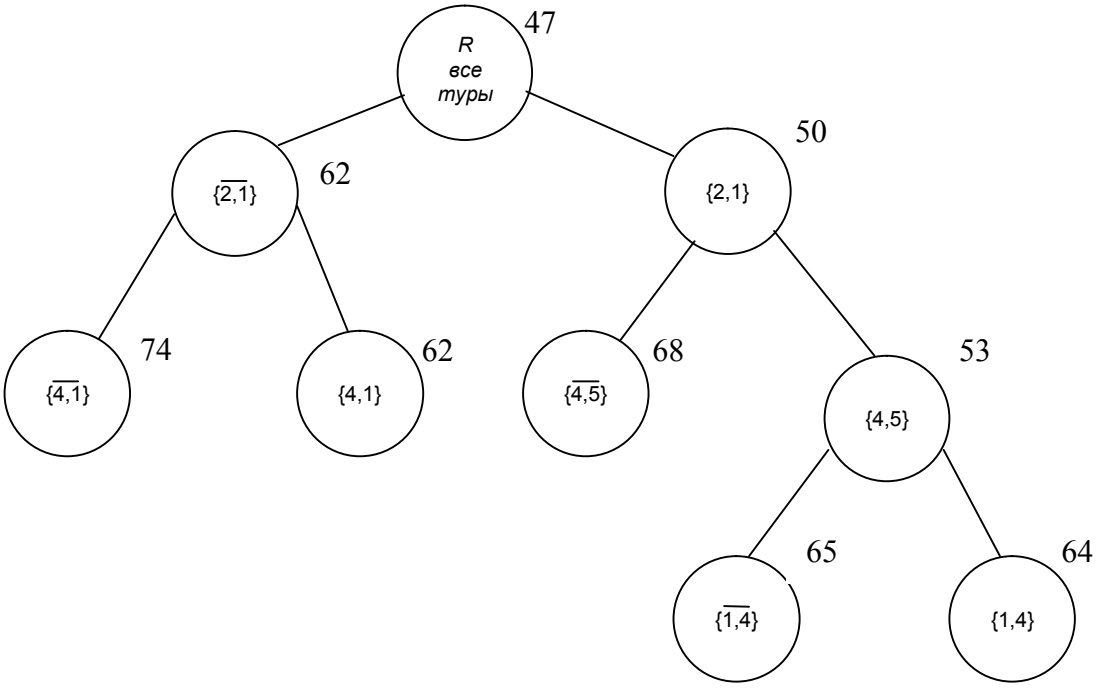


Рисунок 12.6. Фрагмент дерева решений для задачи коммивояжера, полученный
алгоритмом метода ветвей и границ.

Обсуждение алгоритма. Каких результатов в смысле вычислительной сложности можно ожидать для различных матриц стоимостей фиксированной размерности? Обо всех более или менее эффективных алгоритмах, реализующих метод ветвей и границ для задачи коммивояжера, известно или имеются предположения, что их трудоемкость в худшем случае является экспоненциальной. Это предположение основывается на том, что сама задача коммивояжера является NP-трудной, и, следовательно, любой точный алгоритм ее решения обладает надполиномиальной трудоемкостью. В лучшем случае — если размерность матрицы стоимостей все время сокращается, то оценка вычислительной сложности является полиномиальной. Это очевидно, т. к. оценка трудоемкости каждого из этапов алгоритма — полиномиальна по линейному размеру (n) матрицы стоимостей, и в лучшем случае основной цикл выполняется не более чем n раз, если на каждом шаге выбирается одно ребро в тур, который состоит из n ребер. Таким образом, разброс ожидаемого времени выполнения при фиксированной размерности матрицы стоимости n очень велик и зависит от численных значений ее элементов. Это пример количественно-параметрического алгоритма с сильной пара-

метрической зависимостью — алгоритм относится к классу *NPRH*. Теоретический анализ ожидаемой трудоемкости на основе предварительного исследования матрицы стоимостей очень сложен и часто выходит за рамки наших аналитических возможностей.

Модификация алгоритма. При обсуждении этапов приведенной выше схемы алгоритма метода ветвей и границ уже отмечалось, что возможен вариант, при котором вместе с поисковым деревом решений в каждой вершине хранится соответствующая матрица стоимости. Схема модифицированного алгоритма в этом случае выглядит следующим образом.

```
A2(C, n)
   Схема модифицированного алгоритма МВГ для задачи коммивояжера
   n - размерность, C - матрица стоимостей.
Begin
   1. Инициализация.
   2. Приведение матрицы стоимостей C.
   3. Установка корня поискового дерева решений X=R
      приведение исходной матрицы - вычисление w(X)
      и сохранение приведенной матрицы в корне дерева.
   While (w(X) < z0)
      begin
         4. Выбор ребра ветвления(k,l).
         5. Процесс ветвления. Создание вершины Y¯,
            вычисление w(Y¯) и сохранение приведенной матрицы.
         6. Процесс ветвления. Создание вершины Y,
            вычисление w(Y) и сохранение приведенной матрицы.
         If (размер матрицы стоимостей в вершине Y = 2)
            then
               begin
                  7. Проведение исчерпывающей оценки для вершины Y
                  If (w(Y) < z0)
                     then
                                          begin
                                                z0 =
w(Y)(запоминаем тур)
                                            end
               end
         8. Выбор следующей вершины поискового дерева
            решений, и установка X
         9. Чтение фрагмента матрицы C,
            соответствующего выбранной вершине X,
            из структуры хранения поискового
            дерева решени.
      end (while w(X) < z0)
Оптимальное решение со стоимостью z0 найдено
End.
```

Каковы необходимые в этом случае затраты памяти? Ответ на этот вопрос можно получить, если известна функция, задающая число вершин поискового де-

рева решений в зависимости от исходной матрицы стоимостей. Теоретические оценки в этом случае достаточно проблематичны, и информацию о числе вершин поискового дерева можно получить на основе экспериментальных данных. Приводимая ниже формула получена на основе экспериментов с программной реализацией алгоритма A1. В предположении об экспоненциальном росте, вполне согласующимся с экспериментами для других размерностей, коэффициенты функции были получены с использованием метода наименьших квадратов. Функция задает среднее число вершин поискового дерева решений в зависимости от размерности матрицы стоимостей, элементы которой генерировались с использованием стандартного равномерного генератора псевдослучайных чисел, результат опубликован в [12.6]:

$$R(n) = 3{,}6932 \cdot e^{0{,}1819n} \qquad (12.3.2)$$

Несколько значений, вычисленных по формуле (12.3.2), приведены в таблице 12.8.

Таблица 12.8

n	$R(n)$
20	140
30	866
40	5 337
50	32 909
60	202 905
70	1 251 049
80	7 713 582
90	47 559 556
100	293 237 462

Если приближенно считать, что средний размер матрицы стоимостей в вершинах поискового дерева равен половине исходного, то несложные расчеты показывают, что для исходной матрицы с линейным размером 70 требуемый объем оперативной памяти для всех хранимых матриц составляет порядка 730 Мб. Тем не менее, если такого рода затраты памяти являются допустимыми, то экспериментальные данные показывают, что программная реализация алгоритма A2 обеспечивает в среднем двух-трех кратное сокращение времени получения точного решения задачи коммивояжера по сравнению с программной реализацией алгоритма A1.

12.4 Рациональные ресурсно-адаптивные алгоритмические решения по компоненту формирования глобальной матрицы в системе конечно-элементного анализа

Содержание этого параграфа отражает некоторые результаты по совершенствованию алгоритмического обеспечения системы конечно-элементного анализа, опубликованные в [12.7, 12.8]. В проблематике создания ресурсно-эффективного алгоритмического обеспечения, материал этого параграфа демонстрирует варианты комбинированных алгоритмов как по критерию временной, так и емкостной эффективности.

Введение в проблему. Использование метода конечных элементов (МКЭ) для решения различных технических и физических задач имеет давнюю историю. Начиная с середины 70-х годов, МКЭ стал активно применяться для расчетов напряженно-деформированного состояния, нестационарных тепловых и гидравлических процессов и решения многих других актуальных научно-технических задач. В настоящий момент круг задач, решаемых с помощью систем конечно-элементного анализа, охватывает почти все сферы инженерных расчетов: прочность, колебания, устойчивость, динамика, акустика, гидродинамика, аэродинамика и т. д. По сути метод конечных элементов сводит решение уравнений в частных производных, описывающих исследуемый процесс, к решению системы линейных уравнений большой размерности. В связи с этим, программная система, реализующая метод конечных элементов, требует больших объемов оперативной памяти и значительного процессорного времени, особенно при решении обратных задач. В связи с широким распространением систем конечно-элементного анализа и усложнением структуры объектов и состава решаемых задач, актуальной является проблема повышения их ресурсной эффективности. Одним из путей решения данной проблемы является выбор или разработка эффективных алгоритмов, реализующих этапы конечно-элементного анализа.

Ресурсные особенности задачи формирования глобальной матрицы в методе конечных элементов. Для анализа ресурсных особенностей программных систем, реализующих метод конечных элементов, рассмотрим основные этапы,

последовательное выполнение которых приводит к получению решения по анализу исследуемого объекта:

— создание геометрической модели исследуемого объекта;

— выбор типа конечного элемента;

— генерация сети конечных элементов;

— задание физических параметров;

— задание граничных условий;

— формирование расчетной задачи, состоящей из задания связи физических параметров и граничных условий с конечно-элементной моделью объекта;

— формирование глобальной матрицы системы линейных алгебраических уравнений (СЛАУ);

— решение СЛАУ;

— визуализация и анализ результатов решения.

В ходе исторического развития систем конечно-элементного анализа эти этапы были распределены по трем взаимодействующим подсистемам:

— препроцессор — подсистема, осуществляющая подготовку исходных данных; в ее функции входит определение геометрической области и её дискретизация, задание свойств материалов, начальных и граничных условий;

— расчетная подсистема — подсистема, осуществляющая формирование и решение системы линейных алгебраических уравнений (СЛАУ) на основе исходных данных, сформированных препроцессором;

— постпроцессор — подсистема, предназначенная для анализа результатов расчёта, она включает в себя различные средства отображения результатов.

Для связи между подсистемами устанавливаются межмодульные интерфейсы, реализуемые в виде файлов с установленными стандартами. Обычно перечисленные подсистемы чётко выделены в структуре программной системы и реализованы в виде отдельных модулей, которые могут быть объединены единым пользовательским интерфейсом. Такое разделение оправдано снижением затрат на программирование за счёт независимой разработки и тестирования отдельных подсистем.

Отметим, что независимо от конкретной реализации, ядром конечно-элементной системы является расчетная подсистема, отвечающая за непосредственное решение задачи, т. к. остальные подсистемы выполняют вспомогательные функции и являются обслуживающими. Наличие расчётной подсистемы выделяет программное обеспечение систем конечно-элементного анализа в отдельный класс, имеющий свои особенности и собственные требования к задаче построения эффективного программного обеспечения. Можно выделить следующие основные особенности данного класса программных систем:

— большой объём обрабатываемых данных и необходимость использования сложных структур для их хранения. Число элементов конечно-элементных моделей, используемых при решении современных задач, составляет от нескольких тысяч до нескольких десятков тысяч, что выдвигает необходимость обрабатывать в процессе расчёта миллионы переменных, при этом существует тенденция к увеличению числа переменных, что обусловлено усложнением расчётных конечно-элементных моделей и алгоритмов анализа;

— применение трудоёмких циклических алгоритмов конечно-элементного анализа и других вспомогательных численных методов (например, для решения систем линейных алгебраических уравнений). В зависимости от сложности конечно-элементной модели и нелинейности задачи, время расчета может составлять от нескольких часов до нескольких дней, даже при использовании производительной вычислительной техники. Очень часто пользователю системы конечно-элементного анализа приходится выбирать между точностью расчётов и временем решения;

— динамично расширяющийся круг пользовательских задач и областей применения, которые должна решать система конечно-элементного анализа. Этот фактор, совместно с дальнейшим развитием самого метода конечных элементов, приводит к добавлению новых методик и алгоритмов анализа в программную реализацию системы.

Исследование разнообразных программных решений в области систем конечно-элементного анализа позволяет выделить в настоящее время два следующих направления их развития:

— разработка и развитие эффективных численных методов, используемых при реализации метода конечных элементов (аппроксимация искомых значений на конечных элементах при нелинейных процессах, построение эффективных нелинейных алгоритмов, численное интегрирование, численное решение систем алгебраических линейных уравнений и т. д.);

— разработка новых эффективных алгоритмов, основанных на современных методах программирования, позволяющих оптимизировать процесс вычислений, исходя из анализа их работоспособности применительно к реальной архитектуре ЭВМ. Сюда же можно отнести эффективные схемы хранения данных, поскольку алгоритмы манипулирования с большими разреженными матрицами неотделимы от таких схем.

В части разработки и развития численных методов сегодня достигнуты значительные успехи, которые по отношению к рассматриваемой программной системе касаются в первую очередь разработки эффективных вычислительных алгоритмов решения СЛАУ. Более детальная реализация алгоритмического направления развития систем конечно-элементного анализа связана с разработкой и исследованием эффективных алгоритмов для наиболее ресурсоёмких этапов — этапа генерации сети конечных элементов и формирования граничных условий и этапа формирования конечно-элементной матрицы. Поскольку для этапа решения СЛАУ в области численных методов разработано достаточно много эффективных алгоритмов (упомянем, например, метод сопряженных градиентов с предварительным обуславливанием), то речь может идти о выборе наиболее рационального из них, с учетом особенностей сформированной матрицы.

Важнейшим и одним из самых трудоемких этапов решения прикладных задач методом конечных элементов является этап формирования глобальной матрицы системы линейных алгебраических уравнений (СЛАУ). Временные затраты на этапы формирования глобальной матрицы и решения СЛАУ во многом определяют время решения задачи с помощью метода конечных элементов в целом.

Система линейных алгебраических уравнений, получаемая на основе метода конечных элементов, характеризуются сильно разреженной матрицей, в которой значащие (отличные от нуля) коэффициенты, составляют, как правило, не более 2-

3% от общего числа коэффициентов матрицы. Количество значащих коэффициентов в строке матрицы определяется типом используемого конечного элемента, числом конечных элементов, связанных с узлом, которому соответствует данная строка матрицы, а также числом степеней свободы в узле, для которого ищется решение. Так, например, для теплового расчета (одна степень свободы) с помощью восьми точечной призмы число значащих коэффициентов составляет, как правило, не более 14-ти отличных от нуля элементов в строке глобальной матрицы [12.7].

Данная особенность СЛАУ в МКЭ определяет способ хранения коэффициентов СЛАУ в оперативной памяти ЭВМ в форме «ключ — не нулевое значение коэффициента», где ключом является индекс не нулевого коэффициента в строке матрицы. Такая форма представления матрицы позволяет избежать существенных ресурсных затрат оперативной памяти на хранение нулевых элементов. Значительная размерность СЛАУ в МКЭ для реальных задач делает данный способ хранения матрицы практически безальтернативным. Именно большая размерность глобальной матрицы и относительно сложный алгоритм формирования ее элементов приводят к значительным временным затратам на этом этапе в МКЭ, и обусловливают необходимость поиска рациональных алгоритмических решений.

Подход к разработке ресурсно-адаптивного алгоритма формирования глобальной матрицы. Проведенное исследование особенностей задачи формирования матрицы СЛАУ в МКЭ показало, что дальнейшее сокращение времени выполнения этапа формирования глобальной матрицы может быть получено за счет принципиально новых алгоритмических решений, повышающих временную эффективность за счет дополнительных объемов памяти.

В рамках этого подхода был разработан новый алгоритм формирования глобальной матрицы СЛАУ, который получил название алгоритма прохода по строкам [12.8]. Основная идея нового алгоритма состоит в том, чтобы формировать значения коэффициентов глобальной матрицы, переходя по очереди от одного коэффициента строки к следующему за ним. Таким образом, в этом алгоритме исключается необходимость поиска нужной позиции в строке, как это происходит в классическом алгоритме. Реализация предложенной идеи потребовала создания дополнительной структуры данных, получившей название матрицы связей, кото-

рая позволяет получить все необходимые данные для расчета значения каждого коэффициента матрицы.

Для задач с одной степенью свободы, каждый элемент матрицы связей позволяет рассчитать значения одного соответствующего ему коэффициента глобальной матрицы. Каждый элемент матрицы связей представляет собой массив, элементы которого, в свою очередь, содержат геометрические характеристики и характеристики физических параметров материалов конечного элемента, по которым может быть рассчитан его вклад в соответствующий коэффициент глобальной матрицы. Таким образом, расчет значения коэффициента матрицы СЛАУ предполагает обращение к соответствующему элементу матрицы связей и последовательный расчет вкладов каждого конечного элемента, связанного с текущим коэффициентом глобальной матрицы.

Организация дополнительных структур данных была выполнена таким образом, чтобы максимально сократить объем дополнительных данных, а так же время доступа к ним. Это позволило, в основном, не увеличить количество вычислений, необходимых для расчета каждого коэффициента глобальной матрицы по сравнению с классическим алгоритмом [12.8]. В итоге объем дополнительных данных, необходимых для реализации нового алгоритма для различных типов задач приблизительно кратен 5-6-ти объемам матрицы СЛАУ в классическом алгоритме. Однако, при этом, за счет исключения процесса поиска нужных позиций в строке глобальной матрицы, достигается сокращение времени формирования СЛАУ в среднем на 30-35% по сравнению с классическим алгоритмом [12.8].

Выбор между классическим алгоритмом формирования СЛАУ и новым алгоритмом прохода по строкам, по сути, представляет собой достаточно стандартный для разработчиков программного обеспечения выбор между трудоемкостью вычислений и доступным объемом оперативной памяти. Таким образом, для данного случая, функция ресурсной эффективности нового алгоритма формирования глобальной матрицы имеет значимый коэффициент компонента емкостной эфффективности.

Для обеспечения большей гибкости систем конечно-элементного анализа по аппаратным требованиям необходима реализация адаптивного выбора алгоритмов. Поэтому, с точки зрения теории ресурсной эффективности наиболее пер-

спективным является разработка комбинированного алгоритма, который будет использовать как классический подход к формированию глобальной матрицы по локальным матрицам конечных элементов, так и алгоритм прохода по строкам, с использованием дополнительных структур данных.

Такой комбинированный алгоритм может состоять в использовании алгоритма прохода по строкам (формировании матрицы связей) только для первых m строк матрицы СЛАУ, и классического алгоритма для расчета коэффициентов для остальных строк матрицы. Отметим, что в этом случае для расчета коэффициентов нескольких строк матрицы на границе смены алгоритмов необходимо будет использовать оба рассмотренных подхода. Адаптация к реальному компьютеру и рассчитываемой задаче может быть реализована следующим образом: количество строк m, для которых глобальная матрица формируется алгоритмом прохода по строкам с использованием дополнительной памяти, динамически определяется программной системой на основе анализа исходных данных задачи, и информации о характеристиках того компьютера, на котором будет проводиться этот расчет. При этом создание такого комбинированного адаптивного алгоритма предоставит пользователям систем конечно-элементного анализа возможность эффективно использовать то аппаратное обеспечение, которое находится в их распоряжении.

Эффективный комбинированный алгоритм поиска по ключу в структуре хранения глобальной матрицы. Другой подход к принятию рациональных ресурсно-эффективных алгоритмических решений по компоненту формирования глобальной матрицы связан с модификацией классического алгоритма. Классический алгоритм формирования матрицы СЛАУ в МКЭ, предполагает формирование глобальной матрицы (собственно матрицы СЛАУ) на основе локальных матриц отдельно взятых конечных элементов. При этом сначала вычисляются значения коэффициентов для локальной матрицы конечного элемента, а затем рассчитанные значения добавляются в глобальную матрицу. При использовании в качестве конечного элемента восьми точечной призмы это означает, что при записи значений элементов локальной матрицы в глобальную матрицу СЛАУ необходимо провести поиск в общей сложности 36 ключей в 8-и различных строках глобальной матрицы.

Таким образом, задача поиска по ключу в структуре глобальной матрицы обладает значительной удельной трудоемкостью внутри рассматриваемого этапа. Проведенные эксперименты показывают, что на операции поиска нужной позиции в структуре хранения глобальной матрицы, при использовании классического алгоритма формирования СЛАУ, может потребоваться до 50% времени, уходящего на этап формирования СЛАУ в целом. При этом можно утверждать, что ресурс оптимизации алгоритмов поиска по ключу, применительно к матрице СЛАУ в МКЭ, в значительной степени исчерпан.

В настоящее время известно достаточное много алгоритмов поиска по ключу в массиве записей, которые значительно превосходят по эффективности метод простого перебора или бинарного поиска. Однако преимущества этих алгоритмов проявляются при работе с массивами, содержащими от нескольких сотен до нескольких тысяч элементов, в зависимости от алгоритма. Поскольку число значащих коэффициентов в строке глобальной матрицы, как правило, значительно меньше, то эффективные алгоритмы поиска в строке глобальной матрицы могут быть построены на основе методов последовательного перебора и бинарного поиска, чья эффективность выше для строк небольшой длины.

При анализе ресурсной эффективности вычислительных алгоритмов хорошо известна ситуация, когда для некоторой задачи в зависимости от области реальных размерностей множества исходных данных, рациональным по временному или более сложному комплексному критерию, является применение различных алгоритмов ее решения, что позволяет построить эффективный алгоритм методом композиции. Идея композиции состоит в том, что для определенного порога длины массива ключей, достигнутого алгоритмом бинарного поиска, более оптимальным по времени будет использование алгоритма последовательного поиска элементов в текущем фрагменте этого массива.

Очевидно, что пороговое значение длины массива ключей определяется соотношением времен на обобщенную базовую операцию и количеством этих операций на один шаг поиска, задаваемое двумя алгоритмами и языком их реализации. В силу специфики алгоритмов предположение о том, что общее время на один шаг поиска при прямом переборе будет меньше, чем время на один шаг при

бинарном поиске, является обоснованным. Это связано с тем, что шаг бинарного поиска включает в себя, помимо операции сравнения, вычисление середины длины текущего массива и сдвиг одной из границ после сравнения. Следовательно, для малых длин массива ключей прямой перебор будет выполняться быстрее, чем бинарный поиск. Таким образом, возникает задача определения пороговой длины массива ключей для алгоритма бинарного поиска. Для длин фрагмента массива ключей меньше пороговой, рациональнее выполнять поиск прямым перебором с целью минимизации общего времени выполнения программной реализации композиционного алгоритма.

Введем следующие обозначения — пусть: n — длина исходного массива ключей; k — число шагов половинного деления, выполняемых в алгоритме бинарного поиска; a — количество базовых операций языка реализации на одно половинное деление массива; t_a — среднее время на одну обобщенную операцию в алгоритме бинарного поиска; b — количество базовых операций языка реализации на один шаг последовательного поиска ключа; t_b — среднее время на одну обобщенную операцию в алгоритме последовательного поиска. Кроме того, будем предполагать, что при последовательном поиске ключа количество сравнений в среднем, необходимых для получения результата, равно половине длины обрабатываемого массива ключей.

Учитывая, что после k шагов алгоритма бинарного поиска мы получим в среднем фрагмент массива ключей длиной $\frac{n}{2^k}$, и, вводя обозначение $T(n,k)$ для среднего времени выполнения композиционного алгоритма поиска, получаем оценку среднего времени в виде:

$$T(n,k) = a \cdot t_a \cdot k + \frac{1}{2} \cdot b \cdot t_b \cdot \frac{n}{2^k}. \qquad (12.4.1)$$

Приравнивая нулю частную производную по k для выражения (12.4.1) и логарифмируя полученное выражение по основанию 2, определим оптимальное среднее значение k, минимизирующее функцию $T(n,k)$, обозначив его через k^*:

$$k^* = \log_2 n + \log_2\left(\frac{b \cdot t_b}{a \cdot t_a}\right) + \log_2(\ln 2) - 1. \qquad (12.4.2)$$

На основании этой формулы можно определить пороговую среднюю длину фрагмента массива ключей — l^*, ниже которой рационально использовать алгоритм последовательного поиска:

$$l^* = \frac{n}{2^{k^*}} = \frac{2 \cdot a \cdot t_a \cdot \log_2 e}{b \cdot t_b} \approx 2{,}8853 \cdot \frac{a \cdot t_a}{b \cdot t_b} = 2{,}8853 \cdot L\,, \qquad (12.4.3)$$

где значение L задается соотношением

$$L = \frac{a \cdot t_a}{b \cdot t_b},$$

и определяется языком программирования, но слабо зависит от аппаратной среды реализации. В экспериментальном исследовании для задачи поиска элемента в структуре хранения разреженной глобальной матрицы, в языке реализации C++, было получено среднее значение этого отношения равное $2{,}94$ [12.8]. Таким образом, для выбранного языка программирования значение $l^* = 8{,}4827$ по формуле (12.4.3), но поскольку пороговая длина фрагмента массива представляет собой целое число, то оптимальная пороговая длина l_r равна 8 или 9 [12.8].

Принятие решения по переключению на алгоритм последовательного поиска может приниматься не только на основании значения l^*, но и на основании значения k^*. Отметим, что значение k^* в формуле (12.4.2) дает среднее значение числа шагов половинного деления и является действительным числом, но для принятия решения в программной реализации это число k_r должно быть целым:

$$k_r = \left[k^*\right], \text{ или } k_r = \left[k^*\right] + 1,$$

что, приводит к выбору пороговой длины в более широком диапазоне. Поскольку функция логарифма растет достаточно медленно, а возможный диапазон изменения количества не нулевых элементов в строке глобальной матрицы не велик, то значение k_r может быть фиксировано. Например, в диапазоне длин массива ключей от 32 до 56 значение $k_r = 2$. Проведенные на основе формулы (12.4.2) более детальные исследования, подтвержденные программными экспериментами, показывают, что с учетом целочисленности значения k_r, минимум функции $T(n,k)$ достигается при таких значениях k, которым соответствует значение l_r в диапазоне: $8 \le l_r \le 14$. Экспериментальные результаты по определению рациональной

пороговой границы для комбинированного алгоритма поиска приведены на рисунке 12.7. Язык реализации алгоритма — C++, процессор — P IV 1,5 Ггц, значение $t_{ср}$ — среднее время поиска элемента в структуре разреженной матрицы в наносекундах при заданной пороговой длине переключения l_r на алгоритм последовательного поиска. График построен по реальным данным для разреженной глобальной матрицы с количеством ненулевых элементов в строке равным 45 (данные из [12.8]).

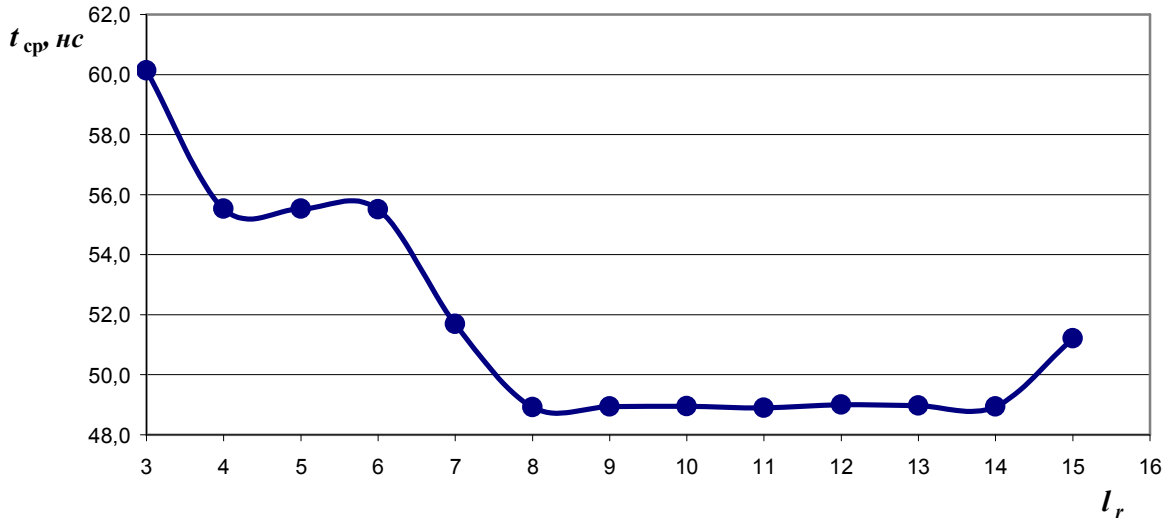


Рис. 12.7 Зависимость среднего времени поиска от пороговой длины переключения
на алгоритм последовательного поиска

С учетом теоретических и экспериментальных данных пороговое значение l_r для переключения на алгоритм последовательного поиска выбиралось в программной реализации из условия: $l_r \le 14$. Полученные в результате экспериментальных исследований данные позволяют говорить о снижении времени расчета в среднем для тепловых нестационарных задач на 20–30 минут при общем расчетном времени порядка 3–5 часов [12.8].

12.5 Рациональная организация аналитической базы данных с использованием алгоритмов решения задачи упаковки с динамической внутренней границей объема

Введение в проблему. Материал этого параграфа основан на статье [12.9], в которой рассматривается задача распределения ограниченного структурированно-

го ресурса с целью рационализации структуры хранения аналитической базы данных (БД). Задача о распределении ограниченного ресурса встречается, в том числе, и в области управления базами данных (БД). В классической постановке для распределения может быть использован весь доступный объем ресурса. Однако встречаются задачи, решение которых предполагает назначение внутренней структуры распределяемому ограниченному ресурсу. Простейший случай организации подобной внутренней структуры — разбиение доступного объема ресурса на квазистатический и динамический разделы, первый из которых предназначается для постоянного хранения наиболее предпочтительного набора объектов, а второй — для динамической загрузки или генерации редко запрашиваемых объектов. Одна из таких задач связана с рационализацией структуры хранения аналитической базы данных в оперативной памяти для индивидуальной информационной системы. В связи с этим рассмотрим особенности индивидуальных информационных систем и постановку задачи рациональной организации данных их аналитического компонента.

Системы управления базами данных (СУБД) индивидуальных информационных систем (ИИС) должны снабжаться функциями автоматизированного администрирования. В отсутствие подобной функциональности индивидуальный пользователь вынужден либо пользоваться услугами администратора базы данных, либо самостоятельно выполнять его роль, однако оба эти варианта для подавляющего большинства пользователей неприемлемы. В основе автоматизации администрирования БД лежит постоянный мониторинг состояния системы и данных. Модули мониторинга, введенные в состав СУБД, собирают информацию о динамике использования разноуровневой памяти как самой СУБД, так и о деятельности пользователя, выражающейся во введении, модификации, удалении и поиске данных, о деятельности модулей автоматического поиска информации в глобальной сети и т.д. На основе собранной информации система автоматизации администрирования БД (САА БД) может решать задачи повышения эффективности функционирования СУБД, в том числе, путем принятия оптимизационных решений, например, переноса редко используемых данных в более медленное хранилище, добавления индекса для ускорения выполнения одного из поисковых

запросов, и т. п. В ряду этих задач можно выделить задачу автоматизированного формирования аналитической подсистемы ИИС.

Как правило, в корпоративных информационных системах обработка текущих (транзакционных) данных и аналитическая обработка данных выполняются разными подсистемами, обладающими раздельными хранилищами данных, причем первые — реляционными, а вторые — либо реляционными, либо многомерными хранилищами. Известно, что при низких и средних объемах данных, эффективность применения многомерных хранилищ выше, чем реляционных [12.10]. Хранилище данных аналитической подсистемы проектируется исходя из планируемой на него нагрузки: выбор измерений для агрегирования выполняют системные аналитики, прогнозируя спектр будущих запросов. Обновление агрегированных данных в аналитической БД на основе данных транзакционной БД выполняется с некоторой периодичностью. Исходные данные после агрегирования удаляются из транзакционной БД и переносятся в архивную БД, обладающую структурой, аналогичной транзакционной БД. Но в ИИС отсутствует возможность упреждающего планирования структуры аналитической подсистемы, поэтому необходимо обеспечить возможность динамического выбора измерений для агрегирования в зависимости от типов запросов, выдаваемых пользователем и приложениями САА БД. По мере накопления статистики запросов, структура аналитической многомерной БД должна уточняться. В результате к некоторому моменту времени будет сформирована аналитическая БД, удовлетворяющая основные запросы пользователя и обеспечивающая возможности для эффективного функционирования приложений САА БД. Для обслуживания относительно редко возникающих запросов необходимо выполнять медленный и неэффективный поиск в архивной БД. Чтобы в этом случае снизить потери, можно использовать следующий подход.

Поскольку для хранения аналитической БД выделяется ограниченный объем оперативной памяти, а разным запросам требуется наличие данных, агрегированных по разным измерениям, то часть выделенной памяти можно отвести под хранение квазистатического, в смысле набора агрегируемых измерений, раздела многомерной БД. Оставшуюся память можно использовать для построения динамического раздела многомерной БД, хранящего агрегированные данные, исполь-

зованные в последнем «редком» запросе, который не мог быть удовлетворен за счет информации квазистатического раздела. Прирост эффективности при данном подходе достигается за счет того, что «редкие» запросы имеют тенденцию к группированию во времени [12.11]: часто после того, как был выдан «редкий» запрос с некоторыми параметрами, этот же запрос повторяется с другими параметрами. Если в процессе выполнения первого запроса подготовиться к выполнению последующих аналогичных запросов, построив в динамическом разделе многомерную БД по требуемым измерениям, время выполнения последующих запросов может быть сокращено. Если поступает другой «редкий» запрос, динамический раздел очищается и в нем строится новая многомерная БД, соответствующая новому запросу. Альтернативой могло бы служить сохранение содержимого динамического раздела, но при этом потребуется нести затраты, связанные с постоянным обновлением агрегатов по входящим в них измерениям, что, учитывая малую частоту обращений к этим измерениям, нерационально.

Содержательная постановка задачи. На основе вышеизложенного возникает задача разбиения доступного объема ресурса общей памяти, выделенной для хранения аналитической БД, и размещения в квазистатическом разделе агрегированных данных в соответствии со статистикой запросов. Анализ поставленной задачи показывает, что она принадлежит к классу задач об оптимальных назначениях с целочисленными параметрами. Точное решение задач этого класса может быть получено, например, на основе использования аппарата метода динамического программирования. Но в связи с относительной вычислительной сложностью метода представляет интерес определение условий, в которых получение точного решения было бы целесообразным.

Содержательная постановка задачи предполагает знание предпочтений, определение которых носит статистический характер. Например, в качестве таких предпочтений по выбору агрегируемых измерений для квазистатического раздела БД могут быть использованы счетчики обращений, накапливаемые модулем мониторинга в динамике функционирования ИИС. В смысле характеристики такой динамики возможно выделение следующих трех ситуаций по отношению к статистически определяемым значениям предпочтений:

— *предпочтения не определены*. Это ситуация начального этапа функционирования ИИС, когда информация о предпочтениях отсутствует или очень мала. В строгом смысле решение задачи в этой ситуации невозможно. Практически в этой ситуации происходит формирование квазистатического раздела из динамически генерируемых многомерных БД с одновременным набором статистики предпочтений.

— *система находится в процессе набора статистики предпочтений.* Это ситуация такого временного этапа функционирования, когда оценка устойчивости статистики не попадает в доверительный интервал. Такие оценки могут быть получены на основе определения минимального объема выборки по известным критериям математической статистики, например, на основе выборочного значения дисперсии, с использованием критерия Стьюдента [12.12]. В этой ситуации получение точного решения задачи нецелесообразно из-за статистической неточности исходных данных о предпочтениях. В качестве альтернативы возможно использование эвристических алгоритмов решения задачи о разбиении доступного объема ресурса общей памяти с меньшей вычислительной сложностью и получением рациональных результатов.

— *система находится в состоянии устойчивой статистики предпочтений.* В этой ситуации возможно точное решение задачи о разбиении доступного объема ресурса общей памяти аналитической БД. Полученные результаты могут лежать в основе организации оптимальной процедуры управления разбиением аналитической БД ИИС на квазистатический и динамический разделы с учетом предположения об устойчивости предпочтений.

Математическая постановка задачи. Рассмотрим общую постановку задачи упаковки, отражающую особенности рассмотренной выше задачи разбиения доступного объема ресурса общей памяти, выделенной для хранения аналитической БД. Пусть задано множество объектов:

$$Y = \{y_i\}, \quad \|Y\| = n, \quad y_i = \{v_i, p_i\}, \quad i = \overline{1,n},$$

где каждый объект y_i обладает целочисленным линейным размером — v_i, или «объемом» в общепринятых терминах задачи упаковки, и ценовой характеристикой — p_i, которая содержательно отражает практически значимые предпочтения

для загрузки данного объекта, например, выборочная частота встречаемости запроса. Целым числом задан так же основной объем упаковки — V. Реальная постановка задачи имеет смысл, если суммарный линейный размер (объем) объектов превышает основной объем упаковки, таким образом, должно быть выполнено следующее условие:

$$\sum_{i=1}^{n} v_i > V. \qquad (12.5.1)$$

Для описания подмножества объектов $\widetilde{M}$, загружаемых в объем V из множества Y, введем в рассмотрение следующий характеристический вектор:

$$X = \{x_i\},\ x_i \in \{0,1\},\ i = \overline{1,n}, \qquad (12.5.2)$$

где значение компонента вектора $x_i = 1$ соответствует принадлежности объекта y_i к подмножеству объектов $\widetilde{M} \subset Y$, загружаемых в объем V. Объекты, составляющие подмножество $\widetilde{M}$ должны максимизировать суммарное предпочтение. Поскольку после загрузки в основном объеме должно остаться свободное место для размещения максимального по размеру незагруженного объекта $y_i \in Y - \widetilde{M}$, то математическая постановка задачи имеет вид:

Максимизировать линейную форму:

$$P_n(X) = \sum_{i=1}^{n} x_i \cdot p_i \to \max, \qquad (12.5.3),$$

при выполнении следующего динамического, в смысле текущего набора загружаемых объектов, условия:

$$\sum_{i=1}^{n} x_i \cdot v_i + \max_{i=1,n} \left\{ (x_i + 1)_{\mathrm{mod}\,2} \cdot v_i \right\} \le V. \qquad (12.5.4)$$

Условие (12.5.4) не является типичным для классической задачи оптимальной по стоимости одномерной упаковки. В связи с динамическим изменением внутренней границы для загружаемых в основной объем объектов, в зависимости от размера максимального незагруженного объекта, будем называть в дальнейшем эту задачу — *задачей упаковки с динамической внутренней границей объема*.

Точный алгоритм решения задачи упаковки с динамической внутренней границей объема. Исследование задачи упаковки (12.5.3) с изменяющимся в

ходе решения граничным значением объема показало, что она может быть решена путем получения решений классической задачи упаковки для всех объемов v:

$$V_1 \le v \le V_2, \quad V_1 = V\text{- max } \{v_i\}, V_2 = V\text{- min } \{v_i\}, \ i = \overline{1,n},$$

которую будем в дальнейшем называть базовой, с последующим выбором той упаковки, которая удовлетворяет условию (12.5.4). Для решения базовой задачи в указанных объемах может быть использован табличный алгоритм для классической одномерной оптимальной по стоимости задачи упаковки (см. параграф 12.2), позволяющий получить набор оптимальных решений для всех промежуточных дискретных значений основного объема. Базовая задача упаковки есть задача максимизации линейной формы (12.5.3) при ограничениях:

$$\sum_{i=1}^{N} x_i \cdot v_i \ \le V. \qquad (12.5.5)$$

Обозначим вектор оптимальной упаковки объектов $y_j, j = \overline{1,k}$, являющийся решением базовой задачи для объема v через X_k^v, а через $f_k(v)$ — стоимость этой оптимальной упаковки в данном объеме. Решение табличным алгоритмом осуществляется на основе функционального уравнения метода динамического программирования, которое для базовой задачи имеет вид:

$$\begin{cases} f_0(v) = 0; \\ f_k(v) = \max\limits_{x_k} \{ x_k \cdot p_k + f_{k-1}(v - x_k \cdot v_k) \}, k = \overline{1,n}, v = \overline{0,V}, x_k \in \{0,1\}. \end{cases}$$

Результатом решения базовой задачи с ограничением (12.5.5) является набор оптимальных значений целевой функции $f_n(v)$ и соответствующих векторов оптимальной упаковки X_n^v для всех объемов v от V_1 до V_2 исходными объектами из Y. Отметим, что вычисления по дискретным значениям объема при $k = n$ могут быть сокращены, т. к. оптимальные упаковки для объемов от 0 до $(V_1 - 1)$ не содержат решений, интересующих нас в смысле условия (12.5.4).

Рассмотрим подзадачу определения максимального объема v^*, в котором выполняется условие (12.5.4). Поскольку базовая задача имеет целочисленный (по v_i и V) характер, то хотя в каждом решении для объема v: $V_1 \le v \le V_2$ и достигается максимум линейной формы (12.5.3), сам объем v может быть упакован не

плотно. В связи с этим рассмотрим специальную функцию $d(v)$, задающую остаток свободного пространства в оптимально упакованном объеме v:

$$d(v)=v-\sum_{i=1}^{n} x_i^v \cdot v_i, \quad x_i^v \in X_n^v. \tag{12.5.6}$$

Функция $d(v)$ неотрицательна в силу выполнения условия, аналогичного (12.5.5) для объема упаковки v при решении базовой задачи. Определим так же функцию $R(v)$, задающую необходимый остаток объема для размещения максимального по объему неупакованного элемента:

$$R(v)=\max\{v_i\},\ i: x_i^v=0,\ i=\overline{1,n}. \tag{12.5.7}$$

С использованием введенных обозначений выполнение ограничения (12.5.4) сводится к нахождению максимального объема v, такого, что:

$$V-v+d(v)\geq R(v). \tag{12.5.8}$$

Поскольку функция $f_n(v)$ является неубывающей в силу основного функционального уравнения, то должно быть выбрано наибольшее значение v, удовлетворяющее условию (12.5.8). Следовательно, оптимальное решение задачи упаковки с динамической внутренней границей объема задается тем вектором $X_n^{v^*}$, для которого значение v^* удовлетворяет условию:

$$\begin{cases} V-v^*+\ \ d(v^*) & \geq R(v^*); \\ V-(v^*+1)+d(v^*+1) & < R(v^*+1); \\ V_1\leq v^*\leq V_2. \end{cases} \tag{12.5.9}$$

Таким образом, предлагаемый алгоритм $A1$ является двух шаговым, при этом на первом шаге решается базовая задача упаковки (12.5.3) с ограничением (12.5.5), а на втором шаге определяется такая внутренняя динамическая граница упаковки, которая доставляет максимум линейной форме (12.5.3) при ограничении (12.5.4), путем нахождения v^* по условию (12.5.9).

Оценка сложности алгоритма $A1$ есть максимальная по асимптотике оценка сложности двух указанных шагов. Решение базовой задачи реализуется, с учетом бинарности значений компонент вектора X, двумя вложенными циклами — по объектам y_i, и по дискретам объема v. С учетом необходимости дублирования

строки таблицы, содержащей вектор X_k^v, имеем оценку для первого шага — $O(n^2V)$. Для второго шага вычисление функций $R(v)$ и $d(v)$ требует не более $O(n)$ операций для каждой проверки условия (12.5.9), которая выполняется не более чем $M = V_2 - V_1$ раз. Очевидно, что $M < V$ в силу определений V_2 и V_1, и в результате, алгоритм $A1$ имеет сложность $O(n^2V)$.

Определение рационального значения дискрета объема. Из оценки сложности алгоритма $A1$ — $O(n^2V)$ очевидно, что сокращение количества операций (при фиксированном n) может быть получено только за счет решения задачи выбора большего значения дискрета объема, уменьшающего численно значения v_i и V. Для точного решения этой задачи необходимо нахождение НОД $(V,v_1,v_2,...,v_n)$. Однако такой путь бесперспективен, т. к. при большом значении n вероятность того, что НОД $(V,v_1,v_2,...,v_n)=1$ близка к единице. Это предположение опирается на теорему Дирихле [12.13], утверждающую, что вероятность p того, что два случайно выбранных натуральных числа взаимно просты, равна

$$p = \frac{6}{\pi^2} \approx 0{,}607927 .$$

В качестве альтернативы рассмотрим следующий эвристический подход к определению рациональной единицы измерения объема в задаче упаковки. Пусть объемы v_i и V заданы целыми числами в некоторых единицах объема, которые будем считать минимально возможными в рамках реальной задачи. Предположим, что $\forall i = \overline{1,n} \quad v_i << V$, и обозначим через k коэффициент упаковки объема V объектами со средним значением объема:

$$k = \left[\frac{V}{\bar{v}}\right], \bar{v} = \frac{1}{N} \cdot \sum_{i=1}^{n} v_i ,$$

где $[a]$ — целая часть числа a. Введем в рассмотрение новую единицу измерения объема, со значением равным m минимальных единиц. Тогда значение объема объекта y_i в новых единицах равно:

$$v_i^{(m)} = \left\lceil \frac{v_i}{m} \right\rceil .$$

Обозначим через Δv_i приращение объема элемента y_i, выраженное в минимальных единицах, при переводе его в новую единицу измерения. Будем предполагать, что значение n велико, а распределение значений v_i по классам эквивалентности по модулю m равномерно. Тогда в среднем $\frac{k}{m}$ элементов y_i точно (без остатка) представляются в новых единицах измерения объема, а $k \cdot \left(1 - \frac{1}{m}\right)$ элементов y_i должны быть округлены до ближайшего большего целого с увеличением в среднем на $\frac{m}{2}$. В силу предположения о равномерности увеличение суммарного объема для k элементов, упакованных в объем V, составит в минимальных единицах:

$$\Delta v^+ = \sum_{i:x_i=1} \Delta v_i \approx k \cdot \left(1 - \frac{1}{m}\right) \cdot \frac{m}{2} = \frac{k \cdot (m-1)}{2} \ . \qquad (12.5.10)$$

Сам общий объем упаковки V при переводе в новые единицы объема будет увеличен в среднем также на $\frac{m}{2}$. Поскольку, в среднем, после упаковки k элементов, в объеме V останется свободное место размером $\frac{\bar{v}}{2}$ минимальных единиц, то увеличение общего доступного объема составит:

$$\Delta V^+ = \frac{m + \bar{v}}{2} . \qquad (12.5.11)$$

Для сохранения старой упаковки общий объем V должен быть увеличен на $\Delta V = \left(\Delta v^+ - \Delta V^+\right)$ минимальных единиц. Условие выбора рационального значения m может быть задано как процентное ($p\%$) ограничение увеличения общего объема ΔV:

$$\Delta V = \left(\Delta v^+ - \Delta V^-\right) \ \le p \cdot V, \quad p = \frac{p\%}{100}, \qquad (12.5.12)$$

Подставляя (12.5.10) и (12.5.11) в (12.5.12) окончательно имеем следующее ограничение на выбор значения m:

$$1 \le m \le \left[\frac{2 \cdot p \cdot V + \bar{v} + k}{(k-1)} \right]. \qquad (12.5.13)$$

Могут быть рассмотрены и другие подходы к определению рационального значения m. Например, в ситуации, когда коэффициент вариации для распределения v_i достаточно велик, можно положить

$$m = \min_{i=1,n} (v_i).$$

Из приведенной выше оценки сложности алгоритма $A1$, очевидно, что трудоемкость вычислений при переходе на новые единицы измерения объема уменьшится в m раз.

Эвристический алгоритм решения задачи упаковки с динамической внутренней границей объема. Использование эвристических алгоритмов упаковки [12.5] позволяет получить рациональное решение с лучшей временной эффективностью. Достаточно хорошие результаты дает эвристический алгоритм, использующий предварительную сортировку объектов по удельному предпочтению на единицу объема (см. оценку Грэхема в [12.5]). Упаковка осуществляется в порядке расположения объектов в отсортированной последовательности с пошаговой проверкой выполнения ограничений. Реализация ограничения (12.5.4) для такого алгоритма сводится к проверке возможности расположения максимального по объему незагруженного объекта в общий объем. Поскольку пошаговая загрузка осуществляется последовательно из отсортированного массива, то на шаге k должна быть выполнена проверка следующего условия:

$$V - \sum_{j=1}^{k-1} v_j + v_k \quad \geq \quad \max_{k+1 \leq i \leq n} \{v_i\}, \qquad (12.5.14),$$

где нумерация $i = \overline{1,n}$ соответствует нумерации в отсортированной последовательности исходных объектов в порядке убывания значений $\frac{p_i}{v_i}$. Если условие (12.5.14) не выполняется, начиная с некоторого значения $k+1$, то предыдущие упакованные объекты (с индексами 1,2,…, k) в объеме V образуют рациональную упаковку с динамической внутренней границей объема.

Реализация условия (12.5.14) требует перевычисления значения максимума на каждом шаге добавления объекта в $\widetilde{M}$. Определенное сокращение трудоемкости проверки этого условия может быть получено за счет пошагового накопления

суммы объемов упакованных элементов и более быстрому поиску максимума. Такой быстрый поиск может быть осуществлен на основе следующих рассуждений. Если на некотором шаге упаковки выбирается очередной объект с объемом, меньшим текущего максимума, то правая часть условия (12.5.14) может не пересчитываться. Для вычисления объема максимального неупакованного объекта введем в рассмотрение функцию $M_k(v)$:

$$\begin{cases} M_0(v) = \max\limits_{1 \le i \le n} \{ v_i \}; \\ M_k(v) = \begin{cases} M_{k-1}(v), & v_k < M_{k-1}(v); \\ \max\limits_{k+1 \le i \le n} \{ v_i \}, & v_k = M_{k-1}(v). \end{cases} \end{cases} \tag{12.5.15}$$

Тогда условие (12.5.14) принимает вид:

$$\begin{cases} V - V_\Sigma(k-1) + v_k^* \quad \ge \quad M_k(v), \quad V_\Sigma(k) = \sum\limits_{j=1}^{k} v_j; \\ V - V_\Sigma(k) + v_{k+1}^* \quad < \quad M_{k+1}(v). \end{cases} \tag{12.5.16}$$

Таким образом, предлагаемый эвристический алгоритм $A2$ является так же двух шаговым. На первом шаге выполняется сортировка исходных объектов в порядке убывания значений $\frac{p_i}{v_i}$. На втором шаге определяется внутренняя динамическая граница упаковки, обеспечивающая рациональную суммарную стоимость, путем нахождения значения v_k^* по условию (12.5.16), причем значения $M_k(v)$ вычисляются в соответствии с формулой (12.5.15).

Получим оценку сложности алгоритма $A2$ для среднего случая. Сортировка на первом шаге при больших значениях n может быть выполнена, например, методом пирамиды с оценкой в среднем случае $O(n \ln n)$ [12.5]. На втором шаге формирование вектора X требует не более k шагов и, следовательно, проверка условия (12.5.16) выполняется не более k раз, причем значение k не может превосходить n в силу постановки задачи. Покажем, что сложность второго шага не хуже, чем $O(n \ln n)$ в среднем. Вычисление левой части в условии (12.5.16) имеет сложность $O(1)$. Вычисление $M_0(v)$, выполняемое однократно, имеет сложность $O(n)$. В предположении о равномерном случайном распределении n значений v_i обычный алгоритм поиска максимума выполняет в среднем $O(\ln n)$ переприсваиваний

[12.14], т. е. смен текущего максимума. Таким образом, пересчет $M_k(v)$ будет выполнен в среднем, с учетом условия $k < N$, не более чем $O(\ln n)$ раз, в остальных случаях $M_k(v)$ вычисляется путем присваивания старого значения со сложностью $O(1)$. Сам пересчет максимума для вычисления нового значения $M_k(v)$ требует в соответствии с (12.5.16) не более $O(n)$ шагов. Объединяя полученные результаты, имеем оценку сложности в среднем для второго шага алгоритма в виде:

$$O(n) + n \cdot O(1) + O(\ln n) \cdot O(n) = O(n \ln n).$$

Таким образом, оценка сложности алгоритма $A2$ в целом составляет $O(n \ln n)$, что асимптотически значительно лучше, чем у алгоритма $A1$, но при этом алгоритм $A2$ позволяет получить рациональное, но не оптимально точное решение. Заметим также, что вычислительная сложность данного эвристического алгоритма не зависит от значений объемов объектов и общего объема упаковки.

Обсуждение результатов. В заключение необходимо отметить, что в содержательной постановке задачи оптимизация формирования квазистатического и динамического разделов структуры хранения аналитической БД ИИС производится на основе анализа статистической информации об обращениях к агрегированным данным по отдельным измерениям. С точки зрения принятия решения по выбору рациональных алгоритмов, рассмотренная задача служит примером того, что такие решения могут приниматься в зависимости от условий реальной эксплуатации программных реализаций алгоритмов. В данном случае таким условием является устойчивость статистики предпочтений, количественный показатель которой может быть определен методами математической статистики. В зависимости от значения этого показателя могут быть использованы два, схематично описанные выше, алгоритма решения задачи упаковки с динамической внутренней границей объема — точный и эвристический. Для точного алгоритма решения этой задачи дополнительная временная эффективность в программной реализации может быть достигнута на основе рационального выбора значения единицы измерения для основного объема упаковки и объемов упаковываемых объектов.

Таким образом, решение задачи выбора рациональных компьютерных алгоритмов на основе анализа их ресурсной эффективности может приниматься и с

учетом особенностей жизненного цикла программных средств и систем. Уважаемые читатели наверняка смогут привести еще целый ряд примеров, показывающих как некоторые изменения информационной среды, происходящие во время жизненного цикла программы, могут оказывать влияние на выбор рационального алгоритма решения задачи.

Задачи и упражнения к главе 12

12.1. Реализуйте программно комбинированный алгоритм сортировки и проведите эксперименты с программной реализацией с целью определения оптимального порога переключения на алгоритм сортировки вставками. Насколько полученный вами результат отличается от значения порога переключения ($k = 12$), полученного на основе анализа функций трудоемкости?

12.2. Определите оптимальный порог переключения на алгоритм сортировки вставками при реализации комбинированного алгоритма сортировки на различных языках программирования и различных компьютерах. Это достаточно интересное исследование позволит понять, как среда реализации алгоритма влияет на решения по рациональному выбору.

12.3. Получите самостоятельно функцию трудоемкости табличного алгоритма упаковки. Совпадает ли Ваш результат с формулой (12.2.5)?

12.4. Предложите структуру хранения поискового дерева решений для решения задачи коммивояжера методом ветвей и границ, предусматривающую хранение матрицы стоимости.

Список литературы к главе 12

[12.1.] Головешкин В. А., Ульянов М. В. Теория рекурсии для программистов. — М.: Издательство «Наука ФИЗМАТЛИТ», 2006. — 296 с.

[12.2.] Беллман Р., Дрейфус Р. Прикладные задачи динамического программирования: Пер. с англ. — М.: Наука, 1965, — 457 с.

[12.3.] 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.

[12.4.] Little J. D. C., Murty K. G., Sweeney D. W., and Karel C. An algorithm for the traveling salesman problem // Operations Research. v11 (1963), pp. 972–989.

[12.5.] Гудман С., Хидетниеми С. Ведение в разработку и анализ алгоритмов. — М.: Мир, 1981. — 368 с.

[12.6.] Ермошин А. С., Плиско В. А. Исследование дерева решений метода ветвей и границ в задаче коммивояжера // Программное и информационное обеспечение систем различного назначения на базе персональных ЭВМ: Межвузовский сборник научных трудов / Под ред. д. т. н., проф. Михайлова Б. М. — М.: МГАПИ, 2006. Вып. 9. С. 76–82.

[12.7.] Ульянов М. В., Александров А. Е. Общие подходы к повышению ресурсной эффективности алгоритмического обеспечения систем конечно-элементного анализа // Автоматизация и современные технологии. 2004. № 9. С. 18–24.

[12.8.] Ульянов М. В., Александров А. Е., Востриков А. А. Эффективные алгоритмы формирования глобальной матрицы для комплекса конечно-элементного анализа // Автоматизация и современные технологии. 2004. № 10. С. 32–36.

[12.9.] Ульянов М. В., Брейман А. Д. Рациональная организация данных аналитического компонента в индивидуальных информационных системах с использованием алгоритма упаковки с динамической внутренней границей объема // Автоматизация и современные технологии. 2004. № 12. С. 17–22.

[12.10.] Inmon W.H. Building the data warehouse: Third edition. – New York: John Wiley and Sons, 2002. – 412 pp.

[12.11.] Deshpande P.M., Ramasamy K., Shukla A., Naughton J.F. Caching multidimensional queries using chunks. // In Proceedings of ACM SIGMOD, June 1998. – pp. 259-270.

[12.12.] Гмурман В.Е. Теория вероятностей и математическая статистика: Учеб. пособие для вузов/ 9-е изд., стер. – М.: Высшая школа, 2003. – 479 с.: ил.

[12.13.] Ноден П., Китте К. Алгебраическая алгоритмика: Пер. с франц. – М.: Мир, 1999. – 720 с., ил.

[12.14.] Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. – М.: МЦНМО, 1999. – 960 с., 263 ил.