БИБЛИОТЕКА ПРОГРАММИСТА

Вильям Спрингер
William M. Springer II

ГИД

ПО COMPUTER SCIENCE

ДЛЯ КАЖДОГО ПРОГРАММИСТА

A Programmer's Guide to Computer Science

William M. Springer II, PhD

БИБЛИОТЕКА ПРОГРАММИСТА

Вильям Спрингер
William M. Springer II

ГИД ПО COMPUTER SCIENCE

ДЛЯ КАЖДОГО ПРОГРАММИСТА

ПИТЕР®
Санкт-Петербург · Москва · Екатеринбург · Воронеж
Нижний Новгород · Ростов-на-Дону · Самара · Минск

2020

ББК 32.973.2-018
УДК 004.3
С74

Спрингер Вильям

С74 Гид по Computer Science для каждого программиста. — СПб.: Питер, 2020. — 192 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1674-4

Колосс на глиняных ногах — так можно назвать программиста без подготовки в области Computer Science. Уверенное владение основами позволяет «не изобретать велосипеды» и закладывать в архитектуру программ эффективные решения. Все это избавляет от ошибок и чрезмерных затрат на тестирование и рефакторинг.

Не беда, если вы чувствуете себя не у дел, когда другие программисты обсуждают аппроксимативный предел. Даже специалисты с опытом допускают ошибки из-за того, что подзабыли Computer Science.

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

ББК 32.973.2-018
УДК 004.3

ISBN 978-1951204006 англ.
ISBN 978-5-4461-1674-4

Оглавление

Часть II. Графы и графовые алгоритмы

Часть V. Теория сложности вычислений

Введение

Зачем нужна эта книга

Многие из моих знакомых разработчиков пришли в профессию из самых разных областей. У одних — высшее образование в области Computer Science; другие изучали фотографию, математику или даже не окончили университет.

В последние годы я заметил, что программисты все чаще стремятся изучить Computer Science по ряду причин:

- чтобы стать хорошими программистами;
- чтобы на собеседованиях отвечать на вопросы про алгоритмы;
- чтобы удовлетворить свое любопытство в области Computer Science или наконец перестать сожалеть о том, что в свое время у них не было возможности освоить этот предмет.

Эта книга для всех вас.

Многие найдут здесь темы, интересные сами по себе. Я попытался показать, в каких реальных (неакадемических) ситуациях эти знания будут полезны. Хочу, чтобы,

прочитав эту книгу, вы получили такие же знания, как после изучения базового курса по Computer Science, а также научились их применять.

Проще говоря, цель этой книги — помочь вам стать более квалифицированным и опытным программистом благодаря лучшему пониманию Computer Science. Мне не под силу втиснуть в одну книгу 20-летний стаж преподавания в колледже и профессиональный опыт... однако я постараюсь сделать максимум того, на что способен. Надеюсь, что вы найдете здесь хотя бы одну тему, о которой сможете сказать: «Да, теперь мне это понятно», — и применить знания в своей работе.

Чего вы не найдете в издании

Смысл книги состоит в том, чтобы читатель смог лучше понимать Computer Science и применять знания на практике, а вовсе не в том, чтобы полностью заменить четыре года обучения.

В частности, это не книга с доказательствами. Действительно, во втором томе книги[1] рассмотрены методы доказательства, однако стандартные алгоритмы обычно приводятся здесь без доказательств. Идея в том, чтобы читатель узнал о существовании этих алгоритмов и научился их использовать, не вникая в подробности. В качестве книги с доказательствами, написанной для студентов и аспирантов, я настоятельно рекомендую

[1] *Springer II W. M., Springer B.* A Programmer's Guide to Computer Science. Vol. 2: A virtual degree for the self-taught developer, 2020.

Introduction to Algorithms[1] («Алгоритмы. Вводный курс») Кормена (Cormen), Лейзерсона (Leiserson), Ривеста (Rivest) и Стейна (Stein) (авторов обычно объединяют под аббревиатурой CLRS).

Это и не книга по программированию: вы не найдете здесь рекомендаций, когда использовать числа типа `int`, а когда — `double`, или объяснений, что такое цикл. Предполагается, что читатель сможет разобраться в листингах на псевдокоде, используемых для описания алгоритмов[2]. Цель книги — связать концепции Computer Science с уже знакомыми читателю методами программирования.

Дополнительные ресурсы

Для тех, кто хочет подробнее изучить ту или иную тему, я включил в сноски ссылки на дополнительные материалы. Кроме того, по адресу http://www.whatwilliamsaid.com/books/ вы найдете тесты для самопроверки к каждой главе.

---

[1] *Cormen T. H., Leiserson C. E., Rivest R. L., Stein C.* Introduction to Algorithms, 3rd Edition. The MIT Press, 2009.

[2] Все программы в этой книге написаны на псевдокоде на основе языка C.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Часть I
Основы Computer Science

1 Асимптотическое время выполнения

1.1. Что такое алгоритм

Предположим, вам нужно научить робота делать бутерброд с арахисовым маслом (рис. 1.1). Ваши инструкции могут быть примерно такими.

1. Открыть верхнюю левую дверцу шкафчика.
2. Взять банку с арахисовым маслом и вынуть ее из шкафчика.
3. Закрыть шкафчик.
4. Держа банку с арахисовым маслом в левой руке, взять крышку в правую руку.
5. Поворачивать правую руку против часовой стрелки, пока крышка не откроется.
6. И так далее...

Это программа: вы описали каждый шаг, который компьютер должен выполнить, и указали всю информацию, которая требуется компьютеру для выполнения каждого шага. А теперь представьте, что вы объясняете

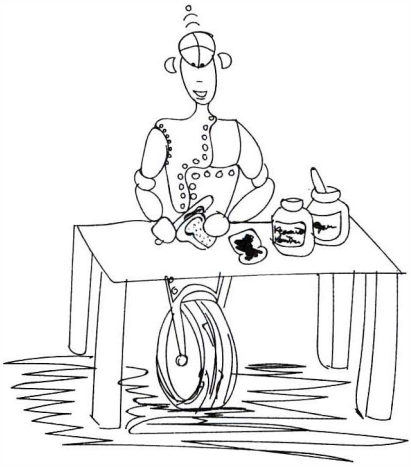


Рис. 1.1. Робот — изготовитель бутербродов

человеку, как сделать бутерброд с арахисовым маслом. Ваши инструкции будут, скорее всего, такими.

1. Достаньте арахисовое масло, джем и хлеб.
2. Намажьте ножом арахисовое масло на один ломтик хлеба.
3. Намажьте ложкой джем на второй ломтик хлеба.
4. Сложите два ломтика вместе. Приятного аппетита!

Это алгоритм: процесс, которому нужно следовать для получения желаемого результата (в данном случае бутерброда с арахисовым маслом и джемом). Обратите внимание, что алгоритм более абстрактный, чем программа. Программа сообщает роботу, откуда именно нужно взять предметы на конкретной кухне, с точным указанием всех необходимых деталей. Это реализация алгоритма, которая предоставляет все важные детали,

но может быть выполнена на любом оборудовании (в данном случае — на кухне), со всеми необходимыми элементами (арахисовое масло, джем, хлеб и столовые приборы).

1.2. Почему скорость имеет значение

Современные компьютеры достаточно быстры, поэтому во многих случаях скорость алгоритма не особенно важна. Когда я нажимаю кнопку и компьютер реагирует за 1/25 секунды, а не за 1/100 секунды, эта разница для меня не имеет значения — с моей точки зрения, компьютер в обоих случаях реагирует мгновенно.

Но во многих приложениях скорость все еще важна, например, при работе с большим количеством объектов. Предположим, что у вас есть список из миллиона элементов, который необходимо отсортировать. Эффективная сортировка занимает одну секунду, а неэффективная может длиться несколько недель. Возможно, пользователь не захочет ждать, пока она закончится.

Мы часто считаем задачу неразрешимой, если не существует известного способа ее решения за разумные сроки, где «разумность» зависит от различных реальных факторов. Например, безопасность шифрования данных часто зависит от сложности разложения на множители (факторизации) больших чисел. Если я отправляю вам зашифрованное сообщение, содержимое которого нужно хранить в секрете в течение недели, то для меня не имеет значения, что злоумышленник перехватит это сообщение и расшифрует его через три года. Задача

не является неразрешимой — просто наш любитель подслушивать не знает, как решить ее достаточно быстро, чтобы решение было полезным.

Важным навыком в программировании является умение оптимизировать только те части программы, которые необходимо оптимизировать. Если пользовательский интерфейс работает на одну тысячную секунды медленнее, чем мог бы, это никого не волнует — в данном случае мы предпочли бы незаметному увеличению скорости удобочитаемую программу. А вот код, расположенный внутри цикла, который может выполняться миллионы раз, должен быть написан максимально эффективно.

1.3. Когда секунды (не) считаются

Рассмотрим алгоритм приготовления бутербродов из раздела 1.1. Поскольку мы хотим, чтобы этот алгоритм можно было использовать для любого количества различных роботов — изготовителей бутербродов, мы не хотим измерять количество секунд, затрачиваемое на выполнение алгоритма, поскольку для разных роботов оно будет различаться. Один робот может дольше открывать шкафчик, но быстрее вскрывать банку с арахисовым маслом, а другой — наоборот.

Вместо того чтобы измерять фактическую скорость выполнения каждого шага, которая будет различаться для разных роботов и кухонь, лучше подсчитать (и минимизировать) количество шагов. Например, алгоритм, который требует, чтобы робот сразу брал и нож и ложку, эффективнее, чем алгоритм, в котором робот открывает

ящик со столовыми приборами, берет нож, закрывает ящик, кладет нож на стол, снова открывает ящик, достает ложку и т. д.

Измерение времени: алгоритм или программа?

Напомню, что алгоритмы являются более обобщенными, чем программы. Для нашего алгоритма приготовления бутербродов мы хотим посчитать число шагов. Если бы мы действительно использовали конкретного робота — изготовителя бутербродов, то нас бы больше интересовало точное время, которое требуется для изготовления каждого бутерброда.

Следует признать, что не все шаги потребуют одинакового времени выполнения; скорее всего, взять нож — быстрее, чем намазать арахисовое масло на хлеб. Поэтому мы хотим не столько узнать точное количество шагов, сколько иметь представление о том, сколько шагов потребуется в зависимости от размера входных данных. В случае с нашим роботом время, необходимое для приготовления бутерброда, при увеличении количества бутербродов не увеличивается (при условии, что нам хватит джема).

Два компьютера могут выполнять алгоритм с разной скоростью. Это зависит от их тактовой частоты, объема доступной памяти, количества тактовых циклов, требуемого для выполнения каждой инструкции, и т. д. Однако обоим компьютерам, как правило, требуется приблизительно одинаковое число инструкций, и мы можем измерить скорость, с которой количество требуемых инструкций увеличивается в зависимости от размера задачи (рис. 1.2). Например, при сортировке

массива чисел, если увеличить его размер в тысячу раз, один алгоритм может потребовать в тысячу раз больше команд, а второй — в миллион раз больше[1].

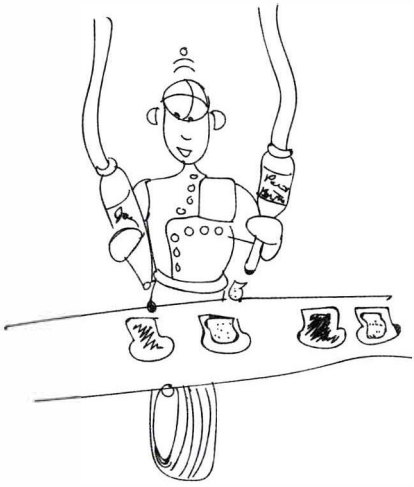


Рис. 1.2. Более эффективный робот — изготовитель бутербродов

Часто мы хотим измерить скорость алгоритма несколькими способами. В жизненно важных ситуациях — например, при запуске двигателей на зонде, который приземляется на Марсе, — мы хотим знать время выполнения при самом неблагоприятном раскладе. Мы можем выбрать алгоритм, который в среднем работает немного медленнее, но зато гарантирует, что его выполнение никогда не займет больше времени, чем мы считаем приемлемым (и наш зонд не разобьется вместо того, чтобы приземлиться). Для более повседневных сценариев мы

[1] Конкретные примеры см. в главе 8.

можем смириться со случайными всплесками времени выполнения, если при этом среднее время не растет; например, в видеоигре мы бы предпочли генерировать результаты в среднем быстрее, смирившись с необходимостью время от времени прерывать длительные вычисления. А бывают случаи, когда мы хотели бы знать наилучшую производительность. Однако в большинстве ситуаций мы просто рассчитываем наихудшее время выполнения, которое все равно часто совпадает со средним временем выполнения.

1.4. Как мы описываем скорость

Предположим, у нас есть два списка целых чисел, которые мы хотим отсортировать: {1, 2, 3, 4, 5, 6, 7, 8} и {3, 5, 4, 1, 2}. Сортировка какого списка займет меньше времени?

Ответ на вопрос: неизвестно. Для одного алгоритма сортировки тот факт, что первый список уже отсортирован, позволит почти сразу завершить работу. Для другого алгоритма решающим фактором может быть то, что второй список короче. Но обратите внимание, что оба свойства входных данных — размер списка и последовательность расположения чисел — влияют на количество шагов, необходимых для сортировки.

Если некоторое свойство, например «отсортированность», изменяет время выполнения конкретного алгоритма только на постоянную величину, мы можем просто его игнорировать, поскольку его влияние незаметно по сравнению с влиянием размера задачи. Например, робот может делать бутерброды с виноградным или

малиновым джемом, но банка с малиновым джемом открывается немного дольше. Если робот делает миллион бутербродов, то влияние выбора джема на время приготовления бутерброда незаметно на фоне количества бутербродов. Поэтому мы можем его игнорировать и просто сказать, что приготовление миллиона бутербродов занимает примерно в миллион раз больше времени, чем приготовление одного бутерброда.

С точки зрения математики мы вычисляем *асимптотическое* время выполнения алгоритма, то есть скорость увеличения времени выполнения в зависимости от размера входных данных. Нашему роботу, который делает бутерброды, требуется некоторое постоянное время c для приготовления одного бутерброда и в n раз больше времени, то есть cn, чтобы сделать n бутербродов. Мы отбрасываем константу и говорим, что наш алгоритм приготовления бутербродов занимает время $O(n)$ (произносится как «O большое от n» или просто «O от n»). Это означает, что время выполнения в худшем случае пропорционально количеству бутербродов, которое будет сделано. Нас интересует не точное количество необходимых шагов, а скорость, с которой это число увеличивается по мере роста размера задачи (в данном случае — количества бутербродов).

1.5. Скорость типичных алгоритмов

Сколько бы бутербродов ни делал наш робот, это не увеличивает время, необходимое для изготовления одного бутерброда. Это *линейный* алгоритм — общее время выполнения пропорционально количеству обрабатываемых элементов. Для большинства задач это лучшее, чего

можно добиться[1]. Типичный пример линейного алгоритма в программировании — чтение списка элементов и выполнение некоторой задачи для каждого элемента списка: время, затрачиваемое на обработку каждого элемента, не зависит от других элементов. Есть цикл, который выполняет постоянный объем работы и осуществляется один раз для каждого из n элементов, поэтому все вместе занимает $O(n)$ времени.

Но чаще количество элементов списка влияет на объем работы, которую необходимо выполнить для отдельного элемента. Алгоритм сортировки может обрабатывать каждый элемент списка, разделяя список на два меньших списка, и повторять это до тех пор, пока все элементы не окажутся в своем собственном списке. На каждой итерации алгоритм выполняет $O(n)$ операций и требует $O(\lg n)$ итераций, что в общей сложности составляет $O(n) \times O(\lg n) = O(n \lg n)$ времени.

Математическое предупреждение

Логарифм описывает, в какую степень необходимо возвести число, указанное в основании, чтобы получить желаемое значение. Например, $\log_{10} 1000 = 3$, так как $10^3 = 1000$.

В компьютерах мы часто делим на 2, поэтому обычно используются логарифмы по основанию 2. Сокращенно $\log_2 n$ обозначается как $\lg n$.[2] Таким образом, $\lg 1 = 0$, $\lg 2 = 1$, $\lg 4 = 2$, $\lg 8 = 3$ и т. д.

---

[1] Поскольку n — это размер входных данных, то в общем случае только для их чтения требуется время $O(n)$.

[2] Строго говоря, lg — это логарифм по основанию 10, но часто именно в Computer Science принимают, что это логарифм по основанию 2. — *Примеч. науч. ред.*

С этого момента время выполнения начинает ухудшаться. Рассмотрим алгоритм, в котором каждый элемент множества сравнивается со всеми остальными элементами множества, — здесь мы выполняем работу, занимающую $O(n)$ времени, $O(n)$ раз, то есть общее время выполнения алгоритма — $O(n^2)$. Это *квадратичное* время.

Все алгоритмы, у которых время выполнения пропорционально количеству входных данных, возведенному в некоторую степень, называются *полиномиальными* алгоритмами; такие алгоритмы принято считать быстрыми. Конечно, алгоритм, решение которого пропорционально количеству входных данных в сотой степени, хоть и является полиномиальным, все же не будет считаться быстрым даже при большом воображении! Однако на практике задачи, о которых известно, что они имеют полиномиальное время решения, как правило, решаются за *биквадратное* (четвертой степени) время или еще быстрее.

Это не означает, что асимптотически более эффективный алгоритм всегда будет работать быстрее, чем асимптотически менее эффективный. Например, ваш жесткий диск вышел из строя и вы потеряли несколько важных файлов. К счастью, вы сделали их резервную копию в Сети[1].

Вы можете загрузить файлы из облака со скоростью 10 Мбит/с (при условии, что у вас хорошее соединение).

---

[1] В этом примере я использовал цифры для резервного копирования из облака Carbonite, но есть и много других. Это ни в коем случае не реклама, а всего лишь первый попавшийся сервис, который я обнаружил при поиске.

Это линейное время — количество времени, которое требуется для извлечения всех потерянных файлов, и оно более или менее прямо пропорционально зависит от размера файлов. Служба резервного копирования также предлагает загрузить файлы на внешний диск и отправить их вам — это действие занимает постоянное время, или $O(1)$, потому что время получения этих данных не зависит от их размера[1]. Если речь идет о восстановлении всего нескольких мегабайтов, то загрузить их напрямую будет намного быстрее. Но, как только файл достигнет такого размера, что его загрузка будет занимать столько же времени, сколько и отправка, удобнее будет использовать внешний диск.

Если полиномиальные алгоритмы быстрые, то какие же алгоритмы медленные? Для некоторых алгоритмов (называемых *экспоненциальными*) число операций ограничено не размером входных данных, возведенным в некоторую постоянную степень, а константой, возведенной в степень, равную размеру входных данных.

В качестве примера рассмотрим попытку угадать числовой код доступа длиной n символов. Если это десять

---

[1] Ну хорошо, почти не зависит — это занимает 1–3 рабочих дня. «Не зависит» в данном случае означает не то, что некая операция всегда занимает одинаковое время, а лишь то, что время выполнения не зависит от количества входных данных. Мы также предполагаем, что компания, занимающаяся резервным копированием, может подготовить вашу копию достаточно быстро, так что они не пропустят отправку почты из-за очень большого размера файла.

цифр от 0 до 9, то количество возможных кодов составляет 10^n. Обратите внимание, что это число растет **намного** быстрее, чем n^{10}: если n равно всего 20, полиномиальный алгоритм работает уже почти в 10 миллионов раз быстрее (рис. 1.3)!

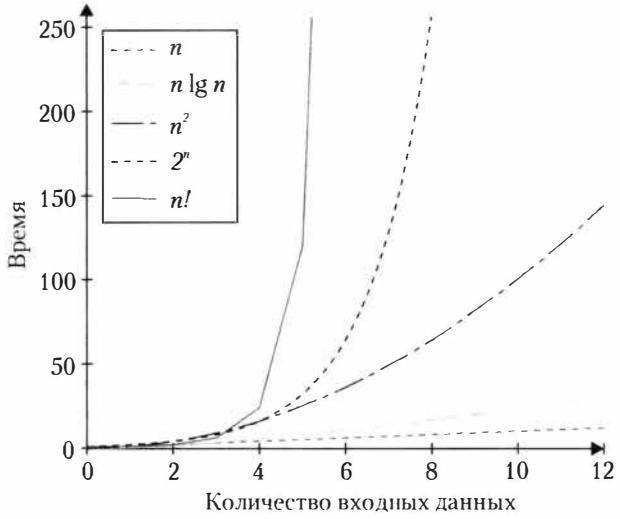


Рис. 1.3. Даже при небольшом количестве входных данных различия в асимптотическом времени выполнения быстро становятся заметными

1.6. Всегда ли полиномиальное время лучше?

Как правило, специалисты по Computer Science заинтересованы получить полиномиальное решение задачи, особенно если оно выполняется за квадратичное время или еще быстрее. Однако для задач разумного

(небольшого) размера экспоненциальные алгоритмы также могут быть приемлемы.

Зачастую мы можем найти приближенное решение задачи за полиномиальное время, однако получить точный (или близкий к точному) ответ можем лишь за экспоненциальное время. Рассмотрим в качестве примера задачу коммивояжера: продавец хотел бы посетить каждый город на своем маршруте ровно один раз и вернуться домой, преодолев минимальное расстояние. (Представьте себе, сколько денег сэкономили бы службы доставки UPS и FedEx, если бы сделали свои маршруты всего лишь немного более эффективными!)

Чтобы получить точный ответ, нужно вычислить все возможные маршруты и сравнить их суммарные расстояния, то есть $O(n!)$ возможных путей. Очень близкое к оптимальному (в пределах до 1 %) решение может быть найдено за экспоненциальное время[1]. Но возможное «достаточно хорошее» (в пределах 50 % от оптимального) решение может быть найдено за полиномиальное время[2]. Это обычный компромисс: мы можем быстро получить достаточно хорошее приближение или медленнее — более точный ответ.

---

[1] Подробный обзор различных подходов вы найдете в статье: *Applegate D. L., Bixby R. E., Chvátal V., Cook W. J.* The Traveling Salesman Problem.

[2] На момент написания этой книги были известны алгоритмы, позволяющие найти решение хуже оптимального не более чем на 50 % за время $O(n^3)$. См., например, статью: *Sebö A., Vygen J.* Shorter Tours by Nicer Ears, 2012.

1.7. Время выполнения алгоритма

Рассмотрим следующий код:

```
foreach (name in NameCollection)
{
    Print "Hello, {name}!";
}
```

Здесь у нас есть коллекция из n строк; для каждой строки выводится короткое сообщение. Вывод сообщения занимает постоянное время[1], то есть $O(1)$. Мы делаем это n раз, то есть $O(n)$. Умножив одно на другое, получим результат: время выполнения кода составляет $O(n)$.

А теперь рассмотрим другую функцию:

```
DoStuff (numbers)
{
    sum = 0;
    foreach (num in numbers)
    {
        sum += num;
    }
    product = 1;
    foreach (num in numbers)
    {
        product *= num;
    }
    Print "The sum is {sum} and the
           product is {product}";
}
```

[1] Это не означает, что для каждой строки требуется одинаковое время; время, необходимое для вывода каждой строки, не зависит от количества строк.

Здесь есть два цикла; каждый из них состоит из $O(n)$ итераций и на каждой итерации выполняет постоянную работу, то есть общее время выполнения каждого цикла составляет $O(n)$. Сложив $O(n)$ и $O(n)$ и отбросив константу, мы снова получим не $O(2n)$, а $O(n)$. Представленную выше функцию также можно написать так:

```
DoStuff (numbers)
{
    sum = 0 , product = 1;
    foreach (num in numbers)
    {
        sum += num;
        product *= num;
    }
    Print "The sum is {sum} and the
           product is {product}";
}
```

Теперь у нас есть только один цикл, который снова выполняет постоянную работу; просто у него константа больше, чем раньше. Помните, что нас интересует, насколько быстро растет время выполнения задачи при увеличении ее размера, а не точное число операций для данного размера задачи.

Теперь попробуем что-нибудь более сложное. Каково время выполнения этого алгоритма?

```
CountInventory (stuffToSell, colorList)
{
    totalItems = 0;
    foreach (thing in stuffToSell)
    {
        foreach (color in colorList)
```

```
        {
            totalItems += thing[color];
        }
    }
}
```

Здесь у нас есть два цикла, причем один вложен в другой. Поэтому мы будем умножать их время выполнения, а не складывать. Внешний цикл запускается один раз для каждого элемента каталога, а внутренний — один раз для каждого предоставленного цвета. Для n элементов и m цветов общее время выполнения равно $O(nm)$. Обратите внимание, что это **не** $O(n^2)$; у нас нет оснований полагать, что между n и m существует взаимосвязь.

Рассмотрим еще одну функцию:

```
doesStartWith47 (numbers)
{
    return (numbers[0] == 47);
}
```

Эта функция проверяет, равен ли 47 первый элемент целочисленного массива, и возвращает результат. Объем работы, который выполняет функция, не зависит от количества входных данных и поэтому равен $O(1)$[1].

Мы часто пишем программы, которые включают в себя двоичный поиск, следовательно, в нашем анализе будут логарифмы.

---

[1] Здесь, конечно, предполагается, что массив передается по ссылке; если массив передается по значению, то время выполнения составит $O(n)$.

Например, рассмотрим следующий код[1]:

```
binarySearch (numarray, left, right, x)
{
    if (left > right) { return -1; }
    int mid = 1 + (right - 1)/2;
    if (numarray[mid] == x) { return mid; }
    if (numarray[mid] > x)
        { return binarySearch (numarray, left,
          mid -1, x); }
    return binarySearch (numarray, mid + 1, right, x);
}
```

Исходя из предположения, что массив отсортирован, мы проверяем, является ли средний элемент массива тем, что мы ищем. Если это так, то возвращаем индекс среднего элемента. Если же нет и средний элемент больше требуемого значения, то мы проделываем то же самое с первой половиной массива, а если больше — то со второй половиной. Для каждого рекурсивного вызова выполняется постоянный объем работы (проверка, что значение `left` не больше, чем `right`; это подразумевает, что мы выполнили поиск по всему массиву и искомое значение не было найдено; затем вычисление средней точки и сравнение ее значения с тем, что мы ищем). Мы выполняем $O(\lg n)$ вызовов, каждый из которых занимает $O(1)$ времени, поэтому общая сложность двоичного поиска составляет $O(\lg n)$.

---

[1] Как показывает практика, лучше перенести проверку left > right в конец, поскольку это менее распространенный случай; мы поступили так, чтобы обойтись без вложенности.

Углубленные темы

Представьте, что у нас есть рекурсивная функция, которая разбивает задачу на две части, размер каждой из них составляет 2/3 от размера оригинала? Формула для вычисления такой рекурсии действительно существует; подробнее о ней и об основной теореме (Master Theorem) вы узнаете в главе 31 второго тома книги.

1.8. Насколько сложна задача?

Если есть алгоритм для решения задачи, определить время выполнения этого алгоритма обычно довольно просто. Но что, если у нас еще нет алгоритма, но уже нужно понять, насколько сложно будет решить задачу?

Мы можем это сделать, сравнивая задачу с другими подобными задачами, для которых известно время выполнения. Мы можем разделить задачи на классы — наборы задач, имеющих сходные характеристики. Нас интересуют два основных класса: задачи, которые решаются за полиномиальное время, и задачи, решение которых можно проверить за полиномиальное время. Оба этих класса мы рассмотрим в следующей главе.

2 Структуры данных

2.1. Организация данных

Одно из главных понятий Computer Science — структуры данных[1]. Говоря о времени выполнения алгоритмов, мы предполагаем, что данные хранятся в соответствующей структуре, которая позволяет эффективно их обрабатывать. Какая структура лучше, зависит от типа данных и от того, какой доступ к ним нужен.

- Необходим ли произвольный доступ, или достаточно последовательного?
- Будут ли данные при записи всегда добавляться в конец списка, или нужна возможность вставлять значения в середину?
- Допускаются ли повторяющиеся значения?
- Что нам важнее: наименьшее возможное время доступа или строгая верхняя граница времени выполнения каждой операции?

---

[1] Многое из изложенного, вероятно, знакомо опытным программистам. Но, поскольку эта информация очень важна для дальнейшего чтения, она включена в книгу. Если вы уже знакомы с этим материалом, смело пропустите главу.

Ответы на все эти вопросы определяют то, как должны храниться данные.

2.2. Массивы, очереди и другие способы построиться

Возможно, самая известная структура данных — это массив, набор элементов, проиндексированных ключом. Элементы массива хранятся последовательно, причем ключ имеет форму смещения относительно начальной позиции в памяти, благодаря чему можно вычислить положение любого элемента по его ключу. Именно поэтому индексы массива[1] обычно начинаются с нуля; первый элемент массива находится на нулевом расстоянии от начала, следующий — на расстоянии одного элемента от начала и т. д. «На расстоянии одного элемента» может означать один байт, одно слово и т. д., в зависимости от размера данных. Важно, что каждый элемент массива занимает одинаковое количество памяти.

Польза массивов состоит в том, что получение или сохранение любого элемента массива занимает постоянное время, а весь массив занимает $O(n)$ места в памяти[2]. Если количество элементов заранее известно, то память не расходуется зря; поскольку позиция каждого эле-

---

[1] Интересно, что в английском языке есть два варианта множественного числа от слова index (индекс): array indices для индексов массива, но database indexes — для индексов базы данных.

[2] Помните: если не указано иное, буквой n обозначается количество элементов.

мента вычисляется просто по смещению относительно начала, нам не нужно выделять место для указателей. Поскольку элементы массива расположены в смежных областях памяти, перебор значений массива, очевидно, выполняется гораздо быстрее, чем для многих других структур данных из-за меньшего количества неудачных обращений к кэш-памяти[1].

Однако требование выделения непрерывного блока памяти может сделать массивы плохим выбором, когда число элементов заранее не известно. С ростом размера массива может понадобиться скопировать его в другое место памяти (при условии, что оно есть). Избежать этой проблемы, предварительно выделив гораздо больше места, чем необходимо, бывает довольно затратно[2]. Другая проблема — вставка и удаление элементов массива занимает много времени ($O(n)$), поскольку приходится перемещать все элементы массива.

На практике массивы используются как сами по себе, так и для реализации многих других структур данных, которые накладывают дополнительные ограничения на манипулирование данными. Например, строка может

---

[1] Это так называемая локальность ссылок: как только мы получили доступ к элементу, вполне вероятно, что в ближайшем будущем также понадобится доступ к другим элементам, находящимся поблизости. Поскольку мы уже загрузили страницу, содержащую первый элемент, в кэш, то получение ближайших элементов (которые, вероятно, находятся на той же странице) происходит быстрее.

[2] Это особенно актуально при работе со встроенными системами, которые, как правило, имеют ограниченный объем памяти.

быть реализована в виде массива символов. Очередь — это последовательный список, в котором элементы добавляются только в один конец списка (постановка в очередь), а удаляются из другого (извлечение из очереди); таким образом, очередь может быть реализована как массив, в котором «начало» перемещается вместе с началом очереди при условии, что максимальное количество элементов в очереди никогда не превышает размер массива. Однако очередь неопределенной длины лучше реализовать на основе двусвязного списка (см. раздел 2.3). К другим структурам, реализуемым на основе массивов, относятся списки, стеки и кучи (см. раздел 2.4), очереди с приоритетом (которые часто создаются на основе куч) и хеш-таблицы (см. раздел 2.5).

2.3. Связные списки

Связный список — это структура данных, в которой каждый элемент содержит данные и указатель на следующий элемент списка (а если это двусвязный список, то также ссылку на предыдущий элемент). Указатель на связный список — это просто указатель на первый элемент, или head, списка; поскольку элементы могут размещаться в разных местах выделенной памяти, для поиска указанного элемента необходимо начать с первого элемента и пройтись по всему списку.

Как уже говорилось, многие структуры данных реализованы на основе массивов или связных списков. Во многом связный список является дополнением массива. Если сильная сторона массива — быстрый доступ к любому элементу (по его ключу), то для того, чтобы найти

элемент списка, необходимо пройти по всем ссылкам, пока не будет найден нужный элемент, что в худшем случае займет $O(n)$ времени. С другой стороны, массив имеет фиксированный размер, а элементы связного списка могут размещаться в любом месте памяти, и список может произвольно увеличиваться до тех пор, пока не будет исчерпана доступная память. Кроме того, вставка и удаление элементов массива очень затратны, а в связном списке эти операции выполняются за постоянное время, если есть указатель на предыдущий узел (рис. 2.1).

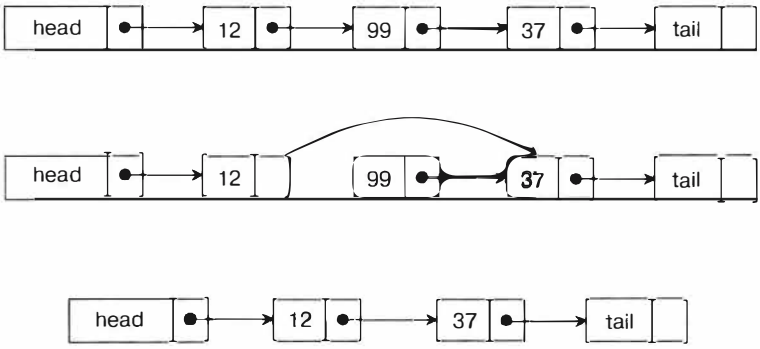


Рис. 2.1. Удаление узла из связного списка

Практическое применение

Представьте себе поезд как пример двусвязного списка: каждый вагон связан с предыдущим и (если он существует) со следующим. В конец поезда можно легко добавить вагоны, но можно вставить вагон и в середину поезда, отсоединив и (пере)присоединив существующие вагоны перед и после добавляемых, или же можно отсоединить

вагоны в середине поезда, откатить их на боковой путь и присоединить оставшиеся вагоны. А вот извлечь вагон напрямую не получится; для этого нужно сначала пройти по всему поезду и отделить нужный вагон от предыдущего.

Теоретическая глупость

Как-то мой преподаватель спросил у группы студентов, как определить, содержит ли связный список цикл. Есть несколько классических решений этой задачи. Один из способов — обратить указатели на элементы списка, когда мы их проходим; если цикл существует, то мы в итоге вернемся к началу списка. Другой способ: обходить список с двумя указателями, так что один указатель ссылается на следующий элемент, а второй — через один элемент. Если в списке существует цикл, то в итоге оба указателя когда-нибудь будут ссылаться на один и тот же элемент.

Однако преподаватель добавил условие: он сказал, что ему все равно, сколько времени займет выполнение нашего метода. Я предложил несколько глупый вариант: вычислить объем памяти, необходимый для хранения одного узла списка, и разделить на него общую память системы. Потом вычислить количество посещенных элементов. Если число посещенных элементов превышает количество узлов, которые могут быть сохранены в памяти, список должен содержать цикл.

Преимущество этого решения неоднозначно: оно требует безумно много времени (определяемого размером памяти, а не размером списка) и является абсолютно правильным.

2.4. Стеки и кучи

2.4.1. Стеки

Стек — это структура данных типа LIFO (Last In, First Out — «последним пришел, первым ушел»), в которой элементы добавляются или удаляются только сверху; это называется поместить элемент в стек (push) или извлечь его из стека (pop) (рис. 2.2).

Рис. 2.2. Элементы всегда помещаются в вершину стека

Стек можно реализовать на основе массива (отслеживая текущую длину стека) или на основе односвязного списка (отслеживая head списка[1]). Как и в случае с очередями, реализация на основе массива проще, но

[1] Вместо того чтобы добавлять новые элементы в конец списка, их можно просто помещать в начало. Тогда извлекаемый элемент всегда будет первым в списке.

она накладывает ограничение на размер стека. Стек, реализованный на основе связного списка, может расти до тех пор, пока хватает памяти.

Стеки поддерживают четыре основные операции, каждая из которых может быть выполнена за время $O(1)$: `push` (поместить элемент в стек), `pop` (извлечь элемент из стека), `isEmpty` (проверить, пуст ли стек) и `size` (получить количество элементов в стеке). Часто также реализуют операцию `peek` (просмотреть верхний элемент стека, но не удалять его), что эквивалентно извлечению верхнего элемента и его повторному помещению в стек.

При помещении элемента в стек может возникнуть исключение, если размер стека ограничен, и в настоящее время стек заполнен (ошибка переполнения), а при извлечении элемента из стека возникает исключение, если в данный момент стек пуст (ошибка обнуления).

Несмотря на то что в стеках не допускается произвольный доступ, они очень полезны в тех компьютерных вычислениях, для которых требуется ведение истории, от операций Undo до рекурсивных вызовов функций. В этом случае стек обеспечивает возможность обратного перебора, если необходимо вернуться к предыдущему состоянию. В разделе 13.3 вы увидите, как стеки используются для реализации магазинных автоматов, способных распознавать контекстно свободные языки.

Типичным примером использования стека является проверка сбалансированности фигурных скобок. Рассмотрим язык, в котором скобки должны быть парными: каждой правой скобке (}) предшествует соответствующая левая скобка ({). Мы можем прочитать строку

и каждый раз, когда встречается левая скобка, помещать ее в стек. Всегда, когда встречается правая скобка, мы будем извлекать левую скобку из стека. Если попытаться извлечь фигурную скобку из стека, но стек окажется пустым, это будет означать, что у правой фигурной скобки нет соответствующей левой скобки. Если в конце строки стек окажется непустым, это будет означать, что считано больше левых скобок, чем правых. В противном случае все скобки в строке окажутся парными.

2.4.2. Кучи

Кучи, подобно стекам, как правило, реализуются на основе массивов. Как и в стеке, в куче можно удалить за один раз только один элемент. Однако это будет не последний добавленный, а наибольший (для max-кучи) или наименьший элемент (для min-кучи). Куча частично упорядочена по ключам элементов, так что элемент с наивысшим (или низшим) приоритетом всегда хранится в корне кучи (рис. 2.3).

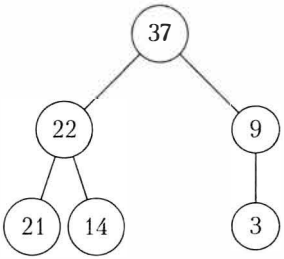


Рис. 2.3. Узлы-братья (узлы с одним и тем же родителем) в куче никак не взаимосвязаны; просто каждый узел имеет более низкий приоритет, чем его родитель

Словом *«куча»* называют структуру данных, которая удовлетворяет свойству упорядоченности кучи: неубывания (min-куча) (значение каждого узла не меньше, чем у его родителя) либо невозрастания (max-куча) (значение каждого узла не больше, чем у родителя). Если не указано иное, то, говоря о куче, мы имеем в виду двоичную кучу, которая является полным двоичным деревом[1], которое удовлетворяет свойству упорядоченности кучи; в число других полезных видов кучи входят левосторонние кучи, биномиальные кучи и кучи Фибоначчи.

Max-куча поддерживает операции поиска максимального значения (find-max) (peek), вставки (insert) (push), извлечения максимального значения (extract-max) (pop) и увеличения ключа (increase-key) (изменения ключа узла с последующим перемещением узла на новое место на графе). Для двоичной кучи после создания кучи из списка элементов за время $O(n)$ каждая из этих операций занимает время $O(\lg n)$[2].

Кучи используются в тех случаях, когда нужен быстрый доступ к наибольшему (или наименьшему) элементу списка без необходимости сортировки остальной части списка. Именно поэтому кучи используются для реализации очередей с приоритетом: нас интересует

---

[1] Двоичное дерево, в котором каждый уровень, за исключением, возможно, нижнего, имеет максимальное количество узлов. Подробнее о деревьях читайте в главе 5.

[2] Подробнее о временной сложности каждой операции читайте в книге Introduction to Algorithms.

не относительный порядок всех элементов, а только то, какой элемент является следующим в очереди, — это всегда текущий корень кучи. Более подробно кучи рассмотрены в разделах 5.3 и 6.5.

2.5. Хеш-таблицы

Предположим, мы хотим определить, содержится ли в массиве некий элемент. Если массив отсортирован, можно выполнить двоичный поиск и найти этот элемент за время $O(\lg n)$; если же нет, можно перебрать весь массив за время $O(n)$. Конечно, если бы мы знали, где находится этот элемент, то мы бы просто обратились туда напрямую за время $O(1)$.

В некоторых случаях, таких как сортировка подсчетом (см. подраздел 8.4.1), в качестве индекса массива используется сам сохраняемый элемент или его ключ, и поэтому можно просто перейти в нужное место без поиска. А что, если бы для произвольного объекта у нас была функция, которая бы принимала ключ этого объекта и преобразовывала его в индекс массива, так что мы бы точно знали, где находится объект? Именно как работают хеш-таблицы.

Первая часть хеш-таблицы — это хеш-функция; она преобразует ключ элемента, который помещается в хеш. Этому ключу соответствует определенное место в таблице. Например, наши ключи представляют собой набор строк, а хеш-функция ставит в соответствие каждой строке количество символов этой строки[1]. Тогда слово

[1] Это не очень хорошая хеш-функция.

cat попадет в ячейку 3, а *penguin* — в ячейку 7. Если место ограничено, то мы делим хеш-код по модулю на размер массива; тогда, если массив ограничен десятью ячейками (0–9), строка *sesquipedalophobia* (хеш-код которой равен 18) попадет в ячейку 7[1, 2].

Что произойдет, если мы уже поместили в хеш-таблицу слово *cat*, но попытаемся вставить туда слово *cat* еще раз? В хеш-таблицах допускается хранить только один экземпляр каждого элемента; в некоторых реализациях в каждой ячейке хранится счетчик, который увеличивается по мере необходимости, чтобы отслеживать количество копий элемента. Но что, если мы попытаемся вставить слово *dog*, которое попадет в ту же ячейку? Есть два способа решить эту проблему. Можно рассматривать каждую ячейку как группу объектов, представленных в виде связного списка (который нужно пройти, чтобы найти правильное животное); это называется методом цепочек. Или же можно просмотреть близлежащие ячейки, найти свободную и поместить *dog* туда — это называется открытой адресацией. Размер хеш-таблицы с цепочками не ограничен, однако производительность будет снижаться по мере увеличения количества элементов в ячейке. При открытой адресации таблица имеет фиксированный

---

1 Если бы мы хешировали слова sesquipedalophobia и hippopotomonstrosesquippedaliophobia по модулю 9, то оба слова попали бы в точку ноль и образовалась бы коллизия, к удовольствию любителей длинных слов и к ужасу страдающих логофобией.

2 На практике мы будем использовать массив, размер которого является простым числом.

максимальный размер; как только все ячейки будут заполнены, в таблицу нельзя будет вставить новые элементы.

Способ организации хеш-таблицы зависит от того, предпочитаем ли мы свести к минимуму коллизии (несколько значений, которые попали в одну ячейку) или объем памяти. Чем больше места выделено под таблицу относительно количества вставляемых элементов, тем меньше вероятность коллизий. За счет увеличения памяти мы получаем повышение скорости: после того как ключ захеширован, сохранение или извлечение элемента (при условии отсутствия коллизий) занимает $O(1)$ времени. Однако при коллизиях наихудшее время извлечения в хеш-таблице (когда для каждого элемента возникает коллизия) составляет $O(n)$.

Как правило, хеш-таблицы используют в тех случаях, когда нужен прямой доступ к неотсортированным данным на основе ключа и при этом существует быстродействующая функция генерации ключа для каждого объекта (при условии, что сами объекты не являются такими ключами). Мы не будем использовать хеш-таблицы, когда нужно сохранять порядок сортировки, или элементы не распределены (то есть много элементов хешируется в малое количество ячеек), или часто необходим доступ к блокам последовательно размещенных элементов, поскольку элементы (как мы надеемся) равномерно распределены по памяти, выделенной для хеш-таблицы, мы теряем преимущество от локальности ссылки.

Практическое применение

В C# есть класс `Hashtable`, позволяющий хранить произвольные объекты. Каждый объект, добавляемый в `Hashtable`, должен реализовывать функцию `GetHashCode()`, она возвращает значение `int32`, которое можно использовать для хеширования объекта. `Dictionary<TKey, TValue>` предоставляет тот же функционал, но только для объектов типа `TValue`, который (при условии, что `TValue` не установлен в `Object`) позволяет программисту избегать упаковки и распаковки сохраненных элементов. Для внутреннего представления в обоих случаях используется структура данных хеш-таблицы, но с разными методами предотвращения коллизий. В `Hashtable` применяется перехеширование (поиск другой ячейки, если первая занята), а в `Dictionary` — метод цепочек (несколько элементов с одинаковым хешем просто добавляются в одну ячейку).

2.6. Множества и частично упорядоченные множества

Множество — это неупорядоченная коллекция уникальных предметов. Существует три основные операции над множествами, каждая из которых принимает два множества в качестве аргументов и возвращает еще одно множество в качестве результата.

Объединение множеств Union(S, T) — это множество, содержащее каждый элемент, который принадлежит хотя бы одному из множеств S и T. Обычно объединение обозначается как $S \cup T$ (рис. 2.4).

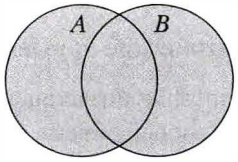


Рис. 2.4. Объединение множеств A и B

Пересечение множеств Intersection(S, T) — это множество элементов, содержащихся как в S, так и в T, обозначается как $S \cap T$ (рис. 2.5).

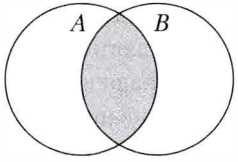


Рис. 2.5. Пересечение множеств A и B

Разность множеств Difference(S, T) (рис. 2.6) обозначается как $S - T$ — это множество, в которое входят все элементы, содержащиеся в S, но отсутствующие в T.

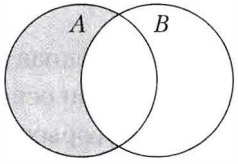


Рис. 2.6. Разность множеств A и B ($A - B$)

Наконец, есть еще одна дополнительная операция, которая возвращает логическое значение: подмножество Subset(S, T). Ее результатом будет истина (true), если S является подмножеством T (то есть каждый элемент

множества S — элемент множества T). Если S является подмножеством T и не равно T, то такое подмножество называется собственным. Мы используем обозначения $S \subseteq T$ (S является подмножеством T), $S \nsubseteq T$ (S не является подмножеством T), $S \subset T$ (S является собственным подмножеством T) и $S \not\subset T$ (S не является собственным подмножеством T). Если $S \subseteq T$ и $T \subseteq S$, то $S = T$. Существуют также дополнительные операции, которые применяются к множествам строк; они рассмотрены в подразделе 13.2.2.

Существует несколько видов множеств (рис. 2.7). Мультимножество — это множество, в котором допускается дублирование элементов, то есть хранятся несколько копий одного и того же значения либо просто ведется подсчет того, сколько раз данное значение присутствует в множестве.

$$\{4, 7, 2, 16, 1004\}$$

Рис. 2.7. Множество целых чисел

Частично упорядоченное множество — множество, в котором некоторые элементы расположены в определенном порядке. Это означает, что для некоторой двоичной операции[1] $\leq$ между элементами b и c может быть истиной $b \leq c$, в других — $c \leq b$ или же между b и c может не существовать отношений.

Если все элементы множества связаны операцией $\leq$, то есть для любых двух элементов f и g множества

[1] Двоичная операция — это такая операция, которая работает с двумя операндами. Например, операция «плюс» используется для получения суммы двух значений.

либо $f \leq g$, либо $g \leq f$, то операция $\leq$ определяет полный порядок множества. Например, стандартная операция «меньше или равно» — это порядок для множества действительных чисел: любые два числа либо равны, либо одно больше другого.

Необходимо, чтобы отношение было рефлексивным (каждый элемент множества меньше или равен самому себе), антисимметричным (если b меньше или равно c, то c не может быть меньше или равно b, если только b не равно c) и транзитивным (если b меньше или равно c, а c меньше или равно d, то b меньше или равно d).

Обратите внимание, что куча — частично упорядоченное мультимножество (рис. 2.8): в ней может существовать несколько копий одного и того же значения и каждое значение имеет определенную связь только со своим родителем и потомками.

$$\{3, 3, 6, 3, 9\}$$

Рис. 2.8. Мультимножество целых чисел

Практический пример

Пусть $\leq$ означает отношение «является потомком», причем по определению каждый человек является потомком самого себя. Тогда такое отношение является антисимметричным (если я твой потомок, то ты не можешь быть моим потомком) и транзитивным (если я твой потомок, а ты потомок дедушки, то я тоже потомок дедушки). Это частичное, а не полное упорядочение, поскольку, возможно, ни я не являюсь твоим потомком, ни ты — моим.

Практическое применение

Реляционные базы данных[1] содержат таблицы, в которых хранятся наборы строк. Упорядоченность может быть задана при выводе данных (в SQL это делается с помощью оператора ORDER BY), но такое ограничение не накладывается на фактически сохраненные данные, поэтому нельзя считать, что строки в базе данных хранятся в каком-либо определенном порядке.

2.7. Специализированные структуры данных

Далее в книге мы рассмотрим более специализированные структуры данных. В главе 5 описаны некоторые распространенные графовые структуры данных (о них — в главе 4). В главе 32 второго тома представлена концепция амортизированного времени выполнения (общего времени, затрачиваемого на серию операций, а не времени выполнения отдельной операции), а в главе 33 описана структура данных, которую мы проанализируем с использованием амортизированного времени выполнения.

[1] Реляционная база данных структурирована в соответствии с отношениями между хранимыми элементами. Реляционные и иерархические базы данных рассмотрены в главе 30 второго тома.

3 Классы задач

Специалисты в Computer Science классифицируют задачи по времени, которое занимает их решение, в зависимости от количества входных данных. Благодаря такой классификации задач мы определяем сложность их решения. На практике это позволяет не тратить времени на задачи, которые не решаются достаточно быстро, чтобы ответ был полезным.

Самые простые задачи, класса *P*, могут быть решены за полиномиальное время. Это все задачи, у которых время решения — количество входных данных, возведенное в некоторую постоянную степень. Принято считать, что такие задачи имеют эффективные решения. К этому классу относятся многие широко известные задачи, например такие, как сортировка списков.

Задачи класса *P* также называют легкоразрешимыми. Как правило, если мы можем доказать, что задача относится к классу *P*, это значит, что ее можно решить за разумное время.

Класс *P* является собственным подмножеством множества задач *EXP*, которые решаются за экспоненциальное

время; любая задача, которая может быть решена за время $O(n^2)$, также может быть решена за время $O(2^n)$.

Математическое предупреждение: возвращаясь к главе 2

Для двух множеств A и B множество A является подмножеством B, а B — надмножеством A, если каждый элемент множества A также является элементом множества B. Это обозначается так: $A \subseteq B$ и $B \supseteq A$.

Если множество B содержит все элементы множества A, а также что-то еще, то B является собственным надмножеством множества A, а A — собственным подмножеством B. Это обозначается так: $A \subset B$ и $B \supset A$.

Например, множество {1, 2, 3} является собственным подмножеством множества {1, 2, 3, 4, 5}.

Но в множество *EXP* входят и другие классы, кроме *P*. Один из них — это недетерминированный полином (Nondeterministic Polynomial, NP). Задача относится к классу *NP*, если она может быть решена недетерминированно[1] за полиномиальное время. Другими словами, существует алгоритм, который решает эту задачу путем

[1] Детерминированный алгоритм с фиксированными входными данными каждый раз проходит через одну и ту же последовательность состояний и возвращает один и тот же результат. С математической точки зрения такой алгоритм отображает область экземпляров задачи на ряд решений. Недетерминированный алгоритм может демонстрировать различное поведение для одного и того же набора входных данных.

принятия ряда решений, причем в каждой точке принятия решения алгоритм случайным образом (и удачно) делает правильный выбор. Пока остается ряд шагов, ведущих к ответу, алгоритм выбирает правильные шаги. Другой (более полезный) способ показать, что задача относится к классу *NP*, — убедиться, что ее решение можно проверить (детерминированно, то есть следуя предварительно определенной последовательности шагов) за полиномиальное время. *NP* является надмножеством *P*; любое решение, которое может быть найдено за полиномиальное время, также может быть проверено за полиномиальное время.

Один из важнейших вопросов Computer Science, на который до сих пор не получен ответ: является ли *NP* собственным надмножеством *P*? Существуют ли задачи, которые относятся к *NP*, но не принадлежат *P*, или это одно и то же множество задач? Другими словами: любая ли задача, решение которой быстро проверяется компьютером, также быстро решается компьютером? Большинство специалистов в области Computer Science считают, что $P \neq NP$, но никаких математических доказательств найдено не было[1] (рис. 3.1).

Рассмотрим задачу разбиения множества чисел (рис. 3.2). Для заданного мультимножества (множества, в котором могут присутствовать повторяющиеся элементы) натуральных чисел нужно определить, можно ли разбить

---

[1] Вопрос о том, справедливо ли утверждение $P = NP$, является одной из семи Задач Тысячелетия — важных математических вопросов, ответы на которые не найдены. За решение каждого из них установлен приз 1 миллион долларов.

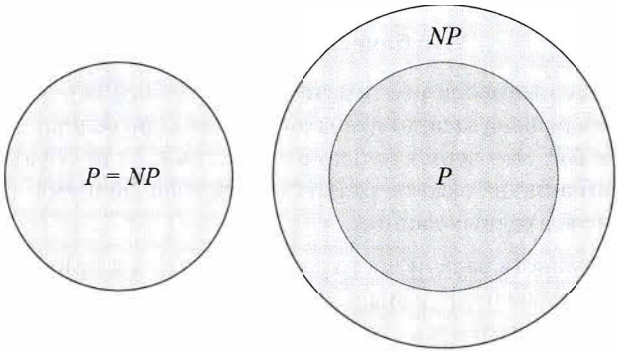


Рис. 3.1. Проблема $P = NP$: это одно и то же множество *(слева)* или же P является собственным подмножеством NP *(справа)*?

такое множество на два подмножества таким образом, чтобы сумма чисел в первом множестве равнялась сумме чисел во втором. Если мультимножество разделено на два подмножества $\{S_1, S_2\}$, то задача решается тривиально: нужно лишь сложить значения в каждом множестве и определить, идентичны ли эти две суммы. Но выяснить, какие значения следует поместить в каждое из множеств, за полиномиальное время невозможно (при условии, что $P \neq NP$).

$$S = \{1, 1, 2, 3, 4, 5, 6\}$$
$$S_1 = \{1, 1, 2, 3, 4\}$$
$$S_2 = \{5, 6\}$$

Рис. 3.2. Простой пример задачи разбиения множества чисел. Мультимножество {1, 1, 2, 3, 4, 5, 6} можно разделить на мультимножества {1, 1, 2, 3, 4} и {5, 6}. Сумма элементов каждого из них равна 11

Углубляясь в подробности

Выше мы говорили о том, что задача разбиения не может быть решена за полиномиальное время при условии, что $P \neq NP$. Это верно только в том случае, если обратить внимание на важное различие между значением и размером входных данных.

Сложность задачи часто зависит от того, как входные данные закодированы. С технической точки зрения задача относится к классу P, если ее можно решить с помощью алгоритма, который выполняется за полиномиальное время, в зависимости от длины входных данных, то есть от числа битов, необходимых для представления входных данных. Алгоритм выполняется за псевдополиномиальное время, если он является полиномиальным, в зависимости от числового значения входных данных, длина которых изменяется экспоненциально. Рассмотрим задачу с количеством входных данных 1 миллиард и алгоритмом, который требует n операций. Если n — это количество цифр, необходимое для представления числа (в данном случае 10, при условии, что основанием системы счисления является 10), то алгоритм является полиномиальным. Если n — значение входных данных (1 000 000 000), то алгоритм будет псевдополиномиальным. В последнем случае количество требуемых операций растет намного быстрее, чем количество битов входных данных.

Некоторые задачи класса *NP* называются *NP*-сложными; *NP*-сложная задача (несколько рекурсивно) определяется как любая задача, которая по крайней мере столь же сложна, как и самая сложная задача класса *NP*. Чтобы понять, что это значит, рассмотрим принцип сокращения.

Задача *B* может быть сокращена до задачи *C*, если решение задачи *C* позволило бы решить задачу *B* за полиномиальное время. Другими словами, если у нас есть оракул, который дает ответ на задачу *C*, то мы можем (за полиномиальное время) преобразовать его в ответ на задачу *B*. Решение задачи является *NP*-сложным, если любая задача класса *NP* может быть сведена к этому решению, то есть эффективное решение задачи также приведет к эффективному решению любой задачи класса *NP*.

Обратите внимание, что сама *NP*-сложная задача не обязательно должна принадлежать к классу *NP*; она должна быть как минимум такой же сложной, как все задачи класса *NP*. Это означает, что некоторые *NP*-сложные задачи (не принадлежащие классу *NP*) могут быть намного сложнее, чем другие задачи, которые принадлежат к классу *NP*.

Задача, которая и является *NP*-сложной, и относится к классу *NP*, называется *NP*-полной. Поскольку каждая *NP*-полная задача может быть сокращена до любой другой *NP*-полной задачи, сокращение *NP*-полной задачи к новой задаче и демонстрация того, что новая задача принадлежит к классу *NP*, достаточны, чтобы доказать, что данная задача является *NP*-полной.

Углубляясь в подробности

Если *NP*-полная задача имеет псевдополиномиальное решение, она называется слабо *NP*-полной. Если же это не так (или $P = NP$), то задача называется сильно *NP*-полной.

Для того чтобы ответить на вопрос, действительно ли $P = NP$, нужно либо предоставить алгоритм, который бы решал NP-полную задачу за (детерминированное) полиномиальное время (в этом случае $P = NP$), либо доказать, что такого алгоритма не существует (в таком случае $P \neq NP$).

Углубленные темы: NP-полное сокращение

Предположим, что у нас есть следующие задачи.

Сумма подмножества. Для заданного мультимножества S целых чисел и значения w определить, существует ли непустое подмножество множества S, сумма элементов которого равна w?

Разделение. Можно ли разделить мультимножество S целых чисел на два подмножества — S_1 и S_2, суммы элементов которых равны?

Поскольку сумма подмножества — NP-полная задача, можно доказать, что разбиение также является NP-полной задачей, следующим образом.

Первый шаг — показать, что задача разбиения относится к классу NP. Для данных подмножеств S_1 и S_2 можно найти сумму элементов каждого множества и сравнить их за время, пропорциональное общему количеству значений. Поскольку решение может быть проверено за линейное (полиномиальное) время, задача принадлежит к классу NP.

Второй шаг — показать, что разбиение является NP-сложным. Для этого мы покажем, что решение новой задачи за полиномиальное время также даст решение за полиномиальное время задачи, которая, как известно,

является *NP*-сложной, поэтому разбиение — столь же сложная задача, как и эта задача.

Допустим, что у нас есть алгоритм, который решает за полиномиальное время и задачу разбиения, и задачу суммы подмножества, которую мы хотели бы решить. Есть множество S, и мы хотим знать, существует ли у него подмножество, сумма элементов которого равна w. Это эквивалентно вопросу, можно ли разбить множество S с суммой $|S|$ на мультимножество S_1 с суммой w_1 и мультимножество S_2 с суммой $w_2 = |S| - w$.

Пусть x — разность между w_1 и w_2. Добавим это число в множество S, а затем выполним алгоритм разбиения для нового множества. Если задача разрешима, то алгоритм вернет два разбиения с одинаковой суммой, которая равна половине от $|S| + x$. Вследствие того, как мы выбрали значение x, удаление этого элемента теперь дает множества S_1 и S_2, одно из которых является решением исходной задачи суммы подмножества. Поскольку дополнительная работа заняла линейное время, решение задачи суммы подмножества заняло полиномиальное время.

Пример:

$S = \{5, 10, 10, 30, 45\}$, $w = 25$

$|S| = 100$, $w_2 = 100 - 25 = 75$, $x = 75 - 25 = 50$

$S_a = \{5, 10, 10, 30, 45, 50\}$

$S_1 = \{5, 10, 10, 50\}$

$S_2 = \{30, 45\}$

Решение: {5, 10, 10}

Этот пример показывает, что задача разбиения является *NP*-полной, а ее решение за полиномиальное время было бы доказательством того, что $P = NP$.

Существует еще несколько классов сложности, но те два, которые мы обсудили в этом разделе, наиболее известны. Другие классы, как правило, определяются в терминах машин Тьюринга, которые мы рассмотрим позже в этой книге.

Дополнительные источники информации

Некоторые наиболее известные *NP*-полные задачи рассмотрены в приложении Б. Более подробную информацию о классах задач вы найдете в главах 13 и 14.

Часть II
Графы и графовые алгоритмы

4 Введение в теорию графов

4.1. Семь кенигсбергских мостов

Я считаю, что введение в теорию графов обязательно должно начинаться с задачи о кенигсбергских мостах. Эта задача важна не сама по себе, а потому, что с нее начался новый раздел математики — теория графов.

В городе Кенигсберге (ныне Калининград) протекает река. Она делит город на четыре части, которые соединяются семью мостами (рис. 4.1). Возникает вопрос: можно ли прогуляться по городу, пересекая каждый мост ровно один раз?

Однажды этот вопрос задали математику Леонарду Эйлеру. Он объявил задачу тривиальной, но она все же привлекла его внимание, поскольку ни одна из существующих областей математики не была достаточной для ее решения. Главным заключением является то, что топологические деформации неважны для решения; другими словами, изменение размера и формы различных деталей не меняет задачу при условии, что не меняются соединения[1].

[1] Эйлер назвал это геометрией положения.

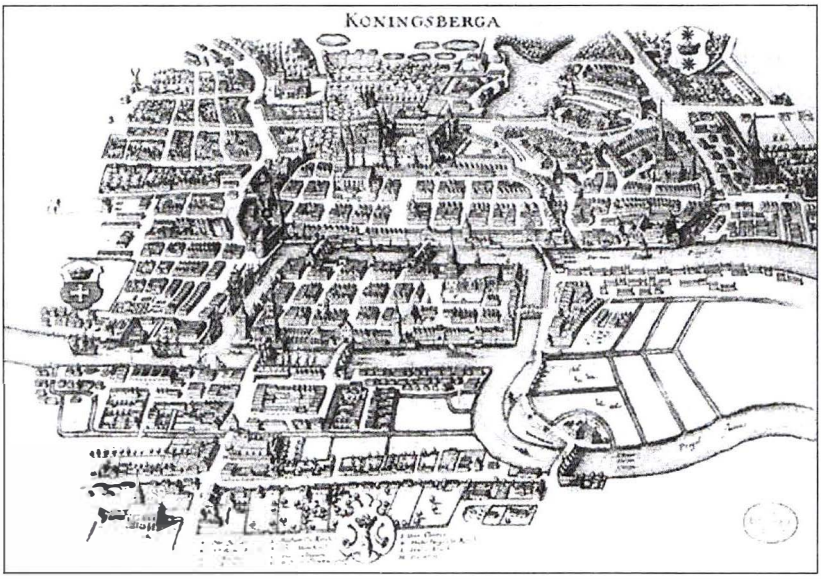


Рис. 4.1. Находящаяся в открытом доступе карта Кенигсберга. Merian-Erben, 1652

Таким образом, мы можем упростить карту, изображенную на рис. 4.1, заменив каждую часть города вершиной, а каждый мост — ребром, соединяющим две вершины. В результате получим граф, показанный на рис. 4.2.

Ключевым логическим заключением[1] является тот факт, что для того, чтобы сначала попасть на сушу, а затем покинуть ее, требуется два разных моста. Поэтому

[1] Solutio Problematis ad Geometriam Situs Pertinentis // Commentarii academiae scientiarum Petropolitanae 8, 1741. P. 128–140. («Решение одной проблемы, относящейся к геометрии положения»). Английский перевод доступен в книге: *Biggs N., Lloyd E. K., Wilson R. J.* Graph Theory, 1736–1936.

любой участок суши, который не является начальной или конечной точкой маршрута, должен быть конечной точкой для четного числа мостов. В случае Кенигсберга к каждой из четырех частей города вело нечетное число мостов, что делало задачу неразрешимой. Путь через граф, который проходит через каждое ребро ровно один раз, теперь называется эйлеровым путем.

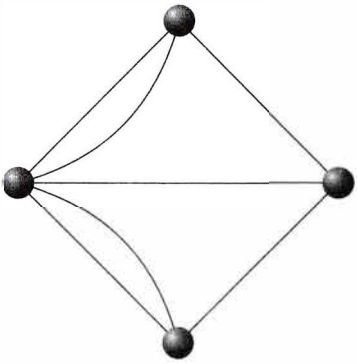


Рис. 4.2. Кенигсбергские мосты в виде графа. Обратите внимание, что каждая вершина имеет нечетную степень, то есть ее касается нечетное количество ребер

4.2. Мотивация

Графы в Computer Science чрезвычайно важны: очень многие задачи можно представить в виде графов. В случае с кенигсбергскими мостами представление города в виде графа позволяет игнорировать неважные детали — фактическую географию города — и сосредоточиться на важном — на связях между различными частями города. Во многих случаях возможность свести

постановку задачи к эквивалентной задаче для определенного класса графов даст полезную информацию о том, насколько сложно ее решить. Некоторые задачи являются *NP*-сложными на произвольных графах, но имеют эффективные (часто $O(n)$) решения на графах, имеющих определенные свойства.

В этой и последующих главах вы познакомитесь с терминами, связанными с графами и некоторыми распространенными структурами данных, которые используют графы. Вы узнаете, как представлять графы визуально, а также в форматах, более удобных для вычислений. После этого вы узнаете об известных графовых алгоритмах и некоторых основных графовых классах. К тому времени, когда вы закончите читать часть II, вы будете понимать, в каких случаях можно применять методы построения графов при решении задач.

Определение

Класс графов — это (как правило, бесконечный) набор графов, который обладает определенным свойством; принадлежность данного графа к этому классу зависит от того, есть ли у этого графа это свойство.

Пример

Двудольные графы — это такие графы, в которых вершины можно разделить на два множества, так что каждое ребро соединяет вершину из одного множества с вершиной из другого множества.

4.3. Терминология

Граф — это способ представления взаимосвязей в множестве данных. Графы часто изображаются в виде кружков, которыми обозначаются вершины, и линий между ними, представляющих ребра. Но вы узнаете и о других способах представления графов. Две вершины являются смежными, если между ними есть ребро, и несмежными, если между ними нет ребра.

Вершины графа также называются узлами; эти два термина, как правило, взаимозаменяемы. Однако точка многоугольника, в которой встречаются два ребра и более, всегда будет вершиной, а участок памяти, в котором содержится вершина и ее набор ребер, — узлом.

4.3.1. Части графов

Я буду часто ссылаться на подграф или порожденный подграф графа. *Подграфом* графа является любое количество вершин графа вместе с любым количеством ребер (которые также принадлежат исходному графу) между этими вершинами. *Порожденный подграф* — это любое подмножество вершин вместе со всеми ребрами графа, соединяющими эти вершины.

Строгое подмножество множества содержит меньше элементов, чем исходное множество; другими словами, собственное подмножество вершин графа содержит меньше вершин, чем исходный граф, в то время как регулярное подмножество может содержать все множество вершин.

4.3.2. Графы со всеми ребрами или без ребер

Полный (под)граф, или клика, — это такой граф, который содержит все возможные ребра между его вершинами. *Независимое* (или внутренне *устойчивое*) множество — это множество вершин без ребер между ними. На рис. 4.3 показан граф K_8; в теории графов буква K с целочисленным индексом означает полный граф с соответствующим количеством вершин.

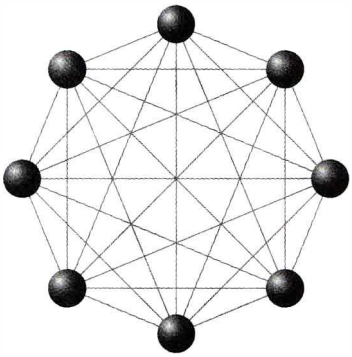


Рис. 4.3. K_8 — полный граф из восьми вершин

Граф или подграф, в котором существует путь от любой вершины к любой другой вершине, называется *связным*; граф, который не является связным, состоит из нескольких компонент связности[1].

[1] Компонентой связности графа является максимально связный подграф — множество вершин графа, такое, для которого существует путь от любой вершины множества к любой другой вершине множества, и при этом ни у одной вершины множества нет ребра, ведущего к вершине, не принадлежащей этому множеству.

Для данного графа G его дополнение G' также является графом с теми же вершинами; для любой пары вершин G' ребро между ними существует тогда и только тогда, когда такого ребра нет в графе G. На рис. 4.4 показан граф, который является дополнением к графу K_6; вместо того чтобы содержать все возможные ребра, он не имеет ни одного ребра.

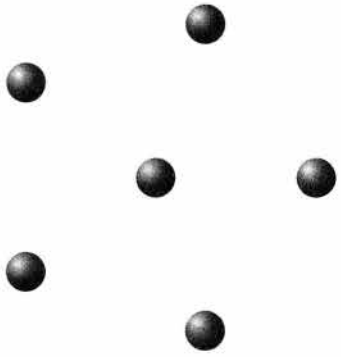


Рис. 4.4. Разъединенный граф с шестью компонентами связности размером 1

Математическое предупреждение

Символ штриха (') используется в математике для представления объекта, который был получен из другого объекта. Здесь я использую этот символ для обозначения графа G', который мы получаем из G, удаляя из него все ребра и добавляя ребра там, где их раньше не было. Это похоже на использование данного символа в теории множеств для обозначения дополнения множества.

При обсуждении языков в главе 13 используется альтернативная нотация — обозначение дополнения языка A как $\bar{A}$. Эта нотация (также широко распространенная) облегчает работу с дополнениями дополнений.

4.3.3. Петли и мультиграфы

Обычно мы работаем с простыми графами, то есть с графами, которые не содержат петель (ребро, которое заканчивается в той же вершине, в которой началось) или кратных ребер (несколько ребер, соединяющих одни и те же вершины). Говоря «граф» и не указывая иное, мы всегда имеем в виду простой граф. Граф с петлями можно назвать непростым графом, а граф с кратными ребрами — мультиграфом. Далее в книге, встретив слово «граф», считайте, что оно означает простой неориентированный граф[1], если не указано иное.

4.4. Представление графов

Когда мы говорим о размере графа, то обычно используем обозначение n для числа вершин и m — для числа ребер[2]. На рис. 4.3 $n = 8$ и $m = 28$, на рис. 4.4 — $n = 6$ и $m = 0$. Количество места в памяти, необходимое для хранения графа, зависит от того, как именно мы его

---

[1] Ориентированные графы мы рассмотрим в разделе 4.5.

[2] Обычно множество вершин обозначают буквой V, а множество ребер — буквой E, поэтому $|V| = n$, а $|E| = m$; то есть n — размер множества V, а m — размер множества E.

храним; как правило, графы хранятся в виде списков смежности и матриц смежности.

4.4.1. Представление графов в виде списков смежности

При использовании представления в виде списка смежности каждая вершина графа сохраняется вместе со списком смежных с ней вершин (рис. 4.5–4.8). Если реализовать это в виде множества связных списков, то объем занимаемого пространства составит $O(n + m)$[1]. В случае *разреженного* графа (с очень небольшим числом ребер) этот объем сократится до $O(n)$. Для *плотного* графа (графа с большим количеством ребер, такого как полный или почти полный граф) этот объем составит $O(n^2)$.

A: BCDEFGH
B: ACDEFGH
C: ABDEFGH
D: ABCEFGH
E: ABCDFGH
F: ABCDEGH
G: ABCDEFH
H: ABCDEFG

Рис. 4.5. Представление списка смежности графа, показанного на рис. 4.3

A:
B:
C:
D:
E:
F:

Рис. 4.6. Представление списка смежности графа, показанного на рис. 4.4

[1] У нас есть n связных списков, некоторые из них могут быть пустыми (если граф разъединенный). Каждое ненаправленное ребро присутствует в обоих списках, поэтому общее количество узлов в списках составляет $2m$. В сумме это дает $O(n + m)$ объема памяти.

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Рис. 4.7. Представление матрицы смежности графа, показанного на рис. 4.3

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Рис. 4.8. Представление матрицы смежности графа, показанного на рис. 4.4

4.4.2. Представление графов в виде матрицы смежности

Еще одним распространенным способом хранения графа является матрица смежности, которая представляет собой матрицу со следующими свойствами:

- каждая ячейка матрицы равна либо 0, либо 1;
- ячейка в позиции (i, j) равна 1 тогда и только тогда, когда существует ребро между вершинами i и j. Это верно и для ячейки в положении (j, i);
- из предыдущего пункта следует, что число единиц в матрице равно удвоенному числу ребер в графе;
- диагональные ячейки всегда равны нулю, потому что ни у одной вершины нет ребра, которое и начинается, и заканчивается в ней[1];

[1] Диагональ матрицы — это ячейки, для которых номер столбца совпадает с номером строки: (0, 0), (1, 1) и т. д. На рис. 4.7 приведен пример матрицы, которая имеет нулевые значения только на диагонали.

- матрица состоит из n строк и n столбцов и поэтому занимает n^2 места. Для плотного графа линейная зависимость от размера матрицы сохраняется[1].

В мультиграфе, где присутствуют петли, некоторые из этих условий не выполняются. В частности, значения могут быть больше 1 (потому что между двумя вершинами может существовать несколько ребер) и диагональ может быть ненулевой (петли могут соединять вершину с самой собой).

4.4.3. Представление графов в памяти

В памяти граф часто хранится в виде набора узлов. Каждый узел представляет одну вершину и содержит набор указателей на другие узлы, так что каждый указатель соответствует ребру, ведущему к другой вершине.

4.4.4. Выбор представлений

Выбор представления графа зависит от плотности графа и от того, как вы планируете его использовать. Для разреженного графа список смежности намного эффективнее по объему памяти, чем матрица смежности, поскольку нам не нужно хранить $O(n^2)$ нулей и можно легко перебрать все существующие ребра. Кроме того, в случае динамического графа (который изменяется со

[1] Размер матрицы — это сумма количества вершин (n) и количества ребер (m), но для плотного графа $m = O(n^2)$.

временем) в списке смежности проще добавлять и удалять вершины.

С другой стороны, в матрице смежности более эффективно организован доступ к ребрам. Чтобы определить, являются ли вершины i и j смежными, достаточно проверить, равно ли $A[i][j]$ единице. Не нужно сканировать весь список, что может занять до $O(n)$ времени. Таким образом, в матрице смежности поиск выполняется не только быстрее, но и занимает постоянное количество времени, что делает матрицы смежности лучшими для приложений, где необходима предсказуемость[1].

4.5. Направленные и ненаправленные графы

В задаче о кенигсбергских мостах все ребра графа были *ненаправленными*; если по мосту можно идти в одном направлении, то можно идти и в обратном. Графы, для которых это верно, называются ненаправленными или просто графами и описывают ситуации, в которых если (вершина) A связана с B, то B также связана с A. Например, если Алиса — двоюродная сестра Веры, то Вера также является двоюродной сестрой Алисы.

В *ориентированном* графе, или *орграфе*, каждое ребро имеет направление, в котором следует данная связь. Если Алиса любит проводить время с Верой и мы обо-

[1] В приложениях реального времени следует быть готовыми пожертвовать некоторой производительностью, чтобы получить более жесткие ограничения на максимальное время, которое может потребоваться для выполнения операции.

значаем это стрелкой из A в B, это еще не говорит о том, что Вера тоже любит проводить время с Алисой. Если это так, то также существует стрелка из B в A. Орграф является *симметричным*, когда для каждого ориентированного ребра в нем существует ребро, соединяющее те же две вершины, но направленное в противоположную сторону. Такой граф эквивалентен ненаправленному графу с одним ребром между каждой парой вершин, где есть пара направленных ребер, поэтому обычные графы можно рассматривать как особый случай орграфов.

И наоборот, возможна ситуация, при которой между любыми двумя вершинами может существовать только одно ориентированное ребро; такой граф называется *направленным* или *антисимметричным*. Например, если Алиса является родителем Влада, то Влад не может быть родителем Алисы (по крайней мере никаким социально приемлемым способом). Если взять ненаправленный граф и задать направление каждому из ребер (то есть превратить ребро в ориентированное), то получим направленный граф.

4.6. Циклические и ациклические графы

Один из способов классификации графов — разделить их на *циклические* или *ациклические*. Ациклический граф имеет не более одного пути между любыми двумя вершинами; другими словами, не существует пути a-b-c-a, где $\{a, b, c\}$ — различные вершины гра-

фа[1]. Циклический граф имеет хотя бы один цикл: можно найти путь, который начинается и заканчивается в одной и той же вершине (рис. 4.9). В случае ориентированных графов добавляется условие, что все ребра цикла должны иметь одинаковое направление — по часовой стрелке или против часовой стрелки; другими словами, в циклическом орграфе можно найти путь от вершины к самой себе, следуя по направлению стрелок. При программировании графовых алгоритмов необходимо соблюдать осторожность при обработке циклов, иначе программа может попасть в бесконечный цикл.

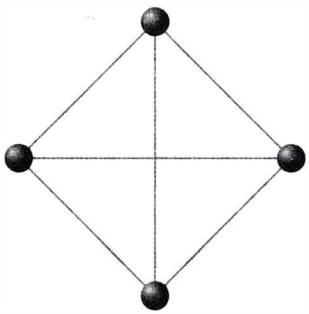


Рис. 4.9. Любой полный граф является циклическим

4.6.1. Деревья

Многие важные алгоритмы в теории графов оперируют с *деревьями*. Дерево — это просто связный граф,

[1] В данном случае на месте с также может быть несколько вершин — это может быть цикл *a-b-c-d-e-f-g-h-a*. Обратите внимание, что каждое ребро можно использовать только один раз: переход по пути *a-b-c* и затем возвращение назад по тем же ребрам — это не цикл!

не имеющий циклов. Как правило, удобно описывать дерево, начиная с корня; мы назначаем одну вершину корнем дерева и определяем остальные вершины по их отношению к корню. Следующие определения для дерева являются эквивалентными:

- ациклический граф, в котором появится *простой* (без повторяющихся вершин) цикл, если добавить в него любое ребро[1];
- связный граф, который перестанет быть связным, если удалить из него любое ребро;
- граф, в котором любые две вершины связаны уникальным простым путем.

Узлы дерева бывают двух типов: внутренние узлы (у которых есть хотя бы один дочерний элемент) и листья (у которых нет дочерних элементов) (рис. 4.10).

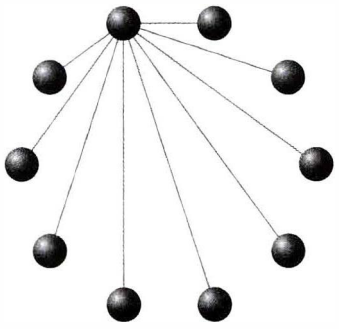


Рис. 4.10. Дерево с десятью вершинами. Это звезда, представляющая собой дерево ровно с одним внутренним узлом

[1] Это должен быть связный граф; понимаете ли вы почему?

4.6.2. Бесхордовые циклы

В большинстве случаев нас особенно интересуют бесхордовые циклы. *Хорда* — это ребро между двумя вершинами цикла, которое само не является частью цикла. Бесхордовый цикл — это цикл, который состоит хотя бы из четырех вершин и не содержит хорд (рис. 4.11). Другими словами, это цикл, в котором нет ребер между любыми двумя вершинами, не являющимися последовательными в цикле[1].

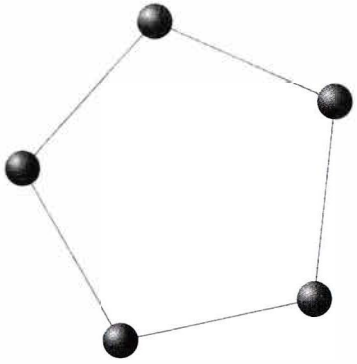


Рис. 4.11. C_5, бесхордовый цикл из пяти вершин

В главе 7 речь пойдет о нескольких классах графов, которые (по крайней мере частично) определяются отсутствием порожденных бесхордовых циклов, являющихся порожденными подграфами, содержащими циклы без хорд. Граф без порожденных бесхордовых

[1] Например, если у нас есть бесхордовый цикл *A-B-C-D-E-A*, то у *A* нет ребра, ведущего к *C* или *D*, поскольку эти вершины не стоят непосредственно перед или после *A* в цикле.

циклов называется, что неудивительно, хордальным графом.

4.7. Раскраска графа

Многие графовые задачи связаны с раскраской — способом маркировки вершин (или ребер) графа. Правильная раскраска вершин — такая, при которой смежные вершины (то есть вершины, между которыми есть ребро) раскрашены в разные цвета. Другими словами, раскраска — это разделение вершин на независимые множества.

> **Математическое предупреждение**
>
> Когда мы говорим о поиске раскраски графа, это не значит, что мы должны в буквальном смысле использовать цвета. Иногда при создании раскраски в роли цветов может выступать набор целых чисел.
>
> Даже если мы используем реальные цвета, компьютер все равно будет обрабатывать их как список целых чисел, а затем при визуализации преобразовывать целые числа в цвета.

Аналогично правильная раскраска ребер — это такая, при которой два ребра, инцидентные[1] одной и той же вершине, раскрашены в разные цвета. Если не указано иное, то, говоря о раскраске графа, мы будем иметь в виду правильную раскраску вершин. Первые резуль-

[1] Два ребра называются инцидентными, если они имеют общую вершину; аналогично две вершины, соединенные общим ребром, называются смежными.

таты раскраски графов включали в себя раскраску плоских графов[1] в виде карт. В гипотезе о четырех цветах[2], выдвинутой в 1852 году, говорится, что для правильной раскраски любой карты, состоящей только из связных областей[3] с границами конечной длины, требуется не более четырех цветов.

Доказательство этого утверждения было представлено Альфредом Кемпе в 1879 году и считалось общепринятым, пока в 1890 году не было доказано, что оно неверно[4]. В итоге задачу решили с помощью компьютера в 1976 году; теперь у нас есть алгоритмы квадратичного времени для любой четырехцветной карты[5]. До сих пор не существует доказательств, не требующих использования компьютера.

Несмотря на то что задача раскраски карты, в сущности, не представляла особого интереса для создателей карт, она интересна с теоретической точки зрения и имеет практическое применение. Так, судоку — это форма

---

1 Это графы, которые можно нарисовать так, чтобы никакие два ребра не пересекались, кроме как в вершине; подробнее см. в разделе 7.2.

2 Доказано, теперь это теорема о четырех цветах.

3 В соответствии с этим требованием континентальная часть США, Аляска и Гавайи будут считаться отдельными регионами. На самом деле Гавайи были бы несколькими регионами.

4 Первая опубликованная исследовательская работа автора этой книги была посвящена контрпримеру к ошибочному доказательству Кемпе.

5 Или же для любого плоского графа.

раскраски графа, «цветами» которого являются числа от единицы до девяти.

Хроматическое число графа — это количество цветов, необходимых для его правильной раскраски. Другая формулировка теоремы о четырех цветах состоит в том, что хроматическое число плоского графа — не больше четырех.

Очевидно, что хроматическое число графа, не имеющего ребер, равно единице (все вершины могут быть одного цвета). Для полного графа из n вершин хроматическое число равно n (каждая вершина смежна со всеми остальными вершинами, поэтому все вершины должны быть разных цветов).

Проверка того, может ли произвольный граф быть двухцветным, занимает линейное время[1] — достаточно окрасить одну вершину в красный цвет, затем окрасить все ее соседние вершины в синий, потом все соседние вершины этих вершин, которые еще не были окрашены, — в красный и т. д. Работа прекращается, когда либо все вершины окрашены, либо обнаружена вершина, у которой соседняя вершина имеет тот же цвет. А вот трехцветная раскраска является *NP*-полной задачей! Известно, что алгоритмы, определяющие, является ли данный граф k-раскрашиваемым, занимают экспоненциальное время. Однако, если знать, что граф принадлежит конкретному классу[2], найти раскраску можно за полиномиальное время.

---

[1] То есть время пропорционально сумме количества вершин и ребер.

[2] Например, идеальные графы, описанные в разделе 7.3.

Алгоритмы раскраски обычно используются в таких приложениях, как планирование, анализ данных, работа в сетях и т. п. Например, рассмотрим задачу назначения времени для собраний длительностью один час, когда на разные собрания приглашаются разные люди и используется разное оборудование. Мы представляем каждое собрание в виде вершины и добавляем ребро между двумя вершинами, если в них задействованы одни и те же человек или оборудование. Нахождение минимальной раскраски говорит нам о необходимости назначить разное время для встреч (рис. 4.12).

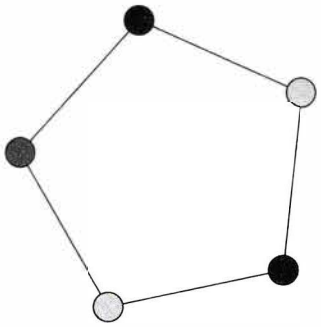


Рис. 4.12. Граф C_5 с минимальной раскраской

4.8. Взвешенные и невзвешенные графы

Вершины графа можно представить как локации, а ребра — как пути между ними, но в действительности не все пути имеют одинаковую длину. В *невзвешенном* графе ребра просто показывают, между какими вершинами есть прямой путь, но существуют также *взвешенные* графы, в которых каждому ребру присвоен

вес. Обычно, но не всегда эти веса являются неотрицательными целыми числами. Вес часто воспринимается как стоимость использования этого ребра.

То, что конкретно означает вес, зависит от того, что описывает граф. На графе крупных городов ребра с весами могут представлять расстояния в милях по кратчайшему шоссе между двумя городами. На электрической схеме веса представляют максимальный ток через данное соединение.

5 Структуры данных на основе графов

При проектировании алгоритмов мы часто используем абстрактные структуры данных в том смысле, что знаем свойства, которыми должна обладать структура, а не ее точная организация. Например, очередь с приоритетом — это структура данных FIFO (first-in, first-out — «первым пришел, первым ушел»), в которой элементы с более высоким приоритетом проходят вне очереди и обслуживаются раньше, чем элементы с более низким приоритетом. При выполнении реального алгоритма мы должны превратить эту абстрактную структуру данных в реальную. Так, очередь с приоритетом может быть реализована с помощью двоичного дерева поиска или же с помощью кучи. С кучей вы познакомились в подразделе 2.4.2; в этой главе более подробно рассмотрим ее внутреннее устройство.

5.1. Двоичные деревья поиска

Двоичное дерево поиска (Binary Search Tree, BST) (рис. 5.1) — это корневое двоичное дерево[1], которое рекурсивно определяется следующим образом: ключ

[1] Двоичное дерево — это дерево, каждый узел которого имеет не более двух дочерних узлов.

корня у него больше или равен ключу его левого потомка и меньше или равен ключу правого потомка (если он есть). Это также верно для поддерева, корнем которого является любой другой узел.

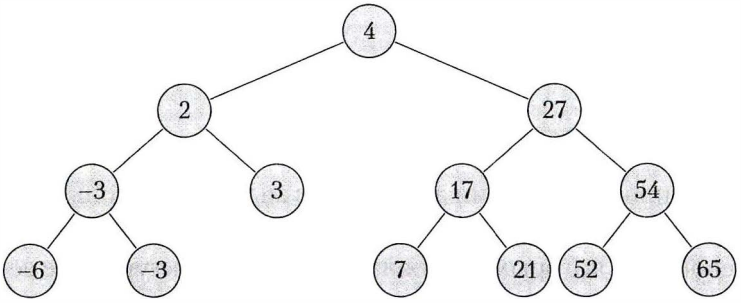


Рис. 5.1. Двоичное дерево поиска

Операции над двоичным деревом поиска занимают время, пропорциональное высоте дерева — длине самой длинной цепочки от корня (высота которого равна нулю) до листа. В общем случае[1] это $\Theta(\lg n)$[2], но в худшем случае (когда у каждого узла есть только один дочерний элемент, что, по существу, превращает дерево в связный список) высота составляет $O(n)$. Некоторые вариации двоичного дерева поиска гарантируют, что высота дерева будет $\Theta(\lg n)$, поэтому мы

---

[1] Предположим, что вы назначили корнем случайный узел. Тогда в среднем половина оставшихся узлов окажется слева от него, а половина — справа. Таким образом, ожидается, что количество узлов на каждом уровне дерева будет примерно вдвое больше, чем на предыдущем, следовательно, во всем дереве будет примерно $\lg n$ уровней.

[2] Θ означает, что время выполнения ограничено как сверху, так и снизу; высота дерева должна быть не менее $\lg n$.

можем быть уверены, что все операции завершатся за $O(\lg n)$ времени. На рис. 5.2 изображено несбалансированное двоичное дерево поиска.

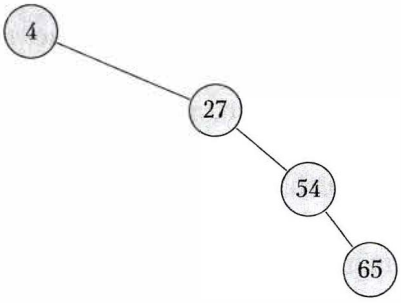


Рис. 5.2. Несбалансированное двоичное дерево поиска. Операции с ним занимают $O(n)$ времени

Двоичное дерево поиска может быть реализовано как коллекция связанных узлов, где каждый узел имеет ключ и указатели на левый и правый дочерние элементы и на родительский элемент. Поскольку узлы расположены так, что любой узел не меньше, чем любой из узлов его левого поддерева, и не больше, чем любой из узлов его правого поддерева, то можно вывести все ключи по порядку, выполнив упорядоченный обход дерева.

```
Процедура PrintInOrder(node x)
begin
    if x ≠ null then
        PrintInOrder(x.left);
        print x.key;
        PrintInOrder(x.right);
    end
end
```

Поиск по двоичному дереву выполняется просто: начиная с указателя на корень дерева и имея заданный ключ, мы сначала проверяем, есть ли у корня этот ключ. Если ключ корня меньше заданного, мы выбираем правый дочерний узел, если же он больше, то левый. Поиск прекращается, когда либо обнаружено совпадение ключей, либо поддерево, для которого мы пытаемся выполнить рекурсию, пусто. Чтобы найти наименьший элемент, мы просто всегда выполняем рекурсию для левого дочернего элемента, а чтобы найти наибольший элемент — для правого. В каждом случае мы будем проверять не более одного узла на каждом уровне дерева, поэтому время выполнения в худшем случае пропорционально высоте дерева.

Как создать и модифицировать двоичное дерево поиска? Первый добавленный узел становится корнем дерева. Чтобы добавить другие узлы, мы ищем значение ключа для вставки; обнаружив нулевой указатель, меняем его так, чтобы он указывал на новый узел, и делаем текущий узел родительским для нового узла[1].

Чтобы удалить узел d, нужно найти его в дереве и затем выполнить следующее:

- если у d нет потомков, просто присвоить null указателю родительского узла, который сейчас ссылается на d;

[1] Если в дереве допускаются дубликаты, нам может встретиться еще один или несколько узлов с таким же значением ключа, прежде чем мы найдем нулевой указатель.

- если у d есть один потомок, этот потомок занимает место d;
- если у d два потомка, то найти его предшественника или преемника (узлы, которые будут появляться непосредственно перед или после d при упорядоченном обходе) и также переместить его вверх, на место d (после чего может потребоваться обработать потомков этого узла так, как если бы он был удален).

5.2. Сбалансированные деревья двоичного поиска

Чтобы гарантировать, что операции над двоичным деревом поиска занимают $O(\lg n)$, а не $O(n)$ времени, необходимо ограничить высоту дерева. Если все ключи заранее известны, мы, конечно же, можем построить сбалансированное дерево (имеющее минимально возможную высоту). На практике элементы дерева меняются время от времени, поэтому мы допускаем, что его высота может быть больше минимальной, но при этом составит $\Theta(\lg n)$.

Двоичное дерево поиска с автоматической балансировкой — это дерево, которое автоматически сохраняет небольшую высоту (по сравнению с количеством требуемых уровней) независимо от добавления или удаления узлов. Деревьями с автоматической балансировкой

являются, в частности, красно-черные деревья[1], косые деревья[2] и декартовы деревья (treaps).

5.3. Кучи

Как показано в подразделе 2.4.2, куча — это структура данных, а именно почти полное двоичное дерево с корнем, ключ которого больше, чем ключ любого из его дочерних элементов (рис. 5.3). И это правило рекурсивно выполняется для любого поддерева, корнем которого является любой дочерний элемент[3]. «Почти полное» означает, что дерево максимально заполнено, за исключением, возможно, самого нижнего уровня, который заполняется слева направо.

---

[1] Красно-черное дерево — это двоичное дерево поиска, в котором все ключи хранятся во внутренних узлах, а листья — это нулевые узлы. Для таких деревьев также существуют дополнительные требования: каждый узел должен быть красным или черным, корень и все листья — черными, потомки красного узла — черными и каждый путь от узла к листу должен содержать одинаковое количество черных узлов. В результате этих ограничений путь от корня до самого дальнего листа не более чем в два раза превышает длину пути от корня до ближайшего листа, поскольку кратчайший возможный путь — это все черные узлы, а в самом длинном из возможных путей красные и черные узлы чередуются, поэтому ни один путь не может быть более чем в два раза длиннее другого.

[2] Косые деревья подробно рассмотрены в главе 33 второго тома.

[3] Или же корень может быть меньше, чем его дочерние элементы, такая куча называется неубывающей. Для согласованности далее мы будем использовать невозрастающую кучу.

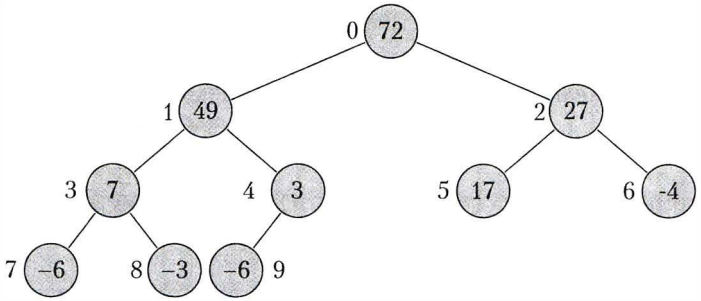


Рис. 5.3. Max-куча с узлами, помеченными их позицией

Предположим, что мы хотим реализовать кучу. В этой куче вершины нумеруются начиная с корня, а затем на каждом уровне — слева направо: корень — вершина 0, его левый потомок — вершина 1, правый потомок — вершина 2, самый левый внук корня — вершина 3 и т. д. (рис. 5.4).

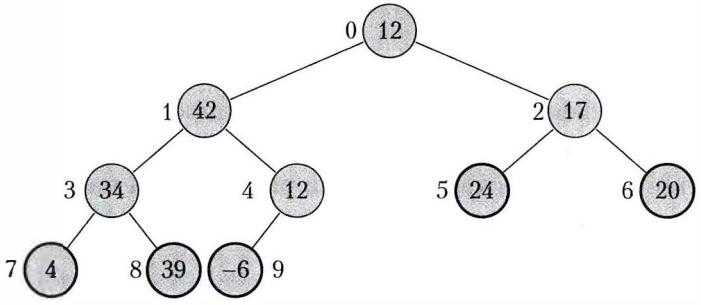


Рис. 5.4. В этом неупорядоченном двоичном дереве узлы с 5-го по 9-й являются корнями кучи размером 1

Обратите внимание, что каждая строка содержит в два раза больше узлов, чем строка, расположенная над ней.

Это означает, что метка левого дочернего элемента каждого узла как минимум вдвое плюс еще на единицу больше, чем номер родительского элемента[1]. Для любого узла k номер его родителя равен $\left\lfloor \frac{k-1}{2} \right\rfloor$, а его дочерние элементы (если они есть), имеют номера $2k + 1$ и $2k + 2$.

Математическое предупреждение

Приведенные выше L-образные операторы представляют функцию округления до ближайшего целого в меньшую сторону, что означает усечение любой дроби; $\lfloor 2.8 \rfloor = 2$. Обратная функция, округление до ближайшего целого в большую сторону, записывается как $\lceil 2.1 \rceil = 3$.

Если вы предпочитаете использовать массив с отсчетом от 1, то родительским узлом для k будет узел номер $\left\lfloor \frac{k}{2} \right\rfloor$, а его дочерние элементы будут иметь номера $2k$ и $2k + 1$.

Можно эффективно определить позицию родительского или дочернего узла, выполнив умножение посредством сдвига битов; если куча реализована на основе массива, то мы сразу получим положение нужного узла.

5.3.1. Построение кучи

Теперь, когда мы определили, как куча хранится в памяти, мы готовы ее построить. Куча строится рекур-

[1] Или же просто удвоить массив с отсчетом от 1.

сивно: создаются маленькие кучи и объединяются в большие. Для начала предположим, что у нас есть массив A размером n, который мы интерпретируем как описанное ранее двоичное дерево (см. рис. 5.4). Обратите внимание, что каждый лист дерева (любой элемент в позиции от $\left\lfloor \frac{n}{2} \right\rfloor$ до $n - 1$) является корнем кучи размером 1.

Теперь рассмотрим внутренние узлы на втором снизу уровне дерева. Если ключ такого узла больше, чем ключ любого из его дочерних элементов, то он является корнем невозрастающей кучи. Если же нет, то мы поменяем его с дочерним элементом, ключ которого больше, чтобы этот дочерний элемент стал корнем невозрастающей кучи, и перейдем к новому дочернему элементу, чтобы убедиться, что новое поддерево сохраняет свойство кучи. Затем мы продолжим объединять кучи таким образом, пока не будет обработан весь массив (рис. 5.5–5.7).

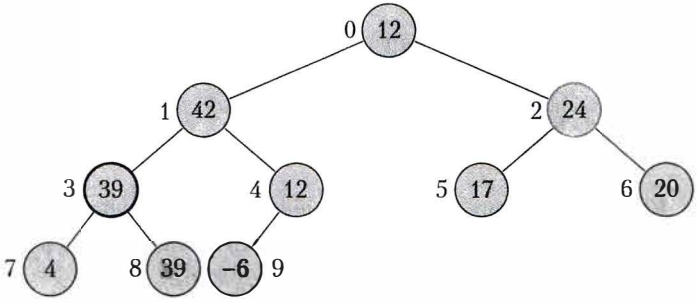


Рис. 5.5. Если лист больше, чем его родитель и сестра, то этот лист меняется местами с родителем

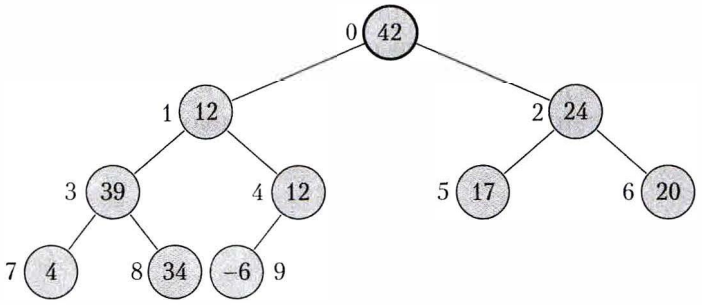


Рис. 5.6. Перенос дочернего элемента на более высокий уровень может уничтожить свойство кучи поддерева, корневым элементом которого был этот дочерний элемент. В данном случае новый узел 1 меньше, чем один из его дочерних узлов

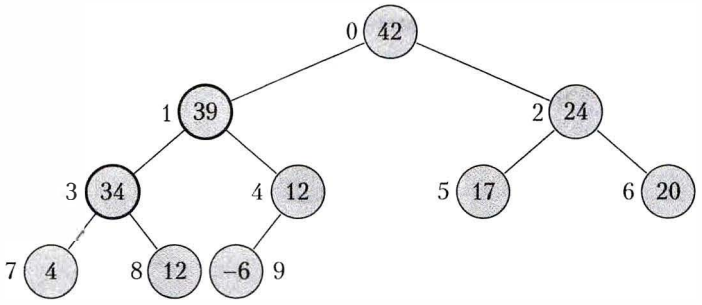


Рис. 5.7. В итоге ни одно значение узла не превышает значение его родительского элемента, так что граф удовлетворяет свойству кучи

Этот процесс можно представить в виде следующего кода. Кроме `.length`, который, как обычно, озна-

чает количество элементов массива, мы определяем `.heapSize` — число элементов массива, которые являются частью кучи.

Алгоритм 1. BuildMaxHeap

Входные данные: массив A
Выходные данные: массив A, отсортированный как невозрастающая куча

```
begin
    A.heapSize = A.length
    for i = ⌈A:length / 2⌉ downto 1 do
        MaxHeapify(A, i)
    end
end
```

В `BuildMaxHeap` мы перебираем все внутренние узлы, начиная с самого нижнего. Для каждого выбранного узла мы проверяем, больше ли он, чем каждый из его дочерних узлов, которых может быть не более двух. Если это так, то, поскольку каждый дочерний элемент уже является корнем невозрастающей кучи, этот узел теперь является корнем новой невозрастающей кучи большего размера.

Если это не так, то мы меняем данный элемент на больший из двух его дочерних элементов. Это может привести к тому, что поддерево с корнем в данном дочернем элементе больше не будет удовлетворять свойству кучи (в случае если новый элемент был меньше

не только чем его дочерний элемент, но также и дочерний элемент этого дочернего элемента, как было в случае на рис. 5.5), поэтому мы снова вызываем алгоритм `MaxHeapify` для этого поддерева.

Алгоритм 2. MaxHeapify

Входные данные: массив A, индекс i

```
begin
    left = LeftChild(i)
    right = RightChild(i)
    if left < A.heapSize и A[left] > A[largest]
        then
            largest = left
        else
            largest = i
    end
    if right < A.heapSize и A[right] > A[largest] then
        largest = right
    end
    if largest ≠ i then
        Swap(A[i], A[largest])
        MaxHeapify(A, largest)
    end
end
```

В худшем случае каждый раз при объединении куч новый корень должен перемещаться в самое основание дерева. Каждый узел, который обрабатывается таким образом, может быть сравнен с $O(\lg n)$ потомками, причем каждое сравнение занимает постоянное количество

времени. Умножив это на n узлов, получим общее время выполнения алгоритма $O(n \lg n)$.

Алгоритм 3. Heapsort

Входные данные: массив A

Выходные данные: отсортированный массив A

```
begin
    BuildMaxHeap(A)
    for i = A.length downto 2 do
        Swap(A[1], A[i])
        A.heapSize = A.heapSize – 1
        MaxHeapify(A, 1)
    end
end
```

В алгоритмах `BuildMaxHeap` и `MaxHeapify` параметр `.heapSize` может показаться избыточным. Проверка того, является ли какой-либо узел меньше `A.heapSize`, всегда будет возвращать `true`, поскольку размер кучи задан равным размеру массива. Однако при запуске `Heapsort` этот узел отмечает конец оставшихся неотсортированных элементов кучи.

В алгоритме `Heapsort` после построения кучи корень (который является самым большим элементом) перемещается в конец массива, где он и остается. Затем мы уменьшаем значение `heapSize` и снова вызываем `MaxHeapify` для остальной части массива.

Каждый раз, когда значение `heapSize` уменьшается, оставшиеся `heapsize - 1` элементов массива, отличные

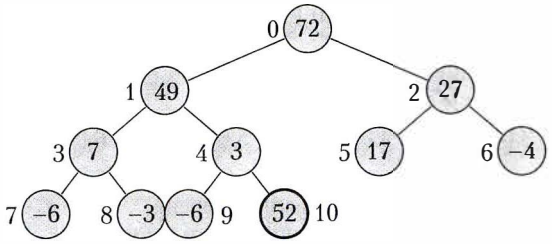
от корня, образуют кучу и вызов `Heapify` для корня восстанавливает свойство кучи. Элемент, перемещенный в конец массива, был самым большим из оставшихся элементов, поэтому после k-й итерации в позициях `[heapsize ... n]` находятся k самых больших элементов в отсортированной последовательности.

5.3.2. Добавление элементов в кучу

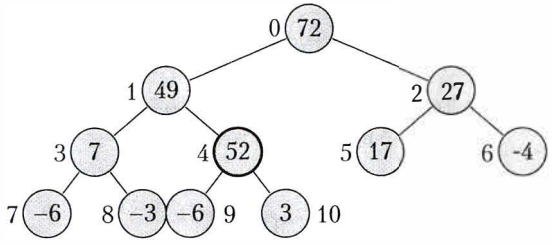
Элемент, добавляемый в кучу, вставляется в ближайшее доступное место (то есть в первую пустую ячейку массива) (рис. 5.8). Если при этом новый элемент больше родительского элемента, это может нарушить свойство неубывания. В таком случае нужно поднять этот элемент вверх, рекурсивно меняя его местами с родительскими элементами, пока они меньше этого элемента. Поскольку максимальная высота дерева равна $O(\lg n)$ и каждое сравнение/обмен занимает время $O(1)$, общее время вставки равно $O(\lg n)$.

5.3.3. Удаление корня из кучи

Чтобы извлечь первый элемент из кучи, достаточно удалить первый элемент массива; однако при этом в куче остается пустое место, которое мы заменяем последним элементом кучи, при необходимости оставляя почти полное дерево (рис. 5.9). Затем мы поднимаем новый корень вверх, заменяя его дочерними узлами с бóльшими значениями, пока дерево снова не приобретет свойство кучи.



Мы помещаем новый элемент (с ключом 52) в первую открытую ячейку, из-за чего куча может потерять (и в данном случае теряет) свойство неубывания



Мы восстанавливаем свойство кучи, поднимая новый узел в правильную позицию

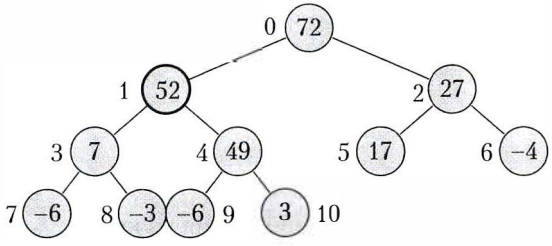
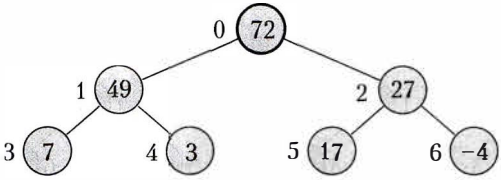
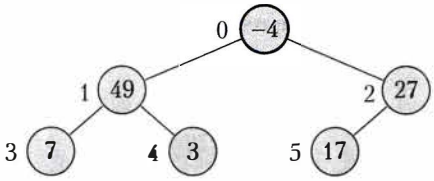


Рис. 5.8. Добавление элемента в кучу



Чтобы удалить корень,
мы заменяем его последним элементом
с самого низкого уровня кучи



Затем мы добавляем новый корень
в нужное место в куче

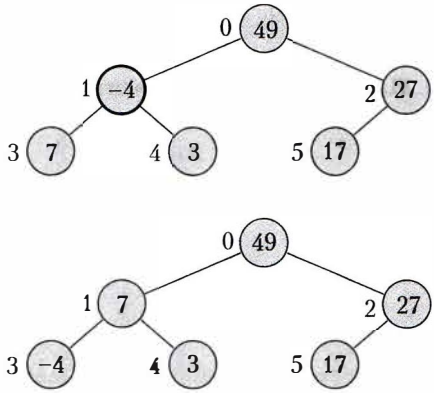


Рис. 5.9. Удаление корня из кучи

6 Хорошо известные графовые алгоритмы

6.1. Введение

Есть алгоритмы на графах, которые встречаются в различных приложениях настолько часто, что знать о них вы просто обязаны. Многие из них представляют собой различные способы сортировки графа или поиска подграфа с определенным свойством. Понимание относительных преимуществ этих фундаментальных алгоритмов — ключ к определению, когда использовать каждый из них в практических приложениях.

Эта глава начинается с описания алгоритмов поиска в ширину (Breadth-First Search, BFS) и в глубину (Depth-First Search, DFS), которые превращают произвольный граф в дерево поиска. В обоих случаях мы назначаем одну из вершин графа корнем дерева и рекурсивно исследуем ребра каждой вершины, пока все вершины графа[1] не будут добавлены в остовное дере-

[1] То есть любая вершина, которой можно достичь, переходя по ребрам от корня; в случае связного графа это будут все вершины.

во[1]. Рассмотрим, для каких типов приложений полезен каждый вид дерева поиска, а затем перейдем к поиску кратчайших путей.

При анализе времени выполнения алгоритмов на графах обычно используются две системы обозначений. Одна из них — это обозначение числа вершин буквой n, а числа ребер — буквой m. Другая — указать размеры множеств вершин и ребер. Для графа G множество вершин обозначается как $V(G)$ [читается «вершины G»), а множество ребер — как $E(G)$ (читается «ребра G»), поэтому $n = |V(G)|$, а $m = |E(G)|$.

Математическое предупреждение

Напомню, что в математике заключение значения в прямые скобки означает получение абсолютного значения или размера; здесь это означает размер множества, поэтому $|V(G)|$ можно прочитать как размер множества вершин G. Если G подразумевается по умолчанию, то можно обращаться к множествам просто как V и E.

6.2. Поиск в ширину

При поиске в ширину мы сначала исследуем все вершины, которые примыкают к корню; затем исследуем

[1] Остовное дерево графа — это дерево, которое содержит все вершины графа.

все вершины, смежные с соседями корня, которые еще не были исследованы, и т. д. Таким образом, вначале мы исследуем все вершины, которые находятся на расстоянии k от корня (который обычно обозначается буквой s, от слова *source* — «источник»), и лишь затем переходим к вершинам на расстоянии $k + 1$.

Когда алгоритм завершает работу, глубина каждого узла дерева — это минимальное количество ребер (то есть длина кратчайшего пути), по которому можно добраться до этого узла из s как в дереве, так и в исходном графе. Чтобы найти этот путь, нужно идти по указателям на родительские узлы, пока не будет достигнут узел s.

Каково время выполнения поиска в ширину? При инициализации для каждой вершины инициализируются три упомянутых выше свойства, что занимает постоянное время для каждой вершины, в сумме $O(n)$. Затем каждая вершина добавляется в очередь, удаляется из очереди, и каждое из свойств изменяется не более одного раза — каждая из этих операций занимает $O(1)$ времени, так что для всех вершин снова получаем $O(n)$. Наконец, мы перебираем всех соседей каждой вершины, чтобы проверить, отмечены ли они, что занимает еще $O(m)$ времени. В сумме получаем общее время выполнения $O(n + m)$. Для хранения графа требуется $n + m$ места в памяти, а очередь занимает $O(n)$ места, поэтому в сумме требуется пространство объемом $O(n + m)$. Это также означает, что алгоритм поиска в ширину работает за линейное

время по отношению к размеру входных данных (который равен $n + m$).

Алгоритм 4. Поиск в ширину

Входные данные: произвольный граф, корнем которого выбрана одна вершина, и все вершины имеют такие свойства:

- distance (расстояние) — изначально равно бесконечности;
- parent (родитель) — изначально равно нулю;
- marked (помечена) — изначально равно false.

Выходные данные: остовное дерево графа, каждая вершина которого максимально приближена к корню

```
begin
    Инициализировать очередь Q
    Задать s.distance = 0
    Пометить s
    Добавить в очередь s
    while Q не пустая do
        Удалить из очереди u
        foreach вершина v ∈ Adj(u) do
            if v.marked then
                продолжить
            end
            Задать v.parent = u
            Задать v.distance = u.distance + 1
            Пометить v
            Добавить в очередь v
        end
    end
end
```

6.3. Применение поиска в ширину

Как показано ранее, поиск в ширину позволяет найти длину кратчайшего пути от исходной вершины *s* до любой другой вершины графа за линейное время. Затем мы можем найти сам путь, следуя по родительским указателям от исходного узла до корня дерева, за время, пропорциональное длине пути. Это означает, что алгоритм поиска в ширину полезен для всех задач, где нужен поиск кратчайших путей. Вот несколько примеров таких задач.

6.3.1. Навигационные системы

Рассмотрим задачу построения маршрутов с помощью GPS. Если картографическая система хранит данные о локальной области в виде графа, вершинами которого являются адреса (или пересечения), а ребрами — улицы (или, точнее, короткие отрезки улиц), то она может выполнять поиск в ширину для дерева, источник которого — ваше текущее местоположение.

6.3.2. Проверка, является ли граф двудольным

При запуске поиска в ширину, если существует ребро между данной вершиной и уже отмеченной вершиной, расстояние которой либо такое же, либо меньше в степени 2, этим двум вершинам будет присвоен один и тот же цвет и граф не будет двудольным. Кроме того, ребро

вместе с одним или несколькими путями к наименьшему общему предку двух вершин является нечетным циклом, который будет признаком того, что граф не двудольный[1]. Двудольные графы используются в теории кодирования[2], в сетях Петри[3], в анализе социальных сетей и облачных вычислениях.

6.4. Поиск в глубину

Как и при поиске в ширину, при поиске в глубину мы начинаем с исходной вершины и рекурсивно проходим остальную часть графа. Но только теперь мы продвигаемся настолько глубоко, насколько можем, по одному пути и только потом исследуем другие пути.

Представьте, что вы изучаете собак. Студент, использующий алгоритм поиска в ширину, может начать с изучения названий всех пород — от австралийской короткохвостой пастушьей собаки до японского шпица — и лишь потом углубиться в детали. Тот же, кто использует поиск в глубину, начнет с австралийской короткохвостой, которая является пастушьей породой, поэтому он прочитает все о пастушьих собаках, что приведет его к чтению

---

1 Сертификат — это доказательство того, что ответ, возвращаемый программой, является правильным; подробнее об этом вы прочитаете в главе 19 второго тома.

2 Теория кодирования изучает, в частности, криптографию, сжатие данных и исправление ошибок.

3 Сети Петри используются для моделирования поведения систем.

о стадах, это, в свою очередь, — к истории о домашних животных, что приведет к... В общем, идею вы поняли. Если область поиска слишком велика или даже бесконечна, может использоваться модифицированная версия поиска в глубину, которая работает только до указанной глубины.

Алгоритм 5. Поиск в глубину

Входные данные: произвольный граф, у каждой вершины которого есть свойства distance (расстояние) (изначально равно нулю) и marked (помечена) (изначально имеет значение false), а также исходная вершина s

Выходные данные: остовное дерево графа

```
begin
    Инициализировать стек S
    S.Push(s)
    while S не пустой do
        u = S.Pop(s)
        if u.marked then
            продолжить
        end
        Пометить u
        foreach вершина v ∈ Adj(u) do
            if v.marked then
                продолжить
            end
            Задать v.parent = u
            S.Push(v)
        end
    end
end
```

Практическое применение

Недавно меня попросили помочь решить задачу планирования. Было несколько заданий, которые требовалось выполнить, со следующими ограничениями:

- ни одно задание нельзя начать раньше определенного времени;
- каждое задание имеет максимальное время выполнения;
- у задания может быть ноль или более заданий, от которых оно зависит, и ноль или более заданий, которые зависят от него; выполнение задания нельзя начать, пока не будут выполнены все задания, от которых оно зависит. Циклических зависимостей нет (поэтому это орграф частично упорядоченного множества);
- на выполнение всех заданий может уйти не более 24 часов.

Поиск кратчайшего пути — распространенная задача, но в данном случае мы фактически хотели найти самый длинный путь для каждой задачи, что и сделали, используя модификацию алгоритма поиска в глубину. Это позволило нам построить такой порядок выполнения заданий, который гарантировал, что во время выполнения любого задания все предки этого задания в графе зависимостей уже были выполнены, так что самое длительное время выполнения задания было равно сумме самого позднего времени, требуемого для выполнения любого из родителей данного задания, и количества времени, отведенного для самого задания. Повторное выполнение алгоритма в обратном порядке

позволило выявить все цепочки зависимостей, которые будут выполняться после выделенного времени, что позволило выяснить, какие задачи необходимо оптимизировать.

6.5. Кратчайшие пути

Рассмотрим задачу наискорейшего перемещения из одного места в другое. В теории графов это задача кратчайшего пути: найти маршрут между двумя вершинами, имеющий наименьший вес. В невзвешенном графе это просто путь с наименьшим количеством ребер; во взвешенном графе это путь с наименьшим суммарным весом ребер.

Рассмотрим вариации этой задачи.

- **Кратчайший путь из одной вершины.** Найти кратчайший путь от исходного узла ко всем остальным узлам графа. Пример: узнать кратчайший путь от пожарной части до каждой точки города.
- **Кратчайший путь в заданный пункт назначения.** Найти кратчайший путь от каждого узла графа до данной точки назначения. Это просто задача поиска кратчайшего пути из одной вершины, для которой направление всех ребер изменено на противоположное. Например, мы можем захотеть узнать кратчайший путь до больницы из любой точки города.
- **Кратчайший путь между всеми парами вершин.** Найти кратчайший путь между всеми парами узлов графа. В идеале наш GPS сможет построить наилучший маршрут из любой точки в любую точку.

6.5.1. Алгоритм Дейкстры

Алгоритм Дейкстры[1] изначально определял кратчайший путь между двумя узлами графа, но был расширен таким образом, что теперь он решает задачу поиска кратчайшего пути из одной вершины. Этот алгоритм применим к любому взвешенному графу (напомню, что невзвешенный граф — это просто взвешенный граф, у которого вес каждого ребра равен 1), у которого все ребра имеют неотрицательный вес.

Алгоритм 6. Алгоритм Дейкстры

Входные данные: граф G и исходная вершина s
Выходные данные: расстояние от s до любого другого узла графа G
Инвариантное: S — множество узлов, для которых определены кратчайшие пути

```
begin
    Инициализировать множество вершин S нулем
    Инициализировать очередь с приоритетами Q
    и поместить в нее все вершины G
    while Q не пустая do
        u = Q.ExtractMin()
        S = S ∪ {u}
        foreach вершина v ∈ Adj(u) do
            Relax(u, v, w)
        end
    end
end
```

[1] www-m3.ma.tum.de/twiki/pub/MN0506/WebHome/dijkstra.pdf.

В алгоритме Дейкстры мы начинаем с исходной вершины s и множества узлов S (в настоящее время пустого), расстояние до которых было определено. Мы добавляем все элементы графа в очередь с приоритетом на основе их расстояния от s; сначала расстояние для s равно нулю; у всех остальных узлов расстояние равно бесконечности. Мы удаляем из очереди наименьший элемент (которым первоначально будет s) и «ослабляем» все его ребра. Далее мы продолжаем удалять наименьшие элементы и «ослаблять» его ребра до тех пор, пока очередь не станет пустой, после чего каждому узлу присваивается длина его кратчайшего пути.

Алгоритм 7. «Ослабление ребра»

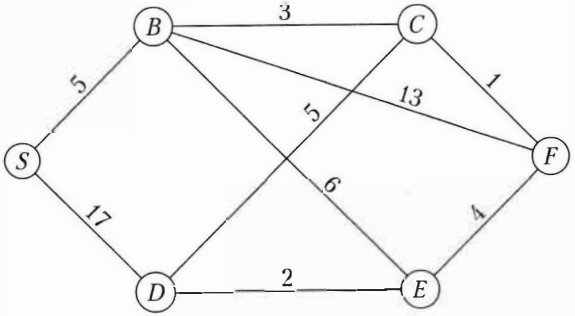
Входные данные: смежные вершины u и v и вес w ребра между ними

Выходные данные: v.d — вес найденного кратчайшего пути от s до v; если путь проходит через ребро от u до v, то v.π (родительский элемент v) присваивается значение u

```
begin
    if (v.d > u.d + w(u, v)) then
        v.d = u.d + w(u, v)
        v.π = u
    end
end
```

«Ослабление» ребра в данном случае означает, что мы берем приоритет текущего узла u, прибавляем к нему расстояние w до нового узла v и сравниваем сумму с текущим приоритетом v. Если сумма оказывается меньше существующего значения $v.d$, это значит, что мы нашли более короткий путь к v и можем заменить старое значение;

мы также помечаем u как нового родителя для v. Каждый раз, когда узел удаляется из очереди с приоритетом, мы знаем, что уже нашли кратчайший путь к этому узлу, потому что любой более короткий путь должен был бы пройти через уже обработанные узлы.



S	B	C	D	E	F
0	1	∞	∞	∞	∞
—	**5**$_S$	∞	17$_S$	∞	∞
—	—	**8**$_B$	17$_S$	11$_B$	18$_B$
—	—	—	13$_C$	11$_B$	**9**$_C$
—	—	—	13$_C$	**11**$_B$	—
—	—	—	**13**$_C$	—	—

Рис. 6.1. Поиск всех кратчайших путей из множества S с использованием алгоритма Дейкстры. Каждая строка таблицы представляет один шаг, на котором мы обрабатываем ранее необработанную вершину с наименьшим весом

Это жадный алгоритм[1], то есть на каждом шаге он делает все, что кажется лучшим на данный момент. Поскольку

[1] Подробнее о жадных алгоритмах и других подходах к решению задач рассказывается в части IV.

мы считаем, что каждый узел в множестве S помечен длиной его кратчайшего пути, алгоритм Дейкстры (в отличие от многих других жадных алгоритмов) гарантированно возвращает оптимальное решение.

«Ослабление» ребра занимает постоянное время, что в сумме дает время $O(m)$ для обработки всех ребер. Поиск элемента с наименьшим приоритетом для последующей обработки занимает время $O(n)$, и это выполняется $O(n)$ раз, что дает общее время $O(n^2)$. В сумме это дает $O(n^2 + m)$, где $m \leq n^2$, поэтому сложность алгоритма Дейкстры составляет $O(n^2)$.

7 Основные классы графов

Многие задачи достаточно сложны для произвольных графов (являются *NP*-сложными), но имеют эффективные (или даже тривиальные) решения для графов определенного класса. И наоборот, задача может не иметь решения на графах определенного класса. Таким образом, мы часто можем избежать проблем, если удастся показать, что данная задача относится к определенному классу графов.

7.1. Запрещенные подграфы

Характеризация *запрещенными подграфами* для класса графов определяет набор структур, которые не должны появляться в графе; наличие или отсутствие таких структур говорит о том, принадлежит ли данный граф к этому классу. Такие запрещенные подструктуры могут быть определены несколькими способами.

- **Графы.** Граф может принадлежать к классу, только если он не содержит подграфа из (возможно, бесконечного) множества. Например, двудольные гра-

фы — это только такие графы, которые не содержат нечетных циклов.

- **Индуцированные графы.** То же самое, что и в предыдущем случае, за исключением того, что здесь нас интересуют только индуцированные подграфы (напомню, что это некоторое подмножество вершин графа вместе со всеми ребрами между этими вершинами). Например, хордальные графы — это такие графы, которые не содержат индуцированного бесхордового цикла длиной не менее четырех.
- **Гомеоморфные графы.** Два графа называются гомеоморфными, если один из них можно получить из другого, удалив вершины второй степени[1] и свернув все ребра в одно.
- **Миноры графа.** Минором графа называется подграф, получаемый посредством стягивания ребер, где стягивание ребер происходит, когда мы выбираем две смежные вершины и объединяем их в одну.

Как вы увидите в следующем разделе, класс графов может иметь несколько характеризаций запрещенными подграфами разных типов.

7.2. Планарные графы

В главе 4 описана задача кенигсбергских мостов, которая включает в себя представление карты в виде графа. Класс графов, представляющих карты на плоскости

[1] Степень вершины — это количество ее ребер. В данном случае это означает, что все вершины имеют ровно два ребра.

(или сфере[1]), имеет специальное название: планарные графы. Формально планарный граф — это такой граф, который может быть построен на плоскости, следовательно, он может быть нарисован так, что все его ребра пересекаются только в вершинах графа. Любая плоская карта может быть представлена в виде плоского графа: для этого достаточно заменить каждую область вершиной и провести ребра там, где раньше были общие границы.

Теорема Куратовского классифицирует планарные графы в терминах запрещенных подграфов: планарные графы — это только такие графы, которые не содержат подграфов, гомеоморфных K_5 (полный граф из пяти вершин) (приведен на рис. 7.1) или $K_{3,3}$ (полный двудольный граф из шести вершин, см. раздел 7.4) (рис. 7.2).

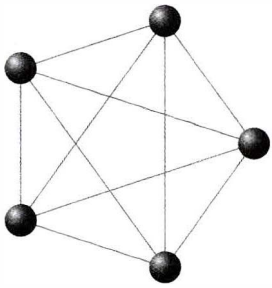


Рис. 7.1. K_5 — полный граф из пяти вершин

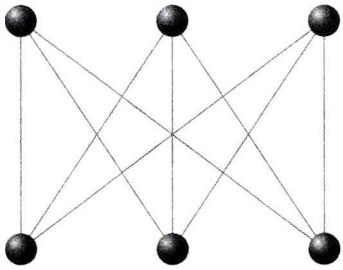


Рис. 7.2. Граф $K_{3,3}$, также известный как «домики и колодцы»

[1] Плоскость — это просто сфера с удаленным северным полюсом. Прости, Санта.

Рассмотрим следующее

Предположим, у нас есть граф K_5, который не является планарным, — нет возможности нарисовать его на плоскости хотя бы без одного пересечения ребер. Если добавить вершину в середину только одного ребра, это не устранит пересечение, поэтому новый граф также не будет планарным. Именно поэтому запрещенным является любой подграф, гомеоморфный K_5 или $K_{3,3}$.

Теорема Вагнера классифицирует планарные графы как такие, которые не содержат те же подграфы в качестве миноров.

Формула Эйлера гласит, что если конечный связный граф из v вершин, e ребер и f граней можно представить на плоскости без пересечений ребер[1], то $v - e + f = 2$.

Определение планарного графа (такого, который может быть построен на плоскости без пересечения ребер), естественно, означает, что такие графы удобны для отображения, так как легче увидеть, куда ведут все ребра (и фактически каждый такой граф можно нарисовать так, чтобы все ребра были отрезками прямых линий[2]). Приложение для рисования графов может разбить граф на плоские компоненты и объединить их, чтобы получить «более приятную» визуализацию. Это также имеет практическое применение в таких областях, как проектирование электронных схем.

---

[1] Это, конечно, доказывает, что граф планарный.

[2] Теорема Фари.

Планарные графы можно разбить на более мелкие части путем удаления $O(\sqrt{n})$ вершин, что помогает в разработке алгоритмов типа «разделяй и властвуй» и при динамическом программировании (см. главу 10) на планарных графах.

7.3. Совершенные графы

В разделе 4.7 представлена концепция раскраски графа, где хроматическое число графа — это минимальное количество цветов, необходимое для его правильной раскраски. Для графа, который содержит полный подграф (порожденный подграф, содержащий все возможные ребра) размером k, очевидно, хроматическое число должно быть не менее k. Совершенным графом называется такой граф, для которого это строгое равенство (хроматическое число строго равно k) и в котором это строгое равенство для хроматического числа также выполняется для всех порожденных подграфов. Другими словами, граф совершенен тогда и только тогда, когда для этого графа и для всех его порожденных подграфов хроматическое число равно размеру наибольшей клики[1].

Свойство совершенности является *наследуемым*; это означает, что если граф совершенен, то все его порожденные подграфы также совершенны. Наследуемость совершенных графов следует из приведенного выше

[1] Словами «клика» и «полный подграф» обозначается одно и то же понятие. Так бывает.

определения: если все порожденные подграфы графа удовлетворяют условию совершенности, то очевидно, что все порожденные подграфы этих подграфов также должны удовлетворять этому условию.

В действительности существует несколько эквивалентных определений совершенного графа[1]. Сильная теорема о совершенных графах характеризует их как такие графы, которые не содержат нечетных отверстий (порожденных бесхордовых циклов нечетной длины) или антиотверстий (дополнений этих циклов). Слабая теорема о совершенных графах характеризует их как дополнения к совершенным графам (то есть дополнение каждого совершенного графа само по себе является совершенным). Эта теорема непосредственно следует из доказательства сильной теоремы о совершенных графах. Еще одной характеризацией совершенных графов является то, что для таких графов произведение размеров наибольшей клики и наибольшего независимого множества больше или равно числу вершин графа. Это также верно для всех порожденных подграфов.

В число задач, которые являются *NP*-сложными на произвольных графах, но решаются за полиномиальное время на совершенных графах, входит раскраска графа, определение максимальной клики и максимального

[1] Чтобы доказать, что эти определения действительно эквивалентны, потребовалось приложить усилия. Подробности читайте в книге Мартина Голамбика (Martin Golumbic) Algorithmic Graph Theory and Perfect Graphs.

независимого множества. Подклассы совершенных графов (некоторые из них рассмотрены в этой главе) включают в себя двудольные графы (то есть такие, у которых хроматическое число равно двум), хордальные графы (такие, у которых нет хордовых циклов длиной больше трех), графы сравнимости (которые отражают частично упорядоченное множество) и подмножества этих классов[1]. Как мы увидим в следующем разделе, эти подклассы являются совершенными графами с дополнительными ограничениями.

7.4. Двудольные графы

Двудольные графы — это графы, для которых хроматическое число равно двум. Здесь они рассмотрены главным образом для того, чтобы вы увидели, насколько легко можно распознать класс графа. Для этого нужно выбрать одну вершину графа из каждого связного компонента, присвоить ей первый цвет и добавить ее в очередь. Каждый раз, когда вершина удаляется из очереди, рассматриваются все ее соседи и, если какая-либо из них еще не окрашена, ей присваивается противоположный цвет, после чего она добавляется в очередь. Если соседняя вершина уже раскрашена в тот же цвет, что и выбранная вершина, то граф не является двудольным (и две вершины образуют часть нечетного

[1] Эти подмножества включают в себя интервальные графы — хордальные графы, которые можно изобразить в виде набора последовательных интервалов и лесов — графов, в которых каждый связный компонент представляет собой дерево.

цикла); в противном случае мы получаем на выходе 2-раскраску графа.

Двудольные графы применяются в сетях Петри (которые используются для описания распределенных систем) и для множества других известных задач, включая задачу об устойчивом браке[1] и задачу назначения[2].

7.5. Интервальные графы

Интервальные графы — это графы, которые можно изобразить в виде набора пересекающихся отрезков, расположенных вдоль общей линии, где каждый отрезок представляет вершину; две вершины являются смежными тогда и только тогда, когда соответствующие сегменты линии пересекаются.

Как и совершенные графы, подклассом которых они являются, интервальные графы имеют несколько характеризаций. Характеризация запрещенными подграфами формулируется в терминах *астероидальных троек*. Они представляют собой наборы из трех вершин графа,

---

[1] В задаче об устойчивом браке каждый человек из группы A сопоставляется с человеком из группы B таким образом, чтобы не было пар, в которых бы он предпочитал того, кто также предпочитает его.

[2] В задаче назначения каждый агент выполняет ровно одно задание (разные задания могут стоить по-разному, в зависимости от того, какой агент их выполняет); задания назначаются таким образом, чтобы свести к минимуму общую стоимость.

где между любыми двумя вершинами существует такой путь, который избегает окрестности третьей вершины. Интервальные графы — это такие графы, которые являются хордальными и не содержат астероидальных троек. На рис. 7.3 представлена астероидальная тройка.

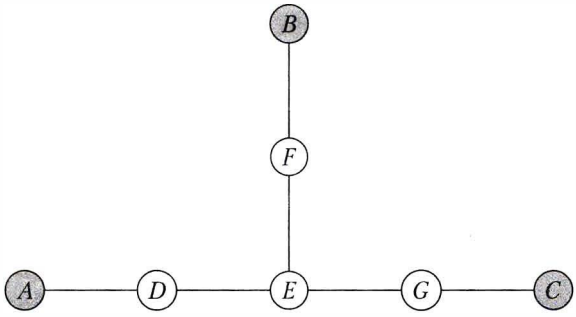


Рис. 7.3. Вершины *A*, *B* и *C* образуют астероидальную тройку; между любыми двумя из этих вершин существует путь, который не содержит никаких вершин, смежных с третьей

Вероятно, самым известным является применение интервальных графов в генетическом анализе; интервальные графы были использованы для определения того, что субэлементы гена с высокой вероятностью связаны друг с другом, образуя линейную структуру[1].

Интервальные графы обычно используются в задачах выделения ресурсов: каждый интервал представляет запрос ресурса, а раскраска графа соответствует выделению этих ресурсов. Например, если каждый интервал

[1] О топологии тонкой генетической структуры читайте работы Сеймура Бензера (Seymour Benzer), 1959.

представляет урок, а два интервала пересекаются, если уроки в какой-то момент совпадают, то хроматическое число графа — это минимальное количество необходимых классных комнат, а раскраска распределяет уроки по этим комнатам.

7.6. Графы дуг окружности

Надмножеством интервальных графов[1] являются графы дуг окружности — такие графы, которые можно изобразить в виде набора пересекающихся дуг окружности (рис. 7.4). Две вершины такого графа будут смежными тогда и только тогда, когда соответствующие дуги пересекаются. Если граф дуг окружности можно представить так, чтобы окружность не была замкнута, то такой граф также интервальный; если разорвать круг в незамкнутой части и растянуть его в линию, то получим интервальное представление.

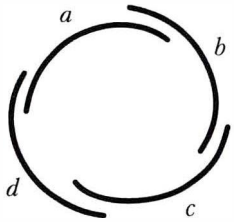


Рис. 7.4. C_4, бесхордовый цикл из четырех вершин, представленный в виде графа дуг окружности

[1] Другими словами, каждый интервальный граф также является графом пересечения дуг окружности. Это строгое отношение надмножеств, поскольку обратное неверно: многие графы пересечения дуг окружности не являются интервальными.

В то время как интервальные графы являются строгим подмножеством как совершенных графов, так и графов дуг окружности, последние классы пересекаются. Класс графов пересечения дуг окружности содержит как совершенные графы (например, интервальные), так и графы, которые не являются совершенными (нечетные бесхордовые циклы). Подобно интервальным графам, графы дуг окружности можно классифицировать по их запрещенным подграфам.

Часть III
Неграфовые алгоритмы

8 Алгоритмы сортировки

Сортировка относится к тем задачам, о которых большинству программистов никогда не приходится думать; за них это делает язык программирования или библиотеки. Как правило, подходящий алгоритм сортировки выбирается автоматически в зависимости от объема входных данных. Например, на платформе .NET массивы сортируются методом вставки, пирамидальной или быстрой сортировки, в зависимости от задачи (https://msdn.microsoft.com/en-us/library/85y6y2d3(v=vs.110).aspx).

Так зачем же создавать собственный алгоритм сортировки? Возможно, вам нужна устойчивая сортировка (то есть такая, в которой два элемента с одинаковым ранжированием будут размещаться в одинаковом порядке друг относительно друга), а предоставляемые готовые алгоритмы сортировки не являются устойчивыми. Или же вы обладаете дополнительной информацией о сортируемых данных, что может значительно сократить время выполнения алгоритма.

Наиболее эффективно изменить алгоритм сортировки можно, если:

- данные и так уже почти отсортированы;
- данные расположены в обратном или почти в обратном порядке;
- данные представляют собой конечное число дискретных значений, относительно малое по сравнению с количеством сортируемых элементов.

В этой главе рассмотрены некоторые характерные алгоритмы сортировки. Сравним их преимущества и недостатки.

Большинство алгоритмов сортировки — сортировка сравнением, когда два элемента сортируемого списка сравниваются с помощью некоторой операции, которая определяет, что один из них меньше или равен другому (размещается перед ним). Сложность алгоритмов сортировки сравнением обычно определяется количеством требуемых операций сравнения.

8.1. Малые и большие алгоритмы сортировки

Первые два вида сортировки, которые мы рассмотрим, имеют неэффективную среднюю сложность. Это означает, что на больших наборах данных они будут слишком медленными, поэтому бесполезными. Однако для небольших наборов данных асимптотически неэффективный алгоритм может работать быстрее, чем асимптотически эффективный алгоритм.

Множество из восьми элементов алгоритм со сложностью $O(n^2)$ отсортируется примерно за 64 сравнения,

а алгоритм со сложностью $O(n \lg n)$ — всего за 24 сравнения, однако каждое сравнение может потребовать меньше дополнительной работы, и в результате асимптотически медленный алгоритм на практике работает быстрее. Однако при сортировке тысячи элементов вряд ли удастся преодолеть стократное асимптотическое ускорение более эффективного алгоритма.

Затем мы рассмотрим три алгоритма со сравнением, которые выполняются за среднее время $O(n \lg n)$, и два алгоритма, в которых невозможно подсчитать количество сравнений, потому что в них значения не сравниваются между собой.

Время выполнения сортировки сравнением в худшем случае составляет $O(n \lg n)$ — это лучшее, что можно получить, и это можно продемонстрировать следующим образом. Каждое сравнение определяет относительный порядок двух значений и поэтому может рассматриваться как внутренний узел сбалансированного двоичного дерева, каждый лист которого является одним из $n!$ возможных вариантов размещения элементов. В сбалансированном двоичном дереве с k листьями любой путь от корня до листа состоит из $\lg k$ внутренних узлов или вариантов выбора (в данном случае сравнений), поэтому существует $\lg n!$ сравнений; согласно приближению Стирлинга, $\lg n!$ имеет сложность $O(n \lg n)$.[1]

---

[1] Это упрощенное объяснение; более подробное объяснение с точными цифрами вы найдете в подразделе 5.3.1 книги «Искусство программирования» Дональда Кнута (Donald Knuth).

8.2. Сортировки для малых наборов данных

8.2.1. Сортировка пузырьком

Сортировка пузырьком (bubble sort) не имеет практической пользы — ее обычно приводят в качестве примера наивного алгоритма сортировки. Однако благодаря своей простоте она дает хорошее представление о сортировке. Алгоритм состоит в том, что каждая пара соседних элементов массива сравнивается и, если эти элементы не расположены в должном порядке, их меняют местами.

Пример: сортировка пузырьком

$$\begin{matrix}3\\1\\4\\2\end{matrix} \rightarrow \begin{matrix}1\\3\\4\\2\end{matrix} \rightarrow \begin{matrix}1\\3\\4\\2\end{matrix} \rightarrow \begin{matrix}1\\3\\2\\4\end{matrix} \quad (8.1)$$

$$\begin{matrix}1\\3\\2\\4\end{matrix} \rightarrow \begin{matrix}1\\3\\2\\4\end{matrix} \rightarrow \begin{matrix}1\\2\\3\\4\end{matrix} \quad (8.2)$$

$$\begin{matrix}1\\2\\3\\4\end{matrix} \rightarrow \begin{matrix}1\\2\\3\\4\end{matrix} \quad (8.3)$$

На k-й итерации k самых больших значений гарантированно находятся в конце списка, поэтому после $n - 1$ итераций список отсортирован с общим временем

выполнения $O(n^2)$. Интересно, что это не только сложность наихудшей ситуации (по которой обычно измеряется эффективность алгоритмов), но и средняя сложность. Однако в лучшем случае время выполнения составит всего $O(n)$. Если список уже полностью или почти отсортирован, то, поскольку сортировка пузырьком способна это обнаружить (при этом не выполняется никаких перестановок), она эффективно работает для таких наборов данных. Это свойство присуще также сортировке вставками, о которой мы поговорим позже. Такая сортировка была бы более полезной во времена ленточных накопителей, когда последовательный доступ выполнялся намного быстрее, чем произвольный.

8.2.2. Сортировка вставками

Сортировка вставками (insertion sort) — еще один способ сортировки, который прост в реализации, но, в отличие от сортировки пузырьком, он действительно полезен для небольших наборов данных. Этот метод также обладает тем преимуществом, что он интуитивно понятен для большинства людей[1].

Сортировка вставками выполняется путем перебора элементов массива. Если размер массива равен единице, то такой массив явно упорядочен. Каждый элемент, добавляемый в конец массива (то есть при просмотре

[1] Вы когда-нибудь играли в бридж или преферанс? Большинство людей берут карты по одной и вставляют каждую карту на свое место среди уже отсортированных карт, которые держат в руке. Это и есть сортировка вставкой!

следующего элемента массива), либо больше, чем все остальные элементы, либо его можно переместить в правильное место массива, сместив все расположенные после него отсортированные элементы в конец на одну ячейку.

Пример: сортировка вставками

$$\begin{matrix}3\\4\\1\\2\end{matrix} \rightarrow \begin{matrix}3\\4\\1\\2\end{matrix} \rightarrow \begin{matrix}1\\3\\4\\2\end{matrix} \rightarrow \begin{matrix}1\\2\\3\\4\end{matrix} \qquad (8.4)$$

Подобно сортировке пузырьком, наилучшая сложность сортировки вставкой (когда массив уже отсортирован) равна $O(n)$, а наихудшая сложность (когда массив отсортирован в обратном порядке) равна $O(n^2)$. На практике алгоритм быстро работает для небольших массивов и обычно применяется для сортировки небольших участков массивов в рекурсивных алгоритмах, таких как быстрая сортировка и сортировка слиянием[1].

У сортировки вставками есть еще несколько приятных особенностей. В частности, она стабильна, выполняется на месте и *онлайн*. Онлайн-алгоритм — это такой алгоритм, в котором значения могут обрабатываться по мере их поступления; они могут быть недоступны все сразу с самого начала.

---

[1] После того как однажды мне пришлось проверить 160 ученических работ и выставить оценки, я привык сортировать вставкой стопки из 8–10 работ, объединять эти стопки методом сортировки слиянием, а затем записывать оценки.

8.3. Сортировка больших наборов данных

8.3.1. Пирамидальная сортировка

Подобно предыдущим алгоритмам сортировки, пирамидальная сортировка делит входные данные на отсортированную и несортированную части и поэтапно перемещает элементы в отсортированную часть. В отличие от предыдущих методов сортировки пирамидальная сортировка требует предварительной обработки данных. Сначала из данных строится куча, а затем самый большой элемент кучи многократно извлекается и вставляется в массив. Построение кучи требует $O(n)$ операций[1]. Извлечение из кучи максимального элемента и восстановление кучи требуют $O(\lg n)$ операций. Поскольку нужно извлечь все n элементов, наилучшая и наихудшая производительность составляет $O(n \lg n)$[2]. Обратите внимание: хотя асимптотическая сложность для наилучшего и наихудшего случаев одинакова, на практике наилучшее время выполнения будет примерно в два раза быстрее[3].

8.3.2. Сортировка слиянием

При сортировке слиянием (merge sort) список сортируется рекурсивно: массив делится на несколько мас-

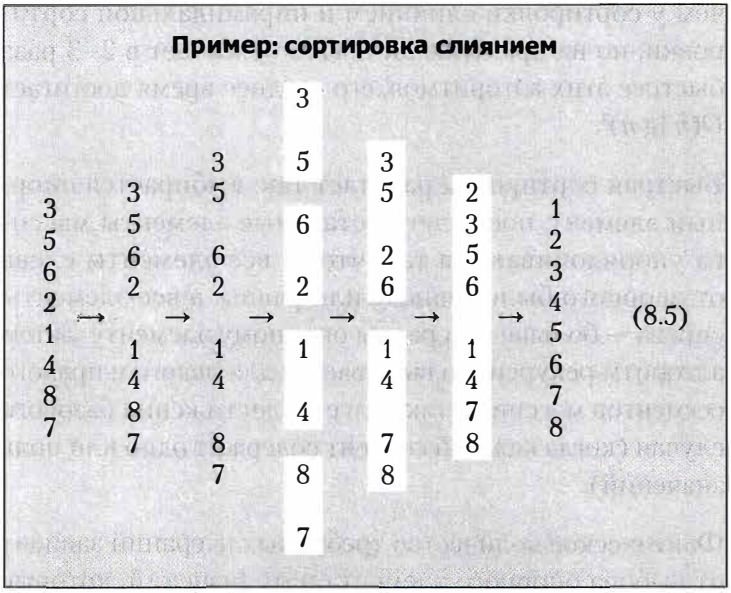
---

[1] $O(n \lg n)$ для некоторых вариаций.

[2] *Schaffer R., Sedgewick R.* The Analysis of Heapsort // Journal of Algorithms. Vol. 15. Issue 1, July, 1993.

[3] *Bolloba's B., Fenner T. I., Frieze A. M.* On the Best Case of Heapsort // Journal of Algorithms. Vol. 20. Issue 2, March, 1996.

сивов меньшего размера, которые сортируются, а затем объединяются. При чистом слиянии можно продолжать деление до тех пор, пока каждый набор не будет содержать только одно значение. Затем множества объединяются; имея два отсортированных набора размером k, мы можем создать полностью упорядоченный объединенный набор, при этом число сравнений составит от k до $2k - 1$.



Это дает $O(\lg n)$ комбинированных шагов, каждый из которых занимает $O(n)$ времени, так что общее время выполнения алгоритма составляет $O(n \lg n)$.

На практике вместо того, чтобы делить исходный набор данных на отдельные элементы, обычно прекращают деление, когда наборы достаточно малы, чтобы

использовать алгоритм с наилучшим временем выполнения для небольших наборов данных (например, сортировку вставками).

8.3.3. Быстрая сортировка

Алгоритм быстрой сортировки (quick sort) интересен тем, что его наихудшее время выполнения $O(n^2)$ хуже, чем у сортировки слиянием и пирамидальной сортировки, но на практике он обычно работает в 2–3 раза быстрее этих алгоритмов, его среднее время достигает $O(n \lg n)$[1].

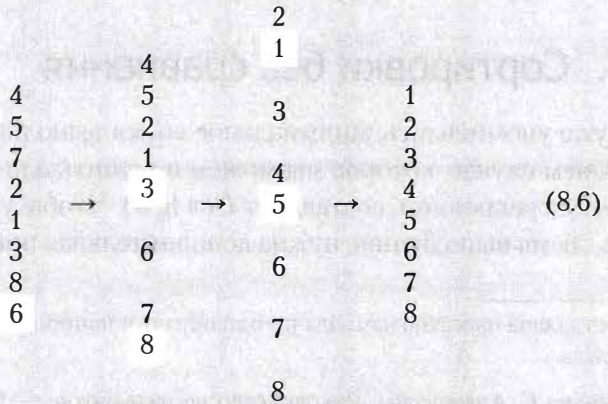
Быстрая сортировка работает так: выбирается опорный элемент, после чего остальные элементы массива упорядочиваются так, чтобы все элементы слева от опорного были меньше или равны, а все элементы справа — больше или равны опорному элементу. Затем алгоритм рекурсивно вызывается для левого и правого сегментов массива и так далее до достижения базового случая (когда каждый сегмент содержит одно или ноль значений).

Фактическое количество требуемых итераций зависит от выбора опорного элемента и от значений, которые необходимо отсортировать. Например, метод Ломуто состоит в том, чтобы просто выбрать в качестве опорного последний элемент массива (или сегмента массива); это легко реализовать, но тогда, если массив уже отсор-

[1] *Скиена С.* Алгоритмы. Руководство по разработке. — СПб.: BHV, 2017.

тирован, время выполнения составит $O(n^2)$, потому что на каждой итерации сегмент будет уменьшаться только на один элемент. Чтобы этого избежать, можно выбрать в качестве опорного элемент, находящийся в середине массива, случайный элемент или медиану из трех случайных элементов. В идеале опорный элемент будет точной медианой сортируемых данных, а каждый из двух новых разделов будет содержать половину оставшихся данных. При известном (неслучайном) методе выбора опорного элемента злоумышленник может увеличить время выполнения до $O(n^2)$, намеренно размещая данные таким образом, чтобы опорный элемент каждый раз находился с одной и той же стороны от элемента, с которым он сравнивается.

Пример: быстрая сортировка



(8.6)

По аналогичным причинам быстрая сортировка также будет плохо работать с массивами, содержащими

большое количество повторяющихся значений[1], но это можно исправить, используя сортировку не на два, а на три раздела (меньше, равно, больше). В этом случае средний раздел не нуждается в дальнейшей сортировке, поэтому при сортировке массивов с большим количеством дубликатов можно получить более быстрые результаты (а если все элементы одинаковы, то сортировка займет линейное время).

Преимущества быстрой сортировки, по сравнению с сортировкой слиянием, включают в себя сортировку на месте и более простой внутренний цикл, благодаря чему в большинстве случаев алгоритм работает быстрее[2]. Недостатками этого метода являются наихудшее время выполнения и отсутствие стабильности[3]; переупорядочение на месте также не работает, если данные представлены не в виде массива, а в виде связного списка.

8.4. Сортировки без сравнения

Как уже упоминалось, минимальное время выполнения в худшем случае, которое мы можем получить для сортировки сравнением, составляет $O(n \lg n)$. Чтобы улучшить время выполнения, нужна дополнительная инфор-

---

1 Эта задача известна как задача голландского национального флага.

2 *Скиена С.* Алгоритмы. Руководство по разработке. — СПб.: BHV, 2017.

3 Причина нестабильности быстрой сортировки, как правило, состоит в том, что во время разбиения могут переставляться элементы с одинаковым ключом.

мация (метаданные) о сортируемых данных. Иногда это позволяет улучшить время сортировки до $O(n)$.

В частности, даже если количество сортируемых элементов достаточно велико, может существовать ограниченное количество ключевых значений, по которым эти элементы должны быть отсортированы. На этом принципе основаны следующие методы сортировки.

8.4.1. Сортировка подсчетом

Предположим, что каждый из n элементов сортируемых данных имеет ключ, который является положительным целым числом со значением не более k. Тогда можно создать массив длиной k и перебрать в цикле все элементы, которые должны быть отсортированы, используя этот массив для отслеживания количества вхождений каждого ключа (выражаясь более формально, мы строим частотную гистограмму для значений ключа).

Теперь у нас есть массив значений, соответствующих количеству появлений каждого ключа. Затем остается еще раз пройти по массиву и заменить каждое значение на количество ключей с меньшими значениями. После этого каждая ячейка массива будет содержать правильную позицию для первого элемента с этим ключом.

Наконец, остается переместить каждый элемент исходных данных в позицию, которая определяется значением, записанным в соответствующей ячейке массива ключей, и увеличить это значение на единицу (чтобы следующий элемент с таким ключом, если он есть, разместить в следующей ячейке выходного массива).

Пример: сортировка подсчетом

Рассмотрим следующий массив, в котором цифра является ключом для сортировки, а буква — прикрепленным значением: [1a, 3b, 2a, 4c, 1c, 4b, 2a, 2c, 2b, 1b, 3c].

Пройдя по массиву, мы обнаружили три единицы, четыре двойки, две тройки и две четверки и получили гистограмму [3, 4, 2, 2].

Это говорит о том, что первый элемент с каждым ключом должен появляться в следующих позициях: [0, 3, 7, 9].

Теперь снова пройдем по исходному массиву. Первый элемент равен 1 и перейдет в ячейку 0. Второй элемент имеет ключ 3, он пойдет в ячейку 3 и т. д. Каждый раз, помещая элемент в выходной массив, мы увеличиваем соответствующее значение в массиве ключей на единицу. Например, после размещения первых семи элементов массив ключей принял вид [2, 5, 8, 11], и мы знаем, что следующий элемент, 2c, попадет в ячейку 5 выходного массива.

Весь процесс требует двух циклов для входного массива (один для заполнения массива ключей и один — для перемещения элементов в выходной массив) и одного цикла для массива ключей, чтобы просуммировать значения и определить конечные позиции ключей. Кроме того, должны быть инициализированы массив ключей (размером k) и выходной массив (размером n). Тогда общее время выполнения, как и занимаемое место в памяти, составит $O(n + k)$. Если k мало по сравнению с n (например, для большого количества элементов, которые ранжируются по шкале от 1 до 100), то это $O(n)$. Это стабильный метод сортировки.

8.4.2. Поразрядная сортировка

Поразрядная сортировка — это более старый метод сортировки[1], который до сих пор остается полезным; в зависимости от допущений, он может быть более эффективным, чем лучшие варианты сортировки сравнением. Его сложность составляет $O(wn)$ для ключей с длиной слова w; если w постоянна, это сводится к $O(n)$. Естественно, это требует дублирования ключей; если каждый сортируемый элемент имеет уникальный ключ, то длина ключа должна быть не менее $\lg n$[2]. Существует несколько вариантов поразрядной сортировки; здесь мы рассмотрим сортировку по младшему разряду (least significant digit, LSD).

Рассмотрим алфавит, из которого взят ключ, и создадим по одной ячейке для каждой буквы этого алфавита. Например, если ключ является десятичным числом, то у нас будет десять ячеек, помеченных цифрами от 0 до 9. Каждый элемент списка мы поместим в ячейку, соответствующую младшему разряду ключа этого элемента. Например, элемент с ключом 378 попадает в ячейку номер 8. Другими словами, мы сгруппировали ключи по младшему разряду, но сохранили их относительный порядок (чтобы это была стабильная сортировка).

---

[1] Метод использовался Германом Холлеритом (Herman Hollerith) для счетно-аналитических машин в 1887 году; одна из компаний, основанных Холлеритом, в итоге стала IBM.

[2] Для основания b и ключа длиной w количество различных ключей равно b^w. Если w меньше $\lg_b n$, то количество возможных ключей меньше количества сортируемых элементов.

Затем мы снова сгруппируем ключи, но теперь по второму с конца разряду; теперь элемент с ключом 378 окажется в ячейке номер 7. И так далее. После того как будут перебраны все цифры в ключе, список отсортирован. Один из способов реализовать это — представить каждую ячейку в виде очереди; после каждого прохода элементы можно возвращать обратно в список.

Почему это работает? Рассмотрим два ключа на последнем этапе процесса. Если их старшие разряды различны, то ключ, который должен стоять первым при сортировке, попадет в предшествующую ячейку, поэтому они будут правильно расположены друг относительно друга. Если старшие разряды ключей одинаковы, то они попадут в одну ячейку в том же порядке, в котором были отсортированы для предыдущего разряда, что по индукции также будет правильным, если эти цифры отличаются, и т. д. Если все цифры одинаковы, то порядок не имеет значения (но элементы расположатся в том же относительном порядке, в котором были представлены в исходном списке).

Каждый элемент помещается в ячейку (за постоянное время) один раз для каждого символа ключа, что дает максимальное время выполнения $O(wn)$.

Часть IV
Методы решения задач

9 А если в лоб?

Computer Science можно рассматривать как способ найти решения задач с помощью компьютеров. Затем работа программиста сводится к трем действиям: определить, какие задачи может решить компьютер, объяснить компьютеру, как их решать, и спрогнозировать, сколько времени займет этот процесс.

Выбор метода решения задачи зависит от того, какие атрибуты решения наиболее важны. У каждого алгоритма есть своя стоимость выполнения, которую можно выразить как сложность по времени и пространству. Сложность по времени — это время выполнения алгоритма в зависимости от объема входных данных. Сложность по пространству — требуемый объем памяти (опять же относительно объема входных данных). *Вычислительная сложность* алгоритма — это количество ресурсов (времени и пространства), необходимых для его выполнения; вычислительная сложность задачи — это минимальная сложность алгоритма, который мог бы ее решить.

Однако ресурсы, необходимые для выполнения программы, — это не единственные расходы (хотя в научных статьях, как правило, упоминают только их). Приходит-

ся также учитывать сложность написания и отладки программы. Иногда лучше реализовать менее эффективный алгоритм, пожертвовав некоторой скоростью в обмен на время программиста (потому что реализация более быстрого алгоритма займет значительно больше времени) или точностью (потому что в более быстром алгоритме выше вероятность программных ошибок).

Практическое применение

Однажды я проводил обзор кода (code review) джуниора. Рецензент предложил внести некое изменение, чтобы сделать код более эффективным.

Я согласился, что предложенный код будет более эффективным, но наложил вето на изменение, потому что из-за этого код стал бы более сложным. В том случае код не относился к чувствительной ко времени части приложения (долю секунды, которую могло бы сэкономить изменение, никто бы не заметил), но дополнительное усложнение повысило бы вероятность того, что в итоге мы бы получили ошибку — либо сразу, либо в будущем, при очередном обновлении кода.

В конце шкалы «эффективность — сложность» находится метод решения задач в лоб, или метод грубой силы. Он очень прост: перебрать все возможные решения и проверить каждое, пока одно из них не окажется правильным. Этот метод годится, если задачу нужно решить один раз, и дополнительное время, необходимое для поиска эффективного решения, превысит экономию времени при выполнении этого решения. Конечно, иногда мы применяем грубую силу просто потому, что не знаем лучшего решения!

Например, если на сейфе установлен пульт из десяти кнопок и вы знаете, что комбинация представляет собой четырехзначное число и больше никакой информации нет (и количество попыток не ограничено), можно найти комбинацию, просто перебрав все возможные числа от 0000 до 9999. Это может занять очень много времени — в среднем 5000 попыток, — но сама процедура очень проста. В цифровом мире это означает, что, если на сайте не используются надежные пароли, а учетная запись не блокируется после некоторого количества неправильных попыток, то с помощью известного имени пользователя можно перебрать все возможные пароли, пока один из них не подойдет.

Решение методом грубой силы, как правило, очень просто реализовать. В то же время его очень сложно реализовать неправильно, что делает его полезным, если размер задачи невелик, а точность крайне желательна.

10 Динамическое программирование

10.1. Задача недостающих полей

Предположим, у нас есть шахматная доска размером $n \times n$, на которой недостает нескольких полей. Как найти наибольший участок доски размером $k \times k$ без недостающих полей?[1]

Если вы раньше не сталкивались с такой задачей, потратьте несколько минут на то, чтобы написать решение и определить время выполнения вашего алгоритма.

Столкнувшись с этой задачей, я рассуждал следующим образом. Каждое поле шахматной доски может принадлежать ко многим наибольшим участкам, но только в одном из них оно может быть верхним левым углом. Если я помечу каждое поле размером наибольшего неповрежденного участка, верхним левым углом которого

[1] Мне задали эту задачу несколько лет назад на собеседовании в известной компании по разработке программного обеспечения; быстро предложив эффективное решение, я вышел на следующий уровень собеседований. С тех пор прошло достаточно времени, чтобы я мог спокойно изложить ее здесь.

он является, то поле с наибольшей такой меткой будет соответствовать искомому участку.

Предположим, что доска представлена в виде матрицы $n \times n$, в которой каждая ячейка содержит 1, если соответствующее поле присутствует, и 0, если оно пропущено. Если текущее значение ячейки равно 0, то соответствующее поле отсутствует и не может быть частью непрерывного участка, поэтому его не нужно изменять. Если же значение равно 1, то мы можем заменить его числом, которое на единицу больше, чем минимальное значение из трех ячеек, расположенных ниже и справа.

Мы изменяем каждую ячейку матрицы один раз, включая проверку, равно ли значение ячейки нулю, проверку значений еще максимум трех ячеек и запись нового значения ячейки. Каждая из этих операций занимает время $O(1)$, поэтому время, необходимое для обработки всей шахматной доски, составляет $O(n^2)$.

Обратите внимание, что это линейное, а не квадратичное время выполнения алгоритма — на шахматной доске n[1] полей (некоторые из них отсутствуют), поэтому общее время, затрачиваемое алгоритмом, пропорционально количеству полей. Если более точно обозначить размер шахматной доски как $\sqrt{n} \times \sqrt{n}$, то получим n полей и общее время выполнения $O(n)$.

---

[1] Мы считаем, что n — это общее количество полей, и придерживаемся обычного соглашения о том, что n — это объем входных данных.

10.2. Работа с перекрывающимися подзадачами

Подход, использованный в этом разделе, называется динамическим программированием. Он применяется, когда задачу можно разделить на несколько подзадач, решение каждой из которых будет использовано несколько раз. Этот подход отличается от принципа «разделяй и властвуй», когда задачу разделяют на подзадачи, которые решаются независимо друг от друга. В задаче с шахматной доской каждая подзадача зависела от решений трех других задач, а решения всех подзадач сохранялись для дальнейшего использования.

Динамическое программирование обычно выполняется путем построения таблиц, как показано выше. Это означает решение задачи методом «снизу вверх», когда мы начинаем с решения наименьших подзадач и продвигаемся вверх до тех пор, пока не придем к ответу. Другой метод — это мемоизация, при которой мы идем сверху вниз, решая подзадачи по мере необходимости и кэшируя результаты для повторного использования[1]. Построение таблиц — предпочтительный вариант, когда нужно решить каждую подзадачу (в моем примере с шахматной доской нужно было найти наибольший

[1] Некоторые считают, что динамическое программирование ограничивается построением таблиц, а мемоизация к нему не относится. Какую бы семантику вы ни предпочли, сами методы остаются такими, как описано здесь.

неповрежденный участок для каждого поля доски), поскольку затраты у этого метода меньше, чем при мемоизации. Если некоторые подзадачи из области решения не обязательно решать, то мемоизация позволяет решать только те подзадачи, которые действительно необходимы.

Алгоритм 8. В каждой ячейке может быть записано число больше 1, только если начальные значения этой ячейки, а также ячеек справа, снизу и снизу справа от нее равны 1

Входные данные: матрица M, каждая ячейка которой содержит либо 1, если соответствующее поле присутствует на доске, либо 0, если его нет

Выходные данные: матрица M, в каждой ячейке которой записан размер самого большого неповрежденного участка доски, для которого соответствующее поле находится в верхнем левом углу

```
begin
    for i = n – 2 to 0 do
        for j = n – 2 to 0 do
            if M[i][j] == 0 then
                продолжить;
            else
                M[i][j] = 1 + min(M[i + 1][j],
                M[i][j + 1], M[i + 1][j + 1]);
            end
        end
    end
end
```

Ключевой момент

Там, где метод «разделяй и властвуй» подразумевает разделение задачи на несколько *независимых* подзадач, динамическое программирование подразумевает разделение задачи на несколько *перекрывающихся* подзадач. Решение каждой подзадачи кэшируется для последующего повторного использования.

10.3. Динамическое программирование и кратчайшие пути

Рассмотрим задачу поиска кратчайшего пути: для заданного графа со взвешенными ребрами нужно найти такой путь из одного узла в другой, который имеет наименьшую стоимость.

Определение

Граф со взвешенными ребрами — это граф, в котором каждое ребро имеет свой вес (стоимость). Стоимость пути из одного узла в другой определяется суммой стоимости всех пройденных ребер.

Предположим, что мы нашли путь между узлами s и t, который содержит третий узел v. Тогда путь из s в t должен содержать кратчайший путь из s в v, поскольку в противном случае мы могли бы заменить этот участок более коротким путем и уменьшить длину кратчайшего пути из s в t, что противоречит начальному условию[1].

[1] Это принцип оптимальности Беллмана.

Ключевой момент

Динамическое программирование (и жадные алгоритмы) полезно для решения задач, имеющих оптимальную подструктуру. Это означает, что оптимальное решение задачи может быть эффективно построено из оптимальных решений ее подзадач. Если самый короткий путь из Мэдисона в Денвер проходит через Омаху, то этот маршрут также должен содержать самый короткий путь из Мэдисона в Омаху и из Омахи в Денвер.

Если задача имеет и оптимальную подструктуру, и перекрывающиеся подзадачи, то она становится кандидатом на решение методом динамического программирования.

Задачи поиска кратчайшего пути представляют собой яркие примеры динамического программирования, поскольку оптимальное свойство подструктуры интуитивно понятно — очевидно, что самый быстрый способ перехода из точки *A* в точку *C* через точку *B* — это также самый быстрый способ перехода из точки *A* в точку *B* и из точки *B* в точку *C*. В число алгоритмов, основанных на этом принципе, входит метод Беллмана — Форда, который находит кратчайший путь из исходной точки в любую вершину графа (или от любой вершины графа до конечной точки) и метод Флойда — Уоршелла — с его помощью вычисляется кратчайший путь между каждой парой вершин графа. В обоих случаях идея состоит в том, чтобы начать с небольшого подмножества узлов, близких к интересующим нас узлам, и постепенно расширять это множество, используя уже найденные узлы для вычисления новых расстояний.

Алгоритм 9. В начале алгоритма в ячейке M[s][t] хранится длина ребра между вершинами s и t, если оно существует. Если сумма расстояний от s до другой вершины i и от i до t меньше расстояния от s до t, то мы заменяем M[s][t] новым значением

Алгоритм: Флойда — Уоршелла
Входные данные: матрица M, в каждой ячейке которой записана длина ребра между соответствующими вершинами, с нулями по диагонали и ∞, если такого ребра не существует
Выходные данные: матрица M, в каждой ячейке которой записана длина кратчайшего пути между соответствующими вершинами

```
begin
    for i = 1 to n do
        foreach s, t do
            M[s][t] = min(M[s][t], (M[s][i] + M[i][t]);
        end
    end
end
```

10.4. Примеры практического применения

10.4.1. Git merge

Еще одна задача, на примере которой обычно демонстрируют возможности динамического программирования, — поиск наибольшей общей подпоследовательности (Longest Common Subsequence). Задача состоит в том, чтобы для двух заданных строк A и B найти самую

длинную последовательность, которая встречается в обеих строках с сохранением последовательности символов. Символы в строках не обязательно должны стоять подряд; например, если $A = \{acdbef\}$ и $B = \{babdef\}$, то $\{adef\}$ будет их общей подпоследовательностью.

При слиянии изменений в Git (merge) выполняется поиск наибольшей общей подпоследовательности для master и рабочей веток. Символы, присутствующие в master, но отсутствующие в наибольшей общей подпоследовательности, удаляются; символы, которые есть в рабочей ветке, но отсутствуют в этой подпоследовательности, добавляются в master.

10.4.2. LaTeX

Систему подготовки документов LaTeX часто используют для создания технических документов. Одна из задач системы набора текста — выравнивание текста одновременно по правому и левому краю; для этого интервалы между словами и символами растягиваются или сжимаются таким образом, чтобы все строки имели одинаковую длину. Другой способ выровнять текст состоит в переносе последнего слова, так что часть слова оказывается в следующей строке. LaTeX[1] пытается найти оптимальные точки разрыва строки, чтобы текст выглядел красиво. Если это сделать не удастся, то несколько строк подряд будут заканчиваться переносом слова или же расстояние между словами в разных строках

[1] С технической точки зрения практически всю работу выполняет система ввода текста TeX; LaTeX построена на основе TeX. Здесь я использую LaTeX из соображений простоты.

будет различаться. В LaTeX существует набор правил для оценки «неудачности» выравнивания. Программа стремится найти «наименее плохой» вариант.

Если в абзаце есть n возможных точек разрыва строки, то существует 2^n возможных вариантов разбиения текста. «Неудачность» выбора для каждой точки разрыва зависит от того, какие точки разрыва были выбраны до нее. Следовательно, у нас снова есть перекрывающиеся подзадачи. Использование методов динамического программирования сокращает время выполнения до $O(n^2)$, которое может быть улучшено с помощью дополнительных методов[1].

[1] Подробнее см. статью: *Donald E. Knuth, Michael F.* Plass Breaking paragraphs into lines // Software: Practice and Experience. Vol. 11. Issue 11, 1981.

11 Жадные алгоритмы

Жадный подход к решению задачи состоит в том, чтобы каждый раз, когда нужно принять решение, выбирать тот вариант, который является локально оптимальным. Другими словами, выбирается тот вариант, который дает лучшее решение текущей подзадачи, даже если это не будет лучшим решением всей задачи. В задаче о коммивояжере из раздела 1.6 это означало, что нужно всегда ехать в ближайший город, который еще не посещался (то есть выбрать ребро с наименьшим весом, которое приведет к еще не посещавшейся вершине). Такой метод работает быстро (нужно только перебрать все ребра, выходящие из текущей вершины, и выбрать то, которое имеет наименьший вес), но не гарантирует, что будет найдено наилучшее решение всей задачи.

Допустим, мы ходим по некоторому участку земли и хотим найти самую высокую точку. Жадный алгоритм предложил бы постоянно идти в гору, пока это возможно. Когда все допустимые пути будут направлены вниз, это будет означать, что мы нашли локальный мак-

симум — любая точка, в которую мы сможем перейти непосредственно из текущего положения, окажется ниже того места, в котором мы находимся в данный момент. Это не означает, что мы действительно нашли самое высокое доступное место, — мы достигли лишь самой высокой точки в локальной области (места, где вы сейчас находитесь, и соседних мест, с которыми вы его сравниваете).

Существуют задачи, для которых жадные алгоритмы дают оптимальное решение; одним из примеров является алгоритм поиска кратчайших путей Дейкстры. Как и в случае динамического программирования, жадные алгоритмы лучше всего работают в тех случаях, когда задача имеет оптимальную подструктуру. Разница между этими подходами заключается в том, что динамическое программирование позволяет гарантированно найти решение задачи для оптимальной подструктуры, поскольку оно учитывает все возможные подзадачи и объединяет их для достижения оптимального решения, тогда как жадный алгоритм просто выбирает ту подзадачу, которая лучше всего выглядит в данный момент.

Рассмотрим такую задачу: добраться домой с работы в час пик. Жадный подход заключается в том, чтобы выбрать любой маршрут домой с наименьшей загруженностью возле работы. В этом случае вы будете двигаться быстрее, но рискуете столкнуться с большим трафиком ближе к дому. Алгоритм динамического программирования будет учитывать весь трафик и выберет маршрут с наименьшими затратами в среднем, даже

если начальный участок этого пути будет сильнее загружен.

Жадные алгоритмы, как правило, являются быстрыми. Поэтому их предпочтительно выбирать, когда они гарантированно найдут либо оптимальное решение задачи, либо хотя бы одно достаточно хорошее решение.

Часть V
Теория сложности вычислений

12 Что такое теория сложности

В главе 1 вы узнали, как оценить время выполнения алгоритма, чтобы можно было выбрать наилучший алгоритм для решения данной задачи (или определить, существует ли достаточно хороший алгоритм для ее решения). В той главе не имело особого значения, какой язык программирования использовать. Там описаны алгоритмы на псевдокоде, однако использование C#, Java или Python дало бы такую же асимптотическую скорость выполнения.

Это не означает, что программы, написанные на любом языке, всегда будут одинаково эффективны, однако все они (теоретически) работают на одном и том же оборудовании и используют одни и те же структуры данных, поэтому мы ожидаем увидеть одинаковое увеличение времени выполнения по мере роста объема задачи. Основное предположение состоит в том, что существует некий теоретический компьютер, во всем подобный тем, что применяются в настоящее время, за исключением бесконечного объема памяти (что, увы, редко встречается в реальных компьютерах).

Одно из применений теории сложности состоит в определении того, что может и чего не может сделать компьютер. Иногда удается значительно сократить время, необходимое для решения задачи, либо используя более эффективный алгоритм, либо вводя дополнительные ресурсы (например, процессоры). Более сложные по своей природе задачи невозможно решить без значительных ресурсов даже при наличии оптимального алгоритма (NP-полные задачи считаются сложными). Иногда задача действительно не решается на определенном типе компьютеров, даже при наличии неограниченного количества ресурсов.

При смене вычислительной модели задача, которая прежде была очень сложной, может стать тривиальной, а тривиальная задача, наоборот, — неразрешимой. Например, существуют квантовые алгоритмы (алгоритмы, работающие на квантовом компьютере), которые выполняются экспоненциально быстрее, чем самые известные классические алгоритмы (те, что выполняются на компьютерах, используемых сегодня повсеместно, не только в исследовательских лабораториях), для одной и той же задачи.

Дополнительная информация

Квантовый компьютер — это сравнительно новая модель вычислений, основанная на квантовой механике. Там, где в классических компьютерах используются биты, которые могут быть равны 0 или 1, в квантовых компьютерах применяются *кубиты*. Последние существуют

в суперпозиции состояний: каждый бит равен 0 и 1 одновременно. Если байт выражает одно значение от 0 до 255, то квантовый байт выражает все 256 значений одновременно.

Если прочитать кубиты, то они свернутся в одно состояние. Другими словами, вместо того, чтобы находиться во всех возможных состояниях сразу, они имеют ровно одно значение. При этом нет гарантии, что это значение будет правильным; алгоритм просто манипулирует кубитами так, что при их измерении они с большой вероятностью придут в правильное состояние.

И наоборот, существуют модели вычислений, неспособные решить задачи, которые тривиально решаются с помощью современных компьютеров. Эти модели имеют такие ограничения, как способ доступа к памяти. Продемонстрировав, что конкретная задача может быть решена с помощью данной модели, мы устанавливаем верхнюю границу вычислительной мощности, необходимой для ее решения, а также получаем доступ к инструментам, созданным для работы с этой моделью.

13 Языки и конечные автоматы

13.1. Формальные языки

Любой человеческий язык, например английский, представляет собой набор букв (в письменной речи) или звуков (в устной речи), а также правил, описывающих, как комбинировать эти буквы или звуки в слова и предложения. Точно так же в математике и Computer Science язык представляет собой набор символов и правила их комбинации.

Языки классифицируются в зависимости от того, насколько мощным должен быть компьютер для их распознавания. Под распознаванием в данном случае понимается способность компьютера точно определить по заданной строке, принадлежит ли она некоторому языку. Такие свойства, как тип доступной памяти, определяют мощность компьютера и, следовательно, сложность языков, которые он способен распознавать.

Формально язык L с алфавитом Σ — это (возможно, бесконечное) множество всех допустимых слов, которые могут быть составлены из символов этого алфавита. Например, мы можем определить алфавит $\Sigma = \{a\}$ для языка $L = a^{2n}$, другими словами, все строки четной

длины, состоящие только из буквы *a*. В этот язык будут входить строки λ (пустая строка — ноль, а ноль — это четное число!), *aa*, *aaaa* и т. д. В языке может быть конечное количество слов, бесконечное количество слов или даже не быть слов вообще ($L = \{\varnothing\}$[1]).

Класс языков может определяться по типу машины, которая способна его распознать, по типу грамматики, которая генерирует этот язык, а также в терминах теории множеств. Мы рассмотрим все эти способы для каждого класса языков.

13.2. Регулярные языки

Регулярные языки — это такие языки, которые могут быть поняты *конечными автоматами*. Конечный автомат (Finite State Machine), — это машина, которая имеет одно начальное состояние, одно или несколько допустимых состояний и переходы между этими состояниями. Чтобы определить, есть ли в языке данная строка, вы начинаете со стартового состояния и следуете переходу для каждой буквы этой строки. Если состояние, достигнутое после последней буквы строки, является допустимым состоянием, то строка принадлежит этому языку.

В конечном автомате, представленном на рис. 13.1, работа начинается с состояния *S*, которое также является допустимым (поскольку нам еще не встретилось никаких букв, а ноль — четное число). Встретив одну

[1] Это пустое множество — не то же самое, что множество, содержащее только пустую строку ($\{\lambda\}$).

букву a, мы переходим в состояние q_1, а встретив вторую букву a — возвращаемся к S. И так далее: продолжаем переходить по ребрам, пока не прочитаем всю строку.

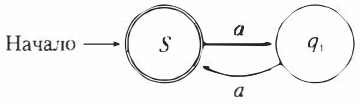


Рис. 13.1. Конечный автомат, который принимает любую строку четной длины, составленную из алфавита $\Sigma = \{a\}$. Двойной кружок указывает на то, что S является допустимым состоянием

Существует два вида конечных автоматов: детерминированные и недетерминированные. В *детерминированных конечных автоматах*, ДКА (Deterministic Finite Automata), если алфавит содержит более одной буквы (как в большинстве языков), каждое состояние автомата должно содержать ребро для каждой буквы алфавита. На рис. 13.2 показан детерминированный конечный автомат для языка, который построен на алфавите $\Sigma = \{a, b\}$ и состоит из любого количества букв a, после которых стоит ровно одна буква b.

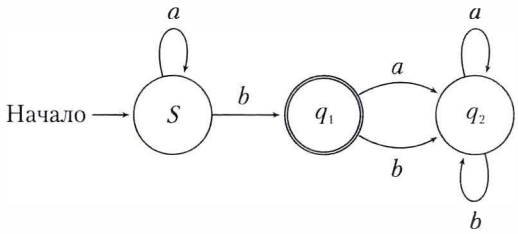


Рис. 13.2. Детерминированный конечный автомат, принимающий строки, состоящие из любого (включая нулевое) количества букв a, после которых стоит одна буква b

Состояние q_2 на рис. 13.2 — то, что называют мертвым состоянием: это состояние не является допустимым и, поскольку, какую бы букву мы ни добавили, мы все равно вернемся в эту же точку, если мы сюда попали, то уже не сможем достичь какого-либо допустимого состояния. В данном случае строка, которая содержит любые символы после буквы *b*, отсутствует в данном языке. Обратите внимание, что, поскольку автомат является детерминированным, переход на каждом шаге однозначно определяется вводимым символом и текущим состоянием.

Альтернативой мертвому состоянию является использование *недетерминированного конечного автомата*, НКА (Nondeterministic Finite Automaton). Это конечный автомат, в котором у каждого состояния для любой буквы может быть ноль, один или несколько переходов. Поскольку в недетерминированном конечном автомате каждое состояние не должно иметь переход для каждой буквы, мертвое состояние не является обязательным. Если для следующей буквы в строке нет перехода, то данная строка отсутствует в этом языке.

С точки зрения описываемых языков автоматы НКА эквивалентны ДКА. Каждый ДКА также является НКА (ДКА должен соблюдать ограничение по одному переходу на одну букву, в отличие от НКА), а любой НКА может быть преобразован в соответствующий ДКА. На рис. 13.3 представлен НКА, который описывает тот же язык, что и ДКА на рис. 13.2. Как правило, предпочтительно использовать НКА вместо ДКА, поскольку НКА часто требуют гораздо меньше переходов.

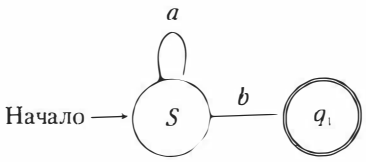


Рис. 13.3. Недетерминированный конечный автомат, который принимает строки, состоящие из любого (включая нулевое) количества букв *a*, после которых стоит ровно одна буква *b*

Предположим, мы хотим распознать более сложный язык: этот язык состоит из одной или нескольких букв *a*, после которых стоит ноль или более букв *c*, ноль или более букв *b*, а затем ровно одна буква *c*. Мы могли бы распознать этот язык с помощью недетерминированного конечного автомата на рис. 13.4. Обратите внимание: в этом автомате используется лямбда-переход[1], чтобы гарантировать, что *c*-цикл не будет повторяться после *b*-цикла.

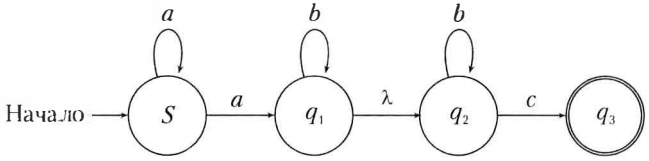


Рис. 13.4. НКА для языка *a* + *c* * *b* * *c*

Какие типы языков не распознаются конечными автоматами? Обратите внимание, что у этих конечных автоматов нет отдельной памяти, что ограничивает длину

[1] Лямбда-переход позволяет переходить из одного состояния в другое, не считывая входные данные.

сравнений числом состояний автомата. Например, мы хотим построить автомат, который бы распознавал язык, состоящий из любого количества парных скобок, причем после левой скобки всегда следует соответствующая правая скобка. Такой автомат мог бы выглядеть так, как показано на рис. 13.5.

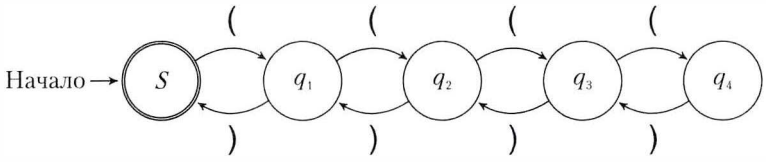


Рис. 13.5. Недетерминированный конечный автомат, который принимает наборы парных скобок и поддерживает до четырех одновременно открытых пар

Такой автомат будет делать именно то, что мы хотим, если число одновременно открытых пар скобок не превысит четырех (четыре левые скобки без соответствующих правых скобок). Если этот предел будет превышен, автомат не сможет распознать строку. Мы можем увеличить автомат, добавив еще несколько состояний, но число состояний всегда будет конечным. В случае $n + 1$ состояний можно отслеживать до n переходов, то есть конечный автомат не сможет распознать любую строку языка, в котором допускаются сравнения любой длины. Позже вы узнаете о методе, позволяющем показать, что язык, который требует такого сравнения, не может быть распознан никаким конечным автоматом, следовательно, такой язык не является регулярным.

13.2.1. Регулярные грамматики

Грамматика — это набор правил для генерации языка. Там, где автоматы для языка позволяют проверить, принадлежит ли данная строка языку, грамматика позволяет генерировать любую строку, которая является частью языка. Грамматика состоит из переменных (обозначаемых заглавными буквами), терминалов (которые являются символами алфавита) и переходов, преобразующих переменные в символы (которые могут быть переменными или терминалами). Например, грамматика, показанная на рис. 13.6, генерирует язык, представленный на рис. 13.5 и состоящий из наборов скобок, в которых одновременно может быть открыто до четырех пар.

$$
\begin{aligned}
S &\rightarrow (A \\
S &\rightarrow \lambda \\
A &\rightarrow (B \\
A &\rightarrow)S \\
B &\rightarrow (C \\
B &\rightarrow)A \\
C &\rightarrow (D \\
C &\rightarrow)B \\
D &\rightarrow)C
\end{aligned}
$$

Рис. 13.6. Грамматика, генерирующая язык парных скобок, в котором не допускается более четырех открытых пар одновременно

Мы будем использовать грамматику, в которой построение строки начинается с S и затем на каждом шаге

переменная заменяется до тех пор, пока получившаяся строка полностью не будет состоять из терминалов. Например: $S \rightarrow (A \rightarrow ((B \rightarrow (() A \rightarrow (() (B \rightarrow (() () A \rightarrow \rightarrow (() ()) S \rightarrow (() ())$.

Такая грамматика называется праволинейной, потому что при каждом преобразовании после переменной (если она есть) стоит терминальный символ. Каждое порождающее правило в праволинейной регулярной грамматике имеет одну из трех форм: переменная присоединяется к терминалу, переменная ставится после терминала или λ. Леволинейная регулярная грамматика идентична праволинейной, только переменная присоединяется к терминалу, терминал ставится после переменной или λ. Обратите внимание, что эти два вида грамматик не смешиваются; для представления любого регулярного языка можно использовать только леволинейную регулярную или праволинейную регулярную грамматику, но не обе одновременно.

Подводные камни на практике

Казалось бы, еще один способ работы с регулярными языками — это регулярные выражения (regex). Действительно, с точки зрения Computer Science это так. Регулярные выражения описывают регулярные языки и обладают теми же возможностями описания, что и регулярные грамматики.

Однако в программировании регулярные выражения расширены и теперь поддерживают многие языки, которые на самом деле не являются регулярными, поскольку включают в себя запоминание введенных

ранее данных (обратные ссылки). Благодаря этому regex в программировании гораздо мощнее, чем регулярные выражения в Computer Science.

Зато теоретические регулярные выражения по крайней мере не подвержены катастрофическим возвратам.

13.2.2. Свойства замыкания

Все слова языка образуют множество[1]. Множества могут быть замкнуты относительно определенных операций. Это означает, что если применить эту операцию к членам множества, мы получим другой элемент данного множества. Например, целые числа замкнуты относительно сложения, вычитания и умножения, потому что сложение, вычитание и умножение двух целых чисел дает еще одно целое число. Целые числа не замкнуты относительно деления, потому что результат деления одного целого числа на другое не обязательно является целым числом.

Аналогично класс языков является замкнутым относительно операции $\circ$, если для любых языков из этого класса результатом применения операции является язык этого же класса. Регулярные языки замкнуты относительно таких операций, как:

- **объединение** $A \cup B$: каждое слово, которое входит хотя бы в один из языков A или B;

[1] Напомню, что множество — это неупорядоченная коллекция элементов.

- **пересечение** $A \cap B$: каждое слово, которое входит и в A, и в B;
- **конкатенация** A с B: множество слов, каждое из которых представляет собой любую строку из A, после которой идет любая строка из B;
- **дополнение** $\overline{A}$: все слова, составленные из того же алфавита Σ, что и A, но не принадлежащие A;
- **разность** $A - B$: все слова, которые есть в A, но которых нет в B;
- **звезда Клини** A^*: ноль копий A и более;
- **плюс Клини** A^+: одна копия A и более.

Следующие свойства либо являются очевидными, либо логически следуют из определения регулярных языков в терминах теории множеств:

- пустой язык $L = \{\varnothing\}$ является регулярным;
- язык, содержащий только пустую строку $L = \lambda$, является регулярным;
- для каждой буквы в алфавите Σ язык, содержащий только эту букву, является регулярным;
- если A и B являются регулярными языками, то $A \cup B$, конкатенация A и B, а также A^* являются регулярными.

Ни один язык с алфавитом Σ, который не может быть построен по этим правилам, не является регулярным.

Чтобы доказать, что регулярный язык замкнут относительно определенной операции, нужно либо показать, что можно создать конечный автомат (или регулярное

выражение), представляющий операцию замыкания, либо показать, что эта операция является объединением уже проверенных операций замыкания.

Пример

Конкатенация. Если A и B являются регулярными языками, то добавление перехода λ из каждого допустимого состояния НКА для A в начальное состояние НКА для B[1] дает НКА для конкатенации A с B.

Дополнение. Если заменить любое допустимое состояние ДКА для языка L на недопустимое состояние (и наоборот), то получим ДКА для дополнения L.

Пересечение. Пересечение множеств A и B (элементы, присутствующие в обоих множествах) — это те же самые элементы, которых нет в множестве элементов, отсутствующих в A, или элементов, отсутствующих в B. Другими словами, это $\overline{\bar{A} \cup \bar{B}}$. Таким образом, зная, что регулярные языки замкнуты относительно объединения, из дополнения следует, что они также замкнуты относительно пересечения.

13.2.3. Лемма о накачке

Чтобы показать, что язык является регулярным, я использую либо конечные автоматы, либо регулярную грамматику, которая описывает этот язык. Как показать, что язык не является регулярным? Неспособность найти подходящий автомат или грамматику не является

[1] Следовательно, эти состояния в A больше не являются допустимыми.

доказательством того, что, возможно, она существует, просто я ее не нашел.

Вместо того чтобы доказывать, что язык не является регулярным, мы используем то, что называется леммой о накачке. Лемма о накачке не доказывает, что язык является регулярным, а лишь показывает, что это не так. Это способ доказательства от противного — мы предполагаем, что язык является регулярным, и показываем, что любой автомат, который допустим для этого языка, также будет считать допустимыми строки, которых нет в данном языке.

Рассмотрим язык $L = \{a^n b^n\}$ (то есть любое количество букв a, после которого следует такое же количество букв b). Предположим, что этот язык — регулярный; тогда существует конечный автомат, для которого этот язык является допустимым. Данный автомат имеет конечное число состояний, обозначим его k. Далее, если этот автомат считает допустимой строку длиной k и более символов, то одно и то же состояние должно посещаться более одного раза (в автомате есть цикл).

Рассмотрим строку $a^k b^k$, которая заведомо есть в языке. Поскольку для нашего автомата этот язык допустимый, то и строка $a^k b^k$ является допустимой для этого автомата. Однако, поскольку число букв a в строке совпадает с количеством состояний, в автомате должен существовать цикл размером j[1], который необходимо выполнить, чтобы распознать эту строку. При чтении букв a мы проходим этот цикл еще один раз, после чего попадаем в допустимое состояние для строки $a^{k+j} b^k$, которой в языке

[1] $|j| > 0$.

заведомо нет. Это противоречит предположению о том, что для данного языка существует конечный автомат, и показывает, что этот язык не регулярный.

13.3. Контекстно свободные языки

Контекстно свободные языки — это надмножество регулярных языков: любой регулярный язык также является контекстно свободным языком. Оба типа языков входят в состав иерархии грамматик Хомского (рис. 13.7), в которой каждая грамматика является собственным надмножеством менее мощных грамматик иерархии.

Рис. 13.7. Иерархия Хомского. Регулярные языки являются собственным подмножеством контекстно свободных языков, которые, в свою очередь, являются собственным подмножеством контекстно зависимых языков, а те являются собственным подмножеством рекурсивно перечислимых языков

Подобно регулярным языкам, контекстно свободные языки могут быть описаны в терминах автоматов,

которые могут их распознавать, грамматик, которые могут их создавать, и свойств замыкания множества. Контекстно свободные языки более мощные, чем регулярные языки, поскольку в них допускается использование памяти, в частности стека (сколь угодно большого размера). Интуитивно понятно, что контекстно свободными языками являются те, в которых нужно помнить не более одного элемента за раз, — так (как вскоре будет доказано), язык $a^n b^n$ будет контекстно свободным, а $a^n b^n c^n$ — нет.

13.3.1. Магазинные автоматы

Магазинные (стековые) автоматы (МП-автоматы) (PushDown Automata) очень похожи на конечные автоматы с двумя исключениями. Чтобы определить, какой переход выполнить, такие автоматы, кроме чтения очередного элемента входных данных, могут использовать верхний элемент стека. Автоматы также могут управлять стеком в процессе перехода.

Подобно конечным автоматам, магазинные состоят из конечного множества состояний (одно или несколько из которых могут быть допустимыми) и переходов между ними. Однако, кроме чтения буквы из входных данных, в процессе этих переходов автомат может извлекать переменную из стека и помещать одну или несколько переменных в стек. В дополнение к алфавиту терминалов, здесь есть множество символов стека (в которое иногда входит специальный символ, обычно Z, обозначающий дно стека).

Магазинный автомат может проверить строку на допустимость двумя способами. Первый способ — дойти до заключительного состояния, как в случае конечных автоматов: МП-автомат считает строку допустимой, если после чтения строки автомат находится в допустимом состоянии. Кроме того, МП-автомат также считает строку допустимой в случае пустого стека. Одни и те же языки можно выразить, используя любой метод доказательства допустимости. Но, как правило, удобно предположить, что МП-автомат должен находиться в допустимом состоянии, а также иметь пустой стек; именно этому соглашению мы и будем следовать.

Рассмотрим автомат, показанный на рис. 13.8.

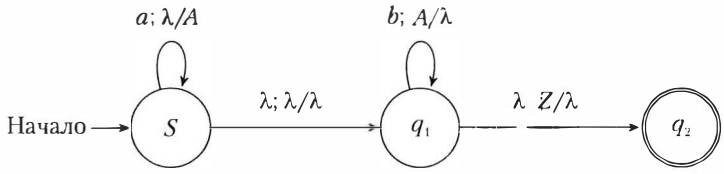


Рис. 13.8. Недетерминированный магазинный автомат, для которого является допустимым язык $L = \{a^n b^n\}$. Предполагается, что в начале стек содержит Z

В первом состоянии S мы можем прочитать любое количество букв a (включая нулевое). Для каждой прочитанной буквы a мы ничего не извлекаем из стека, но помещаем туда A. Закончив чтение букв a, мы выполняем λ-переход (при котором ничего не читаем, не помещаем в стек и не извлекаем оттуда) в состояние q_1. В состоянии q_1 мы считываем столько букв b, сколько это возможно, с учетом того, что при каждом чтении буквы b мы извлекаем

из стека A. Закончив чтение входных данных, мы выполняем последний λ-переход в допустимое состояние, извлекая в процессе перехода символ дна стека.

Как и в случае конечных автоматов, МП-автоматы могут быть детерминированными или недетерминированными. МП-автомат является недетерминированным, если для данной ситуации (текущего состояния, следующего считываемого символа и переменной вверху стека) существует несколько возможных переходов. Детерминированный магазинный автомат имеет не более одного возможного перехода из каждого состояния, после которого можно читать следующий символ из входных данных и верхнего символа стека.

В отличие от конечных автоматов детерминированные и недетерминированные магазинные автоматы не являются взаимозаменяемыми. Недетерминированные МП-автоматы могут реализовать любой контекстно свободный язык, в то время как детерминированные МП-автоматы реализуют собственное подмножество детерминированных контекстно свободных языков.

13.3.2. Контекстно свободная грамматика

Контекстно свободная грамматика, которая генерирует контекстно свободный язык, — это грамматика, левая часть которой всегда представляет собой одну переменную, а правая часть может быть любым числом переменных и терминалов. Это надмножество регулярных грамматик, для которых левая часть также должна представлять собой единственную переменную, но правая часть всегда является только одним терминалом, после

которого следует ноль или одна переменная (или перед ним стоит леволинейная регулярная грамматика). Интуитивно понятно, что контекстно свободными грамматиками являются такие, в которых (поскольку левая часть в правиле контекстно свободной грамматики всегда является единственной переменной) переменная всегда может быть преобразована в любую из возможных замен, независимо от чего-либо (от контекста), что может стоять до или после нее (рис. 13.9).

$$S \rightarrow aSb$$
$$S \rightarrow \lambda$$

Рис. 13.9. Контекстно свободная грамматика, которая генерирует язык $a^n b^n$

13.3.3. Лемма о накачке

Как и в случае с регулярными языками, можно доказать, что язык не является контекстно свободным. Для этого нужно выбрать строку, принадлежащую языку, и показать, что любой магазинный автомат, который генерирует эту строку, также должен генерировать строки, отсутствующие в данном языке. Лемма о накачке для контекстно свободных языков гласит, что если язык L контекстно свободный, то любая строка s из L, имеющая длину p или более символов, может быть записана как $s = uvwxy$, так что выполняются следующие условия:

- $uv^n wx^n y$ принадлежит к языку L для всех $n \geq 0$.

  Это условие говорит о том, что, если v и x повторяются одинаковое число раз (включая ноль), мы все равно получим строку, которая принадлежит языку.

Чтобы отследить, сколько раз мы выполняем первый цикл, и повторить то же количество итераций во втором цикле, можно использовать стек;

- $|vwx| \leq p$.

  Отсюда следует, что количество символов в цикле, включая неповторяющуюся часть (x), не больше p;

- $|vx| \geq 1$.

  Данное условие показывает, что повторяющаяся часть цикла содержит хотя бы один символ.

Предположим, что язык L является контекстно свободным; тогда существует автомат с p состояниями, для которого этот язык будет допустимым. Если строка языка имеет длину не менее p, то путь, который проходит строка, должен содержать цикл длиної не менее единицы. Если выполнить этот цикл еще один раз или не выполнять его вообще, то мы должны закончить в том же конечном состоянии. Следовательно, есть другая строка, которая также должна принадлежать этому языку. Если условие не соблюдается, то это противоречит нашему первоначальному предположению о существовании магазинного автомата, для которого L — допустимый язык и L не является контекстно свободным языком.

Хитрость использования леммы о накачке для контекстно свободных языков состоит в том, что обычно существует несколько способов разбить строку таким образом, чтобы она соответствовала приведенным выше правилам. Мы должны показать, что любой способ разбиения строки все равно выбрасывает нас за пределы языка. Таким образом, главным условием эффектив-

ного использования леммы о накачке является выбор хороших строк, у которых не очень много вариантов разбиения.

Рассмотрим язык $L = \{a^n b^n c^n\}$ и выберем строку $s = a^p b^p c^p$. Если строка целиком состоит из букв a, то в результате накачки мы получим строку, в которой букв a больше, чем b и c, поэтому строка не относится к данному языку. Это верно и для случая, когда строка полностью состоит из букв b или c. Если же начальная строка содержит два символа, то результат будет содержать еще несколько этих символов, после которых будет стоять третий (или меньше, если использовать обратную накачку, то есть вообще не выполнять цикл). Исходная строка не может содержать все три символа, поскольку ее длина не может быть больше p, а между последней a и первой c должно стоять p букв b. Таким образом, любой способ, которым теоретически можно разбить строку, позволяет превратить ее во что-то, чего нет в языке. Следовательно, такой язык не является контекстно свободным.

13.4. Контекстно зависимые языки

Контекстно зависимые языки — это следующий уровень в иерархии Хомского, и они включают в себя контекстно свободные языки. Если регулярные языки реализуются конечными автоматами, а контекстно свободные языки — магазинными автоматами, то контекстно зависимые языки реализуются *линейно ограниченными автоматами* (Linear Bounded Automata). Такой автомат — это недетерминированная машина Тьюринга

(см. главу 14), в которой длина ленты ограничена и зависит от длины входных данных[1].

Контекстно зависимые языки генерируются неукорачивающими грамматиками — это грамматики, которые не содержат каких-либо правил, в которых левая часть длиннее правой. Другими словами, при использовании такой грамматики длина перезаписываемой строки никогда не уменьшается.

$L = \{a^n b^n c^n, n \geq 1\}$ является контекстно зависимым языком, поскольку его можно сгенерировать с помощью следующей грамматики (рис. 13.10).

$$S \rightarrow abc$$
$$S \rightarrow aSBc$$
$$cB \rightarrow Bc$$
$$bB \rightarrow bb$$

Рис. 13.10. Грамматика, которая генерирует язык $a^n b^n c^n, n \geq 1$

13.5. Рекурсивные и рекурсивно перечислимые языки

Рекурсивные языки также реализуются машинами Тьюринга, но они снимают ограничение на длину используемой ленты. Если существует машина Тьюринга, которая в итоге прекращает работу на любом заданном наборе входных данных и правильно признает

[1] Формально длина ленты является линейной функцией длины входных данных.

допустимыми или отклоняет строки некоторого языка, то этот язык является рекурсивным.

Рекурсивно перечислимые языки — это те языки, для которых машина Тьюринга может перечислить все допустимые строки. Другими словами, мы снимаем требование, чтобы машина Тьюринга прекращала работу, если строка не относится к данному языку. Если язык, комплементарный рекурсивно перечислимому языку, также является рекурсивно перечислимым, то такой язык является рекурсивным.

14 Машины Тьюринга

14.1. Чисто теоретический компьютер

Машина Тьюринга — это абстрактная машина, способная моделировать любой алгоритм. Возможно, другие модели вычислений могут работать быстрее, использовать меньше памяти, их легче программировать, но если машина Тьюринга не способна решить какую-то задачу, значит, никакая другая машина также не способна это сделать (насколько мы знаем, это тезис Черча — Тьюринга).

> **Предупреждение**
>
> Говоря о компьютере без каких-либо уточнений, я имею в виду классический (не квантовый) компьютер.

Машина Тьюринга — это конечный автомат, который работает с лентой памяти бесконечной длины, разделенной на отдельные ячейки. У машины есть головка, которая расположена над определенной ячейкой; при запуске алгоритма машина считывает значение этой

ячейки. Затем она может выполнить запись в эту ячейку, переместить головку влево или вправо и перейти в новое состояние. Другой способ представить машину Тьюринга — как магазинный автомат с двумя стеками, в котором один стек представляет часть ленты, расположенную слева от головки, а второй — остальную часть ленты.

Установка различных ограничений на машину Тьюринга может сделать ее эквивалентной другим моделям вычислений. Например, если машина осуществляет только чтение и может перемещать головку только вправо, то она эквивалентна НКА. И наоборот, различные ослабления ограничений на машины Тьюринга — сделать их недетерминированными, добавляя дополнительные ленты и т. д., — не расширяют класс задач, которые может решить машина (хотя это влияет на число операций, необходимых для решения этих задач).

Универсальная машина Тьюринга, способная моделировать другие машины Тьюринга, эквивалентна по мощности (то есть по задачам, которые она способна решать) реальному компьютеру (при условии, что у реального компьютера бесконечная память).

14.2. Построение машины Тьюринга

Машину Тьюринга можно представить так же, как конечные автоматы: в виде последовательности состояний и переходов между этими состояниями (рис. 14.1).

Каждый переход включает в себя считываемый символ (если он есть), записываемый символ (если есть) и то, должна ли головка машины передвигаться влево, вправо или остаться в той же ячейке ленты. Строка принадлежит языку, который является допустимым для машины Тьюринга, если обработка ленты, содержащей эту строку, приведет к остановке машины в допустимом состоянии. Строка не будет допустимой, если не существует корректного перехода, который можно было бы выполнить.

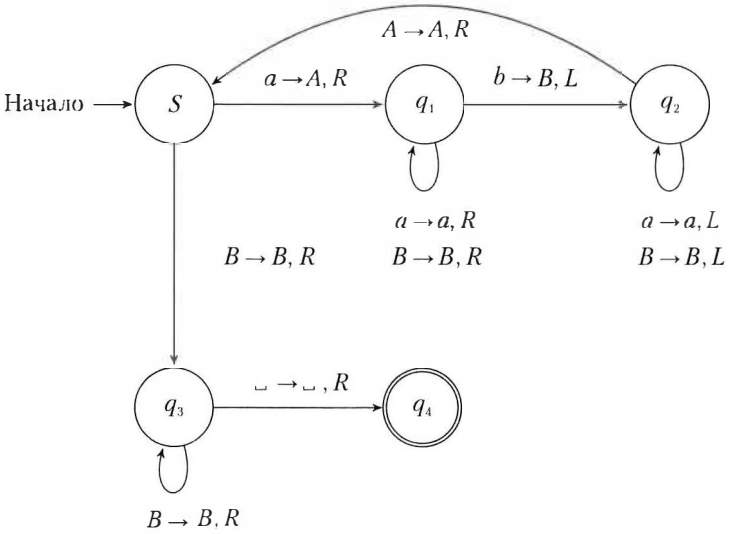


Рис. 14.1. Машина Тьюринга для $a^n b^n$. ●братите внимание, что при последнем переходе читается пустая ячейка. Машина находит буквы a и соответствующие буквы b, повторяя так до тех пор, пока в строке не закончатся символы

14.3. Полнота по Тьюрингу

Система правил, такая как язык программирования, называется Тьюринг-полной, если ее можно использовать для моделирования любой машины Тьюринга. Поскольку машина Тьюринга способна решить любую задачу, которая в принципе может быть решена компьютером, это означает, что Тьюринг-полный язык также способен решить любую задачу, которая может быть решена компьютером с использованием любого языка. Обратите внимание, что здесь ничего не говорится о том, сколько времени потребуется для решения задачи, а лишь то, что машина в итоге придет к решению задачи.

Процедурным языком является Тьюринг-полный язык, если в нем есть условное ветвление и он способен обрабатывать сколь угодно большой объем памяти. Это означает, что (без учета аппаратных ограничений) большинство языков программирования являются Тьюринг-полными.

14.4. Проблема остановки

Если машина Тьюринга способна решить любую задачу, которая в принципе может быть решена компьютером, это означает, что существуют задачи, которые не может решить ни один компьютер.

Одной из таких является проблема остановки: если даны описание произвольной программы и входные данные, как определить, закончит ли когда-нибудь

работу эта программа? Задача считается неразрешимой — не существует машины Тьюринга, которая могла бы ответить на этот вопрос для всех возможных входных данных. Машина для языка L всегда останавливается на любой строке этого языка, но может работать вечно, если строка не принадлежит данному языку и при этом язык не рекурсивный.

Как и в случае *NP*-полноты, можно продемонстрировать, что задача неразрешима, показав, что ее решение также позволило бы решить еще одну задачу, относительно которой уже было доказано, что она неразрешимая.

Послесловие

Вы дошли до конца книги! (Если не считать приложения.)

Сейчас вы, возможно, чувствуете некую незавершенность — как будто вам нужно еще многое узнать. И вы правы.

Эта книга неполна в двух смыслах. Во-первых, ни одна из описанных тем не рассмотрена глубоко (если бы это было так, книга была бы втрое толще). Тем не менее теперь у вас есть общее представление, которое позволит вам осмысленно участвовать в обсуждении вопросов и по мере необходимости искать дополнительную информацию.

Во-вторых, остается еще много тем, которые нужно охватить: доказательства, безопасность, операционные системы, сети... этот список можно продолжать. Вероятно, вы заметили перекрестные ссылки на главы, которых нет в этой книге; вы найдете их в следующем томе. Если вам понравилась данная книга, обязательно дождитесь выхода второго тома.

Я надеюсь, что вы посетите мой сайт http://www.what-williamsaid.com/books/. Там вы найдете тесты для самопроверки, которые позволят вам узнать, хорошо ли вы усвоили каждую главу. Вы можете подписаться на

мои рассылки, чтобы получить уведомление о выходе следующего тома (а также бесплатные дополнительные материалы), и связаться со мной, если у вас возникнут вопросы.

Если вы считаете эту книгу полезной, я буду очень признателен, если вы потратите пару минут и оставите отзыв о ней. Обратная связь с читателями очень важна для авторов!

Приложения

A Необходимая математика

Насколько глубокие познания в математике нужны для изучения Computer Science?

Обычно для поступления на специальности, связанные с Computer Science, нужно сдавать экзамен по алгебре; это необходимо потому, что студенты, у которых возникают сложности с пониманием переменных в алгебре, скорее всего, также с трудом поймут переменные в программировании. Поскольку эта книга ориентирована на программистов-практиков, предполагается, что читатель знаком с концепцией и использованием переменных.

При анализе времени выполнения алгоритма понадобятся алгебра и логарифмы. Логарифм числа — это степень, в которую нужно возвести основание логарифма (обычно для компьютеров это 2), чтобы получить данное число; например, логарифм 16 по основанию 2 равен 4, потому что $2^4 = 16$.

Более сложные темы могут также потребовать углубленного знания математики. В компьютерной графике

используются мнимые числа, а в машинном обучении — численные методы и статистика. Эти темы выходят за рамки данной книги.

Если вы не работаете в специализированной области, то вам, скорее всего, будет достаточно алгебры, логарифмов и теории графов (которые подробно рассмотрены в части II).

Б Классические NP-полные задачи

В этом приложении представлен краткий обзор некоторых классических *NP*-полных задач. Здесь не приводятся доказательства *NP*-полноты, а только дана информация, чтобы вы сами могли распознать эти задачи, когда встретитесь с ними. Обратите внимание, что задачи описаны так, что интересен не размер наилучшего решения, а то, существует ли решение размером s. Это связано с тем, что по определению *NP*-полные задачи — это задачи принятия решений (ответом на которые является «да» или «нет»).

Б.1. SAT и 3-SAT

Задача выполнимости булевых формул, ВЫП (boolean satisfiability problem, SAT), отвечает на вопрос: можно ли для заданной формулы, состоящей из логических переменных, подобрать такие значения этих переменных, чтобы результат формулы был истинным? Например, формула «b и не c» принимает значение «истина», если b истинно, а c — ложно. А вот утвер-

ждение «b и не b» не станет истинным ни при каких значениях b.

Как правило, формула представлена в виде набора условий. Например, если задана формула, значение которой истинно, если истинно любое из приведенных выше утверждений, то ее можно записать как $(b \wedge \neg c) \vee (b \wedge \neg b)$ (это читается так: «b и не c или b и не b»). В данном случае каждое из утверждений содержит два литерала, но вообще оно может содержать любое число литералов. 3-SAT — это та же задача с дополнительным ограничением: каждое условие ограничено максимум тремя литералами.

Б.2. Клика

Для заданного графа G и размера k определите, содержит ли G клику (то есть полный подграф) размером k.

Б.3. Кликовое покрытие

Для заданного графа G и размера k определите, можно ли разбить G на k клик таким образом, чтобы каждая вершина графа принадлежала хотя бы одной из полученных клик.

Б.4. Раскраска графа

Для заданного графа G и размера k определите, можно ли правильно раскрасить G, используя только k цветов.

Б.5. Гамильтонов путь

Для заданного графа G определите, существует ли путь между вершинами графа, который проходит через каждую вершину ровно один раз.

Б.6. Укладка рюкзака

Для заданного набора предметов, каждый из которых имеет вес и ценность, и рюкзака с максимальной вместимостью c определите, можно ли найти набор предметов с общей ценностью не менее v, которые не превышают вместимость рюкзака.

Б.7. Наибольшее независимое множество

Для заданного графа G определите, существует ли независимое множество (множество вершин, в котором нет двух смежных вершин) размером k.

Б.8. Сумма подмножества

Для заданного множества или мультимножества (множества, в котором допускаются повторяющиеся значения) целых чисел и значения s определите, существует ли непустое подмножество, сумма которого равна s? Например, для множества $\{-7, -5, -3, -1, 4, 8, 156\}$ и $s = 0$ таким подмножеством было бы $\{-7, -5, 4, 8\}$.

Вильям Спрингер

Гид по Computer Science для каждого программиста

Перевела с английского *Е. Сандицкая*

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*С. Давид*
Ведущий редактор	*Н. Гринчик*
Научный редактор	*Н. Искра*
Литературный редактор	*А. Дубейко*
Художественный редактор	*В. Мостипан*
Корректор	*Е. Павлович*
Верстка	*Г. Блинов*

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: 06.2020. Наименование: книжная продукция.
Срок годности: не ограничен.

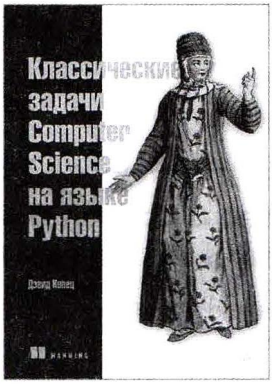
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

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 20.05.20. Формат 60×90/16. Бумага офсетная. Усл. п. л. 12,000.
Тираж 1000. Заказ С-936.
Отпечатано в типографии ООО «ИНФО СИСТЕМ».
420044, Россия, г. Казань, пр. Ямашева, д. 36Б.

Дэвид Копец

КЛАССИЧЕСКИЕ ЗАДАЧИ COMPUTER SCIENCE НА ЯЗЫКЕ PYTHON



Многие задачи в области Computer Science, которые на первый взгляд кажутся новыми или уникальными, на самом деле уходят корнями в классические алгоритмы, методы кодирования и принципы разработки. И устоявшиеся техники по-прежнему остаются лучшим способом решения таких задач! Научитесь писать оптимальный код для веб-разработки, обработки данных, машинного обучения и других актуальных сфер применения Python. Книга даст вам возможность глубже освоить язык Python, проверить себя на испытанных временем задачах, упражнениях и алгоритмах. Вам предстоит решать десятки заданий по программированию: от самых простых (например, найти элементы списка с помощью двоичной сортировки) до сложных (выполнить кластеризацию данных методом k-средних). Прорабатывая примеры, посвященные поиску, кластеризации, графам и пр., вы вспомните то, о чем успели позабыть и овладеете классическими приемами решения повседневных задач.