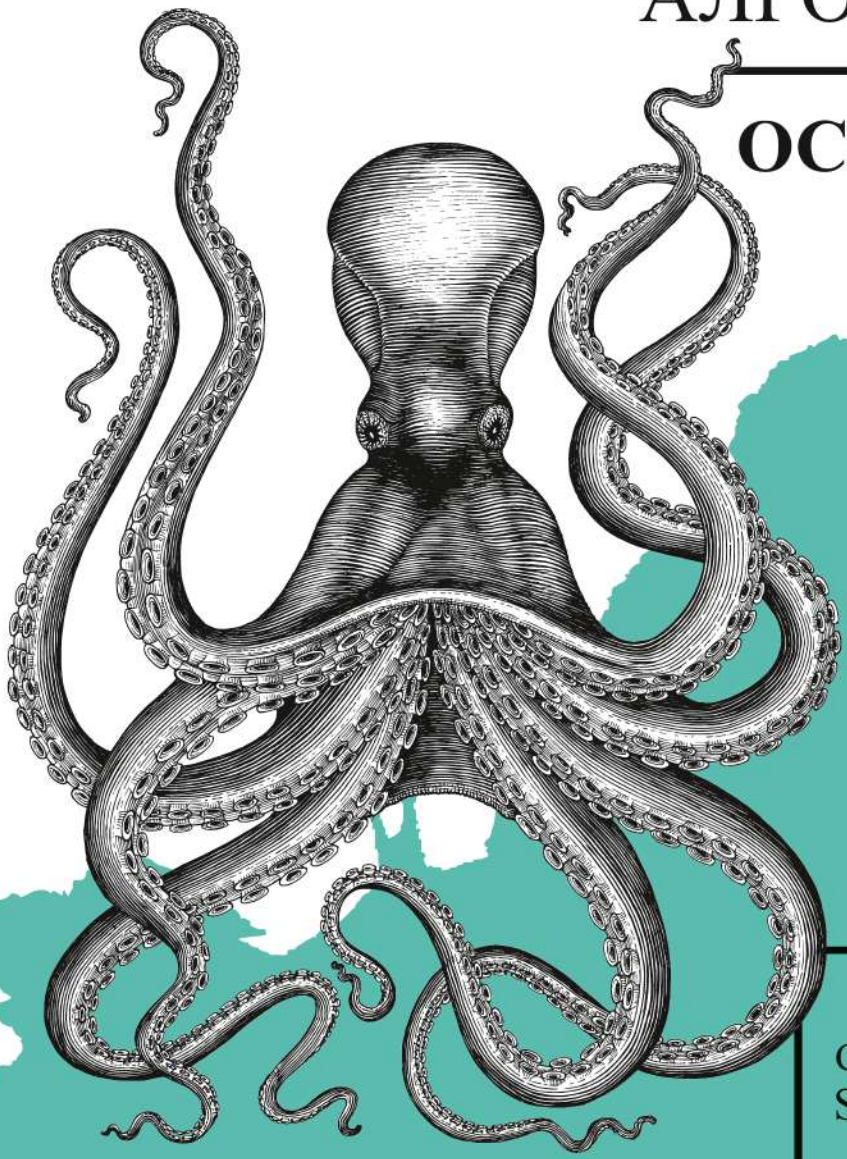
**ТИМ РАФГАРДЕН**

# СОВЕРШЕННЫЙ АЛГОРИТМ

## ОСНОВЫ



COMPUTER SCIENCE

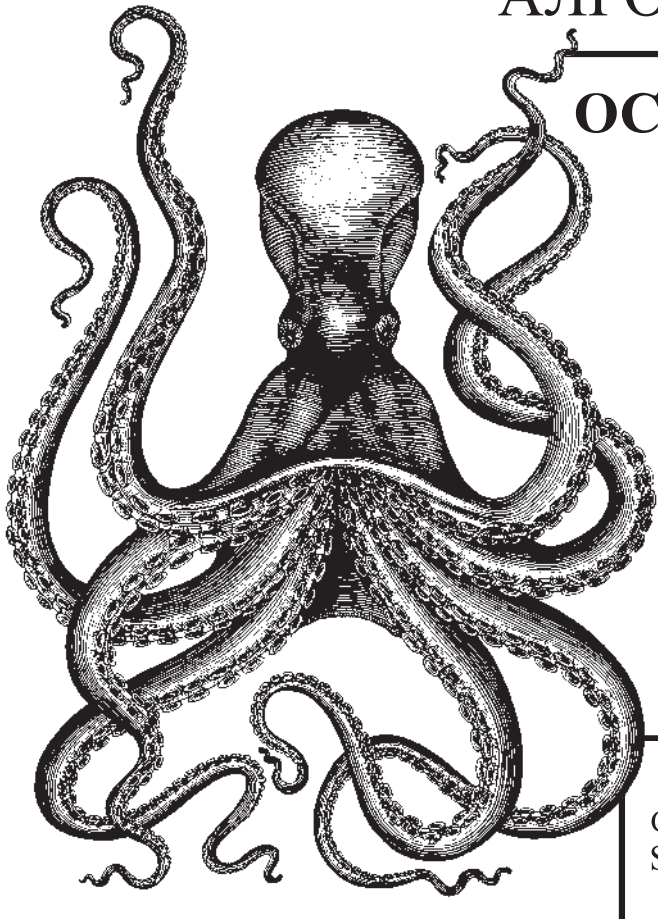
# Algorithms Illuminated
## Part 1: The Basics

Tim Roughgarden

**ТИМ РАФГАРДЕН**

# СОВЕРШЕННЫЙ АЛГОРИТМ

## ОСНОВЫ



COMPUTER SCIENCE

ПИТЕР®

**Санкт-Петербург • Москва • Екатеринбург • Воронеж**
**Нижний Новгород • Ростов-на-Дону**
**Самара • Минск**

2019

ББК 32.973.2-018
УДК 004.42
Р26

**Рафгарден Тим**

Р26 Совершенный алгоритм. Основы. — СПб.: Питер, 2019. — 256 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-0907-4

Алгоритмы — это сердце и душа computer science. Без них не обойтись, они есть везде — от сетевой маршрутизации и расчетов по геномике до криптографии и машинного обучения. «Совершенный алгоритм» превратит вас в настоящего профи, который будет ставить задачи и мастерски их решать как в жизни, так и на собеседовании при приеме на работу в любую IT-компанию. В этой книге Тим Рафгарден — гуру алгоритмов — расскажет об асимптотическом анализе, нотации большое-О, алгоритмах «разделяй и властвуй», рандомизации, сортировки и отбора. Книга «Совершенный алгоритм» адресована тем, у кого уже есть опыт программирования. Вы перейдете на новый уровень, чтобы увидеть общую картину, разобраться в низкоуровневых концепциях и математических нюансах.

Познакомиться с дополнительными материалами и видеороликами автора (на английском языке) можно на сайте www.algorithmsilluminated.org.

**16+** (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018
УДК 004.42



ISBN 978-0999282908 англ.
ISBN 978-5-4461-0907-4

# *Краткое содержание*

# *Оглавление*

*Посвящается Эмме*

# *Предисловие*

Перед вами первая из четырех частей книги, основанной на проводимых мною с 2012 года онлайн-курсах по алгоритмам. Эти курсы, в свою очередь, появились благодаря лекциям для студентов, которые я читал в Стэнфордском университете в течение многих лет.

## О чем эти книги

*Первая часть книги «Совершенный алгоритм»* закладывает основы понимания следующих четырех тем.

**Асимптотический анализ и математическое обозначение О-большое.** Система обозначений, используемая в математическом анализе асимптот, обеспечивает терминологическую базу для обсуждения процессов разработки и анализа алгоритмов. Ключевым понятием здесь является математическое обозначение О-большое, которое представляет собой вариант моделирования гранулярности, используемой для оценки времени исполнения алгоритма. Мы увидим, что «золотой серединой» для четкого и высокоуровневого понимания механизмов разработки алгоритмов является подход, основанный на игнорировании постоянных коэффициентов и членов более низкого порядка, а также концентрация на том, каким образом производительность алгоритма соотносится с размером входных данных.

**Алгоритмы «разделяй и властвуй» и Основной метод.** При разработке алгоритмов нет какой-либо «серебряной пули», отсутствует универсальный способ, который позволял бы запросто решать все вычислительные задачи. Вместе с тем существует несколько общих методов проектирования алгоритмов, которые находят успешное применение в различных областях.

В этой части книги мы рассмотрим методику «разделяй и властвуй». Ее идея заключается в разбиении задачи на ряд мелких подзадач, которые решаются рекурсивно. Затем полученные решения быстро объединяются в решение исходной задачи. Мы рассмотрим быстрые алгоритмы типа «разделяй и властвуй» для решения задач сортировки, умножения целых чисел и умножения матриц, а также для решения основной задачи вычислительной геометрии. Мы также рассмотрим Основной метод (или основную теорему о рекуррентных соотношениях), который является мощным инструментом анализа времени исполнения алгоритмов типа «разделяй и властвуй».

**Рандомизированные алгоритмы.** Рандомизированный алгоритм в ходе своего выполнения использует прием «подбрасывания монеты», и его поведение может зависеть от результатов этих подбрасываний. Удивительно часто рандомизация приводит к простым, изящным и практичным алгоритмам. Каноническим примером является рандомизированный алгоритм быстрой сортировки `QuickSort`. Мы детально рассмотрим как сам этот алгоритм, так и анализ продолжительности его исполнения. Дальнейшие примеры применения рандомизации будут рассматриваться во второй части книги.

**Сортировка и отбор.** В качестве побочного продукта изучения первых трех тем мы узнаем несколько знаменитых алгоритмов сортировки и отбора, включая сортировку слиянием `MergeSort`, быструю сортировку `QuickSort` и линейный отбор (как рандомизированный, так и детерминированный). Эти вычислительные примитивы настолько невероятно быстры, что они не занимают намного больше времени, чем требуется для чтения входных данных. Важно наращивать коллекцию таких «бесплатных примитивов» как для непосредственного применения к данным, так и для использования в качестве строительных блоков для решения более сложных задач.

Для более подробного изучения материала книги внимательно знакомьтесь с разделами «Выводы», которые завершают каждую главу и выделяют наиболее важные вопросы.

**Темы, которые рассматриваются в других трех частях.** *Во второй части книги* рассматриваются различные структуры данных (кучи, сбалансированные деревья поиска, хеш-таблицы, фильтры Блума), графовые примитивы (поиск в ширину и в глубину, связность, кратчайшие пути) и области их применения (от дедупликации до анализа социальных сетей). *Третья*

*часть* посвящена жадным алгоритмам (задачи планирования, определение минимального остовного дерева графа, кластеризация, коды Хаффмана), а также динамическому программированию («задача о рюкзаке», задачи выравнивания рядов, поиска кратчайших путей, построения деревьев оптимального поиска). *Четвертая часть* посвящена раскрытию класса NP-полных задач, их сложности для разработчиков алгоритмов, искусству решения алгоритмически неразрешимых задач, включая эвристический анализ и локальный поиск.

## Навыки, которые вы приобретете

Освоение алгоритмов требует времени и усилий. Ради чего все это?

**Возможность стать более эффективным программистом.** Вы изучите несколько невероятно быстрых подпрограмм для обработки данных и несколько полезных структур для организации данных, которые можете применять непосредственно в ваших собственных программах. Реализация и применение этих алгоритмов расширит и улучшит ваши навыки программирования. Вы также узнаете основные приемы разработки алгоритмов, которые актуальны для решения разнообразных задач в широких областях, получите инструменты для прогнозирования производительности этих алгоритмов. Такие «шаблоны» могут быть вам полезны для разработки новых алгоритмов решения задач, которые возникают в вашей собственной работе.

**Развитие аналитических способностей.** Алгоритмические описания, мыслительная работа над алгоритмами дают большой опыт. Посредством математического анализа вы получите углубленное понимание конкретных алгоритмов и структур данных, описанных в этой книге. Вы приобретете навыки работы с несколькими математическими методами, которые широко применяются для анализа алгоритмов.

**Алгоритмическое мышление.** Научившись разбираться в алгоритмах, трудно не заметить, что они окружают вас повсюду: едете ли вы в лифте, наблюдаете ли за стаей птиц, управляете ли вы своим инвестиционным портфелем или даже наблюдаете за тем, как учится ребенок. Алгоритмическое мышление становится все более полезным и распространенным в дис-

циплинах, не связанных с информатикой, включая биологию, статистику и экономику.

**Знакомство с величайшими достижениями информатики.** Изучение алгоритмов напоминает просмотр эффектного клипа с многочисленными суперхитами минувших шестидесяти лет развития информатики. Вы больше не будете чувствовать себя отстраненным на фуршете для специалистов в области информатики, когда кто-то отпустит шутку по поводу алгоритма Дейкстры. Прочитав эти книги, вы будете точно знать, что он имеет в виду.

**Успешность в собеседованиях.** На протяжении многих лет студенты развлекали меня рассказами о том, как знания, почерпнутые из этих книг, позволяли им успешно справляться с любым техническим вопросом, который им задавали во время собеседования.

## В чем особенность этой книги

Эта книга предназначена только для одного: *постараться научить основам алгоритмизации максимально доступным способом*. Воспринимайте ее как конспект лекций, которые опытный наставник по алгоритмам будет давать вам на протяжении серии индивидуальных уроков.

Существует ряд прекрасных, гораздо более традиционных и энциклопедически выверенных учебников по алгоритмам. Любой из них с пользой украсит эту серию книг дополнительными деталями, задачами и темами. Хотелось бы, чтобы вы поискали и нашли что-то избранное среди этих книг для себя. Кроме того, есть книги, которые ориентируются на программистов, ищущих готовые реализации алгоритмов на конкретном языке программирования. Множество соответствующих примеров также находятся в свободном доступе в интернете.

## Для кого эта книга?

Весь смысл этой книги, как и онлайн-курсов, на основе которых она создана, — быть широко и легко доступной настолько, насколько это возможно. На моих онлайн-курсах широко представлены люди всех возрастов, профессионального опыта и слоев общества, есть немало учащихся и студентов, разработчиков программного обеспечения (как состоявшихся, так и начинающих), ученых и профессионалов со всех уголков мира.

Эта книга не является введением в программирование, и было бы просто идеально, если бы вы уже обладали основными навыками программирования на каком-либо распространенном языке (например, Java, Python, C, Scala, Haskell). Чтобы пройти «лакмусовый» тест, обратитесь к разделу 1.4 — если там все понятно, значит, в остальной части книги у вас все будет в порядке. Если вам требуется развить свои навыки программирования, то для этих целей есть несколько прекрасных бесплатных онлайн-курсов, обучающих основам программирования.

По мере необходимости мы также используем математический анализ, чтобы разобраться в том, как и почему алгоритмы действительно работают. Свободно доступные конспекты лекций «Математика для Computer Science» под авторством Эрика Лемана и Тома Лейтона являются превосходным и освежающим память пособием по системе математических обозначений (например, $\sum$ и $\forall$), основам теории доказательств (метод индукции, доказательство от противного и др.), дискретному распределению вероятностей и многому другому[1]. Краткие обзоры метода индукции и дискретного распределения вероятностей изложены в приложениях А и Б соответственно. Наиболее математически нагруженные разделы помечены звездочками. Если у вас «страх математики» или вы ограничены во времени, можно пропустить их при первом чтении без потери непрерывности в изложении материала.

[1] См. Mathematics for Computer Science, Eric Lehman, Tom Leighton: http://www.boazbarak.org/cs121/LehmanLeighton.pdf

# Дополнительные ресурсы

Эта книга основана на онлайн-курсах, которые в настоящее время запущены в рамках проектов Coursera и Stanford Lagunita. Имеется также ряд ресурсов в помощь вам для повторения и закрепления опыта, который можно извлечь из онлайн-курсов.

**Видео.** Если вы больше настроены смотреть и слушать, чем читать, обратитесь к материалам с «Ютуба», доступным на сайте www.algorithmsilluminated.org. Эти видео затрагивают все темы этой серии книг. Надеюсь, что они пропитаны тем же заразительным энтузиазмом в отношении алгоритмов, который, увы, невозможно полностью воспроизвести на печатной странице.

**Тестовые задания.** Как узнать, что вы действительно усваиваете понятия, представленные в этой книге? Тестовые задания с решениями и объяснениями разбросаны по всему тексту; когда вы сталкиваетесь с одним из них, призываю вас остановиться и подумать об ответе, прежде чем читать далее.

**Задачи в конце главы**. В конце каждой главы вы найдете несколько относительно простых вопросов для проверки усвоения материала, а затем более трудные и менее ограниченные по времени сложные задачи. Решения этих задач в книгу не включены, но для этого читатели могут обратиться ко мне или взаимодействовать между собой через дискуссионный форум книги (см. ниже).

**Задачи по программированию.** В конце большинства глав предлагается реализовать программный проект, целью которого является закрепление детального понимания алгоритма путем создания его рабочей реализации. Наборы данных, а также тестовые примеры и их решения можно найти на www.algorithmsilluminated.org.

**Дискуссионные форумы.** Существенной причиной успеха онлайн-курсов являются реализованные через дискуссионные форумы возможности общения для слушателей. Это позволяет им помогать друг другу в лучшем усвоении материала курса, а также отлаживать свои программы. Читатели этих книг имеют такую же возможность благодаря форумам, доступным на сайте www.algorithmsilluminated.org.

# Благодарности

Эта книга не появилась бы без того энтузиазма и интеллектуального голода, которые демонстрируют тысячи слушателей моих курсов по алгоритмам на протяжении многих лет как на кампусе в Стэнфорде, так и на онлайн-платформах. Я особенно благодарен тем, кто предоставлял подробные отзывы на более ранний проект этой книги, среди них: Тоня Бласт, Юань Цао, Джим Хьюмелсайн, Байрам Кулиев, Патрик Монкелбэн, Кайл Шиллер, Ниссэнка Викермэзинг и Дэниел Зингаро.

Конечно же, всегда приятно получать замечания и предложения от читателей, которые наилучшим образом доходят через упомянутые выше дискуссионные форумы.

*Стэнфордский университет*
*Тим Рафгарден*
*Стэнфорд, Калифорния*
*Сентябрь 2017*

# 1 Введение

Цель этой главы — зародить в вас интерес к изучению алгоритмов. Начав с рассмотрения алгоритмов в целом, мы постараемся выяснить, почему они так важны. Затем мы воспользуемся задачей умножения двух целых чисел, чтобы проиллюстрировать, как алгоритмическая изобретательность может улучшить более простые или, казалось бы, очевидные на первый взгляд решения. Далее мы подробно рассмотрим алгоритм сортировки слиянием `MergeSort`, что важно по нескольким причинам. Во-первых, это широко применяемый на практике и знаменитый алгоритм, который необходимо знать. Во-вторых, его обзор станет неплохой разминкой при подготовке к изучению более изощренных алгоритмов. И наконец, этот алгоритм является классическим введением в методологию проектирования алгоритмов типа «разделяй и властвуй». Главу завершает описание ряда руководящих принципов нашего подхода к анализу алгоритмов, которые будут применяться в оставшейся части книги.

## 1.1. Зачем изучать алгоритмы?

Позвольте мне сначала обосновать актуальность этой книги, указав несколько причин, почему так важно быть мотивированным в изучении алгоритмов. Так что же такое алгоритм, в конце концов? Это набор четко сформулированных правил, в сущности, рецепт для решения некоторой вычислительной задачи. Может быть, у вас имеется куча чисел и вы хотите перераспределить их так, чтобы расположить их в отсортированном порядке. Возможно, у вас есть дорожная карта и необходимо вычислить кратчайший путь от некоторой исходной точки до определенного места назначения. Вполне вероятно, вам требуется выполнить несколько задач до наступления установленных сроков, в таком случае вы заинтересованы в упорядочении выполнения этих задач, для того чтобы вовремя все их завершить.

Итак, зачем же изучать алгоритмы?

**Важность для всех отраслей computer science.** Во-первых, понимание основ алгоритмизации и тесно с ней взаимосвязанной сферы организации структур данных необходимо для выполнения серьезной работы практически в любой отрасли информатики. Например, в Стэнфордском университете для полу-

чения любой степени по Computer Science (будь то бакалавр, магистр наук и даже доктор наук) обязательно требуется пройти курс алгоритмов. Приведу всего несколько примеров:

1. Протоколы маршрутизации в коммуникационных сетях задействуют классические алгоритмы поиска кратчайшего пути.
2. Криптография с открытым ключом опирается на эффективные теоретико-числовые алгоритмы.
3. Компьютерная графика требует вычислительных примитивов, предоставляемых геометрическими алгоритмами.
4. Индексация в базах данных опирается на структуры данных сбалансированных деревьев поиска.
5. Вычислительная биология использует алгоритмы динамического программирования для измерения сходства геномов.

Приведенный список можно продолжать.

**Двигатель технологических инноваций.** Во-вторых, алгоритмы играют ключевую роль в современных технологических инновациях. Приведу только один очевидный пример: поисковые системы используют целую мозаику алгоритмов для эффективного вычисления релевантности различных вебстраниц заданному поисковому запросу. Наиболее известным подобным алгоритмом является алгоритм `PageRank`, используемый в настоящее время Google. В самом деле, в докладе за декабрь 2010 года для Белого дома США президентский консультативный совет на науке и технике написал следующее:

> *«Все знают Закон Мура — предсказание, сделанное в 1965 г. соучредителем Intel Гордоном Муром о том, что плотность транзисторов в интегральных схемах будет удваиваться каждые 1–2 года... Во многих областях прирост производительности за счет улучшения алгоритмов значительно превысил даже впечатляющий рост производительности за счет увеличения скорости процессоров»*[1].

[1] Выдержка из доклада президенту и конгрессу: Проектирование цифрового будущего, декабрь 2010 г. (с. 71).

**Новый взгляд на достижения других наук.** В-третьих (хотя это выходит за рамки этой книги), алгоритмы все более и более используются в качестве «увеличительного стекла», чтобы по-новому взглянуть на научные проблемы за пределами computer science и IT. Например, исследование квантовых вычислений обеспечило новый вычислительный взгляд на квантовую механику. Ценовые колебания на экономических рынках могут плодотворно рассматриваться как алгоритмический процесс. Даже эволюцию можно рассматривать как удивительно эффективный алгоритм поиска.

**Гимнастика для мозга.** Во времена студенчества моими любимыми всегда были сложные предметы. После их освоения всегда оставалось ощущение, что я становился на несколько пунктов IQ умнее, чем в начале изучения. Надеюсь, моя книга позволит вам получить аналогичный опыт.

**Удовольствие!** Я надеюсь, что к концу этой книги вы поймете, почему разработка и анализ алгоритмов просто приносит удовольствие! Это увлекательное занятие, которое требует редкого сочетания точности и креативности. Разумеется, временами эта работа может разочаровывать, но она также очень затягивает. И давайте не будем забывать о том, что обучение алгоритмам, по сути, сопровождает вас с самого детства.

## 1.2. Целочисленное умножение

### 1.2.1. Задачи и решения

Когда вы учились в начальной школе, вы, скорее всего, изучали умножение двух чисел в столбик, что, по сути, является алгоритмом — четко сформулированным набором правил для преобразования входа (два числа) в выход (их произведение). Всегда важно понимать разницу между постановкой (описанием) *решаемой задачи* и описанием *метода ее решения* (то есть алгоритма этой задачи). В этой книге мы будем систематически придерживаться схемы, в которой сначала производится постановка вычислительной задачи (данные на входе и требуемый результат на выходе), а затем дается описание одного или нескольких алгоритмов ее решения.

### 1.2.2. Задача целочисленного умножения

В задаче целочисленного умножения входными данными являются два $n$-разрядных числа, обозначим их $x$ и $y$. Разрядность (длина) $n$ чисел $x$ и $y$ может быть любым положительным целым числом. Однако я призываю вас оперировать большими значениями $n$, в тысячах или более[1]. (Представьте, что вы разрабатываете некое криптографическое приложение, ведь они манипулируют очень большими числами.) В задаче целочисленного умножения требуемым выходом является результат произведения $x \times y$.

**ЗАДАЧА. ЦЕЛОЧИСЛЕННОЕ УМНОЖЕНИЕ**

**Вход:** два $n$-значных неотрицательных целых числа, $x$ и $y$.

**Выход:** произведение $x \times y$.

### 1.2.3. Алгоритм начальной школы

Точно определив вычислительную задачу, мы опишем алгоритм, который ее решает, — тот самый алгоритм, который вы изучали в начальной школе. Мы оценим производительность этого алгоритма числом «примитивных операций», которые он выполняет, в виде функции от количества знаков $n$ в каждом входном числе. Пока же давайте представим примитивную операцию как любую из следующих: (i) сложение двух одноразрядных ($n = 1$) чисел; (ii) умножение двух одноразрядных чисел или (iii) добавление нуля к началу или концу числа.

Чтобы освежить вашу память, рассмотрим конкретный пример умножения $x = 5678$ на $y = 1234$ (здесь $n = 4$) в столбик, см. рис. 1.1. Сначала алгоритм вы-

[1] Если вы хотите перемножить числа с разными длинами (например, 1234 и 56), просто добавьте несколько нулей в начало меньшего числа (например, рассматривайте 56 как 0056). С другой стороны, алгоритмы, которые мы обсудим, могут быть приспособлены для чисел с разной длиной.

числяет «частичное произведение» первого числа и последней цифры второго числа: 5678 × 4 = 22 712. Вычисление этого частичного произведения сводится к умножению каждой цифры первого числа на 4, записи младшего разряда результата, запоминанию («переносу» на следующий этап) старшего разряда и добавлению этих «переносов» (если они есть) на следующем умножении[1]. При вычислении следующего частичного произведения (5678 × 3 = 17 034) мы делаем то же самое, сдвигая результат на один знак влево (фактически добавляя «0» в конце). И так далее для оставшихся двух частичных произведений. Заключительный шаг состоит в том, чтобы сложить все частичные произведения.

Тогда в третьем классе вы, вероятно, согласились, что этот алгоритм является правильным, имея в виду, что неважно, с каких чисел $x$ и $y$ начинать. При условии, что все промежуточные вычисления выполняются правильно, алгоритм в конечном итоге заканчивается получением результата произведения $x \times y$ двух исходных чисел.

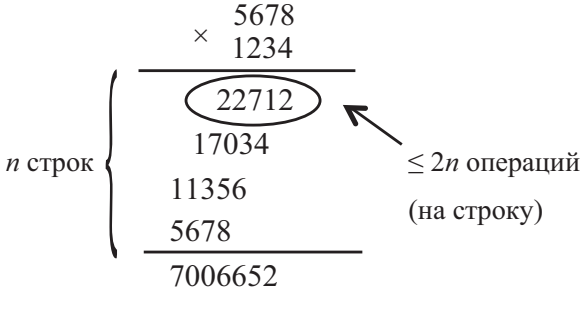



**Рис. 1.1.** Алгоритм целочисленного умножения в столбик

Таким образом, вы никогда не получите неправильный ответ, и алгоритм не может зациклиться.

[1] 8 × 4 = 32, 2 пишем, 3 «переносим», 7 × 4 = 28, плюс 3 равняется 31, 1 пишем, 3 переносим, и так далее...

### 1.2.4. Анализ числа операций

Ваш школьный учитель, возможно, не обсуждал число примитивных операций, необходимых для завершения процедуры умножения в столбик. В нашем примере, для того чтобы вычислить первое частичное произведение, мы умножили 4 раза каждую из цифр 5, 6, 7, 8 первого числа. Это четыре примитивные операции. Мы также выполнили несколько сложений из-за переносов. В общем случае вычисление частичного произведения влечет за собой $n$ умножений (одно умножение на один знак) и не более $n$ сложений (не более одного на один знак). Всего получается не более $2n$ примитивных операций. Первое частичное произведение ничем не отличается от других, и каждое из них требует не более $2n$ операций. Поскольку имеется $n$ частичных произведений — по одному на каждый знак второго числа, — вычисление всех из них требует не более $n \times 2n = 2n^2$ примитивных операций. Чтобы вычислить окончательный ответ, нам все еще нужно все частичные произведения сложить вместе, но для этого требуется сопоставимое число операций (не более чем еще $2n^2$). Подведем итоги:

$$\textit{общее количество операций} \leq \underbrace{\textit{константа}}_{=4} \times n^2.$$

Рассуждая о том, каким образом объем работы, который этот алгоритм выполняет, *возрастает* по мере того, как исходные множители становятся все длиннее и длиннее, мы видим, что объем выполняемых операций растет квадратически, следуя за увеличением разрядности. Если удвоить длину исходных множителей, то требуемый объем операций подскакивает в 4 раза. Увеличьте их длину в 4 раза, и она подскочит в 16 раз, и так далее.

### 1.2.5. Можно ли добиться лучшего?

В зависимости от того, каким третьеклассником вы были, вы вполне могли принять эту процедуру как уникальный или, по крайней мере, оптимальный способ умножения двух чисел. Если вы захотите стать серьезным проектировщиком алгоритмов, то нужно будет преодолеть эту робость. Классическая книга по алгоритмам Ахо, Хопкрофта и Ульмана, после итеративного рассмотрения целого ряда методик разработки алгоритмов, говорит следующее:

*«Для хорошего разработчика алгоритмов, пожалуй, самый важный принцип состоит в том, чтобы отказаться от соглашательства»*[1].

Или как я люблю говорить, что каждый разработчик алгоритмов должен принять как должное:

*Можно ли добиться лучшего?*

Этот вопрос особенно актуален, когда вы сталкиваетесь с очевидным или прямым решением вычислительной задачи. В третьем классе вы, возможно, не задавались вопросом о том, можно ли выстроить путь решения лучше, чем это делает простой классический алгоритм умножения в столбик. Настало время задать этот вопрос и дать на него ответ.

## 1.3. Умножение Карацубы

Разработка алгоритмов — удивительно разносторонняя сфера. Безусловно, существуют другие интересные методы умножения двух целых чисел, помимо того, что вы изучали в третьем классе. В этом разделе описывается один из таких методов, под названием *умножение Карацубы*[2].

### 1.3.1. Конкретный пример

Чтобы почувствовать, что такое умножение Карацубы, давайте снова воспользуемся нашим предыдущим примером с $x = 5678$ и $y = 1234$. Мы выполним последовательность шагов, совершенно отличающуюся от алгоритма начальной школы (умножения в столбик), результатом которого является произведение $x \times y$. Этот новый алгоритм должен показаться вам очень загадочным, похожим на фокус вынимания кролика из шляпы. Позже в этом разделе мы дадим точное объяснение того, что такое умножение Карацубы и почему оно работает.

---

[1] См.: А. Ахо, Дж. Хопкрофт, Дж. Ульман. Построение и анализ вычислительных алгоритмов. М.: Мир, 1979 (Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974, page 70).

[2] Обнаружен в 1960 году Анатолием Карацубой, который в то время был 23-летним студентом.

Сейчас главное — на этом примере оценить, что существует потрясающий воображение массив различных вариантов решения вычислительных задач, в частности целочисленного умножения.

Для начала обозначим первую и вторую половины числа $x$ как $a$ и $b$, чтобы рассматривать их отдельно. Для нашего примера получаем $a = 56$ и $b = 78$. Аналогичным образом поступим с $y$, здесь пусть $c$ и $d$ соответственно обозначают первую и вторую половины числа $y$, то есть $c = 12$ и $d = 34$ (рис. 1.2).

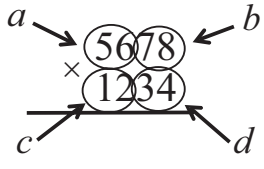



**Рис. 1.2.** Представление четырехзначных чисел как пар двузначных чисел

Затем мы выполним последовательность операций с участием только двузначных чисел *a, b, c* и *d* и, наконец, волшебным образом соберем все элементы вместе. Увидим, что в итоге это приведет нас к результату произведения $x$ и $y$.

**Шаг 1:** Вычислить произведение $a \times c = 56 \times 12$, которое составляет 672 (можете проверить сами).

**Шаг 2:** Вычислить $b \times d = 78 \times 34 = 2652$.

Следующие два шага еще более непостижимые.

**Шаг 3:** Вычислить $(a + b) \times (c + d) = 134 \times 46 = 6164$.

**Шаг 4:** Вычесть результаты первых двух шагов из результата третьего шага: $6164 – 672 – 2652 = 2840$.

Наконец, мы суммируем результаты шагов 1, 2 и 4, но только после добавления четырех конечных нулей к ответу на шаге 1 и двух конечных нулей к ответу на шаге 4.

**Шаг 5:** Вычислить $672 \times 10^4 + 2840 \times 10^2 + 2652 = 6\,720\,000 + 284\,000 + 2652 = 70\,066\,552$.

Что мы видим? Это точно такой же (правильный) результат, который получился у нас при применении алгоритма умножения в столбик в разделе 1.2!

Я не требую от вас сразу интуитивно понять то, что только что произошло. Скорее, я надеюсь, что вы ощущаете некое сочетание озадаченности и интриги, и оценили тот факт, что, похоже, действительно существуют алгоритмы целочисленного умножения, принципиально отличающиеся от того, с которым вы познакомились еще в школе. Как только вы поймете, насколько сфера алгоритмизации разнообразна, вы начнете задумываться над вопросом: может быть, существует что-то более интересное, чем алгоритм умножения в столбик? Разве приведенный выше алгоритм не позволил вам продвинуться в этом понимании?

### 1.3.2. Рекурсивный алгоритм

Прежде чем вплотную заняться умножением Карацубы, давайте немного подробнее рассмотрим, в чем суть простого рекурсивного подхода к целочисленному умножению[1]. Рекурсивный алгоритм целочисленного умножения, очевидно, связан с умножением чисел с меньшим, чем у исходных, количеством знаков (например, 12, 34, 56 и 78 в приведенном выше примере).

В общем случае число $x$ с четным количеством знаков $n$ может быть выражено в виде двух $n/2$-значных чисел (его первой и второй половины, которые мы обозначили как $a$ и $b$):

$$x = 10^{n/2} \times a + b.$$

Аналогично для числа $y$:

$$y = 10^{n/2} \times c + d.$$

[1] Я исхожу из того, что вы слышали о рекурсии в рамках вашего опыта программирования. Рекурсивная процедура — это процедура, которая вызывает саму себя как подпрограмму с меньшими по объему входными данными до тех пор, пока не будет достигнут базовый случай.

Чтобы вычислить произведение $x$ и $y$, воспользуемся двумя приведенными выше представлениями чисел $x$ и $y$ и просто перемножим их. Получаем

$$x \times y = (10^{n/2} \times a + b) \times (10^{n/2} \times c + d) = \\ = 10^n \times (a \times c) + 10^{n/2} \times (a \times d + b \times c) + b \times d. \quad (1.1)$$

Обратите внимание на то, что все умножения в (1.1) либо происходят между парами $n/2$-значных чисел, либо связаны с возведением в степень числа 10[1].

Выражение (1.1) позволяет лучше понять рекурсивный подход к умножению двух чисел. Чтобы получить произведение $x \times y$, мы производим расчет формулы (1.1). Во всех четырех присутствующих в ней произведениях ($a \times c$, $a \times d$, $b \times c$ и $b \times d$) задействуются числа с количеством знаков меньше $n$, поэтому мы можем вычислить каждое из них рекурсивно. Как только наши четыре рекурсивных вызова возвращаются к нам со своими результатами, мы можем вычислить выражение (1.1) в явном виде. Приставляем $n$ конечных нулей к произведению $a \times c$, суммируем произведения $a \times d$ и $b \times c$ (обычное арифметическое сложение) и приставляем $n/2$ конечных нулей к этой сумме, и, наконец, прибавляем эти два выражения к произведению $b \times d$[2]. Теперь давайте обобщим этот алгоритм, который мы назовем `RecIntMult`, в следующем псевдокоде[3].

---

[1] Для простоты мы принимаем допущение, что $n$ является степенью числа 2. Простой прием, который позволяет использовать это допущение, — добавление соответствующего количества нулей к $x$ и $y$, что не более чем удваивает разрядность этих чисел. С другой стороны, когда $n$ — нечетно, всегда можно также разбить $x$ и $y$ на два числа с почти равными длинами.

[2] Рекурсивные алгоритмы, естественно, нуждаются в одном или нескольких базовых случаях для того, чтобы они не зацикливались до бесконечности. Здесь базовый случай: если $x$ и $y$ являются однозначными числами, то необходимо просто перемножить их одной примитивной операцией и выдать результат, закончив алгоритм.

[3] В псевдокоде мы используем символ = для обозначения проверки на эквивалентность и символы := для обозначения присваивания переменной.

**RECINTMULT**

**Вход**: два $n$-значных положительных целых числа, $x$ и $y$.

**Выход**: произведение $x \times y$.

**Допущение**: $n$ является степенью числа 2.

```
if n = 1 then                                    // базовый случай
```
вычислить $x \times y$ за один шаг и выдать результат
```
else                                             // рекурсивный случай
```
$a$, $b$ := первая и вторая половины $x$

$c$, $d$ := первая и вторая половины $y$

рекурсивно вычислить $ac := a \times c$, $ad := a \times d$,

$bc := b \times c$ и $bd := b \times d$

вычислить $10^n \times ac + 10^{n/2} \times (ad + bc) + bd$,

используя арифметическое сложение, и выдать результат.

Является ли алгоритм `RecIntMult` более быстрым, чем алгоритм умножения в столбик? Конечно, можно обратиться к своей интуиции, но лучше все же подождать до главы 4, там вы найдете точный ответ на этот вопрос.

## 1.3.3. Умножение Карацубы

Умножение Карацубы является оптимизированной версией алгоритма `RecIntMult`. Мы снова начинаем с разложения, согласно формуле (1.1), множителей произведения $x \times y$ на составляющие $a$, $b$, $c$ и $d$. В алгоритме `RecIntMult` используется четыре рекурсивных вызова, один для каждого произведения в (1.1) между $n/2$-значными числами. Но нас совсем не интересуют $a \times d$ или $b \times c$ по отдельности, нас интересует их сумма $a \times d + b \times c$. При наличии всего трех промежуточных вычислений, в которых мы заинтересованы — $a \times c$, $a \times d + b \times c$ и $b \times d$, — сможем ли мы обойтись всего тремя рекурсивными вызовами?

Чтобы убедиться в этом, сначала, как и прежде, следует применить два рекурсивных вызова, чтобы вычислить $a \times c$ и $b \times d$.

**Шаг 1:** Рекурсивно вычислить $a \times c$.

**Шаг 2:** Рекурсивно вычислить $b \times d$.

Вместо того чтобы рекурсивно вычислять $a \times d$ или $b \times c$, мы рекурсивно вычисляем произведение между $a + b$ и $c + d$[1].

**Шаг 3:** Вычислить $a + b$ и $c + d$ (обычное арифметическое сложение), и рекурсивно вычислить произведение $(a + b) \times (c + d)$.

Ключевой прием умножения Карацубы обязан математику XIX века Карлу Фридриху Гауссу, который изучал умножение комплексных чисел. Вычитание результатов первых двух шагов из результата третьего шага дает именно то, что мы хотим: срединный коэффициент в (1.1) — выражение $a \times d + b \times c$:

$$\underbrace{(a+b)\times(c+d)}_{=a\times c+a\times d+b\times c+b\times d} - a\times c - b\times d = a\times d + b\times c.$$

**Шаг 4:** Вычесть результаты первых двух шагов из результата третьего шага, чтобы получить $a \times d + b \times c$.

Заключительный шаг представляет собой формулу (1.1), как и в алгоритме `RecIntMult`.

**Шаг 5:** Вычислить формулу (1.1), сложив результаты шагов 1, 2 и 4, после добавления $10^n$ конечных нулей к ответу на шаге 1 и $10^{n/2}$ конечных нулей к ответу на шаге 4.

[1] Числа $a + b$ и $c + d$ могут иметь до $(n/2) + 1$ знаков, но алгоритм по-прежнему работает отлично.

**KARATSUBA**

**Вход**: два $n$-значных положительных целых числа, $x$ и $y$.

**Выход**: произведение $x \times y$.

**Допущение**: $n$ является степенью числа 2.

```
if n = 1 then                                    // базовый случай
```
вычислить $x \times y$ за один шаг и выдать результат
```
else                                             // рекурсивный случай
```
$a$, $b$ := первая и вторая половины $x$

$c$, $d$ := первая и вторая половины $y$

вычислить $p := a + b$ и $q := c + d$, используя

арифметическое сложение,

рекурсивно вычислить $ac := a \times c$, $bd := b \cdot d$ и

$pq := p \times q$

вычислить $adbc := pq - ac - bd$, используя

арифметическое сложение,

вычислить $10^n \times ac + 10^{n/2} \times adbc + bd$, используя

арифметическое сложение, и выдать результат.

Таким образом, умножение Карацубы использует всего три рекурсивных вызова! Экономия на одном рекурсивном вызове должна сэкономить на общем времени исполнения, но на сколько? Алгоритм `Karatsuba` быстрее школьного алгоритма умножения в столбик? Ответ на этот вопрос далеко не очевиден. Однако на его примере будет легко проиллюстрировать применение инструментов анализа времени исполнения алгоритмов типа «разделяй и властвуй», которые мы рассмотрим в главе 4.

## О ПСЕВДОКОДЕ

В этой книге алгоритмы описываются с использованием комбинации высокоуровневого псевдокода и обычного человеческого языка (как в этом разделе).

Я исхожу из того, что у вас есть навыки трансляции таких высокоуровневых (общих) описаний в рабочий код на вашем любимом языке программирования. Несколько других книг и ресурсов в интернете предлагают конкретные реализации различных алгоритмов на определенных языках программирования.

Во-первых, преимущество в предпочтении высокоуровневых описаний над реализациями, специфичными для конкретного языка программирования, заключается в их гибкости: хотя я исхожу из вашей осведомленности в *каком-то* языке программирования, меня не интересует, что это за язык. Во-вторых, этот подход способствует пониманию алгоритмов на глубоком и концептуальном уровне, не обремененном деталями низкого уровня. Опытные программисты и специалисты computer science обычно мыслят и обмениваются информацией об алгоритмах на столь же высоком уровне.

Тем не менее ничто не может заменить детального понимания алгоритма, которое вытекает из разработки своей собственной рабочей программы. Настоятельно рекомендую вам реализовать на любимом языке программирования столько алгоритмов из этой книги, насколько у вас хватит времени. (Это также будет отличным поводом улучшить свои навыки программирования!)

Для развития своих навыков программирования воспользуйтесь задачами по программированию в конце главы и соответствующими тестовыми примерами.

# 1.4. Алгоритм MergeSort

Этот параграф дает возможность войти во вкус анализа времени исполнения нетривиального алгоритма — известного алгоритма сортировки слиянием `MergeSort`.

## 1.4.1. Актуальность

Алгоритм сортировки слиянием `MergeSort` — это достаточно старый алгоритм, и, безусловно, он был известен Джону фон Нейману еще в 1945 году. Зачем начинать современный курс об алгоритмах с такого старого примера?

**Старое, но по-прежнему нужное.** Несмотря на то что алгоритму `MergeSort` уже более 70 лет, он по-прежнему является одним из предпочтительных методов сортировки. Он широко используется на практике и является стандартным алгоритмом сортировки в ряде программных библиотек.

**Это классический алгоритм типа «разделяй и властвуй».** Методология разработки алгоритмов «разделяй и властвуй» представляет собой общий подход к решению задач, который находит приложение в различных областях. Ее основная идея состоит в том, чтобы разбить вашу задачу на более мелкие подзадачи, решить подзадачи рекурсивно и, наконец, объединить решения в итоговое решение исходной задачи. Алгоритм `MergeSort` является идеальным способом познакомиться с методологией «разделяй и властвуй», с учетом тех преимуществ, которые он предлагает, и возникающих при его анализе сложностей.

**Возможность улучшить техническую подготовку.** Предстоящее изучение алгоритма `MergeSort` позволит вам составить ясное представление о том, насколько хорошо имеющиеся у вас навыки соответствуют материалу этой книги. Я исхожу из того, что у вас есть достаточный опыт в программировании и математическом анализе, чтобы (с некоторой доработкой) перевести высокоуровневое описание алгоритма `MergeSort` в рабочую программу на вашем любимом языке программирования и оценить наш анализ времени исполнения алгоритма. Если этот и следующий разделы будут вам понятны, то у вас отличные шансы для понимания остальной части книги.

**Введение в руководящие принципы анализа алгоритмов.** Наш анализ времени исполнения алгоритма `MergeSort` раскрывает ряд более общих руководящих принципов. Таких, как поиск ограничений на время исполнения алгоритма, которые актуальны для любых входных данных заданного размера, а также важность оценки темпов роста времени исполнения алгоритма в зависимости от объема входных данных.

**Разминка для основного метода.** Мы проанализируем алгоритм `MergeSort` с использованием «метода дерева рекурсии», который представляет собой способ подсчета операций, выполняемых рекурсивным алгоритмом. Глава 4 опирается на эти идеи и завершается «основным методом», мощным и простым в использовании инструментом для вычисления границ времени исполнения разнообразных алгоритмов типа «разделяй и властвуй», включая алгоритмы `RecIntMult` и `Karatsuba` из раздела 1.3.

## 1.4.2. Сортировка

Вы, вероятно, уже знакомы с задачей сортировки и некоторыми алгоритмами для ее решения. Но все же, чтобы убедиться, что мы в одной лодке, рассмотрим:

**ЗАДАЧА: СОРТИРОВКА**

**Вход**: массив из $n$ чисел в произвольном порядке.

**Выход**: массив тех же самых чисел, отсортированных от наименьшего до наибольшего.

Например, при наличии входного массива

| 5 | 4 | 1 | 8 | 7 | 2 | 6 | 3 |
|---|---|---|---|---|---|---|---|

Требуемый выходной массив будет следующим:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|

В приведенном выше примере все восемь чисел во входном массиве различны. Сортировка на самом деле не будет как-то сложнее, когда есть дубликаты, она вполне может быть проще. Но для того чтобы обсуждение было максимально простым, давайте договоримся — по-дружески — о том, что числа во входном массиве всегда различны. Я настоятельно рекомендую вам подумать о том, как наши алгоритмы сортировки должны быть изменены (если это вообще потребуется), чтобы справляться с дубликатами[1].

Если вы не заботитесь об оптимизации времени исполнения, то не очень-то сложно найти правильный алгоритм сортировки. Возможно, самый простой способ — сначала просканировать входной массив, чтобы определить минимальный элемент и скопировать его в первый элемент выходного массива; затем просканировать оставшиеся элементы массива, чтобы идентифицировать и скопировать второй наименьший элемент, и так далее. Этот алгоритм называется сортировкой выбором `SelectionSort`. Вы, возможно, слышали о сортировке вставками `InsertionSort`, которая может рассматриваться в качестве более продуманной реализации той же самой идеи итеративного наращивания префикса отсортированного выходного массива. Вероятно, вы также знаете пузырьковую сортировку `BubbleSort`, в которой вы выявляете пары соседних элементов, которые еще не упорядочены, и выполняете многократные обмены до тех пор, пока весь массив не будет отсортирован. Все эти алгоритмы имеют квадратичное время исполнения. Это означает, что число операций, выполняемых на массивах длиной $n$, является функцией от $n^2$, квадрата длины входных данных. Можно ли добиться лучшего? При помощи методики «разделяй и властвуй» алгоритм сортировки слиянием `MergeSort` значительно улучшает более простые алгоритмы сортировки[2].

---

[1] На практике очень часто обрабатываемые данные (именуемые *значением*) связаны с неким числом (которое именуется *ключом*). Например, вы можете отсортировать записи о персонале (с именем, окладом и так далее), использовав номера социального страхования в качестве ключей. Мы фокусируемся на сортировке ключей, понимая, однако, что за каждым ключом закреплены соответствующие данные.

[2] Несмотря на то что сортировка слиянием `MergeSort` доминирует, сортировка вставками `InsertionSort` частенько по-прежнему широко применяется на практике, в особенности когда размеры входных данных небольшие.

### 1.4.3. Пример

Наиболее простой способ понять алгоритм `MergeSort` — это взглянуть на рисунок конкретного примера (рис. 1.3). Мы будем использовать входной массив из параграфа 1.4.2.

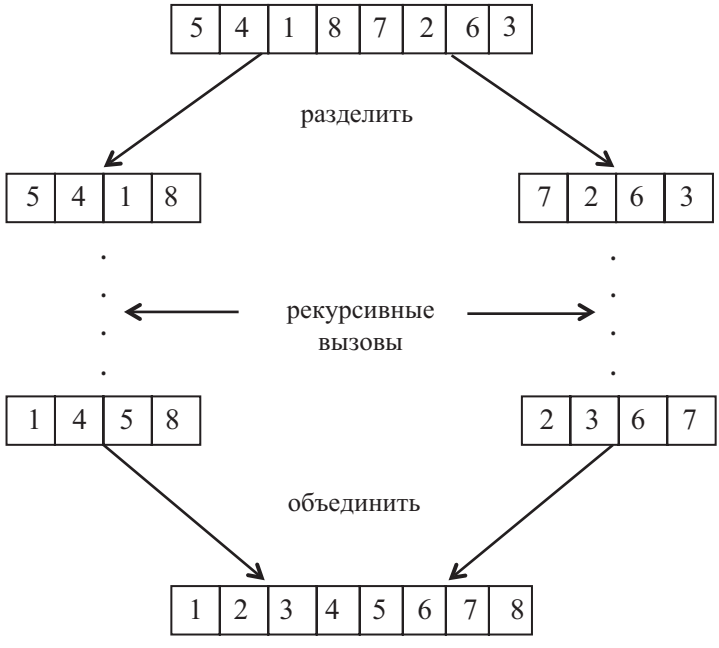



**Рис. 1.3.** Вид с высоты птичьего полета на сортировку слиянием MergeSort на конкретном примере

В качестве рекурсивного алгоритма «разделяй и властвуй» алгоритм `MergeSort` оперирует более мелкими массивами. Самый простой способ разложить задачу сортировки на более мелкие — разбить входной массив пополам. Первая и вторая половины сортируются рекурсивно. Например, на рис. 1.3 первой и второй половинами входного массива являются {5, 4 ,1 ,8} и {7, 2, 6, 3}. Благодаря волшебству рекурсии (или индукции, если хотите), первый рекурсивный вызов правильно сортирует первую половину, возвращая массив {1, 4, 5, 8}. Второй рекурсивный вызов возвращает массив {2, 3, 6, 7}. Заключительный шаг «слияния» объединяет эти два отсортированных массива длиной 4 в один-единственный отсортированный массив из всех

8 чисел. Подробности этого шага приведены ниже, но идея состоит в том, чтобы пройтись по индексам каждого отсортированного подмассива сверху вниз, заполнив выходной массив слева направо в отсортированном порядке.

## 1.4.4. Псевдокод

Рисунок 1.3 может быть преобразован в следующий ниже псевдокод с двумя рекурсивными вызовами и шагом слияния для решения исходной задачи. Как обычно, наше описание высокоуровневое и не обязательно может быть переведено построчно в рабочий программный код (хотя оно достаточно близко к реализации).

---

**MERGESORT**

**Вход**: массив $A$ из $n$ разных целых чисел.

**Выход**: массив с теми же самыми целыми числами, отсортированными от наименьшего до наибольшего.

---

```
// базовые случаи проигнорированы
```

$C$ := рекурсивно отсортировать первую половину $A$

$D$ := рекурсивно отсортировать вторую половину $A$

вернуть `Merge` ($C$, $D$)

---

В псевдокоде пропущено несколько деталей, которые заслуживают комментария. Рекурсивный алгоритм как таковой должен иметь один или несколько базовых случаев, когда больше нет дальнейшей рекурсии и результат возвращается напрямую. Поэтому, если входной массив $A$ содержит только 0 или 1 элемент, алгоритм `MergeSort` возвращает его же (он уже отсортирован). Наш псевдокод не детализирует, как определяются «первая половина» и «вторая половина», когда $n$ нечетно, но очевидная интерпретация (когда одна «половина» имеет на один элемент больше другой) работает нормально. Наконец, псевдокод игнорирует детали реализации в отношении того, как на самом деле передавать два подмассива в соответствующие им рекурсивные вызовы. Эти детали в некоторой степени зависят от языка программирования. Суть

высокоуровневого псевдокода состоит в том, чтобы игнорировать такие детали и сосредоточиваться на общих понятиях, которые выходят за рамки любого конкретного языка программирования.

### 1.4.5. Подпрограмма Merge

Что представляет собой шаг слияния `Merge`? Он применяется после того, как два рекурсивных вызова сделали свою работу и у нас имеется два отсортированных подмассива, *C* и *D*, длиной *n*/2. Идея состоит в том, чтобы пройти оба отсортированных подмассива по порядку и заполнить выходной массив слева направо также в отсортированном порядке[1].

**MERGE**

**Вход**: отсортированные массивы *C* и *D* (длиной *n*/2 каждый).

**Выход**: отсортированный массив B (длиной *n*).

**Упрощающее допущение**: *n* — четное.

```
1 i := 1
2 j := 1
3 for k := 1 to n do
4     if C[i] < D[j] then
5         B[k] := C[i]        // заполнить выходной массив
6         i := i + 1                      // прирастить i
7     else                                  // D[j] < C[i]
8         B[k] := D[j]
9         j := j + 1
```

Мы сканируем выходной массив, используя индекс *k*, и отсортированные подмассивы с индексами *i* и *j*. Все три массива проходятся слева направо. Цикл `for` в строке 3 реализует проход по выходному массиву. В первой итерации подпрограмма определяет минимальный элемент в *C* либо в *D* и копирует его

[1] Мы нумеруем элементы массива, начиная с 1 (а не с 0), и используем синтаксическую конструкцию «*A*[*i*]» для *i*-го элемента массива *A*. В разных языках программирования эти детали различаются.

в первую позицию выходного массива $B$. Минимальный элемент находится либо в $C$ (в этом случае это $C[1]$, поскольку $C$ отсортирован), либо в $D$ (в этом случае это $D[1]$, поскольку $D$ отсортирован). Приращение соответствующего индекса ($i$ или $j$), по сути, исключает только что скопированный элемент из дальнейшего рассмотрения, и затем процесс повторяется, чтобы определить наименьший элемент, остающийся в $C$ или $D$ (второй наименьший в двух подмассивах). В общем случае, наименьший элемент, еще не скопированный в $B$, находится либо в $C[i]$, либо в $D[j]$; подпрограмма явным образом проверяет, какой из них меньше, и действует соответствующим образом. Поскольку каждая итерация копирует наименьший элемент, который все еще рассматривается в $C$ или $D$, выходной массив действительно заполняется в отсортированном порядке.

Как обычно, наш псевдокод преднамеренно слегка неаккуратен, чтобы подчеркнуть, что нас интересует лес, а не деревья. Рабочая программа, естественно, должна отслеживать, когда прохождение $C$ или $D$ выходит за переделы (в процессе приращения индексов $i$ и $j$), и тогда в этой точке остающиеся элементы другого массива копируются в окончательные элементы $B$ (по порядку). Так что для вас сейчас самое время разработать свою собственную программную реализацию алгоритма `MergeSort`.

## 1.5. Анализ алгоритма MergeSort

Каково время исполнения алгоритма сортировки слиянием `MergeSort` (как функции длины $n$ входного массива)? Работает ли он быстрее, чем более простые методы сортировки, такие как сортировка выбором `SelectionSort`, сортировка вставками `InsertionSort` и пузырьковая сортировка `BubbleSort`? Под «временем исполнения» мы понимаем количество строк кода, исполняемых в конкретной реализации алгоритма. Проанализируйте построчно свою реализацию, использовав отладчик, по одной «примитивной операции» за один раз. Нас интересует количество шагов, которые отладчику придется сделать до завершения программы.

### 1.5.1. Время исполнения подпрограммы Merge

Анализ времени исполнения алгоритма `MergeSort` представляет собой задачу, которая на первый взгляд имеет устрашающий вид, поскольку перед нами рекурсивный алгоритм, который неоднократно вызывает сам себя. Поэтому давайте разомнемся на более простой задаче понимания числа операций, выполняемых в результате одного вызова подпрограммы `Merge` с двумя отсортированными массивами длиной $\ell/2$ каждый. Мы можем сделать это напрямую, проанализировав код из раздела 1.4.5 (где $\ell$ соответствует $n$). Первые строки 1 и 2 выполняют инициализацию, и мы зачтем их как две операции. Затем у нас есть цикл `for`, который исполняется в общей сложности $\ell$ раз. Каждая итерация цикла выполняет сравнение в строке 4, присваивание в строке 5 либо в строке 8 и приращение в строке 6 или в строке 9. Индекс $k$ цикла тоже должен увеличиваться при каждой итерации цикла. Это означает, что для каждой из $\ell$ итераций цикла выполняются 4 примитивных операции[1]. Резюмируя, мы приходим к заключению, что для того, чтобы объединить два отсортированных массива длиной $\ell/2$ каждый, подпрограмма `Merge` выполняет не более $4\ell + 2$ операций. В качестве упрощения позвольте мне дополнительно злоупотребить нашей дружбой слегка небрежным, но на самом деле верным допущением, использовав неравенство, которое сделает наши жизни проще: для $\ell \geq 1$, $4\ell + 2 \leq 6\ell$. Таким образом, $6\ell$ также является допустимой верхней границей числа операций, выполняемых подпрограммой `Merge`.

**Лемма 1.1 (время исполнения подпрограммы `Merge`).** *Для каждой пары отсортированных входных массивов C, D длиной $\ell/2$ подпрограмма* `Merge` *выполняет не более $6\ell$ операций.*

[1] Можно придраться к тому, что мы насчитали 4 операции в цикле. Засчитывается ли сравнение индекса цикла $k$ с его верхней границей как дополнительная операция в каждой итерации, что приводит в общей сложности к пяти операциям? В разделе 1.6 объясняется, почему такие различия в ведении подсчета на самом деле не имеют значения. Поэтому давайте договоримся — по-дружески, — что у нас 4 примитивных операции на итерацию.

**О ЛЕММАХ, ТЕОРЕМАХ И ПРОЧЕМ**

В математической практике самые важные технические утверждения имеют статус *теорем*. *Лемма* — это техническое утверждение, которое помогает с доказательством теоремы (во многом как подпрограмма `Merge` помогает с реализацией алгоритма `MergeSort`). *Следствие* — это высказывание, которое непосредственно вытекает из уже доказанного результата, например частного случая теоремы. Термин *утверждение* мы используем для автономных технических высказываний, которые сами по себе не имеют особого значения.

## 1.5.2. Время исполнения алгоритма MergeSort

Как перейти от простого анализа подпрограммы `Merge` к анализу всего рекурсивного алгоритма `MergeSort`, который порождает дальнейшие вызовы самого себя? В особенности пугает быстрое нарастание рекурсивных вызовов, количество которых буквально взрывается экспоненциально, вместе с глубиной рекурсии. Единственное, что нас успокаивает, — это тот факт, что каждому рекурсивному вызову передаются входные данные, значительно меньшие по объему, чем те, с которых мы начали. Здесь играет роль противоборство двух конкурирующих сил: с одной стороны, экспоненциальный рост количества разных подзадач, которые необходимо решать; и с другой — постоянно уменьшающиеся исходные данные, за которые отвечают эти подзадачи. То, что эти две силы друг друга балансируют, стимулирует наши усилия по анализу алгоритма `MergeSort`. В итоге мы докажем нижеследующую полезную теорему о конкретном максимуме количества операций, выполняемых алгоритмом `MergeSort` (в целом, во всех своих рекурсивных вызовах).

**Теорема 1.2 (предел времени исполнения алгоритма `MergeSort`).** *Для каждого входного массива длиной* $n \geq 1$ *алгоритм* `MergeSort` *выполняет не более*

$$6n \log_2 n + 6n$$

*операций, где* $\log_2$ *обозначает логарифм по основанию* 2.

**О ЛОГАРИФМАХ**

Некоторых очень пугает вид логарифма, который на самом деле является очень приземленным понятием. Для положительного целого $n$, $\log_2 n$ попросту означает следующее. Наберите $n$ на калькуляторе и подсчитайте количество раз, которое вам потребуется, чтобы осуществить последовательное деление на 2, прежде чем результат будет равняться 1 или меньше[a]. Например, для того, чтобы довести 32 до 1, требуется пять делений на два, поэтому $\log_2 32 = 5$. Десять делений на два сводят 1024 к 1, поэтому $\log_2 1024 = 10$. Эти примеры делают интуитивно понятным, что $\log_2 n$ гораздо меньше $n$ (сравните 10 с 1024). В особенности это актуально, когда $n$ становится большим. График (рис. 1.4) подтверждает эту интуитивную догадку.

[a] Для особо щепетильных сообщаем, что $\log_2 n$ не является целым числом, если $n$ не является степенью 2, и то, что мы описали, в действительности является округлением $\log_2 n$ до ближайшего целого числа. Это незначительное различие мы можем проигнорировать.

Теорема 1.2 ставит алгоритм `MergeSort` в выигрышное положение и демонстрирует преимущества методологии проектирования алгоритмов «разделяй и властвуй». Мы ранее отмечали, что время исполнения более простых алгоритмов сортировки, например, сортировки выбором `SelectionSort`, сортировки вставками `InsertionSort` и пузырьковой сортировки `BubbleSort`, *квадратично* зависит от размерности $n$ входных данных, а это означает, что число операций, требуемых для исполнения этих алгоритмов, является функцией, описываемой как константа, помноженная на $n^2$. Для `MergeSort`, согласно теореме 1.2, один из этих множителей $n$ заменяется на гораздо меньший $\log_2 n$. Рисунок 1.4 показывает, что, благодаря свойствам логарифма, алгоритм `MergeSort`, как правило, работает намного быстрее, чем более простые алгоритмы сортировки. Особенно это справедливо, когда $n$ становится большим[1].

[1] См. раздел 1.6.3 для дальнейшего пояснения этого вопроса.

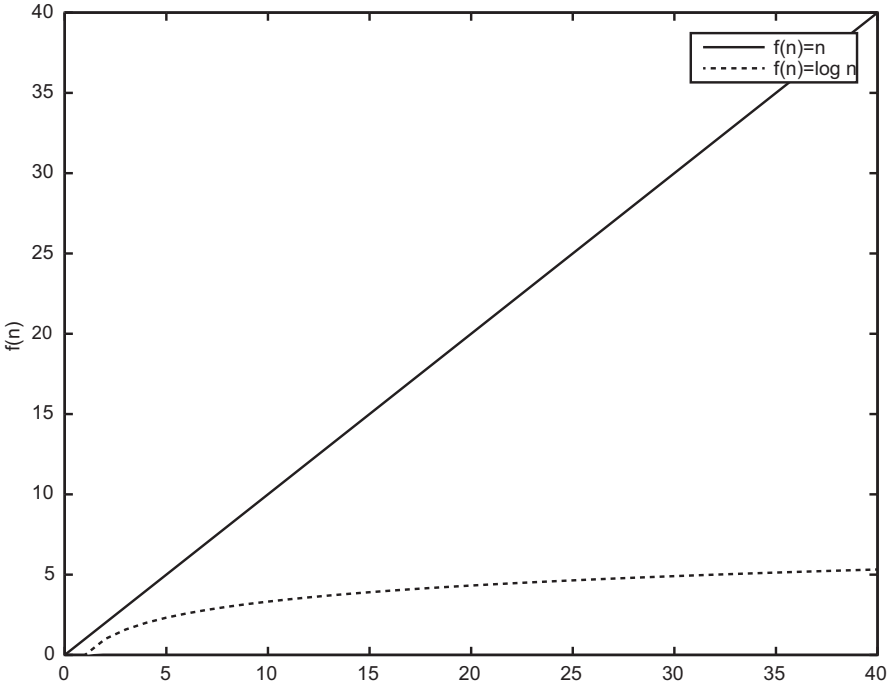



**Рис. 1.4.** Функция логарифма растет намного медленнее, чем исходная функция. Здесь показан логарифм с основанием 2; другие основания приводят к качественно похожим изображениям

## 1.5.3. Доказательство теоремы 1.2

Теперь мы сделаем полный анализ времени исполнения алгоритма `MergeSort` и в результате обоснуем утверждение о том, что рекурсивный подход «разделяй и властвуй» позволяет получать более быстрый алгоритм сортировки, нежели прямые методы. Для простоты мы примем допущение, что длина $n$ входного массива является степенью числа 2. Впрочем, это допущение может и не использоваться после незначительной доработки.

План доказательства утверждения о пределе времени исполнения алгоритма `MergeSort` в теореме 1.2 состоит в том, чтобы использовать *дерево рекурсии,* или дерево рекурсивных вызовов; см. рис. 1.5[1]. Идея метода дерева рекурсии

[1] Судя по всему, по какой-то причине специалисты по computer science считают, что деревья растут вниз.

заключается в том, чтобы изобразить весь объем операций, выполняемых рекурсивным алгоритмом, в виде древовидной структуры, в которой узлы дерева соответствуют рекурсивным вызовам, а дочерние элементы узла соответствуют рекурсивным вызовам, выполняемым этим узлом. Эта древовидная структура предоставляет нам принципиальный способ подсчитать все операции, иными словами, всю работу, проделанную алгоритмом `MergeSort` во всех его рекурсивных вызовах.

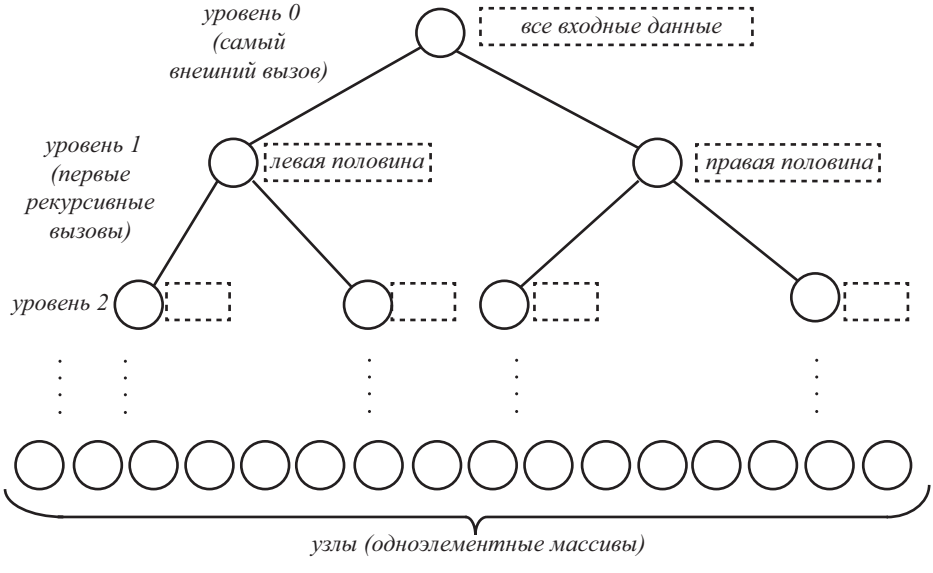



**Рис. 1.5.** Дерево рекурсии для алгоритма MergeSort. Узлы соответствуют рекурсивным вызовам. Уровень 0 соответствует самому первому вызову MergeSort, уровень 1 — следующим рекурсивным вызовам, и так далее

Корень дерева рекурсии соответствует самому первому вызову `MergeSort`, где входными данными является исходный массив. Мы назовем его уровнем 0 дерева. Поскольку каждая активация алгоритма `MergeSort` порождает два рекурсивных вызова, дерево будет двоичным (то есть с двумя дочерними элементами на узел). Уровень 1 дерева имеет два узла, которые соответствуют этим двум рекурсивным вызовам, сделанным самым первым вызовом, один для левой половины входного массива и один для правой половины. Каждый рекурсивный вызов уровня 1 сам будет делать два рекурсивных вызова, и каждый из них будет оперировать с конкретной четвертью перво-

начального входного массива. Этот процесс продолжается до тех пор, пока в конечном итоге рекурсия не закончится массивами размером 0 или 1 (базовые случаи).

**ТЕСТОВОЕ ЗАДАНИЕ 1.1**

Сколько приблизительно уровней имеет это дерево рекурсии, как функция длины $n$ входного массива?

а) постоянное число (независимо от $n$).

б) $\log_2 n$.

в) $\sqrt{n}$.

г) $n$.

(Ответ и анализ решения см. в разделе 1.5.4.)

Использование дерева рекурсии наводит на мысль о чрезвычайно удобном способе учета операций, выполняемых алгоритмом `MergeSort`, а именно спускаясь вниз уровень за уровнем. Чтобы реализовать эту идею, нам нужно понять две вещи: количество разных подзадач на заданном уровне рекурсии $j$ и длину входных данных для каждой из этих подзадач.

**ТЕСТОВОЕ ЗАДАНИЕ 1.2**

Выявите закономерность: заполните *пропуски* в следующем высказывании: на каждом уровне $j = 0, 1, 2, \ldots$ дерева рекурсии имеется [*пропуск*] подзадач, каждая из которых оперирует с подмассивом длиной [*пропуск*].

а) $2^j$ и $2^j$ соответственно.

б) $n/2^j$ и $n/2^j$ соответственно.

в) $2^j$ и $n/2^j$ соответственно.

г) $n/2^j$ и $2^j$ соответственно.

(Ответ и анализ решения см. в разделе 1.5.4.)

Теперь давайте применим полученную закономерность на практике и подсчитаем все операции, которые выполняет алгоритм `MergeSort`. Мы проходим от уровня к уровню, поэтому давайте зафиксируем $j$-й уровень дерева рекурсии. Сколько операций выполняется на $j$-м уровне рекурсии, не считая операций, выполняемых последующими рекурсивными итерациями, вызываемыми с этого уровня? Просматривая код алгоритма `MergeSort`, мы видим, что он делает только три вещи: выполняет два рекурсивных вызова и вызывает подпрограмму `Merge` на объединение результатов. Таким образом, если, как условились, игнорировать операции, выполняемые более поздними рекурсивными вызовами, делаем вывод о том, что работа, выполняемая подзадачей на уровне $j$, — это просто работа, выполняемая подпрограммой `Merge`.

Это и так уже было понятно из леммы 1.1: не более $6\ell$ операций, где $\ell$ — это длина входного массива для этой подзадачи.

Сводя все воедино, мы можем выразить общий объем операций, выполняемых рекурсивными вызовами на уровне $j$ (не учитывая более поздние рекурсивные вызовы), как

$$\underbrace{\text{Количество подзадач уровня } j}_{=2^j} \times \underbrace{\text{работа в расчете на подзадачу уровня } j}_{=\frac{6n}{2^j}}$$

Решение тестового задания 1.2 (см. ниже) показывает, что первый член равняется $2^j$, и длина входных данных в каждую такую подзадачу равняется $n/2^j$. Принимая $\ell = n/2^j$, согласно лемме 1.1, получаем, что каждая подзадача уровня $j$ выполняет не более $6n/2^j$ операций. Таким образом, во всех рекурсивных вызовах на $j$-м уровне рекурсии выполняется не более

$$2^j \times \frac{6n}{2^j} = 6n$$

операций.

Поразительно, но наша верхняя граница объема выполняемых операций алгоритма на заданном уровне $j$ не зависит от $j$! То есть каждый уровень дерева рекурсии производит одинаковое число операций. Это обусловлено идеальным равновесием между двумя конкурирующими силами — количество подзадач удваивается с каждым уровнем, в то время как объем вычислений, выполняемый в расчете на подзадачу, с каждым уровнем сокращается вдвое.

Нас интересует полное число операций, выполняемых на всех уровнях дерева рекурсии алгоритма `MergeSort`. Решение тестового задания 1.1 (см. ниже) показывает, что в дереве рекурсии имеется $\log_2 n + 1$ уровней (уровни от 0 до $\log_2 n$ включительно). Используя найденную нами границу, составляющую $6n$ операций на уровень, получаем, что общее число операций на весь алгоритм, составляет

$$\underbrace{\text{количество уровней}}_{=\log_2 n+1} \times \underbrace{\text{работа в расчете на уровень}}_{\leq 6n} \leq 6n \log_2 n + 6n.$$

Полученное значение соответствует утверждению теоремы 1.2. *Ч. т. д.*[1]

---

**О ПРИМИТИВНЫХ ОПЕРАЦИЯХ**

Мы измеряем время исполнения алгоритмов, в частности `MergeSort`, в терминах количества выполняемых «примитивных операций». Интуитивно понятно, что примитивная операция предназначена для выполнения простой задачи (например, сложение, сравнение или копирование), затрагивая небольшое количество простых переменных (например, 32-разрядные целые числа)[2]. Но имейте в виду: в некоторых высокоуровневых языках программирования одна строка кода может скрывать большое число примитивных операций. Например, строка кода, которая затрагивает каждый элемент длинного массива, транслируется в число примитивных операций, пропорциональное длине массива.

---

## 1.5.4. Ответы на тестовые задания 1.1–1.2

## Ответ на тестовое задание 1.1

**Правильный ответ: (б).** Правильный ответ: $\approx \log_2 n$. Это объясняется тем, что размер входных данных уменьшается в два раза при переходе на каждый последующий уровень рекурсии. Если длина входных данных на уровне 0 равна $n$, то рекурсивные вызовы на уровне 1 оперируют с массивами длиной

---

[1] «Ч. т. д.» — это аббревиатура выражения «что и требовалось доказать» от латинского *quod erat demonstrandum* (Q.E.D.). В математике оно употребляется в конце доказательства, чтобы отметить его завершение.

[2] Могут иметься более точные определения, но нам они не нужны.

$n/2$, рекурсивные вызовы на уровне 2 — с массивами длиной $n/4$, и так далее. Рекурсия отсутствует в базовых случаях, с входными массивами длиной не больше единицы, где нет нужды в рекурсивных вызовах. В иных случаях сколько требуется уровней рекурсии? Количество уровней равняется количеству делений $n$ на 2, чтобы получить число не более 1. Для $n$ как степени 2 это как раз и есть определение логарифма $\log_2 n$. (В более общем случае оно равняется $\log_2 n$, округленному до ближайшего целого числа.)

### Ответ на тестовое задание 1.2

**Правильный ответ: (в).** Правильный ответ заключается в том, что на уровне $j$ рекурсии имеется $2^j$ разные подзадачи, и каждая из них оперирует с подмассивом длиной $n/2^j$. Для начала начните с уровня 0, где имеется один рекурсивный вызов. На уровне 1 имеется два рекурсивных вызова, и в более общем плане (поскольку алгоритм `MergeSort` вызывает себя дважды) количество рекурсивных вызовов на каждом уровне вдвое превышает их количество на предыдущем уровне. Это последовательное удвоение подразумевает, что на каждом уровне $j$ дерева рекурсии имеется $2^j$ подзадачи. Аналогичным образом, поскольку каждый рекурсивный вызов получает только половину входных данных предыдущего, после $j$ уровней рекурсии, длина входных данных падает до $n/2^j$. Ну и наконец, поскольку мы уже знаем, что на уровне $j$ имеется $2^j$ подзадачи и первоначальный входной массив (длиной $n$) разделен между ними поровну, — получается ровно $n/2^j$ элементов на подзадачу.

## 1.6. Основные принципы анализа алгоритмов

Мы уже приобрели первый опыт анализа алгоритма на примере `MergeSort` (см. теорему 1.2). Теперь самое время сделать шаг назад и дать исчерпывающее объяснение трех допущений, которые принимались нами при анализе времени исполнения алгоритма и интерпретации результатов. Мы будем использовать эти три допущения в качестве основных руководящих принципов, которые объясняют, как рассуждать об алгоритмах, и применим их для определения того, что же на самом деле подразумевается нами под термином «быстрый алгоритм».

Цель этих принципов состоит в том, чтобы определить «золотую середину» подхода к анализу алгоритмов, который уравновешивает точность с непротиворечивостью. Точная оценка времени исполнения возможна только для самых простейших алгоритмов; в более общем случае всегда требуются компромиссы. С другой стороны, мы не хотим выплескивать ребенка вместе с водой. Мы хотим, чтобы наши математические выкладки по-прежнему имели предсказательную силу на практике, в отношении того, будет ли алгоритм быстрым или медленным. После того как мы нащупаем оптимальный баланс, нам удастся обосновать полезные оценки пределов времени исполнения десятков фундаментальных алгоритмов, и эти оценки дадут возможность увидеть точную картину того, какие алгоритмы, как правило, работают быстрее других.

### 1.6.1. Принцип № 1: анализ наихудшего случая

Формула предела времени исполнения алгоритма `MergeSort`, $6n \log^2 n + 6n$, доказанная в теореме 1.2, верна для *каждого* входного массива длиной $n$, независимо от того, что он содержит. Мы не принимали никаких допущений о природе входных данных, за исключением рамок их длины $n$. Гипотетически, даже если бы существовал злоумышленник, единственная цель жизни которого состояла бы в том, чтобы что-либо натворить с входными данными, с целью заставить алгоритм `MergeSort` выполняться максимально медленно, то предел времени исполнения этого алгоритма, $6n \log^2 n + 6n$, по-прежнему оставался бы верным и ни при каких условиях не мог бы быть превышен. Этот тип анализа называется *анализом наихудшего случая*, поскольку он дает ограничение времени исполнения, которое действительно даже для «худших» входных данных.

С учетом того, как легко принцип анализа наихудшего случая следует из выполненного нами анализа алгоритма `MergeSort`, вполне резонно задать вопрос, что еще мы можем сделать. Один из альтернативных подходов — «анализ среднего случая», который анализирует среднее время исполнения алгоритма при определенном допущении об относительных частотах различных входных данных. Например, в задаче сортировки мы можем допустить, что все входные массивы равновероятны, и затем изучать среднее время исполнения различных алгоритмов сортировки. Вторая альтернатива — рассматривать производительность алгоритма только на небольшой коллекции «эталонных

экземпляров», которые считаются репрезентативными для «типичных» или «реальных» входных данных.

Как анализ среднего случая, так и анализ эталонных экземпляров могут быть полезны, когда у вас есть предметные знания о вашей задаче и некоторое понимание того, какие входные данные репрезентативнее других. Анализ наихудшего случая, который отличается тем, что вы не делаете абсолютно никаких допущений о входных данных, в особенности подходит для универсальных подпрограмм, предназначенных для работы в различных областях применения. Чтобы быть полезными как можно большему количеству людей, в этой книге основное внимание уделяется таким подпрограммам общего назначения, и, соответственно, используется принцип анализа наихудшего случая, чтобы судить о производительности алгоритма.

Дополнительным преимуществом является то, что анализ наихудшего случая обычно намного более формализован математически, нежели его альтернативы. Это одна из причин, почему принцип анализа наихудшего случая естественным образом очевиден из выполненного нами анализа алгоритма `MergeSort`, несмотря на то что у нас априори не было ориентации на наихудшие входные данные.

## 1.6.2. Принцип № 2: анализ значимых деталей

Второй и третий руководящие принципы тесно взаимосвязаны. Назовем второй из них *анализом значимых деталей* (предупреждение: это не общепринятый термин). Этот принцип констатирует, что мы не должны излишне беспокоиться о малых коэффициентах или членах низших порядков при вычислении пределов времени исполнения алгоритмов. Мы уже встречались с этим концептуальным подходом на практике в нашем анализе алгоритма `MergeSort`: анализируя время исполнения подпрограммы `Merge` (лемма 1.1). Мы сначала обосновали верхний предел числа операций как $4\ell + 2$ (где $\ell$ — это длина выходного массива), а затем перешли к более простой формуле верхнего предела как $6\ell$, несмотря на то что ее недостатком является более крупный коэффициент. Что позволяет нам так легко играть с коэффициентами?

**Математическая простота.** Первым преимуществом анализа значимых деталей является его бóльшая математическая простота, чем альтернатива

с использованием более точных коэффициентов или членов низших порядков. Этот момент был уже очевиден, когда использовался нами при анализе времени исполнения алгоритма `MergeSort`.

**Постоянные коэффициенты зависят от реализации.** Второе обоснование менее очевидно, но чрезвычайно важно. На том уровне гранулярности, который мы будем использовать для описания алгоритмов, как и в случае с алгоритмом `MergeSort`, было бы совершенно неправильно зацикливаться на том, что собой представляют постоянные коэффициенты. Например, во время нашего анализа подпрограммы `Merge` ранее уже присутствовала двусмысленность относительно того, сколько именно «примитивных операций» выполняется на каждой итерации цикла (4, 5 или что-то другое?). Отсюда понятно, что разные программные интерпретации одного и того же псевдокода могут привести к разным постоянным коэффициентам. Двусмысленность только увеличивается, как только псевдокод транслируется в конкретную реализацию на любом высокоуровневом языке программирования, и затем далее транслируется в машинный код — постоянные коэффициенты будут неизбежно меняться в зависимости от используемого языка программирования, конкретной реализации и деталей компилятора и процессора. Поскольку наша цель состоит в том, чтобы сосредоточиться на свойствах алгоритмов, которые выходят за рамки особенностей языка программирования и машинной архитектуры, то очевидно, что эти свойства должны быть независимы от небольших изменений постоянного коэффициента при расчете времени исполнения алгоритма.

**Мы не рискуем потерять в точности прогноза о результате.** Третье оправдание — мы получаем возможность избежать неприятностей. Возможно, вы будете обеспокоены тем, что игнорирование точности постоянных коэффициентов собьет нас с правильного пути, обманом заставив думать, что алгоритм является быстрым, тогда как на практике он на самом деле работает медленно, или наоборот. К счастью, этого не произойдет для алгоритмов, обсуждаемых в этих книгах[1]. Даже при том, что мы не будем отслеживать члены низших порядков и зацикливаться на значениях постоянных коэффициентов, качественные предсказания нашего математического анализа будут очень точны.

---

[1] С одним возможным исключением: детерминированный линейный алгоритм выбора, см. в факультативном разделе 6.3.

Когда анализ приводит к выводу, что алгоритм является быстрым, он на самом деле будет быстрым на практике, и наоборот. Таким образом, в то время как анализ значимых деталей действительно пренебрегает некоторыми деталями, он дает нам то, в чем мы действительно заинтересованы: в точной оценке того, какие алгоритмы демонстрируют тенденцию работать быстрее других[1].

### 1.6.3. Принцип № 3: асимптотический анализ

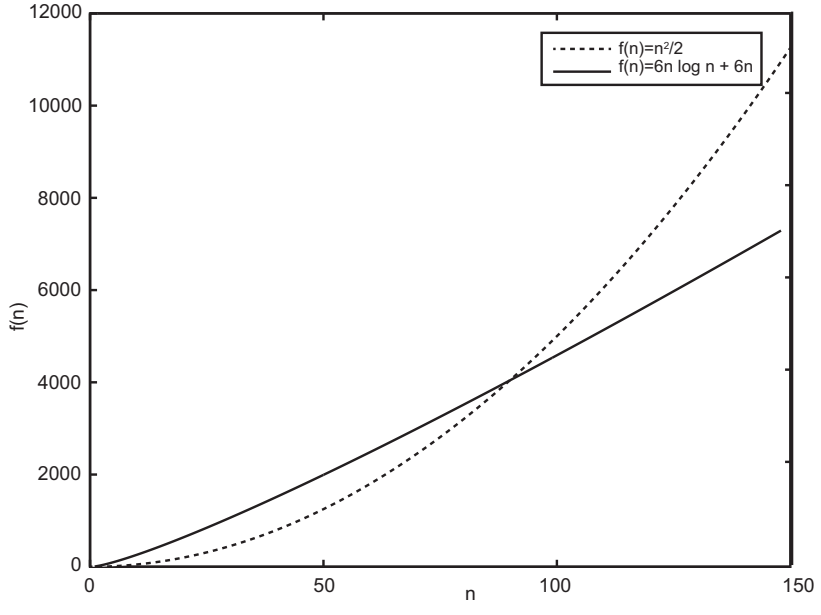
Наш третий и заключительный руководящий принцип состоит в использовании *асимптотического анализа* и акценте на темпе роста времени исполнения алгоритма по мере того, как размер $n$ входных данных становится большим. Это смещение в сторону больших входных данных было уже очевидным, когда мы оценивали предел времени исполнения алгоритма `MergeSort` (теорема 1.2), $6n \log_2 n + 6n$ операций. Мы тогда довольно самоуверенно объявили, что алгоритм `MergeSort` «лучше», чем более простые методы сортировки с квадратичным временем исполнения по отношению к размеру входных данных, как, например, сортировка вставками `InsertionSort`. Но так ли это на самом деле?

Рассмотрим конкретный пример. Пусть имеется алгоритм сортировки, который выполняет не более $\frac{1}{2}n^2$ операций, сортируя массив длиной $n$. Сравним время исполнения этого алгоритма с таковым известного нам алгоритма `MergeSort`, проанализировав

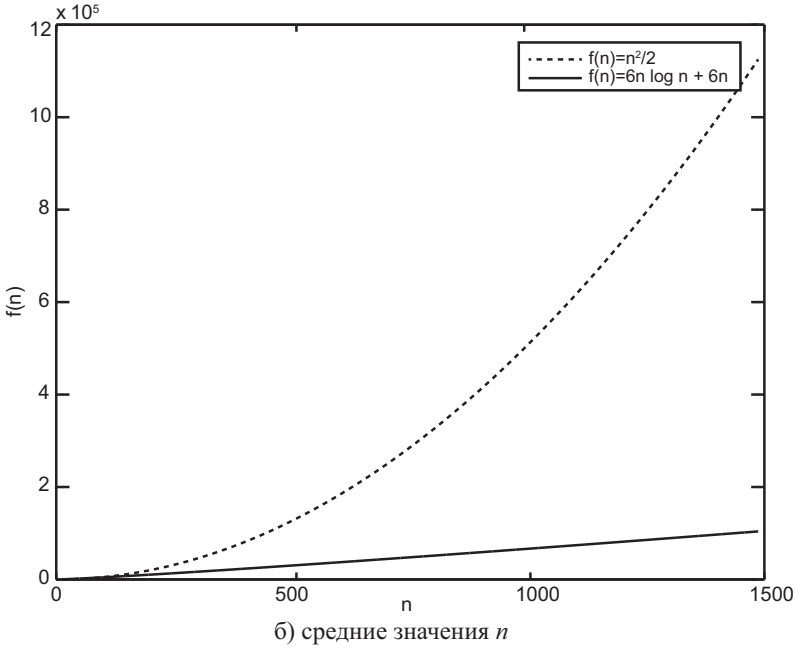
$$6n \log_2 n + 6n \text{ в сопоставлении с } \frac{1}{2}n^2.$$

Глядя на поведение этих двух функций на рис. 1.6, а, мы видим, что значение $\frac{1}{2}n^2$ меньше своего визави при малом $n$ (не более 90 или около того), в то время как значение $6n \log^2 n + 6n$ меньше для более крупного $n$. Поэтому, когда мы говорим, что алгоритм `MergeSort` быстрее, чем более простые методы сорти-

[1] Однако по-прежнему полезно иметь общее представление о релевантных постоянных множителях. Например, в сильно доработанных версиях алгоритма `MergeSort`, которые вы найдете во многих библиотеках программирования, этот алгоритм переключается с `MergeSort` на сортировку вставками `InsertionSort` (из-за его лучшего постоянного коэффициента), как только длина входного массива становится небольшой (например, не более семи элементов).




а) малые значения *n*




б) средние значения *n*

**Рис. 1.6.** По мере того как *n* становится большим, функция $\frac{1}{2}n^2$ растет намного быстрее, чем $6n \log^2 n+6n$. Шкалы оси X и оси Y в (б), соответственно, на один и два порядка больше, чем на (а)

ровки, мы в действительности имеем в виду, что он быстрее при *достаточно больших значениях размера массива входных данных*.

Почему нас больше должны интересовать случаи с большими массивами входных данных? Потому что алгоритмической изобретательности на самом деле требуют только крупные задачи. При современном быстродействии компьютеров почти любой метод сортировки, о котором можно подумать, будет моментально сортировать массив длиной 1000 — нет никакой необходимости изучать алгоритмы «разделяй и властвуй».

С учетом того, что компьютеры постоянно становятся быстрее, вам, возможно, любопытно, не станет ли решение всех вычислительных задач в конечном итоге тривиальным. На самом деле, чем быстрее становятся компьютеры, тем актуальнее становится асимптотический анализ. Наши вычислительные амбиции всегда росли вместе с нашей вычислительной мощью, поэтому с течением времени мы будем рассматривать задачи со все более крупными размерами входных данных. И пропасть в производительности между алгоритмами с различным асимптотическим временем работы будет становиться только шире по мере увеличения размеров входных данных. Например, рис. 1.6, б показывает разницу между функциями $6n \log_2 n + 6n$ и $\frac{1}{2}n^2$ для относительно больших (но по-прежнему скромных) значений $n$. И все равно, к моменту времени, когда $n = 1500$, разница между значениями функций становится 10-кратной. Если бы мы увеличили шкалу $n$ еще в 10, 100 или 1000 раз для того, чтобы получать задачи с интересными размерами, разница между двумя функциями стала бы просто огромной.

Имеется еще одно обоснование полезности принципа асимптотического анализа. Допустим, что у вас имеется фиксированный бюджет времени, например час или день. Каким образом привлечение дополнительных вычислительных мощностей позволит вам влиять на подвластный размер вычислительно разрешимой задачи? С алгоритмом, который выполняется со временем, пропорциональным значению размера входных данных, четырехкратное увеличение вычислительной мощности позволяет решать задачи с размерностью в четыре раза больше, чем раньше. С алгоритмом, который выполняется со временем, пропорциональным квадрату размера входных данных, вы сможете решать задачи, размерность которых только в два раза больше, чем раньше.

## 1.6.4. Что такое «быстрый» алгоритм?

Принимая во внимание наши три руководящих принципа, сформулируем следующее определение «быстрого алгоритма»:

> *«Быстрый алгоритм» — это алгоритм, для которого с увеличением размера входных данных наихудшее время исполнения растет медленно.*

Целесообразность использования категории «наихудшее время исполнения» следует из нашего первого руководящего принципа, описывая который мы обосновали, что важно ориентироваться на время исполнения, которое не исходит из наличия каких-либо предметных знаний, допущений, упрощений применительно к входным данным. Наши второй и третий руководящие принципы, исходящие из понимания того, что значения коэффициентов зависят от особенностей языка и машинной архитектуры и ввиду этого нам надо укрупнять коэффициенты и отметать детали, являются причинами, почему нам важно сосредоточиваться именно на темпе роста времени исполнения алгоритма.

---

**ЛЕГКОВЕСНЫЕ ПРИМИТИВЫ**

Алгоритм с линейным или почти линейным временем исполнения можно представить как примитив-заготовку, который мы можем использовать, по существу, «беззатратно» с точки зрения вычислительной нагрузки, поскольку объем требуемых для его исполнения операций едва ли больше, чем требуется для того, чтобы прочитать входные данные. Классическим примером такого легковесного примитива является сортировка, и мы также изучим еще несколько других. Когда у вас есть невероятно быстрый примитив, относящийся к вашей задаче, почему бы им не воспользоваться? Например, вы всегда можете отсортировать данные на этапе предварительной обработки, даже если вы не совсем представляете, как это пригодится позже. Одна из целей этой книги — пополнить ваш алгоритмический арсенал как можно большим количеством легковесных примитивов, готовых к применению по вашему усмотрению.

---

Что мы имеем в виду, говоря, что время исполнения алгоритма «растет медленно»? Дело в том, что почти для всех задач, рассматриваемых нами, идеалом (этаким Святым Граалем, к которому надо стремиться) является *линейно-вре-*

*менной алгоритм* — тот, для которого время исполнения прямо пропорционально зависит от размера входных данных. Значение установленного нами ранее предела времени исполнения алгоритма `MergeSort`, пропорциональное $n \log n$, является незначительно сверхлинейным и поэтому все же хуже, чем линейное время работы. Нам удастся спроектировать линейно-временные алгоритмы для некоторых задач, но не для всех. В любом случае, линейно-зависимое время — это наилучший сценарий, к которому мы будем стремиться.

### ВЫВОДЫ

- Алгоритм — это набор четко сформулированных правил для решения некоторой вычислительной задачи.
- Число примитивных операций, выполняемых алгоритмом умножения двух $n$-значных чисел в столбик, которому вы обучились в начальной школе, описывается как квадратичная функция числа $n$.
- Умножение Карацубы — это рекурсивный алгоритм целочисленного умножения, в котором, в отличие от обычного рекурсивного алгоритма, используется метод Гаусса с целью экономии на одном рекурсивном вызове.
- Опытные программисты и специалисты в области computer science обычно мыслят и обмениваются информацией об алгоритмах, используя довольно абстрактные описания высокого уровня, а не подробные реализации.
- Алгоритм сортировки слиянием `MergeSort` — это алгоритм типа «разделяй и властвуй», который разбивает входной массив на две половины, рекурсивно сортирует каждую половину и объединяет результаты, используя подпрограмму слияния `Merge`.
- При игнорировании постоянных коэффициентов и членов низших порядков при расчете времени исполнения число операций, выполняемых алгоритмом `MergeSort` для сортировки $n$ элементов, растет как функция $n \log_2 n$. В анализе используется дерево рекурсии, чтобы удобно организовать работу, выполняемую всеми рекурсивными вызовами.

- Поскольку функция $\log_2 n$ растет медленно в зависимости от $n$, то алгоритм `MergeSort`, как правило, быстрее, чем более простые алгоритмы сортировки, которые характеризуются квадратичным ростом числа операций. Для большого $n$ улучшение будет радикальным.
- Три руководящих принципа анализа алгоритмов: (i) анализ наихудшего случая, который дает нам алгоритмы, хорошо работающие без допущений о природе входных данных; (ii) анализ значимых деталей, который уравновешивает предсказательную силу с математической простотой, игнорируя постоянные коэффициенты и члены низших порядков; и (iii) асимптотический анализ, представляющий собой смещение в сторону больших входных данных, то есть данных, требующих алгоритмической изобретательности.
- «Быстрый алгоритм» — это алгоритм, время работы которого в худшем случае растет медленно, в зависимости от роста размера входных данных.
- «Легковесный примитив» — это алгоритм, который выполняется за линейное или почти линейное время, которое едва ли больше того, что требуется для прочтения входных данных.

## Задачи на закрепление материала

**Задача 1.1.** Допустим, мы выполняем алгоритм `MergeSort` с приведенным ниже входным массивом:

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

Перейдем сразу к шагу после завершения двух самых внешних рекурсивных вызовов, но перед заключительным шагом слияния `Merge`. Возьмем два выходных массива с 5 элементами после рекурсивных вызовов, представим их как склеенный 10-элементый массив. Какое число находится в 7-й позиции?

**Задача 1.2.** Рассмотрим следующую модификацию алгоритма `MergeSort`: разделить входной массив на три части (не на половины), рекурсивно отсор-

тировать каждую треть и, наконец, объединить результаты, используя трехаргументную подпрограмму слияния `Merge`. Каким будет время исполнения этого алгоритма как функции длины $n$ входного массива при игнорировании постоянных коэффициентов и членов низших порядков? [Подсказка: обратите внимание, что подпрограмма `Merge` может быть реализована так, чтобы количество операций было линейным как функция суммы длин входных массивов.]

а) $n$.

б) $n \log n$.

в) $n (\log n)^2$.

г) $n^2 \log n$.

**Задача 1.3.** Пусть дано $k$ отсортированных массивов, каждый с $n$ элементами, и вы хотите их объединить в один массив из $kn$ элементов. Один из подходов состоит в том, чтобы использовать подпрограмму слияния `Merge` из раздела 1.4.5 многократно, сначала объединяя первые два массива, затем объединяя результат с третьим массивом, затем с четвертым массивом, и так далее до тех пор, пока вы не выполните слияние результата на $k$-м шаге с последним входным массивом. Каким будет время исполнения этого алгоритма последовательного слияния как функции от $k$ и $n$ при игнорировании постоянных коэффициентов и членов низших порядков?

а) $n \log k$.

б) $nk$.

в) $nk \log k$.

г) $nk \log n$.

д) $nk^2$.

е) $n^2k$.

**Задача 1.4.** Еще раз рассмотрим задачу слияния $k$ отсортированных массивов длиной $n$ в один отсортированный массив длиной $kn$. Рассмотрим алгоритм, который сначала делит группу из $k$ массивов на $k/2$ пар массивов и использует подпрограмму `Merge`, чтобы объединить каждую пару, в результате получая $k/2$ отсортированных массивов длиной $2n$. Пусть алгоритм повторяет этот шаг до тех пор, пока не останется всего один отсортированный массив длиной $kn$. Каким будет время исполнения этого алгоритма как функции от $k$ и $n$ при игнорировании постоянных коэффициентов и членов низших порядков?

а) $n \log k$.

б) $nk$.

в) $nk \log k$.

г) $nk \log n$.

д) $nk^2$.

е) $n^2k$.

## Задача повышенной сложности

**Задача 1.5.** В качестве входных данных имеется неотсортированный массив из $n$ разных чисел, где $n$ — степень числа 2. Предложите алгоритм, который определяет второе по величине число в массиве, при этом алгоритм использует не более $n + \log_2 n - 2$ сравнений. [Подсказка: какой информацией вы будете обладать после вычисления наибольшего числа?]

## Задача по программированию

**Задача 1.6.** Реализуйте алгоритм целочисленного умножения Карацубы на своем любимом языке программирования[1]. Чтобы получить от этой задачи максимальную отдачу, сделайте так, чтобы ваша программа на выбранном вами языке вызывала свой оператор умножения только на операциях с парами одноразрядных чисел.

В качестве конкретного примера вычислите, каким будет результат умножения приведенных ниже двух 64-разрядных чисел?[2]

3141592653589793238462643383279502884197169399375105820974944592

2718281828459045235360287471352662497757247093699959574966967627

[1] Подумайте: сделает ли вашу жизнь проще тот факт, если количество знаков каждого целого числа является степенью 2?

[2] Если вам нужна помощь или же вы хотите сравнить свои результаты с другими читателями, посетите дискуссионный форум на www.algorithmsilluminated.org.

# 2 Асимптотические обозначения

Данная глава посвящена дальнейшему развитию математической абстракции, которая формализует руководящие принципы анализа алгоритмов, выведенные нами в предыдущей главе (раздел 1.6). Нашей целью будет поиск «золотой середины» гранулярности наших допущений (рассуждений) в исследовании алгоритмов. Мы стремимся не учитывать детали второго порядка, в частности постоянные коэффициенты и члены низших порядков, и при этом иметь возможность сосредоточить внимание на том, какую зависимость демонстрирует время исполнения алгоритма по мере увеличения размера входных данных. Формально это делается с помощью математического обозначения О-большое и соотносительных с ним понятий — категориального инструментария, которым должен владеть любой серьезный программист и специалист в области computer science.

## 2.1. Отправная точка

Прежде чем углубляться в математический формализм асимптотических обозначений, давайте убедимся, что для этой темы есть достаточная база, у вас есть четкое понимание того, чего она пытается достичь, и что вы познакомились с несколькими простыми и интуитивно понятными примерами.

### 2.1.1. Актуальность

Асимптотические обозначения обеспечивают базовый категориальный аппарат (набор понятий) для рассмотрения процессов разработки и анализа алгоритмов. Важно понимать, что именно программисты имеют в виду, когда они говорят, что одна часть кода выполняется за время «$O$-большое $n$», в то время как другая выполняется за время «$O$-большое $n$-квадрат».

Этот аппарат получил такое повсеместное распространение потому, что он позволяет достигнуть той самой «золотой середины» рассуждений об алгоритмах. Асимптотические обозначения в достаточной мере грубы, что дает возможность не учитывать все детали, которые вы хотите проигнорировать, — элементы, которые зависят от выбора архитектуры, выбора языка программирования, выбора компилятора и так далее. С другой стороны, они достаточно точны, чтобы позволять делать полезные сравнения между

различными высокоуровневыми алгоритмическими подходами к решению задачи, в особенности на больших объемах входных данных (входных данных, которые требуют алгоритмической изобретательности). Например, асимптотический анализ помогает нам различать лучшие и худшие подходы к сортировке, лучшие и худшие подходы к умножению двух целых чисел и так далее.

## 2.1.2. Высокоуровневая идея

Если вы попросите практикующего программиста объяснить суть системы асимптотических обозначений, то он, скорее всего, скажет примерно следующее:

**СИСТЕМА АСИМПТОТИЧЕСКИХ ОБОЗНАЧЕНИЙ В СЕМИ СЛОВАХ**

устранить постоянные коэффициенты (слишком системно-зависимы) и члены низших порядков (нерелевантны для больших объемов данных)

В дальнейшем мы увидим, что за использованием асимптотических обозначений, конечно, стоит больше, чем просто эти семь слов, но спустя, положим, десять лет, если вы запомните только эти семь слов, они все равно будут выражать самую суть.

Зачем при анализе времени исполнения алгоритма необходимо отбрасывать информацию, такую как постоянные коэффициенты и члены низших порядков? Дело в том, что члены низших порядков по определению становятся все более и более неактуальными, поскольку вы сосредоточиваетесь на больших объемах входных данных, то есть входных данных, которые требуют алгоритмической изобретательности. Между тем постоянные коэффициенты обычно сильно зависят от деталей реализации. Если при анализе алгоритма мы не хотим привязываться к определенному языку программирования, архитектуре или компилятору, имеет смысл использовать формальный подход и не сосредоточиваться на постоянных коэффициентах.

Например, помните наш анализ алгоритма `MergeSort` (раздел 1.4)? Мы определили верхний предел его времени исполнения, равный

$$6n \log_2 n+6n$$

примитивных операций, где $n$ — длина входного массива. Член низших порядков здесь $6n$. Поскольку $n$ растет медленнее, чем $n \log_2 n$, он будет устранен в асимптотических обозначениях. Основной постоянный коэффициент 6 тоже будет устранен, в результате мы получаем намного более простое выражение $n \log n$. В результате этих упрощений условимся говорить, что время работы алгоритма `MergeSort` составляет $O$-большое от $n \log n$, что записывается как $O(n \log n)$, или что алгоритм `MergeSort` является алгоритмом «$O(n \log n)$»[1]. В интуитивном плане высказывание, что нечто работает со временем $O(f(n))$ для функции $f(n)$, означает, что $f(n)$ — это то, с чем вы останетесь после устранения постоянных коэффициентов и членов низших порядков[2]. Применение обозначения $O$-большое позволяет ранжировать алгоритмы по группам согласно их асимптотически наихудшим временам исполнения, например: линейные $O(n)$ алгоритмы, $O(n \log n)$ алгоритмы, квадратичные $O(n^2)$ алгоритмы, $O(1)$ алгоритмы с постоянным временем и так далее.

Буду откровенным: безусловно, я не утверждаю, что постоянные коэффициенты никогда не имеют значения при разработке алгоритма. Скорее, верно то, что, когда вы хотите провести сравнение между принципиально различными способами решения задачи, асимптотический анализ часто является правильным инструментом для понимания того, какой из них будет работать лучше, в особенности на достаточно больших объемах входных данных. В любом случае, как только вы нашли лучший высокоуровневый алгоритмический подход к задаче, пожалуй, вам все же стоит приложить усилия, чтобы улучшить ведущий постоянный коэффициент и, возможно, даже члены низших порядков. Как бы то ни было, если результат вашего

---

[1] Игнорируя постоянные коэффициенты, нам даже не нужно указывать основание логарифма (поскольку разные логарифмические функции различаются только постоянным множителем). См. раздел 4.2.2 для подробного изучения.

[2] Например, даже функция $10^{100} \times n$ технически имеет $O(n)$. В этой книге мы будем изучать только такие пределы времени исполнения алгоритмов, где устраненный постоянный коэффициент относительно мал.

начинания зависит от эффективности вашей реализации конкретного фрагмента кода, найдите в себе силы сделать его быстрым, насколько это возможно.

### 2.1.3. Четыре примера

Мы завершаем этот раздел четырьмя очень простыми примерами. Они настолько просты, что если у вас есть какой-либо предшествующий опыт с обозначением *O*-большое, то вам, вероятно, следует сразу перейти к разделу 2.2, чтобы углубиться в изучение математического формализма. Но если эти понятия для вас совершенно в новинку, то эти простые примеры должны вас правильно сориентировать.

Сначала рассмотрим задачу поиска в некотором массиве заданного целого числа *t*. Давайте проанализируем простой алгоритм, который выполняет линейное сканирование массива, проверяя каждый элемент, чтобы убедиться, что его значение равняется требуемому целому числу *t*.

**ПОИСК В ОДНОМ МАССИВЕ**

**Вход**: массив *A* из *n* целых чисел и искомое целое число *t*.

**Выход**: содержит или нет массив *A* число *t*.

```
for i := 1 to n do
    if A[i] = t then
        return TRUE
return FALSE
```

Этот псевдокод просто проверяет каждый элемент массива по очереди. Если цикл находит целое число *t*, то он возвращает истину, если же цикл заканчивает перебор массива, не найдя *t*, то он возвращает ложь.

Мы еще не дали формального определения обозначению *O*-большое, но интуитивно, благодаря нашим предыдущим рассуждениям вы можете догадаться об асимптотическом времени работы приведенного выше кода.

**ТЕСТОВОЕ ЗАДАНИЕ 2.1**

Каково асимптотическое время исполнения приведенного выше кода для поиска в одном массиве как функции длины $n$ массива?

а) $O(1)$.

б) $O(\log n)$.

в) $O(n)$.

г) $O(n^2)$.

(Решение и его анализ см. в разделе 2.1.4.)

Следующие три примера касаются различных способов объединения двух циклов. Прежде всего, давайте рассмотрим один цикл, за которым следует другой. Допустим, нам теперь даны два целочисленных массива, $A$ и $B$, оба длиной $n$, и мы хотим узнать, находится ли целое число $t$ в одном из них. Давайте снова рассмотрим простой алгоритм, где мы просто выполняем поиск в $A$, и если нам не удается найти $t$ в $A$, то мы тогда выполняем поиск в $B$. Если мы не находим $t$ и в $B$, то мы возвращаем ложь.

**ПОИСК В ДВУХ МАССИВАХ**

**Вход**: массивы $A$ и $B$ из $n$ целых чисел каждый и искомое целое число $t$.

**Выход**: содержит или нет массив $A$ или $B$ число $t$.

```
for i := 1 to n do
    if A[i] = t then
        return TRUE
for i := 1 to n do
    if B[i] = t then
        return TRUE
return FALSE
```

Каково, в обозначении $O$-большое, время исполнения этого более длинного фрагмента кода?

**ТЕСТОВОЕ ЗАДАНИЕ 2.2**

Каково асимптотическое время исполнения приведенного выше фрагмента кода для поиска в двух массивах как функции длины $n$ входных массивов?

а) $O(1)$.

б) $O(\log n)$.

в) $O(n)$.

г) $O(n^2)$.

(Решение и его анализ см. в разделе 2.1.4.)

Далее давайте рассмотрим более интересный пример двух вложенных (не последовательных) циклов. Допустим, мы хотим проверить, включают ли два заданных массива длиной $n$ некоторое общее число или нет. Самое простое решение — проверить все возможности. То есть для каждого индекса $i$ в массиве $A$ и каждого индекса $j$ в массиве $B$ мы проверяем, является ли $A[i]$ тем же числом, что и $B[j]$. Если это так, то мы возвращаем истину. Если мы исчерпываем все возможности, не найдя одинаковых элементов, то мы можем смело вернуть ложь.

**ПРОВЕРКА НА НАЛИЧИЕ ОБЩЕГО ЭЛЕМЕНТА**

**Вход**: массивы $A$ и $B$ из $n$ целых чисел каждый.

**Выход**: содержится ли целое число $t$ в обоих массивах, $A$ и $B$, или нет.

```
for i := 1 to n do
    for j := 1 to n do
        if A[i] = B[j] then
            return TRUE
return FALSE
```

Вопрос, как обычно, один: каково в обозначении $O$-большое время исполнения этого фрагмента кода?

**ТЕСТОВОЕ ЗАДАНИЕ 2.3**

Каково асимптотическое время исполнения приведенного выше фрагмента кода для проверки на наличие общего элемента как функции длины $n$ массивов?

а) $O(1)$.

б) $O(\log n)$.

в) $O(n)$.

г) $O(n^2)$.

(Решение и его анализ см. в разделе 2.1.4.)

Наш последний пример снова связан с вложенными циклами, но на этот раз мы отыскиваем повторяющиеся значения в одном массиве $A$, а не в двух разных массивах. Приведем фрагмент кода, который мы собираемся проанализировать.

**ПРОВЕРКА НА ДУБЛИКАТЫ**

**Вход**: массив $A$ из $n$ целых чисел.

**Выход**: содержит ли массив $A$ некоторое целое число более одного раза или нет.

```
for i := 1 to n do
    for j := i + 1 to n do
        if A[i] = A[j] then
            return TRUE
return FALSE
```

Есть два небольших различия между этим фрагментом кода и предыдущим. Первое и более очевидное изменение заключается в том, что мы сравниваем

$i$-й элемент $A$ с $j$-м элементом $A$, а не с $j$-м элементом некого другого массива $B$. Второе, более тонкое изменение состоит в том, что внутренний цикл теперь начинается с индексной позиции $i + 1$, а не с индексной позиции 1. Почему бы не начать с 1, как и раньше? Потому что тогда он также вернет истину в самой первой итерации (поскольку очевидно, что $A[1] = A[1]$, элемент массива равен самому себе) независимо от того, имеет ли массив какие-либо повторяющиеся значения! Можно, конечно, пропустить все итерации, где $i$ и $j$ равны, но это по-прежнему будет расточительно: каждую пару элементов, $A[h]$ и $A[k]$, не следует сравнивать друг с другом дважды (один раз, когда $i = h$ и $j = k$, и следующий раз, когда $i = k$ и $j = h$), в то время как приведенный выше код сравнивает их только один раз.

Вопрос по заданию, как всегда, один и тот же: каково в обозначении $O$-большое время исполнения этого фрагмента кода?

**ТЕСТОВОЕ ЗАДАНИЕ 2.4**

Каково асимптотическое время работы приведенного выше фрагмента кода для проверки на дубликаты как функции массива длиной $n$?

а) $O(1)$.

б) $O(\log n)$.

в) $O(n)$.

г) $O(n^2)$.

(Решение и его анализ см. в разделе 2.1.4.)

Приведенные выше элементарные примеры должны дать вам достаточно очевидное интуитивное понимание того, что представляет собой обозначение $O$-большое и какой эффект оно дает для решения поставленной задачи. Далее мы перейдем к математической разработке асимптотических обозначений, а также к алгоритмам, еще более интересным, чем рассмотренные ранее.

## 2.1.4. Решения тестовых заданий 2.1–2.4

### Решение тестового задания 2.1

**Правильный ответ: (в).** Правильный ответ — $O(n)$. Другими словами, мы говорим, что алгоритм имеет время исполнения, линейное от $n$. Почему это верно? Точное число выполняемых операций зависит от входных данных — содержится или нет целевое число $t$ в массиве $A$, и, если да, где в массиве оно находится. В худшем случае, когда $t$ не находится в массиве, алгоритм выполнит неудачный поиск, просканировав весь массив (за $n$ итераций цикла), и вернет ложь. Ключевая особенность псевдокода в этом задании состоит в том, что он выполняет постоянное число операций для каждого элемента массива (сравнивая $A[i]$ с $t$, увеличивая индекс $i$ цикла, и так далее). Термин «постоянный», применительно к числу операций, здесь означает некоторое число, не зависящее от $n$, например 2 или 3. Можно спорить о том, какое значение эта константа принимает в приведенном выше псевдокоде, но какой бы она ни была, она удобно устраняется в обозначении $O$-большое. Аналогичным образом, алгоритм выполняет постоянное число операций до начала цикла и после его завершения, и независимо от того, какой именно константа может быть, она представляет собой член низшего порядка, который опускается в обозначении $O$-большое. Поскольку игнорирование постоянных коэффициентов и членов низших порядков приводит нас к границе $n$ общего числа операций, асимптотическое время работы этого алгоритма составляет $O(n)$.

### Решение тестового задания 2.2

**Правильный ответ: (в).** Здесь ответ тот же, $O(n)$. Причина в том, что наихудшее число выполняемых операций (при неудачном поиске) вдвое больше, чем в предыдущем фрагменте псевдокода, — сначала мы выполняем поиск в первом массиве и затем во втором массиве. Этот дополнительный множитель 2 вносит вклад только в ведущий коэффициент в формуле предела времени исполнения этого алгоритма, поэтому он устраняется, когда мы используем обозначение $O$-большое. Таким образом, этот алгоритм, как и предыдущий, является линейным алгоритмом.

## Решение тестового задания 2.3

**Правильный ответ: (г).** На этот раз ответ изменился. Для этого фрагмента псевдокода время работы не $O(n)$, а $O(n^2)$. («$O$-большое $n$-квадрат», так называемый квадратичный алгоритм.) Таким образом, в случае с этим алгоритмом, если умножить длины входных массивов на 10, то время исполнения увеличится в 100 раз (а не в 10 раз, как в случае с линейным алгоритмом).

Почему этот алгоритм имеет время исполнения $O(n^2)$? Его псевдокод снова выполняет постоянное число операций для каждой итерации цикла (то есть для каждого варианта индексов $i$ и $j$), а также постоянное число операций вне циклов. Изменилось то, что теперь мы имеем в общей сложности $n^2$ итераций этого двойного цикла `for` — по одному для каждого варианта $i \in \{1,2, ..., n\}$ и $j \in \{1,2, ..., n\}$. В нашем первом примере было всего $n$ итераций одного цикла `for`. В нашем втором примере, поскольку первый цикл `for` завершился до начала второго, у нас было всего $2n$ итераций. Здесь же для *каждой* из $n$ итераций внешнего цикла `for` код выполняет $n$ итераций внутреннего цикла `for`. В итоге это дает $n \times n = n^2$ итераций.

## Решение тестового задания 2.4

**Правильный ответ: (г).** Ответ на это задание такой же, как и на предыдущее, $O(n^2)$. Время работы снова пропорционально числу итераций двойного цикла `for` (с постоянным числом операций на итерацию). Итак, сколько же имеется итераций? Ответ — примерно $\frac{n^2}{2}$. Один из способов понять это — вспомнить о том, что псевдокод этого алгоритма выполняет примерно половину работы алгоритма из предыдущего задания (поскольку внутренний цикл `for` начинается с $j = i + 1$, а не с $j = 1$). Второй способ — обратить внимание на то, что имеется ровно одна итерация для каждого подмножества $\{i, j\}$ двух разных индексов из $\{1, 2, ..., n\}$, и таких подмножеств мы имеем ровно $\binom{n}{2} = \frac{n(n-1)}{2}$ [1].

[1] $\binom{n}{2}$ произносится «из 2 по $n$» и называется «биномиальным коэффициентом». См. также решение тестового задания 3.1.

# 2.2. Обозначение O-большое

В этом разделе дается формальное определение обозначения *O*-большое. Мы начнем с определения на простом разговорном языке, проиллюстрируем его наглядно и, наконец, дадим математическое определение.

## 2.2.1. Определение на русском языке

Обозначение *O*-большое относится к функциям вида $T(n)$, определенным на положительных целых числах $n = 1, 2, \ldots$ . Для нас $T(n)$ почти всегда будет обозначать предел худшего времени исполнения алгоритма как функцию длины $n$ входных данных. Что же имеется в виду, когда говорят, что $T(n) = O(f(n))$ для некой «канонической» функции $f(n)$, в частности $n$, $n \log n$ или $n^2$? Дадим определение на русском языке.

**ОБОЗНАЧЕНИЕ O-БОЛЬШОЕ (РУССКАЯ ВЕРСИЯ)**

$T(n) = O(f(n))$, тогда и только тогда, когда $T(n)$ в конечном итоге ограничена сверху постоянной, кратной функции $f(n)$.

## 2.2.2. Иллюстрированное определение

Взгляните на рис. 2.1, показывающий определение обозначения *O*-большое. Ось $X$ соответствует параметру $n$, ось $Y$ — значению функции. Пусть $T(n)$ есть функция, соответствующая сплошной линии, а $f(n)$ — функция, соответствующая нижней пунктирной линии. Функция $T(n)$ не ограничена сверху функцией $f(n)$, но умножение $f(n)$ на 3 в результате дает верхнюю пунктирную линию, которая будет лежать выше $T(n)$, после того как мы зайдем достаточно далеко вправо на графике после «точки пересечения» в $n_0$. Поскольку, как мы видим на графике, $T(n)$ действительно в конечном итоге ограничивается сверху постоянной, кратной $f(n)$, мы можем сказать, что $T(n) = O(f(n))$.

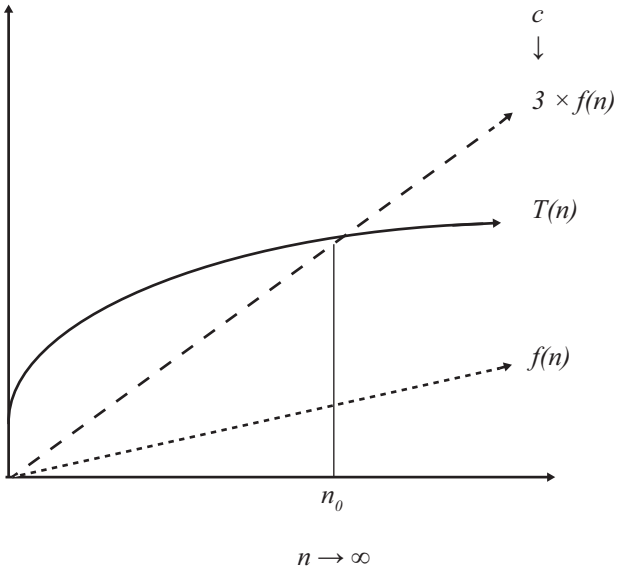



**Рис. 2.1.** Показывает, когда $T(n) = O(f(n))$. Константа $c$ (равная 3 в данном примере) количественно иллюстрирует высказывание «постоянная кратная» от $f(n)$, а константа $n_0$ количественно иллюстрирует высказывание «в конечном итоге»

## 2.2.3. Математическое определение

Приведем математическое определение обозначения *O*-большое. Это та дефиниция, которую вы должны использовать в формальных доказательствах.

**ОБОЗНАЧЕНИЕ О-БОЛЬШОЕ (МАТЕМАТИЧЕСКАЯ ВЕРСИЯ)**

$T(n) = O(f(n))$, тогда и только тогда, когда существуют положительные константы $c$ и $n_0$, такие что

$$T(n) \leq c \times f(n) \tag{2.1}$$

для всех $n \geq n_0$.

Это прямое переложение на математический язык определения на русском языке, приведенного в разделе 2.2.1. Неравенство в (2.1) означает, что $T(n)$ должна быть ограничена сверху выражением, кратным $f(n)$ (при этом константа $c$ обозначает кратное). Обозначение «для всех $n \geq n_0$» означает, что неравенство должно соблюдаться только в конечном итоге, то есть когда $n$ достаточно велико (при этом константа $n_0$ обозначает, насколько велико). Например, на рис. 2.1 константа $c$ соответствует 3, а $n_0$ — точке пересечения между функциями $T(n)$ и $c \times f(n)$.

**Интерпретация определения с точки зрения теории игр.** Если вам требуется доказать, что $T(n) = O(f(n))$ (например, доказать, что асимптотическое время исполнения алгоритма линейно от размера входных данных, что соответствует $f(n) = n$), то ваша задача состоит в том, чтобы подобрать константы $c$ и $n_0$ так, чтобы (2.1) соблюдалось всякий раз, когда $n \geq n_0$. Один из способов добиться этого — использовать теоретико-игровой подход, то есть рассмотреть поиск решения как игру между вами и противником. Вы ходите первым и должны зафиксировать константы $c$ и $n_0$. Ваш противник ходит вторым и может выбрать любое целое число $n$, которое не меньше $n_0$. Вы выигрываете, если соблюдается неравенство (2.1), а ваш противник выигрывает, если соблюдается противоположное неравенство $T(n) > c \times f(n)$.

Если $T(n) = O(f(n))$, то существуют константы $c$ и $n_0$, такие, что (2.1) соблюдается для всех $n \geq n_0$, и у вас есть выигрышная стратегия в этой игре. В противном случае, независимо от того, как вы выберете $c$ и $n_0$, ваш оппонент может выбрать достаточно большое $n \geq n_0$, чтобы перевернуть неравенство и выиграть игру.

---

**ПРЕДОСТЕРЕЖЕНИЕ**

Когда мы говорим, что $c$ и $n_0$ являются константами, мы имеем в виду, что они *не могут зависеть от n*. Например, на рис. 2.1 $c$ и $n_0$ были зафиксированы в значениях 3 и 1000. Затем мы перешли к рассмотрению неравенства (2.1), принимая во внимание, что $n$ растет сколь угодно долго (на графике в направлении оси абсцисс до бесконечности). Будьте внимательны: если вы когда-нибудь (в предполагаемом доказательстве *O*-большое) застанете себя за произнесением фразы «пусть $n_0 = n$» или «пусть $c = \log_2 n$», то вам нужно остановиться и начать заново с выбора констант $c$ и $n_0$, не зависящих от $n$.

---

## 2.3. Два простых примера

Пройдя через тернии формального математического определения обозначения *O*-большое, давайте рассмотрим несколько примеров. Эти примеры не дадут нам никаких новых идей, но они помогут доступнее объяснить с точки зрения здравого смысла, что обозначение *O*-большое достигает своей намеченной цели устранения постоянных коэффициентов и членов низших порядков. Они также являются хорошей разминкой для менее очевидных примеров, с которыми мы столкнемся позже.

### 2.3.1. Для многочленов степени *k* O-большим является O ($n^k$)

Наше первое формальное утверждение состоит в том, что если $T(n)$ это многочлен с некоторой степенью $k$, то $T(n) = O(n^k)$.

**Утверждение 2.1.** *Пусть*

$$T(n) = a_k n^k + \ldots a_1 n + a_0,$$

*где $k \geq 0$ — это неотрицательное целое число и $a_i$ — вещественные числа (положительные или отрицательные). Тогда $T(n) = O(n^k)$.*

Смысл утверждения 2.1 в том, что в случае с многочленом в обозначении *O*-большое вам нужно беспокоиться только о самой высокой степени, которая имеется в многочлене (степень многочлена). Следовательно, обозначение *O*-большое действительно устраняет постоянные множители и члены низших порядков.

*Доказательство утверждения 2.1*. Чтобы доказать это утверждение, нам нужно использовать математическое определение обозначения *O*-большое (раздел 2.2.3). В соответствии с этим определением наша задача — найти пару положительных констант, $c$ и $n_0$ (каждая независимая от $n$), где $c$ количественно истолковывается как постоянная, кратная $n^k$, а $n_0$ количественно истолковывается как «достаточно большое $n$». Чтобы не усложнять ход рассуждений, но, безусловно, оставить налет загадочности, просто вытащим значения этих

констант из шляпы: здесь $n_0 = 1$ и $c$ равняется сумме абсолютных значений коэффициентов многочлена[1]:

$$c = |a_k| + \ldots + |a_1| + |a_0|.$$

Оба этих числа не зависят от $n$. Теперь нам нужно показать, что эти варианты констант удовлетворяют определению, то есть $T(n) \leq cn^k$ для всех $n \geq n_0 = 1$.

Чтобы проверить это неравенство, зададим произвольное положительное целое число $n \geq n_0 = 1$. Нам нужна последовательность верхних пределов времени исполнения алгоритма $T(n)$, кульминацией которых станет верхний предел $c \times n^k$. Сначала применим определение $T(n)$:

$$T(n) = a_k n^k + \ldots + a_1 n + a_0.$$

Если взять абсолютное значение каждого коэффициента $a_i$ в правой части, то выражение становится лишь все больше. ($|a_i|$ может быть больше только $a_i$, и, поскольку $n^i$ положительно, $|a_i|n^i$ может быть больше только $a_i n^i$.) Это означает, что

$$T(n) \leq |a_k| n^k + \ldots + |a_1| n + |a_0|.$$

Почему этот шаг полезен? Теперь, когда коэффициенты не могут быть отрицательными, мы можем использовать аналогичный метод, чтобы превратить разные степени $n$ в общую степень $n$. Поскольку $n \geq 1$, $n^k$ больше только $n^i$ для каждого $i \in \{0, 1, 2, ..., k\}$. Поскольку $|a_i|$ неотрицательно, $|a_i|n^k$ больше только $|a_i|n^i$. Это означает, что

$$T(n) \leq |a_k| n^k + \ldots + |a_1| n^k + |a_0| n^k = \underbrace{\left(|a_k| + \ldots + |a_1| + |a_0|\right)}_{=c} \times n^k.$$

Это неравенство справедливо для каждого $n \geq n_0 = 1$, то есть то, что мы и хотели доказать. *Ч. т. д.*

Как узнать, каким образом выбрать константы $c$ и $n_0$? Обычный подход — реконструировать их от обратного. Это реализуется путем прохождения вычисления, подобного приведенному выше, и выяснения на лету выбора

---

[1] Напомним, что *абсолютное* значение $|x|$ вещественного числа $x$ равняется $x$, когда $x \geq 0$, и $-x$, когда $x \leq 0$. В частности, $|x|$ всегда неотрицательно.

констант, которые позволяют вам провести доказательство. Мы увидим некоторые примеры этого метода в разделе 2.5.

### 2.3.2. Для многочленов степени *k* O-большим не является O ($n^{k-1}$)

Наш второй пример в действительности является непримером: многочлен степени $k$ является $O(n^k)$, однако не является $O(n^{k-1})$.

**Утверждение 2.2.** *Пусть $k > 1$ есть положительное целое число и задано $T(n) = n^k$. Тогда $T(n)$ не равно $O(n^{k-1})$.*

Из утверждения 2.2 следует, что многочлены с разными степенями различаются относительно обозначения *O*-большое. (Если бы это не было истиной, то что-то было бы неправильно с нашим определением обозначения *O*-большое!)

*Доказательство утверждения 2.2.* Чтобы доказать, что одна функция не является *O*-больше другой, как правило, пользуются доказательством от противного. В этом типе доказательства вы допускаете *противоположное* тому, что вы хотите доказать, а затем, основываясь на этом допущении, строите последовательность логически выверенных шагов, кульминацией которых является явно ложное утверждение. Из такого противоречия следует, что исходное противоположное допущение не может быть истинным, а это доказывает требуемое высказывание.

Итак, предположим обратное и допустим, что $n^k$ равно $O(n^{k-1})$; мы переходим к выводу противоречия. Что значит, если $n^k = O(n^{k-1})$? А то, что $n^k$ в конечном итоге ограничивается постоянным кратным $n^{k-1}$. То есть существуют положительные константы $c$ и $n_0$ такие, что

$$n^k \leq c \times n^{k-1}$$

для всех $n \geq n_0$. Поскольку $n$ — это положительное число, мы можем сократить $n^{k-1}$ из обеих частей этого неравенства, и в итоге получим

$$n \leq c$$

для всех $n \geq n_0$. Это неравенство утверждает, что константа $c$ больше любого положительного целого числа, что явно ложно (в качестве контрпримера возьмите $c + 1$, округленное до ближайшего целого). Полученное ложное утверждение говорит о том, что наше исходное допущение, что $n^k = O(n^{k-1})$, не может быть истиным, следовательно, $n^k$ не является $O(n^{k-1})$. *Ч. т. д.*

## 2.4. Обозначение Омега-большое и Тета-большое

Обозначение *O*-большое на сегодняшний день является наиболее важной и общепризнанной категорией для обсуждения асимптотического времени работы алгоритмов. Кроме него необходимо познакомиться с парой родственных понятий, это обозначения Омега-большое и Тета-большое. Если *O*-большое аналогично отношению «меньше или равно (≤)», то Омега-большое и Тета-большое аналогичны, соответственно, отношениям «больше или равно (≥)» и «равно (=)». Давайте теперь рассмотрим их немного подробнее.

### 2.4.1. Обозначение Омега-большое

Формальное определение обозначения Омега-большое совпадает с определением обозначения О-большое. На русском языке мы говорим, что одна функция $T(n)$ является Омега-большое другой функции $f(n)$ тогда и только тогда, когда $T(n)$ в конечном итоге ограничивается снизу константой, кратной $f(n)$. В этом случае мы пишем $T(n) = \Omega(f(n))$. Как и раньше, мы используем две константы, $c$ и $n_0$, для количественной оценки «постоянного кратного» и «в конечном итоге».

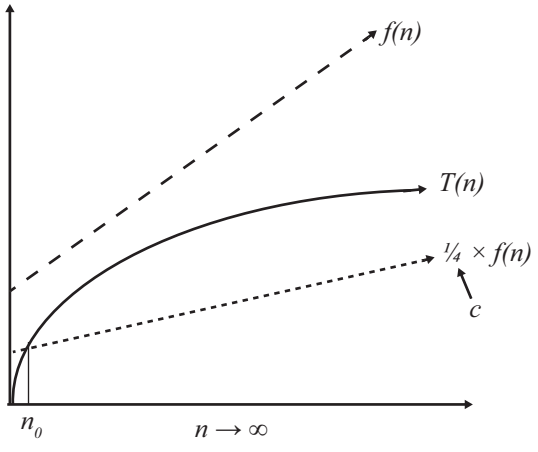
**ОБОЗНАЧЕНИЕ ОМЕГА-БОЛЬШОЕ (МАТЕМАТИЧЕСКАЯ ВЕРСИЯ)**

$T(n) = \Omega(f(n))$, тогда и только тогда существуют положительные константы $c$ и $n_0$ такие, что

$$T(n) \geq c \times f(n)$$

для всех $n \geq n_0$.

Вот как выглядит соответствующий рисунок:




Здесь $T(n)$ снова соответствует функции со сплошной линией. Функция $f(n)$ является верхней пунктирной линией. Эта функция не ограничивает $T(n)$ снизу, но если мы умножим ее на константу $c = 1/4$, то результат (нижняя пунктирная линия) будет ограничивать $T(n)$ снизу для всех $n$ после точки пересечения в $n_0$. Следовательно, $T(n) = \Omega(f(n))$.

## 2.4.2. Обозначение Тета-большое

Обозначение Тета-большое, или просто обозначение Тета, аналогично отношению «равно». Говоря, что $T(n) = \Theta(f(n))$ мы, в сущности, подразумеваем, что $T(n) = \Omega(f(n))$ и, одновременно, $T(n) = O(f(n))$. Эквивалентным образом $T(n)$ в конечном итоге зажата между двумя разными постоянными кратными $f(n)$[1].

[1] Доказательство этой эквивалентности сводится к демонстрации, что один вариант определения удовлетворяется тогда и только тогда, когда удовлетворяется другой. Если $T(n) = \Theta(f(n))$ согласно второму определению, то, взяв константы $c_2$ и $n_0$, мы видим, что $T(n) = O(f(n))$, а взяв константы $c_1$ и $n_0$, мы видим, что $T(n) = \Omega(f(n))$. С другой стороны, предположим, что вы можете доказать, что $T(n) = O(f(n))$, используя константы $c_2$ и $n_0'$, и $T(n) = \Omega(f(n))$, используя константы $c_1$ и $n_0''$. Затем $T(n) = \Theta(f(n))$ в значении второго определения, с константами $c_1$, $c_2$ и $n_0 = \max\{n_0', n_0''\}$.

**ОБОЗНАЧЕНИЕ ТЕТА-БОЛЬШОЕ (МАТЕМАТИЧЕСКАЯ ВЕРСИЯ)**

$T(n) = \Theta(f(n))$, тогда и только тогда существуют положительные константы $c_1$, $c_2$ и $n_0$ такие, что

$$c_1 \times f(n) \le T(n) \le c_2 \times f(n)$$

для всех $n \ge n_0$.

**ПРЕДОСТЕРЕЖЕНИЕ**

Разработчики алгоритмов часто используют обозначение *O*-большое, даже когда обозначение Тета-большое будет более точным. В этой книге мы будем следовать этой традиции. Например, рассмотрим функцию, которая просматривает массив длиной $n$, выполняя постоянное число операций на элемент (например, подпрограмму слияния `Merge` из раздела 1.4.5). Время работы такой подпрограммы, очевидно, равняется $\Theta(n)$, но обычно упоминается только о том, что оно $O(n)$. Это обусловлено тем, что как разработчики алгоритмов, так и мы в наших рассуждениях обычно сосредоточиваемся на верхних границах — гарантиях того, как долго наши алгоритмы могут выполняться.

Следующее тестовое задание предназначено для проверки вашего понимание обозначений О-большое, Омега-большое и Тета-большое.

**ТЕСТОВОЕ ЗАДАНИЕ 2.5**

Пусть $T(n) = \frac{1}{2}n^2 + 3n$. Какое из следующих высказываний истинное? (Может быть более одного правильного ответа.)

а) $T(n) = O(n)$.

б) $T(n) = \Omega(n)$.

в) $T(n) = \Theta(n^2)$.

г) $T(n) = O(n^3)$.

(Решение и пояснение см. в разделе 2.4.5.)

## 2.4.3. Обозначение o-малое

Есть еще один, последний вид асимптотического обозначения, обозначение *o*-малое, которое может встречаться время от времени. Если обозначение *O*-большое эквивалентно отношению «меньше или равно», то обозначение *o*-малое эквивалентно отношению «строго меньше чем»[1].

**ОБОЗНАЧЕНИЕ О-МАЛОЕ (МАТЕМАТИЧЕСКАЯ ВЕРСИЯ)**

$T(n) = o(f(n))$ тогда и только тогда, когда для каждой положительной константы $c > 0$ существует выбор вариантов $n_0$ такой, что

$$T(n) \leq c \times f(n) \qquad (2.2)$$

для всех $n \geq n_0$.

Для доказательства того, что одна функция является *O*-больше другой, требуется всего две константы, $c$ и $n_0$, выбранные раз и навсегда. Чтобы доказать, что одна функция является *o*-меньше другой, требуется доказать более сильное утверждение, а именно что для *каждой* константы $c$, независимо от того, насколько она мала, $T(n)$ в конечном итоге ограничивается сверху постоянным кратным $c \times f(n)$. Обратите внимание, что константа $n_0$, выбранная для количественной оценки «в конечном итоге», может зависеть от $c$ (но не от $n$!), причем меньшие константы $c$ обычно требуют больших констант $n_0$. Например, для каждого положительного целого числа $k$, $n^{k-1} = o(n^k)$[2].

[1] По аналогии, существует обозначение «омега-малое», которое соответствует «строго больше», но у нас не будет возможности его использовать. Обозначения «тета-малое» не существует.

[2] Приведем доказательство. Зададим произвольную константу $c > 0$. Соответственно, выберем $n_0$ равным $\frac{1}{c}$, округленным до ближайшего целого числа. Тогда для всех $n \geq n_0$, $n_0 \times n^{k-1} \leq n^k$ и, следовательно, $n^{k-1} \leq \frac{1}{n_0} \times n^k \leq c \times n^k$, что и требовалось доказать.

## 2.4.4. Откуда взялось обозначение?

Асимптотические обозначения не были изобретены специалистами по computer science — они использовались в теории чисел примерно с начала XX века. Дональд Кнут, родоначальник формального анализа алгоритмов, предложил их использовать в качестве стандартного языка для обсуждения темпов роста и, в частности, времени работы алгоритмов (цитата):

> *«На основе вопросов, обсуждаемых здесь, я предлагаю членам SIGACT*[1] *и редакторам журналов по computer science и математике принять обозначения O, Ω и Θ, как определено выше, если только в ближайшее время не будет найдена лучшая альтернатива»*[2].

## 2.4.5. Решение тестового задания 2.5

**Правильные ответы: (б), (в), (г).** Последние три ответа все правильные, и, надеюсь, интуиция подсказывает вам, почему это именно так. $T(n)$ — это квадратичная функция. Линейным членом $3n$ для больших $n$ можно пренебречь, поэтому следует ожидать, что $T(n) = \Theta(n^2)$ (ответ (c)). Это автоматически означает, что $T(n) = \Omega(n^2)$, и, следовательно, $T(n) = \Omega(n)$ также (ответ (b)). Обратите внимание: $\Omega(n)$ не является особо впечатляющей нижней границей $T(n)$, но тем не менее оно является законным. Аналогично, $T(n) = \Theta(n^2)$ означает, что $T(n) = O(n^2)$ также, и, следовательно, $T(n) = O(n^3)$ (ответ (d)). Доказательство этих высказываний формально сводится к демонстрации соответствующих констант, которые удовлетворяют определениям. Например, использование $n_0 = 1$ и $c = 1/2$ доказывает (b). Использование $n_0 = 1$ и $c = 4$ доказывает (d). Объединение этих наборов констант ($n_0 = 1$, $c_1 = 1/2$, $c_2 = 4$) доказывает (c). Для формального доказательства того, что

---

[1] SIGACT является специальной группой в рамках ACM (Ассоциация вычислительной техники), которая занимается теоретической информатикой, и в частности анализом алгоритмов.

[2] Дональд Кнут, «Большой Омикрон, и большая Омега, и большая Тета» (Donald E. Knuth, «Big Omicron and Big Omega and Big Theta», SIGACT News, Apr.-June 1976, page 23. Reprinted in Selected Papers on Analysis of Algorithms (Center for the Study of Language and Information, 2000).

(а) не является правильным ответом, может быть использован аргумент из доказательства утверждения 2.2.

## 2.5. Дополнительные примеры

Этот раздел предназначен для читателей, которым нужна дополнительная практика по асимптотическим обозначениям. Остальные могут пропустить эти три дополнительных примера и перейти непосредственно к главе 3.

### 2.5.1. Добавление константы к экспоненте

В первую очередь, это еще один пример доказательства того, как одна функция является *O*-больше другой.

**Утверждение 2.3.** *Если*

$$T(n) = 2^{n+10},$$

*то тогда* $T(n) = O(2^n)$.

Таким образом, добавление константы к показателю степени экспоненциальной функции не изменяет ее асимптотический темп роста.

*Доказательство утверждения 2.3.* Чтобы соответствовать математическому определению обозначения *O*-большое (раздел 2.2.3), достаточно найти подходящую пару положительных констант, $c$ и $n_0$ (каждая независима от $n$), таких, что $T(n)$ не более $c \times 2n$ для всех $n \geq n_0$. Доказывая утверждение 2.1, мы просто вытащили эти две константы из шляпы; сейчас же давайте обоснуем этот выбор.

Выведение формулы начинается с $T(n)$ в левой части, за которым следует последовательность только больших чисел, кульминацией которого является постоянное кратное $2n$. Как начнется такое вычисление? Наличие числа «10» в степени раздражает, поэтому, естественно, первым шагом является его выделение:

$$T(n) = 2^{n+10} = 2^{10} \times 2^n = 1024 \times 2^n.$$

Теперь уже лучше; правая часть формулы является постоянной кратной $2n$, и наш вывод предполагает, что мы должны взять $c = 1024$. Учитывая этот выбор $c$, мы имеем $T(n) \leq c \times 2n$ для всех $n \geq 1$, поэтому мы просто берем $n_0 = 1$. Эта пара констант подтверждает, что $T(n)$ действительно $O(2^n)$. *Ч. т. д.*

## 2.5.2. Умножение экспоненты на константу

Далее идет еще один «непример», показывающий, что одна функция не является $O$-больше другой.

**Утверждение 2.4.** *Если*

$$T(n) = 2^{10n},$$

*тогда* $T(n)$ *не является* $O(2^n)$.

Таким образом, умножение показателя степени экспоненциальной функции на константу изменяет ее асимптотический темп роста.

*Доказательство утверждения 2.4.* Как и в случае с утверждением 2.2, обычный способ доказать, что одна функция не является $O$-больше другой, — пойти от обратного. Поэтому допустим противоположное высказывание и предположим, что $T(n)$ на самом деле равняется $O(2^n)$. Согласно определению обозначения $O$-большое это означает, что существуют положительные константы $c$ и $n_0$ такие, что

$$2^{10n} \leq c \times 2^n$$

для всех $n \geq n_0$. Поскольку $2^n$ является положительным числом, мы можем сократить его из обеих частей этого неравенства, в итоге получим

$$2^{9n} \leq c$$

для всех $n \geq n_0$. Но это неравенство явно ложное: правая часть является фиксированной константой (независимой от $n$), в то время как левая часть стремится в бесконечность, когда $n$ становится большим. Это показывает, что наше допущение, что $T(n) = O(2^n)$, не может быть правильным, следовательно, $2^{10n}$ не является $O(2^n)$. *Ч. т. д.*

## 2.5.3. Максимум против суммы

В нашем последнем примере используется обозначение Тета-большое (раздел 2.4.2), асимптотический эквивалент отношения «равняется». Данный пример показывает, что асимптотически нет никакой разницы между взятием поточечного максимума двух неотрицательных функций и их суммой.

**Утверждение 2.5.** *Обозначим функции, преобразующие положительные целые числа в неотрицательные вещественные числа, как $f$ и $g$ и определим*

$$T(n) = \max\{f(n), g(n)\}$$

*для каждого $n \geq 1$. Тогда $T(n) = \Theta(f(n) + g(n))$.*

Одним из следствий утверждения 2.5 является то, что алгоритм, который выполняет постоянное число (имеется в виду независимое от $n$) $O(f(n))$-подпрограмм, выполняется за время $O(f(n))$.

*Доказательство утверждения 2.5.* Напомним, что $T(n) = \Theta(f(n))$ означает, что $T(n)$ в конечном итоге ограничена двумя разными постоянными, кратными $f(n)$. Чтобы соответствовать этому, нам нужно показать три константы: обычную константу $n_0$ и константы $c_1$ и $c_2$, соответствующие меньшим и бо́льшим кратным $f(n)$. Реконструируем значения этих констант.

Рассмотрим произвольное натуральное число $n$. Мы имеем

$$\max\{f(n), g(n)\} \leq f(n) + g(n),$$

поскольку правая часть — это просто левая часть плюс неотрицательное число ($f(n)$ или $g(n)$, в зависимости от того, какое меньше). Аналогично,

$$2 \times \max\{f(n), g(n)\} \geq f(n) + g(n),$$

поскольку левая часть является двумя копиями большего из $f(n)$, $g(n)$ и правая часть является одной копией каждого. Объединив эти два неравенства, мы видим, что

$$\frac{1}{2}\left(f(n) + g(n)\right) \leq \max\left\{f(n),\ g(n)\right\} \leq f(n) + g(n) \qquad (2.3)$$

для каждого $n \geq 1$. Таким образом, $\max\{f(n), g(n)\}$ действительно «вклинивается» между двумя разными кратными $f(n) + g(n)$. Формально выбор $n_0 = 1$, $c_1 = 1$ и $c_2 = 1$ показывает (по (2.3)), что $\max\{f(n), g(n)\} = \Theta(f(n) + g(n))$. *Ч. т. д.*

**ВЫВОДЫ**

- Целью использования асимптотических обозначений является устранение постоянных множителей (которые слишком зависимы от реализации алгоритма) и членов низших порядков (которые не имеют значения для больших входных данных).
- Говорят, что функция $T(n)$ является «О-большое от $f(n)$» (пишется как «$T(n) = O(f(n))$»), если она в конечном итоге (для достаточного большого $n$) ограничена сверху константой, кратной $f(n)$. То есть существуют положительные константы $c$ и $n_0$ такие, что $T(n) \leq c \times f(n)$ для всех $n \geq n_0$.
- Функция $T(n)$ является «Омега-большое функции $f(n)$» (пишется как «$T(n) = \Omega(f(n))$»), если она в конечном итоге ограничена снизу константой, кратной $f(n)$.
- Функция $T(n)$ является «Омега-большое от $f(n)$» (пишется как «$T(n) = \Theta(f(n))$»), если и $T(n) = O(f(n))$, и $T(n) = \Omega(f(n))$.
- *O*-большое эквивалентно отношению «меньше или равно», Омега-большое эквивалентно отношению «больше или равно» и Тета-большое эквивалентно отношению «равняется».

# Задачи на закрепление материала

**Задача 2.1.** Пусть $f$ и $g$ есть неубывающие вещественные функции, определенные на положительных целых числах, причем $f(n)$ и $g(n)$ не менее 1 для всех $n \geq 1$. Допустим, что $f(n) = O(g(n))$, и пусть $c$ есть положительная константа. Верно ли, что $f(n) \log_2(f(n)^c) = O(g(n) \times \log_2(g(n)))$?

а) Да, для всех таких $f$, $g$ и $c$.

б) Никогда, независимо от того, какими являются $f$, $g$ и $c$.

в) Иногда да, иногда нет, в зависимости от константы $c$.

г) Иногда да, иногда нет, в зависимости от функций $f$ и $g$.

**Задача 2.2.** Еще раз допустим две положительные неубывающие функции, $f$ и $g$, такие, что $f(n) = O(g(n))$. Верно ли, что $2^{f(n)} = O(2^{g(n)})$? (Правильных ответов может быть несколько; выберите все верные.)

а) Да, для всех таких $f$ и $g$.

б) Никогда, независимо от того, какими являются $f$ и $g$.

в) Иногда да, иногда нет, в зависимости от функций $f$ и $g$.

г) Да, всякий раз, когда $f(n) \leq g(n)$ для всех достаточно больших $n$.

**Задача 2.3.** Отсортируйте перечисленные ниже функции в порядке увеличения темпа роста, причем в вашем списке $g(n)$ должна следовать за $f(n)$ тогда и только тогда, когда $f(n) = O(g(n))$.

а) $\sqrt{n}$.

б) $10^n$.

в) $n^{1,5}$.

г) $2^{\sqrt{\log_2 n}}$.

д) $n^{5/3}$.

**Задача 2.4.** Отсортируйте перечисленные ниже функции в порядке увеличения темпа роста, причем в вашем списке $g(n)$ должна следовать за $f(n)$, тогда и только тогда $f(n) = O(g(n))$.

а) $n^2 \log_2 n$.

б) $2n$.

в) $2^{2^n}$.

г) $n^{\log_2 n}$.

д) $n^2$.

**Задача 2.5.** Отсортируйте перечисленные ниже функции в порядке увеличения темпа роста, причем в вашем списке $g(n)$ должна следовать за $f(n)$ тогда и только тогда, когда $f(n) = O(g(n))$.

а) $2^{\log_2 n}$.

б) $2^{2^{\log_2 n}}$.

в) $n^{5/2}$.

г) $2^{n^2}$.

д) $n^2 \log_2 n$.

# 3

# *Алгоритмы «разделяй и властвуй»*

Эта глава посвящена практике парадигмы разработки алгоритмов «разделяй и властвуй» применительно к трем основным задачам. Первый пример — это алгоритм подсчета количества инверсий[1] массива (раздел 3.2). Данная задача предназначена для измерения подобия между двумя ранжированными списками. Это так называемая «совместная фильтрация», когда, опираясь на ваши знания о предпочтениях человека и предпочтениях других людей, ему предоставляются полезные рекомендации. Второй алгоритм типа «разделяй и властвуй» — это восхитительный рекурсивный алгоритм Штрассена для умножения матриц, который работает лучше по сравнению с очевидным итеративным методом (раздел 3.3). Третий алгоритм, который является продвинутым и дополнительным материалом, предназначен для решения фундаментальной задачи вычислительной геометрии: вычисление ближайшей пары точек на плоскости (раздел 3.4)[2].

## 3.1. Парадигма «разделяй и властвуй»

Мы уже рассматривали канонический пример алгоритма «разделяй и властвуй», `MergeSort` (раздел 1.4). В более общем плане парадигма разработки алгоритмов «разделяй и властвуй» состоит из трех концептуальных шагов.

**ПАРАДИГМА «РАЗДЕЛЯЙ И ВЛАСТВУЙ»**

1. *Разделить* входные данные на более мелкие подзадачи.
2. *Решить* подзадачи рекурсивным методом.
3. *Объединить* решения подзадач в решение исходной задачи.

[1] В информатике и дискретной математике последовательность имеет инверсию там, где два ее элемента находятся вне своего естественного порядка. Например, пусть дана числовая последовательность 1, 2, 5, 7, 4, 6, тогда пары (5, 4), (7, 4) и (7, 6) образуют инверсии в этой последовательности. — *Примеч. пер.*

[2] Демонстрация в разделах 3.2 и 3.4 черпает вдохновение из главы 5, «Проектирование алгоритмов», Джона Клейнберга и Эвы Тардос (Algorithm Design, Jon Kleinberg, Éva Tardos Pearson, 2005).

Например, в алгоритме `MergeSort` шаг «разделить» разбивает входной массив на левую и правую половины, шаг «решить» использует два рекурсивных вызова для сортировки левого и правого подмассивов, а шаг «объединить» реализуется подпрограммой слияния `Merge` (раздел 1.4.5). В алгоритме `MergeSort` и многих других алгоритмах этот последний шаг требует наибольшей изобретательности. Существуют также алгоритмы типа «разделяй и властвуй», в которых необходимо быть изобретательным уже на первом шаге (см. `QuickSort` в главе 5) или же в спецификации рекурсивных вызовов (раздел 3.2).

## 3.2. Подсчет инверсий за время O (*n* log *n*)

### 3.2.1. Задача

В этом разделе рассматривается задача вычисления количества инверсий массива. *Инверсия* массива — это пара элементов, которые расположены «вне своего естественного порядка». Это означает, что элемент, который в массиве встречается ранее, больше, чем тот, который встречается позже.

**ЗАДАЧА: ПОДСЧЕТ ИНВЕРСИЙ**

**Вход**: массив $A$ разных целых чисел.

**Выход**: количество инверсий — число пар $(i, j)$ индексов массива, где $i < j$ и $A[i] > A[j]$.

Например, если массив $A$ отсортирован, он не имеет инверсий. Вы должны убедиться, что обратное также верно: каждый неотсортированный массив имеет по крайней мере одну инверсию.

### 3.2.2. Пример

Рассмотрим следующий ниже массив длиной 6:

| 1 | 3 | 5 | 2 | 4 | 6 |
|---|---|---|---|---|---|

Сколько инверсий имеет этот массив? В глаза бросаются значения 5 и 2 (соответствуя $i = 3$ и $j = 4$). Имеются две другие неупорядоченные пары: 3 и 2, а также 5 и 4.

**ТЕСТОВОЕ ЗАДАНИЕ 3.1**

Каково наибольшее количество инверсий для 6-элементного массива?

а) 15

б) 21

в) 36

г) 64

(Решение и пояснение см. в разделе 3.2.13.)

## 3.2.3. Совместная фильтрация

Для чего необходимо подсчитывать количество инверсий в массиве? Одна из причин заключается в том, чтобы вычислить числовую меру качественного подобия, которая количественно определяет, насколько близки два ранжированных списка друг к другу. Например, предположим, что я прошу вас и вашего друга ранжировать десять фильмов, которые вы оба смотрели, от любимого до наименее любимого. Являются ли ваши вкусы «похожими» или же они «различаются»? Один из способов решить эту задачу количественно — использовать 10-элементный массив $A$: пусть $A[1]$ содержит оценку вашим другом вашего любимого фильма в его списке, $A[2]$ — личную оценку вашим другом вашего второго любимого фильма... и $A[10]$ — личную оценку вашим другом вашего наименее любимого фильма (вы его поставили на 10-е место). Таким образом, если вашим любимым фильмом является «Звездные войны», но у вашего друга он только пятый в его списке, то $A[1] = 5$. Если ваши рейтинги идентичны, этот массив будет отсортирован и не будет иметь инверсий. Чем больше инверсий массив имеет, тем больше пар фильмов, по которым вы не сходитесь во мнении об их достоинствах, и тем больше различаются ваши предпочтения.

Одна из задач, для которой вам может понадобиться вычисление меры подобия рейтингов, — это *совместная фильтрация* (collaborative filtering), способ создания рекомендаций. Каким образом веб-сайты находят рекомендации для товаров, фильмов, песен, новостей и так далее? При использовании совместной фильтрации идея состоит в том, чтобы идентифицировать других людей, которые имеют аналогичные с вами предпочтения, чтобы затем рекомендовать вам товары, которые были популярны у них. Следовательно, алгоритмы совместной фильтрации осуществляют математическую формализацию качественного понятия «подобия» между предпочтениями людей. Вычисление инверсий частично помогает в этом вопросе.

## 3.2.4. Поиск полным перебором

Как быстро мы можем вычислить количество инверсий в массиве? Если мы лишены воображения, то в нашем распоряжении есть полный перебор, или метод «грубой силы» (brute force).

**ПОЛНЫЙ ПЕРЕБОР ДЛЯ ПОДСЧЕТА ИНВЕРСИЙ**

**Вход**: массив $A$ из $n$ разных целых чисел.

**Выход**: количество инверсий массива $A$.

```
numInv := 0
for i := 1 to n – 1 do
    for j := i + 1 to n do
        if A[i] > A[j] then
            numInv := numInv + 1
return numInv
```

Это, безусловно, правильный алгоритм. А как насчет времени его исполнения? Из решения тестового задания 3.1 мы знаем, что количество итераций цикла возрастает квадратично в зависимости от длины $n$ входного массива. Поскольку алгоритм выполняет постоянное число операций в каждой итерации цикла, его асимптотическое время исполнения равняется $\Theta(n^2)$. Но помните ли вы мантру опытного проектировщика алгоритмов: *можно ли добиться лучшего*?

### 3.2.5. Подход «разделяй и властвуй»

Ответ: да, лучшего добиться можно, и решением будет алгоритм типа «разделяй и властвуй», который исполняется за время $O(n \log n)$, что гораздо лучше по сравнению с алгоритмом поиска методом «грубой силы». Шаг «разделить» будет точно таким же, как и в алгоритме `MergeSort`, с одним рекурсивным вызовом для левой половины массива и одним для правой половины. Чтобы понять остальные операции, которые должны быть выполнены вне двух рекурсивных вызовов, давайте отнесем инверсии $(i, j)$ массива $A$ длиной $n$ к одному из трех типов.

1. *Левая инверсия*: инверсия, в которой оба индекса, $i, j$, находятся в первой половине массива (то есть $i, j \leq \frac{n}{2}$).
2. *Правая инверсия*: инверсия, в которой оба индекса, $i, j$, находятся во второй половине массива (то есть $i, j > \frac{n}{2}$).
3. *Разделенная инверсия*: инверсия, в которой $i$ находится в левой половине, а $j$ — в правой половине (то есть $i \leq \frac{n}{2} < j$).

Например, в шестиэлементном массиве в разделе 3.2.2 все три инверсии являются разделенными.

Первый рекурсивный вызов в первой половине входного массива рекурсивно подсчитывает все левые инверсии (и больше ничего). Точно так же второй рекурсивный вызов подсчитывает все правые инверсии. Дальнейшая задача состоит в подсчете инверсий, не выявленных ни одним рекурсивным вызовом, — разделенных инверсий. Это шаг «объединить» алгоритма, и нам нужно будет реализовать для него специальную линейную функцию, аналогичную подпрограмме слияния `Merge` в алгоритме `MergeSort`.

### 3.2.6. Высокоуровневый алгоритм

Наш подход «разделяй и властвуй» транслируется в приведенный ниже псевдокод; подпрограмма подсчета разделенных инверсий `CountSplitInv` на данный момент не реализована.

**COUNTINV**

**Вход**: массив $A$ из $n$ разных целых чисел.

**Выход**: количество инверсий массива $A$.

```
if n = 0 or n = 1 then                    // базовые случаи
    return 0
else
    leftInv := CountInv(первая половина A)
    rightInv := CountInv(вторая половина A)
    splitInv := CountSplitInv(A)
    return leftInv + rightInv + splitInv
```

Первый и второй рекурсивные вызовы подсчитывают количество левых и правых инверсий. При условии, что подпрограмма `CountSplitInv` правильно вычисляет количество разделенных инверсий, алгоритм `CountInv` правильно вычислит общее количество инверсий.

## 3.2.7. Ключевая идея: задействовать алгоритм MergeSort

Подсчет числа разделенных инверсий массива за линейное время является амбициозной целью. Может существовать много разделенных инверсий: если $A$ состоит из чисел $\frac{n}{2}+1,\ldots,n$ по порядку, за которыми идут числа 1, 2, ... , $\frac{n}{2}$ по порядку, то имеется $n^2/4$ разделенных инверсий. Как вообще подсчитать квадратичное количество элементов только в рамках линейного объема работы?

Движущая идея состоит в том, чтобы разработать наш рекурсивный алгоритм подсчета инверсий таким образом, чтобы он задействовал алгоритм `MergeSort`. Это требует от наших рекурсивных вызовов большего ради того, чтобы облегчить подсчет количества разделенных инверсий[1]. Каждый ре-

[1] Аналогичным образом, иногда доказательство по индукции становится легче реализовать после обоснования вашей индуктивной гипотезы.

курсивный вызов будет отвечать не только за подсчет количества инверсий в массиве, который ему передан, но и за возврат отсортированной версии массива. Мы уже знаем (из теоремы 1.2), что сортировка является легковесным примитивом (раздел 1.6.4), исполняющимся за время $O(n \log n)$, поэтому, если мы нацелены на временную границу $O(n \log n)$, нет причин не использовать сортировку. Мы увидим, что задача слияния двух отсортированных подмассивов идеально подходит для обнаружения всех разделенных инверсий массива.

Вот пересмотренная версия псевдокода из раздела 3.2.6, которая подсчитывает инверсии, а также сортирует входной массив:

**SORT-AND-COUNTINV**

**Вход**: массив $A$ из $n$ разных целых чисел.

**Выход**: отсортированный массив $B$ с теми же самыми целыми числами и количество инверсий массива $A$.

```
if n = 0 or n = 1 then                          // базовые случаи
   return (A,0)
else
    (C, leftInv) := Sort-and-CountInv(первая половина A)
    (D, rightInv) :=
      Sort-and-CountInv(вторая половина A)
    (B, splitInv) := Merge-and-CountSplitInv(C, D)
    return (B, leftInv + rightInv + splitInv)
```

Нам все еще нужно реализовать подпрограмму `Merge-and-CountSplitInv`. Мы знаем, как объединить два отсортированных списка за линейное время, но как воспользоваться этим алгоритмом так, чтобы также подсчитать количество разделенных инверсий?

### 3.2.8. К вопросу о подпрограмме Merge

Чтобы понять, почему объединение отсортированных подмассивов естественным образом обнаруживает разделенные инверсии, давайте вернемся к псевдокоду подпрограммы слияния `Merge`.

**MERGE**

**Вход**: отсортированные массивы $C$ и $D$ (длиной $n/2$ каждый).

**Выход**: отсортированный массив $B$ (длиной $n$).

**Упрощающее допущение**: $n$ — четное.

```
i := 1, j := 1
for k := 1 to n do
    if C[i] < D[j] then
        B[k] := C[i], i := i + 1
    else                                    // D[j] < C[i]
        B[k] := D[j], j := j + 1
```

Обратите внимание на то, что подпрограмма слияния `Merge` выполняет перемещение по одному индексу вниз параллельно по каждому из отсортированных подмассивов ($i$ для $C$ и $j$ для $D$), заполняя выходной массив ($B$) слева направо в отсортированном порядке (используя индекс $k$). На каждой итерации цикла подпрограмма определяет наименьший элемент, который еще не скопирован в $B$. Поскольку $C$ и $D$ отсортированы и все элементы вплоть до $C[i]$ и $D[j]$ уже были скопированы в $B$, единственными двумя кандидатами являются $C[i]$ и $D[j]$. Подпрограмма определяет, какой из двух меньше, а затем копирует его в следующую позицию выходного массива.

Какое отношение имеет подпрограмма `Merge` к подсчету количества инверсий? Начнем с частного случая массива $A$, который вообще не имеет разделенных инверсий — каждая инверсия массива $A$ является либо левой, либо правой инверсией.

**ТЕСТОВОЕ ЗАДАНИЕ 3.2**

Предположим, что входной массив $A$ не имеет разделенных инверсий. Как соотносятся между собой отсортированные подмассивы $C$ и $D$?

а) $C$ содержит наименьший элемент массива $A$, $D$ — второй наименьший, потом $C$ — третий наименьший и так далее.

б) Все элементы массива $C$ меньше всех элементов в $D$.

в) Все элементы массива $C$ больше всех элементов в $D$.

г) Для ответа на этот вопрос недостаточно информации.

(Решение и пояснение см. в разделе 3.2.13.)

Решив тестовое задание 3.2, вы увидите, что подпрограмма `Merge` исполняется довольно предсказуемо на массиве без разделенных инверсий. Поскольку каждый элемент в $C$ меньше каждого элемента в $D$, наименьший остающийся элемент всегда находится в $C$ (до тех пор, пока не закончатся элементы в $C$). Таким образом, подпрограмма `Merge` просто объединяет $C$ и $D$ — сначала она копирует все, что находится в $C$, а затем — все, что в $D$. Это наводит на мысль, что, возможно, разделенные инверсии имеют какое-то отношение к количеству элементов, остающихся в $C$, когда элемент $D$ копируется в выходной массив.

## 3.2.9. Подпрограмма Merge и разделенные инверсии

Чтобы развить нашу интуитивную догадку дальше, давайте рассмотрим исполнение алгоритма `MergeSort` на шестиэлементном массиве $A = \{1, 3, 5, 2, 4, 6\}$ из раздела 3.2.2; см. также рис. 3.1. Левая и правая половины этого массива уже отсортированы, поэтому нет ни левых, ни правых инверсий, соответственно, два рекурсивных вызова возвращают 0. В первой итерации подпрограммы `Merge` первый элемент $C$ («1») копируется в $B$. Это ничего не говорит о каких-либо разделенных инверсиях, и действительно нет никаких разделенных инверсий, которые содержат этот первый элемент. Однако во

второй итерации элемент «2» копируется в выходной массив, даже если *C* по-прежнему содержит элементы «3» и «5». Это выявляет две разделенные инверсии массива *A* — две такие инверсии включают «2». В третьей итерации «3» копируется из *C*, и нет последующих разделенных инверсий, которые содержат этот элемент. Когда «4» копируется из *D*, массив *C* все еще содержит «5», и это копирование выявляет третью и последнюю разделенную инверсию массива *A* (с участием «5» и «2»).

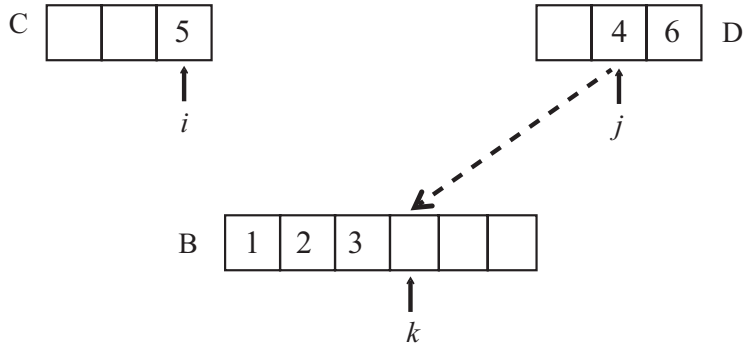



**Рис. 3.1.** Четвертая итерация подпрограммы Merge с учетом отсортированных подмассивов {1, 3, 5} и {2, 4, 6}. Копирование «4» из D, причем «5» все еще в C, выявляет разделенную инверсию с участием этих двух элементов

Приведенная ниже лемма констатирует, что подход, продемонстрированный в приведенном выше примере, соблюдается в общем случае: количество разделенных инверсий, которые содержат элемент *y* второго подмассива *D*, является в точности количеством элементов, остающихся в *C* в итерации подпрограммы `Merge`, в которой *y* копируется в выходной массив.

**Лемма 3.1.** *Пусть A есть массив, и C и D — отсортированные версии первой и второй половин A. Элемент x из первой половины A и элемент y из второй половины A образуют разделенную инверсию, тогда и только тогда в подпрограмме* `Merge` *с входными массивами C и D элемент y копируется в выходной массив перед элементом x.*

*Доказательство:* поскольку выходной массив заполняется слева направо в отсортированном порядке, то копируется сначала меньший из элементов *x*

или $y$. Поскольку $x$ находится в первой половине $A$, а $y$ находится во второй половине, $x$ и $y$ образуют разделенную инверсию, тогда и только тогда $x > y$, и это верно, тогда и только тогда $y$ копируется в выходной массив перед $x$. *Ч. т. д.*

## 3.2.10. Подпрограмма Merge-CountSplitInv

Использовав лемму 3.1, мы можем расширить реализацию подпрограммы `Merge` до реализации `Merge-and-CountSplitInv`. Мы будем наращивать выявленное количество разделенных инверсий, и всякий раз, когда элемент копируется из второго подмассива $D$ в выходной массив $B$, мы увеличиваем нарастающее количество на количество элементов, оставшихся в первом подмассиве $C$.

**MERGE-COUNTSPLITINV**

**Вход**: отсортированные массивы $C$ и $D$ (длиной $n/2$ каждый).

**Выход**: отсортированный массив $B$ (длиной $n$) и количество разделенных инверсий.

**Упрощающее допущение**: $n$ — четное.

```
i := 1, j := 1, splitInv := 0
for k := 1 to n do
    if C[i] < D[j] then
        B[k] := C[i], i := i + 1
    else                                        // D[j] < C[i]
        B[k] := D[j], j := j + 1
        splitInv := splitInv + (n/2 − i + 1)
                                # оставшееся кол-во в C
return (B, splitInv)
```

### 3.2.11. Корректность

Правильность подпрограммы `Merge-and-CountSplitlnv` вытекает из леммы 3.1. Каждая разделенная инверсия включает ровно один элемент $y$ из второго подмассива, и эта инверсия учитывается ровно один раз, когда $y$ копируется в выходной массив. Из этого вытекает корректность всего алгоритма `Sort-and-CountInv` (раздел 3.2.7): первый рекурсивный вызов корректно вычисляет количество левых инверсий, второй рекурсивный вызов — количество правых инверсий, подпрограмма `Merge-CountSplitlnv` — остальные (разделенные) инверсии.

### 3.2.12. Время исполнения

Мы также можем проанализировать время исполнения алгоритма `Sort-and-CountInv`, оперевшись на анализ, который мы уже делали для алгоритма `MergeSort`. Сначала рассмотрим время исполнения одного вызова подпрограммы `Merge-and-CountSplitInv` с учетом двух подмассивов длиной $\ell/2$ каждый. Как и подпрограмма `Merge`, она выполняет постоянное число операций на итерацию цикла плюс постоянное число дополнительных операций за время исполнения $O(\ell)$.

Оглядываясь на наш анализ времени исполнения алгоритма `MergeSort` в разделе 1.5, мы видим, что имелось три важных свойства алгоритма, которые привели к временной границе $O(n \log n)$. Во-первых, каждый вызов алгоритма делает два рекурсивных вызова. Во-вторых, длина входных данных делится пополам с каждым уровнем рекурсии. В-третьих, объем операций, выполняемых в рекурсивном вызове, не считая операций, выполняемых последующими рекурсивными вызовами, линейно зависит от размера входных данных. Поскольку алгоритм `Sort-and-CountInv` также характеризуется этими тремя свойствами, анализ в разделе 1.5 корректен и для этого алгоритма, соответственно, для него также верна временная граница $O(n \log n)$.

**Теорема 3.2 (подсчет инверсий).** *Для каждого входного массива $A$ длиной $n \geq 1$ алгоритм сортировки и подсчета* `Sort-and-CountInv` *вычисляет количество инверсий массива $A$ и выполняется за время $O(n \log n)$.*

### 3.2.13. Решения тестовых заданий 3.1 – 3.2

#### Решение тестового задания 3.1

**Правильный ответ: (а).** Правильный ответ на этот вопрос — 15. Максимально возможное количество инверсий не больше количества способов выбора $i, j \in \{1, 2, ..., 6\}$, где $i < j$. Последнее количество обозначается как $\binom{6}{2}$, то есть «из 6 по 2». В общем случае, $\binom{n}{2} = \frac{n(n-1)}{2}$, и поэтому $\binom{6}{2} = 15$. В шестиэлементном массиве, отсортированном в обратном порядке (6, 5, ..., 1), каждая пара элементов не упорядочена, и поэтому этот массив содержит 15 инверсий.

#### Решение тестового задания 3.2

**Правильный ответ: (б)**. В массиве без разделенных инверсий все элементы в первой половине меньше всех элементов во второй половине. Если бы в первой половине *был* элемент $A[i]$ (где $i \in \left\{1,\ 2,\ \ldots,\ \frac{n}{2}\right\}$), который больше элемента $A[j]$ во второй половине (где $j \in \left\{\frac{n}{2}+1,\ \frac{n}{2}+2,\ \ldots,\ n\right\}$), то пара $(i, j)$ образовывала бы разделенную инверсию.

## 3.3. Умножения матриц по алгоритму Штрассена

В этом разделе рассматривается применение парадигмы проектирования алгоритмов «разделяй и властвуй» к задаче умножения матриц. В результате будет продемонстрирован восхитительный алгоритм умножения матриц Штрассена субкубического времени исполнения. Этот алгоритм является каноническим примером волшебства и силы продуманной алгоритмизации и того, как алгоритмическая изобретательность может улучшить простые решения даже для чрезвычайно фундаментальных задач.

## 3.3.1. Умножение матриц

Пусть **X** и **Y** есть матрицы размером $n \times n$ целых чисел — $n^2$ элементов в каждой. В произведении матриц $\mathbf{Z} = \mathbf{X} \times \mathbf{Y}$ элемент $z_{ij}$ в $i$-й строке и $j$-м столбце матрицы **Z** определяется как скалярное произведение $i$-й строки **X** и $j$-го столбца **Y** (рис. 3.2)[1]. То есть

$$z_{ij} = \sum_{k=1}^{n} x_{ik} y_{kj} . \tag{3.1}$$

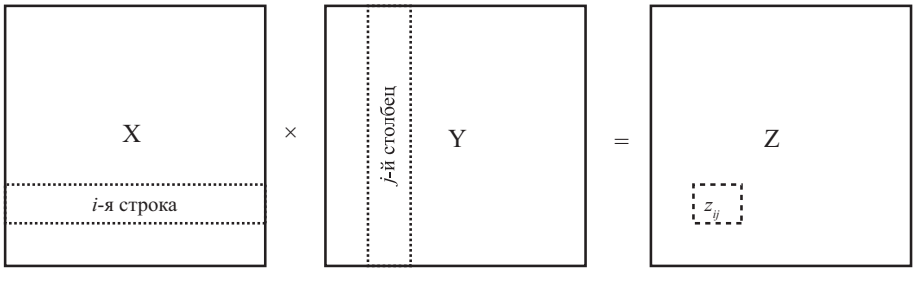



**Рис. 3.2.** Элемент ($i$, $j$) произведения матриц $X \times Y$ представляет собой скалярное произведение $i$-й строки $X$ и $j$-го столбца $Y$

## 3.3.2. Пример (n = 2)

Давайте подробно рассмотрим простой случай умножения матриц для $n = 2$. Мы можем описать две матрицы 2×2, использовав восемь параметров:

$$\underbrace{\begin{pmatrix} a & b \\ c & d \end{pmatrix}}_{\mathrm{X}} \text{ и } \underbrace{\begin{pmatrix} e & f \\ g & h \end{pmatrix}}_{\mathrm{Y}}$$

В произведении матриц $\mathbf{X} \times \mathbf{Y}$ левый верхний элемент является скалярным (точечным) произведением первой строки **X** и первого столбца **Y**, или $ae + bg$. В общем случае, для указанных выше **X** и **Y**,

$$\mathbf{X} \times \mathbf{Y} = \begin{pmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{pmatrix} \tag{3.2}$$

[1] Для вычисления скалярного произведения двух векторов, $a = (a_1, ..., a_n)$ и $b = (b_1, ..., b_n)$ длиной $n$, надо сложить результаты умножения покомпонентно: $a \times b = \sum_{i=1}^{n} a_i b_i$.

### 3.3.3. Простой алгоритм

Рассмотрим алгоритм вычисления произведения двух матриц.

**ЗАДАЧА: УМНОЖЕНИЕ МАТРИЦ**

**Вход**: две целочисленные матрицы размером $n \times n$, $\mathbf{X}$ и $\mathbf{Y}$[1].

**Выход**: произведение матриц $\mathbf{X} \times \mathbf{Y}$.

Размер входных данных пропорционален $n^2$, количеству элементов в каждой из матриц $\mathbf{X}$ и $\mathbf{Y}$. Поскольку мы, разумеется, должны прочитать входные данные и записать выходные данные, то лучшее, на что мы можем надеяться, — это алгоритм со временем работы $O(n^2)$ — линейный по размеру входных данных и квадратичный по размерности. Насколько мы можем приблизиться к этому наилучшему сценарию?

Существует простой алгоритм умножения матриц, который просто переводит математическое определение в код.

**ПРОСТОЕ УМНОЖЕНИЕ МАТРИЦ**

**Вход**: целочисленные матрицы размером $n \times n$, $\mathbf{X}$ и $\mathbf{Y}$.

**Выход**: $\mathbf{Z} = \mathbf{X} \times \mathbf{Y}$.

```
for i := 1 to n do
    for j := 1 to n do
        Z[i][j] := 0
        for k := 1 to n do
            Z[i][j] := Z[i][j] + X[i][k] × Y[k][j]
return Z
```

[1] Алгоритмы, которые мы обсуждаем, также могут быть доработаны для умножения неквадратных матриц, но для простоты мы будем придерживаться квадратного случая.

Каково время исполнения этого алгоритма?

**ТЕСТОВОЕ ЗАДАНИЕ 3.3**

Каково асимптотическое время исполнения простого алгоритма умножения матриц как функции размерности $n$ матрицы? При этом предположим, что сложение или умножение двух элементов матрицы является операцией с постоянным временем.

а) $\Theta(n \log n)$.

б) $\Theta(n^2)$.

в) $\Theta(n^3)$.

г) $\Theta(n^4)$.

(Решение и пояснение см. в разделе 3.3.7.)

## 3.3.4. Подход «разделяй и властвуй»

Алгоритмическая изобретательность, как обычно, исходит из запроса: *можно ли добиться лучшего*? Первой реакцией будет заключение о том, что умножение матрицы должно, в сущности, по определению занимать время $\Omega(n^3)$. Но, вероятно, на вас уже оказал влияние успех алгоритма целочисленного умножения `Karatsuba` (раздел 1.3), где продуманное применение подхода «разделяй и властвуй» существенно улучшает простой школьный алгоритм умножения в столбик[1]. Может ли аналогичный подход работать и для умножения матриц?

Чтобы применить парадигму «разделяй и властвуй» (раздел 3.1), нам нужно выяснить, как разделить входные данные на более мелкие подзадачи и как объединить решения этих подзадач в решение исходной задачи. Самый простой способ разделить квадратную матрицу на меньшие квадратные подматрицы — разрезать ее пополам, как по вертикали, так и по горизонтали. Другими словами, написать

[1] На самом деле мы еще не доказали это, но сделаем в разделе 4.3.

$$\mathbf{X} = \begin{pmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{pmatrix} \text{ и } \mathbf{Y} = \begin{pmatrix} \mathbf{E} & \mathbf{F} \\ \mathbf{G} & \mathbf{H} \end{pmatrix} \tag{3.3}$$

где **A**, **B**, ..., **H** — это матрицы размером $\frac{n}{2} \times \frac{n}{2}$ [1].

Одна восхитительная особенность умножения матриц заключается в том, что блоки одинакового размера ведут себя так же, как отдельные элементы матрицы. То есть для приведенных выше **X** и **Y** мы имеем

$$\mathbf{X} \times \mathbf{Y} = \begin{pmatrix} \mathbf{A} \times \mathbf{E} + \mathbf{B} \times \mathbf{G} & \mathbf{A} \times \mathbf{F} + \mathbf{B} \times \mathbf{H} \\ \mathbf{C} \times \mathbf{E} + \mathbf{D} \times \mathbf{G} & \mathbf{C} \times \mathbf{F} + \mathbf{D} \times \mathbf{H} \end{pmatrix} \tag{3.4}$$

что полностью аналогично уравнению (3.2) для случая $n = 2$. (Это следует из определения умножения матриц, в чем вы должны убедиться самим.) В (3.4) сложение двух матриц просто означает их поэлементное сложение — элемент $(i, j)$ матриц $\mathbf{K} + \mathbf{L}$ является суммой элементов $(i, j)$ матриц **K** и **L**. Разложение матриц и последующие вычисления в (3.4) транслируются естественным образом в рекурсивный алгоритм умножения матриц, `RecMatMult`.

---

**RECMATMULT**

**Вход**: целочисленные матрицы размером $n \times n$, **X** и **Y**.

**Выход**: $\mathbf{Z} = \mathbf{X} \times \mathbf{Y}$.

**Допущение:** $n$ является степенью 2.

---

```
if n = 1 then                                    // базовый случай
  return 1×1 – матрица с элементом X[1][1] × Y[1][1]
else                                             // рекурсивный случай
  A, B, C, D := подматрицы X как в (3.3)
  E, F, G, H := подматрицы Y как в (3.3)
  рекурсивно вычислить восемь матричных произведений,
  которые появляются в (3.4)
  return результат вычисления в (3.4)
```

---

[1] Как обычно, для удобства мы принимаем допущение, что $n$ — четное. И, как обычно, это не имеет значения.

Время исполнения алгоритма `RecMatMult` не сразу очевидно. Ясно то, что существует восемь рекурсивных вызовов, каждый с входными данными половинной размерности. Помимо выполнения этих рекурсивных вызовов, требуется только выполнить сложения матриц согласно (3.4). Поскольку матрица $n \times n$ имеет $n^2$ элементов, а число операций, необходимых для сложения двух матриц, пропорционально количеству элементов, то рекурсивный вызов с парой матриц $\ell \times \ell$ выполняет $\Theta(\ell^2)$ операций, не считая вычислений, выполняемых собственными рекурсивными вызовами.

К сожалению, этот рекурсивный алгоритм имеет время работы $\Theta(n^3)$, такое же, как и у простого алгоритма.

(Это следует из «основного метода», который будет подробно объяснен в следующей главе.)

Итак, неужели все наши усилия были напрасны? Помните, что в задаче целочисленного умножения ключевую роль в победе над алгоритмом умножения в столбик сыграл метод Гаусса, который уменьшил количество рекурсивных вызовов с четырех до трех (раздел 1.3.3). Существует ли аналог метода Гаусса для умножения матриц, который позволяет нам сократить количество рекурсивных вызовов с восьми до семи?

### 3.3.5. Экономия времени на рекурсивном вызове

Высокоуровневый план алгоритма `Strassen` заключается в экономии на одном рекурсивном вызове относительно алгоритма `RecMatMult` в обмен на постоянное число дополнительных матричных сложений и вычитаний.

**STRASSEN (КРАЙНЕ ВЫСОКОУРОВНЕВОЕ ОПИСАНИЕ)**

**Вход**: целочисленные матрицы размером $n \times n$, **X** и **Y**.

**Выход**: $\mathbf{Z} = \mathbf{X} \times \mathbf{Y}$.

**Допущение**: $n$ является степенью числа 2.

```
if n = 1 then                                    // базовый случай
  return 1×1 – матрица с элементом X[1][1] × Y[1][1]
else                                             // рекурсивный случай
  A, B, C, D := подматрицы X, как в (3.3)
  E, F, G, H := подматрицы Y, как в (3.3)
  Рекурсивно вычислить семь (предварительно отобранных)
  произведений
  С участием A, B, . . . , H
  return соответствующие (предварительно отобранные)
  сложения и вычитания матриц, вычисленных на предыдущем
  шаге
```

Экономия времени на одном из восьми рекурсивных вызовов — это большая победа. Она не просто уменьшает время работы алгоритма на 12,5 %. Рекурсивный вызов экономится снова и снова, так что экономия суммируется и — обратите внимание — это приводит к асимптотически превосходному времени исполнения. Мы узнаем точную границу времени работы в разделе 4.3, но сейчас главное понимать, что экономия на рекурсивном вызове дает алгоритм с субкубическим временем исполнения.

Собственно, на этом завершаются все базовые сведения, которые вы должны знать об алгоритме умножения матриц Штрассена. Возможно, вы сомневаетесь в том, что описанным здесь способом можно улучшить очевидный алгоритм обычного умножения матриц? Или же вам все же интересно, каким образом на самом деле выбираются способы произведения и сложения? Если это так, то следующий раздел для вас.

## 3.3.6. Детали

Обозначим через **X** и **Y** две входные матрицы размером $n \times n$ и определим **A**, **B**, ..., **H** как в (3.3). Приведем семь рекурсивных матричных умножений, выполняемых алгоритмом Штрассена:

$$\begin{aligned}
\mathbf{P}_1 &= \mathbf{A} \times (\mathbf{F} - \mathbf{H}) \\
\mathbf{P}_2 &= (\mathbf{A} + \mathbf{B}) \times \mathbf{H} \\
\mathbf{P}_3 &= (\mathbf{C} + \mathbf{D}) \times \mathbf{E} \\
\mathbf{P}_4 &= \mathbf{D} \times (\mathbf{G} - \mathbf{E}) \\
\mathbf{P}_5 &= (\mathbf{A} + \mathbf{D}) \times (\mathbf{E} + \mathbf{H}) \\
\mathbf{P}_6 &= (\mathbf{B} - \mathbf{D}) \times (\mathbf{G} + \mathbf{H}) \\
\mathbf{P}_7 &= (\mathbf{A} - \mathbf{C}) \times (\mathbf{E} + \mathbf{F}).
\end{aligned}$$

Потратив $\Theta(n^2)$ времени на выполнение необходимых матричных сложений и вычитаний, $\mathbf{P}_1$, ..., $\mathbf{P}_7$ могут быть вычислены с использованием семи рекурсивных вызовов на парах матриц размером $\frac{n}{2} \times \frac{n}{2}$. Но достаточно ли этой информации для получения матричного произведения **X** и **Y** за время $\Theta(n^2)$? Приведенное ниже удивительное уравнение дает утвердительный ответ:

$$\begin{aligned}
\mathbf{X} \times \mathbf{Y} &= \left( \begin{array}{c|c} \mathbf{A} \times \mathbf{E} + \mathbf{B} \times \mathbf{G} & \mathbf{A} \times \mathbf{F} + \mathbf{B} \times \mathbf{H} \\ \hline \mathbf{C} \times \mathbf{E} + \mathbf{D} \times \mathbf{G} & \mathbf{C} \times \mathbf{F} + \mathbf{D} \times \mathbf{H} \end{array} \right) \\
&= \left( \begin{array}{c|c} \mathbf{P}_5 + \mathbf{P}_4 - \mathbf{P}_2 + \mathbf{P}_6 & \mathbf{P}_1 + \mathbf{P}_2 \\ \hline \mathbf{P}_3 + \mathbf{P}_4 & \mathbf{P}_1 + \mathbf{P}_5 - \mathbf{P}_3 - \mathbf{P}_7 \end{array} \right)
\end{aligned}$$

Первое уравнение копируется из (3.4). Что касается второго уравнения, то нам нужно удостовериться, что равенство выполняется в каждом из четырех квадрантов. Чтобы опровергнуть свое неверие, обратите внимание на сумасшедшие сокращения в левом верхнем квадранте:

$$\begin{aligned}
\mathbf{P}_5 + \mathbf{P}_4 - \mathbf{P}_2 + \mathbf{P}_6 &= (\mathbf{A} + \mathbf{D}) \times (\mathbf{E} + \mathbf{H}) + \mathbf{D} \times (\mathbf{G} - \mathbf{E}) - \\
&\quad - (\mathbf{A} + \mathbf{B}) \times \mathbf{H} + (\mathbf{B} - \mathbf{D}) \times (\mathbf{G} + \mathbf{H}) = \\
&= \mathbf{A} \times \mathbf{E} + \mathbf{A} \times \mathbf{H} + \mathbf{D} \times \mathbf{E} + \mathbf{D} \times \mathbf{H} + \mathbf{D} \times \mathbf{G} - \\
&\quad - \mathbf{D} \times \mathbf{E} - \mathbf{A} \times \mathbf{H} - \mathbf{B} \times \mathbf{H} + \mathbf{B} \times \mathbf{G} + \\
&\quad + \mathbf{B} \times \mathbf{H} - \mathbf{D} \times \mathbf{G} - \mathbf{D} \times \mathbf{H} = \\
&= \mathbf{A} \times \mathbf{E} + \mathbf{B} \times \mathbf{G}.
\end{aligned}$$

Вычисление для правого нижнего квадранта аналогично, и равенство легко увидеть в двух других квадрантах. Таким образом, алгоритм `Strassen` действительно позволяет умножать матрицы с использованием всего семи рекурсивных вызовов, при этом время его исполнения $\Theta(n^2)$[1]!

### 3.3.7. Решение тестового задания 3.3

**Правильный ответ: (в)**. Правильный ответ — $\Theta(n^3)$. Здесь имеется три вложенных цикла for. Это приводит к $n^3$ итерациям внутреннего цикла (по одной для каждого варианта выбора $i, j, k \in \{1,2, ..., n\}$). В результате алгоритм выполняет постоянное число операций в каждой итерации (одно умножение и одно сложение). В качестве альтернативы для каждого из $n^2$ элементов в $\mathbf{Z}$ алгоритм тратит $\Theta(n)$ время на вычисление (3.1).

## *3.4. Алгоритм со временем O (*n* log *n*) для ближайшей пары

Наш последний пример алгоритма типа «разделяй и властвуй» — это очень крутой алгоритм для решения задачи о ближайшей паре, в которой дается $n$ точек на плоскости, и вам необходимо выявить пару точек, которые лежат ближе всего друг к другу. Это будет наш первый опыт в вычислительной геометрии, области, которая использует алгоритмы для объяснения свойств геометрических объектов и манипулирования ими. Соответствующие алгоритмы имеют приложения в робототехнике, компьютерном зрении и компьютерной графике[2].

---

[1] Начнем с того, что, безусловно, проверить успешность работы алгоритма намного проще, чем придумать его. И как вообще Фолькеру Штрассену удалось его придумать еще в 1969 году? Вот что он сказал (в личной беседе, июнь 2017 года): «Насколько я помню, я понял, что более быстрый некоммутативный алгоритм для некоторого небольшого случая даст лучший показатель. Я попытался доказать, что простой алгоритм оптимален для матриц 2 × 2. Чтобы упростить, я работал по модулю 2, а затем обнаружил более быстрый алгоритм комбинаторно».

[2] Помеченные звездочкой разделы, подобные этому, являются более сложными, и при первом чтении их можно пропустить.

## 3.4.1. Задача

Задача о ближайшей паре касается точек $(x, y) \in R^2$ на плоскости. Для измерения расстояния между двумя точками, $p_1 = (x_1, y_1)$ и $p_2 = (x_2, y_2)$, используется обычное евклидово (прямолинейное) расстояние:

$$d(p_1, p_1) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}. \quad (3.5)$$

**ЗАДАЧА: БЛИЖАЙШАЯ ПАРА**

**Вход**: $n \geq 2$ точек $p_1 = (x_i, y_i)$, ..., $p_n = (x_n, y_n)$ на плоскости.

**Выход**: пара точек, $p_i, p_j$, с наименьшим евклидовым расстоянием $d(p_i, p_j)$.

Для удобства мы допустим, что никакие две точки не имеют одинаковой $x$-координаты или $y$-координаты. Вы должны подумать о том, как доработать алгоритм из этого раздела для того, чтобы адаптировать его под такие совпадения[1].

Задача о ближайшей паре может быть решена за квадратичное время с помощью поиска по методу грубой силы — достаточно вычислить расстояние между каждой $\Theta(n^2)$ парой точек по порядку и определить кратчайшее из них. Что касается задачи подсчета инверсий (раздел 3.2), то нам уже удалось улучшить квадратичный алгоритм поиска по методу полного перебора при помощи алгоритма «разделяй и властвуй». Можно ли добиться лучшего и здесь?

## 3.4.2. Разминка: одномерный случай

Рассмотрим сначала более простой одномерный вариант задачи: дано $n$ точек $p_1$, ..., $p_n \in R^2$ в произвольном порядке. Требуется определить пару, которая

[1] В реальной реализации алгоритм ближайшей пары не будет утруждать себя вычислением квадратного корня в (3.5) — пара точек с наименьшим евклидовым расстоянием ничем не отличается от той, у которой наименьшее квадратичное евклидово расстояние, и последнее расстояние легче вычислить.

обладает минимальным расстоянием $|p_i - p_j|$. Этот частный случай легко решить за время $O(n \log n)$, используя инструменты, которые уже находятся в нашем арсенале. Основное наблюдение заключается в том, что, какой бы ни была ближайшая пара, две точки должны быть расположены последовательно в отсортированной версии множества точек (рис. 3.3).

**ОДНОМЕРНАЯ БЛИЖАЙШАЯ ПАРА**

отсортировать точки;

использовать линейный просмотр отсортированных точек, чтобы определить ближайшую пару.

Первый и второй шаги алгоритма могут быть реализованы соответственно со временем $O(n \log n)$ (при помощи алгоритма `MergeSort`) и со временем $O(n)$ (непосредственно), с общим временем $O(n \log n)$. Таким образом, в одномерном случае действительно существует алгоритм, который работает быстрее поиска по методу полного перебора.

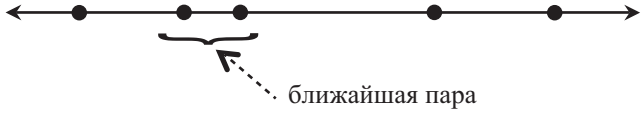



**Рис. 3.3.** В одномерном варианте точки в ближайшей паре появляются последовательно в отсортированной версии множества точек

## 3.4.3. Предварительная обработка

Может ли сортировка помочь решить двумерную версию задачи о ближайшей паре за время $O(n \log n)$? Главная проблема заключается в том, что имеются две разные координаты, которые можно использовать для сортировки точек. Но так как сортировка является легковесным примитивом (см. с. 31), почему бы просто не использовать ее (по обоим измерениям)? То есть на шаге предварительной обработки наш алгоритм создает две копии множества входных точек: копию $P_x$ с точками, отсортированными по $x$-координате, и копию $P_y$,

отсортированную по $y$-координате. Это занимает $O(n \log n)$ времени, которое находится внутри временной границы, на которую мы нацелены.

Каким образом можно использовать отсортированные версии $P_x$ и $P_y$? К сожалению, ближайшая пара точек не будет появляться последовательно ни в $P_x$, ни в $P_y$ (рис. 3.4). Мы должны сделать что-то более хитрое, чем простой линейный просмотр.

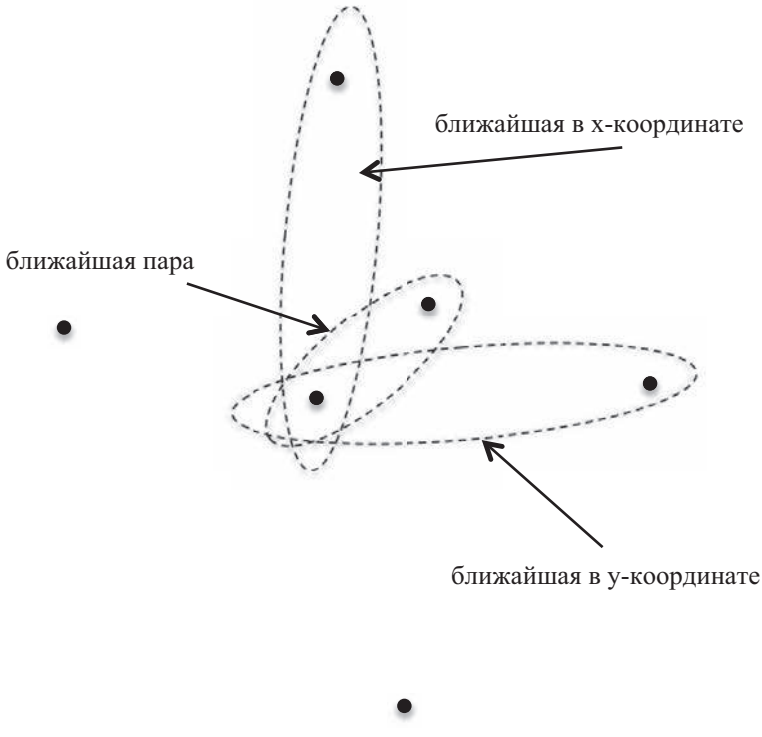



**Рис. 3.4.** В двух измерениях точки в ближайшей паре не обязательно будут появляться последовательно, когда точки отсортированы по координатам *x* или *y*

## 3.4.4. Подход «разделяй и властвуй»

Мы можем сделать лучше при помощи подхода «разделяй и властвуй»[1]. Как разделить входные данные на более мелкие подзадачи и как затем объединить решения этих подзадач в одну для исходной задачи? Что касается первого вопроса, мы используем первый отсортированный массив $P_x$, чтобы разделить входные данные на левую и правую половины. Назовем пару точек *левой парой*, если обе принадлежат левой половине множества точек, *правой парой*, если обе принадлежат правой половине, и *разделенной парой*, если точки принадлежат разным половинам. Например, в точке на рис. 3.4 ближайшая пара является разделенной парой, а пара точек, ближайших к $x$-координате, — левой парой.

Если ближайшая пара является левой парой или правой парой, то она будет идентифицирована одним из двух рекурсивных вызовов. Нам понадобится универсальная функция для оставшегося случая, когда ближайшая пара является разделенной парой. Эта функция играет аналогичную роль в подпрограмме `CountSplitInv` в разделе 3.2.

Приведенный ниже псевдокод резюмирует эти идеи; подпрограмма `ClosestSplitPair` для определения ближайшей разделенной пары на данный момент не реализована.

---

**CLOSESTPAIR (ПРЕДВАРИТЕЛЬНАЯ ВЕРСИЯ)**

**Вход**: две копии $P_x$ и $P_y$ из $n \geq 2$ точек на плоскости, отсортированных по координатам $x$ и $y$ соответственно.

**Выход**: пара $p_i, p_j$ разных точек с наименьшим евклидовым расстоянием между ними.

---

```
// базовый случай для <= 3 точек опущен
```

**1** $L_x$ := первая половина в $P_x$, отсортированная по $x$-координате

**2** $L_y$ := первая половина в $P_x$, отсортированная по $y$-координате

---

[1] Таким образом, парадигма «разделяй и властвуй» используется как на шаге предварительной обработки для реализации алгоритма `MergeSort`, так и еще раз в основном алгоритме.

**3** $R_x$ := вторая половина в $P_x$, отсортированная по $x$-координате

**4** $R_y$ := вторая половина в $P_x$, отсортированная по $y$-координате

**5** $(l_1, l_2)$ := `ClosestPair`($L_x$, $L_y$) `// лучшая левая пара`

**6** $(r_1, r_2)$ := `ClosestPair`($R_x$, $R_y$) `// лучшая правая пара`

**7** $(s_1, s_2)$ := `ClosestSplitPair`($P_x$, $P_y$) `// лучшая разделенная пара`

**8** вернуть лучшую из $(l_1, l_2)$, $(r_1, r_2)$, $(s_1, s_2)$

В опущенном базовом случае, когда имеется две или три входные точки, алгоритм вычисляет ближайшую пару непосредственно за постоянное время $O(1)$. Получить $L_x$ и $R_x$ из $P_x$ легко (нужно просто разделить $P_x$ пополам). Для вычисления $L_y$ и $R_y$ алгоритм может выполнять линейный просмотр $P_y$, помещая каждую точку в конец $L_y$ или $R_y$ в соответствии с $x$-координатой точки. Мы заключаем, что строки 1–4 могут быть реализованы со временем $O(n)$.

При условии правильной реализации подпрограммы `ClosestSplitPair` алгоритм гарантированно вычисляет ближайшую пару точек — три вызова подпрограммы в строках 5–7 охватывают все возможности нахождения ближайшей пары.

**ТЕСТОВОЕ ЗАДАНИЕ 3.4**

Допустим, что мы правильно реализуем подпрограмму `ClosestSplitPair` со временем $O(n)$. Каким будет общее время работы алгоритма `ClosestPair`? (Выберите наименьшую применимую верхнюю границу.)

а) $O(n)$.

б) $O(n \log n)$.

в) $O(n(\log n)^2)$.

г) $O(n^2)$.

(Решение и пояснение см. в разделе 3.4.10.)

### 3.4.5. Тонкая настройка

Решение тестового задания 3.4 делает нашу цель ясной: мы хотим, чтобы $O(n)$-реализация подпрограммы `ClosestSplitPair` привела к общей временной границе $O(n \log n)$ и совпала со временем работы нашего алгоритма для одномерного частного случая.

Мы создадим слегка упрощенную подпрограмму, соответствующую нашим целям. Ключевая особенность: *нам нужна подпрограмма* `ClosestSplitPair`*, чтобы определять ближайшую разделенную пару только тогда, когда она является ближайшей парой в целом*. Если ближайшая пара является левой или правой парой, то подпрограмма `ClosestSplitPair` вполне может выдать мусор — строка 8 псевдокода в разделе 3.4.4 проигнорирует предложенную ей пару точек в любом случае в пользу фактической ближайшей пары, вычисленной одним из рекурсивных вызовов. В нашем алгоритме этому смягченному требованию к правильности отводится решающая роль.

Чтобы реализовать данную идею, мы явным образом передадим подпрограмме `ClosestSplitPair` расстояние $\delta$ между ближайшей парой, которая является левой или правой парой; тогда подпрограмма будет знать, что ей придется искать решение только в разделенных парах с межточечным расстоянием менее $\delta$. Другими словами, мы заменяем строки 7–8 псевдокода в разделе 3.4.4 на следующие ниже.

**CLOSESTPAIR (ДОПОЛНЕНИЕ)**

**7** $\delta := \min\{d(l_1, l_2), d(r_1, r_2)\}$

**8** $(s_1, s_2) :=$ `ClosestSplitPair`$(P_x, P_y, \delta)$

**9** вернуть лучшую из $(l_1, l_2)$, $(r_1, r_2)$, $(s_1, s_2)$

## 3.4.6. Подпрограмма ClosestSplitPair

Теперь мы предоставим реализацию подпрограммы `ClosestSplitPair`, которая выполняется за линейное время и правильно вычисляет ближайшую пару всякий раз, когда та является разделенной парой. Вы можете не поверить, что приведенный ниже псевдокод удовлетворяет этим требованиям, но это так. Высокоуровневая идея состоит в том, чтобы выполнять поиск по методу грубой силы на предварительно ограниченном множестве пар точек.

**CLOSESTSPLITPAIR**

**Вход**: две копии, $P_x$ и $P_y$, из $n \geq 2$ точек на плоскости, отсортированных по координатам $x$ и $y$ и параметр $\delta$.

**Выход**: ближайшая пара при условии, что она — разделенная пара.

```
1 x̄ := наибольшая x-координата в левой половине // медиана
                                  // x-координаты
2 S_y := {точки q_1, q_2, ..., q_ℓ с x-координатой
    между x̄ − δ и x̄ + δ, отсортированные по y-координате}
3 best := δ
4 bestPair := NULL
5 for i := 1 to ℓ − 1 do
6     for j := 1 to min{7, ℓ − i} do
7         if d(q_i, q_i+j) < best then
8             best := d(q_i, q_i+j)
9             bestPair := (q_i, q_i+j)
10 return bestPair
```

Данная подпрограмма начинается в строке 1 с идентификации самой правой точки в левой половине множества точек, которая определяет «медиану $x$-координаты $\bar{x}$». Пара точек является разделенной парой тогда и только тогда, когда одна точка имеет координату $x$ не больше $\bar{x}$, а другая — больше $\bar{x}$. Вычисление $\bar{x}$ занимает постоянное ($O(1)$) время, поскольку $P_x$ содержит точки, отсортированные по координате $x$ (медиана равняется $n/2$-му элементу массива). В строке 2 подпрограмма выполняет шаг фильтрации, отбрасывая все точки, кроме тех, которые лежат на вертикальной полосе шириной $2\delta$ с центром в $\bar{x}$ (рис. 3.5). Множество $S_y$ может быть вычислено за линейное

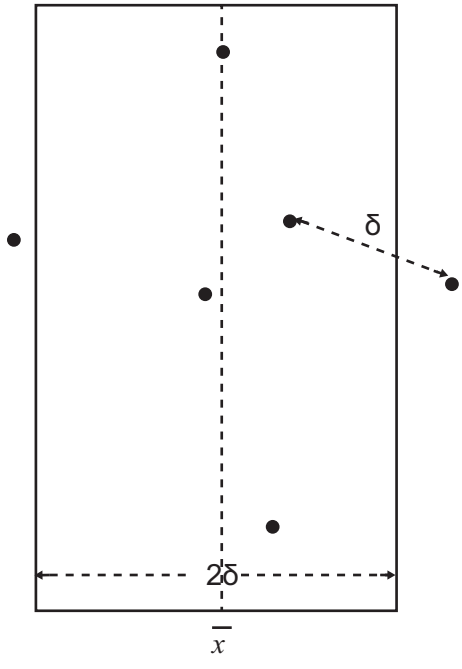



**Рис. 3.5.** Подпрограмма `ClosestSplitPair`. $S_y$ — это множество точек, заключенных в вертикальной полосе. δ — это наименьшее расстояние между левой парой или правой парой точек. Пары разделенных точек имеют одну точку по обе стороны от пунктирной линии

время путем просмотра $P_y$ и удаления любых точек с $x$-координатами вне интересующего диапазона[1]. Строки 5–9 выполняют поиск методом грубой силы в парах точек из множества $S_y$, у которого не более 6 точек в диапазоне (при упорядоченности $S_y$ по $y$-координатам), и вычисляют ближайшую такую пару точек[2]. Все это можно рассматривать как продолжение нашего алгоритма для одномерного случая, в котором мы рассматриваем все «почти последовательные» пары точек. Общее число итераций цикла составляет менее

[1] Этот шаг выполняется, потому что мы отсортировали точку, заданную $y$-координатой раз и навсегда на начальном шаге предварительной обработки. Поскольку мы нацелены на линейную подпрограмму, сейчас нет времени их сортировать!

[2] Если такой пары точек на расстоянии меньше δ нет, то подпрограмма возвращает NULL. В этом случае в `ClosestPair` эта NULL-пара игнорируется, и окончательное сравнение выполняется только между парами точек, возвращаемыми двумя рекурсивными вызовами.

$7\ell \leq 7n = O(n)$, и алгоритм выполняет постоянное число примитивных операций в каждой итерации. Мы заключаем, что подпрограмма `ClosestSplitPair` выполняется за время $O(n)$, как и требуется. Однако почему эта подпрограмма вообще должна найти ближайшую пару?

## 3.4.7. Правильность

Подпрограмма `ClosestSplitPair` выполняется за линейное время, потому что из квадратичного числа возможных пар точек она ищет только линейное их число. Откуда мы знаем, что она не пропустила ближайшую пару? Следующая ниже лемма, которая немного шокирует, гарантирует, что при условии, когда ближайшая пара является разделенной парой, ее точки появляются почти последовательно в отфильтрованном множестве $S_y$.

**Лемма 3.3.** *В подпрограмме* `ClosestSplitPair` *пусть* $(p, q)$ *есть разделенная пара с* $d(p, q) < \delta$, *где* $\delta$ — *это наименьшее расстояние между левой парой или правой парой точек. Тогда:*

(*a*) *p и q будут включены во множество* $S_y$;

(*b*) *не более шести точек в* $S_y$ *имеют y-координату между точками p и q.*

Эта лемма далеко не очевидна, и мы докажем ее в следующем разделе.

Из леммы 3.3 вытекает, что подпрограмма `ClosestSplitPair` корректно выполняет свою задачу.

**Следствие 3.4.** *Когда ближайшая пара является разделенной парой, подпрограмма* `ClosestSplitPair` *ее находит.*

*Доказательство*: допустим, что ближайшая пара $(p, q)$ является разделенной парой, и поэтому $d(p, q) < \delta$, где $\delta$ — минимальное расстояние между левой или правой парой. Лемма 3.3 гарантирует, что и $p$, и $q$ принадлежат множеству $S_y$ в подпрограмме `ClosestSplitPair` и что между ними есть не более шести точек $S_y$ по координате $y$. Поскольку подпрограмма `ClosestSplitPair` выполняет исчерпывающий поиск по всем парам точек, удовлетворяющим этим двум свойствам, она вычислит ближайшую такую пару, которая должна быть фактической ближайшей парой $(p, q)$. *Ч. т. д.*

В преддверии доказательства леммы 3.3 теперь у нас уже есть правильный и невероятно быстрый алгоритм для решения задачи о ближайшей паре.

**Теорема 3.5 (вычисление ближайшей пары).** Для каждого множества $P$ из $n \geq 2$ точек на плоскости алгоритм `ClosestPair` правильно вычисляет ближайшую пару $P$ и выполняется за время $O(n \log n)$.

*Доказательство*: мы с вами уже определили временную границу: алгоритм тратит $O(n \log n)$ времени на шаг предварительной обработки, и остальная часть алгоритма имеет такое же асимптотическое время, что и алгоритм `MergeSort` (с двумя рекурсивными вызовами, каждый с половиной входных данных плюс дополнительные линейные вычисления), который тоже выполняется за $O(n \log n)$.

Если ближайшая пара является левой парой, она выдается первым рекурсивным вызовом (строка 5 в разделе 3.4.4); если это правая пара, она выдается вторым рекурсивным вызовом (строка 6). Если это разделенная пара, то следствие 3.4 гарантирует, что она выдается подпрограммой `ClosestSplitPair`. Во всех случаях ближайшая пара находится среди трех кандидатов, рассмотренных алгоритмом (строка 9 в разделе 3.4.5), и будет выдана в качестве окончательного ответа. *Ч. т. д.*

## 3.4.8. Доказательство леммы 3.3 (а)

Часть (а) леммы 3.3 является более простой для доказательства. Допустим, что существует разделенная пара $(p, q)$ с $p$ в левой половине множества точек и $q$ в правой половине, такая, что $d(p, q) < \delta$, где $\delta$ — это минимальное расстояние между левой или правой парой. Обозначим $p = (x_1, y_1)$ и $q = (x_2, y_2)$ и обозначим через $\bar{x}$ координату $x$ самой правой точки левой половины. Поскольку $p$ и $q$ находятся в левой и правой половинах, соответственно, мы имеем

$$x_1 \leq \bar{x} < x_2.$$

В то же время $x_1$ и $x_2$ не могут быть сильно отдалены. Формально, используя определение евклидова расстояния (3.5), мы можем написать

$$\begin{aligned}\delta &> d(p, q)\\ &= \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}\\ &\geq \sqrt{\max\left\{(x_1 - x_2)^2, (y_1 - y_2)^2\right\}}\\ &= \max\left\{|x_1 - x_2|, |y_1 - y_2|\right\}.\end{aligned}$$

Это означает, что $p$ и $q$ отдалены менее чем на $\delta$ по обеим координатам $x$ и $y$:

$$|x_1 - x_2|, |y_1 - y_2| < \delta. \tag{3.6}$$

Поскольку $x_1 \leq \bar{x}$ и $x_2$, как максимум, на $\delta$ больше $x_1$, мы имеем $x_2 \leq \bar{x} + \delta$[1]. Поскольку $x_2 \geq \bar{x}$ и $x_1$, как максимум, на $\delta$ меньше $x_2$, $x_1 \geq \bar{x} - \delta$.

В частности, $p$ и $q$ имеют $x$-координаты, которые вклиниваются между $\bar{x} - \delta$ и $\bar{x} + \delta$. Все такие точки, в том числе $p$ и $q$, принадлежат множеству $S_y$.

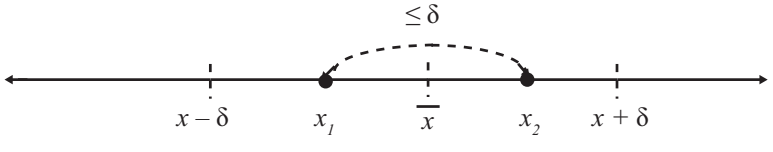



**Рис. 3.6.** Доказательство леммы 3.3(a). И $p$, и $q$ имеют $x$-координаты между $\bar{x} - \delta$ и $\bar{x} + \delta$

## 3.4.9. Доказательство леммы 3.3 (b)

Напомним установленные нами допущения: существует разделенная пара $(p, q)$, в которой $p = (x_1, y_1)$ находится в левой половине множества точек и $q = (x_2, y_2)$ — в правой половине, такая, что $d(p, q) < \delta$, где $\delta$ — это минимальное расстояние между левой или правой парой. Лемма 3.3(b) утверждает, что $p$ и $q$ не только появляются в множестве $S_y$ (что содержится в доказательстве

[1] Представьте себе, что $p$ и $q$ — это люди, связанные за талию веревкой длиной $\delta$. Точка $p$ может перемещаться вправо только на $\bar{x}$, тем самым ограничивая перемещения $q$ величиной $\bar{x} + \delta$ (рис. 3.6).

части (a)), но и что они почти последовательные, где шесть других точек $S_y$ обладают $y$-координатами между $y_1$ и $y_2$.

Для доказательства мы нарисуем восемь ячеек на плоскости по схеме 2 × 4, где каждая ячейка имеет длину стороны δ/2 (рис. 3.7). Два столбца ячеек расположены по обе стороны от медианы $x$-координаты $\bar{x}$. Нижняя часть ячеек выровнена с более нижней из точек $p$ и $q$ в $\min\{y_1, y_2\}$ $y$-координаты[1].

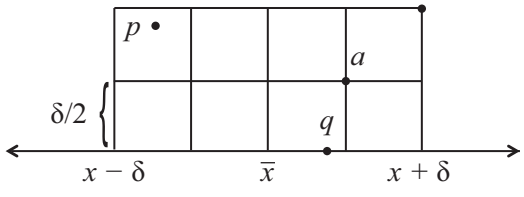



**Рис. 3.7.** Доказательство леммы 3.3 (b). Точки $p$ и $q$ населяют две из этих восьми ячеек, и в каждой ячейке есть не более одной точки

Из доказательства части (a) мы знаем, что и $p$, и $q$ имеют $x$-координаты между $\bar{x} - \delta$ и $\bar{x} + \delta$. Для конкретности предположим, что $q$ имеет меньшую $y$-координату; другой случай аналогичен. Таким образом, $q$ появляется в нижней части некоторой ячейки в нижней строке (в правой половине). Поскольку $y$-координата $p$ может быть только на δ больше $q$ (см. (3.6)), то $p$ также появляется в одной из ячеек (в левой половине). Каждая точка $S_y$ с $y$-координатой между $p$ и $q$ имеет $x$-координаты между $\bar{x} - \delta$ и $\bar{x} + \delta$ (требование для вхождения в множество $S_y$) и $y$-координату между $y_2$ и $y_1 < y_2 + \delta$ и, следовательно, лежит в одной из восьми ячеек. Остаются сомнения, что в этих ячейках есть много точек, которые имеют $y$-координату между $y_1$ и $y_2$. Чтобы показать, что этого не может произойти, давайте докажем, что каждая ячейка содержит не более одной точки. Тогда восемь ячеек содержат не более восьми точек (включая $p$ и $q$), и, следовательно, между $p$ и $q$ в $y$-координате может быть только шесть точек $S_y$[2].

---

[1] Не забывайте: эти ячейки предназначены исключительно для рассуждений о том, почему алгоритм `ClosestPair` является правильным. Сам алгоритм ничего не знает об этих ячейках и остается только псевдокодом в разделах 3.4.4–3.4.6.

[2] Если точка имеет $x$-координату точно в $\bar{x}$, отнесите ее к ячейке слева от координаты. Другие точки на границе нескольких ячеек можно отнести произвольно к одной из них.

Почему каждая ячейка содержит не более одной точки? Эта часть аргумента использует наше наблюдение в разделе 3.4.5 и тот факт, что δ — это наименьшее расстояние между левой парой или правой парой. Чтобы логически вывести противоречие, допустим, что некоторая ячейка имеет две точки, $a$ и $b$ (одна из которых может быть $p$ или $q$). Эта пара точек является либо левой парой (если точки находятся в первых двух столбцах), либо правой парой (если они находятся в последних двух). Наибольшее расстояние друг от друга, на котором могут быть $a$ и $b$, находится в противоположных углах ячейки (см. рис. 3.7), и в этом случае по теореме Пифагора[1] расстояние между $a$ и $b$ составляет $\sqrt{2} \times \frac{\delta}{2} < \delta$. Но это противоречит допущению, что не существует левой или правой пары на расстоянии меньше δ! Из этого противоречия следует, что каждая из восьми ячеек на рис. 3.7 имеет не более одной точки; следовательно, не более шести точек $S_y$ имеют $y$-координату между точками $p$ и $q$. *Ч. т. д.*

## 3.4.10. Решение тестового задания 3.4

**Правильный ответ: (б).** Правильный ответ — $O(n \log n)$. $O(n)$ не является правильным, поскольку, среди прочих причин, алгоритм `ClosestPair` уже тратит время $\Theta(n \log n)$ на шаге предварительной обработки, создавая отсортированные списки $p_x$ и $p_y$. Верхняя граница $O(n \log n)$ вытекает из точно такого же аргумента, что и для алгоритма `MergeSort`: алгоритм `ClosestPair` делает два рекурсивных вызова, каждый со входными данными в два раза меньше, и исполняется за время $O(n)$ вне своих рекурсивных вызовов. (Напомним, что строки 1–4 и 8 могут быть реализованы со временем $O(n)$, и для этого тестового задания мы допускаем, что подпрограмма `ClosestSplitPair` также исполняется за линейное время.) Эти выводы идеально соответствуют нашим заключениям для алгоритма `MergeSort`, который был проанализирован в разделе 1.5. Поэтому мы знаем, что общее число выполняемых операций равняется $O(n \log n)$. Поскольку шаг предварительной обработки тоже выполняется за время $O(n \log n)$, окончательная временная граница равняется $O(n \log n)$.

[1] Для прямоугольного треугольника сумма квадратов сторон равна квадрату гипотенузы.

**ВЫВОДЫ**

- Алгоритм типа «разделяй и властвуй» разбивает исходную задачу на более мелкие подзадачи, рекурсивно решает эти подзадачи и объединяет решения подзадач в решение исходной задачи.
- Вычисление количества инверсий в массиве может быть применимо для измерения подобия между двумя ранжированными списками. Алгоритм поиска методом полного перебора для этой задачи выполняется за время $\Theta(n^2)$ для массивов длиной $n$.
- Существует алгоритм «разделяй и властвуй», который опирается на алгоритм `MergeSort` и вычисляет количество инверсий за время $O(n \log n)$.
- Субкубический алгоритм Штрассена «разделяй и властвуй» для умножения матриц — это поразительный пример того, как алгоритмическая изобретательность может улучшать простые решения. Его ключевая идея состоит в том, чтобы экономить на рекурсивном вызове по сравнению с более простым алгоритмом «разделяй и властвуй», что аналогично подходу, реализованному в умножении Карацубы.
- В задаче о поиске ближайшей пары входными данными являются $n$ точек на плоскости, и ее цель состоит в том, чтобы вычислить пару точек с наименьшим евклидовым расстоянием между ними. Алгоритм поиска методом полного перебора выполняется за время $\Theta(n^2)$.
- Существует изощренный алгоритм типа «разделяй и властвуй», который решает задачу о ближайшей паре за время $O(n \log n)$.

## Задача на закрепление материала

**Задача 3.1.** Рассмотрим следующий ниже псевдокод для вычисления $a^b$, где $a$ и $b$ — это положительные целые числа[1]:

[1] $\lfloor x \rfloor$ обозначает функцию «floor», которая округляет свой аргумент вниз до ближайшего целого числа.

**FASTPOWER**

**Вход**: положительные целые числа $a$ и $b$.

**Выход**: $a^b$.

```
if b = 1 then
    return a
else
    c := b × b
    ans := FastPower(c, ⌊b / 2⌋)
if b is odd then
    return a × ans
else
    return ans
```

Допустим для этой задачи, что каждое умножение и деление может быть выполнено за постоянное время. Каково асимптотическое время исполнения этого алгоритма как функции $b$?

а) $\Theta(\log b)$.

б) $\Theta(\sqrt{b})$.

в) $\Theta(b)$.

г) $\Theta(b \log b)$.

## Задачи повышенной сложности

**Задача 3.2.** Дан *унимодальный* массив из $n$ различных элементов, это означает, что его элементы находятся в строго возрастающем порядке до своего максимального элемента, после чего его элементы находятся в строго убывающем порядке. Разработайте алгоритм вычисления максимального элемента унимодального массива, который выполняется за время $O(\log n)$.

**Задача 3.3.** Дан отсортированный (от наименьшего до наибольшего элемента) массив $A$ из $n$ разных целых чисел, которые могут быть положительными, отрицательными или нулевыми. Определите, существует ли индекс $i$ такой, что $A[i] = i$. Разработайте самый быстрый алгоритм для решения этой задачи.

**Задача 3.4.** (Сложная.) Дана решетка $n$ на $n$ разных чисел. Число является *локальным минимумом*, если оно меньше, чем все его соседи. (*Сосед* числа — это элемент, который находится выше, ниже, слева или справа. Большинство чисел имеют четырех соседей; числа, расположенные по сторонам решетки, имеют трех; четыре угловых числа имеют двух соседей.) Используйте парадигму разработки алгоритмов «разделяй и властвуй» для вычисления локального минимума, со сравнениями между парами чисел, со временем исполнения $O(n)$.

(Примечание: поскольку во входных данных имеется $n^2$ чисел, вы не можете позволить себе просматривать их все.)

[Подсказка: выясните, как выполнять рекурсию на решетке $\frac{n}{2}$ на $\frac{n}{2}$ за время исполнения $O(n)$.]

## Задачи по программированию

**Задача 3.5.** Реализуйте на своем любимом языке программирования алгоритм `CountInv` из раздела 3.2 для подсчета количества инверсий массива. (См. www.algorithmsilluminated.org для тестовых случаев и наборов исходных данных для задач.)

# 4 Основной метод

В этой главе представлен метод «черного ящика» для определения времени работы рекурсивных алгоритмов — подставляешь в формулу несколько ключевых характеристик алгоритма и получаешь верхнюю границу времени работы алгоритма. Этот «основной метод» применяется к большинству алгоритмов «разделяй и властвуй» из тех, с которыми вы когда-либо имели дело, включая алгоритм целочисленного умножения Карацубы (раздел 1.3) и алгоритм Штрассена умножения матриц (раздел 3.3)[1]. Эта глава также иллюстрирует более общую область в изучении алгоритмов: правильная оценка новых алгоритмических идей часто требует неочевидного математического анализа.

После знакомства с рекуррентными соотношениями в разделе 4.1 мы дадим формулировку основного метода (раздел 4.2) и рассмотрим шесть примеров приложений (раздел 4.3). В разделе 4.4 рассматривается доказательство основного метода с акцентом на сущности его трех знаменитых случаев. Правильное построение доказательства производится на основе нашего анализа алгоритма `MergeSort` в разделе 1.5.

## 4.1. К вопросу о целочисленном умножении

Чтобы обосновать основной метод, давайте вспомним основные аспекты нашего обсуждения целочисленного умножения (разделы 1.2–1.3). Задача состоит в том, чтобы умножить два $n$-значных числа, причем примитивными операциями являются сложение или умножение двух однозначных чисел. Итеративный школьный алгоритм требует $\Theta(n^2)$ операций, чтобы умножить два $n$-значных числа. Можем ли мы сделать это проще при помощи метода «разделяй и властвуй»?

### 4.1.1. Алгоритм RecIntMult

Алгоритм `RecIntMult` из раздела 1.3 создает более мелкие подзадачи, разбивая заданные $n$-значные числа $x$ и $y$ на их первую и вторую половины: $x = 10^{n/2} \times$

[1] Основной метод также называется «основной теоремой».

$\times a + b$ и $y = 10^{n/2} \times c + d$, где $a$, $b$, $c$, $d$ — это $n/2$-значные числа (для простоты зададим, что $n$ — четное). Например, если $x = 1234$, то $a = 12$ и $b = 34$. Тогда

$$x \times y = 10^n \times (a \times c) + 10^{n/2} \times (a \times d + b \times c) + b \times d, \tag{4.1}$$

которое показывает, что умножение двух $n$-значных чисел сводится к умножению четырех пар $n/2$-значных чисел, плюс $O(n)$ дополнительная работа (для добавления соответствующих нулей и школьного сложения).

Формально это можно описать как *рекуррентное соотношение*. Обозначим через $T(n)$ максимальное число операций, используемых этим рекурсивным алгоритмом умножения двух $n$-значных чисел, — это количество мы хотим ограничить сверху. Рекуррентное соотношение выражает временную границу $T(n)$ в рамках количества операций, выполняемых рекурсивными вызовами. Рекуррентное соотношение для алгоритма `RecIntMult` равняется

$$T(n) \leq \underbrace{4 \times T\left(\frac{n}{2}\right)}_{\text{работа рекурсивных вызовов}} + \underbrace{O(n)}_{\text{работа вне рекурсивных вызовов}}$$

Как и для рекурсивного алгоритма, для рекуррентного соотношения также нужен базовый случай, в котором устанавливается, какой является $T(n)$ для значений $n$, которые слишком малы для запуска рекурсивных вызовов. Здесь базовым является случай, при котором $n = 1$ и алгоритм просто выполняет одно умножение, поэтому $T(1) = 1$.

## 4.1.2. Алгоритм Karatsuba

Рекурсивный алгоритм Карацубы для целочисленного умножения использует метод Гаусса, чтобы сэкономить на одном рекурсивном вызове. Суть метода заключается в том, чтобы рекурсивно вычислять произведения $a$ и $c$, $b$ и $d$, и $a + b$ и $c + d$ и извлекать средний коэффициент $a \times d + b \times c$ посредством $(a + b)(c + d) - ac - bd$. Этого достаточно для вычисления правой части (4.1) с $O(n)$ посредством дополнительных примитивных операций.

**ТЕСТОВОЕ ЗАДАНИЕ 4.1**

Какое рекуррентное соотношение лучше всего описывает время работы алгоритма `Karatsuba` для целочисленного умножения?

а) $T(n) \leq 2 \times T\left(\frac{n}{2}\right) + O(n^2)$.

б) $3 \times T\left(\frac{n}{2}\right) + O(n)$.

в) $3 \times T\left(\frac{n}{2}\right) + O(n^2)$.

г) $4 \times T\left(\frac{n}{2}\right) + O(n)$.

(Решение и пояснение см. ниже.)

**Правильный ответ: (б)**. Единственное изменение по сравнению с алгоритмом `RecIntMult` состоит в том, что число рекурсивных вызовов пришлось на единицу. Справедливо отметить, что объем работы, выполняемой вне рекурсивных вызовов, больше в алгоритме `Karatsuba`, но только на постоянный множитель, который устраняется в обозначении *O*-большое. Соответствующее рекуррентное соотношение для алгоритма `Karatsuba`, следовательно,

$$T(n) \leq \underbrace{3 \times T\left(\frac{n}{2}\right)}_{\text{работа рекурсивных вызовов}} + \underbrace{O(n)}_{\text{работа вне рекурсивных вызовов}}$$

снова с базовым случаем $T(1) = 1$[1].

## 4.1.3. Сравнение рекуррентных соотношений

На данный момент мы не знаем время работы алгоритма `RecIntMult` или `Karatsuba`, но инспектирование их рекуррентных соотношений наводит на мысль о том, что последний может быть только быстрее, чем первый. Еще

[1] Технически рекурсивный вызов с $a + b$ и $c + d$ может содержать $(\frac{n}{2} + 1)$-значные числа. Давайте договоримся между собой, что мы этот факт проигнорируем — в конечном итоге это не имеет значения.

одним примером для сравнения является алгоритм `MergeSort`, в рамках которого наш анализ в разделе 1.5 приводит к рекуррентному соотношению,

$$T(n) \leq \underbrace{2 \times T\left(\frac{n}{2}\right)}_{\text{работа рекурсивных вызовов}} + \underbrace{O(n)}_{\text{работа вне рекурсивных вызовов}},$$

где $n$ — длина сортируемого массива. Это говорит о том, что временные границы для алгоритмов `RecIntMult` и `Karatsuba` не могут быть больше, чем граница для `MergeSort`, равная $O(n \log n)$. За исключением этих подсказок, мы действительно не имеем понятия, каково время работы любого из этих алгоритмов. Озарение придет вместе с основным методом, который будет рассмотрен ниже.

## 4.2. Формулировка

Основной метод — это именно то, что вам нужно для анализа рекурсивных алгоритмов. На входе он принимает рекуррентное соотношение для алгоритма и — ба-бах! — на выходе пересекает верхнюю границу времени работы алгоритма.

### 4.2.1 Стандартные рекуррентные соотношения

Мы обсудим версию основного метода, работающую с тем, что мы будем называть «стандартными рекуррентными соотношениями», которые имеют три свободных параметра и форму, указанную ниже[1].

[1] Это представление основного метода черпает вдохновение из главы 2 книги «Алгоритмы» Санджоя Дасгупты, Христоса Пападимитриу, Умеша Вазирани (Algorithms, Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani; McGraw-Hill, 2006).

**ФОРМАТ СТАНДАРТНОГО РЕКУРРЕНТНОГО СООТНОШЕНИЯ**

**Базовый случай:** $T(n)$ является не более чем константой для всех достаточно малых $n$[1].

**Общий случай:** для больших значений $n$

$$T(n) \leq a \times T\left(\frac{n}{b}\right) + O\left(n^d\right).$$

**Параметры:**

- $a$ = количество рекурсивных вызовов;
- $b$ = коэффициент сжатия размера входных данных;
- $d$ = экспонента во времени работы «шага объединения».

Базовый случай стандартного рекуррентного отношения утверждает, что, когда размер входных данных настолько мал, что рекурсивные вызовы не нужны, задача может быть решена за время $O(1)$. Это касается всех рассматриваемых нами приложений. Общий случай принимает допущение, что алгоритм делает $a$ рекурсивных вызовов, каждый с подзадачей, имеющей размер на коэффициент $b$ меньше, чем его входные данные, и выполняет $O(n^d)$ работу вне этих рекурсивных вызовов. Например, в алгоритме `MergeSort` есть два рекурсивных вызова ($a = 2$), каждый из которых с массивом, имеющим размер в половину размера входных данных ($b = 2$), и $O(n)$ работа выполняется вне рекурсивных вызовов ($d = 1$). В целом $a$ может быть любым положительным целым числом, $b$ может быть любым действительным числом больше 1 (если $b \leq 1$, то алгоритм не завершится) и $d$ может быть любым неотрицательным вещественным числом, где $d = 0$ указывает только на постоянную ($O(1)$) работу вне рекурсивных вызовов. Как обычно, мы игнорируем тот факт, что, возможно, $\frac{n}{b}$ потребуется округлить вверх или вниз до целого числа, — и, как обычно, это не влияет на наши окончательные выводы. Не будем забывать, что $a$, $b$ и $d$ должны быть

[1] Формально существуют положительные целые числа $n_0$ и $c$, независимые от $n$, такие, что $T(n) \leq c$ для всех $n \leq n_0$.

*константами* — числами, которые не зависят от размера $n$ входных данных[1]. Типичными значениями этих параметров являются 1 (для $a$ и $d$), 2, 3 и 4. Если вы когда-нибудь обнаружите, что говорите что-то вроде «применить основной метод с $a = n$ или $b = \frac{n}{n-1}$», то вы используете его неправильно.

Одно ограничение в стандартных рекуррентных соотношениях состоит в том, чтобы каждый рекурсивный вызов делался с подзадачей одинакового размера. Например, алгоритм, который делает один рекурсивный вызов с первой третью входного массива и еще один с оставшейся частью, приведет к нестандартному рекуррентному соотношению. Большинство (но не все) естественных алгоритмов «разделяй и властвуй» приводят к стандартным рекуррентным соотношениям. Например, в алгоритме `MergeSort` оба рекурсивных вызова работают с задачами, размер которых вдвое меньше размера входного массива. В наших алгоритмах рекурсивного целочисленного умножения рекурсивным вызовам всегда передаются числа, которые вдвое меньше[2].

## 4.2.2. Формулировка и обсуждение основного метода

Теперь мы можем сформулировать основной метод, который представляет верхнюю границу стандартного рекуррентного соотношения как функцию ключевых параметров $a$, $b$ и $d$.

**Теорема 4.1 (основной метод).** *Если $T(n)$ определяется стандартным рекуррентным соотношением с параметрами $a > 1$, $b > 1$ и $d > 0$, то*

$$T(n) = \begin{cases} O\left(n^d \log n\right) & \text{если } a = b^d \quad |\text{Случай 1}| \\ O\left(n^d\right) & \text{если } a < b^d \quad |\text{Случай 2}| \\ O\left(n^{\log_b a}\right) & \text{если } a < b^d \quad |\text{Случай 3}|. \end{cases} \tag{4.2}$$

---

[1] В базовом случае и в члене «$O(n^d)$» есть константы, которые также были устранены, но вывод основного метода не зависит от их значений.

[2] Существуют более общие версии основного метода, которые подходят для более широкого семейства рекуррентных соотношений, но представленной здесь простой версии достаточно для почти любого алгоритма «разделяй и властвуй», с которым вы, вероятно, столкнетесь.

Что это за три случая и почему относительные значения $a$ и $b^d$ так важны? Может ли во втором случае время работы всего алгоритма действительно составлять только $O(n^d)$, в то время как внешний рекурсивный вызов уже делает работу за время $O(n^d)$? И что происходит с экзотическим временем работы в третьем случае? К концу этой главы мы узнаем удовлетворяющие нас ответы на все эти вопросы, и формулировка основного метода будет казаться самой естественной вещью в мире.[1]

### ЕЩЕ РАЗ О ЛОГАРИФМАХ

Еще один озадачивающий аспект теоремы 4.1 касается непоследовательного использования логарифмов. В третьем случае с предосторожностью делается заявление, что рассматриваемый логарифм имеет основание $b$, то есть сколько раз можно делить $n$ на $b$, пока результатом не будет число не более 1. При этом в первом случае основание логарифма вообще не указывается. Причина в том, что *любые две логарифмические функции различаются только постоянным множителем (коэффициентом)*. Например, логарифм по основанию 2 всегда превышает натуральный логарифм (то есть логарифм по основанию $e$, где $e = 2{,}718\ldots$) в $1/\ln 2 \approx 1{,}44$ раза. В первом случае основного метода изменение основания логарифма изменяет только постоянный множитель, который легко опускается в обозначении $O$-большое. В третьем случае логарифм появляется в экспоненте, где разные постоянные множители преобразуются в очень разные временные рамки работы (например, $n^2$ против $n^{100}$).

[1] Границы в теореме 4.1 имеют форму $O(f(n))$, а не $\Theta(f(n))$, потому что в нашем рекуррентном соотношении мы допускаем только верхнюю границу $T(n)$. Если в определении стандартного рекуррентного соотношения заменить $\leq$ на $=$, а $O(n^d)$ на $\Theta(n^d)$, то границы в теореме 4.1 соблюдаются с заменой $O(\cdot)$ на $\Theta(\cdot)$. Верификация этого утверждения станет хорошей проверкой на понимание доказательства в разделе 4.4.

# 4.3. Шесть примеров

Основной метод (теорема 4.1) очень трудно усвоить в первый раз, когда вы его видите. Давайте его конкретизируем шестью разными примерами.

## 4.3.1. К вопросу об алгоритме MergeSort

В качестве проверки работоспособности давайте вернемся к алгоритму, время работы которого мы уже знаем, — к `MergeSort`. Чтобы применить основной метод, от нас требуется только определить значения трех свободных параметров: $a$, количество рекурсивных вызовов; $b$, коэффициент, на который размер входных данных сжимается перед рекурсивными вызовами, и $d$, показатель степени в границе объема работы, выполняемой вне рекурсивных вызовов[1]. В алгоритме `MergeSort` имеется два рекурсивных вызова, поэтому $a = 2$. Каждый рекурсивный вызов получает половину входного массива, поэтому $b = 2$ тоже. Работа, выполняемая вне этих рекурсивных вызовов, определяется подпрограммой `Merge`, которая выполняется в линейное время (раздел 1.5.1), и поэтому $d = 1$. Следовательно,

$$a = 2 = 2^1 = b^d,$$

что возвращает нас к первому случаю основного метода. При подставлении параметров из теоремы 4.1 следует, что время работы алгоритма `MergeSort` $O(n^d \log n) = O(n \log n)$. Таким образом, поторяется наш анализ из раздела 1.5.

[1] У всех рекуррентных соотношений, которые мы рассматриваем, есть базовый случай в форме, необходимой для стандартных рекуррентных соотношений, и с этого момента мы не будем их обсуждать.

## 4.3.2. Двоичный поиск

В качестве второго примера мы рассмотрим задачу поиска заданного элемента в отсортированном массиве. Подумайте, например, о поиске собственного имени в алфавитном списке большой книги[1]. Вы можете искать линейно, начиная с самого начала, но тогда преимущество списка в алфавитном порядке будет утрачено. Более разумный подход заключается в том, чтобы открыть середину книги и рекурсивно искать либо в первой половине (если имя в середине идет после вашего собственного), либо во второй половине (в противном случае). Этот алгоритм, транслированный в задачу поиска в отсортированном массиве, называется двоичным поиском[2].

Каково время работы двоичного поиска? На этот вопрос легко ответить напрямую, но давайте посмотрим, как с ним обращается основной метод.

**ТЕСТОВОЕ ЗАДАНИЕ 4.2**

Каковы соответствующие значения $a$, $b$ и $d$ для алгоритма двоичного поиска?

а) 1, 2, 0 [случай 1].

б) 1, 2, 1 [случай 2].

в) 2, 2, 0 [случай 3].

г) 2, 2, 1 [случай 1].

(Решение и пояснение см. в разделе 4.3.7.)

[1] Читателям средних лет это напомнит телефонную книгу.

[2] Если вы еще не ознакомились с кодом этого алгоритма, посмотрите его в своей любимой книге по основам программирования или в учебном пособии.

### 4.3.3. Рекурсивное целочисленное умножение

Теперь мы перейдем к хорошим вещам, алгоритмам «разделяй и властвуй», для которых мы еще не знаем границу времени работы. Начнем с алгоритма `RecIntMult` для целочисленного умножения. В разделе 4.1 мы видели, что соответствующее рекуррентное соотношение для этого алгоритма выражается формулой

$$T(n) \leq 4 \times T\left(\frac{n}{2}\right) + O(n),$$

и поэтому $a = 4$, $b = 2$ и $d = 1$. Следовательно,

$$a = 4 > 2 = 2^1 = b^d,$$

что возвращает нас к третьему случаю основного метода. В этом случае мы получаем причудливо выглядящую границу времени работы $O(n^{\log_b a})$. Для наших значений параметров $O(n^{\log_2 3}) = O(n^2)$. Следовательно, алгоритм `RecIntMult` совпадает, но не превосходит итеративный школьный алгоритм целочисленного умножения (который использует $\Theta(n^2)$ операций).

### 4.3.4. Умножение Карацубы

Подход «разделяй и властвуй» к целочисленному умножению окупается, только когда используется метод Гаусса, который экономит на рекурсивном вызове. Как мы видели в разделе 4.1, время работы алгоритма `Karatsuba` определяется рекуррентным соотношением

$$T(n) \leq 3 \times T\left(\frac{n}{2}\right) + O(n),$$

которое отличается от предыдущего рекуррентного соотношения только тем, что $a$ снизился с 4 до 3 ($b$ по-прежнему равен 2, $d$ по-прежнему равен 1). Мы ожидаем, что время работы будет где-то между $O(n \log n)$ (граница при $a = 2$, как в алгоритме `MergeSort`) и $O(n^2)$ (граница при $a = 4$, как в алгоритме `RecIntMult`). Если неизвестность вас убивает, основной метод предлагает быстрое решение: у нас есть

$$a = 3 > 2 = 2^1 = b^d,$$

и поэтому мы по-прежнему используем третий случай основного метода, но с увеличенными временными рамками работы: $O(n^{\log_b a}) = O(n^{\log_2 3}) = O(n^{1,59})$.

Таким образом, экономия на рекурсивном вызове приводит к значительно большему времени работы, и алгоритм целочисленного умножения, который вы изучили в третьем классе, не является самым быстрым из возможных[1]!

### 4.3.5. Умножение матриц

В разделе 3.3 рассматривалась задача умножения двух $n \times n$-матриц. Как и в случае с целочисленным умножением, мы обсудили три алгоритма: простой итеративный алгоритм, простой рекурсивный алгоритм `RecMatMult` и гениальный алгоритм `Strassen`. Итеративный алгоритм использует $\Theta(n^3)$ операций (тестовое задание 3.3). Алгоритм `RecMatMult` разбивает каждую из двух входных матриц на четыре $\frac{n}{2} \times \frac{n}{2}$ матрицы (по одной для каждого квадранта), выполняет соответствующие восемь рекурсивных вызовов с более мелкими матрицами и соответствующим образом объединяет результаты (используя простое сложение матриц). Алгоритм `Strassen` ловко идентифицирует семь пар матриц $\frac{n}{2} \times \frac{n}{2}$, произведений которых достаточно для реконструирования произведения первоначальных входных матриц.

**ТЕСТОВОЕ ЗАДАНИЕ 4.3**

Какие временные рамки работы предусматривает основной метод для алгоритмов `RecIntMult` и `Strassen` соответственно?

а) $O(n^3)$ и $O(n^2)$.

б) $O(n^3)$ и $O(n^{\log_2 7})$.

в) $O(n^3)$ и $O(n^3)$.

г) $O(n^3 \log n)$ и $O(n^3)$.

(Решение и пояснение см. в разделе 4.3.7.)

[1] Интересный факт: в языке программирования Python встроенная подпрограмма умножения целочисленных объектов использует алгоритм начальной школы для целых чисел с не более чем 70 знаками и алгоритм `Karatsuba` в противном случае.

### 4.3.6. Фиктивное рекуррентное соотношение

В наших пяти предыдущих примерах два рекуррентных соотношения относятся к первому случаю основного метода, а остальные — к третьему случаю. Кроме того, существуют естественные рекуррентные соотношения, которые относятся ко второму случаю. Например, предположим, что у нас есть алгоритм «разделяй и властвуй», который работает как `MergeSort`, за исключением того, что этот алгоритм работает сложнее вне рекурсивных вызовов, выполняя квадратичный, а не линейный объем работы. То есть рассмотрим рекуррентное соотношение

$$T(n) \leq 2 \times T\left(\frac{n}{2}\right) + O\left(n^2\right).$$

Здесь мы видим

$$a = 2 < 4 = 22 = b^d,$$

что возвращает нас прямо ко второму случаю основного метода для временной границы $O(n^d) = O(n^2)$. Это может показаться нелогичным. С учетом того, что алгоритм `MergeSort` выполняет линейную работу вне двух рекурсивных вызовов и его время работы $O(n \log n)$, можно было ожидать, что квадратичный шаг объединения приведет к времени работы $O(n^2 \log n)$. Основной метод показывает, что это завышенная оценка, и обеспечивает самую высокую верхнюю границу $O(n^2)$. Примечательно, что это означает, что общее время работы алгоритма определяется работой, выполняемой в самом внешнем вызове — все последующие рекурсивные вызовы только увеличивают общее число операций, выполняемых с постоянным множителем[1].

[1] Мы увидим еще один пример случая 2 основного метода, когда будем обсуждать линейный выбор в главе 6.

## 4.3.7. Решения тестовых заданий 4.2–4.3

### Решение тестового задания 4.2

**Правильный ответ: (а).** Двоичный поиск выполняет рекурсию либо с левой половиной входного массива, либо с правой половиной (и никогда с обеими), поэтому существует только один рекурсивный вызов ($a = 1$). Этот рекурсивный вызов выполняется с половиной входного массива, поэтому $b$ снова равняется 2. Вне рекурсивного вызова двоичный поиск выполняет лишь одно единственное сравнение (между срединным элементом массива и элементом, который ищется), чтобы определить, следует ли рекурсивно выполнять поиск в левой или правой половине массива. Это транслируется в $O(1)$-работу вне рекурсивного вызова, поэтому $d = 0$. Поскольку $a = 1 = 2^0 = b^d$, мы снова возвращаемся к первому случаю основного метода и получаем временную границу $O(n^d \log n) = O(\log n)$.

### Решение тестового задания 4.3

**Правильный ответ: (б).** Начнем с алгоритма `RecMatMult` (раздел 3.3.4). Обозначим через $T(n)$ максимальное число примитивных операций, которые алгоритм использует для умножения двух $n \times n$-матриц. Число рекурсивных вызовов равняется 8. Каждый из этих вызовов выполняется с парой матриц $\frac{n}{2} \times \frac{n}{2}$, поэтому $b = 2$. Работа, выполняемая вне рекурсивных вызовов, включает постоянное число сложений матриц, и им нужно время $O(n^2)$ (постоянное время для каждого из $n^2$ элементов матриц). Таким образом, рекуррентное соотношение равняется

$$T(n) \leq 8 \times T\left(\frac{n}{2}\right) + O\left(n^2\right),$$

и поскольку

$$a = 8 > 4 = 2^2 = b^d,$$

мы возвращаемся к третьему случаю основного метода, который определяет временную границу

$$O(n^{\log_b a}) = O(n^{\log_2 8}) = O(n^3).$$

Единственное различие между рекуррентным соотношением для алгоритма `Strassen` и рекуррентным соотношением, приведенным выше, состоит в том, что количество рекурсивных вызовов $a$ падает с 8 до 7. Несомненно, что по алгоритму `Strassen` получается больше матричных сложений, чем по алгоритму `RecMatMult`, но только с постоянным множителем, и, следовательно, $d$ по-прежнему равен 2. Следовательно,

$$a = 7 > 4 = 2^2 = b^d.$$

Мы по-прежнему используем третий случай основного метода, но с увеличенными временными рамками работы:

$$O(n^{\log_b a}) = O(n^{\log_2 7}) = O(n^{2,81}).$$

Следовательно, алгоритм `Strassen` действительно асимптотически превосходит простой итеративный алгоритм[1]!

## *4.4. Доказательство основного метода

В этом разделе доказывается основной метод (теорема 4.1): если $T(n)$ управляется стандартным рекуррентным соотношением вида

$$T(n) \le a \times T\left(\frac{n}{b}\right) + O\left(n^d\right),$$

тогда

$$T(n) = \begin{cases} O\left(n^d \log n\right) & \text{если } a = b^d \quad |\text{Случай 1}| \\ O\left(n^d\right) & \text{если } a < b^d \quad |\text{Случай 2}| \\ O\left(n^{\log_b a}\right) & \text{если } a > b^d \quad |\text{Случай 3}|. \end{cases}$$

[1] Существует целый ряд исследовательских работ, в которых разрабатываются все более сложные алгоритмы умножения матриц со все улучшающимися наихудшими асимптотическими периодами работы (хотя и с большими постоянными множителями, которые исключают практические реализации). Текущий мировой рекорд — это временные рамки работы примерно $O(n^{2,3729})$, и, насколько нам известно, ожидается обнаружение алгоритма со временем $O(n^2)$.

Важно помнить значения трех свободных параметров:

| Параметр | Значение |
|---|---|
| *a* | число рекурсивных вызовов |
| *b* | коэффициент, с которым размер входных данных сжимается в рекурсивном вызове |
| *d* | экспонента работы, выполняемой вне рекурсивных вызовов |

### 4.4.1. Преамбула

Доказательство основного метода важно не потому, что мы заботимся о формальности ради нее самой, а потому, что оно дает фундаментальное объяснение тому, почему все так, как есть, — например, почему в основном методе три случая. Имея это в виду, вы должны различать два типа содержимого в доказательстве. В некоторых моментах мы прибегнем к алгебраическим вычислениям, чтобы понять, что происходит. Эти расчеты стоит увидеть один раз в жизни, но их не особенно важно помнить в долгосрочной перспективе. Стоит помнить концептуальный смысл трех случаев основного метода. Доказательство будет использовать подход дерева рекурсии, который так хорошо послужил нам для анализа алгоритма `MergeSort` (раздел 1.5), и три случая соответствуют трем разным типам деревьев рекурсии. Если вы помните значение трех случаев, нет необходимости запоминать время работы в основном методе — вы сможете реконструировать их по мере необходимости, исходя из вашего концептуального понимания метода.

В целях формального доказательства мы должны явно выписать все постоянные множители в рекуррентном соотношении:

**Базовый случай**: $T(1) \leq c$.

**Общий случай**: для $n > 1$

$$T(n) \leq a \times T\left(\frac{n}{b}\right) + cn^d. \tag{4.3}$$

Для простоты мы принимаем допущение, что константа $n_0$, указывающая, когда начинается базовый случай, равняется 1; доказательство для отличающейся константы $n_0$ почти одинаково. Мы можем допустить, что опущенные

константы в базовом случае и член $O(n^d)$ в общем случае равны одному и тому же числу $c$; если бы они были разными константами, мы могли бы просто работать с бóльшей из двух. Наконец, давайте сосредоточимся на случае, когда $n$ является степенью $b$. Доказательство для общего случая аналогично, без дополнительного концептуального содержания, но более кропотливое.

### 4.4.2. К вопросу о деревьях рекурсии

Высокоуровневый план доказательства является настолько естественным, насколько это возможно: обобщить аргумент дерева рекурсии для алгоритма `MergeSort` (раздел 1.5), чтобы он вмещал другие значения ключевых параметров $a$, $b$ и $d$. Напомним, что дерево рекурсии обеспечивает принципиальный способ отслеживания всей работы, выполняемой рекурсивным алгоритмом, во всех его рекурсивных вызовах. Узлы дерева соответствуют рекурсивным вызовам, а дочерние элементы узла соответствуют рекурсивным вызовам, выполняемым этим узлом (рис. 4.1). Таким образом, корень (уровень 0)

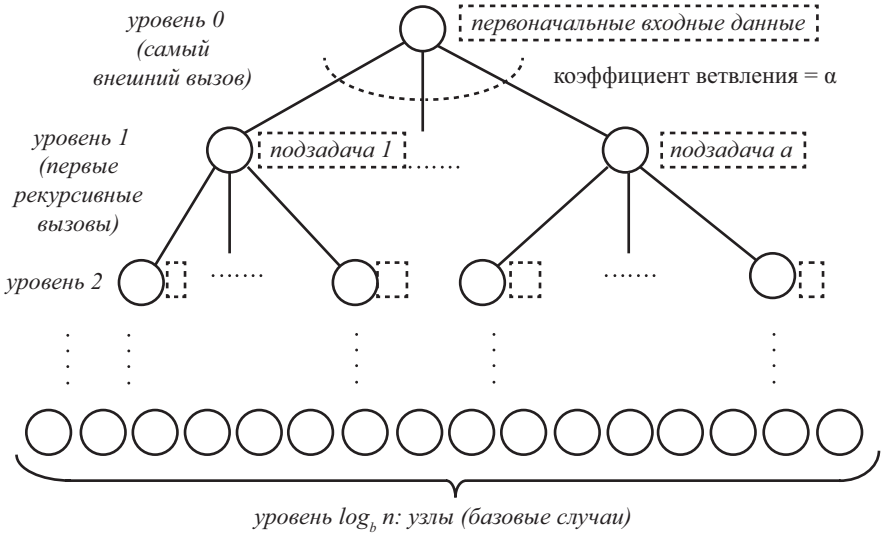



**Рис. 4.1.** Дерево рекурсии, соответствующее стандартному рекуррентному соотношению. Узлы соответствуют рекурсивным вызовам. Уровень 0 соответствует наиболее удаленному вызову, уровень 1 — своим рекурсивным вызовам, и так далее

дерева рекурсии соответствует самому внешнему вызову алгоритма, уровень 1 имеет узлы, соответствующие его рекурсивным вызовам, и так далее. Листья в нижней части дерева соответствуют рекурсивным вызовам, при которых срабатывает базовый случай.

Как и в нашем анализе `MergeSort`, мы хотели бы вести поуровневый учет работы, выполняемой по рекурсивному алгоритму. Этот план требует понимания двух вещей: количества разных подзадач на заданном уровне рекурсии $j$ и длины входных данных для каждой из этих подзадач.

**ТЕСТОВОЕ ЗАДАНИЕ 4.4**

В чем шаблон? Заполните пропуски в следующем высказывании: на каждом уровне $j = 0, 1, 2, \ldots$ дерева рекурсии имеется [пропуск] подзадач, каждая из которых оперирует с подмассивом длиной [пропуск],

а) $a^j$ и $n/a^j$ соответственно.

б) $a^j$ и $n/b^j$ соответственно.

в) $b^j$ и $n/a^j$ соответственно.

г) $b^j$ и $n/b^j$ соответственно.

(Решение и пояснение см. в разделе 4.4.10.)

## 4.4.3. Работа, выполняемая на одном уровне

Вдохновленный нашим анализом `MergeSort`, план состоит в том, чтобы подсчитать общее число операций, выполняемых подзадачами уровня $j$ в алгоритме «разделяй и властвуй», а затем суммировать по всем уровням. Поэтому возьмем уровень $j$ рекурсии крупным планом. Согласно решению тестового задания 4.4 на уровне $j$ имеется $a^j$ разных подзадач, каждая из которых имеет входные данные размером $n/b^j$. Нас интересует размер подзадачи настолько, насколько она определяет объем работы, выполняемой рекурсивным вызовом. Наше рекуррентное соотношение (4.3) утверждает,

что работа, выполняемая в подзадаче уровня $j$, не считая работы, выполняемой в рекурсивных вызовах, не больше постоянной, помноженной на размер входных данных, возведенной в степень $d$: $c(n/b^j)^d$. Суммирование всех подзадач уровня $j$ дает верхнюю границу объема работ, выполняемых на уровне $j$ дерева рекурсии:

$$\text{работа на уровне } j \le \underbrace{a^j}_{\text{число подзадач}} \times \overbrace{c \times \underbrace{\left\lceil \frac{n}{b^j} \right\rceil}_{\text{размер входа}}}^{\text{работа над подзадачу}\,{}^{d}}$$

Давайте упростим это выражение, выделив части, которые зависят от уровня $j$, и части, которые от него не зависят:

$$\text{работа на уровне } j \le cn^d \times \left\lceil \frac{a}{b^d} \right\rceil^j .$$

Правая часть отмечает парадный вход критического соотношения $a/b^d$. Учитывая, что значение $a$ по сравнению с $b^d$ является именно тем, которое диктует соответствующий случай основного метода, мы не должны удивляться, что это соотношение появится в анализе.

## 4.4.4. Суммирование уровней

Сколько имеется уровней? Размер входных данных изначально равен $n$ и уменьшается с коэффициентом $b$ с каждым уровнем. Поскольку мы исходим из того, что $n$ является степенью $b$ и что базовый случай активируется, когда размер входных данных равняется 1, количество уровней в точности равняется количеству раз, когда нужно разделить $n$ на $b$, чтобы достигнуть 1, так называемого $\log_b n$. Просуммировав все уровни $j = 0,1,2, \ldots, \log_b n$, мы получаем следующую непостижимую верхнюю границу времени работы (используя тот факт, что $n^d$ не зависит от $j$ и может быть вынесен наружу):

$$\text{общая работа} \le cn^d \times \sum_{j=0}^{\log_b n} \left\lceil \frac{a}{b^d} \right\rceil^j . \qquad (4.4)$$

Хотите верьте, хотите нет, но мы достигли важной вехи в доказательстве основного метода. Возможно, правая часть (4.4) и выглядит как «суп из букв»,

но при правильной интерпретации из нее можно извлечь ключи, которые «открывают» глубокое понимание основного метода.

### 4.4.5. Добро против зла: потребность в трех случаях

Далее мы представим некоторую семантику временных рамок работы в (4.4) и разовьем интуицию в отношении того, почему границы времени работы в основном методе такие, какие они есть.

Почему соотношение $a$ и $b^d$ так важно? По сути, это сравнение представляет собой перетягивание каната между силами добра и силами зла. Зло представлено $a$, *темпом разрастания подзадачи* (ТРП) — с каждым уровнем рекурсии число подзадач взрывается с коэффициентом $a$, и это немного пугает. Добро принимает форму $b^d$, *темп сжатия работы* (ТСР). Хорошая новость заключается в том, что с каждым уровнем рекурсии объем работы на подзадачу уменьшается с коэффициентом $b^d$ [1]. Тогда ключевой вопрос в следующем: какая сторона победит, силы добра или силы зла? Три случая основного метода в точности соответствуют трем возможным исходам этого перетягивания каната: ничья (ТРП = ТСР), победа добра (ТРП < ТСР) или победа зла (ТРП > ТСР).

Чтобы лучше это понять, потратьте немного времени на размышления об объеме работы, выполняемой на каждом уровне дерева рекурсии (рис. 4.1). Когда объем выполняемой работы увеличивается вместе с уровнем $j$ дерева рекурсии? Когда он уменьшается? Остается ли он всегда одинаковым на каждом уровне?

---

[1] Почему $b^d$, а не $b$? Потому что $b$ — это темп, с которым сжимается размер входных данных, и нас интересует размер входных данных только в той мере, в какой он определяет объем выполняемой работы. Например, в алгоритме «разделяй и властвуй» с квадратичным шагом объединения ($d = 2$), когда размер входных данных сокращается наполовину ($b = 2$), для решения каждой более мелкой подзадачи (поскольку $b^d = 4$) требуется только 25 % работы.

**ТЕСТОВОЕ ЗАДАНИЕ 4.5**

Какое из следующих высказываний истинно? (Выберите все, что применимо.)

а) Если ТРП < ТСР, то объем выполняемой работы уменьшается с уровнем $j$ рекурсии.

б) Если ТРП > ТСР, то объем выполняемой работы увеличивается с уровнем $j$ рекурсии.

в) Никакие выводы о том, как объем работы меняется в зависимости от уровня $j$ рекурсии, сделать нельзя, если ТРП не равен ТСР.

г) Если ТРП = ТСР, то объем выполняемой работы одинаковый на каждом уровне рекурсии.

(Решение и пояснение см. в разделе 4.4.10.)

## 4.4.6. Предсказание границ времени работы

Теперь мы понимаем, почему основной метод включает три случая. Существует три принципиально разных типа деревьев рекурсии — с рабочим уровнем, который остается прежним, уменьшающимся или увеличивающимся, и относительные размеры $a$ (ТРП) и $b^d$ (ТСР) определяют тип дерева рекурсии алгоритма «разделяй и властвуй».

И даже лучше, у нас теперь есть достаточно интуитивного понимания, чтобы точно предсказать временные рамки работы, которые появляются в основном методе. Рассмотрим первый случай, когда $a = b^d$ и алгоритм выполняет одинаковый объем работы на каждом уровне своего дерева рекурсии. Мы, безусловно, знаем, сколько работы сделано в корне, на уровне 0 – $O(n^d)$, согласно явному описанию в рекуррентном соотношении. С работой в объеме $O(n^d)$ в расчете на уровень и с $1 + \log_b n = O(\log n)$ уровнями в этом случае мы должны ожидать временную границу $O(n^d \log n)$ (ср. случай 1 теоремы 4.1)[1].

[1] Сокращение «ср.» означает «сравнить».

Во втором случае побеждают $a < b^d$ и силы добра — объем выполняемых работ уменьшается вместе с уровнем. Следовательно, на уровне 0 выполняется больше работы, чем на любом другом уровне. Самый простой и лучший результат, на который мы могли бы надеяться, заключается в том, что работа, выполняемая в корне, доминирует над временем работы алгоритма. Поскольку $O(n^d)$ работы выполняется в корне, этот наилучший сценарий будет преобразован в общее время работы $O(n^d)$ (ср. случай 2 теоремы 4.1).

В третьем случае, когда подзадачи разрастаются еще быстрее, чем сокращается работа на подзадачу, объем выполняемой работы увеличивается вместе с уровнем рекурсии, причем б*о*льшая часть работы выполняется в листьях дерева. Опять же, самый простой и наилучший сценарий заключается в том, что время работы определяется работой, выполняемой в листьях. Лист соответствует рекурсивному вызову, при котором срабатывает базовый случай, поэтому алгоритм выполняет только $O(1)$ операций на лист. Сколько имеется листьев? Из решения тестового задания 4.4 мы знаем, что на каждом уровне $j$ имеется $a_j$ узлов. Листья находятся на последнем уровне $j = \log_b n$, поэтому имеется $a^{\log_b n}$ листьев. Следовательно, наилучший сценарий транслируется во временную границу $O(a^{\log_b n})$.

Остается загадкой связь между нашими предсказанными временными рамками работы для третьего случая основного метода ($O(a^{\log_b n})$) и фактическими временными рамками, которые появляются в теореме 4.1 ($O(n^{\log_b a})$). Связь состоит..., да, они абсолютно одинаковые! Тождество

$$\underbrace{a^{\log_b n}}_{\text{более интуитивно понятный}} = \underbrace{n^{\log_b a}}_{\text{легче применим}},$$

вероятно, выглядит как ошибка новичка, сделанная студентом-первокурсником, но на самом деле это правда[1]. Таким образом, временные рамки работы $O(n^{\log_b a})$ просто говорят о том, что работа, выполняемая в листьях дерева рекурсии, доминирует в вычислении, причем временные рамки изложены в форме, удобной для подставления параметров (относительно алгоритмов умножения целых чисел и матриц, проанализированных в разделе 4.3).

---

[1] Чтобы проверить это, просто возьмите логарифм по основанию $b$ обеих частей: $\log_b (a^{\log_b n}) = \log_b n \times \log_b a = \log_b a \times \log_b n = \log_b (n^{\log_b a})$. (И, поскольку $\log_b$ является монотонно возрастающей функцией, единственный способ, при котором $\log_b x$ и $\log_b y$ могут быть равными, — это если $x$ и $y$ равны.)

## 4.4.7. Заключительные расчеты: случай 1

Нам все еще нужно проверить, что наша интуиция в предыдущем разделе сработала правильно, и это можно сделать посредством формального доказательства. Кульминацией наших предыдущих расчетов были следующие пугающие верхние временные рамки работы алгоритма «разделяй и властвуй» в зависимости от параметров $a$, $b$ и $d$:

$$\text{общая работа} \leq cn^d \times \sum_{j=0}^{\log_b n} \left[ \frac{a}{b^d} \right]^j. \tag{4.5}$$

Мы получили эти рамки, акцентировав внимание на конкретном уровне $j$ дерева рекурсии (с его $a_j$ подзадачами и $c(n/b^j)^d$ работой на подзадачу) и затем просуммировав уровни.

Когда силы добра и зла находятся в прекрасном равновесии (то есть $a = b^d$) и алгоритм выполняет одинаковый объем работы на каждом уровне, правая часть (4.5) существенно упрощается:

$$cn^d \times \sum_{j=0}^{\log_b n} \underbrace{\left[ \underbrace{\frac{a}{b^d}}_{=1} \right]^j}_{=1 \text{ для каждого } j} = cn^d \times \underbrace{(1+1+\ldots+1)}_{1+\log_b n \text{ раз}}$$

которая равняется $O(n^d \log n)$[1].

## 4.4.8. Отклонение: геометрический ряд

Мы надеемся, что для второго и третьего типов деревьев рекурсии (соответственно уменьшающейся и увеличивающейся работой в расчете на уровень) общее время работы определяется работой, выполняемой на самом сложном уровне (корне и листьях соответственно). Для реализации этой надежды требуется понимание геометрических рядов, которые являются выражениями вида $1 + r + r^2 + \ldots + r^k$ для некоторого вещественного числа $r$ и неотрицательного целого числа $k$. (Для нас $r$ будет критическим соотношением $a / b^d$.) Всякий раз, когда вы видите такое параметризованное выражение, рекоменду-

[1] Помните, что, поскольку разные логарифмические функции различаются постоянным множителем, нет необходимости указывать основание логарифма.

ется иметь в виду пару канонических значений параметров. Например, если $r = 2$, то это сумма положительных степеней двойки: $1 + 2 + 4 + 8 + \cdots + 2^k$. Когда $r = \frac{1}{2}$, то это сумма отрицательных степеней двойки: $1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots + \frac{1}{2^k}$. Когда $r \neq 1$, существует полезная замкнутая формула геометрического ряда[1]:

$$1 + r + r^2 + \ldots + r^k = \frac{1 - r^{k+1}}{1 - r} \cdot \tag{4.6}$$

Для нас важны два следствия этой формулы. Во-первых, когда $r < 1$,

$$1 + r + r^2 + \ldots + r^k \leq \frac{1}{1 - r} = \text{константа } \left(\text{независимая от } k\right).$$

Следовательно, *в каждом геометрическом ряду с* $r < 1$ *доминирует его первый член* — первый член равен 1, а сумма составляет всего $O(1)$. Например, неважно, сколько сложить степеней ½ , результирующая сумма никогда не превышает 2.

Во-вторых, когда $r > 1$,

$$1 + r + r^2 + \ldots + r^k = \frac{r^{k+1} - 1}{r - 1} \leq \frac{r^{k+1}}{r - 1} = r^k \times \frac{r}{r - 1}.$$

Следовательно, *в каждом геометрическом ряду с* $r > 1$ *доминирует его последний член* — последний член равен $r^k$, в то время как сумма является не более чем постоянным множителем $(r/(r - 1))$, умноженным на это число. Например, если суммировать степени от 2 до 1024, итоговая сумма будет меньше 2048.

## 4.4.9. Заключительные вычисления: случаи 2 и 3

Возвращаясь к нашему анализу (4.5), допустим, что $a < b^d$. В этом случае разрастание в подзадачах компенсируется экономией в работе в расчете на подзадачу, а число выполняемых операций уменьшается вместе с уровнем дерева рекурсии. Зададим $r = a / b^d$; поскольку $a$, $b$ и $d$ являются константами (независимыми от размера $n$ входных данных), то и $r$ тоже. Поскольку $r < 1$, геометрический ряд в (4.5) является не более чем константой $1/(1-r)$, а граница времени работы в (4.5) становится

---

[1] Чтобы проверить это тождество, просто умножьте обе части на $1 - r$: $(1 - r)(1 + r + r^2 + \ldots + r^2) = 1 - r + r - r^2 + r^2 - r^3 + r^3 - \ldots - r^{k+1} = 1 - r^{k+1}$.

$$cn^d \times \underbrace{\sum_{j=0}^{\log_b n} r^j}_{=O(1)} = O\left(n^d\right),$$

где выражение $O$-большое устраняет константы $c$ и $1/(1-r)$. Это подтверждает нашу надежду на то, что в случае со вторым типом дерева рекурсии общая сумма выполняемой работы определяется работой, выполняемой в корне.

В последнем случае допустим, что $a > b^d$, где разрастание числа подзадач опережает скорость сжатия работы в расчете на подзадачу.

Установим $r = a/b^d$. Поскольку $r$ теперь больше 1, последний член геометрического ряда доминирует, и граница в (4.5) становится

$$cn^d \times \underbrace{\sum_{j=0}^{\log_b n} r^j}_{=O\left(r^{\log_b n}\right)} = O\left(n^d \times r^{\log_b n}\right) = O\left(n^d \times \left(\frac{a}{b^d}\right)^{\log_b n}\right). \tag{4.7}$$

Это выглядит запутанно, пока мы не взглянем на некоторых чудесных кандидатов на сокращение. Поскольку возведение в степень $b$ и логарифм по основанию $b$ являются обратными операциями, мы можем записать

$$\left(b^{-d}\right)^{\log_b n} = b^{-d\log_b n} = \left(b^{\log_b n}\right)^{-d} = n^{-d}.$$

Следовательно, член $\left(1/b\right)^{\log_b n}$ в (4.7) отменяет член $n^d$, приводя к верхней границе $O(a^{\log_b n})$. Это подтверждает нашу надежду на то, что общее время работы в этом случае будет определяться работой, выполняемой в листьях дерева рекурсии. Поскольку $a^{\log_b n}$ совпадает с $n^{\log_b a}$, мы завершили доказательство основного метода. *Ч. т. д.*

## 4.4.10. Решения тестовых заданий 4.4–4.5

### Решение тестового задания 4.4

**Правильный ответ: (б).** Сначала, по определению, «коэффициентом ветвления» дерева рекурсии является $a$ — каждый рекурсивный вызов, который не запускает базовый случай, делает новые рекурсивные вызовы. Это означает, что количество разных подзадач умножается на $a$ с каждым уровнем. Поскольку на уровне 0 имеется 1 подзадача, на уровне $j$ имеется $a^j$ подзадач.

Что касается второй части решения, опять же по определению, размер подзадачи уменьшается в $b$ раз с каждым уровнем. Поскольку на уровне 0 размер задачи равен $n$, все подзадачи на уровне $j$ имеют размер $n/b^j$ [1].

## Решение тестового задания 4.5

**Правильные ответы: (а), (б), (г).** Сначала предположим, что ТРП < ТСР, а значит, силы добра мощнее сил зла: сжатие в работе, выполняемой в расчете на подзадачу, больше компенсаций за увеличение числа подзадач. В этом случае алгоритм работает меньше с каждым последующим уровнем рекурсии. Следовательно, первое утверждение истинно (а третье утверждение ложно). Второе утверждение истинно по тем же причинам: если подзадачи растут настолько быстро, что опережают экономию в расчете на подзадачу, то каждый уровень рекурсии требует больше работы, чем предыдущий. В заключительном утверждении, когда ТРП = ТСР, существует совершенное равновесие между силами добра и зла. Подзадачи разрастаются, но наша экономия в работе в расчете на подзадачу увеличивается точно такими же темпами. Эти две силы уравновешиваются, и работа, выполняемая на каждом уровне дерева рекурсии, остается неизменной.

[1] В отличие от нашего анализа алгоритма `MergeSort`, тот факт, что число подзадач на уровне $j$ равняется $a^j$, не означает, что размер каждой подзадачи равен $n/a^j$. В алгоритме `MergeSort` входные данные для подзадач уровней $j$ образуют сегмент первоначальных входных данных. Это не относится ко многим другим нашим алгоритмам «разделяй и властвуй». Например, в рекурсивных алгоритмах целочисленного и матричного умножения части первоначальных входных данных повторно используются в разных рекурсивных вызовах.

**ВЫВОДЫ**

- Рекуррентное соотношение выражает границу $T(n)$ времени работы с точки зрения числа операций, выполняемых рекурсивными вызовами.
- Стандартное рекуррентное соотношение $T(n) \le aT\left(\frac{n}{b}\right) + O(n^d)$ определяется тремя параметрами: числом $a$ рекурсивных вызовов, коэффициентом сжатия $b$ размера входных данных и экспонентой $d$ во времени работы шага объединения.
- Основной метод представляет асимптотическую верхнюю границу для каждого стандартного рекуррентного соотношения как функцию $a$, $b$ и $d$: $O(n^d \log n)$, если $a = b^d$; $O(n^d)$, если $a < b^d$, и $O(n^{\log_b a})$, если $a > b^d$.
- Частные случаи включают временную границу $O(n \log n)$ для алгоритма `MergeSort`, временную границу $O(n^{1,59})$ для алгоритма `Karatsuba` и временную границу $O(n^{2,81})$ для алгоритма `Strassen`.
- Доказательство основного метода обобщает аргумент дерева рекурсии, использованный для анализа алгоритма `MergeSort`.
- Количества $a$ и $b^d$ представляют силы зла (темп разрастания подзадач) и силы добра (темп сжатия работы).
- Три случая основного метода соответствуют трем различным типам деревьев рекурсии: те, в которых выполняемая работа в расчете на уровень одинаковая на каждом уровне (совпадение между добром и злом), уменьшается с каждым уровнем (когда добро побеждает) и увеличивается с каждым уровнем (когда зло побеждает).
- Из свойств геометрического ряда вытекает, что работа, выполняемая в корне дерева рекурсии (которая равна $O(n^d)$), доминирует над полным временем работы во втором случае, в то время как работа, выполняемая в листьях ($O(a^{\log_b n}) = O(n^{\log_b a})$), доминирует в третьем случае.

## Задачи на закрепление материала

**Задача 4.1.** Вспомните основной метод (теорема 4.1) и его три параметра: $a$, $b$ и $d$. Что из нижеследующего является лучшей интерпретацией $b^d$?

а) Темп, с которым растет полная работа (в расчете на уровень рекурсии).

б) Темп, с которым растет число подзадач (в расчете на уровень рекурсии).

в) Темп, с которым уменьшается размер подзадач (в расчете на уровень рекурсии).

г) Темп, с которым уменьшается работа в расчете на подзадачу (в расчете на уровень рекурсии).

**Задача 4.2.** Этот и следующие два вопроса дадут вам дальнейший опыт работы с основным методом. Допустим, что время работы $T(n)$ алгоритма ограничено стандартным рекуррентным соотношением, где $T(n) \leq 7 \times T\left(\frac{n}{3}\right) + O(n^2)$. Какая из приведенных ниже границ является наименьшей правильной верхней границей асимптотического времени работы алгоритма?

а) $O(n \log n)$.

б) $O(n^2)$.

в) $O(n^2 \log n)$.

г) $O(n^{2,81})$.

**Задача 4.3.** Допустим, что время работы $T(n)$ алгоритма ограничено стандартным рекуррентным соотношением, где $T(n) \leq 9 \times T\left(\frac{n}{3}\right) + O(n^2)$. Какая из приведенных ниже границ является наименьшей правильной верхней границей асимптотического времени работы алгоритма?

а) $O(n \log n)$.

б) $O(n^2)$.

в) $O(n^2 \log n)$.

г) $O(n^{3,17})$.

**Задача 4.4.** Допустим, что время работы $T(n)$ алгоритма ограничено стандартным рекуррентным соотношением, где $T(n) \leq 5 \times T\left(\frac{n}{3}\right) + O(n)$. Какая из приведенных ниже границ является наименьшей правильной верхней границей асимптотического времени работы алгоритма?

а) $O(n^{\log_5 3})$.

б) $O(n \log n)$.

в) $O(n^{\log_3 5})$.

г) $O(n^{5/3})$.

д) $O(n^2)$.

е) $O(n^{2,59})$.

## Задача повышенной сложности

**Задача 4.5.** Допустим, что время работы $T(n)$ алгоритма ограничено (нестандартным!) рекуррентным соотношением, где $T(1) = 1$ и $T(n) \leq T(\lfloor \sqrt{n} \rfloor) + 1$ для $n > 1$[1]. Какая из приведенных ниже границ является наименьшей правильной верхней границей асимптотического времени работы алгоритма? (Обратите внимание, что основной метод неприменим!)

а) $O(1)$.

б) $O(\log \log n)$.

в) $O(\log n)$.

г) $O(\sqrt{n})$.

[1] Здесь $\lfloor x \rfloor$ обозначает функцию «floor», которая округляет свой аргумент вниз до ближайшего целого числа.

# 5

# *Алгоритм QuickSort*

В этой главе рассматривается быстрая сортировка `QuickSort`, алгоритм, который занимает почетное место в зале алгоритмической славы. Предоставив высокоуровневый обзор того, как этот алгоритм работает (раздел 5.1), мы обсудим, как подразделять массив вокруг «опорного элемента» за линейное время (раздел 5.2) и как выбирать хороший опорный элемент (раздел 5.3). Раздел 5.4 вводит рандомизированный алгоритм `QuickSort`, и раздел 5.5 доказывает, что его асимптотическое среднее время работы равняется $O(n \log n)$ для $n$-элементных массивов. Раздел 5.6 завершает наше обсуждение сортировки доказательством, что никакой алгоритм сортировки «на основе сравнения» не может быть быстрее $O(n \log n)$.

# 5.1. Обзор

Попросите профессионального специалиста в области информатики или программиста перечислить свою десятку лучших алгоритмов, и вы найдете алгоритм быстрой сортировки `Quicksort` во многих списках (включая и мой). И в чем же причина? Мы уже знаем один невероятно быстрый алгоритм сортировки (`MergeSort`) — зачем нужен еще один?

С практической точки зрения алгоритм быстрой сортировки `QuickSort` конкурентоспособен, нередко превосходит алгоритм `MergeSort` и по этой причине является методом сортировки, принятой по умолчанию во многих библиотеках программирования. Большое преимущество алгоритма `QuickSort` над алгоритмом `MergeSort` заключается в том, что он выполняется *на том же месте* — он работает с входным массивом только путем повторяющегося прямого обмена пар элементов, и по этой причине требует выделения лишь небольшого объема дополнительной оперативной памяти для промежуточных вычислений. С эстетической стороны алгоритм `QuickSort` — это просто удивительно красивый алгоритм с одинаково красивым анализом времени работы.

## 5.1.1. Сортировка

Алгоритм `Quicksort` решает задачу сортировки массива, ту же задачу мы решали в разделе 1.4.

**ЗАДАЧА: СОРТИРОВКА**

**Вход**: массив из *n* чисел в произвольном порядке.

**Выход**: массив тех же самых чисел, отсортированных от наименьшего до наибольшего.

Таким образом, если входной массив

| 3 | 8 | 2 | 5 | 1 | 4 | 7 | 6 |
|---|---|---|---|---|---|---|---|

тогда правильный выходной массив имеет вид

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|

Как и в нашем обсуждении алгоритма `MergeSort`, для простоты понимания допустим, что входной массив имеет разные элементы, без дубликатов[1].

## 5.1.2. Разделение вокруг опорного элемента

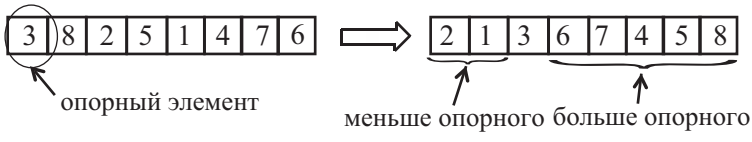
`QuickSort` построен вокруг быстрой подпрограммы «частичной сортировки», задача которой состоит в том, чтобы делить массив вокруг «опорного элемента».

**Шаг 1: Выбрать опорный элемент.** Сначала выбираем один элемент массива, который будет выступать в качестве *опорного элемента*. Раздел 5.3 будет полностью посвящен тому, как именно это должно быть сделано. А пока будем наивны и просто воспользуемся первым элементом массива («3» на рисунке выше).

---

[1] В маловероятном случае, если вам нужно реализовать алгоритм `QuickSort` самостоятельно, имейте в виду, что правильная и эффективная обработка совпадающих элементов немного сложнее, чем в алгоритме `MergeSort`. Подробное пояснение см. в разделе 2.3 «Алгоритмов» (четвертое издание Роберта Седжвика и Кевина Уэйна (Algorithms, Robert Sedgewick, Kevin Wayne, Addison-Wesley, 2011).

**Шаг 2: Перегруппировать входной массив вокруг опорного элемента.** При наличии опорного элемента следующая задача состоит в том, чтобы сгруппировать элементы массива так, чтобы все, что расположено в массиве перед $p$, было меньше $p$, а все, что после $p$, было больше $p$. Например, в случае с приведенным выше входным массивом один из законных способов перегруппирования элементов будет следующий:




Этот пример ясно показывает, что элементы перед опорным не нужно размещать в правильном относительном порядке («1» и «2» инвертированы) и то же касается элементов после опорного. Эта подпрограмма разделения помещает (неопорные) элементы массива в два блока, один для элементов меньше опорного, а другой — для элементов больше опорного.

Приведем два ключевых факта об этой подпрограмме разделения массива.

**Быстрота.** Подпрограмма разделения невероятно быстро внедряется, запускаясь за линейное $O(n)$ время. И, даже лучше, ключом к практической полезности алгоритма `QuickSort` является то, что данная подпрограмма может быть реализована на том же месте с практически нулевым потреблением оперативной памяти, не считая той, которая занимает входной массив[1]. В разделе 5.2 эта реализация описывается подробно.

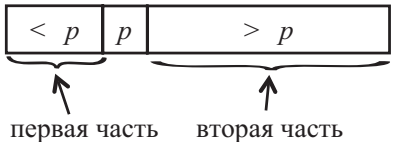
**Значительный прогресс.** Разделение массива вокруг опорного элемента приводит к сортировке массива. Во-первых, опорный элемент оказывается на своем законном месте, то есть в той же позиции, что и в отсортированной версии входного массива (где все элементы меньше опорного находятся перед ним и все элементы больше опорного — после него). Во-вторых, разделение массива сводит задачу сортировки к двум более мелким задачам: сортировке элементов меньше опорного (которые удобно размещаются в своем собствен-

[1] Это контрастирует с алгоритмом `MergeSort` (раздел 1.4), который многократно копирует элементы из одного массива в другой.

ном подмассиве) и элементов больше опорного (также в своем собственном подмассиве). После рекурсивной сортировки элементов в каждом из этих двух подмассивов алгоритм завершает работу[1]!

## 5.1.3. Высокоуровневое описание

В приведенном ниже высокоуровневом описании алгоритма `QuickSort` «первая часть» и «вторая часть» массива относятся к элементам, которые соответственно меньше и больше опорного:




**QUICKSORT (ВЫСОКОУРОВНЕВОЕ ОПИСАНИЕ)**

**Вход**: массив $A$ из $n$ разных целых чисел.

**Выход**: элементы массива $A$ отсортированы от наименьшего до наибольшего.

```
if n ≤ 1 then          // базовый случай – уже отсортирован
   return
выбрать опорный элемент p          // предстоит реализовать
разделить A вокруг p               // предстоит реализовать
рекурсивно отсортировать первую часть A
рекурсивно отсортировать первую часть A
```

Хотя алгоритмы `MergeSort` и `QuickSort` и являются алгоритмами «разделяй и властвуй», порядок операций в них различается. В `MergeSort` сначала выполняются рекурсивные вызовы, а затем шаг объединения, `Merge`. В алгоритме

[1] Одна из подзадач может быть пустой, если в качестве опорного выбран минимальный или максимальный элемент. В этом случае соответствующий рекурсивный вызов может быть пропущен.

QuickSort рекурсивные вызовы происходят после разделения массива, и их результаты не нужно объединять вообще[1]!

### 5.1.4. Забегая вперед

Наш оставшийся список неотложных дел:

1. (Раздел 5.2.) Как реализовать подпрограмму разделения массива?
2. (Раздел 5.3.) Как выбрать опорный элемент?
3. (Разделы 5.4 и 5.5.) Каково время работы алгоритма QuickSort?

Еще один вопрос состоит в следующем: мы действительно уверены, что алгоритм QuickSort всегда правильно сортирует входной массив? До сих пор я мало уделял внимания аргументам формальной правильности, потому что у студентов, как правило, имеется уверенное и точное интуитивное понимание того, почему алгоритмы «разделяй и властвуй» правильные. (Сравните это с пониманием времени работы алгоритмов «разделяй и властвуй», которое обычно далеко от очевидного!) Если у вас остались какие-либо вопросы, достаточно просто формально рассмотреть правильность алгоритма QuickSort, используя доказательство по индукции[2].

---

[1] Алгоритм QuickSort был придуман Тони Хоаром в 1959 году, когда ему было всего 25 лет. Хоар продолжил вносить большой фундаментальный вклад в языки программирования и был награжден премией Тьюринга, вручаемой Ассоциацией вычислительной техники (ACM) — эквивалентом Нобелевской премии в области computer science — в 1980 году.

[2] Следуя шаблону для индуктивных доказательств, рассмотренному в приложении A, обозначим через $P(n)$ высказывание «для каждого входного массива длиной $n$ алгоритм QuickSort его сортирует правильно». Базовый случай ($n = 1$) неинтересен: массив с 1 элементом неизбежно сортируется, и поэтому алгоритм QuickSort автоматически правилен в этом случае. В качестве индукционного перехода зададим произвольное положительное целое число $n \geq 2$. Мы можем принять допущение индуктивной гипотезы (то есть $P(k)$ является истинной для всех $k < n$), что означает, что алгоритм QuickSort правильно сортирует каждый массив с меньшим количеством элементов, чем $n$.

После шага разделения опорный элемент $p$ находится в той же позиции, что и в отсортированной версии входного массива. Элементы перед $p$ точно такие же, как и те, что перед $p$ в отсортированной версии входного массива (возможно, в неправильном относительном порядке), и то же самое касается элементов после $p$. Следователь-

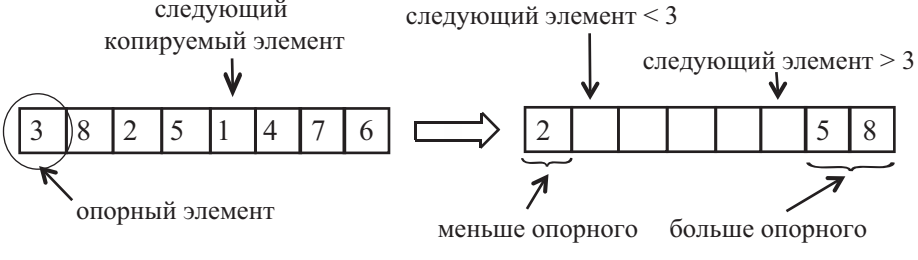
## 5.2. Разделение массива вокруг опорного элемента

Далее мы разберемся с деталями того, как необходимо разделять массив вокруг опорного элемента, то есть реорганизацию массива так, чтобы он выглядел следующим образом:

| $< p$ | $p$ | $> p$ |
|---|---|---|

### 5.2.1. Легкий выход из положения

Легко придумать линейную подпрограмму разделения, если нас не интересует выделение дополнительной оперативной памяти. Один из подходов состоит в том, чтобы выполнить одиночный просмотр входного массива $A$ и скопировать его неопорные элементы один за другим в новый массив $B$ той же длины, заполнив $B$ как спереди (для элементов меньше $p$), так и сзади (для элементов больше $p$). Опорный элемент можно скопировать в оставшуюся ячейку в $B$ после обработки всех неопорных элементов. В качестве работающего примера входного массива приведем снимок середины этого вычисления:




Поскольку объем работы для каждого из $n$ элементов входного массива составляет всего $O(1)$, время работы этой программы равняется $O(n)$.

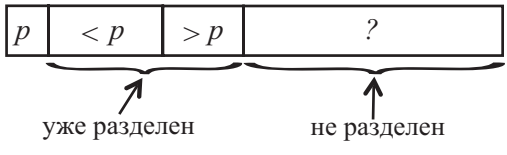
---

но, единственной оставшейся задачей является реорганизация элементов перед $p$ в отсортированном порядке и то же самое для элементов после $p$. Поскольку оба рекурсивных вызова выполняются с подмассивами длиной не более $n - 1$ ($p$ исключается, если иное не происходит), из индуктивной гипотезы следует, что оба вызова правильно сортируют свои подмассивы. На этом завершается индукционный переход и формальное доказательство правильности алгоритма `QuickSort`.

## 5.2.2. Реализация на том же месте: высокоуровневый план

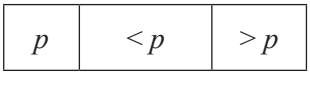
Как разделить массив вокруг опорного элемента, не выделив при этом почти никакой дополнительной оперативной памяти? Наш высокоуровневый подход будет делать единственный просмотр массива, обменивая пары элементов по мере необходимости, чтобы массив был правильно разделен к концу прохождения.

Допустим, что опорный элемент является первым элементом массива; это всегда может быть подкреплено (за время $O(1)$) за счет обмена опорного элемента с первым элементом массива на шаге предварительной обработки. По ходу просмотра и преобразования входного массива мы будем стараться обеспечивать, чтобы он имел следующую форму:



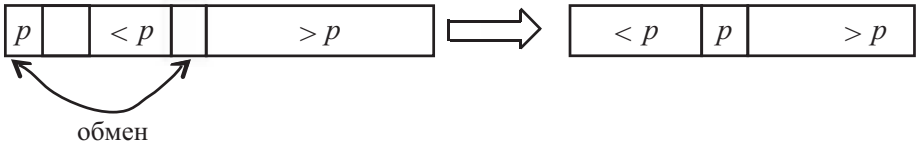

Таким образом, подпрограмма поддерживает следующий инвариант[1]: первым является опорный элемент; далее идут неопорные элементы, которые уже были обработаны, при этом все такие элементы меньше опорного предшествуют всем таким элементам больше опорного; затем в произвольном порядке идут еще не обработанные неопорные элементы.

Если наш план будет успешным, то по завершении линейного просмотра мы преобразуем массив так, чтобы он выглядел следующим образом:




Чтобы завершить разделение, мы можем поменять опорный элемент с последним элементом, который меньше него:

[1] Инвариант алгоритма — это свойство, которое всегда истинно в заданных точках его исполнения (как в конце каждой итерации цикла).



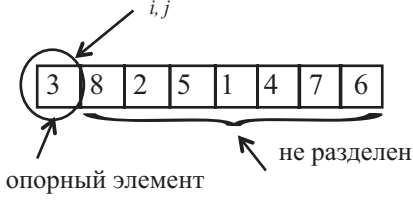

## 5.2.3. Пример

Далее мы пошагово рассмотрим на конкретном примере подпрограмму разделения, работающую на том же месте. Пошаговое рассмотрение примера программы до того, как вы познакомились с ее кодом, может показаться странным, но поверьте: это самый короткий путь к пониманию этой подпрограммы.

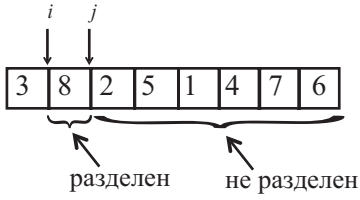
Основываясь на нашем высокоуровневым плане, мы будем отслеживать две границы: границу между неопорными элементами, которые мы уже рассмотрели, и теми, которые еще нет, и внутри первой группы — границу между элементами меньше опорного, и теми, которые больше опорного. Мы будем использовать индексы *j* и *i* соответственно, чтобы отслеживать эти две границы. Наш желаемый инвариант тогда можно перефразировать так:

**Инвариант**: все элементы между опорным и *i* — меньше опорного и все элементы между *i* и *j* — больше опорного.

Индексы *i* и *j* инициализируются границей между опорным элементом и остальными. Тогда между опорным и *j* нет элементов, и инвариант соблюдается в пустом виде:

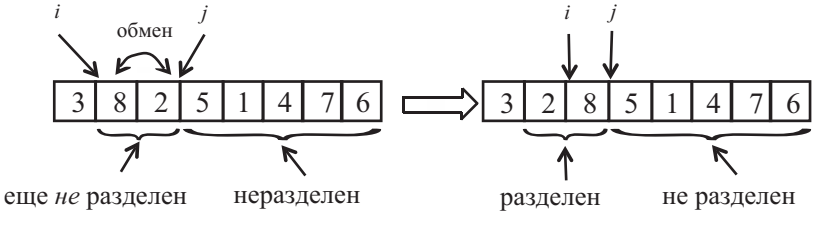



Во время каждой итерации подпрограмма рассматривает один новый элемент и увеличивает *j*. Для поддержания инварианта может потребоваться дополнительная работа. В первый раз, когда мы увеличиваем *j* в нашем примере, мы получаем:

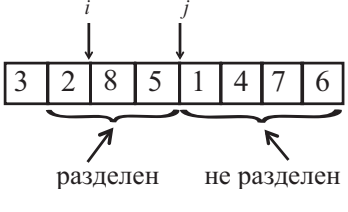



Между опорным элементом и *i* нет элементов, и единственный элемент между *i* и *j* («8») — больше опорного, поэтому инвариант по-прежнему соблюдается.

Теперь интрига нарастает. После приращения *j* во второй раз между *i* и *j* есть элемент, который меньше опорного («2»), это нарушение инварианта. Чтобы восстановить инвариант, мы обмениваем «8» на «2», а также увеличиваем *i*, чтобы он вклинился между «2» и «8» и снова очертил границу между обрабатываемыми элементами меньше и больше опорного:

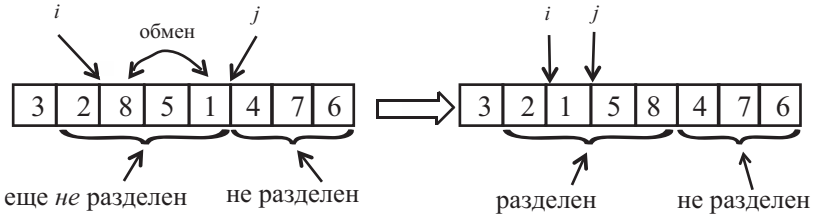



Третья итерация аналогична первой. Мы обрабатываем следующий элемент («5») и увеличиваем *j*. Поскольку новый элемент больше опорного, инвариант продолжает соблюдаться, и больше делать нечего:

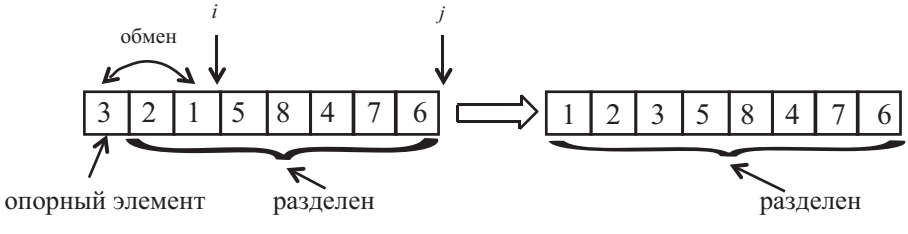



Четвертая итерация аналогична второй. Приращение *j* вносит элемент меньше опорного («1») между *i* и *j*, что нарушает инвариант. Но восстановить инва-

риант достаточно легко — просто надо обменять «1» с первым элементом больше опорного («8») и увеличить *i*, чтобы отразить новую границу между обрабатываемыми элементами меньше и больше опорного:




Последние три итерации обрабатывают элементы, которые больше опорного, поэтому ничего не нужно делать, кроме приращения *j*. После того как все элементы были обработаны и все элементы после опорного были разделены, мы завершаем окончательным обменом опорного элемента и последнего элемента меньше него:




В соответствии с требованием в конечном массиве все элементы меньше опорного идут перед ним, и все элементы больше опорного идут после него. То, что «1» и «2» находятся в отсортированном порядке, является совпадением. Элементы после опорного, очевидно, находятся не в отсортированном порядке.

## 5.2.4. Псевдокод Partition

Псевдокод подпрограммы разделения `Partition` — это именно то, что вы ожидаете увидеть после примера[1].

**PARTITION**

**Вход**: массив $A$ из $n$ разных целых чисел, левая и правая конечные точки, $\ell, r \in \{1, 2, \ldots, n\}$, где $\ell \leq r$

**Постусловие**: элементы подмассива $A[\ell], A[\ell + 1], \ldots, A[r]$ разделены вокруг $A[\ell]$.

**Выход**: конечная позиция опорного элемента.

```
p := A[ℓ]
i := ℓ + 1
for j := ℓ + 1 to r do
    if A[j] < p then          // если A[j] > p, ничего
                                 не делать
        обменять A[j] с A[i]
        i := i + 1            // восстанавливает инвариант
обменять A[ℓ] с A[i - 1]      // поместить опорный элемент
                                 правильно
return i - 1                  // сообщить конечную позицию
                                 опорного элемента
```

Подпрограмма `Partition` принимает в качестве входных данных массив $A$, но работает только с подмассивом элементов $A[\ell], \ldots, A[r]$, где $\ell$ и $r$ — это заданные параметры. Забегая вперед, каждый рекурсивный вызов алгоритма `QuickSort` будет отвечать за определенное непрерывное подмножество первоначального входного массива, а параметры $\ell$ и $r$ будут задавать соответствующие конечные точки.

[1] Если вы заглянете в другие учебники или в интернет, то увидите несколько вариантов этой подпрограммы, которые различаются в деталях. (Есть даже версия в исполнении венгерского ансамбля народных танцев! См. https://www.youtube.com/watch?v=ywWBy6J5gz8.) Эти варианты одинаково подходят для наших целей.

Как и в примере, индекс $j$ отслеживает, какие элементы были обработаны, в то время как $i$ отслеживает границу между обработанными элементами, которые меньше и больше опорного (где $A[i]$ — это самый левый обработанный элемент больше опорного, если таковой имеется). Каждая итерация цикла обрабатывает новый элемент. Как и в примере, когда новый элемент $A[j]$ — больше опорного, инвариант выполняется автоматически, и делать нечего. В противном случае функция восстанавливает инвариант, меняя местами $A[j]$, новый элемент, и $A[i]$, самый левый элемент больше опорного, и увеличивая $i$, чтобы обновить границу между элементами, которые меньше и больше опорного[1, 2]. Последний шаг, как было объявлено ранее, ставит опорный элемент на свое законное место, смещая правый элемент меньше, чем его. Подпрограмма `Partition` завершается, возвращая эту позицию назад в вызов алгоритма `QuickSort`, который ее вызвал.

Эта реализация невероятно быстра. Она выполняет только постоянное число операций для каждого элемента $A[\ell]$, ..., $A[r]$ соответствующего подмассива, и поэтому выполняется за время, линейное по длине этого подмассива. Важно отметить, что подпрограмма работает на этом подмассиве на том же месте, не выделяя никакой дополнительной памяти за пределами $O(1)$ объема, необходимого для отслеживания переменных, таких как $i$ и $j$.

---

[1] Обмен не нужен, если элементы больше опорного еще не встречались, — подмассив обрабатываемых элементов разделяется тривиально. Но дополнительный обмен безвреден (как вы должны убедиться), поэтому мы будем придерживаться нашего простого псевдокода.

[2] Почему этот обмен и приращение индекса всегда восстанавливают инвариант? Инвариант соблюдается перед самым последним приращением $j$ (по индукции, если вы хотите подойти к этому формально). Это означает, что все элементы $A[\ell + 1], \ldots, A[i - 1]$ — меньше опорного, и все элементы $A[i], \ldots, A[j - 1]$ — больше опорного. Единственная проблема заключается в том, что $A[j]$ — меньше опорного. После обмена $A[i]$ с $A[j]$ элементы $A[\ell + 1],\ldots, A[i]$ и $A[i + 1],\ldots, A[j]$, соответственно меньше и больше опорного. После увеличения $i$ элементы $A[\ell + 1], \ldots, A[i - 1]$ и $A[i], \ldots, A[j]$ соответственно меньше и больше опорного, что и восстанавливает инвариант.

### 5.2.5. Псевдокод алгоритма Quicksort

Теперь у нас есть полное описание алгоритма `QuickSort`, в остатке подпрограмма `ChoosePivot`, которая выбирает опорный элемент.

**QUICKSORT**

**Вход**: массив $A$ из $n$ разных целых чисел, левая и правая конечные точки, $\ell, r \in \{1, 2, \ldots, n\}$.

**Постусловие**: элементы подмассива $A[\ell], A[\ell+1], \ldots, A[r]$ отсортированы от наименьшего до наибольшего.

```
if ℓ ≥ r then // 0- или 1-элементный подмассив
    return
i := ChoosePivot(A, ℓ, r) // предстоит реализовать
обменять A[ℓ] с A[i]      // сделать опорный первым
j := Partition(A, ℓ, r)   // j = новая опорная позиция
QuickSort(A, ℓ, j - 1)    // рекурсивно вызвать с первой
                             частью
QuickSort(A, j + 1, r)    // рекурсивно вызвать со второй
                             частью
```

Сортировка $n$-элементного массива $A$ сводится к вызову функции `QuickSort` $(A, 1, n)$[1].

## 5.3. Важность хороших опорных элементов

Действительно ли `Quicksort` — быстрый алгоритм? Ставки высоки: простые алгоритмы сортировки, такие как `InsertionSort`, выполняются за квадратичное ($O(n^2)$) время, и мы уже знаем один алгоритм сортировки (`MergeSort`), который выполняется за время $O(n \log n)$. Ответ на этот вопрос зависит от того, как мы реализуем функцию `ChoosePivot`, которая выбирает один элемент

[1] Массив $A$ всегда передается по ссылке, то есть все вызовы функций работают непосредственно с исходной копией входного массива.

из указанного подмассива. Чтобы `QuickSort` был быстрым, важно, чтобы были выбраны «хорошие» опорные элементы, то есть опорные элементы, которые приводят к двум подзадачам примерно одинакового размера.

### 5.3.1. Наивная реализация подпрограммы ChoosePivot

В нашем обзоре алгоритма `Quicksort` мы упомянули наивную реализацию, которая всегда выбирает первый элемент.

---

`CHOOSEPIVOT` **(НАИВНАЯ РЕАЛИЗАЦИЯ)**

**Вход**: массив $A$ из $n$ разных целых чисел, левая и правая конечные точки $\ell, r \in \{1, 2, \ldots, n\}$.

**Выход**: индекс $i \in \{\ell, \ell + 1, \ldots, r\}$.

---

```
return ℓ
```

---

Разве эта наивная реализация уже недостаточно хороша?

---

**ТЕСТОВОЕ ЗАДАНИЕ 5.1**

Каково время работы алгоритма `QuickSort` с наивной реализацией подпрограммы `ChoosePivot`, если $n$-элементный входной массив уже отсортирован?

а) $\Theta(n)$.

б) $\Theta(n \log n)$.

в) $\Theta(n^2)$.

г) $\Theta(n^3)$.

(Решение и пояснение см. в разделе 5.3.3.)

---

## 5.3.2. Избыточная реализация ChoosePivot

Тестовое задание 5.1 рисует наихудшую картину того, что может произойти в алгоритме `QuickSort`, когда в расчете на рекурсивный вызов удаляется только один элемент. Каким будет *наилучший* сценарий? Идеально сбалансированное разбиение достигается *медианой* элемента массива, то есть элементом, для которого одинаковое количество других элементов меньше него и больше него[1]. Поэтому, если мы хотим приложить усилия в работе с опорным элементом, мы можем вычислить медиану элемента заданного подмассива.

**CHOOSEPIVOT (ИЗБЫТОЧНАЯ РЕАЛИЗАЦИЯ)**

**Вход**: массив $A$ из $n$ разных целых чисел, левая и правая конечные точки $\ell, r \in \{1, 2, \ldots, n\}$.

**Выход**: индекс $i \in \{\ell, \ell + 1, \ldots, r\}$

вернуть позицию медианы элемента $\{A[\ell], ..., A[r]\}$

В следующей главе мы увидим, что медиана элемента массива может быть вычислена за время, линейное по длине массива; давайте примем этот факт на веру для следующего тестового задания[2]. Существует ли какая-то награда за усердную работу по вычислению идеального опорного элемента?

[1] Например, медианой массива, содержащего {1, 2, 3, ..., 9}, будет 5. Для массива четной длины есть два законных варианта выбора медианы, и любой из них подходит для наших целей. Поэтому в массиве, который содержит {1, 2, 3, ..., 10}, можно считать срединным элементом 5 или 6.

[2] На данный момент вы знаете алгоритм со временем $O(n \log n)$ для вычисления медианы массива. (Подсказка: сортировать!)

**ТЕСТОВОЕ ЗАДАНИЕ 5.2**

Каково время работы алгоритма `Quicksort` с реализацией избыточной подпрограммы `ChoosePivot` на произвольном $n$-элементном входном массиве? Исходите из того, что подпрограмма `ChoosePivot` выполняется за время $\Theta(n)$.

Недостаточно информации для ответа.

а) $\Theta(n)$.

б) $\Theta(n \log n)$.

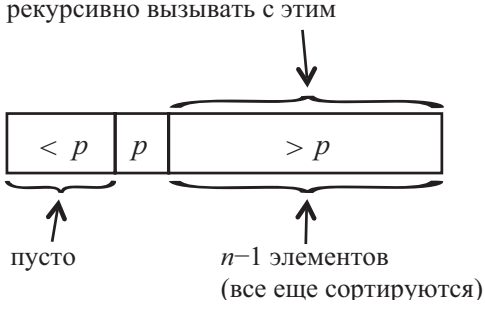
в) $\Theta(n^2)$.

(Решение и пояснение см. в разделе 5.3.3.)

## 5.3.3. Решения тестовых заданий 5.1– 5.2

### Решение тестового задания 5.1

**Правильный ответ: (в).** Сочетание наивно отобранных опорных элементов и уже отсортированного входного массива приводит к тому, что алгоритм `QuickSort` выполняется за время $\Theta(n^2)$, что намного хуже алгоритма `MergeSort` и не лучше простых алгоритмов, таких как `InsertionSort`. Что же произошло? Подпрограмма `Partition` в самом внешнем вызове алгоритма `QuickSort` с первым (наименьшим) элементом в качестве опорного ничего не делает: она проносится по массиву, и, поскольку встречает только элементы, которые больше опорного, она ни разу не меняет местами ни одну пару элементов. После того как этот вызов подпрограммы `Partition` завершится, картина будет следующей:



В непустом рекурсивном вызове этот шаблон повторяется: подмассив уже отсортирован, первый (наименьший) элемент выбран в качестве опорного, и есть один пустой рекурсивный вызов и один рекурсивный вызов, которому передается подмассив из $n - 2$ элементов. И так далее.

В конце подпрограмма `Partition` вызывается с подмассивами длиной $n$, $n - 1$, $n - 2$, ..., 2. Поскольку работа, выполняемая в одном вызове подпрограммы `Partition`, пропорциональна длине подмассива вызова, общий объем работы, выполняемой алгоритмом `QuickSort` в этом случае, пропорционален

$$\underbrace{n + (n-1) + (n-2) + \ldots + 1}_{= \Theta(n^2)}$$

и, следовательно, квадратичен по длине $n$ входных данных[1].

## Решение тестового задания 5.2

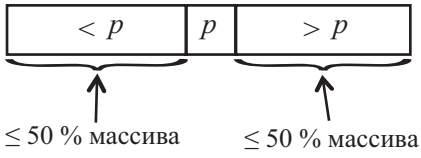
**Правильный ответ: (в).** В этом наилучшем сценарии алгоритм `QuickSort` выполняется за время $\Theta(n \log n)$. Причина в том, что его время работы определяется тем же самым рекуррентным соотношением, которое управляет временем работы `MergeSort`. То есть если $T(n)$ обозначает время работы этой реализации `QuickSort` на массивах длиной $n$, то

$$T(n) = \underbrace{2 \times T\left(\frac{n}{2}\right)}_{\text{поскольку опорный элемент = медиана}} + \underbrace{\Theta(n)}_{\text{ChoosePivot и Partition}}$$

Основная работа, выполняемая вызовом алгоритма `QuickSort` вне рекурсивных вызовов, делается в его подпрограммах `ChoosePivot` и `Partition`. Мы исходим из того, что первая составляет $\Theta(n)$, и раздел 5.2 доказывает, что последняя тоже составляет $\Theta(n)$. Поскольку в качестве опорного элемента мы используем медиану элемента, мы получаем идеальное разбиение входного массива, и каждый рекурсивный вызов получает подмассив с не более чем $\frac{n}{2}$ элементами:

---

[1] Быстрый способ увидеть, что $n + (n - 1) + (n - 2) + \cdots + 1 = \Theta(n^2)$, — обратить внимание на то, что это выражение не более $n^2$ (каждый из $n$ членов составляет не более $n$) и не менее $n^2/4$ (каждый из первых $n/2$ членов составляет не менее $n/2$).




Применение основного метода (теорема 4.1) $a = b = 2$ и $d = 1$ тогда дает $T(n) = \Theta(n \log n)$[1].

## 5.4. Рандомизированный алгоритм Quicksort

Выбор первого элемента подмассива в качестве опорного занимает всего $O(1)$ времени, но может привести к тому, что алгоритм `QuickSort` будет работать за время $\Theta(n^2)$. Выбор медианы элемента в качестве опорного гарантирует общее время работы $\Theta(n \log n)$, но занимает гораздо больше времени (если оно все еще линейное). Можно ли получить лучшее из обоих миров? Есть ли простой и легкий способ выбрать опорный элемент, который приводит к примерно сбалансированному разбиению массива? Ответ — да. Ключевая идея — использовать *рандомизацию*.

### 5.4.1. Рандомизированная реализация подпрограммы ChoosePivot

Рандомизированный алгоритм — это алгоритм, который по ходу работы «подбрасывает монеты» и может принимать решения на основе результатов этих подбрасываний монет. Если вы снова и снова запускаете рандомизированный алгоритм с одними и теми же входными данными, то увидите разное поведение в разных прогонах. Все основные языки программирования включают библиотеки, которые позволяют легко выбирать случайные числа по своему усмотрению, и рандомизация — это инструмент, который должен быть в арсенале любого серьезного проектировщика алгоритмов.

---

[1] Технически мы используем здесь вариант основного метода, который работает с обозначением Θ-большое, а не *O*-большое, но в остальном он такой же, как и в теореме 4.1.

С какой стати возникает потребность ввести случайность в свой алгоритм? Разве алгоритмы не самая детерминированная вещь, которую только можно представить? Оказывается, существуют сотни вычислительных задач, для которых рандомизированные алгоритмы быстрее, эффективнее или легче программировать, чем их детерминированные аналоги[1].

Самый простой способ внедрить случайность в алгоритм `QuickSort`, который оказывается чрезвычайно эффективным, — всегда выбирать опорные элементы *равномерно случайным образом*.

**CHOOSEPIVOT (РАНДОМИЗИРОВАННАЯ РЕАЛИЗАЦИЯ)**

**Вход**: массив $A$ из $n$ разных целых чисел, левая и правая конечные точки $\ell, r \in \{1, 2, \ldots, n\}$.

**Выход**: индекс $i \in \{\ell, \ell + 1, \ldots, r\}$.

вернуть элемент $\{\ell, \ell + 1, \ldots, r\}$, выбираемый равномерно случайным образом

Например, если $\ell = 41$ и $r = 50$, то у каждого из 10 элементов $A[41]$, ..., $A[50]$ есть 10 %-ный шанс быть выбранным в качестве опорного элемента[2].

## 5.4.2. Время работы рандомизированного алгоритма Quicksort

Время работы рандомизированного алгоритма `QuickSort` с опорными элементами, выбираемыми случайным образом, не всегда одинаково. Всегда есть какой-то шанс, хотя и отдаленный, что алгоритм всегда будет выбирать

[1] Специалистам в области computer science потребовалось некоторое время, чтобы это выяснить, причем шлюзы были открыты в середине 1970-х годов вместе с быстрыми рандомизированными алгоритмами для проверки того, является ли целое число простым.

[2] Еще одним не менее полезным способом является рандомизированная перетасовка входного массива на этапе предварительной обработки и затем выполнение наивной реализации алгоритма `QuickSort`.

минимальный элемент в качестве опорного из оставшегося подмассива, что приводит к времени работы $\Theta(n^2)$, наблюдавшемуся в тестовом задании 5.1[1]. Есть также небольшой шанс, что алгоритм, по невероятно счастливому совпадению, всегда в качестве опорного будет выбирать срединный элемент подмассива, в результате приводя к времени работы $\Theta(n \log n)$, наблюдавшемуся в тестовом задании 5.2. Таким образом, время работы алгоритма колеблется между $\Theta(n \log n)$ и $\Theta(n^2)$ — и тогда какой сценарий происходит чаще: наилучший или наихудший? Удивительно, но производительность алгоритма `QuickSort` почти всегда близка к его наилучшей производительности.

**Теорема 5.1 (время работы рандомизированного Quicksort).** *Для каждого входного массива длиной $n \geq 1$ среднее время работы рандомизированного алгоритма* `Quicksort` *равняется $O(n \log n)$.*

Слово «среднее» в формулировке теоремы относится к случайности в самом алгоритме `QuickSort`. Теорема 5.1 *не* исходит из того, что входной массив является случайным. Рандомизированный алгоритм `QuickSort` является универсальным алгоритмом (ср. раздел 1.6.1): независимо от того, что собой представляет ваш входной массив, если выполнять алгоритм с ним снова и снова, среднее время работы будет $O(n \log n)$, то есть достаточно хорошим, чтобы квалифицироваться как бесплатный примитив. В принципе рандомизированный алгоритм `QuickSort` может выполняться за время $\Theta(n^2)$, но на практике вы почти всегда будете наблюдать время работы $O(n \log n)$. Два дополнительных бонуса: константа, скрытая в обозначении $O$-большое в теореме 5.1, достаточно мала (как в `MergeSort`), и алгоритм не тратит время на выделение и управление дополнительной оперативной памятью (в отличие от `MergeSort`).

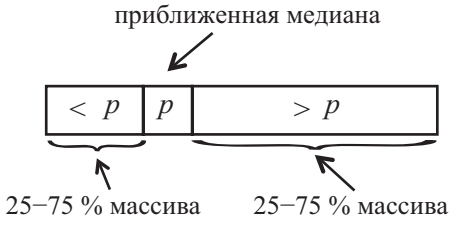
### 5.4.3. Интуитивное понимание: чем хороши случайные опорные элементы?

Для более глубокого понимания того, почему алгоритм `QuickSort` такой быстрый, ничто не может быть лучше исследования доказательства теоремы 5.1,

[1] Даже при незначительных значениях $n$ вероятность того, что при чтении этой книги в вас попадет метеорит, будет выше!

которая объясняется в разделе 5.5. В рамках подготовки к этому доказательству, а также в качестве утешительного приза для читателя, который слишком ограничен во времени, чтобы проглотить раздел 5.5, мы далее развиваем интуитивное понимание того, почему теорема 5.1 должна быть истинной.

Первое представление заключается в том, что для достижения времени работы $O(n \log n)$, как и в наилучшем сценарии в тестовом задании 5.2, использовать медиану элемента в качестве опорного является избыточным решением. Предположим, вместо этого мы используем «приближенную медиану», имея в виду некоторый элемент, который дает нам разбиение 25−75 % или больше. Это эквивалентно тому, что этот элемент больше как минимум 25 % других элементов, а также меньше как минимум 25 % других элементов. Рисунок после разделения вокруг такого опорного элемента будет следующим:




Если каждый рекурсивный вызов выбирает опорный элемент, который является приближенной медианой в данном случае, время работы алгоритма `QuickSort` по-прежнему составит $O(n \log n)$. Мы не можем вывести этот факт непосредственно из основного метода (теорема 4.1), потому что использование не медиан приводит к подзадачам с разными размерами. Но нетрудно обобщить анализ алгоритма `MergeSort` (раздел 1.5) так, чтобы он также применялся и здесь[1].

[1] Нарисуйте дерево рекурсии алгоритма. Всякий раз, когда `QuickSort` рекурсивно себя вызывает с двумя подзадачами, эти подзадачи включают разные элементы (те, которые меньше опорного, и те, которые больше него). Это означает, что для каждого уровня $j$ рекурсии нет наложений между подмассивами разных подзадач уровня $j$, и поэтому сумма длин подмассивов подзадач уровня $j$ не превышает $n$. Общая работа, выполняемая на этом уровне (вызовами подпрограммы `Partition`), линейна по сумме длин подмассивов. Таким образом, как и в алгоритме `MergeSort`, этот алгоритм выполняет $O(n)$ работу в расчете на уровень рекурсии. Сколько уровней имеется? С опорными элементами, которые являются приближенными

Второе представление заключается в том, что в рандомизированном алгоритме `QuickSort` выбор медианы элемента считается невероятным везением (с шансом всего 1 из $n$), а вот выбор приближенной медианы — лишь небольшое везение. Например, рассмотрим массив, содержащий элементы {1, 2, 3, ..., 100}. Любое число от 26 до 75 включительно является приближенной медианой как минимум с 25 элементами меньше медианы и 25 элементами больше нее. А это 50 % чисел в массиве! Таким образом, алгоритм `QuickSort` имеет шанс 50/50 случайным образом выбрать приближенную медиану, как если бы он пытался угадать результат броска симметричной монеты. Это означает, что мы ожидаем, что примерно 50 % вызовов алгоритма `QuickSort` будут использовать приближенные медианы, и мы можем надеяться, что анализ времени работы $O(n \log n)$ в предыдущем абзаце продолжит соблюдаться, возможно, с количеством уровней в два раза большим, чем раньше.

Не ошибитесь: это не формальное доказательство, а просто эвристический аргумент в пользу того, что теорема 5.1 может быть действительно истинной. На вашем месте, учитывая центральное положение алгоритма `QuickSort` в проектировании и анализе алгоритмов, я потребовал бы бесспорного аргумента в пользу того, что теорема 5.1 действительно истинна.

## *5.5. Анализ рандомизированного Quicksort

Рандомизированный алгоритм `QuickSort` кажется отличной идеей, но как действительно узнать, что он будет работать хорошо? В более общем плане, когда вы придумываете новый алгоритм в своей собственной работе, как вы узнаете, является ли он блестящим или же плохо пахнет? Один полезный, но и ситуативный подход — запрограммировать алгоритм и испытать его на куче разных входных данных. Другой подход состоит в том, чтобы развить интуитивное понимание того, почему алгоритм должен работать хорошо, как в разделе 5.4.3, для рандомизированного алгоритма `QuickSort`. Однако четкое

---

медианами, не более 75 % элементов передаются одному и тому же рекурсивному вызову, и поэтому размер подзадачи снижается по меньшей мере с коэффициентом в 4/3 с каждым уровнем. Это означает, что в дереве рекурсии имеется не более $\log_{4/3} n = O(\log n)$ уровней, и поэтому в целом работа выполняется за $O(n \log n)$.

понимание того, что делает алгоритм хорошим или плохим, часто требует математического анализа. Этот раздел поможет вам понять, почему алгоритм `QuickSort` такой быстрый.

Этот раздел исходит из того, что читатель знаком с понятиями из дискретной вероятности, которые рассматриваются в приложении Б: выборочные пространства, события, случайные величины, математическое ожидание и линейность математического ожидания.

### 5.5.1. Предварительные сведения

Теорема 5.1 утверждает, что для каждого входного массива длиной $n \geq 1$ среднее время работы рандомизированного алгоритма `QuickSort` (с опорными элементами, выбираемыми равномерно случайным образом) равняется $O(n \log n)$. Начнем с перевода этого утверждения в формулировку на языке дискретной вероятности.

Зададим для остальной части анализа произвольный входной массив $A$ длиной $n$. Напомним, что выборочное пространство — это множество всех возможных исходов некоторого случайного процесса. В рандомизированном алгоритме `QuickSort` вся случайность заключается в случайном выборе опорных элементов в разных рекурсивных вызовах. Таким образом, мы принимаем выборочное пространство $\omega$ как множество всех возможных исходов случайных вариантов выбора в алгоритме `QuickSort` (то есть всех последовательностей опорных элементов).

Напомним, что случайная величина является численной мерой результата случайного процесса — вещественной функцией, определенной на $\omega$. Случайная величина, которая нас интересует, — это число $RT$ примитивных операций (то есть строк кода), выполняемых рандомизированным алгоритмом `QuickSort`. Это четко сформулированная случайная величина, потому что всякий раз, когда все варианты опорных элементов заранее определены (то есть $\omega \in \Omega$), алгоритм `QuickSort` имеет некоторое фиксированное время работы $\mathrm{RT}(\omega)$. Варьируясь по всем возможным вариантам $\omega$, $RT(\omega)$ колеблется от $\Theta(n \log n)$ до $\Theta(n^2)$ (см. раздел 5.3).

Мы можем обойтись анализом более простой случайной величины, которая подсчитывает только сравнения и игнорирует другие типы выполняемых

примитивных операций. Обозначим через $C$ случайную величину, равную количеству сравнений между парами входных элементов, выполняемых алгоритмом `QuickSort` с заданной последовательностью вариантов опорных элементов. Оглядываясь назад на псевдокод, мы видим, что эти сравнения происходят точно в одном месте: в строке «`if` $A[j] < p$» в подпрограмме `Partition` (раздел 5.2.4), которая сравнивает текущий опорный элемент с неким другим элементом входного подмассива.

Следующая ниже лемма показывает, что сравнения доминируют в общем времени работы алгоритма `QuickSort`. Имеется в виду, что последнее больше, чем первое, только на постоянный множитель. Из этого следует, что, чтобы доказать верхнюю границу $O(n \log n)$ ожидаемого времени работы алгоритма `QuickSort`, нам нужно лишь доказать верхнюю границу $O(n \log n)$ ожидаемого количества выполняемых сравнений.

**Лемма 5.2.** *Существует константа $a > 0$, такая, что для каждого входного массива $A$ длиной не менее 2 и каждой последовательности $\omega$ опорных элементов $RT(\omega) \leq a \times C(\omega)$.*

Мы включаем доказательство для скептиков; пропустите его, если вы находите лемму 5.2 интуитивно очевидной.

*Доказательство леммы 5.2*. Прежде всего, при каждом вызове подпрограммы `Partition` опорный элемент сравнивается ровно один раз с любым другим элементом в заданном подмассиве. Следовательно, число сравнений в вызове является линейным по длине подмассива, и при исследовании псевдокода в разделе 5.2.4 общее число операций в вызове является не более чем константой, умноженной на него. При исследовании псевдокода в разделе 5.2.5 рандомизированный алгоритм `QuickSort` выполняет только постоянное число операций в каждом рекурсивном вызове вне подпрограммы `Partition`[1]. Всего имеется не более $n$ рекурсивных вызовов алгоритма `QuickSort` — каждый элемент входного массива может быть выбран в качестве опорного только

[1] Эта формулировка принимает допущение, что выбор случайного опорного элемента засчитывается за одну примитивную операцию. Доказательство остается действительным, даже если выбор случайного опорного элемента требует $\Theta(\log n)$ примитивных операций (как вы должны убедиться), и это распространяется на типичные реализации на практике генераторов случайных чисел.

один раз, прежде чем он будет исключен из всех последующих рекурсивных вызовов, и поэтому общая работа вне вызовов подпрограммы `Partition` составляет $O(n)$. Если суммировать все рекурсивные вызовы, общее число $RT(\omega)$ операций составит не более чем константу, умноженную на число $C(\omega)$ сравнений, плюс $O(n)$. Поскольку $C(\omega)$ всегда пропорциональна $n$ (или даже $n \log n$), дополнительная $O(n)$ работа может быть поглощена постоянным множителем $a$ леммы, и это завершает доказательство. *Ч. т. д.*

Остальная часть этого раздела посвящена ограничению ожидаемого числа сравнений.

**Теорема 5.3 (сравнения в рандомизированном алгоритме `Quicksort`).** Для каждого входного массива длиной $n \geq 1$ ожидаемое количество сравнений между элементами входного массива в рандомизированном алгоритме `QuickSort` составляет не более $2(n-1)\ln n = O(n \log n)$.

По лемме 5.2 из теоремы 5.3 вытекает теорема 5.1 с другим постоянным множителем, скрытым в обозначении $O$-большое.

## 5.5.2. Схема декомпозиции

Основной метод (теорема 4.1) решал, каким будет время работы каждого алгоритма «разделяй и властвуй», который мы изучали до этого момента, но есть две причины, по которым он не применяется к рандомизированному алгоритму `QuickSort`. Во-первых, время работы алгоритма соответствует случайному рекуррентному соотношению или случайному дереву рекурсии, а основной метод работает с детерминированными рекуррентными соотношениями. Во-вторых, две подзадачи, которые решаются рекурсивно (элементы меньше опорного и элементы больше опорного), как правило, разного размера. Нам нужна новая идея[1].

Чтобы доказать теорему 5.3, мы будем следовать схеме декомпозиции, которая полезна для анализа математического ожидания усложненных случайных величин. Первый шаг — определить (возможно, усложненную) случайную

[1] Существуют обобщения основного метода, которые решают обе эти проблемы, но они несколько сложны и выходят за рамки этой книги.

величину $Y$, которая нас интересует; для нас это число $C$ сравнений между элементами входного массива, выполняемых рандомизированным алгоритмом `QuickSort`, как в теореме 5.3. Второй шаг — выразить $Y$ как сумму более простых случайных величин, в идеале индикаторных (то есть 0–1) случайных величин $X_1, ..., X_m$;

$$Y = \sum_{\ell=1}^{m} X_\ell .$$

Теперь мы находимся в епархии линейности математического ожидания, которое утверждает, что математическое ожидание суммы случайных величин равно сумме их математических ожиданий (теорема Б.1). Третий шаг схемы использует это свойство, чтобы сократить вычисление ожидания $Y$ до простых случайных величин:

$$\mathbf{E}\left[Y\right] = \mathbf{E}\left[\sum_{\ell=1}^{m} X_\ell\right] = \sum_{\ell=1}^{m} \mathbf{E}\left[X_\ell\right].$$

Когда $X_\ell$ — это случайные индикаторные величины, их математические ожидания особенно легко вычислить при помощи определения (Б.1):

$$\mathbf{E}\left[X_\ell\right] = \underbrace{0 \times \mathbf{Pr}\left[X_\ell = 0\right]}_{=0} + 1 \times \mathbf{Pr}\left[X_\ell = 1\right] = \mathbf{Pr}\left[X_\ell = 1\right].$$

На последнем шаге вычисляются математические ожидания простых случайных величин и суммируются результаты[1].

**СХЕМА ДЕКОМПОЗИЦИИ**

1. Определить случайную величину $Y$, которая вас интересует.
2. Выразить $Y$ как сумму индикаторных (то есть 0–1) случайных величин $X_1, \ldots, X_m$:

$$Y = \sum_{\ell=1}^{m} X_\ell .$$

[1] Рандомизированный анализ распределения нагрузки в разделе Б.6 является простым примером этой схемы. Мы также будем использовать эту схему, когда будем говорить о хеш-таблицах в части 2 этой серии книг.

3. Применить линейность математического ожидания:

$$\mathbf{E}[Y] = \sum_{\ell=1}^{m} \mathbf{Pr}[X_\ell = 1].$$

4. Вычислить все $\mathbf{Pr}[X_\ell = 1]$ и сложить результаты, чтобы получить $\mathbf{E}[Y]$.

## 5.5.3. Применение схемы

Чтобы применить схему декомпозиции к анализу рандомизированного алгоритма `QuickSort`, нам нужно разложить случайную величину $C$, которая нас действительно интересует, на более простые (в идеале 0–1) случайные величины. Ключевая идея состоит в том, чтобы разбить общее количество сравнений в соответствии с парой сравниваемых элементов входного массива.

Чтобы конкретизировать, обозначим через $z_i$ $i$-й наименьший элемент во входном массиве, так называемую *$i$-ю порядковую статистику*. Например, в массиве

| 6 | 8 | 9 | 2 |
|---|---|---|---|

$z_1$ относится к «2», $z_2$ — к «6», $z_3$ — к «8» и $z_4$ — к «9». Обратите внимание: $z_i$ обозначает не элемент в $i$-й позиции (неотсортированного) входного массива, а элемент в этой позиции отсортированной версии входного массива.

Для каждой пары индексов $i$, $j \in \{1, 2, ..., n\}$, массива, где $i < j$, мы определяем случайную величину $X_{ij}$ следующим образом:

> для каждого фиксированного варианта выбора из опорных элементов $\omega$ $X_{ij}(\omega)$ — это количество сравнений элементов $z_i$ и $z_j$ в алгоритме `QuickSort`, когда опорные элементы заданы $\omega$.

Для приведенного выше входного массива, например $X_{1,3}$, — это количество раз, когда алгоритм `QuickSort` сравнивает «2» с «8». Нас не интересуют $X_{ij}$ как таковые, за исключением того, что они складываются в случайную величину $C$, которая нас интересует.

Смысл этого определения заключается в реализации второго этапа схемы декомпозиции. Поскольку каждое сравнение включает в себя ровно одну пару элементов входного массива,

$$C(\omega) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}(\omega)$$

для каждого $\omega \in \Omega$. Причудливая двойная сумма в правой части — это просто итеративный обход всех пар $(i, j)$, где $i < j$, и это уравнение просто говорит о том, что $X_{ij}$ учитывает все сравнения, выполняемые алгоритмом `QuickSort`.

**ТЕСТОВОЕ ЗАДАНИЕ 5.3**

Задайте два разных элемента входного массива, скажем, $z_i$ и $z_j$.

Сколько раз можно сравнить $z_i$ и $z_j$ друг с другом во время исполнения `Quicksort`?

а) Ровно один раз.

б) 0 или 1 раз.

в) 0, 1 или 2 раза.

г) Возможно любое количество между 0 и $n - 1$.

(Решение и пояснение см. в разделе 5.5.6.)

Решение тестового задания 5.3 показывает, что все $X_{ij}$ являются индикаторными случайными величинами. Поэтому мы можем применить третий шаг нашей схемы декомпозиции для получения

$$\mathbf{E}[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbf{E}[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbf{Pr}[X_{ij} = 1]. \tag{5.1}$$

Чтобы вычислить то, что нас действительно интересует, то есть ожидаемое количество $\mathbf{E}[C]$ сравнений, все, что нам нужно сделать, это понять числа $\mathbf{Pr}[X_{ij} = 1]$! Каждое из этих чисел является вероятностью того, что некоторые $z_i$ и $z_j$ сравниваются друг с другом в какой-то момент в рандомизированном

алгоритме `QuickSort`, и следующим на повестке дня является вопрос о том, чтобы выявить эти числа[1].

## 5.5.4. Вычисление вероятностей сравнения

Для случая, когда есть вероятность того, что два элемента входного массива сравниваются в рандомизированном алгоритме `QuickSort`, существует убедительная формула.

**Лемма 5.4 (вероятность сравнения).** *Если $z_i$ и $z_j$ обозначают i-й и j-й наименьшие элементы входного массива, где $i < j$, то*

$$\mathbf{Pr}\left[z_i, z_j \text{ сравниваются в рандомизированном } \mathit{QuickSort}\right] = \frac{2}{j-i+1}.$$

Например, если $z_i$ и $z_j$ являются минимальным и максимальным элементами ($i = 1$ и $j = n$), то они сравниваются с вероятностью только $\frac{2}{n}$. Если нет элементов со значением между $z_i$ и $z_j$ ($j = i + 1$), тогда $z_i$ и $z_j$ всегда сравниваются друг с другом.

Зададим $z_i$ и $z_j$, где $i < j$, и рассмотрим опорный элемент $z_k$, выбранный в первом вызове алгоритма `QuickSort`. Каковы будут различные сценарии?

**ЧЕТЫРЕ СЦЕНАРИЯ QUICKSORT**

1. Выбранный опорный элемент меньше обоих элементов, $z_i$ и $z_j$ ($k < i$). Оба элемента, $z_i$ и $z_j$, передаются второму рекурсивному вызову.

[1] В разделе Б.5 большое значение имеет тот факт, что линейность математического ожидания применяется даже к случайным величинам, которые не являются независимыми (в случае, когда знание об одной случайной величине говорит вам что-то о других). Этот факт имеет решающее значение для нас здесь, так как $X_{ij}$ не являются независимыми. Например, если я скажу вам, что $X_{1n} = 1$, то вы знаете, что либо $z_1$, либо $z_n$ был выбран в качестве опорного элемента в самом внешнем вызове алгоритма `QuickSort` (почему?), и это, в свою очередь, делает гораздо более вероятным то, что случайная величина вида $X_{1j}$ или $X_{jn}$ также равна 1.

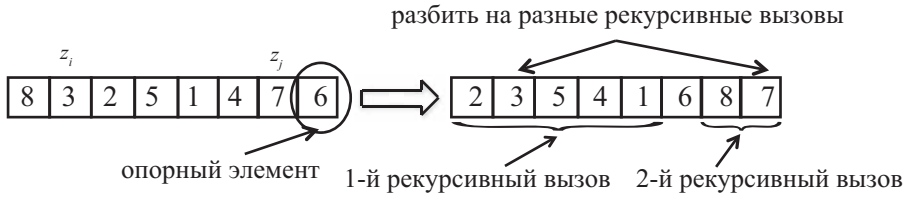
2. Выбранный опорный элемент больше обоих элементов, $z_i$ и $z_j$ ($k > j$). Оба элемента, $z_i$ и $z_j$, передаются первому рекурсивному вызову.
3. Выбранный опорный элемент между $z_i$ и $z_j$ ($i < k < j$). Элемент $z_i$ передается первому рекурсивному вызову и $z_j$ — второму.
4. Выбранный опорный элемент равняется либо $z_i$, либо $z_j$ ($k \in \{i, j\}$). Опорный элемент исключен из обоих рекурсивных вызовов; другой элемент передается первому (если $k = j$) либо второму (если $k = i$) рекурсивному вызову.

---

У нас тут происходят две вещи. Во-первых, вы помните, что каждое сравнение включает в себя текущий опорный элемент. Следовательно, $z_i$ и $z_j$ сравниваются в самом внешнем вызове алгоритма `QuickSort` тогда и только тогда, когда один из них выбран в качестве опорного элемента (сценарий 4). Во-вторых, в сценарии 3 не только $z_i$ и $z_j$ теперь не сравниваются, но они никогда не появятся вместе в одном рекурсивном вызове, и поэтому их нельзя сравнить в будущем. Например, в массиве

| 8 | 3 | 2 | 5 | 1 | 4 | 7 | 6 |
|---|---|---|---|---|---|---|---|

где $z_i = 3$ и $z_j = 7$, если какой-либо из элементов {4, 5, 6} выбран в качестве опорного элемента, то $z_i$ и $z_j$ отправляются в различные рекурсивные вызовы и никогда не сравниваются. Например, если выбрано «6», то картина будет следующей:




Сценарии 1 и 2 являются шаблоном удержания: $z_i$ и $z_j$ еще не сравнивались, но все еще возможно, что они будут сравниваться в будущем. Во время этого шаблона удержания $z_i$ и $z_j$ и все элементы $z_{i+1}$, ..., $z_{j-1}$ со значениями между $z_i$ и $z_j$ существуют параллельно и продолжают передаваться тому же рекурсив-

ному вызову. В конце концов, их коллективное передвижение прерывается рекурсивным вызовом алгоритма `QuickSort`, в котором один из элементов $z_i$, $z_{i+1}$, ..., $z_{j-1}$, $z_j$ выбирается в качестве опорного элемента, вызывая либо сценарий 3, либо сценарий 4[1].

Прокрутив вперед к этому рекурсивному вызову, где как раз все происходит, сценарий 4 (и сравнение между $z_i$ и $z_j$) срабатывает, если выбранным опорным элементом является $z_i$ либо $z_j$, в то время как сценарий 3 (и отсутствие такого сравнения) срабатывает, если в качестве опорного элемента выбран любой из $z_{i+1}$, ..., $z_{j-1}$. Поэтому есть два плохих случая ($z_i$ и $z_j$) вариантов $j-i+1$ ($z_i$, $z_{i+1}$, ..., $z_{j-1}$, $z_j$). Поскольку рандомизированный алгоритм `QuickSort` всегда выбирает опорные элементы равномерно случайным образом, в силу симметрии *каждый элемент $\{z_i, z_{i+1}, ..., z_{j-1}, z_j\}$ с равной вероятностью будет первым опорным элементом, выбранным из множества*. Собирая все вместе, вероятность

**Pr**[$z_i$, $z_j$ сравниваются в некоей точке в рандомизированном `QuickSort`]

совпадает с

**Pr**[$z_i$ или $z_j$ выбирается в качестве опорной до любого из $z_{i+1},\ldots, z_{j-1}$],

которая равняется

$$\frac{\text{количество плохих случаев}}{\text{общее количество вариантов}} = \frac{2}{j-i+1}.$$

Это завершает доказательство леммы 5.4. *Ч. т. д.*

Возвращаясь к нашей формуле (5.1) для ожидаемого числа сравнений, делаемых рандомизированным алгоритмом `QuickSort`, мы получаем потрясающе точное выражение:

$$\mathbf{E}[C] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \mathbf{Pr}\left[X_{ij} = 1\right] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{2}{j-i+1}. \tag{5.2}$$

Чтобы доказать теорему 5.3, осталось показать только то, что правая часть (5.2) на самом деле равняется $O(n \log n)$.

---

[1] Если нет других вариантов, предыдущие рекурсивные вызовы в конечном итоге сводят подмассив только до элементов $\{z_i, z_{i+1}, ..., z_{j-1}, z_j\}$.

### 5.5.5. Заключительные вычисления

Легко доказать верхнюю границу $O(n^2)$ в правой части (5.2): в двойной сумме имеется не более $n^2$ членов, и каждый из них имеет значение не более $\frac{1}{2}$ (достигается, когда $j = i + 1$). Но мы ищем гораздо лучшую верхнюю границу $O(n \log n)$, и мы должны быть умнее, чтобы получить ее, использовав тот факт, что большинство из квадратично многих членов намного меньше $\frac{1}{2}$.

Рассмотрим одну из внутренних сумм в (5.2) для фиксированного значения $i$:

$$\sum_{j=i+1}^{n} \frac{2}{j-i+1} = 2 \times \underbrace{\left( \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n-i+1} \right)}_{n-i \text{ членов}}.$$

Мы можем ограничить каждую из этих сумм сверху наибольшей такой суммой, которая возникает при $i = 1$:

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \le \sum_{i=1}^{n-1} \underbrace{\sum_{j=2}^{n} \frac{2}{j}}_{\text{независимы от } i} = 2(n-1) \times \sum_{j=2}^{n} \frac{1}{j}. \tag{5.3}$$

Насколько большим является $\sum_{j=2}^{n} 1/j$? Давайте взглянем на рис. 5.1.

Рассматривая члены суммы $\sum_{j=2}^{n} 1/j$ в качестве прямоугольников на плоскости, как на рис. 5.1, мы видим, что можем ограничить эту сумму сверху площадью под кривой $f(x) = \frac{1}{x}$ между точками 1 и $n$, также известной как интеграл $\int_1^n \frac{dx}{x}$. Если вы немного помните исчисление, то распознаете решение этого интеграла как натуральный логарифм $\ln x$ (то есть $\ln x$ — это функция, чья производная равняется $\frac{1}{x}$):

$$\sum_{j=2}^{n} \frac{1}{j} \le \int_1^n \frac{1}{x}\, dx = \ln x \Big|_1^n = \ln n - \underbrace{\ln 1}_{=0} = \ln n. \tag{5.4}$$

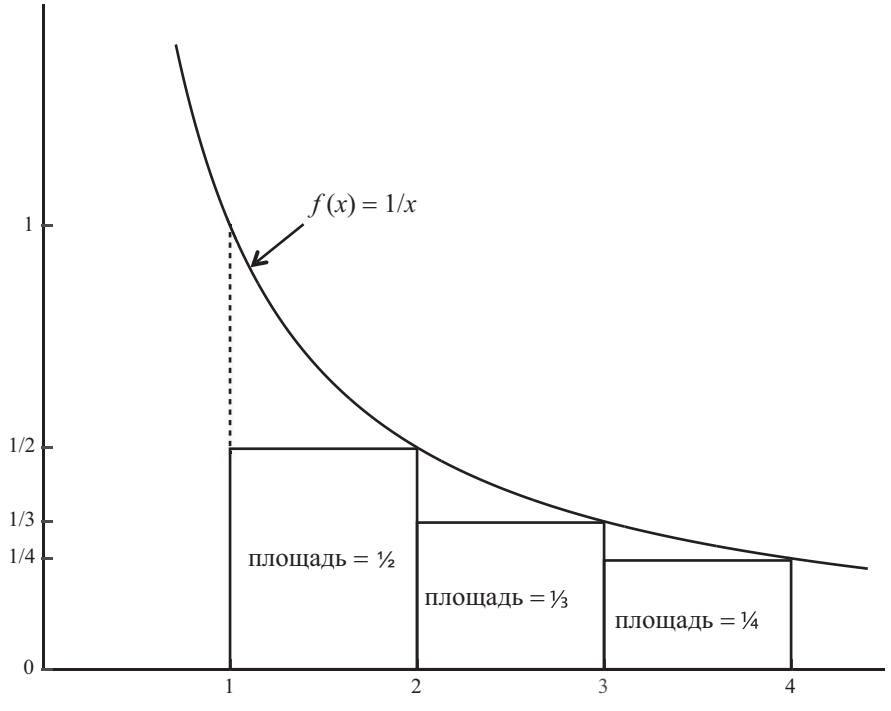



**Рис. 5.1.** Каждое слагаемое суммы $\sum_{j=2}^{n} 1/j$ можно идентифицировать по прямоугольнику шириной 1 (между $x$-координатами $j - 1$ и $j$) и высотой $1/j$ (между $y$-координатами 0 и $1/j$). График функции $f(x) = 1/x$ касается северо-восточного угла каждого из этих прямоугольников, и поэтому площадь под кривой (то есть интеграл) является верхней границей площади прямоугольников

Соединив в цепь уравнения и неравенства в (5.2)–(5.4), мы имеем

$$\mathbf{E}[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \le 2(n-1) \times \sum_{j=2}^{n} \frac{1}{j} \le 2(n-1) \ln n.$$

Следовательно, ожидаемое число сравнений, выполняемых рандомизированным алгоритмом `QuickSort`, а также его ожидаемое время работы по лемме 5.2 действительно составляет $O(n \log n)$! *Ч. т. д.*

### 5.5.6. Решение тестового задания 5.3

**Правильный ответ: (б).** Если в качестве опорного элемента в самом внешнем вызове алгоритма `QuickSort` выбраны $z_i$ или $z_j$, то $z_i$ и $z_j$ будут сравниваться при первом вызове подпрограммы `Partition`. (Помните, что опорный элемент сравнивается с любым другим элементом в подмассиве.) Если $i$ и $j$ различаются более чем на 1, также возможно, что $z_i$ и $z_j$ никогда не будут сравниваться вообще (см. также раздел 5.5.4). Например, минимальный и максимальный элементы не будут сравниваться друг с другом, если один из них не выбран в качестве опорного элемента в самом внешнем рекурсивном вызове (вы видите почему?).

Наконец, как и следовало ожидать от хорошего алгоритма сортировки, $z_i$ и $z_j$ никогда не будут сравниваться друг с другом более одного раза (что было бы избыточным). Каждое сравнение включает в себя текущий опорный элемент, поэтому при первом сравнении $z_i$ и $z_j$ в каком-то вызове (если вообще таковой будет) один из них должен быть опорным элементом. Поскольку опорный элемент исключен из всех будущих рекурсивных вызовов, $z_i$ и $z_j$ больше никогда не появятся вместе в одном рекурсивном вызове (не говоря уже об их сравнении друг с другом).

## *5.6. Сортировка требует $\Omega(n \log n)$ сравнений

Существует ли алгоритм сортировки быстрее, чем `MergeSort` и `QuickSort`, с временем работы большим, чем $\Theta(n \log n)$? Интуитивно понятно, что алгоритм должен просматривать каждый входной элемент один раз, но из этого следует только линейная нижняя граница $\Omega(n)$. Приводимый ниже факультативный раздел показывает, что лучшего варианта для сортировки быть не может — алгоритмы `MergeSort` и `QuickSort` достигают наибольшего асимптотического времени работы.

## 5.6.1. Алгоритмы сортировки на основе сравнения

Приведем формулировку нижней границы $\Omega(n \log n)$.

**Теорема 5.5 (нижняя граница сортировки).** *Существует константа $c > 0$ такая, что для каждого $n \geq 1$ любой алгоритм сортировки на основе сравнения выполняет как минимум $c \times n \log_2 n$ операций на некотором входном массиве длиной $n$.*

Под «алгоритмом сортировки на основе сравнения» мы подразумеваем алгоритм, который обращается к входному массиву только через сравнения между парами элементов и никогда напрямую не обращается к значению элемента. Алгоритмы сортировки на основе сравнения универсальны и не принимают никаких допущений о природе входных элементов, кроме того, что они принадлежат к некоторому полностью упорядоченному множеству. Алгоритм сортировки на основе сравнения можно рассматривать как взаимодействие с входным массивом через API, который поддерживает только одну операцию: при наличии двух индексов, $i$ и $j$ (между 1 и длиной массива $n$), операция возвращает 1, если $i$-й элемент меньше $j$-го элемента, и 0 в противном случае[1].

Например, алгоритм `MergeSort` — это алгоритм сортировки на основе сравнения, ему все равно, сортирует ли он целые числа или фрукты (при условии, что мы договорились о полном упорядочении всех возможных фруктов, например, в алфавитном порядке)[2]. То же касается и алгоритмов `SelectionSort`, `InsertionSort`, `BubbleSort` и `QuickSort`.

---

[1] Например, принятая по умолчанию процедура сортировки в операционной системе UNIX работает именно таким образом. Единственным требованием является пользовательская функция для сравнения пар элементов входного массива.

[2] Для аналогии сравните головоломки Судоку и KenKen. Головоломкам Судоку нужно только понятие равенства между разными объектами, и они будут предельно понятны, если числа 1–9 заменить на девять разных фруктов. Головоломки KenKen используют арифметику, и, следовательно, нужны числа — какова сумма плуота и мангостана?

### 5.6.2. Более быстрая сортировка при более строгих допущениях

Лучший способ понять сортировку на основе сравнения — взглянуть на несколько примеров. Приведем три алгоритма сортировки, которые принимают допущения о природе входных данных, но в обмен превышают нижнюю границу $\Omega(n \log n)$ в теореме 5.5[1].

Блочная сортировка `BucketSort`. Алгоритм `BucketSort` полезен на практике для числовых данных, в особенности если они равномерно распределены по известному диапазону. Например, предположим, что входной массив содержит $n$ элементов от 0 до 1, которые приблизительно равномерно распределены. В уме мы делим интервал [0, 1] на $n$ «корзин», первая зарезервирована для входных элементов между 0 и $\frac{1}{n}$, вторая — для элементов между $\frac{1}{n}$ и $\frac{2}{n}$, и так далее. Первый шаг алгоритма `BucketSort` делает один линейный проход по входному массиву и помещает каждый элемент в его корзину. *Этот шаг не на основе сравнения* — алгоритм `BucketSort` смотрит на фактическое значение входного элемента, чтобы определить, к какой корзине он принадлежит. При этом важно, равняется ли значение входного элемента .17 или .27, даже если мы соблюдаем относительный порядок следования элементов.

Если элементы приблизительно равномерно распределены, наполнение каждой корзины невелико. Второй шаг алгоритма сортирует элементы внутри каждой корзины по отдельности (например, с помощью алгоритма `InsertionSort`). При условии, что в каждой корзине мало элементов, этот шаг также выполняется за линейное время (с постоянным количеством операций, выполняемых в каждой корзине). Наконец, отсортированные списки разных корзин объединяются от первого до последнего. Этот шаг также выполняется за линейное время. Мы заключаем, что линейная сортировка возможна при строгом допущении о природе входных данных.

---

[1] Для более подробного ознакомления см., например, «Введение в алгоритмы» (третье издание), Томас Х. Кормен, Чарлз Е. Лейсерсон, Рональд Л. Ривест и Клиффорд Стейн (Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, MIT Press, 2009).

Сортировка подсчетом `CountingSort`. Алгоритм `CountingSort` является вариацией той же идеи. Здесь мы принимаем допущение, что существует только $k$ разных возможных значений каждого входного элемента (известных заранее), таких как целые числа $\{1, 2, ..., k\}$. Алгоритм устанавливает $k$ корзин, по одной для каждого возможного значения, и за один проход по входному массиву помещает каждый элемент в соответствующую корзину. Выходной массив представляет собой простое объединение этих корзин (по порядку). Алгоритм `CountingSort` выполняется за линейное время при $k = O(n)$, где $n$ — это длина входного массива. Как и алгоритм `BucketSort`, этот алгоритм не основан на сравнениях.

Поразрядная сортировка `RadixSort`. Алгоритм `RadixSort` является расширением алгоритма `CountingSort`, который корректно обрабатывает $n$-элементные целочисленные входные массивы с достаточно большими числами, представленными в двоичном виде (строкой нулей и единиц или «битами»). Первый шаг алгоритма `RadixSort` рассматривает только блок из $\log_2 n$ наименее значимых битов входных чисел и сортирует их соответствующим образом. Поскольку $\log_2 n$ бит могут кодировать только $n$ разных значений, соответствующих числам 0, 1, 2, ..., $n - 1$, записанным в двоичном виде, алгоритм `CountingSort` может быть использован для реализации этого шага за линейное время. Алгоритм `RadixSort` затем повторно сортирует все элементы, используя блок следующих наименее значащих $\log_2 n$ бит, и так далее до тех пор, пока все биты входных данных не будут обработаны. Чтобы этот алгоритм сортировал правильно, важно реализовать подпрограмму `CountingSort` таким образом, чтобы она была *стабильной*, учитывая, что она сохраняет относительный порядок разных элементов с одинаковым значением[1]. Алгоритм `RadixSort` выполняется за линейное время при условии, что входной массив содержит только целые числа от 0 до $n^k$ для некоторой константы $k$.

Эти три алгоритма сортировки демонстрируют, как дополнительные допущения о природе входных данных (например, не слишком большие целые числа) задействуют технические приемы за рамками сравнений (например, разделение на блоки или корзины), и алгоритмы, которые быстрее, чем время $\Theta(n \log n)$. Теорема 5.5 констатирует, что такие улучшения невозможны для

[1] Не все алгоритмы сортировки стабильны. Например, алгоритм `QuickSort` не является стабильным алгоритмом сортировки (вы видите почему?).

универсальных основанных на сравнении алгоритмов сортировки. Давайте посмотрим почему.

## 5.6.3. Доказательство теоремы 5.5

Зададим произвольный детерминированный алгоритм сортировки на основе сравнения[1]. Выход алгоритма можно представить как перестановку (то есть переупорядочение, пермутацию) чисел 1, 2, ..., $n$, где $i$-й элемент выходных данных указывает на позицию $i$-го наименьшего элемента во входном массиве. Например, если входной массив будет следующим:

| 6 | 8 | 9 | 2 |
|---|---|---|---|

то выход правильного алгоритма сортировки может интерпретироваться как массив индексов

| 4 | 1 | 2 | 3 |
|---|---|---|---|

Существует $n! = n \times (n - 1) \cdot \cdot \cdot 2 \times 1$ возможностей правильного выходного массива[2]. Для каждого входного массива существует уникальный правильный выходной массив.

**Лемма 5.6.** *Если алгоритм сортировки на основе сравнения никогда не делает более k сравнений для любого входного массива длиной n, то он генерирует не более* $2^k$ *разных выходных массивов*.

*Доказательство*: мы можем разбить операции, выполняемые алгоритмом, на фазы, где фаза $i$ включает работу, выполняемую алгоритмом после его $(i - 1)$-го сравнения и вплоть до его $i$-го сравнения включительно. (Между сравнениями алгоритм может делать все, что угодно, — ведение учета,

---

[1] Аналогичные аргументы применимы к рандомизированным алгоритмам сортировки на основе сравнения, и ни от одного из таких алгоритмов не ожидается время работы больше, чем $\Theta(n \log n)$.

[2] Существует $n$ вариантов позиций наименьшего элемента во входном массиве, $n - 1$ оставшихся вариантов позиций второго наименьшего элемента и так далее.

выяснение, какое запросить следующее сравнение, и так далее — при условии, что он не обращается к входному массиву.) Конкретные операции, выполняемые в фазе $i$, могут зависеть только от результатов первых $i - 1$ сравнений, поскольку это единственная входная информация, которой алгоритм обладает. Например, эти операции не зависят от фактического значения элемента, участвующего в одном из этих сравнений. Выходной массив в конце алгоритма зависит только от результатов всех сравнений. Если алгоритм никогда не делает более $k$ сравнений, то существует не более $2^k$ разных выполнений алгоритма и, следовательно, не более $2^k$ разных выходных массивов[1]. *Ч. т. д.*

Правильный алгоритм сортировки должен быть способен производить любой из $n!$ возможных правильных выходных массивов. По лемме 5.6, если $k$ есть максимальное число сравнений, выполняемых на $n$-элементных входных массивах, то

$$2^k \geq \underbrace{n!}_{n\times(n-1)\ldots2\times1} \geq \left(\frac{n}{2}\right)^{n/2},$$

где мы использовали тот факт, что все первые $n/2$ членов $n\ (n-1)\cdot\ \cdot\ \cdot 2\times 1$ равняются не менее $\frac{n}{2}$. Взятие логарифма по основанию 2 обеих частей показывает, что

$$k \geq \frac{n}{2}\log_2\left(\frac{n}{2}\right) = \Omega(n\ \log n).$$

Эта нижняя граница применяется к произвольным основанным на сравнении алгоритмам сортировки, завершая доказательство теоремы 5.5. *Ч. т. д.*

[1] Для первого сравнения существует два возможных исхода; каким бы ни был исход и последующее второе сравнение, он также имеет два возможных исхода, и так далее.

### ВЫВОДЫ

- Знаменитый алгоритм `QuickSort` состоит из трех высокоуровневых шагов: во-первых, он выбирает один элемент $p$ из входного массива, который действует как «опорный элемент»; во-вторых, его подпрограмма `Partition` перегруппировывает массив так, чтобы элементы меньше и больше $p$ шли соответственно перед ним и после него; в-третьих, он рекурсивно сортирует два подмассива по обе стороны от опорного элемента.
- Подпрограмма `Partition` может быть реализована так, что она будет выполняться за линейное время и на том же месте, с учетом незначительного потребления дополнительной оперативной памяти. Как следствие, алгоритм `QuickSort` тоже выполняется на том же месте.
- Правильность алгоритма `Quicksort` не зависит от того, как опорные элементы выбираются, но его время работы от этого зависит.
- В наихудшем сценарии время работы составляет $\Theta(n^2)$, где $n$ — это длина входного массива. Это происходит, когда входной массив уже отсортирован и первый элемент всегда используется в качестве опорного элемента. В наилучшем сценарии время работы составляет $\Theta(n \log n)$. Это происходит, когда в качестве опорного всегда используется медиана элемента.
- В рандомизированном алгоритме `QuickSort` опорный элемент всегда выбирается равномерно случайным образом. Его время работы может быть где угодно от $\Theta(n \log n)$ до $\Theta(n^2)$ в зависимости от его случайных подбрасываний монеты.
- Среднее время работы рандомизированного алгоритма `QuickSort` составляет $\Theta(n \log n)$, только на небольшой постоянный множитель меньше, чем его наибольшее время работы.
- В интуитивном плане выбор случайного опорного элемента является хорошей идеей, потому что имеется 50 %-ный шанс получить 25−75 % или лучшее разбиение входного массива.
- Формальный анализ использует схему декомпозиции, чтобы выразить усложненную случайную величину как сумму случайных

величин 0–1, а затем применить линейность математического ожидания.

- Ключевое понимание заключается в том, что $i$-й и $j$-й наименьшие элементы входного массива сравниваются в алгоритме `QuickSort` тогда и только тогда, когда один из них выбран в качестве опорного до того, как элемент со значением строго между ними выбран в качестве опорного.
- Алгоритм сортировки на основе сравнения — это универсальный алгоритм, который обращается к входному массиву только путем сравнения пар элементов и никогда непосредственно не использует значение элемента.
- Ни один алгоритм сортировки на основе сравнения не имеет наихудшего асимптотического времени работы больше, чем $O(n \log n)$.

## Задачи на закрепление материала

**Задача 5.1.** Вспомните подпрограмму `Partition`, используемую алгоритмом `QuickSort` (раздел 5.2). Вам говорят, что следующий ниже массив только что был разделен вокруг некоторого опорного элемента:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 3 | 1 | 2 | 4 | 5 | 8 | 7 | 6 | 9 |

Какой из элементов мог быть опорным элементом? (Перечислите все, которые применимы; может быть более одной возможности.)

**Задача 5.2.** Пусть $\alpha$ есть некоторая константа, не зависящая от длины $n$ входного массива, строго между 0 и $\frac{1}{2}$. Какова вероятность того, что при случайно выбранном опорном элементе подпрограмма `Partition` создаст разбиение, в котором размер обеих результирующих подзадач будет минимум в $\alpha$ раз больше размера исходного массива?

а) $\alpha$.

б) $1 - \alpha$.

в) $1 - 2\alpha$.

г) $2 - 2\alpha$.

**Задача 5.3.** Пусть $\alpha$ есть некоторая константа, не зависящая от длины $n$ входного массива, строго между 0 и $\frac{1}{2}$. Допустим, что вы достигаете приблизительно сбалансированных разбиений из предыдущей задачи в каждом рекурсивном вызове — поэтому всякий раз, когда рекурсивному вызову передается массив длиной $k$, каждому из двух рекурсивных вызовов передается подмассив длиной между $\alpha k$ и $(1 - \alpha)k$. Сколько последовательных рекурсивных вызовов может произойти перед запуском базового случая? Это эквивалентно вопросу: какие уровни дерева рекурсии алгоритма могут содержать листья? Выразите свой ответ в виде диапазона возможных чисел $d$, от минимального до максимального числа рекурсивных вызовов, которые могут потребоваться. [Подсказка: формула, которая связывает логарифмические функции с разными основаниями, следующая: $\log_b n = \frac{\ln n}{\ln b}$.]

а) $0 \le d \le -\frac{\ln n}{\ln \alpha}$.

б) $-\frac{\ln n}{\ln \alpha} \le d \le -\frac{\ln n}{\ln(1-\alpha)}$.

в) $-\frac{\ln n}{\ln(1-\alpha)} \le d \le -\frac{\ln n}{\ln \alpha}$.

г) $-\frac{\ln n}{\ln(1-2\alpha)} \le d \le -\frac{\ln n}{\ln(1-\alpha)}$.

**Задача 5.4.** Определите глубину рекурсии алгоритма `QuickSort` как максимальное число последовательных рекурсивных вызовов, которые он делает, прежде чем попасть в базовый случай, — или, что то же самое, наибольший уровень его дерева рекурсии. В рандомизированном алгоритме `QuickSort` глубина рекурсии является случайной величиной в зависимости от выбранных опорных элементов. Какова минимальная и максимальная глубина рекурсии рандомизированного алгоритма `QuickSort`?

а) Минимум: $\Theta(1)$; максимум: $\Theta(n)$.

б) Минимум: $\Theta(\log n)$; максимум: $\Theta(n)$.

в) Минимум: $\Theta(1)$; максимум: $\Theta(n \log n)$.

г) Минимум: $\Theta(\sqrt{n})$; максимум: $\Theta(n)$.

## Задача повышенной сложности

**Задача 5.5.** Расширьте нижнюю границу $\Omega(n \log n)$ в разделе 5.6, чтобы также применить к ожидаемому времени выполнения рандомизированных алгоритмов сортировки на основе сравнения.

## Задачи по программированию

**Задача 5.6.** Реализуйте алгоритм `QuickSort` на вашем любимом языке программирования. Поэкспериментируйте с производительностью различных способов выбора опорного элемента.

Один из подходов заключается в отслеживании количества сравнений между элементами входного массива, выполняемых алгоритмом `QuickSort`[1]. Для нескольких разных входных массивов определите количество сравнений, выполняемых следующими ниже реализациями подпрограммы `ChoosePivot`:

1. В качестве опорного всегда использует первый элемент.
2. В качестве опорного всегда использует последний элемент.
3. В качестве опорного использует случайный элемент. (В этом случае необходимо выполнить алгоритм 10 раз с заданным входным массивом и усреднить результаты.)
4. В качестве опорного элемента использует *медиану из трех*. Цель этого правила состоит в том, чтобы сделать немного дополнительной работы, чтобы получить гораздо лучшую производительность на входных

[1] Нет необходимости подсчитывать сравнения по одному. При рекурсивном вызове с подмассивом длиной $m$ вы просто можете добавить $m - 1$ в ваш накопительный итог сравнений. (Напомним, что в этом рекурсивном вызове опорный элемент сравнивается с каждым из других $m - 1$ элементов в подмассиве.)

массивах, которые почти отсортированы или отсортированы в обратном порядке.

Если говорить более подробно, эта реализация подпрограммы `ChoosePivot` рассматривает первый, средний и конечный элементы заданного массива. (Для массива с четной длиной $2^k$ использовать $k$-й элемент для «середины».) Затем она определяет, какой из этих трех элементов является медианой (то есть значение какого из них находится между двумя другими), и возвращает этот элемент как опорный[1].

Например, с входным массивом

| 8 | 3 | 2 | 5 | 1 | 4 | 7 | 6 |
|---|---|---|---|---|---|---|---|

подпрограмма будет рассматривать первый (8), средний (5) и последний (6) элементы. Она вернет 6, медиану множества {5, 6, 8}, в качестве опорного элемента.

См. www.algorithmsilluminated.org для тестовых сценариев и наборов данных для задач.

[1] Тщательный анализ будет отслеживать сопоставления, выполняемые при определении медианы трех элементов-кандидатов, в дополнение к сопоставлениям, выполняемым в вызовах подпрограммы `Partition`.

# 6 Линейный выбор

В этой главе изучается задача выбора, целью которой является определение $i$-го наименьшего элемента неотсортированного массива. Эта задача легко решается за время $O(n \log n)$ при помощи сортировки, но мы можем добиться лучшего. Раздел 6.1 описывает чрезвычайно практичный рандомизированный алгоритм, очень похожий на рандомизированный алгоритм `QuickSort`, который в среднем работает в течение линейного времени. В разделе 6.2 представлен интересный анализ этого алгоритма — классный способ подумать о прогрессе, который обеспечивает алгоритм, с позиций простого эксперимента по подбрасыванию монеты, а затем линейность математического ожидания (да, снова она...) ставит печать под сделкой.

Читатели, которым важна теория, могут задаться вопросом, можно ли решить задачу выбора за линейное время, не прибегая к рандомизации. В разделе 6.3 описывается известный детерминированный алгоритм для этой задачи, среди авторов которого больше тех, кто был удостоен премии Тьюринга, чем у любого другого известного мне алгоритма. Он детерминирован (то есть не допускает рандомизации) и основан на гениальной идее «медианы медиан», которая обеспечивает хорошие варианты опорных элементов. Раздел 6.4 доказывает линейную временную границу, что не так-то просто!

В этой главе предполагается, что вы помните подпрограмму `Partition` из раздела 5.2, которая разделяет массив вокруг опорного элемента за линейное время, а также что у вас есть интуитивное понимание того, что делает опорный элемент хорошим или плохим (раздел 5.3).

# 6.1. Алгоритм RSelect

## 6.1.1. Задача выбора

В задаче выбора входные данные такие же, как и в задаче сортировки: массив из $n$ чисел вместе с целым числом $i \in \{1, 2, ..., n\}$. Цель состоит в том, чтобы определить *$i$-ю порядковую статистику* — $i$-е наименьшее значение в массиве.

**ЗАДАЧА: ВЫБОР**

**Вход**: массив из $n$ чисел в произвольном порядке и целое число $i \in \{1, 2, ..., n\}$.

**Выход**: $i$-е наименьшее значение в массиве $A$.

Как обычно, для простоты мы принимаем допущение, что входной массив имеет разные элементы, без дубликатов.

Например, если дан входной массив

| 6 | 8 | 9 | 2 |
|---|---|---|---|

и значение $i$ равняется 2, то правильный результат будет равен 6. Если бы $i$ равнялось 3, то правильный результат был бы 8, и так далее.

При $i = 1$ задача выбора — это просто задача по вычислению минимального элемента массива. В течение линейного времени легко сделать один проход массива и запомнить наименьший встретившийся элемент. По аналогии случай нахождения максимального элемента ($i = n$) также прост. Но как насчет значений $i$ в середине? Например, что, если мы хотим вычислить срединный элемент — *медиану* — массива?

Если быть точным, для массива с нечетной длиной $n$ медиана является $i$-й порядковой статистикой, где $i = (n + 1)/2$. Что касается массива с четной длиной $n$, то давайте согласимся определять медиану как меньшую из двух возможностей, что соответствует $i = \frac{n}{2}$[1].

[1] Зачем вычислять медиану элемента массива? В конце концов, *среднее значение* (то есть среднее арифметическое) достаточно легко вычислить за линейное время — просто суммировать все элементы массива за один проход и разделить на $n$. Одной из причин является вычисление сводной статистики массива, которая более надежна, чем среднее значение. Например, один сильно поврежденный элемент, такой как ошибка ввода данных, может совершенно испортить среднее значение массива, но, как правило, мало повлияет на медиану.

### 6.1.2. Сведение к задаче сортировки

Мы уже знаем быстрый алгоритм для задачи выбора, который опирается на наши алгоритмы быстрой сортировки.

**СВЕДЕНИЕ ВЫБОРА К СОРТИРОВКЕ**

**Вход**: массив $A$ из $n$ разных чисел и целое число $i \in \{1, 2, ..., n\}$.

**Выход**: $i$-я порядковая статистика массива $A$.

```
B := MergeSort(A)
return B[i]
```

Закончив сортировку входного массива мы, безусловно, знаем, где найти $i$-й наименьший элемент, — он болтается в $i$-й позиции отсортированного массива. Поскольку алгоритм `MergeSort` выполняется за время $O(n \log n)$ (теорема 1.2), то и этот двухступенчатый алгоритм — тоже[1].

Но помните мантру любого проектировщика алгоритмов, которая заслуживает уважения: *можно ли добиться лучшего*? Можем ли мы разработать алгоритм для задачи выбора, который будет еще быстрее, чем время $O(n \log n)$? Лучшее, на что мы можем надеяться, — это линейное время ($O(n)$), если мы не будем уделять время даже тому, чтобы просматривать каждый элемент в массиве, то нет никакой надежды на то, что получится всегда правильно идентифицировать, скажем, минимальный элемент. Мы также знаем из теоремы 5.5, что любой алгоритм, который использует подпрограмму сортировки, застревает в наихудшем времени работы $\Omega(n \log n)$[2]. Поэтому, если мы *сможем* получить большее, чем $O(n \log n)$, время работы для задачи выбора, мы докажем, что *выбор принципиально проще сортировки*. Для этого требуется изобретательность — опора на наши алгоритмы сортировки не сработает.

---

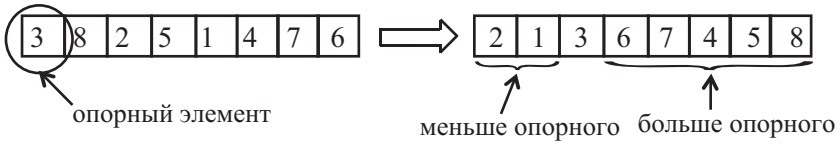
[1] Специалист в области computer science назвал бы это сведением или редукцией задачи выбора к задаче сортировки. Редукция освобождает вас от разработки нового алгоритма с нуля и вместо этого позволяет опираться на существующие алгоритмы. В дополнение к их практической полезности редукции являются чрезвычайно фундаментальной концепцией в информатике, и мы подробно обсудим их в части 4 этой серии книг.

[2] Предполагается, что мы ограничиваемся алгоритмами сортировки на основе сравнения, как в разделе 5.6.

### 6.1.3. Подход «разделяй и властвуй»

Рандомизированный линейный алгоритм выбора `RSelect` следует шаблону, который оказался очень успешным в рандомизированном алгоритме `QuickSort`: выбирается случайный опорный элемент, входной массив разделяется вокруг опорного элемента, и выполняется соответствующая рекурсия. Следующий вопрос на повестке дня — разобраться в соответствующей рекурсии для задачи выбора.

Вспомните, что делает подпрограмма `Partition` в разделе 5.2: при наличии массива и вариантов выбора опорного элемента она перегруппировывает элементы массива так, чтобы все, что меньше и больше опорного элемента, появлялось соответственно до и после него.




Таким образом, опорный элемент в итоге оказывается на своем законном месте, то есть после всех элементов, которые меньше него и перед всеми элементами, которые больше него.

Алгоритм `QuickSort` рекурсивно отсортировал подмассив элементов, которые меньше опорного элемента, а также подмассив элементов, которые больше опорного. Какой аналог для задачи выбора?

**ТЕСТОВОЕ ЗАДАНИЕ 6.1**

Предположим, что мы ищем 5-ю порядковую статистику во входном массиве из 10 элементов. Предположим, что после разделения массива опорный элемент в итоге оказывается в третьей позиции. На какой стороне от опорного элемента мы должны выполнять рекурсию и какую порядковую статистику мы должны искать?

а) 3-ю порядковую статистику на левой стороне от опорного.

б) 2-ю порядковую статистику на правой стороне от опорного.

в) 5-ю порядковую статистику на правой стороне от опорного.

г) Возможно, нам понадобится рекурсия как с левой, так и с правой стороны от опорного элемента.

(Решение и пояснение см. в разделе 6.1.6.)

## 6.1.4. Псевдокод для алгоритма RSelect

Наш псевдокод для алгоритма `RSelect` соответствует высокоуровневому описанию алгоритма `QuickSort` в разделе 5.1 с двумя изменениями. Во-первых, мы должны придерживаться использования случайных опорных элементов, а не иметь общую подпрограмму `ChoosePivot`. Во-вторых, алгоритм `RSelect` делает только один рекурсивный вызов, в то время как алгоритм `QuickSort` делает два. В основном из-за этого различия есть повод надеяться, что алгоритм `RSelect` может быть даже быстрее, чем рандомизированный алгоритм `QuickSort`.

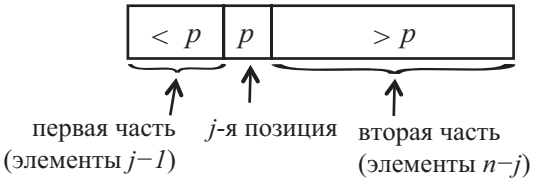
**RSELECT**

**Вход**: массив $A$ из $n \geq 1$ разных чисел и целое число $i \in \{1, 2, ..., n\}$.

**Выход**: $i$-я порядковая статистика массива $A$.

```
if n = 1 then                                  // базовый случай
    return A[1]
выбрать опорный элемент p равномерно случайным образом из A
разделить A вокруг p
j := позиция p в разделенном массиве
if j = i then                                  // вам посчастливилось!
    return p
else if j > i then
    return RSelect(first part of A, i)
else                                           // j < i
    return RSelect(second part of A, i – j)
```

Разделение входного массива вокруг опорного элемента $p$ разбивает массив на три части, что приводит к трем случаям в алгоритме `RSelect`:




Вследствие того, что опорный элемент $p$ занимает свое законное место в разделенном массиве, если он находится в $j$-й позиции, он должен быть $j$-й порядковой статистикой. Если по счастливой случайности алгоритм искал $j$-ю порядковую статистику (то есть $i = j$), то дело сделано. Если же алгоритм ищет меньшее число (то есть $i < j$), оно должно принадлежать первой части разделенного массива. В этом случае рекурсивный обход отбрасывает только те элементы, которые больше $j$-й (и, следовательно, $i$-й) порядковой статистики, поэтому алгоритм по-прежнему ищет $i$-й наименьший элемент среди тех, которые находятся в первом подмассиве. В последнем случае ($i > j$) алгоритм ищет число, большее, чем опорный элемент, и рекурсия имитирует решение тестового задания 6.1. Алгоритм выполняет рекурсивный обход второй части разделенного массива, исключая из дальнейшего рассмотрения опорный элемент и $j - 1$ элементы меньше него. Поскольку алгоритм первоначально искал $i$-й наименьший элемент, теперь он ищет $(i - j)$-й наименьший элемент среди оставшихся.

### 6.1.5. Время работы алгоритма RSelect

Как и рандомизированный алгоритм `QuickSort`, время работы алгоритма `RSelect` зависит от опорных элементов, которые он выбирает. Что может произойти в худшем случае?

**ТЕСТОВОЕ ЗАДАНИЕ 6.2**

Каково время работы алгоритма `RSelect`, если опорные элементы всегда выбираются наихудшим образом?

а) $\Theta(n)$.

б) $\Theta(n \log n)$.

в) $\Theta(n^2)$.

г) $\Theta(2^n)$.

(Решение и пояснение см. в разделе 6.1.6.)

Теперь мы знаем, что алгоритм `RSelect` не работает в течение линейного времени для всех возможных вариантов опорного элемента, но может ли он работать в течение линейного времени в целом на своих случайных вариантах опорного элемента? Начнем с более скромной цели: существуют ли варианты опорных элементов, для которых алгоритм `RSelect` работает в течение линейного времени?

Из чего складывается хороший опорный элемент? Ответ такой же, как для алгоритма `QuickSort` (см. раздел 5.3): хорошие опорные элементы гарантируют, что рекурсивные вызовы получают значительно более мелкие подзадачи. Наихудший сценарий — это опорный элемент, который дает наиболее несбалансированное разбиение с одним пустым подмассивом и другим подмассивом, в котором есть все, кроме опорного элемента (как в тестовой задаче 6.2). Этот сценарий возникает, когда минимальный или максимальный элемент выбран в качестве опорного. Наилучший сценарий — это опорный элемент, который дает наиболее сбалансированное разбиение с двумя подмассивами равной длины[1]. Этот сценарий случается, когда срединный элемент выбран в качестве опорного. Исследование этого сценария может показаться цикличным, так как мы вполне можем сначала попытаться вычислить медиану! Но все равно попытка выяснить наибольшее возможное время работы, которое может иметь

[1] Мы игнорируем счастливый случай, в котором выбранный опорный элемент находится в рамках той порядковой статистики, которая ищется, — это вряд ли произойдет до самых последних нескольких рекурсивных вызовов алгоритма.

алгоритм `RSelect` (лучше бы, чтобы оно было линейным!), является полезным мысленным экспериментом.

Обозначим через $T(n)$ время работы алгоритма `RSelect` на массивах длиной $n$. Если алгоритм `RSelect` волшебным образом выбирает медиану элемента заданного подмассива в каждом рекурсивном вызове, то каждый рекурсивный вызов работает линейно в своем подмассиве (в основном в подпрограмме `Partition`) и делает один рекурсивный вызов с половиной размера подмассива:

$$T(n) \leq \underbrace{T\left(\frac{n}{2}\right)}_{\text{поскольку опорный элемент=медиана}} + \underbrace{O(n)}_{\textit{Partition и т. д.}}$$

Это рекуррентное соотношение главное в основном методе (теорема 4.1): поскольку имеется один рекурсивный вызов ($a = 1$), размер подзадачи падает с коэффициентом 2 ($b = 2$), и линейная работа выполняется вне рекурсивного вызова ($d = 1$), $1 = a < b^d = 2$, и второй случай основного метода показывает нам, что $T(n) = O(n)$. Это важная проверка работоспособности: если алгоритму `RSelect` достаточно повезет, он будет работать в течение линейного времени.

Итак, находится ли время работы алгоритма `RSelect`, как правило, ближе к наилучшей производительности $\Theta(n)$ или его наихудшей производительности $\Theta(n^2)$? С учетом успеха рандомизированного алгоритма `QuickSort` за нашими плечами, мы могли бы надеяться на то, что типичные сценарии выполнения алгоритма `RSelect` имеют производительность, близкую к наилучшему сценарию. И действительно, в то время как в принципе алгоритм `RSelect` может выполняться за время $\Theta(n^2)$, вы почти всегда будете наблюдать за временем работы $O(n)$ на практике.

**Теорема 6.1 (время работы алгоритма `RSelect`).** *Для каждого входного массива длиной $n \geq 1$ среднее время работы алгоритма* `RSelect` *составляет $O(n)$.*

В разделе 6.2. приводится доказательство теоремы 6.1.

Удивительно, но среднее время работы алгоритма `RSelect` только на постоянный множитель больше, чем время, необходимое для чтения входных данных! Поскольку сортировка требует $\Omega(n \log n)$ времени (раздел 5.6), теорема 6.1 показывает, что задача выбора принципиально проще, чем задача сортировки.

Те же самые комментарии относительно среднего времени работы рандомизированного алгоритма `QuickSort` (теорема 5.1) применяются и здесь. Алгоритм `RSelect` универсален в том, что граница времени работы для произвольных входных данных и «среднее» относится только к случайным опорным элементам, выбираемым алгоритмом. Как и в случае с алгоритмом `QuickSort`, константа, скрытая в обозначении *O*-большое в теореме 6.1, достаточно мала, и алгоритм `RSelect` может быть реализован для работы на том же месте, без выделения значительной дополнительной оперативной памяти[1].

## 6.1.6. Решение тестовых заданий 6.1–6.2

### Решение тестового задания 6.1

**Правильный ответ: (б).** После разделения массива мы знаем, что опорный элемент находится на своем законном месте, в котором все меньшие числа расположены перед ним и бо́льшие числа расположены после него. Поскольку опорный элемент оказался в третьей позиции массива, он является третьим по величине элементом. Мы же ищем пятый наименьший элемент, то есть он больше. Поэтому мы можем быть уверены, что 5-я порядковая статистика находится во втором подмассиве, и нам нужно применить рекурсию всего один раз. Какую порядковую статистику мы ищем в рекурсивном вызове? Первоначально мы искали самую маленькую пятую, но теперь мы отбросили опорный элемент и два элемента меньше него. Поскольку $5 - 3 = 2$, мы ищем второй наименьший элемент среди тех, которые передаются рекурсивному вызову.

### Решение тестового задания 6.2

**Правильный ответ: (в).** Наименьшее время работы алгоритма `RSelect` такое же, как и для рандомизированного алгоритма `QuickSort`. Плохой пример такой же, как в тестовом задании 5.1: предположим, что входной массив уже отсортирован, и алгоритм неоднократно выбирает первый элемент в качестве опорного. В каждом рекурсивном вызове первая часть подмассива пуста,

[1] Реализация на том же месте использует левую и правую конечные точки для отслеживания текущего подмассива, как в псевдокоде для алгоритма `QuickSort` в разделе 5.2.5. См. также задачу по программированию 6.5.

в то время как вторая часть имеет все, кроме текущего опорного элемента. Следовательно, длина подмассива каждого рекурсивного вызова меньше предыдущего. Работа, выполняемая в каждом рекурсивном вызове (в основном подпрограммой `Partition`), линейна по длине подмассива. При вычислении медианы элемента существует $\approx \frac{n}{2}$ рекурсивных вызовов, каждый из которых имеет подмассив длиной не менее $\frac{n}{2}$, и поэтому общее время работы составляет $\Omega(n^2)$.

# *6.2. Анализ алгоритма RSelect

Один из способов доказать линейное ожидаемое время работы для алгоритма `RSelect` (теорема 6.1) — следовать той же схеме декомпозиции, которая так хорошо работала для анализа рандомизированного алгоритма `QuickSort` (раздел 5.5), с индикаторными случайными величинами, которые отслеживают сравнения. В случае с алгоритмом `RSelect` мы также можем обойтись более простым экземпляром схемы декомпозиции, которая формализует интуитивное понимание из раздела 5.4.3: (i) случайные опорные элементы, скорее всего, будут довольно хорошими; и (ii) довольно хорошие опорные элементы быстро прогрессируют.

## 6.2.1. Отслеживание прогресса посредством фаз

Мы уже отметили, что вызов алгоритма `RSelect` работает вне рекурсивного вызова, в первую очередь в вызове подпрограммы `Partition`. То есть существует константа $c > 0$ такая, что

> (*) для каждого входного массива длиной $n$ алгоритм `RSelect` выполняет не более $cn$ операций вне рекурсивного вызова.

Поскольку `RSelect` всегда делает только один рекурсивный вызов, мы можем отслеживать его ход по длине подмассива, с которым он в настоящее время работает и который со временем становится все меньше. Для простоты мы будем использовать более примитивную версию этой меры продвижения[1].

[1] Можно сделать более точный анализ, который даст постоянный множитель в границе времени работы.

Предположим, что внешнему вызову алгоритма `RSelect` дан массив длиной $n$. Для целого числа $j \geq 0$ мы говорим, что рекурсивный вызов алгоритма `RSelect` находится в *фазе j*, если длина его подмассива лежит между

$$\left(\frac{3}{4}\right)^{j+1} \times n \quad \text{и} \quad \left(\frac{3}{4}\right)^{j} \times n.$$

Например, самый внешний вызов алгоритма `RSelect` всегда находится в фазе 0, как и любые последующие рекурсивные вызовы, которые работают как минимум с 75 % первоначального входного массива. Рекурсивные вызовы с подмассивами, которые содержат от $\left(\frac{3}{4}\right)^{2} \approx$ от 56 до 75 % исходных элементов, относятся к фазе 1 и так далее. По фазе $j \approx \log_{4/3} n$ подмассив имеет размер не более 1, и нет никаких дальнейших рекурсивных вызовов.

Для каждого целого числа $j \geq 0$ обозначим через $X_j$ случайную величину, равную числу рекурсивных вызовов в фазе $j$. $X_j$ может доходить до 0, поскольку фаза может быть полностью пропущена, и, безусловно, эта величина не может быть больше $n$ максимального количества рекурсивных вызовов, выполняемых алгоритмом `RSelect`. По (*) алгоритму `RSelect` выполняет не более

$$c \times \underbrace{\left(\frac{3}{4}\right)^{j} \times n}_{\substack{\text{макс. длина подмассива} \\ (\text{фаза } j)}}$$

операций в каждом рекурсивном вызове в фазе $j$. Затем мы можем разложить время выполнения алгоритма `RSelect` по различным фазам:

$$\text{время работы } RSelect \leq \sum_{j \geq 0} \underbrace{X_j}_{\substack{\text{кол-во вызовов} \\ (\text{фаза } J)}} \times \underbrace{c\left(\frac{3}{4}\right)^{j} n}_{\substack{\text{работа на вызов} \\ (\text{фаза } j)}}$$

$$= cn \sum_{j \geq 0} \left(\frac{3}{4}\right)^{j} X_j.$$

Эта верхняя граница времени работы алгоритма `RSelect` является усложненной случайной величиной, но она представляет собой взвешенную сумму более простых случайных величин ($X_j$). Ваша автоматическая реакция на этом

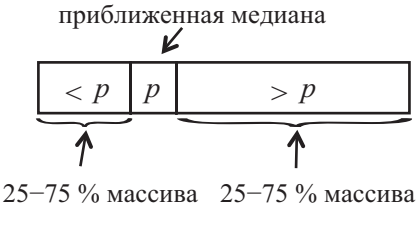
этапе должна заключаться в применении линейности математического ожидания (теорема Б.1), чтобы уменьшить вычисление усложненной случайной величины до более простых единиц:

$$\mathbf{E}\left[\text{время работы } RSelect\right] \leq cn\sum_{j\geq 0}\left(\frac{3}{4}\right)^{j}\mathbf{E}\left[X_j\right]. \tag{6.1}$$

Итак, что же такое $\mathbf{E}[X_j]$?

## 6.2.2. Сведение к задаче подбрасывания монеты

При установлении границы ожидаемого числа $\mathbf{E}[X_j]$ рекурсивных вызовов в фазе $j$ у нас происходит две вещи. Во-первых, каждый раз, когда мы выбираем довольно хороший опорный элемент, мы переходим к более поздней фазе. Как и в разделе 5.4.3, определяем *приближенную медиану* подмассива как элемент, который больше, чем как минимум 25 % других элементов в подмассиве, а также меньше, чем как минимум 25 % других элементов. Рисунок после разделения вокруг такого опорного элемента выглядит следующим образом:




Неважно, какой случай инициируется в алгоритме `RSelect`, рекурсивный вызов получает подмассив длиной не более ¾ подмассива в предыдущем вызове, и поэтому принадлежит более поздней фазе. Этот аргумент доказывает следующее ниже утверждение.

**Утверждение 6.2 (приближенные медианы прогрессируют).** *Если рекурсивный вызов в фазе j выбирает приближенную медиану, тогда следующий рекурсивный вызов принадлежит фазе j* + 1 *или более поздней*.

Во-вторых, как доказано в разделе 5.4.3, рекурсивный вызов имеет приличный шанс выбрать приближенную медиану.

**Утверждение 6.3 (приближенные медианы избыточны).** *Вызов алгоритма* `RSelect` *выбирает приближенную медиану с вероятностью не менее 50 %.*

Например, в массиве, содержащем элементы {1, 2, ..., 100}, каждый из пятидесяти элементов между 26 и 75 включительно является приближенной медианой.

Утверждения 6.2 и 6.3 позволяют нам заменить выяснение числа рекурсивных вызовов в фазе $j$ простым экспериментом по подбрасыванию монеты. Предположим, у вас есть симметричная монета, которая с равной степенью вероятности повернется орлом или решкой. Подбросьте монету несколько раз, остановившись, когда у вас в первый раз выпадет «орел». Пусть $N$ — количество подбрасываний монеты (включая последний бросок). Представьте, что «орел» соответствует выбору приближенной медианы (и окончанию эксперимента).

**Утверждение 6.4 (сведение к задаче подбрасывания монеты).** *Для каждой фазы* $j$, $E[X_j] \leq E[n]$.

*Доказательство*: все различия между определениями $X_j$ и $N$ таковы, что ожидаемое значение (математическое ожидание) первого может быть только меньше:

1. Не может быть рекурсивных вызовов в фазе $j$ (если фаза полностью пропущена), в то время как всегда есть как минимум одно подбрасывание монеты (первое).
2. Каждое подбрасывание монеты имеет ровно 50 % шансов продлить эксперимент (если она повернется решкой). Из утверждений 6.2. и 6.3 следует, что каждый рекурсивный вызов в фазе $j$ имеет не более 50 % вероятности продления фазы — необходимым условием является то, что ему не удается подобрать приближенную медиану.

*Ч. т. д.*

Случайная величина $N$ является *случайной величиной, имеющей геометрическое распределение с параметром* ½. Глядя на ее математическое ожидание в учебнике или в интернете, мы обнаруживаем, что $\mathbf{E}[N] = 2$. В качестве альтернативы есть хитрый способ, чтобы увидеть это, — написать математиче-

ское ожидание $N$ в терминах самого себя. Ключевая идея — в использовании того факта, что случайный эксперимент не имеет последствий (не запоминается): если первое подбрасывание монеты дает «решку», остальная часть эксперимента является копией оригинального. В математике, каким бы ни было математическое ожидание $N$, оно должно удовлетворять соотношению

$$\mathbf{E}[N] = \underbrace{1}_{\text{первый бросок}} + \underbrace{\frac{1}{2}}_{\mathbf{Pr}[\text{решка}]} \times \underbrace{\mathbf{E}[N]}_{\text{последующие броски монеты}}$$

Единственное значение для $\mathbf{E}[N]$, удовлетворяющее этому уравнению, равняется $2$[1].

Из утверждения 6.4 вытекает, что это значение является верхней границей того, что нас интересует, то есть это ожидаемое количество рекурсивных вызовов в фазе $j$.

**Следствие 6.5 (два вызова в расчете на фазу).** *Для каждого $j$* $\mathbf{E}[X_j] \leq 2$.

## 6.2.3. Соединяем все вместе

Теперь мы можем использовать верхнюю границу в следствии 6.5 для $\mathbf{E}[X_j]$, чтобы упростить нашу верхнюю границу (6.1) для ожидаемого времени работы алгоритма `RSelect`:

$$\mathbf{E}[\text{время работы } \mathit{RSelect}] \leq cn\sum_{j\geq 0}\left(\frac{3}{4}\right)^j \mathbf{E}\left[X_j\right] \leq 2cn\sum_{j\geq 0}\left(\frac{3}{4}\right)^j.$$

Сумма $\sum_{j\leq 0}\left(\frac{3}{4}\right)^j$ выглядит запутанно, но этого зверя мы уже приручили. При доказательстве основного метода (раздел 4.4) мы отвлеклись, чтобы обсудить геометрические ряды (раздел 4.4.8) и получили точную формулу (4.6):

$$1 + r + r^2 + \ldots + r^k = \frac{1-r^{k+1}}{1-r}$$

[1] Строго говоря, мы должны также исключить возможность того, что $\mathbf{E}[N] = +\infty$ (что нетрудно сделать).

для каждого вещественного числа $r \neq 1$ и неотрицательного целого числа $k$. При $r < 1$ это количество составляет не более $\frac{1}{1-r}$, неважно, каким большим является $k$. Подставив $r = ¾$, у нас получается

$$\sum_{j \geq 0} \left(\frac{3}{4}\right)^j \leq \frac{1}{1-\frac{3}{4}} = 4,$$

и поэтому

$$\mathbf{E}\left[\text{время работы } RSelect\right] \leq 8cn = O(n).$$

На этом завершаются анализ алгоритма `RSelect` и доказательство теоремы 6.1.

*Ч. т. д.*

## *6.3. Алгоритм DSelect

Алгоритм `RSelect` выполняется за линейное время для каждого входа, в котором математическое ожидание связано со случайными вариантами выбора, выполняемыми алгоритмом. Требуется ли рандомизация для линейного выбора[1]? В этом разделе и далее эта задача решается при помощи детерминированного линейного алгоритма для задачи выбора.

Для задачи сортировки среднее время работы $O(n \log n)$ рандомизированного алгоритма `QuickSort` совпадает со временем работы детерминированного алгоритма `MergeSort`, и оба алгоритма — `QuickSort` и `MergeSort` — являются полезными алгоритмами для применения на практике. С другой стороны, хотя детерминированный линейный алгоритм выбора, описанный в этом разделе, на практике работает нормально, он не конкурирует с алгоритмом `RSelect`. Это происходит по двум причинам: из-за больших постоянных множителей во времени работы алгоритма `DSelect` и выполняемой алгоритмом работы, связанной с выделением и управлением дополнительной оперативной памятью. Тем не менее идеи в алгоритме настолько круты, что я не могу вам о них не рассказать.

[1] Понимание мощных возможностей рандомности в вычислениях в более общем плане является сложным вопросом и продолжает оставаться темой активных исследований в теоретической информатике.

### 6.3.1. Гениальная идея: медиана медиан

Алгоритм `RSelect` быстр, потому что имеется высокая вероятность того, что случайные опорные элементы будут довольно хорошими, обеспечивая примерно сбалансированное разбиение входного массива после разделения, к тому же довольно хорошие опорные элементы быстро прогрессируют. Если нам не разрешено использовать рандомизацию, то каким образом можно вычислить довольно хороший опорный элемент, не делая слишком много работы?

Гениальная идея в детерминированном линейном выборе заключается в использовании «медианы медиан» в качестве прокси для истинной медианы. Алгоритм рассматривает элементы входного массива как спортивные команды и проводит турнир на выбывание в два тура, чемпионом которого является опорный элемент; см. также рис. 6.1.

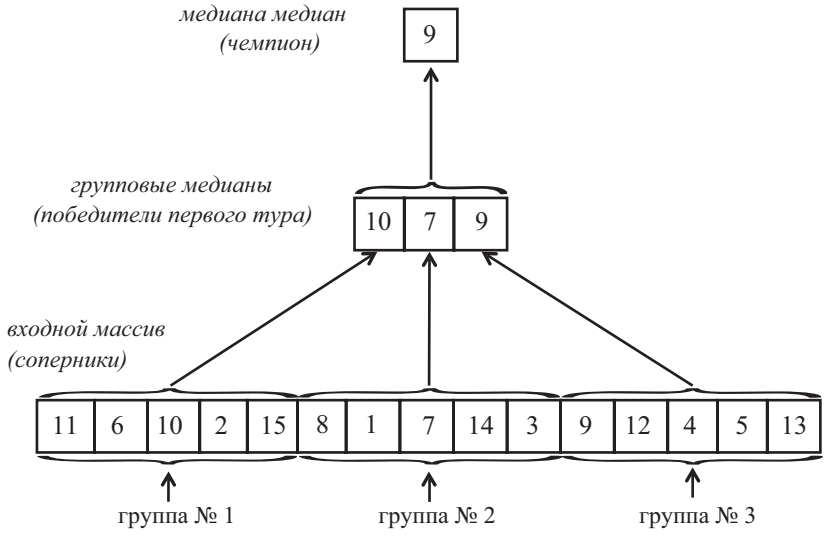



**Рис. 6.1.** Вычисление опорного элемента на основе турнира на выбывание в два тура. В этом примере выбранный опорный элемент — это не медиана входного массива, а довольно близкое к ней значение

Первый тур является групповым этапом с элементами в позициях 1–5 входного массива в качестве первой группы, элементами в позициях 6–10 в качестве

второй группы и так далее. Победитель первого тура группы из пяти элементов определяется как медиана элемента (то есть третий наименьший). Поскольку имеется $\approx \frac{n}{5}$ групп из пяти элементов, есть $\approx \frac{n}{5}$ первых победителей. (Как обычно, мы игнорируем дроби для простоты.) Чемпион турнира определяется как медиана победителей первого тура.

## 6.3.2. Псевдокод для алгоритма DSelect

Как на самом деле вычислить медиану медиан? Реализация первого этапа турнира на выбывание проста, поскольку каждое вычисление медианы включает в себя всего пять элементов. Например, каждое такое вычисление может быть выполнено методом перебора (для каждой из пяти возможностей подробно проверить, является ли он срединным элементом) либо с использованием нашего сведения к задаче сортировки (раздел 6.1.2). Для реализации второго этапа мы вычисляем медиану из $\approx \frac{n}{5}$ победителей первого тура *рекурсивно*.

**DSELECT**

**Вход**: массив $A$ из $n \geq 1$ разных чисел и целое число $i \in \{1, 2, ..., n\}$.

**Выход**: $i$-я порядковая статистика массива $A$.

```
1 if n = 1 then                    // базовый случай
2     return A[1]
3 for h := 1 to n/5 do             // победители первого тура
4     C[h] := срединный элемент из h-й группы из 5 элементов
5 p := DSelect(C, n/10)            // медиана медиан
6 разделить A вокруг p
7 j := позиция p в разделенном массиве
8 if j = i then                    // вам посчастливилось!
9     return p
10 else if j > i then
11     return DSelect(первая часть A, i)
12 else                                          // j < i
13     return DSelect(вторая часть A, i – j
```

Строки 1–2 и 6–13 идентичны алгоритму `RSelect`. Строки 3–5 являются единственной новой частью алгоритма; они вычисляют медиану медианы входного массива, заменяя строку в алгоритме `RSelect`, которая выбирает опорный элемент случайным образом.

Строки 3 и 4 вычисляют победителей первого тура турнира на выбывание, где срединный элемент каждой группы из пяти элементов вычисляется с использованием метода перебора или алгоритма сортировки, и копируют этих победителей в новый массив $C$[1]. Строка 5 вычисляет чемпиона турнира путем рекурсивного вычисления медианы массива $C$; поскольку $C$ имеет длину (ориентировочно) $\frac{n}{5}$, это $\frac{n}{10}$-я порядковая статистика массива $C$. На любом шаге алгоритма никакая рандомизация не используется.

## 6.3.3. Понимание алгоритма DSelect

Рекурсивный вызов алгоритма `DSelect` при вычислении опорного элемента может показаться опасно цикличным. Чтобы понять, что происходит, давайте сначала обозначим общее количество рекурсивных вызовов.

**ТЕСТОВОЕ ЗАДАНИЕ 6.3**

Сколько рекурсивных вызовов, как правило, делает одиночный вызов алгоритма `DSelect`?

а) 0.

б) 1.

в) 2.

г) 3.

(Решение и пояснение см. ниже.)

**Правильный ответ: (в).** Отбросив базовый случай и счастливый случай, в которых опорный элемент оказывается требуемой порядковой статистикой,

[1] Этот вспомогательный массив является причиной, по которой алгоритм `DSelect`, в отличие от алгоритма `RSelect`, не может выполняться на том же месте.

алгоритм `DSelect` делает два рекурсивных вызова. Чтобы понять почему, не перемудрите; просто проверьте псевдокод алгоритма `DSelect` построчно. В строке 5 имеется один рекурсивный вызов и еще один в строке 11 либо 13.

Есть два вызывающих путаницу распространенных вопроса насчет этих двух рекурсивных вызовов. Во-первых, не является ли тот факт, что алгоритм `RSelect` делает всего один рекурсивный вызов, причиной, по которой он работает быстрее, чем наши алгоритмы сортировки? Разве алгоритм `DSelect` не отказывается от этого улучшения, делая два рекурсивных вызова? Раздел 6.4 показывает, что, поскольку дополнительный рекурсивный вызов в строке 5 должен решить только относительно небольшую подзадачу (с 20 % элементов исходного массива), мы все еще можем спасти линейный анализ.

Во-вторых, два рекурсивных вызова играют принципиально разные роли. Целью рекурсивного вызова в строке 5 является определение хорошего опорного элемента для текущего рекурсивного вызова. Цель рекурсивного вызова в строке 11 или 13 обычная — рекурсивно решить более мелкую оставшуюся задачу, оставленную текущим рекурсивным вызовом. Тем не менее рекурсивная структура в алгоритме `DSelect` полностью следует традиции всех других алгоритмов «разделяй и властвуй», которые мы изучали: каждый рекурсивный вызов делает небольшое количество последующих рекурсивных вызовов со строго более мелкими подзадачами и выполняет некоторую дополнительную работу. Если бы мы не беспокоились о том, что алгоритмы, такие как `MergeSort` или `QuickSort`, будут выполняться вечно, то нам не следовало бы беспокоиться и об алгоритме `DSelect`.

### 6.3.4. Время работы алгоритма DSelect

Алгоритм `DSelect` — это не просто четко сформулированная программа, которая завершается за ограниченное количество времени, — он выполняется за *линейное* время, делая больше работы только на постоянный множитель, чем необходимо для чтения входных данных.

**Теорема 6.6 (время работы алгоритма `DSelect`).** *Для каждого входного массива длиной $n \geq 1$ время работы алгоритма* `DSelect` *составляет $O(n)$.*

В отличие от времени работы алгоритма `RSelect`, которое в принципе может быть не больше $\Theta(n^2)$, время работы алгоритма `DSelect` всегда равняется $O(n)$. Тем не менее на практике вам следует предпочесть `RSelect` алгоритму `DSelect`, потому что первый работает на том же месте, а константа, скрытая в среднем времени работы «$O(n)$» в теореме 6.1, меньше константы, скрытой в теореме 6.6.

### СУПЕРКОМАНДА COMPUTER SCIENCE

Одна из целей этой серии книг — сделать известные алгоритмы настолько простыми (по крайней мере, задним числом), чтобы дать вам ощущение, что вы их придумали бы сами, если бы вы были в нужном месте в нужное время. Почти никто этого не чувствует в отношении алгоритма `DSelect`, который был разработан суперкомандой из пяти исследователей, четверо из которых были награждены премией Тьюринга, вручаемой Ассоциацией вычислительной техники (ACM) (все за разные вещи!), эквивалентом Нобелевской премии по информатике[1]. Так что не отчаивайтесь, если вам трудно представить, что вы придумаете алгоритм `DSelect`, даже в самые креативные дни, — победу над Роджером Федерером (не говоря уже о победе над пятью его клонами) на теннисном корте тоже трудно представить!

[1] Этот алгоритм и его анализ были опубликованы в статье «Границы времени для задачи выбора» Мануэля Блюма, Роберта У. Флойда, Вогана Пратта, Рональда Л. Ривеста и Роберта Э. Тарьяна («Times Bounds for Selection», Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan, Journal of Computer and System Sciences, 1973). (Тогда было очень необычно видеть статьи с пятью авторами.) В хронологическом порядке: Флойд получил премию Тьюринга в 1978 году за вклад в алгоритмы, а также в языки программирования и компиляторы; Тарьян получил премию в 1986 году (наряду с Джоном Э. Хопкрофтом) за свою работу по алгоритмам и структурам данных, которые мы обсудим далее в последующих частях этой серии книги; Блюм был удостоен премии в 1995 году, в значительной степени за его вклад в криптографию; и Ривест, которого вы можете узнать по первой букве в криптосистеме RSA, получил ее в 2002 году (с Ади Шамиром и Леонардом Адлеманом) за свою работу по криптографии с открытым ключом. Между тем Пратт прославится достижениями, которые простираются от алгоритмов проверки простоты до совместного основания *Sun Microsystems*!

# *6.4. Анализ алгоритма DSelect

Сможет ли алгоритм `DSelect` действительно выполняться за линейное время? Похоже, что он делает чрезмерно высокий объем работы с двумя рекурсивными вызовами и значительной дополнительной работой вне рекурсивных вызовов. Любой другой алгоритм, который мы видели с двумя или более рекурсивными вызовами, имеет время выполнения $\Theta(n \log n)$ или меньше.

## 6.4.1. Работа вне рекурсивных вызовов

Начнем с выявления количества операций, выполняемых вызовом алгоритма `DSelect` вне его рекурсивных вызовов. Посредством двух шагов, которые требуют значительной работы, вычисляются победители первого тура (строки 3–4), и разделяется входной массив вокруг медианы медиан (строка 6).

Как и в алгоритме `QuickSort` или `RSelect`, второй шаг выполняется за линейное время. Как насчет первого шага?

Сосредоточимся на конкретной группе из пяти элементов. Поскольку это всего лишь постоянное число элементов (независимо от длины $n$ входного массива), вычисление медианы занимает фиксированный промежуток времени. Например, предположим, что мы делаем это вычисление путем сведения к задаче сортировки (раздел 6.1.2), скажем, используя алгоритм `MergeSort`. Мы хорошо понимаем объем работы, выполняемый алгоритмом `MergeSort` (теорема 1.2): не более

$$6m\,(\log_2 m + 1)$$

операций сортировки массива длиной $m$. Вы можете быть обеспокоены тем фактом, что алгоритм `MergeSort` не выполняется за линейное время. Но мы вызываем его только для подмассивов постоянного размера ($m = 5$), и в результате он выполняет постоянное количество операций (не более $6 \times 5 \times \times (\log_2 5 + 1) \leq 120$) на подмассив. Суммируя $\frac{n}{5}$ группы из пяти элементов, которые нужно отсортировать, это не более $120 \times \frac{n}{5} = 24n = O(n)$ операций в целом. Мы делаем вывод о том, что вне рекурсивных вызовов алгоритм `DSelect` выполняет только линейную работу.

## 6.4.2. Грубое рекуррентное соотношение

В главе 4 мы проанализировали алгоритмы «разделяй и властвуй» с использованием рекуррентных соотношений, которые выражают границу времени работы $T(n)$ в терминах числа операций, выполняемых рекурсивными вызовами. Давайте попробуем тот же подход здесь, обозначив через $T(n)$ максимальное количество операций, которые алгоритм `DSelect` выполняет на входном массиве длиной $n$. При $n = 1$ алгоритм `DSelect` просто возвращает единственный элемент массива, поэтому $T(1) = 1$. Для больших $n$ алгоритм `DSelect` делает один рекурсивный вызов в строке 5, еще один рекурсивный вызов в строке 11 или 13 и выполняет $O(n)$ дополнительную работу (для разделения, вычисления и копирования победителей первого тура). Это транслируется в рекуррентное соотношение следующей формы:

$$T(n) \leq T\underbrace{\left(\text{размер подзадачи № 1}\right)}_{=n/5} + T\underbrace{\left(\text{размер подзадачи № 2}\right)}_{=?} + O(n).$$

Чтобы оценить время работы алгоритма `DSelect`, нам нужно понять размеры подзадач, решаемых двумя рекурсивными вызовами. Размер первой подзадачи (строка 5) составляет $\frac{n}{5}$, это количество победителей первого тура.

Мы не знаем размера второй подзадачи — он зависит от того, какой элемент в конечном итоге является опорным, и от того, меньше или больше искомая порядковая статистика этого опорного элемента. Эта неопределенность в размере подзадачи — причина, по которой мы не использовали рекуррентные соотношения для анализа алгоритмов `QuickSort` и `RSelect`.

В особом случае, когда в качестве опорного выбрана истинная медиана элемента входного массива, вторая подзадача гарантированно будет состоять не более чем из $\frac{n}{2}$ элементов. Медиана медиан, как правило, не является истинной медианой (см. рис. 6.1). Достаточно ли она приближена по значению, чтобы гарантировать приблизительно сбалансированное разбиение входного массива и, следовательно, не слишком большую подзадачу в строке 11 или 13?

### 6.4.3. Лемма 30–70

В основе анализа алгоритма `DSelect` лежит указанная ниже лемма, которая количественно оценивает отдачу от тяжелой работы, выполняемой для вычисления медианы медиан: этот опорный элемент гарантирует разбиение входного массива на 30–70 % или больше.

**Лемма 6.7 (лемма 30–70).** *Для каждого входного массива длиной $n \geq 2$ подмассив, передаваемый рекурсивному вызову в строке 11 или 13 из алгоритма* `DSelect`*, имеет длину не более* $\frac{7}{10}n$ [1].

Лемма 30–70 позволяет нам подставить «$\frac{7}{10}n$» вместо «?» в грубом рекуррентном соотношении, приведенном выше: для каждого $n \geq 2$

$$T(n) \leq T\left(\frac{1}{5} \times n\right) + T\left(\frac{7}{10} \times n\right) + O(n). \tag{6.2}$$

Сначала мы докажем лемму 30–70, а затем докажем, что из рекуррентного соотношения (6.2) следует, что алгоритм `DSelect` является линейным алгоритмом.

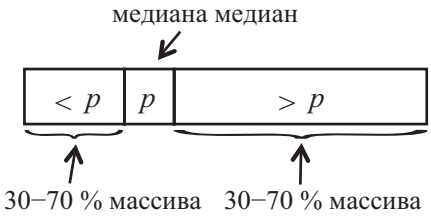
*Доказательство леммы 6.7*. Обозначим через $k = \frac{n}{5}$ количество групп из пяти элементов, а значит, и количество победителей первого тура. Определим $x_i$ как $i$-й наименьший из победителей первого тура. Эквивалентным образом, $x_1$, ..., $x_k$ — это победители первого тура, отсортированные по порядку. Чемпионом турнира, медианой медиан, является $x_{k/2}$ (или $x_{\lceil k/2 \rceil}$, если $k$ — нечетно)[2].

План состоит в том, чтобы доказать, что $x_{k/2}$ не меньше и не больше, чем как минимум 60 % элементов по меньшей мере в 50 % групп. Тогда по меньшей мере 60 % × × 50 % = 30 % элементов входного массива будет не больше медианы медиан и как минимум 30 % будет не меньше:

---

[1] Строго говоря, поскольку одна из «групп из 5» может иметь менее пяти элементов (если $n$ не кратно 5), то $\frac{7}{10}n$ должно быть $\frac{7}{10}n + 2$, округленным до ближайшего целого. Мы будем игнорировать «+2» по той же причине, по которой мы игнорируем дроби, — это деталь, которая усложняет анализ в неинтересную сторону и не влияет на суть.

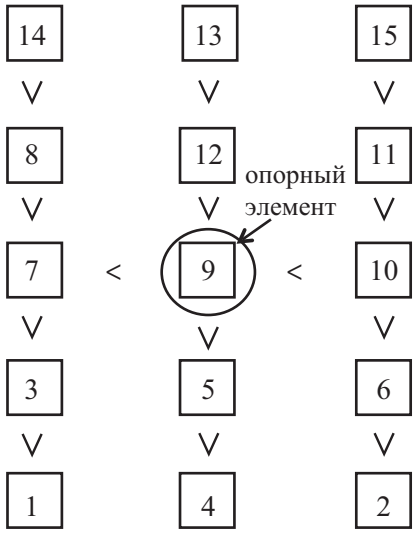
[2] $\lceil x \rceil$ обозначает функцию «ceiling», которая округляет свой аргумент вверх до ближайшего целого числа.




Для реализации этого плана рассмотрим следующий мысленный эксперимент. В уме (не в реальном алгоритме!) мы выкладываем все элементы входного массива в формате двумерной решетки. Имеется пять строк, и каждый из $\frac{n}{5}$ столбцов соответствует одной из групп из пяти элементов. В каждом столбце мы выкладываем пять элементов, отсортированных по порядку снизу вверх. Наконец, мы выкладываем столбцы так, чтобы победители первого тура (то есть элементы в средней строке) были отсортированы слева направо. Например, если мы имеем следующий ниже входной массив

| 11 | 6 | 10 | 2 | 15 | 8 | 1 | 7 | 14 | 3 | 9 | 12 | 4 | 5 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

тогда соответствующая решетка примет следующий вид:




где опорный элемент, медиана медиан, расположен в угловой позиции.

**КЛЮЧЕВОЕ НАБЛЮДЕНИЕ**

Поскольку средняя строка сортируется слева направо, а каждый столбец сортируется снизу вверх, все элементы слева и книзу от опорного меньше опорного, а все элементы справа и кверху от опорного больше опорного[1].

В нашем примере опорный элемент равен «9», элементы слева и ниже равны {1, 3, 4, 5, 7}, и элементы справа и выше равны {10, 11, 12, 13, 15}. Следовательно, как минимум шесть элементов будут исключены из подмассива, передаваемого следующему рекурсивному вызову, — опорный элемент 9 и либо {10, 11, 12, 13, 15} (в строке 11), либо {1, 3, 4, 5, 7} (в строке 13). В любом случае следующий рекурсивный вызов получает не более девяти элементов, а 9 — это менее 70 % из 15.

Доказательство для общего случая то же. На рис. 6.2 показано, как выглядит решетка для произвольного входного массива. Поскольку опорный элемент является медианой элементов в средней строке, как минимум 50 % столбцов находятся слева от того, который содержит опорную точку (учитывая также собственный столбец опорного элемента). В каждом из этих столбцов как минимум 60 % элементов (три наименьших из пяти) не больше медианы столбца и, следовательно, не больше опорного элемента. Таким образом, как минимум 30 % элементов входного массива не больше опорного элемента, и все они будут исключены из рекурсивного вызова в строке 13. Аналогичным образом как минимум 30 % элементов не меньше опорного, и они будут исключены из рекурсивного вызова в строке 11. Это завершает доказательство леммы 30–70. *Ч. т. д.*

## 6.4.4. Решение рекуррентного соотношения

Из леммы 30−70 следует, что размер входных данных уменьшается с постоянным множителем с каждым рекурсивным вызовом алгоритма `DSelect`, и это служит хорошим предзнаменованием линейного времени работы. Но разве

[1] Элементы слева и выше или справа и ниже могут быть либо меньше, либо больше опорного.

это не пиррова победа? Перевешивает ли стоимость вычисления медианы медиан преимущества, получаемые от разделения вокруг довольно хорошего опорного элемента? Ответ на эти вопросы и завершение доказательства теоремы 6.6 вызывают необходимость подумать над решением рекуррентного соотношения в (6.2).

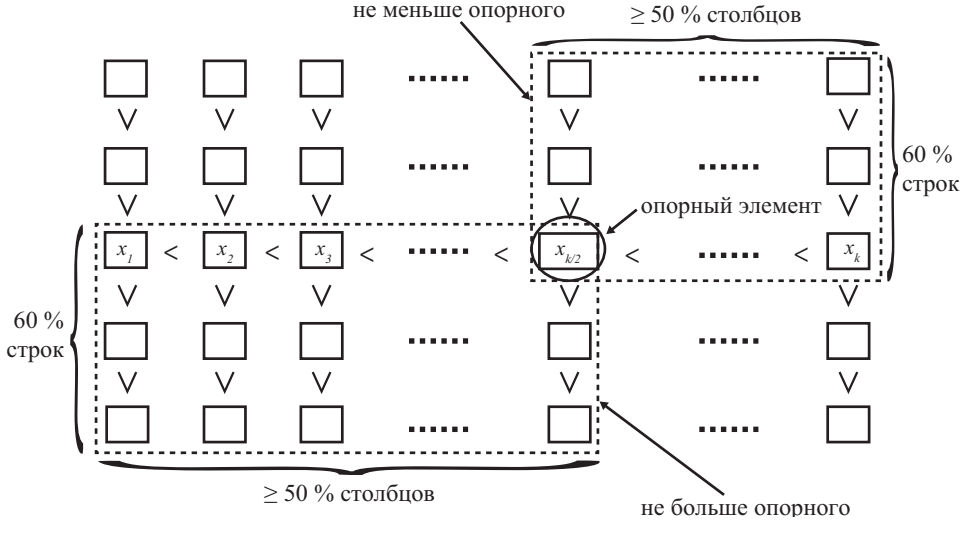



**Рис. 6.2.** Доказательство леммы 30–70. Представьте себе элементы входного массива, выложенные в виде решетки. Каждый столбец соответствует группе из пяти элементов, отсортированных снизу вверх. Столбцы сортируются в порядке их срединных элементов. Рисунок исходит из того, что $k$ — четное; для $k$ нечетного принимается «$x_{\lceil k/2 \rceil}$» вместо «$x_{k/2}$». Элементы к юго-западу от медианы медиан могут быть только меньше нее; элементы к северо-востоку могут быть только больше нее. В результате не менее 60 % × 50 % = 30 % элементов исключаются из каждого из двух возможных рекурсивных вызовов

Поскольку алгоритм `DSelect` выполняет $O(n)$ работу вне рекурсивных вызовов (вычисление победителей первого тура, разделение массива и так далее), существует константа $c > 0$ такая, что для каждого $n \geq 2$

$$T(n) \leq T\left(\frac{1}{5} \times n\right) + T\left(\frac{7}{10} \times n\right) + cn, \tag{6.3}$$

где $T(n)$ — верхняя граница времени работы алгоритма `DSelect` на массивах длиной $n$. Мы можем допустить, что $c \geq 1$ (так как увеличение $c$ не отменяет неравенство (6.3)). Кроме того, $T(1) = 1$. Как мы увидим, важным свойством этого рекуррентного соотношения является то, что $\frac{1}{5} + \frac{7}{10} < 1$.

До сих пор мы опирались на основной метод (глава 4), чтобы оценивать все рекуррентные соотношения, с которыми мы столкнулись, — для алгоритмов `MergeSort`, `Karatsuba`, `Strassen` и других мы просто подставляли в формулу три соответствующих параметра ($a$, $b$ и $d$) и получали верхнюю границу времени работы алгоритма. К сожалению, два рекурсивных вызова в алгоритме `DSelect` имеют разные размеры входных данных, и это исключает применение теоремы 4.1. Можно обобщить аргумент дерева рекурсии в теореме 4.1, чтобы учесть рекуррентное соотношение в (6.3)[1]. Для разнообразия и для того, чтобы добавить еще один инструмент в ваш арсенал, мы перейдем к другому методу.

### 6.4.5. Метод догадок и проверок

Метод *догадок и проверок* (или эвристический метод) для оценки рекуррентных соотношений спонтанен, как это и отражено в названии, но он также чрезвычайно гибок и применяется к сколь угодно сумасшедшим рекуррентным соотношениям.

**Шаг 1: угадать.** Угадать функцию $f(n)$, которая, как вы считаете, удовлетворяет равенству $T(n) = O(f(n))$.

**Шаг 2: проверить.** Доказать по индукции по $n$, что $T(n)$ действительно равняется $O(f(n))$.

---

[1] В качестве эвристического аргумента подумайте о первой паре рекурсивных вызовов `DSelect` — двух узлах на уровне 1 дерева рекурсии данного алгоритма. Один имеет 20 % элементов входного массива, другой — не более 70 %, и работа, выполняемая на этом уровне, линейна по сумме размеров двух подзадач. Следовательно, объем работы, выполняемой на уровне 1, составляет не более 90 % от объема работы на уровне 0, и так далее на последующих уровнях. Это напоминает второй случай основного метода, в котором работа в расчете на уровень с каждым уровнем падает с постоянным множителем (коэффициентом). Эта аналогия наводит на мысль о том, что $O(n)$ работа, выполняемая в корне, должна доминировать над временем работы (ср. раздел 4.4.6).

В общем шаг угадывания — это немного темное искусство. В нашем случае, поскольку мы пытаемся доказать линейную временную границу, мы выдвигаем догадку, что $T(n) = O(n)$[1]. То есть мы угадываем, что существует константа $\ell > 0$ (независимая от $n$) такая, что

$$T(n) \leq \ell \times n \tag{6.4}$$

для каждого положительного целого числа $n$. Если это верно, то, поскольку $\ell$ — константа, следствием нашей надежды будет $T(n) = O(n)$.

При верификации (6.4) мы вольны выбирать $\ell$ какое угодно при условии, что оно не зависит от $n$. Подобно доказательствам на основе асимптотических обозначений, обычный способ выяснить соответствующую константу — реконструировать ее (ср. раздел 2.5). Здесь мы возьмем $\ell = 10c$, где $c$ — это постоянный множитель в рекуррентном соотношении (6.3). (Поскольку $c$ является константой, то и $\ell$ тоже.) Откуда взялось это число? Это наименьшая константа, для которой справедливо приведенное ниже неравенство (6.5).

Мы доказываем (6.4) по индукции. На языке приложения А $P(n)$ — это утверждение, что $T(n) \leq \ell \times n = 10c \times n$. Для базового случая нам нужно доказать непосредственно, что $P(1)$ истинно, имея в виду, что $T(1) \leq 10c$. Рекуррентное соотношение прямо говорит, что $T(1) = 1$ и $c \geq 1$, поэтому, безусловно, $T(1) \leq 10c$.

В качестве индукционного перехода зададим произвольное положительное целое число $n \geq 2$. Нам нужно доказать, что $T(n) \leq \ell \times n$. Индуктивная гипотеза констатирует, что все $P(1)$, ..., $P(n-1)$ — истинны, имея в виду, что $T(k) \leq \ell \times k$ для всех $k < n$. Чтобы доказать $P(n)$, давайте просто доверимся нашей интуиции.

Во-первых, рекуррентное соотношение (6.3) раскладывает $T(n)$ на три члена:

$$T(n) \leq \underbrace{T\left(\frac{1}{5} \times n\right)}_{\underbrace{\leq \ell \times \frac{n}{5}}_{\text{(индукт. гипотеза)}}} + \underbrace{T\left(\frac{7}{10} \times n\right)}_{\underbrace{\leq \ell \times \frac{7n}{10}}_{\text{(индукт. гипотеза)}}} + cn.$$

Мы не можем напрямую осуществлять операции с любым из этих членов, но можем применить индуктивную гипотезу, один раз с $k = \frac{n}{5}$ и один раз с $k = \frac{7n}{10}$:

[1] Для нас термин «надежда и проверка» может быть более подходящим описанием!

$$T(n) \le \ell \times \frac{n}{5} + \ell \times \frac{7n}{10} + cn.$$

Сгруппировав члены,

$$T(n) \le n \underbrace{\left(\frac{9}{10}\ell + c\right)}_{\substack{=\ell \\ \text{т.к. } \ell = 10c}} = \ell \times n.$$

Это доказывает индукционный переход, который верифицирует, что $T(n) \le \ell \times$ $\times n = O(n)$, и завершает доказательство того, что гениальный алгоритм `DSelect` выполняется за линейное время (теорема 6.6). *Ч. т. д.*

**ВЫВОДЫ**

- Цель задачи выбора состоит в вычислении $i$-го наименьшего элемента неотсортированного массива.
- Задачу выбора можно решить за $O(n \log n)$ время, где $n$ — длина входного массива, путем сортировки массива и последующего возврата $i$-го элемента.
- Данная задача также может быть решена путем разделения входного массива вокруг опорного элемента, как в алгоритме `QuickSort`, и выполнения одной рекурсии на соответствующей стороне. Алгоритм `RSelect` всегда выбирает опорный элемент равномерно случайным образом.
- Время работы алгоритма `RSelect` варьируется от $\Theta(n)$ до $\Theta(n^2)$, в зависимости от выбранных опорных элементов.
- Среднее время работы алгоритма `RSelect` составляет $\Theta(n)$. Доказательство сводится к эксперименту с подбрасыванием монеты.
- Гениальная идея в детерминированном алгоритме `DSelect` заключается в том, чтобы в качестве опорного элемента использовать «медиану медиан»: разбить входной массив на группы по пять элементов, непосредственно вычислить медиану каждой группы и рекурсивно вычислить медиану этих $\frac{n}{5}$ победителей первого тура.

- ✶ Лемма 30–70 показывает, что медиана медиан гарантирует разбиение входного массива на 30–70 % или больше.
- ✶ Анализ алгоритма `DSelect` показывает, что затраты в рекурсивном вызове для вычисления медианы медиан, перевешиваются преимуществом разбиения на 30−70 %, что приводит к линейному времени выполнения.

## Задачи на закрепление материала

**Задача 6.1.** Пусть $\alpha$ есть некоторая константа, не зависящая от длины $n$ входного массива, строго между ½ и 1. Предположим, вы используете алгоритм `RSelect` для вычисления медианы элемента массива длиной $n$. Какова вероятность того, что первому рекурсивному вызову передается подмассив длиной не более $\alpha \times n$?

а) $1 - \alpha$.

б) $\alpha - ½$.

в) $1 - ½$.

г) $2\alpha - 1$.

**Задача 6.2.** Пусть $\alpha$ есть некоторая константа, не зависящая от длины $n$ входного массива, строго между ½ и 1. Допустим, что каждый рекурсивный вызов алгоритма `RSelect` выполняется так же, как и в предыдущей задаче, — поэтому всякий раз, когда рекурсивный вызов получает массив длиной $k$, его рекурсивному вызову передается подмассив длиной не более $\alpha k$. Каково максимальное число последовательных рекурсивных вызовов, которые могут произойти перед запуском базового случая?

а) $-\dfrac{\ln n}{\ln \alpha}$.

б) $-\dfrac{\ln n}{\alpha}$.

в) $-\dfrac{\ln n}{\ln(1-\alpha)}$.

г) $-\dfrac{\ln n}{\ln\left(\dfrac{1}{2}+\alpha\right)}$.

## Задачи повышенной сложности

**Задача 6.3.** В этой задаче входными данными является неотсортированный массив из $n$ разных элементов $x_1, x_2, ..., x_n$ с положительными весами $w_1, w_2, ..., w_n$. Обозначим через $W$ сумму $\sum_{i=1}^{n} w_i$ весов. Определите *взвешенную медиану* в качестве элемента $x_k$, для которого суммарный вес всех элементов со значением меньше $x_k$ (то есть $\sum_{x_i<x_k} w_i$) равняется не более $W/2$, а также суммарный вес элементов со значением больше $x_k$ (то есть $\sum_{x_i>x_k} w_i$) равняется не более $W/2$. Обратите внимание, что имеется не более двух взвешенных медиан. Дайте детерминированный линейный алгоритм для вычисления всех взвешенных медиан во входном массиве. [Подсказка: используйте алгоритм `DSelect` в качестве подпрограммы.]

**Задача 6.4.** Предположим, что мы модифицируем алгоритм `DSelect`, разбивая элементы на группы из семи элементов, а не из пяти. (Используйте медиану медиан в качестве опорного элемента, как и раньше.) Выполняется ли этот модифицированный алгоритм также за время $O(n)$? Что делать, если мы используем группы из трех элементов[1]?

## Задачи по программированию

**Задача 6.5.** Реализуйте на своем любимом языке программирования алгоритм `RSelect` из раздела 6.1. Ваша реализация должна работать на том же месте, используя реализацию подпрограммы `Partition` (которую вы, возможно, реализовали для задачи 5.6) и передавая индексы через рекурсию, чтобы отслеживать по-прежнему релевантную часть первоначального входного массива. (См. www.algorithmsilluminated.org для тестовых случаев и наборов данных для задач.)

[1] Для глубокого погружения в эту тему см. статью «Выбор на основе групп из 3 или 4 элементов занимает линейное время», Ке Чен и Адриан Думитреску («Select with Groups of 3 or 4 Takes Linear Time», Ke Chen, Adrian Dumitrescu, arXiv:1409.3600, 2014).

# *Приложения*

# Приложение А. Краткий обзор доказательств по индукции

Доказательства по индукции всплывают в информатике *постоянно*. Например, в разделе 5.1 мы используем доказательство по индукции, чтобы доказать, что алгоритм `QuickSort` всегда правильно сортирует свой входной массив. В разделе 6.4 мы используем индукцию, чтобы доказать, что алгоритм `DSelect` выполняется за линейное время.

Доказательства по индукции могут оказаться интуитивно непонятными, по крайней мере на первый взгляд. Хорошей новостью является то, что после непродолжительной практики они следуют довольно жесткому шаблону и становятся почти автоматическими. В этом приложении объясняется шаблон и приводятся два кратких примера. Если вы никогда раньше не видели доказательств по индукции, вам следует дополнить это приложение другим источником, в котором есть еще больше примеров[1].

## А.1. Шаблон для доказательств по индукции

Для наших целей доказательство по индукции устанавливает истинность утверждения $P(n)$ для каждого натурального числа $n$. Например, доказывая правильность алгоритма `QuickSort` в разделе 5.1, мы можем определить $P(n)$ как в высказывании: «для каждого входного массива длиной $n$ алгоритм `QuickSort` сортирует его правильно». При анализе времени работы алгоритма `DSelect` в разделе 6.4 мы можем определить $P(n)$ как в высказывании: «для каждого входного массива длиной $n$ алгоритм `DSelect` останавливается после выполнения не более $100n$ операций». Индукция позволяет доказывать свойство алгоритма, например правильность или границу времени работы, поочередно устанавливая истинность свойства для каждой длины входных данных.

Аналогично рекурсивному алгоритму, доказательство по индукции состоит из двух частей: *базового случая* (базы индукции) и *индукционного перехода* (шага индукции). Базовый случай доказывает, что $P(n)$ истинно для всех до-

[1] Например, см. главу 2 свободно доступных лекций Эрика Лемана и Тома Лейтона (http://www.boazbarak.org/cs121/LehmanLeighton.pdf).

статочно малых значений $n$ (как правило, $n = 1$). При индукционном переходе вы принимаете допущение, что все $P(1)$, ..., $P(n-1)$ истинны, и доказываете, что $P(n)$, следовательно, тоже истинно.

**Базовый случай**: доказать непосредственно, что $P(1)$ истинно.

**Индукционный переход**: доказать, что для каждого целого числа $n \geq 2$,

$$\text{если } \underbrace{P(1),\ P(2),\ \ldots,\ P(n-1) \text{ истинны}}_{\text{индуктивная гиптоза}},\ \text{то } P(n) \text{ истинно.}$$

При индукционном переходе вы можете *допустить*, что истинность $P(k)$ уже была установлена для всех значений $k$ меньше $n$ — это называется индуктивной гипотезой, — и должны использовать это допущение для установления истинности $P(n)$.

Если вы докажете и базовый случай, и индукционный переход, то $P(n)$ — действительно истинно для каждого положительного целого числа $n$. $P(1)$ истинно по базовому случаю, и применение индукционного перехода снова и снова показывает, что $P(n)$ истинно для сколь угодно больших значений $n$.

## A.2. Пример: замкнутая формула

Мы можем использовать индукцию для получения замкнутой формулы для суммы первых $n$ положительных целых чисел. Обозначим через $P(n)$ утверждение, что

$$1+2+3+\ldots+n=\frac{(n+1)n}{2}.$$

При $n = 1$ левая часть равняется 1, и правая часть равняется $\frac{2\times 1}{2} = 1$. Это показывает, что $P(1)$ истинно и что на этом базовый случай заканчивается. Для индукционного перехода выберем произвольное целое число $n \geq 2$ и предположим, что все $P(1)$, $P(2)$, ..., $P(n-1)$ истинны. В частности, мы можем принять $P(n-1)$, то есть утверждение

$$1+2+3+\ldots+(n-1)=\frac{n(n-1)}{2}.$$

Теперь мы можем добавить $n$ в обе части, чтобы вывести

$$1+2+3+\ldots+n=\frac{n(n-1)}{2}+n=\frac{n^2-n+2n}{2}=\frac{(n+1)n}{2},$$

что доказывает $P(n)$. Поскольку мы установили истинность и базового случая, и индукционного перехода, мы можем прийти к заключению, что $P(n)$ истинно для каждого положительного целого числа $n$.

## A.3. Пример: размер полного двоичного дерева

Далее давайте подсчитаем число узлов в полном двоичном дереве с $n$ уровнями. На рис. A.1 мы видим, что при $n = 4$ уровнях число узлов равняется $15 = 2^4 - 1$. Может ли этот образец быть истинным в целом?

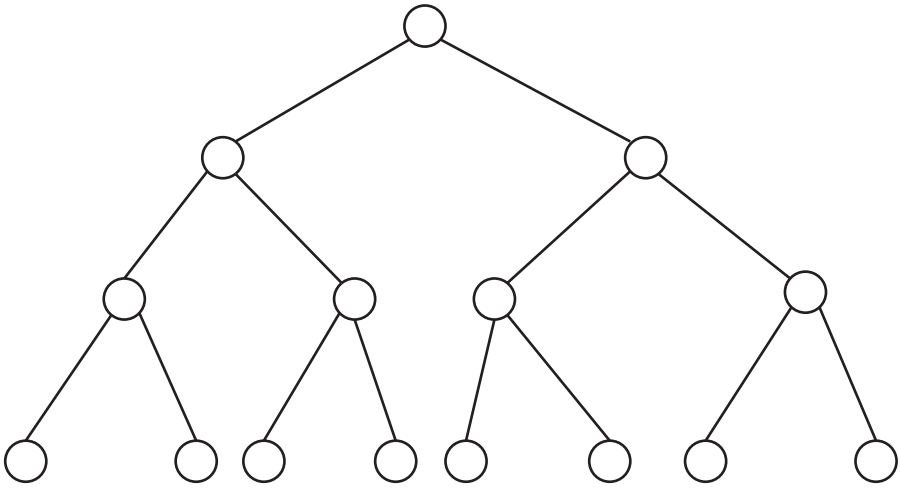


**Рис. A.1.** Полное двоичное дерево с четырьмя уровнями и $2^4 - 1 = 15$ узлами

Для каждого натурального числа $n$ пусть $P(n)$ есть высказывание «полное двоичное дерево с $n$ уровнями имеет $2^n - 1$ узлов». В качестве базового случая обратите внимание на то, что полное двоичное дерево с одним уровнем имеет ровно один узел. Поскольку $2^1 - 1 = 1$, это доказывает, что $P(1)$ истинно. В качестве индукционного перехода задайте положительное целое число $n \geq 2$ и допустите, что все $P(1)$, ..., $P(n-1)$ истинны. Узлы полного двоичного

дерева с $n$ уровнями можно разделить на три группы: (i) корень; (ii) узлы в левом поддереве корня; и (iii) узлы в правом поддереве корня. Левое и правое поддеревья корня сами являются полными двоичными деревьями, каждый из $n-1$ уровней. Поскольку мы допускаем, что $P(n-1)$ истинно, в каждом из левых и правых поддеревьев есть ровно $2^{n-1}-1$ узлов. Сложив узлы в три группы, мы в общей сложности получим

$$\underbrace{1}_{\text{корень}} + \underbrace{2^{n-1}-1}_{\text{левое поддерево}} + \underbrace{2^{n-1}-1}_{\text{правое поддерево}} = 2^n - 1$$

узлов в дереве. Это доказывает утверждение о $P(n)$ и, поскольку $n \geq 2$ было каким угодно, завершает индукционный переход. Мы делаем вывод, что $P(n)$ истинно для каждого натурального числа $n$.

# Приложение Б. Краткий обзор дискретной вероятности

В этом приложении рассматриваются понятия дискретной вероятности, которые необходимы для нашего анализа рандомизированного алгоритма `QuickSort` (теорема 5.1 и раздел 5.5): выборочные пространства, события, случайные величины, математическое ожидание и линейность математического ожидания. Раздел Б.6 завершается примером распределения нагрузки, который связывает все эти понятия воедино. Мы также будем использовать эти понятия в будущих частях этой серии книг, в контексте структур данных, графовых алгоритмов и алгоритмов локального поиска. Если вы видите этот материал в первый раз, вам, вероятно, следует, помимо чтения этого приложения, более тщательно изучить данный вопрос[1]. Если вы видели его раньше, вам необязательно читать это приложение от начала до конца — загляните в ту часть, где содержатся сведения, которые вам нужно освежить.

[1] В дополнение к лекциям Лемана—Лейтона, упомянутым в приложении А, существует бесплатный вики-учебник по дискретной вероятности (https://en.wikibooks.org/wiki/High_School_Mathematics_Extensions/Discrete_Probability).

## Б.1. Выборочные пространства

Нас интересуют случайные процессы, в которых может произойти любое количество разных вещей. *Выборочное пространство*, или пространство элементарных событий, — это множество $\Omega$ всех разнообразных исходов, которые могут произойти, — *универсальное множество*, в котором мы будем назначать вероятности, принимать средние значения и так далее. Например, если наш случайный процесс — это бросок кубика, то $\Omega = \{1, 2, 3, 4, 5, 6\}$. К счастью, при анализе рандомизированных алгоритмов мы почти всегда можем принять $\Omega$ как конечное множество и работать только с дискретной вероятностью, которая гораздо более элементарна, чем общая теория вероятностей.

Каждый элемент $i$ выборочного пространства $\Omega$ сопровождается неотрицательной *вероятностью* $p(i)$, которую можно рассматривать как частоту, с которой исход случайного процесса равняется $i$. Например, если шестигранная игральная кость является геометрически правильной, то $p(i)$ равняется $\frac{1}{6}$ для каждого $i = 1, 2, 3, 4, 5, 6$. В общем случае, поскольку $\Omega$ — это все, что вообще может произойти, вероятности должны равняться 1:

$$\sum_{i \in \Omega} p(i) = 1.$$

Общим частным является случай, когда каждый элемент $\Omega$ одинаково вероятен — так называемое *равномерное распределение*, где $p(i) = \frac{1}{|\Omega|}$ для каждого $i \in \Omega$[1]. Это понятие может показаться довольно абстрактным, поэтому давайте представим два примера. В первом примере случайный процесс представляет собой бросок двух стандартных (шестигранных) игральных костей. Выборочное пространство — это множество из 36 различных исходов, которые могут произойти:

$$\Omega = \underbrace{\{(1,1),(2,1),(3,1),\ \ldots,\ (5,6),(6,6)\}}_{\text{36 упорядоченных пар}}.$$

Если допустить, что кости геометрически правильны, то каждый из этих исходов одинаково вероятен: $p(i) = \frac{1}{36}$ для каждого $i \in \Omega$.

[1] Для конечного множества $S$, $|S|$ обозначает число элементов в $S$.

Второй пример, более близкий к алгоритмам, — это выбор опорного элемента в самом внешнем вызове рандомизированного алгоритма `QuickSort` (раздел 5.4). Любой элемент входного массива может быть выбран в качестве опорного, поэтому

$$\Omega = \underbrace{\{1, 2, 3, \ldots, n\}}_{\text{возможные позиции опорного элемента}}$$

где $n$ — длина входного массива. По определению в рандомизированном алгоритме `QuickSort` каждый элемент одинаково вероятно будет выбран в качестве опорного элемента, и поэтому $p(i) = \frac{1}{n}$ для каждого $i \in \Omega$.

## Б.2. События

*Событие* — это подмножество $S \subseteq \Omega$ выборочного пространства — набор возможных исходов случайного процесса. Вероятность **Pr**[$S$] события $S$ определяется так, как можно было бы ожидать, то есть как вероятность того, что один из исходов $S$ происходит:

$$\mathbf{Pr}[S] = \sum_{i \in S} p(i).$$

Давайте немного потренируемся с этим понятием, использовав наши два примера.

**ТЕСТОВОЕ ЗАДАНИЕ Б.1**

Обозначим через $S$ множество исходов, для которых сумма двух стандартных кубиков равна 7. Какова вероятность события $S$[1]?

а) $\frac{1}{36}$.

б) $\frac{1}{12}$.

[1] Факт, который полезно знать, играя в азартные игры.

в) $\frac{1}{6}$.

г) $\frac{1}{2}$.

(Решение и пояснение см. в разделе Б.2.1.)

Второе тестовое задание касается выбора случайного опорного элемента в самом внешнем вызове алгоритма `QuickSort`. Мы говорим, что опорный элемент является «приближенной медианой», если по крайней мере 25 % элементов массива меньше опорного и по крайней мере 25 % элементов больше опорного.

**ТЕСТОВОЕ ЗАДАНИЕ Б.2**

Обозначим через $S$ событие, что выбранный опорный элемент в самом внешнем вызове алгоритма `QuickSort` является приближенной медианой. Какова вероятность события $S$?

а) $\frac{1}{n}$, где $n$ — это длина массива.

б) $\frac{1}{4}$.

в) $\frac{1}{2}$.

г) $\frac{3}{4}$.

(Решение и пояснение см. в разделе Б.2.2.)

## Б.2.1. Решение тестового задания Б.1

**Правильный ответ: (в).** Имеется шесть исходов, в которых сумма двух кубиков равняется 7:

$$S = \{(6,1), (5, 2), (4, 3), (3, 4), (2, 5), (1, 6)\}.$$

Поскольку каждый исход $\Omega$ равновероятен, $p(i) = \frac{1}{36}$ для каждого $i \in S$ и поэтому

$$\mathbf{Pr}[S] = |S| \times \frac{1}{36} = \frac{6}{36} = \frac{1}{6}.$$

### Б.2.2. Решение тестового задания Б.2

**Правильный ответ: (в).** В качестве мысленного эксперимента представьте, что вы разделили элементы входного массива на четыре группы: наименьшие $\frac{n}{4}$ элементов, следующие наименьшие $\frac{n}{4}$ элементов, следующие наименьшие $\frac{n}{4}$ элементов и, наконец, наибольшие $\frac{n}{4}$ элементов. (Как обычно, мы игнорируем дроби для простоты.) Каждый элемент второй и третьей групп является приближенной медианой: все $\frac{n}{4}$ элементов из первой и последней групп соответственно меньше и больше опорного. И наоборот, если алгоритм выбирает опорный элемент либо из первой, либо из последней группы, то либо элементы меньше опорного составляют только строгое подмножество первой группы, либо элементы больше опорного являются только строгим подмножеством последней группы. В этом случае опорный элемент не является приближенной медианой. Следовательно, событие $S$ соответствует $\frac{n}{2}$ элементам во второй и третьей группах; поскольку каждый элемент одинаково вероятно будет выбран в качестве опорного элемента,

$$\mathbf{Pr}[S] = |S| \times \frac{1}{n} = \frac{n}{2} \times \frac{1}{n} = \frac{1}{2}.$$

## Б.3. Случайные величины

*Случайная величина* — это численная мера выхода случайного процесса. Формально это вещественная функция $X$: $\Omega \to \mathbb{R}$, определенная на выборочном пространстве $\Omega$, вход $i \in \Omega$ в $X$ является выходом случайного процесса, а выход $X(i)$ — числовым значением.

В нашем первом примере мы можем определить случайную величину, которая является суммой двух костей. Эта случайная величина ставит в соответствие выходы (пары $(i, j)$ с $i, j \in \{1, 2, ..., 6\}$) вещественным числам согласно

отображению $(i,\ j) \mapsto i + j$. В нашем втором текущем примере мы можем определить случайную величину, которая является длиной подмассива, передаваемого первому рекурсивному вызову алгоритма `QuickSort`. Эта случайная величина ставит в соответствие каждый выход (то есть каждый выбранный вариант опорного элемента) целому числу от 0 (если выбран минимальный опорный элемент) до $n - 1$, где $n$ — это длина входного массива (если выбран максимальный опорный элемент).

В разделе 5.5 исследуется случайная величина $X$, которая представляет собой время работы рандомизированного алгоритма `QuickSort` на заданном входном массиве. Здесь пространство состояний $\Omega$ — это множество всех возможных последовательностей опорных элементов, которые алгоритм может выбрать, а $X(i)$ — число операций, выполняемых алгоритмом для конкретной последовательности $i \in \Omega$ вариантов опорных элементов[1].

## Б.4. Математическое ожидание

*Математическое ожидание*, или *ожидаемое значение* $\mathbf{E}[X]$ случайной величины $X$, — это ее среднее значение для всего, что может произойти, взвешенное соответствующим образом относительно вероятности различных исходов. Интуитивно это можно понять так: если случайный процесс повторяется снова и снова, то $\mathbf{E}[X]$ представляет собой долгосрочное среднее значение случайной величины $X$. Например, если $X$ — это значение геометрически правильной шестигранной игральной кости, то $\mathbf{E}[X] = 3{,}5$.

В математике, если $X : \Omega \to \mathbb{R}$ есть случайная величина и $p(i)$ обозначает вероятность исхода $i \in \Omega$, то

$$\mathbf{E}[X] = \sum_{i \in \Omega} p(i) \times X(i). \tag{Б.1}$$

Следующие два тестовых задания просят вас вычислить математическое ожидание двух случайных величин, определенных в предыдущем разделе.

[1] Поскольку единственная случайность в рандомизированном алгоритме `QuickSort` состоит в выборе опорных элементов, как только мы зададим эти варианты, у алгоритма `QuickSort` появится некоторое четко определенное время работы.

**ТЕСТОВОЕ ЗАДАНИЕ Б.3**

Каково математическое ожидание суммы двух бросков кубика?

а) 6,5.

б) 7.

в) 7,5.

(Решение и пояснение см. в разделе Б.4.1.)

Возвращаясь к рандомизированному алгоритму `QuickSort`: насколько велика в среднем длина подмассива, передаваемого первому рекурсивному вызову? Эквивалентным образом, сколько элементов будут меньше случайно выбранного опорного элемента в среднем?

**ТЕСТОВОЕ ЗАДАНИЕ Б.4**

Какое из следующих действий наиболее близко к математическому ожиданию размера подмассива, передаваемого первому рекурсивному вызову в алгоритме `QuickSort`?

а) $\frac{n}{4}$.

б) $\frac{n}{3}$.

в) $\frac{n}{2}$.

г) $\frac{3n}{4}$.

(Решение и пояснение см. в разделе Б.4.2.)

## Б.4.1. Решение тестового задания Б.3

**Правильный ответ: (б).** Есть несколько способов понять, почему математическое ожидание равняется 7. Первый способ — вычислить его методом перебора, используя определяющее уравнение (Б.1). С 36 возможными исходами это выполнимо, но утомительно. Более ловкий способ состоит в том, чтобы объединить в пары возможные значения суммы и использовать симметрию. Сумма равновероятно может составлять 2 или 12, равновероятно составлять 3 или 11, и так далее. В каждой из этих пар среднее значение равно 7, поэтому оно также является средним значением в целом. Третий и лучший способ — использовать линейность математических ожиданий, как описано в следующем разделе.

## Б.4.2. Решение тестового задания В.4

**Правильный ответ: (в).** Точное значение математического ожидания равняется $(n - 1)/2$. Есть $1/n$ шанс, что подмассив имеет длину 0 (если опорный элемент является наименьшим элементом), $1/n$ шанс, что он имеет длину 1 (если опорный элемент является вторым наименьшим элементом), и так далее, вплоть до $1/n$ шанса, что он имеет длину $n - 1$ (если опорный элемент является наибольшим элементом). Принимая во внимание определение (Б.1) математического ожидания[1] и основываясь на тождестве $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$, у нас получается

$$\mathbf{E}[X] = \frac{1}{n}\times 0 + \frac{1}{n}\times 1 + \ldots + \frac{1}{n}\times(n-1) = \frac{1}{n}\times\underbrace{\left(1+2+\ldots+(n-1)\right)}_{=\frac{n(n-1)}{2}} = \frac{n-1}{2}.$$

[1] Один из способов увидеть, что $1 + 2 + \cdots + (n - 1) = \frac{n(n-1)}{2}$, — использовать индукцию по $n$ (см. раздел А.2). Для более ловкого доказательства возьмем две копии из левой части и объединим в пары: «1» из первой копии с «$n - 1$» из второй копии, «2» из первой копии с «$n - 2$» из второй копии, и так далее. Это даст $n - 1$ пар со значением $n$ каждая. Поскольку двойная сумма равняется $n(n - 1)$, исходная сумма равна $\frac{n(n-1)}{2}$.

## Б.5. Линейность математического ожидания

### Б.5.1. Формулировка и варианты использования

Наше заключительное понятие — математическое свойство, а не определение. *Линейность математического ожидания* — это математическое ожидание того, что сумма случайных величин равна сумме их отдельных математических ожиданий. Оно невероятно полезно для вычисления математического ожидания усложненной случайной величины, такой как время работы рандомизированного алгоритма `QuickSort`, по которому случайная величина может быть выражена как взвешенная сумма более простых случайных величин.

**Теорема Б.1 (линейность математического ожидания).** *Пусть $X_1$, ..., $X_n$ есть случайные величины, определенные на том же самом выборочном пространстве $\Omega$, и пусть $a_1$, ..., $a_n$ есть вещественные числа. Тогда*

$$\mathbf{E}\left[\sum_{j=1}^{n} a_j \times X_j\right] = \sum_{j=1}^{n} a_j \times \mathbf{E}\left[X_j\right]. \qquad \text{(Б.2)}$$

То есть вы можете взять сумму и математическое ожидание в любом порядке и получить то же самое. Общий случай использования — это когда $\sum_{j=1}^{n} a_j X_j$ является сложной случайной величиной (например, время работы рандомизированного алгоритма `QuickSort`), а $X_j$ являются простыми случайными величинами (например, случайные величины 0–1)[1].

Например, пусть $X$ есть сумма двух стандартных костей. Мы можем записать $X$ как сумму двух случайных величин, $X_1$ и $X_2$, которые являются значениями соответственно первой и второй костей. Математическое ожидание $X_1$ или $X_2$ легко вычислить, использовав определение (Б.1), как $\frac{1}{6}(1+2+3+4+5+6) = 3{,}5$. Линейность математического ожидания тогда дает

$$\mathbf{E}\left[X\right] = \mathbf{E}\left[X_1\right] + \mathbf{E}\left[X_2\right] = 3{,}5 + 3{,}5 = 7,$$

повторяя наш ответ в разделе В.4.1 с меньшим объемом работы.

[1] В стэнфордской версии этого курса за десять недель лекций на доске я очерчиваю квадрат вокруг только одного тождества — линейности математического ожидания.

Чрезвычайно важным моментом является то, что *линейность математического ожидания справедлива даже для случайных величин, которые не являются независимыми*. Нам не нужно будет формально определять понятие независимости в этой книге, но у вас, вероятно, есть хорошее интуитивное понимание того, что это значит: знание чего-то о значении одной случайной величины не говорит вам ничего нового о значениях других. Например, приведенные выше случайные величины $X_1$ и $X_2$ независимы, поскольку предполагается, что две кости бросают независимо друг от друга.

Для примера зависимых случайных величин рассмотрим пару магнетически связанных костей, где вторая кость всегда оказывается со значением большим, чем у первой (или 1, если первая кость показывает 6). Теперь, зная значение любой из них, вы точно знаете, каково значение другой кости. Но мы по-прежнему можем записать сумму $X$ двух костей как $X_1 + X_2$, где $X_1$ и $X_2$ являются значениями двух костей. По-прежнему сохраняется вариант, что $X_1$, рассматриваемый изолированно, равновероятно будет любым из {1, 2, 3, 4, 5, 6}, и то же самое верно для $X_2$. Следовательно, у нас по-прежнему получается $\mathbf{E}[X_1] = \mathbf{E}[X_2] = 3{,}5$, и по линейности математического ожидания у нас по-прежнему получается $\mathbf{E}[X] = 7$.

Почему вы должны удивляться? Внешне тождество в (Б.2) может выглядеть как тавтология. Но если мы перейдем от сумм к произведениям случайных величин, то аналог теоремы Б.1 больше не будет соблюдаться для зависимых случайных величин[1]. Поэтому линейность математического ожидания действительно является частным свойством сумм случайных величин.

[1] Магнетически связанные кости обеспечивают один контрпример. В качестве еще более простого контрпримера предположим, что $X_1$ и $X_2$ либо равны 0 и 1, либо 1 и 0, причем каждый результат имеет 50 %-ную вероятность. Тогда $\mathbf{E}[X_1 \times X_2] = 0$, а $\mathbf{E}[X_1] \times \mathbf{E}[X_2] = \frac{1}{4}$.

## Б.5.2. Доказательство

Полезность линейности математического ожидания сопоставима по простоте только с его доказательством[1].

*Доказательство теоремы Б.1*. Начав с правой части (Б.2) и разложив ее с помощью определения (Б.1) математического ожидания, получаем

$$\sum_{j=1}^{n} a_j \times \mathbf{E}\left[X_j\right] = \sum_{j=1}^{n} a_j \times \left(\sum_{i\in\Omega} p(i) \times X_j(i)\right)$$
$$= \sum_{j=1}^{n} \left(\sum_{i\in\Omega} a_j \times p(i) \times X_j(i)\right)$$

Изменив порядок суммирования, мы имеем

$$\sum_{j=1}^{n} \left(\sum_{i\in\Omega} a_j \times p(i) \times X_j(i)\right) = \sum_{i\in\Omega} \left(\sum_{j=1}^{n} a_j \times p(i) \times X_j(i)\right). \quad \text{(Б.3)}$$

Поскольку вероятность $p(i)$ не зависит от $j = 1, 2, ..., n$, мы можем вытащить ее из внутренней суммы:

$$\sum_{i\in\Omega} \left(\sum_{j=1}^{n} a_j \times p(i) \times X_j(i)\right) = \sum_{i\in\Omega} p(i) \times \left(\sum_{j=1}^{n} a_j \times X_j(i)\right).$$

Наконец, снова используя определение (Б.1) математического ожидания, мы получаем левую часть (Б.2):

$$\sum_{i\in\Omega} p(i) \times \left(\sum_{j=1}^{n} a_j \times X_j(i)\right) = \mathbf{E}\left[\sum_{j=1}^{n} a_j \times X_j\right]$$

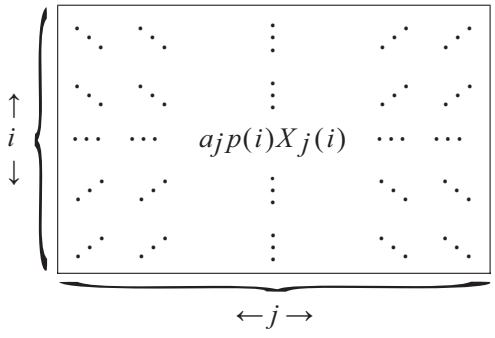
*Ч. т. д.*

Вот и все! Линейность математического ожидания — это просто обратный порядок двойного суммирования.

Говоря о двойном суммировании, уравнение (Б.3) может показаться непрозрачным, если вы подзабыли эти виды алгебраических преобразований. Чтобы

[1] При первом чтении этого доказательства для простоты следует принять, что $a_1 = a_2 = \cdots = a_n = 1$.

думать о нем более приземленно, организуйте $a_j p(i) X_j(i)$ в решетке, в которой строки индексированы по $i \in \Omega$, столбцы индексированы по $j \in \{1, 2, ..., n\}$ и число $a_j p(i) X_j(i)$ находится в ячейке $i$-й строки и $j$-го столбца:




Левая часть (Б.3) сначала суммирует каждый столбец, а затем складывает эти суммы столбцов. Правая часть сначала суммирует строки, а затем складывает эти суммы строк. В том и в другом случае вы получите сумму всех значений в решетке.

## Б.6. Пример: распределение нагрузки

Чтобы связать воедино все предыдущие понятия, рассмотрим пример распределения нагрузки. Предположим, нам нужен алгоритм, который назначает процессы серверам, но мы — суперленивы. Одно из самых простых решений — просто назначить каждый процесс случайному серверу, причем каждый сервер одинаково вероятен. Насколько хорошо это работает[1]?

Для конкретности примем, что есть $n$ процессов, а также $n$ серверов, где $n$ — некоторое положительное целое число. Во-первых, давайте проясним выборочное пространство: множество $\Omega$ — это множество всех $n^n$ возможных способов назначения процессов серверам $n$ вариантами для каждого из $n$ процессов. По определению нашего ленивого алгоритма каждый из этих $n^n$ исходов одинаково вероятен.

[1] Этот пример также имеет отношение к нашему обсуждению хеширования в части 2.

Теперь, когда у нас есть выборочное пространство, мы можем определить случайные величины. Одна интересная величина — это нагрузка на сервер, поэтому определим $Y$ как случайную величину, равную количеству процессов, которые назначаются первому серверу. (История одинакова для всех серверов в силу симметрии, поэтому мы вполне можем сосредоточиться на первом.) Каково математическое ожидание $Y$?

В принципе, мы можем вычислить $\mathbf{E}[Y]$ путем оценки определяющего уравнения (Б.1) методом перебора, но это непрактично для всех значений, кроме наименьших значений $n$. К счастью, поскольку $Y$ может быть выражена как сумма простых случайных величин, линейность математического ожидания может сэкономить нам день. Формально для $j = 1, 2, ..., n$, определим

$$X_J = \begin{cases} 1 & \text{если } j\text{-й процесс приваивается первому серверу} \\ 0 & \text{в противном случае} \end{cases}$$

Случайные величины, которые принимают только значения 0 и 1, часто называются индикаторными случайными величинами, поскольку они показывают, не происходит ли некоторое событие (например, событие — процесс $j$ присваивается первому серверу).

Из определений мы можем выразить $Y$ как сумму значений $X_j$:

$$Y = \sum_{j=1}^{n} X_j .$$

Согласно линейности ожиданий (теорема Б.1) математическое ожидание $Y$ является суммой математических ожиданий $X_j$:

$$\mathbf{E}\left[Y\right] = \mathbf{E}\left[\sum_{j=1}^{n} X_j\right] = \sum_{j=1}^{n} \mathbf{E}\left[X_j\right].$$

Поскольку каждая случайная переменная $X_j$ такая простая, легко вычислить ее математическое ожидание напрямую:

$$\mathbf{E}\left[X_j\right] = \underbrace{0 \times \mathbf{Pr}\left[X_j = 0\right]}_{=0} + 1 \times \mathbf{Pr}\left[X_j = 1\right] = \mathbf{Pr}\left[X_j = 1\right].$$

Поскольку $j$-й процесс равновероятно может быть назначен каждому из $n$ серверов, $\mathbf{Pr}\left[X_j = 1\right] = \frac{1}{n}$. Собрав все вместе, у нас получается

$$\mathbf{E}\left[Y\right] = \sum_{j=1}^{n} \mathbf{E}\left[X_j\right] = n \times \frac{1}{n} = 1.$$

Таким образом, если нас интересуют только средние нагрузки на серверы, то наш сверхленивый алгоритм работает просто отлично! Этот пример и рандомизированный алгоритм `QuickSort` показывают, какую роль играет рандомизация в проектировании алгоритмов: мы часто можем обойтись действительно простой эвристикой, если в процессе делаем случайный выбор.

**ТЕСТОВОЕ ЗАДАНИЕ Б.5**

Рассмотрим группу из $k$ человек. Допустим, что день рождения каждого человека вынимается равномерно случайным образом из 365 возможностей. (И проигнорируем високосные годы.) Какое наименьшее значение $k$ такое, что ожидаемое число пар разных людей с одним и тем же днем рождения равно как минимум единице? [Подсказка: определите индикаторную случайную величину для каждой пары людей. Используйте линейность математического ожидания.]

а) 20.

б) 23.

в) 27.

г) 28.

д) 366.

(Решение и пояснение см. ниже.)

**Правильный ответ: (г).** Зададим положительное целое число $k$ и обозначим множество людей как $\{1, 2, ..., k\}$. Обозначим через $Y$ количество пар людей с одинаковым днем рождения. Как предлагается в подсказке, определим одну случайную величину $X_{ij}$ для каждого варианта $i, j \in \{1, 2, ..., k\}$ людей, где $i < j$. Определим $X_{ij}$ как 1, если $i$ и $j$ имеют одинаковый день рождения, и 0

в противном случае. Следовательно, $X_{ij}$ являются индикаторными случайными переменными, и

$$Y = \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} X_{ij}.$$

По линейности математического ожидания (теорема Б.1)

$$\mathbf{E}\left[Y\right] = \mathbf{E}\left[\sum_{i=1}^{k-1} \sum_{j=i+1}^{k} X_{ij}\right] = \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \mathbf{E}\left[X_{ij}\right]. \tag{Б.4}$$

Поскольку $X_{ij}$ — это индикаторная случайная величина, $\mathbf{E}[X_{ij}] = \mathbf{Pr}[X_{ij} = 1]$. Имеется $(365)^2$ возможностей для дней рождения людей $i$ и $j$, и в 365 из этих возможностей $i$ и $j$ имеют один и тот же день рождения.

Принимая, что все комбинации дней рождения равновероятны,

$$\mathbf{Pr}\left[X_{ij} = 1\right] = \frac{365}{(365)^2} = \frac{1}{365}.$$

Подставив это в (Б.4), мы имеем

$$\mathbf{E}\left[Y\right] = \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \frac{1}{365} = \frac{1}{365} \times \binom{k}{2} = \frac{k(k-1)}{730},$$

где $\binom{k}{2}$ обозначает биномиальный коэффициент «из 2 по $k$» (как в решении тестового задания 3.1). Наименьшее значение $k$, для которого $k(k-1)/730 \geq 1$, равно 28.

*Тим Рафгарден*

**Совершенный алгоритм. Основы**

Перевел с английского *А. Логунов*


| | |
|---|---|
| Заведующая редакцией | *Ю. Сергиенко* |
| Ведущий редактор | *К. Тульцева* |
| Научный редактор | *П. Ковалёв* |
| Литературные редакторы | *А. Бульченко, И. Маланыч* |
| Художественный редактор | *В. Мостипан* |
| Корректоры | *С. Беляева, И. Тимофеева* |
| Верстка | *Л. Егорова* |

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 12.2018. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214,
тел./факс: 208 80 01.

Подписано в печать 12.12.18. Формат 70×100/16. Бумага офсетная. Усл. п. л. 20,640. Тираж 1200. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

**ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает профессиональную, популярную и детскую развивающую литературу**

**Заказать книги оптом можно в наших представительствах**

**РОССИЯ**

**Санкт-Петербург:** м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

**Москва:** м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

**Воронеж:** тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

**Екатеринбург:** ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;
e-mail: office@ekat.piter.com; skype: ekat.manager2

**Нижний Новгород:** тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

**Ростов-на-Дону:** ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

**Самара:** ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,
pitvolga@samara-ttk.ru

**БЕЛАРУСЬ**

**Минск:** ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;
e-mail: og@minsk.piter.com

***Издательский дом «Питер» приглашает к сотрудничеству авторов:***
*тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanovaa@piter.com*
*Подробная информация здесь: http://www.piter.com/page/avtoru*

***Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок:*** *тел./факс: (812) 703-73-73; e-mail: sales@piter.com*

***Заказ книг для вузов и библиотек:***
*тел./факс: (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com*

***Заказ книг по почте:*** *на сайте www.piter.com; тел.: (812) 703-73-74, доб. 6216; e-mail: books@piter.com*

***Вопросы по продаже электронных книг:*** *тел.: (812) 703-73-74, доб. 6217; e-mail: kuznetsov@piter.com*

# КНИГА-ПОЧТОЙ

**ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:**

- на нашем сайте: **www.piter.com**
- по электронной почте: **books@piter.com**
- по телефону: **(812) 703-73-74**

**ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:**

- Наложенным платежом с оплатой при получении в ближайшем почтовом отделении.
- С помощью банковской карты. Во время заказа вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
- Электронными деньгами. Мы принимаем к оплате Яндекс.Деньги, Webmoney и Qiwi-кошелек.
- В любом банке, распечатав квитанцию, которая формируется автоматически после совершения вами заказа.

**ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ДОСТАВКИ:**

- Посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку ваших покупок максимально быстро. Дату отправления вашей покупки и дату доставки вам сообщат по e-mail.
- Вы можете оформить курьерскую доставку своего заказа (более подробную информацию можно получить на нашем сайте www.piter.com).
- Можно оформить доставку заказа через почтоматы,(адреса почтоматов можно узнать на нашем сайте www.piter.com).

**ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:**

- фамилию, имя, отчество, телефон, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

**БЕСПЛАТНАЯ ДОСТАВКА:**

- курьером по Москве и Санкт-Петербургу при заказе на сумму **от 2000 руб.**
- почтой России при предварительной оплате заказа на сумму **от 2000 руб.**