Задача данной книги проста: разобрать «идеи», лежащие в основе программ, и показать, как доказывать их правильность.

Как математически доказать, что заданный алгоритм делает то, что он должен делать? И почему это так важно?

Доказывается правильность классических алгоритмов: целочисленного деления, алгоритм Евклида, ранжирования, др. Помимо традиционных алгоритмов, таких как жадные алгоритмы, алгоритмы динамического программирования и алгоритмы «разделяй и властвуй», книга исследует также рандомизированные и онлайновые алгоритмы. Первые стали повсеместными из-за появления криптографии, а вторые необходимы во многих областях, начиная с операционных систем и заканчивая фондовым рынком.

Книга усеяна задачами. Большинство задач теоретические, но многие требуют реализации алгоритма; для таких задач используется язык программирования **Python 3.**

Несмотря на свою краткость, издание является математически строгим. Желательно предварительное знакомство с дискретной математикой.

Издание предназначено для студентов вузов, специалистов в области информатики и математики, а также широкого круга программистов и разработчиков.

Файлы к книге, в том числе упражнения на языке Python доступны на сайте книги www.msoltys.com/book, а также на сайте издательства www.dmkpress.com.

Основные темы книги:

- предварительные условия;
- жадный алгоритм;
- разделяй и властвуй;
- динамическое программирование;
- онлайновые алгоритмы;
- рандомизированные алгоритмы;
- алгоритмы в линейной алгебре;
- вычислительная основа;
- математическая основа.

Майкл Солтис (Michael Soltys) *является профессором и заведующим кафедрой компьютерных наук Калифорнийского государственного университета на Нормандских островах, США, с 2014 года. Его научные интересы лежат в области алгоритмов, в особенности в области строковых алгоритмов и кибербезопасности. Солтис консультирует бизнес и промышленность в области цифровой криминалистики и информационной безопасности. Регулярно ведет курсы по кибербезопасности и алгоритмам.*

Интернет-магазин:
www.dmkpress.com

Оптовая продажа:
КТК “Галактика”
books@alians-kniga.ru

World Scientific

ДМК ПРЕСС ИЗДАТЕЛЬСТВО
www.дмк.рф

ISBN 978-5-97060-696-4

Введение в анализ алгоритмов

ДМК ПРЕСС ИЗДАТЕЛЬСТВО

Майкл Солтис

Введение в анализ алгоритмов

Michael Soltys
California State University Channel Islands, USA

An Introduction to the Analysis of Algorithms

3rd Edition

World Scientific

NEW JERSEY • LONDON • SINGAPORE • BEIJING • SHANGHAI • HONG KONG • TAIPEI • CHENNAI • TOKYO

Майкл Солтис
Калифорнийский университет Нормандские острова

Введение в анализ алгоритмов

ДМК Пресс
издательство
Москва, 2019

УДК 510.5
ББК 22.18я73
С60

Солтис М.
С60 Введение в анализ алгоритмов / пер. с анг. А. В. Логунова. – М.: ДМК Пресс, 2019. – 278 с.: ил.

ISBN 978-5-97060-696-4

Книга представляет собой краткое, но математически строгое введение в анализ различных алгоритмов с точки зрения доказывания их правильности. Вы ознакомитесь с основными свойствами линейных, ветвящихся и циклических алгоритмов и способами их проверки. Книга содержит большое количество теоретических задач и практических примеров на языке Python.

Издание предназначено для студентов вузов, специалистов в области информатики и математики, а также широкого круга программистов и разработчиков.

УДК 510.5
ББК 22.18я73

ISBN 978-981-3235-90-8 (анг.)
ISBN 978-5-97060-696-4 (рус.)

Посвящается моей семье

Содержание

Предисловие

> Ах, если бы он чуть поменьше знал, насколько лучше он мог бы научить неизмеримо большему!
>
> *Чарльз Диккенс [Dickens (1854)], стр. 7*

Эта книга представляет собой краткое введение в анализ алгоритмов с точки зрения доказывания правильности алгоритма. Приведенная выше цитата относится к г-ну Чадомору, карикатуре на учителя из «Тяжелых времен» Чарльза Диккенса, который душит умы своих учеников слишком большим объемом информации. Мы избежим ошибки Чадомора и возведем краткость в добродетель.

Наша тема заключается в следующем: как математически, без бремени чрезмерного формализма, доказывать, что заданный алгоритм делает то, что он должен делать? И почему это так важно? По словам К. А. Р. Хоара:

> *Что касается фундаментальной науки, то мы до сих пор точно не знаем, как доказывать правильность программ. Нам требуется довольно много устойчивого прогресса в этой области, который можно было бы предвидеть, и ряд прорывов, когда люди внезапно обнаруживают, что, оказывается, существует простой способ сделать то, что всеми до сих пор считалось слишком трудным*[1].

Инженеры по программному обеспечению знают много примеров того, когда дела принимают ужасный оборот из-за программных ошибок; их конкретными фаворитами являются следующие два из них[2]. Отключение электроэнергии на северо-востоке Америки летом 2003 года было вызвано программным дефектом в системе энергоуправления; сигнал тревоги, который должен был сработать, так и не сработал, что привело к цепочке событий, кульминацией которых стало каскадное отключение электроэнергии. Ариан 5, полет 501, первый полет ракеты 4 июня 1996 г., закончился взрывом на 40-й секунде полета; этот убыток в размере 500 млн долл. был вызван переполнением при конвертации 64-разрядного числа с плавающей запятой в 16-разрядное целое число со знаком.

Когда Ричард А. Кларк, бывший национальный координатор по безопасности, спросил Эда Аморосо, руководителя подразделения по сетевой безопасности AT&T Network Security, что нужно сделать в отношении уязвимостей в киберинфраструктуре США, Аморосо сказал:

> *Программное обеспечение – это большая часть проблемы. Мы должны писать программное обеспечение, которое имеет гораздо меньше ошибок и является гораздо безопасным*[3].

---

[1] Из интервью с К. А. Р. Хоаром, разработчиком алгоритма «быстрой сортировки», в [Shustek (2009)].

[2] Эти два примера взяты из публикации [van Vliet (2000)], где можно найти еще много примеров впечатляющих провалов.

[3] См. стр. 272 в [Clarke и Knake (2011)].

Фред Д. Тейлор-младший, подполковник ВВС Соединенных Штатов и сотрудник по национальной безопасности в Гарвардской школе Кеннеди, писал схожим образом:

> *Широкое использование программного обеспечения создало новые и широкие возможности. Наряду с этими возможностями появляются новые факторы уязвимости, ставящие под угрозу глобальную инфраструктуру и нашу национальную безопасность. Повсеместный характер интернета и тот факт, что он раздается посредством общих протоколов и процессов, позволяют любому человеку, обладающему знаниями, создавать программное обеспечение для участия в деятельности по всему миру. Однако у большинства разработчиков программного обеспечения нет стимула создавать более безопасное программное обеспечение*[1].

Безопасность программного обеспечения естественным образом относится к правильности программного обеспечения как ее составная часть.

В то время как цель правильности программы неуловима, мы можем разработать методы и приемы для сокращения ошибок. Задача этой книги довольно скромная: мы хотим представить введение в анализ алгоритмов – «идеи», лежащие в основе программ, и показать, как доказывать их правильность.

Алгоритм может быть правильным, но сама реализация может быть ошибочной. Некоторые синтаксические ошибки в реализации программы могут быть обнаружены компилятором или транслятором, которые, в свою очередь, также сами могут быть дефектными, но могут быть и другие скрытые ошибки. Само оборудование может быть неисправным; библиотеки, на которые опирается программа во время выполнения, могут быть ненадежными и т. д. Основная задача программиста – писать исходный код, который работает в условиях такой непрочной, подверженной ошибкам среды. Наконец, алгоритмическое содержимое компонента программного обеспечения может быть очень малым; большинство строк исходного кода может быть посвящено «черновой» задаче программирования интерфейса. Таким образом, способность правильно рассуждать о добротности алгоритма является лишь одним из многих аспектов рассматриваемой задачи, но важно, хотя бы по педагогической причине обучения, рассуждать об алгоритмах строго.

Мы начинаем эту книгу с главы предварительных условий, содержащей ключевые идеи индукции и инвариантности, а также математический каркас пред- и постусловий и инвариантов циклов. Мы также доказываем правильность некоторых классических алгоритмов, таких как алгоритм целочисленного деления и процедура Евклида для вычисления наибольшего общего делителя двух чисел.

Мы представим три стандартных метода проектирования алгоритмов в одноименных главах: жадные алгоритмы, динамическое программирование и парадигма «разделяй и властвуй». Нас интересует правильность алгоритмов, а не, скажем, эффективность или лежащие в их основе структуры данных. Например, в главе, посвященной жадной парадигме, мы подробно исследуем идею перспективного частичного решения, мощного метода доказательства правильности жадных алгоритмов. Мы включаем онлайновые алгоритмы и состязательный анализ, а также рандомизированные алгоритмы вместе с разделом по криптографии.

---

[1] Журнал «Национальная безопасность» Гарвардской школы права [Фред Д. Тейлор (2011)].

Алгоритмы решают задачи, и многие задачи в этой книге подпадают под категорию оптимизационных задач, будь то минимизация издержек, например алгоритм Краскала для вычисления остовных деревьев минимальной стоимости – раздел 2.1, или максимизация прибыли, например отбор наиболее прибыльного подмножества видов деятельности – раздел 4.4.

Книга усеяна задачами. Большинство задач теоретические, но многие требуют реализации алгоритма; для таких задач мы предлагаем язык программирования Python 3. Ожидается, что читатель самостоятельно изучит Python; см., например, книги [Dierbach (2013)] или [Downey (2015)][1]. Одним из преимуществ языка Python является то, что на нем легко начать писать небольшие фрагменты кода, которые работают, и подавляющая часть кода в этой книге попадает в категорию «маленький фрагмент». Решения большинства задач включены в «ответы на избранные задачи» в конце каждой главы. Решения для большинства упражнений по программированию будут доступны для загрузки с веб-страницы автора[2].

Целевая аудитория этой книги – студенты-выпускники и студенты старших курсов по информатике и математике. Презентация материала является самодостаточной: в первой главе представлены вышеупомянутые идеи пред- и постусловий, инвариантов циклов и завершения. Последняя глава, глава 9 «Математические основы», содержит необходимую информацию по индукции, принципу инвариантности, теории чисел, отношениям и логике. Читателю, незнакомому с дискретной математикой, рекомендуется начать с главы 9 и заняться всеми задачами в ней.

Эта книга опирается на ряд источников. Прежде всего книга [Cormen и соавт. (2009)] является фантастическим справочником для тех, кто изучает алгоритмы. Я также использовал в качестве ссылки элегантно написанную книгу [Kleinberg и Tardos (2006)]. Классикой в этой области является книга [Knuth (1997)], и я основываю свое изложение онлайновых алгоритмов на материале книги [Borodin и El-Yaniv (1998)]. Я научился жадным алгоритмам, динамическому программированию и логике у Стивена А. Кука в Университете Торонто. Сводка отношений в разделе 9.3 основана на лекциях, прочитанных Рышардом Яницким в 2008 году в Университете Макмастера. Раздел 9.4 основан на логических лекциях Стивена А. Кука, преподававшихся в Университете Торонто в 1990-х годах.

Я благодарен Райану Макинтайру, который прочел рукопись 3-го издания и обновил решения на языке Python летом 2017 года.

Как указано в начале этого предисловия, мы стремимся представить краткое, математически строгое введение в прекрасную область алгоритмов. Я полностью согласен с [Su (2010)], что цель образования состоит в том, чтобы культивировать «клич»:

Я издаю свой варварский клич над основанием (вершиной) мира!

Это слова Джона Китинга, цитирующего поэму Уолка Уитмена ([Whitman (1892)]) в фильме «Общество мертвых поэтов». Этот клич – глубокая тоска внутри каждого из нас по эстетическому опыту ([Scruton (2011)]). Надеюсь, настоящая книга предоставит один такой клич или два.

---

[1] PDF-файлы более ранних версий, до 2.0.17 на момент написания, доступны для бесплатного скачивания из Green Tea Press, http://greenteapress.com/wp/think-python.

[2] См. http://www.msoltys.com.

Глава 1

Предварительные условия

> Считается, что более 70% (!) усилий и затрат на разработку сложной программной системы посвящены, так или иначе, исправлению ошибок.
>
> *Алгоритм, стр. 107 [Harel (1987)]*

1.1. Что такое правильность?

Для того чтобы показать, что алгоритм является правильным, мы должны каким-то образом показать, что он делает то, что должен делать. Сложность состоит в том, что алгоритм разворачивается во времени, и сложно работать с переменным числом шагов, то есть циклами while. Мы собираемся ввести математический каркас для доказательства правильности алгоритма (и программы), который называется *логикой Хоара*. В этом математическом каркасе используются индукция и инвариантность (см. раздел 9.1), а также логика (см. раздел 9.4), но мы будем использовать ее неформально. Формальный пример см. в разделе 9.4.4.

Мы делаем два логических утверждения, именуемых *предусловием* и *постусловием*; под правильностью мы имеем в виду, что всякий раз, когда предусловие соблюдается перед исполнением алгоритма, постусловие будет соблюдаться после его исполнения. Под *завершением* (остановом) мы имеем в виду, что всякий раз, когда соблюдается предусловие, алгоритм перестанет работать после конечного числа шагов. Правильность без завершения называется *частичной правильностью*, а правильность сама по себе является частичной правильностью с завершением. Вся эта терминология приводится здесь для того, чтобы связать ту или иную задачу с каким-то алгоритмом, который призван ее решить. Следовательно, мы подбираем пред- и постусловие таким путем, который отражает эту связь и доказывает ее истинность.

Эти понятия можно сделать точнее, но мы должны ввести некую стандартную форму записи: *булевы связки*: $\wedge$ равно «и», $\vee$ равно «или» и $\neg$ равно «не». Мы также используем $\rightarrow$ в качестве логического следствия, то есть $x \rightarrow y$ логически эквивалентно $\neg x \vee y$, и $\leftrightarrow$ является булевой эквивалентностью, а $\alpha \leftrightarrow \beta$ выражает $((\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha))$. $\forall$ – это универсальный квантификатор «для всех», и $\exists$ – экзистенциальный квантификатор «существует». Мы используем «$\Rightarrow$» в качестве аббревиатуры для «влечет за собой», то есть $2|x \Rightarrow x$ является четным, в то время как «$\not\Rightarrow$» является аббревиатурой для «не влечет за собой».

Пусть A равно алгоритму, и пусть $\mathcal{I}_A$ равно множеству всех *хорошо сформированных входных данных*; идея состоит в том, что если $I \in \mathcal{I}_A$, то имеет «смысл» подать

I на вход в A. Понятие «хорошо сформированные» входные данные также можно уточнить, но нам будет достаточно того, что мы опираемся на наше интуитивное понимание – в частности, в алгоритм, который в качестве входных данных берет пары чисел, не будет «подаваться» матрица. Пусть $O = A(I)$ равно выходу из A на I, если он существует. Пусть α_A равно предусловию и β_A равно постусловию алгоритма A; если I удовлетворяет предусловию, то мы пишем $\alpha_A(I)$, а если O удовлетворяет постусловию, то мы пишем $\beta_A(O)$. Тогда частичная правильность A относительно предусловия α_A и постусловия β_A может быть сформулирована как:

$$(\forall I \in \mathcal{I}_A)[(\alpha_A(I) \wedge \exists O(O = A(I))) \rightarrow \beta_A(A(I))]. \quad (1.1)$$

На словах: для любых хорошо сформированных входных данных I, если I удовлетворяет предусловию, а $A(I)$ производит выходные данные (то есть завершается), то эти выходные данные удовлетворяют постусловию.

Полная правильность равна (1.1) вместе с логическим утверждением, что для всех $I \in \mathcal{I}_A$ завершается (и, следовательно, существуют O такие, что $O = A(I)$).

Задача 1.1. Модифицируйте (1.1) так, чтобы выразить полную правильность.

Фундаментальным понятием в анализе алгоритмов является понятие *инварианта цикла*; он представляет собой логическое утверждение, которое остается истинным после каждого исполнения цикла «while» (или «for»). Придумать правильное логическое утверждение и доказать его представляет собой настоящее творческое начинание. Если алгоритм завершается, то инвариант цикла – это логическое утверждение, которое помогает доказать импликацию $\alpha_A(I) \rightarrow \beta_A(A(I))$[1].

После того как показано, что инвариант цикла соблюдается, он используется для доказательства частичной правильности алгоритма. Таким образом, критерий отбора инварианта цикла заключается в том, что он помогает доказать постусловие. В общем случае желаемое доказательство правильности может дать ряд разных инвариантов циклов (и в этом отношении пред- и постусловия); искусство анализа алгоритмов состоит в их разумном отборе. Обычно, для того чтобы доказать, что выбранный инвариант цикла соблюдается после каждой итерации цикла, нам нужна индукция, и обычно нам также нужно предусловие, которое служит в качестве допущения в этом доказательстве.

1.1.1. Сложность

С учетом алгоритма $\mathcal{A}$ и входных данных x временем выполнения $\mathcal{A}$ на x является число шагов, которые требуются $\mathcal{A}$ для того, чтобы завершиться на входных данных x. Деликатный вопрос здесь – определить понятие «шаг», но мы отнесемся к нему неформально: мы будем считать, что у нас есть машина случайного доступа (машина, которая может осуществлять доступ к ячейкам памяти за один шаг), и мы будем считать, что присвоение типа $x \leftarrow y$ выполняется за один шаг, и то же самое касается арифметических операций и проверки булевых выражений (например, $x \geqslant y \wedge y \geqslant 0$). Разумеется, это упрощение не отражает истинное положение дел, если, например, мы манипулируем числами из 4000 бит (как в случае крип-

[1] Инвариантом называется логическое выражение, истинное перед началом выполнения цикла и после каждой итерации цикла, зависящей от переменных, изменяющихся в теле цикла. Инварианты используются для доказательства правильности выполнения цикла, а также при проектировании и оптимизации циклических алгоритмов. – *Прим. перев.*

тографических алгоритмов). Но тогда мы переопределяем шаги в соответствии с контекстом.

Нас интересует *сложность худшего случая*. То есть с учетом алгоритма $\mathcal{A}$ мы обозначаем через $T^{\mathcal{A}}(n)$ максимальное время выполнения $\mathcal{A}$ на любых входных данных x размера n. Здесь «размер» означает число бит в разумной фиксированной кодировке x. Вместо $T^{\mathcal{A}}(n)$ мы, как правило, пишем $T(n)$, так как обсуждаемый алгоритм задается контекстом. Оказывается, что $T(n)$ может быть очень сложным даже для простых алгоритмов, и поэтому мы соглашаемся на асимптотические границы на $T(n)$.

Для того чтобы обеспечить асимптотические аппроксимации для $T(n)$, мы вводим форму записи «*O* *большое*». Рассмотрим функции f и g из $\mathbb{N}$ в $\mathbb{R}$, то есть функции, область которых является натуральными числами, но может варьироваться над вещественными. Мы говорим, что $g(n) \in O(f(n))$, если существуют константы c, $n_0 \in \mathbb{N}$ такие, что для всех $n \geqslant n_0$, $g(n) \leqslant cf(n)$, и форма записи «*o* *малое*», $g(n) \in o(f(n))$, которая означает, что $\lim_{n\to\infty} g(n)/f(n) = 0$. Мы также говорим, что $g(n) \in \Omega(f(n))$, если существуют константы c, n_0 такие, что для всех $n \geqslant n_0$, $g(n) \geqslant cf(n)$. Наконец, мы говорим, что $g(n) \in \Theta(f(n))$, если имеет место, что $g(n) \in O(f(n)) \cap \Omega(f(n))$. Если $g(n) \in \Theta(f(n))$, то $f(n)$ называется *асимптотически плотной границей* для $g(n)$, а это означает, что $f(n)$ является очень хорошей аппроксимацией $g(n)$. Обратите внимание, что на практике мы часто будем писать $g(n) = O(f(n))$ вместо формального $g(n) \in O(f(n))$; небольшое, но удобное злоупотребление математической формой записи.

Например, $an^2 + bn + c = \Theta(n^2)$, где $a > 0$. Для того чтобы это увидеть, обратите внимание, что $an^2 + bn + c \leqslant (a + |b| + |c|)n^2$ для всех $n \in \mathbb{N}$, и поэтому $an^2 + bn + c = O(n^2)$, где мы взяли абсолютное значение b, c, потому что они могут быть отрицательными. С другой стороны, $an^2 + bn + c = a((n + c_1)^2 - c_2)$, где $c_1 = b/2a$ и $c_2 = (b^2 - 4ac)/4a^2$, благодаря чему мы можем найти c_3 и n_0, вследствие чего для всех $n \geqslant n_0$, $c_3 n^2 \leqslant a((n + c_1)^2 - c_2)$, и поэтому $an^2 + bn + c = \Omega(n^2)$.

Задача 1.2. Найдите c_3 и n_0 в терминах a, b, c. Затем докажите, что для $k \geqslant 0$ $\sum_{i=0}^{k} a_i n^i = \Theta(n^k)$; этим показывается упрощающее преимущество «O» большого.

1.1.2. Деление

Что может быть проще целочисленного деления? Даны два целых числа x, y, и мы хотим найти частное и остаток от деления x на y. Например, если $x = 25$ и $y = 3$, то $q = 3$ и $r = 1$. Обратите внимание, что возвращаемые алгоритмом деления q и r обычно обозначаются соответственно как $\mathrm{div}(x, y)$ (*частное*) и $\mathrm{rem}(x, y)$ (*остаток*).

Алгоритм 1.1. Деление

Предусловие: $x \geqslant 0 \wedge y > 0 \wedge x, y \in \mathbb{N}$

```
 1: q ← 0
 2: r ← x
 3: while y ⩽ r do
 4:         r ← r − y
 5:         q ← q + 1
 6: end while
 7: return q, r
```

Постусловие: $x = (q \cdot y) + r \wedge 0 \leqslant r < y$

В качестве инварианта цикла мы предлагаем следующее логическое утверждение:

$$x = (q \cdot y) + r \wedge r \geqslant 0, \tag{1.2}$$

и мы показываем, что (1.2) соблюдается после каждой итерации цикла. Базовый случай (то есть ноль итераций цикла – мы просто находимся перед строкой 3 алгоритма): $q = 0$, $r = x$, поэтому $x = (q \cdot y) + r$ и, поскольку $x \geqslant 0$ и $r = x$, $r \geqslant 0$.

Индукционный шаг: предположим, что $x = (q \cdot y) + r \wedge r \geqslant 0$, и мы еще раз пройдемся по циклу, и пусть q', r' равны новым значениям соответственно q, r (вычисленным в строках 4 и 5 алгоритма). Так как мы исполнили цикл еще раз, то получается, что $y < r$ (это условие проверено в строке 3 алгоритма), а так как $r' = r - y$, то у нас получается $r' \geqslant 0$. Следовательно,

$$x = (q \cdot y) + r = ((q + 1) \cdot y) + (r - y) = (q' \cdot y) + r',$$

и поэтому q', r' по-прежнему удовлетворяет инварианту цикла (1.2).

Теперь мы используем инвариант цикла, для того чтобы показать, что (если алгоритм завершается) постусловие алгоритма деления соблюдается, если соблюдается предусловие. В данном случае это очень просто, так как цикл заканчивается, когда больше не является истинным, что $y \leqslant r$, то есть когда истинно, что $r < y$. С другой стороны, (1.2) соблюдается после каждой итерации, и в особенности последней итерации. Соединяя (1.2) и $r < y$, мы получаем наше постусловие и, следовательно, частичную правильность.

Для того чтобы показать завершение, мы используем *принцип наименьшего числа* (least number principle, LNP). Нам нужно связать некую неотрицательную монотонную убывающую последовательность с алгоритмом; просто рассмотрим r_0, r_1, r_2, …, где $r_0 = x$, и r_i – это значение r после i-й итерации. Обратите внимание, что $r_{i+1} = r_i - y$. Во-первых, $r_i \geqslant 0$, потому что алгоритм входит в цикл while, только если $y \leqslant r$, а во-вторых, $r_{i+1} < r_i$, так как $y > 0$. По принципу наименьшего числа такая последовательность «не может продолжаться вечно» (в том смысле, что множество $\{r_i \mid i = 0, 1, 2, \ldots\}$ является подмножеством натуральных чисел и поэтому имеет наименьший элемент), поэтому алгоритм должен завершиться.

Таким образом, мы показали полную правильность алгоритма деления.

Задача 1.3. Каково время выполнения алгоритма 1.1? То есть сколько шагов требуется для его завершения? Будем считать, что присваивания (строки 1 и 2) и арифметические операции (строки 4 и 5), а также проверка «⩽» (строка 3) все выполняются за один шаг.

Задача 1.4. Предположим, что предусловие в алгоритме 1.1 изменено, скажем, на «$x \geqslant 0 \wedge y > 0 \wedge x, y \in \mathbb{Z}$», где $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$. По-прежнему ли корректен алгоритм в этом случае? Что делать, если он изменяется на следующее «$y > 0 \wedge x, y \in \mathbb{Z}$»? Как бы вы модифицировали этот алгоритм для работы с отрицательными значениями?

Задача 1.5. Напишите программу, которая принимает на входе x и y и выдает на выходе промежуточные значения q и r, и, наконец, частное и остаток от деления x на y.

1.1.3. Евклид

Пусть даны два положительных целых числа a, b, тогда их *наибольший общий делитель* (greatest common divisor), обозначаемый как $\gcd(a, b)$, есть наибольшее целое число, которое делит оба числа. Алгоритм Евклида, показанный как алгоритм 1.2, представляет собой процедуру нахождения наибольшего общего делителя двух чисел. Это один из старейших известных алгоритмов; он появился в «Элементах» Евклида (книга 7, пропозиции 1 и 2) около 300 года до нашей эры.

Обратите внимание, что для вычисления $\text{rem}(n, m)$ в строках 1 и 3 алгоритма Евклида нам необходимо в качестве подпрограммы использовать алгоритм 1.1 (алгоритм деления); это типичная «композиция» алгоритмов. Также обратите внимание, что строки 1 и 3 выполняются слева направо, поэтому, в частности, в строке 3 мы сначала выполняем $m \leftarrow n$, затем $n \leftarrow r$ и, наконец, $r \leftarrow \text{rem}(m, n)$. Это важно для правильной работы алгоритма, так как при выполнении $r \leftarrow \text{rem}(m, n)$ мы используем только что обновленные значения m, n.

Алгоритм 1.2. Евклид

Предусловие: $a > 0 \wedge b > 0 \wedge a, b \in \mathbb{Z}$
1: $m \leftarrow a$; $n \leftarrow b$; $r \leftarrow \text{rem}(m, n)$
2: **while** $(r > 0)$ **do**
3: $m \leftarrow n$; $n \leftarrow r$; $r \leftarrow \text{rem}(m, n)$
4: **end while**
5: **return** n
Постусловие: $n = \gcd(a, b)$

Для того чтобы доказать правильность алгоритма Евклида, мы покажем, что после каждой итерации цикла while соблюдается следующее логическое утверждение:

$$m > 0, n > 0 \text{ и } \gcd(m, n) = \gcd(a, b), \tag{1.3}$$

то есть (1.3) – это наш инвариант цикла. Мы доказываем это по индукции на числе итераций. Базовый случай: после нуля итераций (то есть непосредственно перед началом цикла while – после исполнения строки 1 и перед исполнением строки 2) мы имеем, что $m = a > 0$ и $n = b > 0$, поэтому (1.3) соблюдается тривиально. Обратите внимание, что $A > 0$ и $b > 0$ по предусловию.

На индукционном шаге будем считать, что $m, n > 0$ и $\gcd(a, b) = \gcd(m, n)$, и мы проходим по циклу еще раз, получая m', n'. Мы хотим показать, что $\gcd(m, n) = \gcd(m', n')$. Обратите внимание, что из строки 3 алгоритма мы видим, что $m' = n$, $n' = r = \text{rem}(m, n)$, поэтому, в частности, $m' = n > 0$ и $n' = r = \text{rem}(m, n) > 0$, так как если $r = \text{rem}(m, n)$ было бы равно нулю, то цикл завершился бы (а мы исходим из того, что проходим по циклу еще один раз). Поэтому достаточно доказать логическое утверждение в задаче 1.6.

Задача 1.6. Покажите, что для всех $m, n > 0$, $\gcd(m, n) = \gcd(n, \text{rem}(m, n))$.

Теперь правильность алгоритма Евклида следует из (1.3), так как алгоритм останавливается, когда $r = \text{rem}(m, n) = 0$, поэтому $m = q \cdot n$, и значит $\gcd(m, n) = n$.

Задача 1.7. Покажите, что алгоритм Евклида завершается, и установите его сложность в форме записи «O» большое.

Задача 1.8. Как бы вы сделали этот алгоритм эффективнее? Этот вопрос требует простых улучшений, которые снижают время работы на постоянный коэффициент.

Задача 1.9. Модифицируйте алгоритм Евклида так, чтобы на входе задавались целые числа m, n и на выходе получались целые числа a, b такие, что $am + bn = g = \gcd(m, n)$. Такая модификация называется *расширенным алгоритмом Евклида*. Следуйте этой схеме:

a) используйте принцип наименьшего числа, для того чтобы показать, что если $g = \gcd(m, n)$, то существуют a, b такие, что $am + bn = g$;
b) спроектируйте расширенный алгоритм Евклида и докажите его правильность;
c) обычный расширенный алгоритм Евклида имеет полином времени выполнения в $\min\{m, n\}$; покажите, что это время является временем выполнения вашего алгоритма, или измените свой алгоритм так, чтобы он работал за это время.

Задача 1.10. Напишите программу, которая реализует расширенный алгоритм Евклида. Затем выполните следующий эксперимент: выполните его на случайной подборке входных данных заданного размера для размеров, ограниченных некоторым параметром N; вычислите среднее число шагов алгоритма для каждого размера $n \leqslant N$ входных данных и используйте `gnuplot`[1] для построения графика результата. Как выглядит $f(n)$ – т. е. «среднее число шагов» расширенного алгоритма Евклида на размере n входных данных? Обратите внимание, что размер не совпадает со значением; входные данные размера n являются входами с двоичным представлением из n бит.

1.1.4. Палиндромы

Алгоритм 1.3 проверяет, является ли цепочка символов *палиндромом*, то есть словом, читаемым в обоих направлениях, слева направо и справа налево, например `madamimadam` или `racecar` (на русском: казак, ротатор).

Для того чтобы представить этот алгоритм, нам нужно ввести немного обозначений. Функции *floor* и *ceil* определяются, соответственно, следующим образом: $\lfloor x \rfloor = \max\{n \in \mathbb{Z} | n \leqslant x\}$ и $\lceil x \rceil = \min\{n \in \mathbb{Z} | n \geqslant x\}$, $\lfloor x \rceil$ означает «округление» x и определяется как $\lfloor x \rceil = \lfloor x + 1/2 \rfloor$.

Алгоритм 1.3. Палиндромы

Предусловие: $n \geqslant 1 \wedge A[0 \ldots n-1]$ является массивом символов
1: $i \leftarrow 0$
2: **while** $(i < \lfloor n/2 \rfloor)$ **do**

[1] Gnuplot – это вспомогательная консольная программа для построения графиков (http://www.gnuplot.info). Кроме того, в Python есть графопостроительная библиотека `matplotlib` (https://matplotlib.org).

```
 3:        if (A[i] ≠ A[n − i − 1]) then
 4:                return F
 5:        end if
 6:        i ← i + 1
 7: end while
 8: return T
```

Постусловие: вернуть T тогда и только тогда, когда A является палиндромом

Пусть инвариант цикла равен: после k-й итерации $i = k + 1$ и для всех j таких, что $1 \leqslant j \leqslant k$, $A[j] = A[n - j + 1]$. Докажем, что инвариант цикла соблюдается по индукции на k. Базовый случай: перед тем как состоятся любые итерации, то есть после нуля итераций, нет j таких, что $1 \leqslant j \leqslant 0$, поэтому вторая часть инварианта цикла (бессодержательно) истинна. Первая часть инварианта цикла соблюдается, так как i изначально имеет значение 1.

Индукционный шаг: мы знаем, что после k итераций $A[j] = A[n - j + 1]$ для всех $1 \leqslant j \leqslant k$; после еще одной итерации мы знаем, что $A[k+1] = A[n - (k+1) + 1]$, значит, данное формальное суждение вытекает для всех $1 \leqslant j \leqslant k + 1$. Этим доказывается инвариантность цикла.

Задача 1.11. С помощью инварианта цикла проаргументируйте частичную правильность алгоритма палиндромов. Покажите, что алгоритм завершается.

В Python легко манипулировать символьными цепочками (строками); сегмент символьной цепочки называется *срезом*. Рассмотрим слово `palindrome`; если мы установим переменную `s` равной этому слову:

```
s = 'palindrome'
```

тогда мы можем получить доступ к разным срезам следующим образом:

```
print s[0:5]      palin
print s[5:10]     drome
print s[5:]       drome
print s[2:8:2]    lnr
```

где форма записи `[i:j]` означает сегмент символьной цепочки, начинающийся с i-го символа (и мы всегда начинаем отсчет с нуля!) и вплоть до j-го символа, включая первый, но исключая последний. Форма записи `[i:]` означает от i-го символа вплоть до конца, а `[i:j:k]` означает от i-го символа вплоть до j-го (опять же, не считая сам j-й), беря каждый k-й символ.

Хорошим способом понять разделители символьной цепочки является запись индексов «между» символами, а также в начале и в конце. Например:

$_0\mathtt{p}_1\mathtt{a}_2\mathtt{l}_3\mathtt{i}_4\mathtt{n}_5\mathtt{d}_6\mathtt{r}_7\mathtt{o}_8\mathtt{m}_9\mathtt{e}_{10}$

и обратите внимание, что срез `[i:j]` содержит все символы между индексом i и индексом j.

Задача 1.12. Используя встроенные функциональные средства Python для манипуляции со срезами символьных цепочек, напишите краткую программу, которая проверяет, является ли данная символьная цепочка палиндромом.

1.1.5. Дальнейшие примеры

В этом разделе мы приводим ряд дальнейших примеров алгоритмов, которые принимают в качестве входных данных целые числа и манипулируют ими с помощью цикла while. Мы также приводим пример алгоритма, который очень легко описать, но для которого неизвестно доказательство завершения (алгоритм 1.6). Все они дополнительно подтверждают идею о том, что доказательства правильности являются не просто педантичными упражнениями в математическом формализме, а реальным свидетельством валидности того или иного алгоритмического решения.

Задача 1.13. Дайте алгоритм, который принимает на входе положительное целое число n и выводит на выходе «да», если $n = 2^k$ (то есть n – это степень числа 2) и «нет» в противном случае. Докажите, что ваш алгоритм правилен.

Задача 1.14. Что вычисляет алгоритм 1.4? Докажите свое утверждение.

Алгоритм 1.4. См. задачу 1.14

```
1: x ← m ; y ← n ; z ← 0
2: while (x ≠ 0) do
3:         if (rem(x, 2) = 1) then
4:                 z ← z + y
5:         end if
6:         x ← div(x, 2)
7:         y ← y·2
8: end while
9: return z
```

Задача 1.15. Что вычисляет алгоритм 1.5? Исходите из того, что a, b – это положительные целые числа (то есть исходите из того, что предусловием является, что a, $b > 0$). Для каких начальных a, b этот алгоритм завершается? За сколько шагов он завершается, если он действительно завершается?

Algorithm 1.5. См. задачу 1.15

```
1: while (a > 0) do
2:         if (a < b) then
3:                 (a, b) ← (2a, b − a)
4:         else
5:                 (a, b) ← (a − b, 2b)
6:         end if
7: end while
```

Рассмотрите приведеный ниже алгоритм 1.6.

Algorithm 1.6. Алгоритм Улама

Pre-condition: $a > 0$

```
x ← a
while последние три значения x не равны 4, 2, 1 do
```

```
        if x является четным then
                x ← x/2
        else
                x ← 3x + 1
        end if
end while
```

Этот алгоритм отличается от всех алгоритмов, которые мы видели до сих пор, тем, что у него нет известного доказательства завершения и, следовательно, нет известного доказательства правильности. Посмотрите, как это просто: для любого положительного целого числа a установить $x = a$ и повторять следующее: если x является четным, то разделить его на 2, а если нечетным, то умножить его на 3 и прибавить 1. Повторять это до тех пор, пока последние три полученных значения не будут равны 4, 2,1. Например, если $a = 22$, то можно проверить, что x принимает следующие значения: 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, **4**, **2**, **1**, и алгоритм 1.6 завершается. Предполагается, что независимо от начального значения a до тех пор, пока a является положительным целым числом, алгоритм 1.6 завершается. Эта гипотеза известна как «задача Улама»[1], и, несмотря на десятилетия работы, пока никто не смог эту задачу решить.

На самом деле, как показывает недавняя работа, было продемонстрировано, что варианты задачи Улама неразрешимы. Мы рассмотрим неразрешимость в главе 9, но в работе [Lehtonen (2008)] было показано, что для очень простого варианта задачи, где мы принимаем x равным $3x + t$ для x в отдельном множестве A_t (подробнее см. указанную статью), вообще нет никакого алгоритма, который решит, для какого начального a новый алгоритм завершается и для какого нет.

Задача 1.16. Напишите программу, которая принимает a в качестве входных данных и отображает все значения задачи Улама до тех пор, пока не увидит 4, 2,1, и в этот момент он останавливается. Вы только что написали почти тривиальную программу, для которой нет доказательств завершения. Теперь проделайте эксперимент: вычислите, сколько шагов требуется для того, чтобы достичь 4, 2,1 для всех $a < N$, для некоторого N. Есть ли какие-либо догадки?

1.2. Алгоритмы ранжирования

Алгоритмы, которые мы встречали до сих пор в книге, являются классическими, однако в некоторой степени они являются «игрушечными примерами». В этом разделе мы хотим продемонстрировать силу и полезность некоторых очень хорошо известных «взрослых» алгоритмов. Мы сосредоточимся на трех разных алгоритмах ранжирования. Ранжирование элементов, то есть ранговая градация, является исконной человеческой деятельностью, и мы кратко рассмотрим процедуры ранжирования, которые варьируются от древних, таких как процедура Раймунда Луллия, жившего в XIII веке мистика и философа, до старых, таких

[1] Она также называется «гипотезой Коллатца», «сиракузской задачей», «задачей Какутани» или «алгоритмом Хассе». Хотя следует признать, что роза с любым из этих названий будет пахнуть так же сладко, засилье имен показывает, что данная гипотеза представляет собой весьма заманчивую математическую задачу.

как работа маркиза де Кондорсе, обсуждаемая в разделе 1.2.3, до современного простого и элегантного алгоритма ранжирования страниц PageRank компании Google, обсуждаемого в следующем далее разделе.

1.2.1. Алгоритм PageRank

В 1945 году Ванневар Буш написал статью в Atlantic Monthly под названием «Как мы, возможно, думаем» (*As we may think*) [Bush (1945)], где он продемонстрировал жуткое предвидение идей, которые впоследствии стали Всемирной паутиной. В этой удивительной статье Буш указал на то, что информационно-поисковые системы организованы линейно (будь то книги, базы данных, компьютерная память и т. д.), но сознательный опыт человека демонстрирует то, что он назвал «ассоциативной памятью». То есть человеческий разум имеет семантическую сеть, где мы думаем об одном, и это напоминает нам о другом, и т. д. Буш предложил проект человекоподобной машины, «Memex», которая имела характеристики ur-паутины: оцифрованного человеческого знания, взаимосвязанного ассоциативными связями.

Когда в начале 1990-х Тим Бернерс-Ли наконец реализовал идеи Буша в виде HTML и внедрил Всемирную паутину, веб-страницы были статичными, а ссылки имели навигационную функцию. Сегодня ссылки часто вызывают сложные программы, написанные на Perl, PHP, MySQL и т. д., и в то время как некоторые из них по-прежнему остаются навигационными, многие являются транзакционными, реализуя такие действия, как «добавить в корзину» или «обновить мой календарь».

Поскольку в настоящее время существуют миллиарды активных веб-страниц, возникает вопрос, как выполнять в них поиск, для того чтобы находить соответствующую высококачественную информацию? Мы достигаем этого путем ранжирования тех страниц, которые соответствуют критериям поиска; страницы с хорошим рангом будут появляться вверху – благодаря этому результаты поиска будут иметь смысл для читателя-человека, который должен просмотреть только первые несколько результатов, чтобы (надо надеяться) найти то, чего он хочет. Эти верхние страницы называются *авторитетными страницами*.

Для того чтобы расположить авторитетные страницы рангом выше, мы используем тот факт, что веб состоит не только из страниц, но и из *гиперссылок*, которые соединяют эти страницы. Эта гиперссылочная структура (которая может быть естественным образом смоделирована ориентированным графом) содержит много скрытых аннотаций, которые могут быть использованы для автоматического выведения авторитетности. В этом заключается глубокое наблюдение: в конце концов, элементы, получающие от пользователя высокий ранг, ранжируются так субъективно; эксплуатация гиперссылочной структуры позволяет нам связывать субъективный опыт пользователей с выходными данными алгоритма!

Точнее говоря, создавая гиперссылку, автор выражает неявное одобрение странице. Добывая коллективное суждение, выраженное этими одобрениями, мы получаем картину качества (или субъективного восприятия качества) данной веб-страницы. Это очень похоже на наше восприятие качества научных цитат, когда важная публикация цитируется другими важными публикациями. Теперь возникает вопрос, как преобразовать эти идеи в алгоритм. Судьбоносный ответ был дан хорошо известным сегодня алгоритмом PageRank, авторами которого являются

С. Брин и Л. Пейдж, основатели Google – см. публикацию [Brin и Page (1998)]. Алгоритм PageRank глубоко анализирует гиперссылочную структуру в интернете, для того чтобы сделать вывод об относительной важности страниц.

Рассмотрим рис. 1.1, на котором изображена веб-страница X и все страницы T_1, T_2, T_3, ..., T_n, которые на нее указывают. С учетом страницы X пусть $C(X)$ равно числу несовпадающих ссылок, которые покидают X, то есть это расположенные в X ссылки, которые указывают на страницу за пределами X. Пусть $PR(X)$ равно рангу страницы X. Мы также привлекаем параметр D, который называем *коэффициентом затухания* и который мы объясним позже.

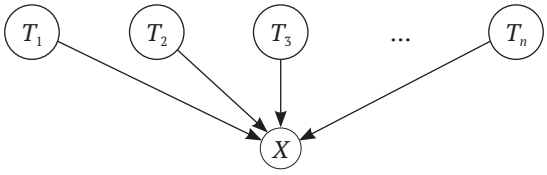


Рис. 1.1 ❖ Вычисление ранга страницы *A*

Тогда ранг страницы X можно вычислить следующим образом:

$$PR(X) = (1 - d) + d\left[\frac{PR(T_1)}{C(T_1)} + \frac{PR(T_2)}{C(T_2)} + \cdots + \frac{PR(T_n)}{C(T_n)}\right]. \quad (1.4)$$

Теперь поясним (1.4): коэффициент затухания d – это константа $0 \leqslant d \leqslant 1$ и обычно устанавливается равной ,85. Данная формула постулирует поведение «случайного серфера», который начинает нажимать ссылки на случайной странице, следуя по ссылке из этой страницы и нажимая ссылки (ни разу не нажимая кнопку «назад») до тех пор, пока случайному серферу не надоест и он не начнет этот процесс с самого начала, перейдя на случайную страницу. Таким образом, в (1.4) $(1 - d)$ является вероятностью случайного выбора X, тогда как $\frac{PR(T_i)}{C(T_i)}$ – вероятность достижения X в случае прихода из T_i, нормализованная по числу исходящих из T_i ссылок. Мы вносим небольшую корректировку в (1.4): нормализуем ее на размер паутины, N, то есть делим $(1 - d)$ на N. Благодаря этому вероятность наткнуться на X корректируется на совокупный размер паутины.

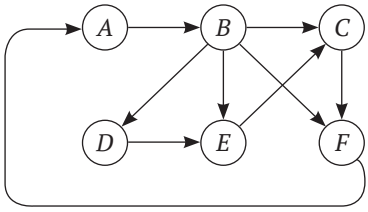
Задача с (1.4) заключается в том, что она выглядит зацикленной. Как изначально вычислить $PR(T_i)$? Алгоритм работает поэтапно, уточняя ранг каждой страницы на каждом этапе. Первоначально мы используем эталитарный подход и присваиваем каждой странице ранг $1/N$, где N – это общее число страниц в интернете. Затем пересчитываем все ранги страниц, используя (1.4) и начальные ранги страниц, и продолжаем. После каждого этапа $PR(X)$ приближается к фактическому значению, и по сути дело сходится довольно быстро. Здесь есть много технических вопросов, таких как знание того, когда остановиться, и обработка вычислений с участием N, которых может быть более триллиона, но выше приведен алгоритм PageRank в конспектном изложении.

Разумеется, интернет представляет собой обширную коллекцию разнородных документов, и (1.4) является слишком простой формулой, чтобы выразить ею абсолютно все, – поиск в Google намного сложнее. Например, не все исходящие

ссылки обрабатываются одинаково: ссылка более крупным шрифтом или выделенная тегом <STRONG> будет иметь больший вес. Документы различаются внутренне по языку, формату, такому как PDF, изображению, тексту, звуку, видео; и внешне с точки зрения репутации источника, частоты обновления, качества, популярности и других переменных, которые теперь учитываются современной поисковой системой. Для получения дополнительной информации об алгоритме PageRank читатель должен обратиться к публикации [Franceschet (2011)].

Более того, присутствие поисковых систем также влияет на интернет. Поскольку поисковые системы направляют трафик, они сами формируют рейтинг сети. Подобный эффект в физике известен как *эффект наблюдателя*, где приборы изменяют состояние того, что они наблюдают. В качестве простого примера рассмотрим замер давления в шинах: чтобы его измерить, вы должны выпустить немного воздуха и, следовательно, чуть изменить давление. Все эти увлекательные вопросы являются предметом анализа больших данных.

Задача 1.17. Рассмотрите следующую небольшую сеть:



Вычислите PageRank разных страниц в этой сети, используя (1.4) с коэффициентом затухания $d = 1$, то есть исходя из того, что вся навигация осуществляется путем следования по ссылкам (без случайных переходов на другие страницы).

Задача 1.18. Напишите программу, которая вычисляет ранги всех страниц в данной сети размера N. Пусть сеть задана матрицей 0–1, где 1 в позиции (i, j) означает, что существует ссылка со страницы i на страницу j. В противном случае в этой позиции находится 0. Используйте (1.4) для вычисления ранга страниц, начиная со значения $1/N$. Вы должны остановиться, когда все значения сойдутся, – всегда ли этот алгоритм завершается? Также отслеживайте все значения в виде дробей a/b, где $\gcd(a, b) = 1$; Python имеет удобную библиотеку для дробей: `import fractions`.

1.2.2. Стабильный брачный союз

Предположим, что мы хотим сопоставить стажеров с больницами или студентов с колледжами; обе задачи являются примерами задачи процесса приема на работу или учебу, и обе имеют решение, которое оптимизирует, в определенной степени, совокупное удовлетворение всех заинтересованных сторон. Решением этой задачи является элегантный алгоритм решения так называемой «*задачи о стабильном брачном союзе*», который используется с 1960-х годов для приема в колледж и для подбора интернов в больницы.

Экземпляр задачи устойчивого брачного союза размера n состоит из двух непересекающихся конечных множеств одинакового размера: множества юношей $B = \{b_1, b_2, \ldots, b_n\}$ и множества девушек $G = \{g_1, g_2, \ldots, g_n\}$. Пусть $<_i$ обозначает ранговую градацию, выполненную юношей b_i, то есть $g <_i g'$ означает, что юноша b_i предпо-

читает девушку g девушке g'. Схожим образом $<^j$ обозначает ранговую градацию, выполненную девушкой g_j. Каждый юноша b_i имеет такую ранговую градацию (линейное упорядочение) $<_i$ множества G, которая отражает его предпочтение в отношении девушек, на которых он хочет жениться. Схожим образом каждая девушка g_j имеет ранговую градацию (линейное упорядочение) $<^j$ множества B, которая отражает ее предпочтение в отношении юношей, за которых она хотела бы выйти замуж.

Паросочетание (или брак) M является взаимно-однозначным соответствием между B и G. Мы говорим, что b и g являются партнерами в M, если они сочетались в M, и пишем $p_M(b) = g$ и $p_M(g) = b$. Паросочетание M является *неустойчивым*, если существует пара (b, g) из $B \times G$ такая, что b и g не являются партнерами в M, и b предпочитает девушку $p_M(b)$ девушке g и g предпочитает юношу $p_M(g)$ юноше b. Говорят, что такая пара (b, g) *блокирует* паросочетание M и называется *блокирующей парой* для M (см. рис. 1.2). Паросочетание M стабильно, если оно не содержит блокирующих пар.

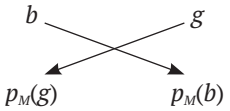


Рис. 1.2 ❖ Блокирующая пара: b и g предпочитают друг друга своим партнерам $p_M(b)$ и $p_M(g)$

Оказывается, всегда существует решение устойчивого брачного союза задачи паросочетания. Это решение может быть вычислено с помощью знаменитого алгоритма Гейла и Шепли ([Gale и Shapley (1962)]), который выдает стабильный брачный союз для любых B, G на входе независимо от ранговой градации[1].

Паросочетание M производится в несколько этапов M_s с целью того, чтобы b_t всегда имел партнера в конце этапа s, где $t \leqslant s$. Однако партнеры b_t не становятся лучше, то есть $p_{M_t}(b_t) \leqslant_t p_{M_{t+1}}(b_t) \leqslant_t \cdots$. С другой стороны, для каждого $g \in G$, если g имеет партнера на этапе t, g будет иметь партнера на каждом этапе $s \geqslant t$, и партнеры не станут хуже, то есть $p_{M_t}(g) \geqslant^t p_{M_{t+1}}(g) \geqslant^t \ldots$. Следовательно, с увеличением s партнеры b_t становятся менее предпочтительными, а партнеры g – более предпочтительными.

В конце этапа s будем считать, что мы произвели паросочетание

$$M_s = \{(b_1, g_{1,s}), \ldots, (b_s, g_{s,s})\},$$

где форма записи $g_{i,s}$ означает, что $g_{i,s}$ является партнершей юноши b_i после окончания этапа s.

Мы будем говорить, что партнеры в M_s *помолвлены*. Идея состоит в том, что на этапе $s + 1$ юноша b_{s+1} попытается получить партнершу, сделав предложение девушкам в G в его порядке предпочтения. Когда b_{s+1} делает предложение девушке g_j, g_j принимает его предложение, если g_j в настоящее время не помолвлена либо

[1] В 2012 году Нобелевская премия по экономике была присуждена Ллойду С. Шепли и Элвину Э. Роту «за теорию устойчивых размещений и практику рыночного проектирования», то есть за алгоритм стабильного брачного союза.

в настоящее время помолвлена с менее предпочтительным юношей b, то есть $b_{s+1} <^j b$. В случае, когда g_j предпочитает юношу b_{s+1} ее нынешнему партнеру b, то g_j разрывает помолвку с b, и b тогда приходится искать новую партнершу.

Задача 1.19. Покажите, что каждому b нужно сделать предложение не более одного раза каждой g.

Из задачи 1.19 мы видим, что каждый юноша может оставить в своем списке предпочтений закладку, и эта закладка движется только вперед. Когда приходит очередь юноши делать выбор, он начинает делать предложения с того места, где находится его закладка, и к тому времени, когда он закончит, его закладка будет двигаться только вперед. Обратите внимание, что на этапе $s + 1$ закладка каждого юноши не могла выйти за пределы номера девушки s в списке, не выбрав кого-то (после этапа s задействуется только s девушек). По мере того как юноши меняются очередями, закладка каждого юноши продвигается, поэтому закладка некоторого юноши (среди юношей в $\{b_1, \ldots, b_{s+1}\}$) в конце концов дойдет до точки, где он должен выбрать девушку.

Алгоритм 1.7. Гейл-Шепли

```
 1: Этап 1: b_1 выбирает свою лучшую g и M_1 ← {(b_1, g)}
 2: for s = 1, ..., s = |B| − 1, этап s + 1: do
 3:         M ← M_s
 4:         b* ← b_{s+1}
 5:         for b* делает предложение всем g в порядке предпочтения: do
 6:                 if g не была помолвлена: then
 7:                         M_{s+1} ← M ∪ {(b*, g)}
 8:                         закончить текущий этап
 9:                 else g была помолвлена с b, но g предпочитает b*: then
10:                         M ← (M − {(b, g)}) ∪ {(b*, g)}
11:                         b* ← b
12:                         повторить со строки 5
13:                 end if
14:         end for
15: end for
```

Обсуждение в предыдущем абзаце показывает, что этап $s + 1$ в алгоритме 1.7 должен завершиться. Озабоченность здесь вызывает то, что случай (ii) этапа $s + 1$ может оказаться зацикленным. Но тот факт, что закладки продвигаются, показывает, что это невозможно.

Более того, в данной процедуре это дает верхнюю границу $(s + 1)^2$ шагов на этапе $(s + 1)$. Это означает, что имеется n этапов, и каждый этап занимает $O(n^2)$ шагов, и, следовательно, алгоритм 1.7 занимает $O(n^3)$ шагов в целом. Вопрос, конечно, в том, что мы подразумеваем под шагом? Компьютеры работают на двоичных цепочках, но здесь принимается неявное допущение, что мы сравниваем числа и получаем доступ к спискам предпочтений за один шаг. Но цена этих операций незначительна при сравнении с нашим идеализированным временем выполнения, и поэтому мы позволяем себе, чтобы эта поэтическая лицензия ограничивала совокупное время выполнения.

Задача 1.20. Покажите, что существует ровно одна девушка, которая не помолвлена на этапе s, но помолвлена на этапе $(s + 1)$, и что для каждой девушки g_j, которая помолвлена в M_s, g_j будет помолвлена в M_{s+1} и $p_{M_{s+1}}(g_j) <^j p_{M_s}(g_j)$. (Следовательно, как только g_j станет помолвленной, она останется помолвленной, и ее партнеры только заработают в предпочтении по мере продвижения этапов.)

Задача 1.21. Предположим, что $|B| = |G| = n$. Покажите, что M_n будет стабильным брачным союзом в конце этапа n.

Мы говорим, что пара (b, g) *допустима*, если существует устойчивое паросочетание, в котором b, g являются партнерами. Мы говорим, что паросочетание является *оптимальным по юноше*, если каждый юноша имеет пару со своей допустимой партнершей с самым высоким рангом. Мы говорим, что паросочетание является *пессимистичным по юноше*, если каждый юноша имеет пару со своей допустимой партнершей с самым низким рангом. Схожим образом мы определяем *оптимальное/пессимистичное* паросочетание *по девушке*.

Задача 1.22. Покажите, что наша версия алгоритма производит стабильное паросочетание, оптимальное по юноше и пессимистичное по девушке. Означает ли это, что упорядочение юношей не имеет значения?

Задача 1.23. Реализуйте алгоритм 1.7.

1.2.3. Попарные сравнения

Фундаментальное приложение алгоритмических процедур состоит в выборе лучшего варианта из многих. Этот отбор требует процедуры ранжирования, которая им руководит, но с учетом сложности мира в информационную эпоху процедура ранжирования и отбора часто осуществляется на основе чрезвычайного количества критериев. Такой отбор может также потребовать от выборщика предоставить обоснование для отбора и убедить кого-то еще, что лучший вариант был все-таки выбран. Например, представьте сценарий, когда команда врачей должна решить, оперировать пациента или нет [Kakiashvili и соавт. (2012)], и насколько важно одновременно отобрать оптимальный курс действий и предоставить веское обоснование для финального отбора. Действительно, обоснование в данном случае может быть не менее важным, чем отбор наилучшего варианта.

Значительные усилия были посвящены исследованию ранжирования в поисковых системах [Easley и Kleinberg (2010)] в случае большого числа сильно разнородных элементов. С другой стороны, была проделана относительно небольшая работа по ранжированию небольших множеств из весьма схожих (однородных) элементов, дифференцированных по большому числу критериев. Сегодняшнее состояние дел состоит из целого ряда специфичных для конкретной области ситуативных процедур, которые сильно зависят от области применения: один подход в медицинской профессии [Kakiashvili и соавт. (2012)], другой в мире менеджмента [Koczkodaj и соавт. (2014)] и т. д.

Попарные сравнения имеют удивительно старую историю для метода, который в определенной мере известен не так широко. Древние начала часто приписывают мистику и философу XIII века Раймунду Луллию. В 2001 году была обнаружена рукопись Луллия под названием *Ars notandi*, *Ars eleccionis*, *Alia ars eleccionis* (см. [Hagele и Pukelsheim (2001); Faliszewski и соавт. (2010)]), где он обсуждал системы

голосования и описал прообраз метода попарных сравнений. Современное начало приписывается маркизу де Кондорсе (см. его работу [Condorcet (1785)], написанную за четыре года до Французской революции и за девять лет до того, как он потерял свою голову). Так же, как и Луллий, Кондорсе применил метод попарных сравнений для анализа результатов голосования. Почти полтора века спустя Терстоун [Thurstone (1927)] этот метод усовершенствовал и использовал психологический континуум со шкальными значениями в качестве медиан распределения суждений.

Можно сказать, что современный метод попарных сравнений начался с работы Саати в 1977 году [Saaty (1977)], который предложил конечную девятибалльную шкалу измерения. Более того, Саати внедрил процесс анализа иерархий (analytic hierarchy process, AHP), который представляет собой формальный метод получения порядков ранговой градации из числовых попарных сравнений. Процесс анализа иерархий широко используется во всем мире для принятия решений в области образования, промышленности, правительства и т. д. В публикации [Koczkodaj (1993)] была предложена меньшая пятибалльная шкала, менее мелкозернистая, чем у Саатив с его девятибалльной, но проще в использовании. Обратите внимание, что хотя процесс анализа иерархий является респектабельным инструментом для практического применения, он тем не менее рассматривается многими [Dyer (1990); Janicki (2011)] как ошибочная процедура, которая порождает произвольное ранжирование.

Пусть $X = \{x_1, x_2, \ldots, x_n\}$ равно конечному множеству ранжируемых объектов. Пусть a_{ij} выражает числовое предпочтение между x_i и x_j. Идея состоит в том, что a_{ij} оценивает, «насколько лучше» x_i по сравнению с x_j. Ясно, что для всех i, j, $a_{ij} > 0$ и $a_{ij} = 1/a_{ji}$. Интуиция подсказывает, что если $a_{ij} > 1$, то по этому фактору x_i предпочтительнее x_j. Так, например, дисплей Apple Retina имеет в четыре раза большее разрешение, чем дисплей Thunderbolt, и поэтому если x_i – это Retina, а x_2 – Thunderbolt, то мы можем сказать, что качество изображения x_i в четыре раза лучше, чем качество изображения x_2, и поэтому $a_{12} = 4$ и $a_{21} = 1/4$. Закрепление значений a_j часто осуществляется субъективно человеческими судьями. Пусть $A = [a_{ij}]$ равно *матрице попарных сравнений*, также именуемой *матрицей предпочтений*. Мы говорим, что матрица попарных сравнений *непротиворечива*, если для всех i, j, k мы имеем, что $a_{ij}a_{jk} = a_{ik}$. В противном случае она *противоречива*.

Теорема 1.24 (Саати). Матрица попарных сравнений a непротиворечива тогда и только тогда, когда существуют $w_1, w_2, \ldots, w_n$ такие, что $a_{ij} = w_i/w_j$.

Задача 1.25. Обратите внимание, что w_i, которые появляются в теореме 1.24, создают ранговую градацию, в которой x_j предпочтительнее x_i тогда и только тогда, когда $w_i < w_j$. Предположим, что A является непротиворечивой матрицей попарных сравнений. Как извлечь значения w_i из A?

На практике субъективные оценивания a_{ij} редко бывают непротиворечивыми, что создает ряд проблем ([Janicki и Zhai (2011)]), а именно: (i) как мы измеряем противоречивость и какой уровень приемлем? (ii) как мы устраняем противоречивости или понижаем их до приемлемого уровня? (iii) как мы выводим значения w_i, начиная с противоречивой ранговой градации A? iv) как мы обосновываем определенный метод устранения противоречий? Противоречивая матрица

имеет ценность в том, что степень противоречивости измеряет в некоторой мере степень субъективности судей. Но мы должны быть в состоянии ответить на изложенные в предыдущем пункте вопросы, прежде чем сможем конструктивным образом воспользоваться противоречивой матрицей.

Задача 1.26. В работе [Bozoki и Rapcsak (2008)] предлагается несколько методов измерения противоречий в матрице (см., в частности, табл. 1 на стр. 161 указанной статьи). Рассмотрите возможность осуществления некоторых из этих мер. Можете ли вы предложить способ устранения противоречий в матрице попарных сравнений?

1.3. Ответы к избранным задачам

Задача 1.1. $(\forall I \in \mathcal{I}_A)[\exists O(O = A(I)) \wedge (\alpha_A(I) \rightarrow \beta_A(A(I)))]$. Тем самым говорится о том, что для любых хорошо сформированных входных данных I есть выходные данные, то есть алгоритм A завершается. Это выражается через $\exists O(O = A(I))$. Кроме того, это выражение говорит, что если хорошо сформированные входные данные удовлетворяют предусловию, указанному как предпосылка $\alpha_A(I)$, то выходные данные удовлетворяют постусловию, формулируемому как консеквент $\beta_A(A(I))$.

Задача 1.2. Очевидно, что

$$an^2 + bn + c \geqslant an^2 - |b|n - |c| = n^2(a - |b|/n - |c|/n^2), \tag{1.5}$$

$|b|$ конечно, поэтому $\exists n_b \in \mathbb{N}$ такие, что $|b|/n_b \leqslant a/4$. Схожим образом $\exists n_c \in \mathbb{N}$ такое, что $|c|/n_c^2 \leqslant a/4$. Пусть $n_0 = \max\{n_b, n_c\}$. Для $n \geqslant n_0$ $a - |b|/n_0 - |c|/n_0^2 \geqslant a - a/4 - a/4 = a/2$. В сочетании с (1.5) это дает:

$$\frac{a}{2}n^2 \leqslant an^2 + bn + c$$

для всех $n \geqslant n_0$. Нам нужно только присвоить c_3 значение $a/2$, чтобы завершить доказательство, что $an^2 + bn + c \in \Omega(n^2)$.

Далее мы имеем дело с общим многочленом с положительным ведущим коэффициентом. Пусть

$$p(n) = \sum_{i=1}^{k} a_i n^i = n^k \sum_{i=1}^{k} a_i / n^{k-i},$$

где $a_k > 0$. Ясно, что $p(n) \leqslant n^k \sum_{i=1}^{k} |a_i|$ для всех $n \in \mathbb{N}$, поэтому $p(n) = O(n^k)$. Более того, каждое a_i является конечным, поэтому для каждого $i \in \mathbb{N}$ такого, что $0 \leqslant i \leqslant k - 1$, $\exists n_i$ такие, что $a_i/n^{k-i} \leqslant a_k/2k$ для всех $n \geqslant n_i$. Пусть n_0 равно максимуму из этих n_i. $p(n)$ можно переписать как $n^k(a_k + \sum_{i=0}^{k-1} a_i/n^{k-i})$, поэтому

$$p(n) \geqslant n^k(a_k - \sum_{i=0}^{k-1} a_i/n^{k-i}).$$

Мы показали, что для $n \geqslant n_0$, $\sum_{i=0}^{k-1} a_i/n^{k-i} \leqslant a_k - k(a_k/2k) = a_k/2$, поэтому пусть $c = a_k/2$. Для всех $n \geqslant n_0$ $p(n) \geqslant (a_k - a_k/2)n^k = cn^k$. Следовательно, $p(n) = \Omega(n^k)$.

Мы показали, что $p(n) \in O(n^k)$ и $p(n) \in \Omega(n^k)$, поэтому $p(n) = \Theta(n^k)$.

Задача 1.3. Цикл while начинается с $r = x$, а затем всякий раз y вычитается; он ограничен значением x (самый медленный случай, когда y =1). Всякий раз, когда

цикл while исполняется, он проверяет, что $y \leqslant r$, и пересчитывает r, q, и поэтому он сто́ит 3 шага. Добавив исходные два присваивания ($q \leftarrow 0$, $r \leftarrow x$), в общей сложности мы получим $3x + 2$ шагов. Обратите внимание, что мы исходим из того, что x, y представлены в двоичном формате (обычная кодировка) и что для кодирования x требуется $\log_2 x$ бит, и, значит, время работы составляет $3 \cdot 2^{\log_2 x} + 2$, то есть время работы экспоненциально длине входных данных! Это нежелательное время работы; если бы x было больши́м, скажем, 1000 бит, а y – малым, то этот алгоритм занял бы больше времени, чем время жизни Солнца (10 млрд лет). Существуют гораздо более быстрые алгоритмы деления, такие как метод Ньютона–Рафсона.

Задача 1.4. Исходное предусловие (при котором алгоритм является правильным) равно:

$$x \geqslant 0 \wedge y > 0 \wedge x, y \in \mathbb{N},$$

где $\mathbb{N} = \{0, 1, 2, \ldots\}$. И значит, в первом случае наша работа уже сделана за нас; любой член $\mathbb{Z}$, который $\geqslant 0$, также находится в $\mathbb{N}$ (и любой член $\mathbb{N}$ находится в $\mathbb{Z}$), поэтому эти предусловия эквивалентны. С учетом того, что этот алгоритм был правильным в исходном предусловии, он также является правильным в новом. Во втором случае он не является правильным: рассмотрим $x = -5$ и $y = 2$, поэтому изначально $r = -5$, и цикл не будет выполняться, и $r \geqslant 0$ в постусловии не будет истинным.

Задача 1.6. Сначала заметим, что если u делит x и y, то для любого $a, b \in \mathbb{Z}$ u также делит $ax + by$. Следовательно, если $i|m$ и $i|n$, то

$$i|(m - qn) = r = \text{rem}(m, n).$$

Поэтому i делит оба числа: n и $\text{rem}(m, n)$, и поэтому i должно быть ограничено по их наибольшему общему делителю, то есть $i \leqslant \gcd(n, \text{rem}(m, n))$. Так как это верно для всех i, это, в частности, верно для $i = \gcd(m, n)$; следовательно, $\gcd(m, n) \leqslant \gcd(n, \text{rem}(m, n))$. И наоборот, предположим, что $i|n$ и $i|\text{rem}(m, n)$. Тогда $i|m = qn + r$, поэтому $i \leqslant \gcd(m, n)$, и опять же, $\gcd(n, \text{rem}(m, n))$ удовлетворяет условию существования такого i, поэтому мы имеем $\gcd(n, \text{rem}(m, n)) \leqslant \gcd(m, n)$. Оба неравенства, взятых вместе, дают нам $\gcd(m, n) = \gcd(n, \text{rem}(m, n))$.

Задача 1.7. Пусть r_i равно r после i-й итерации цикла. Обратите внимание, что $r_0 = \text{rem}(m, n) = \text{rem}(a, b) \geqslant 0$, и фактически каждый $r_i \geqslant 0$ по определению остатка. Более того:

$$\begin{aligned} r_{i+1} &= \text{rem}(m_{i+1}, n_{i+1}) \\ &= \text{rem}(n_i, r_i) \\ &= \text{rem}(n_i, \text{rem}(m_i, n_i)) \\ &= \text{rem}(n_i, r_i) < r_i. \end{aligned}$$

И значит, мы имеем убывающую, но неотрицательную последовательность чисел; по принципу наименьшего числа он должен завершиться. Для того чтобы установить сложность, мы подсчитываем число итераций цикла while, игнорируя перестановки значений местами (поэтому, чтобы получить фактическое число итераций, мы должны умножить результат на два).

Предположим, что $m = qn + r$. Если $q \geqslant 2$, то $m \geqslant 2n$, а так как $m \leftarrow n$, то m уменьшается как минимум наполовину. Если $q = 1$, тогда $m = n + r$, где $0 < r < n$, и исследуем два случая: $r \leqslant n/2$, поэтому n уменьшается, по меньшей мере, наполовину так, как

$n \leftarrow r$, либо $r > n/2$, и в этом случае $m = n + r > n + n/2 = 3/2n$, поэтому так как $m \leftarrow n$, m уменьшается на 1/3. Следовательно, можно сказать, что во всех случаях хотя бы один элемент в паре уменьшается, по крайней мере, на 1/3, и поэтому можно сказать, что время работы ограничено значением k таким, что $3^k = m \cdot n$, и, следовательно, сложность равна $O(\log(m \cdot n)) = O(\log m + \log n)$. Поскольку входные данные считаются двоичными, из этого можно сделать вывод, что время работы линейно по размеру входных данных.

Более жесткий анализ, известный как теорема Ламе, можно найти в публикации [Cormen и соавт. (2009)] (теорема 31.11), в которой говорится, что для любого целого числа $k \geqslant 1$, если $a > b \geqslant 1$ и $b < F_{k+1}$, где F_i – это i-е число Фибоначчи (см. задачу 9.5), то для выполнения алгоритма Евклида требуется менее k итераций цикла while (не считая перестановок значений местами).

Задача 1.8. Когда $m < n$, тогда $\text{rem}(m, n) = m$, и поэтому $m' = n$ и $n' = m$. Следовательно, при $m < n$ мы исполняем одну итерацию цикла, только для того чтобы поменять местами m и n. Для того чтобы быть эффективнее, мы могли бы добавить строку 2.5 в алгоритме 1.2 следующего содержания: **if** $(m < n)$ **then** $\text{swap}(m, n)$.

Задача 1.9. (a) Мы покажем, что если $d = \gcd(a, b)$, то существуют u, v такие, что $au + bv = d$. Пусть $S = \{ax + by | ax + by > 0\}$; явно $S \neq \emptyset$. По принципу наименьшего числа существует наименьшее $g \in S$. Мы покажем, что $g = d$. Пусть $a = q \cdot g + r$, $0 \leqslant r < g$. Предположим, что $r > 0$; тогда

$$r = a - q \cdot g = a - q(ax_0 + by_0) = a(1 - qx_0) + b(-qy_0).$$

Таким образом, $r \in S$, но $r < G$ – противоречие. Поэтому $r = 0$, и, значит, $g | a$, и подобный аргумент показывает, что $g | b$. Остается показать, что g больше, чем любой другой общий делитель a, b. Предположим, $c | a$ и $c | b$, поэтому $c | (ax_0 + by_0)$, и поэтому $c | g$, а это означает, что $c \leqslant g$. Следовательно, $g = \gcd(a, b) = d$.

(b) Расширенный алгоритм Евклида – это алгоритм 1.8. Обратите внимание, что в данном алгоритме присваивания в строке 1 и строке 8 вычисляются слева направо.

Алгоритм 1.8. Расширенный алгоритм Евклида

Предусловие: $m > 0$, $n > 0$

```
 1: a ← 0; x ← 1; b ← 1; y ← 0; c ← m; d ← n
 2: loop
 3:         q ← div(c, d)
 4:         r ← rem(c, d)
 5:         if r = 0 then
 6:                 stop
 7:         end if
 8:         c ← d; d ← r; t ← x; x ← a; a ← t − qa; t ← y; y ← b; b ← t − qb
 9: end loop
```

Постусловие: $am + bn = d = \gcd(m, n)$

Мы можем доказать правильность алгоритма 1.8, используя следующий инвариант цикла, состоящий из четырех логических утверждений:

$$am + bn = d, \quad xm + yn = c, \quad d > 0, \quad \gcd(c, d) = \gcd(m, n). \tag{LI}$$

Базовый случай:

$am + bn = 0 \cdot m + 1 \cdot n = n = d$;
$xm + yn = 1 \cdot m + 0 \cdot n = m = c$;

оба утверждения по строке 1. Тогда $d = n > 0$ по предусловию, и $\gcd(c, d) = \gcd(m, n)$ по строке 1. На индукционном шаге допустим, что переменные с индексом «штрих» являются результатом еще одной полной итерации цикла на переменных без индекса «штрих»:

$$\begin{aligned} a'm + b'n &= (x - qa)m + (y - qb)n && \text{по строке 8} \\ &= (xm - yn) - q(am + bn) \\ &= c - qd && \text{по индукционной гипотезе} \\ &= r && \text{по строкам 3 и 4} \\ &= d'. && \text{по строке 8} \end{aligned}$$

Тогда $x'm = y'n = am + bn = d = c'$, где первое равенство является по строке 8, второе по индукционной гипотезе, а третье по строке 8. Кроме того, $d' = r$ по строке 8, и алгоритм остановится в строке 5, если $r = 0$; с другой стороны, из строки 4 $r = \text{rem}(c, d) \geqslant 0$, поэтому $r > 0$ и, значит, $d' > 0$. В заключение

$$\begin{aligned} \gcd(c', d') &= \gcd(d, r) && \text{по строке 8} \\ &= \gcd(d, \text{rem}(c, d)) && \text{по строке 4} \\ &= \gcd(c, d) && \text{см. задачу 1.6} \\ &= \gcd(m, n). && \text{по индукционной гипотезе} \end{aligned}$$

Для частичной правильности достаточно показать, что если алгоритм завершается, то постусловие соблюдается. Если алгоритм завершается, то $r = 0$, поэтому $\text{rem}(c, d) = 0$ и $\gcd(c, d) = \gcd(d, 0) = d$. С другой стороны, по (LI), мы имеем, что $am + bn = d$, поэтому $am + bn = d = \gcd(c, d)$ и $\gcd(c, d) = \gcd(m, n)$.

(c) На стр. 292–293 в публикации [Delfs и Knebl (2007)] есть хороший анализ версии данного алгоритма. Авторы ограничивают время выполнения в терминах чисел Фибоначчи и получают желаемую границу времени выполнения.

Задача 1.11. Для частичной правильности алгоритма 1.3 мы покажем, что если предусловие соблюдается и если алгоритм завершается, то постусловие будет соблюдено. Итак, примем предусловие и сначала предположим, что A не является палиндромом. Тогда существует наименьшее i_0 (существует одно, и поэтому по принципу наименьшего числа существует наименьшее) такое, что $A[i_0] \neq A[n - i_0 + 1]$, и поэтому после первой $i_0 - 1$ итерации цикла while мы знаем из инварианта цикла, что $i = (i_0 - 1) + 1 = i_0$, и поэтому строка 4 исполняется, и алгоритм возвращает F (False). Следовательно, «A не является палиндромом» ⇒ «вернуть F».

Предположим теперь, что A является палиндромом. Тогда строка 4 ни разу не исполняется (так как такого i_0 не существует), и поэтому после $k = \lfloor \frac{n}{2} \rfloor$-й итерации цикла while мы знаем из инварианта цикла, что $i = \lfloor \frac{n}{2} \rfloor + 1$, и поэтому цикл while больше не исполняется, алгоритм переходит к строке 8 и возвращает T (True). Следовательно, «A является палиндромом» ⇒ «вернуть T».

Следовательно, постусловие «вернуть T тогда и только тогда, когда A является палиндромом» соблюдается. Обратите внимание, что мы использовали только часть инварианта цикла, то есть мы использовали тот факт, что после k-й итерации $i = k + 1$; по-прежнему соблюдается, что после k-й итерации для $1 \leqslant j \leqslant k$ $A[j] = A[n - j + 1]$, но в приведенном выше доказательстве нам этот факт не нужен.

Для того чтобы показать, что алгоритм завершается, пусть $d_i = \lfloor \frac{n}{2} \rfloor - i$. По предусловию мы знаем, что $n \geqslant 1$. Последовательность $d_1, d_2, d_3, \ldots$ является убывающей последовательностью натуральных чисел (потому что $i \leqslant \lfloor \frac{n}{2} \rfloor$), Поэтому по принципу наименьшего числа она конечна, и поэтому цикл завершается.

Задача 1.12. Она очень легкая, как только вы поймете, что в Python срез `[::-1]` генерирует обратную цепочку. Таким образом, чтобы проверить, является ли цепочка `s` палиндромом, нам нужно только написать `s == s[::-1]`.

Задача 1.13. Решение дано алгоритмом 1.9.

Алгоритм 1.9. Степени числа 2

Предусловие: $n \geqslant 1$
$x \leftarrow n$
while $(x > 1)$ **do**
    if $(2|x)$ **then**
        $x \leftarrow x/2$
    else
        остановиться и вернуть «нет»
    end if
end while
return «да»
Постусловие: «да» $\Leftrightarrow$ n является степенью числа 2

Пусть инвариант цикла равен «x является степенью числа 2 тогда и только тогда, когда n является степенью числа 2».

Покажем инвариант цикла по индукции на числе итераций главного цикла. Базовый случай: ноль итераций, и поскольку $x \leftarrow n$, $x = n$, поэтому очевидно, что x является степенью числа 2 тогда и только тогда, когда n является степенью числа 2. На индукционном шаге обратите внимание, что если нам когда-нибудь придется обновлять x, у нас есть $x' = x/2$, и, очевидно, x' является степенью числа 2 тогда и только тогда, когда x является степенью числа 2. Обратите внимание, что алгоритм всегда завершается (принять $x_0 = n$ и $x_{i+1} = x_i/2$ и, как обычно, применить принцип наименьшего числа).

Теперь можно доказать правильность: если алгоритмы возвращают «да», то по завершении последней итерации цикла $x = 1 = 2^0$, и по инварианту цикла n является степенью числа 2. Если, с другой стороны, n является степенью числа 2, то таково и каждое x, поэтому в конечном итоге $x = 1$, и поэтому алгоритм возвращает «да».

Задача 1.14. Алгоритм 1.4 вычисляет произведение m и n, то есть возвращаемое $z = m \cdot n$. Хорошим инвариантом цикла является $x \cdot y + z = m \cdot n$.

Задача 1.17. Мы начинаем с инициализации всех узлов рангом 1/6, а затем повторно применяем следующие ниже формулы, основанные на (1.4):

$\mathrm{PR}(A) = \mathrm{PR}(F)$
$\mathrm{PR}(B) = \mathrm{PR}(A)$
$\mathrm{PR}(C) = \mathrm{PR}(B)/4 + \mathrm{PR}(E)$
$\mathrm{PR}(D) = \mathrm{PR}(B)/4$
$\mathrm{PR}(E) = \mathrm{PR}(B)/4 + \mathrm{PR}(D)$
$\mathrm{PR}(F) = \mathrm{PR}(B)/4 + \mathrm{PR}(C)$

Результат приведен на рис. 1.3.

	0	1	2	3	4	5	6	...	17
A	0,17	0,17	0,21	0,25	0,29	0,18	0,20		0,22
B	0,17	0,17	0,17	0,21	0,25	0,29	0,18		0,22
C	0,17	0,21	0,25	0,13	0,14	0,16	0,19	...	0,17
D	0,17	0,04	0,04	0,04	0,05	0,05	0,07		0,06
E	0,17	0,21	0,08	0,08	0,09	0,11	0,14		0,11
F	0,17	0,21	0,25	0,29	0,18	0,20	0,23		0,22
Всего	1,00	1,00	1,00	1,00	1,00	1,00	1,00	...	1,00

Рис. 1.3 ❖ Схождение алгоритма Pagerank в задаче 1.17. Отметим, что данная таблица получена с помощью электронной таблицы: все значения округляются до двух десятичных знаков, но столбец 1 получается путем размещения 1/6 в каждой строке, столбец 2 получается из столбца 1 с формулами, а все остальные столбцы получаются путем «перетаскивания» столбца 2 до самого конца. Значения сошлись (более или менее) в столбце 17

Задача 1.19. После того как b сделал предложение g в первый раз, независимо от того, было оно успешным или нет, партнеры g могут стать только лучше. Следовательно, для b нет необходимости пробовать еще раз.

Задача 1.20. b_{s+1} делает предложение девушкам в соответствии с его списком предпочтений; в итоге некая g его принимает, и если g, которая приняла предложение b_{s+1}, была свободна, то она является новой партнершей. В противном случае некоторый $b^* \in \{b_1, \ldots, b_s\}$ стал свободным от помолвки, и мы повторяем тот же аргумент. Девушка g разрывает помолвку, только если предложение делает более подходящий юноша b, поэтому верно, что $p_{M_{s+1}}(g_j) <^j p_{M_s}(g_j)$.

Задача 1.21. Предположим, что у нас есть блокирующая пара $\{b, g\}$ (имея в виду, что $\{(b, g'), (b', g)\} \subseteq M_n$, но юноша b предпочитает девушку g девушке g', и девушка g предпочитает юношу b юноше b'). b появился либо после b', либо раньше. Если бы b появился перед b', то g была бы помолвлена с b или с кем-то еще лучше, когда появился b', поэтому g не стала бы помолвленной с b'. С другой стороны, поскольку (b', g) является парой, девушке g не было сделано ни одного лучшего предложения после предложения от юноши b', поэтому b не мог появиться после b'. В любом случае, мы получаем невозможность, и поэтому нет блокирующей пары $\{b, g\}$.

Задача 1.22. Для того чтобы показать, что паросочетание является оптимальным по юноше, мы рассуждаем от противного. Пусть выражение «g является оптимальной партнершей для b» означает, что среди всех стабильных паросочетаний g является лучшей партнершей, которую может получить b.

Мы выполняем алгоритм Гейла-Шепли, и пусть b равно первому юноше, отклоненному его оптимальной партнершей g. Это означает, что g уже находится в паре с некоторым b', и g предпочитает юношу b' юноше b. Более того, g желательна для b', по крайней мере, так же, как его собственная оптимальная партнерша (поскольку предложение b делается впервые во время выполнения алгоритма, когда юноша отклоняется его оптимальной партершей). Поскольку g является оптимальной для b, мы знаем (по определению), что существует некоторое устойчивое паросочетание S, где (b, g) является парой. С другой стороны, у b' оптимальная партнерша ранжируется (разумеется, самим же b'), по крайней мере, так высоко,

как g, и поскольку g занята b, с кем бы ни был b' в паре в S, скажем, g', b' предпочитает девушку g девушке g'. Это дает нам нестабильную пару, потому что $\{b', g\}$ предпочитают друг друга партнерам, которые у них есть в S.

Разумеется, это означает, что упорядоченность юношей не имеет значения, потому что есть уникальное оптимальное по юноше паросочетание, и оно не зависит от упорядоченности юношей.

Для того чтобы показать, что алгоритм Гейла-Шепли является пессимистичным по девушке, мы используем тот факт, что он оптимальный по юноше (что только что показали). Опять же, мы рассуждаем от противного. Предположим, что существует устойчивое паросочетание S, где g находится в паре с b, и g предпочитает юношу b' юноше b, где (b', g) является результатом алгоритма Гейла-Шепли. Согласно оптимальности по юноше, мы знаем, что в S у нас есть (b', g'), где g' не выше в списке предпочтений b', чем g, и поскольку g уже находится в паре с b, мы знаем, что g' находится на самом деле ниже. Это говорит о том, что S нестабильна, так как $\{b', g\}$ скорее будут вместе, чем со своими партнерами.

1.4. Примечания

Эта книга посвящена доказательству разных аспектов алгоритмов, их правильности, их завершению, их времени выполнения и т. д. Искусство математических доказательств поддается освоению с трудом; очень хорошим местом для начала является книга [Velleman (2006)].

В предисловии мы упомянули про отключение электричества на северо-востоке Америки в 2003 году. В то время автор жил в Торонто, Канада, на 14-м этаже жилого дома (который на самом деле был 13-м этажом, но поскольку номер 13 в лифтах Торонто был запрещен, после 12-го этажа следующей кнопкой на лифте была 14). После первых 24 часов аварийные генераторы вышли из строя, и нам всем пришлось подниматься по лестнице на наши этажи; мы покидали здание и вычищали район от пищи и воды, но так как в большинстве мест не было холодильника, было нелегко найти свежие продукты. Короче говоря, мы действительно чувствовали последствия этой алгоритмической ошибки.

В сноске к задаче 1.10 упоминается библиотека Python `matplotlib`. Ниже приведен простой пример построения графика функций $f(x) = x^3$ и $h(x) = -x^3$ над интервалом [0, 10] с использованием этой библиотеки:

```
import matplotlib.pyplot as plt
import numpy as np

def f(x):
    return x**3
def h(x):
    return -x**3

Input = np.arange(0,10.1,.5)
Outputf = [f(x) for x in Input]
Outputh = [h(x) for x in Input]

plt.plot(Input,Outputf,'r.',label='f - метка')
plt.plot(Input,Outputh,'b--',label='h - метка')
plt.xlabel('Это метка оси X')
plt.ylabel('Это метка оси Y')
```

```
plt.suptitle('Это заголовок')
plt.legend()
plt.show()
```

Разумеется, matplotlib имеет целый ряд функциональных возможностей; обратитесь к документации для получения более сложных примеров.

Палиндром `madamimadam` родом из романа «Улисс» Джойса. Мы обсуждали возможности манипулирования символьными цепочками в Python в разделе о палиндромах, раздел 1.1.4, но, возможно, самым мощным языком для манипулирования символьными цепочками является Perl. Например, предположим, что у нас есть текст, содержащий хештеги, которые являются словами символов, начинающихся с #, и мы хотим собрать все эти хештеги в массив. Можно испытать дрожь от перспективы реализации этой задачи, скажем, на языке программирования C, но в Perl это может быть достигнуто в одной строке:

```
@TAGS = ($TEXT =~ m/\#([a-zA-Z0-9]+)/g);
```

где `$TEXT` содержит текст с нулевым или большим числом хештегов, а массив `@TAGS` будет списком всех хештегов, которые встречаются в `$TEXT` без префикса #. Для получения большего удовольствия от Perl см. книгу [Schwartz и соавт. (2011)].

Поисковые системы представляют собой сложные и обширные программные системы, и ранжирование страниц является не единственной технической задачей, которая должна быть решена. Например, разбор ключевых слов для отбора релевантных страниц (страниц, содержащих ключевые слова), прежде чем на этих страницах будет составлена та или иная ранговая градация, также является сложной задачей: поисковая система должна решить ряд задач, таких как *синонимия* (несколько способов сказать одно и то же) и *полисемия* (несколько значений) и многие другие. См. публикацию [Miller (1995)].

Раздел 1.2.2 основан на § 2 книги [Cenzer и Remmel (2001)]. Еще одно изложение задачи стабильного брачного союза см. в главе 1 [Kleinberg и Tardos (2006)]. Ссылка на Маркиза де Кондорсе в первом предложении раздела 1.2.2 взята из докторской диссертации Юнь Чжая ([Zhai (2010)]), написанной под руководством Рышарда Яницкого. В этой докторской работе Юнь Чжай ссылается на работу [Arrow (1951)] как источник замечания относительно ранних попыток Маркиза де Кондорсе выполнить попарное ранжирование. Существует удивительно острое описание Кондорсе и его идей в «Судьбах постоянства» (Fortunes of Permanence) Роджера Кимбалла [Kimball (2012)], стр. 237–244. Кондерсе дал нам метод попарных сравнений, но он был трагической фигурой эпохи Просвещения: он обещал «абсолютное совершенствование человеческой расы» («perfectionnement même de l'espèce humaine»), но его утопические идеи стали предтечей бесчисленных писак, которые настаивали на совершенствовании человека, хочет он того или нет, открывших ящик Пандоры для неизбежных тиранических бесчинств, которые являются результатом утопических мечтаний.

Профессор Томас Л. Саати (теорема 1.24) умер 14 августа 2017 года. Он был выдающимся профессором Школы бизнеса Питтсбургского университета Каца. Правительство Польши присудило профессору Саати национальную награду, после того как примененение его теории процесса анализа иерархий для принятия решений привело к тому, что страна изначально не присоединилась к Европейскому союзу.

Глава 2

Жадный алгоритм

> Быть может, вам будет полезно поразмыслить над тем, что люди на свете в алчности и коварстве не умеют остановиться вовремя и всегда перехватывают через край.
>
> *Д. Копперфилд, [Dickens (1850)]*

Жадные алгоритмы – это алгоритмы, склонные добиваться сиюминутного вознаграждения. Они делают выбор, который является *локально оптимальным*, надеясь, что в конце они придут к *глобальному оптимуму*. Примером жадной процедуры является отсчитывание сдачи продавщицой в круглосуточном мини-маркете. Для того чтобы использовать как можно меньше монет, продавщица как можно дольше выдает монеты самого высокого номинала, переходя к следующему более низкому номиналу, когда это уже невозможно, и повторяет все сначала.

Жадность – это простая стратегия, которая хорошо работает с некоторыми вычислительными задачами, но оказывается безуспешной с другими. В случае отсчитывания наличных, если у нас есть монеты номиналом 1, 5, 25, жадная процедура всегда производит наименьшее возможное число монет, но то же самое не верно для 1, 10, 25. Попробуйте выдать 30, которые жадно составят 25, 1, 1, 1, 1, 1, тогда как оптимальным будет 10, 10, 10.

2.1. Остовные деревья минимальной стоимости

Мы представляем конечные графы с помощью *матриц смежности*. Для ориентированного или неориентированного графа $G = (V, E)$ его матрицей смежности является матрица A_G размера $n \times n$, где $n = |V|$ такое, что элемент (i, j) равен 1, если (i, j) является ребром в G, и 0 в противном случае.

Сама матрица смежности может быть легко закодирована в виде цепочки символов над $\{0, 1\}$. То есть с учетом A_G размера $n \times n$ пусть $s_G \in \{0, 1\}^{n^2}$, где s_G – это просто конкатенация строк A_G. Мы можем проверить непосредственно из s_G, является ли (i, j) ребром, проверив, содержит ли позиция $(i - 1)n + j$ в s_G единицу.

Неориентированный граф G представлен парой (B, E), где V – это множество вершин, или узлов, и $E \subseteq V \times V$ и $(u, v) \in E$ тогда и только тогда, когда $(v, u) \in E$ и $(u, u) \notin E$. *Степенью* вершины v является число ребер, которые касаются v. *Путем* в G между v_1 и v_k является последовательность $v_1, v_2, \ldots, v_k$ такая, что каждая $(v_i, v_{i+1}) \in E$. G является *связным*, если между каждой парой несовпадающих узлов есть путь. *Циклом* является замкнутый путь $v_1, \ldots, v_k, v_1$, причем все $v_1, \ldots, v_k$ не совпадают и $k \geqslant 3$. Граф

является *ациклическим*, если у него нет циклов. Дерево, по определению, является связным ациклическим графом. *Остовное* дерево связного графа G – это подмножество $T \subseteq E$ ребер такое, что (V, T) является деревом. Другими словами, ребра в T должны связывать все узлы G и не содержать циклов.

Если G имеет цикл, то для G существует более одного остовного дерева, и в общем случае G может иметь много остовных деревьев, но каждое остовное дерево имеет одинаковое число ребер.

Лемма 2.1. Каждое дерево с n вершинами имеет ровно $n - 1$ ребер.

Задача 2.2. Докажите лемму 2.1. (Подсказка: сначала покажите, что у каждого дерева есть лист, то есть узел первой степени. Затем покажите лемму по индукции на n.)

Лемма 2.3. Граф с n вершинами и более $n - 1$ ребер должен содержать хотя бы один цикл.

Задача 2.4. Докажите лемму 2.3.

Из лемм 2.1 и 2.3 следует, что если граф является деревом, то есть ациклическим и связным, то он должен иметь $(n - 1)$ ребер. Если у него *нет* $(n - 1)$ ребер, то он либо неациклический, либо несвязный. Если у него меньше $(n - 1)$ ребер, то он, конечно, несвязный, и если у него больше $(n - 1)$ ребер, он, конечно, неациклический.

Ребрам в графе вполне естественно назначают стоимости, так как ребра в целом могут представлять расстояния, пропускную способность или в общем случае затраты на переход из A в B. Пусть $c(e)$ обозначает стоимость ребра e, где $c(e)$ – это неотрицательное вещественное число. Общая стоимость графа G, $c(G)$, представляет собой сумму стоимостей всех ребер в G. Будем говорить, что T есть *остовное дерево минимальной стоимости* для G, если T является остовным деревом для G, и при наличии любого остовного дерева T' для G $c(T) \leqslant c(T')$.

Для графа $G = (V, E)$ и функции стоимости c, связанной с ребрами в E, мы хотим найти остовное дерево минимальной стоимости. Оказывается, что по счастливой случайности с этим справляется очевидный жадный алгоритм под названием *алгоритма Краскала*. Данный алгоритм состоит в следующем: отсортировать ребра в неубывающем порядке стоимостей так, чтобы $c(e_1) \leqslant c(e_2) \leqslant \ldots \leqslant c(e_m)$, и добавлять ребра по одному за раз, за исключением случаев, когда включение ребра образует цикл с уже добавленными ребрами.

Алгоритм 2.1. Краскал

```
1: Отсортировать ребра: c(e_1) ⩽ c(e_2) ⩽ ... ⩽ c(e_m)
2: T ← ∅
3: for i : 1..m do
4:         if T ∪ {e_i} не имеет цикла
5:                 T ← T ∪ {e_i}
6:         end if
7: end for
```

Но как проверить существование цикла, то есть исполнить строку 4 в алгоритме 2.1? В конце каждой итерации цикла for множество T ребер делит вершины V

на коллекцию $V_1, \ldots, V_k$ *связных компонент*. То есть V является непересекающимся объединением $V_1, \ldots, V_k$, каждый V_i образует связный граф, используя ребра из T, и никакое ребро в T не связывает V_i и V_j, если $i \neq j$. Простой способ отследить $V_1, \ldots, V_k$ – применить массив $D[i]$, где $D[i] = j$, если вершина $i \in V_j$. Инициализировать D, установив $D[i] \leftarrow i$ для каждого $i = 1, 2, \ldots, n$.

Для того чтобы проверить, образует ли $e_i = (r, s)$ цикл в пределах T, достаточно проверить $D[r] = D[s]$. Если e_i не образует цикла в пределах T, то мы обновляем: $T \leftarrow T \cup \{(r, s)\}$, и мы совмещаем компоненту $D[r]$ с $D[s]$, как показано в алгоритме 2.2.

Алгоритм 2.2. Слияние компонент

```
k ← D[r]
l ← D[s]
for j : 1..n do
        if D[j] = l then
                D[j] ← k
        end if
end for
```

Задача 2.5. С учетом того, что ребра могут быть упорядочены за m^2 шагов, например с помощью сортировки вставками, каково время работы алгоритма 2.1? Краткое описание алгоритмов сортировки см. в примечаниях (раздел 2.5).

Задача 2.6. Напишите программу, реализующую алгоритм 2.1 с алгоритмом 2.2 для отслеживания *связных компонент*. Исходите из того, что входные данные заданы в виде матрицы смежности размера $n \times n$.

Теперь мы докажем, что алгоритм Краскала работает. То, что алгоритм Краскала порождает остовное дерево, очевидно не сразу, не говоря уже об остовном дереве минимальной стоимости. Для того чтобы увидеть, что результирующая коллекция T ребер является остовным деревом для G, исходя из того, что G связен, мы должны показать, что (V, T) является связным и ациклическим.

Очевидно, что коллекция T является ациклической, потому что мы ни разу не добавляем ребро, которое приводит к циклу. Для того чтобы показать, что (V, T) является связным, мы рассуждаем следующим образом. Пусть u и v равны двум несовпадающим узлам в V. Поскольку G является связным, существует путь p, соединяющий u и v в G. Алгоритм рассматривает каждое ребро e_i графа G по очереди и помещает e_i в T, если только $T \cup \{e_i\}$ не образует цикла. Но в последнем случае в T уже должен быть путь, соединяющий конечные точки e_i, поэтому удаление e_i не делает граф несвязным.

Этот аргумент можно формализовать, показав, что следующее формальное суждение является инвариантом цикла в алгоритме Краскала:

Множество ребер $T \cup \{e_{i+1}, \ldots, e_m\}$ связывает все узлы в V. (2.1)

Лемма 2.7. Алгоритм 2.1 выводит дерево T при условии, что G был связен.

Задача 2.8. Доказать лемму 2.7, показав, что для связного G алгоритм 2.1 выдает T, которое является связным и ациклическим. Для того чтобы доказать, что T

является связным, покажите, что (2.1) является инвариантом цикла. На индуктивном шаге покажите, что если (2.1) соблюдается после исполнения i цикла, то $T \cup \{e_{i+2}, \ldots, e_m\}$ связывает все узлы V после исполнения $(i + 1)$ цикла. Сделайте вывод по индукции, что (2.1) соблюдается для всех i. Наконец, покажите, как использовать этот инвариант цикла, для того чтобы доказать, что T является связным. Как вы будете аргументировать, что T является ациклическим?

Задача 2.9. Предположим, что $G = (V, E)$ не является связным. Покажите, что в этом случае, когда G передается на вход алгоритма Краскала, данный алгоритм вычисляет *остовной лес* G. Сначала определите понятия связной компоненты и остовного леса. Затем приведите формальное доказательство, используя идею инварианта цикла, как в задаче 2.8.

Для того чтобы показать, что остовное дерево, полученное в результате этого алгоритма, на самом деле является остовным деревом минимальной стоимости, мы рассуждаем, что после каждой итерации цикла множество T ребер может быть расширено до остовного дерева минимальной стоимости, используя ребра, которые еще *не были* рассмотрены. Следовательно, после завершения все ребра будут рассмотрены, поэтому T должно само быть остовным деревом минимальной стоимости. Мы говорим, что множество T ребер из G является *перспективным*, если T может быть расширено до остовного дерева минимальной стоимости для G, то есть T является перспективным, если существует остовное дерево минимальной стоимости T такое, что $T \subseteq T'$.

Лемма 2.10. Выражение «T является перспективным» является инвариантом цикла для алгоритма Краскала.

Доказательство. Доказательство выполняется по индукции на числе итераций основного цикла алгоритма Краскала. Базовый случай: на этом этапе алгоритм прошелся по циклу ноль раз, и изначально T является пустым множеством, которое, очевидно, является перспективным (пустое множество является подмножеством любого множества).

Индукционный шаг: допустим, что T является перспективным, и покажем, что T остается перспективным после еще одной итерации цикла.

Обратите внимание, что ребра, используемые для расширения T до остовного дерева, должны поступать из еще не рассмотренных ребер, поскольку уже рассмотренные ребра либо уже находятся в T, либо были отклонены, так как они образуют цикл. Мы исследуем случаи по порядку относительно того, что произойдет после рассмотрения ребра e_i.

Случай 1: e_i отклоняется. T остается неизменным, и оно по-прежнему перспективно. Есть один тонкий момент: T было перспективно перед исполнением цикла, имея в виду, что существовало подмножество ребер $S \subseteq \{e_i, \ldots, e_m\}$, которое расширяло T до остовного дерева минимальной стоимости, то есть $T \cup S$ является остовным деревом минимальной стоимости. Но после исполнения цикла ребра, расширяющие T до остовного дерева минимальной стоимости, будут прибывать из $\{e_{i+1}, \ldots, e_m\}$; но это не проблема, так как e_i не может быть частью S (поскольку тогда $T \cup S$ будет содержать цикл), поэтому $S \subseteq \{e_{i+1}, \ldots, e_m\}$, и поэтому S по-прежнему является кандидатом для расширения T до остовного дерева минимальной стои-

мости, даже после исполнения цикла. Следовательно, T остается перспективным после исполнения цикла, хотя ребра, расширяющие его до остовного дерева минимальной стоимости, прибывают из меньшего множества (то есть не содержащего e_i).

Случай 2: e_i принимается. Мы должны показать, что $T \cup \{e_i\}$ по-прежнему является перспективным. Поскольку T является перспективным, существует остовное дерево минимальной стоимости T_1 такое, что $T \subseteq T_1$. Рассмотрим два подслучая.

Подслучай a: $e_i \in T_1$. Тогда очевидно, что $T \cup \{e_i\}$ является перспективным.

Подслучай b: $e_i \notin T_1$. Тогда, согласно приведенной ниже лемме о замене, в $T_1 - T_2$ есть ребро e_j, где T_2 – это остовное дерево, полученное в результате алгоритма, такое, что $T_3 = (T_1 \cup \{e_i\}) - \{e_j\}$ является остовным деревом. Обратите внимание, что $i < j$, так как в противном случае e_j было бы отклонено из T и тем самым образовался бы цикл в T, и поэтому так же и в T_1. Следовательно $c(e_i) \leqslant c(e_j)$, поэтому $c(T_3) \leqslant c(T_1)$, и поэтому T_3 также должно быть остовным деревом минимальной стоимости. Поскольку $T \cup \{e_i\} \subseteq T_3$, из этого следует, что $T \cup \{e_i\}$ является перспективным.

Этим завершается доказательство индукционного шага. □

Рассмотрим граф на рис. 2.1 и выполним алгоритм Краскала, представленный на рис. 2.2, начиная с верхнего левого графа, продолжая вправо, затем до следующей строки графа, продвигаясь слева направо, заканчивая в нижнем правом углу результирующим остовным деревом минимальной стоимости.

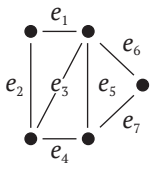


Рис. 2.1 ❖ Все ребра имеют стоимость 1

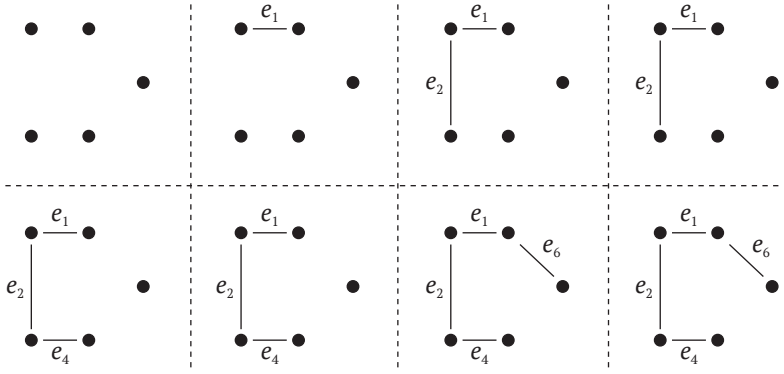


Рис. 2.2 ❖ Прогон алгоритма Краскала на графе из рис. 2.1

Изначально в левом верхнем углу у нас нет ребер и $T = \emptyset$, и на каждой итерации мы рассматриваем следующее ребро, в результате чего получаем:

Итерация	Ребро	Текущий T	Основное дерево минимальной стоимости (ОДМС), расширяющее T
0		$\emptyset$	$\{e_1, e_3, e_4, e_7\}$
1	e_1	$\{e_1\}$	$\{e_1, e_3, e_4, e_7\}$
2	e_2	$\{e_1, e_2\}$	$\{e_1, e_2, e_4, e_7\}$
3	e_3	$\{e_1, e_2\}$	$\{e_1, e_2, e_4, e_7\}$
4	e_4	$\{e_1, e_2, e_4\}$	$\{e_1, e_2, e_4, e_7\}$
5	e_5	$\{e_1, e_2, e_4\}$	$\{e_1, e_2, e_4, e_7\}$
6	e_6	$\{e_1, e_2, e_4, e_6\}$	$\{e_1, e_2, e_4, e_6\}$
7	e_7	$\{e_1, e_2, e_4, e_6\}$	$\{e_1, e_2, e_4, e_6\}$

Обратите внимание, что алгоритм рассматривает ребра в порядке их индексов, то есть $e_1, e_2, e_3, e_4, e_5, e_6, e_7$, и что стоимость всех этих ребер равна 1. (Таким образом, любое упорядочение этих ребер породит остовное дерево минимальной стоимости, но не обязательно оно будет таким же, что и каноническое упорядочение.)

Лемма 2.11 (лемма о замене)[1]. Пусть G равно связному графу, и пусть T_1 и T_2 равны любым двум остовным деревьям для G. Для каждого ребра e в $T_2 - T_1$ существует ребро e' в $T_1 - T_2$ такое, что $T_1 \cup \{e\} - \{e'\}$ является остовным деревом для G (см. рис. 2.3).

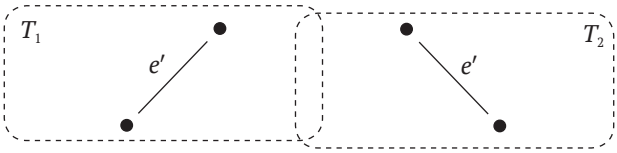


Рис. 2.3 ❖ Лемма о замене

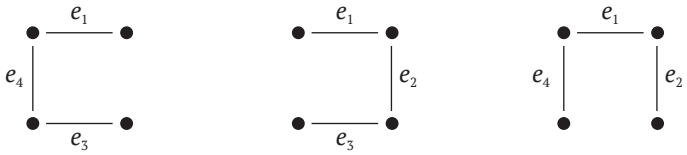


Рис. 2.4 ❖ Пример леммы о замене: левый и средний графы являются двумя разными остовными деревьями одного и того же графа. Предположим, мы добавляем ребро e_4 в среднее дерево; затем мы удаляем e_3 и получаем крайнее правое остовное дерево

Задача 2.12. Докажите эту лемму. (Подсказка: пусть e равно ребру в $T_2 - T_1$. Тогда $T_1 \cup \{e\}$ содержит цикл – могут ли все ребра в этом цикле принадлежать T_2?)

Задача 2.13. Предположим, что ребро e_1 имеет меньшую стоимость, чем любое другое ребро; то есть $c(e_1) < c(e_i)$ для всех $i > 1$. Покажите, что каждое остовное дерево минимальной соимости для G включает e_1.

---

[1] Лемма Стейница о замене (exchange lemma) – утверждение в линейной алгебре о том, что любое множество линейно независимых векторов в линейном пространстве можно дополнить до базиса пространства элементами некоторого заданного базиса. Лема используется в доказывании утверждения об одинаковом числе элементов во всех базисах линейного пространства. https://uk.wikipedia.org/wiki/Лема_Стейніца_про_заміну. – *Прим. перев.*

Задача 2.14. Прежде чем алгоритм 2.1 продолжит работу, он упорядочивает ребра в строке 1 и, предположительно, разрывает совпадения – то есть сортирует ребра с одной и той же стоимостью – произвольным образом. Покажите, что для каждого остовного дерева минимальной стоимости T графа G существует определенный способ разрыва совпадений, благодаря чему алгоритм возвращает T.

Задача 2.15. Напишите программу, которая на входе принимает описание решетки и на выходе выводит ее остовное дерево минимальной стоимости. n-узловая решетка представляет собой граф, состоящий из n^2 узлов, организованных в виде квадратного массива из $n \times n$ точек. Каждый узел может быть связан не более чем с узлами непосредственно над и под ним (если они существуют) и с двумя узлами сразу слева и справа (если они существуют). Пример 4-узловой решетки приведен на рис. 2.5.

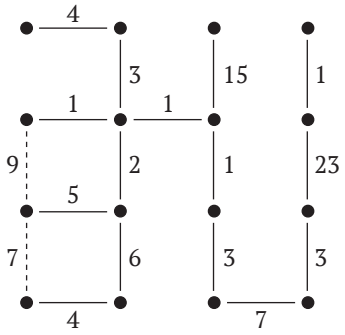


Рис. 2.5 ❖ Пример 4-узловой решетки.
Обратите внимание, что она имеет $4^2 = 16$ узлов и 17 ребер

Какое наибольшее число ребер может быть у n-узловой решетки? У нас есть следующее соглашение об именовании узлов: мы именуем узлы слева направо, строка за строкой, начиная с верхней строки. Таким образом, наша 4-узловая решетка описывается следующим списком смежности:

$$4 : (0, 1; 4), (1, 5; 3), (2, 6; 15), (3, 7; 1), (4, 5; 1), (5, 6; 1), \ldots, \tag{2.2}$$

где первое число – это параметр размера решетки, и первые два числа в каждой тройке обозначают два узла, которые описывают ребро, и третье число после точки с запятой дает стоимость этого ребра.

Если на вход подается список троек, то ваша программа должна сначала проверить, описывает ли этот список решетку, а затем вычислить для решетки ее остовное дерево минимальной стоимости. В нашем примере с 4-узловой решеткой ребра, обозначенные сплошной линией, описывают остовное дерево минимальной стоимости. Также обратите внимание, что ребра в (2.2) не обязательно должны быть заданы в определенном порядке.

Ваша программа должна принимать на входе файл, скажем `graph.txt`, содержащий список, такой как (2.1). Например, `2:(0,1;9),(2,3;5),(1,3;6),(0,2;2)`, и она должна выводить граф, показывающий ребра остовного дерева минимальной стоимости, прямо на экран. Граф должен быть «текстовым», в котором символ «*» описывает

узлы и символы «-» и «|» описывают ребра. В этом примере остовное дерево минимальной стоимости, заданное 2-узловой решеткой, представляется как:

```
* *
| |
*-*
```

2.2. Задания с предельными сроками и прибылями

Мы имеем n заданий, каждое из которых занимает единицу времени, и обработчик, на котором мы хотели бы запланировать их последовательно настолько прибыльно, насколько это возможно. У каждого задания есть связанная с ним прибыль, а также предельный срок; если задание не запланировано к своему предельному сроку, то мы не получаем его прибыль. Поскольку каждое задание занимает одинаковое количество времени, мы полагаем, что *расписание* S состоит из последовательности ячеек заданий, или «слотов», 1, 2, 3, ..., где $S(t)$ – это задание, запланированное в ячейке t.

Формально входными данными является последовательность пар (d_1, g_1), (d_2, g_2), ..., (d_n, g_n), где $g_i \in \mathbb{R}^+$ – это прибыль (прирост), получаемая от задания i, и $d_i \in \mathbb{N}$ – предельный срок для задания i. В разделе 4.5 мы рассмотрим случай, когда задания имеют произвольную длительность – заданную положительным целым числом. Однако когда длительности произвольны, а не имеют одинаковую единицу измерения, жадный подход, «похоже»[1], не справляется.

Расписание представляет собой массив $S(1)$, $S(2)$, ..., $S(d)$, где $d = \max d_i$, то есть d – это последний предельный срок, после которого задания не могут быть запланированы. Если $S(t) = i$, то задание i запланировано на момент времени t, $1 \leqslant t \leqslant d$. Если $S(t) = 0$, то никакого задания на момент времени t не запланировано. Расписание S является *допустимым*, если оно удовлетворяет двум условиям:

- **условие 1:** если $S(t) = i > 0$, то $t \leqslant d_i$, то есть каждое запланированное задание соответствует своему сроку;
- **условие 2:** если $t_1 \neq t_2$, а также $S(t_1) \neq 0$, тогда $S(t_1) \neq S(t_2)$, то есть каждое задание планируется не более одного раза.

Задача 2.16. Напишите программу, которая на входе принимает расписание S и последовательность заданий и проверяет, является ли S допустимым.

Пусть суммарная прибыль расписания S равна $P(S) = \sum_{t=1}^{d} g_{S(t)}$, где $g_0 = 0$. Мы хотим найти допустимое расписание S, прибыль которого $P(S)$ как можно больше; это может быть достигнуто с помощью жадного алгоритма 2.3, который упорядочивает задания в невозврастающем порядке прибыли и размещает их как можно позже в пределах их предельного срока. Удивительно, что этот алгоритм работает, и этот факт, похоже, является научным подтверждением преимуществ волокиты, именуемой также *прокрастинацией*.

---

[1] Мы говорим «похоже», ставя это слово в кавычки, потому что нет известного доказательства того, что жадный алгоритм не будет справляться; такое доказательство потребует строгого определения того, что означает «порождение жадным алгоритмом решения», которое само по себе является сложной задачей (см. [Allan Borodin (2003)]).

Строка 7 в алгоритме 2.3 находит последнюю свободную ячейку, которая соответствует предельному сроку; если такая свободная ячейка не существует, то задание выполняться не будет. То есть если нет t, удовлетворяющего как $S(t) = 0$, так и $t \leqslant d_i$, то последняя команда в строке 7, $S(t) \leftarrow i$, не исполняется и цикл for рассматривает следующее i.

Алгоритм 2.3. Планирование заданий

```
1: Отсортировать задания в невозрастающем порядке прибылей: g_1 ⩾ g_2 ⩾ ... ⩾ g_n
2: d ← max_i d_i
3: for t : 1..d do
4:         S(t) ← 0
5: end for
6: for i : 1..n do
7:         Найти самое крупное t такое, что S(t) = 0 и t ⩽ d_i, S(t) ← i
8: end for
```

Задача 2.17. Реализуйте алгоритм 2.3 для планирования заданий.

Теорема 2.18. Жадное решение задачи планирования заданий является оптимальным. То есть прибыль $P(S)$ расписания S, вычисленная алгоритмом 2.3, является максимально большой.

Расписание является *перспективным*, если его можно расширить до оптимального расписания. Расписание S' расширяет расписание S, если для всех $1 \leqslant t \leqslant d$, если $S(t) \neq 0$, то $S(t) = S'(t)$. Например, $S' = (2, 0, 1, 0, 3)$ расширяет $S = (2, 0, 0, 0, 3)$.

Лемма 2.19. Выражение «S является перспективным» является инвариантом для (второго) цикла for в алгоритме 2.3.

На самом деле, как и в случае алгоритма Краскала в предыдущем разделе, мы должны более точно определить понятие «перспективный» в лемме 2.19: мы говорим, что «S является перспективным после i-й итерации цикла в алгоритме 2.3», если S может быть расширено до оптимального расписания с использованием заданий из числа $\{i + 1, i + 2, \ldots, n\}$, то есть, используя подмножество тех заданий, которые еще не были рассмотрены.

Задача 2.20. Рассмотрим следующие входные данные:

$$\{\underbrace{(1, 10)}_{1}, \underbrace{(1, 10)}_{2}, \underbrace{(2, 8)}_{3}, \underbrace{(2, 8)}_{4}, \underbrace{(4, 6)}_{5}, \underbrace{(4, 6)}_{6}, \underbrace{(4, 6)}_{7}, \underbrace{(4, 6)}_{8}\},$$

где задания были пронумерованы внизу для удобства. Выполним трассировку работы алгоритма 2.3 на этих входных данных. Слева поместим номера заданий в соответствующие ячейки; справа покажем, как оптимальное решение корректируется с целью сохранения свойства «перспективности». Начнем со следующей конфигурации:

$S^0 = \boxed{0}\boxed{0}\boxed{0}\boxed{0}$ и $S^0_{\text{opt}} = \boxed{2}\boxed{4}\boxed{5}\boxed{8}$.

Задача 2.21. Почему из леммы 2.19 следует теорема 2.18? (Подсказка: ответ кроется в непосредственном наблюдении.)

Теперь докажем лемму 2.19.

Доказательство. Доказательство выполняется по индукции. Базовый случай: после 0-й итерации цикла $S = (0, 0, \ldots, 0)$, и мы можем расширить его заданиями $\{1, 2, \ldots, n\}$, то есть в нашем распоряжении есть все задания; поэтому S является перспективным, так как мы можем взять любое оптимальное расписание, и оно будет расширением S.

Индукционный шаг: предположим, что S является перспективным, и пусть S_{opt} равно некому оптимальному расписанию, которое расширяет S. Пусть S' равно результату еще одной итерации цикла, в котором рассматривается задание i. Мы должны доказать, что S' остается перспективным, поэтому цель состоит в том, чтобы показать, что существует оптимальное расписание S_{opt}, которое расширяет S'. Рассмотрим два случая:

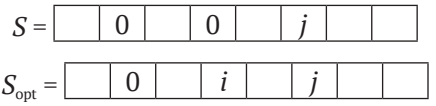


Рис. 2.6 ❖ Если S имеет задание j в некой позиции, то S_{opt} также имеет задание j в той же позиции. Если в заданной позиции S имеет ноль (там не запланировано ни одного задания), то S_{opt} может иметь ноль или другое задание в той же позиции

Случай 1: задание i не может быть запланировано. Тогда $S' = S$, поэтому мы принимаем $S'_{opt} = S_{opt}$, и доказательство завершено. Единственная тонкость состоит в том, что S было расширяемо до S_{opt} заданиями в $\{i, i + 1, \ldots, n\}$, но после i-й итерации в нашем распоряжении задания i больше нет.

Задача 2.22. Покажите, что эта «тонкость», упомянутая в приведенном выше абзаце, не является проблемой.

Случай 2: Задание i планируется на момент времени t_0, поэтому $S'(t_0) = i$ (тогда как $S(t_0) = 0$) и t_0 является самым последним возможным временем для задания i в расписании S. Мы имеем два подслучая.

Подслучай a: задание i запланировано в S_{opt} на момент времени t_1:

- если $t_1 = t_0$, то, как и в случае 1, просто примем $S'_{opt} = S_{opt}$;
- если $t_1 < t_0$, то пусть S'_{opt} равно S_{opt}, за исключением того, что мы переставляем t_0 и t_1, то есть принимаем $S'_{opt}(t_0) = S_{opt}(t_1) = i$ и $S'_{opt}(t_1) = S_{opt}(t_0)$. Тогда S'_{opt} является допустимым (вопрос почему № 1), оно расширяет S' (вопрос почему № 2), и $P(S'_{opt}) = P(S_{opt})$ (вопрос почему № 3).

Случай $t_1 > t_0$ невозможен (вопрос почему № 4).

Подслучай b: задание i не запланировано в S_{opt}. Тогда мы просто определяем S'_{opt} таким же, как S_{opt}, за исключением, что $S'_{opt}(t_0) = i$. Поскольку S_{opt} является допустимым, то и S'_{opt} тоже является допустимым, и поскольку S'_{opt} расширяет S_{opt}, нам нужно только показать, что $P(S'_{opt}) = P(S_{opt})$.

Это вытекает из следующего утверждения:

Утверждение 2.23. Пусть $S_{opt}(t_0) = j$. Тогда $g_j \leqslant g_i$.

Доказательство. Докажем утверждение от обратного: предположим, что $g_j > g_i$ (заметим, что в этом случае $j \neq 0$). Тогда задание j было рассмотрено перед задани-

ем i. Поскольку задание i было запланировано на момент времени t_0, задание j должно быть запланировано на момент времени $t_2 \neq t_0$ (мы знаем, что задание j было запланировано в S, так как $S(t_0) = 0$ и $t_0 \leqslant d_j$, поэтому имелась ячейка для задания j, и поэтому оно было запланировано). Но S_{opt} расширяет S, и $S(t_2) = j \neq S_{opt}(t_2)$ – противоречие. □

Этим завершается доказательство индукционного шага. □

Задача 2.24. Убедитесь, что вы можете ответить на все вопросы «почему» в приведенном выше доказательстве. Кроме того, где в доказательстве утверждения мы используем тот факт, что $j \neq 0$?

Задача 2.25. При каких условиях на входных данных существует уникальное оптимальное расписание? Если существует более одного оптимального расписания, и при наличии одного такого оптимального расписания всегда ли существует распределение заданий, которое по-прежнему находится в невозрастающем порядке прибылей, приводя к тому, что алгоритм выдает это конкретное оптимальное расписание?

2.3. Дальнейшие примеры и задачи

2.3.1. Отсчитывание сдачи

Задача отсчитывания сдачи, кратко описанная во введении к этой главе, состоит в том, чтобы заплатить заданную сумму, используя наименьшее число монет, при этом применяя какой-то фиксированный номинал и неограниченный запас монет каждого номинала.

Рассмотрим следующий ниже жадный алгоритм для решения задачи отсчитывания сдачи, где номиналы равны C = {1, 10, 25, 100}. При $n \in \mathbb{N}$ на входе алгоритм выдает наименьший список L монет (из числа C), сумма которых равна n.

Обратите внимание, что s равно сумме значений монет в L и что, строго говоря, L является *мультимножеством* (в мультимножестве один и тот же элемент может появляться более одного раза).

Задача 2.26. Реализуйте алгоритм 2.4 для отсчитывания сдачи.

Задача 2.27. Покажите, что алгоритм 2.4 (с заданными номиналами) не обязательно дает оптимальное решение. То есть покажите n, для которого выход L содержит больше монет, чем *оптимальное решение*.

Алгоритм 2.4. Отсчитывание сдачи

```
1: C ← {1, 10, 25, 100}; L ← ∅; s ← 0
2: while (s < n) do
3:         найти наибольшее x в C такое, что s + x ⩽ n
4:         L ← L ∪ {x}; s ← s + x
5: end while
6: return L
```

Задача 2.28. Предположим, что $C = \{1, p, p^2, \ldots, p^n\}$, где $p > 1$ и $n \geqslant 0$ – это целые числа. То есть «$C \leftarrow \{1, 10, 25, 100\}$» в строке 1 алгоритма 2.4 заменяется на «$C \leftarrow \{1, p, p^2,$

..., p^n}». Покажите, что с этой серией номиналов (для некоторых фиксированных p, n) приведенный выше жадный алгоритм всегда находит оптимальное решение. (Подсказка: начните с подходящего определения перспективного списка.)

2.3.2. Паросочетание с максимальным весом

Пусть $G = (V_1 \cup V_2, E)$ равно *двудольному графу*, то есть графу с реберным множеством $E \subseteq V_1 \times V_2$ с непересекающимися множествами V_1 и V_2. $w : E \to \mathbb{N}$ назначает вес $w(e) \in \mathbb{N}$ для каждого ребра $e \in E = \{e_1, ..., e_m\}$. Паросочетанием для G является подмножество $M \subseteq E$ такое, что никакие два ребра в M не имеют общей вершины. Вес M равен $w(M) = \sum_{e \in M} w(e)$.

Задача 2.29. Дайте простой жадный алгоритм, который для заданного двудольного графа с весами ребер пытается найти паросочетание с максимально возможным весом.

Задача 2.30. Приведите пример двудольного графа с весами ребер, для которого ваш алгоритм в задаче 2.29 не может найти паросочетание с максимально возможным весом.

Задача 2.31. Предположим, что все веса ребер в двудольном графе не совпадают, и каждый из них является степенью числа 2. Докажите, что в этом случае вашему жадному алгоритму всегда удается найти паросочетание с максимальным весом. (В случае этого вопроса будем считать, что все ребра присутствуют, то есть что $E = V \times V$.)

2.3.3. Кратчайший путь

Следующий пример жадного алгоритма очень красив. Он напоминает одного из старых картографов, который создавал карты мира с белыми пятнами – неизвестными и неизведанными местами.

Предположим, что у нас есть граф $G = (V, E)$, выделенный начальный узел s и функция стоимости для каждого ребра $e \in E$, обозначенная $c(e)$. Нас просят вычислить самые дешевые пути из s во все остальные узлы в G, где стоимость пути равна сумме стоимостей его ребер.

Рассмотрим следующий жадный алгоритм: алгоритм поддерживает множество s разведываемых узлов, и для каждого $u \in S$ он хранит значение $d(u)$, то есть самый дешевый путь внутри S, начиная в s и заканчивая в u.

Изначально $S = \{s\}$ и $d(s) = 0$. Теперь для каждого $v \in V - S$ мы находим кратчайший путь в v, путешествуя внутри разведываемой части S до некоторого $u \in S$, за которым следует единственное ребро (u, v). См. рис. 2.7.

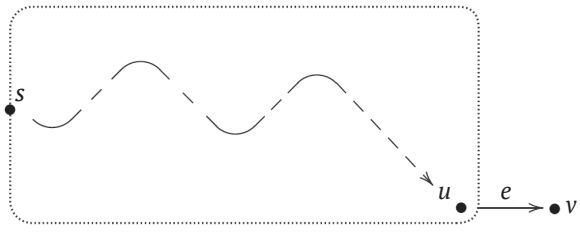


Рис. 2.7 ❖ Вычисление пути из s в u

То есть мы вычисляем:

$$d'(v) = \min_{u \in S, e=(u,v)} d(u) + c(e). \tag{2.3}$$

Мы выбираем узел в $v \in V - S$, для которого минимизируется (2.3), добавляем v в S, устанавливаем $d(v) = d'(v)$ и повторяем. Таким образом, мы добавляем один узел за раз в разведываемую часть, и мы останавливаемся, когда $S = V$.

Этот жадный алгоритм вычисления кратчайшего пути связан с алгоритмом нидерландского ученого *Эдсгера Дейкстры*. Нетрудно заметить, что время работы алгоритма равно $O(n^2)$.

Задача 2.32. Спроектируйте алгоритм в псевдокоде и покажите, что в конце для каждого $u \in V$ $d(u)$ является стоимостью самого дешевого пути из s в u.

Задача 2.33. Открытый протокол предпочтения кратчайшего пути (open shortest path first, OSPF) – это протокол маршрутизации для IP, подробно описанный в рабочем предложении `RFC 2328` (где `RFC` расшифровывается как «request for comment», то есть «запрос комментария», который представляет собой серию меморандумов, опубликованных рабочей группой по стандартам для сети интернет, описывающей работу интернета). Широко применяемый протокол маршрутизации OSPF использует жадный алгоритм Дейкстры для вычисления так называемого «дерева кратчайших путей», который для конкретного узла x в интернете перечисляет лучшие связи со всеми другими узлами в подсети узла x.

Напишите программу, реализующую упрощенный механизм политики динамической маршрутизации. Точнее, требуется реализовать демон управления таблицей маршрутизации, который поддерживает базу данных состояния связи как в протоколе внутренней маршрутизации OSPF. Мы исходим из того, что все узлы являются либо маршрутизаторами, либо сетями (то есть нет «мостов», «концентраторов» и т. д.).

Назовите свою программу `routed` (как в демоне маршрутизации). После запуска в командной строке она ожидает инструкций и выполняет действия:

1) `add rt` (маршрутизаторы) – эта команда добавляет маршрутизаторы в таблицу маршрутизации, где (маршрутизаторы) – это разделенный запятыми список (положительных) целых чисел и целочисленных диапазонов. То есть (маршрутизаторы) могут быть `6,9,10-13,4,8`, которые будут включать маршрутизаторы

   ```
   rt4,rt6,rt8,rt9,rt10,rt11,rt12,rt13
   ```

   Ваша программа должна быть достаточно робастной, чтобы принимать любую такую легальную последовательность (включая одиночный маршрутизатор) и возвращать сообщение об ошибке, если команда пытается добавить уже существующий маршрутизатор (но другие валидные маршрутизаторы в списке (маршрутизаторы) должны добавляться независимо);
2) `del rt` (маршрутизаторы) – удаляет маршрутизаторы, указанные в списке (маршрутизаторы). Если команда пытается удалить несуществующий маршрутизатор, то должно быть возвращено сообщение об ошибке; нам нужна робастность: существующие маршрутизаторы должны быть удалены; при попытке удалить несуществующие маршрутизаторы должно

возвращаться сообщение об ошибке (с указанием маршрутизаторов-«нарушителей»). После показа сообщения об ошибке программа не должна останавливаться;

3) `add nt` (сети) – добавить сети согласно списку (сети); тот же формат, что и для добавления маршрутизаторов. Например, «`add nt 89`» приведет к добавлению `nt89`. Обработка ошибок должна проделываться аналогично случаю добавления маршрутизаторов;
4) `del nt` (сети) – удалить сети, заданные в списке (сети);
5) `con` *x y z* – связать узел *x* и *y*, где *x*, *y* – это существующие маршрутизаторы и сети (например, *x* = `rt8` и *y* = `rt90`, или *x* = `nt76` и *y* = `rtl`) и *z* – это стоимость соединения. Если *x* или *y* не существует, то должно быть возвращено сообщение об ошибке. Обратите внимание, что сеть является ориентированной; то есть следующие две команды не эквивалентны: «`con rt3 rt5 1`» и «`con rt5 rt3 1`».

   Важно отметить, что две сети не могут быть связаны напрямую; попытка это сделать должна генерировать сообщение об ошибке. Если связь между *x* и *y* уже существует, то она обновляется новой стоимостью *z*;
6) `display` – эта команда показывает таблицу маршрутизации, то есть базу данных о состоянии связи. Например, в результате добавления `rt3`, `rt5`, `nt8`, `nt9` и подачи команд «`con rt5 rt3 1`» и «`con rt3 nt8 6`» будет показана следующая ниже таблица маршрутизации:

```
    rt3 rt5 nt8 nt9
rt3      1
rt5
nt8  6
nt9
```

   Обратите внимание, что (согласно рабочему предложению `RFC 2338`, описывающему версию OSPF 2) мы читаем таблицу следующим образом: «сначала столбец, затем строка». Тем самым в таблице говорится, что есть связь из `rt5` в `rt3` со стоимостью 1 и еще одна связь из `rt3` в `nt8` со стоимостью 6;
7) `tree` *x* – эта команда вычисляет дерево кратчайших путей из базы данных состояния связи, где x является корнем. Обратите внимание, что в этом случае *x* должен быть маршрутизатором. Выходные данные должны быть представлены в следующем виде:

$$\begin{aligned} w_1 &: x, v_1, v_2, \ldots, v_n, y_1 \\ &: \text{нет пути в } y_2 \\ w_3 &: x, u_1, u_2, \ldots, u_m, y_3 \\ &\vdots \end{aligned}$$

   где w_1 – это стоимость пути (сумма стоимостей ребер) из *x* в y_1 с промежуточными узлами узла v_i (то есть переходами, для того чтобы попасть из *x* в y_1. Каждый узел y_j в базе данных должен быть перечислен; если нет никакого пути из *x* в y_j, то программа должна об этом сообщить, как в приведенном выше примере выходных данных.

   Продолжая пример базы данных состояния связи в объяснении команды `display`, выходными данными выполнения команды «`tree rt5`» будут:

```
1 : rt5,rt3
7 : rt5,rt3,nt8
  : no path to nt9
```

Так же, как это сделано в стандарте OSPF, дерево путей должно быть вычислено жадным алгоритмом Дейкстры.

Наконец, между двумя узлами может быть несколько путей одинакового значения; в этом случае объясните в комментариях в вашей программе, как ваша схема выбирает один из них;

8) `quit` – убивает демона.

2.3.4. Коды Хаффмана

Еще один важный пример жадного решения дает алгоритм Хаффмана, широко используемый и эффективный метод сжатия данных без потерь. В алгоритме Хаффмана используется таблица частот появления символов, которая предназначена для построения оптимального способа представления каждого символа в виде двоичной строки. См. § 16.3 в книге [Cormen и соавт. (2009)] для получения подробной информации, но следующий ниже пример иллюстрирует ключевую идею.

Предположим, что у нас есть строка s над алфавитом {a, b, c, d, e, f} и $|s| = 100$. Предположим также, что символы в s встречаются с частотами соответственно 44, 14, 11, 17, 8, 6. Так как имеется шесть символов, если бы для их представления мы использовали двоичные кодовые слова фиксированной длины, то нам потребовалось бы три бита, а значит, 300 символов для представления строки.

Вместо кодировки фиксированной длины мы хотим дать частым символам короткое кодовое слово и нечастым символам длинное кодовое слово. Мы рассматриваем только коды, в которых кодовое слово не является префиксом какого-либо другого кодового слова. Такие коды называются *префиксными кодами*; ограничивая внимание префиксными кодами, мы не получаем никакой потери общности, так как можно показать, что любой код всегда можно заменить префиксным кодом, который, по крайней мере, так же хорош.

С помощью префиксного кода кодирование и декодирование выполняются легко; в случае кодирования мы просто сцепляем кодовые слова, представляющие каждый символ файла. Поскольку ни одно кодовое слово не является префиксом любого другого, кодовое слово, начинающееся с закодированной строки, является однозначным, и поэтому декодирование выполняется просто.

Префиксный код может быть задан двоичным деревом, в котором листья помечены символом и его частотой, а каждый внутренний узел помечен суммой частот листьев в его поддереве (см. рис. 2.8). Мы строим код символа, проходя дерево, начиная с корня и записывая 0 для левого ребенка и 1 для правого ребенка.

Пусть $\sum$ равно алфавиту из n символов, и пусть $f : \sum \to \mathbb{N}$ равно функции частот. Алгоритм Хаффмана строит дерево T, соответствующее оптимальному коду, снизу вверх. Он начинается со множества $|\sum|$ листьев и выполняет последовательность из $|\sum| - 1$ операций «слияния» для создания финального дерева. На каждом шагу самые частые объекты подвергаются слиянию; результатом слияния двух объектов является новый объект, частота которого равна сумме частот двух объектов, которые были слиты вместе.

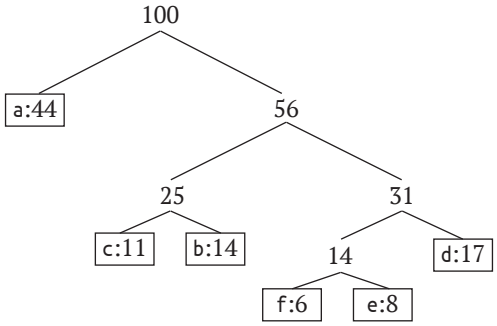


Рис. 2.8 ❖ Бинарное дерево
для префиксного кода переменной длины

Алгоритм 2.5. Хаффман

$n \leftarrow |\sum|$; $Q \leftarrow \sum$
for $i = 1..n - 1$ **do**
    разместить новый узел z
    left[z] $\leftarrow x$ = извлечь-min(Q)
    right[z] $\leftarrow y$ = извлечь-min(Q)
    $f(z) \leftarrow f(x) + f(y)$
    вставить z в Q
end for

Задача 2.34. Рассмотрим файл, состоящий из 100 символов в кодировке ASCII со следующими частотами:

Символ	a	b	c	d	e	f	g	h
Частота	40	15	12	10	8	6	5	4

Используя стандартную кодировку ASCII, этот файл требует 800 бит. Вычислите префиксную кодировку переменной длины для этого файла и общее число бит при использовании данной кодировки.

Задача 2.35. Напишите программу, которая на входе принимает текстовый файл, скажем, над алфавитом ASCII, и использует алгоритм Хаффмана, чтобы сжать его в двоичную строку. Сжатый файл должен иметь заголовок, содержащий привязку символов к битовым строкам с целью правильного разжатия сжатого файла. Ваша программа должна быть способна выполнять как сжатие, так и разжатие. Сравните свое решение со стандартными инструментами сжатия, такими как *вспомогательная программа* `gzip`[1].

[1] Вспомогательная программа `gzip` реализует алгоритм Лемпеля–Зива–Уэлша (Lempel–Ziv–Welch, LZW), то есть алгоритм сжатия данных без потерь, доступный на платформах UNIX. В качестве входных данных он принимает любой файл и выводит сжатую версию с расширением `.gz`. Он описан в рабочих предложениях RFC 1951 и 1952.

2.4. Ответы к избранным задачам

Задача 2.2. Лист – это вершина с одним исходящим ребром; предположим, что листа нет. Выберите вершину, возьмите одно из исходящих из нее ребер. Поскольку каждая вершина имеет, по крайней мере, два смежных ребра, мы продолжаем приходить по одному ребру и уходить по другому. Поскольку существует конечное число ребер, мы должны в конечном итоге сформировать цикл. Противоречие.

Теперь мы покажем по индукции на n, что дерево с n вершинами должно иметь ровно $n - 1$ ребер. Базовый случай: $n = 1$, то есть дерево состоит из одного узла, и, следовательно, оно не имеет ребер; $n - 1 = 1 - 1 = 0$ ребер. Индукционный шаг: предположим, что у нас есть дерево с $n + 1$ узлами. Выберите лист и ребро, которое соединяет его с остальной частью дерева. Удаление этого листа и его ребра приводит к дереву с n узлами, а следовательно – по индукционной гипотезе, – с $n - 1$ ребрами. Таким образом, все дерево имеет $(n - 1) + 1 = n$ ребер, как и требуется.

Задача 2.4. Докажем это по индукции, с базовым случаем $n = 3$ (так как граф – без многочисленных ребер между одной и той же парой узлов – не может иметь цикл с менее чем 3 узлами). Если $n = 3$, и ребер больше, чем $n - 1 = 2$, то ребер должно быть ровно 3. Поэтому граф представляет собой цикл («треугольник»). Индукционный шаг: рассмотрим граф с $n + 1$ узлами ($n \geqslant 3$) и, по крайней мере, $n + 1$ ребрами. Если у графа есть узел с нулем или одним смежным с ним ребром, то, удалив этот узел (и его ребро, если оно есть), мы получим граф с n узлами и не менее n ребрами, и поэтому – по индукционной гипотезе – результирующий граф имеет цикл, и, значит, исходный граф также имеет цикл. В противном случае все узлы имеют, по крайней мере, два смежных ребра. Предположим, что v_0 является таким узлом, и (v_0, x), (v_0, y) – это два ребра. Удалим v_0 из графа, удалим ребра (v_0, x), (v_0, y) и заменим их одним ребром (x, y). Опять же – по индукционной гипотезе – в полученном графе должен быть цикл. Но тогда должен быть цикл и в исходном графе. (Обратите внимание, что существует $n + 1$ узлов, поэтому после удаления v_0 имеется n узлов и $n \geqslant 3$.)

Задача 2.5. Мы знаем, что строки 1–2 алгоритма 2.1 требуют не более $m^2 + 1$ шагов. Мы также должны создать массив D, который требует еще n шагов (где n – это число вершин).

Цикл for в строке 3 будет проходить ровно m итераций. Выражение «$T \cup \{e_i\}$ не имеет цикла» (где $e_i = (r, s)$) эквивалентно выражению «$D[r] \neq D[s]$», поэтому проверка в строке 4 требует только одного шага. Для цели установления верхней границы можно с уверенностью принять допущение, что каждая проверка возвращает «истину», поэтому мы должны пройти весь алгоритм 2.2 целиком в каждой итерации цикла for.

Алгоритм 2.2 требует 2 присваиваний, за которыми следует цикл, выполняемый n раз, и имеет не более 2 шагов; алгоритм 2.2 имеет сложность $O(2n + 2) = O(n)$.

Поэтому составной алгоритм, где алгоритм 2.2 используется для выполнения строки 4 алгоритма 2.1 и сортировка вставками используется для строки 1, явно выполняется за $O(m^2 + n + 1 + m(2n + 2))$. То же самое, если $p = \max(n, m)$, то алгоритм имеет сложность $O(p^2)$.

Другими словами, если число ребер больше числа вершин, то узким местом является алгоритм сортировки. Более того, исходя из допущения, что рассматриваемый граф является связным, число вершин равно не менее $n - 1$; любой граф

с $n - 1$ ребрами либо уже является остовным деревом, либо не является связным, поэтому можно с уверенностью допустить, что $m \geqslant n$. Использование сортировки слиянием, пирамидальной сортировки или быстрой сортировки позволит усовершенствовать сложность до $O(m \log(m))$.

Задача 2.8. Начнем с базового случая: перед первой итерацией t_0 является пустым множеством ($i = 0$). Поскольку G является связным, очевидно, что $\{e_1, e_2, \ldots, e_m\} = E$ связывает все узлы в V.

Далее мы доказываем по индукции. Допустим, что после $i - 1$ итераций $T_{i-1} \cup \{e_i, \ldots, e_m\}$ связывает все узлы в V. На итерации i у нас есть два случая.

Случай 1: $T_{i-1} \cup \{e_i\}$ не имеет цикла, поэтому $T_i = T_{i-1} \cup \{e_i\}$. $T_i \cup \{e_{i+1}, \ldots, e_m\}$ и $T_{i-1} \cup \{e_i, \ldots, e_m\}$ являются одним и тем же множеством, e_i только что перешло из «оставшихся» ребер в T. По гипотезе последнее из указанных реберных множеств связывает все узлы в V, поэтому первое должно связывать тоже.

Случай 2: $T_{i-1} \cup \{e_i\}$ содержит цикл, поэтому $T_i = T_{i-1}$. Рассмотрим любые два узла $u, v \in V$. По гипотезе существует путь из u в v, состоящий из ребер в $T_{i-1} \cup \{e_i, \ldots, e_m\}$. Если e_i в этом пути нет, то доказательство завершено. Путь между u и v попрежнему существует, так как мы только потеряли доступ к e_i. Если $e_i = (a, b)$ находится в этом пути, то мы можем заменить его другим путем из a в b; e_i был в цикле, поэтому еще один такой путь обязательно существует.

Мы нашли путь, связывающий произвольные u и v в $T_i \cup \{e_i, \ldots, e_m\}$ при условии, что он существовал в $T_{i-1} \cup \{e_i, \ldots, e_m\}$, тем самым завершив индукционный шаг и доказав, что (2.4) является инвариантом цикла.

Очевидно, что после всех $i = m$ итераций этот инвариант цикла читается как «$T_i \cup \{e_i, \ldots, e_m\}$ связывает все узлы в V», но e_m было последним ребром, поэтому $\{e_{i+1}, \ldots\}$ является пустым множеством. Следовательно, T_m связывает все узлы в V. По конструкции T_m не может содержать циклов; любое ребро, которое приводило бы к циклу, просто не было бы включено. Поэтому после m итераций T связывает все узлы в V и является ациклическим – T является остовным деревом G.

Задача 2.9. Для заданного неориентированного графа $G = (V, E)$ связная компонента $C = (V_c, E_c)$ графа G является непустым подмножеством V' множества V (вместе с его ребрами) таким, что для всех пар вершин $u, v \in V'$ существует путь из u в v (который мы будем выражать как «u и v связаны»), и, более того, для всех пар вершин x, y, таких, что $x \in V'$ и $y \in V - V'$, x и y не связаны (то есть отсутствует путь из x в y). Мы можем сделать несколько быстрых наблюдений о связных компонентах:

1) связные компоненты любого графа содержат разбиения его реберного и вершинного множеств, поскольку связность является отношением эквивалентности;
2) для любого ребра обе его конечные точки находятся в одной компоненте, поскольку она определяет связывающий их путь;
3) для любых двух вершин в компоненте связности существует связывающий их путь. Схожим образом любые две вершины в разных компонентах не обязательно связаны;
4) для любого пути каждое содержащееся ребро находится в одной и той же компоненте.

Остовной лес – это коллекция остовных деревьев, по одному для каждой связной компоненты. То есть реберное множество $F \subseteq E$ является остовным лесом $G = (V, E)$ тогда и только тогда, когда:

1) F не содержит циклов;
2) $(\forall u, v \in V)$, F связывает u и v тогда и только тогда, когда u и v связаны в G.

Пусть $G = (V, E)$ равно несвязному графу. То есть G имеет более одной компоненты. Обозначим через T_i состояние T в алгоритме Краскала после i итераций. Пусть $C = (V_c, E_c)$ равно компоненте графа G. В качестве доказательства того, что алгоритм Краскала приводит к остовному лесу для G, мы будем использовать следующий инвариант цикла:

Реберное множество $T_i \cup \{e_{i+1}, \ldots, e_m\}$ связывает все узлы в V_c. (2.4)

Базовый случай явно работает; $T_0 \cup \{e_1, \ldots, e_m\} = E$. Каждая вершина в V_c связана в G, и в нашем распоряжении есть каждое ребро в G.

Допустим, что $T_{i-1} \cup \{e_i, \ldots, e_m\}$ связывает все узлы в V_c.

Случай 1: e_i нет в E_c. Очевидно, что e_i не влияет на связность V_c, так как любой путь в C должен быть подмножеством E_c.

Случай 2: $e_i \in E_c$ и $T_{i-1} \cup \{e_i\}$ содержит цикл. Пусть u, v равны узлам, смежным с e_i. T_{i-1} не содержит цикла по конструкции, поэтому e_i завершает цикл в $T_{i-1} \cup \{e_i\}$. Тем самым в T_{i-1} уже есть путь (u, v), который можно использовать для замены e_i в любом другом пути. Следовательно, $T_{i-1} \cup \{e_{i+1}, \ldots, e_m\}$ связывает все, что было связано $T_{i-1} \cup \{e_i, \ldots, e_m\}$, поэтому присваивание $T_i = T_{i-1}$ с «потерей доступа» к e_i сохраняет инвариантность цикла.

Случай 3: $e_i \in E_c$ и $T_{i-1} \cup \{e_i\}$ не содержат цикла. Тогда $T_i = T_{i-1} \cup \{e_i\}$, поэтому $T_i \cup \{e_{i+1}, \ldots, e_m\} = T_{i-1} \cup \{e_i, \ldots, e_m\}$, и поэтому инвариант цикла соблюдается.

Посредством индукции мы показали, что инвариант цикла (2.4) соблюдается. Обратите внимание, что C было произвольной связной компонентой, поэтому T_m для каждой компоненты $C = (V_c, E_c)$ в G, T_m связывает каждый узел в V_c. Очевидно, что если любые два узла в V не связаны в G, то T их не связывает; для этого потребуются ребра не в E. Следовательно, T_m удовлетворяет обоим условиям, наложенным на упомянутый выше остовной лес.

Задача 2.12. Пусть e равно любому ребру в $T_2 - T_1$. Мы должны доказать существование $e' \in T_1 - T_2$ такого, что $(T_1 \cup \{e\}) - \{e'\}$ является остовным деревом. Поскольку $e \notin T_1$, добавив e в T_1, мы получим цикл (по лемме 2.3, которая доказана в задаче 2.4). Цикл имеет, по крайней мере, 3 ребра (граф G имеет, по крайней мере, 3 узла, так как в противном случае он не может иметь два несовпадающих остовных дерева!). Таким образом, в этом цикле существует ребро e' не в T_2. Причина в том, что если бы каждое ребро e' в цикле все-таки принадлежало T_2, то само T_2 имело бы цикл. Убрав e', мы нарушаем цикл, но результирующий граф, $(T_1 \cup \{e\}) - \{e'\}$, по-прежнему является связным и имеет размер $|T_1| = |T_2|$, то есть правильный размер для дерева, поэтому он должен быть ациклическим (в противном случае мы могли бы избавиться от какого-то ребра и иметь остовное дерево меньшего размера, чем T_1 и T_2, – но все остовные деревья имеют одинаковый размер), и поэтому $(T_1 \cup \{e\}) - \{e'\}$ является остовным деревом.

Задача 2.13. Сначала отметим, что если мы передадим в алгоритм Краскала G с ребрами в порядке следования их индексов (то есть пропустим шаг сортировки), то результирующее дерево будет включать e_1 – цикл не может образоваться первым (или вторым) ребром, поэтому e_1 будет добавлен в T на первой итерации. Следовательно, с необходимостью имеется остовное дерево T_1 графа G такое, что $e_1 \in T_1$.

От противного допустим, что существует остовное дерево минимальной стоимости T_2 такое, что $e_1 \notin T_1$. По лемме о замене существует e_j в T_2 такое, что $T_3 = T_2 \cup \{e_1\} - \{e_j\}$ является остовным деревом. Но $c(e_1) < c(e_j)$, поэтому $c(T_3) < c(T_2)$, то есть T_2 не является остовным деревом минимальной стоимости. Мы нашли наше противоречие; не может существовать остовного дерева минимальной стоимости, которое не содержит e_1. Поэтому любое остовное дерево минимальной стоимости включает e_1.

Задача 2.14. Пусть T равно любому остовному дереву минимальной стоимости для графа G. Переупорядочим ребра графа G по стоимостям, как и в алгоритме Краскала. Для любого блока ребер с одинаковой стоимостью поместим эти ребра, которые появляются в T, перед всеми остальными ребрами в этом блоке. Теперь докажем следующий инвариант цикла: множество ребер S, отобранных алгоритмом с изначальным упорядочением, как описано, всегда является подмножеством T. Изначально $S = \emptyset \subseteq T$. На индукционном шаге $S \subseteq T$ и S' является результатом добавления еще одного ребра в S. Если $S' = S$, то делать нечего, и если $S' = S \cup \{e\}$, то мы должны показать, что $e \in T$. Предположим, что это не так. Пусть T' равно результату алгоритма Краскала, который, как мы знаем, является остовным деревом минимальной стоимости. По лемме о замене мы знаем, что существует $e' \notin T'$ такое, что $T \cup \{e\} - \{e'\}$ является остовным деревом, и поскольку T было остовным деревом минимальной стоимости, мы знаем, что $c(e') \leqslant c(e)$, и, следовательно, e' было рассмотрено перед e. Так как e' не находится в T', оно было отклонено, поскольку оно должно было создать цикл в S, и, следовательно, в T – противоречие. Следовательно, $S \cup \{e\} \subseteq T$.

Задача 2.20. Ниже приведен след алгоритма; обратите внимание, что мы модифицируем оптимальное решение лишь настолько, насколько это необходимо для сохранения свойства расширения.

$S^1 =$	1	0	0	0	$S^1_{\text{opt}} =$	1	4	5	8
$S^2 =$	1	3	0	0	$S^2_{\text{opt}} =$	1	3	5	8
$S^3 =$	1	3	0	5	$S^3_{\text{opt}} =$	1	3	8	5
$S^4 =$	1	3	6	5	$S^4_{\text{opt}} =$	1	3	6	5

Задача 2.21. Допустим, что после каждой итерации расписание S является перспективным. После финальной итерации S по-прежнему является перспективным, но единственными незапланированными заданиями являются те, которые не могут расширить S на любой момент времени. Другими словами, S не может быть расширено за пределы бессодержательного переназначения «отсутствующей задачи» на незанятые времена; такие расширения не изменяют стоимость S, поэтому оно должно быть оптимальным.

То же самое, допустим, что последнее сделанное в расписание добавление находится на итерации i. До того, как было запланировано последнее задание, S_{i-1} было перспективным. Кроме того, это последнее задание было единственным оставшимся заданием, которое могло реально расширить S, так как ни одно из них не было запланировано. Очевидно, что прибыль, полученная от планирования этого задания, является одинаковой независимо от того, когда оно заплани-

ровано, поэтому каждое расширение S_{i-1} имеет ту же прибыль, что и S_i.

Задача 2.22. Поскольку $S' = S$ и $S'_{opt} = S_{opt}$, мы должны показать, что S расширяемо до S_{opt} заданиями в $\{i + 1, i + 2, \ldots, n\}$. Поскольку задание i не могло быть запланировано в S и S_{opt} расширяет S (то есть S_{opt} имеет все задания, которые имело S, и, возможно, больше), из этого следует, что i тоже не могло быть в S_{opt}, и поэтому i не было необходимо в расширении S до S_{opt}.

Задача 2.24. Почему № 1. Для того чтобы показать, что S'_{opt} является допустимым, мы должны показать, что никакое задание не запланировано дважды и никакое задание не запланировано после его предельного срока. Первое сделать легко, потому что S_{opt} было допустимым. Что касается второго, то мы аргументируем так: задание, которое было в момент времени t_0, теперь перемещено в $t_1 < t_0$, поэтому, безусловно, если t_0 было до его предельного срока, то же самое касается и t_1. Задание, которое было в момент времени t_i (задание i), теперь перенесено вперед на момент времени t_0, но мы работаем, исходя из допущения, что задание i было запланировано (на этот момент времени) в ячейке t_0, поэтому $t_0 \leqslant d_i$, и, значит, мы ответили. **Почему № 2.** S'_{opt} расширяет S', потому что S_{opt} расширяет S, и единственное различие между ними состоит в позиции t_1 и t_0. Они совпадают в позиции t_0 (обе имеют i), поэтому нам остается только исследовать позицию t_1. Но $S(t_1) = 0$, так как $S_{opt}(t_1) = i$, и S не планирует задание i вообще. Так как единственная разница между S и S' находится в позиции t_0, то из этого следует, что $S'(t_1) = 0$, поэтому не имеет значения, каким является $S'_{opt}(t_1)$, оно будет расширять S'. **Почему № 3.** Они планируют одно и то же множество заданий, поэтому они должны иметь одинаковую прибыль. **Почему № 4.** Предположим, $t_1 > t_0$. Так как S_{opt} расширяет S, из этого следует, что $S(t_1) = 0$. Так как $S_{opt}(t_1) = i$, то из этого следует, что $t_1 \leqslant d_i$. Но тогда алгоритм запланировал бы i в t_1, а не в t_0.

Дело в том, что $j \neq 0$ используется в последнем предложении доказательства утверждения 2.23, где мы делаем вывод о существовании противоречия из $S(t_2) = j \neq S_{opt}(t_2)$. Если бы j был равен 0, то вполне могло быть, что $S(t_2) = j = 0$, но $S_{opt}(T_2) \neq 0$.

Задача 2.27. При номиналах {1, 10, 25, 100} существует ряд значений, для которых алгоритм 2.4 не дает оптимального решения. Рассмотрим, например, случай $n = 33$. Алгоритм 2.4 предоставляет решение {25, 1, 1, 1, 1, 1, 1, 1, 1} (которое содержит 9 «монет»), тогда как оптимальное решение равно {10, 10, 10, 1, 1, 1} с мощностью 6.

Задача 2.28. Дадим определение *перспективного списка* как список, который можно расширить до оптимального списка монет. Теперь покажем, что L является перспективным и инвариантом цикла. Базовый случай: изначально L является пустым, поэтому любое оптимальное решение расширяет L. Следовательно, L является перспективным. Индукционный шаг: допустим, что L является перспективным, и покажем, что L остается перспективным после еще одного исполнения цикла: предположим, что L является перспективным и $s < N$. Пусть L' равно списку, который расширяет L до оптимального решения, то есть L, $L' = L_{opt}$. Пусть x равно самому крупному элементу в C такому, что $s + x \leqslant N$. **Случай (a):** $x \in L'$. Тогда $L' = x$, L'', благодаря чему L, x может быть расширено до оптимального решения L_{opt} списком L''. **Случай (b):** $x \notin L'$. Мы показываем, что этот случай невозможен. С этой целью мы доказываем следующее утверждение.

Утверждение: если $x \notin L'$, то существует подсписок L_0 списка L' такой, что x = сумме элементов в L_0.

Доказательство утверждения: пусть B равно наименьшему числу такому, что $B \geqslant x$, и некий подсписок L' суммируется в число B. Пусть этот подсписок равен $\{e_1, e_2, \ldots, e_l\}$, где $e_i \leqslant e_{i+1}$ (то есть элементы находятся в неубывающем порядке). Так как x является самой крупной монетой, которая укладывается в $N - s$, и сумма монет в L' равна $N - s$, то получается, что каждая монета в L' будет $\leqslant x$. Так как $e_l \neq x$ (поскольку $x \notin L'$), то из этого следует, что $l > 1$. Пусть $D = x - (e_2 + \ldots + e_l)$. По определению B мы знаем, что $D > 0$. Каждое из чисел $x, e_2, \ldots, e_l$ делится на e_1 (чтобы это увидеть, обратите внимание, что все монеты являются степенями p, то есть во множестве $\{1, p, p_2, \ldots, p_n\}$, и $e_l < x$, поэтому $e_1 < x$). Следовательно $D \geqslant e_1$. С другой стороны $x \leqslant e_1 + e_2 + \ldots + e_l$, поэтому мы также знаем, что $D \leqslant e_1$, значит, на самом деле $D = e_1$. Поэтому $x = e_1 + e_2 + \ldots + e_l$, и доказательство завершено. (Конец доказательства утверждения.)

Таким образом, $\{e_1, e_2, \ldots, e_l\}$ может быть заменено одной монетой x. Если $l = 1$, то $x = e_1 \in L'$, что является противоречием. Если $l > 1$, то

$$L, x, L' - \{e_1, e_2, \ldots, e_l\}$$

суммируется в N, но у него меньше монет, чем L, $L' = L_{\text{opt}}$, что является противоречием. Следовательно, случай (b) невозможен.

Задача 2.29. См. алгоритм 2.6.

Алгоритм 2.6. Решение задачи 2.29

```
w(e_1) ⩾ w(e_2) ⩾ ... ⩾ w(e_m)
M ← ∅
for i : 1..m do
        if M ∪ {e_i} не содержит двух ребер с общей вершиной
        then
                M ← M ∪ {e_i}
        end if
end for
```

Задача 2.31. Пусть M_{opt} равно оптимальному паросочетанию. Определим утверждение «M является перспективным» как означающее, что M может быть расширено до M_{opt} ребрами, которые еще не рассматривались. Мы показываем, что утверждение «M является перспективным» является инвариантом цикла нашего алгоритма. Результат будет следовать из него (также следует, что существует уникальное максимальное паросочетание). Базовый случай: $M = \emptyset$, поэтому оно, безусловно, является перспективным. Индукционный шаг: допустим, что M является перспективным, и пусть M' равно M после рассмотрения ребра e_i. Мы показываем, что $e_i \in M' \Leftrightarrow e_i \in M_{\text{opt}}$.

[⇒] Допустим, что $e_i \in M'$, поскольку веса не совпадают, и степени числа 2, $w(e_i) > \sum_{j=i+1}^{m} w(e_j)$ (для того чтобы увидеть, почему это справедливо, см. задачу 9.1), поэтому если только не является истинным, что $e_i \in M_{\text{opt}}$, $w(M_{\text{opt}}) < w$, где w – это результат алгоритма.

[⇐] Допустим, что $e_i \in M_{\text{opt}}$, поэтому $M \cup \{e_i\}$ не имеет конфликта, поэтому алгоритм его добавит.

Задача 2.32. Эта задача относится к алгоритму Дейкстры кратчайшего пути; для получения дополнительной информации см. § 24.3, стр. 658, в книге [Cormen и соавт. (2009)] и § 4.4, стр. 137, в книге [Kleinberg и Tardos (2006)]. Доказательство простое: определим S как перспективный, если для всех узлов v в S $d(v)$ действительно является кратчайшим расстоянием из s в v. Теперь нам нужно показать по индукции на числе итераций алгоритма, что выражение «S является перспективным» является инвариантом цикла. Базовый случай $S = \{s\}$ и $d(s) = 0$, поэтому он, очевидно, соблюдается. На индукционном шаге предположим, что v – это только что добавленный узел, поэтому $S' = S \cup \{v\}$. Предположим, что существует более короткий путь в графе G из s в v; назовем этот путь p (то есть p – это просто последовательность узлов, начинающаяся в s и заканчивающаяся в v). Поскольку p начинается внутри S (в s) и заканчивается за пределами S (в v), из этого следует, что существует ребро (a, b) такое, что a, b – это следующие друг за другом узлы на p, где a находится в S и b находится в $V – S$. Пусть $c(p)$ равно стоимости пути p, и пусть $d'(v)$ равно стоимости, которую нашел алгоритм; мы имеем $c(p) < d'(v)$. Теперь рассмотрим два случая: $b = v$ и $b \neq v$, и убедимся, что оба дают противоречие. Если $b = v$, то алгоритм использовал бы a вместо u. Если $b \neq v$, то стоимость пути из s в b будет даже меньше, чем $c(p)$, поэтому алгоритм добавил бы b вместо v. Следовательно, такого пути p не существует.

2.5. Примечания

В любой книге по алгоритмам есть глава о жадных алгоритмах. Например, глава 16 в книге [Cormen и соавт. (2009)] или глава 4 в книге [Kleinberg и Tardos (2006)].

В задаче 2.5 обсуждается сложность алгоритма Краскала, которая зависит от того, какой алгоритм сортировки используется для расстановки ребер в порядке стоимостей. Упоминается сортировка вставками (каждый элемент в списке вставляется в свою правильную позицию), но есть и ряд других алгоритмов сортировки. Есть также сортировка выбором (найти минимальное значение, поменять его местами со значением в первой позиции и повторить), сортировка слиянием (см. раздел 3.1), пирамидальная сортировка (как и сортировка выбором, но с использованием кучи для эффективности), быстрая сортировка (выбрать элемент, расположить все меньшие элементы перед ним, все более крупные элементы после него и повторять на этих двух частях – следовательно, как и сортировка слиянием, этот алгоритм относится к категории алгоритмов «разделяй и властвуй»), сортировка пузырьком (начинается вначале и сравнивает первые два элемента, и если первый больше второго, меняет их местами и продолжает для каждой пары смежных элементов до конца, начинается снова с первых двух элементов, повторяя до тех пор, пока на последнем проходе не произойдет ни одной перемены мест). Есть еще ряд других вариантов.

Мы также отмечаем, что существует глубокая связь между математической структурой, именуемой *матроидами*, и жадными алгоритмами. Матроид, также именуемый *структурой независимости*, выражает понятие «независимости», как и понятие независимости в линейной алгебре.

Матроид M представляет собой пару (E, I), где E – это конечное множество и I – коллекция подмножеств E (именуемых независимыми множествами) со следующими тремя свойствами:

i) пустое множество находится в I, то есть $\emptyset \in I$;
ii) каждое подмножество независимого множества также независимо, то есть если $x \subseteq y$, то $y \in I \Rightarrow x \in I$;
iii) если x и y являются двумя независимыми множествами и x имеет больше элементов, чем y, то существует элемент в x не в y, который при добавлении в y по-прежнему дает независимое множество. Это свойство называется *свойством замещения независимого множества*.

Последнее свойство, конечно, напоминает нашу лемму о замене, лемма 2.11.

Хороший способ понять смысл этого определения – думать о E как о множестве векторов (в $\mathbb{R}^n$) и об I как о всех подмножествах E, состоящих из линейно независимых векторов; проверьте, что все три свойства соблюдаются.

Обзор связи между матроидами и жадными алгоритмами см. в книге [Papadimitriou и Steiglitz (1998)], глава 12 «Остовные деревья и матроиды».

Для изучения того, какие оптимизационные задачи могут оптимально или приближенно решаться «жадными» алгоритмами, см. [Allan Borodin (2003)].

Хорошо известным алгоритмом вычисления максимального паросочетания в двудольном графе является а*лгоритм Хопкрофта–Карпа*; см., например, книгу [Cormen и соавт. (2009)]. Этот алгоритм работает за полиномиальное время (то есть эффективно), но он не жадный – жадный подход, по-видимому, будет безуспешным, как намекает раздел 2.3.2.

Глава 3

Разделяй и властвуй

Si vis pacem, para bellum (Хочешь мира – готовься к войне).
De Re Militari
[Ренатус (IV или V век нашей эры)]

Divide et impera (разделяй и властвуй) – это римская военная стратегия, состоявшая в обеспечении команды по разделению большой концентрации военной мощи на части, которые сами по себе были слабее, и методичном уничтожении этих частей одна за другой. В этом состоит идея алгоритмов «разделяй и властвуй»: взять большую задачу, разделить ее на более мелкие части, решить эти части *рекурсивно* и объединить эти решения в решение целого.

Парадигматическим примером алгоритма «разделяй и властвуй» является сортировка слиянием, где у нас есть большой список элементов для сортировки; мы разбиваем его на два меньших списка (разделяй), сортируем их рекурсивно (властвуй), а затем объединяем эти два отсортированных списка в один большой отсортированный список. Мы представим этот алгоритм в разделе 3.1. Мы также представим алгоритм «разделяй и властвуй» для умножения двоичных целых чисел – раздел 3.2 и алгоритм достижимости в графе – раздел 3.3.

Подход «разделяй и властвуй» часто используется в ситуациях, когда есть алгоритм грубой силы/исчерпывающего поиска, который решает задачу, но алгоритм «разделяй и властвуй» улучшает время выполнения. Это, в частности, касается двоичного целочисленного умножения. Последним примером в этой главе является алгоритм «разделяй и властвуй» достижимости (алгоритм Саввича), который минимизирует использование памяти, а не время работы.

Для того чтобы проанализировать использование ресурсов (будь то время или пространство) рекурсивной процедуры, мы должны решить рекуррентные соотношения; см., например, книги [Rosen (2007)] или [Cormen и соавт. (2009)] для получения необходимой общей информации, в частности по «основному методу» для решения рекуррентных соотношений. Мы предлагаем краткое обсуждение данной темы в разделе примечаний в конце этой главы.

3.1. Сортировка слиянием

Предположим, что у нас есть два списка чисел, которые уже отсортированы. То есть у нас есть список $a_1 \leqslant a_2 \leqslant \cdots \leqslant a_n$ и $b_1 \leqslant b_2 \leqslant \cdots \leqslant b_m$. Мы хотим объединить эти два списка в один длинный отсортированный список $c_1 \leqslant c_2 \leqslant \cdots \leqslant c_{n+m}$. Алгоритм 3.1 выполняет эту работу.

Алгоритм 3.1. Выполнить слияние двух списков

Предусловие: $a_1 \leqslant a_2 \leqslant \cdots \leqslant a_n$ и $b_1 \leqslant b_2 \leqslant \cdots \leqslant b_m$

```
p1 ← 1; p2 ← 1; i ← 1
while i ⩽ n + m do
        if a_{p1} ⩽ b_{p2} then
                c_i ← a_{p1}
                p1 ← p1 + 1
        else
                c_i ← b_{p1}
                p2 ← p2 + 1
        end if
        i ← i + 1
end while
```

Постусловие: $c_1 \leqslant c_2 \leqslant \cdots \leqslant c_{n+m}$

Задача 3.1. Обратите внимание, что алгоритм 3.1 в представленном виде является неполным; например, предположим, что $n < m$ и все элементы списка a_i меньше b_1. В этом случае после n-й итерации цикла while $p_1 = n + 1$, и еще одна итерация проверяет выражение $a_{p_1} \leqslant b_{p_2}$, в результате чего получается ошибка «индекс за пределами границ». Модифицируйте алгоритм, чтобы исправить эту ошибку.

Алгоритм сортировки слиянием сортирует заданный список чисел, сначала разделяя их на два списка соответственно длины $\lceil n/2 \rceil$ и $\lfloor n/2 \rfloor$, затем рекурсивно сортируя каждый список и, наконец, объединяя результаты с помощью алгоритма 3.1.

В алгоритме 3.2 строка 1 устанавливает L равным списку входных чисел $a_1, a_2, \ldots, a_n$. Они являются целыми числами, не обязательно упорядоченными. Строка 2 проверяет, является ли L пустым или состоит из одного элемента; если это так, то список уже отсортирован – это та ситуация, где рекурсия «опускается», возвращая тот же самый список. В противном случае в строке 5 мы назначаем L_1 первые $\lceil n/2 \rceil$ элементов L и L_2 – последние $\lfloor n/2 \rfloor$ элементов L.

Задача 3.2. Покажите, что $L = L_1 \cup L_2$.

Алгоритм 3.2. Сортировка слиянием

Предусловие: список целых чисел $a_1, a_2, \ldots, a_n$

```
1: L ← a_1, a_2, ..., a_n
2: if |L| ⩽ 1 then
3:         return L
4: else
5:         L1 ← первые ⌈n/2⌉ элементов списка L
6:         L2 ← последние ⌊n/2⌋ элементов списка L
7:         return Merge(Mergesort(L1), Mergesort(L2))
8: end if
```

Постусловие: $a_{i_1} \leqslant a_{i_2} \leqslant \cdots \leqslant a_{i_n}$

В разделе 9.3.6 мы покажем, как использовать теорию неподвижных точек для

доказательства правильности рекурсивных алгоритмов. Для нас это останется теоретической демонстрацией, так как нелегко придумать наименьшую неподвижную точку, которая интерпретирует рекурсию. Мы дадим естественные доказательства правильности с помощью индукции.

Задача 3.3. Докажите правильность алгоритма сортировки слиянием с учетом вашего решения задачи 3.1.

Пусть $T(n)$ ограничивает время работы алгоритма сортировки слиянием на списках длины n. Ясно, что

$$T(n) \leqslant T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + cn,$$

где cn для некоторой константы c – это стоимость слияния двух списков (алгоритм 3.1). Более того, асимптотические границы не затрагиваются функциями округления вверх и вниз, и поэтому мы можем просто сказать, что $T(n) \leqslant 2T(n/2) + cn$. Следовательно, $T(n)$ ограничена сложностью $O(n \log n)$.

Задача 3.4. Реализуйте сортировку слиянием для сортировки списка слов в лексикографическом порядке.

3.2. Умножение двоичных чисел

Рассмотрим пример умножения двух двоичных чисел, используя алгоритм начальной школы, то есть лесенкой, приведенный на рис. 3.1.

	1	2	3	4	5	6	7	8
x					1	1	1	0
y					1	1	0	1
s_1					1	1	1	0
s_2				0	0	0	0	
s_3			1	1	1	0		
s_4		1	1	1	0			
$x \times y$	1	0	1	1	0	1	1	0

Рис. 3.1 ❖ Умножьте 1110 на 1101, т. е. 14 на 13

Этот школьный алгоритм умножения очень прост. Для того чтобы умножить x на y, где x, y – это два двоичных числа, мы проходим по y справа налево; когда мы встречаем 0, то пишем строку из стольких нулей, сколько $|x|$, длина x. Когда мы встречаем 1, мы копируем x. Когда мы переходим к следующему биту числа y, мы сдвигаемся на одну позицию влево. В итоге мы производим знакомую форму «лесенки» – см. s_1, s_2, s_3, s_4 на рис. 3.1 (отныне рис. 3.1 является нашим текущим примером двоичного умножения).

После того как мы получим «лесенку», мы вернемся к верхней ступеньке (строка s_1) и к ее самому правому биту (столбец 8). Для того чтобы получить произведение, мы складываем все элементы в каждом столбце с помощью обычной операции переноса. Например, столбец 5 содержит две единицы, поэтому мы пишем 0 в последней строке (строка $x \times y$) и переносим 1 в столбец 4. Нетрудно заметить, что на умножение двух n-битных целых чисел требуется $O(n^2)$ примитивных би-

товых операций.

Теперь мы представим алгоритм «разделяй и властвуй», который требует только $O(n^{\log 3}) \approx O(n^{1,59})$ операций. Ускорение, полученное из процедуры «разделяй и властвуй», выглядит небольшим, но это улучшение действительно становится значительным, когда n вырастает до очень крупных размеров.

Пусть x и y равны двум n-битным целым числам. Мы разбиваем их на два меньших $n/2$-битных целых числа следующим образом:

$$x = (x_1 \cdot 2^{n/2} + x_0),$$
$$y = (y_1 \cdot 2^{n/2} + y_0).$$

Тем самым x_1 и y_1 соответствуют старшим битам x и y и x_0 и y_0 соответствуют младшим битам x и y. В терминах этих частей произведение x и y выглядит следующим образом:

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0. \end{aligned} \tag{3.1}$$

Процедура «разделяй и властвуй» проявляется скрытно. Для того чтобы вычислить произведение x и y, мы *рекурсивно* вычисляем четыре произведения x_1y_1, x_1y_0, x_0y_1, x_0y_0, и затем мы объединяем их в (3.1) и получаем xy.

Пусть $T(n)$ равно числу операций, необходимых для вычисления произведения двух n-битных целых чисел с помощью процедуры «разделяй и властвуй», которая получается в результате (3.1). Тогда

$$T(n) \leqslant 4T(n/2) + cn, \tag{3.2}$$

так как мы должны вычислить четыре произведения x_1y_1, x_1y_0, x_0y_1, x_0y_0 (вот откуда берется составляющая $4T(n/2)$), а затем мы должны выполнить три сложения n-битных целых чисел (отсюда берется составляющая cn, где c – это некоторая константа). Обратите внимание, что мы не учли произведение на 2^n и $2^{n/2}$ (в (3.1)), так как они просто состоят в сдвиге двоичной строки на соответствующее число бит влево (n для 2^n и $n/2$ для $2^{n/2}$). Эти операции сдвига дешевы и могут игнорироваться при анализе сложности.

Когда мы решаем стандартное рекуррентное соотношение, заданное в (3.2), мы видим, что $T(n) \leqslant O(n^{\log 4}) = O(n^2)$, поэтому кажется, что мы ничего не получили, по сравнению с процедурой грубой силы.

Из (3.1) следует, что мы должны сделать четыре рекурсивных вызова; то есть нам нужно вычислить четыре умножения x_1y_1, x_1y_0, x_0y_1, x_0y_0. Но мы можем обойтись только тремя умножениями и, следовательно, тремя рекурсивными вызовами: x_1y_1, x_0y_0 и $(x_1 + x_0)(y_1 + y_0)$; причина в том, что

$$(x_1 y_0 + x_0 y_1) = (x_1 + x_0)(y_1 + y_0) - (x_1 y_1 + x_0 y_0). \tag{3.3}$$

См. рис. 3.2 со сравнением стоимости операций.

	Умножение	Сложение	Сдвиги
Метод (3.1)	4	3	2
Метод (3.3)	3	4	2

Рис. 3. Сокращение числа умножений на одно увеличивает число сложений и вычитаний на одно – чем-то нужно пожертвовать. Но,

поскольку умножения стоят дороже, сделка стоит того

Алгоритм 3.3 реализует идею, задаваемую в (3.3).

Обратите внимание, что в строках 1 и 2 алгоритма мы разбиваем x и y на две части соответственно x_1, x_0 и y_1, y_0, где x_1, y_1 состоят из $\lfloor n/2 \rfloor$ старших бит и x_0, y_0 состоят из $\lceil n/2 \rceil$ младших бит.

Задача 3.5. Докажите правильность алгоритма 3.3.

Алгоритм 3.3 очевидным образом требует $T(n) \leqslant 3T(n/2) + dn$ операций. Следовательно, его время работы составляет $O(n^{\log 3}) \approx O(n^{1,59})$ – для того чтобы в этом убедиться, прочитайте обсуждение решения рекуррентных соотношений в разделе примечаний этой главы.

Алгоритм 3.3. Рекурсивное двоичное умножение

Предусловие: два n-битных целых числа x и y

```
 1: if n = 1 then
 2:         if x = 1 ∧ y = 1 then
 3:                 return 1
 4:         else
 5:                 return 0
 6:         end if
 7: end if
 8: (x_1, x_0) ← (первые ⌈n/2⌉ бит, последние ⌊n/2⌋ бит) числа x
 9: (y_1, y_0) ← (первые ⌈n/2⌉ бит, последние ⌊n/2⌋ бит) числа y
10: z_1 ← Умножить(x_1 + x_0, y_1 + y_0)
11: z_2 ← Умножить(x_1, y_1)
12: z_3 ← Умножить(x_0, y_0)
13: return z_2 · 2^n + (z_1 − z_2 − z_3) · 2^⌈n/2⌉ + z_3
```

Задача 3.6. Реализуйте алгоритм двоичного умножения. Исходите из того, что входные данные задаются в командной строке в виде двух цепочек нулей и единиц.

3.3. Алгоритм Савича

В этом разделе мы дадим решение «разделяй и властвуй» *задачи достижимости в графе*. Вспомните теоретико-графовые определения, которые были даны в начале раздела 2.1. Здесь мы будем считать, что имеем (ориентированный) граф G, и мы хотим установить, существует ли путь из узла s в узел t; обратите внимание, что мы даже не выполняем поиск кратчайшего пути (как в разделе 2.3.3 или в разделе 4.2); мы просто хотим знать, достижим ли узел t из узла s.

В качестве изюминки по минимизации времени работы алгоритмов мы представим очень умное решение «разделяй и властвуй», которое резко сокращает объем пространства, то есть памяти. Алгоритм Савича решает ориентированную достижимость в пространстве $O(\log^2 n)$, где n – это число вершин в графе. Он заслуживает внимания, так как $O(\log^2 n)$ бит памяти действительно дает очень малое пространство для графа с n вершинами! Мы исходим из того, что граф представлен в виде матрицы смежности размера $n \times n$ (см. стр. 29), и поэтому он занимает

ровно n^2 бит памяти, то есть «рабочей памяти», которую мы используем для реализации стека.

Может показаться бесполезным рекомендовать алгоритм, который занимает $O(\log^2 n)$ бит пространства, когда сами входные данные требуют n^2 бит. Если входные данные уже занимают так много места, какая выгода в том, что требуется небольшое пространство для вычислений? Дело в том, что входные данные не обязательно должны быть представлены полностью. Вместо явного представления граф может быть задан неявно. Например, «граф» $G = (V, E)$ может быть всей *Всемирной паутиной* (WWW) целиком, где V – это множество всех веб-страниц (в данный момент времени), и со страницы x на страницу y имеется ребро, если в x есть *гиперссылка*, указывающая на y. Мы можем быть заинтересованы в существовании пути в WWW, и мы можем запрашивать страницы и их ссылки по частям, не сохраняя представление всего WWW в памяти. Сам размер WWW является таким, что может оказаться полезным знать, что нам требуется лишь столько места, сколько составляет квадрат логарифма числа веб-страниц.

Кстати, мы не говорим, что алгоритм Савича является идеальным решением «*задачи связности гиперссылок WWW*»; мы просто приводим пример огромного графа и алгоритма, который использует очень мало рабочего пространства по отношению к размеру входных данных.

Определим булев предикат $R(G, u, v, i)$ как истинный тогда и только тогда, когда в G существует путь из u в v длиной не более 2^i. Ключевая идея состоит в том, что если существует путь из u в v, то любой такой путь должен иметь срединную точку w; казалось бы, тривиальное наблюдение, которое тем не менее обусловливает очень умную рекурсивную процедуру. Другими словами, существуют пути расстояния не более 2^{i-1} из u в w и из w в v, то есть

$$R(G, u, v, i) \Leftrightarrow (\exists w)[R(G, u, w, i-1) \wedge R(G, w, v, i-1)]. \tag{3.4}$$

Алгоритм 3.4 вычисляет предикат $R(G, u, v, i)$ на основе рекуррентного соотношения, указанного в (3.4). Обратите внимание, что в алгоритме 3.4 мы вычисляем $R(G, u, v, i)$; рекурсивные вызовы выполняются в строке 9, где мы вычисляем $R(G, u, w, i-1)$ и $R(G, w, v, i-1)$.

Алгоритм 3.4. Савич

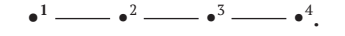
```
 1: if i = 0 then
 2:         if u = v then
 3:                 return ⊤
 4:         else if (u, v) является ребром then
 5:                 return ⊤
 6:         end if
 7: else
 8:         for каждая вершина w do
 9:                 if R(G, u, w, i − 1) и R(G, w, v, i − 1) then
10:                         return ⊤
11:                 end if
12:         end for
```

13: **end if**
14: **return** F

Задача 3.7. Покажите, что алгоритм 3.4 является правильным (то есть он вычисляет $R(G, u, v, i)$ правильно) и требует не более $i \cdot s$ пространства, где s – это число бит, необходимых для учета одного узла. Сделайте вывод, что он требует $O(\log^2 n)$ пространства на графе G с n узлами.

Задача 3.8. Алгоритм 3.4 действительно использует очень мало места для установления связности в графе. Но какова временная сложность этого алгоритма?

Задача 3.9. Ваша задача – написать программу, реализующую алгоритм Савича, такую, что на каждом шаге будет выводиться содержимое стека рекурсии. Предположим, например, что на вход алгоритма поступает следующий граф:



Тогда стек рекурсии будет выглядеть для первых 6 шагов следующим образом:

		$R(1,4,0)$	F	$R(2,4,0)$	F
		$R(1,1,0)$	T	$R(1,2,0)$	T
	$R(1,4,1)$	$R(1,4,1)$	$R(1,4,1)$	$R(1,4,1)$	$R(1,4,1)$
	$R(1,1,1)$	$R(1,1,1)$	$R(1,1,1)$	$R(1,1,1)$	$R(1,1,1)$
$R(1,4,2)$	$R(1,4,2)$	$R(1,4,2)$	$R(1,4,2)$	$R(1,4,2)$	$R(1,4,2)$
Шаг 1	Шаг 2	Шаг 3	Шаг 4	Шаг 5	Шаг 6

3.4. Дальнейшие примеры и задачи

3.4.1. Расширенный алгоритм Евклида

Мы возвращаемся к старому другу из раздела 1.1, а именно к расширенному алгоритму Евклида – см. задачу 1.9 и соответствующее решение на стр. 31, содержащее алгоритм 1.8. Мы представим рекурсивную версию как алгоритм 3.5, где этот алгоритм возвращает три значения, и, следовательно, мы используем форму записи $(x, y, z) \leftarrow (x', y', z')$ в качестве удобного сокращения для $x \leftarrow x'$, $y \leftarrow y'$ и $z \leftarrow z'$. Обратите внимание на интересное сходство между алгоритмом 1.8 и алгоритмом гауссовой редукции решетки 7.3.

Задача 3.10. Покажите, что алгоритм 3.5 работает правильно.

Алгоритм 3.5. Расширенный алгоритм Евклида (рекурсивный)

Предусловие: $m > 0$, $n \geqslant 0$
1: $a \leftarrow m$; $b \leftarrow n$
2: **if** $b = 0$ **then**
3: **return** $(a, 1, 0)$
4: **else**
5: $(d, x, y) \leftarrow \text{Euclid}(b, \text{rem}(a, b))$
6: **return** $(d, y, x - \text{div}(a, b) \cdot y)$
7: **end if**
Постусловие: $mx + ny = d = \gcd(m, n)$

Задача 3.11. Реализуйте расширенный алгоритм Евклида (алгоритм 3.5). Исходите из того, что входные данные заданы в командной строке в виде двух целых чисел.

3.4.2. Быстрая сортировка

Быстрая сортировка является широко применяемым алгоритмом сортировки. Он был разработан в конце 1950-х годов Т. Хоаром[1]. Быстрая сортировка легко определяется: для того чтобы отсортировать элементы списка I, следует выбрать один элемент x из I (назовем этот элемент опорным) и создать два новых списка, S и L: это все те элементы, которые меньше или равны опорному, S, и все те элементы, которые больше опорного, L. Теперь рекурсивно отсортировать S и L, создав S' и L', и новый отсортированный список I' задается членами S', x, L'.

Интересно отметить, что быструю сортировку можно легко реализовать на функциональном языке, так как он работает со списками и она естественным образом будет являться рекурсивной функцией. Например, быстрая сортировка может быть реализована на языке Haskell в нескольких строках кода следующим образом:

```
qsort [] = []
qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger
  where
    smaller = [a | a <- xs, a <= x]
    larger = [b | b <- xs, b > x]
```

Обратите внимание, что в этой реализации (см. стр. 10 книги [Hutton (2007)]) в качестве опоры мы выбрали первый элемент списка; существует рандомизированная версия быстрой сортировки, где опорный элемент выбирается из списка случайно.

Задача 3.12. Реализуйте быструю сортировку и проанализируйте ее сложность. Для сложности укажите время выполнения в худшем и в среднем случаях.

3.4.3. Команда git bisect

Git – это широко используемая программа для управления версиями компьютерных файлов, а также для координации сотрудничества групп людей, работающих над большим программным проектом[2]. На самом деле решения задач программирования, содержащиеся в этой книге, а также реализации всех алгоритмов поддерживаются в общедоступном репозитории Git на GitHub (веб-репозитории Git): https://github.com/michaelsoltys/IAA-Code.

Как объясняется в документации Git, команда `git bisect` использует алгоритм бинарного поиска для отыскания фиксации (коммита) в истории проекта, кото-

---

[1] Это тот же Хоар, который уже цитировался на стр. 10; напомним также, что на стр. 13 мы ввели логику Хоара как механизм доказательства правильности алгоритма и программы.

[2] «Программирование» понимается здесь в широком смысле, так как оно может означать что угодно, от работы над ядром Linux до разработки веб-сайтов и до сотрудничества LaTeX. См. https://git-scm.com для получения дополнительной информации.

рая внесла ошибку. Для того чтобы обратить эту команду вспять, пользователь задает «плохую» фиксацию с заранее известной ошибкой и «хорошую» фиксацию, о которой известно, что она существовала до внесения ошибки. Затем команда `git bisect` выбирает фиксацию между этими двумя конечными точками и спрашивает, является выбранная фиксация «хорошей» или «плохой». Она продолжает сужать диапазон до тех пор, пока не найдет точную фиксацию, которая внесла изменение.

По сути дела, команду `git bisect` можно использовать для поиска фиксации, которая изменила любое свойство проекта; например, фиксацию, которая исправила ошибку, или фиксацию, которая вызвала улучшение результативности эталонных тестов. Для того чтобы поддержать это более общее использование, термины «старый» и «новый» могут применяться вместо терминов «хороший» и «плохой», или могут использоваться любые другие термины.

3.5. Ответы к избранным задачам

Задача 3.1. Задача возникает только тогда, когда $p_1 > n$ или $p_2 > m$. Мы можем изменить условие цикла while на $p_1 \leqslant n \wedge p_2 \leqslant m$. Конечно, это означает, что цикл while завершится рано, когда один входной список не будет полностью учтен в $C = \{c_1, c_2, \ldots\}$. В связи с этим после первого цикла должен быть добавлен еще один цикл. Если $p_1 \leqslant n$, то он должен назначить оставшиеся элементы из $a_{p_1}, \ldots, a_n$ остальным переменным в C; в противном случае $p_2 \leqslant m$, поэтому $b_{p_2}, \ldots, b_m$ должны быть переданы остатку C.

Более элегантное решение становится доступным, если мы потребуем, чтобы элементы каждого списка были конечными. Перед началом цикла while мы можем просто добавить в конец каждого списка бесконечно большой элемент; ясно, что он никогда не будет оценен как меньший, чем конечное значение, или равный ему в противоположном списке, поэтому он никогда не будет назначен элементу C.

Задача 3.2. Для этого достаточно показать, что $\lceil n/2 \rceil + \lfloor n/2 \rfloor = n$. Если n является четным, то $\lceil n/2 \rceil = \lfloor n/2 \rfloor = n/2$ и $n/2 + n/2 = n$. Если n является нечетным, то $n = 2k + 1$ и поэтому $\lceil n/2 \rceil = k + 1$, тогда как $\lfloor n/2 \rfloor = k$ и $(k + 1) + k = 2k + 1 = n$.

Задача 3.3. Мы должны показать, что для списка целых чисел $L = a_1, a_2, \ldots, a_n$ алгоритм возвращает список L', который состоит из чисел в L в неубывающем порядке. Рекурсия сама по себе свидетельствует о правильной индукции; мы используем принцип полной индукции (см. стр. 210). Если $|L| = 1$, то $L' = L$, и доказательство завершено. В противном случае $|L| = n > 1$, и мы получаем два списка L_1 и L_2 (с длинами $\lceil n/2 \rceil$ и $n - \lceil n/2 \rceil$), которые, по индукционной гипотезе, возвращаются упорядоченными. Теперь осталось доказать правильность процедуры слияния, алгоритм 3.1, что также можно сделать по индукции.

Задача 3.5. Очевидно, что базовый случай правилен; с учетом двух 1-битных целых числа, если оба числа не равны 1, то, по крайней мере, одно из них равно 0, поэтому произведение равно 0.

Предположим, что алгоритм правилен для всех $n < n'$. Тогда умножения для нахождения z_1, z_2 и z_3 являются правильными. Поэтому уравнения (3.1) и (3.3) обеспечивают индуктивное доказательство.

Задача 3.8. $O(2^{\log^2 n}) = O(n^{\log n})$, поэтому временная сложность алгоритма Савича сверхполиномиальна и поэтому не очень хороша.

Задача 3.10. Прежде всего обратите внимание на то, что второй параметр уменьшается при каждом рекурсивном вызове, но по определению остатка он является неотрицательным. Таким образом, по принципу наименьшего числа алгоритм завершается. Мы доказываем частичную правильность по индукции на значении второго аргумента. В базовом случае $n = 0$, поэтому в строке 1 $b \leftarrow n = 0$, поэтому в строке 2 $b = 0$, алгоритм завершается в строке 3 и возвращает $(a, 1, 0) = (m, 1, 0)$, поэтому $mx + ny = m \cdot 1 + n \cdot 0 = m$, и доказательство завершено.

На индукционном шаге мы исходим из того, что рекурсивная процедура возвращает правильные значения для всех пар аргументов, где второй аргумент $< n$ (таким образом мы делаем полную индукцию). Из строк 1 и 5 мы имеем:

$$(d, x, y) \leftarrow \text{Extended-Euclid}(b, \text{rem}(a, b)) \\ = \text{Extended-Euclid}(n, \text{rem}(m, n)).$$

Обратите внимание, что $0 \leqslant \text{rem}(m, n) < n$, и поэтому мы можем применить индукционную гипотезу, и у нас есть:

$$n \cdot x + \text{rem}(m, n) \cdot y = d = \gcd(n, \text{rem}(m, n)).$$

Сначала отметим, что в задаче 1.6 мы имеем, что $d = \gcd(m, n)$. Теперь мы работаем над левой частью уравнения. У нас есть:

$$\begin{aligned} & n \cdot x + \text{rem}(m, n) \cdot y \\ &= n \cdot x + (m - \text{div}(m, n) \cdot n) \cdot y \\ &= m \cdot y + n \cdot (x - \text{div}(m, n) \cdot y) \\ &= m \cdot y + n \cdot (x - \text{div}(a, b) \cdot y), \end{aligned}$$

и доказательство завершено, так как это то, что возвращается в строке 6.

3.6. Примечания

Для получения полного обсуждения сортировки слиянием и двоичного умножения см. соответственно § 5.1 и § 5.5 в книге [Kleinberg и Tardos (2006)]. Сортировка слиянием имеет интересную историю (подробнее см. главу 3 «сортировка» в книге [Christian и Griffiths (2016)]): в 1945 году Джон фон Нейманн написал программу, демонстрирующую мощь компьютера с хранимой программой; поскольку он был гением, программа не только иллюстрировала парадигму хранимой программы, но и представила новый способ сортировки: Mergesort. См. [Katajainen и Traff (1997)], где данный алгоритм изучается во всех подробностях.

Для обсуждения анализа рекурсивных алгоритмов см. раздел 9.3.6.

Для получения дополнительной информации об алгоритме Савича (раздел 3.3) см. теорему 7.5 в работе [Papadimitriou (1994)], § 8.1 в книге [Sipser (2006)] или теорему 2.7 в работе [Kozen (2006)].

Задача достижимости широко распространена в информатике. Предположим, что у нас есть граф G с n вершинами. В разделе 2.3.3 мы представили алгоритм $O(n^2)$, жадный по времени для задачи достижимости, обусловленный Дейкстрой. В этой главе, в разделе 3.3, мы представили алгоритм «разделяй и властвуй», который требует $O(\log^2 n)$ пространства, обусловленный Савичем. В разделе 4.2 мы представим алгоритм динамического программирования, который вычисляет кратчайший путь для всех пар узлов в графе – он обусловлен Флойдом и занимает

время $O(n^3)$. В подразделе 4.2.1 представлен еще один динамический алгоритм Беллмана и Форда (который может справляться с ребрами отрицательного веса). В 2005 году Рейнгольд показал, что неориентированная достижимость может быть вычислена в пространстве $O(\log n)$; см. [Reingold (2005)] с описанием этого замечательного, но трудного результата. Обратите внимание, что алгоритм Рейнгольда работает только для неориентированных графов.

См. главу 7 в книге [Rosen (2007)], где приводится введение в решение рекуррентных соотношений, и §4.5, стр. 93–103, в книге [Cormen и соавт. (2009)], где дается очень тщательное изложение «основного метода» для решения рекуррентных соотношений. Здесь мы включим очень краткое обсуждение; мы хотим решать рекуррентные соотношения следующего вида:

$$T(n) = aT(n/b) + f(n), \tag{3.5}$$

где $a \geqslant 1$ и $b > 1$ – это константы, $f(n)$ – асимптотически положительная функция – имея в виду, что существует n_0 такой, что $f(n) > 0$ для всех $n \geqslant n_0$. Для решения такого рекуррентного соотношения существует три случая.

Случай 1: $f(n) = O(n^{\log_b a - \varepsilon})$ для некой константы $\varepsilon > 0$; в этом случае мы имеем $T(n) = \Theta(n^{\log_b a})$.

Случай 2: $f(n) = \Theta(n^{\log_b a} \log^k n)$ с $k \geqslant 0$; в этом случае мы имеем $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.

Наконец, **случай 3:** $f(n) = \Omega(n^{\log_b a + \varepsilon})$ с $\varepsilon > 0$, и $f(n)$ удовлетворяет *условию регулярности*, а именно что $af(n/b) \leqslant cf(n)$ для некой константы $c < 1$ и всех достаточно больших n; в этом случае $T(n) = \Theta(f(n))$.

Например, рекуррентное соотношение, которое появляется при анализе сортировки слиянием, равно $T(n) = 2T(n/2) + cn$, поэтому $a = 2$ и $b = 2$, и поэтому $\log_b a = \log_2 2 = 1$, и, значит, мы можем сказать, что $f(n) = \Theta(n^{\log_b a} \log^k n) = \Theta(n \log n)$, то есть $k = 1$ в случае 2, и, значит, $T(n) = \Theta(n \log n)$, как было отмечено в анализе.

Глава 4

Динамическое программирование

Не помнящий прошлого обречен его повторять.

Джордж Сантайяна
(George Santayana)

Динамическое программирование – это алгоритмический метод, тесно связанный с подходом «разделяй и властвуй», который мы видели в предыдущей главе. Однако, в то время как подход «разделяй и властвуй», по существу, рекурсивен, и поэтому продвигается «сверху вниз», динамическое программирование работает «снизу вверх».

Алгоритм динамического программирования создает массив связанных, но более простых подзадач, а затем вычисляет решение большой сложной задачи, используя решения более простых подзадач, которые хранятся в массиве. Обычно мы хотим максимизировать прибыль или минимизировать стоимость.

Существует три шага в поиске решения задачи динамического программирования: (i) определить класс подзадач, (ii) дать регуррентное соотношение на основе решения каждой подзадачи в терминах более простых подзадач и (iii) дать алгоритм вычисления рекуррентного соотношения.

4.1. Задача о наибольшей монотонной подпоследовательности

Вход: $d, a_1, a_2, \ldots, a_d \in \mathbb{N}$.
Выход: L = длина самой длинной монотонной неубывающей подпоследовательности.

Обратите внимание, что элементы подпоследовательности не обязательно должны располагаться друг за другом подряд, то есть $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ – это монотонная подпоследовательность, при условии что

$$1 \leqslant i_1 < i_2 < \ldots < i_k \leqslant d,$$
$$a_{i_1} \leqslant a_{i_2} \leqslant \ldots \leqslant a_{i_k}.$$

Например, длина наибольшей монотонной подпоследовательности {4, 6, 5, 9,1} равна 3.

Сначала определим массив подзадач: $R(j)$ = длина самой длинной монотонной подпоследовательности, которая заканчивается в a_j. Ответ может быть извлечен из массива R путем вычисления $L = \max_{1 \leqslant j \leqslant n} R(j)$.

Следующий шаг – найти рекуррентное соотношение. Пусть $R(1) = 1$, и для $j > 1$

$$R(j) = \begin{cases} 1, & \text{если } a_i > a_j \text{ для всех } 1 \leqslant i < j \\ 1 + \max_{1 \leqslant i < j} \{R(i) | a_i \leqslant a_j\} & \text{в противном случае.} \end{cases}$$

Мы завершим написанием алгоритма, который вычисляет R; см. алгоритм 4.1.

Алгоритм 4.1. Наибольшая монотонная подпоследовательность

```
R(1) ← 1
for j : 2..d do
        max ← 0
        for i : 1..j – 1 do
                if R(i) > max и a_i ⩽ a_j then
                        max ← R(i)
                end if
        end for
        R(j) ← max + 1
end for
```

Задача 4.1. После того как мы вычислили все значения массива R, как построить фактическую монотонную неубывающую подпоследовательность длины L?

Задача 4.2. Каковы были бы соответствующие пред- и постусловия приведенных выше алгоритмов? Докажите правильность с приемлемым инвариантом цикла.

Задача 4.3. Рассмотрите следующий вариант задачи о наибольшей монотонной подпоследовательности. Вход равен $d, a_1, a_2, \ldots, a_d \in \mathbb{N}$, но выход равен длине самой длинной подпоследовательности $a_1, a_2, \ldots, a_d$, где любые два следующих подряд члена подпоследовательности отличаются не более чем на 1. Например, наибольшая такая подпоследовательность последовательности {7, 6, 1, 4, 7, 8, 20} равна {7, 6, 7, 8}, поэтому в данном случае ответ будет 4. Дайте решение методом динамического программирования.

Задачи 4.4. Реализуйте алгоритм 4.1; ваша программа должна принимать дополнительный шаговый параметр, назовем его s, где, как и в задаче 4.3, любые два следующих подряд элемента подпоследовательности отличаются не более чем на S, то есть $|a_{i_j} - a_{i_{j+1}}| \leqslant s$, для любого $1 \leqslant j < k$.

4.2. Задача кратчайшего пути для всех пар

Вход: ориентированный граф $G = (V, E)$, $V = \{1, 2, \ldots, n\}$, функция стоимости $C(i, j) \in \mathbb{N}^+ \cup \{\infty\}$, $1 \leqslant i, j \leqslant n$, $C(i, j) = \infty$ если (i, j) не является ребром.
Выход: массив D, где $d(i, j)$ – это длина кратчайшего ориентированного пути из i в j.

Напомним, что мы определили неориентированные графы в разделе 2.1; *ориентированный граф* (или *орграф*) – это граф, в котором ребра имеют направление, то есть ребра являются стрелками. Также напомним, что в разделе 2.3.3 мы дали

жадный алгоритм для вычисления кратчайших путей из обозначенного узла s во все узлы в (неориентированном) графе.

Задача 4.5. Постройте семейство графов $\{G_n\}$, где G_n имеет $O(n)$ узлов и экспоненциально много путей, то есть $\Omega(2^n)$ путей. Тем самым сделайте вывод, что исчерпывающий поиск не является возможным решением «задачи кратчайшего пути для всех пар».

Определим массив подзадач: пусть $A(k, i, j)$ равно длине кратчайшего пути из i в j такого, что все промежуточные узлы на пути находятся в $\{1, 2, \ldots, k\}$. Тогда $A(n, i, j) = D(i, j)$ будет решением. Условно, если $k = 0$, то $[k] = \{1, 2, \ldots, k\} = 0$.

Определим рекуррентное соотношение: сначала инициализируем массив для $k = 0$, $A(0, i, j) = C(i, j)$. Теперь мы хотим вычислить $A(k, i, j)$ для $k > 0$. Проектируя рекуррентное соотношение, обратите внимание, что кратчайший путь между i и j либо включает k, либо его не включает. Допустим, что мы знаем $A(k - 1, r, s)$ для всех r, s. Предположим, узел k не включен. Тогда очевидно, что $A(k, i, j) = A(k - 1, i, j)$. Если, с другой стороны, узел k встречается на кратчайшем пути, то он встречается ровно один раз, поэтому $A(k, i, j) = A(k - 1, i, k) + A(k - 1, k, j)$. Следовательно, длина кратчайшего пути получается, беря минимум из этих двух случаев:

$$A(k, i, j) = \min\{A(k - 1, i, j), A(k - 1, i, k) + A(k - 1, k, j)\}.$$

Напишем алгоритм: оказывается, что нам нужно пространство только для двумерного массива $B(i, j) = A(k, i, j)$, потому что для вычисления $A(k, *, *)$ из $A(k - 1, *, *)$ мы можем перезаписать $A(k - 1, *, *)$.

Нашим решением является алгоритм 4.2, именуемый *алгоритмом Флойда* (или алгоритмом Флойда–Уоршелла). Примечательно, что он выполняется за время $O(n^3)$, где n – это число вершин, тогда как в таком графе может быть до $O(n^2)$ ребер. В строках 1–5 мы инициализируем массив B, то есть устанавливаем его равным C. Обратите внимание, что перед исполнением строки 6 имеет место, что $b(i, j) = A(k - 1, i, j)$ для всех i, j.

Алгоритм 4.2. Флойд

```
 1: for i : 1..n do
 2:         for j : 1..n do
 3:                 B(i, j) ← C(i, j)
 4:         end for
 5: end for
 6: for k : 1..n do
 7:         for i : 1..n do
 8:                 for j : 1..n do
 9:                         B(i, j) ← min{B(i, j), B(i, k) + B(k, j)}
10:                 end for
11:         end for
12: end for
13: return D ← B
```

Задача 4.6. Почему работает метод перезаписи в алгоритме 4.2? Опасение вызвано тем, что $B(i, k)$ или $B(k, j)$, возможно, уже были обновлены (если $k < j$ или $k < i$).

Однако перезапись работает; объясните, почему. Мы могли бы избежать трехмерного массива, вместо него храня два двумерных массива, и тогда перезапись не была бы проблемой вообще; как это будет работать?

Задача 4.7. Каковы приемлемые предусловия и постусловия в алгоритме 4.2? Что такое приемлемый инвариант цикла?

Задача 4.8. Реализуйте алгоритм Флойда, используя двумерный массив и метод перезаписи.

4.2.1. Алгоритм Беллмана–Форда

Предположим, что мы хотим найти кратчайший путь из s в t в ориентированном графе $G = (V, E)$, где ребра имеют неотрицательные стоимости. Пусть $\mathrm{Opt}(i, v)$ обозначает минимальную стоимость i-членного пути из v в t, где i-членный путь – это путь, который использует не более i ребер. Пусть p равно оптимальному i-членному пути со стоимостью $\mathrm{Opt}(i, v)$; если такой p не существует, то мы условно договариваемся, что $\mathrm{Opt}(i, v) = \infty$.

Если p использует $i - 1$ ребер, то $\mathrm{Opt}(i, v) = \mathrm{Opt}(i - 1, v)$, и если p использует i ребер, и первое ребро равно $(v, w) \in E$, тогда $\mathrm{Opt}(i, v) = c(v, w) + \mathrm{Opt}(i - 1, w)$, где $c(v, w)$ – это стоимость ребра (v, w). В результате получаем рекурсивную формулу для $i > 0$: $\mathrm{Opt}(i, v) = \min\{\mathrm{Opt}(i - 1, v), \min_{w \in V}\{c(v, w) + \mathrm{Opt}(i - 1, w)\}\}$.

Задача 4.9. Реализуйте алгоритм Беллмана–Форда.

4.3. Простая задача о рюкзаке

Вход: $w_1, w_2, \ldots, w_d, C \in \mathbb{N}$, где C – это грузоподъемность рюкзака (ранца).
Выход: $\max_S\{K(S) | K(S) \leqslant C\}$, где $S \subseteq [d]$ и $K(S) = \sum_{i \in S} w_i$.

Данная задача является *NP-трудной*[1]. Это означает, что мы ожидаемо не сможем найти алгоритм полиномиального времени, который работает в общем случае. Мы даем решение методом динамического программирования, которое работает для относительно небольшого C; отметим, что, для того чтобы наш метод работал со входами $w_1, \ldots, w_d, C$ должно быть (неотрицательными) целыми числами. Название «простая задача о рюкзаке» нередко упоминается под аббревиатурой SKS (simple knapsack problem).

Определим массив подзадач: рассмотрим первые i весов (то есть $[i]$), суммирующих вплоть до промежуточного весового предела j. Определим булев массив R следующим образом:

$$R(i, j) = \begin{cases} \mathsf{T}, & \text{если } \exists S \subseteq [i] \text{ такой, что } K(S) = j \\ \mathsf{F}, & \text{в противном случае} \end{cases}$$

[1] NP – это класс задач, решаемых за полиномиальное время на *недетерминированной* машине Тьюринга. Задача P является NP-трудной, если каждая задача в NP сводима к P за полиномиальное время, то есть каждая задача в NP может быть эффективно переформулирована в терминах P. Когда задача NP-трудна, это указывает на то, что она, вероятно, *неразрешима*, то есть она не может быть решена эффективно в общем случае. Для получения дополнительной информации см. любую книгу по теме вычислительной сложности, например [Papadimitriou (1994); Sipser (2006); Soltys (2009)].

для $0 \leqslant i \leqslant d$ и $0 \leqslant j \leqslant C$. После вычисления всех значений R мы можем получить решение M следующим образом: $M = \max_{j \leqslant C}\{j | R(d, j) = \mathsf{T}\}$.

Определим рекуррентное соотношение: мы инициализируем $R(0, j) = \mathsf{F}$ для $j = 1, 2, \ldots, C$, и $R(i, 0) = \mathsf{T}$ для $i = 0, 1, \ldots, d$.

Теперь мы определим рекуррентное соотношение для вычисления R, для $i, j > 0$, таким образом, что оно будет зависеть от того, включим мы объект i в рюкзак или нет. Предположим, что мы не включаем объект i. Тогда очевидно, что $R(i, j) = \mathsf{T}$ тогда и только тогда, когда $R(i-1, j) = \mathsf{T}$. Предположим, с другой стороны, что объект i включен. Тогда должно иметь место, что $R(i, j) = \mathsf{T}$ тогда и только тогда, когда $R(i-1, j-w_i) = \mathsf{T}$ и $j - w_i \geqslant 0$, то есть существует подмножество $S \subseteq [i-1]$ такое, что $K(S)$ в точности равно $j - w_i$ (и в этом случае $j \geqslant w_i$). Складывая все это вместе, мы получаем следующее рекуррентное соотношение для $i, j > 0$:

$$R(i, j) = \mathsf{T} \Leftrightarrow R(i-1, j) = \mathsf{T} \vee (j \geqslant w_i \wedge R(i-1, j-w_i) = \mathsf{T}). \tag{4.1}$$

Рисунок 4.1 резюмирует вычисление рекуррентного соотношения.

Наконец, мы проектируем алгоритм 4.3, который использует тот же хитроумный прием экономии пространства, что и в алгоритме 4.2; он использует одномерный массив $S(j)$ для отслеживания двумерного массива $R(i, j)$. Это делается путем перезаписи $R(i, j)$ с помощью $R(i+1, j)$.

В алгоритме 4.3 в строке 1 мы инициализируем массив для $i = j = 0$. В строках 2–4 мы инициализируем массив для $i = 0$ и $j \in \{1, 2, \ldots, C\}$. Обратите внимание, что после каждого исполнения i-членного цикла (строка 5) имеет место, что $s(j) = R(i, j)$ для всех j.

R	0	⋯	$j - w_i$	⋯	j	⋯	c
0	T	F⋯F	F	F⋯F	F	F⋯F	F
	T ⋮ T						
$i-1$	T		**c**		**b**		
i	T				**a**		
	T ⋮ T						
d	T						

Рис. 4.1 ❖ Рекуррентное соотношение, задаваемое равенством (4.1), можно интерпретировать следующим образом: мы выставляем T в клетку с надписью **a** тогда и только тогда, когда удовлетворяется хотя бы одно из следующих двух условий: в позиции прямо над ней имеется T, то есть в клетке с надписью **b** (если мы можем построить j с первыми $i-1$ весами, то, безусловно, мы можем построить j с первыми i весами), либо в клетке с надписью **c** имеется T (если мы можем построить $j - w_i$ с первыми $i-1$ весами, то, безусловно, мы можем построить j с первыми i весами). Также обратите внимание, что для заполнения клетки, помеченной буквой **a**, нам нужно смотреть только на две клетки, и ни одна из этих клеток не находится справа; это будет важно при проектировании алгоритма (алгоритма 4.3)

Задача 4.10. Для отслеживания двумерного массива мы используем одномерный массив, но перезапись не является проблемой; объясните, почему.

Задача 4.11. Логическое утверждение $S(j) = R(i, j)$ может быть доказано по индукции на числе раз, которое исполняется i-членным циклом в алгоритме 4.3. Из этого логического утверждения следует, что после завершения алгоритма $S(j) = R(d, j)$ для всех j. Докажите это формально, предоставив пред- и постусловия, инвариант цикла и стандартное доказательство правильности.

Задача 4.12. Постройте входные данные, для которых алгоритм 4.3 допустил бы ошибку, если бы внутренний цикл «for уменьшая j : C..1 do» (строка 6) был заменен на «для j : 1..C».

Задача 4.13. Реализуйте алгоритм 4.3.

Алгоритм 4.3 является хорошей иллюстрацией мощной идеи *детализации программы*. Начнем с идеи вычисления $R(i, j)$ для всех i, j. Затем мы понимаем, что нам действительно нужно только две строки массива в памяти; для того чтобы вычислить строку i, нам нужно только найти строку $i - 1$. Затем мы развиваем эту идею дальше и видим, что, обновляя строку массива i справа налево, нам вообще не требуется строка $i - 1$, – мы можем сделать это *прямо на месте*. Начиная с надежной идеи и последовательно ее обрезая, мы получаем гладкое решение.

Алгоритм 4.3. Простая задача о рюкзаке

```
 1: S(0) ← T
 2: for j : 1..C do
 3:         S(j) ← F
 4: end for
 5: for i : 1..d do
 6:         for уменьшая j : C..1 do
 7:                 if (j ⩾ w_i и S(j − w_i) = T) then
 8:                         S(j) ← T
 9:                 end if
10:        end for
11: end for
```

Но насколько хорошим является наше решение методом динамического программирования с точки зрения сложности задачи? То есть сколько шагов требуется для вычисления решения *пропорционально* размеру входных данных? Мы должны построить таблицу размера $d \times C$ и ее заполнить, поэтому временна́я сложность нашего решения равна $O(d \cdot C)$. На первый взгляд это кажется приемлемым, но во введении к этому разделу мы говорили, что простая задача о рюкзаке является NP-трудной; что же делать?

Дело в том, что изначально исходят из того, что входные данные заданы в двоичном виде, и для кодирования C в двоичном виде нам потребуется только $\log C$ бит, поэтому число столбцов (C) на самом деле экспоненциально зависит от размера входа ($C = 2^{\log C}$). С другой стороны, d – это число весов, и поскольку эти веса должны быть каким-то образом перечислены, размер списка весов, безус-

ловно, больше, чем d (то есть этот список не может быть закодирован – в общем случае – с использованием $\log d$ бит; он требует, по крайней мере, d бит).

Мы можем сказать только то, что если C имеет размер $O(d^k)$ для некой константы k, то наше решение методом динамического программирования работает за полиномиальное время в размере входных данных. Другими словами, у нас есть эффективное решение для «малых» значений C. Говоря по-другому, до тех пор, пока $|C|$ (размер двоичного кодирования C) равен $O(\log d)$, наше решение работает за полиномиальное время.

Задача 4.14. Покажите, как построить фактическое оптимальное множество весов после вычисления R.

Задача 4.15. Определите «естественный» жадный алгоритм решения простой задачи о рюкзаке; пусть $\overline{M}$ равно выходу этого алгоритма, а M равно выходу решения методом динамического программирования, приведенного в этом разделе. Покажите, что либо $\overline{M} = M$, либо $\overline{M} > \frac{1}{2}C$.

Задача 4.15 скрытно вводит понятие аппроксимационных алгоритмов. Как было упомянуто в начале этого раздела (см. сноску на стр. 75), простая задача о рюкзаке является примером NP-трудной задачи, задачи, для которой по нашим подозрениям в общем случае не может быть эффективного решения. То есть большинство экспертов считает, что любой алгоритм, пытающийся решить простую задачу о рюкзаке в общем случае – на бесконечно большом числе входов, будет занимать чрезмерное число шагов (то есть время) для получения решения.

Один из возможных компромиссов заключается в разработке эффективного алгоритма, не дающего оптимального решения – которого, возможно, даже не потребуется – и дающего только решение с некой гарантией в его близости к оптимальному решению. Таким образом, мы просто аппроксимируем оптимальное решение, но, по крайней мере, наш алгоритм работает быстро. Исследование таких компромиссов осуществляется с помощью аппроксимационных алгоритмов.

Наконец, в приведенном ниже разделе мы даем жадное решение простой задачи о рюкзаке в частном случае, когда веса имеют определенное «возрастающее свойство». Оно является примером перспективной задачи, где мы можем ожидать соблюдения какого-то удобного условия во входных данных; условия, которое нам не нужно проверять, но которое допускается как имеющееся. Обратите внимание, что мы использовали термин «перспективный» для доказательства правильности жадных алгоритмов – это другое понятие, чем «перспективная задача».

4.3.1. Задача о рассредоточенном рюкзаке

Вход: $w_1, \ldots, w_d, C \in \mathbb{N}$, такие, что $w_i \geqslant \sum_{j=i+1}^{d}$ для $i = 1, \ldots, d-1$.
Вывод: $S_{\max} \subseteq [d]$, где $K(S_{\max}) = \max_{S \subseteq [d]} \{K(S) | K(S) \leqslant C\}$.

Задача 4.16. Дайте «естественный» жадный алгоритм, который решает задачу о рассредоточенном рюкзаке, заполняя пробелы в алгоритме 4.4.

Задача 4.17. Дайте определение того, что означает быть «перспективным» для промежуточного решения S в алгоритме 4.4. Покажите, что из инварианта цикла «S является перспективным» следует, что жадный алгоритм дает оптимальное решение. Наконец, покажите, что выражение «S является перспективным» является инвариантом цикла.

Алгоритм 4.4. Рассредоточенный рюкзак

```
S ← ∅
for i : 1..d do
        if _____ then
                _____
        end if
end for
```

4.3.2. Общая задача о рюкзаке

Вход: $w_1, w_2, \ldots, w_d, v_1, \ldots, v_d, C \in \mathbb{N}$.
Выход: $\max\limits_{S \subseteq [d]} \{V(S) | K(S) \leqslant C\}$, $K(S) = \sum_{i \in S} w_i$, $V(S) = \sum_{i \in S} v_i$.

Таким образом, общая задача о рюкзаке имеет положительное целочисленное значение v_i, помимо каждого веса w_i, и цель заключается в том, чтобы получить как можно более ценный рюкзак, без превышения C, то есть грузоподъемности рюкзака.

Точнее, $V(S) = \sum_{i \in S} v_i$ – это суммарное значение множества S весов. Цель состоит в том, чтобы максимизировать $V(S)$ с учетом ограничения, что $K(S)$, которое является суммой весов в S, не превышает C. Обратите внимание, что простая задача о рюкзаке является частным случаем общей задачи о рюкзаке, где $v_i = w_i$ для всех $1 \leqslant i \leqslant d$.

Для того чтобы решить общую задачу о рюкзаке, мы начнем с вычисления того же булева массива $R(i, j)$, который использовался для решения простой задачи о рюкзаке. Следовательно, $R(i, j)$ игнорирует значения v_i и зависит только от весов w_i. Далее мы определим еще один массив $V(i, j)$, который зависит от значений v_i следующим образом:

$$V(i, j) = \max\{V(S) | S \subseteq [i] \text{ и } K(S) = j\}, \tag{4.2}$$

для $0 \leqslant i \leqslant d$ и $0 \leqslant i \leqslant C$.

Задача 4.18. Дайте рекуррентное соотношение для вычисления массива $V(i, j)$, используя булев массив $R(i, j)$, – будем считать, что массив $R(i, j)$ уже вычислен. Также дайте алгоритм вычисления $V(i, j)$.

Задача 4.19. Если определение $V(i, j)$, данное в (4.2), изменяется, в результате чего нам требуется только $K(S) \leqslant j$ вместо $K(S) = j$, то булев массив $R(i, j)$ в рекуррентном соотношении не нужен. Дайте рекуррентное соотношение в этом случае.

4.4. Задача выбора мероприятий

Вход: список мероприятий $(s_1, f_1, p_1), \ldots, (s_n, f_n, p_n)$, где $p_i > 0$, $s_i < f_i$ и s_i, f_i, p_i – это неотрицательные вещественные числа.
Выход: множество $S \subseteq [n]$ выбранных мероприятий таких, что никакие два выбранных мероприятия не накладываются и прибыль $P(S) = \sum_{i \in S} p_i$ равна максимально возможной.

Мероприятие i имеет фиксированные время начала s_i, время завершения f_i и прибыль p_i. Для заданного множества мероприятий мы хотим выбрать подмножество ненакладывающихся мероприятий с максимальной общей прибылью. Ти-

пичным примером задачи выбора мероприятий является серия лекций с фиксированным временем начала и завершения, которые должны быть запланированы в одном учебном классе.

Определим массив подзадач: отсортируем мероприятия по времени завершения, $f_1 \leqslant f_2 \leqslant \ldots \leqslant f_n$. Поскольку возможно, что мероприятия завершаются одновременно, мы выбираем несовпадающие времена завершения и обозначаем их $u_1 < u_2 < \ldots < u_k$, где очевидно, что $k \leqslant n$. Например, если мы имеем мероприятия, завершающиеся в моменты времени 1,24; 4; 3,77; 1,24; 5 и 3,77, то мы разбиваем их на четыре группы: мероприятия, завершающиеся в моменты времени $u_1 = 1{,}24$, $u_2 = 3{,}77$, $u_3 = 4$, $u_4 = 5$.

Пусть u_0 равно $\min_{1 \leqslant i \leqslant n} s_i$, то есть самому раннему времени начала. Следовательно,

$$u_0 < u_1 < u_2 < \ldots < u_k,$$

так как понятно, что $s_i < f_i$. Определим массив $A(0 \ldots k)$ следующим образом:

$$A(j) = \max_{S \subseteq [n]} \{P(S) | S \text{ является допустимым и } f_i \leqslant u_j \text{ для каждого } i \in S\},$$

где расписание S является допустимым, если никакие два мероприятия в S не накладываются. Обратите внимание, что $A(k)$ – это максимально возможная прибыль для всех возможных расписаний S.

Задача 4.20. Дайте формальное определение того, что означает быть допустимым для расписания мероприятий, то есть выразите точно, что мероприятия во множестве S «не накладываются».

Определим рекуррентное соотношение $A(0 \ldots k)$. Для того чтобы дать такое рекуррентное соотношение, сначала определим вспомогательный массив $H(1 \ldots n)$ такой, что $H(i)$ – это индекс наибольшего несовпадающего времени завершения, не превышающего время начала мероприятия i. Формально $H(i) = l$, если l – это наибольшее число такое, что $u_l \leqslant s_i$. Для вычисления $H(i)$ нам нужно выполнить поиск в списке несовпадающих времен завершения. Для того чтобы это сделать эффективно, для каждого i применим процедуру двоичного поиска, которая выполняется за логарифмическое время в длине списка несовпадающих времен завершения (попробуйте сначала $l = \lfloor \frac{k}{2} \rfloor$). Так как длина k списка несовпадающих времен завершения не превышает n и нам нужно применить двоичный поиск для каждого элемента массива $H(1..n)$, время, необходимое для вычисления всех элементов массива, составляет $O(n \log n)$.

Мы инициализируем $A(0) = 0$, и мы хотим вычислить $A(j)$, при условии что у нас уже есть $A(0)$, ..., $A(j-1)$. Рассмотрим $u_0 < u_1 < u_2 < \ldots < u_{j-1} < u_j$. Можем ли мы превзойти прибыль $A(j-1)$, запланировав некое мероприятие, которое заканчивается в момент времени u_j? Попробуем все мероприятия, которые заканчиваются в этот момент времени, и вычислим максимальную прибыль в каждом случае. Мы получим следующее рекуррентное соотношение:

$$A(j) = \max\{A(j-1), \max_{1 \leqslant i \leqslant n} \{p_i + A(H(i)) | f_i = u_j\}\}, \tag{4.3}$$

где $H(i)$ – это наибольшее l такое, что $u_l \leqslant s_i$. Рассмотрим пример, приведенный на рис. 4.2.

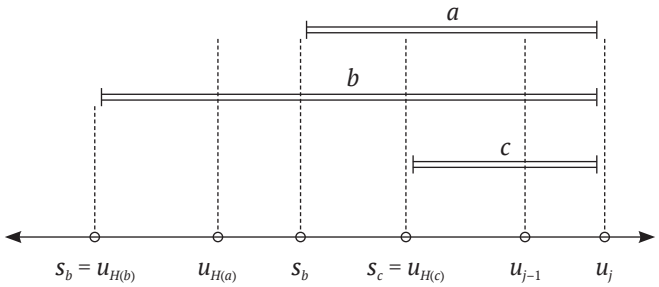


Рис. 4.2 ❖ В данном примере мы хотим вычислить $A(j)$. Предположим, что некое мероприятие, завершающееся в момент времени u_j, должно быть запланировано для получения максимально возможной прибыли. На этом рисунке есть три мероприятия, которые заканчиваются в момент времени u_j: a, b, c, – заданных соответственно тройками (s_a, f_a, p_a), (s_b, f_b, p_b), (s_c, f_c, p_c), где, разумеется, принято допущение, что $u_j = f_a = f_b = f_c$. Вопрос в том, какое из этих трех мероприятий должно быть выбрано. Для того чтобы это установить, мы должны посмотреть на каждое мероприятие a, b, c по очереди и определить, какое наиболее прибыльное расписание мы можем получить, если настаиваем на том, чтобы данное мероприятие было запланировано. Например, если мы настаиваем на том, чтобы было запланировано мероприятие a, то мы должны увидеть, какое самое прибыльное расписание мы можем получить, где все другие мероприятия должны закончиться к s_a, что фактически означает, что все другие мероприятия должны закончиться к $u_{H(a)}$. Обратите внимание, что в этом примере мы имеем, что $u_{H(a)} < s_a$, но $u_{H(b)} = s_b$ и $u_{H(c)} = s_c$. По большому счету, мы должны найти, какое из трех значений $p_a + A(H(a))$, $p_b + A(H(b))$, $p_c + A(H(c))$ является максимальным

Рассмотрим пример на рис. 4.3. Для того чтобы увидеть, как была вычислена нижняя строка правой таблицы на рис. 4.3, обратите внимание, что в соответствии с рекуррентным соотношением (4.3) мы имеем:

$A(2) = \max\{20, 30 + A(0), 20 + A(1)\} = 40,$
$A(3) = \max\{40, 30 + A(0)\} = 40.$

Мероприятие i	1	2	3	4
Начало s_i	0	2	3	2
Конечный f_i	3	6	6	10
Прибыль p_i	20	30	20	30
$H(i)$	0	0	1	0

j	0	1	2	3
u_j	0	3	6	10
$A(j)$	0	20	40	40

Рис. 4.3 ❖ Пример с четырьмя мероприятиями

Таким образом, максимальная прибыль равна $A(3) = 40$.

Задача 4.21. Напишите алгоритм.

Задача 4.22. С учетом того, что A было вычислено, как найти множество мероприятий S такое, что $P(S) = A(k)$? Подсказка: если $A(k) = A(k - 1)$, то мы знаем,

что никакое выбранное мероприятие не заканчивается в момент времени u_k, поэтому мы переходим к рассмотрению $A(k-1)$. Если $A(k) > A(k-1)$, то некое выбранное мероприятие завершается в момент времени u_k. Как мы находим это мероприятие?

Задача 4.23. Реализуйте решение методом динамического программирования для «задачи выбора мероприятия с прибылью». Ваш алгоритм должен вычислить стоимость наиболее прибыльного множества мероприятий, а также вывести исчерпывающий список этих мероприятий.

4.5. Задания с указанием предельных сроков, длительностей и прибылей

Вход: список заданий $(d_1, t_1, p_1), \ldots, (d_n, t_n, p_n)$.
Выход: допустимое расписание $C(1 \ldots n)$ такое, что прибыль C, обозначаемая $P(C)$, является максимально возможной среди допустимых расписаний.

В разделе 2.2 мы рассмотрели задачи планирования заданий для случая, когда каждое задание занимает единицу времени, то есть каждая длительность $d_i = 1$. Теперь мы обобщим ее на случай, когда каждое задание i имеет произвольную длительность d_i, крайний срок t_i и прибыль p_i. Мы исходим из того, что d_i и t_i являются положительными целыми числами, но прибыль p_i может быть положительным вещественным числом. Мы говорим, что расписание $C(1 \ldots n)$ является допустимым, если выполняются следующие два условия (пусть $C(i) = -1$ обозначает, что задание i не запланировано, и, значит, $C(i) \geqslant 0$ указывает на то, что оно запланировано, и обратите внимание, что мы разрешаем планировать задания в момент времени 0):

(1) если $C(i) \geqslant 0$, то $C(i) + d_i \leqslant t_i$ и
(2) если $i \neq j$ и $C(i), C(j) \geqslant 0$, то
    (a) $C(i) + d_i \leqslant C(j)$; или
    (b) $C(j) + d_j \leqslant C(i)$.

Первое условие сродни утверждению, что каждое запланированное задание завершается к предельному сроку, и второе условие сродни утверждению, что никакие два запланированных задания не накладываются. Цель состоит в том, чтобы найти допустимое расписание $C(1 \ldots n)$ для n заданий, для которых прибыль $P(C) = \sum_{C(i) \geqslant 0} p_i$, сумма прибылей запланированных заданий максимизируется.

Задание отличается от мероприятия тем, что оно может быть запланировано в любое время, если только завершается к предельному сроку; мероприятие имеет фиксированные времена начала и завершения. Из-за гибкости планирования заданий найти оптимальное расписание для заданий «труднее», чем выбрать оптимальное подмножество мероприятий.

Обратите внимание, что планирование заданий представляет «как минимум такую же трудность, как и простая задача о рюкзаке». На самом деле экземпляр простой задачи о рюкзаке $w_1, \ldots, w_n, C$ можно рассматривать как задачу планирования заданий, в которой каждая длительность $d_i = w_i$, каждый предельный срок $t_i = C$ и каждая прибыль $p_i = w_i$. Тогда максимальная прибыль любого раписания равна максимальному весу, который можно положить в рюкзак. Эта, казалось бы,

невинная идея о «как минимум такой же трудности» на самом деле является мощным инструментом, широко используемым в области вычислительной сложности для сравнения относительной трудности задач. Пересмотрев общий экземпляр планирования заданий как экземпляр простой задачи о рюкзаке, мы обеспечили *редукцию* планирования расписания заданий в простую задачу о рюкзаке и тем самым показали, что если бы удалось эффективно решить задачу планирования заданий, то у вас автоматически было бы эффективное решение простой задачи о рюкзаке.

Для того чтобы предоставить решение задачи планирования заданий методом динамического программирования, мы начинаем с сортировки заданий по предельным срокам. Следовательно, мы исходим из того, что $t_1 \leqslant t_2 \leqslant \ldots \leqslant t_n$.

Получается, что, для того чтобы определить подходящий массив A для решения задачи, мы должны рассмотреть все возможные целочисленные времена t, $0 \leqslant t \leqslant t_n$ как предельный срок для первых i заданий. Недостаточно рассмотреть только указанный предельный срок t_i, заданный во входных данных задачи. Таким образом, определим массив $A(i, t)$ следующим образом:

$$A(i, j) = \max\left\{P(C): \begin{array}{l} C \text{ – это допустимое решение;} \\ \text{планируются только те задания, которые в } [i]; \\ \text{все запланированные задания заканчиваются ко времени } t \end{array}\right\}.$$

Теперь мы хотим спроектировать рекуррентное соотношение для вычисления $A(i, t)$. В обычном стиле рассмотрим два случая, когда задание i выполняется или не выполняется в оптимальном расписании (и отметим, что задание i не будет выполняться в оптимальном расписании, если $d_i > \min\{t_i, t\}$). Если задание i не встречается, то мы уже знаем оптимальную прибыль.

Если, с другой стороны, задание i в оптимальном расписании все-таки встречается, то мы можем также допустить, что это задание является последним (среди заданий $\{1, \ldots, i\}$), которые должны быть запланированы, потому что оно имеет последний предельный срок. Поэтому мы допускаем, что задание i планируется как можно позже с целью его завершения либо в момент времени t, либо в момент времени t_i в зависимости от того, какое из них меньше, то есть оно заканчивается в момент времени $t_{\min} = \min\{t_i, t\}$.

Задача 4.24. В свете обсуждения в двух приведенных выше абзацах найдите рекуррентное соотношение для $A(i, t)$.

Задача 4.25. Реализуйте свое решение.

4.6. Дальнейшие примеры и задачи

4.6.1. Задача суммирования сплошной подпоследовательности

Вход: вещественные числа $r_1, \ldots, r_n$.
Выход: для каждой сплошной подпоследовательности (то есть подпоследовательности с расположенными подряд элементами) вида $r_i, r_{i+1}, \ldots, r_j$ пусть

$$S_{ij} = r_i + r_{i+1} + \cdots + r_j,$$

где $S_{ii} = r_i$. Найти $M = \max_{1 \leqslant i \leqslant j \leqslant n} S_{ij}$.

Например, на рис. 4.4 мы имеем пример задачи суммирования сплошной подпоследовательности. Там решение равняется $M = S_{35} = 3 + (-1) + 2 = 4$.

Эта задача может быть решена за время $O(n^2)$ путем систематического вычисления всех сумм S_{ij} и нахождения максимума (имеется $\binom{n}{2}$ пар $i, j \leqslant n$ таких, что $i < j$). Однако существует более эффективное решение методом динамического программирования, которое работает за время $O(n)$.

Определим массив $M(1..n)$ следующим образом:

$M(j) = \max\{S_{1j}, S_{2j}, \ldots, S_{jj}\}$.

См. рис. 4.4 с примером.

Задача 4.26. Объясните, как найти решение M из массива $M(1...n)$.

Задача 4.27. Заполните четыре строки кода, указанные в алгоритме 4.5 для вычисления значений массива $M(1...n)$ с учетом $r_1, r_2, \ldots, r_n$.

j	1	2	3	4	5	6	7
r_j	1	−5	3	−1	2	−8	3
$M(j)$	1	−4	3	2	4	−4	3

Рис. 4.4 ❖ Пример вычисления $M(j)$

Алгоритм 4.5. Задача 4.27

```
M(1) ← _____(1)
for j : 2..n do
        if _____(2) then
                M(j) ← _____(3)
        else
                M(j) ← _____(4)
        end if
end for
```

4.6.2. Перетасовка

В этом разделе мы намерены изучить алгоритм, который работает с цепочками символов – строками; см. раздел 8.2 для получения общей информации по строкам, алфавитам и языкам.

Если u, v и w – это строки над алфавитом $\sum$, тогда w – это перетасовка u и v, при условии что существуют (возможно, пустые) строки x_i и y_i такие, что $u = x_1x_2 \cdots x_k$ и $v = y_1y_2 \cdots y_k$ и $w = x_1y_1x_2y_2 \cdots x_ky_k$. Перетасовка иногда также называется «слиянием» или «чередованием». В интуитивном плане данное определение базируется на идее, что w может быть получена из u и v с помощью операции, аналогичной тасованию двух колод карт. Мы используем $w = u \odot v$ для обозначения, что w является перетасовкой u и v; вместе с тем обратите внимание, что, несмотря на данную форму записи, из u и v может быть получено много разных перетасовок w. Строка w называется *квадратом*, при условии что она равна перетасовке строки u с самой собой, а именно при условии $w = u \odot u$ для некоторой строки u. В работе

[Buss и Soltys (2013)] показано, что множество квадратов является NP-полным; это является истинным даже для (достаточно больших) конечных алфавитов. См. раздел 4.3, где расматривается NP-полнота.

В начале 1980-х годов Мэнсфилд [Mansfield (1982, 1983)] и Вармут и Хаусслер [Warmuth и Haussler (1984)] изучали вычислительную сложность операции перетасовки. В работе [Mansfield (1982)] дан полиномиально-временной алгоритм динамического программирования для решения следующей задачи перетасовки: для заданных входов u, v, w может ли w быть выражена как перетасовка u и v, то есть $w = u \odot v$?

Идея в основе алгоритма [Mansfield (1982)] заключается в построении решеточного графа с $(|x| + 1) \times (|y| + 1)$ узлами; левый нижний узел представлен (0, 0), и правый верхний узел представлен $(|x|, |y|)$. Для любых $i < |x|$ и $j < |y|$ мы имеем ребра:

$$\begin{cases} ((i,j),(i+1,j)), \text{если } x_{i+1} = w_{i+j+1} \\ ((i,j),(i,j+1)), \text{если } y_{j+1} = w_{i+j+1}. \end{cases} \tag{4.4}$$

Обратите внимание, что оба ребра могут присутствовать, и это, в свою очередь, вводит экспоненциальное число вариантов, если выполнять поиск наивно.

Путь начинается в (0, 0), и когда i-й раз он идет вверх, мы выбираем x_i и, когда j-й раз он идет вправо, y_j. Таким образом, путь из (0,0) в $(|x|, |y|)$ представляет собой конкретную перетасовку.

Например, рассмотрим рис. 4.5. Слева мы имеем перетасовку 000 и 111, которая дает 010101, и справа мы имеем перетасовку 011 и 011, которая дает 001111. Левый экземпляр имеет уникальную перетасовку, которая дает 010101, и она соответствует уникальному пути из (0,0) в (3, 3). Справа есть несколько возможных перетасовок 011, 011, которые дают 001111 – на самом деле восемь, каждая из которых соответствует несовпадающему пути из (0, 0) в (3, 3).

Алгоритм динамического программирования из работы [Mansfield (1982)] вычисляет частичные решения вдоль левой верхней и правой нижней диагональных линий на решеточном графе.

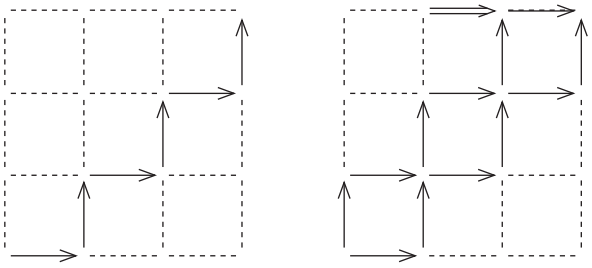


Рис. 4.5 ❖ Слева мы имеем перетасовку 000 и 111, которая дает 010101, и справа мы имеем перетасовку 011 и 011, которая дает 001111. Двойная стрелка на правой диаграмме находится там, показывая, что рядом со сплошными стрелками могут быть и другие стрелки; двойная стрелка равна ((1, 3), (2, 3)), и она находится там, потому что $x_{1+1} = x_2 = 1 = w_5 = w_{1+3+1}$. Ребра размещены согласно (4.4)

Число путей всегда ограничено:

$$\binom{|x| + |y|}{|x|}$$

и эта граница достигается для $\langle 1^n, 1^n, 1^{2n} \rangle$. Таким образом, число путей может быть экспоненциальным по размеру входных данных, и поэтому исчерпывающий поиск вообще невозможен.

Задача 4.28. С учетом обсуждения в этом разделе предложите алгоритм динамического программирования, который, получая на входе w, u, v, проверяет, является ли $w = u \odot v$.

4.7. Ответы к избранным задачам

Задача 4.1. После того как мы вычислили значения R, мы можем пройтись по нему в обратном порядке от конца самой длинной неубывающей подпоследовательности. Такая последовательность должна заканчиваться на индексе j таком, что $R(j)$ является максимальным. Если $R(j) = 1$, то доказательство завершено. В противном случае, для того чтобы найти индекс, предшествующий j, найдите любой индекс $i < j$ такой, что $R(i) = R(j) - 1$ и $a_i \leqslant a_j$; один такой обязательно существует, или $R(j)$ будет меньше. Продолжайте прослеживать в обратном порядке до тех пор, пока не дойдете до начала подпоследовательности, где R равно 1.

Задача 4.2. Алгоритм 4.1 требует только, чтобы на входе у него была конечная последовательность упорядоченных объектов (то есть объектов, для которых «$\leqslant$» имеет смысл). Его постусловие, которое мы стремимся доказать, заключается в том, что для всех j в $\{1, 2, \ldots, d\}$, $R(j)$ – это длина самой длинной неубывающей подпоследовательности, заканчивающейся в a_j.

Мы утверждаем, что после j итераций внешнего цикла «for» $R(j)$ будет длиной самой длинной подпоследовательности, заканчивающейся в a_j, и, более того, что то же самое верно для всех $i < j$. Из первого вытекает второе, так как после того, как значение присваивается элементу $R(i)$, алгоритм ни разу не присваивает его заново.

Доказательство будет по полной индукции над j. Пусть S_j обозначает любую длинную неубывающую подпоследовательность, оканчивающуюся в a_j для любого индекса j. В базовом случае очевидно, что $R(1) = 1$ является правильным присваиванием; пустая подпоследовательность имеет длину менее 1, и единственная другая подпоследовательность, $\{a_1\}$, является тривиально неубывающей с мощностью 1. Допустим, что для всех $i < j$, $R(i)$ было присвоено правильное значение. Если $S_j = \{a_j\}$, то нет $i < j$ такого, что $a_i \leqslant a_j$, поэтому значение max никогда не будет изменено после начального присваивания значения 0. В связи с этим элементу $R(j)$ дается правильное значение, 1. Если, с другой стороны, $|S_j| > 1$, то существует элемент a_i, непосредственно предшествовавший a_j в S_j, где $a_i \leqslant a_j$ и $i < j$. Очевидно, что существует S_j такое, что $S_j = S_i \cup \{a_j\}$, поэтому $|S_j| = |S_i| + 1 = R(i) + 1$.

Допустим, что max $\neq R(i)$ после итерации i внутреннего цикла for. $a_i \leqslant a_j$, поэтому $R(i) <$ max. Следовательно, существует $i' < i$ такой, что $a_{i'} \leqslant a_j$ и $R(i') > R(i)$. Но $S_{i'} \cup \{a_j\}$ является неуменьшающимся, заканчивается на a_j и имеет мощность, большую, чем S_j, – противоречие. Схожим образом max не может быть заново переназначен позже, так как это приведет к такому же противоречию. Следовательно, в конце

итерации j элементу $R(j)$ присваивается правильное значение $R(i) + 1$. Поэтому после итерации d $R(j)$ является правильным для всех j.

Задача 4.3. Для того чтобы найти длину самой длинной подпоследовательности, над которой любые два следующих подряд члена отличаются не более чем на 1, мы можем просто отредактировать условие «if» в алгоритме 4.1. В частности, «$a_i \leqslant a_j$» можно заменить на «$|a_i - a_j| \leqslant 1$».

Задача 4.5. Рассмотрим граф G_n на рис. 4.6. Он содержит $2 + n + n = 2n + 2$ узлов и 2^n путей из s в t; начиная в s, у нас есть выбор – идти в узел 1 или в узел $1'$, и далее у нас всегда есть выбор: идти вверх или вниз, поэтому существует $2 \times 2^{n-1}$ путей, которые приводят нас в n или n'. Наконец, мы просто идем в t. Обратите внимание, что мы привели неориентированный граф; но если придать всем ребрам направление «слева направо», то мы получим пример для ориентированных графов.

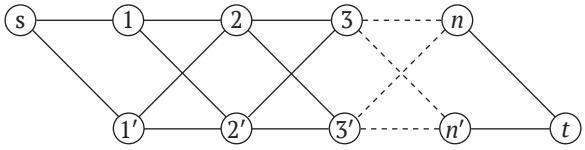


Рис. 4.6 ❖ Экспоненциально много путей (задача 4.5)

Задачи 4.6 и 4.7. Предусловием является то, что $\forall i, j \in [n]$, и мы имеем $B(i, j) = A(0, i, j)$. Постусловие состоит в том, что $\forall i, j \in [n]$, и мы имеем $B(i, j) = A(n, i, j)$. Инвариант цикла состоит в том, что после k-й итерации главного цикла $B(i, j) = A(k, i, j)$. Для того чтобы доказать инвариант цикла, следует отметить, что $B(i, j)$ задается с помощью $\min\{B(i, j), B(i, k) + B(k, j)\}$, поэтому единственное беспокойство вызывает то, что $B(i, k)$ или $B(k, j)$ было уже обновлено, поэтому мы не получаем $A(k - 1, i, k)$ или $A(k - 1, k, j)$, как следовало бы, а наоборот – $A(k, i, k)$ или $A(k, k, j)$. Но оказывается, что $A(k, i, k) = A(k - 1, i, k)$ и $A(k, k, j) = A(k - 1, k, j)$, потому что кратчайший путь из i в k (или из k в j) не содержит k в качестве промежуточного узла.

Задача 4.10. Перезапись не создает проблем, так как значения j рассматриваются в убывающем порядке C, $C - 1$, ...,1. Следовательно, когда делается ссылка, позиция массива $S(j - w_i)$ еще не была обновлена.

Задача 4.11. Предусловием является то, что для всех j, $S(j) = R(0, j)$. Постусловием является то, что для всех j, $S(j) = R(d, j)$. Пусть инвариант цикла равен логическому утверждению, что после i-го шага $S(j) = R(i, j)$. Этот инвариант цикла соблюдается, так как мы начинаем «заполнять» S справа, и мы только изменяем ложь на истину (не истину на ложь – причина в том, что если бы мы могли построить промежуточное значение j с первыми $(i - 1)$ весами, то мы, безусловно, по-прежнему можем построить его с первыми i весами).

Задача 4.12. Рассмотрим $w_1 = w_2 = 1$, $w_3 = 2$ и $C = 3$, поэтому таблица на этих входных данных будет выглядеть следующим образом:

	0	1	2	3
	T	F	F	F
$w_1 = 1$	T	T	F	F
$w_2 = 1$	T	T	T	F
$w_3 = 2$	T	T	T	T

Теперь рассмотрим строку таблицы для $w_i = 1$ и ячейку для столбца с меткой 2. Значение в указанной ячейке равно F, как и должно быть, но если бы цикл for в алгоритме 4.3 не был уменьшающимся циклом, то мы бы обновили эту ячейку значением T, так как для $j = 2$ мы имеем $2 \geqslant w_1$ и $S(2 - w_1) = \mathsf{T}$.

Задача 4.14. Сначала нам нужно найти решение; поэтому мы ищем в последней строке (то есть строке d) наибольшее ненулевое j. Иными словами, решение дается с помощью $M = \max_{0 \leqslant j \leqslant C}[R(d, j) = \mathsf{T}]$. Теперь мы проверяем, что $R(d - 1, M) = \mathsf{T}$. Если да, то мы знаем, что вес w_d не нужен, поэтому мы его не включаем и продолжаем смотреть на $R(d - 2, M)$. Если нет, то поскольку $R(d, M) = \mathsf{T}$, мы знаем, что $M - w_d \geqslant 0 \wedge R(d - 1, M - w_d) = \mathsf{T}$. Поэтому мы включаем w_d и продолжаем смотреть на $R(d - 1, M - w_d)$. Мы останавливаемся, когда достигаем первого столбца массива.

Задача 4.15. Естественный жадный алгоритм, который пытается решить простую задачу о рюкзаке, следующий: упорядочить веса от самого тяжелого до самого легкого и добавлять их в этом порядке как можно дольше. Допустим, что $\overline{M} \neq M$, и пусть S_0 равно результату этой жадной процедуры, то есть подмножеству $\{1, \ldots, d\}$ такому, что $K(S_0) = \overline{M}$. Сначала покажем, что в S_0 есть хотя бы один вес. Если $S_0 = \emptyset$, то $\overline{M} = 0$, и все веса должны быть больше C, но тогда $M = 0$, и поэтому $\overline{M} = M$, что не соответствует действительности по принятому допущению. Теперь покажем, что есть хотя бы один вес не в S_0: если все веса находятся в S, то снова $\overline{M} = \sum_{i=1}^{d} w_i = M$. Наконец, теперь покажем, что $\overline{M} > \frac{1}{2}C$, рассмотрев первый вес, назовем его w_j, который был отклонен, после того как хотя бы один вес был добавлен (обратите внимание, что такой вес должен существовать; мы можем допустить, что нет весов больше, чем грузоподъемность C, а если есть, то мы можем просто их не рассматривать; поэтому первый вес в списке добавляется, и тогда мы знаем, что появится некий вес, который не будет добавлен; мы рассматриваем первый такой вес). Если $w_j \leqslant \frac{1}{2}C$, то сумма весов, которые уже внутри, $> \frac{1}{2}C$, поэтому $\overline{M} > \frac{1}{2}C$. Если $w_j > \frac{1}{2}C$, то, поскольку объекты упорядочены жадным алгоритмом в невозрастающем порядке весов, веса, которые уже находятся внутри, $> \frac{1}{2}C$, поэтому снова $\overline{M} > \frac{1}{2}C$.

Задача 4.16. В первом пробеле поставьте $w_i + \sum_{j \in S} w_j \leqslant C$, и во втором $S \leftarrow S \cup \{i\}$.

Задача 4.17. Определим выражение «S является перспективным» как означающее, что S может быть расширено с использованием еще не рассматривавшихся весов до оптимального решения $S_{\max}$. В конце, когда весов для рассмотрения больше не остается, инвариант цикла по-прежнему истинен, поэтому S само должно быть оптимальным.

Покажем, что выражение «S является перспективным» является инвариантом цикла, по индукции на числе итераций. Базовый случай: $S = \emptyset$, поэтому S явно является перспективным. Индукционный шаг: предположим, что S является перспективным (поэтому S может быть расширено, используя еще не рассматривавшиеся веса, до $S_{\max}$). Пусть S' равно S после еще одной итерации. Предположим $i \in S'$. Поскольку $w_i \geqslant \sum_{j=i+1}^{d} w_j$, из этого следует, что:

$$K(S') \geqslant K(S) + \sum_{j=i+1}^{d} w_j,$$

поэтому S' уже содержит, по крайней мере, столько же веса, сколько любое расширение S, не включая w_i. Если S' является оптимальным, то доказательство завер-

шено. В противном случае S_{max} имеет больший вес, чем S', поэтому оно должно содержать w_i. Предположим $i \notin S'$, тогда мы имеем, что $w_i + \sum_{j \in S} w_j > C$, поэтому $i \notin S_{max}$. В связи с этим S' может быть расширено (используя еще не рассматривавшиеся веса!) до S_{max}. В обоих случаях S' является перспективным.

Задача 4.18. $V(i, j) = 0$, если $i = 0$ или $j = 0$. И для $i, j > 0$, $V(i, j)$ равно

$$\begin{cases} V(i-1, j), & \text{если } j < w_i \text{ или } R(i-1, j-w_i) = \mathsf{F}, \\ \max\{v_i + V(i-1, j-w_i), V(i-1, j)\} & \text{в противном случае.} \end{cases}$$

Для того чтобы убедиться, что это работает, предположим, что $j < w_i$. Тогда вес i не может быть включен, поэтому $V(i, j) = V(i-1, j)$. Если $R(i-1, j-w_i) = \mathsf{F}$, то не существует подмножества $S \subseteq \{1, \ldots, i\}$ такого, что $i \in S$ и $K(S) = j$, поэтому вес i опять не включен и $V(i, j) = V(i-1, j)$.

В противном случае, если $j \geqslant w_i$ и $R(i-1, j-w_i) = \mathsf{T}$, вес i может, а может и не быть включен в S. Мы берем случай, который предлагает бóльшую величину: $\max\{v_i + V(i-1, j-w_i), V(i-1, j)\}$.

Задача 4.19. Изменив определение $V(i, j)$, приведенное в (4.2), на $K(S) \leqslant j$ (вместо $K(S) = j$), мы можем взять рекуррентное соотношение, заданное для V в решении задачи 4.18, и просто избавиться от части «или $R(i-1, j-w_i) = \mathsf{F}$» для получения рекуррентного соотношения для V, которое не требует вычисления R.

Задача 4.20. Предположим, что расписание S содержит мероприятия $\{a_1, a_2, \ldots\}$, где $a_n = (s_n, f_n, p_n)$ являются временем начала, временем завершения и прибылью мероприятия a_n для всех n. Расписание S является допустимым, если для всех a_i, $a_j \in S$ либо $f_i \leqslant s_j$, либо $f_j \leqslant s_i$; то есть первое из двух должно быть завершено до начала второго, поскольку в противном случае они явно накладываются.

Задача 4.21. Алгоритм должен включать вычисление несовпадающих времен окончания, то есть времен u_i, а также вычисление массива H. Здесь мы просто даем алгоритм вычисления, основанный на рекуррентном соотношении (4.3). При этом мы исходим из того, что существует n мероприятий и k несовпадающих времен завершения.

Алгоритм 4.6. Выбор мероприятия

```
A(0) ← 0
for j : 1..k do
        max ← 0
        for i = 1..n do
                if f_i = u_j then
                        if p_i + A(H(i)) > max then
                                max ← p_i + A(H(i))
                        end if
                end if
        end for
        if A(j − 1) > max then
                max ← A(j − 1)
        end if
        A(j) ← max
end for
```

Задача 4.22. Мы покажем, как найти фактическое множество мероприятий. Предположим $k > 0$. Если $A(k) = A(k-1)$, то никакое мероприятие не было запланировано завершиться в момент времени u_k, поэтому мы продолжаем рекурсивно исследовать $A(k-1)$. Если, с другой стороны, $A(k) \neq A(k-1)$, то мы знаем, что некое мероприятие было запланировано завершиться в момент времени u_k. Мы должны выяснить, которое из них. Мы знаем, что в этом случае $A(k) = \max_{1 \leqslant i \leqslant n}\{p_i + A(H(i)) | f_i = u_k\}$, поэтому мы исследуем все мероприятия i, $1 \leqslant i \leqslant n$, а выдаем (первое) мероприятие i_0 такое, что $A(k) = p_{i_0} + A(H(i_0))$ и $f_{i_0} \leqslant u_k$. Теперь мы повторяем эту процедуру с $A(H(i_0))$. Мы заканчиваем, когда $k = 0$.

Задача 4.24. Инициализация: $A(0, t) = 0$, $0 \leqslant t \leqslant t_n$. Для того чтобы вычислить $A(i, t)$ для $i > 0$, сначала определим $t_{\min} = \min\{t, t_i\}$. Теперь

$$A(i,t) = \begin{cases} A(i-1,t), & \text{если } t_{\min} < d_i \\ \max\{A(i-1,t),\ p_i + A(i-1, t_{\min} - d_i)\} & \text{в противном случае} \end{cases}.$$

Обоснование: если задание i запланировано в оптимальном расписании, то оно завершается в момент времени $t_{\min}$ и начинается в момент времени $t_{\min} - d_i$. Если оно запланировано, то максимально возможная прибыль равна $A(i-1, t_{\min} - d_i) + p_i$. В противном случае максимальная прибыль равна $A(i-1, t)$.

Задача 4.26. $M = \max_{1 \leqslant j \leqslant n} M(j)$.

Задача 4.27.

1) $r_1 (= S_{11})$;
2) $M(j-1) > 0$;
3) $M(j-1) + r_j$;
4) r_j.

4.8. Примечания

Любой учебник по алгоритмам будет иметь раздел по динамическому программированию; см., например, главу 15 в книге [Cormen и соавт. (2009)] и главу 6 в [Kleinberg и Tardos (2006)].

В отличие от матроидов, которые служат хорошей абстрактной моделью для жадных алгоритмов, в настоящее время разрабатывается общая модель динамического программирования. См. [Aleknovich и соавт. (2005)].

Материал по операции перетасовки, раздел 4.6.2, взят из работ [Buss и Soltys (2013)] и [Mhaskar и Soltys (2015)]. Первоначальная работа по перетасовкам возникла из абстрактных формальных языков, и позже перетасовки были обусловлены приложениями к моделированию последовательного исполнения параллельных процессов. Насколько известно автору, операция перетасовки впервые была использована на формальных языках Гинзбургом и Спанье [Ginsburg и Spanier (1965)]. Ранние исследования с приложениями к параллельным процессам можно найти у Риддла [Riddle (1973, 1979)] и Шоу [Shaw (1978)]. Ряд авторов, в частности [Gischer (1981); Gruber и Holzer (2009); Jantzen (1981, 1985); Jedrzejowicz (1999); Jedrzejowicz и Szepietowski (2001, 2005); Mayer и Stockmeyer (1994); Ogden и соавт. (1978); Shoudai (1992)], впоследствии изучали различные аспекты сложности операций перетасовки и итерированной перетасовки в сочетании с операциями регулярных выражений и другими конструкциями из теории языков программирования.

В публикации [Mansfield (1983)] были даны алгоритмы с полиномиальным временем для принятия решения о том, может ли строка w быть записана как перетасовка k строк $u_1, \ldots, u_k$, чтобы $w = u_1 \odot u_2 \odot \cdots \odot u_k$ для постоянного целого k. В работе [Mansfield (1983)] далее доказано, что если k разрешено варьироваться, то задача становится NP-полной (посредством редукции из точного покрытия с помощью 3-членных множеств[1]). Уормут и Хаусслер [Warmuth и Haussler (1984)] дали независимое доказательство этого последнего результата и продолжили давать довольно поразительное улучшение, показывая, что эта задача остается NP-полной, даже если k строк $u_1, \ldots, u_k$ являются эквивалентными. То есть при наличии строк u и w вопрос, эквивалентна ли строка w итерированной перетасовке $u \odot u \odot \cdots \odot u$ строки u, является NP-полным. В их доказательстве использовалась редукция из задачи с 3-членными разбиениями[2]. В публикации [Soltys (2013)] показано, что задача об эквивалентности $w = u \odot v$ может быть решена с помощью схем логарифмической глубины, но не схем ограниченной глубины.

Как упоминалось в разделе 4.6.2, строка w определяется как квадрат, если она может быть записана как $w = u \odot u$ для некоторой u. Эриксон [Erickson (2010)] в 2010 году задал вопрос на доске обсуждений Stack Exchange о вычислительной сложности распознавания квадратов и, в частности, решаема ли эта задача за полиномиальное время. Этот вопрос был повторен как открытый вопрос в работе [Henshall и соавт. (2012)]. Онлайновый ответ на [Erickson (2010)], предоставленный Острином [Austrin (2010)], показал, что задача распознавания квадратов разрешима за полиномиальное время, при условии что каждый символ алфавита встречается не более четырех раз в w (путем редукции из задачи 3-SAT[3]); однако общий вопрос остался открытым. Настоящая работа решает эту задачу, доказывая, что задача распознавания квадратов NP-полная, даже над достаточно большим фиксированным алфавитом.

---

[1] 3-членное множество (3-SET) здесь означает, что каждая сентенция содержит ровно три литерала. – *Прим. перев.*

[2] Задача с 3-членными разбиениями (3-partition problem) – это задача, когда требуется разбить $3q$ чисел (допуская дубликаты) на q групп по 3 так, чтобы каждая группа имела одинаковую сумму. См. https://en.wikipedia.org/wiki/3-partition_problem. – *Прим. перев.*

[3] Задача 3-SAT (задача выполнимости формулы с 3 переменными) состоит из объединения сентенций на n булевых переменных, где каждая сентенция является дизъюнкцией из 3 литералов. См. http://jeffe.cs.illinois.edu/teaching/algorithms/notes/30-nphard.pdf. – *Прим. перев.*

Глава 5

Онлайновые алгоритмы

Нескончаемая Вальпургиева ночь.

Сэр Роджер Скратон
[Scruton (2015)]

Представленные до этого алгоритмы были *офлайновыми алгоритмами*, в том смысле что все входные данные целиком подавались в самом начале. В этой главе мы меняем нашу парадигму и рассматриваем *онлайновые алгоритмы*, где входные данные нескончаемы и предоставляются порциями и алгоритм должен принимать решения, основываясь на неполной информации, не зная будущих событий.

Типичным примером их применения является дисциплина кеширования; рассмотрим жесткий диск, с которого данные считываются в память с произвольным доступом. Как правило, память с произвольным доступом намного меньше, и поэтому необходимо решить, какие данные должны перезаписываться новыми данными. Новые запросы данных с жесткого диска поступают непрерывно, и трудно предсказать будущие запросы.

Следовательно, мы должны перезаписывать части памяти с произвольным доступом новыми запросами, но должны выполнять перезапись разумно с целью минимизации будущих промахов: данными, которые требуются, но не присутствуют в памяти с произвольным доступом, и поэтому они должны быть доставлены с жесткого диска. Когда будущие запросы неизвестны, минимизировать число промахов очень сложно.

Правильность в контексте онлайнового алгоритма имеет другой нюанс; она означает, что алгоритм минимизирует стратегические ошибки. То есть онлайновый алгоритм, как правило, будет хуже, чем соответствующий офлайновый алгоритм, который видит все входные данные целиком, но мы хотим, чтобы он был как можно состязательнее с учетом присущих ему ограничений. Следовательно, в контексте онлайновых алгоритмов мы занимаемся оценкой результативности.

В разделе 5.1 мы вводим предмет онлайновых алгоритмов задачей доступа к списку, а затем в разделе 5.2 представляем алгоритмы замещения страниц.

5.1. Задача доступа к списку

Мы заведуем картотечным шкафом, содержащим L промаркированных, но не отсортированных файлов. Мы получаем последовательность запросов доступа к файлам; каждый запрос представляет собой метку файла. После получения запроса файла мы должны его найти, обработать и вернуть в шкаф.

Поскольку файлы не упорядочены, мы должны пролистывать файлы, начиная с самого начала, пока не будет найден запрошенный файл. Если файл находится в позиции i, то мы несем расходы на поиск i для его локализации. Если файла в шкафу нет, то стоимость равна l, равная суммарному числу файлов. После того как мы достанем файл, мы должны вернуть его в шкаф, но мы можем решить реорганизовать шкаф; например, мы можем разместить файл ближе к началу. Стимулом для такой реорганизации является то, что это может сэкономить нам некоторое время поиска в будущем: если определенный файл запрашивается часто, то целесообразно вставить его ближе к началу. Наша цель – найти правило реорганизации, которое минимизирует время поиска.

Пусть $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_n$ равно конечной последовательности из n запросов. Для выполнения поступающего запроса σ_i алгоритм доступа к списку ALG должен отыскать элемент с меткой σ_i, проходя список с самого начала до тех пор, пока он его не найдет. Стоимость извлечения этого элемента является индексом его позиции в списке. Таким образом, если элемент σ_i находится в позиции j, то стоимость его извлечения равна j. Более того, алгоритм может реорганизовать список в любое время.

Работа, связанная с реорганизацией, представляет собой минимальное число перемещений расположенных подряд элементов, необходимое для ее выполнения. Каждое перемещение имеет стоимость 1, однако сразу же после доступа к элементу мы даем ему переместиться бесплатно в любое место ближе к началу этого списка. Эти перемещения являются *бесплатными*, в то время как все остальные перемещения являются платными. Пусть $\mathrm{ALG}(\sigma)$ равно сумме стоимостей обслуживания всех элементов в списке σ, то есть сумме стоимостей всех поисковых операций плюс сумма стоимостей всех *платных перемещений*.

Задача 5.1. Чем оправдано это «бесплатное перемещение»? Другими словами, почему имеет смысл разрешить размещение элемента «бесплатно» сразу после доступа к нему? Наконец, покажите, что с учетом списка из l элементов мы всегда можем переупорядочить его любым удобным для нас способом, делая перемещения только расположенных подряд элементов.

Мы рассматриваем *модель доступа к статическому списку*, где у нас есть список из l элементов и единственными запросами являются запросы на доступ к элементу в списке, то есть нет ни вставок, ни удалений. Для задачи управления списками был предложен целый ряд алгоритмов; мы рассмотрим *алгоритм перемещения в начало* (move to front, MTF), где после доступа к элементу мы перемещаем его в начало списка без изменения относительного порядка других элементов.

Далее, мы исходим из того, что σ состоит *только* из тех элементов, которые появляются в списке MTF – это не критическое упрощение; см. задачу 5.7. Обратите внимание, что $\mathrm{MTF}(\sigma)$ – это просто сумма стоимостей всех поисковых операций, так как мы меняем положение элемента только при его извлечении, в каковом случае мы бесплатно перемещаем его в начало.

Теорема 5.2. Пусть OPT равно оптимальному (офлайновому) алгоритму для модели доступа к статическому списку. Предположим, что алгоритмы OPT и MTF начинаются с одной и той же списковой конфигурации. Тогда для любой последовательности запросов σ, где $|\sigma| = n$, мы имеем, что

$$\mathrm{MTF}(\sigma) \leqslant 2 \cdot \mathrm{OPT}_S(\sigma) + \mathrm{OPT}_P(\sigma) - \mathrm{OPT}_F(\sigma) - n, \tag{5.1}$$

где $OPT_S(\sigma)$, $OPT_P(\sigma)$, $OPT_F(\sigma)$ – это соответственно суммарная стоимость поисковых операций, суммарное число платных перемещений и суммарное число бесплатных перемещений алгоритма OPT на σ.

Доказательство. Представим, что оба алгоритма, MTF и OPT, обрабатывают запросы в σ, в то время как каждый алгоритм работает на своем собственном списке, начиная с одинаковой начальной конфигурации. Вы можете представить, что MTF и OPT работают параллельно, начиная из одного списка, и ни один не начинает обработку σ_i до тех пор, пока к этому не готов другой.

Пусть

$$a_i = t_i + (\Phi_i - \Phi_{i-1}), \tag{5.2}$$

где t_i – это фактическая стоимость, которую несет MTF на обработку этого запроса (таким образом, t_i фактически является позицией элемента σ_i в списке MTF *после* обработки первых $i - 1$ запросов). Φ_i – это *потенциальная функция*, и здесь она определяется как число *инверсий* в списке MTF по отношению к списку OPT. Инверсия определяется как упорядоченная пара элементов x_j и x_k, где x_j предшествует x_k в списке MTF, но x_k предшествует x_j в списке OPT.

Задача 5.3. Предположим, что $l = 3$, и список MTF равен x_1, x_2, x_3, и список OPT равен x_3, x_2, x_1. Какова Φ в этом случае? По сути дела, как вычислить OPT(σ), где σ – это произвольная последовательность запросов, не зная, как работает OPT?

Обратите внимание, что Φ_0 зависит только от начальных конфигураций MTF и OPT, и так как мы исходим из того, что списки изначально идентичны, $\Phi_0 = 0$. Наконец, значение a_i в (5.2) называется *амортизированной стоимостью*, и вкладываемый в нее смысл – показывать стоимость доступа к σ_i, то есть t_i плюс мера увеличения «расстояния» между списком MTF и списком OPT после обработки σ_i, то есть $\Phi_i - \Phi_{i-1}$.

Совершенно очевидно, что стоимость, понесенная MTF на обработку σ, обозначаемая MTF(σ), составляет $\sum_{i=1}^{n} t_i$. Но вместо вычисления $\sum_{i=1}^{n} t_i$, что сложно, мы вычисляем $\sum_{i=1}^{n} a_i$, что намного проще. Взаимосвязь между двумя суммированиями выражается как

$$\mathrm{MTF}(\sigma) = \sum_{i=1}^{n} t_i = \Phi_0 - \Phi_n + \sum_{i=1}^{n} a_i, \tag{5.3}$$

и так как мы договорились, что $\Phi_0 = 0$ и Φ_i всегда положительна, мы имеем, что

$$\mathrm{MTF}(\sigma) \leqslant \sum_{i=1}^{n} a_i. \tag{5.4}$$

Теперь осталось вычислить верхнюю границу для a_i.

Задача 5.4. Покажите второе равенство уравнения (5.3).

Допустим, что i-й запрос, σ_i, находится в позиции j списка OPT и в позиции k списка MTF (то есть это позиция данного элемента, после того как были обработаны первые $(i - 1)$ запросов). Пусть x обозначает этот элемент – см. рис. 5.1.

Мы собираемся показать, что

$$a_i \leqslant (2s_i - 1) + p_i - f_i, \tag{5.5}$$

где s_i – это стоимостные издержки поиска, понесенные OPT для доступа к запросу σ_i, и p_i и f_i – это соответственно платное и бесплатное перемещения, осуществленные OPT при обработке σ_i. Это неравенство показывает, что

$$\begin{aligned}\sum_{i=1}^{n} a_i &\leqslant \sum_{i=1}^{n}((2s_i - 1) + p_i - f_i) \\ &= 2\left(\sum_{i=1}^{n} s_i\right) + \left(\sum_{i=1}^{n} p_i\right) - \left(\sum_{i=1}^{n} f_i\right) - n) \\ &= 2\mathrm{OPT}_S(\sigma) + \mathrm{OPT}_P(\sigma) - \mathrm{OPT}_F(\sigma) - n,\end{aligned}$$

которое вместе с неравенством (5.4) покажет (5.1).

Мы доказываем (5.5) в два шага: на первом шаге свой ход делает MTF, то есть перемещает x из k-й ячейки в начало своего списка, и мы измеряем изменение потенциальной функции *относительно* списковой конфигурации OPT, *перед* тем как OPT сделает свои собственные шаги для обработки запроса на x.

На втором шаге OPT делает свой ход, и теперь мы измеряем изменение потенциальной функции *относительно* списков конфигурации MTF, *после* того как MTF завершил обработку запроса (то есть с элементом x в начале списка MTF).

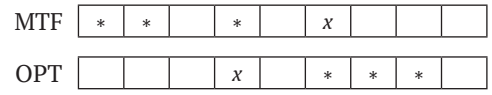


Рис. 5.1 ❖ Элемент x находится в позиции k в MTF и в позиции j в OPT. Обратите внимание, что на рисунке видно, что $j < k$, но в анализе мы такого допущения не делаем. Обозначим через $*$ элементы, расположенные перед x в MTF, но после x в OPT, то есть $*$ обозначают инверсии относительно x. Могут существовать и другие инверсии с участием x, а именно элементы, которые находятся после x в MTF, но перед x в OPT, однако нас они не касаются

Посмотрим на рис. 5.1: предположим, что существует v таких $*$, то есть v инверсий того типа, который представлен на рисунке. Тогда существует, по крайней мере, $(k - 1 - v)$ элементов, которые предшествуют x в обоих списках.

Задача 5.5. Объясните, почему, по крайней мере, $(k - 1 - v)$ элементов предшествуют x в обоих списках.

Но из этого следует, что $(k - 1 - v) \leqslant (j - 1)$, так как x находится в j-й позиции в OPT. Следовательно, $(k - v) \leqslant j$. Так что же происходит, когда MTF перемещает x в начало? С точки зрения инверсии, происходят две вещи: (i) создается $(k - 1 - v)$ новых инверсий относительно списка алгоритма OPT до того, как алгоритм OPT сам займется запросом x. (ii) исключается v инверсий, опять же относительно списка алгоритма OPT до того, как алгоритм OPT сам займется запросом x. Таким образом, вклад в амортизированную стоимость составляет:

$$k + ((k - 1 - v) - v) = 2(k - v) - 1 \overset{(1)}{\leqslant} 2j - 1 \overset{(2)}{=} 2s - 1, \tag{5.6}$$

где (1) следует из показанного выше $(k - v) \leqslant j$ и (2) следует из того факта, что стоимость поиска, понесенная OPT при поиске x, в точности равна j. Обратите

внимание, что равенство (5.6) похоже на (5.5), но отсутствует $+p_i - f_i$. Эти члены будут появляться из рассмотрения второго шага анализа: OPT делает свой ход, и мы измеряем изменение потенциала по отношению к MTF с элементом x в начале списка. Это рассматривается в следующей задаче.

Задача 5.6. На втором шаге анализа алгоритм MTF сделал свой ход, и алгоритм OPT после извлечения x переставляет свой список. Покажите, что каждое платное перемещение вносит в амортизированную стоимость вклад, равный 1, и каждое бесплатное перемещение вносит в амортизированную стоимость вклад, равный –1.

На этом доказательство заканчивается. □

В *модели доступа к динамическому списку* у нас также есть вставки, где стоимость вставки равна $l + 1$ – здесь l является длиной списка – и *удаления*, где стоимость удаления такая же, как стоимость доступа, то есть позиция элемента в списке. Алгоритм MTF всегда удаляет элемент в позиции l.

Задача 5.7. Покажите, что теорема 5.2 по-прежнему соблюдается в динамическом случае.

Инфимум подмножества $S \subseteq R$ – это наибольший элемент r, не обязательно в S, такой, что для всех $s \in S$, $r \leqslant s$. Мы говорим, что онлайновый алгоритм является *c-состязательным*, если существует постоянная α такая, что для всех конечных входных последовательностей $\mathrm{ALG}(\sigma) \leqslant c \cdot \mathrm{OPT}(\sigma) + \alpha$. Инфимум над множеством всех значений c, таких, что ALG является c-состязательным[1], называется *состязательным соотношением* алгоритма ALG и обозначается $\mathcal{R}(\mathrm{ALG})$.

Задача 5.8. Пронаблюдайте, что $\mathrm{OPT}(\sigma) \leqslant n \cdot l$, где l – это длина списка и n равно $|\sigma|$.

Задача 5.9. Покажите, что алгоритм MTF является 2-состязательным алгоритмом и что $\mathcal{R}(\mathrm{MTF}) \leqslant 2 - \frac{1}{l}$.

Задача 5.10. В главах, посвященных онлайновым и рандомизированным алгоритмам (эта глава и следующая), нам нужно генерировать случайные значения. Используйте библиотеку Python `random` для генерирования этих случайных значений; реализуйте алгоритмы OPT и MTF и сравните их на случайной последовательности запросов. Вы, возможно, захотите построить график состязательности алгоритма MTF относительно алгоритма OPT с помощью *консольной программы* `gnuplot`.

Традиционный подход к изучению онлайновых алгоритмов укладывается в рамки *распределительной сложности*, также именуемой сложностью *среднего случая*: строится гипотеза о распределении на событийных последовательностях,

---

[1] Состязательный анализ – это метод, служащий для анализа онлайновых алгоритмов, в котором результативность онлайнового алгоритма (который должен удовлетворять непредсказуемую последовательность запросов, завершая каждый запрос, не имея возможности видеть будущее) сравнивается с результативностью оптимального офлайнового (автономного) алгоритма, который может просматривать последовательность запросов заранее. Вместо термина «состязательный» нередко используется термин «конкурентный». За основу взят первый, с тем чтобы отделить его от понятия «рекуррентный». См. https://en.wikipedia.org/wiki/Competitive_analysis_(online_algorithm). – *Прим. перев.*

и анализируется ожидаемый выигрыш в расчете на событие. Однако в этой главе мы представляем более поздний подход к *состязательному анализу*, при котором выигрыш онлайнового алгоритма измеряется путем сравнения его результативности с *оптимальным офлайновым алгоритмом*. Следовательно, состязательный анализ попадает в рамки *сложности худшего случая*.

5.2. Замещение страниц

Рассмотрим двухуровневую *систему с виртуальной памятью*: каждый уровень, медленный и быстрый, может хранить ряд постоянно-размерных единиц памяти, именуемых *страницами*. В медленной памяти хранится N страниц, и в быстрой памяти хранится k страниц, где $k < N$. Обычно k намного меньше N.

При наличии запроса страницы p_i система должна сделать страницу p_i доступной в быстрой памяти. Если p_i уже находится в быстрой памяти – такая ситуация называется *попаданием*, – то системе ничего делать не нужно. В противном случае при *промахе* система вызывает ошибку – *страничный отказ* – и должна скопировать страницу p_i из медленной памяти в быструю память. При этом система сталкивается со следующей проблемой: какую страницу вытеснить из оперативной памяти, освободив место для p_i. Для того чтобы *минимизировать* число страничных отказов, выбор того, какую страницу вытеснять, должен делаться по-умному.

Типичными примерами пары быстрой и медленной памяти являются соответственно ОЗУ и жесткий диск или соответственно кеш процессора и ОЗУ. В общем случае мы будем называть быструю память «кешем». Из-за его важной роли в производительности почти каждой компьютерной системы замещение страниц широко изучалось начиная с 1960-х годов, и общепринятые схемы замещения страниц перечислены на рис. 5.2.

LRU	*Least Recently Used* (*Последней вошла/последней вышла*)
CLOCK	*Clock Replacement* (*Замещение по часам*)
FIFO	*First-In/First-Out* (*Первым вошла/первой вышла*)
LIFO	*Last-In/First-Out* (*Последней вошла/последней вышла*)
LFU	*Least Frequently Used* (*Наименее часто используемая страница*)
LFD	*Longest Forward Distance* (*Наибольшее расстояние вперед* *(вытеснить страницу, которая будет запрошена последней)*)

Рис. 5.2 ❖ Дисциплины замещения страниц: пять лучших онлайновых алгоритмов; последний, наибольшее расстояние вперед (LFD), является офлайновым. В разделе 5.2.6 мы увидим, что алгоритм LFD на самом деле является оптимальным алгоритмом замещения страниц

Все дисциплины кеширования на рис. 5.2, за исключением последней, являются онлайновыми алгоритмами; то есть это алгоритмы, которые принимают решения на основе прошлых событий, а не будущего. Последний алгоритм, наибольшего расстояния вперед (LFD), заменяет страницу, чей следующий запрос будет

последним, что требует знания о будущих запросах, и, следовательно, он является офлайновым алгоритмом.

5.2.1. Замещение страниц по требованию

Алгоритмы замещения страниц по требованию никогда не вытесняют страницу из кеша, если только нет страничного отказа, то есть они никогда не вытесняют упреждающе. Все дисциплины замещения страниц на рис. 5.2 являются замещением страниц по требованию. Мы рассматриваем *модель страничного отказа*, где взимаем 1 за перенос страницы в быструю память, и мы ничего не взимаем за доступ к уже существующей странице. Как показывает следующая теорема, это очень общая модель.

Теорема 5.11. Любой алгоритм замещения страниц, онлайновый или офлайновый, может быть модифицирован, став алгоритмом замещения страниц по требованию, без увеличения совокупной стоимости на любой последовательности запросов.

Доказательство. В алгоритме замещения страниц по требованию страничный отказ вызывает ровно одно вытеснение (то есть когда кеш заполнен), и между промахами нет никаких вытеснений. Поэтому пусть ALG равен любому алгоритму замещения страниц. Мы покажем, как его модифицировать, сделав его алгоритмом замещения страниц по требованию ALG′ таким, что на любой входной последовательности алгоритм ALG′ несет максимум столько же стоимостных издержек (делает максимум столько же перемещений страниц из медленной памяти в быструю), сколько алгоритм ALG, то есть $\forall\sigma$, $\mathrm{ALG}'(\sigma) \leqslant \mathrm{ALG}(\sigma)$.

Предположим, что алгоритм ALG имеет кеш размером k. Определим алгоритм ALG′ следующим образом: ALG′ также имеет кеш размером k плюс k регистров. Алгоритм ALG′ выполняет симуляцию алгоритма ALG, держа в своих k регистрах номера страниц, которые алгоритм ALG имел бы в своем кеше. Основываясь на поведении алгоритма ALG, алгоритм ALG′ принимает решения о вытеснении страниц[1].

Предположим, запрашивается страница p. Если p находится в кеше алгоритма ALG′, то мы просто обработаем запрос. В противном случае, если происходит страничный отказ, алгоритм ALG′ ведет себя в соответствии со следующими двумя случаями.

Случай 1: если у алгоритма ALG тоже страничный отказ (то есть число p не находится в регистрах) и алгоритм ALG вытесняет страницу из регистра i, освобождая место для p, то алгоритм ALG′ вытесняет страницу из ячейки i в своем кеше, освобождая место для р.

Случай 2: если у алгоритма ALG нет страничной ошибки, то число p должно быть, скажем, в регистре i. В этом случае алгоритм ALG′ вытесняет содержимое ячейки i в своем кеше и перемещает туда p.

Таким образом, алгоритм ALG′ является алгоритмом замещения страниц по запросу.

Теперь мы покажем, что алгоритм ALG′ несет стоимостные издержки не больше алгоритма ALG на любой входной последовательности; то есть алгоритм ALG′ имеет максимум столько же страничных отказов, сколько алгоритм ALG. Для этого мы сопрягаем каждое перемещение страницы алгоритмом ALG′ с перемеще-

[1] Принятое в этом доказательстве допущение состоит в том, что алгоритм ALG не перестраивает свои ячейки – то есть он никогда не переставляет содержимое своего кеша.

нием страницы алгоритмом ALG уникально следующим образом: если алгоритмы ALG′ и ALG имеют страничный отказ, то спарить перемещения страниц. В противном случае если у алгоритма ALG уже была страница в кеше, то, видимо, она была перемещена туда раньше, поэтому спарить то перемещение с текущим перемещением алгоритма ALG′.

Никогда не бывает, чтобы два разных перемещения в алгоритме ALG были сопряжены с одним перемещением в алгоритме ALG. Для того чтобы это увидеть, предположим, что на некоторой входной последовательности мы впервые сталкиваемся с ситуацией, когда два перемещения в алгоритме ALG′ сопряжены с одним и тем же перемещением в алгоритме ALG. Это может произойти только в следующей ситуации: запрашивается страница p, алгоритм ALG′ получает страничный отказ, он перемещает p в свой кеш, и мы спариваем это перемещение с прошлым перемещением в алгоритме ALG, которое уже было спарено! Но это означает, что страница p уже была запрошена, и после того как она была запрошена, она была вытеснена из кеша алгоритма ALG′ (в противном случае у алгоритма ALG′ не было бы страничного отказа).

Алгоритм ALG′ вытеснил страницу p, тогда как алгоритм ALG нет, поэтому они не были в одной ячейке. Но алгоритм ALG′ (первый раз) поместил p в ту же ячейку, что и алгоритм ALG, противоречие. Следовательно, мы не смогли бы спарить перемещение дважды. Таким образом, мы можем спаривать каждое перемещение ALG′ с перемещением ALG один к одному, и, следовательно, алгоритм ALG′ делает максимум столько же ходов, сколько алгоритм ALG (рис. 5.3).

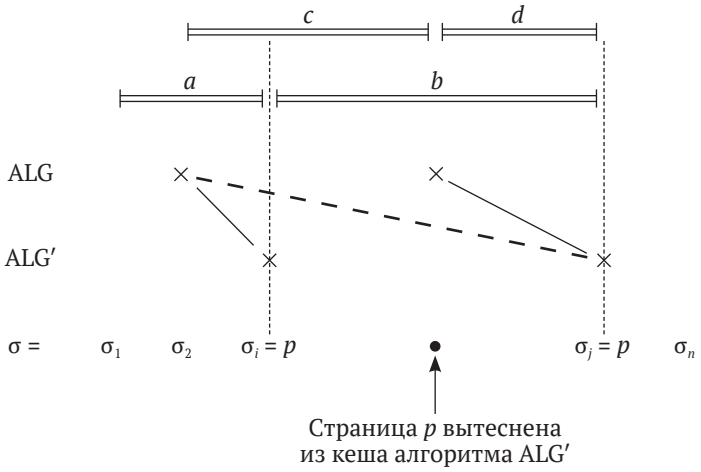


Рис. 5.3 ❖ Предположим, что i и j – это наименьшая пара такая, что существует страница p с тем свойством, что $\sigma_i = \sigma_j = p$. Алгоритм ALG имеет страничный отказ на σ_i и σ_j, и два соответствующих перемещения страниц ALG′ оба сопрягаются с одним перемещением страницы p, выполняемым алгоритмом ALG где-то на отрезке a. Мы покажем, что это невозможно: если алгоритм ALG′ имеет страничный отказ на $\sigma_i = \sigma_j = p$, то это означает, что где-то на отрезке b страница p вытесняется – этот момент обозначается символом «•». Если алгоритм ALG не вытеснил p на отрезке c, то алгоритм ALG также вытесняет страницу p в «•», и поэтому он должен тогда вернуть ее в кеш на отрезке d – мы бы спарили × в σ_j с этим перемещением. Если алгоритм ALG все-таки вытеснил p на отрезке c, то ему снова пришлось бы вернуть ее перед σ_j. В любом случае, существует более позднее перемещение p, которое будет сопряжено со страничным отказом алгоритма ALG в σ_j

Задача 5.12. На рис. 5.3 мы постулируем существование «наименьшей» пары i, j с заданными свойствами. Покажите, что если такая пара существует, то существует «наименьшая» такая пара; что означает «наименьший» в этом случае?

Идея состоит в том, что алгоритм ALG ничего не выигрывает, перемещая страницу в кеш упреждающе (до того, как страница действительно понадобится). Алгоритм ALG′ ожидает запроса перед выполнением того же действия.

В то же время (между временем, когда алгортим ALG доставляет страницу, и временем, когда она запрашивается и алгоритм ALG′ ее доставляет) алгоритм ALG′ может только выиграть, потому что в течение этого времени нет запросов этой страницы, но может быть запрос страницы, которую алгоритм ALG вытеснил упреждающе.

Обратите внимание, что в симуляции алгоритма ALG′ требуется только k дополнительных регистров, необходимых для отслеживания номеров страниц в кеше алгоритма ALG, поэтому данная симуляция является эффективной. □

Теорема 5.11 позволяет нам ограничить наше внимание алгоритмами замещения страниц по запросу и, как следствие, использовать термины «страничный отказ» и «перемещение страницы» взаимозаменяемо, в том смысле, что в контексте замещения страниц по запросу у нас есть перемещение страницы тогда и только тогда, когда у нас есть страничный отказ.

5.2.2. Первым вошел/первым вышел (FIFO)

Когда страница должна быть заменена, выбирается самая старая страница. Отсутствует необходимость вести учет времени перемещения страницы; для хранения всех страниц в памяти нам нужно лишь создать очередь FIFO (First-In/First-Out, первым вошел/первым вышел). Алгоритм FIFO легко понять и запрограммировать, но его производительность не очень хороша в общем случае.

Алгоритм FIFO также страдает от так называемой аномалии Белади. Предположим, что мы имеем последовательность запросов страниц: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5. Тогда у нас больше страничных отказов при $k = 4$, чем при $k = 3$. То есть алгоритм FIFO имеет больше страничных отказов с бóльшим кешем!

Задача 5.13. Для общего i предоставьте последовательность запросов страниц, которая иллюстрирует аномалию Белади, испытываемую алгоритмом FIFO на размерах кеша i и $i + 1$. В вашем анализе исходите из того, что кеш изначально пуст.

5.2.3. Наименее недавно использованная страница (LRU)

Оптимальный алгоритм замещения страниц (OPT) вытесняет страницу, чей следующий запрос является самым последним, и если некоторые страницы никогда не запрашиваются снова, то любая из них вытесняется. С точки зрения онлайновых алгоритмов этот алгоритм непрактичный, поскольку мы не знаем будущего.

Однако если мы используем недавнее прошлое как аппроксимацию ближайшего будущего, то мы заменим страницу, которая не использовалась в течение длительного периода времени. Этот подход представлен алгоритмом замещения наименее недавно использованной страницы (least recently used, LRU).

Замещение в алгоритме LRU связывает с каждой страницей время последнего использования этой страницы. Когда страница должна быть заменена, алгоритм

LRU выбирает ту страницу, которая не использовалась в течение самого длительного периода времени. Алгоритм LRU считается хорошим и часто реализуется – главная проблема заключается в том, как его реализовать; два типичных решения – использовать счетчики и стеки.

Счетчики: отслеживать время последней ссылки на данную страницу, обновляя счетчик каждый раз, когда мы ее запрашиваем. Эта схема требует поиска страниц в таблице для нахождения наименее недавно использованной страницы и записи в память при каждом запросе; очевидной проблемой может быть переполнение часов.

Стек: держать стопку номеров страниц. Всякий раз, когда на страницу ссылаются, она удаляется из стека и помещается сверху. Благодаря этому верх стека всегда является последней используемой страницей, а низ – наименее недавно использованной страницей (LRU). Поскольку элементы удаляются из середины стека, их лучше всего реализовывать с помощью *двусвязного списка*.

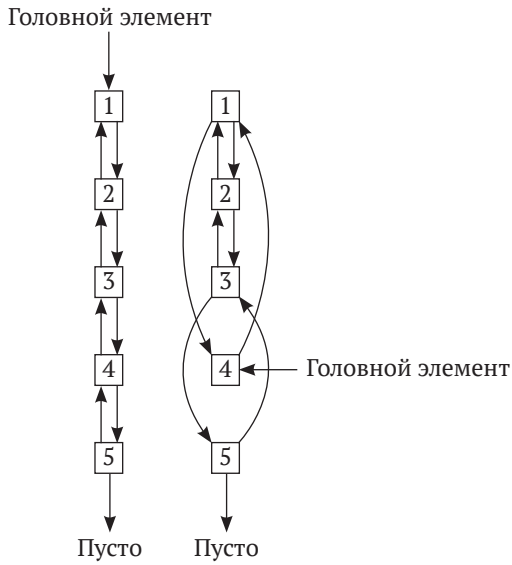


Рис. 5.4 ❖ Реализация стека алгоритма LRU с двусвязным списком. Запрошенная страница – это страница 4; в левом списке отображается состояние до запроса страницы 4, а в правом – состояние после обработки запроса

Сколько операций с указателем необходимо выполнить в примере на рис. 5.4? Шесть, если считать следующим образом: удалить старый головной элемент и добавить новый головной элемент (2 операции), соединить 4 с 1 (2 операции), соединить 3 с 5 (2 операции). Вместе с тем мы могли бы также подсчитать отсоединение 3 от 4 и 4 от 5, что дает еще 4 операции с указателем, давая нам в общей сложности 10 операций. Третьей стратегией было бы не считать отсоединение указателей, в этом случае мы получили бы половину этих операций, 5. На самом деле не имеет значения, как мы считаем, потому что все дело в том, что для перемещения запрошенной страницы наверх (после попадания) нам требуется небольшое постоянное число операций с указателем, независимо от того, как мы их считаем.

Задача 5.14. Перечислите операции с указателем, которые необходимо выполнить, если запрашиваемая страница отсутствует в кеше. Обратите внимание, что вы должны именно перечислить операции с указателем (а не просто дать «волшебное число»), поскольку мы только что показали, что есть три разных (и все разумные) способа их подсчета. Опять же, дело в том, что если страница должна быть доставлена из медленной памяти в кеш, то нам требуется небольшое постоянное число операций с указателем.

Задача 5.15. Мы реализовали алгоритм LRU с помощью двусвязного списка. В чем заключалась бы проблема, если бы вместо этого мы использовали обычный связный список? То есть если бы каждая страница имела только указатель на следующую страницу $i \rightsquigarrow j$, имея в виду, что i запрашивалась более недавно, чем j, но ни одна страница не запрашивалась позже i и раньше j.

Лемма 5.16. Алгоритм LRU не испытывает аномалии Белади (на любом размере кеша и любой последовательности запросов).

Доказательство. Пусть $\sigma = p_1, p_2, \cdots, p_n$ равно последовательности запросов, и пусть $\mathrm{LRU}_i(\sigma)$ равно числу отказов, которые алгоритм LRU испытывает на σ с кешем размера i. Покажем, что для всех i и σ имеет место следующее свойство:

$$\mathrm{LRU}_i(\sigma) \geqslant \mathrm{LRU}_{i+1}(\sigma). \tag{5.7}$$

Показав (5.7), из этого следует, что для любой пары $i < j$ и любой последовательности запросов σ $\mathrm{LRU}_i(\sigma) \geqslant \mathrm{LRU}_j(\sigma)$, и заключим, что алгоритм LRU не испытывает аномалии Белади.

Для демонстрации (5.7) мы определяем свойство двусвязных списков, которое называем *встраиванием*. Мы говорим, что двусвязный список размера i может быть встроен в другой двусвязный список размера $i + 1$, если два двусвязных списка идентичны, за исключением того, что более длинный может иметь еще один элемент в «конце». См. рис. 5.5, где двусвязный список размера 3 слева может быть встроен в двусвязный список размера 4 справа.

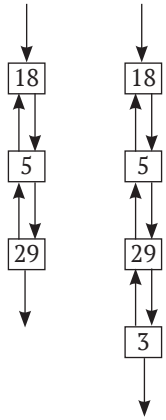


Рис. 5.5 ❖ Список слева может быть встроен в список справа

В начале обработки последовательности запросов, когда кеши заполняются, два списка идентичны, но как только кеши заполнены, кеш LRU_{i+1} будет иметь еще один элемент.

Утверждение 5.17. После обработки каждого запроса двусвязный список LRU_i может быть встроен в двусвязный список LRU_{i+1}.

Доказательство. Мы доказываем это утверждение по индукции на числе шагов. Базовый случай: если $n = 1$, то и LRU_i и LRU_{i+1} имеют отказ и доставляют p_1. Индукционный шаг: предположим, что утверждение соблюдается после шага n; мы покажем, что оно также соблюдается после шага $n + 1$. Рассмотрим следующие случаи: (1) LRU_i имеет попадание на p_{n+1}, (2) LRU_i имеет отказ на p_{n+1}, (2a) LRU_{i+1} также имеет отказ, (2b) LRU_{i+1} не имеет отказа.

Задача 5.18. Покажите, что в каждом случае свойство встраивания сохраняется.

Этим завершается доказательство утверждения. □

Задача 5.19. Используйте это утверждение для доказательства (5.7).

Этим завершается доказательство леммы. □

5.2.4. Маркировочные алгоритмы

Рассмотрим кеш размера k и зададим последовательность запросов σ. Мы разделяем последовательность запросов на фазы следующим образом: фаза 0 является пустой последовательностью. Для каждого $i \geqslant 1$ фаза i является максимальной последовательностью, следующей за фазой $i - 1$, которая содержит не более k несовпадающих запросов страниц; то есть если она существует, то фаза $i + 1$ начинается на запросе, который составляет $(k + 1)$-й несовпадающий запрос страницы с начала i-й фазы. Такое разбиение называется *k-членным фазовым разбиением*. Это разбиение хорошо определено и не зависит от любого конкретного алгоритма обработки σ.

Например, 3-членное фазовое разбиение:

$$\underbrace{1, 2, 1, 2, 1, 2, 3}_{\text{3-членная фаза № 1}}, \underbrace{4, 5, 6, 6, 6, 6, 6, 6, 4}_{\text{3-членная фаза № 2}}, \underbrace{5, 4, 7, 7, 7, 7, 1, 2}_{\text{3-членная фаза № 3}}.$$

Пусть σ равно любой последовательности запросов. Мы рассмотрим ее k-членное фазовое разбиение. Свяжем с каждой страницей бит, именуемой *маркером*. Маркировка выполняется ради анализа (она реализуется не алгоритмом, а «нами», для того чтобы отслеживать действия алгоритма). Для каждой страницы, когда ее маркерный бит установлен, мы говорим, что страница *промаркирована*, и в противном случае она *не промаркирована*.

Предположим, что в начале каждой k-членной фазы мы снимаем маркеры со всех страниц, и мы маркируем страницу при первом ее запросе во время k-членной фазы. Маркировочный алгоритм никогда не вытесняет промаркированную страницу из быстрой памяти.

Например, предположим, что $k = 2$ и σ является последовательностью запросов. Мы показываем 2-членные фазы для σ:

$$\sigma = \underbrace{1, 1, 3, 1}_{\text{2-членная фаза № 1}}, \underbrace{5, 1, 5, 1, 5, 1}_{\text{2-членная фаза № 2}}, \underbrace{3, 4, 4, 4}_{\text{2-членная фаза № 3}}, \underbrace{2, 2, 2, 2}_{\text{2-членная фаза № 4}}. \quad (5.8)$$

См. рис. 5.6, чтобы проанализировать маркировку в этом примере. Обратите внимание, что после каждой фазы с каждой страницы снимается маркер, и мы начинаем маркировать заново, и, за исключением последней фазы, все фазы всегда завершаются (в них есть ровно k несовпадающих запросов, в данном случае 2).

Как только запрос страницы p в фазе i сделан в маркировочном алгоритме, p остается в кеше до конца фазы i – при первом запросе страницы p она маркируется, и она остается помеченной на всю фазу, и маркировочный алгоритм никогда не вытесняет промаркированную страницу.

Интуиция подсказывает, что маркировочные алгоритмы являются хорошими схемами для замещения страницы, потому что в любой данной фазе существует не более k несовпадающих страниц, поэтому все они помещаются в кеш размера k; нет смысла вытеснять их в этой фазе, поскольку мы только можем потерять от вытеснения – вытесненная страница может быть запрошена снова.

Шаг	1	2	3	4	5	Шаг	1	2	3	4	5
1	×					10	×				×
2	×					11			×		
3	×		×			12			×	×	
4	×		×			13			×	×	
5					×	14			×	×	
6	×				×	15		×			
7	×				×	16		×			
8	×				×	17		×			
9	×				×	18		×			

Рис. 5.6 ❖ Маркировка в примере (5.8)

Теорема 5.20. Алгоритм LRU является маркировочным алгоритмом.

Доказательство. Мы аргументируем от противного; предположим, что алгоритм LRU на кеше размера k не является маркировочным алгоритмом. Пусть σ равно последовательности запросов, в которой существует k-членное фазовое разбиение, в течение которого вытесняется некая промаркированная страница p. Рассмотрим первый запрос страницы p в течение этой k-членной фазы:

$$\sigma = p_1, p_2, p_3, \ldots, \ldots, \underbrace{\ldots, p, \ldots, \ldots, \ldots}_{k\text{-членная фаза}}, \ldots, \ldots$$

Сразу после того, как p доставлена, она маркируется в кеше как совсем недавно использованная страница (то есть он помещается вверх двусвязного списка).

Для того чтобы p покинула кеш, алгоритм LRU должен получить страничный отказ, при этом p является наименее недавно используемой страницей. Из этого следует, что во время рассматриваемой k-членной фазы было запрошено $k + 1$ несовпадающих страниц: существует $k - 1$ страниц, которые продвинули p в конец списка, существует p и страница, которая вытеснила p. Противоречие; k-членная фаза имеет не более k несовпадающих страниц. □

5.2.5. Сброс при заполнении (FWF)

Сброс при заполнении FWF (flush when full) – это очень наивный алгоритм замещения страниц, который работает следующим образом: всякий раз, когда возникает страничный отказ и в кеше не остается места, вытеснить все страницы, находящиеся в кеше, – назовем это действие *сбросом*.

Точнее, мы рассмотрим следующую версию алгоритма FWF: каждая ячейка в кеше имеет один связанный с ней бит. Вначале все эти биты равны нулю. Когда запрашивается страница p, алгоритм FWF проверяет только ячейки с промаркированным битом. Если p найдена, то она доставляется. Если p не найдена, то она должна быть доставлена из медленной памяти (даже если она на самом деле находится в кеше, в непромаркированной ячейке). Алгоритм FWF разыскивает ячейку с нулевым битом, и происходит одно из следующих событий: (1) ячейка с нулевым битом (непромаркированная страница) найдена, и в этом случае алгоритм FWF заменяет эту страницу на p; (2) ячейка с нулевым битом не найдена (все страницы промаркированы), и в этом случае алгоритм FWF снимает маркеры со всех ячеек и заменяет любую страницу на p, и он обнуляет маркерный бит страницы p.

Задача 5.21. Покажите, что алгоритм FWF является маркировочным алгоритмом. Покажите, что алгоритм FIFO не является маркировочным алгоритмом.

Задача 5.22. Алгоритм замещения страниц ALG является консервативным, если на любой сплошной входной подпоследовательности, содержащей k или меньше запросов несовпадающих страниц, ALG будет иметь k или меньше страничных ошибок. Докажите, что алгоритмы LRU и FIFO являются консервативными, а алгоритм FWF – нет.

5.2.6. Наибольшее расстояние вперед (LFD)

Оптимальным алгоритмом замещения страниц оказывается *алгоритм с наибольшим расстоянием вперед LFD* (longest forward distance – см. рис. 5.2). Алгоритм LFD вытесняет страницу, которая не будет использоваться в течение длительного периода времени, и в связи с этим он не может быть реализован на практике, потому что он требует знаний о будущем. Тем не менее он очень полезен для сравнительных исследований, то есть для состязательного анализа.

Теорема 5.23. Алгоритм LFD является оптимальным (офлайновым) алгоритмом замещения страниц, то есть OPT = LFD.

Доказательство. Мы покажем, что если ALG является любым алгоритмом замещения страниц (онлайновым или офлайновым), то на любой последовательности запросов σ $\mathrm{ALG}(\sigma) \geqslant \mathrm{LFD}(\sigma)$. Как обычно, $\mathrm{ALG}(\sigma)$ обозначает число страничных отказов алгоритма ALG на последовательности запросов σ. Мы исходим из допущения, что все алгоритмы работают с кешем фиксированного размера k. Нам нужно доказать следующее утверждение.

Утверждение 5.24. Пусть ALG равно любому алгоритму замещения страниц. Пусть $\sigma = p_1, p_2, \ldots, p_n$ равно любой последовательности запросов. Тогда можно построить офлайновый алгоритм ALG_i, удовлетворяющий следующим трем свойствам:

1) алгоритм ALG_i обрабатывает первые $i - 1$ запросов σ точно так же, как алгоритм ALG;
2) если i-й запрос приводит к страничному отказу, то алгоритм ALG_i вытесняет из кеша страницу с «наибольшим расстоянием вперед»;
3) $ALG_i(\sigma) \leqslant ALG(\sigma)$.

Доказательство. Разделим σ на три сегмента следующим образом:

$$\sigma = \sigma_1, p_i, \sigma_2,$$

где σ_1 и σ_2 каждый обозначают блок запросов.

Напомним доказательство теоремы 5.11, где мы симулировали алгоритм ALG с помощью алгоритма ALG′, выполняя «теневое симулирование» содержимого кеша алгоритма ALG на множестве регистров, чтобы алгоритм ALG′ знал, что делать со своим кешем на основе содержимого этих регистров. Здесь мы делаем то же самое: алгоритм ALG_i выполняет симуляцию алгоритма ALG на множестве регистров.

Как и на σ_1, алгоритм ALG_i является алгоритмом ALG, из чего следует, что $ALG_i(\sigma_1) = ALG(\sigma_1)$, а также что они оба испытывают или не испытывают страничный отказ на p_i. Если этого не происходит, то пусть алгоритм ALG_i продолжает вести себя так же, как алгоритм ALG на σ_2 с целью, чтобы $ALG_i(\sigma) = ALG(\sigma)$.

Однако если они все-таки испытывают страничный отказ на p_i, то алгоритм ALG_i вытесняет страницу с наибольшим расстоянием вперед из своего кеша и заменяет ее на p_i. Если алгоритм ALG также вытесняет ту же страницу, то опять же, пусть алгоритм ALG_i ведет себя так же, как алгоритм ALG для остальной части σ с целью, чтобы $ALG_i(\sigma) = ALG(\sigma)$.

Наконец, предположим, что они оба испытывают отказ на p_i, но алгоритм ALG вытесняет некую страницу q, а алгоритм ALG_i вытесняет некую страницу p, и $p \neq q$; см. рис. 5.7. Если оба p, $q \notin \sigma_2$, то пусть алгоритм ALG_i ведет себя в точности как алгоритм ALG, за исключением того, что ячейки с p и q меняются местами (то есть когда алгоритм ALG вытесняет из q-ячейки, алгоритм ALG_i вытесняет из p-ячейки, и когда алгоритм ALG вытесняет из p-ячейки, алгоритм ALG_i вытесняет из q-ячейки).

ALG:		~~q~~			p	
ALG_i:		q			~~p~~	

Рис. 5.7 ❖ Алгоритм ALG вытесняет q, и алгоритм ALG_i вытесняет p, обозначаемые соответственно как ~~q~~ и ~~p~~, и они оба заменяют свою вытесненную страницу на p_i

Если $q \in \sigma_2$, но $p \notin \sigma_2$, то снова пусть алгоритм ALG_i, когда он вынужден вытеснять, действует так же, как алгоритм ALG, и при этом две ячейки взаимозаменяются. Обратите внимание, что в этом случае может случиться так, что $ALG_i(\sigma_2) < ALG(\sigma_2)$, поскольку алгоритм ALG вытеснил q, которая будет запрошена снова, но алгоритм ALG_i вытеснил p, которая не будет запрошена никогда.

Задача 5.25. Объясните, почему случай $q \notin \sigma_2$ и $p \in \sigma_2$ невозможен.

В противном случае можно допустить, что алгоритм ALG_i вытесняет страницу p, алгоритм ALG вытесняет страницу q, $p \neq q$ и

$$\sigma_2 = p_{i+1}, \ldots, q, \ldots, p, \ldots, p_n. \tag{5.9}$$

Допустим, что q, показанная в (5.9), является самым ранним экземпляром q в σ_2. Как и раньше, пусть алгоритм ALG_i действует так же, как алгоритм ALG с q-ячейкой и p-ячейкой. Мы точно знаем, что алгоритм ALG будет иметь отказ на q. Предположим, алгоритм ALG не имеет отказа на p; тогда алгоритм ALG никогда не вытеснял p, поэтому алгоритм ALG_i никогда не вытеснял q, и поэтому у алгоритма ALG_i не было отказа на q. Следовательно, $ALG_i(\sigma_2) \leqslant ALG(\sigma_2)$. □

Теперь мы покажем, как использовать утверждение 5.24, чтобы доказать, что алгоритм LFD на самом деле является оптимальным алгоритмом. Пусть $\sigma = p_1, p_2, \ldots, p_n$ равно любой последовательности запросов. По утверждению мы знаем, что $ALG_1(\sigma) \leqslant ALG(\sigma)$. Применив утверждение снова, получим $(ALG_1)_2(\sigma) \leqslant ALG_1(\sigma)$. Определим $\overline{ALG}_j$ как $(\cdots((ALG_1)_2)\cdots)_j$. Тогда мы получаем, что $\overline{ALG}_j(\sigma) \leqslant \overline{ALG}_{j-1}(\sigma)$.

Обратите внимание, что алгоритм $\overline{ALG}_n$ действует так же, как алгоритм LFD на σ, и, следовательно, мы имеем, что $LFD(\sigma) = \overline{ALG}_n(\sigma) \leqslant ALG(\sigma)$, и доказательство завершено. □

Отныне алгоритм OPT можно считать синонимом алгоритма LFD.

Теорема 5.26. Любой маркировочный алгоритм ALG является $\frac{k}{k-h+1}$-состязательным, где k – это размер кеша и h – размер кеша алгоритма OPT.

Доказательство. Зададим любую последовательность запросов σ и рассмотрим ее k-членное фазовое разбиение. Допустим, на секунду, что последняя фаза последовательности σ завершена (в общем случае последняя фаза может быть незавершенной).

Утверждение 5.27. Для любой фазы $i \geqslant 1$ маркировочный алгоритм ALG испытывает не более k страничных отказов.

Доказательство. Это вытекает из того, что на каждой фазе существует k несовпадающих ссылок на страницы. Как только страница запрашивается, она маркируется и, следовательно, не может быть удалена до завершения фазы. Следовательно, алгоритм ALG не может испытывать отказ дважды на одной и той же странице. □

Если обозначить i-ю k-членную фазу σ через σ_i, то мы можем выразить приведенное выше утверждение как $ALG(\sigma_i) \leqslant k$. Таким образом, если есть s фаз, то $ALG(\sigma) \leqslant s \cdot k$.

Утверждение 5.28. $OPT(\sigma) \geqslant s \cdot (k - h + 1)$, где опять же мы исходим из того, что запросы равны $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_s$, где σ_s является полным.

Доказательство. Пусть p_a равно первому запросу фазы i, а p_b равно последнему запросу фазы i. Предположим сначала, что существует фаза $i + 1$ (то есть i не является последней фазой). Тогда мы разбиваем σ на k-членные фазы (хотя кеш OPT имеет размер k, мы все же разбиваем σ на k-членные фазы):

$$\sigma = \ldots, p_{a-1}, \underbrace{p_a, p_{a+1}, \ldots, p_b}_{\substack{k\text{-членная} \\ \text{фаза № } i}}, \underbrace{p_{b+1}, \ldots, \ldots}_{\substack{k\text{-членная} \\ \text{фаза № } i+1}}$$

После обработки запроса p_a алгоритм OPT имеет не более $h - 1$ страниц в кеше, не включая p_a. Начиная с (и включая) p_{a+1} и вплоть до (и включая) p_{b+1} существует, по крайней мере, k несовпадающих запросов. Следовательно, на этом сегменте алгоритм OPT должен испытать, по крайней мере, $k - (h - 1) = k - h + 1$ отказов. Для того чтобы это увидеть, обратите внимание, что имеется два случая.

Случай 1: p_a снова появляется в $p_{a+1}, \ldots, p_{b+1}$; тогда в сегменте $p_{a+1}, \ldots, p_{b+1}$ существует, по крайней мере, $(k + 1)$ несовпадающих запросов, и поскольку алгоритм OPT имеет кеш размера h, независимо от содержимого кеша, то будет существовать, по крайней мере, $(k + 1) - h = k - h + 1$ страничных отказов.

Случай 2: предположим, что p_a больше не появляется в $p_{a+1}, \ldots, p_{b+1}$, тогда поскольку p_a запрашивается в начале фазы i, то она, безусловно, будет в кеше, когда мы начинаем обработку $p_{a+1}, \ldots, p_{b+1}$. Так как она не запрашивается снова, она занимает место в кеше, поэтому самое большее число $(h - 1)$ ячеек в кеше может быть занято некоторыми элементами, запрошенными в $p_{a+1}, \ldots, p_{b+1}$; поэтому снова у нас есть, по крайней мере, $k - (h - 1) = k - h + 1$ отказов.

Если i является последней фазой (то есть $i = s$), то у нас нет p_{b+1}, поэтому мы можем только сказать, что у нас есть, по крайней мере, $k - h$ отказов, но мы компенсируем это с помощью p_1, которая не учитывалась. □

Из утверждений 5.27 и 5.28 следует, что:

$$\mathrm{ALG}(\sigma) \leqslant s \cdot k \quad \text{и} \quad \mathrm{OPT}(\sigma) \geqslant s \cdot (k - h + 1),$$

с целью, чтобы

$$\frac{\mathrm{ALG}(\sigma)}{s \cdot k} \leqslant 1 \leqslant \frac{\mathrm{OPT}(\sigma)}{s \cdot (k - h + 1)}.$$

Поэтому, наконец:

$$\mathrm{ALG}(\sigma) \leqslant \left(\frac{k}{k - h + 1}\right) \cdot \mathrm{OPT}(\sigma).$$

В случае если σ можно разделить на s полных фаз.

Как уже упоминалось выше, в общем случае последняя фаза может оказаться неполной. Тогда мы повторяем этот анализ с $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_{s-1}$, и для σ_s мы в конце используем α, и, значит, мы получаем:

$$\mathrm{ALG}(\sigma) \leqslant \left(\frac{k}{k - h + 1}\right) \cdot \mathrm{OPT}(\sigma) + \alpha.$$

Задача 5.29. Разложите этот вывод.

Следовательно, в обоих случаях мы получаем, что любой маркировочный алгоритм ALG является $\frac{k}{k-h+1}$-состязательным.

Задача 5.30. Реализуйте все дисциплины на рис. 5.2. Оцените их экспериментально, выполнив их на цепочке случайных запросов и построив график их стоимости – по сравнению с LFD.

5.3. Ответы к избранным задачам

Задача 5.1. Подумайте о картотечном шкафе, упомянутом в начале этой главы. Когда мы перелистываем картотечный шкаф в поисках конкретного файла, мы по пути держим указатель в заданном месте (то есть в этом месте мы «помещаем палец» в качестве закладки), а затем вставляем полученный файл в это место почти без дополнительных издержек на поиск или реорганизацию. Мы также исходим из того, что нет смысла перемещать файл в более позднее место. Наконец, любая перестановка может быть записана как произведение перемещений (см. любой учебник по абстрактной алгебре).

Задача 5.3. Ответ равен 3. Обратите внимание, что в списке из n элементов существует $\binom{n}{2} = \frac{n\cdot(n-1)}{2}$ = неупорядоченных пар (и $n\cdot(n-1)$ упорядоченных пар), поэтому, для того чтобы вычислить Ф, мы перечисляем все эти пары и увеличиваем счетчик на 1 (начиная с 0) всякий раз, когда сталкиваемся с инверсией. Что касается второго вопроса, то обратите внимание, что хотя мы не знаем, как именно работает алгоритм OPT, мы знаем, что он обрабатывает σ с оптимальной стоимостью, то есть он обрабатывает σ самым дешевым способом. Таким образом, мы можем найти OPT(σ) исчерпывающим перечислением: для нашего списка x_1, x_2, …, x_l и последовательности запросов $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_n$ мы строим дерево, где корень промаркирован x_1, x_2, …, x_l, и дети корня – это все $l!$ перестановок списка. Тогда каждый узел, в свою очередь, имеет $l!$ детей; глубина дерева равна n. Мы рассчитываем стоимость каждой ветви и маркируем листья этими стоимостями. Стоимость каждой ветви представляет собой сумму стоимостей всех перемещений, необходимых для создания каждого следующего подряд узла, и стоимостей поисковых операций, связанных с соответствующими списковыми конфигурациями. Самой дешевой ветвью (а их может быть несколько) является именно OPT(σ).

Задача 5.4.

$$\sum_{i=1}^{n} t_i = \sum_{i=1}^{n}(a_i - \Phi_i + \Phi_{i-1}) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n}\Phi_{i-1} - \sum_{i=1}^{n}\Phi_i$$

$$= \sum_{i=1}^{n} a_i + \sum_{i=0}^{n-1}\Phi_i - \sum_{i=1}^{n}\Phi_i = \sum_{i=1}^{n} a_i + \left(\Phi_0 + \sum_{i=1}^{n-1}\Phi_i\right) - \left(\Phi_n + \sum_{i=1}^{n-1}\Phi_i\right)$$

и вычеркивание в последнем члене дают нам $\Phi_0 - \Phi_n + \sum_{i=1}^{n} a_i$.

Задача 5.5. Число элементов перед x в алгоритме MTF равно $(k-1)$, так как x находится в k-й позиции. Из этих $(k-1)$ элементов v равны $*$. Оба списка содержат точно такие же элементы, и $(k-1-v)$ элементов не-$*$ перед x в алгоритме MTF должны все быть перед x в алгоритме OPT (если элемент находится перед x в MTF и после x в OPT, то по определению он будет $*$).

Задача 5.6. В случае платного перемещения единственное изменение в числе инверсий может исходить от двух перемещенных элементов, так как относительный порядок по отношению ко всем другим элементам остается тем же. В случае бесплатного перемещения мы знаем, что алгоритм MTF уже поместил перемещенный элемент x в начало своего списка, и мы знаем, что бесплатные перемещения могут перемещать x только вперед, поэтому число элементов перед x в алгоритме OPT уменьшается на 1.

Задача 5.8. Алгоритм OPT является оптимальным офлайновым алгоритмом, и, следовательно, он должен работать, по крайней мере, так же хорошо, как и любой алгоритм ALG. Предположим, мы выполняем все получаемые запросы один за другим наивным способом, не делая никаких перестановок. Стоимость этой схемы ограничена примерно $n \cdot l$, числом запросов, умноженным на длину списка. Следовательно, $\mathrm{OPT}(\sigma) \leqslant n \cdot l$.

Задача 5.9. По теореме 5.2 мы знаем, что

$$\mathrm{MTF}(\sigma) \leqslant 2 \cdot \mathrm{OPT}_S(\sigma) + \mathrm{OPT}_P(\sigma) - \mathrm{OPT}_F(\sigma) - n$$

и правая часть равна

$$\leqslant 2 \cdot \mathrm{OPT}_S(\sigma) + \mathrm{OPT}_P(\sigma) \leqslant 2 \cdot (\mathrm{OPT}_S(\sigma) + \mathrm{OPT}_P(\sigma)) = 2 \cdot \mathrm{OPT}(\sigma).$$

Это показывает, что алгоритм MTF является 2-состязательным (с $\alpha = 0$). Для второй части мы повторим приведенный выше аргумент, но без «потери» составляющей n, поэтому мы имеем $\mathrm{MTF}(\sigma) \leqslant 2 \cdot \mathrm{OPT}(\sigma) - n$. С другой стороны, $\mathrm{OPT}(\sigma) \leqslant n \cdot l$ (по задаче 5.8), поэтому

$$2 \cdot \mathrm{OPT}(\sigma) - n \leqslant \left(2 - \frac{1}{l}\right) \cdot \mathrm{OPT}(\sigma).$$

Задача 5.12. В доказательстве теоремы 5.11 мы определяем паросочетание между перемещениями страниц (из медленной памяти в кеш) алгоритмов ALG и ALG′. Для того чтобы показать, что паросочетание получается один к одному, мы постулируем существование пары i и j, $i \neq j$, со следующими свойствами: (i) существует страница p такая, что $\sigma_i = \sigma_j = p$, (ii) алгоритм ALG′ испытывает отказ на σ_i и σ_j, и (iii) алгоритм ALG′ должен перемещать p в кеш для обработки σ_i и σ_j, и эти два перемещения сочетаются с таким же перемещением p алгоритмом ALG. Ради аргумента в доказательстве теоремы 5.11 мы хотим «наименьшую» такую пару – поэтому мы используем принцип наименьшего числа (см. стр. 210), для того чтобы показать, что если такие пары вообще существуют, то должны существовать пары, где $i + j$ минимально; мы берем любую такую пару.

Задача 5.13. Рассмотрим следующий список:

$$\underbrace{1, 2, 3 \ldots, i, i+1}_{1}, \underbrace{1, 2, 3, \ldots, i-1}_{2}, \underbrace{i+2}_{3}, \underbrace{1, 2, 3, \ldots, i, i+1, i+2}_{4}.$$

Если у нас есть кеш размера $i + 1$, то мы испытаем $i + 1$ отказов в сегменте 1 (потому что кеш изначально пуст), затем мы имеем $i - 1$ попаданий в сегменте 2, потом у нас есть еще один страничный отказ в сегменте 3, поэтому мы вытесняем 1, и в сегменте 4 мы отстаем на 1 на всем протяжении, поэтому мы имеем $i + 2$ страничных отказов. Следовательно, в общей сложности мы имеем $i + 1 + 1 + i + 2 = 2i + 4$ страничных отказов.

Предположим теперь, что мы имеем кеш размера i. Тогда мы испытаем $i + 1$ страничных отказов в сегменте 1, затем у нас будет $i - 1$ страничных отказов в сегменте 2 и один страничный отказ в сегменте 3, следовательно, $2i + 1$ страничных отказов перед началом этапа 4. Когда начинается сегмент 4, в кеше у нас уже есть страницы с 1 по $i - 1$, поэтому мы имеем попадания, а затем, когда запрашивается $i + 1$, мы имеем отказ, и когда запрашивается $i + 2$, мы имеем попадание и, следовательно, только один отказ в сегменте 4. Таким образом, мы имеем $2i + 2$ стра-

ничных отказов с кешем размера i. Для того чтобы разобраться в этом решении, убедитесь, что вы отслеживаете содержимое кеша после обработки каждого из четырех сегментов. Обратите внимание, что для правильной работы этого примера i должно равняться, по крайней мере, 3.

Задача 5.14. Если запрашиваемой страницы в кеше нет, то мы должны:

1) удалить пустой указатель (+1);
2) отсоединить последний элемент от предпоследнего (+2);
3) добавить пустой указатель из нового последнего элемента (+1);
4) удалить головной элемент (+1);
5) соединить новый первый элемент (запрошенная страница) со старым первым элементом (+2);
6) добавить новый головной элемент (+1), –

в общей сложности 8 операций с указателями.

Задача 5.15. Задача с односвязным списком заключается в том, что для нахождения предшественника страницы нам нужно начинать поиск всегда в начале списка, увеличивая накладные расходы на поддержание стека.

Задача 5.18. Случай 1. Если LRU_i имеет попадание в p_{n+1}, то то же самое происходит в LRU_{i+1}, так как LRU_i может быть встроен в LRU_{i+1} после шага h. Следовательно, ни один связный список не изменяется на шаге $n + 1$, поэтому LRU_i по-прежнему может быть встроен в LRU_{i+1} после шага $n + 1$.

Случай 2a. Если LRU_i и LRU_{i+1} оба имеют отказы на p_{n+1}, то каждый список претерпит два изменения: последняя (то есть наименее недавно использованная) страница будет удалена с «конца», и p_{i+1} будет добавлена в качестве головного элемента. Удаление последней страницы из каждого списка не останавливает встраивание LRU_i в LRU_{i+1}; оно просто приводит к тому, что каждый список заканчивается на одну страницу «раньше», поэтому страница в конце LRU_i после шага n теперь является «лишней» страницей в конце LRU_{i+1}. Ясно, что добавление нового головного элемента в каждый список также не подавляет встраивание, поэтому через $n + 1$ шагов LRU_i по-прежнему может быть встроен в LRU_{i+1}.

Случай 2b. Допустим, что LRU_i имеет отказ на p_{i+1} и LRU_{i+1} его не имеет. Оба списка должны содержать i страниц после шага n, и более того, страницы должны быть одинаковыми и в том же порядке, для того чтобы LRU_i мог быть встроен в LRU_{i+1}. Поэтому наименее недавно использованная страница в LRU_i будет удалена на шаге $n + 1$, но это не остановит LRU_i от его встраивания, так как удаленная страница теперь является лишней страницей в конце LRU_{i+1}. Опять же, очевидно, что добавление одной и той же страницы p_{n+1} в начало обоих списков не влияет на встраивание, поэтому индукция завершена.

Задача 5.19. После $n - 1$ шагов связный список LRU_i может быть встроен в список LRU_{i+1}. Рассмотрим любую страницу $p_n \in \sigma$. Если p_n находится в списке LRU_i, то она находится в том же индексе в LRU_{i+1}, поэтому стоимость доступа к p_n идентична. Если p_n нет в списке LRU_i, то она может быть последним элементом списка LRU_{i+1}, в каковом случае LRU_{i+1} получает доступ к p_n с меньшей стоимостью, чем у LRU_i. В противном случае ее нет ни в одном из списков, поэтому стоимость снова одинакова. Поскольку это верно для каждой $p \in \sigma$, мы можем заключить, что $LRU_i(\sigma) \leqslant LRU_{i+1}(\sigma)$.

Задача 5.21. Алгоритм FWF действительно реализует маркирующий бит, поэтому он представляет собой почти маркировочный алгоритм по определению.

Алгоритм FIFO не является маркировочным алгоритмом, потому что при $k = 3$ и последовательности запросов 1, 2, 3, 4, 2, 1 он вытеснит элемент 2 во второй фазе, даже если он промаркирован.

Задача 5.22. Мы должны исходить из того, что кеш имеет размер k. В противном случае утверждение не является истинным: например, предположим, что у нас есть кеш размера 1 и последовательность: 1, 2, 1, 2. Тогда в этой последовательности из 4 запросов есть только 2 несовпадающих запроса, но с кешем размера 1 было бы 4 отказа для любого алгоритма замещения страниц. С кешем размера k алгоритм LRU никогда не будет вытеснять страницу во время этой сплошной подпоследовательности, как только страница была запрошена. Таким образом, каждый несовпадающий запрос страницы может вызвать только один отказ. То же самое касается и алгоритма FIFO. Таким образом, они оба являются консервативными алгоритмами. Вместе с тем существует возможность, что на полпути через сплошную подпоследовательность кеш алгоритма FWF будет заполнен, и алгоритм FWF вытеснит все страницы. Следовательно, алгоритм FWF может иметь более одного отказа страницы на той же странице во время этой сплошной подпоследовательности.

Задача 5.25. Если $p \in \sigma_2$ и $q \notin \sigma_2$, то q имела бы «более длинное расстояние вперед», чем p, и поэтому p не была бы вытеснена алгоритмом ALG. Скорее, алгоритм ALG вытеснил бы q или какую-то другую страницу, которая не будет запрошена снова.

Задача 5.29. Пусть $\sum_{s-1}$ обозначает первые $s - 1$ полных фаз последовательности σ. Мы знаем, что

$$\mathrm{ALG}(\textstyle\sum_{s-1}) \leqslant \left(\frac{k}{k-h+1}\right) \cdot \mathrm{OPT}(\textstyle\sum_{s-1}).$$

Ясно, что в любом из указанных алгоритмов может произойти не более $k - 1$ отказов в фазе σ_s, так как это не полная k-членная фаза. Поэтому

$$\begin{aligned}\mathrm{ALG}(\sigma) &\leqslant \mathrm{ALG}(\textstyle\sum_{s-1}) + k - 1\\ &\leqslant \left(\frac{k}{k-h+1}\right) \cdot \mathrm{OPT}(\textstyle\sum_{s-1}) + k - 1\\ &\leqslant \left(\frac{k}{k-h+1}\right) \cdot \mathrm{OPT}(\sigma) + k - 1.\end{aligned}$$

5.4. Примечания

Очень полным учебником по онлайновым алгоритмам является [Borodin и El-Yaniv (1998)]. См. также [Dorrigiv и Lopez-Ortiz (2009)] из колонки новостей SIGACT по онлайновым алгоритмам.

Глава 6

Рандомизированные алгоритмы

> Даже на расшировку сообщения, зашифрованного на трехроторной Энигме, может уйти двадцать четыре часа, поскольку бомбы пробивались сквозь миллиарды перестановок. Работа четырехроторной Энигмы, умножающей числа на двадцать шесть порядков, теоретически заняла бы добрую часть месяца.
>
> *Enigma, стр. 27 [Harris (1996)]*

Довольно любопытно, что мы можем проектировать процедуры, которые, столкнувшись с изобилием вариантов, вместо того чтобы кропотливо изучать все возможные ответы на эти варианты, подбрасывают монетку, чтобы решить, в какую сторону идти, и, несмотря на это, «как правило», получают правильный результат.

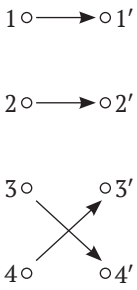
Совершенно очевидно, что когда прибегаем к случайности, мы экономим время, но совершенно удивительно то, что результаты таких процедур могут быть содержательными. То есть существуют задачи, решение которых кажется вычислительно очень трудным, но когда допускается использование случайности, появляется возможность проектировать процедуры, которые решают эти сложные задачи удовлетворительно: результат этой процедуры является правильным с небольшой вероятностью ошибки. На самом деле эта ошибка может быть настолько мала, что становится ничтожной (скажем, 1 к 2^{100} – что равно примерному числу атомов в наблюдаемой Вселенной). Таким образом, многие эксперты считают, что определение «допустимо вычислимого» должно подразумевать «вычислимое за полиномиальное время со случайностью», а не просто «за полиномиальное время». Появление рандомизированных алгоритмов было связано с задачей проверки простоты (примарности), что, в свою очередь, было вызвано расцветом области криптографии. Исторически первый такой алгоритм появился благодаря работе [Solovay и Strassen (1977)]. Проверка простоты остается одной из лучших задач для демонстрации возможностей рандомизированных алгоритмов; в этой главе мы представим алгоритм Рабина–Миллера, который появился после алгоритма Соловея–Штрассена, но он несколько проще. Интересно отметить, что теперь широко известен полиномиально-временной алгоритм проверки простоты (благодаря работе [Agrawal и соавт. (2004)]), но алгоритм Рабина–Миллера по-прежнему

используется на практике, поскольку он эффективнее (и его вероятность ошибки незначительна).

В этой главе мы представим три примера рандомизированных алгоритмов: алгоритм идеального паросочетания, сопоставление со строковым образцом и, наконец, алгоритм Рабина–Миллера. В заключение мы поговорим о криптографии.

6.1. Идеальное паросочетание

Рассмотрим двудольный граф $G = (V \cup V', E)$, где $E \subseteq V \times V'$. Его матрица смежности определяется следующим образом: $(A_G)_{ij} = x_{ij}$, если $(i, j') \in E_G$, и $(A_G)_{ij} = 0$ в противном случае. См. пример, приведенный на рис. 6.1.



$$\begin{pmatrix} x_{11} & 0 & 0 & 0 \\ 0 & x_{22} & 0 & 0 \\ 0 & 0 & 0 & x_{34} \\ 0 & 0 & x_{43} & 0 \end{pmatrix}$$

Рис. 6.1 ❖ Слева мы имеем двудольный граф $G = (V \cup V', E)$, где $V = \{1, 2, 3, 4\}$, $V' = \{1', 2', 3', 4'\}$ и $E \subseteq V \times V'$, $E = \{(1, 1'), (2, 2'), (3, 4'), (4, 3')\}$. Справа мы имеем соответствующую матрицу смежности A_G

Пусть S_n равно множеству всех перестановок n элементов. Точнее, S_n является множеством биекций (взаимно-однозначных соответствий) $\sigma : [n] \to [n]$. Ясно, что $|S_n| = n!$, и из алгебры хорошо известно, что любая перестановка $\sigma \in S_n$ может быть записана как произведение перемещений, или транспозиций (то есть перестановок, которые обмениваются двумя элементами в $[n]$ и оставляют все остальные элементы нетронутыми). Любая перестановка в S_n может быть записана как произведение перемещений, и хотя существует много способов это проделать (то есть представление перемещений не является уникальным), четность числа перемещений постоянна для любой данной перестановки σ. Пусть $\mathrm{sgn}(\sigma)$ равно 1 либо -1, в зависимости от четности σ, соответственно, четной или нечетной.

Напомним формулу Лагранжа для определителя:

$$\det(A) = \sum_{\sigma \in S_n} \mathrm{sgn}(\sigma) \prod_{i=1}^{n} A_{i\sigma(i)}. \tag{6.1}$$

Лемма 6.1. Пусть $G = (V \cup V', E)$ равно графу, где $n = |V| = |V'|$ и $E \subseteq V \times V'$. Тогда граф G имеет идеальное паросочетание (то есть каждая вершина в V может быть спарена с уникальной вершиной в V') тогда и только тогда, когда имеет место, что $\det(A_G) = \sum_{\sigma \in S_n} \mathrm{sgn}(\sigma) \prod_{i=1}^{n} (A_G)_{i\sigma(i)} \neq 0$.

Задача 6.2. Докажите лемму 6.1.

Поскольку $|S_n| = n!$, вычисление суммирования над всеми σ в S_n, как и в (6.1), вычислительно очень дорого, поэтому мы случайно назначаем значения элемен-

там x_{ij}. Целочисленный определитель, в отличие от символического, может быть вычислен очень эффективно – например, с помощью *алгоритма Берковица*. Пусть $A_G(x_1, \ldots, x_m)$, $m = |E_G|$ равно A_G, где ее переменные переименованы в $x_1, \ldots, x_m$. Обратите внимание, что $m \leqslant n^2$ и каждый x_i представляет некоторое x_{ij}. Мы получаем рандомизированный алгоритм для задачи идеального паросочетания – см. алгоритм 6.1.

Алгоритм 6.1. Идеальное паросочетание

Выбрать m случайных целых чисел $i_1, \ldots, i_m$ в $\{1, \ldots, M\}$, где $M = 2m$
вычислить целочисленный определитель матрицы $A_G(i_1, \ldots, i_m)$
if $\det(A_G(i_1, \ldots, i_m)) \neq 0$ **then**
    return да, G имеет идельное паросочетание
else
    return нет, G, вероятно, не имеет идеального паросочетания
end if

Алгоритм 6.1 представляет собой алгоритм Монте-Карло с полиномиальным временем выполнения: ответы «да» являются надежными и окончательными, в то время как ответы «нет» находятся под угрозой ложного отрицания. Ложное отрицание может возникнуть следующим образом: G может иметь идеальное паросочетание, но $(i_1, \ldots, i_m)$ может оказаться корнем многочлена $\det(A_G(x_1, \ldots, x_m))$. Однако, как мы вскоре увидим, вероятность ложного отрицания (то есть вероятность ошибки) можно сделать ничтожно малой.

В строке 1 алгоритма 6.1 мы несколько загадочно говорим: «Выбрать m случайных чисел». Каким образом мы «выбираем» эти случайные числа? Оказывается, ответить на этот вопрос непросто, а получение источника случайности является ахиллесовой пятой рандомизированных алгоритмов. В нашем распоряжении есть наука о генераторах псевдослучайных чисел и другие подходы, но эта огромная тема выходит за рамки настоящей книги, и поэтому мы наивно предположим, что у нас есть «некий источник случайности». Мы хотим показать правильность нашего рандомизированного алгоритма, поэтому нам нужно показать, что вероятность ошибки ничтожно мала. Начнем с леммы Шварца–Зиппеля.

Лемма 6.3 (Шварца–Зиппеля). Рассмотрим многочлены над $\mathbb{Z}$, и пусть $p(x_1, \ldots, x_m) \neq 0$ равно многочлену, где степень каждой переменной $\leqslant d$ (когда многочлен записан как сумма одночленов), и пусть $M > 0$. Тогда число m-членных кортежей $(i_1, \ldots, i_m) \in \{1, 2, \ldots, M\}^m$ таких, что $p(i_1, \ldots, i_m) = 0$, равно $\leqslant mdM^{m-1}$.

Доказательство. Индукция на m (числе переменных). Если $m = 1$, то по основной теореме алгебры $p(x_1)$ может иметь не более $d = 1 \cdot d \cdot M^{1-1}$ корней.

Предположим, что лемма соблюдается для $(m - 1)$, и теперь мы хотим дать верхнюю границу mdM^{m-1} на числе кортежей $(i_1, \ldots, i_m)$ таких, что $p(i_1, \ldots, i_m) = 0$. Сначала мы записываем $p(x_1, \ldots, x_m)$ как $y_d x_m^d + \cdots + y_0 x_m^0$, где каждый коэффициент $y_i = y_i(x_1, \ldots, x_{m-1}) \in \mathbb{Z}[x_1, \ldots, x_{m-1}]$.

Так сколько же существует кортежей $(i_1, \ldots, i_m)$ таких, что $p(i_1, \ldots, i_m) = 0$? Мы разбиваем такие кортежи на два множества: те, которые задают $y_d = 0$, и те, которые этого не делают. Результат ограничен сверху суммой верхних границ двух множеств; теперь мы дадим эти верхние границы.

Множество 1. По индукционной гипотезе y_d равно нулю для не более $(m-1)dM^{m-2}$ числа $(i_1, \ldots, i_{m-1})$ кортежей, и x_m может принимать M значений, поэтому $p(x_1, \ldots, x_m)$ равно нулю для не более $(m-1)dM^{m-1}$ кортежей. Обратите внимание, что здесь мы переоцениваем; мы берем *все* кортежи, которые устанавливают $y_d = 0$.

Множество 2. Для каждой комбинации M^{m-1} значений для $x_1, \ldots, x_{m-1}$ существует не более d корней результирующего многочлена (опять же по основной теореме алгебры), то есть dM^{m-1}. Обратите внимание, что мы снова переоцениваем, так как некоторые из этих установок в $x_1, \ldots, x_m$ приведут к $y_d = 0$.

Добавление двух верхних границ дает нам mdM^{m-1}. □

Лемма 6.4. Алгоритм 6.1 является правильным.

Доказательство. Мы хотим показать, что алгоритм 6.1 для идеального паросочетания является надежным алгоритмом Монте-Карло, что означает, что ответы «да» на 100 % верны, в то время как ответы «нет» допускают незначительную вероятность ошибки.

Если алгоритм отвечает «да», то $\det(A_G(i_1, \ldots, i_m)) \neq 0$ для некоторых случайно отобранных $i_1, \ldots, i_m$, но тогда символический определитель $\det(A_G(x_1, \ldots, x_m)) \neq 0$, и поэтому по лемме 6.1 G имеет идеальное паросочетание. Таким образом, ответы «да» с абсолютной уверенностью указывают на то, что существует идеальное паросочетание.

Предположим, что ответ равен «нет». Тогда мы применяем лемму 6.3 к $\det(A_G(x_1, \ldots, x_m))$ с $M = 2m$ и получаем, что вероятность ложного отрицания равна

$$\leqslant \frac{m \cdot d \cdot M^{m-1}}{M^m} = \frac{m \cdot 1 \cdot (2m)^{m-1}}{(2m)^m} = \frac{m}{2m} = \frac{1}{2}.$$

Теперь предположим, что мы проводим «много независимых экспериментов», имея в виду, что мы выполняем алгоритм 6.1 k число раз, всякий раз выбирая случайное множество $i_1, \ldots, i_m$. Тогда если ответ всегда выходит равным нулю, то мы знаем, что вероятность ошибки равна $\leqslant (\frac{1}{2})^k = \frac{1}{2^k}$. Для $k = 100$ ошибка становится незначительной. □

В последнем абзаце доказательства леммы 6.4 мы говорим, что выполняем алгоритм 6.1 k число раз, и таким образом снижаем вероятность ошибки до того, что она меньше, чем $\frac{1}{2^k}$, которая для $k = 100$ действительно ничтожно мала. Выполнение алгоритма k раз для получения ответа называется *амплификацией* (потому что мы резко уменьшаем вероятность ошибки и тем самым усиливаем уверенность в правильности ответа); обратите внимание, что красота этого подхода заключается в том, что пока мы выполняем алгоритм k раз, вероятность ошибки снижается экспоненциально быстро до $\frac{1}{2^k}$. Глядя на все в перспективе, если $k = 100$, то $\frac{1}{2^{100}}$ – это такой мизер, что по сравнению с вероятностью попадания в землю большого метеорита – во время работы алгоритма – является виртуальной определенностью (и если уж на то пошло, то попадание большого метеорита избавит любого от необходимости выполнять алгоритмы).

Задача 6.5. Покажите, как использовать алгоритм 6.1 для поиска идеального па-

росочетания.

Идеальное паросочетание может быть легко сведено[1] в *задачу о максимальном потоке*: в качестве примера рассмотрим задачу идеального паросочетания, приведенную на рис. 6.1; добавим два новых узла s, t, свяжем s со всеми узлами в левом столбце задачи паросочетания и свяжем t со всеми узлами в правом столбце задачи паросочетания, дадим каждому ребру емкость 1 и зададимся вопросом, есть ли поток $\geqslant n$ (где n – это число узлов в каждой из двух компонент данного двудольного графа) из s в t; см. рис. 6.2.

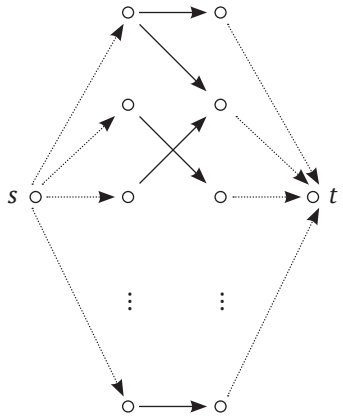


Рис. 6.2 ❖ Сведение идеального паросочетания к максимальному потоку

Поскольку задача о максимальном потоке может быть решена за полиномиальное время без использования случайности, то из этого следует, что идеальное паросочетание также может быть решено за полиномиальное время без случайности. Тем не менее смысл этого раздела состоял в том, чтобы показать простой рандомизированный алгоритм, и именно этого мы и достигли.

6.2. Сопоставление с образцом

В этом разделе мы спроектируем рандомизированный алгоритм сопоставления (паросочетания) с образцом. Рассмотрим множество символьных цепочек, или строк, над {0, 1}, и пусть $M : \{0, 1\} \to M_{2\times 2}(\mathbb{Z})$, то есть M – это отображение из строк в 2×2-матрицы над целыми числами ($\mathbb{Z}$), определяемое следующим образом:

$$M(\varepsilon) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}; \quad M(0) = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}; \quad M(1) = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix},$$

и для строк $x, y \in \{0, 1\}^*$, $M(xy) = M(x)M(y)$, где операция с левой стороны является конкатенацией строк, а операция с правой стороны – умножением матриц.

Задача 6.6. Покажите, что $M(x)$ хорошо определено, то есть независимо от того,

[1] Напомним, что мы кратко рассмотрели идею редуцирования на стр. 83.

как мы оцениваем M на x, мы всегда получаем один и тот же результат. Также покажите, что M взаимно-однозначно.

Задача 6.7. Покажите, что для $x \in \{0,1\}^n$ элементы $M(x)$ ограничены n-м числом Фибоначчи. Формальное определение чисел Фибоначчи см. в задаче 9.5 на стр. 209.

Рассматривая матрицы $M(x)$ по модулю подходящего простого числа p, то есть принимая все элементы $M(x)$ по модулю простого числа p, мы выполняем эффективное рандомизированное сопоставление с образцом. Мы хотим определить, является ли x подстрокой y, где $|x| = n$, $|y| = m$, $n \leqslant m$. Определим

$$y(i) = y_i y_{i+1} \ldots y_{n+i-1}$$

для соответствующего $i \in \{1, \ldots, m - n + 1\}$. Выберем простое число $p \in \{1, \ldots, nm^2\}$, и пусть $A = M(x) \pmod p$ и $A(i) = M(y(i)) \pmod p$. Заметим, что

$$A(i+1) = M^{-1}(y_i)A(i)M(y_{n+i}) \pmod p,$$

что делает вычисление последующих $A(i)$ эффективным.

Алгоритм 6.2. Сопоставление с образцом

Предусловие: $x, y \in \{0,1\}^*$, $|x| = n$, $|y| = m$ и $n \leqslant m$

```
 1: отобрать случайное простое число p ⩽ nm²
 2: A ← M(x) (mod p)
 3: B ← M(y(1)) (mod p)
 4: for i = 1, ..., m − n + 1 do
 5:         if A = B then
 6:                 if x = y(i) then
 7:                         return найдено паросочетание в позиции i
 8:                 end if
 9:         end if
10:         B ← M⁻¹(y_i) · B · M(y_{n+i})
11: end for
```

Какова вероятность получения ложного утверждения? Это вероятность того, что $A(i) = M(y(i)) \pmod p$, даже если $A(i) \neq M(y(i))$. Это меньше, чем вероятность того, что $p \in \{1, \ldots, nm^2\}$ делит (ненулевой) элемент в $A(i) - M(y(i))$. Поскольку эти элементы ограничены условием $F_n < 2^n$, менее n несовпадающих простых чисел могут разделить любой из них. С другой стороны (по теореме о распределении простых чисел), в $\{1, \ldots, nm^2\}$ существует $\pi(nm^2) \approx (nm^2)/(\log(nm^2))$ простых чисел. Таким образом, вероятность ложного утверждения равна $O(1/m)$.

Обратите внимание, что алгоритм 6.2 не имеет ошибки; он рандомизирован, но все потенциальные ответы проверяются на *ложное утверждение* (в строке 6 алгоритма). Проверка этих потенциальных кандидатов называется *снятием отпечатков пальцев*. Идея снятия отпечатков пальцев состоит в том, чтобы проверять только те подстроки, которые «выглядят» как хорошие кандидаты, гарантируя, что когда мы «вынюхиваем» кандидата, мы никогда не пропустим решение (в этом случае если $x = y(i)$ для некоторого i, то $y(i)$ всегда будет кандидатом).

С другой стороны, могут быть j такие, что $x \neq y(j)$, и все же они являются кандидатами; но вероятность этого мала. Использование случайности в алгоритме 6.2

просто снижает средневременную сложность процедуры; такие алгоритмы называются *алгоритмами Лас-Вегас*.

6.3. Проверка простоты

Один из способов определить, является ли число p простым, – попробовать все возможные числа $n < p$ и проверить, являются ли они делителями[1]. Очевидно, что эта процедура методом грубой силы имеет экспоненциально-временную сложность по длине p, и поэтому она имеет запредельную временную стоимость. Хотя полиномиально-временной (детерминированный) алгоритм для примарности (определения простоты) теперь известен (см. работу [Agrawal и соавт. (2004)]), рандомизированный алгоритм Рабина–Миллера для проверки простоты проще и эффективнее и поэтому до сих пор используется на практике.

Малая теорема Ферма (см. теорему 9.22) предоставляет своего рода «тест» на простоту, называемый проверкой Ферма; алгоритм Рабина–Миллера (алгоритм 6.3) основан на этой проверке. Когда мы говорим, что p проходит проверку Ферма в a, мы имеем в виду, что $a^{(p-1)} \equiv 1 \pmod p$. Таким образом, все простые числа проходят проверку Ферма для всех $a \in \mathbb{Z}_p - \{0\}$.

К сожалению, существуют также составные числа n, которые проходят проверки Ферма для каждого $a \in \mathbb{Z}_n^*$; это так называемые *числа Кармайкла*, например 561, 1105, 1729 и т. д.

Лемма 6.8. Если p равно составному числу не Кармайкла, то оно проходит не более половины проверок в $\mathbb{Z}_p^*$. То есть если p равно составному числу не Кармайкла, то для большей половины a во множестве $\mathbb{Z}_p^*$ имеет место, что $a^{(p-1)} \equiv 1 \pmod p$.

Доказательство. Мы говорим, что a является *свидетелем* p, если a не проходит проверку Ферма для p. То есть a является свидетелем, если $a^{(p-1)} \not\equiv 1$. Пусть $S \subseteq \mathbb{Z}_p^*$ состоит из тех элементов $a \in \mathbb{Z}_p^*$, для которых $a^{(p-1)} \equiv 1 \pmod p$. Легко проверить, что S является на самом деле подгруппой $\mathbb{Z}_p^*$. Следовательно, по теореме Лагранжа (теорема 9.2.3) $|S|$ должно делить $|\mathbb{Z}_p^*|$. Предположим теперь, что существует элемент $a \in \mathbb{Z}_p^*$, для которого $a^{(p-1)} \not\equiv 1 \pmod p$. Тогда $S \neq \mathbb{Z}_p^*$, поэтому лучшее, чем оно может быть, – это «половина» $\mathbb{Z}_p^*$, поэтому $|S|$ должно быть не более половины $|\mathbb{Z}_p^*|$. □

Задача 6.9. Дайте альтернативное доказательство леммы 6.8 без групп.

Некое число является *псевдопростым*, если оно является либо простым, либо Кармайкла. Последняя лемма наводит на мысль об алгоритме для проверки псевдопростоты: на входном p проверить истинность $a^{(p-1)} \equiv 1 \pmod p$ для некоторого случайного $a \in \mathbb{Z}_p^* - \{0\}$. Если p не проходит эту проверку (то есть $a^{(p-1)} \not\equiv 1 \pmod p$), то p точно является составным. Если p проходит эту проверку, то p, вероятно, является псевдопростым. Мы показываем, что вероятность ошибки в данном случае составляет $\leqslant 1/2$. Предположим, что p не является псевдопростым. Если $\gcd(a, p) \neq 1$, то $a^{(p-1)} \not\equiv 1 \pmod p$ (по пропозиции 9.20), поэтому, исходя из того что p прошло проверку, действительно должно иметь место, что $\gcd(a, p) = 1$, и поэтому $a \in \mathbb{Z}_p^*$. Но тогда, по лемме 6.8, по крайней мере, половина элементов $\mathbb{Z}_p^*$ являются свиде-

[1] Этот раздел требует немного знаний теории чисел; см. раздел 9.2 для получения всей необходимой общей информации.

телями не псевдопростоты.

Задача 6.10. Покажите, что если $\gcd(a, p) \neq 1$, то $a^{(p-1)} \not\equiv 1 \pmod{p}$.

Неформальный алгоритм псевдопростоты, описанный в предыдущем абзаце, является основой для алгоритма Рабина–Миллера, который мы обсудим далее. Алгоритм Рабина–Миллера расширяет проверку псевдопростоты для работы с числами Кармайкла.

Алгоритм 6.3. Рабин–Миллер

```
 1: Если n = 2, принять; если n является четным и n > 2, отклонить.
 2: Выбрать случайно положительное a в Z_n.
 3: if a(n − 1) ≢ 1 (mod n) then
 4:         отклонить
 5: else
 6:         Отыскать s, h такие, что s является нечетным и n − 1 = s2^h
 7:         Вычислить последовательность a^{s·2^0}, a^{s·2^1}, a^{s·2^2}, ..., a^{s·2^h} (mod n)
 8:         if все элементы в последовательности равны 1 then
 9:                 принять
10:         else if последний элемент, отличный от 1, равен –1 then
11:                 принять
12:         else
13:                 отклонить
14:         end if
15: end if
```

Обратите внимание, что этот алгоритм является полиномиально-временным (рандомизированным) алгоритмом: вычисление степеней $(\mathrm{mod}\, n)$ может быть эффективно выполнено с помощью многократного *возведения в квадрат* – например, если $(n-1)_b = c_r \ldots c_1 c_0$, то вычислить

$$a_0 = a,\ a_1 = a_0^2,\ a_2 = a_1^2, \ldots, a_r = a_{r-1}^2 \pmod{n},$$

и поэтому $a^{n-1} = a_0^{c_0} a_1^{c_1} \cdots a_r^{c_r} \pmod{n}$. Таким образом, получение степеней в строках 6 и 7 не является проблемой.

Задача 6.11. Реализуйте алгоритм Рабина–Миллера. В первой наивной версии алгоритм должен работать на целочисленных входах (встроенном типе `int`). Во второй, более сложной версии алгоритм должен выполняться на входах в виде чисел, закодированных как двоичные строки, с трюком многократного возведения в квадрат, чтобы справиться с большими числами.

Теорема 6.12. Если n является простым числом, то алгоритм Рабина–Миллера его принимает; если n является составным, то алгоритм его отклоняет с вероятностью $\geqslant 1/2$.

Доказательство. Если n является простым числом, то по малой теореме Ферма $a^{(n-1)} \equiv 1 \pmod{n}$, поэтому строка 4 алгоритма не может отклонить n. Предположим, что строка 13 отклоняет n; тогда существует b в $\mathbb{Z}_n$ такое, что $b^2 \not\equiv \pm 1 \pmod{n}$

и $b^2 = 1 \pmod n$. Поэтому $b^2 - 1 \equiv 0 \pmod n$, а следовательно:

$$(b-1)(b+1) \equiv 0 \pmod n.$$

Поскольку $b \not\equiv \pm 1 \pmod n$, то и $(b-1)$, и $(b+1)$ расположены строго между 0 и n, и поэтому простое n не может разделить их произведение. Это создает противоречие, и поэтому такого b не существует, и поэтому строка 13 не может отклонить n.

Если n является нечетным составным числом, то мы говорим, что a является свидетелем («составности») для n, если алгоритм отклоняет на a. Мы показываем, что если n является нечетным составным числом, то, по крайней мере, 50 % a в $\mathbb{Z}_n$ являются свидетелями. Распределение этих свидетелей в $\mathbb{Z}_n$ выглядит очень нерегулярным, но если мы выберем наше a случайно, то мы попадем в свидетеля с вероятностью $\geqslant \frac{1}{2}$.

Поскольку n является составным числом, то либо n является степенью нечетного простого числа, либо n является произведением двух нечетных взаимно простых чисел. В результате получаем два случая.

Случай 1: предположим, что $n = q^e$, где q – это нечетное простое и $e > 1$. Установим $t := 1 + q^{e-1}$. Из биномиального разложения t^n получаем:

$$t^n = (1 + q^{e-1})^n = 1 + nq^{e-1} + \sum_{i=2}^{n} \binom{n}{l} (q^{e-1})^l, \tag{6.2}$$

и поэтому $t^n \equiv 1 \pmod n$. Если $t^{n-1} \equiv 1 \pmod n$, то $t^n \equiv t \pmod n$, что из наблюдения о t и t^n невозможно, следовательно, t является свидетелем в строке 4. Но множество несвидетелей в строке 4, $S_1 := \{a \in \mathbb{Z}_n | a^{(n-1)} \equiv 1 \pmod n\}$, является подгруппой $\mathbb{Z}_n$, и поскольку она не равна $\mathbb{Z}_n$ (t в нем не находится), по теореме Лагранжа S_1 равно не больше половины $\mathbb{Z}_n^*$, и поэтому оно не больше половины $\mathbb{Z}_n$.

Случай 2: предположим, что $n = qr$, где q, r являются взаимно простыми. Среди всех несвидетелей в строке 13 найдем несвидетеля, для которого -1 появляется в наибольшей позиции в последовательности в строке 7 алгоритма (обратите внимание, что -1 является несвидетелем в строке 13, поэтому множество этих несвидетелей не пустое). Пусть x равно такому несвидетелю, и пусть j равно позиции -1 в своей последовательности, где позиции нумеруются, начиная с 0; $x^{s \cdot 2^j} \equiv -1 \pmod n$ и $x^{s \cdot 2^{j+1}} \equiv 1 \pmod n$. Несвидетели в строке 13 являются подмножеством $S_2 := \{a \in \mathbb{Z}_n^* | a^{s \cdot 2^j} \equiv \pm 1 \pmod n\}$, и S_2 является подгруппой $\mathbb{Z}_n^*$.

По китайской теореме об остатках существует $t \in \mathbb{Z}_n$ такое, что

$$\begin{array}{ll} t \equiv x & \pmod q \\ t \equiv 1 & \pmod r \end{array} \quad \Rightarrow \quad \begin{array}{ll} t^{s \cdot 2^j} \equiv -1 & \pmod q \\ t^{s \cdot 2^j} \equiv 1 & \pmod r. \end{array}$$

Следовательно, t является свидетелем, потому что $t^{s \cdot 2^j} \not\equiv \pm 1 \pmod n$, но с другой стороны, $t^{s \cdot 2^{j+1}} \equiv 1 \pmod n$.

Задача 6.13. Покажите, что $t^{s \cdot 2^j} \not\equiv \pm 1 \pmod n$.

Поэтому, как и в случае 1, мы сконструировали $t \in \mathbb{Z}_n^*$, которое не находится в S_2, и поэтому S_2 может быть не больше половины $\mathbb{Z}_n^*$, и, значит, по крайней мере, 50 % элементов в $\mathbb{Z}_n$ являются свидетелями. □

Задача 6.14. Сначала покажите, что множества S_1 и S_2 (в доказательстве теоремы 6.12) действительно являются подгруппами $\mathbb{Z}_n^*$ и что в случае 2 все несвидете-

ли содержатся в S_2. Затем покажите, что, по крайней мере, 50% элементов $\mathbb{Z}_n$ являются свидетелями, когда n является составным, без использования теории групп.

Обратите внимание, что, выполнив алгоритм k раз на независимо выбранном a, мы можем убедиться, что он отклоняет составное число с вероятностью $\geqslant 1 - \frac{1}{2^k}$ (он всегда будет принимать простое число с вероятностью 1). Таким образом, для $k = 100$ вероятность ошибки, то есть ложного утверждения, ничтожна.

6.4. Шифрование с публичным ключом

Шифрование (криптография) имеет широко известные применения в сфере безопасности, например мы можем использовать наши кредитные карты при покупке онлайн, потому что когда мы посылаем наши номера кредитных карт, они шифруются, и даже если они проходят через публичный канал, никто, кроме получателя, не сможет их прочесть. Шифрование также имеет увлекательную историю: от первых применений, зафиксированных Геродотом во время персидских войн V века до нашей эры, до разработок в Блетчли-Парке во время Второй мировой войны – читатель, интересующийся историей криптографии, должен прочитать вот эту увлекательную книгу [Singh (1999)].

Криптосистема с публичным ключом (public key cryptosystem, PKC) состоит из трех множеств: K, множества (из пар) *ключей*, M, множества *простых текстовых сообщений*, и C, множества *зашифрованных текстовых сообщений*. Пара ключей в K равна $k = (k_{\text{priv}}, k_{\text{pub}})$: соответственно, *приватный* (секретный) ключ и *публичный* (открытый) ключ. Для каждого k_{pub} существует соответствующая *шифрующая* функция $e_{k_{\text{pub}}} : M \to C$, и для каждого k_{priv} существует соответствующая *дешифрующая* функция $d_{k_{\text{priv}}} : C \to M$.

Свойство, которому должны удовлетворять *шифрующая* и *дешифрующая* функции, заключается в том, что если $K = (k_{\text{priv}}, k_{\text{pub}}) \in K$, тогда $d_{k_{\text{priv}}}(e_{k_{\text{pub}}}(m)) = m$ для всех $m \in M$. Необходимое допущение состоит в том, что вычислить $d_{k_{\text{priv}}}(c)$, просто зная k_{pub} и c, должно быть трудно. Но с дополнительной *потайной* информацией k_{priv} вычислить $d_{k_{\text{priv}}}(c)$ становится легко.

В следующих разделах мы представляем три разные схемы шифрования: схему Диффи–Хеллмана, которая на самом деле не является схемой шифрования с публичным ключом, а скорее способом согласования секретного ключа по небезопасному каналу, а также схемы Эль-Гамаля и RSA (аббревиатура от фамилий Rivest, Shamir и Adleman). Все три требуют больших простых чисел (на практике длиной около 2000 бит); одно простое число для схем Диффи–Хеллмана и Эль-Гамаля и пара простых чисел для схемы RSA. Но как находить большие простые числа? Ответ, конечно же, будет предусматривать использование алгоритма Рабина–Миллера из предыдущего раздела.

Вот как мы это делаем: по теореме о распределении простых чисел мы знаем, что существует $\pi(n) = n/\log n$ простых чисел $\leqslant n$. Это означает, что среди n-битных целых чисел существует $2^n/n$ простых чисел, примерно 1 в n, и эти простые числа довольно равномерно распределены. Поэтому мы выбираем случайное целое число в заданном диапазоне и применяем к нему алгоритм Рабина–Миллера.

6.4.1. Обмен ключами Диффи–Хеллмана

Если p является простым, то можно показать – хотя доказать это трудно, и мы это здесь опускаем, – что существует $g \in \mathbb{Z}_p^*$ такое, что $\langle g \rangle = \{g^1, g^2, \ldots, g^{p-1}\} = \mathbb{Z}_p^*$. Данное g называется *первообразным корнем* для $\mathbb{Z}_p^*$. Пусть дано $h \in \mathbb{Z}_p^*$, тогда *задача дискретного логарифмирования* является задачей нахождения $x \in \{1, \ldots, p-1\}$ такого, что $g^x \equiv h \pmod{p}$. То есть $x = \log_g(h)$.

Например, $p = 56\,609$ является простым числом, и $g = 2$ является генератором для $\mathbb{Z}_{56\,609}^*$, то есть $\mathbb{Z}_{56\,609}^* = \{2^1, 2^2, 2^3, \ldots, 2^{56\,608}\}$ и $\log_2(38\,679) = 11\,235$.

Задача 6.15. Если $p = 7$, объясните, почему $g = 3$ будет работать как генератор для $\mathbb{Z}_p^*$. Является ли каждое число в $\mathbb{Z}_7^*$ генератором для $\mathbb{Z}_7^*$?

Дискретное логарфимирование считается сложной задачей. Мы намереваемся использовать его, чтобы создать для Алисы и Боба способ договориться о секретном ключе по небезопасному каналу. Сначала Алиса и Боб соглашаются на большое простое число p и целое число $g \in \mathbb{Z}_p^*$. На самом деле g не обязательно должно быть первообразным корнем для p; достаточно и гораздо проще выбрать число g порядка примерно $p/2$. См., например, упражнение 1.31 в книге [Hoffstein и соавт. (2008)]. Числа p, g являются общеизвестными, то есть $k_{\text{pub}} = \langle p, g \rangle$.

Затем Алиса выбирает секрет a и Боб выбирает секрет b. Алиса вычисляет $A := g^a \pmod{p}$, и Боб вычисляет $B := g^b \pmod{p}$. Затем Алиса и Боб обмениваются A и B по небезопасному соединению. С ее стороны Алиса вычисляет $A' := B^a \pmod{p}$, и Боб с его стороны вычисляет $B' := A^b \pmod{p}$. Очевидно, что

$$A' \equiv_p B^a \equiv_p (g^b)^a \equiv_p g^{ab} \equiv_p (g^a)^b \equiv_p A^b \equiv_p B'.$$

Это общее значение $A' = B'$ является их секретным ключом. Таким образом, схема Диффи–Хеллмана на самом деле не является полноценной криптосистемой с публичным ключом; это просто способ для двух сторон договориваться о секретном значении по небезопасному каналу. Также обратите внимание, что вычисление A и B предусматривает вычисление больших степеней g по модулю простого p; если это делается наивно, умножая g на само себя a число раз, то эта процедура непрактична для большого a. Вместо этого мы используем многократное возведение в квадрат; см. стр. 120, где мы обсуждаем эту процедуру.

Задача 6.16. Предположим, что Алиса и Боб соглашаются на $p = 23$ и $g = 5$ и что секрет Алисы $a = 8$, а секрет Боба $b = 15$. Покажите, как работает обмен Диффи–Хеллмана в этом случае. Каким будет результирующий секретный ключ?

Предположим, что Ева подслушивает этот обмен. Из него она может почерпнуть следующую информацию: $\langle p, g, g^a \pmod{p}, g^b \pmod{p} \rangle$. Вычисление из этой информации $g^{ab} \pmod{p}$ (то есть $A' = B'$) называется *задачей Диффи–Хеллмана* (DHP) и считается сложной, когда p является большим простым числом.

Но предположим, что у Евы есть эффективный способ решения задачи Диффи–Хеллмана. Тогда из $g^a \pmod{p}$ она вычисляет a, и из $g^b \pmod{p}$ она вычисляет b, и теперь она может легко вычислить $g^{ab} \pmod{p}$. С другой стороны, неизвестно, дает ли решение задачи Диффи–Хеллмана эффективное решение.

Задача 6.17. Рассмотрите алгоритм Шэнка – алгоритм 6.4. Покажите, что алгоритм Шэнка вычисляет x такое, что $g^x \equiv_p h$ за время $O(n \log n)$, то есть за время $O(\sqrt{p} \log(\sqrt{p}))$.

Задача 6.18. Реализуйте алгоритм 6.4.

Алгоритм 6.4. Малые и гигантские шаги Шэнка

Предусловие: p простое, $\langle g\rangle = \mathbb{Z}_p^*$, $h \in \mathbb{Z}_p^*$

1: $n \leftarrow 1 + \sqrt{p}$
2: $L_1 \leftarrow \{g^0, g^1, g^2, \ldots, g^n\} \pmod p$
3: $L_2 \leftarrow \{hg^0, hg^{-n}, hg^{-2n}, \ldots, hg^{-n^2}\} \pmod p$
4: Найти $g^i \equiv_p hg^{-jn} \in L_1 \cap L_2$
5: $x \leftarrow jn + i$
6: **return** x

Постусловие: $g^x \equiv_p h$

Хотя кажется, что трудно организовать прямую атаку на схему Диффи–Хеллмана, то есть атаковать ее, решив родственную задачу дискретного логарифмирования, существует довольно коварный способ ее взломать, который называется атакой «*человек в середине*» (man-in-the-middle). Он заключается в том, что Ева пользуется отсутствием аутентификации сторон; то есть откуда Боб знает, что он получает сообщение от Алисы, и откуда Алиса знает, что она получает сообщение от Боба? Ева может воспользоваться этим и перехватить предназначенное для Боба сообщение от Алисы и заменить его на $E = g^e \pmod p$, перехватить предназначенное для Алисы сообщение от Боба и также заменить его на $E = g^e \pmod p$, и с того момента читать всю переписку, притворяясь Бобом для Алисы и Алисой для Боба, транслируя сообщение, закодированное с помощью $g^{ae} \pmod p$ в сообщение, закодированное с помощью $g^{be} \pmod p$, и наоборот.

Задача 6.19. Предположим, что $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ является функцией со следующими свойствами:

- для всех $a, b, g \in \mathbb{N}$, $f(g, ab) = f(f(g, a), b) = f(f(g, b), a)$;
- для любого g, $h_g(c) = f(g, c)$ является односторонней функцией, то есть функцией, которую легко вычислить, но обратное которой вычислить трудно[1].

Объясните, как можно использовать f для шифрования с публичным ключом в стиле Диффи–Хеллмана.

6.4.2. Криптосистема Эль-Гамаля

Эта схема является истинной криптографией с публичным ключом, где Алиса и Боб соглашаются о публичных p, g таких, что p – это простое число, и $\mathbb{Z}_p^* = \langle g\rangle$. Алиса также имеет частный a и публикует публичный $A := g^a \pmod p$. Боб хочет отправить сообщение m Алисе, поэтому он создает эфемерный ключ b и отправляет пару c_1, c_2 Алисе, где:

$$c_1 := g^b \pmod p;\ c_2 := mA^b \pmod p.$$

Затем, чтобы прочитать сообщение, Алиса вычисляет:

$$c_1^{-a} c_2 \equiv_p g^{-ab} m g^{ab} \equiv_p m.$$

[1] Существование таких функций является одним из базовых допущений криптографии; дискретный логарифм является примером такой функции, однако доказательство ее существования отсутствует, существует только хорошо обоснованное допущение.

Обратите внимание, что для вычисления c_1^{-a} Алиса сначала вычисляет обратное c_1 в $\mathbb{Z}_p^*$, что она может сделать эффективно, используя *расширенный алгоритм Евклида* (см. алгоритм 1.8 или алгоритм 3.5), а затем вычисляет a-ю степень результата.

Если точнее, то вот как мы вычисляем обратное k в $\mathbb{Z}_n^*$. Наблюдаем, что если $k \in \mathbb{Z}_p^*$, то $\gcd(k, n) = 1$, поэтому, используя алгоритм 1.8, мы получаем s, t такие, что $sk + tn = 1$, и далее s, t могут быть выбраны с целью, чтобы s было в $\mathbb{Z}_n^*$. Для того чтобы это увидеть, сначала получаем любое s, t, а затем добавляем в s соответствующее число положительных или отрицательных кратных n, помещая его во множество $\mathbb{Z}_n^*$, и корректируем t на то же самое число кратных с противоположным знаком.

Задача 6.20. Пусть $p = 7$ и $g = 3$.

1. Пусть $a = 4$ равно секретному ключу Алисы, поэтому

   $A = g^a \pmod p = 3^4 \pmod 7 = 4.$

   Пусть $p = 7, g = 3, A = 4$ равны публичным значениям.
   Предположим, что Боб хочет отправить сообщение $m = 2$ Алисе с эфемерным ключом $b = 5$. Какую соответствующую пару $\langle c_1, c_2 \rangle$ он посылает Алисе? Покажите, какими будут фактические значения и как они вычисляются.
2. Что делает Алиса для прочтения сообщения $\langle 5, 4 \rangle$? То есть как Алиса извлекает m из $\langle c_1, c_2 \rangle = \langle 5, 4 \rangle$?

Задача 6.21. Мы говорим, что можем взломать схему Эль-Гамаля, если у нас есть эффективный способ вычисления m из $\langle p, g, A, c_1, c_2 \rangle$. Покажите, что мы можем взломать схему Эль-Гамаля тогда и только тогда, когда мы можем эффективно решить задачу Диффи–Хеллмана.

Задача 6.22. Напишите приложение, в котором реализуется схема цифровой подписи Эль-Гамаля. Ваша консольная программа должна вызываться следующим образом: `sign 11 6 3 7` – затем примите одну символьную цепочку текста в формате ASCII, пока не появится символ новой строки (то есть пока вы не нажмете клавишу **Enter**). То есть, после того как вы наберете в командной строке `sign 11 6 3 7` и нажмете **Return**, вы наберете сообщение: `'A message'` (сообщение), и после того как вы нажмете **Return** снова, ниже появится цифровая подпись, состоящая из пары положительных целых чисел.

Теперь мы объясним, как получить эту цифровую подпись: сначала преобразуйте символы в символьной цепочке «A message» в соответствующие коды ASCII, а затем получите хеш этих кодов, умножив их все по модулю 11; в результате должно получиться одно число 5.

Для того чтобы это увидеть, посмотрите таблицу:

```
A       65      10
        32      1
m       109     10
e       101     9
s       115     1
s       115     5
a       97      1
g       103     4
```

```
e       101     8
.       46      5
```

Первый столбец содержит символы, второй – соответствующие коды ASCII, и i-я ячейка в третьем столбце содержит произведение первых i кодов по модулю 11. Последняя ячейка в третьем столбце является хеш-значением 5.

Мы подписываем хеш-значение, то есть если сообщение m = `A message`, то мы подписываем hash(m) = 5. Обратите внимание, что мы вызываем функцию `sign` с четырьмя аргументами, то есть мы вызываем ее с p, g, x, k (в нашем рабочем примере соответственно 11, 6, 3, 7).

Здесь p должно быть простым, $1 < g, x, k < p - 1$ и $\gcd(k, p - 1) = 1$. Это условие ввода; в своей программе вам проверять выполнение этого условия не нужно – мы можем допустить, что это так.

Теперь алгоритм подписывает $h(m)$ следующим образом: он вычисляет

$$r = g^k \pmod p$$
$$s = k^{-1}(h(m) - xr) \pmod{(p-1)}.$$

Если s равно нулю, то начните сначала, выбрав другой k (соответствующий требуемым условиям). Подпись m представляет собой именно пару чисел (r, s). В нашем рабочем примере мы имеем следующие значения:

m = `A message`; $h(m) = 5$; $p = 11$; $g = 6$; $x = 3$; $k = 7$,

и поэтому подпись «`A message`» с заданными параметрами будет:

$$\begin{aligned} r &= 6^7 \pmod{11} = 8 \\ s &= 7^{-1}(5 - 3 \cdot 8) \pmod{(11-1)} \\ &= 3 \cdot (-19) \pmod{10} \\ &= 3 \cdot 1 \pmod{10} \\ &= 3, \end{aligned}$$

то есть подпись «`A message`» будет $(r, s) = (8, 3)$.

Задача 6.23. В задаче 6.22:

1) можете ли вы установить (возможные) недостатки этой схемы цифровой подписи? Можете ли вы составить другое сообщение m' такое, что $h(m) = h(m')$?
2) если вы получаете сообщение m и подписную пару (r, s), и вы лишь знаете p, g и $y = g^x \pmod p$, то есть p, g, y являются публичной информацией, как вы можете «верифицировать» подпись – и что подразумевается под верификацией подписи?
3) разыщите в интернете более оптимальный вариант хеш-функции h;
4) покажите, что при использовании без (хорошей) хеш-функции схема подписи Эль-Гамаля экзистенциально поддается подделке; то есть злоумышленница Ева может сконструировать сообщение m и валидную подпись (r, s) для m;
5) на практике k представляет собой случайное число; покажите, что абсолютно необходимо выбирать новое случайное число для каждого сообщения;
6) покажите, что при верификации подписи необходимо проверять, соблюдается ли $1 \leqslant r \leqslant p - 1$, потому что в противном случае Ева будет способна

подписывать сообщение по своему выбору, при условии что она знает одну валидную подпись (r, s) для некого сообщения m, где m такое, что $1 \leqslant m \leqslant p-1$ и $\gcd(m, p-1) = 1$.

6.4.3. Криптосистема RSA

Выберем два нечетных простых числа p, q и установим $n = pq$. Выберем $k \in \mathbb{Z}^*_{\varphi(n)}$, $k > 1$. Объявим f, где $f(m) \equiv m^k \pmod n$. Вычислим l, обратное k в $\mathbb{Z}^*_{\varphi(n)}$. Теперь (n, k) является публичным, и ключ l является секретным, а также и функция g, где $g(C) \equiv C^l \pmod n$. Обратите внимание, что $g(f(m)) \equiv_n m^{kl} \equiv_n m$.

Задача 6.24. Покажите, что $m^{kl} \equiv m \pmod n$. На самом деле, для того чтобы это соблюдалось, существует неявное допущение о m; что это за допущение?

Задача 6.25. Подтвердите наблюдение, что мы могли бы взломать RSA, если бы разложение на составляющие было бы легким.

Теперь мы сделаем два наблюдения о безопасности RSA. Во-первых, простые числа p, q не могут быть выбраны «близко» друг к другу. Для того чтобы понять, что мы имеем в виду, обратите внимание, что

$$n = \left(\frac{p+q}{2}\right)^2 - \left(\frac{p-q}{2}\right)^2.$$

Поскольку p, q близки, мы знаем, что $s := (p-q)/2$ является малым, и $t := (p+q)/2$ лишь слегка больше $\sqrt{n}$, и $t^2 - n = s^2$ представляет собой идеальный квадрат. Поэтому мы пробуем следующие кандидатные значения для t:

$$\lceil\sqrt{n}\rceil + 0, \lceil\sqrt{n}\rceil + 1, \lceil\sqrt{n}\rceil + 2, \ldots$$

До тех пор, пока $t^2 - n$ не будет идеальным квадратом s^2. Ясно, что если s является малым, то мы быстро найдем такое t, и тогда $p = t + s$ и $q = t - s$.

Второе наблюдение состоит в том, что если бы мы могли взломать RSA путем эффективного вычисления l из n и k, то мы могли бы разложить n за рандомизированное полиномиальное время. Поскольку $\varphi(n) = \varphi(pq) = (p-1)(q-1)$, из этого следует, что:

$$\begin{aligned} p + q &= n - \varphi(n) + 1 \\ pq &= n, \end{aligned} \tag{6.3}$$

и из этих двух уравнений мы получаем:

$$(x-p)(x-q) = x^2 - (p+q)x + pq = x^2 - (n - \varphi(n) + 1)x + n.$$

Таким образом, мы можем вычислить p, q, вычислив корни этого последнего многочлена. Используя классическую формулу квадратного уравнения $x = (-b \pm \sqrt{b^2 - 4ac})/2a$, мы получаем, что p, q равняются:

$$\frac{(n - \varphi(n) + 1) \pm \sqrt{(n - \varphi(n) + 1)^2 - 4n}}{2}.$$

Предположим, что Ева способна вычислить l из n и k. Если Ева знает l, тогда она знает, что каким бы ни была $\varphi(n)$, она делит $kl - 1$ и, значит, Ева имеет уравнения (6.3) с той разницей, что $\varphi(n)$ заменена на $(kl-1)/a$ для некоторого a. Это a может

быть вычислено за рандомизированное полиномиальное время, но мы его здесь не приводим. Тем самым следует утверждение.

Если Ева способна факторизовать, то она, очевидно, способна взломать RSA; с другой стороны, если Ева способна взломать RSA – путем вычисления l из n, k, – то, значит, она способна факторизовать за рандомизированное полиномиальное время.

С другой стороны, Ева может быть способна взломать RSA без вычисления l, поэтому из предыдущих наблюдений не вытекает, что взлом RSA представляет такую же трудность, что и факторизация.

6.5. Дальнейшие задачи

В криптографии существует определенный разворот приоритетов в том, что сложная задача становится союзником, а не препятствием. На стр. 75 мы упомянули NP-трудные задачи, то есть задачи, для которых нет допустимых решений, когда экземпляры «достаточно велики».

Простая задача о рюкзаке (см. раздел 4.3) является одной из таких задач, и мы можем использовать ее для определения криптосистемы. *Криптосистема Меркла–Хеллмана* сумм подмножеств основана на задаче о рюкзаке и работает следующим образом. Сначала Алиса создает секретный ключ, состоящий из следующих элементов:

- сверхвозрастающая последовательность: $\mathbf{r} = (r_1, r_2, \ldots, r_n)$, где $r_i \in \mathbb{N}$, и свойство быть «сверхвозврастающим» относится к $2r_i \leqslant r_{i+1}$, для всех $1 \leqslant i < n$;
- пара положительных целых чисел A, B с двумя условиями: $2r_n < B$ и $\gcd(A, B) = 1$.

Публичный ключ состоит из $\mathbf{M} = (M_1, M_2, \ldots, M_n)$, где $M_i = Ar_i \pmod{B}$.

Предположим, что Боб хочет отправить текстовое сообщение $x \in \{0, 1\}^n$, то есть x является двоичной строкой длины n. Тогда он использует открытый ключ Алисы для вычисления $S = \sum_{i=1}^{n} x_i M_i$, где x_i – это i-й бит x, интерпретируемый как целое число 0 или 1. Боб теперь посылает S Алисе.

Для того чтобы прочитать сообщение, Алиса вычисляет $S' = A^{-1}S \pmod{B}$, и она решает задачу о сумме подмножеств с использованием сверхвозврастающего $\mathbf{r}$. Задача о сумме подмножеств для общей последовательности $\mathbf{r}$ является очень сложной, но когда $\mathbf{r}$ является сверхвозврастающим (отметим, что $\mathbf{M}$ принимается как несверхвозврастающая!), эту задачу можно решить простым жадным алгоритмом.

Точнее, Алиса находит подмножество $\mathbf{r}$, сумма которого в точности равна S'. Любое подмножество $\mathbf{r}$ можно отождествить с двоичной строкой длины n, исходя из допущения, что x_i равно 1 тогда и только тогда, когда r_i находится в этом подмножестве. Следовательно, Алиса «извлекает» x из S'.

Например, пусть $\mathbf{r} = (3, 11, 24, 50, 115)$ и $A = 113$, $B = 250$. Проверим, что все условия выполнены, и проверим, что $\mathbf{M} = (89, 243, 212, 150, 245)$. Для того чтобы отправить секретное сообщение $x = 10101$, мы вычисляем

$$S = 1 \cdot 89 + 0 \cdot 243 + 1 \cdot 212 + 0 \cdot 150 + 1 \cdot 245 = 546.$$

Получив S, мы умножаем его на 177, обратное от 113 в mod 250, и получаем 142.

Теперь x может быть извлечено из 142 с помощью простого жадного алгоритма.

Задача 6.26. Две части:

1) покажите, что если $\mathbf{r} = (r_1, r_2, \ldots, r_n)$ является сверхвозрастающей последовательностью, то $r_{i+1} > \sum_{j=1}^{i} r_j$ для всех $1 \leqslant i < n$;
2) предположим, что $\mathbf{r} = (r_1, r_2, \ldots, r_n)$ является сверхвозрастающей последовательностью и что существует подмножество $\mathbf{r}$, сумма которого равна S. Предоставьте (естественный) жадный алгоритм вычисления этого подмножества и покажите, что ваш алгоритм является правильным.

Задача 6.27. Реализуйте криптосистему Меркла–Хеллмана сумм подмножеств. Вызовите программу `sscrypt`, и она должна работать с тремя переключателями: `-e -d -v` для шифрования, дешифрования и верификации. То есть

`sscrypt -e` $M_1\, M_2 \ldots M_n\, x$

шифрует символьную цепочку $x = x_1 x_2 \ldots x_n \in \{0, 1\}^n$ с открытым ключом, задаваемым $\mathbf{M} = (M_1, M_2, \ldots, M_n)$, и выдает на выходе S. С другой стороны,

`sscrypt -d` $r_1\, r_2 \ldots r_n\, A\, B\, S$

расшифровывает символьную цепочку $x = x_1 x_2 \ldots x_n \in \{0, 1\}^n$ из S с использованием секретного ключа, задаваемого $\mathbf{r} = (r_1, r_2, \ldots, r_n)$ и A, B; то есть на выходе она выдает x при $\mathbf{r}$, A, B, S на входе. Наконец,

`sscrypt -v` $r_1\, r_2 \ldots r_n\, A\, B$

проверяет, что $\mathbf{r} = (r_1, r_2, \ldots, r_n)$ является сверхвозрастающей, она проверяет, что $2r_n < B$ и что $\gcd(A, B) = 1$, и на выходе выдает соответствующий публичный ключ, задаваемый $\mathbf{M} = (M_1, M_2, \ldots, M_n)$.

6.6. Ответы к избранным задачам

Задача 6.5. Мы используем алгоритм 6.1, для того чтобы найти идеальное паросочетание (если таковое имеется) следующим образом: выбираем $1 \in V$ и по очереди рассматриваем каждое $(1, i') \in E$, удаляем его из G, получая $G_{1,i'} = ((V - \{1\}) \cup (V' - \{i'\}), E_{1,i'})$, где $E_{1,i'}$ состоит из всех ребер E, за исключением смежных 1 или i', до тех пор, пока для некоторой $i' \in V'$ мы не получим $G_{1,i'}$, для которых алгоритм дает ответ «да». Тогда мы знаем, что существует идеальное паросочетание, которое сочетает 1 и i'. Продолжаем с $G_{1,i'}$.

Задача 6.6. $M(x)$ хорошо определено, потому что матричное умножение ассоциативно. Теперь мы покажем, что из $M(x) = M(y)$ следует $x = y$ (то есть отображение M является взаимно-однозначным). При условии что $M = M(x)$, мы можем «расшифровать» x уникально следующим образом: если первый столбец M больше второго (где сравнение производится покомпонентно), то последний бит x равен нулю, и в противном случае он равен 1. Принимаем M' равным M, где мы вычитаем меньший столбец из большего и повторяем.

Задача 6.7. Для заданной символьной цепочки x, $M(x_1 x_2 \ldots x_n)$ является такой, что «меньший» столбец ограничен числом f_{n-1} и «больший» столбец ограничен f_n. Мы можем показать это индукционно: базовый случай, $x = x_i$, очевиден. В качестве индукционного шага мы принимаем допущение, что он соблюдается для $x \in \{0, 1\}^n$,

и покажем, что он по-прежнему соблюдается для $x \in \{0, 1\}^{n+1}$: это ясно, так как независимо от того, равен ли x_{n+1} значению 0 либо 1, один столбец добавляется в другой, а другой столбец остается неизменным.

Задача 6.9. С учетом того, что p является составным и не Кармайклом, существует, по крайней мере, одно $a \in \mathbb{Z}_p^*$ такое, что $a^{(p-1)} \not\equiv 1 \pmod p$ и $\gcd(p, a) = 1$. Пусть $B = \{b_1, b_2, \ldots\}$ равно множеству несвидетелей. Умножим каждый элемент B на a, получив свидетелей $\{a_1, a_2, \ldots\}$. Каждый из этих свидетелей уникален, так как $a \in \mathbb{Z}_p^*$, поэтому свидетелей, по крайней мере, столько же, сколько и несвидетелей.

Пусть b равно потенциальному несвидетелю; то есть b равняется любому элементу $\mathbb{Z}_p^*$ такому, что $\gcd(p, b) = 1$. Если мы умножим a на любого *лжеца Ферма* (то есть несвидетеля), то мы получим свидетеля. Если существует только один несвидетель, то доказательство завершено. В противном случае пусть b_1, b_2 равны двум несвидетелям. Мы знаем, что $\gcd(p, b_1) = \gcd(p, b_2) = 1$, так как в противном случае b_1 и b_2 были бы свидетелями. Допустим, $ab_1 = ab_2 + kp$, и пусть $g = \gcd(a, p)$.

Задача 6.10. Предположим, что $\gcd(a, p) \neq 1$. По пропозиции 9.20 мы знаем, что если $\gcd(a, p) \neq 1$, то a не имеет (мультипликативного) обратного в $\mathbb{Z}_p$. Таким образом, невозможно, чтобы $a^{(p-1)} \equiv 1 \pmod p$ было истинным, так как из этого следует, что $a \cdot a^{(p-2)} \equiv 1 \pmod p$, и, следовательно, a будет иметь (мультипликативное) обратное.

Задача 6.13. Для того чтобы понять, почему $t^{s \cdot 2^j} \not\equiv \pm 1 \pmod n$, сделаем следующее наблюдение: предположим, что $a \equiv -1 \pmod q$ и $a \equiv 1 \pmod r$, где $\gcd(q, r) = 1$. Предположим, что $n = qr | (a+1)$, тогда $q | (a+1)$ и $r | (a+1)$, и поскольку также $r | (a-1)$, из этого следует, что $r | [(a+1) - (a-1)]$, поэтому $r | 2$, и поэтому $r = 2$, а значит, n должно быть четным, что невозможно, поскольку мы имеем дело с четными n в строке 1 алгоритма.

Задача 6.14. Показать, что S_1 и S_2 являются подгруппами $\mathbb{Z}_n^*$, достаточно легко; в обоих случаях очевидно, что там имеется 1, и замыкание и существование обратного можно легко проверить.

Для того чтобы дать то же самое доказательство без теории групп, мы следуем случаям в доказательстве теоремы 6.12. Пусть t равно свидетелю, сконструированному в случае 1. Если d является (этап 3) несвидетелем, то мы имеем $d^{p-1} \equiv 1 \pmod p$, но тогда $dt \pmod p$ является свидетелем. Более того, если d_1, d_2 являются несовпадающими (этап 3) несвидетелями, то $d_1 t \not\equiv d_2 t \pmod p$. В противном случае $d_1 \equiv_p d_1 \cdot t \cdot t^{p-1} \equiv_p d_2 \cdot t \cdot t^{p-1} \equiv_p d_2$. Таким образом, число (этап 3) свидетелей должно быть, по крайней мере, таким же большим, как и число несвидетелей.

Мы делаем то же самое для случая 2; пусть d равно несвидетелю. Во-первых, $d^{s \cdot 2^j} \equiv \pm 1 \pmod p$ и $d^{s \cdot 2^{j+1}} \equiv 1 \pmod p$ вследствие того, как было выбрано j. Поэтому $dt \pmod p$ является свидетелем, потому что $(dt)^{s \cdot 2^j} \not\equiv \pm 1 \pmod p$ и $(dt)^{s \cdot 2^{j+1}} \equiv 1 \pmod p$.

Во-вторых, если d_1 и d_2 являются несовпадающими несвидетелями, то $d_1 t \not\equiv d_2 t \pmod p$. Причина в том, что $t^{s \cdot 2^{j+1}} \equiv 1 \pmod p$. Следовательно, $t \cdot t^{s \cdot 2^{j+1}-1} \equiv 1 \pmod p$. Поэтому если $d_1 t \equiv d_2 t \pmod p$, то $d_1 \equiv_p d_1 t \cdot t^{s \cdot 2^{j+1}-1} \equiv_p d_2 t \cdot t^{s \cdot 2^{j+1}-1} \equiv_p d_2$. Таким образом, в случае 2 то же самое число свидетелей должно быть, по крайней мере, таким же большим, как и число несвидетелей.

Задача 6.15. $3^1 = 3$, $3^2 = 9 = 2$, $3^3 = 2 \cdot 3 = 6$, $3^4 = 6 \cdot 3 = 4$, $3^5 = 4 \cdot 3 = 5$, $3^6 = 5 \cdot 3 = 1$, все вычисления $\pmod 7$, и, следовательно, $g = 3$ генерирует все числа в $\mathbb{Z}_7^*$. Не каждое число является генератором: например, 4 таковым не является.

Задача 6.16. Алиса и Боб соглашаются использовать простое число $p = 23$ и ос-

нование $g = 5$. Алиса выбирает секрет $a = 8$; посылает Бобу $A = g^a \pmod p$.

$A = 5^8 \pmod{23} = 16$.

Боб выбирает секрет $b = 15$; отправляет Алисе $B = g^b \pmod p$.

$B = 5^{15} \pmod{23} = 19$.

Алиса вычисляет $s = B^a \pmod p$.

$s = 19^8 \pmod{23} = 9$.

Боб вычисляет $s = A^b \pmod p$.

$s = 16^{15} \pmod{23} = 9$.

Как можно видеть, оба в конечном итоге имеют $s = 9$, их общий секретный ключ.

Задача 6.19. Предположим, что мы имеем одностороннюю функцию, как в вопросе. Сначала Алиса и Боб договариваются о публичном g и обмениваются им (поэтому злоумышленник знает g). Тогда пусть Алиса генерирует секрет a и пусть Боб генерирует секрет b. Алиса посылает $f(g, a)$ Бобу, и Боб посылает $f(g, b)$ Алисе. Обратите внимание, что поскольку h_g является односторонней, злоумышленник не может получить a или b из $h_g(a) = f(g, a)$ и $h_g(b) = f(g, b)$. Наконец, Алиса вычисляет $f(f(g, b), a)$, и Боб вычисляет $f(f(g, a), b)$, и по свойствам функции оба равны $f(g, ab)$, что является их секретным общим ключом. Злоумышленник не может правдоподобно вычислить $f(g, ab)$.

Задача 6.20. Для первой части:

$c_1 = g^b \pmod p = 3^5 \pmod 7 = 5$;
$c_2 = mA^b \pmod p = 2 \cdot 4^5 \pmod 7 = 2 \cdot 2 = 4$.

Для второй части:

$$\begin{aligned} m &= c_1^{-a} c_2 \pmod p \\ &= 5^{-4}4 \pmod 7 \\ &= (5^{-1})^4 4 \pmod 7 \\ &= 3^4 4 \pmod 7 \\ &= 4 \cdot 4 \pmod 7 \\ &= 2. \end{aligned}$$

Задача 6.21. Задача Диффи–Хеллмана с $\langle p, g, A \equiv_p g^a, B \equiv_p g^b \rangle$ на входе выдает $g^{ab} \pmod p$ на выходе, и задача Эль-Гамаля с

$$\langle p, g, A \equiv_p g^a, c_1 \equiv_p g^b, c_2 \equiv_p mA^b \rangle \tag{6.4}$$

на входе выдает m на выходе. Мы хотим показать, что мы можем взломать схему Диффи–Хеллмана, то есть эффективно решить задачу Диффи–Хеллмана тогда и только тогда, когда сможем взломать схему Эль-Гамаля, то есть эффективно решить задачу Эль-Гамаля. Ключевое слово здесь – *эффективно*, то есть за полиномиальное время.

($\Rightarrow$) Предположим, что мы можем эффективно решить задачу Диффи–Хеллмана; мы даем эффективную процедуру решения задачи Эль-Гамаля: с учетом входа (6.4) в задачу Эль-Гамаля мы получаем $g^{ab} \pmod p$ из $A \equiv_p g^a$ и $c_1 \equiv g^b$, используя эффективный решатель для задачи Диффи–Хеллмана. Затем мы используем рас-

ширенный евклидов алгоритм, см. задачу 1.9 – и обратите внимание, что расширенный алгоритм Евклида работает за полиномиальное время, для того чтобы получить $(g^{ab})^{-1} \pmod p$. Теперь

$$c_2 \cdot (g^{ab})^{-1} \equiv_p mg^{ab}(g^{ab})^{-1} \equiv_p m = m,$$

где последнее равенство следует из $m \in \mathbb{Z}_p$.

($\Leftarrow$) Предположим, что у нас есть эффективный решатель для задачи Эль-Гамаля. Для того чтобы решить задачу Диффи–Хеллмана, мы конструируем следующий вход в задачу Эль-Гамаля:

$$\langle p, g, A \equiv_p g^a, c_1 \equiv_p g^b, c_2 = 1\rangle.$$

Обратите внимание, что $c_2 = 1 \equiv_p \underbrace{(g^{ab})^{-1}A^b}_{=m}$, поэтому, используя эффективный решатель для задачи Эль-Гамаля, мы получаем $m \equiv_p (g^{ab})^{-1}$, и теперь, используя расширенный алгоритм Евклида, мы получаем обратное от $(g^{ab})^{-1} \pmod p$, то есть просто $g^{ab} \pmod p$, и выдаем его на выходе.

Задача 6.23.

1. Слабость нашей схемы лежит в хеш-функции, которая вычисляет одинаковые хеш-значения для разных сообщений, и на самом деле легко найти сообщения с одинаковым значением хеш-функции – например, добавив пары букв (в любой части сообщения) такие, что соответствующие им значения ASCII являются взаимно обратными по модулю p.

   Приведем пару примеров (заданий) сообщений с одинаковым хеш-значением: «A mess» и «L message». В общем случае по своей природе любая хеш-функция будет иметь такие *коллизии*, то есть сообщения, что:

   $$h(\text{A message.}) = h(\text{A mess}) = h(\text{L message}) = 5,$$

   но есть хеш-функции, коллизионно-устойчивые в том смысле, что вычислительно трудно найти два сообщения m, m' такие, что $h(m) = h(m')$. Хорошая хеш-функция также является односторонней функцией в том смысле, что при наличии значения y вычислительно трудно найти m такое, что $h(m) = y$.
2. Верифицирование подписи означает проверку того, что документ m подписало именно то лицо, которое владеет x. Здесь имеются две тонкости: сначала мы говорим «лицо, которое владеет x», а не «законный владелец x», просто потому, что x может быть скомпрометирован (например, украден). Во-вторых, и именно по этой причине данная схема является такой гениальной, мы можем проверить, что «некто, владеющий x», подписал сообщение, *даже не зная, что такое* x! Мы знаем y, где $y = g^x \pmod p$, но для больших p трудно вычислить x из y (это называется задачей дискретного логарифмирования).

   Ниже мы покажем, как верифицировать, что «некто, владеющий x», подписал сообщение m. Мы проверяем $0 < r < p$ и $0 < s < p - 1$ (см. вопрос 6), вычисляем $v := g^{h(m)} \pmod p$ и $w := y^r r^s \pmod p$; g, p являются публичными, m известно, функция $h : \mathbb{N} \to [p-1]$ тоже известна, и r, s являются заданной подписью. Если v и w совпадают, то подпись валидна.

   Для того чтобы убедиться, что все работает, обратите внимание, что мы опре-

делили $s := k^{-1}(h(m) - xr) \pmod{p-1}$. Следовательно, $h(m) = xr + sk \pmod{p-1}$. Теперь малая теорема Ферма говорит, что $g^{p-1} = 1 \pmod{p}$, и следовательно:

$$g^{h(m)} \overset{(*)}{=} g^{xr+sh} = (g^x)^r (g^k)^s = y^r r^s \pmod{p}.$$

Малая теорема Ферма применяется в равенстве $(*)$: поскольку $h(m) = xr + sk \pmod{p-1}$, из этого следует, что $(p-1) | (h(m) - (xr + sk))$, а это означает, что $(p-1)z = h(m) - (xr + sk)$ для некоторого z, и так как $g^{(p-1)z} = (g^{(p-1)})^z = 1^z = 1 \pmod{p}$, отсюда следует, что $g^{h(m)-(xr+sk)} = 1 \pmod{p}$, и поэтому $g^{h(m)} = g^{xr+sk} \pmod{p}$.

3. Приведем несколько хеш-функций, реализованных в `GPG`, версии 2.0.30[1]: `MD5`, `SHA1`, `RIPEMD160`, `SHA256`, `SHA384`, `SHA512`, `SHA224`.
4. Для того чтобы это увидеть, пусть b, c равны числам таким, что $\gcd(c, p-1) = 1$. Зададим $r = g^b y^c$, $s = -rc^{-1} \pmod{p-1}$ и $m = -rbc^{-1} \pmod{p-1}$. Тогда (m, r, s) удовлетворяет $g^m = y^r r^s$. Так как на практике хеш-функция h применяется к сообщению и в реальности подписывается именно хеш-значение, подделать подпись для получения содержательного сообщения не так просто. Злоумышленник должен найти содержательное сообщение $\tilde{m}$ такое, что $h(\tilde{m}) = h(m)$, и когда h является коллизионно-устойчивой, это очень трудно сделать.
5. Если одинаковое случайное число k используется в двух разных сообщениях $m \neq m'$, то k можно вычислить следующим образом: $s - s' = (m - m')^{k-1} \pmod{p-1}$, и следовательно, $k = (s - s')^{-1}(m - m') \pmod{p-1}$.
6. Пусть m' равно сообщению по выбору Евы, $u = m'm^{-1} \pmod{p-1}$, $s' = su \pmod{p-1}$, r' и целое число такие, что $r' = r \pmod{p}$ и $r' = ru \pmod{p-1}$. Это r' может быть получено так называемой китайской теоремой об остатках (см. теорему 9.30). Затем (m', r', s') принимается верификационной процедурой.

Задача 6.24. Почему $m^{kl} \equiv_n m$? Пронаблюдаем, что $kl = 1 + (-t)\varphi(n)$, где $(-t) > 0$, и поэтому $m^{kl} \equiv_n m^{1+(-t)\varphi(n)} \equiv_n m \cdot (m^{\varphi(n)})^{(-t)} \equiv_n m$, потому что $m^{\varphi(n)} \equiv_n 1$. Обратите внимание, что это последнее формальное суждение не следует непосредственно из теоремы Эйлера (теорема 9.29), потому что $m \in \mathbb{Z}_n$, и не обязательно в $\mathbb{Z}_n^*$. Обратите внимание, что, для того чтобы убедиться, что $m \in \mathbb{Z}_n^*$, достаточно настоять на том, что у нас есть $0 < m < \min\{p, q\}$; поэтому мы разбиваем большое сообщение на мелкие части.

Интересно отметить, что мы можем обойти теорему Эйлера и просто применить малую теорему Ферма: мы знаем, что $m^{(p-1)} \equiv_p 1$ и $m^{(q-1)} \equiv_q 1$, поэтому $m^{(p-1)(q-1)} \equiv_p 1$ и $m^{(q-1)(p-1)} \equiv_q 1$, следовательно, $m^{\varphi(n)} \equiv_p 1$ и $m^{\varphi(n)} \equiv_q 1$. Это означает, что $p | (m^{\varphi(n)} - 1)$ и $q | (m^{\varphi(n)} - 1)$, поэтому, поскольку p, q являются несовпадающими простыми числами, из этого следует, что $(pq) | (m^{\varphi(n)} - 1)$, и поэтому $m^{\varphi(n)} \equiv_n 1$.

Задача 6.25. Если бы факторизовать целые числа было легко, то схему RSA было бы легко взломать: если бы мы смогли факторизовать n, то мы получили бы все простые числа p, q и, следовательно, было бы легко вычислить $\varphi(n) = \varphi(pq) = (p-1)(q-1)$, и из этого мы получаем l, инверсию k.

Задача 6.26. Мы показываем, что для $\forall i \in [n-1]$ имеет место, что $r_{i+1} \sum_{j=1}^{i} r_j$ по индукции на i. Базовый случай равен $i - 1$, поэтому

[1] GNU Privacy Guard (GPG) – свободная программа для шифрования информации и создания электронных цифровых подписей. Последняя на конец 2018 г. версия была 2.2.12. – *Прим. перев.*

$$r_2 \geqslant 2r_1 > r_1 = \sum_{j=1}^{i} r_j,$$

где $r_2 \geqslant 2r_1$ по свойству сверхвозрастания. На индукционном шаге мы имеем

$$r_{i+1} \geqslant 2r_i = r_i + r_i > r_i + \sum_{j=1}^{i=1} r_j = \sum_{j=1}^{i} r_j,$$

где мы использовали свойство сверхвозрастания и индукционную гипотезу.

Приведем алгоритм для второго вопроса:

```
X ← S
Y ← ∅
for i = n...1 do
        if (r_i ⩽ X) then
                X ← X − r_i
                Y ← Y ∪ {i}
        end if
end for
```

и пусть предусловие констатирует, что $\{r_i\}_{i=1}^{n}$ является сверхвозрастающим и что существует $\mathcal{S} \subseteq \{r_i\}_{i=1}^{n}$ такое, что $\sum_{i \in \mathcal{S}} r_i = S$. Пусть постусловие констатирует, что $\sum_{i \in Y} r_i = S$.

Определим следующий инвариант цикла: «Y является перспективным» в том смысле, что оно может быть расширено, причем индексы весов еще не были рассмотрены, в решение. То есть после рассмотрения i существует подмножество E из $\{i-1, \ldots, 1\}$ такое, что $\sum_{j \in X \cup E} r_j = S$.

Базовый случай является тривиальным, так как изначально $X = \emptyset$, и поскольку предусловие гарантирует существование решения, X может быть расширено в это решение.

На индукционном шаге рассмотрим два случая. Если $r_i > X$, то i не добавляется, но Y может быть расширено с помощью $E' \subseteq \{i-1, i-2, \ldots, 1\}$. Причина в том, что по индукционной гипотезе X было расширено в решение некоторым $E \subseteq \{i, i-1, \ldots, 1\}$ и i не было частью расширения, поскольку r_i было слишком большим, чтобы уложиться в то, что уже было в Y, то есть $E' = E$.

Если $r_i \leqslant X$, то $i \in E$, так как по предыдущей части оставшиеся веса не смогут закрыть разрыв между S и $\sum_{j \in Y} r_j$.

6.7. Примечания

Говоря об эпиграфе в начале главы, роман «Энигма» [Harris (1996)] является отличным введением в ранние годы криптоанализа; кроме того, у данного романа есть отличная экранизация 2001 года.

Хотя в этой книге мы не обсуждали задачу минимального-максимального потока, в большинстве вводных пособий в алгоритмы данная тема рассматривается. См., например, главу 7 в книге [Kleinberg и Tardos (2006)]. Кроме того, в работе [Fernandez и Soltys (2013)] обсуждается минимаксный принцип и связывается с несколькими другими фундаментальными принципами комбинаторики.

Генерировать случайные числа очень трудно; см., например, главу 7 в книге [Press и соавт. (2007)].

Алгоритм 6.3, алгоритм Рабина–Миллера, часто встречаемый под аббревиатурой RM, реализован в OpenSSL, инструментальной бибилотеке для протоколов безопасности транспортного уровня (transport layer security, TLS) и уровня защищенных сокетов (secure sockets layer, SSL). Он также представляет собой универсальную криптографическую библиотеку.

Очень большие числа можно проверить на простоту с помощью команды:

```
openssl prime <число>
```

Более новые версии инструментальной библиотеки OpenSSL также могут генерировать простое число из заданного количества бит:

```
openssl prime -generate -bits 2048
```

Также обратите внимание, что можно легко вычислять большие степени числа по модулю простого числа с помощью Python, просто исполните команду:

```
>>> pow(x,y,z)
```

которая возвращает $x^y \pmod z$.

Раздел 6.3 по алгоритму Рабина–Миллера был написан в то время, когда автор проводил академический год в Университете Колорадо в Боулдере, 2007–2008 гг., и этот раздел был значительно улучшен, по сравнению с обсуждениями с Яном Мициельским.

Заслуга в изобретении метода Монте-Карло часто принадлежит Станиславу Уламу, математику польского происхождения, который работал в США с Джоном фон Нейманом над Манхэттенским проектом во время Второй мировой войны. Улам также известен разработкой водородной бомбы с Эдвардом Теллером в 1951 году. Он изобрел метод Монте-Карло в 1946 году, размышляя о вероятности выигрыша в карточной игре пасьянс.

Раздел 6.2 основан на работе [Karp и Rabin (1987)].

Первый полиномиально-временной алгоритм для тестирования простоты (примарности) был описан в работе [Agrawal и соавт. (2004)]. Этот алгоритм известен как «AKS Primality Test» (по фамилиям изобретателей: Agrawal–Kayal–Saxena). Однако алгоритм AKS не выполним; алгоритм Рабина–Миллера по-прежнему является стандартом для тестирования простоты.

На самом деле именно рандомизированная проверка простоты вызвала интерес к рандомизированным вычислениям в конце 1970-х годов. Исторически первый рандомизированный алгоритм для примарности был описан в [Solovay и Strassen (1977)]; хорошее описание этого алгоритма со всеми необходимыми общими сведениями можно найти в § 11.1 публикации [Papadimitriou (1994)], и еще одно в § 18.5 публикации [von zur Gathen и Gerhard (1999)].

Р. Д. Кармайкл впервые отметил существование чисел Кармайкла в 1910 году, вычислил пятнадцать примеров и предположил, что хотя они нечасты, их бесконечно много. В 1956 году Эрдос набросал технику построения больших чисел Кармайкла ([Hoffman (1998)]), и доказательство было дано в работе [Alford и соавт. (1994)] в 1994 году.

Первые три числа Кармайкла равны 561, 1105, 1729, где последнее приведен-

ное в этом списке число называется числом Харди–Рамануджана по известному рассказу британского математика Г. Х. Харди о посещении индийского математика Шринивасой Рамануджаном в больнице. Харди писал: «Я помню, как однажды ходил к нему, когда он приболел в Патни. Я ехал в такси с номером 1729 и заметил, что этот номер показался мне довольно скучным, и я надеялся, что это не будет неблагоприятным предзнаменованием. – Нет, – ответил он, – это очень интересный номер; это наименьшее число, выражаемое как сумма двух кубов двумя разными способами». Читателю предлагается посмотреть фильм «Человек, который познал бесконечность» (The Man Who Knew Infinity) 2015 года о Шринивасе Рамануджане.

Раздел 6.4 основан на материалах [Hoffstein и соавт. (2008)] и [Delfs и Knebl (2007)].

Схема RSA названа по фамилии своих изобретателей: Rivest–Shamir–Adleman.

GnuPG, или GPG, является свободной реализацией стандарта OpenPGP, определенного рабочим предложением RFC4880 (также известным как PGP). GPG позволяет шифровать и подписывать данные и коммуникации, а также имеет полную систему управления ключами. Вот еще несколько примеров использования GPG:

```
gpg --gen-keys
gpg --list-keys
gpg --armor -r 9B070A58 -e example.txt
gpg --armor --clearsign example.txt
gpg --verify example.txt.asc
```

Первая команда генерирует новую пару публичного и секретного ключей. Вторая выводит список всех ключей в связке ключей и отображает сводку о каждом. Третья команда шифрует текстовый файл `example.txt` с публичным ключом с идентификатором 9B070A58, который является ключом автора[1]. Четвертая команда выводит подпись `example.txt`, которая гарантирует, что файл не был модифицирован; подпись прилагается в качестве текста к файлу. Последняя команда верифицирует подпись, полученную в результате выполнения предыдущей команды.

Публичные ключи можно рекламировать на личных домашних страницах или загружать в инфраструктуру открытых ключей (public key infrastructure, PKI). Примером инфраструктуры открытых ключей является сервер публичных ключей MIT PGP, https://pgp.mit.edu, в котором можно выполнять поиск ключей (по идентификаторам, именам, электронным адресам и т. д.):

```
gpg --keyserver hkp://pgp.mit.edu --search-keys 0x9B070A58
```

Обратите внимание, что URL-адрес сервера ключей задается с протоколом HKP, где HKP означает «протокол HTTP сервера ключей OpenPGP» (OpenPGP HTTP Keyserver Protocol).

Подобные операции могут быть выполнены с помощью OpenSSL; например, мы можем генерировать секретные ключи RSA следующим образом:

```
openssl genrsa -out mysecretrsakey.pem 512
```

---

[1] Открытый ключ GPG автора с идентификатором 9B070A58: http://www.msoltys.com/gpg-key.

```
openssl genrsa -out mysecretrsakey.pem 4096
```

Два параметра 512 и 4096 задают размер простых чисел; обратите внимание, что при 4096 генерирование продолжается немного дольше; именно здесь используется алгоритм Рабина–Миллера. Следующий далее параметр создает соответствующий публичный ключ:

```
openssl rsa -in mysecretrsakey.pem -pubout
```

Мы можем генерировать ключ на основе эллиптических кривых:

```
openssl ecparam -out myeckey.pem -name prime256v1 -genkey
```

и получить полный список типов эллиптических кривых:

```
openssl ecparam -list_curves
```

Как уже обсуждалось, мы можем использовать OpenSSL для проверки непосредственно простоты:

```
openssl prime 3294823052308402983 4023
```

Обратите внимание, что возвращаемое число всегда является шестнадцатеричным; удивительно, что такие большие числа могут быть проверены на простоту; проверку можно выполнять так быстро именно благодаря теореме Рабина–Миллера.

Глава 7

Алгоритмы в линейной алгебре

> Kraj bez matematyki nie wytrzyma współzawodnictwa z tymi, którzy uprawiają matematykę (Страна без математики не может конкурировать с теми, кто ее придерживается).
>
> *Хуго Штайнгауз цитируется на стр. 147 в книге [Duda (1977)]*

7.1. Введение

В этой главе требуется понимание элементарной линейной алгебры, но не намного выходящее за пределы линейной независимости, детерминантов и характеристического многочлена. Мы сосредоточимся на матрицах, в некоторых случаях на матрицах над конечными полями. Для читателя, который незнаком с основами линейной алгебры, мы рекомендуем книгу [Halmos (1995)].

Мы говорим, что множество векторов $\{v_1, v_2, \ldots, v_n\}$ является *линейно независимым*, если из $\sum_{i=1}^{n} c_i v_i = 0$ вытекает, что $c_i = 0$ для всех i и что они охватывают векторное пространство $V \subseteq \mathbb{R}^n$, если всякий раз, когда $v \in V$, то существует $c_i \in \mathbb{R}$ такое, что $v = \sum_{i=1}^{n} c_i v_i$. Мы обозначаем это как $V =$ промежуток – текстовый буфер$\{v_1, v_2, \ldots, v_n\}$. Множество векторов $\{v_1, v_2, \ldots, v_n\}$ в $\mathbb{R}^n$ является *базисом* для векторного пространства $V \subseteq \mathbb{R}^n$, если они линейно независимы и охватывают V. Пусть $x \cdot y$ обозначает *скалярное* (*точечное*) *произведение* двух векторов, определенных как $x \cdot y = (x_1, x_2, \ldots, x_n) \cdot (y_1, y_2, \ldots, y_n) = \sum_{i=1}^{n} x_i y_i$, и *норма* вектора x определяется как $||x|| = \sqrt{x \cdot x}$. Два вектора x, y являются *ортогональными*, если $x \cdot y = 0$.

7.2. Гауссово исключение

Мы говорим, что матрица находится в *построчно-ступенчатой форме*, если она удовлетворяет следующим двум условиям: (i) если существуют ненулевые строки, то первый ненулевой элемент таких строк равен 1 (*опорному элементу*), и (ii) первый ненулевой элемент строки $i + 1$ находится справа от первого ненулевого элемента строки i. Короче говоря, матрица находится в построчно-ступенчатой форме, если она выглядит следующим образом:

$$\begin{bmatrix} 1 * \dots * * * \dots * * * \dots * * \\ \quad 1 * \dots * * * \dots * * \\ \quad\quad \ddots \quad\quad 1 * \dots * * \\ \quad\quad\quad 0 \quad\quad\quad\quad 1 \dots \\ \quad\quad\quad\quad \ddots \quad\quad\quad \vdots \ddots \end{bmatrix}, \tag{7.1}$$

где знаки $*$ обозначают произвольные элементы.

Мы определяем функцию гауссова исключения (gaussian elimination, GE), $GE : M_{n\times m} \to M_{n\times n}$, как функцию, которая при передаче ей на вход $n\times m$-матрицы выводит $n\times n$-матрицу $GE(A)$ с тем свойством, что $GE(A)A$ находится в построчно-ступенчатой форме. Мы называем это свойство *условием правильности гауссова исключения*.

Покажем, как вычислить гауссово исключение $GE(A)$ при наличии A. Идея, конечно, состоит в том, что $GE(A)$ равняется произведению элементарных матриц, которое приводит A к построчно-ступенчатой форме. Начнем с определения элементарных матриц. Пусть T_{ij} равно матрице с нулями везде, кроме (i, j)-й позиции, где она имеет единицу. Матрица E является *элементарной матрицей*, если E имеет одну из следующих трех форм:

$I + aT_{ij} \quad i \neq j$ (элементарная матрица 1-го рода)
$I + T_{ij} + T_{ji} - T_{ii} - T_{jj}$ (элементарная матрица 2-го рода)
$I + (c - 1)T_{ii} \quad c \neq 0$ (элементарная матрица 3-го рода)

Пусть A равно любой матрице. Если E является элементарной матрицей 1-го рода, то EA равна A, в которой i-я строка заменена суммой i-й строки и произведения a на j-ю строку. Если E является элементарной матрицей 2-рода, то EA равна A, в которой i-я и j-я строки переставлены местами. Если E является элементарной матрицей 3-го рода, то EA равна A, в которой i-я строка умножена на $c \neq 0$.

Алгоритм гауссова исключения представляет собой вид алгоритмов «разделяй и властвуй» с рекурсивным вызовом меньших матриц. То есть мы вычисляем GE рекурсивно, по числу строк A. Если A является $1\times m$-матрицей, $A = [a_{11}a_{12} \dots a_{1m}]$, то:

$$GE(A) = \begin{cases} [1/a_{1i}], & \text{где } i = \min\{1, 2, \dots, m\} \text{ такое, что } a_{i1} \neq 0, \\ [1], & \text{если } a_{11} = a_{12} = \dots = a_{1m} = 0. \end{cases} \tag{7.2}$$

В первом случае $GE(A) = [1/a_{1i}]$, $GE(A)$ является просто элементарной матрицей размера 1×1 и 3-го рода, $c = a_{i1}$. Во втором случае $GE(A)$ является матрицей тождественности 1×1, то есть элементарной матрицей 1-го рода с $a = 0$. Также отметим, что в первом случае мы делим на a_{1i}. Это не требуется, если лежащее в основе поле равно $\mathbb{Z}_2$, так как ненулевой элемент с необходимостью равен 1. Однако наши аргументы соблюдаются независимо от лежащего в основе поля, поэтому мы хотим сделать функцию независимой от поля гауссова исключения.

Предположим теперь, что $n > 1$. Если $A = 0$, то пусть $GE(A) = I$. В противном случае пусть:

$$GE(A) = \begin{bmatrix} 1 & 0 \\ 0 & GE((EA)[1|1]) \end{bmatrix} E, \tag{7.2}$$

где E – это произведение не более $n + 1$ элементарных матриц, определенных ниже. Обратите внимание, что $C[i|j]$ обозначает матрицу C, в которой строки i и j

удалены, поэтому $(EA)[1/1]$ равна матрице A, умноженной на E слева, где затем первая строка и столбец удаляются из результата. Также обратите внимание, что мы удостоверяемся, что $GE(A)$ имеет соответствующий размер (то есть это матрица размера $n \times n$), размещая $GE((EA)[1/1])$ внутрь матрицы, дополненной 1 в левом верхнем углу и нулями в оставшейся части первой строки и столбца.

Теперь мы определим матрицу E с учетом матрицы A, заданной на входе. Существует два случая: первый столбец A равен нулю либо он не равен нулю.

Случай 1: если первый столбец A равен нулю, пусть j равно первому ненулевому столбцу A (такой столбец существует из допущения, что $A \neq 0$). Пусть i равно индексу первой строки A такому, что $A_{ij} \neq 0$. Если $i > 1$, пусть $E = I_{1i}$ (E переставляет строку 1 и строку i). Если $i = 1$, но $A_{ij} = 0$ для $1 < l \leqslant n$, то $E = I$ (ничего не делать). Если $i = 1$, и $1\ i'_1 < i'_2 < \cdots < i'_k$ являются индексами остальных строк с $A_{i'_l j} \neq 0$, то пусть $E = E_{i'_1} E_{i'_2} \cdots E_{i'_k}$, где $E_{i'_l}$ – это элементарная матрица, которая добавляет первую строку матрицы A в i'_l-ю строку матрицы A, чтобы она очищала j-й элемент i'_l-й строки (над полем $\mathbb{Z}_2$; над большим полем для очистки i'_l-й строки нам может потребоваться кратное первой строки).

Случай 2: если первый столбец A не равен нулю, то пусть a_{i1} равно его первому ненулевому элементу (то есть $a_{j1} = 0$, если $j < i$). Мы хотим вычислить последовательность элементарных матриц, произведение которых будет обозначено как E, которое выполняет приведенную ниже последовательность шагов:

1) они переставляют первую и i-ю строки;
2) они делят первую строку на a_{i1};
3) они используют первую строку, чтобы очистить все остальные элементы в первом столбце.

Пусть $a_{i_1 1}, a_{i_2 1}, \ldots, a_{i_k 1}$ равно списку всех ненулевых элементов в первом столбце A, не включая a_{i1}, упорядоченному так, чтобы:

$$i < i_1 < i_2 < \cdots < i_k.$$

Условимся, что если a_{i1} является единственным ненулевым элементом в первой строке, то $k = 0$. Определим E в виде:

$$E = E_{i_1} E_{i_2} \cdots E_{i_k} E' E'',$$

где $E_{i_j} = I - a_{i_j 1} T_{i_j 1}$, поэтому E_{i_j} очищает первый элемент из i_j-й строки матрицы A. Обратите внимание, что если $k = 0$ (если a_{i_1} является единственным ненулевым элементом в первом столбце матрицы A), то $E = E'' E'$. Пусть

$$E'' = I + \left(\frac{1}{a_{i1}} - 1\right) T_{11} \text{ и } E' = I + T_{i1} + T_{1i} - T_{ii} - T_{11}.$$

Таким образом, E'' делит первую строку на a_{i_1}, и E' переставляет местами первую и i-ю строки.

Алгоритм 7.1. Гауссово исключение

Предусловие: $n \times m$-матрица $A = [a_{ij}]$ над неким полем $\mathbb{F}$

```
1: if n = 1 then
2:         if a_11 = a_12 = ··· = a_1m = 0 then
3:                 return [1]
4:         else
```

```
 5:                return [1/a_{1ℓ}], где ℓ = min_{i∈[n]}{a_{1i} ≠ 0}
 6:        end if
 7: else
 8:        if A = 0 then
 9:                return I
10:        else
11:                if первый столбец матрицы A равен нулю then
12:                        Вычислить E как случай 1.
13:                else
14:                        Вычислить E как случай 2.
15:                end if
16:                return [ 1   0          ] E
                          [ 0   GE((EA)[1|1]) ]
17:        end if
18: end if
```

Постусловие: $GE(A)$ находится в построчно-ступенчатой форме

Задача 7.1. Реализуйте алгоритм 7.1 над $\mathbb{F} = \mathbb{R}$ с использованием арифметики с плавающей запятой.

7.2.1. Формальные доказательства правильности над $\mathbb{Z}_2$

Этот раздел содержит расширенный материал, относящийся к области сложности доказательства. Заинтересованному читателю предлагается первая часть обзора 9.4, в частности 9.4.1.1, которая представляет общие сведения, связанные с пропозициональными системами доказательства на основе пропозиционального исчисления PK и расширенного пропозиционального исчисления EPK[1].

Для того чтобы упростить подачу материала, мы ограничимся полем из двух элементов $\mathbb{Z}_2 = \{0,1\}$, но эти результаты соблюдаются над более общими полями. Однако над бóльшими полями приходится мириться с кодировкой элементов поля булевыми переменными; это тривиально в случае двухэлементного поля $\mathbb{Z}_2$.

Мы определяем булеву формулу RowEchelon(C_{11}, C_{12}, ..., C_{nm})[2] как дизъюнкцию (7.4) и (7.5), показанную ниже:

$$\bigwedge_{1\leqslant i\leqslant n, 1\leqslant j\leqslant m} \neg C_{ij}; \tag{7.4}$$

$$\bigwedge_{1\leqslant i\leqslant n, 1\leqslant j\leqslant m} \left((\neg C_{(i+1)1} \wedge \ldots \wedge \neg C_{(i+1)(j-1)} \wedge C_{(i+1)j}) \supset \bigvee_{1\leqslant k\leqslant j-1} C_{ik} \right). \tag{7.5}$$

Обратите внимание, что (7.4) формулирует, что C является нулевой матрицей, и (7.5) формулирует, что первый ненулевой элемент строки $i + 1$ находится справа от первого ненулевого элемента строки i. Более того, если в $(i + 1)$-й строке есть ненулевой элемент, то в i-й строке также должен иметься ненулевой элемент. Об-

---

[1] Пропозициональное исчисление (propositional kalkul, PK) – это пропозициональная система доказательства, которая работает на секвенциях. Расширенная PK (EPK) позволяет вводить новые переменные и объявлять их эквивалентными любой формуле. – *Прим. перев.*

[2] RowEchelon – построчно-ступенчатый. – *Прим. перев.*

ратите внимание, что нам не нужно указывать условие, что первый ненулевой элемент каждой строки равен 1, так как поле равно $\mathbb{Z}_2$; над более общими полями мы должны были бы также указать это условие.

Мы немного злоупотребим формой записи и иногда будем писать *RowEchelon*(C) вместо *RowEchelon*(C_{11}, C_{12}, ..., C_{nm}).

Теорема 7.2. Расширенное пропозициональное исчисление EPK доказывает правильность гауссова исключения с помощью доказательств размерного многочлена в данной матрице. Точнее, семейство тавтологий, задаваемых

$$\{\bigwedge||C = GE(A)A||_{n,m} \supset RowEchelon(C)\}, \tag{7.6}$$

имеет короткие доказательства EPK.

Доказательство. Докажем, что (7.6) имеет короткие доказательства EPK. Точнее, из приведенных ниже конструкций формальных дериваций можно вывести константу d с целью, чтобы размер этих дериваций (измеряемый числом символов) был ограничен $(n + m)^d$, $n, m \geqslant 1$. Мы не задаем d явно.

Мы строим доказательство (7.6) индуктивно на n. Сначала предположим, что A является $1\times m$-матрицей. Пусть $G = GE(A)$, тогда из (7.2) мы видим, что $G = [1]$, поэтому оно представлено одним определением расширения $G_{11} \leftrightarrow 1$. Теперь определим $C = GA$ с m определениями расширений и покажем, что $\bigwedge||C = A||_{1,m}$. Поскольку A имеет только одну строку и является матрицей над $\mathbb{Z}_2$, из этого следует, что A находится в построчно-ступенчатой форме и, следовательно, *RowEchelon*(C).

Теперь предположим, что A является матрицей размера $(n+1)\times m$. Пусть $G' = GE((EA)[1|1])$, и у нас уже есть набор определений расширений для G по индукции. Следовательно, из

$$G = \begin{bmatrix} 1 & 0 \\ 0 & G' \end{bmatrix} E$$

мы получаем множество определений расширений для $G = GE(A)$. Это множество является коротким, потому что определение E является коротким и потому что определение G' является коротким по индукции. Точнее, E задается не более чем $n + 2$ элементарными матрицами размера $(n+1)\times(n+1)$ каждая; следовательно, оно включает в себя $n + 1$ новых матричных определений, причем каждое определение с размером, ограниченным $O((n+1)^3)$ (вспомним определение $||C = A||_{1,m}$). Каждая элементарная матрица, составляющая E (см. определение E выше), над $\mathbb{Z}_2$, имеет определение постоянного размера (в терминах элементов A). Таким образом, определения расширений для E имеют размер, ограниченный $O((n+1)^4)$. Поэтому G может быть определено с помощью $O((n+1)^4)$ + (число определений расширений для G') определений расширений, что в общей сложности составляет $O(\sum_{k=1}^{n+1} k^4) \leqslant O((n + 1)^5)$ определений расширений для G.

Пусть $C' = G'((EA)[1|1])$ и $C = GA$. По индукции

$$\bigwedge||C' = G'((EA)[1|1])||_n \supset RowEchelon(C')$$

имеет доказательство в расширенном пропозициональном исчислении EPK размера, ограниченного $(n + m)^d$. Теперь мы хотим показать, что при наличии определений расширений для G' и G *RowEchelon*(C) $\supset$ *RowEchelon*(C) имеет короткие доказательства в расширенном пропозициональном исчислении EPK. Поскольку

$$C = GA = \begin{bmatrix} 1 & 0 \\ 0 & G' \end{bmatrix} EA = \begin{bmatrix} \text{первая строка } EA \\ 0 \quad G'((EA)[1|1]) \end{bmatrix} = \begin{bmatrix} \text{первая строка } EA \\ 0 \quad C' \end{bmatrix}.$$

Для того чтобы это увидеть, обратите внимание, что первый столбец EA равен нулю, за исключением, возможно, первого элемента. По выбору E, либо $(EA)_{11} \neq 0$, в каковом случае мы имеем $RowEchelon(C)$, либо первый ненулевой элемент первой строки EA находится слева от первого ненулевого столбца C', в каковом случае мы также имеем $RowEchelon(0)$. Также отметим, что в приведенных выше рассуждениях мы используем ассоциативность итерированных матричных произведений. То есть мы исходим из того, что не важно, как мы заключаем в скобки итерированное матричное произведение, так как по ассоциативности мы всегда получаем один и тот же результат. Это может быть также показано с помощью коротких доказательств в расширенном пропозициональном исчислении EPK. □

Теорема 7.3. Существование обратного $GE(A)$ может быть показано с помощью коротких доказательств расширенной пропозициональной системы EPK.

Доказательство. Мы должны показать, что при наличии $||G = GE(A)||_n$ булевы переменные G_{11}^{-1}, G_{21}^{-1}, ..., G_n^{-n}, которые соответствуют G^{-1}, могут быть сконструированы с помощью коротких определений расширений и что EPK доказывает $||GG^{-1} = I||_n$ с помощью коротких доказательств.

Так же, как мы определили G индуктивно с помощью определений расширений, мы определяем G^{-1} индуктивно. С учетом $E = E_{i_1}E_{i_2}\cdots E_{i_k}E'E''$ мы можем вычислить E^{-1} сразу, приняв $E''^{-1}E'^{-1}E_{i_k}^{-1}\cdots E_{i_2}^{-1}E_{i_1}^{-1}$. Каждая из этих инверсий может быть вычислена очень легко, потому что они являются элементарными матрицами. Поэтому, поскольку мы имеем дело с $\mathbb{Z}_2$, $E'' = E''$, и E' также является своей собственной инверсией, и E_{i_j} является матрицей с единицами по диагонали, и 1 в позиции (p, q), поэтому $E_{i_j}^{-1}$ является матрицей с единицами по диагонали и 1 в позиции (q, p).

Таким образом, мы показали, как вычислить G^{-1}. Нам по-прежнему нужно показать, что семейство тавтологий $\{||GG^{-1} = I||_{n,m}\}$ имеет короткие доказательства EPK для любой $n{\times}m$-матрицы A. Мы можем доказать это индуктивно на числе строк матрицы A, как и в доказательстве теоремы 7.2, поэтому мы его здесь не повторяем. □

Следствие 7.4. С помощью коротких доказательств EPK можно показать, что $GE(A)A$ имеет единицы на главной диагонали, либо ее последняя строка равна нулю.

Доказательство. Истинность этого логического утверждения очевидна из (7.1). Пусть $C = GA$, и предположим, что на диагонали есть нулевой элемент, то есть $\neg\bigwedge_{1\leqslant i\leqslant n} C_{ii} \leftrightarrow 1$. Мы хотим показать, что последняя строка равна нулю, $\bigwedge_{1\leqslant i\leqslant n} C_{ni} \leftrightarrow 0$. Мы знаем, что $RowEchelon(C)$ является валидной и доказуемой в полиразмерном EPK (по теореме (7.2)). Из (7.5) мы можем заключить с помощью коротких доказательств EPK, что:

$$\bigwedge_{1\leqslant j\leqslant k} \neg C_{ij} \supset \bigwedge_{1\leqslant j\leqslant k+1} \neg C_{(i+1)j}. \tag{7.7}$$

То есть если первые k элементов строки i равны нулю, то первые $(k + 1)$ элементов строки $(i + 1)$ равны нулю. Пусть C_{ii} равно нулю, где i – наименьший. Теперь из (7.7) докажем, что:

$$\bigwedge_{1 \leqslant j \leqslant i} \neg C_{ij} \leftrightarrow 0. \tag{7.8}$$

Используя (7.7) неоднократно, для $0 \leqslant k \leqslant n - i$, мы показываем, что первые $(i + k)$ элементов строки $(i + k)$ равны нулю. Таким образом, можно сделать вывод, что первые n элементов n-й строки равны нулю, и, следовательно, n-я (последняя) строка полностью равна нулю.

Фактически обратите внимание, что при наличии *RowEchelon*(C) нам было необходимо только полиразмерное исчисление РК, и этого было достаточно для доказательства, что если некий C_{ii} равен нулю, то последняя строка матрицы C равна нулю.

7.3. Алгоритм Грама-Шмидта

Задача 7.5. Пусть $V \subseteq \mathbb{R}^n$ равно векторному пространству и $\{v_1, v_2, \ldots, v_n\}$ равно его базису. Рассмотрим алгоритм 7.2 и покажем, что он порождает ортогональный базис $\{v_1^*, v_2^*, \ldots, v_n^*\}$ для векторного пространства V. Другими словами, покажем, что $v_i^* \cdot v_j^* = 0$ при $i \neq j$ и что *промежуток*$\{v_1, v_2, \ldots, v_n\}$ = *промежуток*$\{v_1^*, v_2^*, \ldots, v_n^*\}$. Строка 4 содержит деление на квадрат нормы v_j^*; покажите, что это никогда не приведет к попытке деления на ноль.

Алгоритм 7.2. Грам-Шмидт

Предусловие: $\{v_1, \ldots, v_n\}$ – базис для $\mathbb{R}^n$

1: $v_1^* \leftarrow v_1$
2: **for** $i = 2, 3, \ldots, n$ **do**
3: **for** $j = 1, 2, \ldots, (i - 1)$ **do**
4: $\mu_{ij} \leftarrow (v_i \cdot v_j^*) / ||v_j^*||^2$
5: **end for**
6: $v_i^* \leftarrow v_i - \sum_{j=1}^{i-1} \mu_{ij} v_j^*$
7: **end for**

Постусловие: $\{v_1^*, \ldots, v_n^*\}$ – ортогональный базис для $\mathbb{R}^n$

Задача 7.6. Реализуйте алгоритм Грама-Шмидта (алгоритм 7.2), но со следующей особенностью: вместо вычисления над $\mathbb{R}$, действительными числами, вычислите над $\mathbb{Z}_2$, полем из двух элементов, где сложение и умножение определяются следующим образом:

+	0	1
0	0	1
1	1	0

.	0	1
0	0	0
1	0	1

На самом деле эта «особенность» делает реализацию намного проще, так как вам не придется решать вопросы прецизионности, связанные с реализацией операций деления над полем действительных чисел.

7.4. Гауссова редукция решетки

Предположим, что $\{v_1, v_2, \ldots, v_n\}$ – это линейно независимые векторы в $\mathbb{R}^n$. Решетка L, охваченная этими векторами, является множеством $\{\sum_{i=1}^{n} c_i v_i : c_i \in \mathbb{Z}\}$, то есть L

состоит из линейных комбинаций векторов $\{v_1, v_2, \ldots, v_n\}$, где коэффициенты ограничены целыми числами.

Задача 7.7. Предположим, что $\{v_1, v_2\}$ охватывают решетку в $\mathbb{R}^2$. Рассмотрим алгоритм 7.3 и покажем, что он завершается и выводит новый базис $\{v_1, v_2\}$ для L, где v_1 – это кратчайший вектор в решетке L, то есть $||v_1||$ – это как можно меньший среди всех векторов L.

Алгоритм 7.3. Гауссова редукция решетки в размерности 2

Предусловие: $\{v_1, v_2\}$ – линейно-независимы в $\mathbb{R}^2$

```
loop
    if ||v2|| < ||v1|| then
        поменять местами v1 и v2
    end if
    m ← v1·v2/||v1||^2
    if m = 0 then
        return v1, v2
    else
        v2 ← v2 − mv1
    end if
end loop
```

7.5. Вычисление характеристического многочлена

Существует два быстрых алгоритма вычисления характеристического многочлена матрицы: *алгоритм Чанки* и *алгоритм Берковица*. Характеристический многочлен матрицы обычно определяется как $p_A(x) = \det(xI - A)$ для заданной матрицы A. Обозначим через p_A^{CSANKY} и p_A^{BERK} коэффициенты характеристического многочлена A, заданного соответственно в виде векторов-столбцов. Пусть $p_A^{\text{CSANKY}}(x)$ и $p_A^{\text{BERK}}(x)$ обозначают фактические характеристические многочлены с коэффициентами, вычисленными соответствующими алгоритмами.

7.5.1. Алгоритм Чанки

Симметричные многочлены Ньютона определяются следующим образом: $s_0 = 1$, и для $1 \leqslant k \leqslant n$ посредством:

$$s_k = \frac{1}{k}\sum_{i=1}^{k}(-1)^{i-1}s_{k-i}\operatorname{tr}(A^i). \tag{7.9}$$

Тогда $p_A^{\text{CSANKY}}(x) = s_0x^n - s_1x^{n-1} + s_2x^{n-2} - \cdots \pm s_nx^0$.

Задача 7.8. Напишите алгоритм, который реализует алгоритм Чанки с симметричными многочленами Ньютона.

Теперь мы представим алгоритм Чанки с матричными операциями, следуя идеям в §13.4 публикации [von zur Gathen (1993)]. Мы переформулируем (7.9) в матричной форме: $s = Ts - b$, где s, T, b заданы, соответственно, следующим образом:

$$\begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_n \end{pmatrix}, \quad \begin{pmatrix} 0 & 0 & 0 & \dots \\ \frac{1}{2}\mathrm{tr}(A) & 0 & 0 & \dots \\ \frac{1}{2}\mathrm{tr}(A^2) & \frac{1}{3}\mathrm{tr}(A) & 0 & \dots \\ \frac{1}{4}\mathrm{tr}(A^3) & \frac{1}{4}\mathrm{tr}(A^2) & \frac{1}{4}\mathrm{tr}(A) & \dots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}, \quad \begin{pmatrix} \mathrm{tr}(A) \\ \frac{1}{2}\mathrm{tr}(A^2) \\ \vdots \\ \frac{1}{n}\mathrm{tr}(A^n) \end{pmatrix}.$$

Тогда $s = -b(I - T)^{-1}$. Обратите внимание, что $(I - T)$ является обратимой матрицей, поскольку она является нижнетреугольной, с единицами на главной диагонали. Обратная от $(I - T)$ может быть вычислена рекурсивно, используя следующую идею: если C является нижнетреугольной без нулей на главной диагонали, то

$$C = \begin{pmatrix} C_1 & 0 \\ E & C_2 \end{pmatrix} \Rightarrow C^{-1} = \begin{pmatrix} C_1^{-1} & 0 \\ -C_2^{-1}EC_1^{-1} & C_2^{-1} \end{pmatrix}.$$

Существует $O(\log(n))$ шагов, и вся процедура может быть просимулирована в $O(\log^2(n))$ параллельных шагах.

Задача 7.9. Реализуйте алгоритм Чанки с матричными операциями.

7.5.2. Алгоритм Берковица

Алгоритм Берковица, так же как алгоритм Чанки, позволяет нам свести вычисление характеристического многочлена к матричному возведению в степень. Его преимущество в том, что он работает над любым полем.

Алгоритм Берковица вычисляет характеристический многочлен матрицы в терминах характеристического многочлена ее главного минора:

$$A = \begin{pmatrix} a_{11} & R \\ S & M \end{pmatrix}, \tag{7.10}$$

где R – это матрица-строка размера $1\times(n-1)$, S – это матрица-столбец размера $(n-1)\times 1$, и M равно $(n-1)\times(n-1)$. Пусть $p(x)$ и $q(x)$ равны характеристическим многочленам соответственно A и M. Предположим, что коэффициенты p образуют вектор-столбец

$$p = (p_n p_{n-1} \dots p_0)^t, \tag{7.11}$$

где p_i – это коэффициент x^i в $\det(xI - A)$, и схожим образом для q. Тогда:

$$p = C_1 q, \tag{7.12}$$

где C_1 – это нижнетреугольная $(n+1)\times n$-матрица Теплица (*матрица Теплица* означает, что значения на каждой диагонали являются постоянными) и где элементы в первом столбце определяются следующим образом: $c_{i1} = 1$, если $i = 1$, $c_{i1} = -a_{11}$, если $i = 2$, и $c_{i1} = -(RM^{i-3}S)$, если $i \geqslant 3$. Алгоритм Берковица состоит в том, чтобы повторять это для q и продолжать с целью, чтобы p выражалось как произведение матриц. Итак:

$$p_A^{\mathrm{BERK}} = C_1 C_2 \cdots C_n, \tag{7.13}$$

где C_i – это определенная выше $(n+2-i)\times(n+1-i)$-матрица Теплица, за исключением того, что A заменяется ее i-й основной подматрицей. Отметим, что $C_n = (1 - a_{nn})^t$.

Поскольку каждый элемент C_i может быть явно определен в терминах использования матричного возведения в степень и поскольку итерационное матричное произведение может быть сведено к матричному возведению в степень стандартным методом, все произведение (7.13) может быть выражено в терминах использования матричного возведения в степень.

Задача 7.10. Напишите программу, реализующую алгоритм Берковица.

7.5.3. Доказательство свойств характеристического многочлена

Лемма 7.11. Подобные матрицы имеют один и тот же характеристический многочлен; то есть если P является любой обратимой матрицей, то $p_A = p_{PAP^{-1}}$.

Доказательство. Пронаблюдаем, что $\mathrm{tr}(AB) = \sum_i \sum_j a_{ij} b_{ji} = \sum_j \sum_i b_{ji} a_{ij} = \mathrm{tr}(BA)$, поэтому, используя ассоциативность матричного умножения, $\mathrm{tr}(PA^iP^{-1}) = \mathrm{tr}(A^iPP^{-1}) = \mathrm{tr}(A^i)$. Осматривая (7.9), мы видим, что доказательство по индукции на s_i доказывает эту лемму. □

Лемма 7.12. Если A является матрицей вида:

$$\begin{pmatrix} B & 0 \\ C & D \end{pmatrix}, \tag{7.14}$$

где B и D – это квадратные матрицы (не обязательно одинакового размера) и правый верхний угол равен нулю, тогда $p_A(x) = p_B(x) \cdot p_D(x)$.

Доказательство. Пусть s_i^A, s_i^B, s_i^D равны коэффициентам характеристических многочленов (как дано в (7.9)) соответственно A, B, D. Мы хотим показать по индукции на i, что

$$s_i^A = \sum_{j+k=i} s_j^B s_k^D,$$

из чего следует утверждение леммы. Базовый случай: $s_0^A, s_0^B, s_0^D = 1$. На индукционном шаге по определению и по индукционной гипотезе мы имеем, что s_{i+1}^A равно

$$= \sum_{j=0}^{i} (-1)^j s_{i-j}^A \mathrm{tr}(A^{j+1}) = \sum_{j=0}^{i} (-1)^j \left[\sum_{p+q=i-j} s_p^B s_q^D \right] \mathrm{tr}(A^{j+1})$$

и по форме A (то есть (7.14)):

$$= \sum_{j=0}^{i} (-1)^j \left[\sum_{p+q=i-j} s_p^B s_q^D \right] (\mathrm{tr}(B^{j+1}) + \mathrm{tr}(D^{j+1}));$$

чтобы увидеть, как эта формула упрощается, разделим ее на две части:

$$= \sum_{j=0}^{i} (-1)^j \left[\sum_{p+q=i-j} s_p^B s_q^D \right] (\mathrm{tr}(B^{j+1}) + \sum_{j=0}^{i} (-1)^j \left[\sum_{p+q=i-j} s_p^B s_q^D \right] \mathrm{tr}(D^{j+1})).$$

Рассмотрим сначала левую сторону. Когда $q = 0$, p варьируется над $\{i, i-1, \ldots, 0\}$, и $j+1$ варьируется над $\{1, 2, \ldots, i+1\}$, и, следовательно, по определению, мы получаем s_{i+1}^B. Схожим образом, когда $q = 1$, мы получаем s_i^B и т. д. до тех пор, пока не получим s_1^B. Следовательно, мы имеем:

$$= \sum_{j=0}^{i+1} s_{i-j}^B s_j^D + \sum_{j=0}^{i} (-1)^j \left[\sum_{p+q=i-j} s_p^B s_q^D \right] (\mathrm{tr}(D^{j+1}).$$

То же самое рассуждение, но, задавая p вместо q с правой стороны, мы получаем:

$$= \sum_{j=0}^{i+1} s_{i-j}^B s_j^D + \sum_{j=0}^{i+1} s_j^B s_{i-j}^D = \sum_{j+k=i+1} s_j^B s_k^D,$$

что дает нам индукционный шаг и доказательство леммы. □

Для того чтобы показать, что $p_A(A) = 0$, достаточно показать, что $p_A(A)e_i = 0$ для всех векторов e_i в стандартном базисе $\{e_1, e_2, \ldots, e_n\}$. Пусть k равно наибольшему целому числу такому, что

$$\{e_i, Ae_i, \ldots, A^{k-1}e_i\} \tag{7.15}$$

линейно независимо; мы знаем, что $k - 1 < n$ по принципу линейной независимости (тут мы впервые используем линейную независимость). Тогда (7.15) является базисом для подпространства W на $\mathbb{F}^n$, и W является инвариантом относительно A, то есть при наличии любого $w \in W$, $Aw \in W$.

Используя гауссово исключение, запишем $A^k e_i$ как линейную комбинацию векторов в (7.15). Используя коэффициенты этой линейной комбинации, напишем монический многочлен

$$g(x) = x^k + c_1 x^{k-1} + \cdots + c_k x^0 \tag{7.16}$$

такой, что $g(A)e_i = 0$.

Пусть A_W равно A, ограниченной базисом (7.15), то есть A_W является матрицей, представляющей линейное преобразование T_A: $\mathbb{F}^n \to \mathbb{F}^n$, индуцированное матрицей A, ограниченной подпространством W. Матрица A_W^t имеет следующую простую форму:

$$\left(\begin{array}{c|cccc} 0 & 0 & 0 & \ldots & -c_k \\ \hline 1 & 0 & 0 & \ldots & -c_{k-1} \\ 0 & 1 & 0 & \ldots & -c_{k-2} \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & 0 & \ldots & -c_1 \end{array}\right), \tag{7.17}$$

то есть это сопровождающая матрица многочлена $g(x)$. Поскольку $p_A = p_{A^t}$, мы рассматриваем транспонированную версию A_W, так как A_W^t имеет свойство, что ее главная подматрица также является сопровождающей матрицей, и это будет использоваться в доказательстве по индукции в следующей лемме.

Лемма 7.13. Многочлен $g(x)$ является характеристическим многочленом A_W, другими словами, $g(x) = p_{A_W}(x)$.

Доказательство. Мы отбрасываем W из A_W, поскольку нет опасности путаницы (исходная матрица A в доказательстве не появляется); таким образом, A является $k \times k$-матрицей с единицами ниже главной диагонали и нулями везде, кроме (возможно) в последнем столбце, где она имеет отрицания коэффициентов $g(x)$.

Как было отмечено выше, A подразделяется на четыре квадранта, причем левый верхний содержит только 0. Пусть $R = (0 \ldots 0 \ -c_k)$ равно вектору-строке в правом верхнем квадранте. Пусть $S = e_1$ равно вектору-столбцу в левом нижнем квадранте, то есть первому столбцу A без верхнего элемента. Наконец, пусть M равно главной подматрице A, $M = A[1|1]$, правому нижнему квадранту.

Пусть $s_0, s_1, \ldots, s_k$ равно симметричным многочленам Ньютона матрицы A.

Для того чтобы доказать, что $g(x) = p_{A_{T_w}}(x)$, мы докажем что-то более сильное: покажем, что (i) для всех $0 \leqslant i \leqslant k$ $(-1)^i s_i = c_i$ и (ii) $p_A(A) = 0$.

Мы показываем это по индукции на размере матрицы A. Поскольку главная подматрица A (то есть M) является также сопровождающей матрицей, мы исходим из того, что для $i < k$ коэффициенты симметричного многочлена M равны c_i и что $p_M(M) = 0$. (Обратите внимание, что базовый случай индукции является 1×1-матрицей и он тривиален для доказательства.)

Так как для $i < k$, $\mathrm{tr}(A^i) = \mathrm{tr}(M^i)$, то из (7.9) и индукционной гипотезы следует, что для $i < k$, $(-1)^i s_i = c_i$ (отметим, что $s_0 = c_0 = 1$).

Далее мы показываем, что $(-1)^k s_k = c_k$. По определению (то есть по (7.9)) мы имеем, что s_k равно:

$$(1/k)(s_{k-1}\mathrm{tr}(A) - s_{k-2}\mathrm{tr}(A^2) + \cdots + (-1)^{k-2}s_1\mathrm{tr}(A^{k-1}) + (-1)^{k-1}s_0\mathrm{tr}(A^k)),$$

и по индукционной гипотезе и тому факту, что для $i < k$ $\mathrm{tr}(A^i) = \mathrm{tr}(M^i)$, мы имеем:

$$= (1/k)(-1)^{k-1}(c_{k-1}\mathrm{tr}(M) + c_{k-2}\mathrm{tr}(M^2) + \cdots + c_1\mathrm{tr}(M^{k-1}) + c_0\mathrm{tr}(A^k)).$$

Обратите внимание, что $\mathrm{tr}(A^k) = -kc_k + \mathrm{tr}(M^k)$, поэтому:

$$= (1/k)(-1)^{k-1}[c_{k-1}\mathrm{tr}(M) + c_{k-2}\mathrm{tr}(M^2) + \cdots + c_1\mathrm{tr}(M^{k-1}) + c_0\mathrm{tr}(M^k)] + (-1)^k c_k.$$

Заметим, что

$$\mathrm{tr}(c_{k-1}M + c_{k-2}M^2 + \cdots + c_1M^{k-1} + c_0M^k) = \mathrm{tr}(p_M(M)M) = \mathrm{tr}(0) = 0,$$

поскольку $p_M(M) = 0$ по индукционной гипотезе. Поэтому $s_k = (-1)^k c_k$.

Остается доказать, что $p_A(A) = \sum_{i=0}^{k} c_i A^{k-i} = 0$. Сначала покажем, что для $1 \leqslant i \leqslant (k-1)$:

$$A^{i+1} = \left(\begin{array}{c|c} 0 & RM^i \\ \hline M^iS & \sum_{j=0}^{i-1} M^jSRM^{(i-1)-j} + M^{i+1} \end{array}\right). \tag{7.18}$$

(Для A формы, указанной в пункте (7.17), и R, S, M, определенных как в первом абзаце доказательства.) Определяем w_i, X_i, Y_i, Z_i следующим образом:

$$\begin{aligned} A^{i+1} &= \begin{pmatrix} w_{i+1} & X_{i+1} \\ Y_{i+1} & Z_{i+1} \end{pmatrix} = \begin{pmatrix} w_i & X_i \\ Y_i & Z_i \end{pmatrix}\begin{pmatrix} 0 & R \\ S & M \end{pmatrix} \\ &= \begin{pmatrix} X_iS & w_iR + X_iM \\ Z_iS & Y_iR + Z_iM \end{pmatrix}. \end{aligned} \tag{7.19}$$

Мы хотим показать, что самая правая матрица в (7.19) равна правой стороне (7.18). Во-первых, обратите внимание, что:

$$X_{i+1} = \sum_{j=0}^{i} w_{i-j}RM^j \quad w_{i+1} = \sum_{j=0}^{i-1} (RM^jS)w_{i-1-j}. \tag{7.20}$$

С договоренностью, что $w_0 = 1$. Поскольку $w_1 = 0$, прямая индукция показывает, что $w_{i+1} = 0$. Таким образом, на этом этапе самая правая матрица (7.19) может быть упрощена до:

$$\begin{pmatrix} 0 & RM^i \\ Z_iS & Y_iR + Z_iM \end{pmatrix}.$$

Снова по лемме 5.1 из работы [Soltys и Cook (2004)] у нас есть:

$$Y_{i+1} = M^iS + \sum_{j=0}^{i=2}(RM^jS)Y_{i-1-j} \quad z_{i+1} = M^{i+1} + \sum_{j=0}^{i-1} Y_{i-1-j}RM^j.$$

По тому же рассуждению, что и выше, $\sum_{j=0}^{i-2}(RM^jS)Y_{i-1-j} = 0$, поэтому, соединив все это вместе, мы получим правую сторону (7.18).

Используя индукционную гипотезу ($p_M(M) = 0$), легко показать, что первая строка и столбец $p_A(A)$ равны нулю. Кроме того, согласно индукционной гипотезе, член M^{i+1} в главной подматрице $p_A(A)$ исчезает, но оставляет c_kI. Таким образом, из этого следует, что $p_A(A) = 0$, если мы покажем, что

$$\sum_{i=2}^{k} c_{k-i} \sum_{j=0}^{i-2} M^jSRM^{(i-2)-j} \tag{7.21}$$

равно $-c_kI$.

Некоторые наблюдения о (7.21): для $0 \leqslant j \leqslant i - 2 \leqslant k - 2$ первый столбец M^j равен e_{j+1}. И SR является матрицей нулей, в которой $-c_k$ находится в правом верхнем углу. Таким образом, M^jSR является матрицей нулей, за исключением последнего столбца $-c_ke_{j+1}$. Таким образом, $M^jSRM^{(i-2)-j}$ является матрицей нулей везде, кроме строки $(j + 1)$, где она имеет нижнюю строку $M^{(i-2)-j}$, умноженную на $-c_k$. Пусть $\mathbf{m}^{(i-2)-j}$ обозначает вектор-строку размера $1\times(k-1)$, состоящую из нижней строки $M^{(i-2)-j}$. Следовательно, (7.21) равно:

$$-c_k \cdot \begin{pmatrix} \sum_{i=2}^{k} c_{k-i}\mathbf{m}^{(i-2)} \\ \sum_{i=3}^{k} c_{k-i}\mathbf{m}^{(i-3)} \\ \vdots \\ \sum_{i=k}^{k} c_{k-i}\mathbf{m}^{(i-k)} \end{pmatrix}. \tag{7.22}$$

Мы хотим показать, что (7.22) равно $-c_kI$, для того чтобы закончить доказательство $p_A(A) = 0$. Для этого пусть l обозначает l-ю строку матрицы в (7.22), начиная с нижней строки. Мы хотим показать по индукции на l, что l-я строка равна e_{k-l}.

Базовый случай равен $l = 0$:

$$\sum_{i=k}^{k} c_{k-i}\mathbf{m}^{(i-k)} = c_0\mathbf{m}^0 = e_k,$$

и доказательство завершено.

На индукционном шаге обратим внимание, что $\mathbf{m}^{l+1}$ равно $\mathbf{m}^l$, сдвинутому влево на одну позицию, и с

$$\mathbf{m}^l \cdot (-c_{k-1} - c_{k-2} \ldots - c_1)^t \tag{7.23}$$

в последней позиции. Введем еще несколько обозначений: пусть $\mathbf{r}_l$ обозначает строку $k - l$ (7.22). Тем самым $\mathbf{r}_l$ равно вектору-строке размера $1\times(k-1)$. Пусть $\overleftarrow{r}_l$ обозначает $\boldsymbol{r}_l$, сдвинутый на одну позицию влево, и с нулем в последней позиции. Это можно кратко изложить следующим образом:

$$\overleftarrow{r}_l \stackrel{\text{def}}{=} \lambda ij\langle 1, (k-1), e(\mathbf{r}_l, 1, i+1)\rangle.$$

Исходя из (7.22) и (7.23), видно, что:

$$\mathbf{r}_{l+1} = \overleftarrow{r}_l + [\mathbf{r}_l \cdot (-c_{k-1} - c_{k-2} \ldots - c_1)^t]e_k + c_l\mathbf{m}^0.$$

(Здесь знак «·» в квадратных скобках означает скалярное произведение двух векторов.) Используя индукционную гипотезу, $\overleftarrow{\mathbf{r}}_l = e_{k-(l+1)}$ и

$$\mathbf{r}_l \cdot (-c_{k-1} - c_{k-2} \ldots - c_1)^t = e_{k-l} \cdot (-c_{k-1} - c_{k-2} \ldots - c_1)^t = -c_l,$$

поэтому $r_{l+1} = e_{k-l} - c_l e_k + c_l e_k = e_{k-(l+1)}$, как и требовалось. На этом заканчивается доказательство того, что матрица в (7.22) является матрицей тождественности, что, в свою очередь, доказывает, что (7.21) равно $-c_k I$, и на этом заканчивается доказательство $p_A(A) = 0$, что в итоге заканчивает основной аргумент индукции и доказывает лемму. □

Интересно отметить, что лемма 7.13 также может быть доказана (практически) для алгоритма Берковица, и это доказательство на самом деле намного проще: рассмотрим еще раз матрицу, заданную (7.17). Мы индуктивно исходим из того, что p_M^{BERK} (характеристический многочлен главной подматрицы (7.17)) задан $(1\ c_1\ c_2 \ldots c_{k-1})^t$. Поскольку $R = (0 \ldots 0 - c_k)$ и $S = e_1$, $p_A^{\text{BERK}} = B \cdot p_M^{\text{BERK}}$, где B (матрица, заданная алгоритмом Берковица) является матрицей размера $(n+1)\times n$ с единицами на главной диагонали, нулями везде, кроме $+c_k$ в позиции $(n+1, 1)$. Из этого легко увидеть, что p_A^{BERK} задается с помощью $(1\ c_1\ c_2 \ldots c_k)^t$.

Лемма 7.14. Многочлен $g(x)$ делит $p_A(x)$.

Доказательство. Разложим (7.15) до полного базиса $\mathbb{F}^n$:

$$B = \{e_i, Ae_i, \ldots, A^{k-1}e_i, e_{j_1}, e_{j_2}, \ldots, e_{j_{n-k}}\}.$$

Это разложение можно легко выполнить с помощью гауссова исключения, проверяя, какие векторы из стандартного базиса ($\{e_1, e_2, \ldots, e_n\}$) находятся в промежутке, состоящем из (7.15) и тех векторов, которые уже были добавлены, и добавляя только недобавленные. Это единственное место (кроме абзаца после доказательства леммы 7.12), где мы должны использовать принцип линейной независимости.

Пусть P равно изменению базиса для A из стандартного базиса в B. Тогда

$$PAP^{-1} = \begin{pmatrix} A_W & 0 \\ * & E \end{pmatrix},$$

где A_W – это блок размера $k\times k$, E – блок размера $(n-k)\times(k-n)$ (соответствующий разложению), и у нас есть блок нулей выше E, поскольку W инвариантно относительно A. В силу леммы 7.12 следует, что $p_A(x) = p_{PAP^{-1}}(x) = p_{A_W}(x) \cdot p_E(x)$. По лемме 7.13, $p_{A_W} = g(x)$, и поэтому $g(x)$ делит $p_A(x)$. □

Теорема 7.15. Мы можем доказать теорему Кэли–Гамильтона из принципа линейной независимости, когда характеристический многочлен вычисляется алгоритмом Чанки.

Доказательство. По лемме 7.1:

$$p_A(A)e_i = (p_{A_W}(A) \cdot p_E(A))e_i = (g(A) \cdot p_E(A))e_i = p_E(A) \cdot (g(A)e_i) = 0.$$

Поскольку это верно для любого e_i в стандартном базисе, из этого следует, что $p_A(A) = 0$. □

7.6. Ответы к избранным задачам

Задача 7.5. Мы намерены доказать инвариант цикла на внешнем цикле алгоритма 7.2, то есть мы собираемся доказать инвариант цикла на цикле for (индексированном на i), который начинается в строке 2 и заканчивается в строке 7. Наш инвариант состоит из двух частей: после k-й итерации цикла соблюдаются следующие два формальных суждения:

1) множество $\{v_1^*, \cdots, v_{k+1}^*\}$ является ортогональным, и
2) промежуток$\{v_1, \cdots, v_{k+1}\}$ = промежуток$\{v_1^*, \cdots, v_{k+1}^*\}$.

Базовый случай: после нуля итераций цикла for, то есть перед тем, как цикл for был вообще исполнен, то есть со строки 1 алгоритма, $v_1^* \leftarrow v_1$, и поэтому первое формальное суждение является истинным, потому что $\{v_1^*\}$ является ортогональным (множество, состоящее из единственного ненулевого вектора, всегда является ортогональным – и $v_1^* = v_1 \neq 0$, потому что допущение (то есть предусловие) заключается в том, что $\{v_1, \cdots, v_n\}$ является линейно независимым, и поэтому ни один из этих векторов не может быть нулевым), и второе формальное суждение также соблюдается тривиально, так как если $v_1^* = v_1$, то промежуток$\{v_1\}$ = промежуток$\{v_1^*\}$.

Индукционный шаг: предположим, что два условия соблюдаются после первых k итераций цикла; мы покажем, что они продолжают соблюдаться после $k + 1$ итераций. Рассмотрим:

$$v_{k+2}^* = v_{k+2} - \sum_{j=1}^{k+1} \mu_{(k+1)j} v_j^*,$$

которое мы получаем непосредственно из строки 6 алгоритма; обратите внимание, что внешний цикл for индексируется на i, который проходит от 2 до n, поэтому после k-го исполнения строки 2 для $k \geqslant 1$ значение индекса i равно $k + 1$. Покажем первое формальное суждение, то есть что $\{v_1^*, \cdots, v_{k+2}^*\}$ являются ортогональными. Поскольку по индукционной гипотезе мы знаем, что $\{v_1^*, \cdots, v_{k+1}^*\}$ уже являются ортогональными, достаточно показать, что для $1 \leqslant l \leqslant k + 1$, $v_l^* \cdot v_{k+2}^* = 0$, что мы и делаем далее:

$$\begin{aligned} v_l^* \cdot v_{k+2}^* &= v_l^* \cdot \left(v_{k+2} - \sum_{j=1}^{k+1} \mu_{(k+2)j} v_j^* \right) \\ &= (v_l^* \cdot v_{k+2}) - \sum_{j=1}^{k+1} \mu_{(k+2)j} (v_l^* \cdot v_j^*), \end{aligned}$$

и так как $v_l^* \cdot v_j^* = 0$, если только не является истинным, что $l = j$, то мы имеем:

$$= (v_l^* \cdot v_{k+2}) - \mu_{(k+2)l} (v_l^* \cdot v_l^*),$$

и, используя строку 4 алгоритма, мы записываем:

$$= (v_l^* \cdot v_{k+2}) - \frac{v_{k+2} \cdot v_l^*}{||v_l^*||^2}(v_l^* \cdot v_l^*) = 0,$$

где мы использовали тот факт, что $v_l \cdot v_l = ||v_l||^2$ и что $v_{k+2} = v_{k+2} \cdot v_l^*$.

Для второго формального суждения инварианта цикла нам нужно показать, что

$$\text{span}\{v_1, \ldots, v_{k+2}\} = \text{span}\{v_1^*, \ldots, v_{k+2}^*\}, \tag{7.24}$$

приняв по индукционной гипотезе, что промежуток $\{v_1, \ldots, v_{k+1}\}$ = промежуток$\{v_1^*, \cdots, v_{k+1}^*\}$. Аргументация будет основана на строке 6 алгоритма, которая предоставляет нам следующее равенство:

$$v_{k+2}^* = v_{k+2} - \sum_{j=1}^{k+1} \mu_{(k+2)j} v_j^*. \tag{7.25}$$

С учетом индукционной гипотезы, для того чтобы показать (7.24), нам нужно показать только следующие две вещи:

1) $v_{k+2} \in$ промежуток$\{v_1^*, \ldots, v_{k+2}^*\}$ и
2) $v_{k+2}^* \in$ промежуток$\{v_1, \ldots, v_{k+2}\}$.

Используя (7.25), мы немедленно получаем, что $v_{k+2} = v_{k+2}^* + \sum_{j=1}^{k+1} \mu_{(k+2)j} v_j^*$, и поэтому мы имеем (1). Для того чтобы показать (2), мы отмечаем, что

$$\text{промежуток}\{v_1, \ldots, v_{k+2}\} = \text{промежуток}\{v_1^*, \ldots, v_{k+1}^*, v_{k+2}\}$$

по индукционной гипотезе, и поэтому у нас есть то, что нам нужно непосредственно из (7.25).

Наконец, обратите внимание , что в строке 4 алгоритма мы ни разу не делим на ноль, потому что мы всегда делим на $||v_j^*||$, и единственно, когда норма может быть нулем, – это если сам вектор, v_j^*, является нулевым. Но из постусловия мы знаем, что $\{v_1^*, \ldots, v_n^*\}$ является базисом, и поэтому эти векторы должны быть линейно независимыми, и, значит, ни один из них не может быть нулевым.

Задача 7.7. Ссылку на этот алгоритм можно найти в работе [Hoffstein и соавт. (2008)] в §6.12.1. Также в публикации [von zur Gathen и Gerhard (1999)], §16.2, дается обработка алгоритма в более высоких размерностях.

Пусть $p = v_1 \cdot v_2 / ||v_1||^2$, и учтем следующее отношение:

$$\lfloor p \rceil = \lfloor p + 1/2 \rfloor = m \in \mathbb{Z} \Leftrightarrow p \in [m - 1/2, m + 1/2) \subseteq \mathbb{R},$$

где, следуя стандартной терминологии исчисления, множество $[a, b)$ для $a, b \in \mathbb{R}$ обозначает множество всех $x \in \mathbb{R}$ таких, что $a \leqslant x < b$.

Теперь мы даем доказательство завершения алгоритма. Предположим, что $|p| = \frac{1}{2}$. Если $p = -\frac{1}{2}$, то $m = 0$, и алгоритм останавливается. Если $p = \frac{1}{2}$, то $m = 1$, что означает, что мы проходим по циклу еще раз с $v_1' = v_1$ и $||v_2'|| = ||v_2 - v_1|| = ||v_2||$, и, что более важно, в следующем раунде $p = -\frac{1}{2}$, и снова алгоритм завершается.

Если $p = m$, то есть изначально p было целым числом (давая $\vec{CE} = \vec{D'E} = \vec{DE}$ на рис. 7.1), то просто по теореме Пифагора $||\vec{CE}||$ должно быть короче $||AE||$ (поскольку v_1, v_2 – ненулевые, так как $m \neq 0$).

Поэтому мы можем принять, что $|p| \neq \frac{1}{2}$ и $p \neq m$. Два случая, когда $m < p$, давая D', либо $m > p$, давая D, являются симметричными, и поэтому мы рассматриваем только последний случай. Должно быть $|p| > \frac{1}{2}$, иначе m было бы нулем, приводя

к завершению. Обратите внимание, что $||\vec{CD}|| \leqslant \frac{1}{2})||\vec{AB}||$, потому что $\vec{AD} = m\vec{AB}$. Из этого и теоремы Пифагора мы знаем, что:

$$||\vec{AE}|| = ||\vec{AC}||^2 + ||\vec{CE}||^2 = p^2||\vec{AB}||^2 + ||\vec{CE}||^2;$$
$$||\vec{DE}||^2 = ||\vec{CD}||^2 + ||\vec{CE}||^2 \leqslant p^2||\vec{AB}||^2 + ||\vec{CE}||^2,$$

и поэтому $||\vec{AE}||^2 - ||\vec{DE}||^2 \geqslant (p^2 - \frac{1}{4})||\vec{AB}||^2$, и, как мы уже отмечали, если алгоритм не заканчивается в строке 6, это означает, что $|p| > \frac{1}{2}$, и поэтому отсюда следует, что $||\vec{AE}|| > ||\vec{DE}||$, то есть v_2 длиннее, чем $v_2 - mv_1$, и поэтому новый v_2 (строка 9) короче, чем старый.

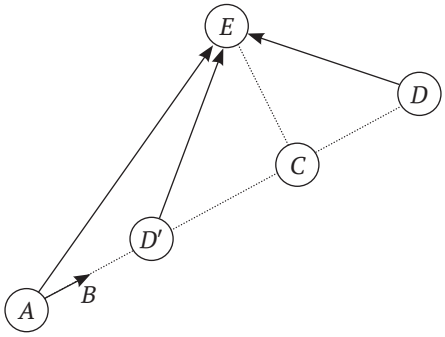


Рис. 7.1 ❖ Проекция v_2, заданного как $\vec{AE}$, в v_1, заданный как $\vec{AB}$. Результирующий вектор равен $\vec{AC} = v_2 - pv_1$, где $p = v_1 \cdot v_2/||v_1||^2$. Принимая $m = p$, вектор $v_2 - mv_1$, задаваемый соответственно $\vec{D'E}$ или $\vec{DE}$ в зависимости от того, истинно ли, что $m < p$, или нет. Разумеется, $D' = C = D$ при $p = m$

Пусть v'_1, v'_2 равны двум векторам, приводящим к одной итерации цикла из v_1, v_2. Как мы уже отмечали выше, при $|p| = \frac{1}{2}$ алгоритм завершается за один или два шага. В противном случае $||v'_1|| + ||v'_2|| < ||v_1|| + ||v_2||$, и так как в решетке существует конечное число пар точек, ограниченных суммой двух норм исходных векторов, и алгоритм заканчивается, когда один из векторов становится нулевым, эта процедура должна заканчиваться за конечное число шагов.

7.7. Примечания

Эта глава основана на нескольких статьях автора. Разделы 7.2 и 7.2.1, гауссово исключение и доказательство его правильности основаны на разделе 3.1 в работе [Soltys (2002b)]. Раздел 7.5 основан на серии работ, в которых автор искал возможные доказательства основных свойств характеристического многочлена (таких свойств, как то, что характеристический многочлен матрицы также является ее аннигилятором и что постоянный член является определителем матрицы). В этой области исследований было изучено несколько алгоритмов: алгоритм Чанки, раздел 7.5.1, основан на работе [Soltys (2005)], а алгоритм Берковица, раздел 7.5.2, основан на работе [Soltys (2002a)]. Оригинальное представление алгоритма Берковица можно найти в работе [Berkowitz (1984)].

Глава 8

Вычислительные основы

Технология состоит в изготовлении метафор из мира природы. Полет – это метафора воздуха, колеса – метафора воды, еда – метафора земли. Метафора огня – электричество.

Э. Л. Доктороу [Doctorow (1971)], стр. 224

8.1. Введение

Первая серьезная попытка построить компьютер была предпринята в 1820-х годах Чарльзом Бэббиджем. Машина называлась *разностной машиной*, вычислялась с помощью десятичной системы счисления и приводилась в действие с помощью рукоятки. Увы, Бэббиджу так и не удалось построить готовое изделие, поскольку изготовление прецизионных деталей было колоссальной инженерной проблемой, учитывая технологию его времени.

Компьютерные программы – это не что иное, как реализации алгоритмов на выбранном языке программирования. Программы работают на аппаратном обеспечении, и подобно тому, как программы являются инстанциациями алгоритмов, аппаратное обеспечение является материальным воплощением конкретной вычислительной модели. В этой главе мы рассмотрим различные модели вычислений, которые затем инстанциируются в машине, работающей на электричестве. Мы введем несколько типов конечных автоматов и завершим презентацией машины Тьюринга.

8.2. Алфавиты, строки и язык

Алфавит – это конечное, непустое множество разных символов, обычно обозначаемых знаком $\sum$. Например, $\sum = \{0, 1\}$, обычный двоичный алфавит, или $\sum = \{a, b, c, \cdots, z\}$, обычные буквы нижнего регистра английского алфавита. *Строка*, или символьная цепочка, также именуемая *словом*, представляет собой конечную упорядоченную последовательность символов, выбранных из некоторого алфавита. Например, 01001110101011 – это строка над двухзначным алфавитом. Форма записи $|w|$ обозначает *длину символьной цепочки* (строки) w, например $|010011101011| = 12$. *Пустая строка* ε – это уникальная строка такая, что $|\varepsilon| = 0$. Мы иногда пишем $\sum_\varepsilon$, чтобы подчеркнуть, что $\varepsilon \in \sum$. $\sum^k$ является множеством строк над $\sum$ длиной ровно k, например если $\sum = \{0, 1\}$, то:

$\sum_0 = \{\varepsilon\}$,

$\sum_1 = \sum$,
$\sum_2 = \{00, 01, 10, 11\}$.

Множество $\sum^*$ называется *звездой Клини* алфавита $\sum$ и является множеством всех строк над $\sum$. Обратите внимание, что $\sum^* = \sum^0 \cup \sum^1 \cup \sum^2 \cup \ldots$, тогда как $\sum^+ = \sum^1 \cup \sum^2 \cup \ldots$. Если x, y являются строками, и $x = a_1a_2 \ldots a_m$, и $y = b_1b_2 \ldots b_n$, тогда их *конкатенацией* является их *смежное расположение*, то есть $x \cdot y = a_1a_2 \ldots a_mb_1b_2 \ldots b_n$. Мы часто пишем xy вместо $x \cdot y$, и $w\varepsilon = \varepsilon w = w$. *Язык* L представляет собой коллекцию строк над некоторым алфавитом $\sum$, то есть $L \subseteq \sum^*$. Например:

$$L = \{\varepsilon, 01, 0011, 000111, \ldots\} = \{0^n1^n | n \geqslant 0\}. \quad (8.1)$$

Обратите внимание, что $\{\varepsilon\} \neq \emptyset$, один из которых является языком, состоящим из единственной строки ε, а другой является пустым языком.

Обозначим через $\sum_\ell$ обобщенный алфавит размера ℓ. Например, пусть $\sum_1 = \{1\}$, $\sum_2 = \{0, 1\}$ и т. д.

Задача 8.1. Каков размер $\sum_2^k$? Каков размер $\sum_\ell^k$? Пусть L равно множеству строк над $\sum$, где ни один символ не может встречаться более одного раза; какова $|L|$?

Пусть $w = w_1w_2 \ldots w_n$, где для каждого i, $w_i \in \sum$. Для того чтобы подчеркнуть структуру массива w, мы иногда представляем его как $w[1 \ldots n]$. Мы говорим, что v является *подсловом* w, если $v = w_iw_{i+1} \ldots w_j$, где $i \leqslant j$. Если $i = j$, то v является единственным символом в w; если $i = 1$ и $j = n$, то $v = w$; если $i = 1$, то v является *префиксом* w (иногда обозначается $v \sqsubseteq w$), и если $j = n$, то v является *суффиксом* w (иногда обозначается как $w \sqsupseteq v$). Мы можем выразить лаконичнее, что v является подсловом, следующим образом: $v = w[i \ldots j]$, и когда разделители не нужно выражать явно, мы используем форму записи $v \leqslant w$. Мы говорим, что v является *подпоследовательностью* w, если $v = w_{i_1}w_{i_2} \ldots w_{i_k}$, для $i_1 < i_2 < \ldots < i_k$.

8.3. Регулярные языки

В этой главе мы рассмотрим разные типы языков, то есть разные типы множеств строк. Мы классифицируем их в соответствии с вычислительными моделями, которые их описывают. Регулярные языки в некотором смысле являются простейшими языками, поскольку они описываются вычислителями без памяти, так называемыми конечными автоматами. Неудивительно, что только некоторые языки являются регулярными, и для описания более сложных языков нам потребуются более прочные модели вычислений, такие как магазинные автоматы (раздел 8.4.2) или машины Тьюринга (раздел 8.5).

8.3.1. Детерминированный конечный автомат

Детерминированный конечный автомат (deterministic finite automaton, DFA) – это модель вычисления, заданная кортежем $A = (Q, \sum, \delta, q_0, F)$, где:

1) Q – это *конечное множество* состояний;
2) $\sum$ – это *алфавит*, то есть конечное множество входных символов;
3) δ: $Q \times \sum \to Q$ – это *переходная функция*, то есть «программа», которая исполняет детерминированный конечный автомат. При условии что $q \in Q$, $a \in \sum$, δ вычисляет следующее состояние $\delta(q, a) = p \in Q$;

4) q_0 – это *исходное состояние*, также именуемое *начальным состоянием* (q_0);
5) F – это множество *финальных* или *принимающих* состояний.

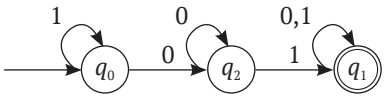
Для того чтобы увидеть, что A принимает строку w, мы «выполняем» A на $w = a_1a_2 \ldots a_n$ следующим образом: $\delta(q_0, a_1) = q_1$, $\delta(q_1, a_2) = q_2$ до тех пор, пока не будет $\delta(q_{n-1}, a_n) = q_n$. Мы говорим, что A *принимает* w тогда и только тогда, когда $q_n \in F$, то есть если q_n является одним из финальных (принимающих) состояний. Точнее: A принимает w, если существует последовательность состояний $r_0, r_1, \ldots, r_n$, где $n = |w|$ такое, что $r_0 = q_0$, $\delta(r_i, w_{i+1}) = r_{i+1}$, где $i = 0, 1, \ldots, n - 1$ и w_j является j-м символом w, и $r_n \in F$. В противном случае мы говорим, что A отклоняет w.

Например, рассмотрим язык

$$L_{01} = \{w \mid w \text{ имеет форму } x01y \in \sum^*\},$$

который является множеством строк, имеющих 01 в качестве подстроки. Поэтому $111 \notin L_{01}$, но $001 \in L_{01}$.

Предположим, что мы хотим спроектировать детерминированный конечный автомат $A = (Q, \sum, \delta, q_0, F)$ для L_{01}. То есть A принимает строки в L_{01} и отклоняет строки не в L_{01}. Пусть $E = \{0, 1\}$, $Q = \{q_0, q_1, q_2\}$ и $F = \{q_1\}$. Существует два способа представить δ: в форме *диаграммы переходов* либо в форме *таблицы переходов*; см. рис. 8.1.



	0	1
q_0	q_2	q_0
q_1	q_1	q_1
q_2	q_2	q_1

Рис. 8.1 ❖ Детерминированный конечный автомат, принимающий $L = \{w \mid w, \text{ имеет форму } x01y \in \sum^*\}$. Слева приведена диаграмма переходов, а справа – таблица переходов

На данный момент мы знаем, что простого представления A в качестве кандидатного детерминированного конечного автомата для L_{01} недостаточно. Мы также должны доказать, что A является правильным. Это будет проще, как только позже в этом разделе мы определим *расширенную переходную функцию*, но пока будет достаточно простого аргумента по индукции на длине $w \in \sum^*$.

Задача 8.2. Докажите, что A является правильным детерминированным конечным автоматом для L_{01}.

Задача 8.3. Спроектируйте детерминированный конечный автомат для $\{w: |w| \geqslant 3$, и его третий символ равен 0$\}$.

Задача 8.4. Спроектируйте детерминированный конечный автомат для $\{w$: каждая нечетная позиция w равна 1$\}$.

Задача 8.5. Рассмотрим следующие два языка:

$B_n = \{a^k = \underbrace{aa\cdots a}_{k} : k \text{ является кратным } n\} \subseteq \{a\}^*$;

$C_n = \{(w)_b \in \{0, 1\}^* : w \text{ делится на } n\}$.

Обратите внимание, что $(w)_b$ является двоичным представлением числа $w \in \mathbb{N}$. Каковы их детерминированные конечные автоматы?

Задача 8.6. Рассмотрим торговый автомат, который на входе принимает монеты, где разрешенные монеты составляют следующий алфавит символов:

(1), (5), (10), (25).

Естественно, строка представляет собой просто упорядоченную последовательность монет. Сконструируйте переходную функцию для торгового автомата, который принимает любую последовательность монет, где итоговая стоимость монет суммируется в кратное 25.

При наличии переходной функции δ ее расширенная переходная функция, обозначаемая как $\hat{\delta}$, определяется индуктивно. Базовый случай: $\hat{\delta}(q, \varepsilon) = q$, и индукционный шаг: если $w = xa$, $w, x \in \sum^*$ и $a \in \sum$, то

$$\hat{\delta}(q, w) = \hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a).$$

Таким образом, $\hat{\delta} : Q \times \sum^* \to Q$, и $w \in L(A) \Leftrightarrow \hat{\delta}(q_0, w) \in F$. Здесь $L(A)$ – это множество всех тех строк (и только тех), которые принимаются A, именуемых *языком* A.

Теперь мы можем определить язык детерминированного конечного автомата A равным

$$L(A) = \{w | \hat{\delta}(q_0, w) \in F\},$$

и мы можем сказать, что язык L является *регулярным*, если существует детерминированный конечный автомат A такой, что $L = L(A)$. Возникает следующий естественный вопрос: какие операции на языках сохраняют их регулярность? Регулярные языки хорошо себя ведут, и многие естественные операции сохраняют их регулярность; мы начинаем с трех основных, которые называются *регулярными операциями*:

i) объединение: $L \cup M = \{w | w \in L$ или $w \in M\}$;
ii) конкатенации: $LM = \{xy | x \in L$ и $y \in M\}$;
iii) звезда Клини (или замыкание Клини):

$$L^* = \{w | w = x_1 x_2 \ldots x_n \text{ и } x_i \in L\}.$$

Мы уже ввели звезду Клини в контексте алфавитов (раздел 8.2), где алфавиты можно рассматривать как особый язык (из строк длиной один). Но есть важная разница в том, как звезда Клини действует на следующих двух: обратите внимание, что $\sum^+ = \sum^* - \{\varepsilon\}$, но это не является истинным в общем случае для языков, которые $L^+ = L^* - \{\varepsilon\}$.

Задача 8.7. Почему $L^+ = L^* - \{\varepsilon\}$ не обязательно является истинным?

Теорема 8.8. Регулярные языки замкнуты при регулярных операциях (объединении, конкатенации и звезде Клини).

Доказательство. Предположим, что у нас есть два регулярных языка, A, B, и поэтому у них есть соответствующие детерминированные конечные автоматы, M_1, M_2. Рассмотрим объединение $A \cup B$: возьмем соответствующие детерминирован-

ные конечные автоматы M_1 и M_2; пусть M равно такому, что $Q_M = Q_{M_1} \times Q_{M_2}$, то есть декартовому произведению двух множеств состояний. Пусть:

$\delta_M((r_1, r_2), a) = (\delta_{M_1}(r_1, a), \delta_{M_2}(r_2, a))$.

Для конкатенации и звезды нам нужно понятие «недетерминизма», которое мы введем в следующем разделе – см. задачу 8.13. □

Ключевой идеей в доказательстве теоремы 8.8 является расширение понятия состояния. Множество состояний представляет собой по-настоящему конечное множество «описателей» различных ситуаций. Эти описатели могут быть буквально чем угодно, например множествами состояний из других машин – как мы увидим, когда далее введем недетерминированные конечные автоматы.

8.3.2. Недетерминированные конечные автоматы

Недетерминированный конечный автомат (nondeterministic finite automaton, NFA) определяется аналогично детерминированному конечному автомату, за исключением того, что переходная функция δ становится *переходным отношением*. Тем самым $\delta \subseteq Q \times \sum \times Q$, то есть на одной и той же паре (q, a) может существовать более одного возможного нового состояния (или ни одного). Эквивалентным образом мы можем смотреть на δ как на $\delta : Q \times \sum \to \mathcal{P}(Q)$, где $\mathcal{P}(Q)$ – это степенное множество множества Q.

Недетерминированные конечные автоматы похожи на детерминированные конечные автоматы, однако, в отличие от последних, они допускают «ветвление». Это означает, что в конкретной конфигурации, где детерминированный конечный автомат находится в одном состоянии, недетерминированный конечный автомат может находиться в нескольких (или в одном, или в никаком). Хорошей его аналогией является механизм ветвления в языке программирования *C*. Поскольку недетерминированный конечный автомат может находиться в нескольких состояниях одновременно, он допускает определенную степень параллелизма.

Например, рассмотрим $L_n = \{w |$ n-й символ с конца равен 1$\}$. Недетерминированный конечный автомат для L_n приведен на рис. 8.2.

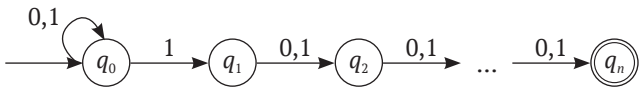


Рис. 8.2 ❖ Недетерминированный конечный автомат для $L_n = \{w |$ n-й символ с конца равен 1$\}$

Задача 8.9. По меньшей мере сколько состояний требуется любому детерминированному конечному автомату, который способен распознавать L_n?

Определение термина «принятие» изменяется незначительно: N принимает w, если $w = y_1 y_2 \ldots y_m$, где $y_i \in \sum_\varepsilon$ с целью, чтобы существовала последовательность состояний $r_0, r_1, \ldots, r_m$ таких, что $r_0 = q_0$, и $r_{i+1} \in \delta(r_i, y_{i+1})$ для $i = 0, 1, \ldots, m - 1$ и $r_m \in F$. То есть w принимается, если существует *заполнение* строки w строками ε, для которого существует принимающая последовательность состояний.

Задача 8.10. При заполнении строки строками ε нам вовсе не нужен сплошной отрезок строк ε длиннее, чем число состояний. Другими словами, если заполне-

ние существует, то его можно найти за конечное число шагов. Объясните, почему, и ограничьте время поиска рабочего заполнения.

Как показано на рис. 8.3, переходы по ε очень удобны при проектировании недетерминированных конечных автоматов.

В недетерминированном конечном автомате на рис. 8.3 мы не хотим иметь точку саму по себе, то есть «.» не является надлежаще сформированным числом в десятичной форме записи; мы хотим, чтобы цифры выступали в качестве либо префикса, либо суффикса; мы могли бы уточнить его дальше, запретив сегменту нулей быть префиксом и т. д. В итоге недетерминированный конечный автомат в данном контексте точно описывает, что мы имеем в виду под числом, и тогда это определение становится способом, которым данный синтаксический анализатор, или парсер, придает значение исходному коду.

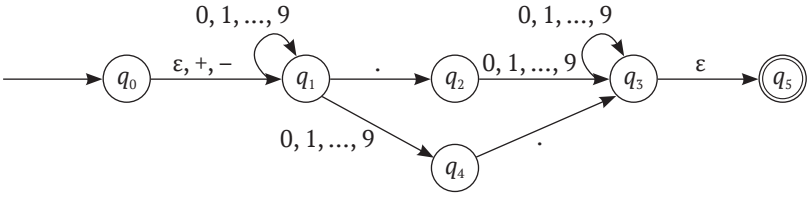


Рис. 8.3 ❖ Недетерминированный конечный автомат для множества десятичных чисел

Для того чтобы определить понятие расширенной переходной функции для недетерминированных конечных автоматов, то есть $\hat{\delta}$, нам нужно понятие *ε-замыкания*. С учетом q ε-close(q) равно множеству всех состояний p, которые достижимы из q, следуя по стрелкам, помеченным символом ε. Формально $q \in$ ε-close(q), и если $p \in$ ε-close(q) и $p \xrightarrow{\varepsilon} r$, то $r \in$ ε-close(q).

Теперь мы можем определить *расширенное переходное отношение* для недетерминированных конечных автоматов следующим образом: $\hat{\delta}(q, \varepsilon) = $ ε-close(q); предположим, что $w = xa$ и $\hat{\delta}(q, x) = \{p_1, p_2, \ldots, p_n\}$, и, более того, $\cup_{i=1}^{n} \delta(p_i, a) = \{r_1, r_2, \ldots, r_m\}$. Тогда

$$\hat{\delta}(q, w) = \bigcup_{i=1}^{m} \varepsilon\text{-close}(r_i).$$

Теорема 8.11. Детерминированный и недетерминированный конечные автоматы являются эквивалентными.

Доказательство. Очевидно, что детерминированные конечные автоматы – это особый случай недетерминированных конечных автоматов. Таким образом, мы должны показать, что каждый недетерминированный конечный автомат может быть перепроектирован как детерминированный конечный автомат. С этой целью мы используем прием, именуемый *построением подмножеств*.

Сначала мы принимаем допущение, что недетерминированный конечный автомат N не имеет ε-переходов. Пусть M равно соответствующему детерминированному конечному автомату, и $Q_M = \mathcal{P}(Q_N)$, где $\mathcal{P}(Q_N)$ – это степенное множество Q_N, означающее, что оно состоит из всех возможных подмножеств множеств Q_N, и

$$\delta_M(Q, a) := \bigcup_{q \in Q} \delta_N(q, a),$$

где $Q \in \mathcal{P}(Q_N)$, и пусть

$$F_M = \{Q \in \mathcal{P}(Q_N) : Q \cap F_N \neq \emptyset\}.$$

Наконец, пусть $(q_M)_0 := \{(q_N)_0\}$. Если N имеет ε-переходы, то

$$\delta_M(Q, a) := \bigcup_{q \in Q} \varepsilon\text{-closure}(\delta_N(q, a)).$$

Отметим, что построение подмножеств имеет стоимость: поскольку $|\mathcal{P}(Q_N)| = 2^{|Q_N|}$, мы видим, что происходит экспоненциальный рост состояний. Чего-то подобного следовало ожидать, поскольку мы симулируем более выразительную модель вычисления (недетерминированный конечный автомат) с помощью более ограниченной (детерминированный конечный автомат). □

Покажем пример преобразования из недетерминированного конечного автомата для L_2 (L_2 – это множество строк, где предпоследний символ равен 0), приведенного на рис. 8.4, в соответствующий детерминированный конечный автомат, приведенный на рис. 8.5.

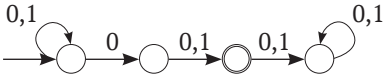


Рис. 8.4 ❖ Недетерминированный конечный автомат для L_2

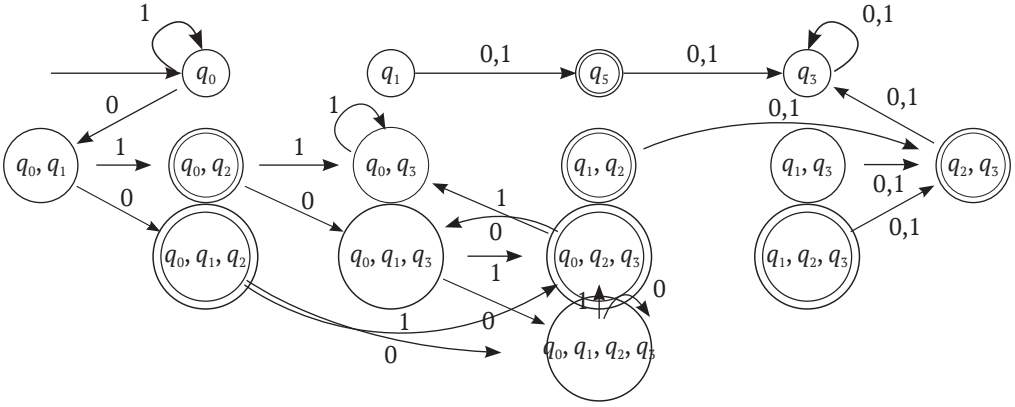


Рис. 8.5 ❖ Детерминированный конечный автомат для L_2

На диаграмме преобразования недетерминированного конечного автомата в детерминированный (рис. 8.4 и 8.5) можно заметить одну вещь: сгенерирована целая куча состояний вместе с соответствующими стрелками – но они не нужны! Это те состояния, которые расположены в правой части диаграммы, что просто расточительно. Вместо этого начнем генерировать состояния и соединения из $\{q_0\}$. Соединяем и генерируем только те состояния, которые необходимы для переходов; остальное игнорируем. Иногда это не помогает, и нужны все состояния.

Следствие 8.12. Язык является регулярным ⇔ он распознается неким детерминированным конечным автоматом ⇔ он распознается неким недетерминированным конечным автоматом.

Задача 8.13. Закончите доказательство теоремы 8.8, то есть покажите, что операции конкатенации и звезды сохраняют регулярность.

8.3.3. Регулярные выражения

Регулярные выражения (regular expression, RE) знакомы всем, кто использует компьютер. Они являются средством поиска закономерностей в тексте. Автор книги является заядлым пользователем текстового редактора VIM[1], и трудно найти редактор с более универсальной функцией сопоставления и замены по шаблону. Например, команда

```
:23,43s/\(.*\n\)\{3\}/&\r/
```

вставляет пустую строку через каждую третью строку между строками 23 и 43 (включительно). Фактически VIM, как и большинство текстовых процессоров, реализует набор команд, выходящих далеко за рамки использования только регулярных выражений.

Регулярное выражение – это синтаксический объект, предназначенный для выражения множества строк, то есть языка. В этом смысле регулярные выражения представляют собой модель вычислений, подобную детерминированным или недетерминированным конечным автоматам. Они определяются формально по структурной индукции. В базовом случае: $a \in \sum, \varepsilon, \emptyset$. На индукционном шаге: если E, F являются регулярными выражениями, то $E + F$, EF, $(E)^*$, (E).

Используя свое интуитивное понимание регулярного выражения, вы должны быть в состоянии решить задачу 8.14.

Задача 8.14. Каковы $L(a)$, $L(\varepsilon)$, $L(\emptyset)$, $L(E + F)$, $L(EF)$, $L(E^*)$? Эта задача просит вас определить семантику регулярного выражения.

Задача 8.15. Дайте регулярное выражение для множества строк из нулей и единиц, не содержащих 101 в качестве подстроки.

Теорема 8.16. Язык является регулярным тогда и только тогда, когда он задан некоторым регулярным выражением.

Мы собираемся доказать теорему 8.16 в двух частях. Сначала предположим, что мы хотим преобразовать регулярное выражение R в недетерминированный конечный автомат A. С этой целью мы используем структурную индукцию, и на каждом шаге конструирования мы гарантируем, что недетерминированный конечный автомат A обладает следующими тремя свойствами (то есть инвариантами конструкции): (i) ровно одно принимающее состояние; (ii) ни одной стрелки в исходное состояние; (iii) ни одной стрелки из принимающего состояния.

Мы следуем договоренности, что если из состояния на определенном символе стрелки нет, то вычисление *отклоняет*. Формально мы можем ввести «мусорное состояние» T, которое представляет собой отклоняющее состояние с петлей на

[1] См. http://www.vim.org.

всех символах в $\sum$, и представить, что существует стрелка на $\sigma \in \sum$ из состояния q в T, если не было стрелки на σ из q. Базовый случай: регулярное выражение R имеет форму: ε, $\emptyset$, $a \in \sum$. В этом случае недетерминированный конечный автомат имеет три соответствующие формы, изображенные на рис. 8.6.

Рис. 8.6 ❖ Недетерминированные конечные автоматы ε, $\emptyset$, a

На индукционном шаге мы строим бо́льшие регулярные выражения из меньших тремя возможными способами: $R + S$, RS, R^*. Соответствующие недетерминированные конечные автоматы строятся в указанном порядке следующим образом:

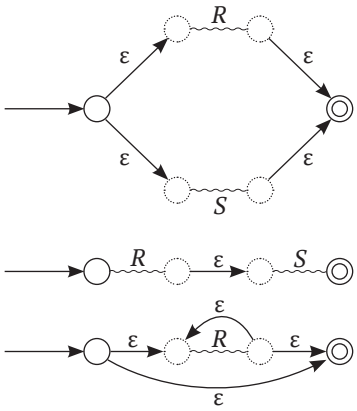
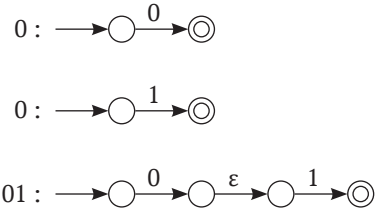
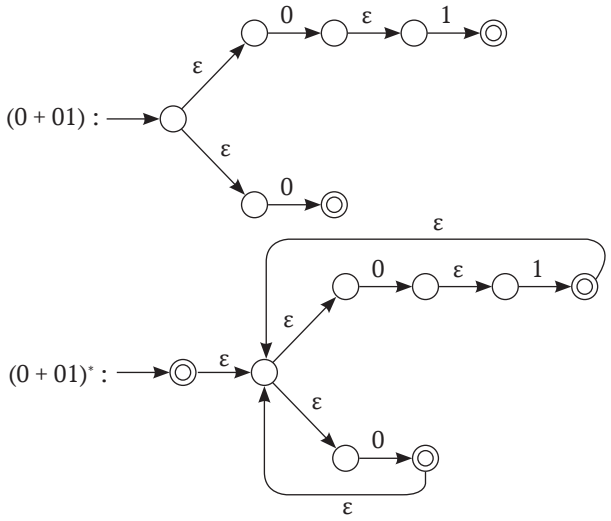


Рис. 8.7 ❖ Недетерминированные конечные автоматы $R + S$, RS, R^*. Мы используем пунктирные круги для обозначения начального и финального состояний предыдущего недетерминированного конечного автомата и волнистую линию для обозначения всех других его состояний

В качестве примера мы преобразовываем регулярное выражение $(0 + 01)^*$ в недетерминированный конечный автомат с помощью приведенной ниже процедуры.



Теперь мы докажем другое направление теоремы 8.16: при наличии недетерминированного конечного автомата мы сконструируем соответствующее регулярное выражение. Мы представляем два способа выполнения этого построения.



8.3.3.1. Метод 1

Этот метод является отличным примером динамического программирования, который мы рассмотрели в главе 4. Предположим, что A имеет n состояний, и пусть $R_{ij}^{(k)}$ обозначает регулярное выражение, языком которого является множество строк w такое, что: w переводит A из состояния q_i в состояние q_j, где все промежуточные состояния имеют свой индекс $\leqslant k$. Тогда R такое, что $L(R) = L(A)$, задается следующим выражением:

$$R := R_{1j_1}^{(n)} + R_{1j_2}^{(n)} + \cdots + R_{1j_k}^{(n)},$$

где $F = \{j_1, j_2, \ldots, j_k\}$. Поэтому теперь мы строим $R_{ij}^{(k)}$ по индукции на k. Для базового случая пусть $k = 0$ и $R_{ij}^{(0)} = x + a_1 + a_2 + \cdots + a_k$, где $i \xrightarrow{a_l} j$ и $x = \emptyset$, если $i \neq j$ и $x = \varepsilon$, если $i = j$. На индукционном шаге $k > 0$, и

$$R_{ij}^{(k)} = \underbrace{R_{ij}^{(k-1)}}_{\text{Путь не посещает } k} + \underbrace{R_{ik}^{(k-1)}(R_{kk}^{(k-1)})^* R_{kj}^{(k-1)}}_{\substack{\text{Посещает } k \\ \text{не более одного раза}}}.$$

Очевидно, что этот процесс создает соответствующее R с нуля.

В качестве примера мы преобразуем детерминированный конечный автомат, который принимает только те строки, которые имеют 00 в качестве подстроки. Такой детерминированный конечный автомат приведен на рис. 8.8.

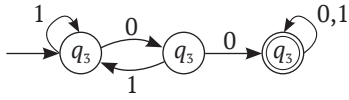


Рис. 8.8 ❖ Детерминированный конечный автомат, который принимает только те строки, которые имеют 00 в качестве подстроки

Тогда:

$R_{11}^{(0)} = \varepsilon + 1$

$R_{12}^{(0)} = R_{23}^{(0)} = 0$

$R_{13}^{(0)} = R_{31}^{(0)} = R_{32}^{(0)} = \emptyset$

$R_{21}^{(0)} = 1$

$R_{22}^{(0)} = \varepsilon$

$R_{33}^{(0)} = \varepsilon + 0 + 1$

Задача 8.17. Завершите построение, вычислив $R^{(1)}$, $R^{(2)}$, $R^{(3)}$ и, наконец, R.

8.3.3.2. Метод 2

Мы преобразовываем детерминированный конечный автомат в регулярное выражение, сначала преобразуя его в обобщенный недетерминированный конечный автомат, то есть недетерминированный конечный автомат, который позволяет использовать регулярные выражения в качестве меток своих стрелок. Мы формально определяем *обобщенный недетерминированный конечный автомат* следующим образом:

$$\delta : (Q - \{q_{\text{accept}}\}) \times (Q - \{q_{\text{start}}\}) \to \mathcal{R},$$

где состояния «начать» и «принять» являются уникальными.

Мы говорим, что G принимает $w = w_1 w_2 \ldots w_n$, $w_i \in \Sigma^*$, если существует последовательность состояний $q_0 = q_{\text{start}}, q_1, \ldots, q_n = q_{\text{accept}}$ такая, что для всех i, $w_i \in L(R_i)$, где $R_i = \delta(q_{i-1}, q_i)$.

Во время транслирования из детерминированного конечного автомата в обобщенный недетерминированный, если нет стрелки $i \to j$, то мы помечаем это знаком $\emptyset$. Для каждого i мы обозначаем петлю знаком ε. Теперь мы исключаем состояния из G до тех пор, пока не останется только $q_{\text{start}} \xrightarrow{R} q_{\text{accept}}$. Исключение состояний осуществляется, как показано на рис. 8.9.

На этом заканчивается доказательство теоремы 8.16.

8.3.4. Алгебраические законы для регулярных выражений

Регулярные выражения подчиняются ряду алгебраических законов; эти законы могут быть использованы для упрощения регулярного выражения либо для переформулирования регулярного выражения по-другому.

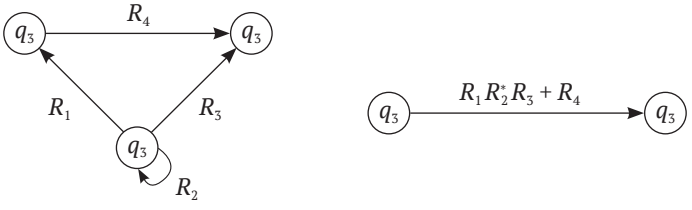


Рис. 8.9 ❖ Шаг в редукции состояний

Закон	Описание
$R + P = P + R$	Коммутативность +
$(R + P) + Q = R + (P + Q)$	Ассоциативность +
$(RP)Q = R(PQ)$	Ассоциативность конкатенации
$\emptyset + R = R + \emptyset = R$	$\emptyset$-тождество для +
$\varepsilon R = R\varepsilon = R$	ε-тождество для конкатенации
$\emptyset R = R\emptyset = \emptyset$	$\emptyset$-аннигилятор для конкатенации
$R(P + Q) = RP + RQ$	Левая дистрибутивность
$(P + Q)R = PR + QR$	Правая дистрибутивность
$R + R = R$	Идемпотентный закон для объединения

Обратите внимание, что коммутативность конкатенации, $RP = PR$, явно отсутствует, поскольку для регулярного выражения она не верна вообще; действительно, в качестве строк $ab \neq ba$. Вот еще шесть законов, связанных со звездой Клини:

$(R^*)^* = R^*$; $\quad \emptyset^* = \varepsilon$; $\quad \varepsilon^* = \varepsilon$;
$R^+ = RR^* = R^*R$; $\quad R^* = R^+ + \varepsilon$; $\quad (R + P)^* = (R^*P^*)^*$.

Обратите внимание, что $R^* = R^+ + \varepsilon$ не означает, что $L(R^+) = L(R^*) - \{\varepsilon\}$.

Теперь вопрос в том, как мы можем проверить, является ли данное формальное суждение валидным алгебраическим законом? Ответ будет захватывающим, потому что он противоречит всему, чему мы научились в математике: мы можем увидеть, что заявленный закон валиден, проверив его на конкретном примере. Следовательно, мы можем проверить универсальное формальное суждение с помощью единственного экземпляра. Другими словами, для того чтобы проверить, является ли $E = F$, где E, F – это регулярные выражения с переменными (R, P, Q, ···), следует конвертировать E, F в конкретные регулярные выражения C, D, заменив переменные символами. Затем проверить, истинно ли, что $L(C) = L(D)$, и если да, то мы можем заключить, что $E = F$. Например, для того чтобы показать, что $(R + P)^* = (R^*P^*)^*$, мы заменяем R, P на a, $b \in \sum$, получив $(a + b)^* = (a^*b^*)^*$, и мы проверяем, является ли этот конкретный экземпляр истинным. Он является истинным, и поэтому мы можем заключить, что $(R + P)^* = (R^*P^*)^*$ является истинным. Данное свойство часто называют «тестом на алгебраические законы для регулярных выражений».

8.3.5. Свойства замыкания в регулярных языках

Перечислим операции на языках, которые сохраняют регулярность. Отметим, что первые три операции были представлены в теореме 8.8.

1. **Объединение:** если L, M являются регулярными, то и $L \cup M$ тоже.
2. **Конкатенация:** если L, M являются регулярными, то и $L \cdot M$ тоже.
3. **Звезда Клини:** если L является регулярным, то и L^* тоже.
4. **Дополнение:** если L является регулярным, то и $L_c = \sum^* - L$ тоже.
5. **Пересечение:** если L, M являются регулярными, то и $L \cap M$ тоже.
6. **Реверс:** если L является регулярным, то и $L^R = \{w^R | w \in L\}$ тоже, где $(w_1w_2 \ldots w_n)^R = w_nw_{n-1} \ldots w_1$.
7. **Гомоморфизм:** $h : \sum^* \to \sum^*$, где

   $h(w) = h(w_1w_2 \ldots w_n) = h(w_1)h(w_2) \ldots h(w_n)$.

   Например, $h(0) = ab$, $h(1) = \varepsilon$, тогда $h(0011) = abab$. $h(L) = \{h(w) | w \in L\}$. Если L является регулярным, то и $H(L)$ тоже.

8. **Обратный гомоморфизм:** $h^{-1}(L) = \{w|h(w) \in L\}$. Пусть A равно детерминированному конечному автомату для L; строим детерминированный конечный автомат для $h^{-1}(L)$ следующим образом: $\delta(q, a) = \hat{\delta}_A(q, h(a))$.
9. **Ненадлежащий префикс:** если A является регулярным, то и язык тоже.

   NOPREFIX(A) = {$w \in A$: ненадлежащий префикс строки w находится в A}.

10. **Не расширяется:** если A является регулярным, то и язык тоже.

    NOEXTEND(A) = {$w \in A$: w является ненадлежащим префиксом любой строки в A}.

Задача 8.18. Покажите, что вышеуказанные операции сохраняют регулярность.

8.3.6. Сложность преобразований и принятия решений

В этом разделе мы прорезюмируем сложность, то есть наиболее известный алгоритм для трансформаций между разными формализациями регулярных языков. Мы будем использовать форму записи $A \hookrightarrow B$ для обозначения преобразования из формализма A в формализм B.

1. Недетерминированный конечный автомат ↪ детерминированный конечный автомат: $O(n^3 2^n)$.
2. Детерминированный конечный автомат ↪ недетерминированный конечный автомат: $O(n)$.
3. Детерминированный конечный автомат ↪ регулярное выражение: $O(n^3 4^n)$.
4. Регулярное выражение ↪ недетерминированный конечный автомат: $O(n)$.

Задача 8.19. Обоснуйте сложности для каждого приведенного выше преобразования.

Теперь рассмотрим следующие свойства принятия решений для регулярных языков:

1. Является ли данный язык пустым?
2. Находится ли данная строка в данном языке?
3. Являются ли два данных языка фактически одним и тем же языком?

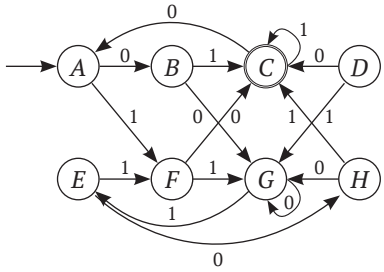
Задача 8.20. В чем сложность трех указанных задач принятия решения? Обратите внимание, что в каждом случае необходимо уточнить, что означает «данный». То есть каким образом данный язык «задается».

8.3.7. Эквивалентность и минимизация автоматов

Мы часто заинтересованы в поиске минимального детерминированного конечного автомата для данного языка. Мы говорим, что два состояния эквивалентны, если для всех строк w, $\hat{\delta}(p, w)$ принимает $\Leftrightarrow$ $\hat{\delta}(q, w)$ принимает. Если два состояния не эквивалентны, они различимы.

У нас есть рекурсивная процедура (разделяй и властвуй) для нахождения пар различимых состояний. Сначала если p принимает и q нет, то $\{p, q\}$ является парой различимых состояний. Это «нижний» случай рекурсии. Если $r = \delta(p, a)$ и $s = \delta(q, a)$, где $a \in \sum$ и $\{r, s\}$ уже найдены различимыми, то $\{p, q\}$ различимы; это рекурсивный случай. Мы хотим формализовать это с помощью так называемого

алгоритма заполнения таблицы, представляющего собой рекурсивный алгоритм поиска различимых пар состояний.



	A	*B*	*C*	*D*	*E*	*F*	*G*
B	×						
C	×	×					
D	×	×	×				
E		×	×	×			
F	×	×	×		×		
G	×	×	×	×	×	×	
H	×		×	×	×	×	×

Рис. 8.10 ❖ Пример детерминированного конечного автомата и соответствующей таблицы. Различимые состояния отмечены знаком «×»; таблица заполняется только ниже диагонали, так как она симметрична

Задача 8.21. Спроектируйте алгоритм рекурсивного заполнения таблицы. Докажите, что в вашем алгоритме, если два состояния алгоритмом не различаются, эти два состояния эквивалентны.

Теперь мы применим алгоритм заполнения таблицы, чтобы показать эквивалентность автоматов и их минимизировать. Пусть D_1, D_2 равны двум детерминированным конечным автоматам. Для того чтобы увидеть их эквивалентность, то есть $L(D_1) = L(D_2)$, выполним алгоритм заполнения таблицы на их «объединении» и проверим, являются ли $q_0^{D_1}$ и $q_0^{D_2}$ эквивалентными.

Обратите внимание, что эквивалентность состояний является отношением эквивалентности (см. раздел 9.3). Мы можем использовать этот факт, чтобы минимизировать детерминированные конечные автоматы. Для данного детерминированного конечного автомата мы выполняем алгоритм заполнения таблицы, найдя все эквивалентные состояния и, следовательно, все классы эквивалентности. Мы называем каждый класс эквивалентности *блоком*. В примере на рис. 8.10 будут следующие блоки:

$\{E, A\}, \{H, B\}, \{C\}, \{F, D\}, \{G\}$.

Состояния внутри каждого блока эквивалентны, и блоки не пересекаются.

Теперь мы построим минимальный детерминированный конечный автомат с состояниями, которые заданы блоками следующим образом: $\gamma(S, a) = T$, где $\delta(p, a) \in T$ для $p \in S$. Мы должны показать, что γ хорошо определено; предположим, мы выбираем другое $q \in S$. По-прежнему ли истинно, что $\delta(q, a) \in T$? Предположим, что нет, то есть $(q, a) \in T'$, поэтому $\delta(p, a) = t \in T$ и $\delta(q, a) = t' \in T'$. Поскольку $T \neq T'$, $\{t, t'\}$ является *различимой парой*. Но тогда такой парой является и $\{p, q\}$, что противоречит тому, что они оба находятся в S.

Задача 8.22. Покажите, что из этой процедуры мы получаем минимальный детерминированный конечный автомат.

Задача 8.23. Реализуйте алгоритм минимизации. Допустим, что вход задан в виде таблицы переходов, где алфавит задан в виде {0, 1}, и строки таблицы представляют состояния, где первая строка обозначает начальное состояние. Обозначьте

соответствующие принимающим состояниям строки специальным символом, к примеру символом *.

Обратите внимание, что в соответствии с этим соглашением вам не нужно подписывать строки и столбцы входных данных, за исключением символа *, обозначающего принимающие состояния. Таким образом, приведенная на рис. 8.1 таблица переходов будет представлена следующим образом:

```
  2 0
  1 1
* 2 1
```

8.3.8. Нерегулярные языки

Легко показать, что язык является регулярным; для этого нам нужно только продемонстрировать одну из моделей вычислений, которая описывает регулярные языки: детерминированный конечный автомат, недетерминированный конечный автомат или регулярное выражение. Следовательно, доказательство станет *экзистенциальным* и будет состоять в том, что при заданном (предполагаемом) регулярном языке L мы должны показать существование машины A такой, что $L(A) = L$.

Но как показать, что язык не является регулярным? Предположительно мы должны показать, что для каждой машины A, $L(A) \neq L$, что по контрасту является *универсальным доказательством*. Это, интуитивно, кажется более сложной пропозицией, потому что мы не сможем перечислить бесконечно много машин и проверить каждую из них. Следовательно, нам нужен новый прием; на самом деле мы предлагаем два таких приема: «лемму о накачке» и теорему Майхилла–Нероуда. Тем самым мы входим в очень сложную область теории вычислений: доказательство результатов невозможности. К счастью, результаты невозможности для регулярных языков, то есть показывающие, что данный язык не является регулярным, довольно просты. Это происходит потому, что регулярные языки описываются относительно слабыми машинами. Чем сильнее модель вычислений, тем труднее дать для нее результаты невозможности.

Мы заинтересованы в свойствах регулярных языков, потому что важно понимать вычисления «без памяти». Многие встраиваемые устройства, такие как ритмоводители, не имеют памяти или аккумулятора для поддержания памяти. Регулярные языки могут разрешаться с помощью приборов без памяти.

8.3.8.1. Лемма о накачке

Лемма 8.24 (лемма о накачке). Пусть L равно регулярному языку. Тогда существует постоянная n (в зависимости от L) такая, что для всех $w \in L$, $|w| \geqslant n$, мы можем разбить w на три части $w = xyz$ таких, что:

1) $y \neq \varepsilon$;
2) $|xy| \leqslant n$;
3) для всех $k \geqslant 0$, $xy^kz \in L$.

Доказательство. Предположим, что L является регулярным. Тогда существует детерминированный конечный автомат A такой, что $L = L(A)$. Пусть n равно числу состояний A. Рассмотрим любое $w = a_1a_2 \ldots a_m$, $m \geqslant n$:

$$\overset{p_0}{\uparrow}\overbrace{a_1}^{x}\overset{p_1}{\uparrow}\overbrace{a_2}^{x}\overset{p_2}{\uparrow}\overbrace{a_3 \ldots a_i}^{x}\overset{p_i}{\uparrow}\overbrace{a_{i+1} \ldots a_j}^{y}\overset{p_j}{\uparrow}\overbrace{a_{j+1} \ldots a_m}^{z}\overset{p_m}{\uparrow}.$$ □

Задача 8.25. Покажите, что $L = \{0^n 1^n | n \geqslant 0\}$ не является регулярным.

Задача 8.26. Покажите, что $L = \{1^p | \ p$ является простым$\}$ не является регулярным.

8.3.8.2. Теорема Майхилла–Нероуда

Теорема Майхилла–Нероуда обеспечивает определение регулярных языков, которое дается без упоминания модели вычислений. Она характеризует регулярные языки в терминах реляционных свойств строк. См. раздел 9.3, для того чтобы освежить свои знания об отношениях эквивалентности.

Начнем с некоторых определений и наблюдений. С учетом языка $L \subseteq \sum^*$ пусть $\equiv_L$ равно отношению на $\sum^* \times \sum^*$ такому, что $x \equiv_L y$, если для всех z, $xz \in L \Leftrightarrow yz \in L$.

Задача 8.27. Покажите, что $\equiv_L$ на самом деле является отношением эквивалентности.

Предположим, что некоторый детерминированный конечный автомат D распознает L, и $k = |Q_D|$. Мы говорим, что X является множеством, которое *попарно различимо языком L* тогда и только тогда, когда для каждых двух несовпадающих $x, y \in X$, $x \not\equiv_L y$. Покажем, что если $|Q_D| = k$, то $|X| \leqslant k$. Предположим, что $\{x_1, x_2, \ldots, x_{k+1}\} \subseteq X$. Поскольку существует k состояний, существует два x_i, x_j, не совпадающих так, что

$$\begin{aligned}
&\hat{\delta}_D(q_0, x_i) = \hat{\delta}(q_0, x_j) \\
&\Rightarrow \forall z[\hat{\delta}_D(q_0, x_i z) = \hat{\delta}(q_0, x_j z)] \\
&\Rightarrow \forall z[x_i z \in L \Leftrightarrow x_j z \in L] \\
&\Rightarrow x_i \equiv_L x_j.
\end{aligned}$$

Таким образом, невозможно, чтобы $|X| > k$. Обозначим через индекс(L) мощность $|X|$ наибольшего попарно различимого множества $X \subseteq L$.

Теорема 8.28 (Майхилла–Нероуда). L является регулярным тогда и только тогда, когда индекс(L) является конечным. Более того, индекс(L) является размером наименьшего детерминированного конечного автомата для L.

Доказательство. Предположим, что индекс(L) = k, и пусть $X = \{x_1, x_2, \ldots, x_k\}$; сначала отметим, что для любого $x \in \sum^*$, $x \equiv_L x_i$ для некоторого (уникального) $x_i \in X$; в противном случае $X \cup \{x\}$ будет более крупным множеством, которое «попарно различимо языком L». Уникальность следует за транзитивностью.

Пусть D является таким, что $Q_D = \{q_1, \ldots, q_k\}$ и

$$\delta_D(q_i, a) = q_j \Leftrightarrow x_i a \equiv_L x_j.$$

Тот факт, что существует (уникальный) x_j такой, что $x_i a \equiv_L x_j$, следует из приведенного выше наблюдения. Тем самым $\hat{\delta}(q_i, w) = q_j \Leftrightarrow x_i w \equiv_L x_j$.

Пусть $F_D = \{q_i \in Q_D : x_i \in L\}$, и пусть $q_0 := q_i$ такое, что $x_i \equiv_L \varepsilon$.

Легко показать, что наш D работает: $x \in L \Leftrightarrow x \equiv_L x_i$ для некоторого $x_i \in L$. Для того чтобы это увидеть, обратите внимание, что $x \equiv_L x_i$ для уникального x_i, и если

это $x_i \notin L$, то $x\varepsilon \in L$, тогда как $x_i\varepsilon \notin L$, поэтому мы получаем противоречие $x \not\equiv_L x_i$. Наконец, $x \equiv_L x_i$ тогда и только тогда, когда $\hat{\delta}(q_0, x) = q_i \in F_D$. □

8.3.9. Автоматы на членах

См. раздел 9.4 для получения необходимой общей информации по математической логике. В первопорядковой логике словарь (вокабуляр)

$$\mathcal{V} = \{\mathbf{f}_1, \mathbf{f}_2, \mathbf{f}_3, \ldots; \mathbf{R}_1, \mathbf{R}_2, \mathbf{R}_3, \ldots\}$$

представляет собой множество символов функций (**f**) и отношений (**R**). Каждая функция и отношение имеют арность, то есть «сколько аргументов функция или отношение принимает». Функция с арностью 0 называется константой.

Мы определяем $\mathcal{V}$-члены (члены, когда $\mathcal{V}$ понимается из контекста) по структурной индукции следующим образом: любая константа **c** (то есть арность(**c**) = 0) является членом, и если $\mathbf{t}_1, \ldots, \mathbf{t}_n$ являются n-членами, и **f** является символом функции с арностью n, то $\mathbf{ft}_1 \ldots \mathbf{t}_n$ тоже является членом. То есть члены конструируются путем смежного расположения. Пусть $\mathcal{T}$ равно множеству всех членов. Обратите внимание, что, в отличие от первопорядковой логики, мы не вводим переменные.

Задача 8.29. Покажите, что члены «однозначно читаемы» (подсказка: сравните с теоремой 9.80.)

$\mathcal{V}$-алгебра (алгебра, когда $\mathcal{V}$ понимается из контекста) является интерпретацией данного словаря $\mathcal{V}$. То есть $\mathcal{A}$ представляет собой $\mathcal{V}$-алгебру, если она состоит из непустого множества A (именуемого *универсумом* $\mathcal{A}$) вместе с интерпретацией всех символов функций и отношений $\mathcal{V}$. То есть при наличии $\mathbf{f} \in \mathcal{V}$ арности n $\mathcal{A}$ обеспечивает интерпретацию для **f** в том смысле, что она назначает **f** смысл $f : A_n \to A$. Мы пишем $\mathbf{f}^{\mathcal{A}}$ для обозначения f, или просто $\mathbf{f}^{\mathcal{A}} = f$.

$\mathcal{A}$ назначает каждому члену **t** интерпретацию $\mathbf{t}^{\mathcal{A}} \in A$.

Задача 8.30. Определите $\mathbf{t}^{\mathcal{A}}$ для произвольных членов. Какая структура данных может быть естественным образом связана с осуществлением такой интерпретации? Какова естественная интерпретация для отношений, то есть какова интерпретация $(\mathbf{Rt}_1 \ldots \mathbf{t}_n)^{\mathcal{A}}$? Четко укажите разницу в «типе» между $\mathbf{f}^{\mathcal{A}}$ и $\mathbf{R}^{\mathcal{A}}$.

Мы говорим, что алгебра $\mathcal{A}$ является автоматом, если универсум $\mathcal{A}$ конечен, а $\mathcal{V}$ имеет один унарный символ отношения **R**. Мы говорим, что $\mathcal{A}$ принимает член $\mathbf{t} \in \mathcal{T}$, если $\mathbf{t}^{\mathcal{A}} \in \mathbf{R}^{\mathcal{A}}$. Как и в случае детерминированного конечного автомата, мы делаем $L(\mathcal{A})$ равным множеству $\mathbf{t} \in \mathcal{T}$, которое принимается алгеброй $\mathcal{A}$.

Задача 8.31. Пусть $\sum$ равно конечному алфавиту, и пусть $\mathcal{V} = \sum' \cup \{\mathbf{c}\}$, где **c** – это новый символ, обозначающий функцию с арностью 0, и каждый $a \in \sum$ интерпретируется как несовпадающий унарный функциональный символ a в $\sum'$ (тем самым $|\sum| = |\sum'|$). Покажите, что язык L над $\sum$ является регулярным тогда и только тогда, когда некоторый автомат $\mathcal{A}$ принимает $L' = \{\mathbf{a}_n \ldots \mathbf{a}_2\mathbf{a}_1\mathbf{c} : a_1a_2 \ldots a_n \in L\}$.

Мы говорим, что подмножество $L \subseteq \mathcal{T}$ является регулярным, если $L = L(\mathcal{A})$ для некоторого автомата $\mathcal{A}$. Обратите внимание, что это определение регулярности является более широким, поскольку не все функции в $\mathcal{V}$ обязательно унарны (когда они унарны, как показала задача 8.31, это определение «регулярности» соответствует классическому определению «регулярности»).

Задача 8.32. Покажите, что регулярные языки (в этой новой обстановке) замкнуты при объединении, дополнении и пересечении.

8.4. Контекстно-свободные языки

В 1961 году Хомский пообещал произвести полную грамматику английского языка – через полгода он сказал, что это невозможно, и отказался от проекта. Однако его аспиранты продолжили работать в этой области, и разные подходы к лингвистике соответствовали разным этапам мышления Хомского и студентов, которые были у него в то время. Хомский хотел использовать грамматики для *генерирования речи*; до того времени грамматика использовалась только для анализа текста. Его подход называется *структурной лингвистикой*, тенденцией отделять грамматику от смысла (напоминает мне логический позитивизм Карнапа). В информатике грамматика (компьютер) и смысл (человек) всегда разделены; взаимодействие между синтаксисом и семантикой является одной из самых богатых концепций в информатике.

Первым языком, разработанным в соответствии с принципами Хомского, был ALGOL (великий прародитель C, C++, Pascal и др.). ALGOL был основан на грамматиках Хомского и, следовательно, нечитаем для людей; поэтому первые программисты на языке ALGOL ввели понятие *отступа*.

В 1960-е годы люди мыслили не в терминах алгоритмов, а скорее в терминах воображаемых машин, то есть в терминах аппаратных средств. *Магазинные автоматы* (автоматы с магазинной памятью, pushdown automaton, PDA) – это машины, соответствующие *контекстно-свободным грамматикам* (context-free grammar, CFG), так же как детерминированные конечные автоматы соответствуют регулярным языкам. Основное различие между детерминированными конечными автоматами и магазинными автоматами заключается в том, что детерминированные конечные автоматы представляются собой «алгоритмы», которые не требуют динамического выделения памяти (без операции `malloc`), а магазинные автоматы предусматривают динамическое выделение памяти, хотя и в самом примитивном типе структуры данных: стеке.

8.4.1. Контекстно-свободные грамматики

Контекстно-свободная грамматика (context-free grammar, CFG) выражается кортежем $G = (V, T, P, S)$, где буквы обозначают множество переменных, терминалов, продукций и заданную начальную переменную.

Например, в грамматике языка палиндромов используется следующая продукция: $P \to \varepsilon|0|1|0P0|1P1$. А грамматика для языка (редуцированных) алгебраических выражений равна $G = (\{E, T, F\}, \sum, P, E)$, где $\sum = \{a, +, \times, (,)\}$ и P равно следующему множеству продукций:

$E \to E + T|T$;
$T \to T \times F|F$;
$F \to (E)|a$.

Здесь мы используем E для выражений, T для членов (термов) и F для составляющих (сомножителей, факторов). В рамках нормальных интерпретаций знаков

+ и × три приведенные выше продукции в указанном порядке отражают следующие структурные факты об алгебраических выражениях:

- выражение – это член либо сумма выражения и члена;
- член – это либо фактор, либо произведение члена и фактора;
- фактор – это скобочное выражение либо терминал «*a*».

Таким образом, самым простым выражением будет выражение, состоящее из одного члена, который, в свою очередь, состоит из одного фактора: *a*.

Рассмотрим строку $\alpha A\beta$ над алфавитом $(V \cup T)^*$, где $A \in V$ и $A \to \gamma$ является продукцией. Тогда мы можем сказать, что $\alpha A\beta$ *порождает* $\alpha\gamma\beta$ в символах: $\alpha A\beta \Rightarrow \alpha\gamma\beta$. Мы используем $\overset{*}{\Rightarrow}$ для обозначения 0 или более шагов. Теперь мы можем определить язык грамматики как $L(G) = \{w \in T^* | S \overset{*}{\Rightarrow} w\}$.

Лемма 8.33. $L((\{P\}, \{0, 1\}, \{P \to \varepsilon|0|1|0P0|1P1\}, P))$ есть множество палиндромов над {0,1}.

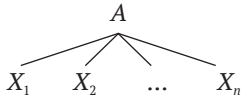
Доказательство. Предположим, w – это палиндром. Мы покажем по индукции на $|w|$, что $P \overset{*}{\Rightarrow} w$. Базовый случай: $|w| \leqslant 1$, поэтому $w = \varepsilon, 0, 1$, и поэтому используем единственное правило $P \to \varepsilon, 0, 1$. Индукционный шаг: для $|w| \geqslant 2$, $w = 0x0, 1x1$, и по индукционной гипотезе $P \overset{*}{\Rightarrow} w$.

Предположим, что $P \overset{*}{\Rightarrow} w$. Мы показываем по индукции на числе шагов в деривации, что $w = w^R$. Базовый случай: деривация имеет один шаг. Индукционный шаг:

$$P \Rightarrow 0P0 \overset{*}{\Rightarrow} 0x0 = w,$$

где 0 вместо этого можно заменить на 1. □

Предположим, что у нас есть грамматика $G = (V, T, P, S)$ и $S \overset{*}{\Rightarrow} \alpha$, где $\alpha \in (V \cup T)^*$. Тогда α называется *сентенциальной формой* (этой конкретной грамматики G). Пусть $L(G)$ равно множеству этих сентенциальных форм, которые находятся в T^*. Другими словами, как и в случае регулярных языков, $L(G)$ является языком G. Мы определим *дерево разбора* для (G, w) следующим образом: это корневое дерево, в котором S помечает корень, и листья помечены слева направо символами w. Для каждого внутреннего узла, то есть всех узлов, кроме листьев, метки имеют следующую форму:



где $A \to X_1X_2X_3 \ldots X_n$ – это правило в P.

Существует ряд способов продемонстрировать, что данное слово w может быть сгенерировано грамматикой G, то есть доказать, что $w \in L(G)$. Эти способы следующие: *рекурсивный вывод*, *деривация*, *левосторонняя деривация*, *правосторонняя деривация* и *порождение дерева разбора*. Рекурсивный вывод похож на деривацию, за исключением того, что мы генерируем деривацию из w в S. Лево(право)сторонняя деривация – это деривация, которая всегда применяет правило к крайне левой (правой) переменной в промежуточной сентенциальной форме.

Мы говорим, что грамматика *неоднозначна*, если есть слова, которые имеют два разных дерева разбора. Например, $G = (\{E\}, [0–9], \{E \to E+E, E*E\}, E)$ неоднозначна, так как деревья разбора, соответствующие этим двум деривациям, различны:

$E \Rightarrow E + E \Rightarrow E + E * E$;
$E \Rightarrow E * E \Rightarrow E + E * E$.

Проблема в том, что деревья разбора назначают строке смысл, и два разных дерева разбора назначают два возможных смысла, отсюда и «неоднозначность».

Задача 8.34. Покажите, что расширенные регулярные языки, как определено в разделе 8.3.9, содержатся в классе контекстно-свободных языков.

8.4.2. Магазинные автоматы

Магазинные автоматы (PDA) – это недетерминированные конечные автоматы со стеком. Формальное определение магазинного автомата дается следующим образом: $P = (Q, \sum, \Gamma, \delta, q_0, F)$, где:

1) Q – конечное множество состояний;
2) $\sum$ – конечный входной алфавит;
3) Γ – конечный стековый алфавит;
4) $\delta(q, x, a) = \{(p_1, b_1), \ldots, (p_n, b_n)\}$;
5) q_0 – начальное состояние;
6) F – принимающие состояния.

Задача 8.35. Каким является простой магазинный автомат для $\{ww^R | w \in \{0, 1\}^*\}$?

P вычисляет следующим образом: он принимает заданную строку w в $\sum*$, если $w = w_1 w_2 \ldots w_m$, где $w_i \in \sum_\varepsilon$, где $|w| = n \leqslant m$. То есть существует ε-заполнение w такое, что существует последовательность состояний $r_0, r_1, \ldots, r_m$ в Q и последовательность содержимого стека $s_0, s_1, \ldots, s_m \in \Gamma^*$ такая, что соблюдаются следующие три условия:

1) $r_0 = q_0$ и $s_0 = \varepsilon$;
2) $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, где $s_i = at$, $s_{i+1} = bt$ и $a, b \in \Gamma_\varepsilon$ и $t \in \Gamma^*$. То есть M перемещается правильно в соответствии с состоянием, стеком и следующим входным символом;
3) $r_m \in F$.

Конфигурация – это кортеж (q, w, γ): состояние, оставшиеся входные данные, содержимое стека. Если $(p, \alpha) \in \delta(q, a, X)$, то $(q, aw, X\beta) \rightarrow (p, w, \alpha\beta)$.

Лемма 8.36. Если $(q, x, \alpha) \overset{*}{\Rightarrow} (p, y, \beta)$, то $(q, xw, \alpha\gamma) \overset{*}{\Rightarrow} (p, yw, \beta\gamma)$.

Задача 8.37. Докажите лемму 8.36.

Существует два эквивалентных способа точно определить, что именно означает для магазинного автомата принять входное слово. Существует принятие по финальному состоянию, где:

$$L(P) = \{w | (q_0, w, \$) \overset{*}{\Rightarrow} (q, \varepsilon, \alpha), q \in F\},$$

и принятие по пустому стеку:

$$L(P) = \{w | (q_0, w, \$) \overset{*}{\Rightarrow} (q, \varepsilon, \varepsilon)\}.$$

При проектировании магазинных автоматов удобнее использовать одно из этих определений, чем другое, но, как демонстрирует следующая теорема, оба определения выражают одно и то же множество языков.

Лемма 8.38. *L* принимается магазинным автоматом по конечному состоянию тогда и только тогда, когда он принимается магазинным автоматом по пустому стеку.

Задача 8.39. Докажите лемму 8.38.

Теорема 8.40. Контекстно-свободные грамматики и магазинные автоматы являются эквивалентными.

Доказательство. Сначала мы покажем, как выполнять трансляцию контекстно-свободной грамматики в эквивалентный магазинный автомат. Левая сентенциальная форма представляет собой особый способ выразить конфигурацию, где:

$$\underbrace{x}_{\in T^*}\ \overbrace{A\alpha}^{\text{хвост}}.$$

Хвост появляется в стеке, и x является префиксом потребленных к этому моменту входных данных. Идея состоит в том, что входные данные для магазинного автомата заданы $w = xy$ и $A\alpha \stackrel{*}{\Rightarrow} y$.

Предположим, что магазинный автомат находится в конфигурации $(q, y, A\alpha)$ и что он использует правило $A \to \beta$ и входит в $(q, y, \beta\gamma)$. Магазинный автомат симулирует грамматику следующим образом: выполняется разбор начального сегмента β, и если есть терминальные символы, то они сравниваются с входными данными и удаляются до тех пор, пока первая переменная β не появится наверху стека. Этот процесс повторяется, и магазинный автомат принимает по пустому стеку.

Например, рассмотрим $P \to \varepsilon|0|1|0P0|1P1$. Соответствующий магазинный автомат имеет переходы:

$\delta(q_0, \varepsilon, \$) = \{(q, P\$)\}$;
$\delta(q, \varepsilon, P) = \{(q, 0P0), (q, 0), (q, \varepsilon), (q, 1P1), (q, 1)\}$;
$\delta(q, 0, 0) = \delta(q, 1, 1) = \{(q, \varepsilon)\}$;
$\delta(q, 0, 1) = \delta(q, 1, 0) = \emptyset$;
$\delta(q, \varepsilon, \$) = (q, \varepsilon)$.

Вычисление представлено на рис. 8.11.

Z	P	1	P	0	P	0	P	0	0	1	Z
	Z	P	1	P	0	P	0	0	1	Z	
		1	Z	0	1	0	0	1	Z		
		Z		1	Z	0	1	Z			
				Z		1	Z				
						Z					

Рис. 8.11 ❖ Вычисление для $P \Rightarrow 1P1 \Rightarrow 10P01 \Rightarrow 100P001 \Rightarrow 100001$

Теперь мы дадим схематичное описание того, как выполнять трансляцию из магазинного автомата в контекстно-свободную грамматику. Идея заключается в «чистом выталкивании» одного символа стека, потребляя некоторые входные данные. Переменными являются: $A_{[pXq]}$ для $p, q \in Q, X \in \Gamma$. $A_{[pXq]} \stackrel{*}{\Rightarrow} w$ тогда и только

тогда, когда w переводит магазинный автомат из состояния p в состояние q и выталкивает X из стека. Продукции: для всех p, $S \to A_{[q_0\$p]}$, и всякий раз, когда мы имеем:

$$(r, Y_1Y_2 \ldots Y_k) \in \delta(q, a, X),$$

мы привлекаем правило:

$$A_{[qXr_k]} \to aA_{[rY_1r_1]}A_{[r_1Y_2r_2]} \ldots A_{[r_{k-1}Y_kr_k]},$$

где $a \in \sum \cup \{\varepsilon\}$, $r_1, r_2, \ldots, r_k \in Q$ – это все возможные списки состояний.

Если $(r, \varepsilon) \in \delta(q, a, X)$, то мы имеем $A_{[qXr]} \to a$.

Задача 8.41. Покажите, что $A_{[qXp]} \overset{*}{\Rightarrow} w$ тогда и только тогда, когда $(q, w, X) \overset{*}{\Rightarrow} (p, \varepsilon, \varepsilon)$.

Это завершает доказательство леммы. □

Магазинный автомат является детерминированным, если $|\delta(q, a, X)| \leqslant 1$, и второе условие состоит в том, что если для некоторого $a \in \sum |\delta(q, a, X)| = 1$, то $|\delta(q, \varepsilon, X)| = 0$. Мы называем такие машины детерминированными магазинными автоматами (deterministic PDA).

Лемма 8.42. Если L является регулярным, то $L = L(P)$ для некоторого детерминированного магазинного автомата P.

Доказательство. Просто прослеживаем, что детерминированный конечный автомат – это детерминированный магазинный автомат. □

Обратите внимание, что детерминированные магазинные автоматы, которые принимают по конечному состоянию, не эквивалентны детерминированным магазинным автоматам, которые принимают по пустому стеку. Для того чтобы изучить связь между принятием по состоянию либо по пустому стеку в контексте детерминированных магазинных автоматов, введем следующее свойство языков: L имеет *префиксное свойство*, если существует пара (x, y), $x, y \in L$, такая, что $y = xz$ для некоторого z. Например, $\{0\}^*$ имеет префиксное свойство.

Лемма 8.43. L принимается детерминированным магазинным автоматом по пустому стеку $\Leftrightarrow$ L принимается детерминированным магазинным автоматом по конечному состоянию, и L не имеет *префиксного свойства*.

Лемма 8.44. Если L принимается детерминированным магазинным автоматом, то L является *однозначным*.

8.4.3. Нормальная форма Хомского

В этом разделе мы покажем, что каждая контекстно-свободная грамматика может быть помещена в особо простую форму, называемую *нормальной формой Хомского* (Chomsky normal form, CNF). Контекстно-свободная грамматика находится в нормальной форме Хомского, если все правила принимают одну из следующих трех форм:

1) $S \to \varepsilon$, где S – это начальная переменная;
2) $A \to BC$, где A, B, C – это переменные, возможно, повторяющиеся;
3) $A \to a$, где A – это переменная, и a – это символ алфавита (не ε).

Нормальная форма Хомского имеет много желательных свойств, но одним из наиболее важных следствий является так называемый *алгоритм CYK* (алгоритм 8.1, раздел 8.4.4), представляющий собой алгоритм динамического программирования для решения $w \in L(G)$ для данного слова w и контекстно-свободной грамматики G.

Теперь мы покажем, как преобразовывать произвольную контекстно-свободную грамматику в нормальную форму Хомского. В последующем обсуждении S – это переменная, $X \in V \cup T$, $w \in T^*$ и $\alpha, \beta \in (V \cup T)^*$. Мы говорим, что символ X является полезным, если существует деривация такая, что $S \stackrel{*}{\Rightarrow} \alpha X \beta \stackrel{*}{\Rightarrow} w$.

Мы говорим, что X *генерирует*, если $X \stackrel{*}{\Rightarrow} w \in T^*$, и мы говорим, что X *достижим*, если существует деривация $S \stackrel{*}{\Rightarrow} \alpha X \beta$. Полезный символ будет генерировать и будет достижимым. Таким образом, если сначала исключить негенерирующие символы, а затем из оставшейся грамматики – недостижимые символы, то останутся только полезные.

Задача 8.45. Докажите, что если сначала исключить негенерирующие символы, а затем из оставшейся грамматики исключить недостижимые символы, то останутся только полезные символы.

Покажем, как мы устанавливаем множество генерирующих символов и множество достижимых символов. Безусловно, каждый символ в T генерирует, и если $A \to \alpha$ является продукцией, и каждый символ в α генерирует (или $\alpha = \varepsilon$), то A тоже генерирует. Схожим образом S достижим, и если A достижим, и $A \to \alpha$ является продукцией, то каждый символ в α достижим.

Утверждение 8.46. Если L имеет контекстно-свободную грамматику, то $L - \{\varepsilon\}$ является контекстно-свободной грамматикой без продукций в форме $A \to \varepsilon$ и без продукций в форме $A \to B$.

Доказательство. Переменная допускает *наличие пустого значения*, если $A \stackrel{*}{\Rightarrow} \varepsilon$. Для вычисления переменных, допускающих пустое значение: если $A \to \varepsilon$ является продукцией, то A допускает пустое значение; если $B \to C_1 C_2 \ldots C_k$ является продукцией, и все C_i допускают пустое значение, то и B тоже. После того как у нас все переменные допускают пустое значение, мы исключаем ε-продукции следующим образом: исключаем все $A \to \varepsilon$.

Если $A \to X_1 X_2 \ldots X_k$ является продукцией и $m \leqslant k$ из X_i допускают пустое значение, то добавить 2^m версий правила присутствия/отсутствия, допускающих пустое значение переменных (если $m = k$, то не добавлять случай, когда они все отсутствуют).

Устранение единичных продукций: $A \to B$. Если $A \stackrel{*}{\Rightarrow} B$, то (A, B) является единичной парой. Найти все пары: (A, A) является единичной парой, и если (A, B) является единичной парой, и $B \to C$ является продукцией, то (A, B) является единичной парой. Чтобы устранить единичные продукции: вычислить *все* единичные пары, и если (A, B) является единичной парой и $B \to \alpha$ является неединичной продукцией, то добавить продукцию $A \to \alpha$. Отбросить все единичные продукции. □

Теорема 8.47. Каждый контекстно-свободный язык имеет контекстно-свободную грамматику в нормальной форме Хомского.

Доказательство. Чтобы преобразовать G в нормальную форму Хомкого, начнем с устранения всех ε-продукций, единичных продукций и бесполезных символов. Расположим все тела длиной $\geqslant 2$ так, чтобы они состояли только из переменных (путем введения новых переменных), и, наконец, разобьем тела длиной $\geqslant 3$ на каскад продукций, каждая из которых имеет тело длиной ровно 2. □

8.4.4. Алгоритм CYK

С учетом грамматики G в нормальной форме Хомского и строки $w = a_1a_2 \ldots a_n$ мы можем проверить, является ли истинным, что $w \in L(G)$, используя динамический алгоритм CYK[1] (алгоритм 8.1). При передаче в него G, $w = a_1a_2 \ldots a_n$ алгоритм 8.1 строит $n \times n$-таблицу T, где каждый элемент содержит подмножество V. В конце $w \in L(G)$ тогда и только тогда, когда начальная переменная S содержится в позиции $(1, n)$ таблицы T. Главная идея – поставить переменную X_1 в позицию (i, j), если X_2 находится в позиции (i, k), X_3 находится в позиции $(k+1, j)$ и $X_1 \to X_2X_3$ является правилом. Смысл этого заключается в том, что X_1 находится в позиции (i, k) тогда и только тогда, когда $X_1 \overset{*}{\Rightarrow} a_i \ldots a_k$, то есть подстрока $a_i \ldots a_k$ входной строки может быть сгенерирована из X_1. Пусть $V = \{X_1, X_2, \ldots, X_m\}$.

Алгоритм 8.1. CYK

```
for i = 1..n do
        for j = 1..m do
                Поместить переменную x_j в (i, i) тогда и только тогда, когда X_j →
                a_i является правилом грамматики G
        end for
end for
for 1 ⩽ i < j ⩽ n do
        for k = i..(j − 1) do
                if (∃X_p ∈ (i, k) ∧ ∃X_q ∈ (k + 1, j) ∧ ∃X_r → X_pX_q) then
                        Поставить X_r в (i, j)
                end if
        end for
end for
```

В примере на рис. 8.12 мы показываем, какие ячейки в таблице необходимо использовать для вычисления содержимого (2, 5).

Задача 8.48. Покажите правильность алгоритма 8.1.

Задача 8.49. Реализуйте алгоритм CYK. Выберите условные обозначения для представления контекстно-свободных грамматик, и хорошо задокументируйте их в своем исходном коде. Вы можете исходить из того, что грамматика задана в нормальной форме Хомского, или же вы можете выполнять ее явную проверку. Для того чтобы сделать проект еще более амбициозным, вы можете реализовать трансляцию общей грамматики в нормальную форму Хомского.

[1] Назван в честь изобретателей: Кока–Янгера–Касами (Cocke–Younger–Kasami).

×	(2, 2)	(2, 3)	(2, 4)	(2, 5)
×	×			(3, 5)
×	×	×		(4, 5)
×	×	×	×	(5, 5)

Рис. 8.12 ❖ Вычисление элемента (2, 5): обратите внимание, что нам требуются все элементы в одной строке и одном столбце (за исключением тех, которые находятся ниже главной диагонали). Таким образом, алгоритм CYK динамически вычисляет элементы по диагоналям, начиная с главной диагонали и заканчивая правым верхним углом

8.4.5. Лемма о накачке для контекстно-свободных языков

Лемма 8.50 (лемма о накачке для контекстно-свободных языков). Существует p такое, что любое s, $|s| \geqslant p$, можно записать как $s = uvxyz$, и:

1) uv^ixy^iz находится в языке, для всех $i \geqslant 0$;
2) $|vy| > 0$;
3) $|vxy| \leqslant p$.

Доказательство. Следуя рассуждениям по принципу Дирихле (на основе голубиных клеток), используемому для демонстрации леммы о накачке для регулярных языков (см. раздел 8.3.8.1, лемма 8.24), рисунка 8.13 должно быть достаточно для того, чтобы убедить читателя: оказывается, что этот аргумент лучше всего выполняется вместе с трансляцией грамматики в нормальную форму Хомского (раздел 8.4.3). Затем находим длину входных данных, которая гарантирует высоту дерева не менее $|V| + 1$. Подробности оставляются читателю. □

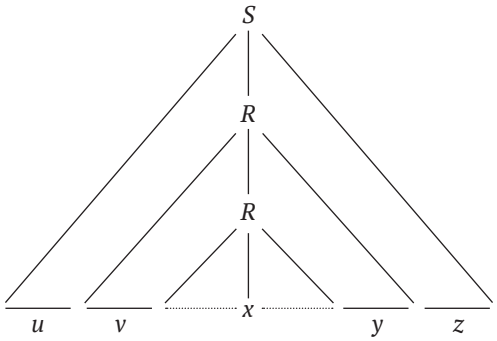


Рис. 8.13 ❖ Если слово достаточно длинное, то высота дерева разбора довольно велика, для того чтобы заставить некоторую переменную (R) повторяться вдоль некоторой ветви

Задача 8.51. Закончите доказательство леммы 8.50.

Задача 8.52. Покажите, что $L = \{0^n1^n2^n | n \geqslant 1\}$ не является нормальной формой Хомского.

8.4.6. Дальнейшие замечания по нормальной форме Хомского

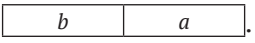
Нормальные формы Хомского не имеют таких же широких свойств замыкания, как регулярные языки (см. раздел 8.3.5). Нормальные формы Хомского замыкаются при объединении, конкатенации, звезде Клини ($*$), гомоморфизмах и реверсах. Касаясь гомоморфизма, обратите внимание, что гомоморфизм может быть применен к деривации. Что касается реверсов, то следует просто заменить каждую $A \to \alpha$ на $A \to \alpha^R$.

Нормальные формы Хомского не замыкаются при пересечении или дополнении. Для того чтобы увидеть, что они не замыкаются при пересечении, обратите внимание, что $L_1 = \{0^n1^n2^i | n, i \geqslant 1\}$ и $L_2 = \{0^i1^n2^n | n, i \geqslant 1\}$ являются нормальными формами Хомского, но $L_1 \cap L_2 = \{0^n1^n2^n | n \geqslant 1\}$ таковыми не являются.

Для того чтобы увидеть, что нормальные формы Хомского не замыкаются при дополнении, обратите внимание, что язык $L = \{ww : w \in \{a, b\}^*\}$ не является контекстно-свободным, однако L^c является контекстно-свободным. Оказывается, что демонстрация того, что L^c является контекстно-свободным языком, является нетривиальной; проектирование контекстно-свободной грамматики сопряжено с трудностями: в первую очередь стоит отметить, что никакие нечетные строки не имеют формы ww, поэтому первым правилом должно быть:

$S \to O|E$
$O \to a|b|aaO|abO|baO|bbO;$

здесь O генерирует все нечетные строки. С другой стороны, E генерирует строки четной длины, не имеющие форму ww, то есть все строки имеют форму:

$X =$  $Y =$  .

Нам нужно правило:

$E \to X|Y,$

и теперь

$X \to PQ,$ $Y \to VW,$
$P \to RPR,$ $V \to SVS,$
$P \to a,$ $V \to b,$
$Q \to RQR,$ $W \to SWS,$
$Q \to b,$ $W \to a,$
$R \to a|b,$ $S \to a|b.$

Обратите внимание, что все R могут быть заменены любым a или b, что дает нам желаемое свойство.

Задача 8.53. Покажите, что если L является контекстно-свободным языком и R является регулярным языком, то $L \cap R$ является контекстно-свободным языком.

Задача 8.54. Мы знаем, что контекстно-свободные языки замкнуты при подстановках (типе гомоморфизма): для каждого $a \in \sum$ мы выбираем L_a, который называем $s(a)$. Для любого $w \in \sum^*$, $s(w)$ является языком из $x_1x_2 \ldots x_n$, $x_i \in s(a_i)$. Покажите, что если L является контекстно-свободным языком и $s(a)$ является контекстно-свободным языком $\forall a \in \sum$, то $s(L) = \bigcup_{w \in L} s(w)$ тоже является контекстно-свободным языком.

В то время как алгоритм CYK позволяет нам решить, находится ли данная строка w на языке некоторой данной контекстно-свободной грамматики G, существует много свойств контекстно-свободной грамматики, которые, к сожалению, неразрешимы. Что это означает? Это означает, что существуют вычислительные проблемы в отношении контекстно-свободной грамматики, для которых нет алгоритмов. Например:

1) является ли данная контекстно-свободная грамматика G неоднозначной?
2) является ли данный контекстно-свободный язык внутренне неоднозначным?
3) является ли пересечение двух контекстно-свободных языков пустым?
4) с учетом G^1, G^2 является ли истинным, что $L(G^1) = L(G^2)$?
5) эквивалентна ли данная контекстно-свободная грамматика $\sum^*$?

Показать, что у конкретной задачи нет алгоритма, который ее решает, очень трудно. По сути дела, мы должны ввести новый метод, для того чтобы показать, что пять приведенных выше вопросов являются «неразрешимыми». Мы делаем это в разделе 8.5. Для нетерпеливых см. раздел 8.5.9.

8.4.7. Другие грамматики

Контекстно-чувствительные грамматики (context-sensitive grammar, CSG) имеют правила формы:

$$\alpha \rightarrow \beta,$$

где $\alpha, \beta \in (T \cup V)^*$ и $|\alpha| \leqslant |\beta|$. Язык является *контекстно-чувствительным*, если он имеет контекстно-чувствительную грамматику. В элегантной связи со сложностью, как оказалось, контекстно-чувствительные языки точно описывают множество тех языков, которые могут быть решены недетерминированными машинами Тьюринга за линейное время (см. следующий раздел).

Система перезаписи (также именуемая *полутуевской системой*[1]) – это грамматика, в которой нет никаких ограничений; $\alpha \rightarrow \beta$ для произвольных $\alpha, \beta \in (V \cup T)^*$.

Системы перезаписи соответствуют наиболее общей модели вычислений в том смысле, что все, что может быть решено алгоритмически, может быть решено с помощью системы перезаписи. Таким образом, язык имеет систему перезаписи тогда и только тогда, когда он «вычислим», что является темой следующих разделов.

8.5. Машины Тьюринга

Машина Тьюринга – это автомат с конечным управлением и бесконечной лентой, где бесконечная лента выражает идею «неограниченного пространства». Первоначально входные данные помещаются на ленту, считывающе-записывающая головка ленты расположена на первом символе входных данных, и состояние равно q_0. Все остальные ячейки содержат пустоты.

---

[1] Полутуевская система – это система перезаписи слов, близкая к грамматике типа 0. Единственное различие заключается в том, что в полутуевских системах нет разделения на терминальные и нетерминальные символы и нет выделенного начального символа. См. https://pl.wikipedia.org/wiki/System_p%C3%B3%C5%82thueowski. – *Прим. перев.*

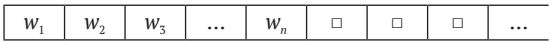


Рис. 8.14 ❖ Исходное содержимое ленты; считывающе-записывающая головка сканирует w_1

Формально *машина Тьюринга* представляет собой кортеж $(Q, \sum, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, где входной алфавит $\sum$ содержится в ленточном алфавите Γ, и □ – это «пустой» символ, то есть $\sum \cup \{\square\} \subseteq \Gamma$. *Переходная функция* $\delta(q, X) = (p, Y, D)$, где D – это направление движения ленты, «влево» или «вправо», иногда обозначаемое как «←» или «→».

Конфигурация представляет собой строку upv, где $u, v \in \Gamma^*$ и $p \in Q$, имея в виду, что состояние равно p, считывающе-записывающая головка сканирует первый символ v, и лента содержит только пустоты после последнего символа v. Первоначально конфигурация равна q_0w, где $w = w_1w_2 \ldots w_n$, $w_i \in \sum$ является входами, и первый символ w, w_1, помещается в самую левую ячейку ленты. Соблюдая особую осторожность, мы говорим, что символ, находящийся слева от последнего символа v, имеет свойство быть □ и иметь наименьший индекс среди всех этих ячеек ленты, удовлетворяя двум условиям: (i) он находится справа от головки; (ii) справа от него нет других символов, кроме □.

Если $\delta(q_i, b) = (q_j, c, L)$, то конфигурация uaq_ibv порождает конфигурацию uq_jacv, и если $\delta(q_i, b) = (q_j, c, R)$, то uaq_ibv порождает $uacq_jv$. Иногда выражение «C_1 порождает C_2» записывается как $C_1 \to C_2$. Мы исходим из того, что машина Тьюринга останавливается, когда она входит в принимающее или отклоняющее состояние, и мы определяем *язык машины Тьюринга M*, обозначаемый $L(M)$, следующим образом: $L(M) = \{w \in \sum^* | q_0w \overset{*}{\Rightarrow} \alpha q_{\text{accept}}\beta\}$.

Задача 8.55. Спроектируйте машину Тьюринга M такую, что $L(M)$ является языком палиндромов.

Разные варианты машин Тьюринга эквивалентны; это понятие называется *робастностью*. Например, лента бесконечна только в одном направлении, либо несколько лент. Между разными моделями легко выполнить «трансляцию».

Принимаемые машинами Тьюринга языки называются *рекурсивно перечислимыми* (recursively enumerable, RE), или *распознаваемыми*, или *распознаваемыми по Тьюрингу* (в Sipser). Язык L является рекурсивно перечислимым, если существует машина Тьюринга M, которая останавливается в принимающем состоянии для всех $x \in L$ и не принимает $x \notin L$. Другими словами, L является рекурсивно перечислимым, если существует M такая, что $L = L(M)$ (но M не обязательно останавливается на всех входах).

Язык L является *рекурсивным*, или *разрешимым*, или *разрешимым по Тьюрингу* (в Sipser), если существует машина Тьюринга M, которая останавливается в q_{accept} для всех $x \in L$ и останавливается в q_{reject} для всех $x \notin L$. Другими словами, L является разрешимым, если существует машина Тьюринга M такая, что $L = L(M)$ (то есть M распознает/принимает L), а также M всегда останавливается. Рекурсивные языки соответствуют языкам, которые могут распознаваться *алгоритмически*.

8.5.1. Недетерминированные машины Тьюринга

Напомним, что в разделе 8.3.2 мы дали определение недетерминированным конечным автоматам. Недетерминизм допускает возможность нескольких возмож-

ных ходов на одной и той же конфигурации. Эта идея сейчас используется в контексте машин Тьюринга.

Недетерминированная машина Тьюринга похожа на нормальную машину Тьюринга, за исключением того, что переходная функция теперь является переходным отношением; тем самым существует несколько возможных ходов на данном состоянии и символе:

$\delta(q, a) = \{(q_1, b_1, D_1), (q_2, b_2, D_2), \ldots, (q_k, b_k, D_k)\}$.

Так же, как и для недетерминированного конечного автомата, недетерминизм не усиливает модель вычислений, по крайней мере, не в контексте разрешимости. Но он допускает более удобный проектный формализм.

Например, рассмотрим машину Тьюринга N, которая определяет следующий язык $L(N) = \{w \in \{0, 1\}^* |$ последний символ w равен 1$\}$. Описание N вместе с вычислительным деревом машины N на входных данных 011 можно найти на рис. 8.15.

Теорема 8.56. Если N является недетерминированной машиной Тьюринга, то существует детерминированная машина Тьюринга D такая, что $L(N) = L(D)$.

Доказательство. D пробует все возможные ходы из N, используя поиск «сначала в ширину». D поддерживает последовательность конфигураций на ленте 1:

...	config_1	config_2	config_1^*	...

и использует вторую ленту для черновых заметок. Конфигурация, отмеченная символом «*», является текущей конфигурацией. D копирует ее на вторую ленту и проверяет, является ли она принимающей. Если это так, то машина принимает. Если это не так и N имеет k возможных ходов, то D добавляет k новых конфигураций, получаемых в результате этих ходов на ленте 1, и помечает следующую конфигурацию в списке как текущую. Если максимальное число возможных вариантов N равно m, то есть m – это *степень недетерминированности* N, и N делает n ходов перед принятием, то D исследует $1 + m + m^2 + m^3 + \cdots + m^n \approx nm^n$ конфигураций. □

$\delta(q_0, 0) = \{(q_0, 0, \rightarrow), (q, 0, \rightarrow)\}$
$\delta(q_0, 1) = \{(q_0, 1, \rightarrow), (r, 1, \rightarrow)\}$
$\delta(r, \square) = \{(q_{\text{accept}}, \square, \rightarrow)\}$
$\delta(r, 0/1) = \{(q, 0, \rightarrow)\}$

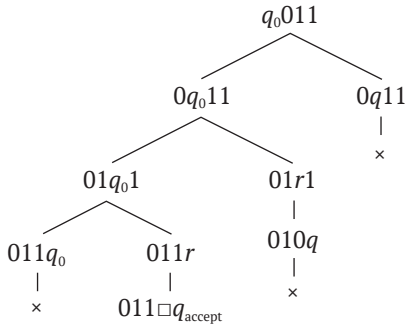


Рис. 8.15 ❖ Определение N вместе с выполнением на 011

Фактически в приведенном выше доказательстве происходит то, что D *симулирует* N; идея симуляции будет важной нитью в теме вычислимости. У нас всегда может быть одна машина Тьюринга, которая симулирует другую; «другая» машина Тьюринга может быть закодирована в состояниях симулятора. Это неудиви-

тельно, поскольку машина Тьюринга является «конечным объектом», который может быть «закодирован» конечным числом символов (подробнее об этом ниже). Кроме того, описание «другой» машины можно поместить на ленту, и симулятор проверяет это описание для симулирования каждого хода на другой выделенной ленте. Короче говоря, тот факт, что это можно сделать, не должен удивлять, учитывая, что машины Тьюринга выражают то, что мы понимаем под термином «компьютер». Кроме того, у нас также есть симуляторы в «реальном мире» – например, мы можем использовать программу `VMware` для симулирования ОС Windows на компьютере под управлением ОС Linux.

Задача 8.57. Покажите, как M_1 может симулировать M_2. Одна идея состоит в том, чтобы иметь состояния (s_{on}, p) и (s_{off}, p), где некоторые из p находятся в Q_{M_2} и некоторые соответствуют действиям M_1. Здесь s_{on}, s_{off} указывают на то, включена симуляция или нет, и состояния машины M_1 являются такими парами.

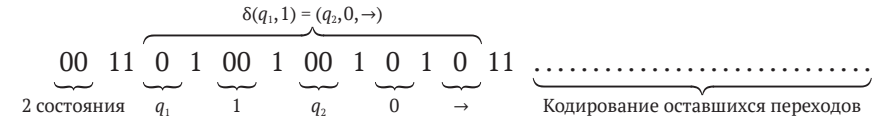
8.5.2. Варианты кодирования

Напомним, что детерминированный конечный автомат B представляет собой кортеж $(Q, \sum, \delta, q_0, F)$; мы исходим из того, что $\sum = \{0,1\}$ и $Q = \{q_1, q_2, \ldots, q_n\}$, где q_0 всегда равно q_1. Будем также считать, что $F = \{q_{i_1}, \ldots, q_{i_k}\}$. Тогда

$$\langle B\rangle := 0^n 10^{l_1^0} 10^{l_1^1} 10^{l_2^0} 10^{l_2^1} 1 \ldots 0^{l_n^0} 10^{l_n^1} 10^{i_1} 10^{i_2} 1 \ldots 10^{i_k},$$

где $0^{l_j^0} 10^{l_j^1}$ означает, что на q_j детерминированный конечный автомат B переходит в q_{l_0} на 0 и в q_{l_1} на 1, начальное 0^n обозначает, что существует n состояний, а конечное $0^{i_1} 10^{i_2} 1 \ldots 10^{i_k}$ обозначает принимающие состояния. Обратите внимание, что в этом представлении нет двух подряд 1, поэтому $\langle B, w\rangle := \langle B\rangle 11 w$ является хорошей кодировкой пары (B, w), так как кодировка B, $\langle B\rangle$ и кодировка w разделены символами 11.

Схожим образом мы можем закодировать каждую машину Тьюринга строкой над {0,1}. Например, если M является машиной Тьюринга $(\{q_1, q_2\}, \{0, 1\}, \delta, \ldots)$ и $\delta(q_1, 1) = (q_2, 0, \rightarrow)$ является одним из переходов, то она может быть закодирована как



Не каждая строка будет валидной кодировкой машины Тьюринга; например, в нашем обозначении строка 1 ничего не кодирует. Будем говорить, что строка $x \in \{0, 1\}^*$ является *хорошо сформированной строкой*, если существует машина Тьюринга M и строки w, такие что $x = \langle M, w\rangle$; это означает, что x является надлежащей кодировкой пары (M, w). Легко увидеть, что мы можем создать решатель, который проверяет, что x является хорошо сформированной строкой, или, другими словами, что язык хорошо сформированной строки разрешим.

8.5.3. Разрешимость

Теорема 8.58. Регулярные языки разрешимы, и контекстно-свободные языки также разрешимы.

Ниже приведены примеры разрешимых языков:

- $A_{\text{DFA}} := \{\langle B, w\rangle : B$ – это детерминированный конечный автомат, который принимает входную строку $w\}$;
- $A_{\text{NFA}} := \{\langle B, w\rangle : B$ – это недетерминированный конечный автомат, который принимает входную строку $w\}$;
- $A_{\text{REX}} := \{\langle R, w\rangle : R$ – это регулярное выражение, которое принимает входную строку $w\}$;
- $E_{\text{DFA}} := \{\langle A\rangle : A$ – это детерминированный конечный автомат такой, что $L(A) = \emptyset\}$;
- $EQ_{\text{DFA}} := \{\langle A, B\rangle : A, B$ – это детерминированные конечные автоматы такие, что $L(A) = L(B)\}$;
- $E_{\text{CFG}} := \{\langle G\rangle : G$ – это контекстно-свободная грамматика такая, что $L(G) = \emptyset\}$.

Для EQ_{DFA} используется симметрическая разность: $C = (A \cap \overline{B}) \cup (\overline{A} \cap B)$.

Теорема 8.59. Если L разрешим, то и его дополнение тоже разрешимо.

Доказательство. Пусть $\overline{L} = \sum^* - L$ равно дополнению L, и предположим, что L разрешим машиной M. Пусть M' равно следующей модификации M: при подаче x на вход M' работает так же, как M. Однако когда M собирается принять, M' отклоняет, и когда M собирается отклонить, M' принимает. Ясно, что $\overline{L} = L(M')$. Поскольку M всегда останавливается, то так же делает и M', поэтому по определению L разрешим. □

Теорема 8.60. Если L и $\overline{L}$ оба являются рекурсивно перечислимыми, то L разрешим.

Доказательство. Пусть $L = L(M_1)$ и $\overline{L} = L(M_2)$. Пусть M равно следующей машине Тьюринга: при подаче x на вход она симмулирует M_1 на x на одной ленте и M_2 на x на другой ленте. Поскольку $L \cup \overline{L} = \sum$, x должны быть приняты одним либо другим. Если M_1 принимает, то то же делает и M; если M_2 принимает, то M отклоняет. □

8.5.4. Тезис Черча–Тьюринга

> *Интуитивное понятие алгоритма выражается через формальное определение машины Тьюринга.*

Это положение называется «тезисом», потому что понятие алгоритма интуитивно, то есть расплывчато. У всех нас есть интуитивное понимание понятия «алгоритм» как рецепта, процедуры, набора инструкций, которые для любых входных данных определенного вида дают желаемый результат.

Рассмотрим язык:

$A_{\text{TM}} = \{\langle M, w\rangle : M$ – это машина Тьюринга, и M принимает $w\}$.

Этот язык иногда называют *универсальным языком*. Он является распознаваемым, поскольку он распознается *универсальной машиной Тьюринга*; это такая машина, которая при подаче на вход $\langle M, w\rangle$, где M – это машина Тьюринга и w – строка, проверяет, что $\langle M, w\rangle$ представляет собой хорошо сформированную строку, и если это так, она симулирует M на w и дает ответы соответственно тому, как отвечает $M(w)$. Вместе с тем обратите внимание, что U не решает A_{TM}.

Универсальная машина была революционной идеей Тьюринга; она представляла собой концепцию, которая шла вразрез с инженерными принципами своего времени. В эпоху Тьюринга инженерная практика заключалась в том, чтобы иметь «одну машину для решения одной задачи». Тем самым иметь универсальную машину, которая способна решать «все» задачи, означало плыть против течения. Но наши современные компьютеры – это именно универсальные машины Тьюринга, то есть мы не строим компьютер для выполнения *одного* алгоритма, а наоборот – наши компьютеры могут выполнять все то, что мы на них *запрограммируем*.

Нетрудно увидеть, что универсальная машина Тьюринга может быть сконструирована, но необходимо позаботиться о том, чтобы установить правила кодирования машин Тьюринга и правила для кодировки $\langle M, w\rangle$. Универсальная машина Тьюринга может иметь несколько лент, одна из которых зарезервирована для $\langle M\rangle$, то есть лента, содержащая «программу», а другая лента, на которой U имитирует вычисление $M(w)$. В 1960-х годах Марвин Мински, глава отдела искусственного интеллекта в Массачусетском технологическом институте, предложил в то время самую маленькую универсальную машину Тьюринга: 7 состояний и 2 символа. В настоящее время с 2008 года рекорд принадлежит Алексу Смиту, предложившему универсальную машину Тьюринга с 2 состояниями и 3 символами.

Проектирование универсальной машины Тьюринга является по-настоящему важным упражнением для тех, кто серьезно занимается информатикой. Такое упражнение включает в себя разработку языка программирования (то есть (M)) и интерпретатора/компилятора (то есть U, способного *симулировать* любой M с любыми данными на входе). Такое упражнение материализует многие понятия информатики и делает их более узнаваемыми, когда они появляются позже.

8.5.5. Неразрешимость

Теорема 8.61. A_{TM} является неразрешимым.

Доказательство. Предположим, что он является разрешимым и что H его решает. Тогда $L(H) = A_{TM}$, и H всегда останавливается (заметим, что $L(H) = L(U)$, но, как мы уже говорили, не гарантируется, что U является решателем). Определим новую машину D (здесь D означает «диагональ», так как этот аргумент следует «диагональному аргументу Кантора»):

$$D(\langle M\rangle) := \begin{cases} \text{принять}, & \text{если } H(\langle M, \langle M\rangle\rangle) \text{ – отклонить}, \\ \text{отклонить}, & \text{если } H(\langle M, \langle M\rangle\rangle) \text{ – принять}, \end{cases}$$

то есть D делает «противоположное». Тогда мы видим, что $D(\langle D\rangle)$ принимает тогда и только тогда, когда она отклоняет. Противоречие; поэтому A_{TM} не может быть разрешимым. □

Каково практическое следствие этой теоремы? Представьте, что вы разрабатываете отладчик для некого языка программирования – что-то в стиле GDB[1] для C. По словам команды «Проекта GNU», отладчик GDB позволяет вам видеть, что происходит «внутри» другой программы во время ее выполнения – или что другая программа делала в момент сбоя. Очень полезной функциональной возможностью будет запрос отладчика о том, остановится ли ваша программа на тех или

[1] См. https://www.gnu.org/software/gdb/. – *Прим. перев.*

иных входных данных. Например, вы выполнили свою программу на некоторых входных данных x, и ничего не происходило в течение длительного времени, пока не было нажато сочетание клавиш **Ctrl**+**D** для прерывания исполнения. Не нажали ли вы слишком быстро? Возможно, если бы вы подождали чуть дольше, то ваш ответ пришел бы; или, возможно, она никогда бы не остановилась сама по себе. «Возможность остановки» в отладчике даст вам ответ. Однако теорема 8.61 говорит, что эта функциональная возможность *не может быть* реализована.

Давайте убедимся, что мы понимаем то, что утверждает теорема 8.61: она говорит, что A_{TM} неразрешим и поэтому нет машины Тьюринга, которая останавливается на любом $\langle M, w\rangle$ с правильным ответом. Это не отменяет возможности разработки машины Тьюринга, которая останавливается на некоторых (возможно, даже бесконечно многих) $\langle M, w\rangle$ с правильным ответом. Теорема 8.61 гласит, что не существует алгоритма, который работает правильно для любых входных данных.

См. статью Моше Варди (Moshe Vardi), касающуюся завершения/неразрешимости, из июльского номера ACM Communications за 2011 год «Решение неразрешимого» (Solving the unsolvable).

Задача 8.62. Существует ли машина Тьюринга M такая, что $L(M) = A_{TM} - L'$, где $|L'| < \infty$? То есть M решает A_{TM} для всех, кроме конечного числа $\langle M, w\rangle$.

Функция трудолюбивого бобра (busy beaver, BB)[1], $\sum(n, m)$, выдает максимальное число ячеек, которые могут быть записаны с помощью машины Тьюринга с n состояниями и M символами алфавита, начинающимися на пустой ленте. Задав $m = 2$ и приняв $\sum(n)$ равным $\sum(n, 2)$, известно, что $\sum(2) = 4$; $\sum(3) = 6$; $\sum(4) = 13$; $\sum(5) \geqslant 4098$; $\sum(6) \geqslant 3{,}5\times10^{18267}$. Функция трудолюбивого бобра неразрешима; предположим, что она разрешима. Тогда мы могли бы использовать ее для того, чтобы решить A_{TM} следующим образом: передав на вход $\langle M, w\rangle$, конструируем машину Тьюринга M', которая на пустой ленте пишет w, и возвращается в первую ячейку, и симулирует M на w. Затем вычисляем $i = \sum(|Q_{M'}|, |\Gamma_{M'}|)$ и симулируем M'. Если M' хоть раз пересекает i-ю ячейку, то мы знаем, что $M(w)$ не останавливается; если M' ограничена в пределах первых i ячеек, то она либо остановится, либо войдет в «цикл». Мы можем обнаружить эту «петлю», отслеживая разные конфигурации и убеждаясь, что они не повторяются. При ограниченном пространстве число конфигураций ограничено $|Q_M|i|\Gamma|^i$.

Следствие 8.63. $\overline{A_{TM}}$ не является рекурсивно перечислимым.

Доказательство. Поскольку A_{TM} является рекурсивно перечислимым (так как $L(U) = A_{TM}$), по теореме 8.60 мы знаем, что если бы $\overline{A_{TM}}$ был также рекурсивно перечислимым, то A_{TM} был бы разрешимым, что по теореме 8.61 не так. □

Перечислитель – это машина Тьюринга, которая имеет рабочую ленту, пустую на входе, и выходную ленту, на которой она записывает строки, разделенные не-

[1] Трудолюбивый бобер (busy beaver) – это машина Тьюринга с заранее заданным числом состояний N, которая, начиная с пустой ленты (все нули), генерирует самую длинную последовательность единиц (или в другой формулировке: выполняет как можно больше шагов), после чего останавливается. См. https://pl.wikipedia.org/wiki/Pracowity_b%C3%B3br. – *Прим. перев.*

которыми символами, скажем символом #, никогда не перемещаясь влево. Идея состоит в том, что она «пересчитывает» строки на языке. Язык является *перечислимым*, если существует перечислитель E такой, что $L = L(E)$.

Теорема 8.64. Язык распознаваем тогда и только тогда, когда он перечислим.

Доказательство. Если язык L перечислим, то пусть M симулирует на входных данных w перечислитель языка L и принимает, если на входе появляется w. Для другого направления мы должны быть осторожнее: предположим, что L распознаваем машиной M. Пусть перечислитель E работает следующим образом: в фазе i он симулирует M на первых i строках $\sum^*$ (в *лексикографическом порядке*), каждая для i шагов. Когда M принимает некоторую строку, E выдает ее. В этом заключается идея согласования по типу «ласточкина хвоста»[1]. □

8.5.6. Редукции

Используя понятие редукции, или сведения, мы можем показать, что многие другие языки не являются рекурсивно перечислимыми или неразрешимыми. Рассмотрим язык:

HALT_{TM}: = {⟨M, w⟩ : M является машиной Тьюринга, которая останавливается на w}.

Этот язык неразрешим, и мы можем показать это следующим образом: предположим, что он разрешим и что его решателем является H. Рассмотрим H', который, получая на входе ⟨M, w⟩, выполняет H(⟨M, w⟩). Если H принимает, то наш H' симулирует M на w и отвечает соответствующим образом; в противном случае H' отклоняет. Ясно, что $L(H') = \text{A}_{\text{TM}}$, но поскольку H был решателем, то таковым является и H'. Но это противоречит неразрешимости A_{TM}. Следовательно, мы только что показали от противного, что HALT_{TM} не может быть разрешимым, то есть он неразрешим.

Рассмотрим сейчас

E_{TM}: = {⟨M⟩: M является машиной Тьюринга такой, что $L(M) = \emptyset$}.

Этот язык неразрешим: предположим, что E_{TM} разрешим; пусть R равно машине Тьюринга, которая его решает. Рассмотрим машину Тьюринга R', спроектированную следующим образом: получая на входе ⟨M, w⟩, она сначала конструирует машину M_w, где на x машина M_w сначала проверяет, является ли истинным, что $x = w$; если нет, то M_w отклоняет. В противном случае M_w выполняет M на w и принимает, если принимает M. Наконец, R' симулирует R на ⟨M_w⟩. Ясно, что $L(R') = \text{A}_{\text{TM}}$, и поскольку R' является решателем, то этого не может быть.

Рассмотрим язык

$\text{REGULAR}_{\text{TM}}$:= {⟨M⟩ : M является машиной Тьюринга и $L(M)$ является регулярным}.

Этот язык неразрешим; предположим, что он был разрешимым, и R – ее решатель. Мы проектируем S следующим образом: получая на входе ⟨M, w⟩, S сначала конструирует M', которая работает следующим образом: M' при подаче на вход x проверяет, имеет ли x форму 0^n1^n, и принимает, если да. Если x не имеет этой фор-

[1] Ласточкин хвост (dovetailing) – это когда симулируется две или более машины Тьюринга параллельно на одной машине Тьюринга. – *Прим. перев.*

мы, то она выполняет M на w и принимает, если M принимает. Наконец, S выполняет R на $\langle M_2 \rangle$. Обратите внимание, что $L(M')$ является либо нерегулярным языком $\{0^n1^n\}$ (если M отклоняет w), либо регулярным языком $\{0,1\}^*$ (если M принимает w).

8.5.7. Теорема Райса

Оказывается, что нетривиальные свойства языков машин Тьюринга неразрешимы. Что мы подразумеваем под «нетривиальными свойствами»? Мы имеем в виду, например, свойство не принимать никакие строки, то есть E_{TM}.

Более формально, *свойство* $\mathcal{P}$ – это всего лишь подмножество $\{\langle M \rangle : M$ является машиной Тьюринга$\}$. Мы говорим, что свойство является нетривиальным, если $\mathcal{P} \neq \emptyset$ и $\overline{\mathcal{P}} \neq \emptyset$. Далее, нам потребуется следующее: если заданы две машины Тьюринга M_1 и M_2 такие, что $L(M_1) = L(M_2)$, тогда либо обе $\langle M_1 \rangle$ и $\langle M_2 \rangle$ находятся в $\mathcal{P}$, либо обе не находятся в $\mathcal{P}$. То есть, находится или нет $\langle M \rangle$ в $\mathcal{P}$, зависит только от свойств языка M, а не от, скажем, синтаксических свойств машины, таких как число состояний.

Теорема 8.65 (Райса). Каждое нетривиальное свойство неразрешимо.

8.5.8. Задача соответствий Поста

Напомним, что теорема Майхилла–Нероуда (раздел 8.3.8.2) дает характеристику регулярных языков без упоминания модели вычислений. Задача Поста делает то же самое для неразрешимых языков; она дает пример конкретного неразрешимого языка без упоминания машин Тьюринга или любой другой модели вычислений. Она показывает, что неразрешимость является не причудой конкретной модели вычислений, а немутируемым свойством некоторых языков.

Экземпляр *задачи соответствий Поста* (Post's correspondence problem, PCP) состоит из двух конечных списков строк над некоторым алфавитом $\sum$. Два списка должны быть одинаковой длины:

$A = w_1, w_2, \ldots, w_k$;
$B = x_1, x_2, \ldots, x_k$.

Для каждого i пара (w_i, x_i) называется *соответствующей парой*. Мы говорим, что этот экземпляр задачи соответствий Поста имеет решение, если существует последовательность одного или нескольких индексов:

$$i_1, i_2, \ldots, i_m, \quad m \geqslant 1,$$

где индексы могут повторяться так, что:

$$w_{i_1} w_{i_2} \ldots w_{i_m} = x_{i_1} x_{i_2} \ldots x_{i_m}.$$

Задача соответствий Поста состоит в следующем: при наличии двух списков (A, B) равной длины имеет ли она решение? Мы можем выразить задачу соответствий Поста как язык:

$L_{PCP} := \{\langle A, B \rangle | (A, B)$ экземпляр задачи соответствий Поста с решением$\}$.

Например, рассмотрим (A, B):

$A = 1, 10111, 10$;
$B = 111, 10, 0$.

Тогда $i_1 = 2$, $i_2 = 1$, $i_3 = 1$, $i_4 = 3$ является решением в виде:

$$\underbrace{10111}_{w_2}\underbrace{1}_{w_1}\underbrace{1}_{w_1}\underbrace{10}_{w_3} = \underbrace{10}_{x_2}\underbrace{111}_{x_1}\underbrace{111}_{x_1}\underbrace{0}_{x_3}.$$

Обратите внимание, что $i_1 = 2$, $i_2 = 1$, $i_3 = 1$, $i_4 = 3$, $i_5 = 2$, $i_6 = 1$, $i_7 = 1$, $i_8 = 3$ составляет еще одно решение.

Задача 8.66. Покажите, что $A = 10, 011, 101$ и $B = 101, 11, 011$ не имеет решения.

Модифицированная задача соответствий Поста (modified PCP) имеет дополнительное требование в том, что первая пара в решении должна быть первой парой (A, B). Поэтому $i_1, i_2, \ldots, i_m$, $m \geqslant 0$ является решением для (A, B) экземпляра модифицированной задачи соответствий Поста, если:

$$w_1 w_{i_1} w_{i_2} \ldots w_{i_m} = x_1 x_{i_1} x_{i_2} \ldots x_{i_m}.$$

Мы также говорим, что $i_1, i_2, \ldots, i_r$ является *частичным решением* (модифицированной) задачи соответствий Поста, если один из следующих префиксов является префиксом другого:

$$(w_1) w_{i_1} w_{i_2} \ldots w_{i_r} \qquad (x_1) x_{i_1} x_{i_2} \ldots x_{i_r}.$$

В случае модифицированной задачи соответствий Поста мы далее требуем, чтобы $i_1 = 1$.

С учетом всех этих элементов мы можем теперь показать, что задача соответствий Поста является неразрешимой. Мы собираемся сделать это в три этапа: во-первых, мы покажем, что если задача соответствий Поста разрешима, то неразрешима и модифицированная задача соответствий Поста. Во-вторых, мы показываем, что если модифицированная задача разрешима, то разрешим и A_{TM}. В-третьих, так как A_{TM} неразрешим, то неразрешима и (модифицированная) задача соответствий Поста.

Лемма 8.67. Если задача соответствий Поста разрешима, то разрешима и модифицированная задача соответствий Поста.

Доказательство. Мы показываем, что с учетом экземпляра (A, B) модифицированной задачи соответствий Поста мы можем создать экземпляр (A', B') задачи соответствий Поста такой, что:

(A, B) имеет решение $\Leftrightarrow$ (A', B') имеет решение.

Пусть (A, B) равно экземпляру модифицированной задачи соответствий Поста над алфавитом $\sum$. Тогда (A', B') является экземпляром задачи соответствий Поста над алфавитом $\sum' = \sum \cup \{*, \$\}$, где $*$, $ – это новые символы.

Если $A = w_1, w_2, w_3, \ldots, w_k$, то $A' = *\bar{w}_1*, \bar{w}_1*, \bar{w}_2*, \bar{w}_3*, \ldots, \bar{w}_k*, \$$.

Если $B = x_1, x_2, x_3, \ldots, x_k$, то $B' = *\bar{x}_1, *\bar{x}_1, *\bar{x}_2, *\bar{x}_3, \ldots, *\bar{x}_k, *\$$, где если $x = a_1 a_2 a_3 \ldots a_n \in \sum^*$, то $\bar{x} = a_1 * a_2 * a_3 * \ldots * a_n$.

Например, если (A, B) является экземпляром, если модифицированная задача соответствий Поста задана как: $A = 1, 10111, 10$ и $B = 111, 10, 0$, то (A', B') является экземпляром задачи соответствий Поста, задаваемой парой: $A' = *1*, 1*, 1*0*1*1*1*, 1*0*, \$$ и $B' = *1*1*1, *1*1*1, *1*0, *0, *\$$. Задача 8.68 заканчивает доказательство. □

Задача 8.68. Закончите доказательство леммы 8.5.8.

Лемма 8.69. Если модифицированная задача соответствий Поста разрешима, то разрешим A_{TM}.

Доказательство. С учетом пары (M, w), мы конструируем экземпляр (A, B) модифицированной задачи соответствий Поста такой, что:

ТМ M принимает $w \Leftrightarrow (A, B)$ имеет решение.

Основная идея заключается в следующем: экземпляр модифицированной задачи соответствий Поста (A, B) симулирует в своих частичных решениях вычисление M на w. То есть частичные решения будут иметь вид:

$$\#\alpha_1\#\alpha_2\#\alpha_3\# \ldots,$$

где α_1 – это начальная конфигурация M на w, и для всех i конфигурация A дает конфигурацию α_{i+1}.

Частичное решение из списка B всегда будет «на одну конфигурацию впереди» списка A; списку A будет разрешено «догонять» только тогда, когда M принимает w. Для упрощения мы исходим из того, что машины Тьюринга не печатают пустые символы (то есть они не печатают «□») с целью, чтобы конфигурации имели форму $\alpha q \beta$, где $\alpha, \beta \in (\Gamma - \{\square\})*$ и $q \in Q$.

Задача 8.70. Покажите, что машина Тьюринга, которая не может печатать пустые символы, эквивалентна по мощности тем машинам Тьюринга, которые могут их печатать.

Пусть M равно машине Тьюринга и $w \in \sum^*$; мы конструируем экземпляр (A, B) модифицированной задачи соответствий Поста следующим образом:

1) A: #
   B: $\#q_0w\#$;
2) A: $a_1, a_2, \ldots, a_n, \#$
   B: $a_1, a_2, \ldots, a_n, \#$
   где $a_i \in (\Gamma - \{\square\})^*$;
3) для симулирования хода M для всех $q \in Q - \{q_{accept}\}$:

Список A	Список B	
qa	bp	если $\delta(q, a) = (p, b, \rightarrow)$
cqa	pcb	если $\delta(q, a) = (p, b, \leftarrow)$
$q\#$	$bp\#$	если $\delta(q, \square) = (p, b, \rightarrow)$
$cq\#$	$pcb\#$	если $\delta(q, \square) = (p, b, \leftarrow)$

4) если конфигурация в конце B принимает (то есть имеет форму $\alpha q_{accept} \beta$), то нам нужно разрешить A догнать B. Таким образом, для всех $a, b \in (\Gamma - \{\square\})^*$ нам нужны следующие соответствующие пары:

Список A	Список B
$aq_{accept}b$	q_{accept}
aq_{accept}	q_{accept}
$q_{accept}b$	q_{accept}

5) наконец, после использования указанных выше пунктов 4 и 3 мы получаем $x\#$ и $x\#q_{accept}\#$, где x – это длинная строка. Таким образом, для того чтобы догнать, нам нужна $q_{accept}\#\#$ в A и # в B. □

Например, рассмотрим следующую машину Тьюринга M с состояниями $\{q_1, q_2, q_3\}$, где q_1 сокращает q_{init}, q_3 сокращает q_{accept} и где δ задается таблицей переходов:

	0	1	□
q_1	$(q_2, 1, \rightarrow)$	$(q_2, 0, \leftarrow)$	$(q_2, 1, \leftarrow)$
q_2	$(q_3, 1, \leftarrow)$	$(q_1, 0, \rightarrow)$	$(q_2, 0, \rightarrow)$

Из этого M и ввода $w = 01$ мы получаем следующую модифицированную задачу соответствий Поста:

Правило	Список A	Список B	Источник
1	#	$\#q_1 01\#$	
2	0 1 #	0 1 #	
3	$q_1 0$ $0q_1 1$ $1q_1 1$ $0q_1\#$ $1q_1\#$ $0q_2 0$ $1q_2 0$ $q_2 1$ $q_2\#$	$1q_2$ $q_2 00$ $q_2 10$ $q_2 01\#$ $q_2 11\#$ $q_3 00$ $q_3 10$ $0q_1$ $0q_2\#$	$\delta(q_1, 0) = (q_2, 1, \rightarrow)$ $\delta(q_1, 1) = (q_2, 0, \leftarrow)$ $\delta(q_1, 1) = (q_2, 0, \leftarrow)$ $\delta(q_1, B) = (q_2, 1, \leftarrow)$ $\delta(q_1, B) = (q_2, 1, \leftarrow)$ $\delta(q_2, 0) = (q_3, 0, \leftarrow)$ $\delta(q_2, 0) = (q_3, 0, \leftarrow)$ $\delta(q_2, 1) = (q_1, 0, \rightarrow)$ $\delta(q_2, B) = (q_2, 0, \rightarrow)$
4	$0q_3 0$ $0q_3 1$ $1q_3 0$ $1q_3 1$ $0q_3$ $1q_3$ $q_3 0$ $q_3 1$	q_3 q_3 q_3 q_3 q_3 q_3 q_3 q_3	
5	$q_3\#\#$	#	

Машина Тьюринга M на входе принимает $w = 01$ последовательностями ходов, представленных следующими конфигурациями:

$q_1 01 \rightarrow 1q_2 1 \rightarrow 10q_1 \rightarrow 1q_2 01 \rightarrow q_3 101.$

Мы исследуем последовательность частичных решений, которая имитирует это вычисление M на w и в конечном итоге приводит к решению. Мы должны начать с первой пары (модифицированной задачи соответствий Поста):

A : #
B : $\#q_1 01\#$

Единственный способ расширить это частичное решение – использовать соответствующую пару $(q_1 0, 1q_2)$, поэтому мы получаем:

A : $\#q_1 0$
B : $\#q_1 01\#1q_2$

Теперь с помощью копирования пар мы получаем:

A : $\#q_1 01\#1$
B : $\#q_1 01\#1q_2 1\#1$

Следующая соответствующая пара $(q_2 1, 0q_1)$:

A : $\#q_1 01\#1q_2 1$
B : $\#q_1 01\#1q_2 1\#10q_1$

Теперь осторожно! Мы копируем только следующие два символа, в результате получив:

A : $\#q_1 01\#1q_2 1\#1$
B : $\#q_1 01\#1q_2 1\#10q_1\#1$

потому что нам нужно $0q_1$, так как считывающе-записывающая головка движется влево, и мы используем следующую соответствующую пару ($0q_1\#$, $q_2 01\#$) и получаем:

A : $\#q_1 01\#1q_2 1\#10q_1\#$
B : $\#q_1 01\#1q_2 1\#10q_1\#1q_2 01\#$

Теперь мы можем сразу использовать еще одну соответствующую пару ($1q_2 0$, $q_3 10$), в результате получив:

A : $\#q_1 01\#1q_2 1\#10q_1\#1q_2 0$
B : $\#q_1 01\#1q_2 1\#10q_1\#1q_2 01\#q_3 10$

и обратите внимание, что у нас есть принимающее состояние! Мы используем две копирующие пары, в результате получив:

A : $\#q_1 01\#1q_2 1\#10q_1\#1q_2 01\#$
B : $\#q_1 01\#1q_2 1\#10q_1\#1q_2 01\#q_3 101\#$

и теперь мы можем начать использовать правила в 4, чтобы A догнало B:

A : ... $\#q_3 1$
B : ... $\#q_3 101\#q_3$

и мы копируем три символа:

A : ... $\#q_3 101\#$
B : ... $\#q_3 101\#q_3 01\#$

И снова немного наверстываем:

A : ... $\#q_3 101\#q_3 0$
B : ... $\#q_3 101\#q_3 01\#q_3$

Комируем два символа:

A : ... $\#q_3 101\#q_3 01\#$
B : ... $\#q_3 101\#q_3 01\#q_3 1\#$

и догоняем:

A : ... $\#q_3 101\#q_3 01\#q_3 1$
B : ... $\#q_3 101\#q_3 01\#q_3 1\#q_3$

и компируем:

A : ... $\#q_3 101\#q_3 01\#q_3 1\#$
B : ... $\#q_3 101\#q_3 01\#q_3 1\#q_3\#$

И теперь все заканчиваем соответствующей парой ($q_3\#\#$, $\#$), заданной правилом 5, получив совпадающие строки:

A : ... $\#q_3 101\#q_3 01\#q_3 1\#q_3\#\#$
B : ... $\#q_3 101\#q_3 01\#q_3 1\#q_3\#\#$

Таким образом, с учетом экземпляра $\langle M, w\rangle$ автомата A_{TM} мы строим экземпляр $(A, B)_{\langle M, w\rangle}$ модифицированной задачи соответствий Поста с целью, чтобы соблюдалось следующее отношение:

M принимает $w \Leftrightarrow (A, B)_{\langle M, w\rangle}$ имеет решение. (8.2)

Другими словами, мы свели A_{TM} к модифицированной задаче соответствий Поста, и наше сведение задается (вычислимой) функцией $f : \{0, 1\}^* \to \{0, 1\}^*$, которая определяется следующим образом: $f(\langle M, w\rangle) = \langle (A, B)_{\langle M, w\rangle}\rangle$. Это показывает, что если модифицированная задача соответствий Поста разрешима, то разрешим и A_{TM}. Для того чтобы это увидеть, предположим, что модифицированная задача соответствий Поста разрешима; тогда у нас есть решатель для A_{TM}: получив на вход $\langle M, w\rangle$, наш решатель вычисляет $x = f(\langle M, w\rangle)$ и выполняет решатель для модифицированной задачи соответствий Поста на x. По (8.2) мы знаем, что ответ «да» означает, что M принимает w.

8.5.9. Неразрешимые свойства контекстно-свободных языков

Теперь мы можем использовать тот факт, что задача соответствий Поста неразрешима, для того чтобы показать, что ряд вопросов о контекстно-свободных языках неразрешим. Пусть (A, B) равно экземпляру задачи соответствий Поста, где $A = w_1, w_2, \ldots, w_k$ и $B = x_1, x_2, \ldots, x_k$. Пусть G_A и G_B связаны с контекстно-свободными языками, заданными:

$A \to w_1 A a_1 | w_2 A a_2 | \cdots | w_k A a_k | w_1 a_1 | w_2 a_2 | \cdots | w_k a_k$;
$B \to x_1 B a_1 | x_2 B a_2 | \cdots | x_k B a_k | x_1 a_1 | x_2 a_2 | \cdots | x_k a_k$,

где $a_1, a_2, \ldots, a_k$ – это новые символы не в алфавите (A, B).

Пусть $L_A = L(G_A)$ и $L_B = L(G_B)$, и поэтому L_A и L_B состоят соответственно из всех строк формы:

$w_{i_1} w_{i_2} \ldots w_{i_m} a_{i_m} \ldots a_{i_2} a_{i_1}$;
$x_{i_1} x_{i_2} \ldots x_{i_m} a_{i_m} \ldots a_{i_2} a_{i_1}$.

Теорема 8.71. Вопрос, является ли контекстно-свободный язык неоднозначным, неразрешим.

Доказательство. Пусть G_{AB} равно контекстно-свободной грамматике, состоящей из G_A, G_B, с вброшенным правилом $S \to A|B$. Таким образом, G_{AB} является неоднозначной $\Leftrightarrow$ задача соответствий Поста (A, B) имеет решение. Обратите внимание, что целью новых символов a_i в G_A и G_B является обеспечение того, чтобы соответствующие пары находились в одних и тех же позициях. □

Теорема 8.72. Предположим, что G_1, G_2 являются контекстно-свободными грамматиками, а R является регулярным выражением, тогда следующие ниже задачи неразрешимы:

1) $L(G_1) \cap L(G_2) \stackrel{?}{=} \emptyset$;
2) $L(G_1) \stackrel{?}{=} L(G_2)$;
3) $L(G_1) \stackrel{?}{=} L(R)$;
4) $L(G_1) \stackrel{?}{=} T^*$;
5) $L(G_1) \stackrel{?}{\subseteq} L(G_2)$;
6) $L(R) \stackrel{?}{\subseteq} L(G_2)$.

Доказательство. Сначала мы покажем, что $\overline{L_A}$, где определенный выше $L_A = L(G_A)$ также является контекстно-свободной грамматикой; мы показываем это, давая магазинный автомат P. $\Gamma_P = \sum_A \cup \{a_1, a_2, \ldots, a_k\}$. До тех пор, пока P видит символ в $\sum_A$, он сохраняет его в стек. Как только P видит a_i, он выталкивает элемент из стека, чтобы убедиться, что верх строки равен w_i^R. (i) если нет, то принимает независимо от того, что будет дальше. (ii) если да, то существуют два подслучая: (iia) если стек еще не пуст, то продолжает. (iib) если стек пуст и входные данные закончились, то отклоняет. Если после a_i, P видит символ в $\sum_A$, то он принимает.

Теперь мы готовы показать, что шесть перечисленных в теореме задач на самом деле неразрешимы:

1) пусть $G_1 = G_A$ и $G_2 = G_B$, тогда $L(G_1) \cap L(G_2) \neq \emptyset$ тогда и только тогда, когда задача соответствий Поста (A, B) имеет решение;
2) пусть G_1 равно контекстно-свободной грамматике для $\overline{L_A} \cup \overline{L_B}$ (контекстно-свободные грамматики замкнуты при объединении). Пусть G_2 равно контекстно-свободной грамматике для регулярного языка $(\sum \cup \{a_1, a_2, \ldots, a_k\})^*$. Обратите внимание, что $L(G_1) = \overline{L_A} \cup \overline{L_B} = \overline{L_A \cap L_B}$ = все, что угодно, кроме решений задачи соответствий Поста (A, B).
   $\therefore L(G_1) = L(G_2)$ тогда и только тогда, когда (A, B) не имеет решения;
3) показано в п. 2, потому что $L(G_2)$ является регулярным языком;
4) снова показано в п. 2;
5) обратите внимание, что $A = B$ тогда и только тогда, когда $A \subseteq B$ и $B \subseteq A$, поэтому следует из п. 2;
6) по пп. 3 и 5.

Это показывает, что важные свойства контекстно-свободных грамматик неразрешимы. □

8.6. Ответы к избранным задачам

Задача 8.1. $\sum_2^k$ – это множество уникальных строк длины k, которые могут быть построены с помощью $\sum_2$, то есть обобщенного алфавита, содержащего два символа. Хорошим примером $\sum_2$ является стандартный двоичный алфавит {0,1}. В строке длины k на этом алфавите есть k символов, каждый из которых может быть либо 1, либо 0; другими словами, чтобы построить строку длины k, принимается k «решений», каждое с двумя вариантами. Таким образом, существует 2^k возможностей. Можно показать, что каждая из этих возможностей является уникальной. Схожим образом в $\sum_l^k$ существует l^k уникальных слов.

Далее рассмотрим множество строк над $\sum_l$, где ни один символ не может быть повторен ни в одной строке. Пусть n равно длине такой строки. Ясно, что это просто перестановка длины n из множества $\sum_l$ без возврата, где $0 \leqslant n \leqslant l$. В связи с этим число уникальных строк длины n равно $l!/(l-n)!$. Тем самым суммарное количество строк длины $n \in \{0, 1, 2, \ldots, l\}$ равно

$$W(l) = \sum_{n=0}^{l} \frac{l!}{(l-n)!}.$$

Хотя это решение является правильным, мы можем сделать лучше с помощью небольшого анализа нашего результата. Начнем с выделения l!.

$$W(l) = l! \cdot \sum_{n=0}^{l} \frac{l!}{(l-n)!} = l! \cdot \sum_{n=0}^{l} \frac{1}{n!}.$$

Напомним, что разложение e^x в ряд Тейлора равно $\sum_{n=0}^{\infty}(x^n/n!)$, поэтому мы можем переписать:

$$W(l) = l! \cdot \left(e - \sum_{n=(l+1)}^{\infty} \frac{1}{n!} \right) = l! \cdot \left(e - \left(\frac{1}{(l+1)!} + \frac{1}{(l+2)!} + \cdots \right) \right).$$

Далее мы распределяем:

$$W(n) = l! \cdot e - \left(\frac{1}{(l+1)} + \frac{1}{(l+1)(l+2)} + \cdots \right).$$

Рассмотрим фрагмент в скобках; он явно меньше, чем $(1/l + 1/l^2 + \cdots)$ – геометрический ряд, сумма которого равна $1/(l-1)$. Обратите внимание, что если $l > 2$, то эта сумма меньше 1, поэтому $W(l) > l! \cdot e - 1$. Сумма также является положительной, поэтому $W(l) < l! \cdot e$. У нас есть $l! \cdot e - 1 < W(l) < l! \cdot e$; этого в сочетании с тем, что $W(l)$ является целым числом, достаточно, чтобы показать, что $W(l) = l! \cdot e$.

Задача 8.2. Рассмотрим строку S в форме $x01y$. После того как последний элемент x был «выполнен», мы находимся в одном из состояний q_0, q_1, q_2. Ниже мы покажем, что независимо от состояния после x мы будем в состоянии q_1 после последующей подстроки 01.

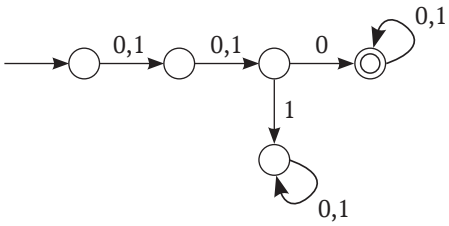
$\delta(q_0, 0) = q_2$ и $\delta(q_2, 1) = q_1$;
$\delta(q_1, 0) = q_1$ и $\delta(q_1, 1) = q_1$;
$\delta(q_2, 0) = q_2$ и $\delta(q_2, 1) = q_1$.

С этого места каждый элемент a из y равен либо 0, либо 1; в любом случае $\delta(q_1, a) = q_1$, поэтому после обработки подстроки 01 мы остаемся в состоянии q_1 до конца S. Поскольку q_1 является конечным состоянием, S принимается A.

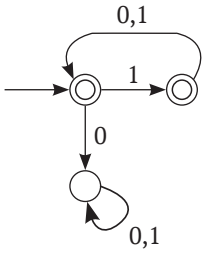
Остается увидеть, что любая строка без подстроки 01 отклоняется A. Рассмотрим такую строку, S'. Если она не содержит нули, то она полностью состоит из единиц. Кроме того, мы начинаем в состоянии q_0, и $\delta(q_0, 1) = q_0$, поэтому состояние равно q_0 для всей строки, и так как q_0 не является конечным состоянием, S' отклоняется. Схожим образом, если S' содержит только один 0 и он является конечным символом, то S' выглядит как 1…10 с некоторым произвольным числом 1 в промежутке. В этом случае мы остаемся в состоянии q_0 до 0 в конце, поэтому конечное состояние равно q_2 – которое также не является конечным состоянием. Если, с другой стороны, S' содержит, по крайней мере, один 0, не в конце строки, то рассмотрим первый 0. После этого первого 0 нет единиц, так как первая такая 1 обязательно будет концом подстроки 01, которой S' не имеет. Таким образом, $S' = xy$, где x – это строка из единиц (или пустая строка) и y – строка из нулей. В конце x мы по-прежнему находимся в состоянии q_0, так как $\delta(q_0,1) = q_0$ по-прежнему является неподвижной точкой. Тогда первый 0 в y приводит к $\delta(q_0, 0) = q_2$, и остальные 0 ничего не делают, потому что $\delta(q_2, 0) = q_2$ также является неподвижной точкой. Таким образом, в конце S' состояние равно q_2, которое не является конечным состоянием; S' отклоняется A.

Гораздо более эффективное доказательство может быть дано посредством индукции над длиной строки – это оставлено читателю.

Задача 8.3.



Задача 8.4.



Задача 8.5. Для B_n, для каждого n, мы хотим построить (другой) детерминированный конечный автомат D_n такой, что $L(D_n) = B_n$. Пусть D_n состоит из состояний $Q = \{q_0, q_1, \ldots, q_{n-1}\}$, и пусть переходная функция δ определена как $\delta(q_i, 1) = q_{(i+1) \pmod n}$, и пусть $F = \{q_0\}$.

Задача 8.6. Пусть $Q = \{q_0, q_1, \ldots, q_{24}\}$, и определим δ следующим образом:

$\delta(q_i, (1)) = q_{(i+1) \pmod{25}}$;
$\delta(q_i, (5)) = q_{(i+5) \pmod{25}}$;
$\delta(q_i, (10)) = q_{(i+10) \pmod{25}}$;
$\delta(q_i, (25)) = q_i$.

Наконец, пусть $F = \{q_0\}$. Этот детерминированный конечный автомат будет принимать только величины, кратные 25. Конечно, торговый автомат должен уметь справляться с недопустимыми входами (например, аркадными жетонами или монетами с неподдерживаемыми значениями). Обозначим любой недопустимый ввод как (I). Очевидно:

$\delta(q_i, (I)) = q_i$.

Более того, на таком вводе δ будет вызывать дополнительное действие, для того чтобы «выплюнуть» недопустимую монету.

Задача 8.7. Просто потому, что ε уже может быть в L, и поэтому если $\varepsilon \in L$, то $\varepsilon \in L^+$. С другой стороны, напомним о допущении, что $\varepsilon \in \sum$ для любого $\sum$.

Задача 8.9. Ответ равен $O(2^n)$. Для того чтобы это увидеть, обратите внимание, что способ построения детерминированного конечного автомата заключается в следующем: дерево начинается с q_0, ответвляется на всех возможных строках из n элементов. Каждый лист – это состояние q_w, где $w \in \{0, 1\}^n$. Принимающие листья – это листья, где w начинается с 1. Предположим, что у нас есть лист q_{ax} (то

есть $w = ax$), тогда $\delta(q_{ax}, b) = q_{xb}$. Обратите внимание, что гораздо проще создать детерминированный конечный автомат для L'_n, где L'_n – это множество строк, где n-е символы в самом начале равны 1.

Задача 8.13. Для конкатенации соедините все принимающие состояния «первого» детерминированного конечного автомата ε-стрелками с начальным состоянием «второго» детерминированного конечного автомата.

Задача 8.14. Базовые случаи состоят в следующем:

$L(a) = \{a\}$;
$L(\varepsilon) = \{\varepsilon\}$;
$L(\emptyset) = \emptyset$.

Далее индукционные правила:

$L(E + F) = L(E) \cup L(F)$;
$L(EF) = \{xy | x \in L(E) \wedge y \in L(F)\}$;
$L(E^*) = \{x_0 x_1 \ldots x_n | \forall i(x_i \in L(E)) \wedge n \in \mathbb{N}\}$.

Задача 8.15. Множество двоичных строк без подстроки 101 может быть выражено:

$0^*(1^*00(0^*))^*1^*0^*$.

Выражение $(1^*00(0^*))^*$ обозначает конкатенацию элементов из множества строк, состоящего из произвольного числа ведущих единиц, за которыми следует как минимум два 0. Идея здесь заключается в том, что за каждой 1 (за исключением последней 1) сразу же следует либо еще одна 1, либо, по меньшей мере, два 0, что делает подстроку 101 невозможной. Остальная часть выражения представляет собой простое заполнение; 0^* в начале обозначает любой ведущий 0, и 1^*0^* обозначает любую концевую подстроку формы 11…100…0.

Задача 8.17. Мы можем интуитивно сконструировать регулярное выражение для всех двоичных строк с помощью подстроки 00 – например, $(\varepsilon + 0 + 1)^*00(\varepsilon + 0 + 1)^*$. Этот метод полезен в более сложных случаях. Обратите внимание, что нам не нужно вычислять R_{ij}^k для всех k, i, j. Единственным конечным состоянием является q_3, поэтому $R = R_{13}^{(3)}$.

$$R_{13}^{(3)} = R_{13}^{(2)} + R_{13}^{(2)}(R_{33}^{(2)})^*R_{33}^{(2)}. \tag{8.3}$$

Поэтому нам нужно найти $R_{13}^{(2)}$ и $R_{33}^{(2)}$.

$$R_{13}^{(2)} = R_{13}^{(1)} + R_{12}^{(1)}(R_{22}^{(1)})^*R_{23}^{(1)}. \tag{8.4}$$

$$R_{33}^{(2)} = R_{33}^{(1)} + R_{32}^{(1)}(R_{22}^{(1)})^*R_{23}^{(1)}. \tag{8.5}$$

Поэтому мы должны вычислить

$R_{12}^{(1)}, R_{13}^{(1)}, R_{22}^{(1)}, R_{23}^{(1)}, R_{32}^{(1)}$ и $R_{33}^{(1)}$.

$$\begin{aligned} R_{12}^{(1)} &= R_{12}^{(0)} + R_{11}^{(0)}(R_{11}^{(0)})^*R_{12}^{(0)} \\ &= 0 + (\varepsilon + 1)(\varepsilon + 1)^*0 \\ &= (\varepsilon + 1)^*0. \end{aligned}$$

Обратите внимание, что $(\varepsilon + 1)(\varepsilon + 1)^* = (\varepsilon + 1)^*$, потому что $\varepsilon \in L(\varepsilon + 1)$.

$$\begin{aligned} R_{13}^{(1)} &= R_{13}^{(0)} + R_{11}^{(0)}(R_{11}^{(0)})^* R_{13}^{(0)} \\ &= \emptyset + (\varepsilon + 1)(\varepsilon + 1)^* \emptyset \\ &= \emptyset. \end{aligned}$$

Этот последний шаг истинен, поскольку для любого регулярного выражения R: $R\emptyset = \emptyset R = \emptyset$.

$$\begin{aligned} R_{22}^{(1)} &= R_{22}^{(0)} + R_{21}^{(0)}(R_{11}^{(0)})^* R_{12}^{(0)} \\ &= \varepsilon + 1(\varepsilon + 1)^* 0. \\ R_{23}^{(1)} &= R_{23}^{(0)} + R_{21}^{(0)}(R_{11}^{(0)})^* R_{13}^{(0)} \\ &= 0 + 1(\varepsilon + 1)^* \emptyset \\ &= 0. \\ R_{32}^{(1)} &= R_{32}^{(0)} + R_{31}^{(0)}(R_{11}^{(0)})^* R_{13}^{(0)} \\ &= \emptyset + \emptyset(\varepsilon + 1)^* 0 \\ &= \emptyset. \end{aligned}$$

При соответствующих наблюдениях можно реализовать ряд сокращений. Например, $\delta(q_3, a) = q_3$ для всех a, поэтому выйти из состояния q_3 невозможно. Если $j \neq 3$, то $R_{3j}^{(n)} = \emptyset$.

$$\begin{aligned} R_{33}^{(1)} &= R_{33}^{(0)} + R_{31}^{(0)}(R_{11}^{(0)})^* R_{13}^{(0)} \\ &= \varepsilon + 0 + 1 + \emptyset(\ldots)^* \ldots \\ &= \varepsilon + 0 + 1. \end{aligned}$$

Теперь мы можем найти $R_{13}^{(2)}$ и $R_{33}^{(2)}$ с помощью уравнений 8.4 и 8.5.

$$\begin{aligned} R_{13}^{(2)} &= R_{13}^{(1)} + R_{21}^{(1)}(R_{22}^{(1)})^* R_{23}^{(1)} \\ &= \emptyset + 1(\varepsilon + 1)^* 0(\varepsilon + 1(\varepsilon + 1)^* 0)^* 0 \\ &= (\varepsilon + 1)^* 0(\varepsilon + 1(\varepsilon + 1)^* 0)^* 0. \\ R_{33}^{(2)} &= R_{33}^{(1)} + R_{32}^{(1)}(R_{22}^{(1)})^* R_{23}^{(1)} \\ &= \varepsilon + 0 + 1 + \emptyset(\ldots)^* \ldots \\ &= \varepsilon + 0 + 1. \end{aligned}$$

Наконец, мы можем использовать уравнение 8.3, чтобы найти $R_{13}^{(3)} = R$.

$$\begin{aligned} R_{13}^{(3)} &= R_{13}^{(2)} + R_{13}^{(2)}(R_{33}^{(2)})^* R_{33}^{(2)} \\ &= (\varepsilon + 1)^* 0(\varepsilon + 1(\varepsilon + 1)^* 0)^* 0 + (\varepsilon + 1)^* 0(\varepsilon + 1(\varepsilon + 1)^* 0)^* 0(\varepsilon + 0 + 1)^*(\varepsilon + 0 + 1) \\ &= (\varepsilon + 1)^* 0(\varepsilon + 1(\varepsilon + 1)^* 0)^* 0(\varepsilon + 0 + 1)^*. \end{aligned}$$

Разумеется, это выражение не упрощено; законы в табл. 8.3.4 могут его улучшить. Результат не должен содержать ε.

Задача 8.18. Объединение: $L = L(R)$ и $M = L(S)$, поэтому $L \cup M = L(R + S)$. Дополнение: $L = L(A)$, поэтому $L^c = L(A')$, где A' – это детерминированный конечный автомат, полученный из A следующим образом: $F_{A'} = Q - FA$. Пересечение: $L \cap M = \overline{\overline{L} \cup \overline{M}}$. Реверс: с учетом регулярного выражения E определяем E^R по структурной индукции. Единственная хитрость заключается в том, что $(E_1E_2)^R = E_2^R E_1^R$. Гомоморфизм: наличие регулярного выражения E определяем $h(e)$ подходящим образом.

Задача 8.19. В случае i обратите внимание, что нам требуется $O(n^3)$ шагов для вычисления e-замыканий всех состояний, и существует 2^n состояний. В случае iii обратите внимание, что существует n выражений $R_{ij}^{(k)}$, и на каждом этапе размер

увеличивается вчетверо (так как нам нужно четырехэтапных (k – 1) выражений для построения одного для этапа k). Случай iv – хитрость здесь заключается в использовании эффективного метода разбора для регулярного выражения; существует $O(n)$ методов.

Задача 8.20. Для случая i используйте автоматное представление: вычислите множество достижимых состояний из q_0. Если хотя бы одно принимающее состояние достижимо, то оно не пустое. Что, если дано только представление в виде регулярного выражения? Для случая ii выполните трансляцию любого представления в детерминированный конечный автомат и выполните строку на этом автомате. Для случая iii используйте эквивалентность и минимизацию автоматов.

Задача 8.21. Здесь мы представим общее доказательство «естественного алгоритма», который вы должны были спроектировать для заполнения таблицы. Мы используем аргументацию от противного с принципом наименьшего числа. Пусть $\{p, q\}$ равно различимой паре, для которой алгоритм оставил соответствующую ячейку пустой, и, более того, из всех таких «плохих» пар $\{p, q\}$ имеет самую короткую различающую строку w. Пусть $w = a_1a_2 \ldots a_n$, $\hat{\delta}(p, w)$ принимает, тогда как $\hat{\delta}(q, w)$ нет. Сначала $w \neq \varepsilon$, так как тогда p, q были бы найдены различимыми в базовом случае алгоритма. Пусть $r = \delta(p, a_1)$ и $s = \delta(q, a_1)$. Тогда $\{r, s\}$ различаются строкой $w' = a_2a_3 \ldots a_n$, и поскольку $|w'| < |w|$, они были обнаружены алгоритмом. Но тогда $\{p, q\}$ была бы найдена на следующем этапе.

Задача 8.22. Рассмотрим детерминированный конечный автомат A, на котором мы выполняем приведенную выше процедуру, для того чтобы получить M. Предположим, что существует N такая, что $L(N) = L(M) = L(A)$), и N имеет меньше состояний, чем M. Выполним алгоритм заполнения таблицы на M, N вместе (переименовывая состояния, чтобы они не имели общих состояний). Поскольку $L(M) = L(N)$, их начальные состояния неразличимы. Таким образом, каждое состояние в M неразличимо хотя бы от одного состояния в N. Но тогда два состояния M неразличимы от одного и того же состояния N…

Задача 8.25. Предположим, это так. По PL[1] $\exists p$ такое, что $|w| \geqslant p \Rightarrow w = xyz$, где $|xy| \leqslant p$ и $y \neq \varepsilon$. Рассмотрим $s = 0^p1^p = xyz$. Поскольку $|xy| \leqslant p$, $y \neq \varepsilon$, очевидно, что $y = 0^j$, $j > 0$. И $xy^2z = 0^{p+j}1^p \in L$, что является противоречием.

Задача 8.26. Предположим, это так. По PL $\exists n$… Рассмотрим некоторое простое $p \geqslant n + 2$. Пусть $1^p = xyz$, $|y| = m > 0$. Поэтому $|xz| = p - m$. Рассмотрим $xy^{(p-m)}z$, которые должны быть в L. Но $|xy^{(p-m)}z| = |xz| + |y|(p - m) = (p - m) + m(p - m) = (p - m)(1 + m)$. Теперь $1 + m > 1$, поскольку $y \neq \varepsilon$, и $p - m > 1$, так как $p \geqslant n + 2$ и $m = |y| \leqslant |xy| \leqslant n$. Поэтому длина $xy^{(p-m)}z$ не является простым числом, и, следовательно, она не может быть в L – противоречие.

Задача 8.27. Для того чтобы показать, что $\equiv_L$ является отношением эквивалентности, нам нужно показать, что оно является рефлексивным, симметричным и транзитивным. Оно явно является рефлексивным; $xz \in L \Leftrightarrow xz \in L$ является истинным, независимо от контекста, поэтому $x \equiv_L x$. Его симметрия и транзитивность следуют непосредственно из симметричной и транзитивной природы «⇔».

Задача 8.29. Мы назначаем веса символам в $\mathcal{V}$; любой предикатный символ (то есть символ функции или отношения) с арностью n имеет вес n – 1. Вес строки $w = w_1w_2 \ldots w_n$ равен сумме ее символов. Мы делаем следующие утверждения:

---

[1] См. https://en.wikipedia.org/wiki/PL_(complexity). – *Прим. перев.*

1) каждый член имеет вес –1;
2) каждый надлежащий начальный сегмент весит не менее 0.

Примечание: надлежащий начальный сегмент – это строка, которая не является членом, но может быть расширена до члена путем конкатенации дополнительных символов (символа) справа.

Базовый случай: для 0-арных символов это явно верно; они весят –1, и единственным надлежащим начальным сегментом является пустая строка ε, которая имеет вес 0.

Это можно расширить по структурной индукции за счет включения всех членов. Рассмотрим член $T = ft_1 \ldots t_n$, где f – это n-арный предикатный символ. f имеет вес $n - 1$, и каждый член t_i весит –1, поэтому чистый вес T равен $(n - 1) + n \cdot (-1)$, или –1. Кроме того, любой надлежащий начальный сегмент из T состоит из f (вес $n - 1$), первые i членов t_i (чистый вес $-i$) с $i < n$, и, возможно, надлежащий начальный сегмент члена t_{i+1}, чей вес является неотрицательным. Таким образом, чистый вес такого сегмента составляет не менее $(n - 1) - i \geqslant 0$.

Пусть $T_1 = f_1 t_{11} \ldots t_{1n}$ и $T_2 = f_2 t_{21} \ldots t_{2m}$, и будем считать, что $T_1 \overset{\text{syn}}{=} T_2$ ($A \overset{\text{syn}}{=} B$ означает, что A и B являются идентичными строками). Очевидно, что $f_1 = f_2 = f$; они являются идентичными, единичными символами, поэтому они должны представлять одну и ту же функцию или отношение. В связи с этим $n = m$. Более того, t_{11} и t_{21} начинаются с одного индекса, и ни один не может быть начальным сегментом другого, поэтому они также должны заканчиваться на том же индексе. Этот аргумент может быть индуктивно расширен на все остальные входные члены для f. T_1 и T_2 представляют результат идентичных входных членов в идентичном порядке на одной и той же функции или отношении.

Задача 8.30. Пусть $\mathbf{t}$ равно $\mathbf{ft}_1 \ldots \mathbf{t}_n$ для некоторого n-арного функционального символа $\mathbf{f}$ и членов $\mathbf{t}_i$. Тогда $\mathbf{t}^{\mathcal{A}} = \mathbf{f}^{\mathcal{A}}(\mathbf{t}_1^{\mathcal{A}}, \ldots, \mathbf{t}_n^{\mathcal{A}})$.

Схожим образом $(\mathbf{Rt}_1 \ldots \mathbf{t}_n)^{\mathcal{A}}$ является идентичным $(\mathbf{t}_1^{\mathcal{A}}, \ldots, \mathbf{t}_n^{\mathcal{A}}) \in \mathbf{R}^{\mathcal{A}}$. Разница заключается в интерпретации: член с ведущим символом отношения является либо истинным, либо ложным, в зависимости от того, является или нет соответствующая упорядоченная последовательность членов элементом интерпретируемого отношения, в то время как член с ведущим символом функции является просто результирующим элементом A.

Задача 8.31. Предположим, что автомат $\mathcal{A}$ в конечном универсуме A принимает $L' = \{\mathbf{a}_n \ldots \mathbf{a}_2 \mathbf{a}_1 \mathbf{c} : a_1 a_2 \ldots a_n \in L\}$. Очевидно, что $(\mathbf{a}_n \ldots \mathbf{a}_1 \mathbf{c})^{\mathcal{A}} = \mathbf{a}_n^{\mathcal{A}}(\mathbf{a}_{n-1}^{\mathcal{A}}(\cdots(\mathbf{a}_1^{\mathcal{A}}(\mathbf{c}^{\mathcal{A}}))\cdots)) \in A$; другими словами, $L(\mathcal{A}) \subseteq A$. Мы определяем недетерминированный конечный автомат для L: начальное состояние $c = q_0$, и остальные состояния $\{q_1, q_2, \ldots, q_m\}$ – это оставшиеся элементы $A - |A|$ является конечным, поэтому данный недетерминированный конечный автомат имеет конечное число состояний. Мы определяем переходную функцию следующим образом: $\delta(q_i, a) = \mathbf{a}^{\mathcal{A}}(q_i)$. Наконец, принимающие состояния F – это просто те состояния, которые принимаются $\mathcal{A}$. L распознается недетерминированным конечным автоматом, поэтому он должен быть регулярным.

С учетом регулярного языка L пусть D равно наименьшему детерминированному конечному автомату для L. Мы знаем, что индекс(L) и Q_D являются конечными и одинаковы из теоремы 8.28; пусть $A = Q_D$. Мы назначаем q_0 метку $\mathbf{c}$ (то есть $\mathbf{c}^{\mathcal{A}} = q_0$) и выбираем вот такую интерпретацию: $\mathbf{a}^{\mathcal{A}}(\mathbf{t}^{\mathcal{A}}) = \delta(\mathbf{t}^{\mathcal{A}}, a)$. Наконец, пусть $\mathbf{R}^{\mathcal{A}} = F_D$. Мы построили автомат, который принимает L'.

Задача 8.32. «Метод» принятия для автоматов напрямую соответствует пересечениям, объединениям и дополнениям. С учетом автоматов $\mathcal{A}$ и $\mathcal{B}$ с отношениями принятия $\mathbf{R}^{\mathcal{A}}$ и $\mathbf{R}^{\mathcal{B}}$ и универсумов A и B автомат, который принимает $L(\mathcal{A}) \cup L(\mathcal{B})$, легко задается универсумом $A \cup B$ и отношением $\mathbf{R}^{\mathcal{A}} \cup \mathbf{R}^{\mathcal{B}}$. Единственный нюанс состоит в том, что некоторые символы в $\mathcal{V}_{\mathcal{A}}$ или $\mathcal{V}_{\mathcal{B}}$, возможно, потребуется заменить новыми символами (с тем же смыслом, что и символы, которые они заменяют), с тем чтобы избежать двойной интерпретации конкретного символа. Пересечения можно обрабатывать примерно так же. Замыкание при дополнении проистекает из конечной природы A; $\mathbf{R}^{\mathcal{A}}$ должно содержать конечное число элементов $\mathcal{P}(A)$, и его можно просто заменить на $\mathcal{P}(A) - \mathbf{R}^{\mathcal{A}}$, создав автомат для дополнения $L(\mathcal{A})$.

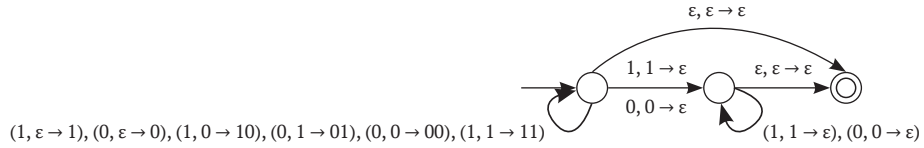
Задача 8.35. P = $\{Q, \sum, \Gamma, \delta, q_0, F\}$, где:

$$\begin{matrix} Q = \{q_0, q_1, q_2\} \\ \Sigma = \Gamma = \\ F = \{q_2\} \end{matrix} \qquad \{1, 0\}$$

и переходная функция δ определена ниже. Отметим, что ε в качестве элемента $\sum^*$ обозначает ε-заполнение, тогда как ε в качестве выходного элемента из стека (то есть $\delta(q_n, x, \varepsilon)$) означает, что стек пуст.

$$\begin{aligned} &\delta(q_0, 0, \varepsilon) = \{(q_0, 0)\} \\ &\delta(q_0, 1, \varepsilon) = \{(q_0, 1)\} \\ &\delta(q_0, \varepsilon, \varepsilon) = \{(q_2, \varepsilon)\} \\ &\delta(q_0, 0, 1) = \{(q_0, 01)\} \\ &\delta(q_0, 1, 0) = \{(q_0, 10)\} \\ &\delta(q_0, 0, 0) = \{(q_0, 00), (q_1, \varepsilon)\} \\ &\delta(q_0, 1, 1) = \{(q_0, 11), (q_1, \varepsilon)\} \\ \delta(q_1, 1, 1) = &\delta(q_1, 0, 0) = \{(q_1, \varepsilon)\} \\ &\delta(q_1, \varepsilon, \varepsilon) = \{q_2, \varepsilon\}. \end{aligned}$$

Обратите внимание, что любые неопределенные переходы отображаются в подразумеваемое «мусорное состояние». На приведенной ниже схеме стрелка из q_i в q_j с меткой $a, b \to c$ означает, что $(q_j, c) \in \delta(q_i, a, b)$.



Задача 8.37. Допустим $(q, x, \alpha) \overset{*}{\Rightarrow} (p, y, \beta)$. Докажем, что $(q, x, \alpha\gamma) \overset{*}{\Rightarrow} (p, y, \beta\gamma)$ по индукции на числе шагов.

Доказательство. Базовый случай: $(q, x, \alpha) \to (p, y, \beta)$. Тогда $x = ay$ для некоторого a такого, что $(p, b) \in \delta(q, a, \alpha_1)$ и $b\alpha_2\alpha_3 \cdots = \beta$. В связи с этим $xw = ayw$ для любой w, и $b\alpha_1\alpha_3 \ldots \gamma = \beta\gamma$. Таким образом, $(q, xw, \alpha\gamma) \to (q, xw, \beta\gamma)$.

Индукционный шаг: если $(q, x, \alpha) \overset{*}{\Rightarrow} (p, y, \beta)$ за n шагов, то существует некий кортеж (o, z, σ) такой, что $(q, x, \alpha) \overset{*}{\Rightarrow} (o, z, \sigma)$ за $n - 1$ шагов и $(o, z, \sigma) \to (p, y, \beta)$. Индукционная гипотеза гарантирует, что $(q, x, \alpha\gamma) \overset{*}{\Rightarrow} (o, z, \sigma\gamma)$, и еще одно применение базового случая гарантирует, что $(o, z, \sigma\gamma) \to (p, y, \beta\gamma)$. Таким образом, $(q, x, \alpha\gamma) \overset{*}{\Rightarrow} (p, y, \beta\gamma)$. □

Задача 8.39. Пусть P равно магазинному автомату, который принимает по конечному состоянию. Мы выполним модификацию P, чтобы он принимал тот же язык по пустому стеку. Пусть q_1 равно начальному состоянию P. Для каждого a такого, что $\delta(q_1, a, \varepsilon) = \{(q_{i_1}, \beta_{i_1}), \dots\}$, заменим этот переход на $\delta(q_1, a, \varepsilon) = \{(q_{i_1}, \beta_{i_1}\$) \dots\}$. Для каждого принимающего состояния q_f в P и каждого $s \in \Gamma_P \cup \{\$\}$ такого, что $\delta(q_f, \varepsilon, s)$, является пустым, пусть $\delta(q_f, \varepsilon, s) = \{(q_f, \varepsilon)\}$. Очевидно, что если у нас закончились входные данные на принимающем состоянии, эта модификация позволяет P «очистить стек», не покидая, что приводит к принятию по пустому стеку. Для каждого отклоняющего состояния q_r и каждого $a \in \sum$, если $\delta(q_r, a, \varepsilon) = \{(q_{i_1}, \beta_{i_1}), \dots\}$, определено, то заменяем это определение на $\delta(q_r, a, \$) = \{(q_{i_1}, \beta_{i_1}\$)\}$; в противном случае оставляем $\delta(q_r, a, \$)$ неопределеным, поэтому стек не может опустошиться на отклоняющем состоянии. Этот измененный магазинный автомат принимает $L(P)$ по пустому стеку.

Далее, пусть P равно магазинному автомату, который принимает по пустому стеку, и пусть q_1 равно начальному состоянию. Для всех a таких, что $\delta(q_1, a, \varepsilon)$, определяется как некоторое множество конфигураций $\{(q_{i_1}, \beta_{i_1}) \dots\}$, убираем этот переход, и вместо него пусть будет $\delta(q_1, a, \varepsilon) = \{(q_{i_1}, \beta_{i_1}\$) \dots\}$. Добавим единственное принимающее состояние q_f, и для каждого состояния q_n пусть $\delta(q_n, \varepsilon, \$) = \{(q_f, \varepsilon)\}$. Любой вход, который был бы принят по пустому стеку в исходном P, «приземляется» на q_f по конструкции, и, более того, другой способ достичь q_f отсутствует (мы просто определили каждый переход к нему), поэтому не принимаются никакие входы, которые были бы отклонены первоначальным определением P. Таким образом, этот модифицированный магазинный автомат принимает такой же язык, что и P, по финальному состоянию.

Задача 8.41. Допустим, что $A_{[qXp]} \overset{*}{\Rightarrow} w$. Тогда, по определению, w переводит магазинный автомат из состояния p в состояние q и выталкивает X из стека. В связи с этим если w – это все оставшиеся входные данные, то X – это весь стек, и магазинный автомат находится в состоянии q, то магазинный автомат остановится на состоянии p с пустым стеком после обработки w; то есть $(q, w, X) \overset{*}{\Rightarrow} (p, \varepsilon, \varepsilon)$.

Далее допустим, что $(q, w, X) \overset{*}{\Rightarrow} (p, \varepsilon, \varepsilon)$. Тогда w переводит магазинный автомат из состояния p в состояние q, и в ходе этого процесса он выталкивает все содержимое X из стека; по определению $A_{[qXp]} \overset{*}{\Rightarrow} w$. Таким образом, $(A_{[qXp]} \overset{*}{\Rightarrow} w) \Leftrightarrow ((q, w, X) \overset{*}{\Rightarrow} (p, \varepsilon, \varepsilon))$.

Задача 8.48. Пусть G равно контекстно-свободной грамматике в нормальной форме Хомского, и допустим, $w \in L(G)$, где $w = a_1 a_2 \dots a_n$. Ясно, что для каждого терминала $a_l \neq \varepsilon$ в w должна быть переменная X_{i_l} такая, что $X_{i_l} \to \alpha a_l \beta$. Поскольку G находится в нормальной форме Хомского и $a_l \neq \varepsilon$, должно быть истинным, что $\alpha = \beta = \varepsilon$, поэтому $X_{i_l} \to a_l$ является правилом. Следовательно, для всех $i \in [1, n]$, (i, i) будет заселена переменной в первом цикле for.

Но из нормальной формы Хомского можно получить больше; каждое правило имеет вид $A \to BC$ и $A \to a$; то есть каждое правило увязывает переменную либо с терминалом, либо с двумя конкатенированными переменными. Таким образом, если $S \overset{*}{\Rightarrow} a_1 a_2 \dots a_n$, то ясно, что $S \overset{*}{\Rightarrow} X_{i_1} X_{i_2} \dots X_{i_n}$, где $X_{i_l} \to a_l$ – это правило для всех l.

Рассмотрим формальное суждение $S \overset{*}{\Rightarrow} X_{i_1} X_{i_2} \dots X_{i_n}$ более подробно; мы знаем, что с точки зрения введения переменных единственные из имеющихся правил имеют форму $S \to AB$, поэтому первый «→» из S в деривации $X_{i_1} X_{i_2} \dots X_{i_n}$ должен быть в этой форме. Очевидно, что существует целое число o такое, что $A \overset{*}{\Rightarrow} X_{i_1} X_{i_2} \dots$

X_{i_o} и $B \overset{*}{\Rightarrow} X_{i_{o+1}} X_{i_{o+2}} \ldots X_{i_n}$. В связи с этим если $(1, o)$ и $(o + 1, n)$ заселяются элементами A и B, то $(1, n)$ будет впоследствии дано S. Мы можем продолжить этот анализ рекурсивно, для того чтобы показать, что A и B будут поставлены на свои правильные места, и поэтому S в конечном итоге будет в $(1, n)$, и w будет принято. Единственным нюансом здесь является порядок, в котором проверяются пары i, j; как показано на рис. 8.12, мы начинаем с главной диагонали и движемся в направлении «вправо вверх», по одной диагонали за раз.

Откуда мы знаем, что никакие новые слова не принимаются? То есть как мы можем быть уверены, что нет $w \notin L(G)$, которые будут приняты алгоритмом CYK?

Задача 8.51. Пусть L равно контекстно-свободному языку с грамматикой G для $L - \{\varepsilon\}$. Допустим, что G находится в нормальной форме Хомского и, более того, что она не имеет допускающих пустое значение переменных (см. п. 8.46). Пусть G имеет n переменных. Рассмотрим $s \in L(G)$ такую, что $|s| \geqslant 2^n$. Тогда s должна иметь путь в дереве разбора длиной не менее $n + 2$ – путь длиной $n + 1$ необходим для достижения строки из 2^n переменных вследствие нормальной формы Хомского, и требуется дополнительный шаг для увязки этих переменных с терминалами. В связи с этим в дереве разбора существует путь, содержащий $n + 1$ переменных; существует только n переменных, поэтому, по крайней мере, одна повторяется. То есть существует переменная R такая, что $R \overset{*}{\Rightarrow} vRy$ для $v, y \in \sum^*$; более того, с необходимостью существует такая переменная, что это происходит не более чем за n шагов, поэтому результат имеет длину не более 2^n. Из-за природы порождения «из переменной в переменную» в нормальной форме Хомского и отсутствия допускающих пустое значение переменных $|vy| > 0$. Поскольку $R \overset{*}{\Rightarrow} vRy$, также является истинным то, что $R \overset{*}{\Rightarrow} vvRyy$; мы можем продолжать «расширять» R таким образом, получая любое (равное) число повторных переменных v и y. Наконец, $R \overset{*}{\Rightarrow} x$ для некоторой строки из терминалов x, тем самым заканчивая доказательство. Обратите внимание, что u и z в лемме являются (возможно, пустыми) строками из начального $S \overset{*}{\Rightarrow} uRz$ для достижения первого R, где S – это начальная переменная. Таким образом, мы имеем $S \overset{*}{\Rightarrow} uv^i xy^i z$, где $|vy| > 0$ и $|vxy| \leqslant 2^n$.

Задача 8.52. Допустим от обратного, что $L = \{0^n 1^n 2^n | n \geqslant 1\}$ является контекстно-свободным языком. Пусть P равно длине накачки для L. Рассмотрим $s = 0^p 1^p 2^p$. По лемме 8.50, s должна быть $uvxyz$... Если v или y содержит более одного уникального терминала, то, конкатенируя его более чем один раз, мы создадим строку, которая не может быть подстрокой любого элемента из L, например если $v = 01$, тогда $v^2 = 0101$, которая не может появиться в любой $w \in L$. Но если v и y каждая являются конкатенациями единственного терминала, то только два из трех терминалов получают увеличение длины в своих соответствующих подстроках. Например, если $v = 11$ и $y = 22$, тогда $uv^2xy^2z = 0^p 1^{p+2} 2^{p+2} \notin L$. Поэтому независимо от состава v и y им не удается удовлетворить условия леммы о накачке.

Задача 8.53. Пусть L равно контекстно-свободному языку, и R равно регулярному языку, оба из которых определены на алфавите $\sum$. Существует магазинный автомат P для L и детерминированный конечный автомат D для R, где оба принимают по финальному состоянию. Обозначим через d_i проиндексированные состояния в D, и через p_j проиндексированные состояния в P. Для каждого состояния d_i в D мы создаем магазинный автомат P_i, который является копией P с двумя ключевыми различиями: финальные состояния в P_i являются финальными, только если d_i является финальным состоянием в D и никаких переходов еще не

было (чтобы мы действительно копировали только состояния). Обозначим через q_{ij} состояние в P_j, которое соответствует d_i в D. Наконец, мы определяем переходы следующим образом: $(q_{kl}, t) \in \delta(q_{ij}, a, s)$ тогда и только тогда, когда $\delta_D(d_i, a) = d_k$ и $(p_l, t) \in \delta_P(p_j, a, s)$. Обратите внимание, что по конструкции q_{ij} является принимающим состоянием тогда и только тогда, когда d_i находится в F_D и p_j находится в F_P. Этот магазинный автомат в связи с этим принимает w тогда и только тогда, когда $w \in L \wedge w \in R$. Таким образом, $L \cap R$ принимается магазинным автоматом, поэтому он должен быть контекстно-свободным языком.

Задача 8.54. Пусть L равно контекстно-свободному языку на $\sum$, представленному контекстно-свободной грамматикой G с терминалами T. Обратите внимание, что $T = \sum$, исходя из того что каждый элемент алфавита $\sum$ достижим, и в противном случае мы можем просто удалить те, которые недостижимы. Для каждого $a \in T$ у нас есть контекстно-свободный язык L_a и соответствующая контекстно-свободная грамматика $G_a = \{V_a, T_a, P_a, S_a\}$. Без потери общности мы можем допустить, что $V_a \cap V_b = \emptyset$ для всех $a, b \in T$ таких, что $a \neq b$, потому что новые переменные могут быть введены по желанию. В G мы можем заменить каждый терминал a во всех продукциях на S_a, и добавить все продукции в P_a. Мы создали контекстно-свободную грамматику для $s(L)$, поэтому он должен быть контекстно-свободным языком.

Задача 8.55. Мы спроектируем машину Тьюринга M такую, что $L(M)$ является языком бинарных палиндромов. У нас есть 8 состояний: $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_{\text{accept}}, q_{\text{reject}}\}$, где q_0 – это начальное состояние. Мы определяем δ следующим образом:

$\delta(q_0, 1) = (q_1, \square, \rightarrow)$	$\delta(q_0, 0) = (q_2, \square, \rightarrow)$	$\delta(q_0, \square) =$ qaccept
$\delta(q_1, 1) = (q_1, 1, \rightarrow)$	$\delta(q_1, 0) = (q_1, 0, \rightarrow)$	$\delta(q_1, \square) = (q_3, \square, \leftarrow)$
$\delta(q_2, 1) = (q_2, 1, \rightarrow)$	$\delta(q_2, 0) = (q_2, 0, \rightarrow)$	$\delta(q_2, \square) = (q_4, \square, \leftarrow)$
$\delta(q_3, 1) = (q_5, \square, \leftarrow)$	$\delta(q_3, 0) = q_{\text{reject}}$	$\delta(q_3, \square) = q_{\text{accept}}$
$\delta(q_4, 1) = q_{\text{reject}}$	$\delta(q_4, 0) = (q_5, \square, \leftarrow)$	$\delta(q_4, \square) = q_{\text{accept}}$
$\delta(q_5, 1) = (q_5, 1, \leftarrow)$	$\delta(q_5, 0) = (q_5, 0, \leftarrow)$	$\delta(q_5, \square) = (q_0, , \rightarrow)$

Допустим, что ввод начинается с 1. Тогда M перейдет из q_0 в q_1, переписывая эту 1 как пустое место, и останется в q_1, двигаясь вправо до тех пор, пока он не попадет в первое пустое место (сразу после конца ввода). В этом месте он двинется влево (к текущему крайне правому непустому месту) и в q_3. Если этот крайне правый вход равен 1 (то есть если он совпадает с 1 слева), то он заменит эту 1 на $\square$, перейдет в q_5 и будет продолжать двигаться влево до тех пор, пока не попадет в пустое место, и в этот момент он перезапустит данный процесс. Если, с другой стороны, он встречает здесь 0, то вход начинается с 1 и заканчивается нулем, поэтому он не является палиндромом и отклоняется. Наконец, если это значение равно $\square$, то все входные данные были перезаписаны символом $\square$, указывая на то, что входные данные были палиндромом, и приводя к их принятию.

8.7. Примечания

Что касается разностной машины Бэббиджа, представленной во введении к настоящей главе, то студентам, изучающим деловую информатику, может быть интересно узнать, что окончательный провал предприятия Бэббиджа был вызван

отсутствием у него деловой хватки; см. стр. 563–570, [Johnson (1991)].

Материал в этой главе опирается на великолепное введение в теорию вычислений [Sipser (2006)]. В частности, доказательство теоремы 8.65 можно найти в решении упражнения 5.28 на стр. 215 книги [Sipser (2006)], а теорема 8.64 является теоремой 3.21 в указанной книге.

Для дальнейшего чтения читатель также адресуется к книге [Kozen (2006)]. В частности, раздел 8.3.9 основан на стр. 109 указанной книги.

Исторический материал в первом абзаце раздела 8.4 взят из лекций Анджея Эренфойхта, читавшихся в Университете Колорадо в Боулдере зимой 2008 года. В 1971 году Эренфойхта был одним из основателей кафедры компьютерных наук в Университете Колорадо. Он сформулировал игру Эренфойхта-Фрайссе, используя метод взад-вперед, описанный Роландом Фрайссе в своей диссертации. Последовательность Эренфойхта-Мысельски также названа его именем. Двое из его учеников, Юджин Майерс и Дэвид Хаусслер, внесли значительный вклад в секвенирование генома человека.

Регулярно-языковые операции ix и x в разделе 8.3.5 взяты из задачи 1.40 в книге [Sipser (2006)]. Материал по теореме Майхилла–Нероуда, раздел 8.3.8.2, мотивирован книгой [Sipser (2006)] [Прим. 1.51 и 1.52].

Контекстно-свободные грамматики являются основой синтаксических анализаторов, или парсеров. Существует много инструментов, которые реализуют идеи, упомянутые в этом разделе; например, Lex, Yacc, Flex, Bison и другие. Подробнее о них можно прочитать здесь: http://dinosaur.compilertools.net.

Раздел 8.4.3 основан на § 7.1 книги [Hopcroft и соавт. (2007)].

В разделе 8.2 мы обсуждаем ur-понятия, такие как символы и слова. Интригующей областью изучения таких объектов является семиотика, изучение знаков и символов, их использование или интерпретация. С давних пор «маркировочные знаки» используются для хранения и обработки информации. Около 8000 лет назад люди использовали символы для обозначения слов и понятий. Истинные формы письма развивались в течение следующих нескольких тысяч лет, и особое значение имеют цилиндрические печати. Они прокатывались по влажным глиняным табличкам, в результате чего получались рельефные конструкции. Многие музеи имеют цилиндрические печати в лазурите[1], относящиеся к Ассирийской культуре, найденные в Вавилоне, Ирак, возраст которых оценивается в 4100–3600 лет. Рельефные конструкции были клинописными символами, которые обозначали понятия, а затем звуки и слоги.

Читателю предлагается посетить, хотя бы онлайн, артефакты, выставленные в Смитсоновском музее естественной истории, Вашингтон, округ Колумбия. Там можно найти выгравированную доску из охры[2] с примитивной символикой из пещеры Бломбос, Южная Африка, которой, по оценкам, 77 000–75 000 лет. Кроме того, кости Ишанго из Конго, по оценкам, имеют возраст 25 000–20 000 лет, которые являются костью ноги бабуина, с тремя рядами счетных меток для сложения и умножения (археологи не уверены, какие из них какие). И наконец, оленьи рога

---

[1] Лазурит – редкий полудрагоценный камень, который издревле ценился за свой насыщенный синий цвет.

[2] Охра – землистый пигмент, содержащий железную окись, как правило, с глиной, варьирующийся по цвету от светло-желтого дл коричневого или красного цвета.

с отметками, из Мадленской культуры, Франция, по оценкам имеющие возврат 17 000–11 500 лет.

В наборе текста разные стилевые формы английского алфавита называются шрифтами. Шрифты PostScript – это спецификации контурных шрифтов, разработанные компанией Adobe Systems для профессионального цифрового набора, который используется в формате файла PostScript для кодирования информации о шрифтах. Контурные шрифты (или векторные шрифты) представляют собой множества векторных изображений, то есть множество линий и кривых для определения границы глифа.

С алгебраической точки зрения мы можем сказать, что $\sum^*$ вместе с оператором конкатенации $\cdot$ является моноидом, где $\cdot$ является ассоциативной операцией, а ε – элементом тождественности. Это одна из многих точек сопряжения между символьными цепочками, или строками, и красивой областью алгебры, именуемой теорией групп (см. раздел 9.2.3).

Глава 9

Математическая основа

> И из математических рассуждений возникает истинный философский вопрос, вопрос, который не сможет решить никакое количество биологии: а именно что такое математика? Что же такое числа, множества и трансфинитные кардиналы?
>
> *Сэр Роджер Скратон*
> *[Scruton (2014)], стр. 6*

9.1. Индукция и инвариантность

9.1.1. Индукция

Пусть $\mathbb{N} = \{0, 1, 2, \ldots\}$ равно множеству натуральных чисел. Предположим, что S – это подмножество $\mathbb{N}$ со следующими двумя свойствами: первое, $0 \in S$, и второе, всякий раз, когда $n \in S$, то также и $n + 1 \in S$. Тогда, используя *принцип индукции*, можно сделать вывод, что $S = \mathbb{N}$.

Мы будем использовать принцип индукции с более удобной формой записи; пусть P равно свойству натуральных чисел, другими словами, P – это унарное отношение, такое, что $P(i)$ либо истинно, либо ложно. Отношение P можно очевидным образом идентифицировать множеством S_P, то есть $i \in S_P$ тогда и только тогда, когда $P(i)$ истинно. Например, если P – это свойство примарности, то P(2) и P(3) истинны, но P(6) является ложным, и $S_P = \{2, 3, 5, 7, 11, \ldots\}$. Используя эту форму записи, принцип индукции может быть сформулирован как

$$[P(0) \wedge \forall n(P(n) \rightarrow P(n + 1))] \rightarrow \forall m P(m) \tag{9.1}$$

для любого (унарного) отношения P над $\mathbb{N}$. На практике мы используем (9.1) следующим образом: сначала доказываем, что P(0) соблюдается (это базовый случай). Затем мы показываем, что $\forall n(P(n) \rightarrow P(n + 1))$ (это индукционный шаг). Наконец, используя (9.1) и модус попенс, мы заключаем, что $\forall m P(m)$.

В качестве примера пусть P равно логическому утверждению «сумма первых i нечетных чисел равна i^2». Мы следуем соглашению, что сумма пустого множества чисел равна нулю; таким образом, P(0) соблюдается, поскольку множество первых нулевых нечетных чисел является пустым множеством. P(1) является истинным, так как $1 = 1^2$, и P(3) также является истинным, поскольку $1 + 3 + 5 = 9 = 3^2$. Мы хотим показать, что на самом деле $\forall m P(m)$, то есть P всегда является истинным, и поэтому $S_P = \mathbb{N}$.

Обратите внимание, что $S_P = \mathbb{N}$ не означает, что все числа являются нечетными – совершенно очевидное ложное логическое утверждение. Мы используем натуральные числа для индексирования нечетных чисел, то есть $o_1 = 1$, $o_2 = 3$, $o_3 = 5$, $o_4 = 7, \ldots$, и наша индукция осуществляется над этим индексированием (где o_i – это i-е нечетное число, то есть $o_i = 2i - 1$). То есть мы доказываем, что для всех $i \in \mathbb{N}$, $o^1 + o^2 + o^3 + \cdots + o^i = i^2$; наше логическое утверждение $P(i)$ является именно формальным суждением «$o^1 + o^2 + o^3 + \cdots + o^i = i^2$».

Теперь мы используем индукцию: базовый случай равен $P(0)$, и мы уже показали, что он соблюдается. Предположим теперь, что это логическое утверждение соблюдается для n, то есть сумма первых n нечетных чисел равна n^2, то есть $1 + 3 + 5 + \cdots + (2n - 1) = n^2$ (это наша индукционная гипотеза, или индукционное допущение). Рассмотрим сумму первых $(n + 1)$ нечетных чисел,

$$\boxed{1 + 3 + 5 + \cdots + (2n - 1)} + (2n + 1) = \boxed{n^2} + (2n + 1) = (n + 1)^2,$$

и тем самым мы только что доказали индукционный шаг, и по принципу индукции мы имеем $\forall m P(m)$.

Задача 9.1. Докажите, что $1 + \sum_{j=0}^{i} 2^j = 2^{i+1}$.

Иногда удобно начать нашу индукцию с более высокого значения, чем в 0. Мы имеем следующий обобщенный принцип индукции:

$$[P(k) \wedge (\forall n \geqslant k)(P(n) \to P(n + 1))] \to (\forall m \geqslant k)P(m) \tag{9.2}$$

для любого предиката P и любого числа k. Обратите внимание, что (9.2) легко следует из (9.1), если мы просто примем $P(i)$ равным $P(i + k)$ и выполним привычную индукцию на предикате $P'(i)$.

Задача 9.2. Примените индукцию, для того чтобы доказать, что для $n \geqslant 1$

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

Задача 9.3. Для каждого $n \geqslant 1$ рассмотрите площадь размера $2^n \times 2^n$ квадратов, в которой один квадрат отсутствует. Покажите, что результирующую площадь можно заполнить фигурами в форме латинской буквы «L» – то есть кластерами из трех квадратов, где три квадрата не образуют линию.

Задача 9.4. Предположим, что мы переформулируем обобщенный принцип индукции (9.2) как

$$[P(k) \wedge \forall n(P(n) \to P(n + 1))] \to (\forall m \geqslant k)P(m). \tag{9.2′}$$

Какова связь между (9.2) и (9.2)?

Задача 9.5. Последовательность Фибоначчи определяется следующим образом: $f_0 = 0$ и $f_1 = 1$ и $f_i + 2 = f_i + 1 + f_i$, $i \geqslant 0$. Докажите, что для всех $n \geqslant 1$ мы имеем:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{pmatrix},$$

где левая сторона – это n-я степень матрицы размера 2×2.

Задача 9.6. Напишите программу, вычисляющую n-е число Фибоначчи с помощью матричного умножения задачи 9.5.

Задача 9.7. Докажите следующее: если m делит n, то f_m делит f_n, то есть $m|n \Rightarrow f_m|f_n$.

Принцип полной индукции подобен принципу индукции, за исключением того, что на индукционном шаге мы показываем, что если P(i) соблюдается для всех $i \leqslant n$, то P(n + 1) тоже соблюдается, то есть индукционный шаг сейчас равен $\forall n((\forall i \leqslant n)$ P(i) → P(n + 1)).

Задача 9.8. Используйте принцип полной индукции, для того чтобы доказать, что каждое число (в n) больше 1 может быть записано как произведение одного или нескольких простых чисел.

Задача 9.9. Предположим, что у нас есть плитка (швейцарского) шоколада, состоящая из нескольких долек, расположенных в прямоугольном узоре. Наша задача – разломить плитку на дольки (она всегда ломается вдоль линий между дольками) с минимальным количеством разламываний. Сколько разламываний потребуется? Выдвиньте обоснованную догадку и докажите ее по индукции.

Принцип наименьшего числа говорит, что каждое непустое подмножество натуральных чисел должно иметь наименьший элемент. Прямым следствием данного принципа является то, что каждая убывающая неотрицательная последовательность целых чисел должна завершиться; то есть если $R = \{r_1, r_2, r_3, \ldots\} \subseteq \mathbb{N}$, где $r_i > r_{i+i}$ для всех i, то R – это *конечное* подмножество $\mathbb{N}$. Мы намерены применить принцип наименьшего числа, для того чтобы показать завершение алгоритмов.

Задача 9.10. Покажите, что принцип индукции, принцип полной индукции и принцип наименьшего числа являются эквивалентными принципами.

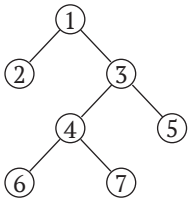
Существует три стандартных способа перечисления узлов бинарного дерева. Мы представляем их ниже вместе с рекурсивной процедурой, которая перечисляет узлы в соответствии с каждой схемой представления.

Инфиксная: левое поддерево, корень, правое поддерево.

Префиксная: корень, левое поддерево, правое поддерево.

Постфиксная: левое поддерево, правое поддерево, корень.

См. пример на рис. 9.1.



Инфиксная: 2, 1, 6, 4, 7, 3, 5
Префиксная: 1, 2, 3, 4, 6, 7, 5
Постфиксная: 2, 6, 7, 4, 5, 3, 1

Рис. 9.1 ❖ Бинарное дерево с соответствующими представлениями

Обратите внимание, что для инфиксной, префиксной и постфиксной схем представления некоторые авторы используют другое название; они называют их соответственно порядковый (inorder), предпорядковый (preorder) и постпорядковый (postorder) обход.

Задача 9.11. Покажите, что при любых двух представлениях мы можем получить из них третье, или, другими словами, из любых двух представлений мы можем

восстановить дерево. Покажите, используя индукцию, что ваша реконструкция правильная. Затем покажите, что одного представления недостаточно.

Задача 9.12. Напишите программу, которая принимает на входе два из трех описаний, а выводит третье. Один из способов представить входные данные – разместить их в текстовом файле, состоящем из двух строк, например

```
инфикс: 2,1,6,4,7,3,5
постфикс: 2,6,7,4,5,3,1
```

и соответствующий выход будет: `префикс: 1,2,3,4,6,7,5`. Обратите внимание, что каждая строка входных данных должна указывать схему описания.

9.1.2. Инвариантность

Метод инвариантности – это метод доказательства логических утверждений об исходах процедур. Даный метод определяет некоторое свойство, которое остается истинным в течение исполнения процедуры. Затем, как только процедура завершается, мы используем это свойство, чтобы доказывать логические утверждения об исходе.

В качестве примера рассмотрим доску размера 8×8, из которой были удалены две клетки из противоположных углов (см. рис. 9.2). Площадь доски равна 64 – 2 = 62 клетки. Теперь предположим, что у нас 31 кость домино размером 1×2. Мы хотим показать, что доска не может быть ими покрыта.

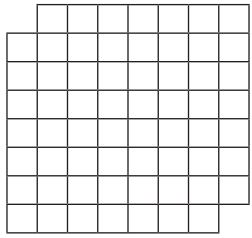


Рис. 9.2 ❖ Доска размера 8×8

Верифицирование с помощью *грубой силы* (то есть исследование всех возможных покрытий доски) будет чрезвычайно трудоемким. Однако, применяя метод инвариантности, мы рассуждаем следующим образом: раскрасим клетки как шахматную доску. Каждая кость домино, покрывающая две смежные клетки, покрывает 1 белую и 1 черную клетку, и, следовательно, каждое размещение покрывает столько белых клеток, сколько оно покрывает черных клеток. Обратите внимание, что число белых клеток и число черных клеток различаются на 2 – противоположные углы, лежащие на одной диагонали, имеют один и тот же цвет – и, следовательно, никакое размещение костей домино не дает покрытия; готово!

Более формально, мы кладем кости домино по одной на доску в любом удобном для нас виде. Инвариант состоит в том, что после размещения каждой новой кости домино число покрытых белых квадратов совпадает с числом покрытых черных квадратов. Мы приводим доказательство, что это является инвариантом, по индукции на числе размещенных костей домино. Базовый случай – это ког-

да было размещено ноль костей домино (и поэтому было покрыто ноль черных и ноль белых клеток). На индукционном шаге мы добавляем еще одну кость домино, которая, независимо от того, как мы ее размещаем, покрывает одну белую и одну черную клетки, тем самым сохраняя свойство. В конце концов, когда мы закончим размещение костей домино, у нас должно быть столько белых клеток, сколько черных клеток, что невозможно из-за характера окраски доски (то есть число черных и белых клеток не то же самое). Обратите внимание, что этот аргумент легко распространяется на доску размера $n \times n$.

Задача 9.13. Пусть n равно нечетному числу, и предположим, что мы имеем множество $\{1, 2, \ldots, 2n\}$. Мы выбираем во множестве любые два числа a, b, удаляем их из множества и заменяем их на $|a - b|$. Продолжайте повторять это до тех пор, пока во множестве не останется только одно число; покажите, что это оставшееся число должно быть нечетным.

Следующие три задачи имеют общую тему, связанную с общественными собраниями. Мы всегда исходим из того, что отношения симпатий и антипатий, дружеские или недружеские, симметричны: то есть если a нравится b, то и b тоже нравится a и т. д. См. раздел 9.3 для получения общей информации по отношениям – симметричные отношения определены на стр. 217.

Задача 9.14. В загородном клубе каждому члену не нравится не более трех других членов. Существует два теннисных корта; покажите, что каждый член может быть приписан одному из двух кортов так, что не более одного человека, который ему не нравится, также играет на том же корте.

Мы используем лексику «загородный клуб» и «теннисный корт», но ясно, что задача 9.14 представляет собой типичную ситуацию, с которой можно столкнуться в информатике: например, многопоточная программа, которая выполняется на двух процессорах, где пара потоков, когда они используют много одинаковых ресурсов, воспринимается как «соперники». Потоки, требующие одинаковых ресурсов, должны планироваться на разных процессорах, насколько это возможно. В некотором смысле эти, казалось бы, невинные задачи являются притчами информатики.

Задача 9.15. Вы принимаете званый обед, где $2n$ человек будут сидеть за круглым столом. Как это нередко происходит в любой социальной клике, довольно часто имеет место неприязнь, но вы знаете, что любому, сидящему за столом, не нравятся $(n - 1)$ людей; покажите, что вы можете рассадить людей так, чтобы никто не сидел рядом с кем-то, кого он недолюбливает.

Задача 9.16. На встрече происходит обмен рукопожатиями. Мы называем человека нечетным лицом, если он обменялся нечетным числом рукопожатий. Покажите, что в любой момент существует четное число нечетных лиц.

9.2. Теория чисел

В этом разделе мы будем работать со множеством целых и натуральных чисел:

$$\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}, \mathbb{N} = \{0, 1, 2, \ldots\}.$$

9.2.1. Простые числа

Мы говорим, что *x делит y*, и записываем $x|y$, если $y = qx$. Если $x|y$, то будем говорить, что x является *делителем* (также фактором) y. Используя терминологию, представленную в разделе 1.1.2, $x|y$ тогда и только тогда, когда $y = \text{div}(x, y) \cdot x$. Мы говорим, что число p является *простым*, если его делителями являются само это число и 1.

Утверждение 9.17. Если p является простым числом и $p|a_1a_2 \ldots a_n$, то $p|a_i$ для некоторого i.

Доказательство. Достаточно показать, что если $p|ab$, то $p|a$ или $p|b$. Пусть $g = \gcd(a, p)$. Тогда $g|p$, и поскольку p – это простое число, то существует два случая. Случай 1, $g = p$, то поскольку $g|a$, то $p|a$. Случай 2, $g = 1$, поэтому существуют u и v такие, что $au + pv = 1$ (см. алгоритм 1.8), поэтому $abu + pbv = b$. Поскольку $p|ab$, и $p|p$, из этого следует, что $p|(abu + pbv)$, поэтому $p|b$. □

Теорема 9.18 (основная теорема арифметики). Пусть дано $a \geqslant 2$. С учетом этого a можно записать как $a = p_1^{e_1}p_2^{e_2} \cdots p_r^{e_r}$, где p_i – это простые числа, и, помимо перестановки простых чисел, данная факторизация уникальна.

Доказательство. Сначала мы покажем существование факторизации, а затем ее уникальность. Доказательство существования выполняется по полной индукции; базовый случай $a = 2$, где 2 – это простое число. Рассмотрим целое число $a > 2$. Если a – простое, то оно является своей собственной факторизацией (как и в базовом случае). В противном случае если a – составное, то $a = b \cdot c$, где $1 < b, c < a$; применяем индукционную гипотезу к b и c.

Для того чтобы показать уникальность, предположим, что $a = p_1p_2 \ldots p_s = q_1q_2 \ldots q_t$, где мы выписали все простые числа, то есть вместо записи p^e мы e раз пишем $p \cdot p \cdots p$. Поскольку $p_1|a$, из этого следует, что $p_1|q_1q_2 \cdots q_t$. Поэтому по утверждению 9.17 $p_1|q_j$ для некоторого j, но тогда $p_1 = q_j$, так как они оба простые. Теперь удалим p_1 из первого списка и q_j из второго списка и продолжим. Совершенно очевидно, что мы не сможем завершить произведением простых чисел, равным 1, поэтому два списка должны быть идентичными. □

9.2.2. Модулярная арифметика

Пусть $m \geqslant 1$ равно целому числу. Мы говорим, что a и b *конгруентны по модулю* m, и записываем $a \equiv b \pmod m$ (или иногда $a \equiv_m b$), если $m|(a - b)$. Еще один способ сказать это состоит в том, что a и b имеют одинаковый остаток при делении на m; используя терминологию раздела 1.1, мы можем сказать, что $a \equiv b \pmod m$ тогда и только тогда, когда $\text{rem}(a, m) = \text{rem}(b, m)$.

Задача 9.19. Покажите, что если $a_1 \equiv_m a_2$ и $b_1 \equiv_m b_2$, то $a_1 \pm b_1 \equiv_m a_2 \pm b_2$ и $a_1 \cdot b_1 \equiv_m a_2 \cdot b_2$.

Пропозиция 9.20. Если $m \geqslant 1$, то $a \cdot b \equiv_m 1$ для некоторого b тогда и только тогда, когда $\gcd(a, m) = 1$.

Доказательство. ($\Rightarrow$) Если существует b такое, что $a \cdot b \equiv_m 1$, то мы имеем $m|(ab - 1)$ и, таким образом, существует c такое, что $ab - 1 = cm$, то есть $ab - cm = 1$. И поскольку $\gcd(a, m)$ делит и a, и m, то он также делит $ab - cm$, и, значит, $\gcd(a, m)|1$, и поэтому он должен быть равен 1.

($\Leftarrow$) Предположим, что $\gcd(a, m) = 1$. По расширенному алгоритму Евклида (см. алгоритм 1.8) существуют u, v такие, что $au + mv = 1$, поэтому $au - 1 = -mv$, поэтому $m|(au - 1)$, и, значит, $au \equiv_m 1$. Поэтому пусть $b = u$. □

Пусть $\mathbb{Z}_m = \{0, 1, 2, \ldots, m - 1\}$. Назовем $\mathbb{Z}_m$ множеством целых чисел по модулю m. Для того чтобы сложить или умножить во множестве $\mathbb{Z}_m$, мы складываем и умножаем соответствующие целые числа, а затем берем остаток от деления на m в качестве результата. Пусть $\mathbb{Z}_m = \{a \in \mathbb{Z}_m | \gcd(a, m) = 1\}$. По пропозиции 9.20 мы знаем, что $\mathbb{Z}_m^*$ – это подмножество $\mathbb{Z}_m$, состоящее из тех элементов, которые имеют мультипликативные инверсии в $\mathbb{Z}_m$.

Функция $\varphi(n)$ называется *эйлеровой функцией* и является числом элементов меньше n, взаимно простых с n, то есть $\varphi(n) = |\mathbb{Z}_m^*|$.

Задача 9.21. Если мы можем факторизовать, мы также можем вычислить $\varphi(n)$. Покажите, что если $n = p_1^{k_1} p_2^{k_2} \cdots p_l^{k_l}$, то $\varphi(n) = \prod_{i=1}^{l} p_i^{k_i - 1}(p_i - 1)$.

Теорема 9.22 (малая теорема Ферма). Пусть p равно простому числу, и $\gcd(a, p) = 1$. Тогда $a^{p-1} \equiv 1 \pmod p$.

Доказательство. Для любого a такого, что $\gcd(a, p) = 1$, все следующие произведения

$$1a, 2a, 3a, \ldots, (p - 1)a, \tag{9.3}$$

взятые по модулю p, попарно не совпадают. Для того чтобы это увидеть, предположим, что $ja \equiv ka \pmod p$. Тогда $(j - k)a \equiv 0 \pmod p$, и поэтому $p|(j - k)a$. Но так как по принятому допущению $\gcd(a, p) = 1$, отсюда следует, что $p \nmid a$, и, значит, по утверждению 9.17 действительно должно иметь место, что $p|(j - k)$. Но так как $j, k \in \{1, 2, \ldots, p - 1\}$, из этого следует, что $-(p - 2) \leqslant j - k \leqslant (p - 2)$, поэтому $j - k = 0$, то есть $j = k$.

Таким образом, числа в списке (9.3) являются просто переупорядочиванием списка $\{1, 2, \ldots, p - 1\}$. Следовательно:

$$a^{p-1}(p - 1)! \equiv_p \prod_{j=1}^{p-1} j \cdot a \equiv_p \prod_{j=1}^{p-1} j \equiv_p (p - 1)!. \tag{9.4}$$

Поскольку все числа в $\{1, 2, \ldots, p - 1\}$ имеют инверсии в $\mathbb{Z}_p$, так как $\gcd(i, p) = 1$ для $1 \leqslant i \leqslant p - 1$, их произведение также имеет инверсию. То есть $(p - 1)!$ имеет инверсию, и поэтому, умножая обе стороны (9.4) на $((p - 1)!)^{-1}$, мы получаем результат. □

Задача 9.23. Приведите второе доказательство малой теоремы Ферма, используя биномиальное разложение, то есть $(x + y)^n = \sum_{j=0}^{n} \binom{n}{j} x^j y^{n-j}$ применительно к $(a + 1)^p$.

9.2.3. Теория групп

Мы говорим, что $(G, *)$ является *группой*, если G является множеством и $*$ операцией, такой, что если $a, b \in G$, то $a * b \in G$ (это свойство называется замыканием). Кроме того, операция $*$ должна удовлетворять следующим трем свойствам:

1) *закон существования нейтрального элемента*: существует $e \in G$ такой, что $e * a = a * e = a$ для всех $a \in G$;

2) *закон существования обратного элемента*: для каждого $a \in G$ существует элемент $b \in G$ такой, что $a * b = b * a = e$. Этот элемент b называется *обратным*, и можно показать, что он уникален; следовательно, он часто обозначается как a^{-1};
3) *ассоциативный закон*: для всех $a, b, c \in G$ мы имеем $a * (b * c) = (a * b) * c$.

Если $(G, *)$ также удовлетворяет *коммутативному закону*, то есть если для всех $a, b \in G$, $a * b = b * a$, то это называется *коммутативной*, или *абелевой*, *группой*.

Типичными примерами групп являются $(\mathbb{Z}_n, +)$ (целые числа по модулю n при сложении) и $(\mathbb{Z}_n^*)$ (целые числа по модулю n при умножении). Обратите внимание, что обе эти группы абелевы. Разумеется, нас интересуют эти две группы; но есть и ряд других: $(\mathbb{Q}, +)$ – бесконечная группа (рациональных чисел при сложении), $\mathrm{GL}(n, \mathbb{F})$ (это группа из обратимых $n \times n$-матриц на поле $\mathbb{F}$) и S_n (*симметрическая группа* над n элементами, состоящая из перестановок $[n]$, где $*$ – это композиция функций).

Задача 9.24. Покажите, что $(\mathbb{Z}_n, +)$ и $(\mathbb{Z}_n^*, \cdot)$ являются группами, проверив, что соответствующая операция удовлетворяет трем аксиомам группы.

Пусть $|G|$ обозначает число элементов в G (заметим, что G может быть бесконечной, но мы имеем дело главным образом с конечными группами). Если $g \in G$ и $x \in \mathbb{N}$, то $g^x = g * g * \cdots * g$, x раз. Если из контекста ясно, что операцией является $*$, то мы используем смежное расположение ab вместо $a * b$.

Предположим, что G является конечной группой и $a \in G$; тогда наименьшим $d \in \mathbb{N}$ таким, что $a^d = e$, называется *порядок* a, и он обозначается как $\mathrm{ord}_G(a)$ (или просто $\mathrm{ord}(a)$, если группа G ясна из контекста).

Пропозиция 9.25. Если G является конечной группой, то для всех $a \in G$ существует $d \in \mathbb{N}$ такое, что $a^d = e$. Если $d = \mathrm{ord}_G(a)$ и $a^k = e$, то $d | k$.

Доказательство. Рассмотрим список $a^1, a^2, a^3, \ldots$. Если G является конечной, то должно существовать $i < j$ такое, что $a^i = a^j$. Тогда, $(a^{-1})^*$, примененное к обеим сторонам, дает $a^{j-i} = e$. Пусть $d = \mathrm{ord}(a)$ (по принципу наименьшего числа мы знаем, что оно должно существовать!). Предположим, что $k \geqslant d$, $a^k = e$; пусть q, r равно соответственно делителю и остатку. Тогда $e = a^k = a^{dq+r} = (a^d)^q a^r$. Поскольку $a^d = e$, из этого следует, что $a^r = e$, противореча минимальности $d = \mathrm{ord}(a)$, если только не $r = 0$. □

Если $(G, *)$ является группой, то мы говорим, что H является *подгруппой* группы G, и записываем $H \leqslant G$, если $H \subseteq G$ и H замыкается при $*$. То есть H является подмножеством группы G, и H сама является группой. Обратите внимание, что для любой G всегда имеет место $\{e\} \leqslant G$ и $G \leqslant G$; эти два элемента называются *тривиальными подгруппами* группы G. Если $H \leqslant G$ и $g \in G$, то gH называется *левым сомножеством* группы G, и оно представляет собой просто множество $\{gh | h \in H\}$. Обратите внимание, что gH не обязательно является подгруппой группы G.

Теорема 9.26 (Лагранжа). Если G является конечной группой и $H \leqslant G$, то $|H|$ делит $|G|$, то есть порядок H делит порядок G.

Доказательство. Если $g_1, g_2 \in G$, то два сомножества g_1H и g_2H либо идентичны, либо $g_1H \cap g_2H = \emptyset$. Для того чтобы это увидеть, предположим, что $g \in g_1H \cap g_2H$,

поэтому $g = g_1h_1 = g_2h_2$. В частности, $g_1 = g_2h_2h_1^{-1}$. Таким образом, $g_1H = (g_2h_2h_1^{-1})H$, и поскольку легко можно проверить, что $(ab)H = a(bH)$ и что $hH = H$ для любого $h \in H$, из этого следует, что $g_1H = g_2H$.

Следовательно, для конечной $G = \{g_1, g_2, \ldots, g_n\}$ коллекция множеств $\{g_1H, g_2H, \ldots, g_nH\}$ является разбиением G на подмножества, которые либо пересекаются, либо идентичные; из всех подколлекций идентичных сомножеств мы выбираем представителя, в силу чего $G = g_{i_1}H \cup g_{i_2}H \cup \ldots \cup g_{i_m}H$, и поэтому $|G| = m|H|$, и доказательство завершено. □

Задача 9.27. Пусть $H \leqslant G$. Покажите, что если $h \in H$, тогда $hH = H$, и что в общем случае для любого $g \in G$ $|gH| = |H|$. Наконец, покажите, что $(ab)H = a(bH)$.

Задача 9.28. Если G – это группа и $\{g_1, g_2, \ldots, g_k\} \subseteq G$, то множество $g_1, g_2, \ldots, g_k$ определяется следующим образом:

$$\{x_1x_2 \cdots x_p | p \in \mathbb{N}, x_i \in \{g_1, g_2, \ldots, g_k, g_1^{-1}, g_2^{-1}, \ldots, g_k^{-1}\}\}.$$

Покажите, что множество $g_1, g_2, \ldots, g_k$ (именуемое подгруппой, генерируемой $\{g_1, g_2, \ldots, g_k\}$) – это подгруппа группы G. Также покажите, что когда G конечна, $|g| = \mathrm{ord}_G(g)$.

9.2.4. Приложения теории групп к теории чисел

Теорема 9.29 (Эйлера). Для каждого n и каждого $a \in \mathbb{Z}_n^*$, то есть для каждой пары a, n такой, что $\gcd(a, n) = 1$, мы имеем $a^{\varphi(n)} \equiv 1 \pmod n$.

Доказательство. Сначала легко проверить, что $(\mathbb{Z}_n^*, \cdot)$ является группой. Тогда по определению $\varphi(n) = |\mathbb{Z}_n^*|$, и поскольку $a \leqslant \mathbb{Z}_n^*$ по теореме Лагранжа следует, что $\mathrm{ord}(a) = |a|$ делит $\varphi(n)$. □

Обратите внимание, что малая теорема Ферма (уже представленная как теорема 9.22) является непосредственным следствием теоремы Эйлера, так как, когда p является простым, $\mathbb{Z}_n^* = \mathbb{Z}_p - \{0\}$ и $\varphi(p) = (p - 1)$.

Теорема 9.30 (китайская теорема об остатках). С учетом двух множеств чисел одинакового размера, $r_0, r_1, \ldots, r_n$ и $m_0, m_1, \ldots, m_n$ таких, что

$$0 \leqslant r_i < m_i \qquad 0 \leqslant i \leqslant n \tag{9.5}$$

и $\gcd(m_i, m_j) = 1$ для $i \neq j$, значит, существует r такой, что $r \equiv r_i \pmod m_i$ для $0 \leqslant i \leqslant n$.

Доказательство. Мы даем доказательство путем подсчета; мы показываем, что разные значения r, $0 \leqslant r < \prod m_i$, представляют собой разные последовательности. Для того чтобы это увидеть, обратите внимание, что если $r \equiv r' \pmod m_i$ для всех i, то $m_i|(r - r')$ для всех i и, значит, $(\prod m_i)|(r - r')$, так как m_i являются попарно взаимно простыми. Поэтому $r \equiv r' \pmod{(\prod m_i)}$, и, значит, $r = r'$, так как оба r, $r' \in \{0, 1, \ldots, (\prod m_i) - 1\}$.

Но суммарное число последовательностей $r_0, \ldots, r_n$ такое, что соблюдается (9.5), составляет ровно $\prod m_i$. Следовательно, каждая такая последовательность должна быть последовательностью остатков некоторого r, $0 \leqslant r < \prod m_i$. □

Задача 9.31. Доказательство теоремы 9.30 (китайской теоремы об остатках) является неконструктивным. Покажите, как эффективно получить r, которое удов-

летворяет требованиям теоремы, то есть за полиномиальное время в n – таким образом, чтобы, в частности, не использовать поиск методом грубой силы.

При наличии двух групп $(G_1, *_1)$ и $(G_2, *_2)$ отображение $h : G_1 \to G_2$ является *гомоморфизмом*, если он соблюдает операцию групп; формально для всех $g_1, g_1' \in G_1$, $h(g_1 *_1 g_1') = h(g_1) *_2 h(g_1')$. Если гомоморфизм h также является биекцией, тогда он называется *изоморфизмом*. Если между двумя группами G_1 и G_2 существует изоморфизм, то мы называем их *изоморфными* и записываем $G_1 \cong G_2$.

Если $(G_1, *_1)$ и $(G_2, *_2)$ – это две группы, то их произведение, обозначаемое $(G_1 \times G_2, *)$, представляет собой просто $\{(g_1, g_2) : g_1 \in G_1, g_2 \in G_2\}$, где $(g_1, g_2) * (g_1', g_2')$ равно $(g_1 *_1 g_1', g_2 *_2 g_2')$. Произведение n групп, $G_1 \times G_2 \times \cdots \times G_n$, может быть определено аналогичным образом; используя эту форму записи, китайская теорема об остатках может быть сформулирована на языке теории групп следующим образом.

Теорема 9.32 (китайская теорема об остатках, версия II). Если $m_0, m_1, \cdots, m_n$ – попарно взаимно простые числа, то

$$\mathbb{Z}_{m_0 \cdot m_1 \cdot \ldots \cdot m_n} \cong \mathbb{Z}_{m_0} \times \mathbb{Z}_{m_1} \times \cdots \times \mathbb{Z}_{m_n}.$$

Задача 9.33. Докажите теорему 9.32.

9.3. Отношения

В этом разделе мы представляем основы отношений. При наличии двух множеств X и Y $X \times Y$ обозначает множество (упорядоченных) пар $\{(x, y) | x \in X \wedge y \in Y\}$, и отношение R равно подмножеству $X \times Y$, то есть $R \subseteq X \times Y$. Тем самым элементы R имеют форму (x, y), и мы записываем $(x, y) \in R$ (можно также записать xRy, Rxy или $R(x, y)$). В дальнейшем мы исходим из допущения, что мы квантифицируем над множеством X и что $R \subseteq X \times X$; мы говорим, что

1) R является *рефлексивным*, если $\forall x, (x, x) \in R$;
2) R является *симметричным*, если $\forall x \forall y$, $(x, y) \in R$ тогда и только тогда, когда $(y, x) \in R$;
3) R является *антисимметричным*, если $\forall x \forall y$, если $(x, y) \in R$ и $(y, x) \in R$, то $x = y$;
4) R является *транзитивным*, если $\forall x \forall y \forall z$, если $(x, y) \in R$ и $(y, z) \in R$, то также имеет место, что $(x, z) \in R$.

Предположим, что $R \subseteq X \times Y$ и $S \subseteq Y \times Z$. *Композиция* R и S определяется следующим образом:

$$R \circ S = \{(x, y) | \exists z, xRz \wedge zSy\}. \tag{9.6}$$

Пусть $R \subseteq X \times X$; мы можем определить $R^n := R \circ R \circ \cdots \circ R$ рекурсивно следующим образом:

$$R^0 = id_X := \{(x, x) | x \in X\} \tag{9.7}$$

и $R^{i+1} = R^i \circ R$. Обратите внимание, что в (9.7) существует два разных равенства; «=» является обычным равенством, а «:=» является определением.

Теорема 9.34. Следующие три отношения эквивалентны:

1) R является транзитивным;
2) $R^2 \subseteq R$;
3) $\forall n \geqslant 1, R^n \subseteq R$.

Задача 9.35. Докажите теорему 9.34.

Существует два стандартных способа представления *конечных* отношений, то есть отношений на $X \times Y$, где X и Y – это конечные множества. Пусть $X = \{a_1, \ldots, a_n\}$ и $Y = \{b_1, \ldots, b_m\}$, тогда мы можем представить отношение $R \subseteq X \times Y$:

1) в виде матрицы $M_R = (m_{ij})$, где:

$$m_{ij} = \begin{cases} 1 & (a_i, b_j) \in R \\ 0 & (a_i, b_j) \notin R \end{cases},$$

2) и в виде ориентированного графа $G_R = (V_R, E_R)$, где $V_R = X \cup Y$ и $_{a_i}\bullet \to \bullet_{b_j}$ является ребром в E_R тогда и только тогда, когда $(a_i, b_j) \in R$.

9.3.1. Замыкание

Пусть P равно свойству[1] отношений, например транзитивности или симметрии. Пусть $R \subseteq X \times X$ равно отношению, со свойством P либо без него. Отношение S, удовлетворяющее следующим трем условиям:

1) S имеет свойство P;
2) $R \subseteq S$; (9.8)
3) $\forall Q \subseteq X \times X$, «$Q$ имеет P» из $R \subseteq Q$ следует, что $S \subseteq Q$,

называется замыканием R относительно P. Обратите внимание, что в некоторых случаях замыкания может не существовать. Также заметим, что условие 3 может быть заменено на

$$S \subseteq \bigcap_{Q \text{ имеет P}, R \subseteq Q} Q. \tag{9.9}$$

См. рис. 9.3 с примером рефлексивного замыкания.

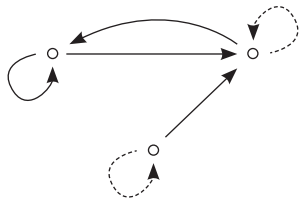


Рис. 9.3 ❖ Пример рефлексивного замыкания: без пунктирных линий эта диаграмма представляет собой отношение, которое не является рефлексивным; с пунктирными линиями оно рефлексивно, и это на самом деле наименьшее рефлексивное отношение, содержащее три точки и четыре сплошные линии

Теорема 9.36. Для $R \subseteq X \times X$ $R \cup id_X$ является рефлексивным замыканием R.

Задача 9.37. Доказите теорему 9.36.

---

[1] Мы встречали понятие абстрактного свойства в разделе 9.1.1. Единственная разница заключается в том, что в разделе 9.1.1 свойство P(i) было над $i \in n$, в то время как здесь при наличии множества X свойство определено над $Q \in \mathcal{P}(X \times X)$, то есть P($Q$), где $Q \subseteq X \times X$. В этом разделе, вместо того чтобы писать P(Q), мы говорим «Q имеет свойство P».

См. рис. 9.4 с примером симметричного замыкания.

Теорема 9.38. При наличии отношения $R \subseteq X \times Y$ отношение $R^{-1} \subseteq Y \times X$ определяется как $\{(x, y) | (y, x) \in R\}$. Для $R \subseteq X \times X$ $R \cup R^{-1}$ является симметричным замыканием R.

Задача 9.39. Докажите теорему 9.38.

См. рис. 9.5 с примером транзитивного замыкания.

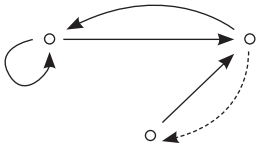


Рис. 9.4 ❖ Пример симметричного замыкания: без пунктирной линии эта диаграмма представляет несимметричное отношение; с пунктирными линиями оно симметрично

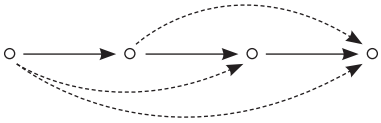


Рис. 9.5 ❖ Пример транзитивного замыкания: без пунктирной линии эта диаграмма представляет отношение, которое не является транзитивным; с пунктирными линиями оно – транзитивно

Теорема 9.40. $R^+ := \bigcup_{i=1}^{\infty} R^i$ является транзитивным замыканием R.

Доказательство. Мы проверяем, что R^+ имеет три условия, приведенных в (9.8). Во-первых, мы проверяем, имеет ли R^+ данное свойство, то есть является ли оно транзитивным:

$$\begin{aligned} xR^+y \wedge yR^+z &\Leftrightarrow \exists m, n \geqslant 1, xR^m y \wedge yR^n z \\ &\Rightarrow \exists m, n \geqslant 1, x(R^m \circ R^n)z \\ &\Leftrightarrow \exists m, n \geqslant 1, xR^{m+n}z \\ &\Leftrightarrow xR^+z. \end{aligned} \qquad (\dagger)$$

Таким образом, R^+ является транзитивным.

Во-вторых, мы проверяем, что $R \subseteq R^+$ – это следует из определения R^+. Мы проверяем последнее условие. Предположим, что S является транзитивным и $R \subseteq S$. Поскольку S является транзитивным, то по теореме 9.34 $S^n \subseteq S$ для $n \geqslant 1$, то есть $S^+ \subseteq S$, и поскольку $R \subseteq S$, то $R^+ \subseteq S^+$, поэтому $R^+ \subseteq S$.

Задача 9.41. Обратите внимание, что когда в доказательстве теоремы 9.40 мы показываем, что R^+ само является транзитивным, вторая строка, помеченная знаком (†), является импликацией, а не эквивалентностью, как другие строки. Почему она не является эквивалентностью?

Теорема 9.42. $R^* = \bigcup_{i=1}^{\infty} R^i$ является рефлексивным и транзитивным замыканием R.

Доказательство. $R^* = R^+ \cup id_X$. □

9.3.2. Отношение эквивалентности

Пусть X равно множеству, и пусть I равно индексному множеству. Семейство множеств $\{A_i | i \in I\}$ называется *разбиением* X тогда и только тогда, когда

1) $\forall i, A_i \neq \emptyset$;
2) $\forall i \neq j, A_i \cap A_j = \emptyset$;
3) $X = \bigcup_{i \in I} A_i$.

Обратите внимание, что $X = \bigcup_{x \in X}\{x\}$ – это *мельчайшее возможное разбиение*, то есть множество всех одноэлементных множеств. Отношение $R \subseteq X \times X$ называется *отношением эквивалентности* тогда и только тогда, когда

1) R является рефлексивным;
2) R является симметричным;
3) R является транзитивным.

Например, если x, y являются строками над $\{0,1\}^*$, то отношение, задаваемое $R = \{(x, y) | \text{длина}(x) = \text{длина}(y)\}$, является отношением эквивалентности. Еще одним примером является $xRy \Leftrightarrow x = y$, то есть отношение равенства является отношением эквивалентности в полном смысле слова. Еще один пример: $R = \{(a, b) | a \equiv b \pmod m\}$ является отношением эквивалентности (где «≡» – это отношение конгруэнтности, определенное на стр. 213).

Теорема 9.43. Рассмотрим отношение эквивалентности. Тогда следующее соблюдается:

1) $a \in [a]$;
2) $a \equiv b \Leftrightarrow [a] = [b]$;
3) $a \not\equiv b$, значит, $[a] \cap [b] = \emptyset$;
4) любые два класса эквивалентности либо равны, либо не пересекаются.

Теорема 9.44. Пусть $F : X \to X$ равно любой тотальной функции (то есть функции, определенной на всех ее входах). Тогда отношение R на X, определенное как xRy $F(x) \Leftrightarrow F(y)$, является отношением эквивалентности.

Задача 9.45. Докажите теорему 9.44.

Пусть R равно отношению эквивалентности на X. Для каждого $x \in X$ множество $[x]_R = \{y | xRy\}$ является *классом эквивалентности x по отношению к R*.

Теорема 9.46. Пусть $R \subseteq X \times X$ равно отношению эквивалентности. Следующие отношения эквивалентны:

1) aRb;
2) $[a] = [b]$;
3) $[a] \cap [b] \neq \emptyset$.

Доказательство. (1) ⇒ (2) Предположим, что aRb, и пусть $c \in [a]$. Тогда aRc, поэтому cRa (по симметрии). Поскольку $cRa \wedge aRb$, cRb (транзитивность), то bRc (симметрия), поэтому $c \in [b]$. Следовательно, $[a] \subseteq [b]$, и схожим образом $[b] \subseteq [a]$.

(2) ⇒ (3) очевидно, поскольку $[a]$ непустое, так как $a \in [a]$.

(3) ⇒ (1) пусть $c \in [a] \cap [b]$, тогда aRc и bRc, поэтому по симметрии $aRc \wedge cRb$.

И поэтому по транзитивности aRb. □

Следствие 9.47. Если R является отношением эквивалентности, то $(a, b) \notin R$ тогда и только тогда, когда $[a] \cap [b] = \emptyset$.

Для каждого отношения эквивалентности $R \subseteq X \times X$ пусть X/R обозначает множество всех классов эквивалентности множества R.

Теорема 9.48. X/R является разбиением множества X.

Доказательство. С учетом теоремы 9.46 единственное, что еще предстоит доказать, – это что $X = \bigcup_{A \in X/R} A$. Поскольку каждое $A = [a]$ для некоторого $a \in X$, из этого следует, что $\bigcup_{A \in X/R} A = \bigcup_{a \in X} [a] = X$. □

Пусть R_1, R_2 равны отношениям эквивалентности. Если $R_1 \subseteq R_2$, то мы говорим, что R_1 является детализацией R_2.

Лемма 9.49. Если R_1 есть уточнение R_2, то $[a]_{R_1} \subseteq [a]_{R_2}$ для всех $a \in X$.

Если X/R является конечным, то индекс(R) := $|X/R|$, то есть индекс R (в X) является размером X/R.

Теорема 9.50. Если $R_1 \subseteq R_2$, то индекс(R_1) $\geqslant$ индекс(R_2).

Задача 9.51. Докажите теорему 9.50.

9.3.3. Частичные порядки

В этом разделе, вместо того чтобы использовать R для представления отношения над множеством X, мы будем использовать разные варианты неравенства: $(X, \preccurlyeq)$, $(X, \sqsubseteq)$, $(X, \leqslant)$.

Отношение $\preccurlyeq$ над X, где $\preccurlyeq \subseteq X \times X$, называется частичным порядком, или частично упорядоченным множеством, если оно:

1) рефлексивно;
2) антисимметрично;
3) транзитивно.

Отношение «$\prec$» (где $\prec \subseteq X \times X$) является *четким частичным порядком*, если оно:

1) $x \prec y \Rightarrow \neg(y \prec x)$;
2) транзитивное.

Эти два стандартных отношения, «$\preccurlyeq$» и «$\prec$», естественным образом связаны в следующей далее теореме.

Теорема 9.52. Отношение $\preccurlyeq$, определенное как $x \preccurlyeq y \Leftrightarrow x \prec y \vee x = y$, является частичным порядком. То есть с учетом четкого частичного порядка «$\prec$» мы можем распространить его на частичный порядок «$\preccurlyeq$» с помощью стандартного символа равенства «=».

Пусть $(X, \preccurlyeq)$ равно частичному порядку. Мы говорим, что x, y *сравнимы*, если $x \preccurlyeq y$ или $y \preccurlyeq x$. В противном случае они *несравнимы*. Пусть $x \sim y$ сокращенно обозначают, что x и y несравнимы, то есть $x \sim y \Leftrightarrow \neg(x \preccurlyeq y) \wedge \neg(y \preccurlyeq x)$. В общем случае для каждой пары x, y только одно из следующего является истинным:

$$x \prec y, y \prec x, x = y, x \sim y.$$

Разумеется, в контексте частичных порядков, представленных символом «$\preccurlyeq$», смысл символа «$\prec$» выглядит следующим образом: $x \prec y \Leftrightarrow x \preccurlyeq y \wedge x \neq y$.

Частичный порядок $(X, \preccurlyeq)$ является *полным* (*тотальным*), или *линейным*, если все x, y сравнимы, то есть $\sim = \emptyset$. Вот несколько примеров частичных порядков: если

X является множеством, то $(\mathcal{P}(X), \subseteq)$ – это частично упорядоченное множество. Например, если $X = \{1, 2, 3\}$, то представление в виде диаграммы Хассе этого частично упорядоченного множества будет таким, как показано на рис. 9.6.

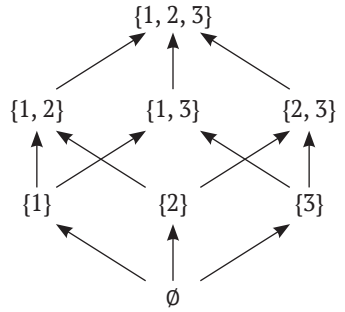


Рис. 9.6 ❖ Представление в виде диаграммы Хассе частично упорядоченного множества ($\{1, 2, 3\}, \subseteq$). Диаграммы Хассе являются транзитивными редукциями – вытекающие из транзитивности отношения не включены

Пусть $\mathbb{Z}^+$ равно множеству натуральных чисел, и пусть $a|b$ равно отношению «a делит b» (которое мы определяем на стр. 213). Тогда $(\mathbb{Z}^+, |)$ является частичным порядком.

Если $(X_1, \preccurlyeq_1)$, $(X_2, \preccurlyeq_2)$ являются двумя частичными порядками, то покомпонентный порядок $(X_1 \times X_2, \preccurlyeq_C)$ определяется следующим образом:

$$(x_1, x_2) \preccurlyeq (y_1, y_2) \Leftrightarrow x_1 \preccurlyeq_1 y_1 \wedge x_2 \preccurlyeq_2 y_2,$$

и он тоже является частичным порядком.

Лексикографический порядок $(X_1 \times X_2, \preccurlyeq_L)$ определяется следующим образом:

$$(x_1, x_2) \preccurlyeq_L (y_1, y_2) \Leftrightarrow (x_1 \preccurlyeq_1 y_1) \vee (x_1 = y_1 \wedge x_2 \preccurlyeq_2 y_2).$$

Наконец, $(X, \preccurlyeq)$ является *стратифицированным порядком* тогда и только тогда, когда $(X, \preccurlyeq)$ является частичным порядком, и, более того, $(x \sim y \wedge y \sim z) \Rightarrow (x \sim z \vee x = z)$. Определим $a \approx b \Leftrightarrow a \sim b \vee a = b$.

Теорема 9.53. Частичный порядок $(X, \preccurlyeq)$ является стратифицированным порядком тогда и только тогда, когда $\approx = \sim \cup id_X$ является отношением эквивалентности.

В математике номенклатура может быть для читателей величайшим бичом. Строка символов «$\approx = \sim \cup id_X$» представляет собой отличный пример запутывания; как ее понять? Действительно, она очень лаконична, но для ее прочтения требуется практика. Здесь мы говорим, что порядок, который мы назвали «$\approx$», на самом деле равен порядку, который мы получаем, взяв объединение порядка «$\sim$» и «id_X».

Задача 9.54. Докажите теорему 9.53.

Теорема 9.55. Частичный порядок $(X, \preccurlyeq)$ является стратифицированным порядком тогда и только тогда, когда существует полный порядок $(T, \preccurlyeq_T)$ и функция $f : X \to T$ такая, что f раскрывает и f представляет собой «порядковый гомоморфизм», то есть $a \preccurlyeq b \Leftrightarrow f(a) \preccurlyeq_T f(b)$.

Задача 9.56. Докажите теорему 9.55.

9.3.4. Решетки

Пусть $(X, \preccurlyeq)$ равно частичному порядку, и пусть $A \subseteq X$ равно некоторому подмножеству, и $a \in X$. Тогда:

1) a является минимальным в X, если $\forall x \in X, \neg(x \prec a)$;
2) a является максимальным в X, если $\forall x \in X, \neg(a \prec x)$;
3) a является наименьшим элементом в X, если $\forall x \in X, a \preccurlyeq x$;
4) a является наибольшим элементом в X, если $\forall x \in X, x \preccurlyeq a$;
5) a является верхней границей A, если $\forall x \in A, x \preccurlyeq a$;
6) a является нижней границей A, если $\forall x \in A, a \preccurlyeq x$;
7) a является наименьшей верхней границей (супремумом) A, обозначаемой $\sup(A)$, если
   а) $\forall x \in A, x \preccurlyeq a$,
   б) $\forall b \in X, (\forall x \in A, x \preccurlyeq b) \Rightarrow a \preccurlyeq b$;
8) a является наибольшей нижней границей (инфимуммом) A, обозначаемой $\inf(a)$, если
   а) $\forall x \in A, a \preccurlyeq x$,
   б) $\forall b \in X, (\forall x \in A, b \preccurlyeq x) \Rightarrow b \preccurlyeq a$.

Задача 9.57. Следует отметить, что в определениях 1–8 (в англоязычном оригинале текста) мы иногда используем определенный артикль «the», а иногда неопределенный артикль «a». В первом случае он подразумевает *уникальность*, во втором – наличие нескольких кандидатов. Убедитесь в уникальности, где это применимо, и приведите пример частичного порядка, где есть несколько кандидатов для данного элемента в других случаях. Наконец, важно отметить, что $\sup(A)$, $\inf(A)$ могут существовать или не существовать; приведите примеры, когда их не существует.

Частичный порядок $(X, \preccurlyeq)$ является вполне *упорядоченным множеством*, если он является полным порядком и для каждого $A \subseteq X$, такого что $A \neq \emptyset$, A, имеет наименьший элемент.

Частичный порядок является *плотным*, если $\forall x$ и y, если $x < y$, то $\exists z, x < z < y$. Например, $(\mathbb{R}, \leqslant)$ со стандартным определением для «$\leqslant$» является полным плотным порядком, но он не является вполне упорядоченным множеством; например, интервал $(2, 3]$, который соответствует подмножеству $\mathbb{R}$, состоящему из x, таких что $2 < x \leqslant 3$, не имеет наименьшего элемента.

Частичный порядок $(X, \preccurlyeq)$ является *решеткой*, если $\forall a, b \in X$, то обе операции, $\inf(\{A, B\})$ и $\sup(\{A, B\})$, существуют в X. Например, каждый полный порядок является решеткой, и $(\mathcal{P}(X), \subseteq)$ является решеткой для каждого $\preccurlyeq$. Этот последний пример обусловливает следующую форму записи: $a \sqcup b := \sup(\{a, b\})$ и $a \sqcap b := \inf(\{a, b\})$.

Задача 9.58. Докажите, что для решетки $(\mathcal{P}(X), \subseteq)$ мы имеем:

$A \sqcup B = A \cup B$;
$A \sqcap B = A \cap B$.

Не каждый частичный порядок является решеткой; рис. 9.7 дает простой пример.

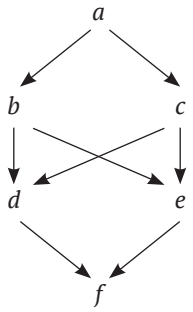


Рис. 9.7 ❖ Пример частичного порядка, который не является решеткой. В отличие от $\inf(\{b, c\}) = a$ и $\sup(\{d, e\}) = f$, супремума $\{b, c\}$ не существует

Теорема 9.59. Пусть $(X, \preccurlyeq)$ равно решетке. Тогда $\forall a, b \in X$,

$$a \preccurlyeq b \Leftrightarrow a \sqcap b = a \Leftrightarrow a \sqcup b = b.$$

Задача 9.60. Докажите теорему 9.59.

Теорема 9.61. Пусть $(X, \preccurlyeq)$ равно решетке. Тогда следующее соблюдается для всех $a, b, c \in X$:

1) $a \sqcup b = b \sqcup a$ и $a \sqcap b = b \sqcap a$ (коммутативность);
2) $a \sqcup (b \sqcup c) = (a \sqcup b) \sqcup c$ и $a \sqcap (b \sqcap c) = (a \sqcap b) \sqcap c$ (ассоциативность);
3) $a \sqcup a = a$ и $a \sqcap a = a$ (идемпотентность);
4) $a = a \sqcup (a \sqcap b)$ и $a = a \sqcap (a \sqcup b)$ (поглощение).

Задача 9.62. Докажите свойства, перечисленные в качестве теоремы 9.61.

Решетка $(X, \preccurlyeq)$ является полной тогда и только тогда, когда $\forall A \subseteq X$, $\sup(A)$, $\inf(A)$ обе существуют. Обозначим $\bot = \inf(X)$ и $\top = \sup(X)$.

Теорема 9.63. $(\mathcal{P}(x), \preccurlyeq)$ является полной решеткой, и следующее соблюдается: $\forall \mathcal{A} \subseteq \mathcal{P}(X)$, $\sup(\mathcal{A}) = \bigcup_{A \in \mathcal{A}} \mathcal{A}$ и $\inf(\mathcal{A}) = \bigcup_{A \notin \mathcal{A}} \mathcal{A}$, и $\bot = \emptyset$ и $\top = X$.

Задача 9.64. Докажите теорему 9.63.

Теорема 9.65. Каждая конечная решетка является полной.

Доказательство. Пусть $A = \{a_1, \ldots, a_n\}$. Определим $b = a_1 \sqcap \ldots \sqcap a_n$ (где скобки ассоциированы справа). Тогда $b = \inf(A)$. Та же идея для супремума. □

9.3.5. Теория неподвижных точек

Предположим, что F – это функция, и рассмотрим уравнение $\vec{x} = F(\vec{x})$. Решение $\vec{a}$ этого уравнения является *неподвижной точкой F*.

Пусть $(X, \preccurlyeq)$ и $(Y, \sqsubseteq)$ равны двум множествам. Функция $f: X \to Y$ является *монотонной* тогда и только тогда, когда $\forall x, y \in X, x \preccurlyeq y \Rightarrow f(x) \sqsubseteq f(y)$. Например, $f_B : \mathcal{P}(X) \to \mathcal{P}(X)$, где $B \subseteq X$, определенное $\forall x \subseteq X$ функцией $f_B(x) = B - x$, *не является монотонной*. С другой стороны, $g_B(x) = B \cup x$ и $h_B(x) = B \cap x$ обе являются монотонными.

Пусть $(X, \vec{x})$ равно частичному порядку, и пусть $f : X \to X$. Значение $x_0 \in X$ такое, что $x_0 = f(x_0)$ является, как мы видели, неподвижной точкой f. Неподвижная точка

может не существовать, например f_B в предыдущем абзаце не имеет неподвижной точки при $B \neq 0$, поскольку в таком случае уравнение $x = B - x$ не имеет решения. Также может существовать много неподвижных точек; например, $f(x) = x$ имеет $|X|$ число неподвижных точек.

Теорема 9.66 (Кнастера–Тарского (1)). Пусть $(X, \preccurlyeq)$ равно полной решетке, и пусть $f : X \to X$ равна монотонной функции. Тогда наименьшая неподвижная точка f существует и равна $\inf(\{x | f(x) \preccurlyeq x\})$.

Доказательство. Пусть $x_0 = \inf(\{x | f(x) \preccurlyeq x\})$. Сначала покажем, что $x_0 = f(x_0)$. Пусть $B = \{x | f(x) \preccurlyeq x\}$, и обратите внимание, что $B = \emptyset$, поскольку $\top = \sup(X) \in B$. Пусть $x \in B$, поэтому мы имеем $x_0 \preccurlyeq x$, отсюда, поскольку f является монотонной, $f(x_0) \preccurlyeq f(x)$, то есть

$$f(x_0) \preccurlyeq f(x) \preccurlyeq x.$$

Это верно для каждого x в B, поэтому $f(x_0)$ является нижней границей для B, и поскольку x_0 является наибольшей нижней границей B, из этого следует, что $f(x_0) \preccurlyeq x_0$.

Поскольку f монотонна, из этого следует, что $f(f(x_0)) \preccurlyeq f(x_0)$, а значит, $f(x_0)$ находится в B. Но тогда $x_0 \preccurlyeq f(x_0)$, а значит, $x_0 = f(x_0)$.

Остается показать, что x_0 является наименьшей неподвижной точкой. Пусть $x' = f(x')$. Это означает, что $f(x') \preccurlyeq x'$, то есть $x' \in B$. Но тогда $x_0 \preccurlyeq x'$. □

Теорема 9.67 (Кнастера–Тарского (2)). Пусть $(X, \preccurlyeq)$ равно полной решетке, и пусть $f : X \to X$ равно монотонной функции. Тогда существует наибольшая неподвижная точка уравнения $x = f(x)$, и она равна $\sup(\{x | f(x) \preccurlyeq x\})$.

Обратите внимание, что эти теоремы не являются конструктивными, но в случае конечного X существует конструктивный способ нахождения наименьшей и наибольшей неподвижных точек.

Теорема 9.68 (Кнастера–Тарского: конечные множества). Пусть $(X, \preccurlyeq)$ равно решетке, $|X| = m$, $f : X \to X$ равно монотонной функции. Тогда $f^m(\bot)$ равно наименьшей неподвижной точке, и $f^m(\top)$ равно наибольшей неподвижной точке.

Доказательство. Так как $|X| = m$, $(X, \preccurlyeq)$ является полной решеткой, $\bot = \inf(X)$ и $\top = \sup(X)$ обе существуют. Поскольку f является монотонной, и $\bot \preccurlyeq f(\bot)$, мы имеем $f(\bot) \preccurlyeq f(f(\bot))$, то есть $f(\bot) \preccurlyeq f^2(\bot)$. Продолжая применять монотонность, мы получаем:

$$f^0(\bot) = \bot \preccurlyeq f(\bot) \preccurlyeq f^2(\bot) \preccurlyeq f^3(\bot) \preccurlyeq \cdots \preccurlyeq f^i(\bot) \preccurlyeq f^{i+1}(\bot) \preccurlyeq \cdots.$$

Рассмотрим приведенную выше последовательность вплоть до $f^m(\bot)$. Она имеет длину $(m + 1)$, но X имеет только m элементов, поэтому существует $i < j$ таких, что $f^i(\bot) = f^j(\bot)$. Поскольку $\preccurlyeq$ является порядком, то из этого следует, что

$$f^i(\bot) = f^{i+1}(\bot) = \cdots = f^j(\bot),$$

поэтому $x^0 = f^i(\bot)$ является неподвижной точкой, так как

$$f(x_0) = f(f^i(\bot)) = f^{i+1}(\bot) = f^i(\bot) = x_0.$$

Ясно, что $f^{j+1}(\bot) = f(f^j(\bot)) = f(x_0) = x_0$, поэтому, по сути дела. $\forall k \geqslant i$, $x_0 = f^k(\bot)$, и, значит, $f^m(\bot) = x_0$, поэтому $f^m(\bot)$ является неподвижной точкой.

Теперь предположим, что x является еще одной неподвижной точкой f, то есть $x = f(x)$. Поскольку $\bot \preccurlyeq x$ и f является монотонной, мы заключаем, что $f(\bot) \preccurlyeq f(x) = x$, то есть $f(\bot) \preccurlyeq x$. Опять же, поскольку f монотонна, $f(f(\bot)) \preccurlyeq f(x) = x$, поэтому $f^2(\bot) \preccurlyeq x$. Следовательно, повторяя эту процедуру достаточно много раз, мы получаем $f^i(\bot) \preccurlyeq x$ для каждого i, и поэтому мы получаем $x_0 = f^m(\bot) \preccurlyeq x$.

Мы приводим аналогичную аргументацию для «наибольшего». □

Ситуация гораздо лучше для стандартной решетки $(\mathcal{P}(X), \subseteq)$, если X является конечным множеством.

Теорема 9.69. Пусть X равно конечному множеству, $|X| = n$, $f : \mathcal{P}(X) \to \mathcal{P}(X)$ является монотонной. Тогда $f^{n+1}(\emptyset)$ – это наименьшая неподвижная точка и $f^{n+1}(X)$ – наибольшая неподвижная точка.

Доказательство. Обратите внимание, что предыдущая теорема говорит, что $f^2(\emptyset)$ является наименьшей неподвижной точкой и $f^2(X)$ – наибольшей неподвижной точкой, так как $|\mathcal{P}(X)| = 2^n$, $\bot = \emptyset$ и $\top = X$, для решетки $(\mathcal{P}(X), \subseteq)$. Но эта теорема утверждает, что $(n + 1)$ вместо 2^n. Причина в том, что $\emptyset \subseteq f(\emptyset) \subseteq f^2(\emptyset) \subseteq \cdots \subseteq f^{n+1}(\emptyset)$ *должно* иметь два повторяющихся множества (так как $|X| = n$). □

Задача 9.70. Рассмотрите решетку $(\mathcal{P}(\{A, b, c\}), \subseteq)$, а также функции $f(x) = x \cup \{a, b\}$ и $g(x) = x \cap \{a, b\}$. Вычислите их соответствующие наименьшие/наибольшие неподвижные точки.

Пусть $(X, \preccurlyeq)$ равно полной решетке. Функция $f : X \to X$ называется:

1) *восходяще-непрерывной* тогда и только тогда, когда $\forall A \subseteq X$, $f(\sup(A)) = \sup(f(A))$;
2) *нисходяще-непрерывной* тогда и только тогда, когда $\forall A \subseteq X, f(\inf(A)) = \inf(f(A))$;
3) *непрерывной*, если она является и восходяще-, и нисходяще-непрерывной.

Лемма 9.71. Если $f : X \to X$ является восходяще-(нисходяще-)непрерывной, то, значит, она является монотонной.

Доказательство. Пусть f является восходяще-непрерывной и $x \preccurlyeq y$, значит, $x = \inf(\{x, y\})$ и $y = \sup(\{x, y\})$ и

$$f(x) \preccurlyeq \sup(\{f(x), f(y)\}) = \sup(f(\{x, y\})) = f(\sup(\{x, y\})) = f(y).$$

Аналогичный аргумент для нисходяще-непрерывной. □

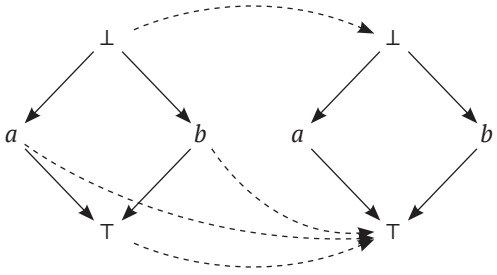


Рис. 9.8 ❖ Пример упорядочения над $X = \{a, b, \bot, \top\}$ функцией $f : X \to X$, обозначенного пунктирными линиями. То есть $f(\bot) = \top$ и $f(a) = f(b) = f(\top) = \top$. Путем непосредственной проверки может быть установлено, что f является монотонной, но она не является нисходяще-непрерывной

Задача 9.72. Покажите, что функция f на рис. 9.8 не является восходяще-непрерывной. Приведите пример монотонной функции g, которая не является ни восходяще-, ни нисходяще-непрерывной.

Теорема 9.73 (Клини). Если $(X, \preccurlyeq)$ является полной решеткой, $f : X \to X$ является восходяще-непрерывной функцией, тогда $x_0 = \sup(\{f^n(\bot) | n = 1, 2, \ldots\})$ является наименьшей неподвижной точкой f.

Доказательство. Обратите внимание, что $\bot \preccurlyeq f(\bot)$, поэтому по монотонности функции f мы имеем, что

$$\bot \preccurlyeq f(\bot) \preccurlyeq f^2(\bot) \preccurlyeq f^3(\bot) \preccurlyeq \cdots \tag{9.10}$$

и

$$f(x_0) = f(\sup(\{f^n(\bot) | n = 1, 2, \ldots\})),$$

и так как f является восходяще-непрерывной

$$= \sup(f(\{f^n(\bot) | n = 1, 2, \ldots\}))$$
$$= \sup(\{f^{n+1}(\bot) | n = 1, 2, \ldots\})$$

и по (9.10)

$$= \sup(\{f^n(\bot) | n = 1, 2, \ldots\}) = x_0,$$

то $f(x_0) = x_0$, то есть x_0 является неподвижной точкой.

Пусть $x = f(x)$. Мы имеем $\bot \preccurlyeq X$ и f является монотонной, поэтому $f(\bot) \preccurlyeq f(x) = x$, то есть $f(\bot) \preccurlyeq X$, $f^2(\bot) \preccurlyeq f(x) = x$, и т. д., то есть $f^n(\bot) \preccurlyeq x$ для всех n, поэтому по определению sup:

$$x_0 = \sup(\{f^n(\bot) | n = 1, 2, \ldots\}) \preccurlyeq x,$$

таким образом, x_0 является наименее неподвижной точкой. □

9.3.6. Рекурсия и неподвижные точки

До сих пор мы доказывали правильность циклов while и for, но есть и другой способ «организации циклов» с использованием *рекурсивных процедур*, то есть алгоритмов, которые «вызывают самих себя». Примеры таких алгоритмов мы увидим в главе, посвященной методу «разделяй и властвуй».

Существует робастная теория правильности рекурсивных алгоритмов, основанная на теории неподвижных точек, в частности на теореме Клини (теорема 9.73). Мы кратко проиллюстрируем этот подход на примере. Рассмотрим рекурсивный алгоритм 9.1.

Алгоритм 9.1. $F(x, y)$

```
1: if x = y then
2:         return y + 1
3: else
4:         F(x, F(x − 1, y + 1))
5: end if
```

Для того чтобы увидеть, как работает этот алгоритм, рассмотрим вычисление $F(4, 2)$. Сначала в строке 1 констатируется, что $4 \neq 2$, и поэтому мы должны вычислить $F(4, F(3, 3))$. Мы сначала рекурсивно вычисляем $F(3, 3)$, поэтому в строке 1 теперь констатируется, что $3 = 3$, и поэтому в строке 2 y устанавливается равным 4, и это является возвращаемым значением, то есть $F(3, 3) = 4$, поэтому теперь мы можем вернуться и вычислить $F(4, F(3, 3)) = F(4, 4)$, тем самым в строке 1 мы опять рекурсивно констатируем, что $4 = 4$, и поэтому в строке 2 y устанавливается равным 5, а это является возвращаемым значением, то есть $F(4, 2) = 5$. С другой стороны, легко заметить, что

$$F(3, 5) = F(3, F(2, 6)) = F(3, F(2, F(1, 7))) = \cdots,$$

и эта процедура никогда не заканчивается, так как x никогда не будет равен y. Таким образом, F не является тотальной функцией, то есть она не определена на всех $(x, y) \in \mathbb{Z} \times \mathbb{Z}$.

Задача 9.74. Какова область определения F, вычисляемой алгоритмом 9.1? То есть область F равна $\mathbb{Z} \times \mathbb{Z}$, тогда как область определения является наибольшим подмножеством $S \subseteq \mathbb{Z} \times \mathbb{Z}$ таким, что F определена для всех $(x, y) \in S$. Мы уже увидели, что $(4, 2) \in S$, тогда как $(3, 5) \notin S$.

Теперь мы рассмотрим три разные функции, все заданные алгоритмами, которые не являются рекурсивными: алгоритмы 9.2, 9.3 и 9.4, вычисляющие соответственно функции f_1, f_2 и f_3.

Функция f_1 имеет интересное свойство: если бы мы заменили F в алгоритме 9.1 на f_1, то мы бы вернули F. Другими словами, при наличии алгоритма 9.1, если бы мы заменили строку 4 на $f_1(x, f_1(x - 1, y + 1))$ и вычисляли f_1 с помощью(нерекурсивного) алгоритма 9.2 для f_1, то измененный таким образом алгоритм 9.1 теперь вычислял бы $F(x, y)$. Поэтому мы говорим, что функция f_1 является *фиксированной точкой* рекурсивного алгоритма 9.1.

Алгоритм 9.2. $f_1(x, y)$

```
if x = y then
        return y + 1
else
        return x + 1
end if
```

Например, вспомните, что мы уже показали, что $F(4, 2) = 5$, используя рекурсивный алгоритм 9.1 для вычисления F. Заменим строку 4 алгоритма 9.1 на $f_1(x, f_1(x - 1, y + 1))$ и вычисим $F(4, 2)$ заново; поскольку $4 \neq 2$, перейдем непосредственно к строке 4, где мы вычисляем $f_1(4, f_1(3, 3)) = f_1(4, 4) = 5$. Обратите внимание, что это последнее вычисление не было рекурсивным, так как мы вычислили f_1 непосредственно с помощью алгоритма 9.2, и что мы получили то же самое значение.

Рассмотрим теперь f_2, f_3, вычисляемые соответственно алгоритмами 9.3, 9.4.

Алгоритм 9.3. $f_2(x, y)$

```
if x ⩾ y then
        return x + 1
else
        return y – 1
end if
```

Алгоритм 9.4. $f_3(x, y)$

```
if x ⩾ y ∧ (x – y является четным) then
        return x + 1
end if
```

Обратите внимание, что если в алгоритме 9.4 не имеет место случай, что $x \geqslant y$ и $x - y$ не является четным, то результат не определен. Таким образом, f_3 является частичной функцией, и если $x < y$ или $x - y$ не является четным, то (x, y) не находится в ее области определения.

Задача 9.75. Докажите, что f_1, f_2, f_3 являются неподвижными точками алгоритма 9.1.

Функция f_3 имеет одно дополнительное свойство. Для каждой пары целых чисел x, y таких, что $f_3(x, y)$ определена, то есть $x \geqslant y$ и $x - y$ является четным, обе функции $f_1(x, y)$ и $f_2(x, y)$ также определены и имеют то же значение, что и $f_3(x, y)$. Мы говорим, что f_3 менее определена или равна f_1 и f_2, и пишем $f_3 \sqsubseteq f_1$ и $f_3 \sqsubseteq f_2$; то есть мы определили (неформально) частичный порядок на функциях $f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \times \mathbb{Z}$.

Задача 9.76. Покажите, что $f_3 \sqsubseteq f_1$ и $f_3 \sqsubseteq f_2$. Напомним понятие области определения, введенное в задаче 9.74. Пусть S_1, S_2, S_3 равны областям определения соответственно f_1, f_2, f_3. Вы должны показать, что $S_3 \subseteq S_1$ и $S_3 \subseteq S_2$.

Можно показать, что f_3 обладает этим свойством не только по отношению к f_1 и f_2, но и по отношению ко всем неподвижным точкам алгоритма 9.1. Кроме того, $f_3(x, y)$ является единственной функцией, имеющей это свойство, и поэтому f_3 считается *наименьшей (определенной) неподвижной точкой* алгоритма 9.1. Важным применением теоремы Клини (теоремы 9.73) является то, что каждый рекурсивный алгоритм имеет единственную неподвижную точку.

9.4. Логика

Мы представим основы пропозициональной и предикатной логики с целью определения арифметики Пеано. Арифметика Пеано является стандартной формализацией теории чисел, и она является логической основой для раздела 9.4.4 – формальная верификация. Наше рассмотрение логики ограничено предоставлением этих общих сведений, но читатель может найти больше ресурсов в разделе примечаний.

9.4.1. Пропозициональная логика

Пропозициональные (булевы) формулы строятся из *пропозициональных (булевых) переменных*[1] $p_1, p_2, p_3, \ldots$ и логических связок $\neg, \wedge, \vee$, перечисленных в предисловии на стр. 13.

Для наших переменных мы часто используем различные метки (например, a, b, $c, \ldots, x, y, z, \ldots, p, q, r, \ldots$ и т. п.) как «метапеременные», которые обозначают переменные, и мы определяем пропозициональные формулы по структурной индукции: любая переменная p является формулой, и если α, β – это формулы, то ими являются и $\neg\alpha$, $(\alpha \wedge \beta)$ и $(\alpha \vee \beta)$. Например, p, $(p \vee q)$, $(\neg(p \wedge q) \wedge (\neg p \vee \neg q))$. Напомним также из предисловия, что $\rightarrow$ и $\leftrightarrow$ являются соответственно связками импликации и эквивалентности.

Задача 9.77. Определите пропозициональные формулы с помощью контекстно-свободной грамматики.

Символ	Вес
$\neg$	0
$\wedge$, $\vee$, (	1
), p, для каждой переменной p	–1

Рис. 9.9 ❖ Закрепление «весов» за символами

Лемма 9.78. Закрепим веса за всеми символами, как показано на рис. 9.9. Вес любой формулы α равен –1, но вес надлежащего начального сегмента $\geqslant 0$. Следовательно, никакой надлежащий начальный сегмент формулы не является формулой.

Доказательство. По структурной индукции на длине α. Базовый случай: $w(p) = -1$ для любой переменной p. Индукционный шаг имеет три случая: $\neg\alpha$, $(\alpha \wedge \beta)$ и $(\alpha \vee \beta)$. Это показывает, что любая хорошо сформированная формула имеет вес –1. Теперь мы покажем, что любой надлежащий начальный сегмент имеет вес $\geqslant 0$. В базовом случае (единственная переменная p) начальные сегменты отсутствуют; на индукционном шаге предположим, что данное утверждение соблюдается для α и β (то есть любой начальный сегмент α и любой начальный сегмент β имеют вес $\geqslant 0$). Тогда то же относится и к $\neg\alpha$, так как любой начальный сегмент $\neg\alpha$ содержит $\neg$ (и $w(\neg) = 0$) и некоторый (возможно, пустой) начальный сегмент α. □

Задача 9.79. Закончите детали доказательства леммы 9.78.

Пусть $\alpha \overset{\text{syn}}{=} \alpha'$ подчеркивает, что α и α' равны как цепочка символов, то есть мы имеем не семантическое тождество, а синтаксическое тождество.

Теорема 9.80 (теорема об уникальной читаемости). Предположим, что α, β, α', β' являются формулами, c, c' являются бинарными связками и $(\alpha c \beta) \overset{\text{syn}}{=} (\alpha' c' \beta')$. Тогда $\alpha \overset{\text{syn}}{=} \alpha'$ и $\beta \overset{\text{syn}}{=} \beta'$ и $c \overset{\text{syn}}{=} c'$.

---

[1] Пропозициональные переменные иногда называются *атомами*. Очень тщательное и, возможно, теперь рассматриваемое немного старомодным обсуждение «имен» в логике (что такое «переменная», что такое «константа» и т. д.) можно найти в книге [Church (1996)], разделы 01 и 02.

Обратите внимание, что эта теорема говорит, что грамматика для генерирования формул однозначна. Или, другими словами, она говорит, что существует только один кандидат на главную связку, то есть что дерево разбора любой формулы уникально. Напомним, что в задаче 9.11 сравнивались инфиксная, префиксная, постфиксная формы записи; булевы формулы даны в инфиксной форме записи в том смысле, что бинарные операторы ($\wedge$, $\vee$) помещаются между операндами, и все же оно однозначно (тогда как задача 9.11 говорит, что, для того чтобы однозначно представить дерево, нам нужно два из трех представлений из списка {инфиксное, префиксное, постфиксное}). Разница в том, что в случае булевых формул у нас есть скобки для разделения подформул.

Задача 9.81. Покажите, что теорема 9.80 является следствием леммы 9.78. (Подсказка: определите вес формулы как сумму весов всех символов в ней.)

Закрепление истинностного значения представляет собой отображение τ : {переменные} $\to$ {T, F}. Здесь {T, F} означает «true» (истина) и «false» (ложь), иногда обозначаемые соответственно 0, 1. Закрепление истинностного значения τ может быть расширено для назначения T либо F каждой формуле следующим образом:

(1) $(\neg\alpha)^\tau = T$ тогда и только тогда, когда $\alpha^\tau = F$;

(2) $(\alpha \wedge \beta)^\tau = T$ тогда и только тогда, когда $\alpha^\tau = T$ и $\beta^\tau = T$;

(3) $(\alpha \vee \beta)^\tau = T$ тогда и только тогда, когда $\alpha^\tau = T$ или $\beta^\tau = T$.

Ниже приведены стандартные определения: мы говорим, что закрепление истинностного значения τ *удовлетворяет* формуле α, если $\alpha^\tau = T$, и τ *удовлетворяет* множеству формул Φ, если τ удовлетворяет всем $\alpha \in \Phi$. В свою очередь, множество формул Φ *удовлетворимо*, если некоторое τ удовлетворяет ему; в противном случае Φ *неудовлетворимо*. Мы говорим, что α является логическим следствием множества Φ, записываемого $\Phi \vDash \alpha$, если τ удовлетворяет α для каждого τ такого, что τ удовлетворяет Φ. Формула α является *валидной*, если $\vDash \alpha$, то есть $\alpha^\tau = T$ для всех τ. Валидная пропозициональная формула называется *тавтологией*. α и β являются *эквивалентными формулами* (записываемыми $\alpha \Leftrightarrow \beta$), если $\alpha \vDash \beta$ и $\beta \vDash \alpha$. Обратите внимание, что знаки «$\Leftrightarrow$» и «$\leftrightarrow$» имеют разные смыслы: один является семантическим логическим утверждением, а другой – синтаксическим логическим утверждением. При этом одно соблюдается тогда и только тогда, когда соблюдается другое.

Например, следующие утверждения являются тавтологиями: $p \vee \neg p$, $p \to p$, $\neg(p \wedge \neg p)$. Экземпляр логического следствия: $(p \wedge q) \vDash (p \vee q)$. Наконец, пример эквивалентности: $\neg(p \vee q) \Leftrightarrow (\neg p \wedge \neg q)$. Это последнее формальное суждение называется *законом де Моргана*.

Задача 9.82. Покажите, что если $\Phi \vDash \alpha$ и $\Phi \cup \{\alpha\} \vDash \beta$, то $\Phi \vDash \beta$.

Задача 9.83. Докажите следующую теорему о двойственности: пусть α' равно результату взаимной замены $\vee$ и $\wedge$ в α и замены p на $\neg p$ для каждой переменной p. Тогда $\neg\alpha \Leftrightarrow \alpha'$.

Задача 9.84. Докажите теорему об интерполяции Крейга: пусть α и β равны любым двум пропозициональным формулам. Пусть Var(α) равно множеству переменных, которые встречаются в α. Пусть S = Var(α) $\cap$ Var(β). Предположим, что S не пустое. Если $A \to B$ является валидным, то существует формула C такая, что

Var(C) = S, называемая «интерполянтом», такая, что $A \to C$ и $C \to B$ обе являются валидными.

Один из способов констатировать, что формула α с n переменными является тавтологией, состоит в верифицировании, что $\alpha^{\tau} = T$ для всех 2^n закреплений истинностных значений τ за переменными α. Подобный исчерпывающий метод может использоваться для верификации, что $\Phi \vDash \alpha$ (если Φ является конечным). Еще один способ – использовать понятие формального доказательства; ниже мы представим систему доказательств на основе пропозиционального исчисления PK (propositional kalkul), существующую благодаря немецкому логику Гентцену.

В провозициональной секвенциальной системе исчисления PK каждая строка в доказательстве является секвенцией вида:

$$S = \alpha_1, \ldots, \alpha_k \to \beta_1, \ldots, \beta_l,$$

где → – это новый символ и $\alpha_1, \ldots, \alpha_k$ и $\beta_1, \ldots, \beta_l$ – последовательности формул ($k, l \geqslant 0$), именуемые *цедентами* (соответственно *антецедент* и *сукцедент*).

Закрепление истинностного значения τ удовлетворяет секвенции S тогда и только тогда, когда τ фальсифицирует некоторую α_i или τ удовлетворяет некоторую β_i, то есть тогда и только тогда, когда τ удовлетворяет формуле:

$$\alpha_S = (\alpha_1 \wedge \cdots \wedge \alpha_k) \to (\beta_1 \vee \cdots \vee \beta_l).$$

Если антецедент является пустым, то → α эквивалентно α, и если сукцедент является пустым, то α → эквивалентно ¬α. Если оба цедента – антецедент и сукцедент – являются пустыми, то → является ложным (является неудовлетворимым).

У нас есть аналогичные определения валидности и логического следствия для секвенций. Например, приведем следующие валидные секвенции: α → α, → α, ¬α, α ∧ ¬α →.

Формальным доказательством в пропозициональном исчислении PK является конечное корневое дерево, в котором узлы помечены секвенциями. *Секвенция в корне* (внизу) – это то, что доказывается: *конечная секвенция* (endsequent). *Секвенции в листьях* (вверху) – это *логические аксиомы*, и они должны иметь форму α → α, где α – это формула. Каждая секвенция, отличная от логических аксиом, должна вытекать из своей родительской секвенции (секвенций) по одному из правил логического вывода, перечисленных на рис. 9.10.

Задача 9.85. Приведите доказательства в пропозициональном исчислении PK для каждой из следующих валидных секвенций: $\neg p \vee \neg q \to \neg(p \vee q)$, $\neg(p \vee q) \to \neg p \wedge \neg q$ и $\neg p \wedge \neg q \to \neg(p \vee q)$, а также $(p_1 \wedge (p_2 \wedge (p_3 \wedge p_4))) \to (((p_1 \wedge p_2) \wedge p_3) \wedge p_4)$.

Задача 9.86. Покажите, что правила сокращения могут быть выведены из правила усечения (с обменами и ослаблениями).

Задача 9.87. Предположим, что мы допустили наличие ↔ как примитивной связки вместо той, которая была введена по определению. Дайте соответствующие правила введения слева и справа для ↔.

Слабые структурные правила:

Обмен влево: $\dfrac{\Gamma_1, \alpha, \beta, \Gamma_2 \to \Delta}{\Gamma_1, \beta, \alpha, \Gamma_2 \to \Delta}$ Обмен вправо: $\dfrac{\Gamma \to \Delta_1, \alpha, \beta, \Delta_2}{\Gamma \to \Delta_1, \beta, \alpha, \Delta_2}$

Сокращение влево: $\dfrac{\Gamma, \alpha, \alpha \to \Delta}{\Gamma, \alpha \to \Delta}$ Сокращение вправо: $\dfrac{\Gamma \to \Delta, \alpha, \alpha}{\Gamma \to \Delta, \alpha}$

Ослабление влево: $\dfrac{\Gamma \to \Delta}{\alpha, \Gamma \to \Delta}$ Ослабление вправо: $\dfrac{\Gamma \to \Delta}{\Gamma \to \Delta, \alpha}$

Правило усечения:

$$\frac{\Gamma \to \Delta, \alpha \quad \alpha, \Gamma \to \Delta}{\Gamma \to \Delta}$$

Правила для введения связок:

$\neg$-влево: $\dfrac{\Gamma \to \Delta, \alpha}{\neg\alpha, \Gamma \to \Delta}$ $\neg$-вправо: $\dfrac{\alpha, \Gamma \to \Delta}{\Gamma \to \Delta, \neg\alpha}$

$\wedge$-влево: $\dfrac{\alpha, \beta, \Gamma \to \Delta}{(\alpha \wedge \beta), \Gamma \to \Delta}$ $\wedge$-вправо: $\dfrac{\Gamma \to \Delta, \alpha \quad \Gamma \to \Delta, \beta}{\Gamma \to \Delta, (\alpha \wedge \beta)}$

$\vee$-влево: $\dfrac{\alpha, \Gamma \to \Delta \quad \beta, \Gamma \to \Delta}{(\alpha \vee \beta), \Gamma \to \Delta}$ $\vee$-вправо: $\dfrac{\Gamma \to \Delta, \alpha, \beta}{\Gamma \to \Delta, (\alpha \vee \beta)}$

Рис. 9.10 ❖ Правила пропозиционального исчисления PK. Обратите внимание, что Γ, Δ обозначают конечные секвенции формул

Для каждого правила пропозиционального исчисления PK севенция внизу является логическим следствием секвенции (секвенций) вверху; назовем это *принципом разумности правила*. Например, в случае с $\vee$-вправом это может быть показано следующим образом: предположим, что τ удовлетворяет верхнюю секвенцию; предположим теперь, что оно удовлетворяет Γ. Тогда, поскольку τ удовлетворяет верхнюю, оно должно удовлетворить одно из перечисленного: Δ, α или β. Если оно удовлетворяет Δ, то доказательство завершено; если оно удовлетворяет одно из α, β, то оно удовлетворяет $\alpha \vee \beta$, и доказательство также завершено.

Задача 9.88. Проверьте принцип разумности правила: убедитесь, что каждое правило является разумным, то есть низ каждого правила является логическим следствием верха.

Теорема 9.89 (о разумности пропозиционального исчисления PK). Каждая доказуемая в пропозициональном исчислении PK секвенция валидна.

Доказательство. Покажем, что конечная секвенция в каждом доказательстве в пропозициональном исчислении PK валидна, по индукции на числе секвенций в доказательстве. В базовом случае доказательство представляет собой одну строку; аксиома $\alpha \to \alpha$, и она, очевидно, валидна. На индукционном шаге для каждого правила нужно только верифицировать, что если все верхние секвенции валидны, то нижняя секвенция валидна. Это вытекает из *принципа разумности правила*. □

Следующий далее принцип называется *принципом инверсии*: для каждого правила пропозиционального исчисления РК, за исключением ослабления, если валидна нижняя секвенция, то валидны все верхние секвенции.

Задача 9.90. Проверьте непосредственно каждое правило и докажите принцип инверсии. Приведем пример с ослабляющим правилом, для которого этот принцип не срабатывает.

Теорема 9.91 (о полноте пропозиционального исчисления РК). Каждая валидная пропозициональная секвенция доказуема в РК без использования усечения или сокращения.

Доказательство. Покажем, что каждая валидная секвенция $\Gamma \to \Delta$ имеет доказательство в пропозициональном исчислении РК по индукции на суммарном числе связок $\wedge$, $\vee$, $\neg$, встречающихся в $\Gamma \to \Delta$.

Базовый случай: ноль связок, и поэтому каждая формула в $\Gamma \to \Delta$ является переменной, а так как она является валидной, некая переменная p должна быть и в Γ, и в Δ. Следовательно, $\Gamma \to \Delta$ может быть выведено из $p \to p$ за счет ослаблений и обменов.

Индукционный шаг: предположим, что γ не является переменной в Γ или Δ. Тогда она имеет форму $\neg\alpha$, $(\alpha \wedge \beta)$, $(\alpha \vee \beta)$. Тогда $\Gamma \to \Delta$ может быть выведено одним из правил введения связок с использованием обменов.

Верхняя секвенция (секвенции) будет иметь на одну связку меньше, чем $\Gamma \to \Delta$, и будет валидна по принципу инверсии; следовательно, она имеет доказательства в пропозициональном исчислении РК по индукционной гипотезе. □

Задача 9.92. Каковы пять правил, которые не используются на индукционном шаге в приведенном выше доказательстве?

Задача 9.93. Рассмотрите пропозициональное исчисление РК′, которое похоже на РК, но где аксиомы должны иметь форму $p \to p$, то есть α должна быть переменной в логических аксиомах. Является ли РК′ по-прежнему завершенным?

Задача 9.94. Предположим, что $\{\to \beta_1, \ldots, \to \beta_n\} \vDash \Gamma \to \Delta$. Дайте доказательство в пропозициональном исчислении РК для $\Gamma \to \Delta$, где все листья являются либо логическими аксиомами $\alpha \to \alpha$, либо одной из нелогических аксиом $\to \beta_i$. (Подсказка: ваше доказательство потребует использования правила усечения.) Теперь приведем доказательство того, что при наличии конечного Φ такого, что $\Phi \vDash \Gamma \to \Delta$, существует доказательство РК для $\Gamma \to \Delta$, где все листья являются логическими аксиомами или секвенциями в Φ. Этим показывается, что РК также является импликационно завершенным.

9.4.1.1. Расширенное пропозициональное исчисление РК

Существует естественное расширение системы импликационного исчисления РК в то, что называется расширенным РК (extended РК, ЕРК). Стандартный метод математического доказательства состоит в возможности сокращения сложных формул, которые затем могут быть использованы в оставшейся части доказательства, вместо переписывания длинных формул всякий раз, когда они необходимы. Это можно проделывать на уровне пропозициональной логики, допуская аксиомы вида:

$p \leftrightarrow \alpha$,

где p – это новая переменная, которая еще не появлялась в доказательстве, и α – любая формула. Сила этой конструкции вытекает из вложенности этих определений, то есть α может задействовать некоторые ранее определенные новые переменные.

Задача 9.95. Покажите, что любое доказательство в пропозициональном исчислении EPK может быть переписано как доказательство в пропозициональном исчислении PK. Что происходит в общем случае с размером нового доказательства в пропозициональном исчислении PK?

Интересное наблюдение, выходящее за рамки этой книги, состоит в том, что, в отличие от пропозиционального исчисления PK, которое соответствует рассуждениям с булевыми формулами, расширенное пропозициональное исчисление EPK соответствует рассуждениям с булевыми схемами. См. работы [Cook и Nguyen (2010)], [Krajıcek (1995)] или [Cook и Soltys (1999)].

9.4.2. Первопорядковая логика

Первопорядковая логика, или логика первого порядка, также именуется предикатным исчислением. Начнем с определения языка $\mathcal{L} = \{f_1, f_2, f_3, \ldots, R_1, R_2, R_3, \ldots\}$, равного множеству символов функций и отношений. Каждый символ функции и отношения имеет связанную с ним *арность*, то есть число аргументов, которые он принимает. *$\mathcal{L}$-члены* (*$\mathcal{L}$-термы*) определяются по структурной индукции следующим образом: каждая переменная является членом: $x, y, z, \ldots, a, b, c, \ldots$; если f является символом n-арной функции и $t_1, t_2, \ldots, t_n$ являются членами, то ими являются и $ft_1t_2 \ldots t_n$. Символ 0-арной функции является константой (мы используем c и e в качестве метасимволов для констант). Например, если f является символом бинарной функции (с арностью 2) и G является символом унарной функции (с арностью 1), то $fgex$, fxy, $gfege$ являются членами.

Задача 9.96. Покажите теорему об уникальной читаемости для членов. См. теорему 9.80, для того чтобы освежить свои знания по уникальной читаемости в пропозициональном случае.

Например, язык арифметики, так называемая *арифметика Пеано*, задается языком $\mathcal{L}_A = [0, \mathrm{s}, +, \cdot; =]$. Вместо формальной префиксной формы записи для символов функций $\cdot$, + языка $\mathcal{L}_A$ мы используем инфиксную форму записи (определенную на стр. 234). То есть мы пишем $(t_1 \cdot t_2)$ вместо $\cdot t_1t_2$, и мы пишем $(t_1 + t_2)$ вместо $+t_1t_2$. Например, приведем следующие $\mathcal{L}_A$-члены: $sss0$, $sss0$, $((x + sy) \cdot (ssz + s0))$. Обратите внимание, что мы используем инфиксную форму записи со скобками, так как в противном случае форма записи была бы неоднозначной.

Мы строим $\mathcal{L}$-формулы следующим образом:

1) $Rt_1t_2 \ldots t_n$ – это *атомарная формула*, R – это символ n-арного предиката, $t_1, t_2, \ldots, t_n$ – это члены;
2) если α, β – это формулы, то ими являются и $\neg\alpha$, $(\alpha \vee \beta)$, $(\alpha \wedge \beta)$;
3) если α – это формула и x – это переменная, то $\forall x\alpha$ и $\exists x\alpha$ также являются формулами.

Например, $(\neg\forall xPx \vee \exists x\neg Px)$, $(\forall x\neg Qxy \wedge \neg\forall zQfyz)$ являются первопорядковыми формулами.

Задача 9.97. Покажите, что множество $\mathcal{L}$-формул может быть задано контекстно-свободной грамматикой.

Мы также используем инфиксную форму записи с предикатом равенства; то есть мы пишем $r = s$ вместо $=rs$ и $r \neq s$ вместо $\neg =rs$.

Вхождение x в α ограничено, если оно находится в подформуле α в форме $\forall x\beta$ или $\exists x\beta$ (то есть в области действия квантификатора). В противном случае это вхождение *свободно*. Например, в $\exists y(x = y + y)$ переменная x свободна, а y связана. В $Px \wedge \forall xQx$ переменная x появляется и как свободная, и как ограниченная. Член t или формула α *замкнуты*, если они не содержат свободных переменных. Замкнутая формула называется *сентенцией*, или *предложением*.

Теперь мы представим способ назначения смысла первопорядковым формулам: *семантику Тарского*; мы будем использовать стандартную терминологию и ссылаться на семантику Тарского как на *основные семантические определения* (basic semantic definitions, BSD).

Структура (или интерпретация) придает смысл членам и формулам. $\mathcal{L}$-структура $\mathcal{M}$ состоит из:

1) непустого множества M, именуемого универсумом дискурса;
2) для каждой n-арной f $f^{\mathcal{M}} : M^n \to M$;
3) для каждого n-арного P $P^{\mathcal{M}} \subseteq M^n$.

Если $\mathcal{L}$ содержит $=$, то $=^{\mathcal{M}}$ должно быть обычным $=$. Таким образом, равенство является особым – оно всегда должно быть истинным равенством. С другой стороны, $<^{\mathcal{M}}$ может быть чем угодно, не обязательно отношением порядка, к которому мы привыкли.

Каждая $\mathcal{L}$-сентенция становится либо истинной, либо ложной при интерпретации $\mathcal{L}$-структурой $\mathcal{M}$. Если сентенция α становится истинной при $\mathcal{M}$, то мы говорим, что $\mathcal{M}$ удовлетворяет α, или $\mathcal{M}$ является моделью для α, и пишем $\mathcal{M} \vDash \alpha$.

Если α имеет свободные переменные, то они должны получать значения из M (универсума дискурса), прежде чем α может получить истинностное значение при $\mathcal{M}$. Закрепление значения за объектом, или объектное закрепление (object assignment) σ, для структуры $\mathcal{M}$ является отображением из переменных в универсум M. В данном контексте $t^{\mathcal{M}}[\sigma]$ – это элемент в M, задаваемый структурой $\mathcal{M}$ и объектным закреплением σ. $\mathcal{M} \vDash \alpha[\sigma]$ означает, что $\mathcal{M}$ удовлетворяет α, когда его свободным переменным назначаются значения посредством σ.

Это должно быть определено очень тщательно; мы покажем, как вычислять $t^{\mathcal{M}}[\sigma]$ по структурной индукции:

1) $x^{\mathcal{M}}[\sigma]$ равно $\sigma(x)$;
2) $(ft_1t_2 \ldots t_n)^{\mathcal{M}}[\sigma]$ равно $f^{\mathcal{M}}(t_1^{\mathcal{M}}[\sigma], t_2^{\mathcal{M}}[\sigma], \ldots, t_n^{\mathcal{M}}[\sigma])$.

Если x является переменной и m находится в универсуме дискурса, то есть $m \in M$, то $\sigma(m/x)$ является тем же самым объектным закреплением, что и σ, за исключением того, что x отображается в m. Теперь мы представим определение $\mathcal{M} \vDash \alpha[\sigma]$ по структурной индукции:

1) $\mathcal{M} \vDash (Pt_1 \ldots t_n)[\sigma]$ тогда и только тогда, когда $(t_1^{\mathcal{M}}[\sigma], \ldots, t_n^{\mathcal{M}}[\sigma]) \in P^{\mathcal{M}}$;
2) $\mathcal{M} \vDash \neg\alpha[\sigma]$ тогда и только тогда, когда $\mathcal{M} \nvDash \alpha[\sigma]$;
3) $\mathcal{M} \vDash (\alpha \wedge (\vee)\beta)[\sigma]$ тогда и только тогда, когда $\mathcal{M} \vDash \alpha[\sigma]$ и(или) $\mathcal{M} \vDash \beta[\sigma]$;
4) $\mathcal{M} \vDash (\forall(\exists)\, x\alpha)[\sigma]$ тогда и только тогда, когда $\mathcal{M} \vDash \alpha[\sigma(m/x)]$ для всех (некоторых) $m \in \mathcal{M}$.

Если t замкнута, то мы пишем $t^{\mathcal{M}}$; если α является сентенцией, то мы пишем $\mathcal{M} \vDash \alpha$.

Например, пусть $\mathcal{L} = [;R,=]$ (R – бинарный предикат), и пусть $\mathcal{M}$ равно $\mathcal{L}$-структуре с универсумом $\mathbb{N}$ и такой, что $(m, n) \in R^{\mathcal{M}}$ тогда и только тогда, когда $m \leqslant n$. Тогда $\mathcal{M} \vDash \exists x \forall y Rxy$, но $\mathcal{M} \nvDash \exists y \forall x Rxy$.

Стандартная структура $\underline{\mathbb{N}}$ для языка $\mathcal{L}_{\text{A}}$ имеет универсум $M = \mathbb{N}$, $s^{\underline{\mathbb{N}}}(n) = n + 1$ и $0, +, \cdot, =$ получают свои обычные смыслы на натуральных числах. Например, $\underline{\mathbb{N}} \vDash \forall x \forall y \exists z(x + z = y \vee y + z = x)$, но $\underline{\mathbb{N}} \nvDash \forall x \exists y(y + y = x)$.

Мы говорим, что формула α *удовлетворима* тогда и только тогда, когда $\mathcal{M} \vDash \alpha[\sigma]$ для некоторого $\mathcal{M}$ & σ. Пусть Φ обозначает множество формул; тогда $\mathcal{M} \vDash \Phi[\sigma]$ тогда и только тогда, когда $\mathcal{M} \vDash \alpha[\sigma]$ для всех $\alpha \in \Phi$. $\Phi \vDash \alpha$ тогда и только тогда, когда $((\forall \mathcal{M}, \sigma), (\mathcal{M} \vDash \Phi[\sigma] \rightarrow \mathcal{M} \vDash \alpha[\sigma])$, то есть α является логическим следствием Φ. Мы говорим, что формула α является *валидной* и пишем $\vDash \alpha$ тогда и только тогда, когда $\mathcal{M} \vDash \alpha[\sigma]$ для всех $\mathcal{M}$ & σ. Мы говорим, что α и β *логически эквивалентны*, и пишем $\alpha \Leftrightarrow \beta$ тогда и только тогда, когда для всех $\mathcal{M}$ & σ ($\mathcal{M} \vDash \alpha[\sigma]$ тогда и только тогда, когда $\mathcal{M} \vDash \beta[\sigma]$).

Обратите внимание, что является символом «метаязыка» (русского), в отличие от $\wedge, \vee, \exists, \ldots$, которые являются символами первопорядковой логики. Кроме того, если Φ представляет собой всего одну формулу, то есть $\Phi = \{\beta\}$, то мы пишем $\beta \vDash \alpha$ вместо $\{\beta\} \vDash \alpha$.

Задача 9.98. Покажите, что $(\forall x\alpha \vee \forall x\beta) \vDash \forall x(\alpha \vee \beta)$ для всех формул α и β.

Задача 9.99. Имеет ли место, что $\forall x(\alpha \vee \beta) \vDash (\forall x\alpha \vee \forall x\beta)$?

Предположим, что t, u являются членами. Тогда:

- $t(u/x)$ – результат замены всех вхождений x в t на u;
- $\alpha(u/x)$ – результат замены всех свободных вхождений x в α на u.

Семантически $(u(t/x))^{\mathcal{M}}[\sigma] = u^{\mathcal{M}}[\sigma(m/x)]$, где $m = t^{\mathcal{M}}[\sigma]$.

Например, пусть $\mathcal{M}$ равно $\underline{\mathbb{N}}$ (стандартной структуре) для $\mathcal{L}_{\text{A}}$. Предположим $\sigma(x) = 5$ и $\sigma(y) = 7$. Пусть:

u равно члену $x + y$;
t равно члену $ss0$.

Тогда:

$u(t/x)$ равно $ss0 + y$ и $(u(t/x))^{\underline{\mathbb{N}}}[\sigma] = 2 + 7 = 9$.

Схожим образом $m = t^{\underline{\mathbb{N}}} = 2$, поэтому $u^{\underline{\mathbb{N}}}[\sigma(m/x)] = 2 + 7 = 9$.

Задача 9.100. Докажите, что $(u(t/x))^{\mathcal{M}}[\sigma] = u^{\mathcal{M}}[\sigma(m/x)]$, где $m = t^{\mathcal{M}}[\sigma]$, используя структурную индукцию на u.

Задача 9.101. Применим ли результат задачи 9.100 к формулам α? То есть является ли истиной, что $\mathcal{M} \vDash \alpha(t/x)[\sigma]$ тогда и только тогда, когда $\mathcal{M} \vDash \alpha[\sigma(m/x)]$, где $m = t^{\mathcal{M}}[\sigma]$?

Например, предположим, что α равно $\forall y\neg(x = y + y)$. Здесь написано «x является нечетным». Но $\alpha(x + y/x)$ равно $\forall y\neg(x + y = y + y)$, которое всегда ложно, независимо от значения $\sigma(x)$. Задача в том, что y в члене $x + y$ был «пойман» квантором $\forall y$.

Член t свободно замещаем на x в α тогда и только тогда, когда нет ни одного свободного вхождения x в α в подформуле α формы $\forall y\beta$ или $\exists y\beta$, где y входит в t.

Теорема 9.102 (теорема о замещении). Если t свободно замещаем на x в α, то для всех структур $\mathcal{M}$ и всех объектных закреплений σ имеет место, что $\mathcal{M} \vDash \alpha(t/x)[\sigma]$ тогда и только тогда, когда $\mathcal{M} \vDash \alpha[\sigma(m/x)]$, где $m = t^{\mathcal{M}}[\sigma]$.

Задача 9.103. Докажите теорему о замещении. (Подсказка. Используйте структурную индукцию на α и основные семантические определения.)

Если член t не является свободно замещаемым на x в α, то причиной тому является то, что некая переменная y в t поймана квантификатором $\forall y$ или $\exists y$ в α. Один из способов это исправить состоит в простом переименовании «ограниченной» переменной y в α в некую новую переменную z. Это переименование не меняет смысл α.

Пусть $a, b, c, \ldots$ обозначают свободные переменные, и пусть $x, y, z, \ldots$ обозначают ограниченные переменные. Первопорядковая формула α называется *надлежащей формулой*, если она удовлетворяет ограничению, что она не имеет ни одного свободного вхождения любой «ограниченной» переменной и ни одного ограниченного вхождения любой «свободной» переменной. Схожим образом *надлежащий член* не имеет ни одной «ограниченной» переменной. Обратите внимание, что подформула надлежащей формулы не обязательно является надлежащей, и надлежащая формула может содержать члены, которые не являются надлежащими.

Секвенциальная система исчисления LK является расширением пропозициональной системы исчисления PK, где теперь все формулы в секвенции $\alpha_1, \ldots, \alpha_k \to \beta_1, \ldots, \beta_l$ должны быть надлежащими формулами. Пропозициональная система исчисления LK является пропозициональной системой исчисления PK вместе с четырьмя правилами введения кванторов, приведенными на рис. 9.11.

$$\text{Введение } \forall\text{:}\quad \frac{\alpha(t), \Gamma \to \Delta}{\forall x\alpha(x), \Gamma \to \Delta} \qquad \frac{\Gamma \to \Delta, \alpha(b)}{\Gamma \to \Delta, \forall x\alpha(x)}$$

$$\text{Введение } \exists\text{:}\quad \frac{\alpha(b), \Gamma \to \Delta}{\exists x\alpha(x), \Gamma \to \Delta} \qquad \frac{\Gamma \to \Delta, \alpha(t)}{\Gamma \to \Delta, \exists x\alpha(x)}$$

Рис. 9.11 ❖ Расширение исчисления PK до исчисления LK

Существуют некоторые ограничения в использовании правил, приведенных на рис. 9.11. Во-первых, t – это надлежащий член, и $\alpha(t)$ (соответственно $\alpha(b)$) – это результат замещения t (соответственно b) для всех свободных вхождений x в $\alpha(x)$. Обратите внимание, что t, b можно свободно заместить на x в $\alpha(x)$, поскольку $\forall x\alpha(x)$, $\exists x\alpha(x)$ имеют надлежащие формулы. Свободная переменная b не должна входить в заключение в $\forall$ справа и $\exists$ слева.

Задача 9.104. Покажите, что четыре новых правила разумны.

Задача 9.105. Приведите конкретный пример секвенции $\Gamma \to \Delta, \alpha(b)$, которая является валидной, но нижняя секвенция $\Gamma \to \Delta, \forall x\alpha(x)$ не является валидной, поскольку ограничение, накладываемое на b, нарушается (b входит в Γ, или Δ, или $\forall x\alpha(x)$). Сделайте то же самое для $\exists$ слева.

Доказательство в исчислении LK валидной первопорядковой секвенции может быть получено тем же методом, что и в пропозициональном случае. Напишите

целевую секвенцию внизу и двигайтесь вверх, используя правила введения в обратном порядке. Если есть выбор, какой квантификатор удалить следующим, выбирайте ∀ справа либо ∃ слева (продвигаясь назад), так как эти правила несут ограничение.

9.4.3. Арифметика Пеано

Вспомним язык арифметики $\mathcal{L}_A = [0, s, +, \cdot; =]$. Аксиомы для арифметики Пеано следующие:

P1 $\forall x(sx \neq 0)$;
P2 $\forall x \forall y(sx = sy \to x = y)$;
P3 $\forall x(x + 0 = x)$;
P4 $\forall x \forall y(x + sy = s(x + y))$;
P5 $\forall x(x \cdot 0 = 0)$;
P6 $\forall x \forall y(x \cdot sy = x \cdot y + x)$

плюс индукционная схема:

$$\forall y_1 \dots \forall_{y_k}[(\alpha(0) \to \forall x(\alpha(x) \to \alpha(sx))) \to \forall x \alpha(x)], \tag{9.11}$$

где α – это любая $\mathcal{L}_A$-формула, а (9.11) – сентенция. Обратите внимание, что это определение является формальным определением индукции, приведенным в разделе 9.1.1.

У нас также есть схема аксиом равенства.

E1 $\forall x(x = x)$;
E2 $\forall x \forall y(x = y \to y = x)$;
E3 $\forall x \forall y \forall z((x = y \wedge y = z) \to x = z)$;
E4 $\forall x_1 \dots \forall x_n \forall y_1 \dots \forall y_n(x_1 = y_1 \wedge \dots \wedge x_n = y_n) \to fx_1 \dots x_n = fy_1 \dots y_n$;
E5 $\forall x1 \dots \forall x_n \forall y_1 \dots \forall y_n(x_1 = y_1 \wedge \dots \wedge x_n = y_n) \to Px_1 \dots x_n \to Py_1 \dots y_n$,

где E4 и E5 соблюдаются для всех n-арных функциональных и предикатных символов. В $\mathcal{L}_A$, который является интересуемым нас языком, s является унарным, $+, \cdot$ являются бинарными и $=$ является бинарным.

Пусть арифметика Пеано на основе исчисления LK равна системе исчисления LK, где листьям разрешено быть P1-6 и E1-5, помимо обычных аксиом $\alpha \to \alpha$. Например, $\to \forall x(x = x)$ будет валидным листом.

Задача 9.106. Покажите, что арифметика Пеано на основе LK доказывает, что все ненулевые элементы имеют предшественника.

Задача 9.107. Покажите, что арифметика Пеано на основе LK доказывает следующее: ассоциативный и коммутативный закон сложения, ассоциативные и коммутативные законы умножения и что умножение распределяется над сложением. Скрупулезно укажите, какие аксиомы вы используете.

9.4.4. Формальная верификация

Доказательства правильности, которые мы приводили до этого, считаются «неформальными» математическими доказательствами. Нет ничего плохого в неформальном доказательстве, и во многих случаях такое доказательство – это все,

что нужно для того, чтобы убедиться в валидности небольшого «фрагмента исходного кода». Вместе с тем существует ряд обстоятельств, когда требуется обширная формальная валидация исходного кода; в этом случае вместо неформального бумажно-карандашного типа аргументации мы часто используем компьютеризированную верификацию программного обеспечения. Например, управление по контролю за продуктами и лекарствами США требует сертификации программного обеспечения в тех случаях, когда медицинские устройства зависят от программного обеспечения для их эффективной и безопасной работы. Когда требуется формальная верификация, все должно быть изложено четко, на формальном языке и кропотливо доказано строка за строкой. В этом разделе мы приведем пример такой процедуры.

Пусть $\{\alpha\}P\{\beta\}$ означает, что если формула α истинна перед исполнением P, P исполняется и завершается, то формула β будет истинна, то есть α, β являются соответственно предусловием и постусловием программы P. Они обычно задаются в виде формул в некой формальной теории, такой как первопорядковая логика над некоторым языком $\mathcal{L}$. Мы исходим из допущения, что этим языком является арифметика Пеано; см. раздел 9.4.

Используя конечное множество правил для верификации программ, мы хотим показать, что $\{\alpha\}P\{\beta\}$ соблюдается, и заключить, что *программа правильна по отношению к спецификации* α, β. Так как наш пример является небольшим, мы будем использовать ограниченное множество правил для верификации программ, приведенное на рис. 9.12.

Следствие слева и справа

$$\frac{\{\alpha\}P\{\beta\} \quad (\beta \to \gamma)}{\{\alpha\}P\{\gamma\}} \qquad \frac{(\gamma \to \alpha) \quad \{\alpha\}P\{\beta\}}{\{\gamma\}P\{\beta\}}$$

Композиция и присвоение

$$\frac{\{\alpha\}P_1\{\beta\} \quad \{\beta\}P_2\{\gamma\}}{\{\alpha\}P_1P_2\{\gamma\}} \qquad \frac{}{\{\alpha(t)\}x := t\{\alpha(x)\}}$$

If

$$\frac{\{\alpha \wedge \beta\}P_1\{\gamma\} \quad \{\alpha \wedge \neg\beta\}P_2\{\gamma\}}{\{\alpha\}, \textbf{if } \beta \textbf{ then } P_1 \textbf{ else } P_2\{\gamma\}}$$

While

$$\frac{\{\alpha \wedge \beta\}P\{\alpha\}}{\{\alpha\} \textbf{ while } \beta \textbf{ do } P \{\alpha \wedge \neg\beta\}}$$

Рис. 9.12 ❖ Небольшое множество правил для верификации программы

Правило «If» (если) гласит следующее: предположим, что имеет место, что $\{\alpha \wedge \beta\}P_1\{\gamma\}$ и $\{\alpha \wedge \neg\beta\}P_2\{\gamma\}$. Это означает, что P_1 является (частично) правильным по отношению к предусловию $\alpha \wedge \beta$ и постусловию γ, тогда как P_2 является (частично) правильным по отношению к предусловию $\alpha \wedge \neg\beta$ и постусловию γ. Тогда

программа «**if** β **when** P_1 **else** P_2» является (частично) правильной по отношению к предусловию α и постусловию γ, потому что если α соблюдается перед его исполнением, то либо β, либо ¬β должно быть истинным, и поэтому соответственно исполняется либо P_1, либо P_2, в обоих случаях давая нам β.

Правило «While» (до тех пор, пока) гласит следующее: предположим, что имеет место $\{\alpha \wedge \beta\}P\{\alpha\}$. Это означает, что P является (частично) правильным по отношению к предусловию $\alpha \wedge \beta$ и постусловию α. Тогда программа «**while** β **do** P» является (частично) правильной по отношению к предусловию α и постусловию $\alpha \wedge \neg\beta$, потому что если α соблюдается перед его исполнением, то либо β соблюдается, в каковом случае цикл while исполняется еще раз, при этом соблюдается $\alpha \wedge \beta$, и поэтому α по-прежнему соблюдается после исполнения P, либо β является ложным, в каковом случае β является истинным, и цикл завершается с $\alpha \wedge \neg\beta$.

В качестве примера мы верифицируем вычисление $y = A \cdot B$. Обратите внимание, что в алгоритме 9.5, описывающем программу, которая вычисляет $y = A \cdot B$, мы используем символ «=» вместо обычного «←», так как теперь мы доказываем правильность фактической программы, а не ее представление в псевдокоде.

Алгоритм 9.5. `mult(A,B)`

Предусловие: $B \geqslant 0$
$a = A$;
$b = B$;
$y = 0$;
while $b > 0$ **do**
$y = y + a$;
$b = b - 1$;
end while
Постусловие: $y = A \cdot B$

Мы хотим показать:

$$\{B \geqslant 0\}\texttt{mult(A,B)}\{y = AB\}. \tag{9.12}$$

Каждое прохождение по циклу while добавляет a в y, но $a \cdot b$ уменьшается на a, потому что b уменьшается на 1. Пусть инвариант цикла равен: $(y + (a \cdot b) = A \cdot B) \wedge b \geqslant 0$. Для того чтобы сэкономить пространство, запишем tu вместо $t \cdot u$. Пусть $t \geqslant u$ является аббревиатурой $\mathcal{L}_A$-формулы $\exists x(t = u + x)$, и пусть $t \leqslant u$ является аббревиатурой $u \geqslant t$.

1 $\{y + a(b-1) = AB \wedge (b-1) \geq 0\}$`b=b-1;`$\{y + ab = AB \wedge b \geq 0\}$
присвоение
2 $\{(y + a) + a(b-1) = AB \wedge (b-1) \geq 0\}$`y=y+a;`$\{y + a(b-1) = AB \wedge (b-1) \geq 0\}$
присвоение
3 $(y + ab = AB \wedge b - 1 \geq 0) \rightarrow ((y + a) + a(b-1) = AB \wedge b - 1 \geq 0)$
теорема
4 $\{y + ab = AB \wedge b - 1 \geq 0\}$`y=y+a;`$\{y + a(b-1) = AB \wedge b - 1 \geq 0\}$
следствие слева 2 и 3
5 $\{y + ab = AB \wedge b - 1 \geq 0\}$`y=y+a;b=b-1;`$\{y + ab = AB \wedge b \geq 0\}$
композиция на 4 и 1

6 $(y + ab = AB) \wedge b \geq 0 \wedge b > 0 \rightarrow (y + ab = AB) \wedge b - 1 \geq 0$
теорема
7 $\{(y + ab = AB) \wedge b \geq 0 \wedge b > 0\}$`y=y+a; b=b-1;`$\{y + ab = AB \wedge b \geq 0\}$
следствие слева 5 и 6

```
while (b>0)
```

8 $\{(y + ab = AB) \wedge b \geq 0\}$ `y=y+a;` `b=b-1;` $\{y + ab = AB \wedge b \geq 0 \wedge \neg(b > 0)\}$
while на 7
9 $\{(0 + ab = AB) \wedge b \geq 0\}$ `y=0;` $\{(y + ab = AB) \wedge b \geq 0\}$
присвоение

```
y=0;
while (b>0)
  y=y+a;
  b=b-1;
```

10 $\{(0 + ab = AB) \wedge b \geq 0\}$ $\{y + ab = AB \wedge b \geq 0 \wedge \neg(b > 0)\}$
композиция на 9 и 8
11 $\{(0 + aB = AB) \wedge B \geq 0\}$ `b=B;` $\{(0 + ab = AB) \wedge b \geq 0\}$
присвоение

```
b=B;
y=0;
while (b>0) {y + ab = AB ∧ b ≥ 0 ∧ ¬(b > 0)}
  y=y+a;
  b=b-1;
```

12 $\{(0 + aB = AB) \wedge B \geq 0\}$
композиция на 11 и 10
13 $\{(0 + AB = AB) \wedge B \geq 0\}$ `a=A;` $\{(0 + aB = AB) \wedge B \geq 0\}$
присвоение
14 $\{(0 + AB = AB) \wedge B \geq 0\}$ `mult(A,B)` $\{y + ab = AB \wedge b \geq 0 \wedge \neg(b > 0)\}$
композиция на 13 и 12
15 $B \geqslant 0 \rightarrow ((0 + AB = AB) \wedge B \geq 0)$
теорема
16 $(y + ab = AB \wedge b \geq 0 \wedge \neg(b > 0)) \rightarrow y = AB$
теорема
17 $\{B \geqslant 0\}$ `mult(A,B)` $\{y + ab = AB \wedge b \geq 0 \wedge \neg(b > 0)\}$
следствие слева на 15 и 14
18 $\{B \geqslant 0\}$ `mult(A,B)` $\{y = AB\}$
следствие справа на 16 и 17

Задача 9.108. Далее приведен скорее проект, чем упражнение. Дайте формальные доказательства правильности алгоритма деления и алгоритма Евклида (алгоритмы 1.1 и 1.2). Для того чтобы дать полное доказательство, вам нужно будет использовать арифметику Пеано, которая представляет собой формализацию теории чисел – именно то, что необходимо для этих двух алгоритмов. Подробности арифметики Пеано приведены в разделе 9.4.

9.5. Ответы к избранным задачам

Задача 9.1. Ясно, что соблюдается базовый случай: $1 + \sum_{j=0}^{0} 2j = 1 + 1 = 2^{0+1}$ (то есть P(0)). Для индукции предположим, что он соблюдается для некоторого $n \in \mathbb{N}$; то есть $1 + \sum_{j=0}^{n} 2j = 2^{n+1}$. Тогда:

$$1 + \sum_{j=0}^{n+1} 2^j = 2^{n+1} + 1 + \sum_{j=0}^{n} 2^j.$$

Здесь мы применяем индукционную гипотезу:

$$= 2^{n+1} + 2^{n+1} = 2^{n+2}.$$

Мы показали, что P(0) истинно и, более того, что $P(n) \to P(n+1)$. Следовательно, $\forall m P(m)$.

Задача 9.2. Базовый случай: $n = 1$, тогда $1^3 = 1^2$. На индукционном шаге:

$$(1 + 2 + 3 + \cdots + n + (n+1))^2$$
$$= (1 + 2 + 3 + \cdots + n)^2 + 2(1 + 2 + 3 + \cdots + n)(n+1) + (n+1)^2,$$

и по индукционной гипотезе:

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

Задача 9.3. Важно правильно истолковать постановку задачи: когда в ней говорится, что отсутствует одна клетка, это означает, что может отсутствовать любая клетка. Таким образом, базовый случай: при заданной площади 2×2 клеток существует четыре возможных варианта отсутствия клетки; но в каждом случае оставшиеся клетки образуют букву «L». Эти четыре возможности показаны на рис. 9.13.

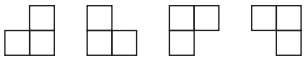


Рис. 9.13 ❖ Четыре разные формы в виде буквы «L»

Предположим, что утверждение соблюдается для n, и рассмотрим площадь размера $2^{n+1} \times 2^{n+1}$ клеток. Разделим ее на четыре квадранта одинакового размера. Независимо от того, какая клетка по нашему выбору отсутствует, она будет находиться в одном из четырех квадрантов; этот квадрант может быть заполнен фигурами «L» (то есть фигурами в форме, приведенной на рис. 9.13) по индукционной гипотезе. Что касается остальных трех квадрантов, поместим в них фигуру «L» таким образом, чтобы она охватывала все три квадранта («L» пересекает центр, оставаясь в этих трех квадрантах). Остальные клетки каждого квадранта теперь могут быть заполнены фигурами «L» по индукционной гипотезе.

Задача 9.4. Поскольку $\forall n(P(n) \to P(n+1)) \to (\forall n \geqslant k)(P(n) \to P(n+1))$, тогда (9.2) $\Rightarrow$ (9.2′). С другой стороны, (9.2′) $\not\Rightarrow$ (9.2).

Задача 9.5. Базовый случай $n = 1$, и он является непосредственным. На индукционном шаге допустим, что равенство соблюдается для экспоненты n, и покажем, что оно соблюдается для экспоненты $n + 1$:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} f_{n+1} + f_n & f_{n+1} \\ f_n + f_{n-1} & f_n \end{pmatrix}.$$

Крайне правая матрица может быть упрощена с использованием определения *чисел Фибоначчи*, как хотелось бы.

Задача 9.7. $m|n$ тогда и только тогда, когда $n = km$, поэтому покажем, что $f_m|f_{km}$ по индукции на k. Если $k = 1$, то доказывать нечего. В противном случае $f_{(k+1)m} = f_{km+m}$. Теперь, используя отдельный индуктивный аргумент, покажем, что для $y \geqslant 1$ $f_{x+y} = f_y f_{x+1} + f_{y-1} f_x$, и закончим доказательство. Для того чтобы показать это последнее формальное суждение, пусть $y = 1$, и отметим, что $f_y f_{x+1} + f_{y-1} f_x = f_1 f_{x+1} + f_0 f_x = f_{x+1}$. Теперь допустим, что $f_{x+y} = f_y f_{x+1} + f_{y-1} f_x$ соблюдается. Рассмотрим:

$$\begin{aligned} f_{x+(y+1)} &= f_{(x+y)+1} = f_{(x+y)} + f_{(x+y)-1} = f_{(x+y)} + f_{x+(y-1)} \\ &= (f_y f_{x+1} + f_{y-1} f_x) + (f_{y-1} f_{x+1} + f_{y-2} f_x) \\ &= f_{x+1}(f_y + f_{y-1}) + f_x(f_{y-1} + f_{y-2}) \\ &= f_{x+1} f_{y+1} + f_x f_y. \end{aligned}$$

Задача 9.8. Обратите внимание, что это является почти основной теоремой арифметики; чего не хватает, так это того факта, что до переупорядочения простых чисел это представление является уникальным. Доказательство этому можно найти в разделе 9.2, теорема 9.18.

Задача 9.9. Пусть наше логическое утверждение P(n) равно: минимальное количество разламываний для разламывания шоколадной плитки, состоящей из n долек, равно $(n - 1)$. Обратите внимание, что это говорит о том, что $(n - 1)$ разламываний достаточно и $(n - 2)$ нет. Базовый случай: не требует разламываний одна-единственная долька. Индукционный шаг: предположим, что у нас $m + 1$ долек. Независимо от того, как мы разламываем плитку на две меньшие части долек a и b, $a + b = m + 1$.

По индукционной гипотезе, часть «a» требует $a - 1$ разламываний, и часть «b» требует $b - 1$ разламываний, поэтому вместе число разламываний равняется

$$(a - 1) + (b - 1) + \boxed{\mathbf{1}} = a + b - 1 = m + 1 - 1 = m,$$

и доказательство завершено. Обратите внимание, что **1** в прямоугольнике происходит от первоначального разламывания, которое делит шоколадную плитку на части «a» и «b».

Таким образом, «нудный» способ разламывания плитки шоколада (сначала на ряды, а затем каждый ряд отдельно на дольки) на самом деле является оптимальным.

Задача 9.10. Пусть принцип индукции равен: $[P(0) \wedge (\forall n)(P(n) \rightarrow P(n + 1))] \rightarrow (\forall m) P(m)$ (где n, m варьируются над натуральными числами), и пусть принцип наименьшего числа гласит: *каждое непустое подмножество натуральных чисел имеет наименьший элемент*. Эти два принципа эквивалентны в том смысле, что один может быть показан из другого. Действительно:

- **принцип наименьшего числа ⇒ принцип индукции:** предположим, что мы имеем $[P(0) \wedge (\forall n)(P(n) \rightarrow P(n + 1))]$, но это вовсе не означает, что имеет место $(\forall m)P(m)$. Тогда множество S из m, для которого P(m) ложно, является непустым. По принципу наименьшего числа мы знаем, что S имеет наименьший элемент. Мы знаем, что этот элемент не равен 0, так как в качестве допущения было принято P(0). Таким образом, этот элемент может быть выражен как $n + 1$ для некоторого натурального числа n. Но так как $n + 1$ является наименьшим таким числом, P(n) должно соблюдаться. Это противо-

речие, поскольку мы исходили из того, что $(\forall n)(\mathrm{P}(n) \to \mathrm{P}(n+1))$, и здесь мы имеем n такое, что $\mathrm{P}(n)$, но не $\mathrm{P}(n+1)$;

- **принцип индукции ⇒ принцип наименьшего числа:** предположим, что S является непустым подмножеством натуральных чисел. Предположим, что оно не имеет наименьшего элемента; пусть $\mathrm{P}(n)$ равно следующему логическому утверждению: «все элементы до и включая n не находятся в S». Мы знаем, что $\mathrm{P}(0)$ должно быть истинным, так как в противном случае 0 будет в S, и тогда он будет наименьшим элементом (по определению 0). Предположим, что $\mathrm{P}(n)$ истинно (поэтому ни один из $\{0, 1, 2, \ldots, n\}$ не находится в S). Предположим, что $\mathrm{P}(n+1)$ ложно: тогда $n+1$ обязательно будет в S (поскольку мы знаем, что ни один из $\{0, 1, 2, \ldots, n\}$ не находится в S), и таким образом $n+1$ будет наименьшим элементом в S. Следовательно, мы показали $[\mathrm{P}(0) \wedge (\forall n)(\mathrm{P}(n) \to \mathrm{P}(n+1))]$. Поэтому по принципу индукции мы можем заключить, что $(\forall m)\mathrm{P}(m)$. Но это означает, что S – пустое. Противоречие. Таким образом, S должно иметь наименьший элемент;
- **принцип индукции ⇒ принцип полной индукции**: для этого направления мы используем принцип наименьшего числа, который мы только что показали как эквивалентный принципу индукции. Предположим, что у нас есть принцип индукции; допустим, что $P(0)$ и $\forall n((\forall i \leqslant n)P(i) \to P(n+1))$. Мы хотим показать, что $\forall n P(n)$, поэтому мы докажем это с помощью принципа индукции: задан базовый случай, $P(0)$. Для того чтобы показать, как $\forall j(P(j) \to P(j+1))$, предположим, что это не соблюдается; тогда существует j такой, что $\mathrm{P}(j)$ и $\neg P(j)$; пусть j равно наименьшему такому j; он существует по принципу наименьшего числа, и $j \neq 0$ тем, что дано. Поэтому $P(0)$, $P(1)$, $P(2)$, …, $P(j)$, но $\neg P(j+1)$. Однако это противоречит $\forall n((\forall i \leqslant n)P(i) \to P(n+1))$, и, следовательно, это невозможно. Отсюда следует, что $\forall j(P(j) \to P(j+1))$, и поэтому по принципу индукции у нас есть $\forall n P(n)$ и, следовательно, принцип полной индукции.

Последнее направление, принцип полной индукции ⇒ принцип индукции, следует непосредственно из того, что принцип полной индукции имеет «более сильный» индукционный шаг.

Задача 9.11. Мы используем пример на рис. 9.1. Предположим, что мы хотим получить дерево из инфиксного (2164735) и префиксного (1234675) кодирований: из префиксного кодирования нам известно, что 1 является корнем, и при этом из инфиксного кодирования нам известно, что левое поддерево имеет инфиксное кодирование 2, и поэтому префиксное кодирование 2, а правое поддерево имеет инфиксное кодирование 64735 и поэтому префиксное кодирование 34675, и мы продолжаем рекурсивно.

Задача 9.13. Рассмотрим следующий инвариант: сумма S чисел, находящихся сейчас во множестве, является нечетной. Теперь докажем, что этот инвариант соблюдается. Базовый случай: $S = 1 + 2 + \cdots + 2n = n(2n+1)$, который является нечетным. Индукционный шаг: предположим, что S нечетно, пусть S равен результату еще одной итерации, поэтому

$$S' = S + |a - b| - a - b = S - 2\min(a, b),$$

и поскольку $2\min(a, b)$ является четным, и S было нечетным по индукционной гипотезе, из этого следует, что S' также должно быть нечетным. В конце, когда останется только одно число, к примеру x, $S = x$, поэтому x является нечетным.

Задача 9.14. Для решения этой задачи необходимо предоставить и алгоритм, и инвариант для него. Алгоритм работает следующим образом: изначально разделить клуб на две группы. Пусть H равно общей сумме соперников, которых каждый член имеет в своей группе. Теперь повторить следующий цикл: до тех пор, пока существует m, у которого есть, по крайней мере, два соперника в его собственной группе, перемещать m в другую группу (где m должен иметь не более одного соперника). Тем самым, когда m переходит из дома в дом, H уменьшается. Здесь инвариантом является «H монотонно убывает». Далее мы знаем, что последовательность натуральных чисел не может уменьшаться вечно, поэтому, когда H достигает своего абсолютного минимума, мы получаем требуемое распределение.

Задача 9.15. Сначала расположите гостей любым способом; пусть H равно числу соседних враждебно настроенных пар. Мы найдем алгоритм, который сокращает H всегда, когда $H > 0$. Предположим, что $H > 0$, и пусть (A, B) равно враждебно настроенной паре, сидящей бок о бок по часовой стрелке A, B. Пройдемся вдоль стола по часовой стрелке, пока мы не найдем еще одну пару (A', B') такую, что A, A' и B, B' являются друзьями. Такая пара должна существовать: существует $2n - 2 - 1 = 2n - 3$ кандидатов на A' (это все люди, сидящие по часовой стрелке после B, у которых есть сосед, сидящий рядом с ними, снова по часовой стрелке, и этот сосед не является ни A, ни B). Так как A имеет не менее n друзей (среди людей кроме самого себя), то из этих $2n - 3$ кандидатов, по крайней мере, $n - 1$ из них являются друзьями A. Если каждый из этих друзей был недругом B, сидящего рядом с ним (опять же, идя по часовой стрелке), тогда B имел бы не менее n недругов, что не представляется возможным, поэтому должно быть A' друзей с A таким образом, чтобы сосед A' (по часовой стрелке) был B' и B' был другом B; см. рис. 9.14.

Обратите внимание, что при $n = 1$ ни у кого нет недругов, и поэтому этот анализ применим при $n \geqslant 2$, в каковом случае $2n - 3 \geqslant 1$.

Теперь ситуация вокруг стола …, A, $\boxed{B, \ldots, A'}$, B', …. Переверните всех в прямоугольнике (то есть отобразите прямоугольник зеркально), чтобы сократить H на 1. Продолжайте повторять эту процедуру до тех пор, пока $H > 0$; в конце концов, $H = 0$ (по принципу наименьшего числа), и в этот момент не будет соседей, которые не нравятся друг другу.

$$A, B, c_1, c_2, \ldots, c_{2n-3}, c_{2n-2}$$

Рис. 9.14 ❖ Список гостей, сидящих за столом по часовой стрелке, начиная с A. Нас интересуют друзья A среди $c_1, c_2, \ldots, c_{2n-3}$, чтобы убедиться, что имеется сосед справа, и этот сосед не является ни A и ни B; конечно, стол огибается вокруг в месте, где сидит гость c_{2n-2}, поэтому следующим соседом гостя c_{2n-2} по часовой стрелке является A. Так как A имеет не более $n - 1$ недругов, A имеет не менее n друзей (не считая самого себя; любовь к себе не считается дружбой). Эти n друзей гостя A находятся среди гостей c, но если мы исключим c_{2n-2}, то из этого следует, что A имеет, по крайней мере, $n - 1$ друзей среди $c_1, c_2, \ldots, c_{2n-3}$. Если сосед c_i по часовой стрелке, $1 \leqslant i \leqslant 2n - 3$, то есть c_{i+i} был в каждом случае недругом B, то, поскольку B уже имеет недруга A, то из этого будет следовать, что B имеет n недругов, что невозможно

Задача 9.16. Мы разбиваем участников на множество E четных лиц и множество O нечетных лиц. Мы наблюдаем, что во время церемонии рукопожатия мно-

жество O не может изменить своего паритета. Действительно, если два нечетных человека пожимают друг другу руки, то $|O|$ уменьшается на 2. Если два четных человека пожимают друг другу руки, то $|O|$ увеличивается на 2, а если четный и нечетный люди пожимают руки, то $|O|$ не меняется. Так как изначально $|O| = 0$, четность множества сохраняется.

Задача 9.19. Если $a_1 \equiv_m a_2$, то существует некоторое $a \in \{0, 1, 2, \ldots, m-1\}$ такое, что $a_1 = \alpha_1 m + a$ и $a_2 = \alpha_2 m + a$, где α_1 и α_2 – это целые числа. Аналогичным образом у нас есть $b_1 = \beta_1 m + b$ и $b_2 = \beta_2 m + b$. Таким образом,

$$\begin{aligned} a_1 \pm b_1 &= (\alpha_1 \pm \beta_1)m + (a \pm b) \\ &\equiv_m a \pm b \\ &\equiv_m (\alpha_2 \pm \beta_2)m + (a \pm b) \\ &= a \pm b_2 \end{aligned}$$

и

$$\begin{aligned} a_1 \cdot b_1 &= (\alpha_1 m + a)\cdot(\beta_1 m + b) \\ &= \alpha_1\beta_1 \cdot m^2 + (\alpha_1 b + \beta_1 a)\cdot m + a \cdot b \\ &\equiv_m a \cdot b \\ &\equiv_m \alpha_2\beta_2 \cdot m^2 + (\alpha_2 b + \beta_2 a)\cdot m + a \cdot b \\ &= a_2 \cdot b_2, \end{aligned}$$

где каждый «$\equiv_m$» является истинным, потому что дополнительные кратные m равны 0; то есть $\forall k \in \mathbb{Z}, k \cdot m \equiv_m 0$.

Задача 9.21. Базовый вариант: пусть n равно простому числу. Очевидно, что $n = n^1$ – это разложение n на простые множители, и каждый элемент $\mathbb{Z}_n - \{0\}$ является взаимно простым с n (то есть для каждого положительного целого числа $i < n$, $\gcd(n, i) = 1$, так как n является простым). Следовательно, $\varphi(n) = |\mathbb{Z}_n| - 1 = n - 1 = n^{1-1}(n-1)$, завершая базовый случай. Рассмотрим любое составное $n = p_1^{k_1} \cdot \cdots \cdot p_i^{k_i}$. Очевидно, что мы можем разбить простой множитель p, чтобы получить n_0 такое, что $n = p \cdot n_0$. Рассмотрим два случая.

Случай 1: $p | n_0$. Пусть $m \in \mathbb{Z}^*_{n_0}$. Очевидно, что $\gcd(m, p) = \gcd(m, n_0) = 1$, так как в противном случае m и n_0 имели бы между собой общий множитель p, и мы знаем $m \in \mathbb{Z}^*_{n_0}$. Допустим от противного, что $\exists i \in \{0, 1, 2, \ldots, p-1\}$ такой, что $\gcd(m + in_0, pn_0) = o > 1$. $o | pn_0$, поэтому $o = p$ или $o | n_0$, но $o < p$, поэтому $o | n_0$. Следовательно, $o | in_0$, и мы уже знаем, что $o \nmid m$, так как $\gcd(m, n_0) = 1$, поэтому $o \nmid (m + in_0)$. Мы нашли наше противоречие, o не может быть делителем $m + in_0$, если он не делит $m + in_0$ равномерно по определению. Таким образом, $\forall i \in \mathbb{Z}_p$, $\gcd(m + in_0, pn_0) = 1$. Более того, m был произвольным элементом $\mathbb{Z}^*_{n_0}$, поэтому это работает для каждого такого $m - \varphi(n) \geqslant p \cdot \varphi(n_0)$. Ясно, что для любого $q \in \mathbb{Z}_{n_0} - \mathbb{Z}^*_{n_0}$, $q + in_0 \notin \mathbb{Z}^*_n$, поэтому $\mathbb{Z}^*_n$ не имеет никаких «дополнительных» элементов; $\varphi(n) = p \cdot \varphi(n_0)$. На этом индукция для данного случая завершается.

Случай 2: $p \nmid n_0$. Этот случай очень похож на предыдущий; единственное различие заключается в том, что в конце мы должны удалить все кратные p, так как эти элементы имеют общий множитель p с n. Существует ровно $\varphi(n_0)$ таких кратных p, так как все остальные кратные имели другой общий множитель с самим n_0 и в связи с этим не были включены. Тем самым $\varphi(n) = p \cdot \varphi(n_0) - \varphi(n_0) = (p-1) \cdot \varphi(n_0)$, завершая индукцию.

Для того чтобы прояснить, почему эти рекуррентные соотношения доказывают индукцию, рассмотрим, что происходит с $\prod_{i=1}^{l} p_i^{k_i-1}(p_i - 1)$, либо когда степень простого числа увеличивается на 1 (случай 1), либо когда включается новое простое число (случай 2).

Задача 9.23. $(a + 1)^p \equiv_p \sum_{j=0}^{p} \binom{p}{j} a^{p-j}1^j \equiv_p (a^p + 1) + \sum_{j=1}^{p-1} \binom{p}{j} a^{p-j}$.

Обратите внимание, что $\binom{p}{j}$ делится на p для $1 \leqslant j \leqslant p - 1$, и поэтому мы имеем, что $\sum_{j=1}^{p-1} \binom{p}{j} a^{p-j} \equiv_p 0$. Таким образом, мы можем доказать наше утверждение по индукции на a. Случай $a = 1$ тривиален, и на индукционном шаге мы используем приведенные выше наблюдения, для того чтобы сделать вывод, что $(a + 1)^p \equiv_p (a^p + 1)$, и мы применяем индукционную гипотезу, получая $a^p \equiv_p a$. После того как мы доказали, что $a^p \equiv_p a$, доказательство завершается, поскольку для a такого, что $\gcd(a, p) = 1$, мы имеем обратное a^{-1}, поэтому мы умножаем обе стороны на него и получаем $a^{p-1} \equiv_p 1$.

Задача 9.24. Сначала мы рассматриваем $(\mathbb{Z}_n, +)$. Ясно, что замыкание удовлетворено, так как добавление в $\mathbb{Z}_n$ выполняется по модулю n, поэтому результат сложения должен быть в $\mathbb{Z}_n$. Нейтральный элемент (тождественность) равен 0; $0 + i \equiv_n i$ для любого $i \in \mathbb{Z}_n$. Мы также можем легко найти обратное: $i^{-1} = n - i$, потому что $i + (n - i) = n \equiv_n 0$. Наконец, сложение по модулю n является ассоциативным для любого n, поэтому все три аксиомы удовлетворены.

Далее рассмотрим $(\mathbb{Z}_n^*, \cdot)$. При наличии $a, b \in \mathbb{Z}_n^*$, $\gcd(a \cdot b, n) = 1$ с регулярным умножением, поэтому $\gcd(a \cdot b, n) = 1$ тоже с модулярным умножением – разница только в удалении «лишних» кратных n. Таким образом, мы имеем замыкание. $\gcd(n, 1) = 1$ является независимым от значения n, поэтому $1 \in \mathbb{Z}_n^*$. Очевидно, что 1 удовлетворяет требованиям нейтрального элемента при умножении. С учетом любого элемента $a \in \mathbb{Z}_n^*$ мы знаем $\gcd(a, n) = 1$, поэтому мы можем найти целые числа x, y такие, что $ax + ny = 1$. Более того, $ny \equiv_n 0$, поэтому $ax \equiv_n 1$. Если $x \notin \mathbb{Z}_n$, то существует $x' \in \mathbb{Z}_n$ такой, что $x' \equiv_n x$. То же самое $ax' \equiv_n 1$; из ax мы удалили только кратное an, поэтому эффект $(\mathrm{mod}\, n)$ равен 0. Так как $ax' \equiv_n 1$, x' не должен иметь никаких общих множителей с n, поэтому $x' \in \mathbb{Z}_n^*$. Таким образом, мы имеем обратное. Опять же, ассоциативность является тривиальной, так как она гарантируется выбранной операцией, умножением по модулю n.

Задача 9.27. Пусть $H \leqslant G$, и допустим, что $h \in H$. Поскольку H является группой, мы знаем, что $h^{-1} \in H$ тоже является группой. Мы также знаем, что $e \in H$, где e – это нейтральный элемент G (и, конечно же, H), опять же просто потому, что H является группой. Поскольку H замкнута, мы знаем, что для всех $a \in H$, также $h^{-1}a \in H$, в связи с чем $hh^{-1}a = a \in hH$. Таким образом, $H \subseteq hH$. Далее рассмотрим любой $a' \in hH$; ясно, что $a' = ha$ для некоторого $a \in H$. Поскольку $h \in H$ тоже и H является замкнутой, a' должно быть в x. Следовательно $hH \subseteq H$, завершая доказательство, что $hH = H$.

Пусть $g \in G$, и рассмотрим gH; очевидно, что $|gH| \leqslant |H|$, так как каждый элемент gH требует уникального $h \in H$. Допустим, что $|gH| < |H|$. Тогда существует два уникальных элемента H, h_1 и h_2, таких, что $gh_1 = gh_2$. Но G является группой, поэтому g имеет обратное, g^{-1}. Значит, $g^{-1}gh_1 = g^{-1}gh_2$, или то же самое $h_1 = h_2$ – противоречие. Тем самым $|gH| = |H|$.

Допустим, что $h' \in (ab)H$. Тогда $\exists h \in H$ такая, что $(ab)h = h'$. Группы являются ассоциативными, поэтому $h' = a(bh)$, и в связи с этим $h' \in H$. Доказывание того, что любой элемент $a(bH)$ также находится в $(ab)H$, является почти идентичным.

Задача 9.28. Для обозначения результата операции заданной группы мы будем использовать термин «произведение». Обратите внимание, что $\langle g_1, g_2, \ldots, g_k\rangle$ – это просто коллекция произведений произвольной перестановки элементов $G' = \{g_1, \ldots, g_k, g_1^{-1}, \ldots, g_k^{-1}\}$ с возвратом. Ясно, что если мы умножаем любой $g \in G'$ на самого себя либо другой элемент G, результат находится в сгенерированной подгруппе (что побуждает включать тождество, при условии что инверсии включены в G'). Более того, учитывая любые два созданных элемента $x_1x_2 \cdots x_{p_1}$ и $y_1y_2 \cdots y_{p_2}$, произведение $x_1 \cdots x_{p_1}y_1 \cdots y_{p_2}$ отвечает требованиям, которые должны быть включены в $\langle g_1, g_2, \ldots, g_k\rangle$. Таким образом, сгенерированная подгруппа замыкается. Она явно также включает в себя инверсии, так как инверсия произведения $x_1 \cdots x_k$ является просто произведением $x_k^{-1} \cdots x_1^{-1}$. Ассоциативность обеспечивается охватывающей группой G. Таким образом, сгенерированная подгруппа действительно является группой. Что касается $|g|$, обратите внимание, что любой элемент может быть записан как произведение элементов g и g^{-1}. Другими словами, каждый элемент $|\langle g\rangle|$ может быть записан в виде g^n для некоторого целого числа n. Но $g^{\mathrm{ord}(G)} = 1$, значит, $g^n = g^{(n \bmod \mathrm{ord}(G))}$. Так как существует только $\mathrm{ord}(G)$ несовпадающих элементов в $\mathbb{Z}_{\mathrm{ord}(G)}$, то существует также только $\mathrm{ord}(G)$ несовпадающих элементов g.

Задача 9.31. Сконструируем r поэтапно так, чтобы на этапе i оно удовлетворяло первым i конгруэнциям, то есть на этапе i мы имеем, что $r \equiv r_j \pmod{m_j}$ для $j \in \{0, 1, \ldots, i\}$. Этап 1 является легким: просто установим $r \leftarrow r_0$. Предположим, что первые i этапов завершены; пусть $r \leftarrow r + (\prod_{j=0}^{i} m_j)x$, где x удовлетворяет

$$x \equiv (\prod_{j=0}^{i} m_j)^{-1}(r_{i+1} - r) \pmod{m_{i+1}}.$$

Мы знаем, что обратное от $(\prod_{j=0}^{i} m_j)$ существует (в $\mathbb{Z}_{m_{i+1}}$), поскольку $\gcd(m_{i+1}, (\prod_{j=0}^{i} m_i)) = 1$, и, более того, это обратное может быть эффективно получено с помощью расширенного алгоритма Евклида.

Задача 9.33. Мы докажем это, если $m_0, m_1, \ldots, m_n$ являются попарно взаимно простыми числами, тогда

$$\mathbb{Z}_{m_0 \cdot m_1 \cdot \ldots \cdot m_n} \cong \mathbb{Z}_{m_0} \times \mathbb{Z}_{m_1} \times \cdots \times \mathbb{Z}_{m_n}$$

через индукцию над n. Пусть $M = m_0 \cdot m_1 \cdot \ldots \cdot m_n$. Теорема 9.30 обеспечивает удобную биекцию из $\mathbb{Z}_M$ в $\mathbb{Z}_{m_0} \times \cdots \times \mathbb{Z}_{m_n}$:

$$f(r) = (r \bmod m_0, r \bmod m_1, \ldots, r \bmod m_n)$$

для всех $r \in \mathbb{Z}_M$. Обратите внимание, что операции в этих двух группах не определены, потому что они вытекают из контекста; для $\mathbb{Z}_M$ операцией является сложение по модулю M. Мы будем обозначать ее как «$+_M$» (на самом деле для любого натурального числа n мы будем обозначать сложение по модулю n как «$+n$», когда это удобно). Для $\mathbb{Z}_{m_0} \times \cdots \times \mathbb{Z}_{m_n}$ она является поэлементным модулярным сложением – то есть при наличии $x, y \in \mathbb{Z}_{m_0} \times \cdots \times \mathbb{Z}_{m_n}$ «$x * y$» будет означать $(x_0 +_{m_0} y_0, \ldots, x_n +_{m_n} y_n)$.

$$\begin{aligned} f(r +_M r') &= (r_0 +_M r_0 \bmod m_0, \ldots, r_n +_M r'_n \bmod m_n) \\ &= (r + r' \bmod m_0, \ldots, r + r' \bmod m_n) \\ &= f(r) * f(r'), \end{aligned}$$

где мы можем использовать нормальное сложение вместо модулярного сложения, потому что для всех i, $m_i | M$. Мы уже знали, что f является биекцией; теперь мы знаем, что она является изоморфизмом, поэтому две группы являются изоморфными.

Задача 9.35. (1) ⇒ (2) Предположим, что R является транзитивным, и пусть $(x, y) \in R^2$. Тогда по определению (9.6) мы знаем, что существует такое z, что xRz и zRy. По транзитивности мы имеем, что $(x, y) \in R$. (2) ⇒ (3) Предположим, что $R^2 \subseteq R$. Мы показываем по индукции на n, что $R^n \subseteq R$. Базовый случай, $n = 1$, тривиален. На индукционном шаге предположим, что $(x, y) \in R^{n+i} = R^n \circ R$, поэтому по определению (9.6) существует z такой, что $xRnz$ и zRy. По индукционному допущению это означает, что xRz и zRy, поэтому $(x, y) \in R^2$, и поскольку $R^2 \subseteq R$, то из этого следует, что $(x, y) \in R$, и доказательство завершено. (3) ⇒ (1) Предположим, что для всех n $R^n \subseteq R$. Если xRy и yRz, то $xRz \in R^2$, и поэтому $xRz \in R$, и, значит, R является транзитивным.

Задача 9.37. С учетом $R \subseteq X \times X$ пусть $S = R \cup id_X$. Очевидно, что S является рефлексивным, так как только id_X содержит каждую пару, необходимую для обеспечения рефлексивности. Рассмотрим любое S', для которого существует пара x, y такая, что xSy и $\neg xS'y$. Если xRy, то $R \not\subseteq S'$. В противном случае $(x, y) \in id_X$, поэтому $x = y$; существует элемент x такой, что $\neg xS'x$, поэтому S' не является рефлексивным. В обоих случаях S' не является рефлексивным замыканием R. Поэтому $R \subseteq S$, S является рефлексивным, и любое множество, удовлетворяющее этим двум условиям, содержит каждый элемент S. Следовательно, S является рефлексивным замыканием R.

Задача 9.39. Пусть $S = R \cup R^{-1}$. Очевидно, что $R \subseteq S$, и S является явно симметричным. Рассмотрим S' такое, что $S \not\subseteq S'$. Существует пара $(x, y) \in S$ такая, что $(x, y) \notin S'$. Если $(x, y) \in R$, то $R \not\subseteq S'$. В противном случае $(y, x) \in R$; если $(y, x) \in S'$, то S' не является симметричным, но если $(y, x) \notin S'$, то $R \not\subseteq S'$. В любом случае, S' либо не замкнуто, либо не содержит R. S, с другой стороны, содержит R, является симметричным и подмножеством любого множества, которое удовлетворяет этим условиям. Поэтому оно является симметричным замыканием R.

Задача 9.41. Причина в том, что в первой строке мы выбрали *конкретный* y: $xR^+y \wedge yR^+z \Leftrightarrow \exists m, n \geqslant 1, xR^m y \wedge yR^n z$. С другой стороны, из формального суждения $\exists m, n \geqslant 1, x(R^m \circ R^n)z$ мы можем только заключить, что существует y такой, что $\exists m, n \geqslant 1, xR^m y' \wedge y'R^n z$, и не обязательно имеет место, что $y = y'$.

Задача 9.45. R является рефлексивным, поскольку $F(x) = F(x)$; R является симметричным, поскольку из $F(x) = F(y)$ следует $F(y) = F(x)$ (равенство является симметричным отношением); R является транзитивным, потому что $F(x) = F(y)$ и $F(y) = F(z)$ означает, что $F(x) = F(z)$ (опять же по транзитивности равенства).

Задача 9.51. Из леммы 9.49 мы знаем, что $\forall a \in X, [a]_{R_1} \subseteq [a]_{R_2}$. Следовательно, отображение $f : X/R_1 \to X/R_2$, заданное $f([a]_{R_1}) = [a]_{R_2}$, является сюръективным, и, следовательно, $|X/R_1| \geqslant |X/R_2|$.

Задача 9.54. Мы показываем направление слева направо. Ясно, что $\approx$ является рефлексивным, поскольку оно содержит id_X. Теперь предположим, что $a \approx b$; тогда $a \sim b$ или $a = b$. Если $a = b$, то $b = a$ (так как равенство, очевидно, является симметричным отношением), и поэтому $b \approx a$. Если $a \sim b$, то по определению несопоставимости $\neg(a \preccurlyeq b) \wedge \neg(b \preccurlyeq a)$, что логически эквивалентно $\neg(b \preccurlyeq a) \wedge \neg(a \preccurlyeq b)$, а следовательно, $b \sim a$, и поэтому в данном случае также $b \approx a$. Наконец, мы хотим

доказать транзитивность: предположим, что $a \approx b \wedge b \approx c$; если $a = b$ и $b = c$, то $a = c$, и мы имеем $a \approx c$. Схожим образом, если $a = b$ и $b \sim c$, то $a \sim c$, и поэтому $a \approx c$, и если $a \sim b$ и $b = c$, тогда и $a \sim c$, а также $a \approx c$. Единственный оставшийся случай – это $a \sim b$ и $b \sim c$, и именно здесь мы используем тот факт, что $\preccurlyeq$ является стратифицированным порядком, так как из этого вытекает, что $a \sim c \vee a = c$, что дает нам $a \approx c$.

Задача 9.56. Мы показываем направление слева направо. Естественный способ продвижения вперед здесь состоит в том, чтобы приравнять T множеству, состоящему из разных классов эквивалентности X при $\sim$. То есть $T = \{[a]\sim : a \in X\}$. Тогда T является полноупорядоченным при $\preccurlyeq_T$, определенном следующим образом: для $X, X' \in T$ таких, что $X \neq X'$ и $X = [x]$ и $X' = [x']$, мы имеем, что $X \preccurlyeq_T X'$ тогда и только тогда, когда $x \preccurlyeq_T x'$. Отметим также, что с учетом двух несовпадающих X, X' в T и любой пары представителей x, x' всегда имеет место, что $x \preccurlyeq_T x'$ или $x' \preccurlyeq_T x$, поскольку если бы ни тот и ни другой не имел место, то мы бы имели $x \sim x'$, и, следовательно, $[x] = [x']$ и $X = X'$. Тогда функция $f : X \to T$, заданная как $f(x) = [x]$, удовлетворяет требованиям.

Задача 9.57. Пусть $X = \{a, b, c, d, e\}$. Рассмотрим частичный порядок, заданный упорядоченными парами $\{(a, c), (a, d), (a, e), (b, c), (b, d), (b, e), (c, d), (c, e)\}$ (где рефлексивные пары (то есть $(a, a), (b, b)\ldots$) были опущены. Ясно, что a является минимальным, так как нет элемента $x \neq a$ такого, что $x \preccurlyeq a$. Схожим образом b является минимальным, d, e являются максимальными. Вместе с тем не существует наименьшего элемента или наибольшего элемента; наши минимальные элементы a, b несравнимы, как максимальные d, e. Обратите внимание, что в случае конечного линейного частичного порядка это было бы невозможно, потому что каждый элемент был бы сравним. X также не имеет супремума или инфимума, опять же потому, что никакой минимальный или максимальный элемент не может быть сравнен с другими. Существуют также простые примеры линейных частичных порядков без инфимума или супремума. Рассмотрим, например, частичный порядок $(\mathbb{R}^+, \preccurlyeq)$, где $\mathbb{R}^+$ – это положительные действительные числа и $x \preccurlyeq y$ тогда и только тогда, когда $x, y \in \mathbb{R}^+ \wedge x \leqslant y$. Очевидно, что этот частичный порядок не имеет супремума – всегда имеется более крупное действительное число. Менее очевидно, что у него нет инфимума! Существует интуитивный кандидат на инфимум: 0. Однако $\preccurlyeq$ определено только для пар элементов в $\mathbb{R}^+$, поэтому 0 несравним ни с чем. Если вместо этого мы используем частичный порядок $(\mathbb{R}, \leqslant)$, то $\mathbb{R}^+ \subset \mathbb{R}$ имеет инфимум: 0.

Пусть $A \subset X$ равно $\{b, c, \mathrm{d}\}$. Порция нашего частичного порядка на X, которая применима к A: $\{(b, c), (b, \mathrm{d}), (c, \mathrm{d})\}$. В отличие от X, A имеет четкий инфимум, супремум, наибольший элемент и наименьший элемент, хотя его охватывающий X не является линейным.

Задача 9.58. Пусть X равно множеству, и рассмотрим частичный порядок $(\mathcal{P}(X), \subseteq)$. При наличии $A, B \in \mathcal{P}(X)$ мы стремимся доказать, что $A \sqcup B = A \cup B$. Очевидно, что $A, B \subseteq A \cup B$. Более того, в любом собственном подмножестве $A \cup B$ наверняка отсутствует элемент A или B, поэтому для всех $C \in \mathcal{P}(X)$, $A, B \subseteq \mathrm{C} \Rightarrow A \cup B \subseteq C$. Таким образом, $A \cup B = \inf(\{A, B\})$. Доказательство, что $A \sqcap B = A \cap B$ подчиняется примерно тому же процессу, но при этом подмножества и надмножества реверсированы.

Задача 9.60. Мы доказываем следующую часть: $a \preccurlyeq b \Leftrightarrow a \sqcap b = a$. Предположим, что $a \preccurlyeq b$. Так как $(X, \preccurlyeq)$ представляет собой решетку, оно является частичным

порядком, и поэтому $a \preccurlyeq a$ (рефлексивность), и, значит, a является нижней границей множества $\{a, b\}$. Поскольку $(X, \preccurlyeq)$ представляет собой решетку, существует $\inf\{a, b\}$, и, следовательно, $a \preccurlyeq \inf\{a, b\}$. С другой стороны, $\inf\{a, b\} \preccurlyeq a$, и поэтому по антисимметрии частичного порядка мы имеем $a = \inf\{a, b\} = a \sqcap b$. В другом направлении $a \sqcap b = a$ означает, что $\inf\{a, b\} = a$, и поэтому $a \preccurlyeq \inf\{a, b\}$, и, значит, $a \preccurlyeq b$.

Задача 9.62. (1) непосредственно вытекает из наблюдения, что $\{a, b\}$ и $\{b, a\}$ являются одним и тем же множеством. (2) следует из наблюдения, что $\inf\{a, \inf\{b, c\}\} = \inf\{a, b, c\} = \inf\{\inf\{a, b\}, c\}$, и то же самое для супремума. (3) следует непосредственно из наблюдения, что $\{a, a\} = \{a\}$ (мы имеем дело со множествами, а не с «мультимножествами»). В случае (4) закон поглощения, показываем, что $a = a \sqcup (a \sqcap b)$. Прежде всего обратите внимание, что $a \preccurlyeq \sup\{a, *\}$ (где «$*$» обозначает что угодно, в частности $a \sqcap b$). С другой стороны, $a \sqcap b \preccurlyeq a$ по определению, $a \preccurlyeq a$ по рефлексивности, и поэтому a является верхней границей для множества $\{a, a \sqcap b\}$. Следовательно, $\sup\{a, a \sqcap b\} \preccurlyeq a$, и тем самым по антисимметрии $a = \sup\{a, a \sqcap b\}$, то есть $a = a \sqcup (a \sqcap b)$. Другой закон поглощения может быть доказан аналогичным образом.

Задача 9.64. Для того чтобы показать, что $(\mathcal{P}(X), \subseteq)$ является полным, достаточно доказать другие свойства, перечисленные в теореме 9.63, так как формула для супремума и инфимума сильнее их существования. Сначала мы докажем, что $\forall \mathcal{A} \subseteq \mathcal{P}(X)$, $\sup(\mathcal{A}) = \bigcup_{A \in \mathcal{A}} A$. Ясно, что оно удовлетворяет требованию быть верхней границей. Более того, в любом собственном подмножестве $\bigcup_{A \in \mathcal{A}} A$ отсутствует, по крайней мере, один из элементов в $A \in \mathcal{A}$, поэтому он не является верхней границей $\mathcal{A}$. Таким образом, $\bigcup_{A \in \mathcal{A}} A$ является супремумом $\mathcal{A}$. Обратите внимание, что это следует непосредственно из результатов задачи 9.58. Доказательство того, что $\inf(\mathcal{A}) = \bigcap_{A \in \mathcal{A}} A$, очень похоже (и тоже следует непосредственно из задачи 9.58). Остальные факты, $\bot = \emptyset$ и $\top = X$, должны быть очень интуитивными; они также являются непосредственными выводами, которые могут быть сделаны из формул супремума и инфимума этой задачи.

Задача 9.70. Например, наименее неподвижная точка f задана формулой $f^4(\emptyset) = f^3(\{a, b\}) = f^2(\{a, b\}) = f(\{a, b\}) = \{a, b\}$.

Задача 9.72. Обратите внимание, что с $\sup\{a, b\} = \top$, и поэтому $f(\sup\{a, b\}) = f(\top) = \top$. С другой стороны, $f(\{a, b\}) = \{\bot\}$, так как $f(a) = f(b) = \bot$. Следовательно, $\sup(f(\{a, b\}) = \sup(\{\bot\}) = \bot$. См. рис. 9.15 для функции g, которая является монотонной, но не является ни восходяще-, ни нисходяще-непрерывной.

Задача 9.74. Пусть $S \subseteq \mathbb{Z} \times \mathbb{Z}$ равно множеству, состоящему строго из этих пар целых чисел (x, y) таких, что $x \geqslant y$ и $x - y$ является четным. Мы намерены доказать, что S является областью определения F. Прежде всего если $x < y$, то $x \neq y$, и поэтому мы продолжаем вычислением $F(x, F(x - 1, y + 1))$, и теперь мы должны вычислить $F(x - 1, y + 1)$; если $x < y$, то однозначно $x - 1 < y + 1$; это условие сохраняется, и поэтому мы завершаем вычислением $F(x - i, y + i)$ для всех i, и, значит, эта рекурсия никогда «не достигнет дна». Предположим, что $x - y$ является нечетным. Тогда $x \neq y$ (так как 0 – четно!), поэтому снова мы переходим к $F(x, F(x - 1, y + 1))$; если $x - y$ является нечетным, то таковым является и $(x - 1) - (y + 1) = x - y - 2$. Опять же, мы завершаем тем, что нам приходится вычислить $F(x - i, y + i)$ для всех i, и поэтому рекурсия никогда не заканчивается. Ясно, что все пары в S^c не находятся в области определения F.

Предположим теперь, что $(x, y) \in S$. Тогда $x \geqslant y$ и $x - y$ является четным; тем самым $x - y = 2i$ для некоторого $i \geqslant 0$. Мы показываем, по индукции на i, что алгоритм

завершается на таких (x, y) и выводит $x + 1$. Базовый случай: $i = 0$, поэтому $x = y$, и, значит, алгоритм возвращает $y + 1$, то есть $x + 1$. Предположим теперь, что $x - y = 2(i + 1)$. Тогда $x \neq y$, и поэтому мы вычисляем $F(x, F(x - 1, y + 1))$.

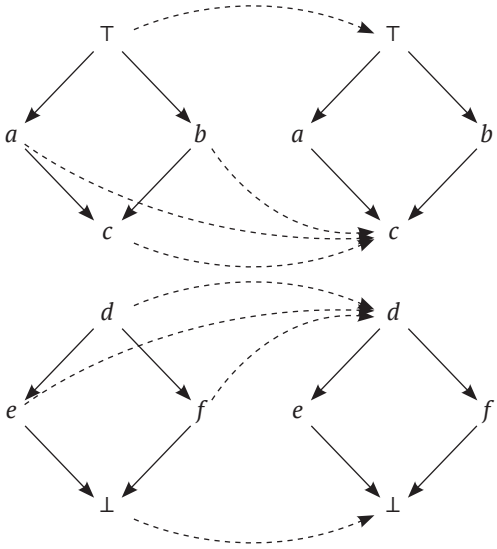


Рис. 9.15 ❖ Пример упорядочения над $X = \{a, b, c, d, e, f, \bot, \top,\}$ с функцией $g : X \to X$, обозначенного пунктирными линиями. В то время как g является монотонной, она не является ни восходящее-, ни нисходяще-непрерывной

Но

$$(x - 1) - (y + 1) = x - y - 2 = 2(i + 1) - 2 = 2i$$

для $i \geqslant 0$, и поэтому по индукции $F(x - 1, y + 1)$ завершается и выводит $(x - 1) + 1 = x$. Поэтому теперь мы должны вычислить $F(x, x)$, то есть просто $x + 1$, и доказательство завершено.

Задача 9.75. Мы показываем, что f_1 является неподвижной точкой алгоритма 9.1. Напомним, что в задаче 9.74 мы показали, что область определения функции F, вычисляемой алгоритмом 9.1, равна $S = \{(x, y) : x - y = 2i, i \geqslant 0\}\}$. Теперь мы показываем, что если мы заменяем F в алгоритме 9.1 на f_1, то новый алгоритм, который является алгоритмом 9.6, по-прежнему вычисляет F, хотя и не рекурсивно (поскольку f_1 определяется алгоритмом 9.2, который не является рекурсивным).

Алгоритм 9.6. Алгоритм 9.1, в котором F заменена на f_1

```
1: if x = y then
2:         return y + 1
3: else
4:         f1(x, f1(x − 1, y + 1))
5: end if
```

Мы действуем следующим образом: если $(x, y) \in S$, то $x - y = 2i$ при $i \geqslant 0$. Из задачи 9.74 мы знаем, что на такой (x, y) функция $F(x, y) = x + 1$. Теперь рассмотрим вывод алгоритма 9.6 на такой паре (x, y). Если $i = 0$, то он возвращает $y + 1 = x + 1$, поэтому доказательство завершено. Если $i > 0$, то он вычисляет

$$f_1(x, f_1(x - 1, y + 1)) = f_1(x, x) = x + 1,$$

и доказательство завершено. Для того чтобы понять, почему $f_1(x - 1, y + 1) = x$, обратите внимание, что существует два случая: во-первых, если $x - 1 = y + 1$, то алгоритм f_1 (алгоритм 9.2) возвращает $(y + 1) + 1 = (x - 1) + 1 = x$. Во-вторых, если $x - 1 > y + 1$ (и это единственная другая возможность), то алгоритм 9.2 тоже возвращает $(x - 1) + 1 = x$.

Задача 9.76. Сначала покажем, что $f_3 \sqsubseteq f_1$. Допустим, $(x, y) \in S_3$. Тогда $x \geqslant y$, и $(x - y)$ является четным. Ясно, что $f_3(x, y) = x + 1$. Если $x \neq y$, то $f_1(x, y) = (x + 1)$; в противном случае $f_1(x, y) = (y + 1) = (x + 1)$. В обоих случаях $f_1(x, y)$ определена и, более того, равна $f_3(x, y)$. Следовательно $f_3 \sqsubseteq f_1$. Далее рассмотрим $f_2(x, y)$. $x \geqslant y$, поэтому f_2 возвращает $(x + 1) = f_3(x, y)$. Тем самым $f_3 \sqsubseteq f_2$.

Задача 9.77. Пусть грамматика G_{prop} имеет алфавит $\{p, 1, \wedge, \vee, \neg, (,)\}$ и множество правил, заданных

$$S \to pX|\neg S|(S \wedge S)|(S \vee S);$$
$$X \to 1|X1.$$

Переменными являются: $\{p1, p11, p111, p1111, \ldots\}$, то есть они кодированы в унарной форме записи.

Задача 9.79. По индукционной гипотезе $w(\alpha) = w(\beta) = 1$, поэтому $w(\neg\alpha) = 0 + (-1) = -1$, и поскольку левая и правая круглые скобки уравновешивают друг друга, в том смысле что $w((t)) = w(() + w(t) + w()) = 1 + w(t) + (-1) = w(t)$, результат быстро следует для $(\alpha \wedge \beta)$ и $(\alpha \vee \beta)$. Для того чтобы показать, что любой надлежащий начальный сегмент $(\alpha \circ \beta)$ (где $\circ \in \{\wedge, \vee\}$) имеет вес $\geqslant 0$, мы записываем его следующим образом:

$$(\alpha \circ \beta) \overset{\text{syn}}{=} (\alpha_1\alpha_2 \ldots \alpha_m \circ \beta_1\beta_2 \ldots \beta_n),$$

где α_i и β_j – это соответственно символы α и β. Несколько случаев естественным образом проявляется: если начальный сегмент состоит только из открывающей скобки (, то его вес равен 1. Если начальный сегмент заканчивается элементами α_i, но не кончается в α_m, тогда по индукции он имеет вес $\geqslant 1$. Если он заканчивается ровно в α_m, то по индукции он имеет вес 0. Если он заканчивается в $\circ$, то он имеет вес 1. Схожим образом мы имеем дело с начальным сегментом, заканчивающимся в середине элементов β_j, в β_n и в последней закрывающей скобке).

Задача 9.81. Предположим, $\alpha \overset{\text{syn}}{\neq} \alpha'$. Тогда $\alpha c\beta \overset{\text{syn}}{=} \alpha' c'\beta'$, α и α' оба являются начальными сегментами одной и той же символьной цепочки (строки). В связи с этим один должен быть начальным сегментом другого; мы исходим из допущения без потери общности, что α является первыми n элементами α' и что α' содержит более n элементов. Очевидно, что α представляет собой надлежащий начальный сегмент, так как α' – это валидная формула. Лемма 9.78 допускает, что вес α является неотрицательным, но α является формулой, поэтому ее вес равен –1. Допущение, что $\alpha \overset{\text{syn}}{\neq} \alpha'$ приводит к противоречию, значит, $\alpha \overset{\text{syn}}{=} \alpha'$. В связи с этим c и c' имеют об-

щий индекс в идентичных строках; они являются одинаковой бинарной связкой. Более того, β и β′ должны тогда начинаться в одинаковом индексе $\alpha c \beta \overset{\text{syn}}{=} \alpha' c' \beta'$,\ и продолжаться до конца, поэтому $\beta \overset{\text{syn}}{=} \beta'$.

Задача 9.82. Предположим, что мы имеем $\Phi \models \alpha$ и $\Phi \cup \{\alpha\} \models \beta$. И предположим, что τ является закреплением истинностного значения, которое удовлетворяет Φ. Тогда по первому допущению оно должно удовлетворять α, и поэтому τ удовлетворяет $\Phi \cup \{\alpha\}$, и, следовательно, по второму допущению оно должно удовлетворять β.

Задача 9.83. По структурной индукции на α. Понятно, если α – это просто переменная p, то α′ равно $\neg p$, и $\neg\alpha \Leftrightarrow \alpha'$. Индукционный шаг следует сразу из законов де Моргана.

Задача 9.84. Пусть переменные α равны $\alpha(\bar{x}, \bar{y})$ и переменные β равны $\beta(\bar{y}, \bar{z})$. Форма записи $\bar{x}$ обозначает множество булевых переменных; используя это обозначение, множество $S = \mathrm{Var}(\alpha) \cap \mathrm{Var}(\beta) = \{\bar{y}\}$. Определим булеву функцию f следующим образом:

$$f(\bar{y}) = \begin{cases} \text{если } \exists x \text{ такое, что } \alpha(\bar{x}, \bar{y}) = 1 \\ \text{в противном случае} \end{cases}.$$

Здесь мы немного злоупотребляем формой записи, смешивая булевы функции и булевы формулы; $\bar{y}$ работает «сверхурочно»: он является и аргументом f, и закреплением истинностного значения переменной α. Но смысл ясен. Пусть $C_f(\bar{y})$ равно булевой формуле, связанной с f; она может быть получена, например, конъюнктивной нормальной формой. C_f – это наша формула: предположим, что $\tau \models \alpha$; тогда τ явно удовлетворяет C_f (по его определению). Если $\tau \models C$, то должно существовать $\bar{x}$ такое, что $\alpha(\bar{x}, \tau)$ является истинным, и, следовательно, $\beta(\tau, \bar{z})$ является истинным по исходному допущению.

Обратите внимание, что мы могли бы определить f двояко с помощью β; каким образом?

Задача 9.85. Мы предлагаем доказательство того, что $\neg(p \vee q) \to \neg p \wedge \neg q$. Обоснование каждого шага приводится справа. Правила «ослабления» и «обмена» обозначаются соответственно символами «w» и «e». Схожим образом «левый» и «правый» обозначаются символами «l» и «r».

$$\dfrac{\dfrac{\dfrac{\dfrac{p \to p}{p \to p, q}\,\mathrm{w}}{p \to p \vee q}\,\vee\mathrm{r}}{\neg(p \vee q) \to \neg p}\,\neg\,\mathrm{l,r} \qquad \dfrac{\dfrac{\dfrac{\dfrac{q \to q}{q \to p, q}\,\mathrm{w,e}}{q \to p \vee q}\,\vee\mathrm{r}}{\neg(p \vee q) \to \neg q}\,\neg\,\mathrm{l,r}}{\neg(p \vee q) \to \neg q \wedge \neg q}\,\wedge\mathrm{r}$$

Задача 9.86. Ниже приводится доказательство в пропозициональном исчислении PK левого сокращения; в связи с этим мы исходим из того, что Γ, α, $\alpha \to \Delta$ является истинным.

$$\dfrac{\dfrac{\alpha \to \alpha}{\Gamma, \alpha \to \Delta, \alpha} \qquad \dfrac{\Gamma, \alpha, \alpha \to \Delta}{\alpha, \Gamma, \alpha \to \Delta}}{\Gamma, \alpha \to \Delta}$$

Задача 9.87. Правило правого введения для ↔:

$$\frac{\alpha, \Gamma \to \Delta, \beta \qquad \beta, \Gamma \to \Delta, \alpha}{\Gamma \to \Delta, (\alpha \leftrightarrow \beta)}$$

Задача 9.88. Напомним, что каждая секвенция записывается в форме «*антецедент → сукцедент*», где антецедент – это конъюнкция и сукцедент – это дизъюнкция. Правила *обмена* являются прямым результатом коммутативности и ассоциативности операторов «и» и «или». Схожим образом правила *ослабления* являются результатом свойств этих операторов. Учитывание дополнительной формулы в антецеденте может приводить к получению значения *ложь*, когда в других случаях оно было бы *истинным*, но это не может сделать секвенцию *ложной*, когда в других случаях она была *истинной*. Схожим образом включение новой формулы в сукцедент может привести к тому, что результирующая дизъюнкция вновь будет возвращать истину, но это опять же не будет приводить к тому, что в других случаях истинная секвенция будет ложной.

Допустим, что $\Gamma \to \Delta, \alpha$ и $\alpha, \Gamma \to \Delta$. То есть $\Gamma \to (\Delta \vee \alpha)$ и $(\alpha \wedge \Gamma) \to \Delta$. Допустим, что Γ истинно, и допустим, ради противоречия, что Δ ложно. Тогда истина → (ложь ∨ α), а значит, α истинно. Поэтому (истина ∧ истина) → Δ, следовательно, Δ должно быть истинным. Очевидно, что мы нашли противоречие; Δ должно быть истинным всякий раз, когда Γ истинно, доказывая правило усечения.

В задаче 9.86 мы показываем, что *правильность* правил сокращения может быть доказана при использовании правил *обмена*, *ослабления* и *усечения*.

Теперь мы начинаем правила введения. Для ¬-левого правила пусть $\Gamma \to \Delta \vee \alpha$. Допустим, что Γ и $\neg\alpha$ являются истинными, и допустим для противоречия, что Δ ложно. Тогда мы имеем: истина → ложь ∨ ложь; это закрепление значений противоречит гипотезе. Таким образом, $\neg\alpha \wedge \Gamma \to \Delta$. Схожий аргумент может быть дан для ¬-правого правила.

∧-левое и ∨-правое правила следуют из коммутативной и ассоциативной природы антецедента и сукцедента. ∧-правое и ∨-левое правила могут быть быстро доказаны с помощью доказательства от противного, аналогично правилам ¬.

Задача 9.90. Каждое правило обмена является своей собственной инверсией, поэтому каждое является инвертируемым в результате своей собственной правильности. Схожим образом правила введения ¬ являются инверсиями друг друга. Правила сокращения доказываются непосредственно посредством логических утверждений, что $(\Gamma \wedge \alpha \wedge \alpha) \Leftrightarrow (\Gamma \wedge \alpha)$ и $(\Delta \vee \alpha \vee \alpha) \Leftrightarrow (\Delta \vee \alpha)$. Схожим образом правило усечения является результатом двух явно истинных наблюдений: $(\alpha \wedge \Gamma) \to \Gamma$ и $\Delta \to (\Delta \vee \alpha)$.

∧-правое и ∨-левое правила явно не меняют смысла секвенции при условии, что конъюнкции и дизъюнкции коммутативны и ассоциативны.

В ∧-правом правиле пусть $\Gamma \to \Delta, (\alpha \wedge \beta)$. Если Γ является ложным, тогда обе верхние секвенции являются истинными вне зависимости от значения их сукцедентов. Допустим, что Γ истинно. Опять же, если Δ является истинным, то доказательство завершено – допустим, что Δ является ложным. Тогда $(\alpha \wedge \beta)$ оба являются истинными по гипотезе, поэтому $\Delta \vee \alpha$ и $\Delta \vee \beta$ являются истинными. Тем самым $\Gamma \to \Delta, \alpha$ и $\Gamma \to \Delta, \beta$, и доказательство завершено.

Для того чтобы доказать, что ∨-левое правило является инвертируемым, пусть $(\alpha \vee \beta), \Gamma \to \Delta)$ равно истине. Допустим, что α и Γ оба являются истинными. Тогда $(\alpha \vee \beta) \wedge \Gamma$ является истинным, поэтому Δ должно быть истинно. Следовательно, α,

$\Gamma \to \Delta$ является истинным. Идентичная аргументация может быть выдвинута для того, чтобы доказать, что $\beta, \Gamma \to \Delta$, завершая доказательство.

Наконец, мы приведем пример, в котором инверсия правила ослабления оказывается безуспешной. Пусть $\alpha, \Gamma \to \Delta$ равно истине. Ясно, что если α является ложным и Γ является истинным, то никакого заключения о Δ сделать невозможно – оно может быть либо истинным, либо ложным. В случае если Δ является ложным, мы находим противоречие логическому утверждению, что $\Gamma \to \Delta$.

Задача 9.92. Пять правил не нужны: правила сокращения, ослабления и усечения. Нам нужны правила обмена, для того чтобы доказательства соответствовали точному порядку следования заданных правил, и нам нужно любое правило введения связок, которое применимо.

Задача 9.93. Любая нетривиальная формула может быть записана в одной из следующих форм: $\alpha \wedge \beta$, $\alpha \vee \beta$ или $\neg\alpha$. Для того чтобы доказать, что РК′ является полным, нам нужно только показать, что формулы в этих формах могут быть введены из их компонентных частей. Мы обеспечиваем их построения. В первую очередь $\wedge$:

$$\dfrac{\dfrac{\dfrac{\alpha \to \alpha}{\alpha, \beta \to \alpha} \qquad \dfrac{\beta \to \beta}{\alpha, \beta \to \beta}}{\alpha, \beta \to \alpha \wedge \beta}}{\alpha \wedge \beta \to \alpha \wedge \beta}.$$

Затем $\vee$:

$$\dfrac{\dfrac{\dfrac{\alpha \to \alpha}{\alpha \to \alpha, \beta} \qquad \dfrac{\beta \to \beta}{\beta \to \alpha, \beta}}{\alpha \vee \beta \to \alpha, \beta}}{\alpha \vee \beta \to \alpha \vee \beta}.$$

И наконец, $\neg\alpha \to \neg\alpha$ является результатом двух быстрых применений правил введения $\neg$ к $\alpha \to \alpha$.

Задача 9.96. Мы закрепляем вес за каждым символом в стиле, схожем с рис. 9.9. Каждый n-арный предикатный символ имеет вес, равный $(n - 1)$. Например, символ 4-арной функции имеет вес 3. В качестве расширения этого правила константы (которые на самом деле являются символами 0-арной функции) имеют вес -1. Переменные, которые (как и константы) представляют собой полные члены, также имеют вес -1. Мы утверждаем, что каждый член весит -1 и что каждый надлежащий начальный сегмент весит, по крайней мере, 0. Сначала рассмотрим тривиальный случай: член, состоящий из одной константы или переменной. Его вес равен -1, и единственным надлежащим начальным сегментом является пустой сегмент, который имеет вес 0.

Более того, это свойство является явно индуктивным. Любой нетривиальный член является n-арным предикатным символом с весом $(n - 1)$, за которым следуют n членов. Если каждый из этих членов имеет вес -1, то результирующий член имеет вес $(n - 1) - n = -1$. Любой начальный сегмент состоит из этого n-арного символа, менее n полных членов и до 1 неполного члена; этот член весит $\geqslant 0$, и очевидно, что недостаточно полных членов, чтобы преодолеть вес $(n - 1)$, предписанный начальным предикатом – любой надлежащий начальный сегмент должен иметь вес $\geqslant 0$. Из этого следует, что две одинаковые строки не могут представлять

один и тот же предикат и ряд членов, поскольку из этого вытекало бы, что некоторый включенный член является надлежащим начальным сегментом другого; более подробное объяснение этого последнего шага см. в решении задачи 9.81.

Задача 9.98. Докажем это с помощью основных семантических определений: пусть $\mathcal{M}$ равно любой структуре и σ равно любому объектному закреплению. Предположим, $\mathcal{M} \models (\forall x\alpha \vee \forall x\beta)[\sigma]$. Тогда $\mathcal{M} \models \forall x\alpha[\sigma]$ или $\mathcal{M} \models \forall x\beta[\sigma]$.

Случай (1): $\mathcal{M} \models \forall x\alpha[\sigma]$. Тогда $\mathcal{M} \models \alpha[\sigma(m/x)]$ для всех $m \in M$. Тогда $\mathcal{M} \models (\alpha \vee \beta)[\sigma(m/x)]$ для всех $m \in M$. Поэтому $\mathcal{M} \models \forall x(\alpha \vee \beta)[\sigma]$.

Случай (2): $\mathcal{M} \models \forall x\beta[\sigma]$; та же идея, что и выше.

Следовательно, $\mathcal{M} \models \forall x(\alpha \vee \beta)[\sigma]$. По определению логического следствия $(\forall x\alpha \vee \forall x\beta) \models \forall x(\alpha \vee \beta)$.

Задача 9.99. Нет, не обязательно. Мы используем определение логического следствия, для того чтобы доказать это. Для того чтобы доказать, что правая сторона *не является* логическим следствием левой стороны, мы должны продемонстрировать модель $\mathcal{M}$, объектное закрепление σ и формулы α, β такие, что $\mathcal{M} \models \forall x(\alpha \vee \beta)[\sigma]$, но $\mathcal{M} \not\models (\forall x\alpha \vee \forall x\beta)[\sigma]$.

Пусть α и β равно соответственно Px и Qx (P, Q – унарные предикаты). Теперь определим $\mathcal{M}$ и σ. Поскольку формулы являются сентенцией, то не нужно определять σ. $\mathcal{M}$: пусть универсум дискурса равен $M = \mathbb{N}$. Нам все еще нужно придать смысл P, Q в $\mathcal{M}$, пусть $P^{\mathcal{M}} = \{0, 2, 4, \ldots\}$ и $Q^{\mathcal{M}} = \{1, 3, 5, \ldots\}$. Тогда $\mathcal{M} \models \forall x(Px \vee Qx)$ (поскольку каждое число является четным или нечетным).

Но $\mathcal{M} \not\models (\forall xPx \vee \forall xQx)$ (потому что не является истиной, что либо все числа четные, либо все числа нечетные).

Задача 9.100. Базовый случай: пусть $u = ft_1t_2 \ldots t_n$, где некоторые из входных членов t_i функции f могут быть x, но ни один из них в прочих случаях не включает x. Если ни один из членов не является x, то ясно, что

$$(u(t/x))^{\mathcal{M}}[\sigma] = u^{\mathcal{M}}[\sigma] = u^{\mathcal{M}}[\sigma(m/x)]$$

для любого m; x не входит в u, поэтому замещения ничего не делают при применении к u. В противном случае существуют некоторые i такие, что $t_i = x$; ниже мы исходим из того, что x присутствует один раз в членах, но эта деталь не имеет значения.

$$\begin{aligned}(u(t/x))^{\mathcal{M}}[\sigma] &= ((ft_1 \ldots x \ldots t_n)(t/x))^{\mathcal{M}}[\sigma] \\ &= (ft_1 \ldots t \ldots t_n)^{\mathcal{M}}[\sigma] \\ &= f^{\mathcal{M}}(t_1^{\mathcal{M}}[\sigma], \ldots t^{\mathcal{M}}[\sigma], \ldots t_n^{\mathcal{M}}[\sigma]).\end{aligned}$$

Схожим образом

$$\begin{aligned}u^{\mathcal{M}}[\sigma(m/x)] &= (ft_1 \ldots x \ldots t_n)^{\mathcal{M}}[\sigma(m/x)] \\ &= f^{\mathcal{M}}(t_1^{\mathcal{M}}[\sigma(m/x)], \ldots, x^{\mathcal{M}}[\sigma(m/x)], \ldots, t_n^{\mathcal{M}}[\sigma(m/x)]).\end{aligned}$$

Здесь мы используем знание о том, что не x-члены не содержат x.

$$\begin{aligned}&= f^{\mathcal{M}}(t_1^{\mathcal{M}}[\sigma], \ldots, m, \ldots t_n^{\mathcal{M}}[\sigma]) \\ &= f^{\mathcal{M}}(t_1^{\mathcal{M}}[\sigma], \ldots t^{\mathcal{M}}[\sigma], \ldots t_n^{\mathcal{M}}[\sigma]).\end{aligned}$$

Поэтому в этом случае $(u(t/x))^{\mathcal{M}}[\sigma] = u^{\mathcal{M}}[\sigma(m/x)]$.

Индукция очень проста в сравнениях: если это применимо к каждому члену, который вводится в любую данную функцию, то это применимо и к выходу функции вследствие рекурсивного характера оценивания членов.

Задача 9.101. Например, предположим, что α равно $\forall y\neg(x = y + y)$. Здесь говорится, что «x является нечетным». Но $\alpha(x + y/x)$ равно $\forall y\neg(x + y = y + y)$, что всегда является ложным, независимо от значения $\sigma(x)$. Проблема в том, что y в члене $x + y$ «пойман» квантором $\forall y$.

Задача 9.103. Если α является атомарной формулой, то она имеет форму $Pt_1 \ldots t_n$. В задаче 9.100 мы показываем, что если t_i является членом, то $(t_i(t/x))^{\mathcal{M}}[\sigma] = t_i^{\mathcal{M}}[\sigma(m/x)]$, где $m = t^{\mathcal{M}}[\sigma]$. Таким образом, следующие утверждения являются эквивалентными:

$\mathcal{M} \vDash \alpha(t/x)[\sigma]$;
$\mathcal{M} \vDash ((Pt_1 \ldots t_n)(t/x))[\sigma]$;
$(t_1(t/x)^{\mathcal{M}}[\sigma], \ldots, t_n(t/x)^{\mathcal{M}}[\sigma]) \in P^{\mathcal{M}}$;
$(t_1^{\mathcal{M}}[\sigma(m/x)], \ldots, t_n^{\mathcal{M}}[\sigma(m/x)]) \in P^{\mathcal{M}}$;
$\mathcal{M} \vDash (Pt_1 \ldots t_n)[\sigma(m/x)]$;
$\mathcal{M} \vDash \alpha[\sigma(m/x)]$.

В связи с этим для любой атомной формулы α, $\mathcal{M} \vDash \alpha(t/x)[\sigma]$ тогда и только тогда, когда $\mathcal{M} \vDash \alpha[\sigma(m/x)]$.

Пусть α, β равны любым двум формулам с этим свойством. Следующие утверждения являются эквивалентными:

$\mathcal{M} \vDash ((\alpha \wedge \beta)(t/x))[\sigma]$;
$\mathcal{M} \vDash (\alpha(t/x))[\sigma]$ и $\mathcal{M} \vDash (\beta(t/x))[\sigma]$;
$\mathcal{M} \vDash \alpha[\sigma(m/x)]$ и $\mathcal{M} \vDash \beta[\sigma(m/x)]$;
$\mathcal{M} \vDash (\alpha \wedge \beta)[\sigma(m/x)]$.

Более того, то же самое можно сказать о ∨ и ∧. Наконец, мы имеем следующие эквмвалентности:

$\mathcal{M} \vDash (\forall y(\alpha(t/x)))[\sigma]$;
$\mathcal{M} \vDash \alpha(t/x)[\sigma(n/y)]$ для всех $n \in \mathcal{M}$.

Мы применяем, что y не входит в t, для того чтобы приравнять два приведенных выше и два приведенных ниже утверждения:

$\mathcal{M} \vDash \alpha[\sigma(m/x)(n/y)]$ для всех $n \in \mathcal{M}$;
$\mathcal{M} \vDash (\forall y\alpha)[\sigma(m/x)]$.

Здесь первые два идентичны вторым двум, потому что y не входит в t (в противном случае t не был бы свободно замещаемым для x), поэтому два замещения не пересекаются (то есть они не влияют на какие-либо общие члены). Та же самая аргументация может быть применена к ∃ как ∀.

Задача 9.104. Два правила не требуют обоснования:

$$\frac{\alpha(t), \Gamma \to \Delta}{\forall x\alpha(x), \Gamma \to \Delta}$$

и

$$\frac{\Gamma \to \Delta, \alpha(t)}{\Gamma \to \Delta, \exists x\alpha(x)}.$$

Очевидно, что $\forall x\alpha(x) \Rightarrow \alpha(t)$, поэтому если из $\alpha(t) \wedge \Gamma$ вытекает, что Δ истинно, то таковым является и $\forall x\alpha(x) \wedge \Gamma$. Схожим образом если из Γ вытекает, что $\Delta \vee \alpha(t)$ является истинным, то из этого также вытекает, что $\Delta \vee \exists x\alpha(x)$, так как $\alpha(t) \Rightarrow \exists x\alpha(x)$.

Два других менее тривиальны; на первый взгляд, не сразу ясно, что они являются правильными. Ключевое понимание исходит из нашей ранее определенной номенклатуры; где t в приведенном выше примере – это конкретный член, b – свободная переменная, поэтому мы можем считать ее произвольным элементом M (или, что то же самое, любым элементом M). Давайте сначала взглянем на правило правого введения для $\forall$:

$$\frac{\Gamma \to \Delta, \alpha(b)}{\Gamma \to \Delta, \forall x\alpha(x)}.$$

Поскольку b является произвольной (то есть не указано никакого закрепления σ для дальнейшей конкретизации смысла b), верх должен быть истинным с любым применимым x, закрепленным за b, отсюда и результат.

Далее мы рассмотрим правило левого введения для $\exists$:

$$\frac{\alpha(b), \Gamma \to \Delta}{\exists x\alpha(x), \Gamma \to \Delta}.$$

Опять же, ключ в том, что за b ничего не закреплено. Предпосылка заключается в том, что для любого b, $\alpha(b) \wedge \Gamma \Rightarrow \Delta$. В связи с этим из существования x, удовлетворяющего условию $\alpha(x)$, вытекает, что это упомянутое x может быть «подставлено вместо» b в предпосылке с целью, чтобы $\alpha(x) \wedge \Gamma \Rightarrow \Delta$.

Задача 9.105. Пусть M равно натуральным числам. Для наших целей σ не имеет значения, поэтому мы оставим его неопределенным. Рассмотрим следующую секвенцию: $(b = y + y) \to \alpha(b)$, где $\alpha(b)$ обозначает «b является четной». Очевидно, что эта секвенция является истинной. Рассмотрим теперь результат $\forall$-справа: $(b = y + y) \to \forall x\alpha(x)$. Эта секвенция контатирует «если b является четной, то каждый x является четным», что очевидно ложно.

Далее рассмотрим тривиальную секвенцию $\beta(b) \to (b > 2)$, где $\beta(b)$ обозначает «$b \geqslant 3$». Она очевидно является истинной, но если мы применим $\exists$-слева, то получим: $\exists x\beta(x) \to (b > 2)$. Другими словами, из существования натурального числа $x \geqslant 3$ вытекает, что $b > 2$; но b является свободной переменной, она может быть равна 1 либо 0 в зависимости от σ.

Задача 9.106. Пусть $\alpha(x)$ равно $(x = 0 \vee \exists y(x = sy))$. Мы обрисуем доказательство неформально, но, разумеется, доказательство может быть формализовано в арифметике Пеано на основе LK. Базовый случай: $x = 0$, и арифметика Пеано на основе LK легко доказывает $\alpha(0)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\to \forall x(x = x)}{\to 0 = 0, \forall x(x = x)}\ \text{ослабление и обмен} \qquad \dfrac{0 = 0 \to 0 = 0}{\forall x(x = x) \to 0 = 0}\ \forall\text{-слева}}{}\ \text{усечение}}{\dfrac{\to 0 = 0}{\to 0 = 0, \exists y(0 = sy)}\ \text{ослабление}}}{\to 0 = 0 \vee \exists y(0 = sy)}\ \vee\text{-справа}$$

Индукционный шаг: показываем, что арифметика Пеано на основе LK доказывает $\forall x(\alpha(x) \rightarrow \alpha(sx))$, то есть мы должны дать доказательство секвенции с использованием арифметики Пеано на основе LK:

$$\rightarrow \forall x(\neg(x = 0 \vee \exists y(x = sy)) \vee (sx = 0 \vee \exists y(sx = sy))).$$

Это не сложно и оставлено читателю. Из формул $\alpha(0)$ и $\forall x(\alpha(x) \rightarrow \alpha(sx))$ и с использованием аксиомы

$$\rightarrow (\alpha(0) \wedge \forall x(\alpha(x) \rightarrow \alpha(sx))) \rightarrow \forall x\alpha(x)$$

теперь мы можем заключить (всего за несколько шагов): $\rightarrow \forall x\alpha(x)$, что мы и хотели доказать. Таким образом, арифметика Пеано на основе LK доказывает $\forall x\alpha(x)$.

9.6. Примечания

Эпиграфом к этой главе является цитата из плодовитого писателя-философа сэра Роджера Скратона. «Напитки в Хельсинки» (Drinks in Helsinki), глава из книги [Scruton (2005)], настолько смешна, насколько это возможно в серьезной рукописи, и она напоминает этому автору о его собственном опыте в Турку, Финляндия (представляя работу [Soltys (2004)]).

$\mathbb{N}$ (множество натуральных чисел) и принцип индукции очень тесно связаны; строгое определение $\mathbb{N}$ как теоретико-множественного объекта заключается в следующем: это *уникальное множество*, удовлетворяющее следующим трем свойствам: (i) оно содержит 0, (ii) если в нем находится n, то же самое относится и к $n+1$, и (iii) оно удовлетворяет принципу индукции (который в данном контексте формулируется следующим образом: если S есть подмножество $\mathbb{N}$ и S удовлетворяет указанным выше (i) и (ii), то на самом деле $S = \mathbb{N}$).

Ссылки в настоящем абзаце сделаны из уважения к эпохальному труду Кнута «Искусство программирования» [Knuth (1997)]. Для подробного изучения алгоритма Евклида см. § 1.1. Задача 9.2 проистекает из § 1.2.1, задача #8, стр. 19. См. § 2.3.1, стр. 318 за дополнительными общими сведениями по обходу деревьев. Относительно истории понятия пред- и постусловий и инвариантов цикла см. стр. 17. В частности, относительно материала, связанного с расширенным алгоритмом Евклида, см. стр. 13, алгоритм E, в книге [Knuth (1997)], стр. 937 в книге [Cormen и соавт. (2009)], и стр. 292, алгоритм A.5, в книге [Delfs и Knebl (2007)]. Мы приводим рекурсивную версию данного алгоритма в разделе 3.4.

См. книгу [Zingaro (2008)], если требуется книга, посвященная идее инвариантов в контексте доказательства правильности алгоритмов. Замечательным источником задач по принципу инвариантности, то есть раздел 9.1.2, является глава 1 в книге [Engel (1998)].

Пример с доской 8×8 и двумя отсутствующими клетками (рис. 9.2) взят из книги [Dijkstra (1989)].

За дополнительными алгебраическими сведениями обращайтесь к книгам [Dummit и Foote (1991)] и [Alperin и Bell (1995)]. Относительно теории чисел, в особенности связанной с криптографией, см. книгу [Hoffstein и соавт. (2008)]. Классическим учебником по теории чисел является книга [Hardy и Wright (1980)].

Раздел об отношениях основан на рукописных лекционных слайдах Рышарда Яницкого. Элементарное введение в отношения можно найти в главе 8 книги

[Rosen (2007)], а для получения очень быстрого введения в отношения (вплоть до определения классов эквивалентности) читателю предлагается прочитать восхитительный раздел 7 книги [Halmos (1960)].

Другой взгляд на частичные порядки предлагается в книге [Мендельсон (1970)], глава 3. В этой книге автор подходит к частичным порядкам с точки зрения *булевой алгебры*, которые определяются следующим образом: множество B вместе с двумя бинарными операциями $\curlywedge$, $\curlyvee$ (обычно обозначаемыми $\wedge$, $\vee$, но мы применяем их для «и», «или», и поэтому здесь мы используем их «фигурные» версии, чтобы подчеркнуть, что «$\curlywedge$» и «$\curlyvee$» не обязательно являются стандартными булевыми операторами) на B, операции сингулярности $'$ на B и двумя конкретными элементами 0 и 1 множества B, и удовлетворяющее множество аксиом: $x \curlywedge y = y \curlywedge x$ и $x \curlyvee y = y \curlyvee x$, дистрибутивность $\curlywedge$ над $\curlyvee$, и наоборот, а также $x \curlywedge 1 = x$ и $x \curlyvee 0 = x$, $x \curlyvee x = 1$ и $x \curlywedge x = 0$, и наконец, $0 \neq 1$. Булева алгебра обычно обозначается секступлем $\mathcal{B} = \langle B, \curlywedge, \curlyvee, ', 0, 1\rangle$, и при этом исходят из того, что она удовлетворяет только что перечисленным аксиомам.

При наличии в булевой алгебре $\mathcal{B}$ мы определяем бинарное отношение $\preccurlyeq$ следующим образом:

$$x \preccurlyeq y \Leftrightarrow x \curlywedge y = x.$$

Оно оказывается эквивалентным нашему понятию порядка в решетке. Затем Мендельсон абстрагирует три свойства рефлексивности, антисимметрии и транзитивности и говорит, что любое отношение, удовлетворяющее всем трем, является частичным порядком – и не каждый частичный порядок является решеткой.

Существует ряд отличных введений в логику; например, книги [Buss (1998)] и [Bell и Machover (1977)]. Этот раздел основан на лекциях по логике Стивена Кука в Университете Торонто.

Задача 9.83 является, конечно, примером общего булева «принципа двойственности». Теоретико-доказательная версия этого принципа дается, например, как теорема 3.4 в книге [Мендельсон (1970)], где *дуал* пропозиции относительно булевой алгебры B является пропозицией, полученной путем замены $\curlyvee$ на $\curlywedge$ и $\curlywedge$ на $\curlyvee$ (см. стр. 293, где мы определили эти символы). Мы также заменяем 0 на 1 и 1 на 0. Тогда если высказывание выводимо из обычных аксиом булевой алгебры, то и его двойственное.

Раздел 9.3.6 о правильности рекурсивных алгоритмов основан на главе 5 книги [Manna (1974)].

Библиография

Agrawal M., Kayal N. and Saxena N. (2004). Primes is in P, Annals of Mathematics 160, 2, p. 781–793.

Aleknovich M., Borodin A., Buresh-Oppenheim J., Impagliazzo R., Magen A. and Pitassi T. (2005). Toward a model of backtracking and dynamic programming.

Alford W. R., Granville A. and Pomerance C. (1994). There are infinitely many Carmichael numbers, Annals of Mathematics 139, 3, p. 703–722.

Allan Borodin C. R., Morten N. Nielsen (2003). (incremental) priority algorithms, Algorithmica 37, 4, p. 295–326.

Alperin J. L. and Bell R. B. (1995). Groups and Representations (Springer).

Arrow K. (1951). Social Choice and Individual Values (J. Wiley).

Austrin P. (2010). How hard is unshuffling a string (reply), CS Theory Stack Exchange reply to[Erickson (2010)] // http://cstheory.stackexchange.eom/q/692.

Bell J. and Machover M. (1977). A course in mathematical logic (North-Holland).

Berkowitz S. J. (1984). On computing the determinant in small parallel time using a small number of processors, Information Processing Letters 18, 3, p. 147–150.

Borodin A. and El-Yaniv R. (1998). Online Computation and Competitive Analysis (Cambridge University Press).

Bozoki S. and Rapcsak T. (2008). On saaty's and koczkodaj's inconsistencies of pairwise comparison matrices, Journal of Global Optimization 42, 2, p. 157–175, doi:10.1007/s10898-007-9236-z.

Brin S. and Page L. (1998). The anatomy of a large-scale hypertextual web search engine, in Proceedings of the seventh international conference on World Wide Web 7, WWW7 (Elsevier Science Publishers B. V., Am-sterdam, The Netherlands, The Netherlands), p. 107–117, URL: http://dl.acm.org/citation.cfm?id=297805.297827.

Bush V. (1945). As we may think, The Atlantic Monthly.

Buss S. R. (1998). An introduction to proof theory, in S. R. Buss (ed.), Handbook of Proof Theory (North Holland), p. 1–78.

Buss S. R. and Soltys M. (2013). Unshuffling a square is NP-hard, Journal of Computer and System Sciences 80, 4, p. 766–776, doi: http://dx.doi.org/10.1016/j.jcss.2013.11.002, URL: http://www.sciencedirect.com/science/article/pii/S002200001300189X.

Cenzer D. and Remmel J. B. (2001). Proof-theoretic strength of the stable marriage theorem and other problems, Reverse Mathematics, p. 67–103.

Christian B. and Griffiths T. (2016). Algorithms to Live By: the Computer Science of Human Decisions (Henry Holt and Company, LLC).

Church A. (1996). Introduction to Mathematical Logic (Princeton University Press).

Clarke R. A. and Knake R. (2011). Cyber War: The Next Threat to National Security and What to Do About It (Ecco; Reprint edition).

Condorcet (1785). Essai sur l'application de l'analyse 'a la probabilite des decisions rendues a la pluralite des vois, Paris.

Cook S. A. and Nguyen P. (2010). Logical Foundations of Proof Complexity (Cambridge Univeristy Press).

Cook S. A. and Soltys M. (1999). Boolean programs and quantified propositional proof systems, Bulletin of the Section of Logic 28, 3, p. 119–129.

Cormen T. H., Leiserson C. E., Rivest R. L. and Stein C. (2009). Introduction to Algorithms, 3rd edn. (McGraw-Hill Book Company), third Edition.

Delfs H. and Knebl H. (2007). Introduction to Cryptography (Springer).

Dickens C. (1850). David Copperfield (Penguin Classics).

Dickens C. (1854). Hard Times (Everyman's Library).

Dickens C. (2013). Introduction to Computer Science using Python: A compu-tational problem solving focus (Wiley).

Dijkstra E. W. (1989). On the cruelty of really teaching computing science, Communications of the ACM 32, 12.

Doctorow E. L. (1971). The Book of Daniel (Plume Penguin).

Dorrigiv R. and Lopez-Ortiz A. (2009). On developing new models, with paging as a case study, ACM SIGACT News 40, 4.

Downey A. (2015). Think Python: How to Think Like a Computer Scientist, 2nd edn. (Green Tea Press).

Duda H. (1977). Zajecia pozalekcyjne z matematyki w szkole podstawowej. Zbiory i relacje (Wydawnictwa Szkolne i Pedagogiczne).

Dummit D. S. and Foote R. M. (1991). Abstract Algebra (Prentice Hall).

Dyer J. S. (1990). Remarks on the analytic hierarchy process, Manage. Sci. 36, 3, p. 249–258, doi:10.1287/mnsc.36.3.249, URL: http://dx.doi.org/10.1287/mnsc.36.3.249.

Easley D. and Kleinberg J. (2010). Networks, crowds, and markets: reasoning about a highly connected world (Cambridge).

Engel A. (1998). Problem-Solving Strategies (Springer).

Erickson J. (2010). How hard is unshuffling a string? CS The¬ory Stack Exchange posting // http://cstheory.stackexchange.com/questions/34/how-hard-is-unshuffling-a-string.

Faliszewski P., Hemaspaandra E. and Hemaspaandra L. A. (2010). Using complexity to protect elections, Communications of the ACM 53, 11, p. 74, doi:10.1145/1839676.1839696, URL: http://dx.doi.org/10.1145/1839676.1839696.

Fernandez A. G. and Soltys M. (2013). Feasible combinatorial matrix theory, in K. Chatterjee and J. Sgall (eds.), Mathematical Foundations of Computer Science 2013, Lecture Notes in Computer Science, Vol. 8087 (Springer Berlin Heidelberg), ISBN 978-3-642-40312-5, p. 777–788, doi: 10.1007/978-3-642-40313-2_68, URL: http://dx.doi.org/10.1007/978-3-642-40313-2_68.

Franceschet M. (2011). Pagerank: standing on the shoulders of giants, Communications of the ACM 54, 6.

Fred D. Taylor J. (2011). Software: The broken door of cyberspace security, Harvard Law School National Security Journal URL: http://harvardnsj.org/2011/02/software-the-broken-door-of-cyberspace-security/.

Gale D. and Shapley L. S. (1962). College admissions and the stability of marriage, American, Mathematical Monthly 69, p. 9–14.

Ginsburg S. and Spanier E. (1965). Mapping of languages by two-tape devices, Journal of the Association of Computing Machinery 12, 3, p. 423–434.

Gischer J. (1981). Shuffle languages, Petri nets, and context-sensivite grammars, Communications of the ACM 24, 9, p. 597–605.

Gruber H. and Holzer M. (2009). Tight bounds on the descriptional complexity of regular expressions, in Proc. Intl. Conf. on Developments in Language Theory (DLT) (Springer Verlag), p. 276–287.

Hagele G. and Pukelsheim F. (2001). Llull's writings on electoral systems, Studia Lulliana 41, p. 3–38, URL: https://www.math.uni-augsburg.de/htdocs/emeriti/pukelsheim/2001a.html.

Halmos P. R. (1960). Naive Set Theory (Springer-Verlag).

Halmos P. R. (1995). Linear algebra problem book (The mathematical association of America).

Hardy G. H. and Wright E. M. (1980). An Introduction to the Theory of Num-bers, 5th edn. (Oxford University Press).

Harel D. (1987). Algorithmics: The Spirit of Computing (The Addison-Wesley Publishing Company), ISBN 0-201-19240-3.

Harris R. (1996). Enigma (Ballantine Books).

Henshall D., Rampersad N. and Shallit J. (2012). Shuffling and unshuffling, Bulletin of the EATCS 107, p. 131–142.

Hoffman P. (1998). The Man Who Loved Only Numbers: The Story of Paul Erdos and the Search for Mathematical Truth (Hyperion).

Hoffstein J., Pipher J. and Silverman J. H. (2008). An Introduction to Mathematical Cryptography (Springer).

Hutton G. (2007). Programming in Haskell (Cambridge University Press, New York, NY, USA), ISBN 0521871727, 9780521871723.

Janicki R. (2011). Approximations of arbitrary relations by partial orders: Clas¬sical and rough set models, in J. F. P. et al (ed.), Transactions on Rough Sets XIII, LNCS, Vol. 6499 (Springer-Verlag Berlin Heidelberg).

Janicki R. and Zhai Y. (2011). Remarks on pairwise comparison numerical and non-numerical rankings, Lecture Notes in Computer Science 6954, p. 290–300.

Jantzen M. (1981). The power of synchronizing operations on strings, Theoretical Computer Science 14, p. 127–154.

Jantze M. (1985). Extending regular expressions with iterated shuffle, Theoretical Computer Science 38, p. 223–247.

Jedrzejowicz J. (1999). Structural properties of shuffle automata, Grammars 2, 1, p. 35–51.

Jedrzejowicz J. and Szepietowski A. (2001). Shuffle languages are in P, Theoretical Computer Science 250, 1–2, p. 31–53.

Jedrzejowicz J. and Szepietowski A. (2005). On the expressive power of the shuffle operator matched with intersection by regular sets, Theoretical Informatics and Applications 35, p. 379–388.

Johnson P. (1991). The Birth of the Modern (Phoenix Giant).

Kakiashvili T., Koczkodaj W. W. and Woodbury-Smith M. (2012). Improving the medical scale predictability by the pairwise comparisons method: evidence from a clinical data study, Comput Methods Programs Biomed 105, 3, p. 210–6, doi:10.1016/j.cmpb.2011.09.011.

Karp R. M. and Rabin M. O. (1987). Efficient randomized pattern-matching algorithms, IBM Journal ofResearch and Development 31, 2, p. 249–260.

Katajainen J. and Traff J. L. (1997). A meticulous analysis of mergesort programs, in Proceedings of the Third Italian Conference on Algorithms and Complexity, CIAC '97 (Springer-Verlag, London, UK, UK), ISBN 3-540-62592-5, p. 217–228, URL: http://dl.acm.org/citation.cfm?id=648256.752881.

Kimball R. (2012). The fortunes of permanence: Culture and anarchy in an Age of Amnesia (St. Augustine's Press).

Kleinberg J. and Tardos E. (2006). Algorithm Design (Pearson Education).

Knuth D. E. (1997). The Art of Computer Programming, Vol. 1, Fundamental Algorithms, 3rd edn. (Addison Wesley).

Koczkodaj W. (1993). A new definition of consistency of pairwise comparisons, Mathematical and Computer Modelling 18, 7, p. 79–84, doi:http://dx.doi.org/10.1016/0895-7177(93)90059-8, URL: http://www.sciencedirect.com/science/article/pii/0895717793900598.

Koczkodaj W. W., Kulakowski K. and Ligeza A. (2014). On the quality evaluation of scientific entities in poland supported by consistencydriven pairwise comparisons method, Scientometrics 99, p. 911–926, doi: 10.1007/s11192-014-1258-y.

Kozen D. (2006). Theory of Computation (Springer).

Krajicek J. (1995). Bounded Arithmetic, Propositional Logic, and Complexity Theory (Cambridge).

Lehtonen E. (2008). Two undecidable variants of collatz's problems, Theoretical Computer Science 407, 1–3, p. 596–600, doi:10.1016/j.tcs.2008.08.029, URL: http://dx.doi.org/10.1016/j.tcs.2008.08.029.

Manna Z. (1974). Mathematical Theory of Computation (McGraw-Hill Computer Science Series).

Mansfield A. (1982). An algorithm for a merge recognition problem, Discrete Applied Mathematics 4, 3, p. 193–197, doi: http://dx.doi.org/10.1016/0166-218X(82)90039-7, URL: http://www.sciencedirect.com/science/article/pii/0166218X82900397.

Mansfield A. (1983). On the computational complexity of a merge recognition problem, Discrete Applied Mathematics 1, 3, p. 119–122.

Mayer A. J. and Stockmeyer L. J. (1994). The complexity of word problems – this time with interleaving, Information and Computation 115, p. 293–311.

Mendelson E. (1970). Boolean algebra and switching circuits (McGraw Hill).

Mhaskar N. and Soltys M. (2015). String shuffle: Circuits and graphs, Journal of Discrete Algorithms 31, 0, p. 120–128, doi: http://dx.doi.org/10.1016/j.jda.2015.01.003, URL: http://www.sciencedirect.com/science/article/pii/S1570866715000040, 24th International Workshop on Combinatorial Algorithms (IWOCA 2013).

Miller G. A. (1995). Wordnet: A lexical database for english, Communications of the ACM .

Ogden W. F., Riddle W. E. and Rounds W. C. (1978). Complexity of expressions allowing concurrency, in Proc. 5th ACM Symposium on Principles of Programming Languages (POPL), p. 185–194.

Papadimitriou C. H. (1994). Computational Complexity (Addison-Wesley).

Papadimitriou C. H. and Steiglitz K. (1998). Combinatorial Optimization: Algorithms and Complexity (Dover).

Press W. H., Vetterling W. T., Teukolsky S. A. and Flannery B. P. (2007). Numerical Recipes: The Art of Scientifc Computing, 3rd edn. (Cambridge University Press).

Reingold O. (2005). Undirected st-connectivity in log-space, in STOC'05: Proceedings of the thirty-seventh annual ACM symposium on Theory of computing, p. 376–385.

Renatus P. F. V. (4th or 5th century AD). De re militari.

Riddle W. E. (1973). A method for the description and analysis of complex software systems, SIGPLAN Notices 8, 9, p. 133–136.

Riddle W. E. (1979). An approach to software system modelling and analysis, Computer Languages 4, 1, p. 49–66.

Rosen K. H. (2007). Discrete Mathematics and Its Applications, 6th edn. (Mc-Graw Hill).

Saaty T. L. (1977). A scaling method for priorities in hierarchical structures, Journal of Mathematical Psychology 15, p. 234–281.

Schwartz R. L., Foy B. D. and Phoenix T. (2011). Learning Perl, 6th edn. (O'Reilly Media, Inc.), ISBN 1449303587, 9781449303587.

Scruton R. (2005). Gentle Regrets: Thoughts from a Life (Continuum).

Scruton R. (2011). Beauty: A Very Short Introduction (Oxford University Press).

Scruton R. (2014). The soul of the world (Princeton University Press).

Scruton R. (2015). Living with a mind, First Things.

Shaw A. C. (1978). Software descriptions with flow expressions, IEEE Transactions on Software Engineering SE-4, 3, p. 242–254.

Shoudai T. (1992). A P-complete language describable with iterated shuffle, Information Processing Letters 41, 5, p. 233–238.

Shustek, L. (2009). Interview, Communications of the ACM 52, 3, pp. 38–41.

Singh S. (1999). The Code Book: The evolution of secrecy, from Mary, Queen of Scots, to Quantum Cryptography (Doubleday).

Sipser M. (2006). Introduction to the Theory of Computation (Thompson), second Edition.

Solovay R. and Strassen V. (1977). A fast monte-carlo test for primality, SIAM Journal of Computing 6, p. 84–86.

Soltys M. (2002a). Berkowitz's algorithm and clow sequences, Electronic Journal of Linear Algebra 9, p. 42–54.

Soltys M. (2002b). Extended Frege and Gaussian Elimination, Bulletin of the Section of Logic 31, 4, p. 1–17.

Soltys M. (2004). LA, permutations, and the Hajos calculus, in J. Diaz, J. Karhumaki, A. Lepisto and D. Sannella (eds.), Automata, Languages and Programming, 31st International Colloquium (ICALP), Lecture Notes in Computer Science, Vol. 3142, European Association of Theoretical Computer Science, University of Turku (Springer), p. 1176–1187, doi: http://dx.doi.org/10.1007/978-3-540-27836-8_97.

Soltys M. (2005). Feasible proofs of matrix properties with Csanky's algoritm, in C.-H. L. Ong (ed.), Computer Science Logic, 19th International Workshop (CSL), Lecture Notes in Computer Science, Vol. 3634, Oxford University Computing Laboratory (Springer), p. 493–508, doi: http://dx.doi.org/10.1007/11538363_34.

Soltys M. (2009). An introduction to computational complexity (Jagiellonian University Press).

Soltys M. (2013). Circuit complexity of shuffle, in T. Lecroq and L. Mouchard (eds.), International Workshop on Combinatorial Algorithms 2013, Lecture Notes in Computer Science, Vol. 8288 (Springer), p. 402–411, doi:10. 1007/978- 3- 642-45278-9.34.

Soltys M. and Cook S. (2004). The proof complexity of linear algebra, Annals of Pure and Applied Logic 130, 1–3, p. 207–275, doi: http://dx.doi.org/10.1016/j.apal.2003.12.005.

Su F. E. (2010). Teaching research: encouraging discoveries, American Mathematical Monthly .

Thurstone L. L. (1927). A law of comparative judgement, Psychological Review 34, 278–286.

van Vliet H. (2000). Software Engineering: Principles and, Practice, 2nd edn. (Wiley).

Velleman D. J. (2006). How To Prove It, 2nd edn. (Cambridge University Press).

von zur Gathen J. (1993). Parallel linear algebra, in J. H. Reif (ed.), Synthesis of Parallel Algorithms (Morgan and Kaufman), p. 574–617.

von zur Gathen J. and Gerhard J. (1999). Modern computer algebra (Cambridge University Press).

Warmuth M. K. and Haussler D. (1984). On the complexity of iterated shuffle, Journal of Computer and System Sciences 28, 3, p. 345–358.

Whitman W. (1892). Song of myself.

Zhai Y. (2010). Pairwise comparisons based non-numerical ranking, Ph.D. thesis, McMaster University.

Zingaro D. (2008). Invariants: A generative approach to programming (College Publications).

Предметный указатель

P

Т

У

Ф

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Майкл Солтис

Введение в анализ алгоритмов

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Логунов А. В.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать офсетная.
Усл. печ. л. 22,59. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**