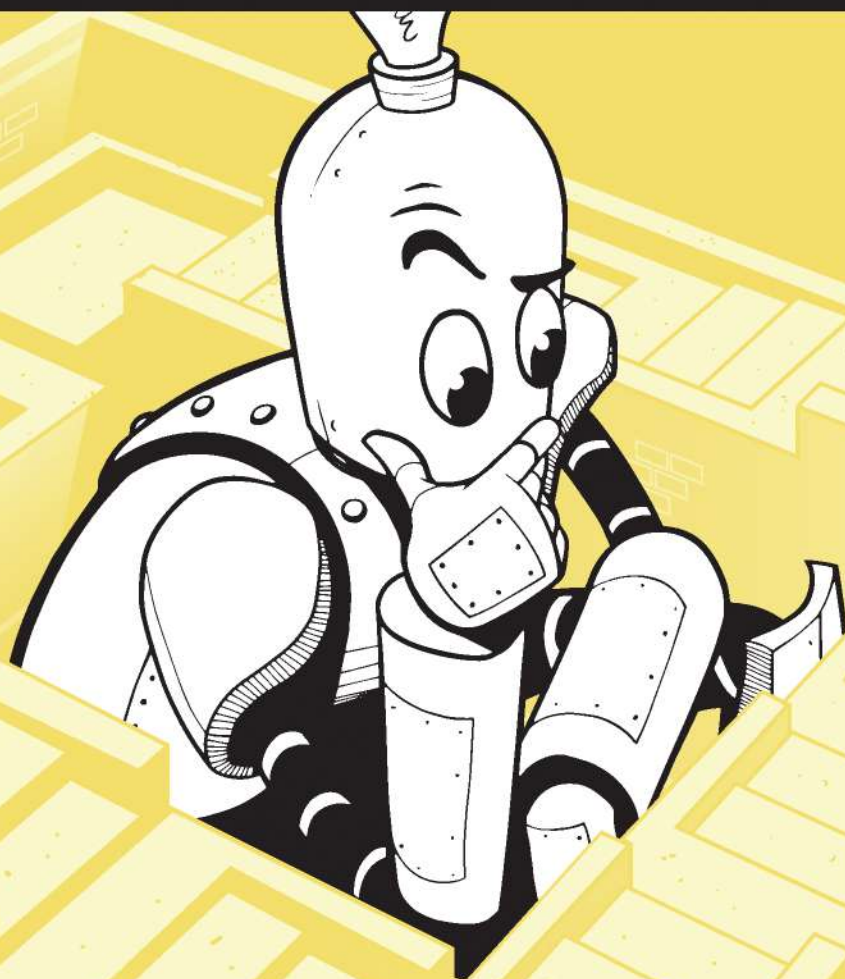


# АЛГОРИТМЫ НА ПРАКТИКЕ

РЕШЕНИЕ РЕАЛЬНЫХ ЗАДАЧ

ДАНИЭЛЬ ЗИНГАРО



# **ALGORITHMIC THINKING**

**A Problem-Based Introduction**

by Daniel Zingaro



**no starch  
press**

San Francisco

# АЛГОРИТМЫ НА ПРАКТИКЕ

РЕШЕНИЕ РЕАЛЬНЫХ ЗАДАЧ

ДАНИЭЛЬ ЗИНГАРО



Санкт-Петербург • Москва • Минск

2023

ББК 32.972.2-018  
УДК 004.021  
3-63

### **Зингаро Даниэль**

3-63 Алгоритмы на практике. — СПб.: Питер, 2023. — 432 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1853-3

«Алгоритмы на практике» научат решать самые трудные и интересные программистские задачи, а также разрабатывать собственные алгоритмы. В качестве примеров для обучения взяты реальные задания с международных соревнований по программированию. Вы узнаете, как классифицировать задачи, правильно подбирать структуру данных и выбирать алгоритм для решения. Поймете, что выбор структуры данных — будь то хеш-таблица, куча или дерево — влияет на скорость выполнения программы и на эффективность алгоритма. Разберетесь, как применять рекурсию, динамическое программирование, двоичный поиск.

Никакого условного псевдокода, все примеры сопровождаются исходным кодом на языке Си с подробными объяснениями.

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

ББК 32.972.2-018  
УДК 004.021

Права на издание получены по соглашению с No Starch Press. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1718500808 англ.

© 2021 by Daniel Zingaro. Algorithmic Thinking: A Problem-Based Introduction, ISBN 9781718500808, published by No Starch Press Inc. 245 8th Street, San Francisco, California United States 94103 Russian edition published under license by No Starch Press Inc.

ISBN 978-5-4461-1853-3

© Перевод на русский язык ООО «Прогресс книга», 2022  
© Издание на русском языке, оформление ООО «Прогресс книга», 2022  
© Серия «Библиотека программиста», 2022

# Краткое содержание

<b>Предисловие</b> .....	15
<b>Благодарности</b> .....	17
<b>Введение</b> .....	19
<b>Глава 1.</b> Хеш-таблицы .....	31
<b>Глава 2.</b> Деревья и рекурсия .....	64
<b>Глава 3.</b> Мемоизация и динамическое программирование .....	107
<b>Глава 4.</b> Графы и поиск в ширину .....	160
<b>Глава 5.</b> Кратчайший путь во взвешенных графах .....	208
<b>Глава 6.</b> Двоичный поиск .....	244
<b>Глава 7.</b> Кучи и деревья отрезков .....	295
<b>Глава 8.</b> Система непересекающихся множеств .....	353
<b>Послесловие</b> .....	401
<b>Приложение А.</b> Время выполнения алгоритма .....	403
<b>Приложение Б.</b> Потому что не могу удержаться .....	410
<b>Приложение В.</b> Сводка по задачам .....	426

# Оглавление

<b>Предисловие</b> .....	15
<b>Благодарности</b> .....	17
От издательства .....	18
<b>Введение</b> .....	19
Онлайн-ресурсы .....	20
Для кого эта книга .....	20
Язык программирования .....	21
Почему Си? .....	21
Ключевое слово Static .....	21
Добавление файлов .....	22
Освобождение памяти .....	22
Темы .....	23
Сайты с задачами .....	24
Структура описания задачи .....	26
Задача. Очереди за продуктами .....	27
Условие .....	27
Решение .....	28
Примечания .....	30
<b>Глава 1. Хеш-таблицы</b> .....	31
Задача 1. Уникальные снежинки .....	31
Условие .....	31
Упрощаем задачу .....	33
Решение основной задачи .....	35

Решение 1: последовательное сравнение .....	38
Решение 2: сокращение числа вычислений .....	42
Хеш-таблицы .....	48
Проектирование хеш-таблицы .....	48
Зачем использовать хеш-таблицы? .....	50
Задача 2. Сложносоставные слова .....	51
Условие .....	51
Определение сложносоставных слов .....	52
Решение .....	52
Задача 3. Проверка орфографии: удаление буквы .....	57
Условие .....	57
Размышление о хеш-таблицах .....	58
Ad hoc-подход .....	60
Выводы .....	63
Примечания .....	63
<b>Глава 2. Деревья и рекурсия .....</b>	<b>64</b>
Задача 1. Трофеи Хэллоуина .....	64
Условие .....	64
Двоичные деревья .....	66
Решаем пример .....	68
Представление двоичных деревьев .....	69
Сбор конфет .....	73
Принципиально другое решение .....	79
Обход минимального количества улиц .....	84
Считывание входных данных .....	87
Когда использовать рекурсию? .....	94
Задача 2. Расстояние до потомка .....	94
Условие .....	94
Считывание входных данных .....	97
Количество потомков одного узла .....	101
Количество потомков всех узлов .....	102
Упорядочивание узлов .....	103

Вывод информации .....	104
Функция main .....	105
Выводы .....	105
Примечания .....	106
<b>Глава 3. Мемоизация и динамическое программирование .....</b>	<b>107</b>
Задача 1. Страсть к бургерам .....	107
Условие .....	107
Разработка плана .....	108
Описание оптимальных решений .....	110
Решение 1. Рекурсия .....	112
Решение 2. Мемоизация .....	116
Решение 3. Динамическое программирование .....	122
Мемоизация и динамическое программирование .....	126
Шаг 1. Структура оптимальных решений .....	126
Шаг 2. Рекурсивное решение .....	127
Шаг 3. Мемоизация .....	127
Шаг 4. Динамическое программирование .....	128
Задача 2. Экономные покупатели .....	129
Условие .....	129
Описание оптимального решения .....	130
Решение 1. Рекурсия .....	133
Функция main .....	137
Решение 2. Мемоизация .....	139
Задача 3. Хоккейное соперничество .....	141
Условие .....	141
О принципиальных матчах .....	142
Описание оптимальных решений .....	144
Решение 1. Рекурсия .....	147
Решение 2. Мемоизация .....	150
Решение 3. Динамическое программирование .....	151
Оптимизация пространства .....	154



Задача 4. Учебный план .....	156
Условие .....	156
Решение. Мемоизация .....	157
Выводы .....	158
Примечания .....	159
<b>Глава 4. Графы и поиск в ширину .....</b>	<b>160</b>
Задача 1. Погоня за пешкой .....	160
Условие .....	160
Оптимальное перемещение .....	163
Лучший результат коня .....	172
Блуждающий конь .....	174
Оптимизация времени .....	178
Графы и BFS .....	179
Что такое графы? .....	179
Графы и деревья .....	180
BFS в графах .....	182
Задача 2. Лазание по канату .....	184
Условие .....	184
Решение 1. Поиск возможностей .....	185
Решение 2. Модификация .....	190
Задача 3. Перевод книги .....	199
Условие .....	199
Построение графа .....	200
BFS .....	204
Общая стоимость .....	206
Выводы .....	206
Примечания .....	207
<b>Глава 5. Кратчайший путь во взвешенных графах .....</b>	<b>208</b>
Задача 1. Мышиный лабиринт .....	208
Условие .....	209
BFS не подходит .....	209

Быстрые пути во взвешенных графах .....	211
Построение графа .....	215
Реализация алгоритма Дейкстры .....	217
Две оптимизации .....	220
Алгоритм Дейкстры .....	222
Время выполнения алгоритма Дейкстры .....	222
Ребра с отрицательными весами .....	223
Задача 2. Дорога к бабушке .....	225
Условие .....	226
Матрица смежности .....	227
Построение графа .....	228
Странные пути .....	229
Подзадача 1: кратчайшие пути .....	233
Подзадача 2: количество кратчайших путей .....	235
Выводы .....	243
Примечания .....	243
<b>Глава 6. Двоичный поиск .....</b>	<b>244</b>
Задача 1. Кормление муравьев .....	244
Условие .....	244
Новая форма задачи с деревом .....	247
Считывание входных данных .....	248
Проверка пригодности решения .....	250
Поиск решения .....	253
Двоичный поиск .....	254
Время выполнения двоичного поиска .....	255
Определение допустимости .....	256
Поиск по упорядоченному массиву .....	257
Задача 2. Прыжки вдоль реки .....	257
Условие .....	258
Жадная идея .....	259
Проверка допустимости .....	261

---

Поиск решения .....	266
Считывание входных данных .....	269
Задача 3. Качество жизни .....	269
Условие .....	270
Упорядочивание прямоугольников .....	272
Двоичный поиск .....	275
Проверка допустимости .....	276
Ускоренная проверка допустимости .....	278
Задача 4. Двери пещеры .....	284
Условие .....	285
Решение подзадачи .....	286
Использование линейного поиска .....	288
Использование двоичного поиска .....	290
Выводы .....	293
Примечания .....	293
<b>Глава 7. Кучи и деревья отрезков .....</b>	<b>295</b>
Задача 1. Акция в супермаркете .....	295
Условие .....	295
Решение 1. Максимум и минимум в массиве .....	296
Max-куча .....	300
Min-кучи .....	313
Решение 2. Кучи .....	315
Кучи .....	318
Два дополнительных варианта применения .....	318
Выбор структуры данных .....	319
Задача 2. Построение декартовых деревьев .....	320
Условие .....	320
Рекурсивный вывод декартовых поддеревьев .....	322
Сортировка по меткам .....	323
Решение 1. Рекурсия .....	324
Запросы максимума на отрезке .....	327

Деревья отрезков .....	329
Решение 2. Дерево отрезков .....	338
Деревья отрезков .....	339
Задача 3. Сумма двух элементов .....	340
Условие .....	340
Заполнение дерева отрезков .....	341
Запрос к дереву отрезков .....	346
Обновление дерева отрезков .....	347
Функция main .....	350
Выводы .....	351
Примечания .....	352
<b>Глава 8. Система непересекающихся множеств .....</b>	<b>353</b>
Задача 1. Социальная сеть .....	354
Условие .....	354
Моделирование в виде графа .....	355
Решение 1. BFS .....	358
Система непересекающихся множеств .....	363
Решение 2. Система непересекающихся множеств .....	367
Оптимизация 1. Объединение по размеру .....	370
Оптимизация 2. Сжатие пути .....	374
Система непересекающихся множеств .....	377
Три требования к связям .....	377
Применение системы непересекающихся множеств .....	378
Оптимизации .....	378
Задача 2. Друзья и враги .....	378
Условие .....	379
Аугментация: враги .....	380
Функция main .....	385
Поиск и объединение .....	386
SetFriends и SetEnemies .....	387
AreFriends и AreEnemies .....	389

Задача 3. Уборка комнаты .....	390
Условие .....	390
Равнозначные ящики .....	391
Функция <code>main</code> .....	397
Поиск и объединение .....	398
Выводы .....	399
Примечания .....	400
<b>Послесловие</b> .....	401
<b>Приложение А. Время выполнения алгоритма</b> .....	403
Оценка скорости выполнения... и не только .....	403
Нотация «О-большое» .....	405
Линейное время .....	405
Постоянное время .....	406
Дополнительный пример .....	407
Квадратичное время .....	408
«О-большое» в книге .....	409
<b>Приложение Б. Потому что не могу удержаться</b> .....	410
Уникальные снежинки: неявные связные списки .....	410
Страсть к бургерам: реконструкция решения .....	413
Погоня за пешкой: кодирование ходов .....	415
Алгоритм Дейкстры и использование куч .....	417
Мышиный лабиринт: отслеживание с помощью куч .....	418
Мышиный лабиринт: реализация с кучами .....	421
Сжатие сжатия пути .....	423
Шаг 1. Больше никаких тернарных «если» .....	423
Шаг 2. Более понятный оператор присваивания .....	424
Шаг 3. Понятная рекурсия .....	425
<b>Приложение В. Сводка по задачам</b> .....	426

## **ОБ АВТОРЕ**

Даниэль Зингаро — доцент кафедры информатики в Университете Торонто, он неоднократно награждался за выдающиеся успехи в преподавании. Основная область его научных интересов — особенности обучения компьютерным наукам: он исследует, как студенты усваивают (а иногда не усваивают) материал в сфере computer science.

## **О НАУЧНОМ РЕДАКТОРЕ**

Ларри Юли Чжан — доцент кафедры информатики в Университете Торонто. Область его преподавательских и исследовательских интересов включает алгоритмы, структуры данных, операционные системы, компьютерные сети, социальные сети и компьютерное образование. Он состоял в программных комитетах конференций ACM SIGCSE, ACM ITiCSE и WCCCE.

# Предисловие

Начинающему теннисисту сложно послать мяч в нужное место корта (особенно бэкхендом). Только спустя месяцы практики начинает раскрываться притягательность этого вида спорта. Технический арсенал теннисиста постепенно расширяется — осваивается резаный бэкхенд, удар над головой, укороченный удар. Стратегия игрока переходит на более высокий уровень абстракции, где возможны выход к сетке с подачи, выход к сетке после резаного удара, игра на задней линии. Постепенно вырабатывается интуитивное понимание того, какие приемы и стратегии будут более эффективными против разных игроков, поскольку не существует универсального рецепта победы над любым соперником.

Программирование можно сравнить с теннисом. Начинающему программисту сложно доходчиво «объяснить» компьютеру, как нужно реализовать то или иное решение задачи. Но стоит превзойти уровень белого пояса, и работа с задачами начинает приносить удовольствие. Прежде всего, какой подход к решению выбрать? Хотя универсального средства эффективного решения всех задач не существует, есть надежные продвинутые инструменты и стратегии: хеш-таблицы, деревья поиска, рекурсия, мемоизация, динамическое программирование, поиск по графу и т. д. А опытному глазу многие задачи и алгоритмы сами указывают, какие инструменты подходят для них лучше всего. Ваш алгоритм выполняет повторяющийся поиск или минимальные вычисления? Ускорьте его с помощью хеш-таблицы или min-кучи соответственно. Общее решение основной задачи можно выстроить из вторичных решений ее подзадач? Используйте рекурсию! Эти подзадачи пересекаются? Ускорьте алгоритм с помощью мемоизации!

Будь то теннис или программирование, не получится перейти на более высокий уровень без двух вещей: практики и хорошего наставника. В случае программирования оба условия выполнят книга *«Алгоритмы на практике»* и ее автор Даниэль Зингаро. Он познакомит вас со всеми упомянутыми концепциями, но при этом не ограничится простым их перечнем. Под руководством Зингаро вы на примерах практических задач научитесь грамотному применению в работе правильных алгоритмических инструментов. Все это вы освоите с помощью книги, написанной понятным языком и с юмором. Удачи вам в решении задач!

Тим Рафгарден  
Нью-Йорк, NY  
Май 2020



# Благодарности

Работа с ребятами из No Stretch Press была абсолютной идиллией. Они чрезвычайно сосредоточены на издании книг, помогающих читателям учиться. Здесь я нашел единомышленников! Лиз Чедвик (Liz Chadwick) поддержала мою книгу с самого начала (и не поддержала другую, за что я ей признателен). Было удовольствием работать с Алексой Фрид (Alex Freed), которая редактировала рукопись. Она терпелива, добра и не просто исправляла ошибки, а стремилась помочь мне улучшить текст. Я благодарю всех, кто участвовал в процессе создания книги, включая литературного редактора Дэвида Кузенса (David Couzens), выпускающего редактора Кэсси Андрэдис (Kassie Andreadis), креативного директора Дерек Яи (Derek Yee) и дизайнера обложки Роба Гейла (Rob Gale).

Я выражаю признательность Университету Торонто за поддержку моего труда. Спасибо Ларри Чжану (Larry Zhang), научному редактору, за внимательную вычитку рукописи. В течение нескольких лет я преподавал совместно с ним, и именно наше сотрудничество помогло мне выработать правильный образ мышления касательно алгоритмов и того, как обучать их использованию.

Благодарю Тима Рафгардена (Tim Roughgarden) за предисловие к моей книге. Книги и видео Тима представляют собой образцы ясности и наглядности, к которым нам нужно стремиться, рассказывая людям об алгоритмах.

Благодарю моих коллег Яна Варенхольда (Jan Vahrenhold), Махику Путане (Mahika Phutane) и Нааз Сибия (Naaz Sibia) за их рецензии на черновой вариант книги.

Спасибо всем авторам использованных в книге задач, а также участникам соревнований по программированию. Выражаю признательность администраторам

ресурса ДМОJ за их поддержку моей работы. Отдельная благодарность — Тюдору Бриндусу (Tudor Brindus) и Раду Погонариу (Radu Pogonariu) за их помощь в выборе и доработке задач.

Спасибо моим родителям за то, что взяли на себя все, буквально все, и просили меня об одном — учиться.

Я благодарю Дояли, мою спутницу, за заботу и за то, что часть времени, которое мы могли бы провести вместе, она пожертвовала на написание моей книги.

В завершение я хочу выразить признательность всем вам за то, что читаете эту книгу и стремитесь к знаниям.

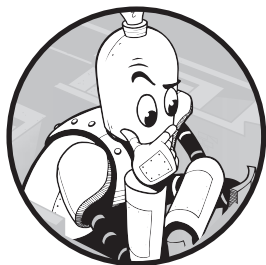
## **От издательства**

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

Мы будем рады узнать ваше мнение!

На веб-сайте издательства [www.piter.com](http://www.piter.com) вы найдете подробную информацию о наших книгах.

# Введение



Я предполагаю, что вы знакомы с такими языками, как Си, С++, Java или Python... и разбираетесь в теме. Тяжело объяснять людям, незнакомым с программированием, почему написание кода для решения задач является настолько захватывающим.

Я также надеюсь, что вы готовы поднять свои навыки программирования на новый уровень. Для меня будет честью помочь вам в этом.

Я мог бы начать с разъяснения всяких изощренных техник, рассказывая о том, какие они полезные, и сравнивая их с другими изощренными техниками. Но я не буду этого делать. Такие знания очень недолго хранятся в памяти, ожидая возможности применения на практике (которая порой так и не появляется).

Вместо этого я на протяжении всей книги буду ставить конкретные трудные задачи. Надеюсь, что вы не сможете справиться с ними сразу, используя знакомые вам подходы. Вы — программист. Вам нужно решать задачи. Пришло время освоить хитрые техники. Книга построена на постановке сложных задач с их последующим решением, при этом вы будете пополнять имеющиеся у вас знания.

Здесь вы не увидите типичных задачек из учебников, не будете искать оптимальный путь умножения последовательности матриц или вычисления чисел Фибоначчи. Я обещаю, что вы не будете решать и головоломку ханойской башни. Существует множество прекрасных учебников, где все это прописано, но я подозреваю, что многих из вас такие головоломки действительно мотивируют.

Мой подход предполагает использование задач, которые вы ранее вряд ли встречали. Каждый год тысячи людей участвуют в соревнованиях по программированию, и для этих соревнований постоянно требуются новые задачи, чтобы оценивать реальные навыки участников, а не способность быстрее всех переделывать старое на новый лад или гуглить решение. Эти задачи удивительны, они базируются на классических подходах, но вносят в них изменения и интересный контекст, подталкивая людей к поиску новых решений. В таких задачах заключается практически безграничный объем знаний по программированию и вычислениям.

Начнем с основ. *Структура данных* — это способ организации данных для быстрого выполнения нужных операций. *Алгоритм* — это последовательность шагов при решении задачи. Иногда можно создать быстрый алгоритм без применения сложных структур данных. В других же случаях правильно подобранная структура может существенно ускорить решение. Моя задача — не сделать из вас участника соревнований по программированию, хотя это было бы хорошим бонусом, а научить работать со структурами данных и алгоритмами на примерах интересно поданных задач спортивного программирования. Пишите мне об успехах в своем обучении. Пишите также, если материалы этой книги заставят вас улыбнуться.

## Онлайн-ресурсы

Примеры листингов, приведенные в книге, вы можете скачать с сайта издательства «Питер» по этому QR-коду:



## Для кого эта книга

Книга предназначена для программистов, которые хотят научиться решать трудные задачи. Вы изучите множество структур данных и алгоритмов, узнаете об их преимуществах, видах задач, где эти алгоритмы и структуры применимы, и способах решения.

Весь код в книге написан на языке программирования Си. Но это не учебник по Си. Если у вас уже есть опыт работы с Си или C++, то смело окунайтесь в материал. Если же вы знакомы только с такими языками, как Java или Python, то я думаю,

что большую часть необходимого вы уясните по ходу чтения. Но все же желательно разобрать некоторые концепции Си заранее или при первой необходимости. В частности, я буду использовать указатели и динамическое выделение памяти. Поэтому, независимо от имеющегося опыта, вам может потребоваться освежить эти темы в памяти.

Лучшая, на мой взгляд, книга по работе с Си — это второе издание *C Programming: A Modern Approach* К. Н. Кинга (K. N. King). Даже если вы хорошо знакомы с Си, все равно рекомендую ее прочесть. Она оказывается отличным помощником, когда особенности этого языка вызывают сложности в понимании кода.

## Язык программирования

Итак, я выбрал для этой книги именно Си, а не один из высокоуровневых языков вроде C++, Java или Python. Далее я вкратце опишу причину такого выбора, а также поясню пару других связанных с Си вещей.

### Почему Си?

Основная причина выбора именно языка Си состоит в том, что я хотел рассказать вам о структурах данных и алгоритмах, начиная с азов. Когда нам понадобится хеш-таблица, мы будем создавать ее сами, не полагаясь на словари, хеш-карты или аналогичные структуры данных других языков. Когда нам неизвестна максимальная длина строки, мы создаем расширяемый массив, поскольку Си не станет выделять память за нас. Я хочу, чтобы вы полностью контролировали происходящее и за кадром ничего не оставалось. В этом-то мне и поможет Си.

Решение задач по программированию на Си служит хорошей основой в случае, если дальше вы захотите работать с C++. Если вы всерьез увлечетесь спортивным программированием, то стоит отметить, что C++ является самым популярным языком среди участников соревнований. Причина этого — его богатая стандартная библиотека и возможности генерировать код, ориентированный на высокую производительность.

### Ключевое слово *Static*

В Си регулярные локальные переменные хранятся в так называемом *стеке вызовов*. При каждом вызове функции часть памяти стека задействуется для сохранения локальных переменных. Далее, когда функция делает возврат, эта память освобождается для последующего использования для других локальных переменных. Однако стек вызовов мал и не подходит для крупных массивов, которые будут встречаться в книге. Поэтому мы будем использовать ключевое слово *static*. Для локальной переменной

оно изменяет продолжительность хранения с динамической на статическую, в результате чего переменная сохраняет свое значение между вызовами функций. Побочным эффектом является то, что такие переменные *не* сохраняются в памяти вместе с регулярными локальными переменными, иначе при завершении функции их значения утрачивались бы. Они помещаются в отдельную область памяти, где им не приходится бороться за место под солнцем с другим содержимым стека вызовов.

При использовании ключевого слова **static** нужно иметь в виду, что такие переменные инициализируются всего один раз! В листинге 1 приводится краткий пример.

**Листинг 1.** Локальная переменная с ключевым словом **static**

```
int f(void) {
    static int x = 5; ❶
    printf("%d\n", x);
    x++;
}

int main(void) {
    f();
    f();
    f();
    return 0;
}
```

Я использовал **static** для локальной переменной **x** ❶. Без него все три раза выводом была бы 5. Тем не менее благодаря **static** мы получаем:

```
5
6
7
```

## Добавление файлов

С целью экономии места я не добавляю в листинги строки **#include**, которые необходимы для запуска программ Си. Все будет в порядке, если вы будете добавлять самостоятельно следующий код:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

## Освобождение памяти

В отличие от Java или Python, Си требует ручной настройки выделяемой памяти. Для выделения памяти служит функция **malloc**, а для освобождения — функция **free**.

Однако я не буду освобождать память по двум причинам. Во-первых, добавление соответствующих функций будет отвлекать от основной обучающей задачи кода.

Во-вторых, эти программы занимают место в памяти недолго: ваш код будет выполняться в нескольких тестовых примерах, и на этом все. Операционная система возвращает всю неосвобожденную память при закрытии программы, так что беспокоиться не о чем, даже если вы будете выполнять код по несколько раз. Конечно же, невыполнение освобождения памяти на практике является безответственным подходом: никто не будет рад программе, которая чем дольше выполняется, тем больше потребляет памяти. Если вы хотите попрактиковаться в освобождении памяти, то можете сами добавлять в приводимые на страницах книги программы вызовы функции `free`.

## Темы

Структуры данных и алгоритмы — слишком обширная область для полного описания в одной книге. Поэтому при отборе тем я опирался на три критерия.

Прежде всего, я выбирал темы с широкой областью применения, чтобы каждую из них можно было использовать при решении не только задач, подобных тем, что приведены в книге, но и многих других. В каждой главе рассматривается не менее двух задач. Первая обычно служит для того, чтобы представить структуру данных или алгоритм вместе с одним из классических случаев применения. Последующие задачи представляют дополнительные возможности рассматриваемой структуры данных или алгоритма. Например, в главе 5 изучается алгоритм Дейкстры. Если вы поищите информацию о нем в Google, то узнаете, что он используется для поиска кратчайших путей. И действительно, в первой задаче главы данный алгоритм применяется именно для этой цели. Однако во второй задаче мы подстраиваем его для поиска не одного, а нескольких кратчайших путей. Надеюсь, что по мере знакомства с каждой главой вы будете все больше узнавать о возможностях, ограничениях и тонкостях каждой техники решения.

Второй критерий — это простота реализации, чтобы не перегружать общий поток повествования. Я хотел, чтобы решение любой задачи укладывалось максимум в 150 строк кода, включая считывание входных данных, само решение и вывод. Поэтому структуры данных и алгоритмы, чья реализация занимает 200 или 300 строк, из практических соображений не рассматривались.

Наконец, я выбирал темы, доказательства корректности которых на мой взгляд являются убедительными и интуитивно понятными. Моя цель — научить вас конкретным структурам данных и алгоритмам, потому что вы наверняка читаете эту книгу, чтобы освоить эффективные подходы к решению задач. При этом я также надеюсь, что вам будет интересно узнать, *почему* та или иная техника работает. Поэтому я тайно преследовал еще одну цель: убедить вас, что рассматриваемые структуры данных или алгоритмы являются корректными. Формальных доказательств или чего-то подобного здесь не будет. Тем не менее если мой секретный

замысел сработает, то наряду с изучением самих структур данных и алгоритмов вы также убедитесь в их корректности. Не стоит ограничиваться чтением кода и восхищаться тем, как он всякий раз волшебным образом срабатывает. Магии здесь нет, и все принципы, заставляющие код работать, находятся в досягаемой для вашего понимания области.

Если у вас возникнет желание выйти за рамки материала книги, то я рекомендую заглянуть в приложение Б, где я добавил кое-какую дополнительную информацию к главам 1, 3, 4, 7 и 8.

Многим читателям будет полезно по ходу чтения книги заниматься дополнительной практикой и изучать сторонние материалы. В разделах «Примечания» в конце каждой главы приводятся ссылки на дополнительные ресурсы. Многие из них содержат дополнительные примеры кода и задачи. Помимо этого есть онлайн-ресурсы, которые содержат структурированный список задач и стратегий по их решению. Из всех известных мне ресурсов наиболее обширным в этом отношении является портал *Methods to Solve* братьев Стивена и Феликса Халимов: <https://cpbook.net/methodstosolve>.

## Сайты с задачами

Каждая выбранная мной задача доступна на сайтах соревнований по программированию. Таких сайтов — множество, и на каждом из них содержатся сотни разных задач. Я старался, чтобы общее число источников было небольшим, но при этом достаточным для гибкого выбора наиболее подходящих задач. Каждый сайт потребует регистрации, и я советую сделать это сразу, чтобы потом не пришлось отвлекаться от решения предложенных в книге задач.

Вот список ресурсов, которые будут использоваться.

Название платформы	Адрес
Codeforces	<a href="https://codeforces.com">codeforces.com</a>
DMOJ	<a href="https://dmoj.ca">dmoj.ca</a>
Kattis	<a href="https://open.kattis.com">open.kattis.com</a>
POJ	<a href="https://poj.org">poj.org</a>
SPOJ	<a href="https://spoj.com">spoj.com</a>
UVA	<a href="https://uva.onlinejudge.org">uva.onlinejudge.org</a>

Описание каждой задачи начинается с указания ресурса, где ее можно найти, и номера/кода задачи.



Одни задачи написаны участниками сообществ, другие взяты из программ известных соревнований. Вот некоторые платформы, откуда были позаимствованы использованные в книге задания:

- Международная олимпиада по информатике (IOI) — престижное соревнование для учащихся старших классов школ. Каждую страну представляют по четыре человека, выступающих в индивидуальном зачете. Программа конкурса рассчитана на два дня, в течение которых участники решают множество задач по программированию.
- Канадский конкурс по программированию (ССС) и канадская олимпиада по программированию (ССО) — ежегодные соревнования для учащихся старших классов школ, организуемые Университетом Ватерлоо. СССР проходит в отдельных школах, после чего победители принимают участие в ССО в Университете Ватерлоо. Победители ССО представляют Канаду на IOI. Когда я был школьником, то неоднократно участвовал в СССР, но ни разу не попал на ССО — даже близко к этому не подошел.
- DWITE — онлайн-соревнование по программированию, проводившееся для подготовки студентов к участию в ежегодных конкурсах. К сожалению, DWITE больше не функционирует, но старые задачи — а они весьма хороши! — по-прежнему доступны.
- ACM East Central North America Regional Programming Contest — ежегодный конкурс для студентов университетов восточной части центрального региона Северной Америки, который проводится ассоциацией ACM. Победители приглашаются на финал международного студенческого соревнования по программированию (ACM International Collegiate Programming Contest, ICPC). В отличие от других упомянутых здесь конкурсов, на которых участники выступают в индивидуальном зачете, это соревнование является командным.
- SAPO — Южно-Африканская олимпиада по программированию. Ежегодное соревнование, которое состоит из трех раундов с растущей сложностью. По результатам конкурса отбираются участники, представляющие ЮАР на IOI.
- COCI — Хорватское открытое соревнование по информатике. Онлайн-конкурс, который проводится несколько раз в год. По его результатам формируется хорватская команда для участия в IOI.
- USACO — Олимпиада по программированию в США. Онлайн-соревнование, организуемое несколько раз в год. Самая сложная его часть — открытый чемпионат. В каждом соревновании участникам предлагаются четыре уровня сложности задач: бронзовый, серебряный, золотой и платиновый. По результатам отбирается команда для участия в IOI.

Полный список источников задач приведен в приложении Б. Когда вы отправляете код задачи, ресурс компилирует вашу программу и проверяет ее на тестовых примерах. Если она проходит все эти тесты за установленное время, то код принимается как верный. Принятые решения обозначают как **АС**. Если же ваша программа не проходит один или более тестов, то она не принимается. Такие программы обозначаются как **WA** (Wrong Answer). Третий вариант результата проверки относится к слишком медленным программам, которые обозначаются **TLE** (Time-Limit Exceeded). Обратите внимание, что **TLE** не означает корректность кода — если время истекает, то оставшиеся тесты не проводятся и возможны необнаруженные ошибки, ведущие к оценке **WA**.

На момент издания книги каждая из представленных в ней окончательных версий решений успешно проходила все тесты за установленное время. В рамках допустимого я стремился сделать код читаемым и отдавал предпочтение ясности, а не скорости, поскольку эта книга посвящена изучению структур данных и алгоритмов, а не выдавливанию максимальной производительности из программы, которая и без того справится со своей задачей.

## Структура описания задачи

Прежде чем решать задачу, нужно четко определиться, что именно от нас требуется. Необходимо точно понимать саму задачу, а также предполагаемый способ считывания входных и генерации выходных данных. Поэтому каждая задача начинается с описания трех ее составляющих:

- **Условие.** Здесь я представляю контекст задачи и ее цель. Очень важно читать условие внимательно, чтобы в точности понимать, какую именно задачу вы решаете. Иногда невнимательное прочтение или неверная интерпретация даже, казалось бы, незначительных нюансов может вести к ошибочным решениям. Например, в одной из задач от нас потребуется купить «не менее» заданного количества яблок. Если же вместо этого вы будете покупать «ровно столько» яблок, то программа провалит некоторые из тестов.
- **Входные данные.** Автор задачи предоставляет тестовые примеры, которые необходимо выполнить, чтобы решение было зачтено правильным. Мы должны считывать из входных данных каждый тестовый пример, чтобы иметь возможность его обработать. Откуда нам знать, сколько там примеров? Какие данные находятся на каждой строке каждого примера? Если в них есть числа, то каков их диапазон? Если там строки, то каков предел их длины? Всю эту информацию я предоставляю в данном разделе.
- **Выходные данные.** Может очень расстроить ситуация, в которой у нас будет программа, производящая верный ответ, но при этом проваливающая те-

стовые кейсы из-за вывода результата в неверном формате. Часть описания задачи, относящаяся к выводу, указывает, как именно он должен быть представлен. Например, в этом разделе будет описываться, сколько строк вывода должно получаться для каждого кейса, что размещать в каждой строке, нужно ли оставлять пустые строки до или после тестовых случаев и т. д. Кроме того, я устанавливаю для каждой задачи ограничение по времени: если программа не делает вывод для всех тестовых кейсов за этот установленный промежуток, она не проходит.

Я менял формулировки условий задач, не меняя их сути.

Для большинства задач книги мы будем считывать входные данные из стандартного ввода (`stdin`) и записывать результат в стандартный вывод (`stdout`) (только в двух задачах в главе 6 `stdin` и `stdout` не потребуются). Это означает, что нужно будет использовать такие функции Си, как `scanf`, `getchar`, `printf` и т. д. без явного открытия и закрытия файлов.

## Задача. Очереди за продуктами

Разберем пример описания задачи. Я буду давать в скобках комментарии, указывая на наиболее важные моменты. Когда условие задачи станет понятным, мы ее решим. В отличие от других задач книги, это можно будет сделать с помощью конструкций и понятий, которые, надеюсь, вам уже знакомы. Если вы справитесь с этой задачей сами или пробежитесь по моему решению без особых затруднений, значит, вы вполне готовы к грядущим испытаниям. Если же у вас возникнут серьезные трудности, то могу порекомендовать для начала еще раз ознакомиться с основами программирования и/или порешать базовые задачки.

Начнем мы с задачи с платформы DMOJ под номером `1kp18c2p1` (найдите ее на сайте, чтобы после получения решения сразу отправить результат на проверку).

### Условие

За продуктами выстроилось  $n$  очередей из известного количества людей. Далее по одному человеку будет прибывать еще  $m$  людей, занимая место в самой короткой очереди (то есть той, где меньше всего людей). Нужно определить количество людей в каждой очереди, к которой присоединяется каждый из  $m$  людей (внимательно обдумайте условие. Дальше идет пример, поэтому, если что-то остается неясным, попробуйте сопоставить его с описанием условия).

Предположим, что всего есть три очереди. В очереди 1 стоит три человека, в очереди 2 стоят двое, в очереди 3 — пятеро. Далее приходят еще четыре человека (прежде

чем читать далее, попробуйте понять, что на этом этапе будет происходить). Первый человек встает в очередь с двумя людьми, то есть очередь 2. Теперь в ней стоят трое. Второй прибывший присоединяется к очереди из трех людей, то есть к 1-й или 2-й, пусть будет 1-я. Теперь в очереди 1 стоят четыре человека. Третий пришедший присоединяется к очереди 2, где стоят трое, и ее размер увеличивается до 4. Последний подошедший встает в очередь с четырьмя людьми, выбирая между очередями 1 и 2. Пусть он выберет 1-ю, и в ней теперь будет пять человек.

### Входные данные

Входные данные содержат один тестовый пример. В первой строке находятся два положительных целых числа:  $n$  — количество очередей;  $m$  — количество людей. Их максимальное значение — 100. Вторая строка содержит  $n$  положительных целых чисел, описывающих количество людей в каждой очереди до прибытия новых. Каждое из этих чисел не больше 100.

Вот пример входных данных:

```
3 4
3 2 5
```

(Обратите внимание, что здесь представлен всего один тестовый пример, следовательно, нужно считывать именно две строки входных данных.)

### Выходные данные

Для каждого из  $m$  подошедших людей нужно вывести строку, содержащую количество людей в очереди, к которой они присоединяются.

Рабочий вывод для предложенного примера будет таким:

```
2
3
3
4
```

Время на решение тестового кейса ограничено тремя секундами (учитывая, что нужно обработать не более 100 новых людей для каждого случая, трех секунд вполне достаточно. Для решения не потребуются изощренные структуры данных или алгоритмы).

### Решение

В задачах, где используются структуры данных, которые сложно создавать вручную, можно начинать со считывания входных данных. В других случаях я откладываю

этот код напоследок, поскольку обычно мы тестируем создаваемые функции путем их вызова с образцами значений. Нет нужды заниматься парсингом входных данных до момента, пока мы не будем готовы решить всю задачу.

Ключевые данные для решения — это количество людей в каждой очереди. Для их сохранения вполне подойдет массив, где каждая очередь будет занимать свой индекс. Этот массив я назову `lines`.

Каждый из прибывающих людей присоединяется к самой короткой очереди, поэтому нам потребуется вспомогательная функция, которая будет определять выбор очереди. Эта функция приведена в листинге 2.

#### Листинг 2. Индекс самой короткой очереди

```
int shortest_line_index(int lines[], int n) {
    int j;
    int shortest = 0;
    for (j = 1; j < n; j++)
        if (lines[j] < lines[shortest])
            shortest = j;
    return shortest;
}
```

Теперь, имея массив `lines`, а также значения `n` и `m`, можно решить тестовый пример. Соответствующий код дан в листинге 3.

#### Листинг 3. Решение задачи

```
void solve(int lines[], int n, int m) {
    int i, shortest;
    for (i = 0; i < m; i++) {
        shortest = shortest_line_index(lines, n);
        printf("%d\n", lines[shortest]);
        lines[shortest]++; ❶
    }
}
```

В каждой итерации внешнего цикла `for` мы вызываем вспомогательную функцию для извлечения индекса самой короткой очереди. Затем мы выводим длину этой очереди. Подошедший человек присоединяется именно к ней, поэтому ее размер надо увеличить на один ❶.

Далее осталось только считать входные данные и вызвать `solve`. Это выполняется в листинге 4.

#### Листинг 4. Функция `main`

```
#define MAX_LINES 100

int main(void) {
```

```
int lines[MAX_LINES];
int n, m, i;
scanf("%d%d", &n, &m);
for (i = 0; i < n; i++)
    scanf("%d", &lines[i]);
solve(lines, n, m);
return 0;
}
```

Объединение функций `shortest_line_index`, `solve` и `main` с добавлением в начале обязательных строк `#include` дает законченное решение, которое можно отправлять на проверку. При этом нужно правильно указывать язык программирования: для наших программ следует выбирать GCC, C99, C11 или иное обозначение компилятора для Си.

Если вы хотите протестировать код локально, прежде чем отправлять его на проверку, то это вполне можно сделать. Поскольку наши программы производят чтение из `stdin`, можно запустить программу и ввести тестовые данные вручную. Это приемлемо для небольших примеров, но будет утомительным, если процесс придется повторять либо требуется ввести большой объем данных. Более разумным вариантом будет сохранить входные данные в файле, а затем выполнить из командной строки *перенаправление ввода*, чтобы программа производила чтение из этого файла, а не с клавиатуры. Например, если сохранить тестовый пример для текущей задачи в `food.txt`, а скомпилированная программа будет называться `food`, то попробуйте следующее:

```
$ food < food.txt
```

Такой способ упрощает изменение тестового примера: просто отредактируйте содержимое `food.txt`, а затем запустите программу снова с перенаправлением ввода.

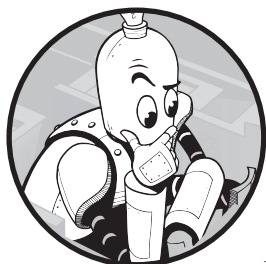
Поздравляю! Вы решили нашу первую задачу. Более того, теперь вы освоили структуру представления, которой будут соответствовать все последующие задачи книги.

## Примечания

Задача «Очереди за продуктами» входила в программу конкурса LKP (Contest 2) на платформе DMOJ.

# 1

## Хеш-таблицы



В этой главе мы решим две задачи, опираясь на средства эффективного поиска. В первой из них нужно будет проверить идентичность снежинок. Во второй мы будем определять, какие из слов являются сложносоставными.

Вы увидите, что некоторые корректные подходы к решению оказываются недостаточно быстрыми. Для существенного повышения скорости вычислений мы используем структуру данных под названием «хеш-таблица», которую также изучим в деталях.

В завершение главы рассмотрим третью задачу: определим, сколькими способами можно удалить букву из слова, чтобы получить другое слово. В ней будут показаны риски необдуманного использования рассмотренной структуры данных, ведь при освоении чего-то нового всегда хочется сразу начать применять это везде!

### Задача 1. Уникальные снежинки

Рассмотрим задачу под номером `sso07p2` с сайта DMOJ.

#### Условие

Дана коллекция снежинок, нужно определить, содержит ли она идентичные снежинки.

Каждая снежинка описывается шестью целыми числами, каждое из которых характеризует длину одного ее луча. Вот пример:

3, 9, 15, 2, 1, 10

При этом значения длин лучей могут повторяться, скажем, вот так:

8, 4, 8, 9, 2, 8

Но как понять, являются ли две снежинки идентичными? Рассмотрим несколько примеров.

Сначала сравним две снежинки:

1, 2, 3, 4, 5, 6

и

1, 2, 3, 4, 5, 6

Очевидно, что они идентичны, потому что числа одной совпадают с числами другой, находящимися в соответствующих позициях.

А вот второй пример:

1, 2, 3, 4, 5, 6

и

4, 5, 6, 1, 2, 3

Эти снежинки тоже идентичны, так как если начать с 1 во второй снежинке и продвигаться вправо, то сперва идут значения 1, 2, 3, после чего происходит возврат в начало и последовательность продолжается значениями 4, 5, 6. Оба этих сегмента вместе формируют снежинку, идентичную первой.

Каждую снежинку можно представить как круг. Тогда два предложенных здесь образца будут идентичными, потому что при правильном выборе начальной точки сопоставления во второй снежинке и следовании по часовой стрелке мы получаем первую снежинку.

Попробуем пример посложнее:

1, 2, 3, 4, 5, 6

и

3, 2, 1, 6, 5, 4



На основе только что рассмотренных вариантов можно сделать вывод, что эти снежинки не идентичны. Если начать с 1 во второй снежинке и продвигаться по кругу вправо до возвращения к точке отсчета, то мы получим 1, 6, 5, 4, 3, 2. Эта форма далека от 1, 2, 3, 4, 5, 6 первой снежинки.

Тем не менее если начать с 1 во второй снежинке и перемещаться не вправо, а влево, тогда мы получим 1, 2, 3, 4, 5, 6. Перемещение влево от 1 дает 1, 2, 3, после чего происходит переход к правому краю и дальнейшее продвижение по значениям 4, 5, 6.

Таким образом, две снежинки считаются идентичными и в случае, если числа совпадают при продвижении влево от точки отсчета.

Обобщая сказанное, можно заключить, что две снежинки идентичны, если они описываются одинаковыми последовательностями чисел либо если сходство обнаруживается при перемещении по этим последовательностям влево или вправо.

### Входные данные

Первая строка входных данных является целым числом  $n$ , описывающим количество сравниваемых снежинок. Значение  $n$  будет находиться в диапазоне от 1 до 100 000. Каждая из следующих  $n$  строк характеризует одну снежинку: содержит шесть целых чисел, каждое из которых может иметь значение в диапазоне от 0 до 10 000 000.

### Выходные данные

На выходе должна быть получена одна текстовая строка:

- Если идентичных снежинок не обнаружено, следует вывести:  
No two snowflakes are alike (нет одинаковых снежинок).
- Если есть хотя бы две идентичные снежинки, следует вывести:  
Twin snowflakes found (найжены одинаковые снежинки).

Время выполнения вычислений ограничено двумя секундами.

### Упрощаем задачу

Одна из универсальных стратегий для решения задач по программированию состоит в проработке их упрощенных версий. Давайте немного разомнемся, снизив сложность нашей задачи.

Предположим, что мы работаем не со снежинками, состоящими из множества целых чисел, а с отдельными целыми числами. У нас есть коллекция, и нужно узнать, содержит ли она одинаковые числа. Проверить одинаковость двух целых чисел в Си можно с помощью оператора `==`. Мы будем тестировать все варианты пар чисел, и если найдем одинаковые числа хотя бы в одной паре, то остановимся и выведем:

```
Twin integers found (найлены одинаковые числа).
```

Если мы не обнаружим одинаковых чисел, то вывод будет таким:

```
No two integers are alike (нет одинаковых чисел).
```

Теперь создадим функцию `identify_identical` с двумя вложенными циклами для сравнения пар целых чисел, как показано в листинге 1.1.

#### Листинг 1.1. Поиск одинаковых целых чисел

```
void identify_identical(int values[], int n) {
    int i, j;
    for (i = 0; i < n; i++) {
        for (j = i+1; j < n; j++) {❶
            if (values[i] == values[j]) {
                printf("Twin integers found.\n");
                return;
            }
        }
    }
    printf("No two integers are alike.\n");
}
```

Мы передаем числа в функцию через массив `values`. Также мы передаем `n`, то есть количество чисел в массиве.

Обратите внимание, что мы начинаем внутренний цикл с `i + 1`, а не с `0` ❶. Если начать с `0`, то `j` будет равняться `i` и мы будем сравнивать элемент с самим собой, получая ложноположительный результат.

Протестируем `identify_identical` с помощью небольшой функции `main`:

```
int main(void) {
    int a[5] = {1, 2, 3, 1, 5};
    identify_identical(a, 5);
    return 0;
}
```

При выполнении кода вывод покажет, что функция определила совпадающую пару единиц. В дальнейшем я ограничусь минимумом тестов, но при этом важно, чтобы вы сами экспериментировали.

## Решение основной задачи

Теперь попробуем изменить функцию `identify_identical` для решения задачи со снежинками. В код нужно будет внести два изменения.

1. Нужно работать не с одним, а с шестью целыми числами сразу. Для сравнения хорошо подойдет двумерный массив, в котором каждая строка будет снежинкой с шестью столбцами (по одному для каждого элемента).
2. Как мы выяснили, снежинки могут оказаться одинаковыми в трех случаях. К сожалению, это подразумевает, что для их сравнения уже не получится использовать `==`. Необходимо учесть возможности «перемещения вправо» и «перемещения влево» (не говоря уже о том, что `==` в Си для сравнения массивов не используется). Так что основным изменением имеющегося алгоритма станет правильное сопоставление снежинок.

Для начала напишем две вспомогательные функции: одну для проверки «перемещения вправо» и вторую для проверки «перемещения влево». Каждая из них будет получать параметры двух снежинок и стартовую точку обхода второй снежинки.

### Проверка вправо

Заголовок функции для `identical_right`:

```
int identical_right(int snow1[], int snow2[], int start)
```

Чтобы определить, совпадают ли снежинки при «передвижении вправо», можно просканировать `snow1` с индекса `0` и `snow2` с индекса `start`. В случае обнаружения разногласия между сопоставляемыми элементами будет возвращаться `0`, указывая, что снежинки не идентичны. Если же все сопоставляемые элементы совпадут, возвращаться будет `1`. Таким образом, `0` означает `false`, а `1` — `true`.

В листинге 1.2 приводится первый вариант кода для этой функции.

**Листинг 1.2.** Определение идентичности снежинок перемещением вправо (с ошибкой!)

```
int identical_right(int snow1[], int snow2[],
                   int start) { //ошибка!
    int offset;
    for (offset = 0; offset < 6; offset++) {
        if (snow1[offset] != snow2[start + offset]) ❶
            return 0;
    }
    return 1;
}
```

К сожалению, этот код не работает нужным образом. Проблема в выражении `start + offset` ❶. Если `start = 4` и `offset = 3`, тогда `start + offset = 7` и мы обращаемся к элементу `snow2[7]`.

Этот код не учитывает, что нам нужно перейти к левой стороне `snow2`. Если код может использовать ведущий к ошибке индекс 6 или выше, то индекс нужно сбрасывать вычитанием шестерки. Это позволит продолжить с индекса 0 вместо индекса 6, индекса 1 вместо 7 и т. д. Попробуем исправить ошибку в листинге 1.3.

**Листинг 1.3.** Определение идентичности снежинок перемещением вправо

```
int identical_right(int snow1[], int snow2[],
                  int start) {
    int offset, snow2_index;
    for (offset = 0; offset < 6; offset++) {
        snow2_index = start + offset;
        if (snow2_index >= 6)
            snow2_index = snow2_index - 6;
        if (snow1[offset] != snow2[snow2_index])
            return 0;
    }
    return 1;
}
```

Такой вариант работает, но его можно улучшить. Одно из возможных изменений — это использование `%`, оператора вычисления остатка от деления (`mod`). Он вычисляет остаток, то есть `x % y` возвращает остаток от целочисленного деления `x` на `y`. Например, `6 % 3` будет равно нулю, так как остатка от операции деления шести на три не будет. Вычисление `6 % 4` даст 2, поскольку в остатке при делении шести на четыре будет два.

Оператор `mod` можно применить здесь для более простой реализации перехода к началу последовательности чисел. Заметьте, что `0 % 6` даст ноль, `1 % 6` даст один, ..., `5 % 6` даст пять. Каждое из этих чисел меньше шести, в связи с чем будет само являться остатком от деления на шесть. Числа от нуля до пяти соответствуют действительным индексам `snow`, поэтому хорошо, что `%` их не меняет. А для индекса 6 операция `6 % 6` даст ноль: шесть на шесть делится без остатка, перенося нас к началу последовательности чисел снежинки.

Обновим функцию `identical_right` с использованием оператора `%`. В листинге 1.4 показан ее доработанный вариант.

**Листинг 1.4.** Определение идентичности снежинок перемещением вправо с вычислением остатка

```
int identical_right(int snow1[], int snow2[], int start) {
    int offset;
    for (offset = 0; offset < 6; offset++) {
```

```
    if (snow1[offset] != snow2[(start + offset) % 6])
        return 0;
}
return 1;
}
```

Использовать прием с получением остатка или нет — дело ваше. Он сокращает одну строку кода и является шаблоном, который знаком многим программистам. Тем не менее применить его не всегда возможно, даже для задач, требующих сброса индекса, — например, `identical_left`. Как раз сейчас мы к этому и перейдем.

### Проверка влево

Функция `identical_left` очень похожа на `identical_right`, но здесь нам нужно сначала перемещаться влево, а затем переходить к правой стороне. При обходе снежинки вправо мы не должны были использовать индекс 6 и выше. На этот раз нужно избегать обращения к индексу -1 и ниже.

К сожалению, в данном случае решение с получением остатка не подойдет. В Си  $-1 / 6$  дает ноль, оставляя остаток -1, то есть  $-1 \% 6$  будет -1. Нам же нужно, чтобы операция  $-1 \% 6$  давала пять.

В листинге 1.5 приведен код функции `identical_left`.

#### Листинг 1.5. Определение идентичности снежинок перемещением влево

```
int identical_left(int snow1[], int snow2[], int start) {
    int offset, snow2_index;
    for (offset = 0; offset < 6; offset++) {
        snow2_index = start - offset;
        if (snow2_index < 0)
            snow2_index = snow2_index + 6;
        if (snow1[offset] != snow2[snow2_index])
            return 0;
    }
    return 1;
}
```

Обратите внимание на сходство между этой функцией и функцией из листинга 1.3. Все, что мы изменили, — это выполнили вычитание смещения вместо его добавления и изменили граничную проверку с 6 на -1.

### Объединение функций проверки

Используя вспомогательные функции `identical_right` и `identical_left`, напомним функцию `are_identical`, которая будет сообщать, идентичны две снежинки или нет. В листинге 1.6 приведен код для этой функции. Мы проводим сравнение

снежинок с перемещениями вправо и влево для каждой возможной стартовой точки в snow2.

**Листинг 1.6.** Определение идентичности снежинок

```
int are_identical(int snow1[], int snow2[]) {
    int start;
    for (start = 0; start < 6; start++) {
        if (identical_right(snow1, snow2, start)) ❶
            return 1;
        if (identical_left(snow1, snow2, start)) ❷
            return 1;
    }
    return 0;
}
```

Вначале мы сравниваем snow1 и snow2, перемещаясь вправо по snow2 ❶. В случае их идентичности возвращается 1 (true). Далее идет проверка перемещением влево ❷.

Здесь есть смысл приостановиться и протестировать функцию are\_identical на примере нескольких пар снежинок. Выполните это самостоятельно, прежде чем продолжать.

## Решение 1: последовательное сравнение

Когда нам нужно сравнить две снежинки, мы применяем вместо оператора == функцию are\_identical. Теперь их сопоставление стало таким же простым, как сравнение двух целых чисел.

Давайте перепишем функцию identify\_identical из листинга 1.1 так, чтобы она использовала новую are\_identical из листинга 1.6. Мы будем сравнивать пары снежинок и получать одно из двух сообщений, в зависимости от результата проверки их идентичности. Код приведен в листинге 1.7.

**Листинг 1.7.** Поиск идентичных снежинок

```
void identify_identical(int snowflakes[][6], int n) {
    int i, j;
    for (i = 0; i < n; i++) {
        for (j = i+1; j < n; j++) {
            if (are_identical(snowflakes[i], snowflakes[j])) {
                printf("Twin snowflakes found.\n");
                return;
            }
        }
    }
    printf("No two snowflakes are alike.\n");
}
```

Функция `identify_identical` для снежинок почти полностью совпадает с ее версией для целых чисел из листинга 1.1. Все, что мы сделали, — это заменили `==` на функцию, сравнивающую снежинки.

### Считывание входных данных

Текущее решение пока не закончено. Нужно еще написать код для считывания снежинок из стандартного потока ввода. Сперва вернемся к описанию задачи в начале главы. Нужно считать строку с целым числом  $n$ , указывающим общее число снежинок, после чего считать каждую из  $n$  строк как отдельную снежинку.

В листинге 1.8 приведена функция `main`, которая обрабатывает входные данные и затем вызывает `identify_identical` (листинг 1.7).

#### Листинг 1.8. Функция `main` для Решения 1

```
#define SIZE 100000

int main(void) {
    static int snowflakes[SIZE][6]; ❶
    int n, i, j;
    scanf("%d", &n);
    for (i = 0; i < n; i++)
        for (j = 0; j < 6; j++)
            scanf("%d", &snowflakes[i][j]);
    identify_identical(snowflakes, n);
    return 0;
}
```

Обратите внимание, что массив `snowflakes` теперь стал статическим ❶. Дело в том, что он огромен, и, если не сделать его статическим, размер необходимого пространства просто превысит объем доступной для данной функции памяти. С помощью ключевого слова `static` массив помещается в отдельную область памяти, где его размер уже не вызовет сложностей. Тем не менее эту возможность нужно использовать осторожно. Стандартные локальные переменные инициализируются при каждом вызове функции, а статические сохраняют значение, полученное от предыдущего вызова функции (см. «Ключевое слово `static`» на с. 21).

Также заметьте, что созданный массив может содержать до 100000 снежинок ❶. Вас может обеспокоить такая растрата памяти. Что, если на входе окажется всего несколько снежинок? В конкурсных задачах считается нормальным жестко прописывать требования памяти для максимального количества входных данных, так как ваше решение наверняка будут проверять методом стресс-тестирования.

Концовка функции проста. Мы считываем количество снежинок с помощью `scanf` и используем это число для управления количеством итераций цикла `for`. В каждой

итерации перебор происходит шесть раз, по разу для каждого целого числа. Далее вызывается `identify_identical`, которая выводит результат.

Объединив функцию `main` с другими ранее написанными функциями, мы получаем завершенную программу, которую можно отправлять на проверку. Попробуйте... и вы получите ошибку «Time-Limit Exceeded» (превышен лимит времени). Похоже, здесь еще есть над чем поработать.

## Выявление проблемы

Наше первое решение оказалось слишком медленным, и мы получили ошибку «Time-Limit Exceeded». Проблема кроется в двух вложенных циклах `for`, которые сравнивают все снежинки между собой. В результате, когда число  $n$  оказывается большим, выполняется огромное число операций сравнения.

Давайте определим, сколько же таких операций выполняет наша программа. Поскольку мы сравниваем каждую пару снежинок, можно переформулировать этот вопрос в отношении именно пар. Например, если даны четыре снежинки 1, 2, 3 и 4, то наша схема выполняет шесть сравнений: снежинок 1 и 2, 1 и 3, 1 и 4, 2 и 3, 2 и 4, а также 3 и 4. Каждая пара формируется выбором одной из  $n$  снежинок в качестве первой и одной из оставшихся  $n - 1$  снежинок в качестве второй.

Для каждого из  $n$  решений для первой снежинки есть  $n - 1$  решений для второй. В общей сложности это дает  $n(n - 1)$  решений. Но формула  $n(n - 1)$  удваивает фактическое количество сравнений снежинок, то есть, к примеру, включает и сравнение 1 с 2, и сравнение 2 с 1. Наше же решение сравнивает их только один раз, поэтому мы можем добавить в формулу делитель 2, получив  $n(n - 1)/2$  сравнений для  $n$  снежинок.

На первый взгляд такое вычисление не выглядит медленным, но давайте подставим в  $n(n - 1)/2$  разные значения  $n$ . Если взять 10, получится  $10(9)/2 = 45$ . С выполнением 45 сравнений любой компьютер справится за считанные миллисекунды. Что насчет 100? Теперь получается 4950: тоже никаких проблем. Выходит, что для небольших значений  $n$  все работает хорошо, но условие задачи говорит, что количество снежинок может достигать 100 000. Давайте подставим в формулу  $n(n - 1)/2$  это значение. Теперь получается 4 999 950 000 операций сравнения. Если выполнить тест для 100 000 снежинок на обычном ноутбуке, то вычисление может занять до четырех минут. А по условиям задачи необходимо уложиться в 2 секунды. В качестве ориентира можно считать, что современный компьютер способен выполнять в секунду около 30 миллионов операций. Это значит, что попытка уложить 4 миллиарда сравнений в 2 секунды, безусловно, обречена на провал.

Если раскрыть скобки в  $n(n - 1)/2$ , получится  $n^2/2 - n/2$ . Самая большая степень здесь — это квадрат. Поэтому создатели алгоритма и назвали его  $O(n^2)$ , или



алгоритмом с *квадратичным временем*.  $O(n^2)$  произносится как «О-большое от  $n$  в квадрате» и является показателем скорости, с которой количество работы растет по отношению к размеру задачи. Краткая информация об «О-большом» приведена в приложении А.

Нам приходится проводить так много сравнений, потому что идентичные снежинки могут находиться в любых частях массива. Если найти способ собирать похожие снежинки в группы, то можно будет быстрее проводить сравнение. Для группировки похожих снежинок можно попробовать отсортировать массив.

### Сортировка снежинок

В Си есть библиотечная функция `qsort`, позволяющая легко упорядочить массив. Главное, что нам потребуется, — это функция сравнения, которая получает ссылки на два сортируемых элемента и возвращает отрицательное целое число, если первый элемент меньше второго, 0, если они равны, и положительное целое число, если первый больше второго. Можно использовать `are_identical` для определения, являются ли две снежинки одинаковыми, и если да, то возвращать 0.

Как при этом определить, что одна снежинка больше или меньше другой? Нужно просто принять соответствующее правило. Например, можно установить, что «меньшей» снежинкой считается та, чей первый отличающийся элемент меньше, чем соответствующий элемент другой снежинки. Это реализуется в листинге 1.9.

#### Листинг 1.9. Функция сравнения для упорядочивания

```
int compare(const void *first, const void *second) {
    int i;
    const int *snowflake1 = first;
    const int *snowflake2 = second;
    if (are_identical(snowflake1, snowflake2))
        return 0;
    for (i = 0; i < 6; i++)
        if (snowflake1[i] < snowflake2[i])
            return -1;
    return 1;
}
```

К сожалению, упорядочивание таким образом не поможет решить проблему. Ниже приведен тестовый пример с четырьмя снежинками, который наверняка будет решен с ошибкой:

```
4
3 4 5 6 1 2
2 3 4 5 6 7
4 5 6 7 8 9
1 2 3 4 5 6
```

Первая и четвертая снежинки идентичны, но выводится сообщение «No two snowflakes are alike». Что же не так?

Вот два факта, которые `qsort` выявит в процессе выполнения:

1. Снежинка 4 меньше снежинки 2.
2. Снежинка 2 меньше снежинки 1.

Исходя из этого, `qsort` заключает, что снежинка 4 меньше снежинки 1, и непосредственно их не сравнивает. Здесь она опирается на транзитивное свойство: если  $a$  меньше  $b$  и  $b$  меньше  $c$ , то однозначно  $a$  должно быть меньше  $c$ . Кажется, что наши определения «меньше» и «больше» имеют смысл.

К сожалению, не вполне понятно, как можно определить правила сортировки снежинок, чтобы применить транзитивность, но исключить ошибки, подобные описанной выше. Тем не менее расстраиваться не стоит, так как мы можем реализовать быстрое решение вообще без упорядочивания.

Как правило, при обработке данных группировка схожих значений оказывается эффективной. К тому же хорошие алгоритмы сортировки выполняются быстро — однозначно быстрее  $O(n^2)$ , но в нашем случае сортировку применить не получится.

## **Решение 2: сокращение числа вычислений**

Сравнение всех пар снежинок и попытка их упорядочивания оказались слишком трудоемкими вариантами решения. Поэтому мы постараемся избежать сравнения тех снежинок, различие которых очевидно. Например, дано две снежинки:

1, 2, 3, 4, 5, 6

и

82, 100, 3, 1, 2, 999

Сразу понятно, что они не могут быть идентичными, и тратить время на их сравнение бессмысленно.

Числа во второй снежинке существенно отличаются от чисел первой. Для выявления очевидного отличия снежинок без их прямого сравнения можно предложить сопоставление их первых элементов, так как 1 сильно отличается от 82. Теперь рассмотрим такие снежинки:

3, 1, 2, 999, 82, 100

и

82, 100, 3, 1, 2, 999

Они оказываются идентичными несмотря на то, что 3 сильно отличается от 82.

Быстро проверить, могут ли две снежинки быть идентичными, можно на основе *суммы* их элементов. При сложении значений приведенных выше образцов снежинок для 1, 2, 3, 4, 5, 6 мы получаем значение 21, а для 82, 100, 3, 1, 2, 299 значение 1187. Таким образом, мы говорим, что *код* для первой снежинки равен 21, а для второй — 1187.

Далее мы будем помещать «снежинки 21» в одну корзину, а «снежинки 1187» в другую и никогда не станем сравнивать снежинки из этих корзин между собой. Такой сортировке можно подвергнуть каждую снежинку: суммировать ее элементы, получить код  $x$  и затем сохранить ее вместе с другими снежинками, имеющими код  $x$ .

Конечно же, обнаружение двух снежинок с кодом 21 не будет означать, что они идентичны. Например, 1, 2, 3, 4, 5, 6 и 16, 1, 1, 1, 1, 1 имеют код 21, но при этом очевидно не идентичны.

Это нормально, так как наше правило «суммирования» предназначено для отсеивания снежинок, которые точно не могут быть идентичными. Это позволяет избежать сравнения всех пар — приема, ставшего причиной неэффективности решения 1, — и сопоставлять только те из них, которые не были отфильтрованы.

В решении 1 мы последовательно сохраняли каждую снежинку в массиве: первую в индексе 0, вторую в индексе 1 и т. д. Здесь же стратегия сохранения изменится: расположение снежинок в массиве будет определяться кодами их сумм. Это значит, что для каждой снежинки мы будем вычислять код и использовать его в качестве индекса, определяющего место ее хранения.

Здесь потребуется ответить на два вопроса:

1. Как вычислять код каждой снежинки?
2. Что делать, если коды нескольких снежинок будут совпадать?

Начнем с вычисления кода.

### Вычисление кода суммы

На первый взгляд вычисление кода выглядит простым. Можно просто складывать все числа каждой снежинки:

```
int code(int snowflake[]) {  
    return (snowflake[0] + snowflake[1] + snowflake[2]  
           + snowflake[3] + snowflake[4] + snowflake[5]);  
}
```

Такой вариант отлично подойдет для многих снежинок, таких как 1, 2, 3, 4, 5, 6 и 82, 100, 3, 1, 2, 999. Но рассмотрите снежинку с огромными значениями лучей, например:

1000000, 2000000, 3000000, 4000000, 5000000, 6000000

Здесь значение кода суммы составит 21 000 000. Если мы планируем использовать его в качестве индекса, то потребуются объявить массив размером 21 миллион элементов, что будет вопиющей тратой памяти.

Наша цель — придерживаться массива, имеющего место под 100 000 элементов. Поэтому код снежинки должен соответствовать диапазону между 0 и 99 999 (минимальный и максимальный индексы массива). Один из способов выполнения этого условия — использование уже знакомого нам оператора %. Вычисление остатка неотрицательного целого числа  $x$  даст целое число между 0 и  $x - 1$ . Какой бы ни была сумма длин лучей снежинки, если мы вычислим ее остаток по модулю 100 000, то получим допустимый индекс массива.

Но у этого метода есть и недостаток: взятие остатка приведет к вероятности получения большего числа снежинок с одинаковым кодом. Например, суммы для 1, 1, 1, 1, 1 и 100001, 1, 1, 1, 1, 1 будут разными — 6 и 100006, но остаток по модулю 100 000 в обоих случаях равен 6. Этот риск допустим: мы просто понадеемся, что такие ситуации будут редкими, а при их возникновении выполним парное сравнение.

Листинг 1.10 вычисляет код суммы снежинки и его остаток по модулю.

#### Листинг 1.10. Вычисление кода снежинки

```
#define SIZE 100000

int code(int snowflake[]) {
    return (snowflake[0] + snowflake[1] + snowflake[2]
        + snowflake[3] + snowflake[4] + snowflake[5]) % SIZE;
}
```

#### Коллизии кодов снежинок

В решении 1 для сохранения снежинки в индексе  $i$  массива `snowflakes` мы использовали следующий фрагмент:

```
for (j = 0; j < 6; j++)
    scanf("%d", &snowflakes[i][j]);
```

Этот вариант работал, так как в каждой строке двумерного массива сохранялась только одна снежинка.

Теперь же нам нужно разобраться с коллизией типа 1, 1, 1, 1, 1, 1 и 100001, 1, 1, 1, 1, 1, когда из-за одинакового остатка по модулю две снежинки будут сохраняться

в одном индексе массива. Это значит, что каждый элемент массива будет уже не одной снежинкой, а коллекцией из некоторого числа снежинок.

Для сохранения элементов в одном индексе можно использовать *связный список*, структуру данных, связывающую каждый элемент со следующим. В нашем случае каждый элемент в массиве будет указывать на первую снежинку в связном списке. К остальным снежинкам можно будет обращаться через указатели `next`.

Рассмотрим типичную реализацию связного списка. Каждый узел `snowflake_node` будет содержать снежинку и указатель на следующий компонент-снежинку. Для сбора этих двух компонентов узла мы используем структуру (`struct`). Также мы задействуем ключевое слово `typedef`, что позволит в дальнейшем использовать `snowflake_node` вместо полной инструкции `struct snowflake_node`:

```
typedef struct snowflake_node {
    int snowflake[6];
    struct snowflake_node *next;
} snowflake_node;
```

Нам придется обновить функции `main` и `identify_identical`, так как ранее они использовали массивы.

## Новая функция `main`

Доработанный код `main` представлен в листинге 1.11.

### Листинг 1.11. Функция `main` для решения 2

```
int main(void) {
    static snowflake_node *snowflakes[SIZE] = {NULL}; ❶
    snowflake_node *snow; ❷
    int n, i, j, snowflake_code;
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        snow = malloc(sizeof(snowflake_node)); ❸
        if (snow == NULL) {
            fprintf(stderr, "malloc error\n");
            exit(1);
        }
        for (j = 0; j < 6; j++)
            scanf("%d", &snow->snowflake[j]); ❹
        snowflake_code = code(snow->snowflake); ❺
        snow->next = snowflakes[snowflake_code]; ❻
        snowflakes[snowflake_code] = snow; ❼
    }
    identify_identical(snowflakes);
    //Рекомендуется освободить всю память, выделенную malloc
    return 0;
}
```

Разберем этот код. Прежде всего обратите внимание, что вместо двумерного массива чисел мы теперь используем одномерный массив указателей на узлы снежинок ❶. Мы также объявили переменную `snow` ❷, которая будет указывать на создаваемые узлы снежинок.

Функция `malloc` отвечает за выделение памяти для каждого узла `snowflake_node` ❸. После считывания и сохранения шести чисел снежинки ❹ мы используем `snowflake_code` для сохранения кода ❺, вычисленного с помощью функции из листинга 1.10.

Осталось только добавить снежинку в массив `snowflakes`, что равнозначно добавлению узла в связный список. Для этого мы добавляем снежинку в начало этого списка. Сначала мы направляем указатель `next` на первый узел списка ❻, после чего устанавливаем указатель начала списка на внедренный узел ❼. При этом важен порядок: если поменять эти две строки местами, то мы потеряем доступ к элементам, уже находящимся в списке.

Обратите внимание, что в плане корректности не важно, в какую часть списка добавляется новый узел. Его можно внедрить в начало, конец или любое другое место по нашему желанию. Но нас интересует самый быстрый вариант, который подразумевает добавление в начало, потому что исключает необходимость обхода списка. К примеру, если список будет содержать миллион элементов, то для добавления элемента в его конец придется переходить по указателям `next` миллион раз, что, естественно, сильно замедлит весь процесс.

Проработаем краткий пример с функцией `main`:

```
4
1 2 3 4 5 6
8 3 9 10 15 4
16 1 1 1 1 1
100016 1 1 1 1 1
```

Каждый элемент массива `snowflakes` в начале имеет значение `NULL`, то есть является пустым связным списком. По мере заполнения массива элементы начинают указывать на узлы снежинок. Сложение чисел первой снежинки дает 21, значит, она помещается в индекс 21. Вторая идет в индекс 49. Третья помещается в индекс 21 к первой. Теперь индекс 21 стал связным списком с *двумя* снежинками:

```
16, 1, 1, 1, 1, 1 и 1, 2, 3, 4, 5, 6.
```

А что насчет четвертой снежинки? Она также помещается в индекс 21, и теперь этот связный список расширился до трех снежинок. Кстати, а не было ли среди них одинаковых? Нет, и это подтверждает тот факт, что существование связного

списка с несколькими элементами не является достаточным доказательством наличия идентичных снежинок. Необходимо сравнить содержащиеся в таком списке снежинки, что станет заключительной деталью пазла.

## Новая функция `identify_identical`

Нам нужно, чтобы `identify_identical` выполняла парное сравнение снежинок в каждом связанном списке. Соответствующий код приведен в листинге 1.12.

**Листинг 1.12.** Определение идентичных снежинок в связанном списке

```
void identify_identical(snowflake_node *snowflakes[]) {
    snowflake_node *node1, *node2;
    int i;
    for (i = 0; i < SIZE; i++) {
        node1 = snowflakes[i]; ❶
        while (node1 != NULL) {
            node2 = node1->next; ❷
            while (node2 != NULL) {
                if (are_identical(node1->snowflake, node2->snowflake)) {
                    printf("Twin snowflakes found.\n");
                    return;
                }
                node2 = node2->next;
            }
            node1 = node1->next; ❸
        }
    }
    printf("No two snowflakes are alike.\n");
}
```

Сначала мы помещаем `node1` в первый узел связанного списка ❶. Далее используем переменную `node2`, чтобы шаг за шагом перемещаться вправо от `node1` ❷ вплоть до конца списка. Таким образом первая снежинка в списке сравнивается со всеми остальными. Потом перемещаем `node1` во второй узел ❸ и сравниваем вторую снежинку с каждой снежинкой справа от нее. Этот процесс повторяется, пока `node1` не достигает конца связанного списка.

Этот код похож на функцию `identify_identical` из решения 1 в листинге 1.7, которая выполняла парные сравнения всех снежинок. Новый код реализует такое же сравнение, но только внутри одного связанного списка. Но что, если кто-то придумает тестовый пример, в котором все снежинки попадут в один список? Не упадет ли в таком случае производительность до уровня решения 1?

Проверьте решение 2, и увидите, что в этот раз оно намного более эффективно! В данном случае мы использовали структуру данных, известную как хеш-таблица.

## Хеш-таблицы

*Хеш-таблица* состоит из двух компонентов:

1. Массива, элементы которого расположены в *ячейках*.
2. *Хеш-функции*, получающей объект и возвращающей его код в качестве индекса в массиве.

Возвращаемый хеш-функцией код называется *хеш-кодом*. Этот индекс означает место *хеширования* объекта.

Внимательно изучите код в листингах 1.10 и 1.11. Вы увидите, что они содержат оба указанных компонента. Функция `code`, которая получает снежинку и вычисляет ее код (число между 0 и 99 999), является хеш-функцией. А массив `snowflakes` представляет собой массив ячеек, каждая из которых содержит связный список.

### Проектирование хеш-таблицы

При создании хеш-таблицы можно учитывать множество разных факторов. Разберем три из них.

Первый касается размера. В задаче «Уникальные снежинки» согласно ее условиям размер 100 000 соответствовал максимальному числу снежинок. При этом можно было использовать массив как большего, так и меньшего размера. Меньший массив экономит память. Например, в начальный момент массив из 50 000 элементов содержит вдвое меньше значений `NULL`, чем массив из 100 000 элементов. Однако меньший размер ведет к увеличению количества объектов в одной ячейке. Попадание объектов в одну ячейку называется *коллизией*. При возникновении множества коллизий формируются длинные связные списки. А в идеале эти списки должны быть короткими, чтобы не приходилось подолгу обходить все их элементы. Крупные же массивы менее подвержены проблеме коллизий.

Если говорить в общем, то нужно искать компромисс между памятью и временем выполнения. Сделайте хеш-таблицу маленькой, и количество коллизий угрожающе возрастет. Сделайте ее слишком большой, и проблемой станет излишний расход памяти.

Второй вопрос касается хеш-функции. В нашем примере эта функция складывает числа снежинки и вычисляет остаток их суммы по модулю 100 000. Здесь важно, что в случае идентичных снежинок она гарантирует их размещение в одной ячейке (при этом не идентичные тоже могут попасть в одну ячейку). Поэтому мы можем выполнять поиск идентичных снежинок только внутри связных списков, а не между ними.



При решении задачи с помощью хеш-таблицы хеш-функция должна исходить из четкого критерия идентичности объектов. Если объекты идентичны, то они должны хешироваться в одну ячейку. В случае, когда два объекта должны быть в точности равны, чтобы считаться «идентичными», сопоставлять объект с ячейкой может быть намного сложнее, чем в рассмотренной ситуации со снежинками. Ознакомьтесь с хеш-функцией `oaat` в листинге 1.13 в качестве примера.

**Листинг 1.13.** Запутанная хеш-функция

```
#define hashsize(n) ((unsigned long)1 << (n))
#define hashmask(n) (hashsize(n) - 1)

unsigned long oaat(char *key, unsigned long len,
                  unsigned long bits) {
    unsigned long hash, i;
    for (hash = 0, i = 0; i < len; i++) {
        hash += key[i];
        hash += (hash << 10);
        hash ^= (hash >> 6);
    }
    hash += (hash << 3);
    hash ^= (hash >> 11);
    hash += (hash << 15);
    return hash & hashmask(bits);
}

int main(void) { //пробный вызов oaat
    long snowflake[] = {1, 2, 3, 4, 5, 6};
    //2^17 - это наименьшая степень 2, дающая больше 100000
    unsigned long code = oaat((char *)snowflake,
                             sizeof(snowflake), 17);
    printf("%u\n", code);
    return 0;
}
```

Для вызова `oaat` передаются три параметра:

- **key** — данные, которые нужно хешировать;
- **len** — длина этих данных;
- **bits** — количество бит, которому должен быть равен итоговый хеш-код.

Возведение 2 в степень `bits` дает нам максимальное значение хеш-кода. Например, если выбрать 17, то этим максимальным значением будет  $2^{17} = 131\,072$ .

Как работает `oaat`? Внутри цикла `for` она начинает с добавления текущего байта ключа (`key`). Эта часть аналогична тому, что мы делали, когда складывали числа в снежинке (листинг 1.10). Сдвиги влево и исключающие ИЛИ (OR) используются для поэтапного смешивания ключа. С помощью такого смешивания хеш-функции

реализуют *лавинный эффект*, который означает, что небольшое изменение в битах ключа окажет сильное влияние на его хеш-код. Если вы намеренно не предложите этой хеш-функции «патологические» исходные данные или огромное число ключей, то вероятность коллизий будет мала. Это подчеркивает важный момент: при использовании одной хеш-функции *всегда* существует коллекция данных, которая приведет к изобилию коллизий и, как следствие, к падению производительности. Сложная хеш-функция, наподобие `oaat`, не сможет защитить от этого. Однако если нас не беспокоит возможность злоумышленного ввода, то мы вполне можем обойтись простой хеш-функцией и надеяться, что она справится с распределением данных.

По правде говоря, именно поэтому решение с хеш-таблицей (решение 2) для задачи со снежинками оказалось успешным. Мы использовали хорошую хеш-функцию, которая распределяет множество неидентичных снежинок по разным ячейкам. Поскольку задача защиты от атак перед нами не стояла, мы не беспокоились о том, что злоумышленник изучит код и найдет, как вызвать в нем миллионы коллизий.

Третьим фактором является выбор типа структуры данных для ячеек. В задаче про уникальные снежинки мы использовали связные списки. Подобное их применение называется *методом цепочек*.

В другом подходе, известном как *открытая адресация*, каждая ячейка содержит не более одного элемента и связные списки отсутствуют. Чтобы не возникало коллизий, обход ячеек выполняется до тех пор, пока не будет найдена пустая. В качестве примера предположим, что нужно внедрить объект в ячейку под номером 50, но она уже занята. В этом случае мы пробуем ячейку 51, затем 52, потом 53, останавливаясь на первой свободной. Однако простая последовательность перебора может привести к низкой производительности, если хеш-таблица содержит множество элементов. Поэтому на практике для таких случаев применяются более продвинутые схемы поиска.

Метод цепочек обычно легче реализуется, чем открытая адресация, поэтому мы использовали в задаче со снежинками именно такой подход. Тем не менее открытая адресация имеет свои преимущества, включая экономию памяти за счет исключения узлов связных списков.

## **Зачем использовать хеш-таблицы?**

Применение хеш-таблиц резко ускоряет выполнение задачи со снежинками. На ноутбуке средней производительности тестовый пример со 100 000 элементов выполнялся бы всего две секунды! В хеш-таблице нет необходимости в парном сравнении всех элементов и их сортировке, а требуется лишь небольшая обработка набора связных списков. При небольшом числе коллизий можно ожидать, что

каждый связный список будет содержать всего несколько элементов. В таком случае выполнение всех парных сравнений внутри ячейки будет требовать небольшого и постоянного количества вычислений. В связи с этим мы предполагаем, что хеш-таблица даст решение в *линейном времени* — за  $n$  шагов (в сравнении с формулой  $n(n-1)/2$ , которая использовалась для решения 1). В терминах «О-большого» можно сказать, что мы ожидаем решение  $O(n)$ .

Всякий раз, когда вы замечаете, что при решении задачи выполняется повторяющийся поиск элемента, подумайте о применении хеш-таблицы. Хеш-таблица преобразует медленный перебор элементов массива в быстрый просмотр. Также можно попробовать использовать сортировку элементов массива. В упорядоченном массиве можно применить технику, известную как бинарный поиск (см. главу 6). Тем не менее даже сортировка массива с последующим бинарным поиском не может сравниться по скорости с хеш-таблицей.

## Задача 2. Сложносоставные слова

Перейдем к следующей задаче и рассмотрим для начала простой вариант решения, основанный на медленном поиске. После этого мы добьемся радикального прироста скорости с помощью хеш-таблицы. Здесь поиск решения пойдет быстрее, так как теперь нам известно, что мы ищем.

Рассмотрим задачу под номером 10391 с платформы UV.

### Условие

Дан список слов, в котором каждое слово представлено в виде строки в нижнем регистре. Например, список, содержащий строки `crea`, `create`, `open` и `te`. Поставлена задача определить, какие из них являются *сложносоставными словами*, то есть представляют конкатенацию двух других строк из списка. В приведенном примере данному условию удовлетворит только строка `create`, потому что состоит из `crea` и `te`.

### Входные данные

Входные данные содержат одно слово на строку в алфавитном порядке. Максимальное число строк — 120 000.

### Выходные данные

От нас требуется вывести каждое сложносоставное слово на отдельной строке в алфавитном порядке.

Время решения должно укладываться в три секунды.

## Определение сложносоставных слов

Как определять сложносоставные слова после их считывания?

Представим слово `create`. Это слово может оказаться сложносоставным в пяти случаях, если отдельными являются следующие слова:

- 1) `c` и `reate`;
- 2) `cr` и `eate`;
- 3) `cre` и `ate`;
- 4) `crea` и `te`;
- 5) `creat` и `e`.

В первой итерации нужно просмотреть список в поиске `c` и `reate`. Если оба поиска окажутся успешными, значит, сложносоставное слово найдено. Во второй итерации нужно просмотреть список в поиске `cr` и `eate`. Далее процесс поиска повторяется для каждой из пяти возможностей. Причем все это пока делается только для слова `create`. Нам же потребуется проверять и другие слова, которых может быть вплоть до 120 000. Поиск получится очень обширным, и повторяющийся перебор огромного списка окажется чрезвычайно времязатратным. Но у нас есть возможность использовать хеш-таблицу.

## Решение

Используем хеш-таблицу связанных списков. Конечно же, не обойдется и без хеш-функции.

Мы не станем применять функцию, аналогичную той, что была в задаче со снежинками, потому что это приведет к коллизиям между анаграммами вроде `cat` и `act`. В данном случае слова нужно различать не только по их буквам, но и по расположению этих букв. Естественно, некоторые коллизии неизбежны, но необходимо сделать все возможное для ограничения их числа. В этом нам поможет хеш-функция `oaat` из листинга 1.13.

При этом в решении будут использованы четыре вспомогательных функции.

## Считывание строки

Начнем со вспомогательной функции считывания строки, приведенной в листинге 1.14.

**Листинг 1.14.** Считывание строки

```
/*на основе https://stackoverflow.com/questions/16870485 */
char *read_line(int size) {
    char *str;
    int ch;
    int len = 0;
    str = malloc(size);
    if (str == NULL) {
        fprintf(stderr, "malloc error\n");
        exit(1);
    }
    while ((ch = getchar()) != EOF && (ch != '\n')) { ❶
        str[len++] = ch;
        if (len == size) {
            size = size * 2;
            str = realloc(str, size); ❷
            if (str == NULL) {
                fprintf(stderr, "realloc error\n");
                exit(1);
            }
        }
    }
    str[len] = '\0'; ❸
    return str;
}
```

К сожалению, в условии задачи не указана максимальная длина строки.

Мы не можем жестко задать некоторую длину слова вроде 16 или 100, так как не контролируем входные данные. Следовательно, функция `read_line` получает начальный размер, который, как мы надеемся, будет достаточен. Вызывая эту функцию, мы задаем начальный размер 16, так как он охватывает большую часть английских слов. Функцию `read_line` можно использовать для считывания символов ❶ вплоть до достижения максимальной длины массива. Если массив заполнится, а слово на этом не закончится, то используется функция `realloc`, которая удвоит размер массива ❷, создав дополнительное пространство для оставшихся символов. В конце мы не забываем завершить `str` нулевым знаком ❸, иначе она окажется недопустимой строкой.

**Поиск по хеш-таблице**

Далее переходим к листингу 1.15, где прописана функция, которая будет искать в хеш-таблице заданное слово.

**Листинг 1.15.** Поиск слова

```

#define NUM_BITS 17

typedef struct word_node {
    char **word;
    struct word_node *next;
} word_node;

int in_hash_table(word_node *hash_table[], char *find,
                  unsigned find_len) {
    unsigned word_code;
    word_node *wordptr;
    word_code = oaat(find, find_len, NUM_BITS); ❶
    wordptr = hash_table[word_code]; ❷
    while (wordptr) {
        if ((strlen(*(wordptr->word)) == find_len) && ❸
            (strncmp(*(wordptr->word), find, find_len) == 0))
            return 1;
        wordptr = wordptr->next;
    }
    return 0;
}

```

Функция `in_hash_table` получает хеш-таблицу и слово, которое в ней надо найти. Если оно будет найдено, функция вернет 1; в противном случае 0. Третий параметр, `find_len`, задает количество символов в `find`, составляющих искомое слово. Этот параметр необходим, так как позволяет найти начало строки. Без него мы не будем знать, сколько всего символов нужно сравнить.

Функция вычисляет хеш-код слова ❶ и использует его для нахождения соответствующего связанного списка для поиска ❷. Хеш-таблица содержит указатели на строки, а не сами строки, что объясняет наличие \* в начале выражения `*(wordptr->word)` ❸ (в листинге 1.17 вы увидите, что хеш-таблица содержит именно указатели, избавляя от необходимости сохранять повторяющиеся копии строк).

**Определение сложносоставных слов**

Теперь можно проверить все возможные варианты разделения слова для определения, является ли оно сложносоставным. В листинге 1.16 приводится соответствующий код.

**Листинг 1.16.** Определение сложносоставных слов

```

void identify_compound_words(char *words[],
                             word_node *hash_table[],
                             int total_words) {

    int i, j;
    unsigned len;

```

```

for (i = 0; i < total_words; i++) {❶
    len = strlen(words[i]);
    for (j = 1; j < len; j++) {❷
        if (in_hash_table(hash_table, words[i], j) && ❸
            in_hash_table(hash_table, &words[i][j], len - j)) {
            printf("%s\n", words[i]);
            break; ❹
        }
    }
}
}
}

```

Функция `identify_compound_words` подобна `identify_identical` из задачи про снежинки (листинг 1.12). Для каждого слова ❶ она генерирует все возможные разделения ❷, затем просматривает хеш-таблицу в поиске префикса (части до точки разделения) и суффикса (части после точки разделения). В качестве точки разделения используется `j` ❸. Вначале поиск проводится для первых `j` символов слова `i`. Затем ищется часть слова `i`, начинающаяся с индекса `j` (длина которой равна `len - j`). Если оба поиска оказываются успешными, значит, текущее слово является сложносоставным. Обратите внимание на использование `break` ❹. Без него при наличии нескольких допустимых разделений слово выводилось бы много раз.

Кого-то может удивить использование и хеш-таблицы, и массива `words`. Узлы в хеш-таблице будут указывать на строки в `words`, но зачем здесь две структуры? Разве нельзя просто использовать хеш-таблицу без массива `words`? Причина в том, что нужно выводить слова упорядоченно. Хеш-таблицы не поддерживают сортировку — они просто разбрасывают элементы по разным местам. Упорядочивание найденных сложносоставных слов можно выполнить в качестве шага постобработки, но тогда мы будем делать работу дважды. Поскольку слова во входных данных отсортированы изначально, то, последовательно проходя массив `words`, мы автоматически получаем эффект упорядочивания.

## Функция `main`

Функция `main` приведена в листинге 1.17.

### Листинг 1.17. Функция `main`

```

#define WORD_LENGTH 16

int main(void) {
    static char *words[1 << NUM_BITS] = {NULL}; ❶
    static word_node *hash_table[1 << NUM_BITS] = {NULL}; ❷
    int total = 0;
    char *word;
    word_node *wordptr;
    unsigned length, word_code;
}

```

```

word = read_line(WORD_LENGTH);
while (*word) {
    words[total] = word; ❸
    wordptr = malloc(sizeof(word_node));
    if (wordptr == NULL) {
        fprintf(stderr, "malloc error\n");
        exit(1);
    }
    length = strlen(word);
    word_code = oaat(word, length, NUM_BITS);
    wordptr->word = &words[total];
    wordptr->next = hash_table[word_code]; ❹
    hash_table[word_code] = wordptr; ❺
    word = read_line(WORD_LENGTH);
    total++;
}
identify_compound_words(words, hash_table, total);
return 0;
}

```

Размещенный в начале странный фрагмент кода  $1 \ll \text{NUM\_BITS}$  ❶, ❷ служит для определения размера хеш-таблицы и массива `words`. В листинге 1.15 мы установили значение `NUM_BITS` равным 17;  $1 \ll 17$  является сокращением для вычисления  $2^{17}$ , что дает 131 072. Это наименьшая степень 2, обеспечивающая превышение 120 000 (максимального числа слов для считывания). Хеш-функция `oaat` требует, чтобы хеш-таблица содержала количество элементов, определяемое степенью 2, поэтому размер хеш-таблицы и массива `words` мы устанавливаем равным  $2^{17}$ .

После объявления структур данных можно задействовать вспомогательную функцию для их заполнения. Мы сохраняем каждое слово в массиве `words` ❸, а в хеш-таблице сохраняем указатели на эти слова ❹, ❺. Техника добавления каждого указателя в хеш-таблицу — такая же, как в задаче со снежинками: каждая ячейка является связным списком, и мы добавляем каждый указатель в начало одного из этих списков. По завершении считывания всех слов происходит вызов функции `identify_compound_words`, которая находит и выводит сложносоставные слова.

При решении задачи хеш-таблица и массив `words` работают параллельно, позволяя добиться молниеносной реализации: хеш-таблица дает быстрый поиск, а `words` позволяет обработать слова упорядоченно. Первое предложенное решение без использования хеш-таблицы оказалось бы намного медленнее. Вернемся еще раз к листингу 1.16 и представим, что дано  $n$  слов. С помощью хеш-таблицы каждый поиск ❸ составит небольшое постоянное количество шагов. Без нее каждый такой поиск потребует сканирования массива `words`, на что будет требоваться  $n$  шагов. Как и в задаче с уникальными снежинками, увеличение скорости, получаемое за счет хеш-таблицы, равнозначно улучшению от  $O(n^2)$  до  $O(n)$ .



## Задача 3. Проверка орфографии: удаление буквы

Иногда кажется, что задачу можно решить определенным способом, потому что она похожа на какие-то другие задачи. Рассмотрим пример, в котором использование хеш-таблицы выглядит оправданным, но в дальнейшем становится понятно, что она чрезмерно усложняет решение.

Это задача с Codeforces под номером 39J — «Проверка орфографии» (Spelling Check). Ее можно легко найти через Google по запросу *Codeforces 39J*.

### Условие

В задаче даются две строки (слова), первая из которых имеет на один символ больше, чем вторая. Цель — определить количество способов, которыми можно удалить один символ из первой строки, чтобы получить вторую строку. Например, существует только один способ получить из *favour* слово *favor* — удалить из первой строки *u*. Но есть три способа получить из *abcdxxxef* слово *abcdxxef* — удалить любой из символов *x* в первой строке.

Контекстом для задачи выступает система проверки правописания. Первой строкой может быть *bizarre* (ошибочно набранное слово), а второй *bizarre* (его верный вариант). В этом случае есть два способа исправить ошибку — удалить одну из двух *z* в первом слове. Однако на деле задача является более общей и не связана исключительно со словами английского языка или ошибками правописания.

Решение должно укладываться в две секунды.

### Входные данные

Входные данные — это два «слова», размещенные на двух строках. Каждое слово может содержать до миллиона символов.

### Выходные данные

Если не существует способа удалить символ из первого слова для получения второго, выводом будет *0*. В противном случае выводятся две строки:

- на первой выводится количество способов, которыми можно удалить символ из первого слова для получения второго;
- на второй выводится разделенный пробелами список индексов тех символов первого слова, которые можно удалить для получения второго слова. Задача

требуется начинать индексацию с 1, а не 0 (это не совсем удобно, но мы будем осторожны).

К примеру, для входных данных

```
abcdxxef  
abcdxxef
```

вывод будет таким:

```
3  
5 6 7
```

Значения 5 6 7 — это индексы трех символов x в первом слове, так как отсчет идет от единицы, а не от нуля.

## Размышление о хеш-таблицах

Я довольно долго искал задачи для этой книги. Мне требовалось, чтобы решения были алгоритмически сложными, а условия достаточно простыми и понятными. В случае с рассматриваемой задачей я действительно полагал, что решение с применением хеш-таблицы будет оптимальным.

В задаче 2 про сложносоставные слова в числе входных данных был дан список слов. Это удобно, так как мы просто передавали каждое слово из списка в хеш-таблицу и затем использовали ее для поиска префиксов и суффиксов. Однако в задаче 3 список слов не задан. При первой попытке решить ее я создал хеш-таблицу, вставив в нее каждый возможный префикс (приставку) и суффикс второго (то есть более короткого) слова. Например, для слова *abc* я вставил префиксы *a*, *ab*, *abc* и суффиксы *c* и *bc*. *abc* тоже является суффиксом, но это сочетание символов я уже добавил. Затем я перешел к поочередному рассмотрению всех символов первого слова. Удаление каждого символа равнозначно разделению слова на префикс и суффикс. Теперь мы вернулись к задаче со сложносоставными словами, где в хеш-таблице ведется поиск префикса и суффикса. Если она их содержит, то удаление символа будет одним из способов преобразования первого слова во второе.

Этот способ решения выглядит заманчиво, не так ли? Хотите попробовать? Можете даже использовать часть кода из задачи со сложносоставными словами.

Но следует учесть, что каждая строка в этой задаче может иметь размер до миллиона символов. Мы однозначно не можем хранить все префиксы и суффиксы в хеш-таблице, так как для этого потребуется слишком много памяти. Я поэкспериментировал с указателями в хеш-таблице, которые бы обозначали начало и конец префиксов и суффиксов. Это решает проблему использования памяти, но не освобождает от необходимости сравнивать чрезвычайно длинные строки при

выполнении поиска с помощью хеш-таблицы. В задачах со снежинками и сложно-составными словами элементы таблицы были малы: 6 целых чисел для снежинки и около 16 для очень длинного английского слова. Это ничто. Здесь же ситуация принципиально другая, так как могут встречаться строки длиной до миллиона символов. Сравнение таких строк будет сверхзатратным по времени.

Еще одна проблема — вычисление хеш-кода префиксов и суффиксов. Если вызвать `oaat` для строки длиной 900 000, а затем снова для той же строки с одним дополнительным символом, то работа удваивается, хотя по факту к уже хешированной строке добавляется всего один символ.

Но я был настойчив и верил, что хеш-таблица является лучшим способом решения, поэтому альтернативы не рассматривал. Хотя в этот момент следовало взглянуть на задачу свежим взглядом. Вместо этого я изучил *инкрементные хеш-функции*, которые очень быстро генерируют хеш-код для элемента, похожего на ранее хешированный. Например, если у меня уже есть хеш-код для `abcde`, то вычисление хеш-кода для `abcdef` с помощью инкрементной хеш-функции будет очень быстрым, так как она может опереться на уже сделанную работу, а не начинать все сначала.

Еще одним открытием стало то, что если сравнение огромных слов оказывается чрезмерно затратным, то их вообще не следует сравнивать. Можно надеяться, что хеш-функция достаточно хороша, нам повезет с тестовыми примерами и коллизий не случится. Если мы ищем элемент в хеш-таблице и находим совпадение... что же, будем надеяться, что оно реальное, а не ложноположительное. Сделав такое допущение, можно использовать структуру, которая проще массива хеш-таблицы. В массиве `prefix1` каждый индекс `i` содержит хеш-код для префикса длиной `i` из первой строки. В каждом из трех других массивов то же правило действует для суффиксов первой строки, приставок второй строки и суффиксов второй строки.

Вот код для создания массива `prefix1`:

```
//long long - очень большой целочисленный тип из C99
unsigned long long prefix1[1000001];
prefix1[0] = 0;
for (i = 1; i <= strlen(first_string); i++)
    prefix1[i] = prefix1[i-1] * 39 + first_string[i];
```

Другие массивы можно создать аналогичным образом.

Важно использовать здесь беззнаковые целые числа. В Си переполнение правильно определяется для беззнаковых целых чисел, но не для чисел со знаком. Если слово окажется достаточно длинным, возникает переполнение, что вносит нежелательную неопределенность в результат.

Обратите внимание, насколько легко вычисляется хеш-код для `prefix1[i]` на основе хеш-кода для `prefix1[i-1]`: это просто умножение, сопровождаемое добавлением нового символа. Почему я применил именно умножение на 39 и добавление символа? Дело в том, что такой вариант не вызывал коллизий в тестовых примерах Codeforces. Да, я понимаю, что это неудовлетворительное решение.

Тем не менее беспокоиться не о чем, так как есть способ получше. Для его реализации мы рассмотрим задачу более внимательно.

## Ad hoc-подход

Давайте проанализируем ранее рассмотренный пример:

```
abcdxxef  
abcdxxef
```

Предположим, что мы удаляем `f` из первой строки (индекс 9). Делает ли это ее равной второй строке? Нет; значит, 9 в списке индексов содержаться не будет. У этих строк есть длинный префикс из совпадающих символов. Если быть точными, то это 6 символов: `abcdxx`. Далее они расходятся, одна строка продолжается символом `x`, а другая символом `e`. Если этого не исправить, то добиться равных строк не удастся. Символ `f` находится слишком далеко справа, чтобы его удаление привело к уравниванию строк.

Отсюда можно сделать первый вывод: если длина *самого длинного общего префикса* (в нашем примере это шесть, то есть длина `abcdxx`) равна  $p$ , то подходящим решением будет только удаление символов с индексами  $\leq p + 1$ . В нашем примере нужно рассмотреть удаление `a`, `b`, `c`, `d`, а также первого, второго и третьего `x`. Удаление символов справа от  $p + 1$  не исправит разницу в символах по индексу  $p + 1$ , а значит, и строки равными не сделает.

Обратите внимание, что только некоторые из возможных вариантов удаления реально сработают. Например, удаление `a`, `b`, `c` или `d` из первой строки не даст вторую строку. Подойдет только любое из удалений `x`. Поэтому, установив верхнюю границу рассматриваемых индексов, нужно определить и нижнюю.

Чтобы найти нижнюю границу, рассмотрим удаление из первой строки символа `a`. Станут ли тогда обе строки равными? Нет. Логика рассуждения здесь та же: справа от `a` есть отличающиеся символы, которые нельзя согласовать путем удаления `a`. Если длина *самого длинного общего суффикса* (в текущем примере это четыре, то есть `xxef`) равна  $s$ , то нужно рассматривать только индексы  $\geq n - s$ , где  $n$  является длиной первой строки. В нашем примере следует рассмотреть только индексы  $\geq 5$ . В абзаце выше мы также заключили, что нужно брать в расчет только индексы  $\leq 7$ . Объединяя эти условия, мы видим, что нас интересуют индексы 5, 6 и 7. Именно их удаление из первой строки сделает ее равной второй.

Обобщим. Нужные индексы находятся в диапазоне от  $n - s$  до  $p + 1$ . Известно, что до индекса  $p + 1$  обе строки одинаковы в каждом своем индексе. Также известно, что строки одинаковы после индекса  $n - s$ . Следовательно, при удалении индекса из этого диапазона две строки становятся одинаковыми. Если же данный диапазон окажется пуст, это будет означать отсутствие индексов, удаление которых привело бы к равенству строк, и выводом будет  $\emptyset$ . В противном случае мы используем цикл `for` для перебора индексов и `printf` для создания разделенного пробелами списка. Можно переходить к коду!

### Самый длинный общий префикс

В листинге 1.18 прописана вспомогательная функция для вычисления длины самого длинного общего для обеих строк префикса.

**Листинг 1.18.** Вычисление самого длинного общего префикса

```
int prefix_length(char s1[], char s2[]) {
    int i = 1;
    while (s1[i] == s2[i])
        i++;
    return i - 1;
}
```

Здесь `s1` — это первая строка, а `s2` — вторая. В качестве стартового индекса строк используется 1. Начиная с индекса 1, цикл продолжается, пока символы строк остаются одинаковыми (в случае `abcde` и `abcd` при сопоставлении `e` с нулевым ограничителем в конце `abcd` индекс `i` верно остановится на значении 5). Когда цикл завершится, индекс `i` будет индексом первого не совпавшего символа. Следовательно, длина самого длинного общего префикса будет `i - 1`.

### Самый длинный общий суффикс

Код для вычисления самого длинного общего суффикса приведен в листинге 1.19.

**Листинг 1.19.** Вычисление самого длинного общего суффикса

```
int suffix_length(char s1[], char s2[], int len) {
    int i = len;
    while (i >= 2 && s1[i] == s2[i-1])
        i--;
    return len - i;
}
```

Этот код очень похож на листинг 1.18. Однако на этот раз сравнение происходит справа налево, а не слева направо. По этой причине нам нужен параметр `len`, который указывает длину первой строки. Заключительное сравнение в цикле — `i == 2`. Если взять `i == 1`, то мы обратимся к `s2[0]`, то есть недопустимому элементу строки.

## Функция main

В листинге 1.20 приводится функция main.

**Листинг 1.20.** Функция main

```
#define SIZE 1000000

int main(void) {
    static char s1[SIZE + 2], s2[SIZE + 2]; ❶
    int len, prefix, suffix, total;
    gets(&s1[1]); ❷
    gets(&s2[1]); ❸

    len = strlen(&s1[1]);
    prefix = prefix_length(s1, s2);
    suffix = suffix_length(s1, s2, len);
    total = (prefix + 1) - (len - suffix) + 1; ❹
    if (total < 0) ❺
        total = 0; ❻

    printf("%d\n", total); ❼
    for (int i = 0; i < total; i++) { ❽
        printf("%d", i + len - suffix);
        if (i < total - 1)
            printf(" ");
        else
            printf("\n");
    }
    return 0;
}
```

Выражение `SIZE + 2` задает размер двухсимвольных массивов ❶. Максимальное количество символов равно миллиону, но нам нужен дополнительный элемент для нуль-терминатора. Для этого достаточен один заключительный элемент, потому что индексирование строк начинается с индекса 1, а индекс 0 «пропускается».

Мы считываем первую ❷ и вторую строки ❸. Обратите внимание на передачу указателя на индекс 1 каждой строки: функция `gets` сохраняет символы с индекса 1, а не с индекса 0. После вызова вспомогательных функций производится вычисление количества индексов, которые можно удалить из `s1`, чтобы получить `s2` ❹. Если их число будет отрицательным ❺, то мы устанавливаем для него значение 0 ❻, делая вызов `printf` корректным ❼. Далее с помощью цикла `for` ❽ выводятся искомые индексы. Вывод нужно начать с `len - suffix`, поэтому мы добавляем `len - suffix` к каждому целому числу `i`.

Таким образом, получено решение в линейном времени без сложного кода и хеш-таблицы. Прежде чем применять хеш-таблицу, подумайте, не приведет ли это к излишне громоздкому решению. Действительно ли необходим поиск элементов в массивах или у задачи есть более простые решения?

## Выводы

Хеш-таблица представляет собой структуру, позволяющую организовать данные таким образом, чтобы определенные операции выполнялись очень быстро. Такие таблицы ускоряют поиск конкретного элемента. Для ускорения других операций требуются иные структуры данных. Например, в главе 7 мы познакомимся с кучей — специальной структурой данных, которая может использоваться для нахождения максимального или минимального элемента массива.

Структурирование данных является основным подходом к хранению и управлению информацией. Задачи этой главы помогут вам отличать случаи, в которых оптимально использование хеш-таблиц. При этом хеш-таблицы применимы к широкому спектру задач, выходящих за рамки рассмотренных здесь примеров. Будьте внимательны и старайтесь находить эффективные решения вместо повторяющихся медленных операций поиска.

## Примечания

Задача «Уникальные снежинки» взята из канадской олимпиады по программированию 2007 года (2007 Canadian Computing Olympiad).

Задача «Сложносоставные слова» взята из местного конкурса, проводившегося в Ватерлоо в сентябре 1996 года (1996 Waterloo Local Contest).

Задача «Проверка орфографии» взята из первого командного конкурса для школьников 2010 года (2010 School Team Contest #1), организованного платформой Codeforces. Решение с префиксами-суффиксами, использованное после отказа от решения с хеш-таблицей, позаимствовано из заметки, размещенной на сайте Codeforces (<https://codeforces.com/blog/entry/786>).

Хеш-функция `oaat` (one-at-a-time) придумана Бобом Дженкинсом (Bob Jenkins) (см. <http://burtleburtle.net/bob/hash/doobs.html>).

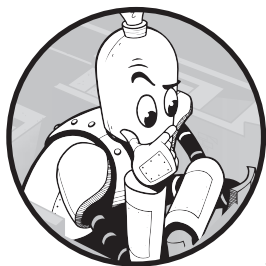
Для лучшего ознакомления с вариантами применения и реализации хеш-таблиц рекомендую почитать работу Тима Рафгардена (Tim Roughgarden) *Algorithms Illuminated (Part 2): Graph Algorithms and Data Structures*<sup>1</sup>.

---

<sup>1</sup> Рафгарден Т. Совершенный алгоритм. Графовые алгоритмы и структуры данных. СПб.: Питер.

# 2

## Деревья и рекурсия



В этой главе будут рассмотрены две задачи, требующие обработки иерархических данных и ответов на возникающие при этом вопросы. Первая — это задача о сборе конфет у соседей в Хэллоуин. Вторая связана с запросами к генеалогическому дереву. Так как естественным способом обработки данных являются циклы, сначала мы попробуем их.

Впрочем, вскоре станет очевидно, что задачи выходят за рамки того, что можно представить с помощью циклов, и подход к решению потребует изменения. А к концу главы у вас сформируется четкое понимание рекурсии — техники, применяемой в случаях, когда общее решение задачи подразумевает ряд решений более мелких подзадач.

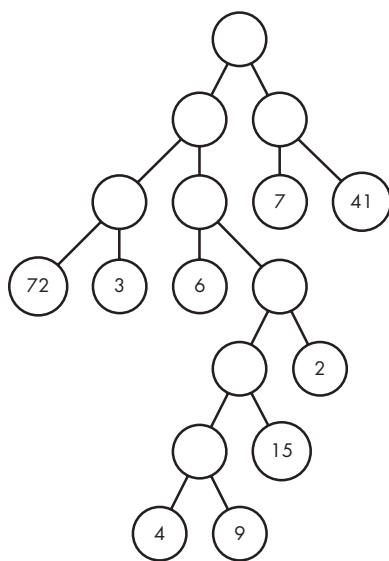
### Задача 1. Трофеи Хэллоуина

Вначале рассмотрим задачу под номером `dwite12c1p4` с сайта DMOJ.

#### Условие

Итак, наступил Хэллоуин, праздник, который начинается с веселых нарядов, сбора конфет по соседям, а заканчивается обычно болью в животе из-за переедания сладостей. По условиям задачи нужно максимально эффективно собрать все конфеты в домах одного квартала. Расположение домов задано, причем выглядит оно весьма необычно (рис. 2.1).





**Рис. 2.1.** Пример плана квартала

Круги с числами на диаграмме — это дома. Каждое число обозначает количество конфет, которое нас ждет в этом доме. Оно может быть одно- или двухзначным. Самый верхний круг является стартовой локацией. Круги без чисел представляют перекрестки, где нужно выбирать, в каком направлении продолжать движение. Линии, соединяющие круги, — это улицы. Перемещение от круга к кругу соответствует прогулке по одной из улиц.

Начнем с планирования перемещения по кварталу. Если от верхнего круга двигаться по правой улице, то вы попадете на перекресток. Если далее снова выбрать правую улицу, то в итоге вы окажетесь в доме, где получите 41 конфету. Затем можно вернуться назад к началу пути. Таким образом, в общей сложности будут пройдены 4 улицы и получена 41 конфета.

Однако целью является собрать *все* конфеты, пройдя при этом минимальное количество улиц. Закончить прогулку можно сразу же по окончании сбора всех трофеев, то есть возвращаться к стартовой локации не нужно.

### Входные данные

Входные данные представлены пятью строками, каждая из которых может содержать не более 255 символов, описывающих квартал.

Как строка может кодировать диаграмму? Этот случай не похож на первую задачу из главы 1, где каждая снежинка характеризовалась всего шестью целыми

числами. Здесь мы имеем круги, соединяющие их линии и числа в некоторых из этих кругов.

Как и в задаче со снежинками, можно все упростить, проигнорировав некоторые нюансы. По этой причине я отложу описание способа представления входных данных на потом. Хотя затравку все же дам: есть весьма компактный способ корректно представить эти диаграммы в виде строк. Как говорится: «Оставайтесь с нами!»

### Выходные данные

Программа должна выводить пять текстовых строк, соответствующих входным. Каждая строка должна содержать два целых числа, разделенных пробелом: количество улиц, пройденных для получения всех конфет, и общее количество собранных конфет.

На расчет всех тестовых примеров дается две секунды.

### Двоичные деревья

На рис. 2.2 показана доработанная версия квартала с рис. 2.1, в которой пустые круги заполнены буквами. Эти буквы в коде использоваться не будут, но являются уникальными обозначениями кругов.

Такая форма представления данных известна как *двоичное дерево*. Оба слова этого выражения имеют важное значение, и далее мы их подробно разберем.

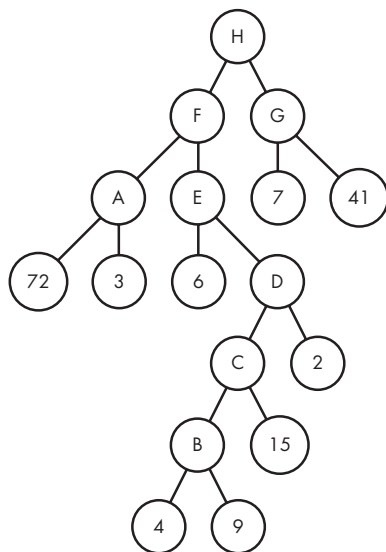


Рис. 2.2. Пример плана квартала с буквенными метками

## Дерево

*Дерево* — это структура, состоящая из *узлов* (кругов) и *ребер* между узлами (линий, представляющих улицы). Верхний узел — круг *H* — называется *корнем*. В качестве синонима узла также нередко используется термин *вершина*. Впрочем, в этой книге я буду называть их узлами.

Между узлами дерева существует связь «родитель — потомок». Например, *H* является родителем *F* и *G*, потому что с ними его соединяют ребра. Также принято говорить, что *F* и *G* являются *дочерними узлами* *H*. Если еще конкретней, то *F* является *левым дочерним узлом*, а *G* *правым дочерним узлом* *H*. Если же у узла нет потомков, то он называется *концевым узлом* или *листом*. В рассматриваемой задаче концевыми узлами являются те, которые содержат конфеты.

Большая часть терминологии, используемой в информатике в отношении деревьев, происходит из генеалогии. Например, *F* и *G* являются *братьями*, потому что имеют общего родителя. *E* является примером *потомка* *H*, потому что достигим при движении вниз от *H*. Потомками (или детьми) также могут называться и дочерние узлы родителя.

*Высота* дерева определяется максимальным количеством ребер, которые можно пройти по нисходящему пути от корневого узла до концевого. Какова высота нашего образца дерева? Например, в нем есть следующий нисходящий путь:  $H \rightarrow G \rightarrow 7$ . На этом пути присутствуют два ребра (от *H* к *G* и от *G* к 7), что дает высоту два. Тем не менее можно найти и намного более длинный маршрут вниз:  $H \rightarrow A \rightarrow E \rightarrow D \rightarrow C \rightarrow B \rightarrow 4$ . На этом пути шесть ребер, значит, высота нашего дерева равна шести. Перепроверьте, нет ли в нем другого более длинного нисходящего пути.

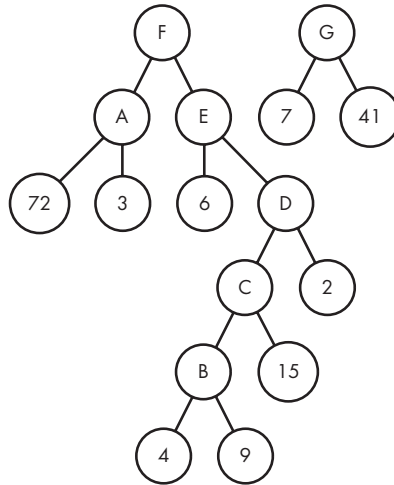
Деревья имеют характерную повторяющуюся структуру, что упрощает их обработку. К примеру, если удалить корневой узел *H* вместе с ребрами от *H* к *F* и от *H* к *G*, то у нас получатся два *поддерева* (рис. 2.3).

Обратите внимание, что каждое из двух поддеревьев само по себе является жизнеспособным: у него есть корень, узлы и ребра, а также необходимая структура. Можно и дальше разделять эти деревья на более мелкие части, каждая из которых также будет представлять самостоятельное дерево. Дерево можно рассматривать как комбинацию более мелких деревьев, каждое из которых также может состоять из еще более мелких деревьев.

## Двоичность

В контексте деревьев термин *двоичный* означает, что каждый узел может иметь не более двух дочерних узлов. Один узел двоичного дерева может не иметь детей либо иметь одного или двоих, но не более. В рассматриваемой задаче условия еще жестче:

каждый узел либо не имеет детей, либо имеет два дочерних узла — узлы с одним потомком отсутствуют. Такая структура называется *полным* двоичным деревом.



**Рис. 2.3.** Дерево, разделенное на два

### Решаем пример

Приступим к решению задачи со сбором конфет на образце дерева. Нам нужно вернуть минимальное количество улиц, которые необходимо пройти для получения всех конфет, а также общее количество конфет. Начнем с последнего, так как посчитать конфеты будет проще.

Для подсчета конфет достаточно сложить все значения из узлов. В рассматриваемом случае получится  $7 + 41 + 72 + 3 + 6 + 2 + 15 + 4 + 9 = 159$ .

А теперь выясним минимальное количество улиц, которые нужно обойти, чтобы все эти конфеты собрать. Имеет ли значение, как мы будем обходить дерево, поскольку в итоге нужно посетить каждый дом? Может быть, самый быстрый путь получится, если не заходить ни в один дом дважды?

Давайте обойдем дерево, посещая сначала левых потомков, а затем правых. При таком подходе у нас получится следующий порядок посещения: Н, F, A, 72, A, 3, A, F, E, 6, E, D, C, B, 4, B, 9, B, C, 15, C, D, 2, D, E, F, H, G, 7, G, 41. Обратите внимание, что конечной точкой оказывается дом 41, а не Н, поскольку возвращаться к корню дерева после сбора всех конфет не требуется. На этом пути мы пройдем 30 ребер. Их количество легко вычислить путем вычитания 1 из общего числа пройденных узлов, которых в нашем случае 31. Неужели обход 30 улиц — это лучшее, на что мы способны?

Если постараться, можно добиться результата получше: самый эффективный маршрут подразумевает обход всего 26 улиц. Уделите время самостоятельному поиску этого пути. Как и в первом случае с 30 улицами, вы будете посещать узлы без домов по несколько раз, а сами дома по одному. Но при этом можно сократить путь на четыре улицы, рационально выбрав *последний* дом маршрута.

## Представление двоичных деревьев

Для создания кода решения требуется представить модель квартала на Си. Вы увидите, что очень удобно преобразовывать строки входных данных, описывающих деревья, в соответствующие структуры, представляющие связи между узлами. В этом разделе мы рассмотрим такие структуры. Пока мы не сможем считывать строки и преобразовывать их в деревья, но напомним код деревьев. Это даст нам основу для решения задачи.

### Определение узлов

При решении задачи с уникальными снежинками мы сохраняли эти снежинки с помощью связанного списка. Каждый узел содержал описание снежинки, а также указатель на следующую снежинку в цепочке:

```
typedef struct snowflake_node {
    int snowflake[6];
    struct snowflake_node *next;
} snowflake_node;
```

Аналогичную структуру можно задействовать для представления двоичного дерева. В деревьях, моделирующих квартал, узлы с домами содержат значения количества конфет, а другие узлы значений не содержат. Для двух видов узлов нам вполне хватит одной структуры. Нужно просто убедиться, что в узлах домов были указаны верные значения конфет. А для узлов без домов инициализировать переменную *candy* не потребуется, поскольку рассматривать их мы все равно не будем.

В итоге у нас получается отправная точка:

```
typedef struct node {
    int candy;
    // ... что еще нужно добавить?
} node;
```

В связанном списке каждый узел указывает на следующий узел в цепочке либо имеет значение NULL. Таким образом, мы можем перемещаться в другие узлы. Но в дереве одного указателя *next* для узла будет недостаточно, так как у неконцевого узла имеются как левый, так и правый потомки. Это значит, что для каждого узла потребуются два указателя, как показано в листинге 2.1.

**Листинг 2.1. Структура узла**

```
typedef struct node {  
    int candy;  
    struct node *left, *right;  
} node;
```

Мы видим, что указатель `parent` сюда не включен. Стоит ли добавить `*parent`, который позволит помимо потомков обращаться еще и к родителю узла? Это может быть полезным для ряда задач, но в данном случае необязательно. Нам потребуется способ перемещаться вверх по дереву (от потомка к родителю), но делать это можно неявно, не следуя открыто по указателям на родителей. Чуть позже мы рассмотрим этот вопрос подробнее.

**Построение дерева**

Создав тип `node`, можно приступить к построению образцов деревьев. Мы начнем с нижней части и будем объединять поддеревья вплоть до достижения корневого узла. Рассмотрим, как это делается, на примере нашего дерева.

Начнем с узлов 4 и 9 в нижнем ярусе. Далее объединим их общим родителем, создав поддерево с корнем B.

Узел 4 описывается следующим кодом:

```
node *four = malloc(sizeof(node));  
four->candy = 4;  
four->left = NULL;  
four->right = NULL;
```

Это узел дома, значит, нужно не забыть указать для него число конфет. Также важно установить его левого и правого потомков как `NULL`. Если этого не сделать, они останутся неинициализированными и будут указывать на незаданную область памяти, что вызовет проблемы при попытке обратиться к ней.

Далее рассмотрим узел 9. Это еще один дом, значит, структура кода будет та же:

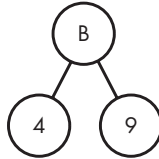
```
node *nine = malloc(sizeof(node));  
nine->candy = 9;  
nine->left = NULL;  
nine->right = NULL;
```

Теперь у нас есть два узла, которые пока что не являются частью дерева. Следующим шагом будет их объединение общим родителем:

```
node *B = malloc(sizeof(node));  
B->left = four;  
B->right = nine;
```

Узел В получил указатель `left` на дом 4 и указатель `right` на дом 9. Переменная `candy` в нем не инициализирована, что вполне нормально, так как узлы без домов не имеют соответствующего значения.

Получившееся поддерево представлено на рис. 2.4.



**Рис. 2.4.** Первые три узла в дереве

Прежде чем создавать поддерево С, необходимо сделать пояснение. Создание узла с домом подразумевает четыре действия: выделение для него памяти, задание числа конфет, а также установку левого и правого потомков как `NULL`. Аналогичным образом создание узла без дома подразумевает выполнение трех действий: выделение узла, указание на левого и правого потомков. Теперь вместо того, чтобы выполнять эти шаги по отдельности, мы зададим их во вспомогательных функциях, приведенных в листинге 2.2.

#### Листинг 2.2. Вспомогательные функции для создания узлов

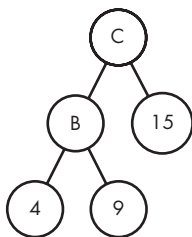
```
node *new_house(int candy) {
    node *house = malloc(sizeof(node));
    if (house == NULL) {
        fprintf(stderr, "malloc error\n");
        exit(1);
    }
    house->candy = candy;
    house->left = NULL;
    house->right = NULL;
    return house;
}

node *new_nonhouse(node *left, node *right) {
    node *nonhouse = malloc(sizeof(node));
    if (nonhouse == NULL) {
        fprintf(stderr, "malloc error\n");
        exit(1);
    }
    nonhouse->left = left;
    nonhouse->right = right;
    return nonhouse;
}
```

Теперь перепишем код для узлов 4, 9 и В с использованием вспомогательных функций и заодно добавим код для узлов 15 и С:

```
node *four = new_house(4);
node *nine = new_house(9);
node *B = new_nonhouse(four, nine);
node *fifteen = new_house(15);
node *C = new_nonhouse(B, fifteen);
```

На рис. 2.5 показано итоговое дерево из пяти узлов.



**Рис. 2.5.** Первые пять узлов в дереве

Обратите внимание, что у С есть левый потомок, являющийся узлом без дома (В), и правый потомок, представленный узлом с домом (fifteen). Функция `new_nonhouse` допускает подобную асимметрию (один потомок без дома, а другой с домом). Поэтому можно комбинировать и сопоставлять узлы с домами и без домов любым нужным нам образом.

На данном этапе у нас получилось поддерево из пяти узлов с корнем С. Теперь можно использовать С для подсчета числа конфет (можно было бы считать и в других узлах, так как при пошаговом построении дерева число конфет остается в памяти. Однако позже мы создадим функцию преобразования строки в дерево, которая будет давать нам только значения для корня дерева, так что не станем хитрить).

Предлагаю небольшое упражнение. Каким будет вывод для строки ниже?

```
printf("%d\n", C->right->candy);
```

Если вы ответили 15, то оказались правы. Мы обратились к правому потомку С, являющемуся узлом с домом fifteen, и получили число конфет.

А в этом случае?

```
printf("%d\n", C->left->right->candy);
```

Здесь выводом будет 9: переходя к левому, а затем к правому дочерним узлам, мы попадаем из С в nine.



А теперь попробуйте такой вариант:

```
printf("%d\n", C->left->left);
```

Упс! На своем ноутбуке я получаю значение **10752944**. Причина в том, что мы выводим значение указателя, а не конфет. С этим нужно быть внимательными.

Ну и наконец, какой вывод ожидать здесь?

```
printf("%d\n", C->candy);
```

Эта инструкция бесполезна. Здесь мы выводим элемент `candy` для узла без дома, но, как нам известно, значения `candy` содержатся только в домах.

Теперь мы готовы к решению задачи. Закончите код для построения дерева и приступим.

## Сбор конфет

Перед нами стоят две основные задачи: вычислить минимальное количество улиц, которые нужно обойти для сбора всех конфет, и посчитать общее количество конфет в дереве. Для каждой задачи мы напишем вспомогательную функцию и начнем с самого простого — вычисления общего числа конфет. Функция в данном случае будет выглядеть так:

```
int tree_candy(node *tree)
```

Эта функция получает указатель на корневой узел и возвращает целое число — общее количество конфет в дереве.

Если мы работаем со связными списками, то можно использовать цикл, как мы делали при решении задачи с уникальными снежинками. Тело этого цикла обрабатывает текущий узел, затем через элемент `next` переходит к следующему узлу. На каждом шаге можно переместиться только в одном направлении — вниз по связному списку. Однако структура двоичных деревьев более сложна. Каждый неконцевой узел имеет левое и правое поддеревья, которые нужно обойти, иначе мы просто пропустим часть дерева.

Чтобы показать обход дерева, вернемся к примеру на рис. 2.2. Какое направление выбрать из узла Н? Можно перейти направо к G, а затем снова направо, собрав 41 конфету. Но мы окажемся в тупике, а несобранных конфет еще много. Вспомните, что узел может хранить указатели только на своих потомков, но не на родителя. Поэтому, оказавшись в узле 41, мы теряем возможность вернуться к G.

Значит, при переходе от Н к G нам нужно сделать запись, что позже необходимо обработать поддерево F, иначе затем мы не сможем вернуться к нему. Аналогично

до перехода из G в 41 надо записать, что необходимо позже обработать поддерево 7. Оказавшись в 41, мы видим, что здесь нет поддеревьев для обработки, а в записях у нас числятся F и 7, к которым нужно вернуться.

Если далее мы выбираем обработку 7, то теперь общее количество конфет составит  $41 + 7 = 48$ . Затем мы обрабатываем поддерево F. Выбор любого варианта перемещения из F оставляет необработанным целое поддерево, значит, это также нужно записать.

Таким образом, если мы используем цикл, то для каждого неконцевого узла нужно сделать две вещи: выбрать одно из его поддеревьев и записать, что второе также ожидает обработки. Выбор поддерева сводится к следованию по указателю `left` или `right` — здесь проблем нет. А вот сохранить информацию о том, что позднее нужно посетить другое поддерево, будет сложнее. Здесь нам понадобится новый инструмент.

### Сохранение ожидающих поддеревьев в стеке

В любой момент у нас может быть несколько ожидающих посещения поддеревьев. При этом нужна возможность добавлять в эту коллекцию очередное поддерево, а также удалять из нее сведения о пройденных поддеревьях.

Управление этим процессом можно организовать с помощью массива. Определим его достаточно большим для хранения необходимого числа ожидающих поддеревьев. Для получения информации о количестве поддеревьев в очереди мы задействуем переменную `highest_used`, которая будет отслеживать верхний индекс массива. Например, если значение `highest_used` равно 2, значит, в индексах 0, 1 и 2 содержатся ссылки на ожидающие поддерева, а остальная часть массива пока не используется. Если же `highest_used` равно 0, значит, используется только индекс 0. Если массив пуст, то `highest_used` равен -1.

Новые элементы получают индекс `highest_used + 1`. Для добавления в любое другое место сначала пришлось бы сдвигать существующие элементы вправо, иначе один из них был бы переписан. Удалять из массива проще всего элемент с индексом `highest_used`. Удаление любого другого элемента требует смещения оставшихся влево для заполнения освободившегося места.

Предположим, что, используя эту схему, мы сначала добавляем ссылку на поддерево F, после чего добавляем ссылку на поддерево 7. Эта операция помещает поддерево F в индекс 0, а поддерево 7 в индекс 1. На данный момент значение `highest_used` равно 1. Теперь, когда мы удаляем элемент из этого массива, какое будет удалено поддерево: F или 7?

Правильный ответ — 7. Как правило, удалению подлежит последний добавленный в массив элемент.

В информатике такой принцип доступа к данным называется «*последним вошел — первым вышел*», *LIFO* (last in, first out). Коллекции данных с доступом по принципу *LIFO* называются *стеками*. Операция добавления элемента в стек называется *push*, а извлечения — *pop*. *Вершину* стека образует элемент, который был добавлен последним и будет извлечен первым.

В окружающей нас жизни стеки встречаются повсеместно. Предположим, у вас есть только что вымытые тарелки, которые вы складываете в шкафчик в стопку. Последняя добавленная (*push*) на полку тарелка будет находиться на вершине стека, и она же будет первой, которую вы извлечете (*pop*) из шкафчика, когда соберетесь поесть. Это и есть схема *LIFO*.

Стек также лежит в основе функции отмены последнего действия в текстовом процессоре наподобие *Word*. Предположим, что вы вводите одно слово, затем второе, потом третье. Теперь вы жмете *undo*. Третье слово удаляется, так как его вы ввели последним.

## Реализация стека

Перейдем к реализации. Для начала упакуем массив и *highest\_used* в единую структуру. Так мы сохраним переменные вместе и получим возможность создавать столько стеков, сколько захотим (для подсчета конфет нам нужен только один стек, но соответствующий код вы сможете использовать и в задачах, требующих нескольких стеков). Вот описание структуры:

```
#define SIZE 255

typedef struct stack {
    node * values[SIZE];
    int highest_used;
} stack;
```

Напомню, что каждая строчка ввода может содержать не более 255 символов. Каждый символ будет представлять не более одного узла. Следовательно, каждое дерево, с которым мы будем работать, будет иметь не более 255 узлов. Именно поэтому наш массив *values* имеет место под 255 элементов. Обратите также внимание, что каждый элемент в *values* имеет тип *node \**, являющийся указателем на *node*. Можно сохранять узлы и не прибегая к указателям, но это будет менее эффективно в плане потребления памяти, так как каждый узел при добавлении в стек будет дублироваться.

Для каждой операции со стеком мы пропишем отдельную функцию. Начнем с *new\_stack*, которая будет создавать стек. Далее нам потребуются *push\_stack* и *pop\_stack* для добавления и удаления элементов из стека соответственно. Последней мы создадим *is\_empty\_stack*, которая будет проверять, пуст стек или нет.

Функция `new_stack` приведена в листинге 2.3.

**Листинг 2.3.** Создание стека

```
stack *new_stack(void) {
    stack *s = malloc(sizeof(stack)); ❶
    if (s == NULL) {
        fprintf(stderr, "malloc error\n");
        exit(1);
    }
    s->highest_used = -1; ❷
    return s;
}
```

Сначала мы выделяем под стек память ❶. Далее устанавливаем `highest_used` равной -1 ❷. Напомню, что -1 означает пустой стек. Обратите внимание, что мы не инициализируем элементы `s->values`, поскольку стек пуст.

Я поместил `stack_push` и `stack_pop` в один листинг 2.4, чтобы подчеркнуть симметрию операций.

**Листинг 2.4.** Добавление и извлечение элементов из стека

```
void push_stack(stack *s, node *value) {
    s->highest_used++; ❶
    s->values[s->highest_used] = value; ❷
}

node * pop_stack(stack *s) {
    node * ret = s->values[s->highest_used]; ❸
    s->highest_used--; ❹
    return ret; ❺
}
```

В `push_stack` мы сначала выделяем место под новый элемент ❶, затем помещаем туда `value` ❷.

Функция `pop_stack` удаляет элемент с индексом `highest_used`. Однако если поручить ей только удаление, то пользы будет мало: она будет извлекать элемент, но что именно извлечено, мы не узнаем. Чтобы это исправить, мы сохраняем элемент, который собираемся удалить, в переменной `ret` ❸. Затем, уменьшая `highest_used` на единицу, мы удаляем элемент ❹, а в завершение возвращаем его ❺.

Я не включил в `push_stack` и `pop_stack` проверку на ошибки. Имейте в виду, что `push_stack` приведет к сбою, если вы попытаете передать больше максимально допустимого числа элементов. Но в нашем случае опасаться нечего, так как мы сделали стек максимально большим для любого возможного ввода. Подобным образом, `pop_stack` вызовет сбой, если попытаться извлечь элемент из пустого стека, но мы предусмотрительно заранее проверяем стек на пустоту. Конечно

же, для более общих задач стеки должны создаваться с большим запасом надежности.

Для выяснения, не пуст ли стек, служит функция `is_empty_stack` из листинга 2.5, в которой с помощью оператора `==` выполняется проверка равенства `highest_used` и `-1`.

**Листинг 2.5. Проверка наличия элементов в стеке**

```
int is_empty_stack(stack *s) {
    return s->highest_used == -1;
}
```

Прежде чем вычислять общее количество конфет в дереве, давайте протестируем созданный стек на небольшом примере, приведенном в листинге 2.6. Я рекомендую не пожалеть нескольких минут на самостоятельную трассировку этого кода. Попробуйте предугадать результат, а затем запустите код на выполнение и проверьте, соответствует ли вывод вашим ожиданиям.

**Листинг 2.6. Пример использования стека**

```
int main(void) {
    stack *s;
    s = new_stack();
    node *n, *n1, *n2, *n3;
    n1 = new_house(20);
    n2 = new_house(30);
    n3 = new_house(10);
    push_stack(s, n1);
    push_stack(s, n2);
    push_stack(s, n3);
    while (!is_empty_stack(s)) {
        n = pop_stack(s);
        printf("%d\n", n->candy);
    }
    return 0;
}
```

Теперь давайте разберем, что же этот код делает. Сначала создается новый стек `s`. Далее создаются три узла с домами: `n1` содержит 20 конфет, `n2` — 30, а `n3` — 10.

Мы передаем эти поддеревья (состоящие из одного узла каждое) в стек: сначала `n1`, затем `n2` и в конце `n3`. Далее, пока стек не опустеет, мы поочередно извлекаем элементы и выводим соответствующие числа конфет. Извлекаются элементы в обратном порядке, поэтому вызовы `printf` дают нам 10, 30, 20.

## Решение со стеком

Теперь у нас есть средства отслеживания ожидающих поддеревьев: выбирая одно из них, второе мы помещаем в стек. Для вычисления общего количества конфет

важно, что стек дает нам возможность отложить поддерево (помогает помнить о нем) и затем извлечь его, чтобы обработать в подходящее время.

Для этого также можно использовать *очередь*, структуру данных, которая будет предоставлять элементы по принципу «*первым вошел — первым вышел*», *FIFO* (first in, first out). Это изменит порядок посещения поддеревьев и порядок суммирования конфет, хотя результат мы получим тот же. Я выбрал стек, потому что его проще создать.

Теперь мы готовы реализовать `tree_candy`. Нам нужно обработать два случая: первый — при рассмотрении узла без дома, и второй — при рассмотрении узла с домом.

Чтобы узнать, имеет ли текущий узел дом, мы будем проверять его указатели `left` и `right`. При отсутствии дома они оба будут указывать на поддеревья, то есть не будут нулевыми. В этом случае мы будем сохранять указатель на левое поддерево в стеке и продолжать движение к правому поддереву. Код для узла без дома выглядит так:

```
if (tree->left && tree->right) {
    push_stack(s, tree->left);
    tree = tree->right;
}
```

Если же `left` и `right` окажутся нулевыми, значит, перед нами узел с домом. В таких узлах находятся конфеты, поэтому первым делом нужно добавить их число к общей сумме:

```
total = total + tree->candy;
```

Из дома нет пути вниз по дереву. Если стек пуст — нет ожидающих обхода поддеревьев, значит, мы закончили обход. Если же стек еще не опустел, то нужно извлечь из него поддерево и обработать.

```
total = total + tree->candy;
if (is_empty_stack(s))
    tree = NULL;
else
    tree = pop_stack(s);
```

Полный код для `tree_candy` при использовании стека приведен в листинге 2.7.

**Листинг 2.7.** Вычисление общего количества конфет при помощи стека

```
int tree_candy(node *tree) {
    int total = 0;
    stack *s = new_stack();
    while (tree != NULL) {
        if (tree->left && tree->right) {
            push_stack(s, tree->left);
            tree = tree->right;
```

```
    } else {  
        total = total + tree->candy;  
        if (is_empty_stack(s))  
            tree = NULL;  
        else  
            tree = pop_stack(s);  
    }  
}  
return total;  
}
```

Пусть  $n$  будет количеством узлов в дереве. На каждом шаге цикла `while` переменная `tree` связывается с новым узлом. Таким образом, мы посещаем каждый узел по одному разу, при этом они добавляются в стек и извлекаются из него также не более одного раза. В итоге каждый узел задействован в определенном количестве шагов, и мы получаем алгоритм с линейным временем, или  $O(n)$ .

## Принципиально другое решение

Функция `tree_candy` работает, но при этом она достаточно сложна. Нам пришлось создать стек, отслеживать ожидающие деревья, а также возвращаться к ним при достижении тупика. В функциях для деревьев использование стека не является идеальным решением по двум причинам:

1. Код стека используется каждый раз, когда нам нужно идти по одному пути и позже возвращаться для обхода другого. Обработка деревьев изобилует задачами, требующими данного шаблона.
2. Сложность основанного на стеке кода растёт параллельно со сложностью задачи. Впереди у нас куда более сложные задачи, чем подсчет конфет. Для них потребуется не только стек ожидающих поддеревьев, но также управление потоком информации для отслеживания обработки, которую необходимо произвести для каждого поддерева.

Поэтому мы перепишем код под задачи с более высоким уровнем абстракции, полностью удалив из него стек.

## Рекурсивные определения

Основанная на стеке функция `tree_candy` описывает *конкретные шаги*, необходимые для решения задачи: добавить элемент в стек; перейти по дереву в этом направлении; при достижении тупика извлечь элемент из стека; остановиться по завершении обработки всего дерева. Рассмотрим решение, которое сосредоточено вокруг *структуры* задачи. Этот метод решает основную задачу через серию решений ее подзадач. В нашем случае будут действовать два правила:

- **Правило 1.** Если корень дерева является узлом с домом, тогда общее количество конфет в дереве равно их количеству в этом доме.
- **Правило 2.** Если корень дерева является узлом без дома, тогда число конфет в дереве равняется сумме их количеств в левом и правом поддеревьях.

Такое описание называют *рекурсивным*. Правило 2 демонстрирует пример рекурсивного решения задачи через ее подзадачи. Нам требуется вычислить общее количество конфет в дереве. Согласно правилу 2, это общее количество можно получить, сложив результаты решения меньших задач, то есть суммируя конфеты в левом и правом поддеревьях.

В этот момент мои студенты обычно впадают в негодование. Как такое описание вообще может помочь что-то решить? Даже если и может, как все это записать в коде? Проблему усугубляют книги и руководства, наделяющие рекурсию мистическим свойством, в которое нужно верить без понимания. Однако здесь вовсе не требуются ни вера, ни безрассудство.

Давайте проработаем небольшой пример, чтобы понять, почему предложенное рекурсивное описание верно.

Рассмотрите дерево, состоящее из одного дома с четырьмя конфетами:



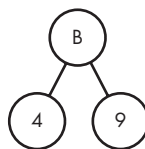
Согласно правилу 1, ответом будет четыре. При каждой встрече данного дерева в дальнейшем нужно просто помнить, что ответ равен четырем.

Хорошо. Теперь рассмотрим дерево, состоящее из одного дома с девятью конфетами:



Опять же, согласно правилу 1, ответом будет девять, и когда мы встретим это дерево в дальнейшем, то просто вспомним, что его сумма равна девяти.

А теперь решим задачку с деревом побольше:





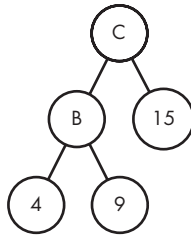
На этот раз правило 1 уже не применяется, так как корневой узел дерева не является домом. Тем не менее у нас есть правило 2. Нам известно, что в левом дереве всего четыре конфеты — ведь мы это дерево уже видели. То же касается и правого узла, который содержит девять конфет. Следовательно, согласно правилу 2, все дерево содержит  $4 + 9 = 13$  конфет. Запомним это.

Теперь идем дальше. Вот еще одно дерево из одного дома, в котором находится 15 конфет:



Согласно правилу 1, в этом дереве всего 15 конфет. Опять же запомним.

А теперь рассмотрим дерево из пяти узлов:



Здесь уже применяется правило 2, так как корневой узел не является домом. Нам нужно получить общее число конфет в левом и правом поддеревьях. Для левого мы уже это значение знаем, оно равно 13. Нам также известно общее число конфет в правом поддереве — 15. Согласно правилу 2, дерево содержит  $13 + 15 = 28$  конфет.

Эту логику можно продолжать использовать для нахождения общего количества конфет и в деревьях больших размеров. По аналогии с данным примером вычисляйте сначала небольшие поддеревья, а затем более крупные. При этом всегда будут применяться правила 1 и 2, позволяя получать ответы для малых деревьев, которые можно использовать в дальнейшем.

Теперь я закодирую эти правила как функцию на Си. Обратимся к листингу 2.8.

**Листинг 2.8.** Вычисление общего числа конфет с помощью рекурсии

```
int tree_candy(node *tree) {  
    if (!tree->left && !tree->right) ❶  
        return tree->candy;  
    return tree_candy(tree->left) + tree_candy(tree->right); ❷  
}
```

Обратите внимание, как непосредственно записаны в коде правила 1 и 2. Здесь присутствует инструкция `if`, чье условие оказывается верно, когда левые и правые поддеревья имеют значение `NULL` ❶. Отсутствие поддеревьев означает, что `tree` является узлом с домом и нам нужно применять правило 1, что мы и делаем. В частности, мы возвращаем количество конфет в узле `tree`. Если же правило 1 не применяется, то мы понимаем, что `tree` является узлом без дома и можно использовать правило 2, вернув сумму конфет из левого и правого поддеревьев ❷... Но здесь мы приостановимся.

Как именно работает правило 2? Общее количество конфет в поддеревьях получается через вызов для каждого из них функции `tree_candy`, но мы же уже находимся в `tree_candy`!

Вызов функции из самой функции называется *рекурсивным вызовом*. Если функция совершает такой вызов, то говорят, что она использует *рекурсию*. Одна из самых больших ошибок, которые можно допустить в этом месте, — пытаться отследить, как реализуется рекурсия в компьютере. Я воздержусь от обсуждения низкоуровневых деталей, касающихся организации компьютером подобных вызовов. Достаточно сказать, что он использует стек для отслеживания ожидающих вызовов функции. Это очень похоже на то, как мы ранее применяли стек в коде для решения `tree_candy`. По этой причине наш рекурсивный код, как и код, основанный на стеке, является решением  $O(n)$ .

Я много раз наблюдал, как попытка вручную отследить рекурсивные вызовы погружает любопытствующего в пучину сложностей. Дело просто в уровне абстракции. Позвольте компьютеру выполнять рекурсию так же, как позволяете ему реализовывать циклы и вызовы функций.

Вот как я предлагаю осмыслить рекурсивный код:

- Если корень дерева является домом, вернуть количество находящихся в нем конфет.
- В противном случае, когда корень дерева представлен узлом без дома, вернуть сумму общих чисел конфет в левом и правом поддеревьях.

При написании рекурсивного кода легко ошибиться. Одна из типичных ошибок состоит в неосторожном отбрасывании информации, когда на деле ее нужно возвращать.

Данную ошибку демонстрирует следующий код:

```
int tree_candy(node *tree) { //с ошибкой!  
    if (!tree->left && !tree->right)  
        return tree->candy;  
    tree_candy(tree->left) + tree_candy(tree->right); ❶  
}
```

Баг в том, что мы ничего не возвращаем ❶, так как отсутствует ключевое слово `return`. На деле же нужно возвращать сумму, а не отбрасывать ее.

Еще одна распространенная ошибка — совершение рекурсивного вызова для чего-либо, не являющегося меньшей подзадачей текущей задачи. Вот пример:

```
int tree_candy(node *tree) { //с ошибкой!  
    if (!tree->left && !tree->right)  
        return tree->candy;  
    return tree_candy(tree); ❶  
}
```

Снова взгляните на команду `return` ❶. Функция заключает в себе инструкцию: вычислить общее число конфет в дереве через... вычисление общего числа конфет в дереве. И она не будет работать для дерева, чей корень является узлом без дома, поскольку заполнит память ожидающими вызовами, пока программа не даст сбой.

### Рекурсия: немного практики

Прежде чем приступить к решению задачи про Хэллоуин, давайте поучимся использовать рекурсию. Напишем несколько функций, аналогичных `tree_candy`.

Сначала, используя указатель на корень двоичного дерева, вернем количество его узлов. Если узел окажется концевым, значит, перед нами дерево из одного узла, и верным возвращаемым значением будет 1. Если перед нами не концевой узел, то количество узлов в дереве определяется как один (текущий) узел плюс сумма узлов левого и правого поддеревьев. Таким образом, имеем два правила:

- **Правило 1.** Если корень является концевым узлом, тогда количество узлов в дереве равно 1.
- **Правило 2.** Если корень не является концевым узлом, тогда количество узлов дерева равно 1 плюс сумма чисел узлов в левом и правом поддеревьях.

Правила, подобные первому, называются *базовыми случаями*, потому что не требуют использования рекурсии. Правила, подобные второму, называются *рекурсивными случаями*, так как требуют рекурсивного решения более мелких подзадач. Каждой рекурсивной функции необходимо не менее одного базового и одного рекурсивного случая: базовый сообщает, что делать, когда задача проста, а рекурсивный указывает действия для более сложных ее вариантов.

Преобразуя эти правила в код, мы получаем функцию, приведенную в листинге 2.9.

**Листинг 2.9. Вычисление количества узлов**

```
int tree_nodes(node *tree) {  
    if (!tree->left && !tree->right)  
        return 1;  
    return 1 + tree_nodes(tree->left) + tree_nodes(tree->right);  
}
```

Далее мы напишем функцию для возвращения количества концевых узлов дерева. Если узел является концевым, то возвращаем 1. Если же он не концевой, мы его не считаем. Таким образом, учитываются только концевые узлы в левом и правом поддеревьях. Код приведен в листинге 2.10.

**Листинг 2.10. Вычисление количества концевых узлов**

```
int tree_leaves(node *tree) {  
    if (!tree->left && !tree->right)  
        return 1;  
    return tree_leaves(tree->left) + tree_leaves(tree->right);  
}
```

Единственное отличие между этим и предыдущим листингами состоит в операции прибавления единицы в последней строке. Рекурсивные функции зачастую очень похожи друг на друга, но при этом могут выполнять совершенно разные вычисления.

## **Обход минимального количества улиц**

Возможно, вам потребуется вернуться к описанию задачи, чтобы вспомнить ее условия. Теперь мы знаем, как получать общее количество конфет, но это только один из двух нужных нам результатов. Помимо него нам требуется определить минимальное число улиц, которые нужно пройти для сбора этих конфет. Если решать эту задачу с помощью рекурсии, то конфет мы не получим!

### **Вычисление количества улиц**

Ранее я представил вариант с проходом 30 улиц (рис. 2.2) и попросил вас найти решение получше. При оптимальном решении мы проходим 26 улиц. В этом случае четыре улицы сокращаются за счет того, что можно закончить обход дерева сразу после получения последних конфет. В данной задаче нет требования по завершении сбора конфет вернуться к корню дерева.

А что, если бы возвращение также входило в нашу прогулку? Логично предположить, что мы бы получили ошибочный ответ, так как прошли бы больше улиц, чем необходимо. Хотя верно и то, что возвращение к корню существенно упрощает задачу. В этом случае нам не нужно выискивать вариант обхода дерева,

который минимизирует общее число пройденных улиц (если в итоге мы все равно вернемся к корню, то не нужно прокладывать путь так, чтобы оптимальным образом выбрать последний посещаемый дом). Может быть, мы подсчитаем длину пути с возвратом к корню, а затем просто вычтем лишние пройденные улицы? Это выход!

Мы поступим так же, как и с `tree_candy`, определяя базовый и рекурсивный случаи.

Что мы делаем, если корень дерева оказывается домом — сколько улиц проходим, начиная от этого дома, чтобы вновь вернуться к нему? Ответ — нисколько. Улицы обходить не потребуется.

А что мы делаем, если корень не содержит дом? Вернемся к рис. 2.3, где я разделил дерево на две части. Предположим, что нам известно количество улиц, необходимых для обхода каждого из поддеревьев *F* и *G*. Вычислить их можно рекурсивно. Теперь вернемся к полной схеме дерева (рис. 2.2): сколько дополнительных улиц необходимо пройти? Одну улицу от *H* к *F*, после чего, закончив с поддеревом *F*, нужно пройти одну улицу от *F* обратно к *H*. То же самое необходимо проделать для *G*: пройти от *H* к *G*, а затем, по завершении обхода *G*, вернуться от него к *H*. Это и есть дополнительные улицы, помимо того количества, что мы получаем с помощью рекурсии.

Снова сформулируем два правила:

- **Правило 1.** Если корень дерева является узлом с домом, значит, улиц для обхода нет.
- **Правило 2.** Если корень дерева является узлом без дома, значит, общее число улиц для обхода будет равно сумме улиц, пройденных в левом и правом поддеревьях, плюс 4.

В листинге 2.11 эти правила преобразованы в код.

**Листинг 2.11.** Вычисление количества улиц при возвращении обратно к корню

```
int tree_streets(node *tree) {
    if (!tree->left && !tree->right)
        return 0;
    return tree_streets(tree->left) + tree_streets(tree->right) + 4;
}
```

Если вы выполните обход по рис. 2.2, начав с *H*, собрав все конфеты и закончив в *H*, то пройдете 32 улицы. Неважно, как вы обходите дерево: если вы посещаете каждый дом по одному разу и не слоняетесь по улицам бесцельно, то у вас будет получаться 32. Минимальное количество улиц, которое можно обойти без необходимости возвращаться к корню, равно 26. Так как  $32 - 26 = 6$ , получается, что мы превышаем верный ответ на шесть.

Поскольку возвращаться к корню не требуется, есть смысл организовать прогулку так, чтобы последний посещаемый узел находился максимально далеко от корневого. Например, завершение в доме с 7 конфетами окажется неудачным решением, потому что он находится всего в двух улицах от Н. Но взгляните на отдаленные дома 4 и 9 внизу. Было бы здорово закончить наш путь в одном из них. Если, к примеру, мы решим закончить обход в 9, то это сэкономит нам шесть улиц пути: от 9 до В, от В к С, от С к D, от D к Е, от Е к F и от F к Н.

Таким образом, нашей целью будет закончить маршрут в максимально удаленном от корня доме. Если этот дом и корень разделяют шесть улиц, значит, есть путь из шести ребер между корнем и некоторым концевым узлом. И это является точным определением высоты дерева! Если мы сможем рекурсивно вычислить высоту дерева, то потом вычтем ее из результата `tree_streets`. Так мы попадаем в самый отдаленный от корня дом, сокращая путь на максимальное число улиц.

В качестве небольшого отступления скажу, что на самом деле не обязательно знать, какой дом является самым удаленным. Несущественно даже то, как именно выстроить путь, чтобы этот дом оказался последним. Нужно лишь убедиться, что мы *можем* выстроить путь так, чтобы этот дом стал последним. Я приведу на примере рис. 2.2 краткое доказательство, которое, надеюсь, убедит вас. Сравните высоты поддеревьев F и G, а затем, начав в Н, полностью обойдите то поддерево, чья высота окажется меньше, — в данном случае это G. Далее повторите этот процесс для поддеревьев E. Сравните высоты A и E, после чего полностью обойдите поддерево A (так как оно ниже). Продолжайте действовать по этому алгоритму, пока не обойдете все поддерева. Таким образом последним посещенным вами домом окажется самый удаленный от Н.

### Вычисление высоты дерева

Перейдем к написанию функции `tree_height`, еще одной демонстрации рекурсивного подхода, основанного на двух правилах.

Высота дерева, состоящего из одного дома, равна нулю, так как ребра в нем отсутствуют.

Дерево, чей корень не является домом, представлено на рис. 2.3. Высота поддерева F равна пяти, а высота G — единице. Эти подзадачи можно решить рекурсивно. Высота исходного дерева с корнем Н будет на единицу больше, чем максимальная высота F и G, так как наличие ребер Н увеличивает ее на единицу. Исходя из этого, мы получаем два правила:

- **Правило 1.** Если корень дерева является узлом с домом, тогда высота дерева равна нулю.

- **Правило 2.** Если корень дерева является узлом без дома, тогда высота дерева будет на единицу больше, чем максимальная высота левого и правого поддеревьев.

В листинге 2.12 вспомогательная функция `max` выбирает максимальное из двух чисел. В остальном `tree_height` не требует комментариев.

**Листинг 2.12.** Вычисление высоты дерева

```
int max(int v1, int v2) {
    if (v1 > v2)
        return v1;
    else
        return v2;
}

int tree_height(node *tree) {
    if (!tree->left && !tree->right)
        return 0;
    return 1 + max(tree_height(tree->left), tree_height(tree->right));
}
```

Теперь у нас есть `tree_candy` для вычисления общего количества конфет, а также `tree_streets` и `tree_height` для вычисления минимального количества улиц. Объединяя эти три компонента, мы получаем функцию, которая решает задачу для данного дерева. Ее код приведен в листинге 2.13.

**Листинг 2.13.** Решение задачи для заданного дерева

```
void tree_solve(node *tree) {
    int candy = tree_candy(tree);
    int height = tree_height(tree);
    int num_streets = tree_streets(tree) - height;
    printf("%d %d\n", num_streets, candy);
}
```

Попробуйте вызвать эту функцию для деревьев, которые создали в подразделе «Построение дерева» на с. 70.

## Считывание входных данных

Мы уже очень близки к цели, но пока ее не достигли. Да, мы можем решить задачу, если у нас будут параметры дерева, но, если вы помните, входные данные представлены в виде строк текста. Нам понадобится преобразовать каждую из этих строк в дерево, после чего мы уже сможем задействовать `tree_solve`. Теперь можно раскрыть способ, которым деревья представляются в виде текста.

### Представление дерева в виде строки

Я продемонстрирую соответствие между строкой текста и деревом на нескольких примерах.

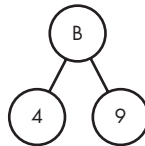
Дерево из одного узла с домом представляется просто числом конфет. Например, дерево с четырьмя конфетами



представляется просто как

4

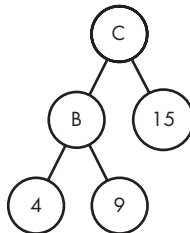
Дерево, чей корень является узлом без дома, представляется (рекурсивно!) заключенными в скобки и разделенными пробелом числами, обозначающими количество конфет в поддеревьях (слева направо). Например, приведенное ниже дерево из трех узлов



представляется так:

(4 9)

Дерево из пяти узлов



будет представлено так:

((4 9) 15)

Здесь левое поддерево — это (4 9), а правое поддерево — это 15.



В формате правил мы получаем следующее:

- **Правило 1.** Если текст состоит только из цифр целого числа  $s$ , то дерево состоит из одного узла с домом, содержащим  $s$  конфет.
- **Правило 2.** Если текст начинается с открывающей скобки, то корень дерева является узлом без дома. После открывающей скобки текст содержит цифры количества конфет в левом поддереве, пробел, цифры количества конфет в правом поддереве и закрывающую скобку.

### Считывание узла без дома

Запишем функцию `read_tree` со следующей сигнатурой:

```
node *read_tree(char *line)
```

Функция получает строку и возвращает дерево. Начнем с реализации правила 2, поскольку правило 1 подразумевает тонкую работу по преобразованию символов в целые числа.

Правило 2 является рекурсивным и требует двух вызовов `read_tree` — по одному для считывания левого и правого поддеревьев:

```
node *tree;
tree = malloc(sizeof(node));
if (line[0] == '(') {
    tree->left = read_tree(&line[1]); ❶
    tree->right = read_tree(???); ❷
    return tree;
}
```

После выделения памяти для корня дерева мы совершаем рекурсивный вызов для считывания левого поддерева ❶. Мы устанавливаем указатель на индекс 1 в `line`, чтобы рекурсивный вызов получил строку, не включающую открывающую скобку из индекса 0. Тем не менее далее у нас возникает проблема ❷. Откуда начинать чтение правого поддерева? Сколько символов характеризуют левое поддерево? Мы не знаем! Можно написать отдельную функцию, чтобы выяснить, где заканчивается левое поддерево. Например, можно посчитать количество открывающих и закрывающих скобок, хотя это выглядит излишним: если `read_tree` успешно считала левое поддерево, значит, она знает, где оно окончилось? Если бы только был способ передать эту информацию обратно для исходного вызова `read_tree`, то мы бы определили, какую часть строки передавать для второго рекурсивного вызова.

Добавление параметра в рекурсивную функцию является универсальным и мощным способом решения подобных задач. Если рекурсивный вызов содержит информацию, не передаваемую через то, что он возвращает, может быть целесообразным

добавление параметра. Если этот параметр будет указателем, его можно использовать как для передачи дополнительной информации в рекурсивные вызовы, так и для получения информации из них.

В данном случае нам нужно сообщить рекурсивному вызову, откуда начинается его часть строки. Кроме того нужно, чтобы рекурсивный вызов мог сообщить, откуда мы должны продолжить обработку строки. Для этого мы добавим параметр целочисленного указателя `pos`. Однако нам не нужно добавлять его в `read_tree`, так как пользователю, вызывающему `read_tree`, этот параметр не интересен. Пользователь должен просто иметь возможность передать строку, не заботясь о параметре `pos`, который является внутренним.

Мы сохраним `read_tree` в прежнем виде, добавив только параметр `line`. Затем `read_tree` будет вызывать `read_tree_helper`, и именно `read_tree_helper` будет содержать параметр `pos` и запускать рекурсию.

Код для `read_tree` представлен в листинге 2.14. Она передает указатель на 0 в `read_tree_helper`, потому что начать обработку мы хотим именно с индекса 0 (начала строки).

**Листинг 2.14.** Вызов вспомогательной функции с указателем на `int`

```
node *read_tree(char *line) {
    int pos = 0;
    return read_tree_helper(line, &pos);
}
```

Теперь можно снова попробовать реализовать правило 2:

```
node *tree;
tree = malloc(sizeof(node));
if (line[*pos] == '(') {
    (*pos)++; ❶
    tree->left = read_tree_helper(line, pos);
    (*pos)++; ❷
    tree->right = read_tree_helper(line, pos);
    (*pos)++; ❸
    return tree;
}
```

Эта функция будет вызываться с параметром `pos`, указывающим на первый символ, поэтому сначала мы перемещаем `pos` на один символ, пропуская открывающую скобку ❶. Теперь `pos` идеально расположен в начале левого поддеревья. Далее мы совершаем рекурсивный вызов для считывания левого поддеревья. Этот рекурсивный вызов сдвинет `pos` на индекс символа, следующего за левым поддеревом. А поскольку за ним следует пробел, этот пробел пропускается ❷. Теперь мы находимся в начале правого поддеревья. Далее мы рекурсивно считываем это поддерево

и затем пропускаем закрывающую скобку ❸, парную пропущенной в начале открывающей ❶. Пропуск закрывающей скобки очень важен, так как данная функция отвечает за обработку всего поддерева, включая его закрывающие скобки. Если завершающий пропуск не сделать, то при следующем вызове функции указатель будет установлен на закрывающей скобке вместо ожидаемого пробела. После пропуска закрывающей скобки остается только вернуть дерево.

### Считывание узла с домом

Разобравшись с правилом 2, можно заняться правилом 1. Для этого нам понадобится преобразовывать часть строки в целое число. Давайте напишем небольшую программу, которая позволит это сделать. Она будет получать строку, которая, как мы предполагаем, представляет узел с домом, и выводить значение числа конфет. Как ни странно, но, если быть недостаточно аккуратным, то можно получить неожиданные результаты. Предупреждаю: в листинге 2.15 мы намеренно неаккуратны.

#### Листинг 2.15. Считывание значения числа конфет (с ошибкой!)

```
#define SIZE 255

int main(void) { //с ошибкой!
    char line[SIZE + 1];
    int candy;
    gets(line);
    candy = line[0];
    printf("%d\n", candy);
    return 0;
}
```

Запустите программу и введите число 4. Скорее всего, на выходе вы получите 52. Запустите еще раз и введите 9. В этом случае вывод покажет 57. А теперь то же с 0. Должно получиться 48. Наконец, выполните программу с каждым значением от 0 до 9. Если 0 дает на выходе 48, тогда 1 даст 49, 2 даст 50, и т. д.

Здесь мы получаем символьный код для каждой цифры. Важнейший момент состоит в том, что коды для целых чисел последовательны. Поэтому можно вычесть символьный код нуля, чтобы сдвинуть получаемые значения в нужный диапазон. Сделав эту корректировку, мы получим код, приведенный в листинге 2.16. Опробуйте его.

#### Листинг 2.16. Считывание значения числа конфет

```
#define SIZE 255

int main(void) {
    char line[SIZE + 1];
    int candy;
    gets(line);
    candy = line[0] - '0';
}
```

```
    printf("%d\n", candy);  
    return 0;  
}
```

Эта небольшая программа работает для целых чисел, состоящих из одной цифры. Но условия задачи про Хэллоуин требуют, чтобы мы также обрабатывали целочисленные значения конфет, состоящие из двух цифр. Предположим, что считываем цифру 2, а затем цифру 8. Нам нужно совместить их и получить в итоге число 28. Для этого можно просто умножить первую цифру на 10 (это даст 20), а затем прибавить восемь (получив 28). В листинге 2.17 показана еще одна тестовая программка, позволяющая проверить правильность нашего решения. Здесь мы предполагаем, что вводится строка из двух цифр.

**Листинг 2.17.** Считывание значения числа конфет, состоящего из двух цифр

```
#define SIZE 255  
  
int main(void) {  
    char line[SIZE + 1];  
    int digit1, digit2, candy;  
    gets(line);  
    digit1 = line[0] - '0';  
    digit2 = line[1] - '0';  
    candy = 10 * digit1 + digit2;  
    printf("%d\n", candy);  
    return 0;  
}
```

Это все, что нужно для описания правила 1, и теперь можно записать следующее:

```
--snip--  
tree->left = NULL; tree->right = NULL;  
tree->candy = line[*pos] - '0'; ❶  
(*pos)++; ❷  
if (line[*pos] != ')') && line[*pos] != ' ' &&  
    line[*pos] != '\0') {  
    tree->candy = tree->candy * 10 + line[*pos] - '0'; ❸  
    (*pos)++; ❹  
}  
return tree;
```

Так как мы создаем узел с домом, то начинаем с установки значения NULL для левого и правого поддеревьев. Затем преобразуем символ в цифру ❶ и переводим указатель дальше ❷. Если значение числа конфет состоит из одной цифры, то мы сохранили его верно. Если же оно состоит из двух цифр, то нужно умножить первую на 10 и прибавить вторую. Следовательно, нам надо определить, состоит ли значение из одной или двух цифр. Если перед нами не закрывающая скобка, пробел или ноль-терминатор в конце строки, значит, скорее всего, перед нами вторая цифра. Тогда мы включаем ее в значение числа конфет ❸ и переходим дальше.

В листинге 2.18 объединен код для правил 2 и 1.

**Листинг 2.18.** Преобразование строки в дерево

```
node *read_tree_helper(char *line, int *pos) {
    node *tree;
    tree = malloc(sizeof(node));
    if (tree == NULL) {
        fprintf(stderr, "malloc error\n");
        exit(1);
    }
    if (line[*pos] == '(') {
        (*pos)++;
        tree->left = read_tree_helper(line, pos);
        (*pos)++;
        tree->right = read_tree_helper(line, pos);
        (*pos)++;
        return tree;
    } else {
        tree->left = NULL;
        tree->right = NULL;
        tree->candy = line[*pos] - '0';
        (*pos)++;
        if (line[*pos] != ')') {
            if (line[*pos] != '\0') {
                tree->candy = tree->candy * 10 + line[*pos] - '0';
                (*pos)++;
            }
        }
        return tree;
    }
}
```

Осталось только создать функцию `main` для считывания каждого тестового примера и его решения. Листинг 2.19 содержит все необходимое.

**Листинг 2.19.** Функция `main`

```
#define SIZE 255
#define TEST_CASES 5

int main(void) {
    int i;
    char line[SIZE + 1];
    node *tree;
    for (i = 0; i < TEST_CASES; i++) {
        gets(line);
        tree = read_tree(line);
        tree_solve(tree);
    }
    return 0;
}
```

## Когда использовать рекурсию?

Не всегда понятно, удастся ли с помощью рекурсии получить оптимальное решение задачи. Вот верный признак: если задачу можно решить комбинированием решений более мелких подзадач, то стоит попробовать рекурсию. В этой главе во всех случаях мы решали с помощью рекурсии две подзадачи для получения ответа на основную задачу. Задачи, делящиеся на две подзадачи, очень распространены, но бывают и такие, которые требуется делить на три, четыре и более подзадач.

Как понять, что разделение на подзадачи поможет решить исходную задачу и какие это вообще должны быть подзадачи? К этому вопросу мы вернемся в главе 3, когда на основе полученных знаний будем изучать мемоизацию и динамическое программирование. А пока подумайте, смогли бы вы решить какую-либо задачу, если бы кто-то подсказал вам решения ее подзадач? Например, вернитесь к подсчету общего количества конфет в дереве. Это не такая уж легкая задача. Что, если бы кто-то сообщил вам общее число конфет в левом и правом поддеревьях? Это существенно упростило бы решение. Упрощение задачи за счет нахождения решений ее подзадач является мощной возможностью, которую предоставляет рекурсия.

Перейдем к еще одной задаче, где рекурсия также окажется полезна. Попробуйте в процессе чтения условия определить, где и почему рекурсия будет включена в решение.

## Задача 2. Расстояние до потомка

Теперь мы перейдем от двоичных деревьев к общим, в которых узлы могут иметь более двух детей.

Это задача с сайта DMOJ под номером esna05b.

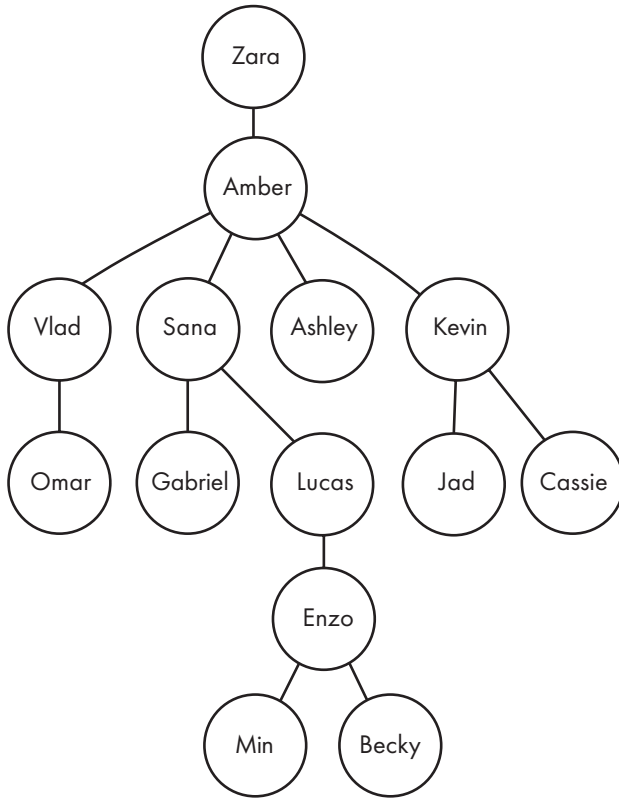
### Условие

Дано генеалогическое древо и некоторое расстояние  $d$ . Количество потомков, которые находятся на расстоянии  $d$  от узла, определяют его рейтинг (количество баллов). Требуется вывести узлы с наиболее высоким рейтингом. В разделе «Вывод» я подробно объясню, по каким правилам определяется число выводимых узлов.

Чтобы понять условие, взгляните на генеалогическое древо на рис. 2.6.

Рассмотрим узел Amber. От Amber идут четыре дочерних узла, значит, на расстоянии одного шага от нее находятся четыре потомка. При этом у Amber также есть пять внуков, которые находятся уже на расстоянии двух шагов. Обобщая, можно

сказать, что для любого узла количество потомков на расстоянии  $d$  равно числу узлов, расположенных ниже родительского узла не более чем на  $d$  ребер.



**Рис. 2.6.** Образец генеалогического дерева

### Входные данные

Первая строка входных данных представляет число тестовых примеров. Каждый пример состоит из следующих строк:

- строки, содержащей два целых числа,  $n$  и  $d$ . Число  $n$  сообщает, сколько еще строк в этом тестовом примере, а  $d$  указывает интересное нас расстояние до потомков;
- $n$  строк, используемых для построения дерева. Каждая состоит из имени узла, целого числа  $m$  и  $m$  имен узлов потомков. Длина имени — не более 10 символов. Эти строки могут идти в любом порядке, то есть родители не обязательно будут идти перед потомками.

В каждом тестовом примере может быть не более 1000 узлов.

Ниже представлен пример входных данных, которые сгенерируют дерево, представленное на рис. 2.6. В примере нужно вывести узлы с наибольшим числом потомков на расстоянии два:

```
1
7 2
Lucas 1 Enzo
Zara 1 Amber
Sana 2 Gabriel Lucas
Enzo 2 Min Becky
Kevin 2 Jad Cassie
Amber 4 Vlad Sana Ashley Kevin
Vlad 1 Omar
```

## Вывод

Вывод для каждого тестового примера содержит две части.

Сначала идет следующая строка:

```
Tree i:
```

Здесь  $i$  — номер примера.

Затем выводятся имена с максимальным рейтингом (где число баллов узла равно количеству потомков, находящихся от него на расстоянии  $d$ ), упорядоченные по убыванию. Имена с одинаковым рейтингом выводятся в алфавитном порядке.

Для определения количества выводимых имен следует руководствоваться двумя правилами:

- Если имен с потомками на расстоянии  $d$  не более трех, то следует выводить их все.
- Если имен с потомками на расстоянии  $d$  более трех, то следует начать с вывода трех, имеющих наиболее высокий рейтинг. Назовем эти имена условно  $n_1$ ,  $n_2$  и  $n_3$ ; они сортируются от большего к меньшему. Кроме них следует выводить все имена, чей рейтинг равен рейтингу  $n_3$ . Например, если у нас есть имена с восемью, восемью, пятью, пятью, пятью и двумя потомками на расстоянии  $d$ , мы выведем информацию для пяти имен: с восемью, восемью, пятью, пятью и пятью потомками.

Для каждого подлежащего выводу имени выводится строка, которая содержит само имя, затем пробел и затем количество потомков на расстоянии  $d$ .

Вывод для каждого тестового примера отделяется от следующего пустой строкой.



Вот как это должно выглядеть для приведенного выше образца входных данных:

```
Tree 1:  
Amber 5  
Zara 4  
Lucas 2
```

Время на вычисление всех тестовых примеров составляет одну секунду.

## Считывание входных данных

Одно из важных различий между этой задачей и задачей про Хэллоуин состоит в том, что теперь мы уже работаем не с двоичными деревьями. Здесь узел может иметь любое количество детей. Нам придется изменить структуру кода, так как указатели `left` и `right` больше работать не будут. Вместо этого мы используем массив детей `children` и целое число `num_children` для записи количества детей, хранящихся в массиве. Мы также задействуем компонент `name`, который будет хранить имя узла (*Zara*, *Amber* и т. д.), и компонент `score` для сохранения вычисленного количества потомков. Структура узла представлена в листинге 2.20.

### Листинг 2.20. Структура `node`

```
typedef struct node {  
    char *name;  
    int num_children;  
    struct node **children;  
    int score;  
} node;
```

В задаче про Хэллоуин деревья хранились как рекурсивно определяемые выражения, из которых мы могли рекурсивно считывать левые и правые поддеревья. В данном же случае так не получится, поскольку узлы идут в произвольном порядке. К примеру, может быть задано:

```
Zara 1 Amber  
Amber 4 Vlad Sana Ashley Kevin
```

Здесь мы узнаем о родителе *Amber* раньше, чем о ее детях. Однако мы можем встретить и такой вариант:

```
Amber 4 Vlad Sana Ashley Kevin  
Zara 1 Amber
```

Здесь мы сперва узнаем о детях *Amber* и лишь потом о детях *Zara*.

Нам известно, что узлы и связи «родитель — потомок», которые мы считываем из файла, в конечном итоге сформируют одно дерево. Однако нет гарантии, что по

ходу обработки строк у нас будет получаться одно дерево. Например, мы считаем строки:

```
Lucas 1 Enzo  
Zara 1 Amber
```

В них говорится, что Enzo является ребенком Lucas, а Amber — ребенком Zara, но на этом все. Пока у нас получаются два несвязанных поддерева, объединение которых произойдет уже с помощью следующих строк.

По этой причине формировать одно связанное дерево по мере считывания строк невозможно. Вместо этого мы будем поддерживать массив указателей на узлы. Каждый раз, встречая новое имя, мы будем создавать узел и добавлять указатель на этот узел в массив. Поэтому целесообразно создать функцию для поиска имен узлов в массиве.

### Поиск узла

В листинге 2.21 реализуется функция `find_node`. Параметр `nodes` является массивом указателей на узлы, `num_nodes` сообщает количество указателей, а `name` представляет искомое имя.

#### Листинг 2.21. Поиск узла

```
node *find_node(node *nodes[], int num_nodes, char *name) {  
    int i;  
    for (i = 0; i < num_nodes; i++)  
        if (strcmp(nodes[i]->name, name) == 0)  
            return nodes[i];  
    return NULL;  
}
```

*Линейный поиск* — это поэлементный поиск в массиве. В функции мы используем его для поиска в массиве `nodes` и... так, стоп! Разве мы ищем не по массиву? Это же место для специальной хеш-таблицы (глава 1). Предлагаю вам самостоятельно применить хеш-таблицу и сравнить производительность. А мы, для упрощения решения и поскольку обработке подлежит не более 1000 узлов, продолжим использовать (медленный) линейный поиск.

Далее выполняется сравнение искомого имени с каждым именем в массиве. Если `strcmp` возвращает 0, значит, строки равны и возвращается указатель на соответствующий узел. Если по достижении конца массива совпадений имен не найдено, возвращается `NULL`.

### Создание узла

Если имя в массиве не найдено, для него нужно создать новый узел. Это подразумевает вызов функции `malloc`, которая, как мы увидим, потребуется и в других местах

программы. В связи с этим я написал вспомогательную функцию `malloc_safe`. Загляните в листинг 2.22 — там стандартная `malloc` дополнена проверкой ошибок.

**Листинг 2.22.** Функция `malloc safe`

```
void *malloc_safe(int size) {
    char *mem = malloc(size);
    if (mem == NULL) {
        fprintf(stderr, "malloc error\n");
        exit(1);
    }
    return mem;
}
```

Вспомогательная функция `new_node` из листинга 2.23 использует `malloc_safe` для создания нового узла.

**Листинг 2.23.** Создание узла

```
node *new_node(char *name) {
    node *n = malloc_safe(sizeof(node));
    n->name = name;
    n->num_children = 0;
    return n;
}
```

Вначале под новый узел выделяется память, затем устанавливается компонент узла `name`. Далее количество потомков узла определяется равным 0, поскольку мы не всегда знаем, сколько у узла детей. Предположим, что первой считывается следующая строка:

```
Lucas 1 Enzo
```

Нам известно, что у Lucas есть один ребенок, но при этом мы не знаем, сколько детей у Enzo. Компонент, вызывающий `new_node`, может установить новое значение числа его детей, как только такая информация станет доступна. Здесь это сразу же выполняется для Lucas, но не для Enzo.

## Создание генеалогического древа

Теперь можно перейти к считыванию данных и созданию дерева. Соответствующая функция приведена в листинге 2.24. Здесь `nodes` является массивом указателей на узлы, под который вызывающий компонент выделил пространство; `num_lines` указывает количество строк для считывания.

**Листинг 2.24.** Преобразование строк в дерево

```
#define MAX_NAME 10

int read_tree(node *nodes[], int num_lines) {
    node *parent_node, *child_node;
```

```

char *parent_name, *child_name;
int i, j, num_children;
int num_nodes = 0;
for (i = 0; i < num_lines; i++) {❶
    parent_name = malloc_safe(MAX_NAME + 1);
    scanf("%s", parent_name);
    scanf("%d", &num_children);
    parent_node = find_node(nodes, num_nodes, parent_name); ❷
    if (parent_node == NULL) {
        parent_node = new_node(parent_name);
        nodes[num_nodes] = parent_node;
        num_nodes++;
    }
    else
        free(parent_name); ❸
    parent_node->children = malloc_safe(sizeof(node) * num_children); ❹
    parent_node->num_children = num_children; ❺
    for (j = 0; j < num_children; j++) {
        child_name = malloc_safe(MAX_NAME + 1);
        scanf("%s", child_name);
        child_node = find_node(nodes, num_nodes, child_name);
        if (child_node == NULL) {
            child_node = new_node(child_name);
            nodes[num_nodes] = child_node;
            num_nodes++;
        }
        else
            free(child_name);
        parent_node->children[j] = child_node; ❻
    }
}
return num_nodes;
}

```

Внешний цикл **for** ❶ однократно перебирает каждую входную строку из `num_lines`. Каждая из этих строк содержит имя родителя и одно или более имен детей; первым обрабатывается родитель. Мы выделяем память, считываем имя родителя, а также количество его детей. Далее посредством вспомогательной функции `find_node` определяем, встречался ли этот узел ранее ❷. Если нет, то используем `new_node` для создания узла, сохраняем указатель на этот новый узел в массиве `nodes` и инкрементируем количество узлов. Если узел уже находится в массиве `nodes`, то выделенная под имя родителя память освобождается, поскольку использоваться она не будет ❸.

Затем память выделяется для указателей на детей этого родителя ❹, и сохраняется их количество ❺. Каждый дочерний узел обрабатывается аналогично родительскому. Если дочерний узел существует и все его члены установлены, указатель на этого потомка сохраняется в массиве `children` родителя ❻. Обратите внимание, что код для выделения памяти или установки количества дочерних узлов отсутствует, как и в случае с родителем. Если имя дочернего узла ранее встречалось, значит, в процессе

начальной обработки его дети уже устанавливались. Если же данное имя встречается первый раз, то мы установим его детей, когда узнаем о них. Если дочерний узел является концевым, количество его детей будет равно начальному значению 0.

В завершение мы получаем количество узлов в дереве, которое необходимо знать при поочередной обработке узлов.

### **Количество потомков одного узла**

По условию задачи нам нужно подсчитать количество потомков на расстоянии  $d$  для каждого узла, чтобы найти узлы, у которых больше всего потомков. Более скромной целью, которую мы реализуем в этом разделе, будет вычисление числа потомков на расстоянии  $d$  от одного узла. Напишем такую функцию:

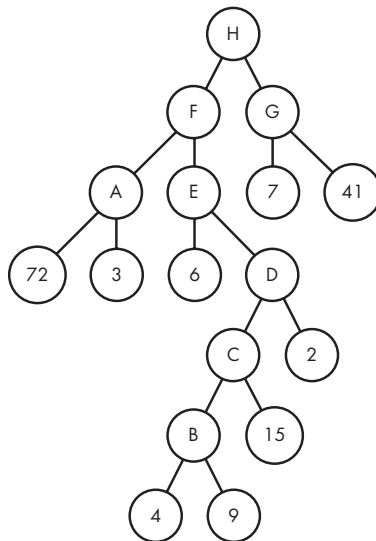
```
int score_one(node *n, int d)
```

Здесь  $n$  является узлом, для которого мы будем вычислять число потомков.

Если  $d$  равно 1, то нам нужно знать количество детей  $n$ . Все просто: ранее мы сохранили компонент `num_children` для каждого узла. Все, что теперь нужно сделать, это вернуть следующее:

```
if (d == 1)
    return n->num_children;
```

А что, если  $d$  больше 1? Представим эту ситуацию в знакомом нам контексте двоичных деревьев. Вот дерево из задачи про Хэллоуин:



Предположим, что нам дан узел двоичного дерева и мы хотим узнать количество его потомков на заданном расстоянии. Если бы мы знали их число на этом расстоянии от корней левого и правого поддеревьев, помогло бы это нам?

Не очень. Предположим, что нужно узнать количество потомков  $N$  на расстоянии два. Мы вычисляем количество потомков  $F$  и  $G$  на расстоянии два. Это никак не помогает, поскольку каждый из этих узлов находится от узла  $N$  на расстоянии три, но нас настолько удаленные узлы не интересуют.

Как это исправить? Мы просто вычислим количество потомков  $F$  и  $G$  на расстоянии одного шага от этих узлов. Теперь каждый из них окажется на нужном нам удалении от  $N$ , равном двум шагам.

Тогда, чтобы узнать число узлов на расстоянии  $d$ , мы вычисляем их количество на расстоянии  $d - 1$  в левом и правом поддеревьях.

В контексте генеалогического древа, где узел может иметь больше двух детей, мы этот подход несколько обобщим: количество узлов, удаленных от него на расстояние  $d$ , является суммой чисел узлов на расстоянии  $d - 1$  каждого его поддерева.

Определим два правила для узла  $n$ :

- **Правило 1.** Если  $d$  равно одному, то количество потомков на расстоянии  $d$  равняется количеству детей  $n$ .
- **Правило 2.** Если  $d$  больше одного, то количество потомков на расстоянии  $d$  равно сумме числа узлов на расстоянии  $d - 1$  в каждом поддереве  $n$ .

Соответствующий код приведен в листинге 2.25.

**Листинг 2.25.** Количество потомков одного узла

```
int score_one(node *n, int d) {
    int total, i;
    if (d == 1)
        return n->num_children;
    total = 0;
    for (i = 0; i < n->num_children; i++)
        total = total + score_one(n->children[i], d - 1);
    return total;
}
```

## **Количество потомков всех узлов**

Для вычисления количества потомков на расстоянии  $d$  от всех узлов мы поместим `score_one` в цикл (листинг 2.26).

**Листинг 2.26.** Количество потомков всех узлов

```
void score_all(node **nodes, int num_nodes, int d) {
    int i;
    for (i = 0; i < num_nodes; i++)
        nodes[i]->score = score_one(nodes[i], d);
}
```

Здесь мы используем переменную `score` в каждой структуре `node`: после выполнения функции `score` содержит количество потомков на расстоянии  $d$  для каждого узла. Теперь осталось только выяснить, у каких узлов наибольший показатель.

## Упорядочивание узлов

В нашей провальной попытке упорядочить снежинки в главе 1 (подраздел «Выявление проблемы») мы познакомились с функцией Си `qsort`. Ее можно использовать для сортировки узлов, чтобы расположить их в зависимости от числа потомков на расстоянии  $d$  в порядке убывания. Если количество таких потомков для узлов совпадает, то мы сортируем их в алфавитном порядке.

Для использования `qsort` нужно написать функцию сравнения, получающую указатели на два элемента и возвращающую отрицательное целое число, если первый меньше второго, 0 при их равенстве и положительное целое число, если первый больше второго. Эта функция представлена в листинге 2.27.

**Листинг 2.27.** Функция сравнения для упорядочивания

```
int compare(const void *v1, const void *v2) {
    const node *n1 = *(const node **)v1;
    const node *n2 = *(const node **)v2;
    if (n1->score > n2->score)
        return -1;
    if (n1->score < n2->score)
        return 1;
    return strcmp(n1->name, n2->name);
}
```

Функции `qsort` обычно имеют сходную сигнатуру: получают два пустых указателя. Эти указатели являются постоянными `const`, поэтому элементы, на которые они указывают, изменять нельзя.

Пустые указатели должны быть преобразованы до того, как мы будем выполнять сравнения или станем иным способом обращаться к внутренним элементам. Помните, что `qsort` вызывает метод `compare` с указателями на два элемента из нашего массива, но поскольку он является массивом указателей, то в `compare` передаются указатели на указатели на элементы. Следовательно, мы сначала приводим пустые указатели к типу `const node**`, а затем применяем `*`, чтобы

получить значения для `n1` и `n2` с типом `const node**`. Теперь можно использовать `n1` и `n2` как указатели на узлы.

Мы начинаем со сравнения количеств баллов каждого узла. Эти рейтинги уже вычислены как количество потомков на расстоянии `d`. Если у `n1` потомков больше, чем у `n2`, тогда возвращается `-1`, указывая, что `n1` должен идти перед `n2`. Аналогичным образом, если `n1` будет иметь меньше потомков на расстоянии `d`, чем `n2`, вернется `1`, указывая, что `n1` должен идти после `n2`.

Таким образом, единственный вариант перехода к последней строке — это когда `n1` и `n2` будут иметь одинаковое количество потомков на расстоянии `d`. В этом случае нужно разделить совпадающие узлы по их именам. Функция `strcmp` возвращает отрицательное число, ноль или положительное число, если первая строка по алфавитному порядку меньше, равна или больше второй строки соответственно.

## Вывод информации

После упорядочивания узлов необходимо вывести их имена из начала массива `nodes`. В листинге 2.28 приведена реализующая это функция.

**Листинг 2.28.** Вывод узлов

```
void output_info(node *nodes[], int num_nodes) {
    int i = 0;
    while (i < 3 && i < num_nodes && nodes[i]->score > 0) {❶
        printf("%s %d\n", nodes[i]->name, nodes[i]->score);
        i++;
        while (i < num_nodes && ❷
                nodes[i]->score == nodes[i-1]->score) {
            printf("%s %d\n", nodes[i]->name, nodes[i]->score);
            i++;
        }
    }
}
```

Переменная `i` подсчитывает количество выведенных узлов. Внешний цикл `while` ❶ управляется тремя условиями, которые вместе определяют, нужно ли продолжать выводить узлы. Если все три условия верны, значит, нужно произвести дополнительный вывод, для чего мы входим в тело цикла `while`. Затем мы выводим информацию для текущего узла и увеличиваем `i`, чтобы перейти к рассмотрению следующего узла. Теперь, пока этот новый узел совпадает с предыдущим, нужно продолжать выводить узлы, не обращая внимания на правило «максимум три узла». Эту логику реализуют условия внутреннего цикла ❷: если выведено более трех узлов и рейтинг текущего узла совпадает с рейтингом предыдущего, мы входим в тело внутреннего цикла `while` и выводим информацию для соответствующего узла.



## Функция *main*

Осталось только объединить функции и добавить логику для обработки тестовых примеров, что мы и делаем в листинге 2.29.

**Листинг 2.29.** Функция *main*

```
#define MAX_NODES 1000

int main(void) {
    int num_cases, case_num;
    int n, d, num_nodes;
    node **nodes = malloc_safe(sizeof(node) * MAX_NODES); ❶
    scanf("%d", &num_cases);
    for (case_num = 1; case_num <= num_cases; case_num++) {
        printf("Tree %d:\n", case_num); ❷
        scanf("%d %d", &n, &d);
        num_nodes = read_tree(nodes, n);
        score_all(nodes, num_nodes, d);
        qsort(nodes, num_nodes, sizeof(node*), compare);
        output_info(nodes, num_nodes);
        if (case_num < num_cases) ❸
            printf("\n");
    }
    return 0;
}
```

Здесь мы начинаем с выделения указателей для максимального количества узлов, которое может встретиться в тестовом примере ❶. Далее идет считывание количества тестовых примеров и однократный перебор каждого. Напомню, что вывод для каждого примера должен состоять из двух частей: информации о его номере и об узлах. Первая часть обрабатывается одним вызовом `printf` ❷. При обработке второй мы опираемся на ранее созданные функции: считываем дерево, решаем задачу для каждого узла, упорядочиваем их, после чего выводим нужную информацию.

В конце кода идет проверка, находимся ли мы в последнем тестовом примере ❸. При этом обеспечивается вывод пустой строки между тестами.

## Выводы

Рекурсивные решения эффективны, просты, понятны и легко проверяются... По крайней мере такое ощущение складывается, если много читаешь о рекурсии и плотно общаешься с ее поклонниками. Однако на примере своих студентов я наблюдаю многочисленные проблемы при изучении рекурсии. Для того чтобы разделить точку зрения экспертов, требуется время. Не волнуйтесь, если поначалу вам будет сложно придумывать рекурсивные решения. Проявляйте настойчивость!

Многие преподаватели и авторы книг придумали собственные подходы и примеры представления рекурсии. Я призываю вас искать дополнительные материалы по теме, чтобы расширить полученные после прочтения этой книги знания.

В следующей главе мы продолжим тему рекурсии, используя ее для другого вида задач.

## Примечания

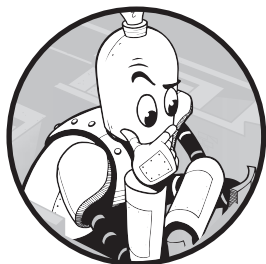
Задача «Трофеи Хэллоуина» входила в первый этап конкурса по программированию 2012 года (2012 DWITE Programming Competition).

Задача «Расстояние до потомка» взята из конкурса по программированию 2005 года (2005 ACM East Central North America Regional Programming Contest).

Рекурсия очень развернуто рассматривается в книге Эрика Робертса (Eric Roberts) *Thinking Recursively with Java* (издательство Wiley, 2005).

# 3

## Мемоизация и динамическое программирование



В этой главе мы рассмотрим четыре задачи, которые можно решить с помощью рекурсии. Мы увидим, что рекурсия только в теории считается универсальным инструментом, а на практике ее применение может привести к объемным вычислениям при том, что задача не будет решена. Однако не стоит беспокоиться: вы изучите две мощные техники — мемоизацию и динамическое программирование, которые позволят добиться

поразительного прироста в производительности, сократив время выполнения программы с часов или дней до секунд. Причем эти техники подходят для множества разнообразных задач. Если бы вы собирались прочесть всего одну главу книги, то я бы порекомендовал выбрать именно эту.

### Задача 1. Страсть к бургерам

Рассмотрим задачу с платформы UV под номером 10465.

#### Условие

Гомер Симпсон любит поесть и выпить. Ему дано  $t$  минут, чтобы съесть несколько бургеров и выпить пива. При этом есть два вида бургеров, первый из них можно съесть за  $m$  минут, а второй за  $n$  минут.

Гомер больше любит бургеры, чем пиво, поэтому он бы предпочел все  $t$  минут потратить на еду. Однако это не всегда возможно. К примеру, если  $m = 4$ ,  $n = 9$ , а  $t = 15$ , то никакая комбинация 4-минутных и 9-минутных бургеров не составит в точности 15 минут. В таком случае Гомер затратит максимально возможное время на уничтожение бургеров, а в оставшиеся минуты будет пить пиво. Задача — определить количество бургеров, которое он в итоге сможет съесть.

### Входные данные

Каждый тестовый пример представлен строкой из трех целых чисел:  $m$  — количества минут на поедание бургера первого вида;  $n$  — количества минут на поедание бургера второго вида; и  $t$  — количества минут, отведенных на бургеры и пиво. Каждое из значений  $m$ ,  $n$  и  $t$  меньше 10 000.

### Выходные данные

Для каждого тестового примера необходимо вывести следующие данные:

- Если Гомер сможет затратить ровно  $t$  минут на поедание бургеров, то следует вывести максимальное число съеденных бургеров.
- В противном случае следует вывести максимальное число съеденных бургеров за максимальное время и число оставшихся минут, в течение которых Гомер будет пить пиво.

Время на решение всех тестовых примеров  $t$  — три секунды.

### Разработка плана

Начнем с рассмотрения нескольких тестовых примеров. Вот первый:

4 9 22

В этом случае бургеры первого вида съедаются за 4 минуты ( $m = 4$ ), второго вида — за 9 минут ( $n = 9$ ), а всего Гомеру дается 22 минуты ( $t = 22$ ). Это пример, в котором Гомер может затратить все время на поедание бургеров. При этом он сможет съесть максимум три бургера. Значит, правильным выводом будет 3.

В число этих трех бургеров войдут один 4-минутный и два 9-минутных. На все про все у Гомера уйдет  $1 \times 4 + 2 \times 9 = 22$  минуты. Здесь стоит заметить, что нам не нужно выводить конкретное количество каждого вида бургеров, достаточно вывести общее. Тем не менее ниже я буду приводить количество съедаемых бургеров каждого вида, но только для того, чтобы наглядно показать верность решения.

Вот еще один тестовый пример:

4 9 54

Верным выводом здесь будет число 11, полученное как сумма девяти 4-минутных бургеров и двух 2-минутных. В отличие от предыдущего случая, здесь у Гомера есть несколько вариантов потратить все 54 минуты на поедание бургеров. К примеру, он может съесть шесть 9-минутных — это займет все 54 минуты, но нас интересует максимально возможное число бургеров, которое точно впишется в заданное время.

Как указано в условии задачи, у Гомера не всегда получится полностью уложиться в  $t$  минут, исключительно поедая бургеры. Рассмотрим очередной тестовый пример:

4 9 15

Сколько бургеров съест Гомер в этом случае? Максимум три, а именно три 4-минутных версии. Таким образом, Гомер затратит 12 минут на еду, а в оставшиеся  $15 - 12 = 3$  минуты будет пить пиво. Итак, он съедает три бургера, и при этом у него остается три минуты на пиво — решена ли задача? Нет.

Еще раз внимательно перечитайте ее условие и обратите внимание, что требуется вывести максимальное количество бургеров, которые Гомер может съесть за максимально возможное время. Это значит, что, если у него не получается полностью уложиться в отведенный промежуток времени, нужно максимизировать *время*, которое он проведет за едой. С таким условием верным выводом для данных 4 9 15 будет 2 2: первая двойка означает съедание двух бургеров (одного 4-минутного и одного 9-минутного за 13 минут), а вторая двойка относится к оставшимся двум минутам ( $15 - 13$ ), в течение которых Гомер будет пить пиво.

В примерах 4 9 22 и 4 9 54 Гомеру дается 22 и 54 минуты соответственно. В этих случаях можно потратить все время на поедание бургеров. Однако в примере 4 9 15 нет возможности истратить на еду все 15 минут. Что делать тогда?

Один из возможных способов — попробовать уложить съедание бургеров в 14 минут. Если это удастся, ответ получен, и мы сообщаем максимальное число бургеров, которое Гомер может съесть за 14 минут, и количество минут, в течение которых он будет пить пиво, то есть 1. Так мы выполним условие максимизации времени еды. Нам уже известно, что 15 минут в данном случае не подходит, поэтому следующим наилучшим вариантом становится 14.

Проверим, удастся ли заполнить все 14 минут поеданием 4-минутных и 9-минутных бургеров. А вот и нет. Как и в случае с 15 минутами, это невозможно.

Однако можно уложиться в 13 минут, съев два бургера: один 4-минутный и один 9-минутный. В итоге у Гомера останется две минуты на пиво, а мы на выходе получим те самые 2 2.

Таким образом, мы будем проверять, может ли Гомер есть бургеры на протяжении ровно  $t$  минут. Если да, то на этом все: мы выводим максимальное число бургеров, которые он успевает съесть. Если нет, то выясняем, может ли Гомер есть бургеры на протяжении всех  $t - 1$  минут. Если да, то выводим максимальное число бургеров и остаток минут на питье пива. Если же и в этом случае уложиться в заданное время не удастся, переходим к варианту с временем  $t - 2$ , затем  $t - 3$  минут и т. д., остановившись на значении, которое удастся полностью заполнить поеданием бургеров.

## Описание оптимальных решений

Рассмотрим тестовый пример 4 9 22. Нужно, чтобы любая предложенная в качестве решения комбинация бургеров и пива составляла ровно 22 минуты. При этом предпочтительным вариантом будет тот, в котором заданный промежуток времени заполняется только поеданием 4-минутных и 9-минутных бургеров. Решение, выполняющее требования задачи, называется *допустимым*. К примеру, вариант, при котором Гомер 4 минуты ест бургеры и 18 минут пьет пиво, считается допустимым. А вот вариант, в котором Гомер 8 минут ест бургеры и 18 минут пьет пиво, уже недопустим, так как  $8 + 18$  не равно 22. Также недопустим вариант, когда он 5 минут ест бургеры и 17 минут пьет пиво, потому что поедание 4-минутных и 9-минутных бургеров никак не уложить в 5 минут.

«Страсть к бургерам» относится к *задачам оптимизации*. Такие задачи подразумевают выбор *оптимального* (лучшего) решения из всех допустимых. При этом допустимых решений с разной эффективностью может быть много. Некоторые окажутся малоэффективны, например пить пиво все 22 минуты, другие оптимальны, а остальные при этом лишь близки к оптимальным и, возможно, будут отличаться от них на один или два бургера. Наша же цель — разобраться во всей этой путанице и найти оптимальное решение.

Предположим, что на поедание первого бургера первого вида уходит  $m$  минут, второго  $n$  минут и всего дано  $t$  минут.

Если  $t = 0$ , то верный вывод будет 0, потому что промежуток из нуля минут можно заполнить поеданием нуля бургеров. Следовательно, далее мы сосредоточимся на возможных действиях в случаях, когда  $t$  больше нуля.

Попробуем представить, как должно выглядеть оптимальное решение для  $t$  минут. Конечно же, мы не можем иметь конкретного представления вроде «Гомер съедает 4-минутный бургер, затем 9-минутный, потом еще один 4-минутный, затем...». Пока

для решения этой задачи мы ничего не сделали, так что о получении подобной степени детализации можно только мечтать.

Тем не менее кое-что здесь отметить можно. С одной стороны, сказанное далее может показаться абсолютной бессмыслицей, с другой же — окажется, что в его основе лежит мощная стратегия по решению множества различных задач оптимизации.

Вот о чем я. Предположим, что Гомер может потратить ровно  $t$  минут на поедание бургеров. Последний бургер, который от съест, должен быть  $m$ -минутным или  $n$ -минутным.

А разве может последний бургер быть еще каким-то? Гомер может есть только  $m$ -минутные и  $n$ -минутные бургеры, так что для последнего съедаемого бургера есть всего два варианта, как и два варианта для того, как должно выглядеть окончание оптимального решения.

Если нам известно, что последний съедаемый бургер в оптимальном решении является  $m$ -минутным, то мы знаем, что без него остается  $t - m$  минут. Нам нужно постараться заполнить эти  $t - m$  минут именно поеданием бургеров без употребления пива: помните, что по текущему условию Гомер может заполнить все  $t$  минут именно едой. Если мы сможем потратить эти  $t - m$  минут оптимально, позволив Гомеру съесть максимальное число бургеров, это будет означать нахождение оптимального решения исходной задачи для  $t$  минут. Далее мы возьмем количество бургеров, которое он может съесть за  $t - m$  минут, и добавим  $m$ -минутный бургер, чтобы заполнить оставшиеся  $m$  минут.

Далее рассмотрим вариант, в котором последним съедаемым бургером в оптимальном решении будет  $n$ -минутный. В этом случае у Гомера остается  $t - n$  минут. И снова, учитывая, что все  $t$  минут уходят на поедание бургеров, мы знаем, что можно затратить оставшиеся  $t - n$  минут именно на еду. Если мы сможем затратить  $t - n$  минут оптимально, то получим оптимальное решение для исходной задачи. В этом случае мы возьмем количество бургеров, которые он может съесть за  $t - n$  минут, и прибавим один  $n$ -минутный бургер, заполнив оставшиеся  $n$  минут.

Но не смахивает ли это на фарс? Мы только что предположили, что знаем, какой бургер будет последним. Однако знать мы этого не можем. Нам известно лишь, что последний бургер является  $m$ -минутным либо  $n$ -минутным, но каким именно — мы не знаем.

Самое замечательное состоит в том, что нам и не нужно этого знать. Можно предположить, что последним будет  $m$ -минутный бургер и решать задачу оптимизации с учетом этого выбора. Затем мы предполагаем, что последним будет  $n$ -минутный бургер, и решаем задачу оптимизации уже относительно него. В первом случае мы

решаем задачу с  $t - m$  минут. Если мы описываем решение задачи через ее подзадачи, то можно попробовать подход с рекурсией, как мы делали в главе 2.

## Решение 1. Рекурсия

Начнем с написания вспомогательной функции, находящей решение для  $t$  минут. Закончив с ней, напишем функцию, решающую задачу для  $t$  минут,  $t - 1$  минут,  $t - 2$  минут и т. д., пока не сможем полностью уложить поедание бургеров в целое количество минут, максимально близкое к  $t$ .

### Вспомогательная функция: решение для заданного числа минут

Каждая задача и подзадача описывается тремя параметрами:  $m$ ,  $n$  и  $t$ . Поэтому тело функции будет выглядеть следующим образом:

```
int solve_t(int m, int n, int t)
```

Если Гомер может потратить ровно  $t$  минут на еду, то вернется количество бургеров, которые ему удастся за это время съесть. Если же он не может потратить на бургеры ровно  $t$  минут, то есть ему придется не менее одной минуты пить пиво, — тогда вернется  $-1$ . Вывод нуля или большего числа означает, что задача была решена и Гомер обошелся без пива. Если же возвращается  $-1$ , это значит, что без пива не обойтись.

Если мы вызываем `solve_t(4, 9, 22)`, то ожидаем получить на выходе 3: три — это максимальное число бургеров, которые Гомер может съесть ровно за 22 минуты. Если же вызвать `solve_t(4, 9, 15)`, то следует ожидать возвращения  $-1$ , то есть никакая комбинация 4-минутных и 9-минутных бургеров не даст ровно 15 минут.

Мы уже разобрались с ситуацией, когда  $t = 0$ : в этом случае за ноль минут Гомер съедает ноль бургеров.

```
if (t == 0)
    return 0;
```

Это базовый случай нашей рекурсии. Для реализации оставшейся части функции нам понадобится стратегия, изложенная в предыдущем подразделе. Помните, что задачу ровно для  $t$  минут мы решаем с учетом последнего съедаемого Гомером бургера. Возможно, это  $m$ -минутный бургер. Чтобы проверить это, мы решаем подзадачу для  $t - m$  минут. Конечно же, последний бургер может быть  $m$ -минутным, только если всего дано не менее  $m$  минут. В коде эта логика прописывается следующим образом:



```
int first;
if (t >= m)
    first = solve_t(m, n, t - m);
else
    first = -1;
```

Переменная `first` используется для хранения оптимального решения подзадачи  $t - m$ , где  $-1$  означает «нет решения». Если  $t \geq m$ , то есть шанс, что последним является  $m$ -минутный бургер. Поэтому мы делаем рекурсивный вызов для вычисления оптимального числа бургеров, которое Гомер сможет съесть ровно за  $t - m$  минут. Такой рекурсивный вызов вернет число больше  $-1$ , если решение ровно за установленное время возможно, либо  $-1$ , если невозможно. Если  $t < m$ , то рекурсивный вызов не делается: мы устанавливаем `first = -1`, указывая, что  $m$ -минутный бургер не может использоваться в поиске оптимального решения для  $t$  минут.

Теперь перейдем к проверке, идет ли последним  $n$ -минутный бургер. Код для этого случая будет аналогичен предыдущему, но на этот раз в нем будет использоваться переменная `second`, а не `first`:

```
int second;
if (t >= n)
    second = solve_t(m, n, t - n);
else
    second = -1;
```

Подведем итоги:

- Переменная `first` характеризует решение для подзадачи  $t - m$ . Если она равна  $-1$ , значит, заполнить  $t - m$  минут бургерами не получится. Если же `first` имеет любое другое значение, то она сообщает оптимальное число бургеров, которые Гомер может съесть ровно за  $t - m$  минут.
- Переменная `second` характеризует решение для подзадачи  $t - n$ . Если она равна  $-1$ , значит, заполнить  $t - n$  минут бургерами нельзя. В остальных случаях она выводит оптимальное число бургеров, которое может съесть Гомер ровно за  $t - n$  минут.

Есть вероятность, что обе переменные, `first` и `second`, будут иметь значение  $-1$ . Для `first` оно будет означать, что  $m$ -минутный бургер не может быть последним. А для `second` это значит, что последним не может быть  $n$ -минутный бургер. Если последний бургер не может быть ни  $m$ -, ни  $n$ -минутным, то у нас не остается вариантов и мы вынуждены будем заключить, что за  $t$  минут невозможно съесть целое число бургеров:

```
if (first == -1 && second == -1)
    return -1;
```

И наоборот, если `first` или `second` либо обе переменные больше `-1`, значит, для `t` минут существует не менее одного решения. В этом случае мы выберем большее из `first`- и `second`-значений в качестве наилучшего решения подзадачи. Если к этому значению прибавить единицу, включив тем самым последний бургер, то мы получим оптимальное решение исходной задачи для `t` минут:

```
return max(first, second) + 1;
```

Полная функция дана в листинге 3.1.

**Листинг 3.1.** Решение для `t` минут

```
int max(int v1, int v2) {
    if (v1 > v2)
        return v1;
    else
        return v2;
}

int solve_t(int m, int n, int t) {
    int first, second;
    if (t == 0)
        return 0;
    if (t >= m)
        first = solve_t(m, n, t - m); ❶
    else
        first = -1;
    if (t >= n)
        second = solve_t(m, n, t - n); ❷
    else
        second = -1;
    if (first == -1 && second == -1)
        return -1; ❸
    else
        return max(first, second) + 1; ❹
}
```

Независимо от того, убедил ли я вас в верности этой функции, рассмотрим ее поведение на практике.

Начнем с `solve_t(4, 9, 22)`. Рекурсивный вызов `first` ❶ выдает решение для 18 минут ( $22 - 4$ ) и возвращает 2, потому что таково максимальное количество бургеров, которое Гомер может съесть ровно за 18 минут. Рекурсивный вызов `second` ❷ решает подзадачу для 13 минут ( $22 - 9$ ). При этом он также возвращает 2, потому что именно столько бургеров Гомер может съесть за 13 минут. Это значит, что и `first`, и `second` равны в данном случае 2. Прибавление же заключительного 4- или 9-минутного бургера дает для исходной 22-минутной задачи решение 3 ❹.

А теперь попробуем решить `solve_t(4, 9, 20)`. Рекурсивный вызов `first` ❶ решает подзадачу для 16 минут ( $20 - 4$ ) и дает результат 4. А рекурсивный вызов `second` ❷ решает подзадачу для 11 минут ( $20 - 9$ ), но у нас не получится затратить ровно 11 минут на поедание 4-минутных и 9-минутных бургеров. Значит, второе рекурсивное решение возвращает -1. Максимальным значением из `first` и `second` в таком случае будет 4 (значение `first`), и в итоге возвращается 5 ❹.

Мы рассмотрели пример, в котором оба рекурсивных вызова давали решения подзадач с одинаковым числом бургеров, и пример, где решение подзадачи давал только один рекурсивный вызов. Теперь же рассмотрим случай, в котором каждый рекурсивный вызов возвращает решение подзадачи, но при этом одно из решений оказывается лучше. Возьмем `solve_t(4, 9, 36)`. Рекурсивный вызов `first` ❶ дает 8, то есть максимальное число бургеров, которое Гомер съест ровно за 32 минуты ( $36 - 4$ ). В свою очередь, рекурсивный вызов `second` ❷ дает 3, максимальное количество бургеров, которые Гомер съест за 27 минут ( $36 - 9$ ). Наибольшим значением является 8, значит, в качестве итогового решения мы возвращаем 9 ❸.

В завершение разберем `solve_t(4, 9, 15)`. Рекурсивный вызов `first` ❶ решает задачу для 11 минут ( $15 - 4$ ) и, поскольку это невозможно, возвращает -1. Рекурсивный вызов `second` ❷ даст аналогичный результат: решение для 6 минут ( $15 - 9$ ) невозможно, и он также вернет -1. Это значит, что для 15 минут решения без остатка нет, и мы возвращаем -1 ❸.

## Функции `solve` и `main`

В разделе «Разработка плана» на с. 108 указано, что если мы сможем затратить ровно  $t$  минут на еду, то выводим максимальное число съеденных бургеров. В противном случае Гомер не менее одной минуты будет пить пиво. Чтобы узнать продолжительность употребления пенного напитка, мы будем пробовать решить задачу для  $t - 1$ ,  $t - 2$ ,  $t - 3$  минут и т. д., пока не определим время, которое можно заполнить едой. К счастью, с помощью функции `solve_t` можно установить любое нужное нам значение времени. Мы начнем со значения  $t$ , а далее при необходимости перейдем к  $t - 1$ ,  $t - 2$  и т. д. Соответствующий код отражен в листинге 3.2.

### Листинг 3.2. Решение 1

```
void solve(int m, int n, int t) {
    int result, i;
    result = solve_t(m, n, t); ❶
    if (result >= 0)
        printf("%d\n", result); ❷
    else {
        i = t - 1;
        result = solve_t(m, n, i); ❸
        while (result == -1) {
```

```
        i--;  
        result = solve_t(m, n, i); ❹  
    }  
    printf("%d %d\n", result, t - i); ❺  
}  
}
```

Сначала задача решается для  $t$  минут ❶. Если мы получим результат не менее нуля, то выведем максимальное число бургеров ❷ и остановимся.

Если же Гомеру не удастся все  $t$  минут есть бургеры, то мы установим для  $i$  значение  $t - 1$ , поскольку оно является следующим оптимальным значением времени. Далее задача решается для этого нового значения  $i$  ❸. Если на выходе вернется не  $-1$ , это будет означать успех, и мы закончим цикл `while`. Если же попытка успехом не увенчается, цикл `while` будет выполняться до момента решения подзадачи. Внутри этого цикла происходит декрементирование значения  $i$  и решение подзадачи ❹. В итоге цикл `while` неизбежно завершится. В крайнем случае, время на еду сократится до нуля минут. Выход из цикла `while` будет означать нахождение наибольшего времени  $i$ , в течение которого Гомер может есть бургеры. В этот момент `result` будет содержать значение максимального числа бургеров, а  $t - i$  отражать время питья пива. Именно эти значения мы и выводим ❺.

Вот и все. Мы используем рекурсию, чтобы решить `solve_t` конкретно для  $t$ . При этом `solve_t` отлично подошла для разных тестовых примеров. Невозможность решить задачу ровно за  $t$  минут не создает проблемы: мы используем цикл внутри `solve`, поочередно перебирая значения времени в порядке убывания. Теперь нужно, чтобы функция `main` считывала входные данные и вызывала `solve`. Соответствующий код приведен в листинге 3.3.

### Листинг 3.3. Функция `main`

```
int main(void) {  
    int m, n, t;  
    while (scanf("%d%d%d", &m, &n, &t) != -1)  
        solve(m, n, t);  
    return 0;  
}
```

Вот он — момент завершенности и гармонии, когда наконец можно отправлять решение 1 на проверку. Сделайте это, а я подожду... и еще подожду... и еще...

## Решение 2. Мемоизация

Решение 1 не пройдет, не потому что оно ошибочно, а потому что слишком медленное. Если отправить его на проверку, то в ответ придет сообщение об ошибке

«Time-Limit Exceeded». Такое же сообщение мы получили для решения 1-й задачи с уникальными снежинками. В том случае неэффективность отражала проделывание лишней работы. Здесь же, как вскоре станет ясно, неэффективность заключается в выполнении не ненужных, а нужных действий, но только с излишним их повторением.

В условии задачи сказано, что  $t$  может быть любым целым числом минут менее 10 000. Конечно же, в таком случае следующий тестовый пример не должен представлять никаких проблем:

```
4 2 88
```

Значения  $m$  и  $n$ , а именно 4 и 2, очень малы. В сравнении с 10 000 значение  $t$ , равное 88, также очень мало. Вас может удивить и даже несколько расстроить то, что наш код для этого примера не впишется в отведенные на решение задачи три секунды. На моем ноутбуке она решается около 10 секунд. Представляете, 10 секунд для столь незначительного значения 88. Давайте-ка сразу для наглядности попробуем пример с чуть большим значением:

```
4 2 90
```

Здесь мы всего лишь увеличили время с 88 до 90, и на моем ноутбуке этот пример выполняется около 18 секунд — почти вдвое дольше, чем предыдущий со значением 88! Если же увеличить значение до 92, то время выполнения снова почти удвоится, и так далее по возрастающей. Неважно, насколько быстр выполняющий вычисления компьютер, вы никогда не уложите во временной лимит задачи даже для значения 100. Если экстраполировать эту тенденцию, то будет сложно даже вообразить, сколько времени займет выполнение кода для тестового примера, где  $t$  будет выражено тысячами. Алгоритм, в котором фиксированное инкрементирование размера задачи ведет к удвоению времени выполнения, называется *экспоненциальным* или алгоритмом, выполняющимся за экспоненциальное время.

Мы установили, что наш код работает медленно, но почему? В чем кроется его неэффективность?

Рассмотрим пример с  $m, n, t$ . Функция `solve_t` содержит три параметра, но изменяется при этом только третий, то есть  $t$ . Таким образом, есть только  $t + 1$  различных способов вызова `solve_t`. Например, если  $t$  в тестовом примере равна 4, тогда вызовами, которые можно сделать к `solve_t`, станут те, в которых значения  $t$  будут равны 4, 3, 2, 1 и 0. Когда мы вызываем `solve_t` с некоторым  $t$ , например 2, то нет смысла повторять такой вызов снова: мы уже получили ответ, и бессмысленно выполнять рекурсивный вызов для повторных вычислений.

### Отсчет вызовов функции

Я собираюсь добавить в решение 1 код, подсчитывающий количество вызовов `solve_t`. Новые версии функций `solve_t` и `solve` приведены в листинге 3.4. В них добавлена глобальная переменная `total_calls`, инициализируемая на входе `solve` как 0, которая увеличивается на 1 при каждом вызове `solve_t`. Тип переменной установлен как `long long`, поскольку `long` или `int` не позволят охватить взрывной рост вызовов функции.

#### Листинг 3.4. Доработанное решение 1

```
unsigned long long total_calls;

int solve_t(int m, int n, int t) {
    int first, second;
    total_calls++; ❶
    if (t == 0)
        return 0;
    if (t >= m)
        first = solve_t(m, n, t - m);
    else
        first = -1;
    if (t >= n)
        second = solve_t(m, n, t - n);
    else
        second = -1;
    if (first == -1 && second == -1)
        return -1;
    else
        return max(first, second) + 1;
}

void solve(int m, int n, int t) {
    int result, i;
    total_calls = 0; ❷
    result = solve_t(m, n, t);
    if (result >= 0)
        printf("%d\n", result);
    else {
        i = t - 1;
        result = solve_t(m, n, i);
        while (result == -1) {
            i--;
            result = solve_t(m, n, i);
        }
        printf("%d %d\n", result, t - i);
    }
    printf("Total calls to solve_t: %llu\n", total_calls); ❸
}
```

При запуске `solve_t` переменная `total_calls` увеличивается на 1 ❶, регистрируя вызов функции. В `solve` мы инициализируем `total_calls` с нуля ❷, чтобы перед обработкой очередного тестового примера счетчик вызовов обнулялся. Для каждого примера код выводит количество вызовов `solve_t` ❸.

Запустим этот код со следующими входными данными:

```
4 2 88
4 2 90
```

На выходе мы получим:

```
44
Total calls to solve_t: 2971215072
45
Total calls to solve_t: 4807526975
```

Мы совершили миллиарды бесполезных вызовов, при том что 88 и 90 отличаются всего на две единицы. Становится очевидным, что одни и те же подзадачи решаются огромное число раз.

### Запоминание ответов

Немного разберемся в природе происхождения такого количества вызовов. Предположим, что мы вызываем `solve_t(4, 2, 88)`. Она делает два рекурсивных вызова: `solve_t(4, 2, 86)` и `solve_t(4, 2, 84)`. Пока все хорошо. А теперь посмотрим, что произойдет с вызовом `solve_t(4, 2, 86)`. Она совершит два собственных рекурсивных вызова, первый из которых будет к `solve_t(4, 2, 84)` — в точности как один из сделанных `solve_t(4, 2, 88)`. Получается, что работа `solve_t(4, 2, 84)` продлевается дважды, хотя хватило бы и одного раза.

Тем не менее бессмысленное повторение на этом лишь начинается. Рассмотрим два вызова `solve_t(4, 2, 84)`. По той же логике каждый из них ведет к двум вызовам `solve_t(4, 2, 80)`, то есть всего их получается четыре. Но и здесь вполне хватило бы одного.

Что ж, его бы хватило, если бы можно было сохранить результат первого вычисления. Если запоминать результат вызова `solve_t` при первом вычислении, то можно будет обращаться к нему позже, когда этот ответ вновь понадобится.

*Запоминать, а не повторно получать* — таков принцип техники, известной как *мемоизация*. Этот термин происходит от слова «мемоизовать» — сохранять или запоминать. Слово, конечно, не самое благозвучное, но используется повсеместно.

Применение мемоизации включает два шага:

1. Объявление массива с размером, достаточным для хранения решений всех возможных подзадач. В задаче с бургерами  $t$  может иметь значение до 10 000, значит, массива под 10 000 элементов будет достаточно. Как правило, такой массив именуется **мемо**. Элементы **мемо** инициализируются как не содержащие значения.
2. В начале рекурсивной функции добавляется код для проверки того, было ли ранее получено решение для текущего варианта задачи. Сюда входит проверка соответствующего индекса **мемо**: если он не содержит значения, значит, данную подзадачу нужно решать, в противном случае там находится ответ, который просто возвращается без выполнения рекурсии. При каждом решении новой подзадачи ее результат сохраняется в **мемо**.

Расширим решение 1, добавив мемоизацию.

### Реализация мемоизации

Для объявления и инициализации массива **мемо** хорошо подходит функция `solve`, поскольку именно она задействуется первой. Для представления неизвестного значения мы используем -2: положительные числа использовать нельзя, так как они обозначают количество бургеров, а -1 уже зарезервировано для «отсутствия возможного решения». Обновленная функция `solve` приведена в листинге 3.5.

#### Листинг 3.5. Решение 2 с мемоизацией

```
#define SIZE 10000

void solve(int m, int n, int t) {
    int result, i;
    int memo[SIZE]; ❶
    for (i = 0; i <= t; i++)
        memo[i] = -2;
    result = solve_t(m, n, t, memo);
    if (result >= 0)
        printf("%d\n", result);
    else {
        i = t - 1;
        result = solve_t(m, n, i, memo);
        while (result == -1) {
            i--;
            result = solve_t(m, n, i, memo);
        }
        printf("%d %d\n", result, t - i);
    }
}
```



Размер, с которым объявляется массив `memo`, определяется наибольшими возможными размерами тестовых примеров ❶. Затем в цикле от 0 до `t` значение каждого элемента устанавливается как -2.

Также сделано небольшое, но важное изменение в вызовах `solve_t`. Теперь `solve_t` проверяет массив `memo` и определяет, решалась ли ранее текущая подзадача. Если нет, в него будет записано соответствующее решение.

Новый код `solve_t` приведен в листинге 3.6.

**Листинг 3.6.** Решение с мемоизацией для `t` минут

```
int solve_t(int m, int n, int t, int memo[]) {
    int first, second;
    if (memo[t] != -2) ❶
        return memo[t];
    if (t == 0) {
        memo[t] = 0;
        return memo[t];
    }
    if (t >= m)
        first = solve_t(m, n, t - m, memo);
    else
        first = -1;
    if (t >= n)
        second = solve_t(m, n, t - n, memo);
    else
        second = -1;
    if (first == -1 && second == -1) {
        memo[t] = -1;
        return memo[t];
    } else {
        memo[t] = max(first, second) + 1;
        return memo[t];
    }
}
```

Общий план решения 1 остается тем же, что и в листинге 3.1: если `t` равно 0, решается базовый случай; в противном случае задача решается для `t - m` и `t - n` минут, после чего выбирается лучшее решение.

К этой структуре добавлена мемоизация. Существенное сокращение времени расчетов достигается благодаря проверке, находится ли решение для `t` в массиве `memo` ❶; если да, то возвращается результат этого решения. Рекурсии в этом случае нет. Вместо нее происходит мгновенный возврат из функции.

Если же решение в `memo` не найдено, то начинается работа по старому алгоритму, за исключением того, что при каждом выводе решения оно сначала сохраняется

в мемо. Перед каждой из инструкций `return` мы сохраняем возвращаемое значение в мемо.

### Проверяем мемоизацию

Выше было показано, что решение 1 обречено на провал, поскольку даже небольшие примеры выполняются слишком долго и эта медлительность обусловлена чрезмерным количеством вызовов функций. Каковы же результаты решения 2?

Опробуем его на входных данных, которые рассматривали в решении 1:

```
4 2 88
4 2 90
```

На моем ноутбуке время выполнения оказывается ничтожно малым.

Сколько же происходит вызовов функции? Давайте, как мы это уже делали в решении 1, добавим код для подсчета числа вызовов. В этом случае для тех же входных данных мы получим следующий вывод:

```
44
Total calls to solve_t: 88
45
Total calls to solve_t: 90
```

88 вызовов, когда  $t$  равно 88, и 90 при  $t$ , равном 90. Разница между решениями 2 и 1 просто колоссальна. Здесь мы перешли от экспоненциального алгоритма к линейному. В частности, теперь мы получили алгоритм вида  $O(t)$ , где  $t$  — это количество минут, заданное в тестовом примере.

А теперь время судей. Если отправить решение 2 на проверку, то оно благополучно пройдет все тесты.

Это определенно знаковый рубеж, но еще не последнее слово в истории о Гомере и его любимых бургерах.

## Решение 3. Динамическое программирование

Мы перекинем мост от мемоизации к динамическому программированию, прояснив цель использования рекурсии в решении 2. Рассмотрим код `solve_t` из листинга 3.7. Он аналогичен коду в листинге 3.6, но теперь мы обратим внимание на два рекурсивных вызова.

**Листинг 3.7.** Решение для  $t$  минут с акцентом на рекурсивных вызовах

```
int solve_t(int m, int n, int t, int memo[]) {
    int first, second;
```

```
if (memo[t] != -2)
    return memo[t];
if (t == 0) {
    memo[t] = 0;
    return memo[t];
}
if (t >= m)
    first = solve_t(m, n, t - m, memo); ❶
else
    first = -1;
if (t >= n)
    second = solve_t(m, n, t - n, memo); ❷
else
    second = -1;
if (first == -1 && second == -1) {
    memo[t] = -1;
    return memo[t];
} else {
    memo[t] = max(first, second) + 1;
    return memo[t];
}
}
```

В первом рекурсивном вызове ❶ произойдет одно из двух действий. Первый вариант — если решение подзадачи находится в `memo`, функция его возвращает. Второй вариант — когда решения в `memo` нет и осуществляются рекурсивные вызовы. Аналогичным образом работает и второй рекурсивный вызов ❷.

Когда при совершении рекурсивного вызова решение подзадачи находится в массиве `memo`, возникает вопрос: а зачем вообще делать рекурсивный вызов? Единственное, что он выполнит, это проверит `memo` и вернет результат, что можно сделать и иными средствами. Если же решение подзадачи в массиве не находится, то без рекурсивного вызова уже не обойтись.

Предположим, что можно организовать все так, что массив `memo` будет всегда содержать очередное искомое решение подзадачи. Нас интересует оптимальное решение при `t`, равном 5? Оно находится в `memo`. А что насчет `t`, равном 18? Оно тоже есть в `memo`. Если решения подзадач будут всегда находиться в `memo`, рекурсивный вызов вообще не понадобится. Можно будет просто находить эти решения.

Здесь мы и встречаемся с отличием мемоизации от динамического программирования. Функция, использующая мемоизацию, делает рекурсивный вызов для решения подзадачи. Возможно, эта подзадача уже решена, а может, и нет — независимо от этого, на момент возвращения ответа она окажется решенной. Функция, использующая *динамическое программирование*, организует работу таким образом, чтобы к моменту, когда нужен ответ для подзадачи, она была уже решена. В таком случае повода для применения рекурсии не остается: мы просто ищем решение.

В случае мемоизации рекурсия требуется для решения подзадачи, а динамическое программирование избавляет от необходимости применять в подзадаче рекурсию.

Наше решение с динамическим программированием обходится без функции `solve_t` и систематически решает подзадачи для всех значений `t` в `solve`. Код приведен в листинге 3.8.

**Листинг 3.8.** Решение 3 с динамическим программированием

```
void solve(int m, int n, int t) {
    int result, i, first, second;
    int dp[SIZE];
    dp[0] = 0; ❶
    for (i = 1; i <= t; i++) {
        if (i >= m) ❷
            first = dp[i - m]; ❸
        else
            first = -1;
        if (i >= n) ❹
            second = dp[i - n];
        else
            second = -1;
        if (first == -1 && second == -1)
            dp[i] = -1; ❺
        else
            dp[i] = max(first, second) + 1; ❻
    }

    result = dp[t]; ❼
    if (result >= 0)
        printf("%d\n", result);
    else {
        i = t - 1;
        result = dp[i];
        while (result == -1) {
            i--;
            result = dp[i]; ❸
        }
        printf("%d %d\n", result, t - i);
    }
}
```

Традиционное имя для массива динамического программирования — `dp`. Можно было бы назвать его и `memo`, поскольку он служит той же цели, что и при мемоизации, но мы будем придерживаться негласного правила и используем имя `dp`. После объявления массива мы решаем базовую задачу, явно записывая, что оптимальным решением для нуля минут будет съедание нуля бургеров ❶. Далее идет цикл, управляющий порядком, в котором решаются подзадачи. Здесь их решение происходит

в порядке возрастания — от подзадач с наименьшим числом минут (1) до подзадач с наибольшим значением времени ( $t$ ). Переменная  $i$  определяет, какая именно подзадача решается. Внутри цикла происходит уже знакомая нам проверка того, есть ли смысл тестировать в качестве последнего бургера его  $m$ -минутный вариант ②. Если да, то в массиве  $dp$  происходит поиск решения для подзадачи  $i - m$  ③.

Обратите внимание, что мы просто извлекли значение из массива ③ без использования рекурсии. Это возможно, так как нам известно, что  $i - m$  меньше  $i$ , а значит, подзадача для  $i - m$  уже решалась. Именно поэтому решение подзадач происходит в порядке от меньшего к большему: более крупным подзадачам требуются решения от меньших, значит, нужно обеспечить, чтобы к этому моменту они уже имелись в массиве.

Следующая инструкция `if` ④ аналогична предыдущей ② и обрабатывает случай, когда последний бургер является  $n$ -минутным. Как и ранее, выполняется поиск решения подзадачи в массиве  $dp$ . Мы точно знаем, что подзадача  $i - n$  уже решена, потому что итерация  $i - n$  произошла до итерации  $i$ .

Теперь у нас есть решения для обеих подзадач. Осталось только сохранить оптимальное решение в  $dp[i]$  ⑤ ⑥.

После того как мы создали массив  $dp$ , решая задачи от 0 до  $t$ , можно искать и сравнивать эти решения. Таким образом мы просто ищем решение задачи для  $t$  ⑦ и выводим его, если оно существует. Если же нет, то переходим к дальнейшему поэтапному его поиску для все меньших подзадач ⑧.

Прежде чем продолжать, рассмотрим пример массива  $dp$  для следующих входных данных:

4 9 15

После решения всех подзадач этот массив будет содержать следующие данные:

Индекс	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Значение	0	-1	-1	-1	1	-1	-1	-1	2	1	-1	-1	3	2	-1	-1

Можно проследить решение каждой подзадачи, анализируя код в листинге 3.8. Например, для  $dp[0]$  максимальное количество бургеров, которое Гомер может съесть за 0 минут, равно 0 ①. В случае  $dp[1]$  — это -1, потому что оба теста, ② ④, проваливаются, и сохраняется -1 ⑤.

В качестве заключительного примера разберем процесс получения значения 3 для  $dp[12]$ . Поскольку 12 больше 4, первая проверка проходит успешно ②. Далее

`first` устанавливается как элемент массива `dp[8]` ❸, который содержит значение 2. Аналогичным образом 12 больше 9, значит, и вторая проверка тоже проходит успешно ❹, а `second` устанавливается равным `dp[3]`, который содержит значение -1. Максимальное значение переменных `first` и `second` — это 2, значит, `dp[12]` равен 3, то есть на единицу больше этого максимума ❺.

## Мемоизация и динамическое программирование

Мы решили задачу про Гомера и бургеры за четыре шага. Во-первых, описали оптимальное решение; во-вторых, написали рекурсивное решение; в-третьих, добавили мемоизацию; в-четвертых, избавились от рекурсии за счет явного решения подзадач в порядке возрастания. Эти четыре шага составляют общий план выработки решений для многих других оптимизационных задач.

### Шаг 1. Структура оптимальных решений

На первом шаге нужно разделить оптимальное решение задачи на оптимальные решения подзадач. В «Страсти к бургерам» мы сделали это, поставив вопрос, какой бургер Гомер съест последним. Будет ли это  $m$ -минутный бургер? В этом случае останется подзадача заполнения  $t - m$  минут. Если же это будет  $n$ -минутный бургер, то останется подзадача заполнения  $t - n$  минут. Конечно, мы не знаем заранее, какой бургер будет последним, но можно решить эти две подзадачи и выяснить.

В подобных рассуждениях зачастую остается неявным требование того, чтобы оптимальное решение задачи содержало не просто некоторое решение подзадач, а его *оптимальный* вариант. Давайте проясним этот момент.

Когда в задаче с бургерами предполагается, что в оптимальном решении последним бургером будет  $m$ -минутный, мы заявляем, что решение для подзадачи  $t - m$  будет частью общей задачи  $t$ . Более того, оптимальное решение для  $t$  должно включать оптимальное решение для  $t - m$ : если это условие не выполнить, то решение для  $t$  просто не окажется оптимальным, так как его можно улучшить, используя более эффективное решение для  $t - m$ . С помощью той же логики можно показать, что если последним бургером в оптимальном решении будет  $n$ -минутный, то оставшиеся  $t - n$  минут должны быть заполнены путем оптимального решения для  $t - n$ .

Позвольте прояснить все сказанное на примере. Предположим, что  $m = 4$ ,  $n = 9$  и  $t = 54$ . Значение оптимального решения равно 11. Существует оптимальное решение  $S$ , в котором последним бургером является 9-минутный. Я утверждаю, что  $S$  должно состоять из этого 9-минутного бургера при оптимальном решении

для 45 минут. Оптимальное решение для 45 минут — 10 бургеров. Если бы  $S$  использовало некое субоптимальное решение для первых 45 минут, то  $S$  не было бы оптимальным для 11 бургеров. Например, если  $b$  для первых 45 минут субоптимальное решение было 5 бургеров, то общее количество бургеров в  $S$  было бы всего 6.

Если оптимальное решение задачи складывается из оптимальных решений подзадач, мы говорим, что задача имеет *оптимальную подструктуру*. Если задача обладает оптимальной подструктурой, значит, методы решения из этой главы будут к ней применимы.

Я читал и слышал утверждения, что решение оптимизационных задач с помощью мемоизации или динамического программирования стереотипно. И, если вы видели одну такую задачу, то можете считать, что видели все, а при встрече новой достаточно будет просто «щелкнуть пальцами». Я с этим не согласен. Такая точка зрения противоречит практике реализации как начального этапа описания структуры оптимальных решений, так и определения, окажется ли выбранная структура эффективной.

В этой главе мы будем ориентироваться на наш пошаговый план и решим несколько дополнительных задач с помощью мемоизации и динамического программирования. Я уверен, что для освоения огромного спектра задач, которые можно решать с помощью этих подходов, требуется практическая тренировка и умение делать обобщения на основе как можно большего количества разных задач.

## Шаг 2. Рекурсивное решение

Шаг 1 предполагает, что не только мемоизация и динамическое программирование могут привести к решению; он оставляет возможность применения также и рекурсивного подхода. Попробуйте найти оптимальное решение с помощью каждой из этих возможностей, решая подзадачи с помощью рекурсии. В задаче с бургерами мы приняли, что оптимальное решение для  $t$  минут может состоять из  $m$ -минутного бургера и оптимального решения для  $t - m$  минут либо из  $n$ -минутного бургера и оптимального решения для  $t - n$  минут. В связи с этим необходимо решить подзадачи  $t - m$  и  $t - n$ , а поскольку они меньше, чем  $t$ , для них можно использовать рекурсию. Как правило, количество рекурсивных вызовов зависит от числа кандидатов, соревнующихся на звание оптимального решения.

## Шаг 3. Мемоизация

Если мы преуспеем на предыдущем шаге, значит, у нас будет верное решение задачи. Хотя, как мы видели из задачи про бургеры, такое решение может потребовать совершенно неоправданных затрат времени на выполнение. Причина в том, что

одни и те же подзадачи решаются снова и снова, порождая явление, называемое *наложением подзадач*. На самом деле, не будь проблемы с наложением, можно было бы остановиться и здесь: рекурсия нас вполне бы устроила. Вспомните главу 2 и две задачи, которые в ней решались. Тогда мы вполне обошлись рекурсией, которая сработала эффективно, потому что каждая подзадача решалась по одному разу. К примеру, в задаче про сбор конфет мы вычисляли общее количество конфет в дереве. Две подзадачи выясняли общее число конфет в левом и правом поддеревьях. В том случае они не зависели друг от друга, то есть для решения подзадачи левого поддерева не требовались данные из правого и наоборот.

Если наложения подзадач нет, можно ограничиться использованием рекурсии. Если же наложение присутствует, настанет черед мемоизации. Как мы видели в задаче с Гомером, мемоизация подразумевает сохранение ответа подзадачи при ее первичном решении. Если этот ответ понадобится в дальнейшем, его находят в массиве, но не вычисляют повторно. Да, подзадачи по-прежнему накладываются, но решаются они в этом случае только один раз.

## Шаг 4. Динамическое программирование

Скорее всего, решение, полученное на третьем шаге, окажется достаточно быстрым. Такое решение продолжает использовать рекурсию, но уже без риска повторения одной и той же работы. Однако, как будет показано ниже, иногда требуется избавиться от рекурсии. Добиться этого можно за счет систематического решения подзадач в порядке их возрастания от самых малых до самых больших. Это и называется динамическим программированием — использование цикла вместо рекурсии для явного последовательного решения подзадач.

Так что же лучше: мемоизация или динамическое программирование? Для многих задач они примерно равнозначны, и в таких случаях можно выбирать, исходя из личных предпочтений. Я предпочитаю мемоизацию. Далее в главе будет пример (задача 3), где таблицы `memo` и `dp` имеют более двух измерений. В подобных задачах я зачастую с трудом нахожу все базовые случаи и определяю границы таблицы `dp`.

Мемоизация используется по мере необходимости. Рассмотрим тестовый пример из «Страсти к бургерам», в котором у нас есть 2-минутный и 4-минутный бургеры и 90 минут времени. Решение с мемоизацией не будет обрабатывать нечетные значения времени, такие как 89, 87 или 85 минут, потому что такие подзадачи не возникнут при вычитании из 90 произведений двоек или четверок. Динамическое программирование, наоборот, решает все подзадачи до достижения 90. Кажется, что мемоизация должна обеспечивать преимущества: в самом деле, если огромные пространства подзадач не используются, тогда мемоизация будет быстрее динамического программирования. Но нужно учесть дополнительную нагрузку, возникающую при использовании рекурсии со всеми вызовами и возвращениями



из функций. Если вам захочется выяснить истину, то просто напишите код для обоих вариантов решения и сравните их скорость.

Нередко разработчики описывают решения с мемоизацией как *нисходящие*, а решения с динамическим программированием как *восходящие*. «Нисходящими» они называются, потому что для решения больших подзадач мы рекурсивно переходим вниз к меньшим подзадачам. В «восходящих» же решениях мы, наоборот, начинаем снизу — с меньших подзадач — и продвигаемся вверх.

Лично меня увлекают и мемоизация, и динамическое программирование. С их помощью можно решать множество задач разных видов. Мне неизвестны другие столь же эффективные методы построения алгоритмов. Многие инструменты, которые мы изучаем в этой книге, например хеш-таблицы из главы 1, позволяют ускорять вычисления. Но задачи можно решать вообще без этих инструментов — конечно, не за то время, которое устанавливается при соревнованиях, но все равно достаточно быстро для эффективного практического применения. Тем не менее мемоизация и динамическое программирование имеют важные преимущества. Они улучшают рекурсивные подходы, преобразуя на удивление медленные алгоритмы в невероятно быстрые. Надеюсь, что в оставшейся части главы мне удастся увлечь вас этими методами, так чтобы интерес сохранился и после завершения чтения.

## Задача 2. Экономные покупатели

В «Страсти к бургерам» нам удалось решить задачу, рассматривая только по две подзадачи. Здесь же мы увидим, что каждая подзадача может требовать более тщательной проработки.

Рассмотрим задачу с платформы UV под номером 10980.

### Условие

Вы хотите купить яблоки в магазине. На яблоки установлена фиксированная цена, к примеру \$1.75 за штуку. Помимо этого, у магазина есть  $m$  ценовых схем, в каждой из которых установлена стоимость  $p$  за  $n$  яблок. Например, в одной схеме три яблока стоят \$4.00, в другой схеме два яблока стоят \$2.50. Вам нужно купить *не менее*  $k$  яблок по наименьшей цене.

### Входные данные

Каждый тестовый пример состоит из следующих строк:

- Цена за одно яблоко и число ценовых схем  $m$  для данного примера. Максимальное значение  $m$  равно 20.

- $m$  строк, в каждой из которых указано число яблок  $n$  и их общая стоимость  $p$ . Значение  $n$  находится в диапазоне между 1 и 100.
- Строки с числами  $k$ , указывающими количество яблок, которое нужно купить. Их значения находятся между 0 и 100.

Каждая цена во входных данных представлена числом с плавающей точкой и имеет два десятичных знака.

В описании задачи я привел в качестве примера цены за яблоко \$1.75. Помимо этого, приведены две ценовые схемы: три яблока за \$4.00 и два яблока за \$2.50. Предположим, что нужно определить минимальные стоимости при покупке не менее одного и при покупке не менее четырех яблок. Вот входные данные для этого примера:

```
1.75 2
3 4.00
2 2.50
1 4
```

### Выходные данные

Для каждого тестового примера требуется вывести следующее:

- Номер примера **Case c:**, где  $c$  является номером тестового примера, отсчет номеров начинается с 1.
- Для каждого числа  $k$  — фразу **Buy  $k$  for \$ $d$**  (купить  $k$  за \$ $d$ ), где  $d$  будет самым дешевым вариантом покупки не менее  $k$  яблок.

Вот каким будет вывод для приведенных выше входных данных:

```
Case 1:
Buy 1 for $1.75
Buy 4 for $5.00
```

Время на решение всех тестовых примеров — три секунды.

### Описание оптимального решения

Условие задачи гласит, что нужно купить *не менее*  $k$  яблок по минимальной цене. Это означает, что покупка  $k$  яблок не является единственным вариантом: можно купить и больше, если это окажется дешевле. Начнем с попытки решить задачу для  $k$  яблок во многом аналогично тому, как мы делали в задаче с бургерами. Тогда мы нашли способ при необходимости переходить от ровно  $t$  минут к их меньшему числу. Будем надеяться, что нам удастся проделать аналогичное и здесь, начав

с  $k$  яблок и определив самую низкую стоимость для  $k, k + 1, k + 2$  и т. д. Если ничего вдруг не сломается...

Прежде чем, вспомнив название этой главы, перейти к мемоизации и динамическому программированию, мы убедимся, что эти инструменты нам действительно нужны.

Что лучше: купить три яблока за \$4.00 (схема 1) или два за \$2.50 (схема 2)? На этот вопрос можно попробовать ответить, вычислив стоимость одного яблока для каждой из предложенных схем. В схеме 1 получится  $\$4.00/3 = \$1.33$  за яблоко, а в схеме 2 каждое яблоко выйдет по  $\$2.50/2 = \$1.25$ . Похоже, что схема 2 выгоднее схемы 1. Давайте также предположим, что можем просто купить одно яблоко за \$1.75. Таким образом, у нас получится последовательность чисел от наименьшей до самой высокой стоимости яблока: \$1.25, \$1.33, \$1.75.

Теперь представим, что хотим купить *ровно*  $k$  яблок. Как это будет выглядеть в алгоритме: на каждом шаге брать самое дешевое яблоко, пока не будет куплено  $k$  яблок?

Если бы мы хотели купить ровно четыре яблока в предложенном примере, то начали бы со схемы 2, потому что она позволяет приобрести их по наилучшей цене за штуку. Использование схемы 2 один раз означает покупку двух яблок за \$2.50, после чего остается купить еще два. Можно снова использовать эту же схему и приобрести еще пару яблок, потратив дополнительно \$2.50. В этом случае мы заплатим за четыре яблока \$5.00 и получим лучший результат.

Обратите внимание, что интуитивной оценки алгоритма и его эффективности в одном тестовом примере недостаточно, чтобы сделать заключение о том, насколько он верен. Алгоритм, основанный на использовании наименьшей стоимости яблока, имеет недостаток, что можно доказать на примерах. Попробуйте самостоятельно найти такие случаи, прежде чем продолжать чтение.

Рассмотрим такой пример. Предположим, что мы хотим купить три яблока, а не четыре. Мы снова начинаем со схемы 2, которая дает нам два яблока за \$2.50. Теперь нужно купить всего одно яблоко, и единственный вариант — это заплатить за него \$1.75. Тогда общая стоимость составит \$4.25. Но есть способ получше — нужно использовать схему 1, затратив всего \$4.00. Так мы избавимся от необходимости платить высокую цену за одно дополнительное яблоко.

Возникает желание дополнить наш алгоритм корректирующими его правилами. Например, «если существует ценовая схема ровно для нужного количества яблок, то следует использовать ее». Однако можно легко опровергнуть этот доработанный алгоритм, если добавить во входные данные несколько схем, в которых магазин продает яблоки по сильно завышенным ценам, например три яблока за \$100.00.

При использовании мемоизации и динамического программирования в поиске наилучшего решения мы пробуем все возможные варианты, после чего выбираем наилучший. Вспомните задачу с бургерами. Должен ли Гомер закончить трапезу на  $m$ -минутном или  $n$ -минутном бургере? Ответа мы не знаем, поэтому пробуем оба варианта. В противоположность этому, *жадный алгоритм* не перебирает большое число вариантов, а пробует только один. Использование наилучшей цены яблока, как мы сделали выше, является примером жадного алгоритма, потому что на каждом шаге он выбирает действие без рассмотрения других вариантов. Иногда жадные алгоритмы работают. Более того, поскольку они зачастую выполняются быстрее и легче реализуются, то могут быть более эффективными, чем алгоритмы с динамическим программированием. Для данной же задачи жадные алгоритмы — как опробованный выше, так и любые другие — недостаточно эффективны.

В задаче с бургерами мы рассуждали, что если можно есть бургеры в течение ровно  $t$  минут, то последним бургером в оптимальном решении окажется  $n$ -минутный или  $m$ -минутный. Подобным образом, для текущей задачи нам нужно предположить, что оптимальное решение для покупки  $k$  яблок должно оканчиваться одним из нескольких возможных вариантов. Вот утверждение: если доступной ценовой схемой является схема 1, схема 2, ..., схема  $m$ , тогда последним нашим действием должно быть использование одной из этих  $m$  ценовых схем. Других вариантов у нас нет, ведь так?

Вообще-то не совсем. Последним действием в оптимальном решении может стать покупка и одного яблока. Этот вариант доступен нам всегда. Вместо того чтобы решать две подзадачи, как в разделе с бургерами, мы решим  $m + 1$  подзадач: по одной для каждой из  $m$  ценовых схем и дополнительную для покупки одного яблока.

Предположим, что оптимальное решение для покупки  $k$  яблок оканчивается тратой  $p$  долларов на  $n$  яблок. Тогда нужно купить  $k - n$  яблок и прибавить их стоимость к  $p$ . При этом важно проверить, что общее оптимальное решение для  $k$  яблок содержит в себе оптимальное решение для  $k - n$  яблок. Таково требование к оптимальной подструктуре для мемоизации и динамического программирования. Как и в «Страсти к бургерам», оптимальная подструктура необходима. Если решение для  $k$  не использует оптимальное решение для  $k - n$ , тогда оно не сможет быть оптимальным, так как не окажется в той же степени эффективным, в какой было бы при использовании оптимального решения для  $k - n$ .

Конечно же, мы заранее не знаем, какое действие предпримем в решении последним, чтобы оно оказалось оптимальным. Используем ли мы схему 1, схему 2, схему 3 или купим одно яблоко? Кто вообще это знает? Как и в любом алгоритме с мемоизацией или динамическим программированием, мы пробуем все варианты и выбираем наилучший.

Прежде чем рассмотреть рекурсивное решение, отмечу, что для любого числа  $k$  можно найти способ купить ровно  $k$  яблок. Неважно, будет ли это одно, два, пять или любое другое их число, можно купить именно столько. Причина в том, что у нас всегда есть вариант покупки одного яблока и его можно повторить любое количество раз.

Сравните это с задачей про бургеры, где были такие значения  $t$ , которые не удавалось без остатка заполнить едой. Вследствие такого различия задач здесь нам не придется беспокоиться о случае, в котором рекурсивный вызов для подзадачи не сможет найти решение.

## Решение 1. Рекурсия

Как и в задаче с бургерами, первым делом нужно написать вспомогательную функцию.

### Вспомогательная функция: решение для заданного числа яблок

Напишем функцию `solve_k`, которая будет аналогична `solve_t` из задачи «Страсть к бургерам». Вот ее заголовок:

```
double solve_k(int num[], double price[], int num_schemes,
               double unit_price, int num_items)
```

Ее параметры:

- **num** — массив чисел яблок в ценовых схемах, по одному элементу для каждой схемы. К примеру, если даны две ценовые схемы, первая для трех яблок и вторая для двух, то массив будет выглядеть как `[3, 2]`.
- **price** — массив цен, по одному элементу для каждой схемы. К примеру, если даны две ценовые схемы, первая со стоимостью `4.00` и вторая со стоимостью `2.50`, то массив будет выглядеть как `[4.00, 2.50]`. Обратите внимание, что **num** и **price** вместе предоставляют исчерпывающую информацию о ценовых схемах.
- **num\_schemes** — количество ценовых схем. Это значение `m` из тестового примера.
- **unit\_price** — цена одного яблока.
- **num\_items** — количество яблок, которое нужно купить.

Функция `solve_k` возвращает минимальную цену `num_items` яблок.

Код `solve_k` приведен в листинге 3.9. При ознакомлении с ним я настоятельно рекомендую вам сравнить его с `solve_t` из листинга 3.1 задачи про бургеры. Какие

отличия вы сможете найти? Чем они обусловлены? Решения с мемоизацией и динамическим программированием используют одинаковую структуру кода. Если нам удастся освоить эту структуру, то можно будет сосредоточиться на отличиях и специфике каждой задачи.

**Листинг 3.9.** Решение для num\_items элементов

```
double min(double v1, double v2) { ❶
    if (v1 < v2)
        return v1;
    else
        return v2;
}

double solve_k(int num[], double price[], int num_schemes,
               double unit_price, int num_items) {
    double best, result;
    int i;
    if (num_items == 0) ❷
        return 0; ❸
    else {
        result = solve_k(num, price, num_schemes, unit_price, ❹
                        num_items - 1);
        best = result + unit_price; ❺
        for (i = 0; i < num_schemes; i++)
            if (num_items - num[i] >= 0) { ❻
                result = solve_k(num, price, num_schemes, unit_price, ❼
                                num_items - num[i]);
                best = min(best, result + price[i]); ❽
            }
        return best;
    }
}
```

Код начинается с небольшой функции min ❶: она нужна для сравнения решений и выбора наименьшего. В задаче с бургерами мы использовали аналогичную функцию max, так как искали максимальное количество бургеров. Здесь же нас интересует минимальная стоимость. Оптимизационные задачи могут быть задачами как по *максимизации* («Страсть к бургерам»), так и по *минимизации* («Экономные покупатели») — внимательно читайте условия, чтобы убедиться в правильности выбора направления оптимизации.

Что мы делаем, если нужно решить задачу для 0 яблок ❷? Возвращаем 0 ❸, потому что минимальная стоимость покупки нуля яблок равна \$0.00. Базовыми случаями являются ноль затраченных на еду минут для задачи с бургерами и ноль купленных яблок в текущей задаче. Как всегда при применении рекурсии, любой задаче по оптимизации требуется хотя бы один базовый случай.

Если условие базового случая не выполняется, значит, `num_items` является положительным целым числом и потребуется найти оптимальный способ покупки ровно такого количества яблок. Для отслеживания наилучшего найденного на данный момент варианта (наименьшей стоимости) используется переменная `best`.

Один из способов — это оптимально решать задачу для `num_items - 1` яблок ❹ и прибавлять стоимость конечного яблока ❺.

Теперь становится очевидным значительное структурное отличие этой задачи от задачи с бургерами, которое заключается в наличии цикла внутри рекурсивной функции. В «Страсти к бургерам» цикл не требовался, потому что решить нужно было всего две подзадачи. Мы просто пробовали первую и следом вторую. Здесь же каждой ценовой схеме соответствует своя подзадача, и нужно перебрать их все.

Мы выполняем проверку, может ли вообще использоваться текущая ценовая схема ❻: если ее количество яблок не больше нужного, то схему можно пробовать. Далее совершается рекурсивный вызов для решения подзадачи, которая получается после вычитания количества яблок, соответствующего данной ценовой схеме ❼ (это действие аналогично более раннему рекурсивному вызову, в котором мы отнимали единицу ❹). Если решение этой подзадачи в сумме с ценой текущей схемы оказывается на данный момент лучшим вариантом, `best` соответствующим образом обновляется ❸.

## Функция `solve`

Мы добились оптимального решения для  $k$  яблок, но в условии задачи есть одна деталь, с которой мы еще не разобрались: «Нам нужно купить *не менее*  $k$  яблок, затратив минимальное количество средств». Почему вообще важна эта разница между ровно  $k$  яблоками и «не менее  $k$  яблок»? Сможете ли вы найти тестовый пример, в котором более  $k$  яблок окажутся дешевле, чем  $k$ ?

Вот один такой вариант. Предположим, что одно яблоко стоит \$1.75. Даны две ценовые схемы: по схеме 1 можно купить четыре яблока за \$3.00, а по схеме 2 два яблока стоят \$2.00. Нам нужно купить не менее трех яблок. Входные данные этого примера выглядят так:

```
1.75 2
4 3.00
2 2.00
3
```

Самый дешевый способ купить три яблока — это потратить \$3.75: \$1.75 за одно яблоко и \$2.00 за два согласно схеме 2. Однако можно потратить и меньше, если

купить сразу четыре яблока, а не три. Для этого нужно просто воспользоваться ценовой схемой 1, что будет стоить всего \$3.00. Таким образом, верный вывод для данного кейса будет:

```
Case 1:  
Buy 3 for $3.00
```

Хотя на деле покупается четыре яблока, а не три, вывод `Buy 3` будет верен. По условиям задачи всегда выводится минимальное число яблок, которое требуется купить, независимо от того, сколько было куплено по факту с целью экономии денег.

Далее нам понадобится функция `solve`, подобная той, что была в листинге 3.2 задачи про бургеры. Тогда мы перебирали значения по убыванию, пока не находили решение. Здесь же мы будем перебирать значения по возрастанию, отслеживая по ходу процесса минимум. Вот первый вариант кода:

```
double solve(int num[], double price[], int num_schemes,  
             double unit_price, int num_items) {  
    double best;  
    int i;  
    best = solve_k(num, price, num_schemes, ❶  
                  unit_price, num_items);  
    for (i = num_items + 1; i < ???; i++) ❷  
        best = min(best, solve_k(num, price, num_schemes,  
                                unit_price, i));  
    return best;  
}
```

Мы инициализируем `best` с оптимальным числом для покупки ровно `num_items` яблок ❶. Далее с помощью цикла `for` выполняется перебор количества яблок по возрастанию ❷. Цикл `for` останавливается, когда... Так. А откуда нам знать, что уже можно остановиться? Возможно, нужно купить 3 яблока, но самым дешевым вариантом будет покупка 4, 5, 10 или даже 20. Такой проблемы в задаче с бургерами не было, потому что там мы спускались вниз по направлению к нулю, а не поднимались.

Спасает нас то, что максимальное число яблок в ценовой схеме не может превышать 100. Но чем это поможет?

Предположим, что нужно купить не менее 50 яблок. Может ли покупка 60 штук оказаться оптимальным вариантом? Конечно! Возможно, последняя ценовая схема в наилучшем решении для 60 яблок будет состоять из 20 штук. Тогда можно будет совместить эти 20 яблок с оптимальным решением для 40 яблок и в общем получить 60.

Снова предположим, что нужно купить 50 яблок. Будет ли иметь смысл приобретение 180? Что ж, подумаем о возможном оптимальном решении. Наибольшая



возможная ценовая схема может быть предложена для 100 яблок. Прежде чем ее использовать, нам бы пришлось купить не менее 80 яблок, которые были бы дешевле, чем 180. Но 80 уже больше 50. Следовательно, купить 80 яблок выгоднее, чем 180. Значит, покупка 180 яблок не может быть оптимальным решением, если их нужно всего 50.

На самом деле, для 50 яблок максимальным количеством, которое вообще есть смысл рассматривать, является 149. Если покупать 150 или более яблок, то вычитание последней ценовой схемы гарантированно даст более дешевый способ купить 50 или более яблок.

Условия задачи устанавливают равным 100 не только максимальное число яблок в ценовой схеме, но также наибольший возможный объем их покупки. В случае, когда нужно приобрести 100 яблок, максимальным рассматриваемым числом должно стать  $100 + 99 = 199$ . Объединение всех этих умозаключений дает нам функцию `solve`, которая приведена в листинге 3.10.

#### Листинг 3.10. Решение 1

```
#define SIZE 200

double solve(int num[], double price[], int num_schemes,
             double unit_price, int num_items) {
    double best;
    int i;
    best = solve_k(num, price, num_schemes, unit_price, num_items);
    for (i = num_items + 1; i < SIZE; i++)
        best = min(best, solve_k(num, price, num_schemes,
                                unit_price, i));
    return best;
}
```

Теперь нам нужна только функция `main`, и можно будет отправлять результаты на проверку.

## Функция `main`

Обратимся к листингу 3.11, где дана функция `main`. Она не завершена, но ей недостает только одной вспомогательной функции, `get_number`, которую я вскоре опишу.

#### Листинг 3.11. Функция `main`

```
#define MAX_SCHEMES 20

int main(void) {
    int test_case, num_schemes, num_items, more, i;
    double unit_price, result;
    int num[MAX_SCHEMES];
    double price[MAX_SCHEMES];
```

```

test_case = 0;
while (scanf("%lf%d", &unit_price, &num_schemes) != -1) { ❶
    test_case++;
    for (i = 0; i < num_schemes; i++)
        scanf("%d%lf", &num[i], &price[i]); ❷
    scanf(" "); ❸
    printf("Case %d:\n", test_case);
    more = get_number(&num_items);
    while (more) {
        result = solve(num, price, num_schemes, unit_price,
                        num_items);
        printf("Buy %d for $%.2f\n", num_items, result);
        more = get_number(&num_items);
    }
    result = solve(num, price, num_schemes, unit_price, ❹
                  num_items);
    printf("Buy %d for $%.2f\n", num_items, result); ❺
}
return 0;
}

```

Сначала мы читаем первую строку очередного тестового примера из входных данных ❶. Последующий вызов `scanf` ❷ представляет собой вложенный цикл, который считывает количество яблок и их стоимость в каждой ценовой схеме. В своем третьем вхождении `scanf` ❸ считывает символ переноса строки в конце последней строки ценовых схем. Таким образом, мы переходим к строке, содержащей количество яблок, которое нужно купить. Однако мы не можем просто продолжать вызывать `scanf` для считывания этих чисел, так как между ними нужно делать остановки. Для этого вводится вспомогательная функция `get_number`, описанная ниже. При наличии оставшихся для считывания чисел она возвращает 1, если же перед ней последнее число строки, возвращает 0. Это объясняет последующий код ❹ ❺: когда цикл заканчивается в связи с завершением считывания последнего числа строки, нам все еще нужно решить этот последний пример.

Код для `get_number` приведен в листинге 3.12.

### Листинг 3.12. Функция для считывания целых чисел

```

int get_number(int *num) {
    int ch;
    int ret = 0;
    ch = getchar();
    while (ch != ' ' && ch != '\n') { ❶
        ret = ret * 10 + ch - '0';
        ch = getchar();
    }
    *num = ret; ❷
    return ch == ' '; ❸
}

```

Эта функция считывает целые числа с помощью подхода, напоминающего листинг 2.17. Цикл продолжается до тех пор, пока не дойдет до пробела или символа новой строки ❶. По завершении цикла считанные данные сохраняются в указателе, передаваемом функции ❷. Я использую указатель, потому что возвращаемое значение применяется для другой цели: указывает, является ли это число последним в строке ❸. То есть если `get_number` возвращает 1 (так как нашла пробел после считанного числа), то на текущей строке еще есть числа; если же она возвращает 0, значит, это последнее число строки.

Теперь у нас есть завершенное решение, но его быстродействие очень мало. Даже на обработку сравнительно небольших тестовых примеров уйдут века, так как мы неизбежно будем перебирать все варианты вплоть до 199 яблок.

Без мемоизации здесь не обойтись.

## Решение 2. Мемоизация

При мемоизации в задаче с бургерами мы ввели массив `memo` в функцию `solve` (листинг 3.5), поскольку каждый ее вызов производился для отдельного примера. Однако в задаче с яблоками у нас есть строка, в которой каждое число указывает количество яблок, которое необходимо купить, и решение нужно получить для каждого из этих чисел. В этом случае будет расточительным перезаписывать массив `memo` до завершения обработки всего тестового примера.

В связи с этим мы объявим и инициализируем `memo` в `main`. Обновленная версия функции приведена в листинге 3.13.

### Листинг 3.13. Функция `main` с мемоизацией

```
int main(void) {
    int test_case, num_schemes, num_items, more, i;
    double unit_price, result;
    int num[MAX_SCHEMES];
    double price[MAX_SCHEMES];
    double memo[SIZE]; ❶
    test_case = 0;
    while (scanf("%lf%d", &unit_price, &num_schemes) != -1) {
        test_case++;
        for (i = 0; i < num_schemes; i++)
            scanf("%d%lf", &num[i], &price[i]);
        scanf(" ");
        printf("Case %d:\n", test_case);
        for (i = 0; i < SIZE; i++) ❷
            memo[i] = -1; ❸
        more = get_number(&num_items);
        while (more) {
            result = solve(num, price, num_schemes, unit_price
                           num_items, memo);
```

```

        printf("Buy %d for $%.2f\n", num_items, result);
        more = get_number(&num_items);
    }
    result = solve(num, price, num_schemes, unit_price,
                  num_items, memo);
    printf("Buy %d for $%.2f\n", num_items, result);
}
return 0;
}

```

Мы объявляем массив **memo** ❶ и устанавливаем для каждого его элемента значение -1 (неизвестное значение) ❷ ❸. Обратите внимание, что инициализация **memo** для каждого тестового примера происходит всего один раз. Второе и последнее отличие заключается в том, что массив добавляется в качестве нового параметра вызовов `solve`.

Обновленный код для `solve` приведен в листинге 3.14.

**Листинг 3.14.** Решение 2 с мемоизацией

```

double solve(int num[], double price[], int num_schemes,
             double unit_price, int num_items, double memo[]) {
    double best;
    int i;
    best = solve_k(num, price, num_schemes, unit_price,
                  num_items, memo);
    for (i = num_items + 1; i < SIZE; i++)
        best = min(best, solve_k(num, price, num_schemes,
                                unit_price, i, memo));
    return best;
}

```

Помимо добавления **memo** в конец списка параметров, мы передаем его в вызовы `solve_k`. На этом все.

В завершение рассмотрим изменения, необходимые для мемоизации `solve_k`. Мы будем сохранять в `memo[num_items]` минимальную стоимость покупки ровно `num_items` яблок. Смотрим листинг 3.15.

**Листинг 3.15.** Решение для `num_items` элементов, с мемоизацией

```

double solve_k(int num[], double price[], int num_schemes,
              double unit_price, int num_items, double memo[]) {
    double best, result;
    int i;
    if (memo[num_items] != -1) ❶
        return memo[num_items];
    if (num_items == 0) {
        memo[num_items] = 0;
        return memo[num_items];
    } else {

```

```
result = solve_k(num, price, num_schemes, unit_price,
                 num_items - 1, memo);
best = result + unit_price;
for (i = 0; i < num_schemes; i++)
    if (num_items - num[i] >= 0) {
        result = solve_k(num, price, num_schemes, unit_price,
                          num_items - num[i], memo);
        best = min(best, result + price[i]);
    }
memo[num_items] = best;
return memo[num_items];
}
```

Напомню, что при использовании мемоизации вначале выполняется проверка на наличие искомого решения в массиве **memo**. Если для подзадачи `num_items` сохранено какое-либо значение кроме `-1`, мы его возвращаем. В противном случае, как и в любой другой мемоизованной функции, новое решение подзадачи сначала сохраняется в `memo` и лишь затем возвращается.

На этом мы достигли логического завершения нашей задачи: решение с мемоизацией можно отправить на проверку, где оно должно успешно пройти все тесты. Если вы хотите дополнительно попрактиковаться в динамическом программировании, то имеете отличную возможность преобразовать мемоизованное решение в решение с динамическим программированием. В противном случае здесь можно поставить точку.

## Задача 3. Хоккейное соперничество

В первых двух задачах этой главы использовались одномерные массивы `memo` или `dp`. Давайте теперь рассмотрим задачу, для решения которой потребуется двумерный массив.

Я живу в Канаде, так что без упоминания хоккея не обойтись. Хоккей, как и футбол, — командный вид спорта, но всегда с голами.

Рассмотрим задачу с сайта DMOJ под номером `cco18p1`.

### Условие

Команда «Гуси» сыграла  $n$  игр, каждая из которых имела один из двух исходов: победа (W) или проигрыш (L). Ничейных игр не было. Нам известны исходы всех игр «Гусей», а также количество голов, которые они забили в каждой игре. Например, мы знаем, что их первая игра закончилась победой (W) и что в ней они забили четыре гола (следовательно, их противники забили меньше). Команда «Ястребы»

также отыграла  $n$  игр, в которых либо победила, либо проиграла. В этом случае нам тоже известно, как закончилась для них каждая игра и сколько голов они забили.

Некоторые игры команды провели друг против друга, в то время как остальные — против других соперников.

Информации о том, кто и с кем играл, не дано. Например, мы можем знать, что «Гуси» победили в определенной игре, забив в ней четыре гола, но нам неизвестно, с кем именно они играли — это могли быть «Ястребы» или любая другая команда.

*Принципиальный матч* — это игра, в которой «Гуси» играли против «Ястребов». Наша задача — определить максимальное число голов, которые могли быть забиты в принципиальных матчах.

### Входные данные

Входные данные содержат всего один тестовый пример, но информация приводится в пяти строках:

- Первая строка содержит  $n$ , количество игр, сыгранных каждой командой. Значение  $n$  находится между 1 и 1000.
- Вторая строка, содержащая  $n$  символов W (победа) и L (поражение), сообщает исход каждой сыгранной «Гусями» игры. К примеру, WLL означает, что команда выиграла первую игру, а затем проиграла вторую и третью.
- На третьей строке приводится  $n$  целых чисел, сообщающих количество забитых «Гусями» голов в каждой игре. К примеру, 4 1 2 означает, что команда забила четыре гола в первой игре, один во второй и два в третьей.
- Четвертая строка подобна второй и сообщает исход каждой игры «Ястребов».
- Пятая строка аналогична третьей и указывает количество голов, забитых «Ястребами» в каждой игре.

### Выходные данные

Следует вывести только одно число — максимальное количество голов, забитых в возможных принципиальных матчах.

Время на решение тестового примера — одна секунда.

## О принципиальных матчах

Прежде чем переходить к структуре оптимальных решений, давайте лучше разберем задачу, проработав ряд тестовых примеров.

Начнем со следующего:

```

3
WWW
2 5 1
WWW
7 8 5

```

В этом случае точно не могло быть принципиальных игр, поскольку в них одна сторона должна проиграть, а другая, соответственно, выиграть. Здесь же мы видим, что и «Гуси», и «Ястребы» выиграли все матчи, значит, эти команды друг против друга не играли. Поскольку в данном случае принципиальные игры невозможны, значит, и голов в них не было. Верным ответом будет 0.

А теперь разберем пример, в котором «Ястребы» все игры проигрывают:

```

3
WWW
2 5 1
LLL
7 8 5

```

Были ли принципиальные игры в этом случае? Ответ по-прежнему отрицательный. «Гуси» выиграли первую игру, забив два гола. Чтобы эта игра оказалась принципиальной, в ней «Ястребы» должны проиграть, забив при этом менее двух голов. А поскольку забивали они не менее пяти голов, значит, ни одна из этих игр не имеет отношения к первой игре «Гусей». Аналогичным образом рассуждаем об остальных играх. «Гуси» выиграли свою вторую игру, забив пять голов, а третью — со всего одним голом, но в списке «Ястребов» нет игр, в которых они забивали бы менее пяти голов. Это значит, что вторая и третья игры «Гусей» прошли против других соперников. В данном примере верным выводом снова будет 0.

Теперь перейдем к ненулевым случаям. Вот пример:

```

3
WWW
2 5 1
LLL
7 8 4

```

Единственное отличие в том, что здесь в последней игре «Ястребы» забили четыре гола, а не пять. Этого достаточно, чтобы получить возможную принципиальную игру. В частности, вторая игра «Гусей», в которой они победили и забили пять голов, может быть принципиальной, если она соответствует третьей игре «Ястребов», в которой они проиграли, забив четыре гола. Тогда в данной игре было забито девять голов, значит, верным выводом будет 9.

А теперь такой пример:

```
2
WW
6 2
LL
8 1
```

Взгляните на последнюю сыгранную каждой командой игру: «Гуси» победили, забив два гола, а «Ястребы» проиграли, забив один. Эта игра может оказаться принципиальной с общим числом голов, равным трем. Первая же игра не может быть принципиальной, так как «Гуси» победили с шестью голами, а «Ястребы» в таком случае не могли проиграть, забив восемь. Выходит, добавить к общему количеству голов в принципиальных матчах больше нечего. Будет ли тогда 3 верным выводом?

А вот и нет! Мы сопоставили только последние игры, а нужно было сопоставить также первую игру «Гусей» со второй «Ястребов». Этот матч тоже мог быть принципиальным, но уже с семью голами: верным выводом будет 7.

Рассмотрим еще один пример. Попробуйте самостоятельно определить максимум, прежде чем прочитать ответ ниже:

```
4
WLWW
3 4 1 8
WLLL
5 1 2 3
```

Верным ответом является 20 — сумма результатов двух возможных принципиальных матчей: сопоставления второй игры «Гусей» с первой «Ястребов» (9 голов) и четвертой игры «Гусей» с четвертой «Ястребов» (11 голов).

## Описание оптимальных решений

Рассмотрим оптимальное решение этой задачи, которое максимизирует количество голов, забитых в принципиальных матчах. Как оно может выглядеть? Предположим, что игры для каждой команды пронумерованы от 1 до  $n$ .

*Вариант 1.* В первом случае в качестве принципиальной берется финальная игра  $n$ , сыгранная «Гусями», и финальная игра  $n$ , сыгранная «Ястребами». В этой игре было забито  $g$  голов. Далее можно исключить обе эти игры и оптимально решить меньшую подзадачу для первых  $n - 1$  игр «Гусей» и первых  $n - 1$  игр «Ястребов». Сумма решения этой подзадачи и  $g$  окажется общим оптимальным решением. Однако такой вариант возможен, только если результаты игр  $n$  реально могут соответствовать принципиальной игре. Например, если обе команды закончили



эту игру с результатом W, то она не может быть принципиальной, и вариант 1 применить нельзя.

Помните тестовый пример из предыдущего раздела?

```

4
WLWW
3 4 1 8
WLLL
5 1 2 3

```

Вот пример варианта 1: мы сопоставляем два последних значения голов, 8 и 3, после чего оптимально решаем подзадачу для оставшихся игр.

*Вариант 2.* Еще один вариант — это когда оптимальное решение не связано с последними играми. В этом случае мы отделяем игру  $n$ , сыгранную «Гусями», и игру  $n$ , сыгранную «Ястребами», после чего оптимально решаем подзадачу для первых  $n - 1$  игр «Гусей» и первых  $n - 1$  игр «Ястребов».

Первый тестовый пример из предыдущего раздела иллюстрирует вариант 2:

```

3
WWW
2 5 1
WWW
7 8 5

```

Значения 1 и 5 справа не являются частью оптимального решения. В итоге оптимальное решение для остальных игр и будет общим оптимальным решением.

Пока что мы рассмотрели варианты решения, в которых результаты игр  $n$  либо используются, либо нет. Можно ли на этом закончить? Нет, и чтобы в этом убедиться, рассмотрите следующий тестовый пример:

```

2
WW
6 2
LL
8 1

```

Вариант 1, сопоставляющий 2 и 1, ведет к максимуму из трех голов в принципиальных играх. Вариант 2, отбрасывающий 2, и 1, ведет к максимуму из нуля голов в принципиальных играх. Тем не менее общий максимум здесь семь. Значит, с использованием варианта 1 и варианта 2 охватить все виды оптимальных решений не получится.

Нам нужна возможность отбрасывать игру «Гусей», но не «Ястребов». В частности, в примере выше требуется отбросить вторую игру «Гусей», а затем решить подзадачу,

состоящую из первой игры «Гусей» и *обеих* игр «Ястребов». Для симметрии нам также нужна возможность отбросить вторую игру «Ястребов» и решить получающуюся подзадачу для первой игры «Ястребов» и *обеих* игр «Гусей». Эти два дополнительных варианта мы и опишем ниже.

*Вариант 3.* Оптимальное решение не связано с  $n$ -й игрой «Гусей». В этом случае мы отделяем игру  $n$  этой команды и оптимально решаем подзадачу для первых  $n - 1$  игр «Гусей» и первых  $n$  игр «Ястребов».

*Вариант 4.* Оптимальное решение не связано с  $n$ -й игрой «Ястребов». В этом случае мы отделяем игру  $n$ , сыгранную этой командой, и оптимально решаем подзадачу для первых  $n$  игр «Гусей» и первых  $n - 1$  игр «Ястребов».

Варианты 3 и 4 усложняют структуру решения задачи, независимо от того, используется ли в нем рекурсия, мемоизация или динамическое программирование. В предыдущих задачах главы наши подзадачи описывались всего одним параметром:  $t$  в «Страсти к бургерам» и  $k$  в «Экономных покупателях». При отсутствии вариантов 3 и 4 мы и в задаче с хоккеем обошлись бы одним параметром,  $n$ , который указывал бы, что решается подзадача для первых  $n$  игр, сыгранных «Гусями» и «Ястребами». Но в вариантах 3 и 4 значения  $n$  для разных команд больше не стыкуются: при изменении одного второе может не меняться. К примеру, если мы решаем подзадачу в отношении пяти игр, сыгранных «Гусями», это не значит, что мы также будем рассматривать пять игр, сыгранных «Ястребами». То же касается и обратной ситуации: подзадача в отношении первых пяти игр «Ястребов» не свидетельствует о таком же числе игр, сыгранных «Гусями».

Следовательно, для решения подзадач потребуются два параметра:  $i$  — количество игр «Гусей» и  $j$  — количество игр «Ястребов».

В задачах оптимизации количество параметров подзадач может быть равно как единице, так и большему числу. Я рекомендую вначале пробовать использовать один параметр и рассматривать возможные варианты в поиске оптимального решения. Не исключено, что каждый вариант можно проработать путем решения подзадач с одним параметром. Тогда дополнительные параметры просто не понадобятся. Тем не менее иногда один или более вариантов потребуют такого решения подзадачи, которое нельзя реализовать с одним параметром. В таких случаях число параметров придется увеличить.

Выгода от добавления дополнительных параметров заключается в расширении пространства соответствующих подзадач, в котором ищется оптимальное решение. Платить за это приходится необходимостью решать больше подзадач. Стремление использовать минимальное число параметров — один, два или, возможно, три — является ключом к построению быстрых решений оптимизационных задач.

## Решение 1. Рекурсия

Пришло время написать рекурсивное решение. На этот раз функция `solve` будет выглядеть так:

```
int solve(char outcome1[], char outcome2[], int goals1[],
          int goals2[], int i, int j)
```

Как всегда, здесь используются параметры двух типов: входные данные тестового примера и информация о текущей подзадаче. Вот их краткое описание:

- **outcome1** — массив символов W и L для «Гусей».
- **outcome2** — массив символов W и L для «Ястребов».
- **goals1** — массив голов, забитых «Гусями».
- **goals2** — массив голов, забитых «Ястребами».
- **i** — количество рассматриваемых в текущей подзадаче игр «Гусей».
- **j** — количество рассматриваемых в текущей подзадаче игр «Ястребов».

Последние два параметра являются специфичными для текущей подзадачи и единственными, которые изменяются при рекурсивных вызовах.

Если начинать каждый массив с индекса 0, как это по умолчанию принято в Си, то придется помнить, что информация для некоторой игры *k* находится не в индексе *k*, а в индексе *k* - 1. Например, информация о четвертой игре будет находиться в индексе 3. Чтобы этого избежать, мы будем сохранять информацию об играх, начиная с индекса 1. Таким образом, данные о четвертой игре будут находиться в индексе 4. С подобной схемой хранения вероятность ошибки уменьшится.

Код для рекурсивного решения приведен в листинге 3.16.

### Листинг 3.16. Решение 1

```
int max(int v1, int v2) { ❶
    if (v1 > v2)
        return v1;
    else
        return v2;
}

int solve(char outcome1[], char outcome2[], int goals1[],
          int goals2[], int i, int j) {
    int first, second, third, fourth; ❷
    if (i == 0 || j == 0) ❸
        return 0;
    if ((outcome1[i] == 'W' && outcome2[j] == 'L' && ❹
        goals1[i] > goals2[j]) ||
```

```

    (outcome1[i] == 'L' && outcome2[j] == 'W' &&
     goals1[i] < goals2[j]))
    first = solve(outcome1, outcome2, goals1, goals2, i - 1, j - 1) + ⑤
           goals1[i] + goals2[j];
else
    first = 0;
    second = solve(outcome1, outcome2, goals1, goals2, i - 1, j - 1); ⑥
    third = solve(outcome1, outcome2, goals1, goals2, i - 1, j); ⑦
    fourth = solve(outcome1, outcome2, goals1, goals2, i, j - 1); ⑧
    return max(first, max(second, max(third, fourth))); ⑨
}

```

В данном случае перед нами задача по максимизации: требуется найти максимальное количество забитых в принципиальных матчах голов. Первой идет функция `max` ① — она используется для определения наилучшего варианта. За ней следует объявление целочисленных переменных, по одной для каждого из четырех вариантов ②.

Начнем с базовых случаев: что нужно вернуть, если  $i$  и  $j$  равны 0? В таком случае подзадача будет решаться для нуля игр «Гусей» и нуля игр «Ястребов». А поскольку игр не было, то и принципиальных не было тоже. Нет принципиальных игр — нет забитых в них голов, поэтому нужно вернуть 0.

Тем не менее это не единственный базовый случай. К примеру, рассмотрим подзадачу, в которой «Гуси» не играли ( $i = 0$ ), а «Ястребы» сыграли три игры ( $j = 3$ ). Принципиальных игр при таких условиях быть не могло, поскольку «Гуси» вообще не играли. Аналогичная ситуация складывается, когда «Ястребы» не играли: даже если «Гуси» и сыграют с кем-то, это точно будут не «Ястребы».

На этом базовые случаи исчерпаны. То есть если  $i$  или  $j$  будет иметь значение 0, то общее число забитых в принципиальных играх голов составит ноль ③.

Разобравшись с базовыми случаями, нужно опробовать четыре возможных варианта в поиске оптимального решения и выбрать лучший.

*Вариант 1.* Напомню, что этот вариант действителен, только если последняя игра «Гусей» и последняя игра «Ястребов» могут быть принципиальным матчем. Для этого есть две возможности:

1. «Гуси» побеждают, а «Ястребы» проигрывают. При этом «Гуси» забивают больше голов, чем «Ястребы».
2. «Гуси» проигрывают, а «Ястребы» выигрывают. При этом «Гуси» забивают меньше голов, чем «Ястребы».

Эти две возможности мы и рассматриваем ④. Если игра может быть принципиальной, то мы вычисляем оптимальное решение для этого случая ⑤: оно состоит

из оптимального решения для  $i - 1$  игр «Гусей» и  $j - 1$  игр «Ястребов» плюс общее число голов в данной принципиальной игре.

*Вариант 2.* В этом варианте подзадача решается для первых  $i - 1$  игр «Гусей» и  $j - 1$  игр «Ястребов» ⑥.

*Вариант 3.* Здесь подзадача решается для первых  $i - 1$  игр «Гусей» и  $j$  игр «Ястребов» ⑦. Обратите внимание, что  $i$  изменяется, а  $j$  нет. Именно поэтому в данной подзадаче требуются два параметра, а не один.

*Вариант 4.* Тут решается подзадача для первых  $i$  игр «Гусей» и  $j - 1$  игр «Ястребов» ⑧. Опять же, один из параметров меняется, а другой нет. Причем это благо, что нам не требуется хранить их с одним значением.

Вот и все: `first`, `second`, `third` и `fourth` являются четырьмя претендентами на оптимальное решение. Теперь нужно определить, какое из них дает максимальное число очков, а после сравнения мы возвращаем результат ⑨. Самый глубокий вызов `max` вычисляет максимум между `third` и `fourth`. На следующем уровне очередной вызов `max` вычисляет максимум между победителем из предыдущего вызова и `second`. В завершение вычисляется максимум между победителем из предыдущего вызова и `first`.

Мы почти у цели. Осталось только прописать функцию `main`, считывающую пять строк входных данных и вызывающую `solve`. Ее код приведен в листинге 3.17. По сравнению с функцией `main` из задачи с яблоками она выглядит более компактной.

### Листинг 3.17. Функция `main`

```
#define SIZE 1000

int main(void) {
    int i, n, result;
    char outcome1[SIZE + 1], outcome2[SIZE + 1]; ①
    int goals1[SIZE + 1], goals2[SIZE + 1]; ②
    scanf("%d ", &n); ③
    for (i = 1; i <= n; i++)
        scanf("%c", &outcome1[i]);
    for (i = 1; i <= n; i++)
        scanf("%d ", &goals1[i]);
    for (i = 1; i <= n; i++)
        scanf("%c", &outcome2[i]);
    for (i = 1; i <= n; i++)
        scanf("%d ", &goals2[i]);
    result = solve(outcome1, outcome2, goals1, goals2, n, n);
    printf("%d\n", result);
    return 0;
}
```

Вначале объявляются массивы результатов (W и L) ❶ и голов ❷. Прибавление единицы отвечает за повышение стартового индекса с 0 до 1, о чем говорилось выше. Если просто использовать SIZE, то допустимые индексы будут идти от нуля до 999. Нам же нужно также включить индекс 1000.

Затем следует считывание целого числа в первой строке ❸, что дает количество игр, сыгранных «Гусями» и «Ястребами». Между %d и закрывающей кавычкой добавлен пробел, чтобы scanf не считывал символ новой строки.

Далее идет считывание информации о победах и поражениях (W и L) «Гусей», а затем о забитых этой командой голах. После то же самое проделывается для «Ястребов». В завершение происходит вызов solve. Задачу нужно решить для всех  $n$  игр «Гусей» и всех  $n$  игр «Ястребов», в связи с чем последние аргументы — два  $n$ .

Пройдет ли данное решение проверку? По правде говоря, можно смело прогнозировать ошибку Time-Limit Exceeded.

## Решение 2. Мемоизация

В «Страсти к бургерам» и «Экономных покупателях» мы использовали для мемо одномерный массив, поскольку в их подзадачах было всего по одному параметру: время и количество яблок соответственно. В «Хоккейном соперничестве» используются два параметра, поэтому потребуется массив мемо с двумя измерениями. В элементе мемо[i][j] сохраняется решение подзадачи для первых  $i$  игр «Гусей» и первых  $j$  игр «Ястребов». За исключением перехода от одного к двум измерениям в мемо, сама методика остается прежней: возвращать решение, если оно уже сохранено, и вычислять, а затем сохранять, если нет.

Обновленная функция main приведена в листинге 3.18.

### Листинг 3.18. Функция main с мемоизацией

```
int main(void) {
    int i, j, n, result;
    char outcome1[SIZE + 1], outcome2[SIZE + 1];
    int goals1[SIZE + 1], goals2[SIZE + 1];
    static int memo[SIZE + 1][SIZE + 1];
    scanf("%d ", &n);
    for (i = 1; i <= n; i++)
        scanf("%c", &outcome1[i]);
    for (i = 1; i <= n; i++)
        scanf("%d ", &goals1[i]);
    for (i = 1; i <= n; i++)
        scanf("%c", &outcome2[i]);
    for (i = 1; i <= n; i++)
        scanf("%d ", &goals2[i]);
    for (i = 0; i <= SIZE; i++)
```

```

    for (j = 0; j <= SIZE; j++)
        memo[i][j] = -1;
    result = solve(outcome1, outcome2, goals1, goals2, n, n, memo);
    printf("%d\n", result);
    return 0;
}

```

Обратите внимание, что массив `memo` огромен — более 1 миллиона элементов, поэтому мы делаем его статичным, как в листинге 1.8.

Мемоизованная функция `solve` представлена в листинге 3.19.

#### Листинг 3.19. Решение 2 с мемоизацией

```

int solve(char outcome1[], char outcome2[], int goals1[],
          int goals2[], int i, int j, int memo[SIZE + 1][SIZE + 1]) {
    int first, second, third, fourth;
    if (memo[i][j] != -1)
        return memo[i][j];
    if (i == 0 || j == 0) {
        memo[i][j] = 0;
        return memo[i][j];
    }
    if ((outcome1[i] == 'W' && outcome2[j] == 'L' &&
         goals1[i] > goals2[j]) ||
        (outcome1[i] == 'L' && outcome2[j] == 'W' &&
         goals1[i] < goals2[j]))
        first = solve(outcome1, outcome2, goals1, goals2, i - 1, j - 1, memo) +
                 goals1[i] + goals2[j];
    else
        first = 0;
    second = solve(outcome1, outcome2, goals1, goals2, i - 1, j - 1, memo);
    third = solve(outcome1, outcome2, goals1, goals2, i - 1, j, memo);
    fourth = solve(outcome1, outcome2, goals1, goals2, i, j - 1, memo);
    memo[i][j] = max(first, max(second, max(third, fourth)));
    return memo[i][j];
}

```

Это решение успешно проходит проверку тестовыми примерами. Если бы требовалось просто решить эту задачу, то на этом можно было бы закончить. Но здесь есть возможность узнать больше о динамическом программировании.

## Решение 3. Динамическое программирование

В предыдущем разделе мы видели, что для мемоизации нужен двумерный массив `memo`, а не одномерный. Для разработки решения на основе динамического программирования нам также потребуется двумерный массив `dp`. В листинге 3.18 массив `memo` был объявлен так:

```
static int memo[SIZE + 1][SIZE + 1];
```

И для массива `dp` мы используем аналогичный код:

```
static int dp[SIZE + 1][SIZE + 1];
```

Как и в `memo`, элемент `dp[i][j]` будет хранить решение подзадачи для первых `i` игр «Гусей» и первых `j` игр «Ястребов». Тогда нам нужно решить каждую из этих подзадач и вернуть `dp[n][n]`.

В мемоизованных решениях задач по оптимизации нам не нужно определять порядок решения подзадач. Мы лишь совершаем рекурсивные вызовы, которые возвращают решения для соответствующих им подзадач. Однако при использовании динамического программирования порядок решения определять необходимо. Нельзя решать подзадачи в произвольной последовательности, потому что тогда необходимое решение в нужный момент может оказаться недоступным.

Предположим, что мы хотим заполнить `dp[3][5]` — элемент для первых трех игр «Гусей» и первых пяти игр «Ястребов». Еще раз взглянем на четыре варианта оптимального решения:

- Вариант 1 требует нахождения `dp[2][4]`.
- Вариант 2 также требует нахождения `dp[2][4]`.
- Вариант 3 требует нахождения `dp[2][5]`.
- Вариант 4 требует нахождения `dp[3][4]`.

Нам необходим такой порядок решения, чтобы эти элементы `dp` уже были сохранены к моменту сохранения `dp[3][5]`.

Подзадачи с одним параметром, как правило, решаются от наименьшего индекса к наибольшему. Если же параметров в подзадаче больше, все усложняется, поскольку заполнять массив можно в различном порядке, и не все варианты обеспечат доступность решения подзадачи в нужный момент.

В хоккейной задаче можно определить `dp[i][j]`, если у нас уже будут сохранены решения для `dp[i - 1][j - 1]` (варианты 1 и 2), `dp[i - 1][j]` (вариант 3) и `dp[i][j - 1]` (вариант 4). Один из способов добиться этого — решить все подзадачи `dp[i - 1]` до решения любой из подзадач `dp[i]`. Например, в таком случае `dp[2][4]` решалась бы до `dp[3][5]`, что требуется для вариантов 1 и 2. Кроме того, `dp[2][5]` решалась бы до `dp[3][5]`, что требуется для варианта 3. Это значит, что решение строки `i - 1` перед строкой `i` удовлетворяет требованиям вариантов 1–3.

Чтобы выполнить требование варианта 4, можно решить подзадачи `dp[i]` от наименьшего до наибольшего значений индекса `j`. Так, к примеру, `dp[3][4]` решалась бы до `dp[3][5]`.



Говоря коротко, мы решаем все подзадачи в строке 0 слева направо, потом все подзадачи в строке 1 слева направо и так далее, пока не будут решены все подзадачи в строке n.

Функция `solve` для решения на основе динамического программирования представлена в листинге 3.20.

**Листинг 3.20.** Решение 3 с динамическим программированием

```
int solve(char outcome1[], char outcome2[], int goals1[],
          int goals2[], int n) {
    int i, j;
    int first, second, third, fourth;
    static int dp[SIZE + 1][SIZE + 1];
    for (i = 0; i <= n; i++)
        dp[0][i] = 0;
    for (i = 0; i <= n; i++)
        dp[i][0] = 0;
    for (i = 1; i <= n; i++) ❶
        for (j = 1; j <= n; j++) { ❷
            if ((outcome1[i] == 'W' && outcome2[j] == 'L' &&
                 goals1[i] > goals2[j]) ||
                (outcome1[i] == 'L' && outcome2[j] == 'W' &&
                 goals1[i] < goals2[j]))
                first = dp[i-1][j-1] + goals1[i] + goals2[j];
            else
                first = 0;
            second = dp[i-1][j-1];
            third = dp[i-1][j];
            fourth = dp[i][j-1];
            dp[i][j] = max(first, max(second, max(third, fourth)));
        }
    return dp[n][n]; ❸
}
```

В начале инициализируются подзадачи базового случая, в которых как минимум один из индексов равен 0. Далее выполняется двойной цикл `for` ❶ ❷, управляющий порядком решения подзадач вне базового случая. Сначала перебираются строки ❶, а потом элементы в каждой строке ❷, что, как мы выяснили, является подходящим порядком для решения подзадач. После заполнения таблицы возвращается решение для исходной задачи ❸.

Массив, создаваемый двумерным алгоритмом динамического программирования, можно представить в виде таблицы. Это помогает лучше понять процесс его заполнения элементами. Рассмотрим следующий тестовый пример:

```
4
WLWW
3 4 1 8
WLLL
5 1 2 3
```

Массив для него выглядит следующим образом:

<b>4</b>	0	9	18	19	20
<b>3</b>	0	9	9	9	9
<b>2</b>	0	9	9	9	9
<b>1</b>	0	0	4	5	5
<b>0</b>	0	0	0	0	0
	<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>

Разберем, к примеру, вычисление элемента в строке 4 и столбце 2, то есть `dp[4][2]`. Это подзадача для первых четырех игр «Гусей» и первых двух игр «Ястребов». При рассмотрении четвертой игры «Гусей» и второй игры «Ястребов» видно, что «Гуси» победили, забив 8 голов, а «Ястребы» проиграли, забив 1 гол. Значит, эта игра могла быть принципиальной. В данном случае допустим вариант 1. Всего в этой игре было забито 9 голов. К этим девяти мы прибавляем значение из строки 3, столбца 1, которым также является 9. В результате получается сумма 18. Пока что это наш максимум, теперь нужно попробовать варианты 2–4, чтобы проверить, не окажутся ли они лучше. В результате мы увидим, что в каждом из них получается значение 9. Следовательно, в `dp[4][2]` мы сохраняем 18, то есть максимальное значение из всех возможных вариантов.

Итоговый ответ, конечно же, находится в правой верхней ячейке, соответствующей подзадаче всех  $n$  игр как для «Гусей», так и для «Ястребов». Это значение, 20, мы и возвращаем в качестве оптимального решения. Другие количественные значения в таблице полезны только тем, что они помогают в итоге получить 20.

В код функции `main` (листинг 3.17) мы вносим одно небольшое изменение: нужно лишь удалить последнее `n`, передаваемое в `solve`, получив в итоге:

```
result = solve(outcome1, outcome2, goals1, goals2, n);
```

## Оптимизация пространства

В разделе «Шаг 4. Динамическое программирование» на с. 128 я отметил, что мемоизация и динамическое программирование примерно эквивалентны. *Примерно*, потому что иногда одно выгоднее использовать, чем другое. Задача с хоккеем содержит пример типичной оптимизации, которую можно выполнить с помощью динамического программирования, но не мемоизации. В данном случае оптимизация касается не скорости, а пространства.

Зададимся вопросом: при решении подзадачи в строке  $i$  массива `dp` к каким строкам идет обращение? Вернемся к четырем рассмотренным вариантам, и вы увидите, что используются только строки  $i - 1$  (предыдущая) и  $i$  (текущая). Строки  $i - 2$ ,  $i - 3$  и другие нам не требуются. В таком случае хранение всего двумерного массива в памяти будет излишне затратным. Предположим, что решаем подзадачи в строке 500. Все что нам нужно — это обратиться к строке 500 и строке 499. При этом нам не нужно держать в памяти строки 498, 497 и с меньшими номерами, потому что рассматривать их больше не потребуется.

Вместо использования двумерной таблицы можно задействовать только два одномерных массива: один для предыдущей строки, другой для текущей. Эта оптимизация реализуется в листинге 3.21.

**Листинг 3.21.** Решение 3 с оптимизацией использования памяти

```
int solve(char outcome1[], char outcome2[], int goals1[],
          int goals2[], int n) {
    int i, j, k;
    int first, second, third, fourth;
    static int previous[SIZE + 1], current[SIZE + 1];
    for (i = 0; i <= n; i++) ❶
        previous[i] = 0; ❷
    for (i = 1; i <= n; i++) {
        for (j = 1; j <= n; j++) {
            if ((outcome1[i] == 'W' && outcome2[j] == 'L' &&
                 goals1[i] > goals2[j]) ||
                (outcome1[i] == 'L' && outcome2[j] == 'W' &&
                 goals1[i] < goals2[j]))
                first = previous[j-1] + goals1[i] + goals2[j];
            else
                first = 0;
            second = previous[j-1];
            third = previous[j];
            fourth = current[j-1];
            current[j] = max(first, max(second, max(third, fourth)));
        }
        for (k = 0; k <= SIZE; k++) ❸
            previous[k] = current[k]; ❹
    }
    return current[n];
}
```

Мы инициализируем массив `previous`, заполненный нулями ❶ ❷, решая таким образом все подзадачи в строке 0. В оставшейся части кода вместо обращения к строке  $i - 1$  теперь используется `previous`. Помимо этого, там, где ранее шло обращение к строке  $i$ , теперь используется массив `current`. Как только новая строка успешно решается и сохраняется в `current`, мы копируем `current` в `previous` ❸ ❹, чтобы можно было использовать его для следующей строки.

## Задача 4. Учебный план

Мы добрались до последнего короткого примера. Первые три задачи этой главы требовали максимизации («Страсть к бургерам» и «Хоккейное соперничество») или минимизации («Экономные покупатели») результата. Закончить же главу я хочу задачей несколько иного плана: вместо поиска оптимального решения мы будем подсчитывать количество возможных. Вы увидите, что и в этом случае можно применять мемоизацию и динамическое программирование.

Рассмотрим задачу с платформы UV под номером 10910.

### Условие

Для завершения учебного курса необходимо как минимум  $p$  оценок ( $p$  может быть любым целым числом). Студент полностью прошел  $n$  курсов. В сумме он заработал  $t$  оценок, но мы не знаем, сколько из них студент получил на каждом курсе. Отсюда вопрос: каким числом различных способов студент мог пройти все курсы?

В качестве примера предположим, что он прошел два курса, на которых заработал девять оценок. При этом для завершения каждого курса требуется получить не менее трех оценок. В этом случае есть четыре способа, которыми студент мог завершить все курсы:

- Три оценки на курсе 1 и шесть оценок на курсе 2.
- Четыре оценки на курсе 1 и пять оценок на курсе 2.
- Пять оценок на курсе 1 и четыре оценки на курсе 2.
- Шесть оценок на курсе 1 и три оценки на курсе 2.

### Входные данные

Первая строка входных данных — это целое число  $k$ , указывающее количество тестовых примеров для выполнения. Каждый из  $k$  тестовых примеров представлен строкой, состоящей из трех целых чисел:  $n$  (количество завершенных курсов),  $t$  (общее число полученных оценок) и  $p$  (число оценок, необходимое для завершения каждого курса). Каждое значение  $n$ ,  $t$  и  $p$  находится в диапазоне между 1 и 70.

Вот входные данные для рассмотренного выше примера:

```
1
2 9 3
```

## Выходные данные

Для каждого тестового примера следует вывести число способов, которыми можно распределить оценки так, чтобы студент завершил все курсы. Для примера выше выводом будет число 4.

Время на решение тестовых примеров — три секунды.

## Решение. Мемоизация

Обратите внимание, что в данном случае отсутствует оптимальный способ распределения оценок. Решение, в котором студент с запасом завершает один курс и с трудом дотягивает остальные, будет считаться равнозначным другим (как преподавателю, мне трудно с этим согласиться).

Поскольку оптимального решения нет, не нужно продумывать его структуру. Вместо этого мы определим, как должно выглядеть решение в принципе. На первом курсе студент должен заработать не менее  $p$  оценок и не более  $t$ . Затем в каждом случае мы получаем новую подзадачу, в которой уже будет на один курс меньше. Предположим, что студент зарабатывает  $m$  оценок на первом курсе. Тогда мы решаем подзадачу для  $n - 1$  курсов, на которых студент получил  $t - m$  оценок.

Вместо использования `min` или `max` для выбора лучшего решения мы используем сложение для получения их общего количества.

Набравшись опыта, вы сможете быстро определять, можно ли пропустить очевидно неподходящее рекурсивное решение и сразу использовать динамическое программирование или мемоизацию. Мемоизация добавляет к рекурсивному решению минимум кода, поэтому ее применение выглядит более разумным выбором. В листинге 3.22 представлено завершенное мемоизованное решение задачи «Учебный план».

### Листинг 3.22. Решение с мемоизацией

```
#define SIZE 70

int solve(int n, int t, int p, int memo[SIZE + 1][SIZE + 1]) {
    int total, m;
    if (memo[n][t] != -1)
        return memo[n][t];
    if (n == 0 && t == 0) ❶
        return 1;
    if (n == 0) ❷
        return 0;
    total = 0;
    for (m = p; m <= t; m++)
```

```

    total = total + solve(n - 1, t - m, p, memo);
    memo[n][t] = total;
    return memo[n][t];
}

int main(void) {
    int k, i, x, y, n, t, p;
    int memo[SIZE + 1][SIZE + 1];
    scanf("%d", &k);
    for (i = 0; i < k; i++) {
        scanf("%d%d", &n, &t, &p);
        for (x = 0; x <= SIZE; x++)
            for (y = 0; y <= SIZE; y++)
                memo[x][y] = -1;
        printf("%d\n", solve(n, t, p, memo));
    }
    return 0;
}

```

Какова ситуация с базовыми случаями ❶ ❷? Базовыми являются случаи, когда количество курсов  $n$  равно 0, но тут возникают два варианта. Во-первых, предположим, что  $t$  также равно 0. Сколькими способами можно распределить ноль отметок на нуле курсов? Здесь легко ошибиться и сказать, что ответом будет «ноль», но правильный ответ — «один». То есть задачу можно закончить, вообще не распределяя оценки. И это определенно будет способом пройти ноль курсов! А теперь, что если  $n$  будет равно 0, а  $t$  будет больше 0? Вот здесь ответом действительно будет «ноль», так как нет способа распределить положительное число оценок по нулю курсов.

Оставшаяся часть программы перебирает допустимые числа оценок ( $m$ ) для текущего курса и решает подзадачу, где рассматривается на один курс меньше и оценок нужно распределить на  $m$  меньше.

## Выводы

Мы рассмотрели основы мемоизации и динамического программирования: определение структуры оптимального решения, разработку рекурсивного алгоритма, повышение его скорости с помощью мемоизации и возможную замену рекурсии заполнением таблицы. Когда вы освоите решение задач с одно- и двумерными таблицами, я рекомендую вам поработать и с такими, которые потребуют трех и более измерений. Принципы в этом случае будут аналогичны описанным выше, но вам придется дополнительно потрудиться, чтобы выявить и связать подзадачи.

Идеи, относящиеся к динамическому программированию, зачастую встречаются и в других алгоритмах. К примеру, в следующей главе вы увидите, что мы снова будем сохранять результаты для их последующего поиска. В главе 6 мы рассмотрим

задачу, в которой динамическое программирование играет вспомогательную роль, ускоряя вычисления, необходимые основному алгоритму.

## Примечания

Задача «Хоккейное соперничество» вошла в программу канадской олимпиады по программированию 2018 года (2018 Canadian Computing Olympiad).

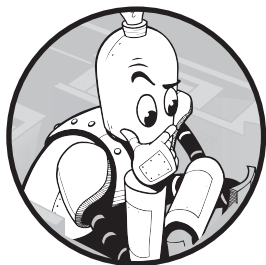
Многие учебники по алгоритмам более глубоко разбирают теорию и применение мемоизации и динамического программирования. Лично мне больше всех нравится книга «Algorithms Design»<sup>1</sup> Джона Клейнберга (Jon Kleinberg) и Евы Тардос (Éva Tardos).

---

<sup>1</sup> Клейнберг Дж., Тардос Е. Алгоритмы: разработка и применение. Классика Computers Science. СПб.: Питер.

# 4

## Графы и поиск в ширину



В этой главе мы рассмотрим три задачи, в которых нужно будет решить головоломку за минимальное число ходов. Как быстро сможет конь съесть пешку? Насколько быстро сможет студент взобраться по канату на уроке физкультуры? Какова минимальная цена перевода книги на другие языки? Получить ответы на эти вопросы позволит алгоритм поиска в ширину (breadth-first search, BFS). BFS решает упомянутые задачи и в общем подходит всякий раз, когда требуется решить головоломку за минимальное количество ходов. Параллельно с этой техникой мы познакомимся с графами, мощным способом моделирования и решения задач, в которых рассматриваются объекты и связи между ними.

### Задача 1. Погоня за пешкой

Рассмотрим задачу с платформы DMOJ под номером `ccc99s4`.

#### Условие

Два человека играют в настольную игру, у одного игрока есть пешка, а у второго конь (не беспокойтесь, знание шахматной теории вам не потребуется).

Игровая доска разделена на  $r$  рядов. Первый ряд находится вверху, а ряд  $r$  внизу. Доска также разделена на  $s$  столбцов, первый столбец расположен слева, а столбец  $s$  справа.



Первой ходит пешка, затем конь, затем снова пешка, затем конь и так далее, пока игра не завершится. За каждый ход фигура обязана переместиться, оставаться на месте запрещается.

Пешка не выбирает, как именно переместиться, и каждый ход продвигается на одну клетку вверх. Конь же каждый ход имеет до восьми вариантов перемещения по клеткам:

- вверх на 1, вправо на 2;
- вверх на 1, влево на 2;
- вниз на 1, вправо на 2;
- вниз на 1, влево на 2;
- вверх на 2, вправо на 1;
- вверх на 2, влево на 1;
- вниз на 2, вправо на 1;
- вниз на 2, влево на 1.

Я написал «до восьми вариантов», а не «ровно восемь», потому что ходы, ведущие фигуру за границы доски, не допускаются. Например, если поле поделено на 10 вертикалей и конь находится на вертикали 9, то ходы, при которых конь смещается на две вертикали вправо, недопустимы.

На схеме ниже отражены возможные ходы коня:

		f		e		
	b				a	
			K			
	d				c	
		h		g		

Конь обозначен на схеме буквой К, а буквы от а до h указывают его возможные ходы.

Игра завершается при выполнении одного из трех условий: конь съедает пешку; возникает ничья (пат); конь проигрывает, поскольку пешка достигает верхнего ряда:

- Конь побеждает, если совершает ход и занимает клетку, в которой стоит пешка, до того как пешка достигнет верхнего ряда. Для победы завершающий ход должен принадлежать именно коню. Если же последний ход сделает пешка, то выигрыш коня не засчитывается.
- Пат возникает, когда конь совершает ход и занимает клетку над пешкой до того, как пешка достигает верхнего ряда. Опять же, последний ход должен быть именно за конем. Единственное исключение — ситуация, когда игра начинается с пата, то есть стартовая позиция коня окажется в клетке над пешкой.
- Конь проигрывает, если пешка достигает верхнего ряда до того, как он победит или возникнет пат. То есть если пешка добирается до верхнего ряда прежде, чем конь ее съест или займет клетку над ней, то конь проигрывает. Как только пешка достигает верхнего ряда, игра останавливается.

Целью решения является определить результат лучшего варианта для коня и количество его ходов, необходимых для получения этого результата.

### Входные данные

Первая строка входных данных содержит число тестовых примеров, каждый из которых состоит из шести строк:

- количества горизонталей на доске — от 3 до 99;
- количества вертикалей на доске — от 2 до 99;
- номера стартовой горизонтали пешки;
- номера стартовой вертикали пешки;
- номера стартовой горизонтали коня;
- номера стартовой вертикали коня.

Гарантируется, что пешка и конь в начале игры стоят на разных клетках доски и что конь начинает игру из позиции, откуда ему будет доступно не менее одного хода.

## Выходные данные

Для каждого тестового примера требуется вывести строку с одним из трех сообщений:

- Если конь может победить:  
`Win in  $m$  knight move(s).`
- Если конь победить не может, но может добиться пата:  
`Stalemate in  $m$  knight move(s).`
- Если конь не может победить или добиться пата:  
`Loss in  $m$  knight move(s).`

Здесь  $m$  — минимальное число ходов, совершенных конем.

Время на решение тестовых примеров — две секунды.

## Оптимальное перемещение

В реальной игре, например в крестиках-ноликах или шахматах, каждый игрок может выбирать, как именно ему ходить. Но в нашем случае такой выбор есть только у коня. Пешка перемещается по жестко определенному маршруту, и ее положение всегда заранее известно. Это хорошая новость, потому что задача была бы намного сложнее, если бы выбор был у обоих игроков.

У коня есть разные способы победить или привести игру к ничьей. Предположим, что он может победить. Каждый способ, которым он может достичь победы, требует определенного числа ходов. Нам же нужно определить их минимальное количество.

## Анализ положения на доске

Разберем простой пример:

1  
7  
7  
1  
1  
4  
6

На этой доске по семь горизонталей и вертикалей. Пешка начинает с первого ряда первой вертикали, а конь с горизонтали 4 и вертикали 6.

При оптимальном перемещении конь может победить за три хода. На схеме ниже показано возможное решение:

<b>7</b>							
<b>6</b>		К2					
<b>5</b>				К1			
<b>4</b>	К3 Р3					К	
<b>3</b>	Р2						
<b>2</b>	Р1						
<b>1</b>	Р						
	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>

Здесь К обозначает стартовую позицию коня, Р — стартовую позицию пешки. К1, К2 и К3 указывают расположение коня после ходов 1, 2 и 3 соответственно. Р1, Р2 и Р3 обозначают перемещения пешки.

Координаты (х, у) относятся к ряду х, столбцу у. Как и было указано выше, пешка просто продвигается вверх по своей вертикали, начиная с (1, 1), к (2, 1), затем к (3, 1) и, наконец, к (4, 1). А вот конь движется следующим образом:

- Начиная с (4, 6), он перемещается на одну клетку вверх и на две влево, попадая в (5, 4). Пешка в этот момент находится в (2, 1).
- Из (5, 4) он переходит на одну клетку вверх и на две влево в (6, 2). Пешка находится в (3, 1).
- Из (6, 2) конь переходит на две клетки вниз и одну влево, оказываясь в (4, 1) и настигая там пешку!

У коня есть и другие способы победить. К примеру, вот что может произойти, если конь немного «схалтурит»:

<b>7</b>							
<b>6</b>		K2					
<b>5</b>	K4 P4			K1			
<b>4</b>	P3		K3			K	
<b>3</b>	P2						
<b>2</b>	P1						
<b>1</b>	P						
	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>

В этом случае конь съедает пешку уже за четыре хода, а не за три. Хотя он все равно побеждает, это *не* самый быстрый способ. Нам же нужно найти вариант с минимумом ходов.

Предположим, что у нас есть алгоритм для определения минимального числа ходов, которые потребуются коню, чтобы попасть из стартовой позиции в заданную. Тогда можно определить число ходов коня, необходимое для попадания в каждую точку нахождения пешки. Если конь сможет дойти туда одновременно с пешкой, то он победит. Если же он победить не может, тогда нужно проделать аналогичный расчет для патовых ситуаций: определить количество ходов коня, необходимое для попадания в клетку над каждой позицией пешки.

Чтобы построить такой алгоритм, можно проанализировать возможности ходов, начиная со стартовой позиции коня. На доске есть всего одна клетка, которой можно достичь за ноль ходов, — стартовая точка коня. Затем можно рассмотреть клетки, которые достижимы за один ход. Далее можно рассматривать те, которые достигаются на втором ходу, и т. д. Остановка произойдет, когда будет достигнута искомая клетка. Тогда мы и узнаем минимальное количество ходов, за которое в нее можно попасть.

Продemonстрируем этот процесс на том же тестовом примере: по семь горизонталей и вертикалей, конь начинает в (4, 6) (пешку пока проигнорируем). Давайте вычислим минимальное число ходов, необходимое коню, чтобы попасть из (4, 6) в (4, 1).

На схемах ниже числа в клетках указывают минимальное расстояние от стартовой точки коня. Как указано выше, единственная клетка, которой можно достичь за

ноль ходов, — это стартовая позиция, то есть (4, 6). Назовем это шагом 0 процесса анализа:

7							
6							
5							
4						0	
3							
2							
1							
	1	2	3	4	5	6	7

Из (4, 6) попробуем все восемь возможных ходов и определим клетки, достижимые за один ход. При этом нельзя двигаться на одну клетку вверх и на две вправо или на одну вниз и на две вправо, потому что так фигура выйдет за правую границу поля.

Таким образом, у коня остается шесть доступных клеток, расположенных в одном ходе от стартовой позиции. Это шаг 1:

7							
6					1		1
5				1			
4						0	
3				1			
2					1		1
	1	2	3	4	5	6	7

Оптимальную клетку (4, 1) мы еще не нашли, поэтому проводим анализ для каждой новой клетки, определенной на первом шаге. Так мы получим позиции, расположенные в двух ходах от стартовой. К примеру, рассмотрим клетку (6, 5); из нее можно попасть в следующие точки:

- вверх на 1, вправо на 2 — (7, 7);
- вверх на 1, влево на 2 — (7, 3);
- вниз на 1, вправо на 2 — (5, 7);
- вниз на 1, влево на 2 — (5, 3);
- вверх на 2, вправо на 1 — недопустимый ход;
- вверх на 2, влево на 1 — недопустимый ход;
- вниз на 2, вправо на 1 — (4, 6);
- вниз на 2, влево на 1 — (4, 4).

Эти клетки находятся в двух ходах от стартовой позиции — за исключением (4,6), чье значение (0) мы заполнили ранее. Рассмотрение всех допустимых ходов из всех клеток, расположенных в одном ходе от стартовой, составляет второй шаг анализа. Таким образом, мы находим клетки, удаленные уже на два хода:

<b>7</b>			2		2		2
<b>6</b>		2			1	2	1
<b>5</b>			2	1	2		2
<b>4</b>		2		2		0	
<b>3</b>			2	1	2		2
<b>2</b>		2			1	2	1
<b>1</b>			2		2		2
	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>

Обратите внимание, что других позиций коня, удаленных на два шага, быть не может. Каждая такая позиция должна исходить из клетки, расположенной в одном

ходе от стартовой, но мы уже исследовали все возможные ходы из всех клеток, удаленных на один ход.

Позиция (4, 1) по-прежнему не найдена, значит, мы продолжаем. Нам предстоит шаг 3, на котором рассматриваются все клетки, достижимые из тех, что удалены на два хода от старта. Так мы получаем позиции, достижимые за три хода коня:

<b>7</b>		3	2	3	2	3	2
<b>6</b>	3	2	3		1	2	1
<b>5</b>		3	2	1	2	3	2
<b>4</b>	3	2	3	2	3	0	3
<b>3</b>		3	2	1	2	3	2
<b>2</b>	3	2	3		1	2	1
<b>1</b>		3	2	3	2	3	2
	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>

Итак, клетка (4, 1) заполняется значением 3. Следовательно, для ее достижения из точки (4, 6) требуются минимум три хода. Не найди мы и на этом шаге необходимую клетку, пришлось бы продолжить поиск: искать те, что удалены на четыре хода, затем на пять и т. д.

Эта техника — нахождение всех клеток, находящихся в нуле ходов от старта, затем в одном ходе, затем в двух, трех и т. д., — называется *поиском в ширину* или BFS. Слово «ширина» подразумевает весь диапазон возможностей. При использовании BFS сначала выполняется поиск по всему диапазону, достижимому из каждой клетки, затем происходит переход к новым клеткам. Этот метод быстр, экономичен для памяти и понятен в реализации. Он очень эффективен, когда необходимо найти минимальное расстояние от одной позиции до другой. Его мы и задействуем!

### Реализация поиска в ширину

Начнем с определений типов данных, которые немного упростят код. Во-первых, каждая позиция на доске формируется горизонталью (рядом) и вертикалью (столбцом), поэтому их мы объединим в структуру:



```
typedef struct position {  
    int row, col;  
} position;
```

Во-вторых, игровая доска — это двумерный массив, для которого тоже можно определить тип. Мы позволим сохранять в нем числа, которые будут соответствовать количеству ходов. В задаче возможны не более 99 горизонталей и 99 вертикалей, но мы выделим одну дополнительную горизонталь и одну вертикаль, чтобы можно было начать их индексацию с 1, а не 0:

```
#define MAX_ROWS 99  
#define MAX_COLS 99  
  
typedef int board[MAX_ROWS + 1][MAX_COLS + 1];
```

И наконец, зададим тип массива для хранения позиций, обнаруживаемых в процессе BFS. Его размер мы определим достаточно большим, чтобы он мог уместить все клетки доски:

```
typedef position positions[MAX_ROWS * MAX_COLS];
```

Теперь можно переходить к самому поиску. Вот реализация соответствующей функции:

```
int find_distance(int knight_row, int knight_col,  
                 int dest_row, int dest_col,  
                 int num_rows, int num_cols)
```

Параметры `knight_row` и `knight_col` обозначают стартовую позицию коня, а `dest_row` и `dest_col` указывают целевую. Параметры `num_rows` и `num_cols` сообщают количество горизонталей и вертикалей доски: они понадобятся для определения допустимости хода. Эта функция возвращает минимальное количество ходов, необходимое для достижения конем целевой точки. Если возможности ее достичь нет, возвращается -1.

Поиском в ширину в данном случае управляют два массива:

- **cur\_positions** содержит позиции, найденные на текущем шаге BFS.
- **new\_positions** содержит позиции, найденные на следующем шаге BFS. К примеру, если **cur\_positions** содержит позиции, обнаруженные на шаге 3, то **new\_positions** будет хранить позиции на шаге 4.

Соответствующий код приведен в листинге 4.1.

**Листинг 4.1.** Определение минимального количества ходов коня методом BFS

```
int find_distance(int knight_row, int knight_col,  
                 int dest_row, int dest_col,  
                 int num_rows, int num_cols) {
```

```

positions cur_positions, new_positions;
int num_cur_positions, num_new_positions;
int i, j, from_row, from_col;
board min_moves;
for (i = 1; i <= num_rows; i++)
    for (j = 1; j <= num_cols; j++)
        min_moves[i][j] = -1;
min_moves[knight_row][knight_col] = 0; ❶
cur_positions[0] = (position){knight_row, knight_col}; ❷
num_cur_positions = 1;

while (num_cur_positions > 0) { ❸
    num_new_positions = 0;
    for (i = 0; i < num_cur_positions; i++) {
        from_row = cur_positions[i].row;
        from_col = cur_positions[i].col;
        if (from_row == dest_row && from_col == dest_col) ❹
            return min_moves[dest_row][dest_col];

        add_position(from_row, from_col, from_row + 1, from_col + 2, ❺
                     num_rows, num_cols, new_positions,
                     &num_new_positions, min_moves);
        add_position(from_row, from_col, from_row + 1, from_col - 2,
                     num_rows, num_cols, new_positions,
                     &num_new_positions, min_moves);
        add_position(from_row, from_col, from_row - 1, from_col + 2,
                     num_rows, num_cols, new_positions,
                     &num_new_positions, min_moves);
        add_position(from_row, from_col, from_row - 1, from_col - 2,
                     num_rows, num_cols, new_positions,
                     &num_new_positions, min_moves);
        add_position(from_row, from_col, from_row + 2, from_col + 1,
                     num_rows, num_cols, new_positions,
                     &num_new_positions, min_moves);
        add_position(from_row, from_col, from_row + 2, from_col - 1,
                     num_rows, num_cols, new_positions,
                     &num_new_positions, min_moves);
        add_position(from_row, from_col, from_row - 2, from_col + 1,
                     num_rows, num_cols, new_positions,
                     &num_new_positions, min_moves);
        add_position(from_row, from_col, from_row - 2, from_col - 1,
                     num_rows, num_cols, new_positions,
                     &num_new_positions, min_moves);
    }

    num_cur_positions = num_new_positions; ❻
    for (i = 0; i < num_cur_positions; i++)
        cur_positions[i] = new_positions[i];
}
return -1;
}

```

Вначале мы очищаем массив `min_moves`, устанавливая все его значения равными `-1`, поскольку мы еще не вычислили количество ходов. Единственная клетка, для которой нам известно минимальное число ходов, — это стартовая позиция коня, которую мы инициализируем как `0` ❶. Эта стартовая клетка также является точкой, откуда начинается BFS ❷. Цикл `while` выполняется до тех пор, пока ближайший шаг BFS не обнаружит хотя бы одну новую клетку ❸. В цикле `while` мы рассматриваем каждую такую позицию. Если при этом обнаруживается целевая ❹, то возвращается минимальное количество ходов. В противном случае поиск продолжается.

Анализ всех восьми возможных ходов из заданной клетки реализуется через восемь вызовов вспомогательной функции `add_position`. Она добавляет новые позиции в `new_positions` и соответствующим образом обновляет `num_new_positions`. Рассмотрим первые четыре параметра: они сообщают текущие горизонталь и вертикаль, а также новые координаты, получающиеся в результате одного из восьми ходов. К примеру, первый вызов ❺ относится к ходу, перемещающему фигуру на две клетки вверх и на одну вправо. Код для `add_position` мы рассмотрим ниже.

Мы прошли через каждую клетку в `cur_positions` и нашли новые клетки, находящиеся на один ход дальше. На этом завершается соответствующий шаг BFS. Чтобы подготовиться к следующему, мы копируем новые позиции из `new_positions` в `cur_positions` ❻. Таким образом, в очередной итерации цикла `while` эти новые клетки будут использованы для нахождения последующих позиций.

Если при достижении нижней строки кода целевая позиция так и не будет достигнута, вернется `-1`, сообщая, что достичь целевой клетки из стартовой позиции коня невозможно.

А теперь перейдем к вспомогательной функции `add_position`, которая представлена в листинге 4.2.

#### Листинг 4.2. Добавление позиции

```
void add_position(int from_row, int from_col,
                 int to_row, int to_col,
                 int num_rows, int num_cols,
                 positions new_positions, int *num_new_positions,
                 board min_moves) {
    struct position new_position;
    if (to_row >= 1 && to_col >= 1 &&
        to_row <= num_rows && to_col <= num_cols &&
        min_moves[to_row][to_col] == -1) {
        min_moves[to_row][to_col] = 1 + min_moves[from_row][from_col]; ❶
        new_position = (position){to_row, to_col};
        new_positions[*num_new_positions] = new_position;
        (*num_new_positions)++;
    }
}
```

Инструкция `if` содержит пять условий, все они должны быть выполнены, чтобы `to_row` и `to_col` оказалась допустимой позицией: номера горизонтали и вертикали должны быть не меньше 1 и не больше общего числа рядов и столбцов соответственно. А что означает последнее условие `min_moves[to_row][to_col]`?

Оно определяет, встречалась ли ранее эта клетка. Если нет, то у нее будет значение `-1`, то есть можно записывать количество ходов. Если же значение отлично от `-1`, значит, оно было записано на одном из предыдущих шагов BFS, то есть эта позиция уже была достигнута за меньшее число ходов, чем можно сохранить сейчас. Таким образом, любое значение, кроме `-1`, указывает, что минимальное количество ходов уже установлено и менять его не надо.

Если все пять условий выполняются, значит, мы нашли новую позицию. Позиция `(from_row, from_col)` была найдена на предыдущем шаге BFS, а `(to_row, to_col)` — на текущем. Следовательно, минимальное количество ходов до `(to_row, to_col)` на один больше, чем минимальное количество ходов до `(from_row, from_col)` ❶. Благодаря тому, что `(from_row, from_col)` передается с предыдущего шага, соответствующее значение уже сохранено в `min_moves`, и мы его находим, не вычисляя повторно.

Вы можете заметить здесь характерные признаки мемоизации и динамического программирования. Все верно: в BFS используется тот же прием поиска элементов вместо их повторного вычисления. Однако в данном случае нет понятий максимизации или минимизации на основе решений подзадач или совмещения этих решений для получения общего. Поэтому создатели алгоритмов обычно не относят BFS к динамическому программированию, а классифицируют его как алгоритм поиска или анализа.

## Лучший результат коня

Мы создали BFS, красиво оформленный в функции `find_distance`. Теперь определим количество ходов с учетом перемещения пешки вверх по ее вертикали и используем `find_distance`, чтобы выяснить, удастся ли коню настичь пешку. К примеру, если пешка совершает три хода, оказываясь в некоторой клетке, и конь тоже может сделать три хода, оказавшись в этой же позиции, то он побеждает. Если же конь не может победить, тогда мы пробуем аналогичную технику для поиска патовых ситуаций: пешка вновь перемещается по своей вертикали, а мы на этот раз проверяем, сможет ли конь организовать пат. Если и патовые ситуации невозможны, тогда конь проигрывает. Эта логика прописана в листинге 4.3. Функция `solve` получает шесть параметров: стартовые горизонталь и вертикаль пешки, стартовые горизонталь и вертикаль коня, а также общее количество горизонталей и вертикалей на доске. В результате она выводит одну строку с результатом: победой коня, его проигрышем или ничьей.

**Листинг 4.3.** Лучший результат коня (с ошибкой!)

```

void solve(int pawn_row, int pawn_col, //с ошибкой!
           int knight_row, int knight_col,
           int num_rows, int num_cols) {
    int cur_pawn_row, num_moves, knight_takes;

    cur_pawn_row = pawn_row; ❶
    num_moves = 0;
    while (cur_pawn_row < num_rows) {
        knight_takes = find_distance(knight_row, knight_col,
                                    cur_pawn_row, pawn_col,
                                    num_rows, num_cols);
        if (knight_takes == num_moves) { ❷
            printf("Win in %d knight move(s).\n", num_moves);
            return;
        }
        cur_pawn_row++;
        num_moves++;
    }

    cur_pawn_row = pawn_row; ❸
    num_moves = 0;
    while (cur_pawn_row < num_rows) {
        knight_takes = find_distance(knight_row, knight_col,
                                    cur_pawn_row + 1, pawn_col,
                                    num_rows, num_cols);
        if (knight_takes == num_moves) {
            printf("Stalemate in %d knight move(s).\n", num_moves);
            return;
        }
        cur_pawn_row++;
        num_moves++;
    }

    printf("Loss in %d knight move(s).\n", num_rows - pawn_row - 1); ❹
}

```

Разберем три части листинга.

Первая часть — это код, проверяющий, может ли конь победить. Начинаем с сохранения горизонтали пешки в новой переменной ❶ — это значение мы будем изменять, перемещая пешку по доске, поэтому нужно запомнить, откуда она начала движение. Цикл `while` выполняется до тех пор, пока пешка не достигнет верхнего ряда. В каждой итерации мы вычисляем количество ходов, которые нужно сделать коню, чтобы попасть в одну клетку с пешкой. Если коню удастся это сделать ❷, то он побеждает. Если же он победить не может, то пешка достигнет верхней горизонтали доски и мы продолжим выполнять код под циклом `while`.

Здесь начинается вторая часть ❸. Задача — определить, может ли конь добиться пата. Код здесь такой же, что и в первом фрагменте, за исключением того, что в цикле `while` проверяется количество ходов, необходимое коню для достижения горизонтали над пешкой, а не той горизонтали, в которой находится пешка.

Третья часть представлена одной строкой ❹ и выполняется, только если конь не может ни победить, ни добиться пата. Этот код просто выводит сообщение о поражении.

Так обрабатывается один тестовый пример. Чтобы обработать все, потребуется небольшая функция `main`, которая показана в листинге 4.4.

#### Листинг 4.4. Функция `main`

```
int main(void) {
    int num_cases, i;
    int num_rows, num_cols, pawn_row, pawn_col, knight_row, knight_col;
    scanf("%d", &num_cases);
    for (i = 0; i < num_cases; i++) {
        scanf("%d%d", &num_rows, &num_cols);
        scanf("%d%d", &pawn_row, &pawn_col);
        scanf("%d%d", &knight_row, &knight_col);
        solve(pawn_row, pawn_col, knight_row, knight_col,
              num_rows, num_cols);
    }
    return 0;
}
```

Довольны? Мы получили завершенное решение. С помощью BFS мы оптимизируем количество ходов, совершаемых конем, а также проверяем, побеждает он, проигрывает или же добивается ничьей. Попробуйте отправить решение на проверку. Все еще довольны?

### Блуждающий конь

В главах 1 и 3 мы уже встречались с решениями, которые были верными, но слишком медленными для успешного прохождения проверки. Здесь же я предоставил принципиально *неверное* решение, которое для некоторых тестовых примеров выдаст ошибочный вывод (честно говоря, быстроедействие решения также оставляет желать лучшего).

#### Исправляем ошибки

Принципиальная ошибка состоит в том, что не учтен вариант, когда конь оказывается «слишком быстр». То есть он может достигнуть целевой позиции до того, как

в эту клетку попадет пешка. Но и проверка на равенство числа ходов пешки и коня слишком ограничивающая. Это прояснит следующий тестовый пример:

1  
5  
3  
1  
1  
3  
1

Дана доска с пятью горизонталями и тремя вертикалями. Пешка начинает с первого ряда вертикали 1, а конь — с ряда 3 вертикали 1. Вывод текущей версии кода для этого кейса будет таким:

```
Loss in 3 knight move(s).
```

Выводится 3, а не 4, потому что коню не разрешается ходить после того, как пешка достигнет верхнего ряда. Это означает, что нет такой победной или патовой позиции, для которой минимальное количество ходов коня будет совпадать с числом ходов пешки. По крайней мере, тут все верно. Однако в решении упущен вариант победы коня, причем в два хода. Уделите немного времени, чтобы найти это решение.

У коня нет шанса победить за один ход, когда пешка находится в (2, 1). Тем не менее после второго хода пешка оказывается в (3, 1), и конь также может занять эту позицию за два хода. Вот каким образом:

- Ход 1: перейти из (3, 1) в (5, 2).
- Ход 2: вернуться из (5, 2) в (3, 1).

Минимальное число ходов коня для попадания в (3, 1) равно нулю, так как это его стартовая позиция. При этом он может сходить на другую клетку и потом вернуться обратно, на что уйдет два хода.

Проверим себя: измените стартовую позицию коня с (3, 1) на (5, 3). Сможете ли вы определить, как конь победит в три хода?

Обобщенно можно сказать, что если конь может попасть в клетку минимум за  $m$  ходов, то он также может попасть в нее за  $m + 2$ ,  $m + 4$  и т. д. ходов. Все, что ему нужно сделать для этого — повторять перемещение в другую клетку и возвращение.

Для нашего решения это означает, что на каждом шаге у коня есть два способа победить или устроить пат: либо если его минимальное количество ходов совпадает с количеством ходов пешки, либо если его минимальное количество ходов на четное число больше количества ходов пешки.

Таким образом, вместо

```
if (knight_takes == num_moves) {
```

нам нужно вот что:

```
if (knight_takes >= 0 && num_moves >= knight_takes &&
    (num_moves - knight_takes) % 2 == 0) {
```

Здесь мы проверяем, кратна ли двум разница между числами ходов пешки и коня.

В коде листинга 4.3 присутствуют две ошибки. Исправив обе, мы получим верный код, приведенный в листинге 4.5.

#### Листинг 4.5. Лучший результат коня

```
void solve(int pawn_row, int pawn_col,
           int knight_row, int knight_col,
           int num_rows, int num_cols) {
    int cur_pawn_row, num_moves, knight_takes;

    cur_pawn_row = pawn_row;
    num_moves = 0;
    while (cur_pawn_row < num_rows) {
        knight_takes = find_distance(knight_row, knight_col,
                                    cur_pawn_row, pawn_col,
                                    num_rows, num_cols);

        if (knight_takes >= 0 && num_moves >= knight_takes && ❶
            (num_moves - knight_takes) % 2 == 0) {
            printf("Win in %d knight move(s).\n", num_moves);
            return;
        }
        cur_pawn_row++;
        num_moves++;
    }

    cur_pawn_row = pawn_row;
    num_moves = 0;
    while (cur_pawn_row < num_rows) {
        knight_takes = find_distance(knight_row, knight_col,
                                    cur_pawn_row + 1, pawn_col,
                                    num_rows, num_cols);

        if (knight_takes >= 0 && num_moves >= knight_takes && ❷
            (num_moves - knight_takes) % 2 == 0) {
            printf("Stalemate in %d knight move(s).\n", num_moves);
            return;
        }
        cur_pawn_row++;
        num_moves++;
    }

    printf("Loss in %d knight move(s).\n", num_rows - pawn_row - 1);
}
```



Как и было сказано, мы всего лишь изменили два условия ❶ ❷. Теперь этот код пройдет проверку.

### Проверяем себя

Если вы достаточно убеждены в верности последнего решения, можете этот раздел не читать. В противном случае я хочу развеять сомнения, которые у вас могли возникнуть.

Предположим, что конь попадает в целевую клетку на четное количество ходов раньше пешки и что на это уходит  $m$  ходов. Также предположим, что конь покидает эту клетку и возвращается в нее столько раз, сколько нужно, оказываясь в ней после  $m + 2$ ,  $m + 4$  и т. д. ходов, и в конечном итоге съедает пешку. Было бы замечательно, если бы конь мог использовать иную последовательность действий для поимки пешки через  $m + 1$ ,  $m + 3$  и т. д. ходов, потому что добавление нечетного количества ходов дало бы лучший минимум, чем добавление четного. Однако так произойти не может.

Попробуйте небольшой эксперимент: выберите для коня стартовую и целевую клетки, а затем найдите минимальное количество ходов, необходимое ему для перемещения до цели. Обозначим это количество ходов как  $m$ . А теперь попробуйте найти способ, которым конь попадет из той же стартовой точки в ту же целевую, используя на один ход больше, на три больше и т. д. К примеру, если кратчайший путь занимает два хода, попробуйте найти способ сделать три. У вас это не получится.

При каждом ходе коня номер его горизонтали изменяется на два, а вертикали — на один, или наоборот. К примеру, номер горизонтали может смениться с шести на четыре, а номер вертикали — с четырех на пять. Изменение номера на два не делает его из четного нечетным и наоборот, а вот изменение на один — делает. Это значит, что в смысле четности или нечетности при каждом ходе один из двух номеров (горизонтали или вертикали) остается прежним, а другой меняется. Когда номер изменяется с четного на нечетный или наоборот, мы говорим, что меняется его *четность*.

Пусть  $k$  будет нечетным целым числом. Теперь можно рассмотреть, почему конь, совершив  $m$  ходов, не сможет попасть в ту же клетку, что и за  $m + k$  ходов. Предположим, что конь может сделать  $m$  ходов, чтобы оказаться в клетке  $s$ , что  $m_1$  из этих ходов изменяют четность по горизонтали, а  $m_2$  — по вертикали.

Допустим, что  $m_1$  и  $m_2$  — четные числа. В этом случае такие ходы не могут изменять четность по горизонтали или вертикали: если мы начнем с некоторого числа и сменим его четность четное количество раз, то в итоге она не изменится. Если же

мы совершаем некоторую последовательность ходов и она изменяет четность по горизонтали или вертикали нечетное количество раз, то эта последовательность не может вернуть в  $s$ , потому что приведет к клетке, четность горизонтали или вертикали которой будет отличаться от четности  $s$ .

Итак,  $m$ , представляющее общее количество ходов  $m_1 + m_2$ , — четное, так как суммой двух четных чисел является четное число. Но  $m + k$  должно быть нечетным, а поскольку  $m + k$  — нечетное, оно не может состоять из четных чисел ходов и по горизонтали, и по вертикали: хотя бы одно из этих значений должно быть нечетным и, следовательно, менять четность горизонтали или вертикали. Именно поэтому  $m + k$  ходов не могут привести к завершению перемещения коня в  $s$ . Не будем рассматривать три других случая —  $m_1$  четное,  $m_2$  нечетное;  $m_1$  нечетное,  $m_2$  четное;  $m_1$  и  $m_2$  нечетные, — так как суть будет та же.

## Оптимизация времени

Наше текущее решение (листинг 4.5) может совершать множество вызовов BFS. Каждый раз, когда пешка перемещается вверх на один ряд, мы используем BFS (вызывая `find_distance`) для определения, может ли конь ее поймать.

Предположим, что пешка начинает в клетке  $(1, 1)$ . Мы запускаем поиск из стартовой позиции коня до  $(1, 1)$ , и он анализирует варианты. Допустим, что конь не может поймать здесь пешку. Тогда нужно запустить BFS из стартовой позиции коня до  $(2, 1)$ . Так мы проверим еще некоторые ходы. Однако  $(1, 1)$  и  $(2, 1)$  очень близки, и второй поиск наверняка повторно обнаружит многие ходы в клетки, кратчайшие расстояния до которых были найдены еще при первом вызове BFS. К сожалению, вызовы BFS независимы друг от друга, поэтому второй повторно проделывает часть работы первого. Третий вызов, в свою очередь, повторяет много действий за всеми предшествующими вызовами и т. д.

BFS, безусловно, быстр, и в следующем разделе я поясню причины этого более подробно. Тем не менее все же стоит попробовать уменьшить количество вызовов.

И здесь есть хорошие новости: можно уменьшить число вызовов BFS вплоть до одного! Вспомните BFS из листинга 4.1. Там прописан код ❹, сокращающий поиск целевой позицией. Однако если этот код удалить, то BFS будет исследовать всю доску, вычисляя кратчайшее расстояние до каждой клетки. Внесение этого изменения позволит сделать всего один вызов BFS, а затем мы будем просто искать интересующие нас данные в массиве `min_moves`.

Сделайте это. Внесите необходимые изменения в код, чтобы BFS вызывался только один раз.

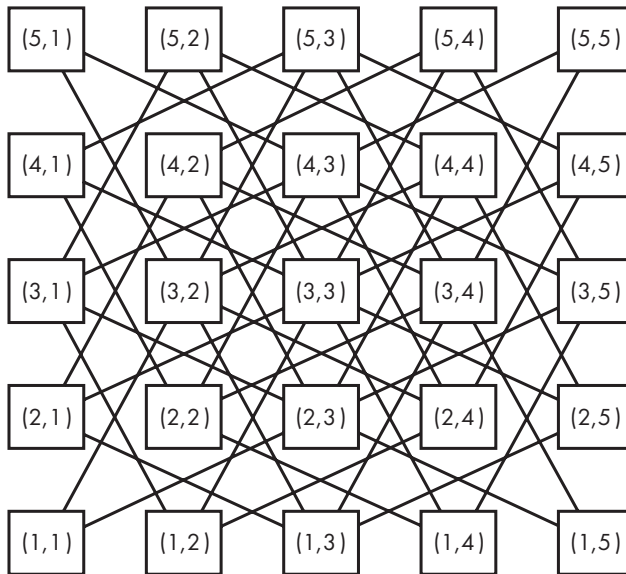
Неоптимизированный код решения этой задачи выполняется за 0,1 секунды. Внесение же указанной оптимизации для использования всего одного вызова BFS уменьшает время расчетов до 0,02 секунды, то есть прирост скорости составляет 500%. Причем такая оптимизация дает возможность использовать BFS для нахождения кратчайшего расстояния от стартовой позиции не только до некоторой другой, но и до *всех* других возможных позиций. В следующем разделе мы поговорим о BFS более подробно. Я думаю, что гибкость этого метода поиска вас весьма удивит.

## Графы и BFS

В задаче с погоней за пешкой мы убедились, что BFS — мощный поисковый алгоритм. Для его выполнения требуется структура, называемая *графом*. Решая предыдущую задачу, мы просто не задумывались об его построении, но на самом деле в основе BFS лежал именно граф.

### Что такое графы?

На рис. 4.1 приведен пример графа.



**Рис. 4.1.** Граф ходов коня

Как и дерево, граф состоит из *вершин* (квадратов) и связывающих их *ребер* (линий). В приведенном выше графе ребра указывают допустимые перемещения коня. К примеру, из вершины (5, 1) конь может перейти по двум ребрам: в (4, 3) либо в (3, 2). Других ребер от (5, 1) нет, значит, из этой позиции больше ходить некуда.

Теперь можно объяснять, как мы неявно использовали граф в решении задачи с конем и пешкой. Предположим, что (5, 1) — это стартовая позиция коня. Наш BFS пробует все восемь ходов из этой точки, но шесть из них ведут за границы доски. На языке графа шесть из них не имеют ребер, идущих от (5, 1). BFS находит две допустимые вершины, которые достижимы по ребрам от (5, 1) к (4, 3) и от (5, 1) к (3, 2). Далее исследование продолжается по ребрам от этих двух вершин и т. д.

Я начертил граф в виде сетки, чтобы наглядно отображалась его основа — шахматная доска. На практике же способ изображения графа значения не имеет. Важны лишь его вершины и ребра. Можно даже изобразить граф с хаотично разбросанными вершинами, но смысл его будет прежним. Однако если граф привязан к определенной внутренней геометрии, лучше отображать его соответствующим образом для облегчения понимания.

Чтобы решить задачу погони за пешкой, нам не требовалось явно представлять граф в коде, так как мы выясняли возможные ходы (ребра) из каждой вершины по мере продвижения по шахматной доске. И все же иногда граф нужно явно выражать в коде, подобно линиям представлений деревьев в главе 2. Как это делается, мы увидим при решении задачи 3.

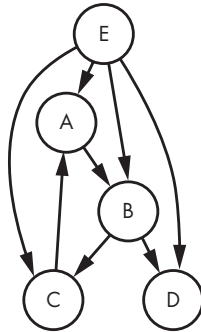
## Графы и деревья

У графов и деревьев много общего. Обе эти структуры используются для представления связей между вершинами (узлами). На деле каждое дерево является графом, но при этом существуют графы, которые не относятся к деревьям. По своей сути графы способны выражать более сложные схемы связей.

Например, в графах возможны циклы. Наличие в графе *цикла* определяется возможностью начать с вершины и вернуться к ней без повторного использования ребер или вершин (повторяются только первая и последняя вершины в цикле). Рассмотрим еще раз рис. 4.1. В этом графе мы видим цикл  $(5, 3) \rightarrow (4, 5) \rightarrow (3, 3) \rightarrow (4, 1) \rightarrow (5, 3)$ .

Кроме того, графы могут быть *направленными*. Деревья и графы, которые мы рассматривали до сих пор, были *ненаправленными*, то есть если два узла, *a* и *b*, связаны

ребром, то можно проходить по этому ребру в обоих направлениях, от  $a$  к  $b$  и от  $b$  к  $a$ . Граф на рис. 4.1 также ненаправленный. Например, можно перейти по ребру из  $(5, 3)$  в  $(4, 5)$  и использовать то же ребро для перехода обратно из  $(4, 5)$  в  $(5, 3)$ . Однако иногда требуется ограничить перемещение только одним направлением. *Направленный граф* — это граф, в котором каждое ребро указывает допустимое направление движения. Его пример показан на рис. 4.2.



**Рис. 4.2.** Направленный граф

В этом графе можно перемещаться из  $E$  к любой другой вершине, но не обратно. Таким образом, ребра являются однонаправленными.

Направленные графы по сравнению с ненаправленными позволяют отразить дополнительную информацию. На кафедре компьютерных наук, где я преподаю, для некоторых курсов требуется предварительное прохождение одного или нескольких других. Например, у нас есть курс *Программирование на Си*, который требует от студентов предварительного прохождения курса *Проектирование ПО*. Эта связь отражается направленным ребром *проектирование ПО*  $\rightarrow$  *программирование на Си*. Если же использовать здесь ненаправленное ребро, то мы по-прежнему будем знать о связи курсов, но не о последовательности, в которой они должны быть пройдены. На рис. 4.3 показан краткий граф очередности прохождения курсов.

Третья черта, которая делает графы более универсальными, чем деревья, — возможность *несвязности*. Все деревья и графы, которые мы до сих пор встречали, были *связными*. Это значит, что в них можно попасть из любого узла в любой другой узел. А теперь взгляните на образец несвязного графа на рис. 4.4.

Несвязный он, потому что, например, в нем нельзя пройти из *Введения в программирование* к *Истории древнего мира*. Несвязность возникает, когда граф естественным образом составлен из разделенных частей.

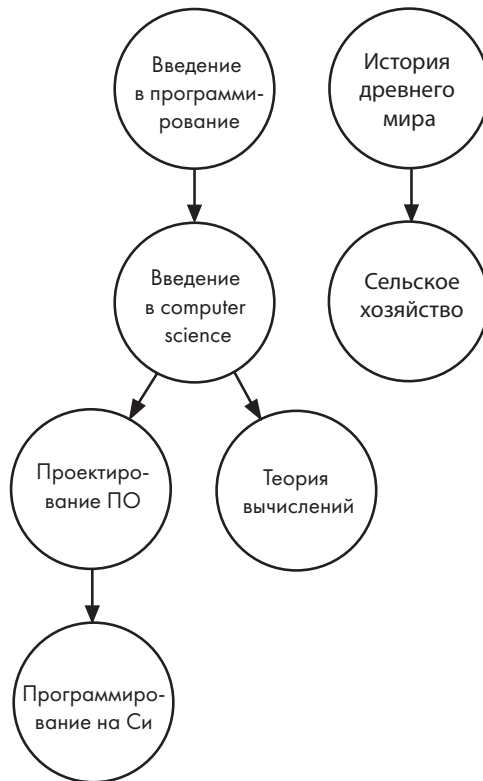


Рис. 4.3. Граф очередности прохождения курсов

## BFS в графах

BFS применим не только для ненаправленных графов (как мы делали в задаче с конем и пешкой), но и для направленных. Алгоритм при этом будет тот же: мы совершаем допустимые ходы из текущей вершины, попутно исследуя возможности. BFS известен как алгоритм *кратчайшего пути*: из всех путей между двумя вершинами он позволяет найти кратчайший в смысле количества ребер. Если нас интересует минимизация количества ребер, он решает задачу нахождения *кратчайшего пути из исходной точки*.

Быстрота BFS определяется не тем, направленный он исследует граф или ненаправленный, а тем, сколько раз мы его вызываем, а также числом ребер в графе. Время выполнения вызова BFS пропорционально количеству ребер, достижимых из стартовой точки. Дело в том, что BFS просматривает каждое ребро один раз,



**Рис. 4.4.** Граф с несвязной очередностью прохождения курсов

определяя, ведет ли оно к новой вершине. Мы называем BFS алгоритмом с линейным временем, так как его скорость линейна по отношению к количеству ребер: если на исследование 5 ребер уходит 5 шагов, то на 10 ребер уйдет 10 шагов. Поэтому мы будем использовать число ребер для оценки количества выполняемых BFS шагов.

В задаче с погоней за пешкой была дана доска с  $r$  рядов и  $c$  столбцов. Каждая вершина имеет не более восьми ребер, значит, по всей доске может быть максимум  $8rc$  ребер. Следовательно, выполнение одного поиска потребует  $8rc$  шагов. Для самой большой доски размером  $99 \times 99$  получится менее 80 000 шагов. Если же мы сделаем  $r$  или более вызовов BFS, как может произойти в листинге 4.5, то будет выполнено  $8r^2c$  шагов. Теперь анализ доски  $99 \times 99$  уже не выглядит столь простой задачей: для ее обхода может потребоваться более семи миллионов шагов. Именно поэтому нам очень помогает уменьшение количества вызовов BFS.

Всякий раз, когда задача подразумевает набор связанных объектов (позиций на доске, курсов, людей и т. д.), целесообразно попробовать смоделировать ее в виде

графа. После создания такой модели можно использовать большое число быстрых алгоритмов обработки графа, одним из которых является BFS.

## Задача 2. Лазание по канату

Рассмотрим задачу с DMOJ под номером `ws18c1s3`.

В задаче про коня и пешку игра проходила на доске известных размеров. Здесь же такой наглядности не будет, поэтому придется самим определять схему. Но стратегия снова будет состоять в моделировании допустимых ходов при помощи BFS.

### Условие

На уроке физкультуры одному из учеников, Бобу, нужно взобраться вверх по канату. Канат этот бесконечно длинный, но Боба просят забраться только на высоту не менее  $h$  метров.

Боб начинает с нулевой высоты. При движении вверх он обучен только одному действию — прыжку ровно на  $j$  метров. То есть если  $j$  равно 5, то он не сможет подпрыгнуть на четыре, шесть или любое другое число метров, только на 5. Кроме того, Боб умеет спускаться вниз на любое целое число метров: один, два, три и т. д.

Каждое подпрыгивание или спуск считается одним движением. К примеру, если Боб подпрыгивает на пять метров, затем спускается на два, потом снова подпрыгивает на пять и следом спускается на восемь, то считается, что он совершил четыре действия.

А теперь самая забавная часть условия: Алиса нанесла на некоторые участки каната чесоточный порошок. Если такой участок простирается от высоты  $a$  до высоты  $b$ , то весь он, включая конечные точки  $a$  и  $b$ , считается покрытым порошком. Оказавшись на участке с чесоточным порошком, Боб не может ни подпрыгнуть вверх, ни спуститься вниз.

Цель — определить минимальное количество действий, необходимое Бобу для достижения высоты  $h$  или более.

### Входные данные

Входные данные содержат один тестовый пример, состоящий из следующих строк:

- Строка с целыми числами  $h$ ,  $j$  и  $n$ , где  $h$  — минимальная высота, которую должен достичь Боб;  $j$  — высота его прыжка;  $n$  — количество участков, на которые Алиса нанесла чесоточный порошок. Каждое из этих чисел не более 1 000 000, а  $j$  меньше или равно  $h$ .



- $n$  строк, каждая из которых содержит два целых числа. Первое число задает высоту начала участка каната, покрытого порошком, а второе — высоту конца этого участка. Каждое из этих чисел не может быть больше  $h - 1$ .

### Выходные данные

Требуется вывести минимальное количество действий, необходимых Бобу, чтобы достичь высоты  $h$  или выше. Если такого способа нет, следует вывести  $-1$ .

Время на решение тестового примера — четыре секунды.

### Решение 1. Поиск возможностей

Начнем со сравнения с задачей о погоне за пешкой. Обратите внимание, что в обоих случаях наша цель — минимизировать количество действий или ходов. Будь то конь на доске или Боб на канате, задача аналогична. Можно отметить, что конь перемещался по двумерной доске, а Боб по одномерному канату, но это лишь меняет подход к описанию их позиций. Самому алгоритму поиска неважно, будет ли он обрабатывать два измерения или одно. Если здесь и есть отличие, то только в том, что меньшее число измерений несколько упрощает поиск.

А что с числом возможных действий в каждой позиции? Для коня их было не более восьми. В противоположность этому число действий Боба зависит от его позиции. Например, если он находится на высоте 4 метра, то у него есть пять возможных действий: подпрыгнуть вверх; спуститься вниз на один, два, три или четыре метра. Если же Боб находится на высоте 1000, то у него есть 1001 возможное действие. Поэтому при определении числа доступных действий мы будем учитывать текущую позицию Боба.

А как же чесоточный порошок? В погоне за пешкой ничего подобного не было. Давайте рассмотрим тестовый пример, чтобы понять, с чем мы имеем дело:

```
10 4 1
8 9
```

Бобу нужно забраться на высоту не менее 10 метров. Подпрыгнуть он может на 4. Значит, если на канате порошка не будет, то он сможет подпрыгнуть с 0 до 4, потом до 8 и затем до 12 метров. Всего три действия.

А теперь внимание: Бобу нельзя прыгать с 4 до 8 метров, потому что на высоте 8 начинается участок с чесоточным порошком (согласно условию, он нанесен на высоте от 8 до 9 метров). С учетом наличия на канате порошка на достижение цели уходит уже четыре действия. Например, Боб может подпрыгнуть с 0 до 4, затем спуститься до 3, затем подпрыгнуть до 7 и совершить завершающий прыжок до 11 метров. Последним прыжком он проскакивает участок с порошком.

На первый взгляд у Боба есть возможность прыгнуть с 4 до 8 метров, ведь он умеет прыгать на 4 метра, но по факту этому мешает чесоточный порошок. Данная ситуация, по сути, аналогична случаю, когда конь не мог ходить из-за того, что ему мешали границы доски. Недопустимые ходы коня мы обнаруживали в BFS и в следующий шаг анализа позиций их не включали. Случай с чесоточным порошком мы обработаем похожим образом: любое действие, которое приведет к попаданию Боба на участок с порошком, код BFS будет запрещать.

Вернемся к недопустимым ходам коня, выводящим его за пределы доски: нужно ли нам в текущей задаче беспокоиться о чем-то подобном? Канат бесконечно длинный, так что мы не нарушим правил, позволив Бобу взбираться все выше и выше. Однако в какой-то момент ему нужно будет остановиться. В противном случае BFS будет бесконечно искать и изучать новые позиции. Я прибегну к доводам из задачи «Экономные покупатели» главы 3, которые помогли избежать подобной проблемы при покупке яблок. Тогда мы решили, что если требуется купить не менее 50 яблок, то нужно рассматривать покупку не более 149 штук, потому что в ценовую схему может быть включено не более 100 яблок. Здесь же по условию задачи  $j$ , то есть высота прыжка Боба, не может быть больше  $h$ , то есть минимальной целевой высоты. Поэтому нам не следует позволять Бобу взбираться на высоту  $2h$  и более. Подумайте, что означает нахождение Боба на высоте  $2h$  или более? Тогда перед последним действием Боб уже находился на высоте  $2h - j$ , и на достижение этой высоты Бобу потребовалось на одно действие меньше, чем на достижение  $2h$ . Это значит, что покорение Бобом высоты  $2h$  или более не может быть самым быстрым способом взобраться не ниже  $h$ .

### Реализация поиска в ширину

Я использую в качестве основы решение задачи о погоне за пешкой, внося в него необходимые изменения. Тогда каждая позиция коня состояла из ряда и столбца, и для хранения этих элементов данных я создал специальную структуру. Здесь же позиция на канате является простым целым числом, поэтому структура для нее не потребуется. Начнем с определения типов для «доски» и определяемых позиций:

```
#define SIZE 1000000

typedef int board[SIZE * 2];
typedef int positions[SIZE * 2];
```

Конечно, несколько странно называть канат «доской» (board), но так как служить этот массив будет в точности той же цели, что и в задаче с конем и пешкой, то оставим это имя.

Мы будем совершать один вызов BFS, вычисляющий минимальное число действий, за которое Боб может добраться из нулевой позиции в каждую допустимую.

Сравните код для BFS из листинга 4.6 с кодом `find_distance` из листинга 4.1, а также с тем его вариантом, который, надеюсь, вы написали после прочтения раздела «Оптимизация времени» на с. 178.

**Листинг 4.6.** Вычисление минимального числа действий Боба при помощи BFS

```
void find_distances(int target_height, int jump_distance,
                  int itching[], board min_moves) {
    static positions cur_positions, new_positions;
    int num_cur_positions, num_new_positions;
    int i, j, from_height;
    for (i = 0; i < target_height * 2; i++)
        min_moves[i] = -1; ❶
    min_moves[0] = 0;
    cur_positions[0] = 0;
    num_cur_positions = 1;

    while (num_cur_positions > 0) {
        num_new_positions = 0;
        for (i = 0; i < num_cur_positions; i++) {
            from_height = cur_positions[i];

            add_position(from_height, from_height + jump_distance, ❷
                        target_height * 2 - 1,
                        new_positions, &num_new_positions,
                        itching, min_moves);
            for (j = 0; j < from_height; j++) ❸
                add_position(from_height, j,
                            target_height * 2 - 1,
                            new_positions, &num_new_positions,
                            itching, min_moves);
        }

        num_cur_positions = num_new_positions;
        for (i = 0; i < num_cur_positions; i++)
            cur_positions[i] = new_positions[i];
    }
}
```

У функции `find_distance` есть четыре параметра:

- **target\_height** — минимальная высота, на которую нужно взобраться. Это значение  $h$  из условия тестового примера.
- **jump\_distance** — высота, на которую Боб может подпрыгнуть. Это значение  $j$  из условия тестового примера.
- **itching** — параметр, указывающий на наличие на канате чесоточного порошка. Если `itching[i]` равен 0, значит, на высоте  $i$  порошка нет. В противном случае он есть (далее нужно будет составить массив для покрытых порошком

участков каната, но это несложный процесс, после чего нам уже не придется беспокоиться об отдельных участках — можно будет просто обращаться к индексам созданного массива).

- **min\_moves** — массив, в котором будет сохраняться минимальное число ходов, необходимое для достижения каждой позиции.

Как и в листинге 4.1 задачи про коня и пешку, каждая позиция инициализируется со значением **-1** ❶, поскольку BFS эту позицию еще не нашел. При этом используется индекс одномерного (не двумерного) массива. В остальном структура кода BFS аналогична решению предыдущей задачи про шахматы.

Но при этом в коде, добавляющем позиции, все же есть интересное структурное изменение. Боб может подпрыгивать только на одну высоту, поэтому рассматривается всего одно действие прыжка ❷: Боб начинает в `from_height` и заканчивает, если такая позиция допустима, в `from_height + jump_distance`. Максимальная высота, которой Бобу разрешено достичь, задается выражением `target_height * 2 - 1`. Для спуска же жестко закодировать доступные действия нельзя, так как они зависят от высоты его текущей позиции.

Для обработки спуска мы используем цикл ❸, просматривая все целевые высоты от нуля до (но не включительно) `from_height` — текущей позиции Боба. Этот цикл является единственным существенным отличием от BFS в задаче про коня и пешку.

Для завершения кода BFS нужно реализовать вспомогательную функцию `add_position`. Ее код приведен в листинге 4.7.

#### Листинг 4.7. Добавление позиции

```
void add_position(int from_height, int to_height, int max_height,
                 positions new_positions, int *num_new_positions,
                 int itching[], board min_moves) {
    if (to_height <= max_height && itching[to_height] == 0 &&
        min_moves[to_height] == -1) {
        min_moves[to_height] = 1 + min_moves[from_height];
        new_positions[*num_new_positions] = to_height;
        (*num_new_positions)++;
    }
}
```

Боб хочет переместиться из `from_height` в `to_height`. Это действие допустимо, если оно пройдет три проверки. Во-первых, Боб не может подпрыгнуть выше максимально допустимой высоты. Во-вторых, он не может заканчивать прыжок на участке с чесоточным порошком. В-третьих, нужно, чтобы массив `min_moves` не содержал записи количества действий, необходимых для достижения `to_height`. Если же значение уже сохранено, значит, есть более быстрый способ попасть в позицию `to_height`. Прохождение всех трех проверок означает, что найдена новая

допустимая позиция, и мы определяем количество действий, необходимых для ее достижения, после чего сохраняем позицию для следующего этапа BFS.

### Поиск оптимальной высоты

Итоговая позиция Боба заранее не известна. Это может быть целевая высота  $h$  из условия тестового примера. Однако в зависимости от  $j$  и наличия порошка она может оказаться выше, чем  $h$ . Нам известно минимальное количество действий, необходимых, чтобы попасть в каждую из возможных позиций. Теперь нужно проверить всех претендентов на звание лучшей позиции, выбрав ту, которая минимизирует количество действий. Соответствующий код приведен в листинге 4.8.

#### Листинг 4.8. Минимальное количество действий

```
void solve(int target_height, board min_moves) {
    int best = -1; ❶
    int i;
    for (i = target_height; i < target_height * 2; i++)
        if (min_moves[i] != -1 && (best == -1 || min_moves[i] < best)) ❷
            best = min_moves[i];
    printf("%d\n", best);
}
```

Есть вероятность, что Боб не сможет добраться до заданной целевой высоты, поэтому сначала для переменной `best` устанавливается значение `-1` ❶. После для каждого варианта высоты выполняется проверка, может ли Боб туда попасть. Если может, и с меньшим количеством действий, чем текущее значение `best` ❷, то `best` соответствующим образом обновляется.

Теперь у нас есть весь код для обработки тестового примера и вывода его результата. Осталось лишь считать входные данные с помощью функции `main` из листинга 4.9.

#### Листинг 4.9. Функция `main`

```
int main(void) {
    int target_height, jump_distance, num_itching_sections;
    static int itching[SIZE * 2] = {0};
    static board min_moves;
    int i, j, itch_start, itch_end;
    scanf("%d%d", &target_height, &jump_distance, &num_itching_sections);
    for (i = 0; i < num_itching_sections; i++) {
        scanf("%d%d", &itch_start, &itch_end);
        for (j = itch_start; j <= itch_end; j++) ❶
            itching[j] = 1; ❷
    }
    find_distances(target_height, jump_distance, itching, min_moves);
    solve(target_height, min_moves);
    return 0;
}
```

Как это принято для больших массивов, `itching` и `min_moves` статичны. Элементы `itching` инициализируются как 0, что означает отсутствие на канате чесоточного порошка. Для каждого участка каната, на который этот порошок нанесен, выполняется цикл, который перебирает все целые числа в диапазоне ❶ и устанавливает соответствующий элемент `itching` равным 1 ❷. После заполнения массива каждый его элемент будет сообщать, нанесен ли порошок в соответствующем месте каната.

Вот и все. Мы получили решение, использующее один вызов BFS. Время отправлять его на проверку. Некоторые бы сейчас сказали: «Дело в шляпе», но увы... в ответ вернется ошибка «Time-Limit Exceeded».

## Решение 2. Модификация

Рассмотрим тестовые примеры большего размера, чтобы оценить время выполнения. Для упрощения чесоточный порошок мы учитывать не будем. Вот первый пример:

```
30000 5 0
```

Здесь 30 000 означает требуемую высоту, а 5 — высоту прыжка Боба. На моем ноутбуке выполнение этого тестового примера занимает 8 секунд. Удвоим нужную высоту:

```
60000 5 0
```

Теперь поиск происходит около 30 секунд, то есть почти в четыре раза дольше, чем в предыдущем случае. При этом установленный лимит в 4 секунды существенно превышен. Но давайте сделаем еще одну попытку, снова удвоив требуемую высоту:

```
120000 5 0
```

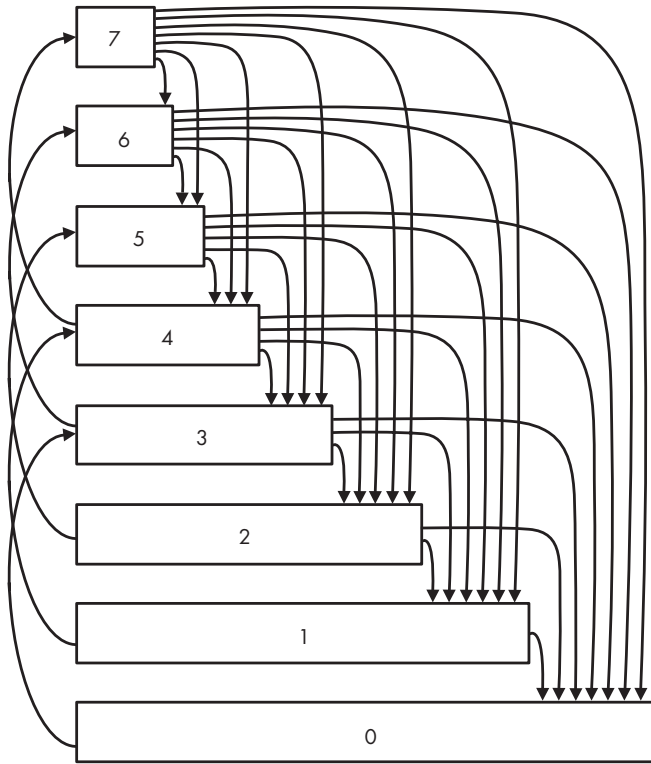
Процесс снова замедлился и на решение ушло около 130 секунд, что еще в четыре раза больше, чем в предыдущем случае. Можно сделать вывод, что удваивание требуемой высоты ведет к увеличению времени выполнения примерно в четыре раза. Это не настолько плохо, как в разделе «Решение 2. Мемоизация» главы 3, но однозначно слишком медленно.

### Слишком много ребер

В разделе «BFS в графах» на с. 182 я предупреждал, что при использовании поиска в ширину нужно оценивать два момента: количество вызовов BFS и число ребер в графе. В отношении первого параметра мы добились наилучшего результата, так

как совершаем всего один вызов BFS. Значит, для улучшения решения нам нужно найти способ уменьшить количество ребер в графе.

Рассмотрим граф для небольшого примера, показанный на рис. 4.5. Разобравшись с ним, можно будет экстраполировать результаты и на большие примеры, чтобы понять корень проблемы.



**Рис. 4.5.** Граф действий Боба

Приведенный граф показывает возможные действия до высоты 7 метров при условии, что Боб может подпрыгивать на три метра. Это пример направленного графа. Заметьте, к примеру, что здесь есть спуск с 6 к 5, но нет подъема с 5 к 6.

Граф содержит ребра подъемов, которые кодируют возможные прыжки Боба, и ребра спусков, кодирующие возможные спуски. Ребра подъемов идут снизу вверх, а ребра спусков сверху вниз. К примеру, ребро с 0 к высоте 3 является ребром подъема, а вышеупомянутое ребро от 6 к 5 — ребром спуска.

Количество ребер подъемов нас не беспокоит. Для каждой вершины возможно не более одного прыжка. Если всего будет  $n$  вершин, тогда и ребер подъемов будет не более  $n$ . Если мы решим смоделировать граф для высоты 8, а не 7, тогда добавим всего одно ребро подъема.

А вот ребра спуска множатся куда быстрее. Обратите внимание, что с высоты 1 есть одно ребро спуска, с высоты 2 — два, с высоты 3 — три и т. д. Это значит, что для каната высотой  $h$  у нас будет  $1 + 2 + 3 + \dots + h$  ребер спуска. Если мы хотим узнать, сколько таких ребер есть для заданной высоты, то можно сложить все целые числа от одного до значения этой высоты. Тем не менее есть удобная формула, которая даст ответ значительно быстрее:  $h(h + 1)/2$ . К примеру, для высоты 50 получится  $50 \times 51/2 = 1275$  ребер спуска. Если же высота составит два миллиона, то придется пройти два триллиона таких ребер.

Очень похожую формулу мы встретили в разделе «Выявление проблемы» главы 1, когда подсчитывали количество пар снежинок. Обе эти формулы квадратичны, то есть соответствуют показателю скорости  $O(h^2)$ , и именно этот квадратичный рост числа ребер спуска значительно замедляет наш алгоритм.

## Изменение графа

Если мы хотим уменьшить число ребер, то придется изменить действия, которые граф кодирует. Мы не можем изменить фактические правила игры, но *можем* модифицировать действия в графовой модели этой игры. Конечно же, граф можно изменить при условии, что новый вариант BFS выдаст верный результат.

В данной ситуации привлекательным выглядит вариант отобразить доступные действия в графе одно к одному. Так мы поступили в задаче с конем и пешкой и в итоге добились успеха. Но несмотря на привлекательность такого решения, оно не единственно возможное. Можно использовать граф, в котором будет меньшее количество вершин или ребер, при условии, что он будет давать верное решение начальной задачи.

Предположим, что хотим спуститься на некоторое расстояние с высоты пять метров. Один из вариантов — спуститься на четыре метра. Действительно, если рассматривать метод решения 1, то получим ребро, ведущее с высоты 5 к высоте 1. С другой стороны, этот спуск можно рассмотреть как четыре спуска по одному метру. Таким образом, Боб сначала спустится с 5 к 4, затем с 4 к 3, затем к 2 и, в конечном итоге, к 1. Мы получаем вариант, когда каждое ребро спуска равно одному метру. Больше нет длинных ребер, которые были при переходе от 5 к 3, от 5 к 2, от 5 к 1 или от 5 к 0. Теперь из каждой вершины будет по одному ребру спуска, ведущему на один метр вниз. Это кардинально сократит их общее число!



Но здесь нужно быть внимательными. Нельзя позволить каждому такому метровому спуску засчитываться за действие. Если Боб спускается на четыре метра, используя четыре метровых ребра, то нам нужно считать это за одно действие, а не за четыре.

Представьте, что у нас есть два каната: 0 и 1. Канат 0 у нас был изначально. Алиса его испортила, и теперь местами на нем может находиться чесоточный порошок. А канат 1 — новый, его изобрели мы сами в целях моделирования, поэтому на нем нет чесоточного порошка.

Кроме того, когда Боб находится на канате 1, ему нельзя двигаться вверх. Суть этой схемы состоит в том, что, когда Боб захочет спуститься, он будет перемещаться с каната 0 на канат 1, а после снова возвращаться на канат 0.

Говоря конкретнее, мы рассматриваем следующие ситуации:

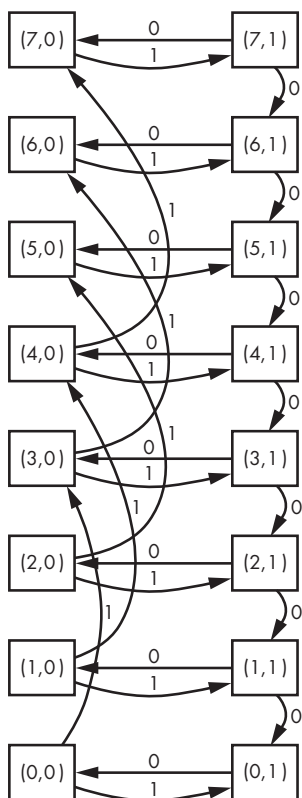
- Когда Боб находится на канате 0, у него есть два варианта действий: подпрыгнуть на  $j$  метров или перейти на канат 1. Любой вариант засчитывается как одно действие.
- Когда Боб находится на канате 1, у него также есть два варианта действий: спуститься на 1 метр или перейти на канат 0. Но за действия эти перемещения не считаются.

Боб, как и прежде, подпрыгивает, используя канат 0. Когда же он хочет спуститься, то переходит на канат 1 (это стоит ему одного действия), спускается по канату 1 на нужную высоту (действия не требуются), а затем возвращается на канат 0, тоже не затрачивая действия. Общий спуск в таком случае обходится Бобу всего в одно действие. Результат — тот же, что раньше! Никто и не узнает, что вместо одного мы используем два каната.

Сравните рис. 4.5 и его число ребер с рис. 4.6, который иллюстрирует прием с двумя канатами.

Все верно, мы удвоили количество вершин, но это нормально — для BFS важно не количество вершин, а число ребер. А с этим у нас полный порядок, так как из каждой вершины исходит не более двух ребер: на канате 0 находится ребро подъема и переход на канат 1; на канате 1 есть ребро спуска и переход на канат 0. Это значит, что для высоты  $h$  у нас будет всего около  $4h$  ребер, то есть мы имеем линейную зависимость и избежали сложных квадратичных процессов с  $h^2$ .

Выше я указал для каждого ребра, стоит оно действия (1) или нет (0). Это наш первый пример *взвешенного* графа, где каждое ребро имеет вес или стоимость.



**Рис. 4.6.** Граф действий Боба при использовании двух канатов

### Добавление позиций

Вот мы и вернулись опять к двумерной доске (привет, «Погоня за пешкой»). Одно измерение нужно для указания высоты позиции Боба, а второе — для каната, на котором он находится. Стандартный термин для второго измерения — *состояние*. Когда Боб находится на канате 0, мы будем говорить, что он в состоянии 0. Если же он находится на канате 1, значит, и состояние его тоже 1. Далее для удобства мы вместо слова «канат» будем использовать термин «состояние».

Вот новые определения типов:

```
typedef struct position {
    int height, state;
} position;

typedef int board[SIZE * 2][2];
typedef position positions[SIZE * 4];
```

Вместо того чтобы начинать с `find_distance`, как делалось выше в этой главе, мы начнем с функций `add_position`. Именно с «функций» во множественном числе, потому что я буду кодировать каждый тип действия в отдельную функцию. Здесь у нас четыре типа действия: прыжок вверх, спуск, перемещение из состояния 0 в состояние 1 и возвращение из состояния 1 в состояние 0. Следовательно, потребуются четыре функции `add_position`.

### Прыжок вверх

В листинге 4.10 приведен код для перехода по ребру подъема.

#### Листинг 4.10. Добавление позиции: прыжок вверх

```
void add_position_up(int from_height, int to_height, int max_height,
                    positions pos, int *num_pos,
                    int itching[], board min_moves) {
    int distance = 1 + min_moves[from_height][0]; ❶
    if (to_height <= max_height && itching[to_height] == 0 &&
        (min_moves[to_height][0] == -1 || ❷
         min_moves[to_height][0] > distance)) {
        min_moves[to_height][0] = distance;
        pos[*num_pos] = (position){to_height, 0};
        (*num_pos)++;
    }
}
```

В этой функции происходит подпрыгивание с `from_height` до `to_height`. Такое действие допустимо только в состоянии 0. Следовательно, при обращении к любому индексу `min_moves` мы будем использовать в качестве второго индекса 0.

Этот код аналогичен листингу 4.7, но с некоторыми важными корректировками.

Во-первых, я изменил имя `new_positions` на `pos`, а `num_new_positions` на `num_pos`. Причину перехода к более обобщенным именам параметров мы разберем после того, как пройдемся по всем четырем функциям.

Во-вторых, чтобы упростить сравнение между этими четырьмя функциями, я добавил переменную `distance` ❶. Она указывает количество действий, необходимых для достижения `to_height`, если начинать с `from_height`. При этом к значению переменной добавляется 1, так как на прыжок расходуется одно действие.

И наконец, я изменил часть условия `if`, которая проверяет, была ли найдена новая позиция ❷. Причина в том, что позиция может быть найдена при переходе по ребру, который засчитывается за действие. Нам нужно предусмотреть возможность, когда минимальное количество действий будет улучшено и обновлено благодаря ребру, не требующему действий (прыжок — не бесплатное действие, поэтому здесь

это изменение не требуется, но я оставил его для согласованности между всеми четырьмя функциями).

## Спуск

Теперь перейдем к листингу 4.11, где прописан код для спуска.

### Листинг 4.11. Добавление позиции: спуск

```
void add_position_down(int from_height, int to_height,
                      positions pos, int *num_pos,
                      board min_moves) {
    int distance = min_moves[from_height][1]; ❶
    if (to_height >= 0 &&
        (min_moves[to_height][1] == -1 ||
         min_moves[to_height][1] > distance)) {
        min_moves[to_height][1] = distance;
        pos[*num_pos] = (position){to_height, 1};
        (*num_pos)++;
    }
}
```

Спуск возможен только в состоянии 1. Поэтому при каждом обращении к `min_moves` второй индекс равен 1. Чесоточный порошок здесь не актуален, поэтому Боб может спускаться на любое расстояние, не беспокоясь о нем. В завершение отмечу важный момент: к вычисленному расстоянию не прибавляется 1 ❶. Напомню: за действие этот переход не считается.

## Смена состояний

Еще две функции необходимы для перехода Боба из состояния 0 в 1 (листинг 4.12) и обратно — из состояния 1 в 0 (листинг 4.13).

### Листинг 4.12. Добавление позиции: переход из состояния 0 в 1

```
void add_position_01(int from_height,
                    positions pos, int *num_pos,
                    board min_moves) {
    int distance = 1 + min_moves[from_height][0];
    if (min_moves[from_height][1] == -1 ||
        min_moves[from_height][1] > distance) {
        min_moves[from_height][1] = distance;
        pos[*num_pos] = (position){from_height, 1};
        (*num_pos)++;
    }
}
```

**Листинг 4.13.** Добавление позиции: переход из состояния 1 в 0

```
void add_position_10(int from_height,
                    positions pos, int *num_pos,
                    int itching[], board min_moves) {
    int distance = min_moves[from_height][1];
    if (itching[from_height] == 0 &&
        (min_moves[from_height][0] == -1 ||
         min_moves[from_height][0] > distance)) {
        min_moves[from_height][0] = distance;
        pos[*num_pos] = (position){from_height, 0};
        (*num_pos)++;
    }
}
```

Переход из состояния 0 в 1 требует одного действия, а обратный переход действия не требует. Заметьте, что перемещаться из состояния 1 в 0 можно, только если на этой высоте нет чесоточного порошка. Отсутствие этой проверки позволило бы останавливать спуск на участке с порошком, что является нарушением правил.

## 0-1 BFS

Пришло время подставить это состояние в код `find_distance` листинга 4.6. Однако следует быть внимательными, чтобы не ошибиться в подсчете ходов.

Вот пример. Я буду использовать  $(h, s)$  для описания Боба, находящегося на высоте  $h$  в состоянии  $s$ . Предположим, что Боб может подпрыгивать на три метра. Начинает он с позиции  $(0,0)$ , для попадания в которую ему нужно ноль действий. Начав исследование вариантов из  $(0,0)$ , мы определим в качестве новой позиции  $(0,1)$  и запишем, что попасть в нее можно за одно действие. Она будет добавлена к позициям для следующего шага BFS. Мы также найдем  $(3,0)$  и снова запишем, что для ее достижения нужно одно дополнительное действие. Это еще одна позиция для очередного шага BFS. Все это вычисляется по стандартному алгоритму BFS.

Продолжая поиск из  $(3,0)$ , мы найдем новые позиции,  $(3,1)$  и  $(6,0)$ . Обе будут добавлены в следующий шаг BFS, и обе окажутся достижимы минимум в два действия.

Однако позиция  $(3,1)$  может вызвать сбой при стандартном BFS. Из нее достижима  $(2,1)$ , но нам лучше не использовать эту позицию для следующего шага BFS. Согласно логике алгоритма,  $(2,1)$  находится на одно действие дальше, чем  $(3,1)$ , однако это не так. Обе позиции удалены от  $(0,0)$  на одинаковое количество действий, так как спуск в состоянии 1 действий не требует. Это значит, что  $(2,1)$  не добавляется в следующий шаг BFS, а идет в *текущий* вместе с  $(3,1)$  и остальными позициями, которые достижимы за два действия.

Таким образом, при каждом перемещении по ребру, которое требует действия, мы добавляем новую позицию в следующий шаг BFS (мы так делали всегда). Однако когда мы переходим по ребру, не затрачивая действия, то добавляем соответствующую позицию в текущий шаг BFS, чтобы обработать ее вместе с другими позициями, находящимися на том же расстоянии. Именно поэтому на с. 194 раздела «Добавление позиций» я отказался от использования `new_positions` и `num_new_positions` в `add_position`. Две из четырех функций действительно будут добавлять действия к новым позициям, но остальные две будут добавлять их к текущим.

Этот вариант алгоритма называется *0-1 BFS*, потому что он работает в графах, переход по некоторым ребрам которых не требует действий.

В завершение рассмотрим код BFS, приведенный в листинге 4.14.


**Листинг 4.14.** Минимальное количество действий Боба при использовании 0-1 BFS

```
void find_distances(int target_height, int jump_distance,
                  int itching[], board min_moves) {
    static positions cur_positions, new_positions;
    int num_cur_positions, num_new_positions;
    int i, j, from_height, from_state;
    for (i = 0; i < target_height * 2; i++)
        for (j = 0; j < 2; j++)
            min_moves[i][j] = -1;
    min_moves[0][0] = 0;
    cur_positions[0] = (position){0, 0};
    num_cur_positions = 1;

    while (num_cur_positions > 0) {
        num_new_positions = 0;
        for (i = 0; i < num_cur_positions; i++) {
            from_height = cur_positions[i].height;
            from_state = cur_positions[i].state;

            if (from_state == 0) { ❶
                add_position_up(from_height, from_height + jump_distance,
                               target_height * 2 - 1,
                               new_positions, &num_new_positions,
                               itching, min_moves);
                add_position_01(from_height, new_positions, &num_new_positions,
                               min_moves);
            } else {
                add_position_down(from_height, from_height - 1,
                                 cur_positions, &num_cur_positions, min_moves);
                add_position_10(from_height,
                               cur_positions, &num_cur_positions,
                               itching, min_moves);
            }
        }
    }
}
```

```
    }  
  }  
  num_cur_positions = num_new_positions;  
  for (i = 0; i < num_cur_positions; i++)  
    cur_positions[i] = new_positions[i];  
}  
}
```

Новый код проверяет, находится текущая позиция в состоянии 0 или 1 . В каждом случае рассматриваются два варианта. В состоянии 0 задействуются новые позиции (для очередного шага BFS); в состоянии 1 используются текущие позиции.

В остальном код практически идентичен коду из решения 1 — нужно лишь убедиться в правильности ссылок на состояние 0. Отправив это решение на проверку, вы увидите, что оно успешно пройдет все тесты и с приличным запасом уложится во временной лимит.

## Задача 3. Перевод книги

В задачах «Погоня за пешкой» и «Лазание по канату» входные данные не были представлены в виде графа, поэтому BFS пошагово формировал граф в ходе работы. А в нашей следующей задаче граф задан изначально.

Рассмотрим задачу с платформы DMOJ под номером `espa16d`.

### Условие

Вы написали книгу на английском и хотите перевести ее на  $n$  других языков. Для этого вы нашли  $m$  переводчиков. Каждый переводчик знает два языка и может выполнить перевод за установленную плату. Например, один из них знает испанский и бенгали и берет за услугу \$1800. Это означает, что ему можно поручить перевод книги с испанского на бенгали или с бенгали на испанский, заплатив \$1800.

Для получения версии книги на нужном языке может потребоваться серия из нескольких переводов. Например, если требуется перевести книгу с английского на бенгали, но переводчика в этой языковой паре нет, то можно вначале перевести ее с английского на испанский, а затем с испанского на бенгали.

Чтобы уменьшить число ошибок, вы должны минимизировать количество переводов, необходимое для получения версии на нужном языке. Если для достижения минимального числа переводов на целевой язык возможно несколько путей, то выбирать нужно самый дешевый. Цель заключается в минимизации числа переводов до целевого языка при наименьших затратах.

### Входные данные

Входные данные представлены одним тестовым примером, состоящим из следующих строк:

- Строки, содержащей два целых числа  $n$  и  $m$ , где  $n$  — количество целевых языков,  $m$  — число переводчиков. Всего возможно не более 100 целевых языков и не более 4500 переводчиков.
- Строки, содержащей  $n$  названий целевых языков. English в их число не входит.
- $m$  строк, каждая из которых содержит информацию об одном переводчике. Каждая из этих строк содержит три параметра, разделенных пробелами: первый язык, второй язык и положительное целочисленное значение стоимости перевода между ними. Для каждой языковой пары предлагается не более одного переводчика.

### Выходные данные

Требуется вывести минимальную стоимость перевода книги на все целевые языки, минимизировав при этом количество переводов для каждого языка. Если перевести книгу на все целевые языки невозможно, следует вывести `impossible`.

Время решения тестового примера ограничено одной секундой.

### Построение графа

Начнем с построения графа на основе входных данных. Это упростит анализ возможных переводов с каждого языка.

Рассмотрим небольшой пример:

```
3 5
Spanish Bengali Italian
English Spanish 500
Spanish Bengali 1800
English Italian 1000
Spanish Italian 250
Bengali Italian 9000
```

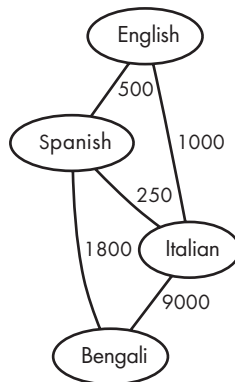
Сможете самостоятельно построить граф? Что следует выбрать вершинами, а что ребрами? Будет граф направленным или ненаправленным, взвешенным или невзвешенным?

Как обычно, ребра отражают допустимые действия. Здесь действие соответствует переводу с одного языка на другой. Вершины же представляют сами языки. Ребро,



проходящее от языка  $a$  к языку  $b$ , означает, что перевод возможен. Переводчик может перевести с  $a$  на  $b$  и наоборот, значит, граф ненаправленный. При этом он взвешенный, потому что каждое ребро (перевод) имеет вес (стоимость).

Граф приведен на рис. 4.7.



**Рис. 4.7.** Граф переводов

Стоимость перевода для пары английский–испанский составляет \$500, для пары английский–итальянский — \$1000 и для перевода с испанского на бенгали — \$1800. В общем получается \$3300. Не соблазняйтесь низкой стоимостью перевода с испанского на итальянский (\$250), поскольку его использование увеличит количество шагов перевода с английского на итальянский, а нам нужно их минимальное число, даже если в итоге это выйдет дороже. В действительности применение здесь BFS обусловлено тем, что нас в первую очередь интересует минимальное число шагов, а не минимальная стоимость услуги. Для минимизации стоимости мы будем использовать более мощные инструменты, с которыми познакомимся в главе 5.

Вместо прямого использования имен — «английский», «испанский» и т. д. — я буду ассоциировать их с целыми числами. Английский будет языком 0, а каждый целевой язык получит уникальное число больше 0. Это позволит работать с целыми числами, как мы делали в других задачах этой главы.

Для представления графа я буду использовать так называемый *список смежностей* (вершина  $b$  является смежной к вершине  $a$ , если их соединяет ребро, отсюда и происходит этот термин). Это обычный массив, в котором каждый индекс соответствует одной вершине и хранит связный список соединенных с этой вершиной ребер. Я использую связные списки ребер, а не их массивы, потому что изначально неизвестно количество ребер, относящихся к данной вершине.

Ниже приведены макросы и определения типов:

```
#define MAX_LANGS 101
#define WORD_LENGTH 16

typedef struct edge {
    int to_lang, cost;
    struct edge *next;
} edge;

typedef int board[MAX_LANGS];
typedef int positions[MAX_LANGS];
```

У `edge` есть параметры `to_lang` и `cost`, что вполне логично. Однако у него нет `from_lang`, который известен исходя из того, в каком индексе списка смежности находится это ребро.

В главе 2 при сохранении деревьев я использовал объект `struct node` вместо `struct edge`. Причина такого выбора заключалась в том, что именно узлы в той задаче являлись сущностями, связанными с информацией, в частности, числом конфет и количеством потомков. В текущей же задаче используется `struct edge`, потому что с информацией (стоимостью перевода) ассоциируются ребра, а не узлы.

Связный список проще всего добавить в начало. Один из побочных эффектов такого выбора состоит в том, что ребра для вершины будут занесены в список в порядке, обратном их считыванию. Например, если считать ребро, ведущее от вершины 1 к вершине 2, а затем — ребро от вершины 1 к вершине 3, то в связном списке ребро, идущее к вершине 3, окажется *перед* ребром, идущим к вершине 2. Учтите это при трассировке кода.

Теперь можно рассмотреть построение графа. Это происходит в функции `main`, приведенной в листинге 4.15.

#### Листинг 4.15. Функция `main` для построения графа

```
int main(void) {
    static edge *adj_list[MAX_LANGS] = {NULL};
    static char *lang_names[MAX_LANGS];
    int i, num_targets, num_translators, cost, from_index, to_index;
    char *from_lang, *to_lang;
    edge *e;
    static board min_costs;
    scanf("%d%d\n", &num_targets, &num_translators);
    lang_names[0] = "English"; ❶

    for (i = 1; i <= num_targets; i++)
        lang_names[i] = read_word(WORD_LENGTH); ❷

    for (i = 0; i < num_translators; i++) {
        from_lang = read_word(WORD_LENGTH);
```

```

to_lang = read_word(WORD_LENGTH);
scanf("%d\n", &cost);
from_index = find_lang(lang_names, from_lang);
to_index = find_lang(lang_names, to_lang);
e = malloc(sizeof(edge));
if (e == NULL) {
    fprintf(stderr, "malloc error\n");
    exit(1);
}
e->to_lang = to_index;
e->cost = cost;
e->next = adj_list[from_index];
adj_list[from_index] = e; ❸
e = malloc(sizeof(edge));
if (e == NULL) {
    fprintf(stderr, "malloc error\n");
    exit(1);
}
e->to_lang = from_index;
e->cost = cost;
e->next = adj_list[to_index];
adj_list[to_index] = e; ❹
}
find_distances(adj_list, num_targets + 1, min_costs);
solve(num_targets + 1, min_costs);
return 0;
}

```

Массив `lang_names` сопоставляет индексы с названиями языков. Как и планировалось, английскому (English) присваивается номер 0 ❶. Затем каждое последующее целое число (1, 2,...) сопоставляется с названиями языков по мере их считывания ❷. Для этого несколько раз используется вспомогательная функция `read_word`. Она похожа на представленную в листинге 1.14, за исключением того, что останавливается после считывания пробела или символа новой строки (см. листинг 4.16).

Напомню, что наш граф — ненаправленный, и, если добавлять ребро от  $a$  к  $b$ , то следует также добавить ребро от  $b$  к  $a$ . Поэтому для каждого переводчика мы добавляем в граф по два ребра: одно из `from_index` к `to_index` ❸, а второе из `to_index` к `from_index` ❹. Индексы `from_index` и `to_index` создаются функцией `find_lang`, которая ищет название языка (см. листинг 4.17).

В обращениях к вспомогательным функциям используется `num_targets + 1`, потому что `num_targets` дает количество целевых языков, а прибавление единицы позволяет учесть также английский язык.

#### Листинг 4.16. Чтение слова

```

/*на основе https://stackoverflow.com/questions/16870485 */
char *read_word(int size) {
    char *str;

```

```

int ch;
int len = 0;
str = malloc(size);
if (str == NULL) {
    fprintf(stderr, "malloc error\n");
    exit(1);
}
while ((ch = getchar()) != EOF && (ch != ' ') && (ch != '\n')) {
    str[len++] = ch;
    if (len == size) {
        size = size * 2;
        str = realloc(str, size);
        if (str == NULL) {
            fprintf(stderr, "realloc error\n");
            exit(1);
        }
    }
}
str[len] = '\0';
return str;
}

```

**Листинг 4.17.** Поиск языка

```

int find_lang(char *langs[], char *lang) {
    int i = 0;
    while (strcmp(langs[i], lang) != 0)
        i++;
    return i;
}

```

**BFS**

Код функции `add_position` в листинге 4.18 не содержит неожиданностей.

**Листинг 4.18.** Добавление позиции

```

void add_position(int from_lang, int to_lang,
                 positions new_positions, int *num_new_positions,
                 board min_moves) {
    if (min_moves[to_lang] == -1) {
        min_moves[to_lang] = 1 + min_moves[from_lang];
        new_positions[*num_new_positions] = to_lang;
        (*num_new_positions)++;
    }
}

```

Теперь можно переходить непосредственно к BFS в листинге 4.19.

**Листинг 4.19.** Вычисление минимальной стоимости перевода при помощи BFS

```

void find_distances(edge *adj_list[], int num_langs, board min_costs) {
    static board min_moves; ❶
    static positions cur_positions, new_positions;
}

```

```

int num_cur_positions, num_new_positions;
int i, from_lang, added_lang, best;
edge *e;
for (i = 0; i < num_langs; i++) {
    min_moves[i] = -1;
    min_costs[i] = -1;
}
min_moves[0] = 0;
cur_positions[0] = 0;
num_cur_positions = 1;
while (num_cur_positions > 0) {
    num_new_positions = 0;
    for (i = 0; i < num_cur_positions; i++) {
        from_lang = cur_positions[i];
        e = adj_list[from_lang]; ❷

        while (e) {
            add_position(from_lang, e->to_lang,
                          new_positions, &num_new_positions, min_moves);
            e = e->next;
        }
    }

    for (i = 0; i < num_new_positions; i++) { ❸
        added_lang = new_positions[i];
        e = adj_list[added_lang];
        best = -1;
        while (e) {
            if (min_moves[e->to_lang] + 1 == min_moves[added_lang] && ❹
                (best == -1 || e->cost < best))
                best = e->cost;
            e = e->next;
        }
        min_costs[added_lang] = best;
    }

    num_cur_positions = num_new_positions;
    for (i = 0; i < num_cur_positions; i++)
        cur_positions[i] = new_positions[i];
}
}

```

Для каждого языка мы сохраняем в `min_costs` ребро с минимальной стоимостью. Если вернуться к рис. 4.7, то мы бы сохранили 500 для испанского, 1000 для итальянского и 1800 для бенгали. В другой функции, которую рассмотрим чуть позже, мы будем складывать все эти числа для получения общей стоимости переводов.

Минимальное число действий требуется только для этой функции, а не внешней программы, поэтому мы объявляем его как локальную переменную ❶. Во внешнюю часть возвращается только `min_costs`.

Проверка каждого возможного действия приводит к обходу связного списка ребер для текущей вершины ❷. Это дает нам все значения `new_postions`. Теперь мы знаем все языки для следующего шага BFS, но стоимость добавления каждого из них пока еще не известна. Суть в том, что из `cur_positions` может исходить несколько ребер, достигающих одной и той же вершины в `new_positions`. Обратитесь еще раз к рис. 4.7. Для получения бенгали требуются два перевода, значит, он обнаруживается на втором шаге BFS, но нас интересует ребро, идущее от испанского, а не от итальянского.

Следовательно, необходим новый цикл `for` ❸, и с реализуемой им задачей мы в этой главе еще не сталкивались. Переменная `added_lang` отслеживает новые вершины для очередного шага BFS. Мы находим самое дешевое ребро, связывающее вершины `added_lang` и обнаруженные на текущем шаге BFS. Каждая из последних будет находиться на расстоянии на один меньше, чем `added_lang`, что объясняет первое условие в инструкции `if` ❹.

## Общая стоимость

После сохранения цен осталось только сложить их для получения общей стоимости перевода на все целевые языки. Соответствующий код приведен в листинге 4.20.

**Листинг 4.20.** Минимальная общая стоимость

```
void solve(int num_langs, board min_costs) {
    int i, total = 0;
    for (i = 1; i < num_langs; i++)
        if (min_costs[i] == -1) { ❶
            printf("Impossible\n");
            return;
        } else {
            total = total + min_costs[i];
        }
    printf("%d\n", total); ❷
}
```

Задача считается невыполнимой, если хотя бы один из целевых языков оказывается недоступен ❶. В противном случае выводится общая стоимость ❷.

Можно отправлять решение на проверку!

## Выводы

В этой главе мы написали множество страниц кода. Конечно же, я рассчитываю, что этот опыт позволит вам решать задачи с графами самостоятельно. В долгосрочной же перспективе хотелось бы, чтобы вы оценили важность проведения

моделирования на начальном этапе поиска решения. Рассмотрение задачи через призму BFS позволяет переходить от разнородных проблем шахмат, уроков физкультуры и поиска переводчиков в единую область графов. Если запрос в Google фразы «как лазать по канату» вряд ли выдаст что-то полезное, то по запросу «поиск в ширину» вы получите множество образцов кода, примеров и пояснений. Из комментариев, которые оставляют программисты на сайтах олимпиад, можно увидеть, что они общаются в терминах алгоритмов, а не частных условий задач. Зачастую для донесения смысла специалисты просто говорят «задача BFS». Читая эту книгу, вы учитесь языку моделирования и осваиваете путь от модели к рабочему коду. В следующей главе нас ждет еще одна встреча с графами, и мы увидим всю мощь их взвешенных видов.

## Примечания

Задача «Погоня за пешкой» взята из программы Канадской олимпиады по программированию 1999 года (1999 Canadian Computing Competition). Задача «Лазание по канату» взята из программы соревнования Вуберна 2018 года (2018 Woburn Challenge) (онлайн-раунд 1, старшая лига). Задача «Перевод книги» взята из программы соревнования по программированию восточной части центрального региона Северной Америки 2016 года, организованного ассоциацией ACM (2016 ACM East Central North America Regional Programming Contest).

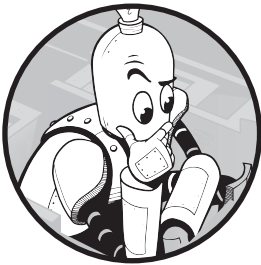
В этой главе мы изучили BFS, но если вы продолжите работать с графовыми алгоритмами, то рекомендую также освоить *поиск в глубину* (depth-first search, DFS). Лучше понять эти и другие графовые алгоритмы поможет книга *Algorithms Illuminated (Part 2): Graph Algorithms and Data Structures* Тима Рафгардена (Tim Roughgarden)<sup>1</sup>.

---

<sup>1</sup> Рафгарден Т. Совершенный алгоритм. Графовые алгоритмы и структуры данных. СПб.: Питер.

# 5

## Кратчайший путь во взвешенных графах



В этой главе мы обобщим то, что узнали о поиске кратчайших путей. В предыдущей главе требовалось определить минимальное число действий, необходимых для достижения цели. А что, если нас интересует не число действий, а минимальное время или расстояние? Допустим, что есть одно медленное действие, на которое уходит 10 минут, но при этом есть альтернатива из трех быстрых, в общей сложности занимающих 8 минут. В таком случае стоит предпочесть три быстрых действия, чтобы сэкономить время.

В этой главе мы изучим алгоритм Дейкстры для нахождения кратчайших путей во взвешенных графах. С его помощью мы определим количество мышей, которые могут сбежать из лабиринта в течение заданного времени, а также число кратчайших путей до дома бабушки. Пример с бабушкиным домом напомним и сделанный нами в главе 4 вывод, что должным образом измененные алгоритмы, такие как BFS или Дейкстры, способны на большее, нежели простое нахождение кратчайшего пути. Мы изучаем известные алгоритмы, чтобы получить гибкие инструменты решения задач. Что ж, приступим!

### Задача 1. Мышиный лабиринт

Рассмотрим задачу с платформы UV под номером 1112.



## Условие

Лабиринт состоит из ячеек и коридоров. Каждый коридор ведет из некоторой ячейки  $a$  в некоторую ячейку  $b$ , при этом на его прохождение уходит время  $t$ . К примеру, для попадания из ячейки 2 в ячейку 4 может потребоваться пять единиц времени. На обратный же переход из ячейки 4 в ячейку 2 может уйти 70 единиц времени, либо проход из ячейки 4 в 2 может отсутствовать — то есть коридоры  $a \rightarrow b$  и  $b \rightarrow a$  независимы. Одна из ячеек лабиринта обозначается как выход.

В каждой ячейке (включая выход) находится лабораторная мышь. Эти мыши обучены пробегать до выхода за минимальное время. Наша задача — определить количество мышей, которые могут достичь клетки выхода в течение заданного времени.

## Входные данные

Первая строка входных данных указывает количество тестовых примеров и сопровождается пустой строкой. Тестовые примеры так же отделены друг от друга пустой строкой. Каждый тестовый пример состоит из следующих строк:

- число  $n$  — количество ячеек в лабиринте. Ячейки пронумерованы от 1 до  $n$ , максимальное значение  $n$  — 100;
- число  $e$  — номер ячейки выхода в интервале от 1 до  $n$ ;
- число  $t$  — лимит времени, целое число больше или равное нулю;
- число  $m$  — количество коридоров в лабиринте;
- $m$  строк, каждая из которых описывает коридор лабиринта. Любая такая строка содержит три числа: номер первой ячейки  $a$  (между 1 и  $n$ ); номер второй ячейки  $b$  (между 1 и  $n$ ); время (больше или равное нулю), необходимое для прохода от  $a$  к  $b$ .

## Выходные данные

Для каждого тестового примера необходимо вывести количество мышей, достигших ячейки выхода за установленное время  $t$ . Вывод для каждого тестового примера должен отделяться от следующего пустой строкой.

Лимит времени на решение тестовых примеров — три секунды.

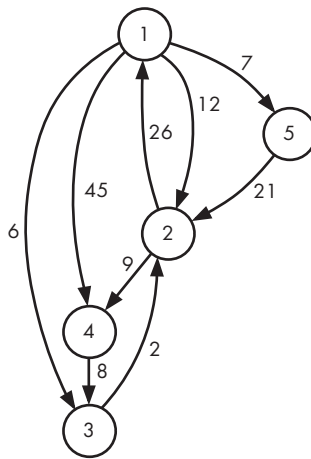
## BFS не подходит

У трех задач из главы 4 и текущей задачи про мышей есть ряд важных сходных черт. Можно смоделировать мышиный лабиринт в виде графа, где вершинами будут ячейки, а ребрами коридоры. Граф получится направленный (как и в задаче

«Лазание по канату»), потому что по коридору из ячейки  $a$  в ячейку  $b$  можно двигаться только в одном направлении.

В трех задачах главы 4 всю основную работу выполнил поиск в ширину. Его ключевая особенность заключается в нахождении кратчайших путей. В текущей задаче нам тоже пригодится знание самых коротких маршрутов, потому что они позволят определить, сколько времени требуется мыши для достижения выхода.

Наряду с этими сходствами есть важнейшее отличие: граф мышинного лабиринта *взвешенный*. У каждого ребра есть дополнительная характеристика — время, необходимое для прохода по нему. Пример приведен на рис. 5.1.



**Рис. 5.1.** Граф мышинного лабиринта

Предположим, что выход находится в ячейке 4. Какое время потребуется мыши в ячейке 1, чтобы попасть в ячейку 4? Напрямую из ячейки 1 в ячейку 4 пролегал одно ребро, поэтому если считать ребра (как в BFS), то ответом будет 1. Однако количество ребер нас сейчас не интересует: вместо этого нам нужен самый быстрый путь, то есть минимальная сумма весов ребер. Вес ребра  $1 \rightarrow 4$  равен 45. Это не быстрейший путь. Самый быстрый маршрут из ячейки 1 в ячейку 4 состоит из трех ребер: от ячейки 1 до ячейки 3 (шесть единиц времени); из ячейки 3 в ячейку 2 (две единицы времени); из ячейки 2 в ячейку 4 (девять единиц времени). Итого получается  $6 + 2 + 9 = 17$  единиц. Именно из-за того, что ориентируемся мы теперь на веса ребер, а не их количество, BFS нам не подходит и потребуются другой алгоритм.

Хотя подождите: в главе 4 были взвешенные графы, для которых мы применяли BFS. Как это получилось? Вернемся к рис. 4.6, изображающему граф подъема по

канату, в котором некоторые ребра имели вес 1, а другие вес 0. Мы смогли использовать BFS только благодаря ограниченности диапазона весов. А теперь взглянем еще раз на рис. 4.7 с графом перевода книги. Это полноценный взвешенный граф с произвольными весами ребер. Здесь мы тоже смогли использовать BFS, но лишь потому, что основная мера расстояния заключалась в количестве ребер. Только после определения расстояния в ребрах в игру вступали их веса, помогая найти максимально дешевый способ.

Для мышиного лабиринта количество ребер совершенно не важно. Путь из  $a$  в  $b$  может иметь сотню ребер, но при этом общее время прохода пять единиц. Другой путь из  $a$  в  $b$  может иметь только одно ребро, которое проходится за 80 единиц времени. BFS нашел бы второй путь, а нам нужен первый.

## Быстрейшие пути во взвешенных графах

BFS работает путем перебора вершин, постепенно удаляясь от стартовой вершины на все большее число ребер. Алгоритм, который я представляю в этом разделе, работает похожим образом: определяет быстрейший путь, также постепенно удаляясь от стартовой вершины, но учитывает вес ребер, а не их количество.

BFS работает пошагово, причем вершины, обнаруживаемые на следующем шаге, оказываются на одно ребро дальше, чем вершины на текущем шаге. Во взвешенных графах мы эту идею использовать не сможем, потому что последние найденные быстрейшие пути не обязательно помогут определить лучший путь до следующей вершины. Поэтому для нахождения очередного быстрейшего пути придется работать иначе.

Для наглядности используем схему на рис. 5.1 и найдем быстрейшие маршруты из вершины 1 до каждой вершины графа. Так мы выясним, сколько нужно времени мыши, чтобы добраться до ячейки выхода.

Для каждой вершины мы будем использовать два элемента данных:

- **done** — переменная, принимающая значения `true/false`. Если она равна `false`, значит, для данной вершины быстрейший путь еще не был найден. Если же она равна `true`, значит, путь уже известен. Если значение `done` для вершины установлено как `true`, значит, ее обработка закончена и найденное значение пути больше изменяться не будет.
- **min\_time** — продолжительность быстрейшего пути от стартовой вершины. По мере обработки все большего числа вершин `min_time` может сокращаться, так как может расти число вариантов достижения рассматриваемой вершины.

Кратчайший маршрут из вершины 1 до вершины 1 равен 0, поскольку идти никуда не нужно. Отсюда мы и начнем: со значения *min\_time* для вершины 1, равного 0, и отсутствия значений *min\_time* для других вершин:

Вершина	Обработана	min_time
1	false	0
2	false	
3	false	
4	false	
5	false	

Мы отмечаем вершину 1 как обработанную, после чего определяем *min\_time* для каждой другой вершины на основе весов ребер, ведущих к ней из вершины 1. Вот очередной вариант таблицы состояния:

Вершина	Обработана	min_time
1	true	0
2	false	12
3	false	6
4	false	45
5	false	7

Итак, вот утверждение, которое лежит в основе дальнейших действий: быстрее всего путь из вершины 1 к вершине 3 равен 6, и способа добиться лучшего результата нет. Я сделал утверждение относительно именно вершины 3, потому что она имеет наименьшее значение *min\_time* из всех необработанных вершин.

Утверждение, что ответ будет 6, может выглядеть на данном этапе несколько самонадеянно. Что если есть другой, более быстрый путь до вершины 3? Им может оказаться маршрут, идущий к вершине 3 через другие точки.

Сейчас я объясню, почему это невозможно и почему мое утверждение является верным. Представьте, что есть более быстрый путь *p* из вершины 1 до вершины 3. Этот путь должен начинаться в вершине 1 по некоторому ребру *e*. Затем он должен проходить через ноль или более других ребер, заканчиваясь в вершине 3. Смотрите: прохождение по *e* уже занимает не менее 6 временных единиц, потому что 6 — это

минимальное количество времени, необходимое для перехода из вершины 1 в другую вершину. Любые другие ребра на пути  $p$  только увеличат это значение, поэтому прохождение  $p$  никак не может занимать менее 6 единиц времени.

Итак, вершина 3 обработана: быстрееший путь до нее нам известен. Теперь нужно использовать эту вершину для проверки, можно ли улучшить какой-нибудь из путей до вершин, которые еще не обработаны. Вспомним, что в *min\_time* хранятся значения быстрееших путей до уже обработанных вершин. На достижение вершины 3 уходит 6 единиц времени, и есть ребро из вершины 3 до вершины 2, требующее 2 единицы времени, так что теперь у нас есть способ добраться из вершины 1 в вершину 2 всего за 8 единиц. Следовательно, мы обновляем значение *min\_time* для вершины 2: вместо 12 устанавливаем 8:

Вершина	Обработана	min_time
1	true	0
2	false	8
3	true	6
4	false	45
5	false	7

Вершины 2, 4 и 5 еще не рассмотрены. Какую из них следует обработать следующей? Вершину 5, так как она имеет минимальное *min\_time*. Можно ли использовать вершину 5 для обновления какого-либо другого пути? Из 5 есть исходящее ребро до вершины 2, но переход из 1 в 5 (7 единиц времени), а затем по ребру от 5 до 2 (21 единица) занимает больше времени ( $7 + 21 = 28$ ), чем уже известный нам путь из 1 в 2 (8 единиц). Поэтому значение *min\_time* вершины 2 остается прежним. Единственным изменением в следующем варианте таблицы будет определение вершины 5 как обработанной.

Вершина	Обработана	min_time
1	true	0
2	false	8
3	true	6
4	false	45
5	true	7

Остались еще две вершины. Значение *min\_time* вершины 2 равно 8, а вершины 4 — больше (45). Снова выбираем вершину с меньшим значением. Опять же, более быстрого пути из 1 в 2 быть не может. Любой маршрут *p* из вершины 1 до вершины 2 должен начинаться с обработанных вершин, и в определенный момент в него надо будет включить ребро из обработанной вершины в необработанную. Назовем это ребро  $x \rightarrow y$ , где *x* — это обработанная вершина, а *y* — необработанная. Таким образом, *p* приводит из вершины 1 в вершину *y*. Затем путь может продолжаться в некотором направлении, чтобы попасть из вершины *y* в вершину 2... но это все несерьезно. Переход из вершины 1 в вершину *y* уже занимает не менее 8 единиц времени, иначе вершина *y* была бы выбрана для обработки раньше вершины 2. Как бы путь *p* ни вел из вершины *y* в вершину 2, на это всегда уйдет больше времени. Так что *p* не может быть короче 8.

Добавление вершины 2 дает нам два ребра для проверки на существование более короткого пути: до вершин 1 и 4. Первое не представляет интереса, потому что вершина 1 уже обработана. А ребро со значением 9 единиц времени из 2 в 4 пригодится! Переход из вершины 1 в вершину 2 занимает 8 единиц, а из 2 в 4 — еще 9 единиц, итого получается 17. Это лучше, чем ранее известный путь от вершины 1 до вершины 4 длительностью 45 единиц. Теперь мы получили следующую таблицу:

Вершина	Обработана	min_time
1	true	0
2	true	8
3	true	6
4	false	17
5	true	7

Необработанной осталась только вершина 4. Для всех остальных мы нашли быстрые пути. Но это значит, что вершина 4 уже не поможет найти лучшие варианты, поэтому можно обозначить ее как обработанную и подвести итог:

Вершина	Обработана	min_time
1	true	0
2	true	8
3	true	6
4	true	17
5	true	7

В итоге мышь из ячейки 1 может добраться до выхода в ячейке 4 за 17 единиц времени. Этот расчет можно провести для каждой другой вершины, чтобы выяснить, сколько каждой мыши требуется времени для достижения выхода.

Такой метод решения называется алгоритмом Дейкстры в честь Эдсгера В. Дейкстры, одного из выдающихся ученых в области информатики. Если заданы стартовая вершина  $s$  и взвешенный граф, алгоритм Дейкстры позволяет вычислить наиболее эффективные пути из  $s$  до каждой вершины графа. Именно это нам и нужно для решения задачи с мышиным лабиринтом. Давайте перейдем к считыванию входных данных для построения графа, а затем посмотрим, как реализовать алгоритм.

## Построение графа

Используем полученный при решении предыдущих задач опыт использования деревьев и графов. Мы построим граф, как делали это для задачи «Перевод книги» в главе 4 (раздел «Построение графа»). Единственное отличие состоит в том, что там граф был ненаправленным, а здесь все графы, наоборот, будут направленными. Поскольку нам заданы номера вершин, не придется делать сопоставление между названиями языков и числами.

Представленный на рис. 5.1 граф соответствует следующим входным данным:

```
1
5
4
12 ①
9
1 2 12
1 3 6
2 1 26
1 4 45
1 5 7
3 2 2
2 4 9
4 3 8
5 2 21
```

Значение 12 ① устанавливает временное ограничение для мыши, в течение которого она должна добраться до выхода (можете убедиться, что мыши из ячеек 2, 3 и 4 могут достичь выхода в рамках данного лимита).

Как и при переводе книги, я буду использовать представление графа в виде списка смежности. С каждым ребром будет сохраняться номер ячейки, к которой оно ведет, время для его прохождения, а также указатель на следующий элемент.

Ниже представлен макрос и определение типов:

```
#define MAX_CELLS 100

typedef struct edge {
    int to_cell, length;
    struct edge *next;
} edge;
```

Входные данные считываются функцией `main` из листинга 5.1.

**Листинг 5.1. Функция `main` для построения графа**

```
int main(void) {
    static edge *adj_list[MAX_CELLS + 1];
    int num_cases, case_num, i;
    int num_cells, exit_cell, time_limit, num_edges;
    int from_cell, to_cell, length;
    int total, min_time;
    edge *e;

    scanf("%d", &num_cases);
    for (case_num = 1; case_num <= num_cases; case_num++) {
        scanf("%d%d%d", &num_cells, &exit_cell, &time_limit);
        scanf("%d", &num_edges);
        for (i = 1; i <= num_cells; i++) ❶
            adj_list[i] = NULL;
        for (i = 0; i < num_edges; i++) {
            scanf("%d%d%d", &from_cell, &to_cell, &length);
            e = malloc(sizeof(edge));
            if (e == NULL) {
                fprintf(stderr, "malloc error\n");
                exit(1);
            }
            e->to_cell = to_cell;
            e->length = length;
            e->next = adj_list[from_cell];
            adj_list[from_cell] = e; ❷
        }

        total = 0;
        for (i = 1; i <= num_cells; i++) {
            min_time = find_time(adj_list, num_cells, i, exit_cell); ❸
            if (min_time >= 0 && min_time <= time_limit) ❹
                total++;
        }
        printf("%d\n", total);
        if (case_num < num_cases)
            printf("\n");
    }
    return 0;
}
```



Согласно условиям задачи, за строкой с числом тестовых примеров следует пустая строка и каждую пару тестовых примеров также разделяет пустая строка. Однако при использовании `scanf` нам не нужно об этом беспокоиться: считывая число, `scanf` пропускает начальные пробелы (включая символы новой строки).

Первое, что мы делаем для каждого тестового примера — это очищаем список смежности, устанавливая значения для каждой ячейки равными `NULL` ❶. Если этого не сделать, то каждый новый тестовый пример будет включать информацию о ребрах из предыдущих примеров (это я знаю наверняка, так как сам совершил такой промах, заплатив за него тремя часами своей жизни). Так что перед каждым тестовым примером обязательно нужно делать очистку списка.

Перед инициализацией каждое ребро добавляется в связный список для `from_cell` ❷. В связный список для `to_cell` ничего не добавляется, потому что граф здесь — направленный.

Задача требует найти быстрые пути из каждой ячейки до ячейки выхода. Поэтому для каждой ячейки вызывается `find_time` ❸ — вспомогательная функция, реализующая алгоритм Дейкстры. Получая стартовую ячейку `i` и целевую ячейку `exit_cell`, он возвращает `-1`, если пути нет, либо время быстрого пути. Каждая ячейка, из которой можно достичь ячейки выхода за время, меньшее или равное `time_limit`, приводит к инкрементированию `total` на один. После определения быстрых путей из всех ячеек выводится полученное значение `total`.

## Реализация алгоритма Дейкстры

Пришло время реализовать алгоритм, следуя схеме, приведенной в разделе «Быстрые пути во взвешенных графах». Вот наша функция:

```
int find_time(edge *adj_list[], int num_cells,
              int from_cell, int exit_cell)
```

Ее четыре параметра определяют список смежности, количество ячеек, стартовую ячейку и ячейку выхода. Алгоритм Дейкстры будет вычислять наименьшее время пути из стартовой ячейки до всех других ячеек, включая выход. По завершении можно будет вернуть время быстрого пути до ячейки выхода. Может показаться чрезмерно затратным вычислять время до каждой ячейки только для того, чтобы в итоге все отбросить, оставив время до ячейки выхода. Здесь возможны различные оптимизации, к которым я перейду в следующем подразделе. Сейчас же давайте разберемся с базовым вариантом кода.

Тело алгоритма Дейкстры создается двумя вложенными циклами `for`. Внешний цикл `for` выполняется по разу для каждой ячейки; на каждой итерации изменяется значение `done` для одной ячейки и обновляются величины быстрых путей.

Внутренний цикл `for` находит ячейку, у которой значение `min_time` минимально среди всех необработанных ячеек. Код приведен в листинге 5.2.

**Листинг 5.2.** Вычисление быстрого пути до выхода при помощи алгоритма Дейкстры

```
int find_time(edge *adj_list[], int num_cells,
              int from_cell, int exit_cell) {
    static int done[MAX_CELLS + 1];
    static int min_times[MAX_CELLS + 1];
    int i, j, found;
    int min_time, min_time_index, old_time;
    edge *e;
    for (i = 1; i <= num_cells; i++) { ❶
        done[i] = 0;
        min_times[i] = -1;
    }
    min_times[from_cell] = 0; ❷

    for (i = 0; i < num_cells; i++) {
        min_time = -1;
        found = 0; ❸
        for (j = 1; j <= num_cells; j++) { ❹
            if (!done[j] && min_times[j] >= 0) { ❺
                if (min_time == -1 || min_times[j] < min_time) { ❻
                    min_time = min_times[j];
                    min_time_index = j;
                    found = 1;
                }
            }
        }
        if (!found) ❼
            break;
        done[min_time_index] = 1;

        e = adj_list[min_time_index];
        while (e) {
            old_time = min_times[e->to_cell];
            if (old_time == -1 || old_time > min_time + e->length) ❸
                min_times[e->to_cell] = min_time + e->length;
            e = e->next;
        }
    }
    return min_times[exit_cell]; ❹
}
```

В массиве `done` хранится информация об обработке ячеек: `0` означает «не обработана», а `1` — «обработана». Массив `min_times` предназначен для хранения продолжительностей быстрых путей из стартовой до других ячеек.

Цикл `for` ❶ используется для инициализации этих двух массивов: он устанавливает все значения `done` как `0` (ложные), а значения `min_times` как `-1` (не

найдено). Затем `min_times` устанавливается для ячейки `from_cell` равной 0 ❷, указывая, что протяженность кратчайшего пути из стартовой ячейки до самой себя равна нулю.

Переменная `found` помогает отслеживать, может ли новая ячейка быть найдена алгоритмом Дейкстры. При каждой итерации внешнего цикла `for` она начинается со значения 0 (ложно) ❸ и устанавливается на 1 (верно), если ячейка может быть найдена. Но тут возникает вопрос: как ячейка может не быть найдена? В разделе «Быстрейшие пути во взвешенных графах», например, мы нашли все ячейки. Однако бывают графы, где между стартовой ячейкой и некоторой другой пути *нет*. В таких графах будут ячейки, которые алгоритм Дейкстры не находит. Если новые ячейки не могут быть найдены, значит, пора остановиться.

Теперь мы переходим к внутреннему циклу `for` ❹. Его задача состоит в определении следующей обрабатываемой ячейки. Этот цикл оставит `min_time_index` с индексом этой ячейки, а `min_time` со значением пути. Подходящими для продолжения расчета ячейками считаются те, которые не обработаны и содержат значение `min_times` не меньше 0 (то есть не -1) ❺. Нас интересуют только необработанные ячейки, потому что для обработанных уже найдены окончательные значения быстрейших путей. Нам также нужно, чтобы значение `min_times` было не меньше 0: если оно будет -1, то эту ячейку пока найти нельзя и о кратчайшем пути до нее у нас нет данных. Если же находится ячейка с более быстрым путем, чем другие ❻, мы обновляем `min_time` и `min_time_index`, а также устанавливаем `found` на 1, обозначая успешное обнаружение ячейки.

Если новая ячейка не была найдена, происходит остановка ❼. В противном случае обнаруженная ячейка получает статус обработанной, и мы перебираем ее исходящие ребра в поиске более быстрых путей. Для каждого ребра `e` мы проверяем, обеспечивает ли оно более быстрый путь в ячейку `e -> to_cell` ❸. Этот путь рассчитывается как сумма `min_time` (времени, необходимого для перехода из `from_cell` в `min_time_index`) и времени, необходимого для преодоления ребра `e` (из `min_times_index` в `e->to_cell`).

Не следует ли нам при рассмотрении ребра `e` сначала убедиться, что `e->to_cell` еще не обработана, и лишь потом проверять, существует ли более быстрый путь ❸? Хотя эту проверку выполнить можно, пользы от нее не будет. Обработанные ячейки уже содержат окончательные значения быстрейших путей, так что здесь нет шансов найти новый, еще более быстрый путь.

После вычисления быстрейших путей до всех ячеек у нас однозначно будет вычислен самый быстрый путь до ячейки выхода. Осталось лишь вернуть его время ❾.

На этом все! Можете отправлять решение на проверку, его код должен успешно пройти все тесты.

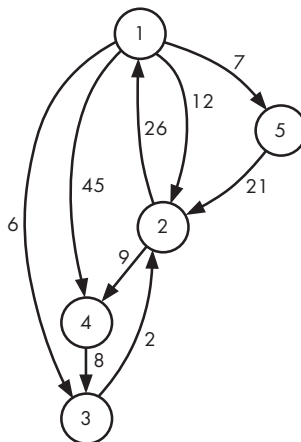
## Две оптимизации

Есть несколько приемов, с помощью которых можно ускорить наш алгоритм Дейкстры. Наиболее широко применяемая оптимизация, обеспечивающая существенное ускорение, достигается за счет структуры данных, называемой *кучей*. В текущей версии нашего кода слишком много ресурсов тратится на поиск очередной вершины для обработки, поскольку приходится сканировать все необработанные вершины. Куча преобразует этот медленный линейный поиск в быстрый поиск по дереву. Поскольку кучи используются во многих решениях помимо алгоритма Дейкстры, я рассмотрю их позже — в главе 7. Здесь же я предлагаю провести оптимизации, которые специфичны для задач типа мышиного лабиринта.

Во-первых, вспомним, что после обработки ячейки быстрейший путь до нее более не меняется. В таком случае, когда мы заканчиваем обработку ячейки выхода, смысл продолжать искать быстрейшие пути для других ячеек пропадает. Таким образом, мы можем раньше завершить выполнение алгоритма Дейкстры.

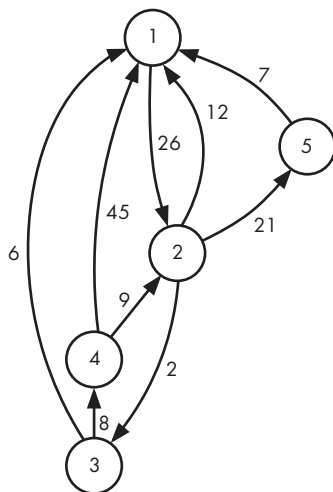
Во-вторых, для лабиринта из  $n$  ячеек мы вызываем алгоритм Дейкстры  $n$  раз, по разу для каждой ячейки. Для ячейки 1 мы вычисляем быстрейшие пути, а затем сохраняем только быстрейший путь до ячейки выхода. То же самое мы делаем для ячейки 2, ячейки 3 и т. д., отбрасывая все найденные пути за исключением тех, что включают ячейку выхода. Вместо этого рассмотрим выполнение алгоритма Дейкстры всего один раз, взяв ячейку выхода как стартовую. В таком случае алгоритм найдет кратчайший путь от выхода до ячейки 1, от ячейки выхода до ячейки 2 и т. д. Тем не менее это не совсем то, что нам нужно, потому что граф направленный: быстрейший путь от ячейки выхода до ячейки 1 *не обязательно* будет таковым при движении в обратном направлении.

Еще раз взглянем на рис. 5.1:



В разделе «Быстрые пути во взвешенных графах» мы узнали, что лучший путь из ячейки 1 до ячейки 4 равен 17, но кратчайший путь из ячейки 4 до ячейки 1 равен 36.

Быстрый маршрут из ячейки 1 до ячейки 4 задействует ребра  $1 \rightarrow 3$ ,  $3 \rightarrow 2$  и  $2 \rightarrow 4$ . Если мы собираемся запустить алгоритм из ячейки 4, тогда нужно, чтобы он нашел ребра  $4 \rightarrow 2$ ,  $2 \rightarrow 3$  и  $3 \rightarrow 1$ . Каждое из этих ребер является *обратным* ребру в исходном графе. На рис. 5.2 показан *обратный граф*.



**Рис. 5.2.** Обратный граф для мышиного лабиринта

Теперь можно запускать алгоритм — всего один вызов из ячейки 4, чтобы восстановить быстрые пути до всех вершин.

Создать обратный граф вместо оригинального целесообразно в функции `main` (листинг 5.1) при считывании графа.

Вместо

```
e->to_cell = to_cell;  
e->length = length;  
e->next = adj_list[from_cell];  
adj_list[from_cell] = e;
```

нам нужно использовать

```
e->to_cell = from_cell;  
e->length = length;  
e->next = adj_list[to_cell];  
adj_list[to_cell] = e;
```

Вот и все, теперь связывающие ребра ведут к `from_cell` и добавляются в связный список для `to_cell`. Если вы внесете это изменение и адаптируете код так, чтобы он вызывал алгоритм Дейкстры всего раз (из ячейки выхода), то получите куда более быструю программу. Попробуйте!

## Алгоритм Дейкстры

Алгоритм Дейкстры справляется там, где BFS помочь не в силах. BFS находит кратчайший путь в плане количества ребер в невзвешенном графе, а алгоритм Дейкстры находит быстрее пути в плане весов ребер во взвешенном.

Как и BFS, алгоритм Дейкстры получает стартовую вершину, а затем находит самые быстрые маршруты из нее до каждой другой вершины графа. Как и BFS, после этого он решает задачу *кратчайшего пути из одного источника*, только во взвешенных графах.

Честно говоря, алгоритм Дейкстры *может* находить быстрее пути и в невзвешенных графах. Просто возьмите такой граф и присвойте каждому ребру вес, равный единице. Тогда алгоритм Дейкстры найдет быстрее пути путем минимизации числа ребер, в точности как это делает BFS.

Так почему бы не решать каждую задачу о кратчайшем пути, будь то взвешенный или невзвешенный граф, с помощью алгоритма Дейкстры? И в самом деле, есть задачи, в которых сложно выбрать между BFS и алгоритмом Дейкстры. К примеру, я подозреваю, что для решения задачи с лазанием по канату из главы 4 многие предпочли бы использовать измененный алгоритм Дейкстры вместо BFS. Но если задача требует только минимизировать количество действий, поручать ее следует BFS, так как он, как правило, проще в реализации и выполняется намного быстрее. Хотя и алгоритм Дейкстры тоже нельзя назвать медленным.

## Время выполнения алгоритма Дейкстры

Опишем время выполнения алгоритма Дейкстры при решении графа, приведенного на рис. 5.2. Количество вершин в графе обозначим как  $n$ .

Цикл инициализации ❶ выполняется  $n$  раз, совершая постоянное число шагов в каждой итерации. Таким образом, его общая работа пропорциональна значению  $n$ . Следующая часть инициализации ❷ — это один шаг. Независимо, занимает инициализация  $n$  или  $n + 1$  шагов — это ничего не меняет, поэтому данный нюанс мы проигнорируем и скажем, что она занимает  $n$  шагов.

Далее начинается реальная работа алгоритма. Его внешний цикл `for` совершает до  $n$  итераций. В каждой итерации внутренний цикл `for` выполняет  $n$  итераций для

нахождения следующей вершины. В результате внутренний цикл `for` выполняется в общей сложности  $n^2$  раз. Каждая такая итерация связана с постоянным количеством работы, а значит, общая работа внутреннего цикла `for` пропорциональна  $n^2$ .

Также алгоритм Дейкстры выполняет перебор ребер каждой вершины. Если есть  $n$  вершин, значит, каждая из них определенно имеет не более  $n$  исходящих ребер. Следовательно, алгоритм для перебора ребер одной вершины предпринимает  $n$  шагов и прodelывает это для каждой из  $n$  вершин. Итого добавляется еще  $n^2$  шагов.

Если подытожить, то у нас есть  $n$  работы при инициализации,  $n^2$  работы во внутреннем цикле `for` и  $n^2$  работы при проверке ребер. Наибольшая экспонента здесь 2, следовательно, это алгоритм  $O(n^2)$ , иначе говоря, квадратичный.

В разделе «Выявление проблемы» главы 1 мы заменили алгоритм с квадратичным временем на линейный. С точки зрения скорости приведенная реализация алгоритма Дейкстры вроде бы не особенно впечатляет. Хотя все дело в том, что за  $n^2$  времени он решает не одну, а  $n$  задач, по одной для каждого быстреешего пути от стартовой вершины.

Я решил представить в этой книге алгоритм Дейкстры, но для нахождения быстреешего пути есть много других методов. Некоторые из них очень эффективно находят такой путь между любыми двумя вершинами графа. С их помощью решается задача *кратчайших путей для всех пар*. Один из них называется *алгоритмом Флойда — Уоршелла*, но выполняется он за  $O(n^3)$  времени. Интересно, что с помощью алгоритма Дейкстры тоже можно найти быстреешие пути для всех пар, причем с той же скоростью мы можем выполнить алгоритм Дейкстры  $n$  раз — по разу для каждой стартовой вершины. Это будет  $n$  вызовов алгоритма  $n^2$  и всего  $O(n^3)$  работы.

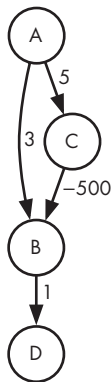
Неважно, взвешенный дан граф или невзвешенный, требуется минимальное расстояние от одного источника или между всеми парами — алгоритм Дейкстры справится с любой такой задачей. Вы спросите: «Он что — всемогущ?» Ответ отрицательный!

## Ребра с отрицательными весами

До текущего момента мы неявно делали допущение, что веса ребер неотрицательны. К примеру, в мышинном лабиринте веса обозначают время прохода по ребрам. Проход по ребру точно не может повернуть время вспять, так что отрицательных весов там быть не могло. Аналогичным образом во многих других графах ребра с отрицательными весами не встречаются, потому что не имеют смысла. Например, рассмотрим граф, где вершины — это города, а ребра — это стоимость перелетов между ними. Ни одна авиакомпания не согласится платить нам, чтобы мы летели ее рейсами, поэтому каждое ребро будет представлять неотрицательную стоимость перелета.

А теперь рассмотрим игру, в которой некоторые ходы дают очки, а другие очки отнимают. Второй вид ходов при представлении игры в виде графа соответствует

ребрам с отрицательными весами. А как на это отреагирует алгоритм Дейкстры? Выясним это с помощью образца графа на рис. 5.3.



**Рис. 5.3.** Граф с отрицательным ребром

Попробуем выяснить быстрее пути из вершины A. Как обычно, алгоритм Дейкстры начинается с присваивания значения 0 вершине A и установки статуса этой вершины как обработанной. Расстояние от A до B равно 3, от A до C — 5, а от A до D не определено (и осталось пустым):

Вершина	Обработана	min_distance
A	true	0
B	false	3
C	false	5
D	false	

Затем алгоритм Дейкстры определяет кратчайший путь до вершины B равным 3 и помечает вершину B как обработанную. Также он обновляет кратчайший путь до D:

Вершина	Обработана	min_distance
A	True	0
B	True	3
C	False	5
D	False	4

В связи с установкой статуса вершины B как обработанной мы утверждаем, что 3 является быстрее путем от A до B, но тогда возникает проблема, потому что 3



не является самым быстрым путем. Быстрейшим маршрутом будет  $A \rightarrow C \rightarrow B$  с общим весом в  $-495$ . Ради интереса давайте продолжим выполнение алгоритма и посмотрим, к чему это приведет. Следующей обработанной вершиной отмечается D:

Вершина	Обработана	min_distance
A	true	0
B	true	3
C	false	5
D	true	4

Кратчайший путь до D тоже определен неверно — он должен быть равен  $-494$ . Поскольку все вершины, кроме C, обработаны, то C уже ничего не изменит:

Вершина	Обработана	min_distance
A	true	0
B	true	3
C	true	5
D	true	4

Даже если позволить алгоритму Дейкстры изменить здесь кратчайший путь до B с 3 на  $-495$ , кратчайший путь до D так и останется ошибочным. Пришлось бы каким-то образом снова возвращаться к B, хотя она уже обработана, то есть как бы сказать: «Я знаю, что отметил B обработанной, но я передумал». В любом случае классический алгоритм Дейкстры, как я вам его представил, решает этот пример с ошибками.

Данный алгоритм не работает, если ребра графа могут иметь отрицательные веса. Для таких случаев следует использовать *алгоритм Беллмана — Форда* или уже упомянутый алгоритм Флойда — Уоршелла.

Ну а мы перейдем к очередной задаче, в которой об отрицательных весах ребер беспокоиться не придется. Здесь мы снова задействуем алгоритм Дейкстры или, точнее сказать, адаптируем его под решение новой задачи о быстрых путях...

## Задача 2. Дорога к бабушке

Иногда задача требует не только нахождения быстрого пути, но и дополнительной информации о нем. Хороший пример — задача с платформы DMOJ под номером `sac08p3`.

### Условие

Брюс планирует путешествие до дома своей бабушки. Есть  $n$  городов, пронумерованных от 1 до  $n$ . Брюс начинает путь в городе 1, а бабушка живет в городе  $n$ . Между каждой парой городов есть дорога определенной длины.

По пути к бабушке Брюсу надо купить коробку печенья. В некоторых городах есть кондитерские, и Брюсу необходимо заехать хотя бы в один такой город.

Во-первых, требуется определить минимальное расстояние, которое Брюсу нужно преодолеть, чтобы попасть от стартовой точки в дом бабушки, купив по пути печенье. Этот параметр не сообщает нам, сколько у Брюса есть вариантов попасть к старушке.

Возможно, есть всего один способ, а все остальные маршруты требуют преодоления большего расстояния. Также возможно, что есть несколько одинаково коротких путей. Во-вторых, задача требует определить количество маршрутов минимальной длины.

### Входные данные

Входные данные содержат один тестовый пример, состоящий из следующих строк:

- Целого числа  $n$ , указывающего количество городов. Города пронумерованы от 1 до  $n$ . Всего может быть от 2 до 700 городов.
- $n$  строк, каждая из которых содержит  $n$  чисел. Первая из этих строк сообщает протяженность дорог от города 1 до каждого города (города 1, города 2 и т. д.). Вторая строка указывает протяженность дорог от города 2 до каждого города, и т. д. Расстояния от городов до них самих равны нулю. Любое другое расстояние не меньше 1. Расстояние от города  $a$  до города  $b$  равно расстоянию от города  $b$  до города  $a$ .
- Целого числа  $m$ , отражающего количество городов с кондитерскими. Значение  $m$  не меньше единицы.
- Строки, содержащей  $m$  целых чисел — номеров городов с кондитерскими.

### Выходные данные

Нам необходимо вывести на одной строке минимальное расстояние от города 1 до города  $n$  (при условии попутной покупки печенья) и отделенное пробелом количество маршрутов с минимальной дистанцией по модулю 1 000 000.

Время на решение тестового примера — одна секунда.

## Матрица смежности

В данном случае граф будет представляться иначе, чем в задачах «Мышиный лабиринт» и «Перевод книги» из главы 4. В тех задачах каждое ребро было представлено двумя вершинами и весом. Например, так:

1 2 12

Эта запись означает, что есть ребро из вершины 1 в вершину 2 с весом 12.

В задаче же «Дорога к бабушке» граф представляется как *матрица смежности*, которая является двумерным массивом чисел — весов ребер.

Вот пример исходных данных:

```
4
0 3 8 2
3 0 2 1
8 2 0 5
2 1 5 0
1
2
```

Число 4 в первой строке сообщает количество городов. Последующие четыре строки представляют собой матрицу смежности. Рассмотрим их.

0 3 8 2

Первая строка содержит веса всех ребер, исходящих из города 1. Здесь мы видим, что ребро из города 1 до города 1 имеет вес 0, из города 1 до города 2 — вес 3, из города 1 до города 3 — вес 8, из города 1 до города 4 — вес 2.

Следующая строка сообщает ту же информацию для города 2:

3 0 2 1

Обратите внимание, что ребра есть между любой парой городов. Это значит, что речь идет о *полном графе*.

Эта матрица смежности избыточна. Например, число в строке 1 и столбце 3 указывает, что вес ребра из города 1 до города 3 равен 8. Однако поскольку в условии сказано, что дорога из города  $a$  до города  $b$  имеет ту же длину, что и дорога из города  $b$  до города  $a$ , мы снова видим эту 8 в строке 3 и столбце 1 (следовательно, наш граф — ненаправленный). Также матрица содержит диагональ нулей, которые указывают, что расстояние от каждого города до него самого равно нулю. Их мы будем просто игнорировать.

## Построение графа

В этой задаче нам потребуется дважды проявить находчивость. Во-первых, нужно проложить пути через город с кондитерской. Среди всех путей нас интересует кратчайший. Во-вторых, потребуется отследить не только кратчайший путь, но и количество способов его реализации. Я бы сказал, что будет вдвойне весело!

Начнем со считывания входных данных и построения графа. В данный момент у нас для этого все есть. Вооружившись графом, мы будем готовы к дальнейшим действиям.

Мы будем считывать матрицу, попутно создавая списки смежности. Отслеживать индексы городов придется самостоятельно, поскольку матрица смежности в явном виде их не содержит.

В принципе, есть возможность считать и использовать матрицу напрямую, минуя создание списков смежности. Каждая строка  $i$  сообщает расстояние до каждого города, значит, вместо перебора списка смежности можно просто перебрать строку  $i$  алгоритмом Дейкстры. Поскольку граф у нас полный, то не придется учитывать необходимость пропуска несуществующих ребер. Тем не менее здесь я все же задействую списки смежности для согласования решения с тем, что мы делали ранее.

Определим константы и структуру `edge`, которые будем использовать:

```
#define MAX_TOWNS 700

typedef struct edge {
    int to_town, length;
    struct edge *next;
} edge;
```

Код для считывания матрицы дан в листинге 5.3.

### Листинг 5.3. Функция `main` для построения графа

```
int main(void) {
    static edge *adj_list[MAX_TOWNS + 1] = {NULL};
    int i, num_towns, from_town, to_town, length;
    int num_stores, store_num;
    static int store[MAX_TOWNS + 1] = {0};
    edge *e;

    scanf("%d", &num_towns);
    for (from_town = 1; from_town <= num_towns; from_town++) ❶
        for (to_town = 1; to_town <= num_towns; to_town++) {
            scanf("%d", &length);
            if (from_town != to_town) { ❷
```

```

    e = malloc(sizeof(edge));
    if (e == NULL) {
        fprintf(stderr, "malloc error\n");
        exit(1);
    }
    e->to_town = to_town;
    e->length = length;
    e->next = adj_list[from_town];
    adj_list[from_town] = e;
}

scanf("%d", &num_stores); ❸
for (i = 1; i <= num_stores; i++) {
    scanf("%d", &store_num);
    store[store_num] = 1;
}
solve(adj_list, num_towns, store);
return 0;
}

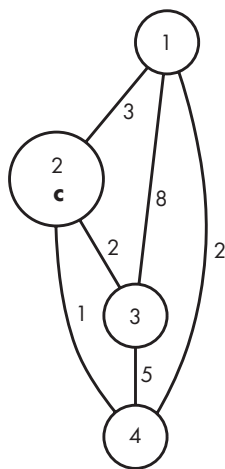
```

После сохранения количества городов используется двойной цикл `for` для считывания матрицы смежности. Каждая итерация внешнего цикла `for` ❶ отвечает за считывание одной строки для `from_town`. Внутренний цикл `for` считывает по одному значению `length` для каждого `to_town`. Теперь нам известно, где начинается и заканчивается ребро, а также его длина. Далее нужно добавить его в список, но только если оно не относится к ребрам с нулевым весом, ведущим от города к нему самому. Если ребро находится между разными городами ❷, то оно добавляется в список смежности для `from_town`. Поскольку граф ненаправленный, мы должны убедиться, что в итоге это ребро будет добавлено и в список смежности для `to_town`. Подобную проверку мы выполняли в явном виде в листинге 4.15, когда решали задачу про перевод книги. Но здесь ребро будет добавлено позже при обработке строки для `to_town`, поэтому от нас не требуется дополнительных действий. Например, если `from_town` равно 1, а `to_town` равно 2, то будет добавлено ребро  $1 \rightarrow 2$ . Позже, когда `from_town` будет равно 2, а `to_town` — 1, добавится уже ребро  $2 \rightarrow 1$ .

Осталось только считать информацию о том, в каких городах есть кондитерские, начиная с количества таких городов ❸. Для отслеживания городов с кондитерскими служит массив `store`, где `store[i]` равно 1 (верно), если в городе `i` есть кондитерская, и 0 (ложно), если кондитерской нет.

## Странные пути

Попробуем обработать тестовый пример, представленный в разделе «Матрица смежности». Соответствующий граф приведен на рис. 5.4, где `c` указывает город с кондитерской.

**Рис. 5.4.** Граф пути к бабушке

Брюс начинает в городе 1, а попасть ему нужно в город 4. Кондитерская есть только в городе 2. Каков будет путь с кратчайшим расстоянием? Хотя Брюс может проехать напрямую из города 1 в город 4 по ребру длиной 2, это не будет допустимым решением задачи. Нельзя забывать, что на кратчайшем пути обязательно должен присутствовать город с кондитерской. Для данного графа это означает, что мы должны включить в маршрут город 2 (в других тестовых примерах может быть множество городов с кондитерскими, и потребуется включить в маршрут хотя бы один из них).

Рассмотрим вариант маршрута из города 1 до города 4:  $1 \rightarrow 2$  (расстояние 3)  $\rightarrow 4$  (расстояние 1). Общее расстояние равно 4, и это действительно кратчайший путь из города 1 в город 4 через город 2.

Но этот путь — не единственный оптимальный. Есть еще один:  $1 \rightarrow 4$  (расстояние 2)  $\rightarrow 2$  (расстояние 1)  $\rightarrow 4$  (расстояние 1). Странность этого маршрута в том, что город бабушки (4) посещается *дважды*. Мы начинаем с поездки из города 1 в город 4, но здесь путь завершить нельзя, так как печенье еще не куплено. Тогда мы отправляемся из города 4 в город 2, чтобы зайти в кондитерскую. В завершение мы возвращаемся из города 2 в город 4, но на этот раз прибываем в него уже с печеньем, получая таким образом допустимый вариант пути.

Этот путь кажется циклическим, однако если взглянуть иначе, то цикла здесь нет. При первом посещении города 4 печенья у нас не было, при повторном же посещении оно есть. Следовательно, эти два визита не являются повторами: верно, что город 4 был посещен дважды, но верно и то, что состояние (отсутствие коробки печенья и ее наличие) было разным.

Можно сделать вывод, что один и тот же город нельзя посетить более двух раз. Если посетить его, например, три раза, то два из этих визитов будут подразумевать одинаковое состояние. Допустим, что при первых двух визитах состояние было одинаковым — без коробки печенья. Тогда это действительно будет циклом, который потребует прохождения дополнительного расстояния. Поэтому его удаление даст более короткий путь.

Получается, что недостаточно просто знать, в каком городе мы находимся. Нам также нужно знать, было ли куплено печенье. С подобной проблемой мы уже однажды справлялись, когда решали задачу про лазание по канату в главе 4. В разделе «Изменение действий» я рассуждал на тему добавления второго каната для более точного моделирования задачи. Здесь мы также воспользуемся этой идеей — добавим состояние, сообщающее о наличии или отсутствии коробки печенья. В состоянии 0 коробка отсутствует, а в состоянии 1 она есть. Тогда допустимым будет любой путь, приводящий к дому бабушки в состоянии 1. Прибытие к дому бабушки в состоянии 0 не может считаться завершением допустимого пути.

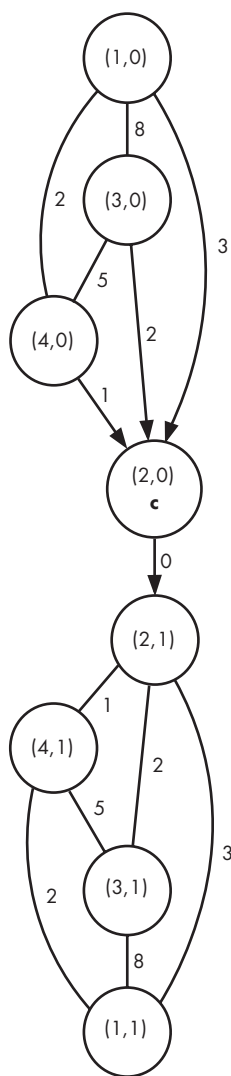
Взгляните на рис. 5.5, который дополняет рис. 5.4 отслеживанием состояния. Здесь **с** представляет город с кондитерской. Ребра без стрелок по-прежнему являются ненаправленными, но теперь здесь также есть и направленные ребра.

Для создания этого графа мы выполняем следующее:

- Добавляем четыре новые вершины городов, по одной для каждого исходного города на графе. Исходные вершины находятся в состоянии 0, а новые вершины — в состоянии 1.
- Сохраняем все исходные ребра за исключением тех, что исходят из города 2 (город с кондитерской). Если мы достигаем этого города в состоянии 0, то переходим в состояние 1. При этом исходящее из него ребро будет направленным к (2, 1) и имеющим нулевой вес, потому что на изменение состояния время не затрачивается. Алгоритму Дейкстры нельзя доверять работу только в графах с отрицательными ребрами (см. раздел «Ребра с отрицательными весами»), а ребра с нулевым весом допустимы.
- Соединяем вершины в состоянии 1, используя те же ребра, которые соединяли вершины в состоянии 0.

Когда, находясь в состоянии 0, мы достигаем города с кондитерской, то покупаем печенье и переходим в состояние 1. Когда мы оказываемся в состоянии 1, граф уже не позволяет вернуться в состояние 0, потому что потерять коробку печенья невозможно.

Итак, мы начинаем в городе 1 в состоянии 0 и должны прибыть в город 4, находясь в состоянии 1. Для этого нужно сначала перейти из состояния 0 в состояние 1,



**Рис. 5.5.** Граф пути к бабушке с состоянием

а затем добраться до города 4, используя ребра состояния 1. Когда в графе присутствуют несколько городов с кондитерскими, задача становится все более запутанной, потому что нужно выбирать, в каком из городов лучше всего перейти из состояния 0 в состояние 1. Что ж, это может запутывать нас, но не алгоритм Дейкстры, так как нам нужно лишь запросить кратчайший путь в графе.



### Подзадача 1: кратчайшие пути

До сих пор мы говорили о том, как моделировать задачу в виде графа и находить кратчайшие пути, но не о том, как находить *количество* таких путей. Эти две подзадачи я разберу по очереди. В конце этого подраздела у нас будет наполовину готовое решение, верно выводящее расстояние кратчайшего пути. Но количество путей мы вывести еще не сможем. Однако уже в следующем подразделе мы разберемся способ получения и этого результата. Время Дейкстры!

При использовании нашей новой модели (с состояниями 0 и 1) граф, считываемый из входных данных, больше не соответствует графу, который будет исследоваться алгоритмом Дейкстры. Одна из возможных идей состоит в создании представления списка смежности нового графа из списка смежности исходного. То есть надо начать с пустого графа, имеющего вдвое больше вершин, и добавлять необходимые ребра. Это выполнимо, но проще будет, не трогая граф, логически добавлять состояние в код алгоритма Дейкстры (при решении задачи с канатом в главе 4 перед нами не стояло такой проблемы, потому что входные данные не содержали графа).

Напишем следующую функцию:

```
void solve(edge *adj_list[], int num_towns, int store[])
```

Здесь `adj_list` является списком смежности, `num_towns` представляет количество городов (и номер города бабушки), а `store` сообщает, есть ли в городе `i` кондитерская.

Далее мы сделаем то же, что и в задаче с мышинным лабиринтом (см. листинг 5.2). Но при этом на каждом шаге будем учитывать влияние состояния на ход решения и вносить необходимые изменения. Пройдемся по коду, приведенному в листинге 5.4, попутно сравнивая его с листингом 5.2 для выявления сходства.

#### Листинг 5.4. Вычисление кратчайшего пути к бабушке с помощью алгоритма Дейкстры

```
void solve(edge *adj_list[], int num_towns, int store[]) {
    static int done[MAX_TOWNS + 1][2];
    static int min_distances[MAX_TOWNS + 1][2];
    int i, j, state, found;
    int min_distance, min_town_index, min_state_index, old_distance;
    edge *e;

    for (state = 0; state <= 1; state++) ❶
        for (i = 1; i <= num_towns; i++) {
            done[i][state] = 0;
            min_distances[i][state] = -1;
        }
    min_distances[1][0] = 0; ❷
```

```

for (i = 0; i < num_towns * 2; i++) { ❸
    min_distance = -1;
    found = 0;
    for (state = 0; state <= 1; state++)
        for (j = 1; j <= num_towns; j++) {
            if (!done[j][state] && min_distances[j][state] >= 0) {
                if (min_distance == -1 || min_distances[j][state] < min_distance) {
                    min_distance = min_distances[j][state];
                    min_town_index = j;
                    min_state_index = state;
                    found = 1;
                }
            }
        }
    if (!found)
        break;
    done[min_town_index][min_state_index] = 1; ❹

    if (min_state_index == 0 && store[min_town_index]) { ❺
        old_distance = min_distances[min_town_index][1];
        if (old_distance == -1 || old_distance > min_distance)
            min_distances[min_town_index][1] = min_distance;
    } else {
        e = adj_list[min_town_index]; ❻
        while (e) {
            old_distance = min_distances[e->to_town][min_state_index];
            if (old_distance == -1 || old_distance > min_distance + e->length)
                min_distances[e->to_town][min_state_index] = min_distance +
                                                                e->length;
            e = e->next;
        }
    }
}
printf("%d\n", min_distances[num_towns][1]); ❼
}

```

Влияние состояния на массивы заметно с самого начала, поскольку теперь `done` и `min_distances` стали двумерными. Первое измерение — номер города, а второе — состояние. Мы инициализируем элементы обоих состояний в соответствующем блоке ❶.

Стартовая точка — это город 1, состояние 0, значит, расстояние инициализируется равным 0 ❷.

Как всегда, нам нужно продолжить выполнение алгоритма до тех пор, пока он не перестанет обнаруживать новые вершины. `num_towns` городов существует и в состоянии 0, и в состоянии 1, поэтому максимальное число вершин равно `num_towns*2` ❸.

Вложенные циклы для `state` и `j` находят очередную вершину. По завершении этих циклов ④ определяются значения двух важных переменных: индекса города `min_town_index` и индекса состояния `min_state_index`.

Дальнейшие действия зависят от того, в каком состоянии мы находимся, и от того, есть ли в городе кондитерская. Если мы находимся в состоянии 0 и в городе есть кондитерская ⑤, то `adj_list` игнорируется и рассматривается только переход в состояние 1. Вспомните, что длина перехода из `[min_town_index][0]` в `[min_town_index][1]` равна 0, значит, путь к `[min_town_index][1]` будет иметь ту же длину, что и кратчайший путь до `[min_town_index][0]`. В характерном для алгоритма Дейкстры стиле мы обновляем значение кратчайшего пути, если новый путь оказывается короче.

Если мы находимся в состоянии 0, но не в городе с кондитерской или в состоянии 1, доступные ребра будут совпадать с исходящими ребрами текущего города в исходном графе, поэтому мы проверяем все ребра из `min_town_index` ⑥. Теперь мы оказываемся в ситуации, аналогичной мышиному лабиринту, выискивая быстрые пути с помощью ребра `e`. Главное, будьте внимательны и используйте одно и то же значение `min_state_index`, поскольку ни одно из этих ребер не изменяет состояние.

В заключение остается вывести протяженность кратчайшего пути ⑦. В качестве первого индекса используется `num_towns` (номер города бабушки), а в качестве второго индекса 1 (наличие коробки печенья).

Если проверить программу на тестовом примере, приведенном в разделе «Матрица смежности», то мы должны получить верный вывод — 4. И для любого другого тестового примера будет выводиться правильное значение кратчайшего пути. Теперь перейдем к количеству таких путей.

## Подзадача 2: количество кратчайших путей

В алгоритм достаточно внести всего несколько изменений, чтобы он смог находить не только расстояние кратчайшего пути, но также и количество таких путей. Изменения эти будут весьма тонкими, поэтому вначале разберем несколько шагов примера, чтобы вы поняли смысл и логику наших действий. Затем я приведу новый код, после чего представлю более подробное доказательство его корректности.

### Пример

Проследим работу алгоритма Дейкстры на рис. 5.5 от вершины (1, 0). Помимо отслеживания обработки вершин и определения минимального расстояния до каждой вершины мы также будем сохранять значение параметра `num_paths`, указывающего

число кратчайших путей до этой вершины. Значение *num\_paths* сбрасывается при каждом нахождении более короткого пути.

Вначале обрабатывается стартовая вершина (1, 0). Кратчайший путь устанавливается равным 0, а сама вершина помечается как обработанная. Поскольку есть всего один путь от стартовой вершины до нее самой (путь без ребер), количество ведущих до нее путей устанавливается равным 1. Ребра, идущие от стартовой вершины, используются для инициализации других вершин, и для каждой из них указывается наличие одного пути (пути от стартовой вершины):

Вершина	Обработана	min_distance	num_paths
(1,0)	true	0	1
(2,0)	false	3	1
(3,0)	false	8	1
(4,0)	false	2	1
(1,1)	false		
(2,1)	false		
(3,1)	false		
(4,1)	false		

Теперь, как это обычно бывает в алгоритме Дейкстры, сканируются необработанные вершины и выбирается одна с минимальным значением *min\_distance*. Это значит, что выбирается вершина (4, 0). Алгоритм Дейкстры гарантирует, что для этого пути установлен кратчайший путь, значит, его можно отметить как обработанный. Затем необходимо просмотреть ребра, исходящие из (4, 0), чтобы проверить, можно ли найти более короткие пути до других вершин. Мы и в самом деле можем найти более короткий путь к (3, 0): раньше его значение было 8, но теперь стало 7, потому что можно добраться до (4, 0) по ребру длиной 2, а затем из (4, 0) до (3, 0) по ребру длиной 5. Каково будет число кратчайших путей для (3, 0)? Что же, здесь было значение 1, значит, логичным кажется установить 2. Однако это будет ошибкой, потому что тогда мы учтем путь с расстоянием 8, но он уже не является кратчайшим. Правильный ответ — 1, так как сюда ведет всего один путь с расстоянием 7.

Из (4, 0) идет ребро к (2, 0), которое нельзя упускать из внимания. Ранее найденный кратчайший путь до (2, 0) равен 3. Обеспечит ли ребро от (4, 0) до (2, 0) более короткий путь? Расстояние до (4, 0) равно 2, а ребро из (4, 0) до (2, 0) имеет длину 1, значит, у нас есть новый способ добраться до (2, 0), преодолев расстояние 3.

Таким образом, переход в (4, 0) с последующим использованием ребра, ведущего в (2, 0), дает нам новый способ добраться до (2, 0). В сумме мы получаем  $1 + 1 = 2$  кратчайших пути до (2, 0):

Вершина	Обработана	min_distance	num_paths
(1,0)	true	0	1
(2,0)	false	3	2
(3,0)	false	7	1
(4,0)	true	2	1
(1,1)	false		
(2,1)	false		
(3,1)	false		
(4,1)	false		

Следующая вершина для обработки — (2, 0). Из (2, 0) к (2, 1) идет ребро с нулевым весом, а дистанция до (2, 0) равна 3, значит, кратчайший путь до (2, 1) также будет равен 3. Есть два способа попасть в (2, 0), преодолев это минимальное расстояние, значит, и до (2, 1) также есть два варианта пути. Вот что у нас получилось:

Вершина	Обработана	min_distance	num_paths
(1,0)	true	0	1
(2,0)	true	3	2
(3,0)	false	7	1
(4,0)	true	2	1
(1,1)	false		
(2,1)	false	3	2
(3,1)	false		
(4,1)	false		

Следующая вершина для обработки — (2, 1), вершина, позволяющая найти длину кратчайшего пути до нашей цели (4, 1). До (2, 1) есть два кратчайших пути, значит, и до (4, 1) также есть два кратчайших пути. Вершина (2, 1) также помогает найти новые быстрее пути до (1, 1) и (3, 1):

Вершина	Обработана	min_distance	num_paths
(1,0)	true	0	1
(2,0)	true	3	2
(3,0)	false	7	1
(4,0)	true	2	1
(1,1)	false	6	2
(2,1)	true	3	2
(3,1)	false	5	2
(4,1)	false	4	2

Вершина (4, 1) попадает в категорию обработанных, значит, ответ у нас есть: кратчайший путь равен 3, а количество таких путей равно 2 (наш код не будет останавливаться в этой точке, чтобы алгоритм Дейкстры продолжил работу, находя быстрее пути и их количество для других вершин. Рекомендую вам продолжить решение примера до конца).

Обобщить алгоритм можно двумя правилами:

- **Правило 1.** Предположим, что вершина  $u$  позволяет найти более короткий путь к узлу  $v$ . Тогда количество кратчайших путей до  $v$  будет равно числу кратчайших путей до  $u$  (все старые пути до  $v$  утрачивают свою актуальность и более не считаются, так как теперь нам известно, что они не являются кратчайшими).
- **Правило 2.** Предположим, что вершина  $u$  позволяет найти путь к узлу  $v$ , который имеет ту же длину, что и текущий кратчайший путь к  $v$ . Тогда количество путей к  $v$  будет суммой количества ранее найденных кратчайших путей до  $v$  и числа кратчайших путей до  $u$  (все старые пути к  $v$  по-прежнему учитываются).

Предположим, что мы выбрали некую вершину  $n$  и по мере выполнения алгоритма наблюдаем, как меняется минимальное расстояние до нее, а также количество кратчайших путей. Нам не известно, каким будет кратчайший путь до  $n$ : может сохраниться уже найденное значение, либо алгоритм Дейкстры может найти новое значение позже. Если значение сохранится, то стоит суммировать количество путей к  $n$ , так как в итоге эта величина может потребоваться для вычисления числа кратчайших путей к другим вершинам. Если же на данный момент кратчайший путь нам не известен, тогда суммирование числа путей, ведущих к  $n$ , было бессмысленным.

Но это нормально, потому что мы все равно просто сбросим значение количества путей, если найдем более короткий.

## Код

Решение этой задачи я начал с листинга 5.4 и внес необходимые изменения для поиска количества кратчайших путей. Обновленный код приведен в листинге 5.5.

**Листинг 5.5.** Вычисление длины кратчайшего пути и количества таких путей до города бабушки

```
void solve(edge *adj_list[], int num_towns, int store[]) {
    static int done[MAX_TOWNS + 1][2];
    static int min_distances[MAX_TOWNS + 1][2];
    static int num_paths[MAX_TOWNS + 1][2]; ❶
    int i, j, state, found;
    int min_distance, min_town_index, min_state_index, old_distance;
    edge *e;

    for (state = 0; state <= 1; state++)
        for (i = 1; i <= num_towns; i++) {
            done[i][state] = 0;
            min_distances[i][state] = -1;
            num_paths[i][state] = 0; ❷
        }
    min_distances[1][0] = 0;
    num_paths[1][0] = 1; ❸

    for (i = 0; i < num_towns * 2; i++) {
        min_distance = -1;
        found = 0;
        for (state = 0; state <= 1; state++)
            for (j = 1; j <= num_towns; j++) {
                if (!done[j][state] && min_distances[j][state] >= 0) {
                    if (min_distance == -1 || min_distances[j][state] < min_distance) {
                        min_distance = min_distances[j][state];
                        min_town_index = j;
                        min_state_index = state;
                        found = 1;
                    }
                }
            }
        if (!found)
            break;
        done[min_town_index][min_state_index] = 1;

        if (min_state_index == 0 && store[min_town_index]) {
            old_distance = min_distances[min_town_index][1];
            if (old_distance == -1 || old_distance >= min_distance) { ❹
                min_distances[min_town_index][1] = min_distance;
            }
        }
    }
}
```

```

        if (old_distance == min_distance) ❸
            num_paths[min_town_index][1] += num_paths[min_town_index][0];
        else
            num_paths[min_town_index][1] = num_paths[min_town_index][0];
            num_paths[min_town_index][1] %= 1000000; ❹
    }
} else {
    e = adj_list[min_town_index];
    while (e) {
        old_distance = min_distances[e->to_town][min_state_index];
        if (old_distance == -1 ||
            old_distance >= min_distance + e->length) {
            min_distances[e->to_town][min_state_index] = min_distance +
                                                    e->length;

            if (old_distance == min_distance + e->length) ❺
                num_paths[e->to_town][min_state_index] +=
                    num_paths[min_town_index][min_state_index];
            else
                num_paths[e->to_town][min_state_index] =
                    num_paths[min_town_index][min_state_index];
            num_paths[e->to_town][min_state_index] %= 1000000; ❻
        }
        e = e->next;
    }
}
}
printf("%d %d\n", min_distances[num_towns][1], num_paths[num_towns][1]); ❼
}

```

Я добавил массив `num_paths`, отслеживающий количество путей, найденных для каждой вершины ❶, и установил все его элементы равными 0 ❷. Единственный ненулевой элемент в `num_paths` — это стартовая вершина (1, 0), у которой есть один путь с нулевой протяженностью (путь, начинающийся и заканчивающийся в стартовой вершине) ❸.

Остальные изменения состоят в обновлении `num_paths`. Как было сказано, здесь возможны два случая. Если мы находим другой маршрут до вершины, используя текущее расстояние пути до нее, то добавляем его к ранее известному количеству путей. А вот второй случай может вызвать ошибку, если не расширить проверку условия «больше чем» проверкой равенства ❹. Если использовать тот же код, что и ранее,

```
if (old_distance == -1 || old_distance > min_distance) {
```

то количество путей будет обновляться только при нахождении более короткого и не будет возможности накапливать кратчайшие пути от нескольких источников. Поэтому мы используем не `>`, а `>=`:

```
if (old_distance == -1 || old_distance >= min_distance) {
```



Это позволяет учесть все варианты кратчайшего пути, если его длина не меняется.

Рассмотрим реализацию двух вариантов обновления количества путей. В нашем коде есть два места, в которых алгоритм Дейкстры может найти кратчайшие пути. Во-первых, это код, который отслеживает ребро с весом 0 из состояния 0 ❸. Если кратчайший путь остается прежним, мы его прибавляем; если же обнаруживается новый кратчайший путь, мы делаем сброс. Второе дополнение ❷ касается кода, перебирающего ребра, которые исходят из текущей вершины. В обоих случаях мы используем оператор вычисления по модулю ❹ ❸, чтобы количество кратчайших путей не превысило 1 000 000.

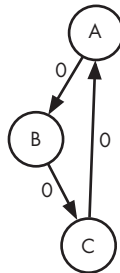
Заключительным изменением является обновление вызова `printf` ❹, чтобы вывести количество кратчайших путей к бабушке.

Теперь можно отправлять решение на проверку. В завершение же предлагаю немного поговорить о корректности.

### Корректность алгоритма

В графах «Дороги к бабушке» нет ребер с отрицательным весом, поэтому мы уверены, что алгоритм Дейкстры верно найдет все длины кратчайших путей. Ребра с нулевым весом — по одному из каждого города с кондитерской в состоянии 0 до того же города в состоянии 1 — не представляют проблемы для алгоритма Дейкстры при поиске кратчайших путей.

Однако является ли корректной обработка ребер с нулевым весом при поиске *количества* кратчайших путей? Если мы допустим существование произвольно расположенных ребер с весом 0, то количество кратчайших путей может стать *бесконечным*. Взгляните на рис. 5.6, где ребра с нулевым весом ведут от А к В, от В к С и от С к А. К примеру, кратчайший путь от А к С равен 0, и у нас есть бесконечное число таких путей:  $A \rightarrow B \rightarrow C$ ,  $A \rightarrow B \rightarrow C \rightarrow A \rightarrow B \rightarrow C$  и т. д.



**Рис. 5.6.** Граф с бесконечным количеством кратчайших путей

К счастью, в графах задачи про путешествие к бабушке циклы из ребер с нулевым весом невозможны. Предположим, что из вершины  $u$  к вершине  $v$  ведет ребро с нулевым весом. Это означает, что  $u$  находится в состоянии 0, а  $v$  в состоянии 1. Мы никогда не сможем попасть из  $v$  обратно в  $u$ , потому что наши графы не дают возможности возвращаться из состояния 1 в 0.

Закончу я следующим утверждением: если вершина отмечена обработанной, значит, общее число кратчайших путей до нее окончательно определено.

Наш алгоритм продолжает выполняться, находя кратчайшие пути и определяя их количество, как вдруг... допускает ошибку и устанавливает некоторую вершину  $n$  как обработанную, пропустив этап нахождения ее кратчайших путей. Нам нужно доказать, что подобная ошибка возникнуть не может.

Предположим, что некоторые кратчайшие пути к  $n$  заканчиваются ребром  $m \rightarrow n$ . Если вес  $m \rightarrow n$  больше нуля, то кратчайший путь до  $m$  окажется короче, чем кратчайший путь до  $n$  (его длина равна кратчайшему пути до  $n$  минус, вес ребра  $m \rightarrow n$ ). Алгоритм Дейкстры анализирует вершины, постепенно удаляясь от стартовой, поэтому вершина  $m$  должна быть к этому моменту обработана. Когда алгоритм Дейкстры устанавливает  $m$  как обработанную, он уже прошел по всем ребрам, ведущим от  $m$ , включая  $m \rightarrow n$ . Поскольку количество путей от  $m$  было установлено корректно ( $m$  же обработана), алгоритм включает их все в число путей к  $n$ .

А что, если  $m \rightarrow n$  — это ребро с нулевым весом? Нам нужно, чтобы  $m$  была обработана до  $n$ , иначе правильности определения путей при последующем исследовании исходящих из  $m$  ребер доверять будет нельзя. Нам известно, что ребра с нулевыми весами исходят из вершины в состоянии 0 в вершину в состоянии 1, значит,  $m$  должна находиться в состоянии 0, а  $n$  — в состоянии 1. Тогда кратчайший путь до  $m$  должен быть таким же, что и кратчайший путь до  $n$ , поскольку ребро с нулевым весом к кратчайшему пути до  $m$  ничего не добавляет. Тогда в определенный момент, когда  $m$  и  $n$  еще не обработаны, алгоритму нужно будет выбирать, какую из них отметить обработанной первой. Он выберет  $m$ , поскольку я написал код, в котором при возникновении подобной ситуации преимущество получает вершина с состоянием 0, а не 1.

Действовать нужно осторожно. Вот тестовый пример, который показывает, почему вершины в состоянии 0 нужно обрабатывать до вершин в состоянии 1:

```

4
0 3 1 2
3 0 2 1
1 2 0 5
2 1 5 0
2
2 3

```

Проследите действия измененного алгоритма Дейкстры на этом примере. Каждый раз, когда возникает вопрос, какую из вершин отметить обработанной, выбирайте имеющую состояние 0. Тогда вы получите верный ответ: длину кратчайшего пути 4 и четыре кратчайших пути. Далее проведите вычисления еще раз, но в случаях равнозначности вершин выбирайте ту, что находится в состоянии 1. В ответе будет получена длина пути 4, однако вместо четырех кратчайших путей будет найдено всего два.

## Выводы

Алгоритм Дейкстры разработан для нахождения быстрееших путей в графах. В этой главе мы узнали, как создать модель задачи в виде взвешенного графа и как использовать сам алгоритм. Более того, алгоритм Дейкстры, подобно BFS в главе 4, позволяет решать схожие, но отличающиеся контекстом задачи. В задаче про путешествие к бабушке мы нашли количество кратчайших путей с помощью модифицированного алгоритма. Причем нам не пришлось полностью переделывать решение. Перед нами не всегда будет стоять задача найти именно и только кратчайший путь. Если бы алгоритм Дейкстры был столь ограничен и находил только кратчайший путь, то он бы не помог в случае изменения контекста. В действительности, тогда бы это был мощный алгоритм, но из категории «все или ничего». К счастью, алгоритм Дейкстры применяется более широко. Если вы продолжите работать с графовыми алгоритмами после прочтения этой книги, то наверняка будете встречаться с идеями, вытекающими из алгоритма Дейкстры. Задач могут быть миллионы, а число алгоритмов значительно меньше. Лучшие из них строятся на идеях столь гибких, что позволяют выходить за рамки своего изначального предназначения.

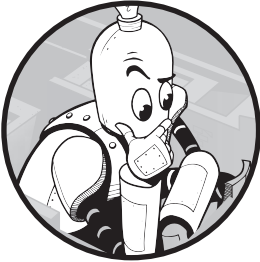
## Примечания

Задача «Мышиный лабиринт» взята из программы соревнования юго-западного региона Европы 2001 года, организованного ассоциацией ACM (2001 ACM Southwestern Europe Regional Contest). Задача «Дорога к бабушке» взята из финального раунда Южноафриканской олимпиады по программированию 2008 года (2008 South African Programming Olympiad).

Для более глубокого изучения поиска в графах и вариантов его применения в задачах соревновательного программирования я рекомендую книгу 2013 года «Competitive Programming» («Спортивное программирование») Стивена Халима (Steven Halim) и Феликса Халима (Felix Halim).

# 6

## Двоичный поиск



Эта глава целиком посвящена двоичному поиску. Если вы не знаете, что это такое, — чудесно! Меня вдохновляет возможность обучить вас высокопроизводительной технике системного выбора оптимального решения из зиллиарда возможных. Если же двоичный поиск вам знаком и вы считаете, что он используется только для поиска по упорядоченному массиву, то и это прекрасно, потому что здесь вы узнаете, что данный алгоритм способен на значительно большее! При этом, чтобы не отвлекаться на многим знакомый материал, а сосредоточиться на новом, мы не будем затрагивать поиск по упорядоченным массивам.

Что общего между минимизацией количества жидкости, необходимой для кормления муравьев, максимизацией минимального расстояния прыжка между камнями, поиском наилучшего района города для проживания и щелканьем переключателями для открытия дверей в пещере? Вскоре мы это выясним!

### Задача 1. Кормление муравьев

Рассмотрим задачу с платформы DMOJ под номером `sosi14c4p4`.

#### Условие

У Боби есть террариум в форме дерева. Каждое ребро этого дерева является трубкой, по которой стекает жидкость. Некоторые ребра образованы супертрубками,

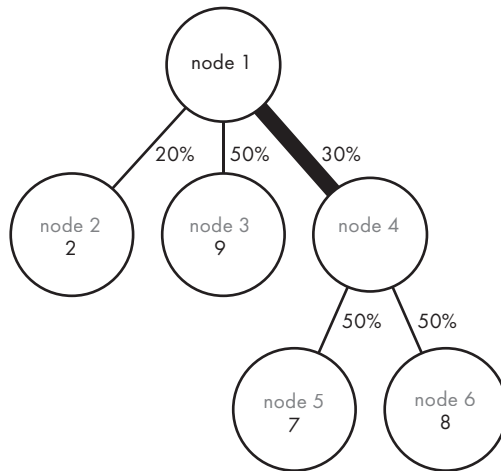
по которым может протекать большее количество жидкости. В каждом листе (концевом узле) дерева находится один ручной муравей (согласен, что контекст фантастический, но сама задача очень интересная).

У каждой трубки-ребра есть значение веса, указывающее процент протекающей по ней жидкости. К примеру, из узла  $n$  исходят три трубки, по которым стекает 20%, 50% и 30% жидкости от общего объема. Если в узел  $n$  поступает 20 литров жидкости, тогда по первой трубке стекает  $20 \times 0,2 = 4$  литра, по второй  $20 \times 0,5 = 10$  литров, по третьей  $20 \times 0,3 = 6$  литров.

Каждая супертрубка может работать в стандартном или суперрежиме — в зависимости от выбора Боби. Если включен суперрежим, то количество протекающей жидкости возводится в квадрат.

Боби заливает жидкость в корень дерева. Его цель — дать каждому муравью не меньше необходимого количества, минимизируя при этом заливаемый объем.

Конкретизируем описание на примере террариума на рис. 6.1.



**Рис. 6.1.** Образец террариума

Узлы пронумерованы от 1 до 6. Концевые узлы (2, 3, 5 и 6) имеют дополнительные значения, указывающие необходимое каждому муравью количество жидкости. Также указаны процентные значения для ребер. Обратите внимание, что сумма этих значений для ребер, исходящих из одного узла, всегда равна 100%.

В дереве есть одна супертрубка, ведущая из узла 1 к узлу 4. Она обозначена жирной линией. Предположим, что в корень заливается 20 литров жидкости.

Супертрубка получает 30% от 20 литров. Если суперрежим этой трубки отключен, то через нее протекает 6 литров. Если же этот режим активен, то через нее пройдет  $6^2 = 36$  литров.

### Входные данные

Входные данные содержат один тестовый пример, состоящий из следующих строк:

- целого числа  $n$ , задающего количество узлов в дереве. Значение  $n$  находится между 1 и 1000. Узлы дерева пронумерованы от 1 до  $n$ , а корнем является узел 1;
- $(n - 1)$  строк, используемых для построения дерева. Каждая содержит данные об одной трубке и состоит из четырех целых чисел: номеров двух узлов, соединяемых этой трубкой, ее процентного значения (между 1 и 100) и указания о возможности суперрежима (0 означает отсутствие, 1 — наличие);
- строки, содержащей  $n$  целых чисел, по одному для каждого узла, сообщающих количество литров, необходимых муравью в этом узле. Каждому муравью требуется от 1 до 10 литров жидкости. Все неконцевые узлы (в которых нет муравья) содержат значение  $-1$ .

Вот входные данные, которые сгенерируют образец террариума с рис. 6.1:

```
6
1 2 20 0
1 3 50 0
1 4 30 1
4 5 50 0
4 6 50 0
-1 2 9 -1 7 8
```

Обратите внимание, что первая строка (число 6) указывает количество узлов дерева, а не количество строк, его выстраивающих. Число строк, создающих дерево (в данном случае пять), всегда будет на единицу меньше числа узлов.

### Выходные данные

Требуется вывести минимальное количество литров жидкости, которое Боби должен залить в корневой узел дерева, чтобы напоить всех муравьев. Точность должна быть выше четырех цифр после запятой. Верное значение не будет превышать двух миллиардов.

Время на решение тестового примера — 2,5 секунды.

## Новая форма задачи с деревом

Как и в главе 2, здесь мы работаем с деревьями. В данном случае для исследования дерева террариума можно использовать рекурсию (алгоритм поиска по полному графу, например BFS, окажется излишним, поскольку циклов здесь нет).

Решения двух задач из главы 2 основывались на структуре дерева и хранящихся в узлах значениях:

- В «Трофеях Хэллоуина» мы вычисляли общее количество конфет, складывая значения в концевых узлах, а также общее число пройденных улиц, используя высоту и форму дерева.
- В «Расстоянии до потомка» мы вычисляли количество потомков на нужной глубине на основе количества детей каждого узла.

То есть все, что требовалось: число конфет, высота и форма дерева — уже было дано и закодировано в самом дереве. В новой задаче нужно найти минимальный объем жидкости, которую Боби должен залить в корень, но дерево не содержит значений объема. Есть информация о пропускной способности трубок в процентах, о наличии у трубок суперрежима и об аппетитах муравьев, но ничто напрямую не информирует нас об объеме жидкости, которую нужно залить в корень. В частности, возможность повышения пропускаемого объема через супертрубки дополнительно усложняет связь между количеством жидкости, которое необходимо муравьям, и количеством, которое нужно залить.

Поскольку дерево не предоставляет необходимую информацию в готовом виде, возьмем значение объема наугад. Итак, Боби, давай зальем в корень 10 литров.

Вероятно, вас сильно удивит, если верным ответом окажется именно 10, ведь это число я выбрал наугад. Вас также может удивить, что, выбрав произвольное число и проанализировав происходящее, мы можем многое узнать.

Вернемся к рис. 6.1 и предположим, что заливаем в корневой узел 10 литров.

Итак, 20% от 10 — это 2, значит, 2 литра жидкости поступят к муравью в узле 2. Идеально: именно 2 литра ему и надо. Продолжим.

Поскольку 50% от 10 равно 5, то муравей в узле 3 получает 5 литров жидкости. Вот теперь у нас проблема, потому что ему нужно 9 литров. При этом трубка между узлами 1 и 3 не имеет суперрежима, значит, изменить ситуацию нельзя и 10 верным решением не является.

Можно продолжить подставлять наугад и другие количества жидкости, аналогичным образом моделируя ее течение по дереву. И все же, если 10 литров оказалось

недостаточно, теперь следует ограничить диапазон до значений *больше* 10. Поскольку 10 литров было мало, любое меньшее значение также окажется недостаточным. Нет смысла пробовать 2, 7, 9, 5 литров или любое другое количество меньше 10.

Теперь давайте возьмем 20 литров. На этот раз муравей в узле 2 получает 4 литра, что вполне нормально, так как ему достаточно двух. Муравей в узле 3 получает 10 литров, что также допустимо, потому что ему нужно 9.

Трубка между узлами 1 и 4 проводит 30% жидкости, то есть 6 из 20 литров. Но при этом она еще и является супертрубкой! Если задействовать суперрежим, то она прокачает  $6^2 = 36$  литров. Значит, в узел 4 протекает 36 литров. Теперь муравьи 5 и 6 будут довольны: каждый из них получает по 18 литров при том, что в узел 5 нужно 7, а в узел 6 — 8 литров.

В отличие от 10 литров, 20 оказалось допустимым решением, но является ли оно оптимальным (то есть минимальным)? Возможно, но не обязательно. Однозначно нам известно лишь то, что проверять количество более 20 литров смысла нет. 20 уже дает нам допустимое решение, так зачем пробовать 25 или 30, которые точно хуже?

Теперь мы сократили область задачи до нахождения оптимального решения между 10 и 20 литрами. Можно продолжать подбирать числа, с каждым шагом сужая диапазон, пока он не станет столь мал, что его конечные точки окажутся точным решением.

Если рассуждать в общем, то какое количество литров следует выбирать изначально? Оптимальный объем может иметь значение вплоть до 2 миллиардов, так что, начиная с 10, можно оказаться очень далеко от истины. Установив начальное значение, что следует делать дальше? Оптимальное решение может быть значительно больше или меньше выбранного значения, значит, прибавление или вычитание 10 на каждом шаге может не слишком помочь в его поиске.

Все эти вопросы важны, и я на них отвечу... но не сразу. Сначала я хочу разобраться с тем, как считывать входные данные (чтобы иметь возможность исследовать дерево) и определять, является ли некоторое количество литров допустимым решением.

Затем я покажу вам супербыстрый алгоритм для поиска в больших диапазонах. Диапазон в два миллиарда? Мы выполним расчет для него, не напрягаясь.

## **Считывание входных данных**

В главе 2 в основе представления деревьев использовалась структура `node`. Затем в задаче «Перевод книги» главы 4 я ввел и использовал представление графа в виде списка смежности при помощи структуры `edge`. Тогда я объяснял, что использование `node` или `edge` зависит от того, где именно содержатся дополнительные атрибуты



графа — в узлах или ребрах. В текущей задаче информацию несут не только ребра (процент и наличие суперрежима), но и концевые узлы (количество необходимой каждому муравью жидкости). Следовательно, кажется резонным использовать и `edge`, и `node`. Я же предпочел применить только структуры `edge`. Согласно условию задачи, нумерация узлов начинается с 1, но без структуры `node` нам будет негде хранить данные о количестве нужной каждому муравью жидкости. По этой причине я расширяю список смежности массивом `liquid_needed`, где `liquid_needed[i]` указывает объем жидкости, необходимый муравью в узле `i`.

Вот макрос и определение типа, которые будут использоваться в нашем коде:

```
#define MAX_NODES 1000

typedef struct edge {
    int to_node, percentage, superpipe;
    struct edge *next;
} edge;
```

По аналогии с задачей про перевод книги из главы 4, а также двумя задачами из главы 5, эти структуры `edge` можно выстроить в цепочку через указатели `next`, сформировав таким образом связный список ребер. Если ребро находится в связном списке для узла `i`, то мы знаем, что родительским узлом этого ребра является `i`. Переменная `to_node` указывает на дочерний узел, соединенный ребром с родительским. `percentage` является целым числом между 1 и 100, которое сообщает процентное значение для трубки (ребра). `superpipe` — флаг, который имеет значение 1, если трубка имеет суперрежим, и 0, если такой режим отсутствует.

Теперь можно считывать дерево из входных данных, как показано в листинге 6.1.

#### Листинг 6.1. Функция `main` для построения дерева

```
int main(void) {
    static edge *adj_list[MAX_NODES + 1] = {NULL};
    static int liquid_needed[MAX_NODES + 1];
    int num_nodes, i;
    int from_node, to_node, percentage, superpipe;
    edge *e;
    scanf("%d", &num_nodes);

    for (i = 0; i < num_nodes - 1; i++) {
        scanf("%d%d%d", &from_node, &to_node, &percentage, &superpipe);
        e = malloc(sizeof(edge));
        if (e == NULL) {
            fprintf(stderr, "malloc error\n");
            exit(1);
        }
        e->to_node = to_node;
        e->percentage = percentage;
```

```

    e->superpipe = superpipe;
    e->next = adj_list[from_node];
    adj_list[from_node] = e; ❶
}

for (i = 1; i <= num_nodes; i++)
    scanf("%d", &liquid_needed[i]); ❷
solve(adj_list, liquid_needed);
return 0;
}

```

Данный код похож на код из листинга 4.15 (задача про перевод книги), но при этом проще. В частности, каждое ребро считывается из входных данных, затем устанавливаются его члены, после чего оно добавляется в список ребер для **from\_node** ❶. При этом можно было бы предположить существование соответствующего ребра из **to\_node**, потому что граф ненаправленный, но я такую возможность исключил: жидкость стекает вниз по дереву, но не вверх, значит, добавление обратных ребер излишне усложнит код, исследующий дерево.

После считывания информации о дереве остается только считать значения количества жидкости, необходимого каждому муравью. Для этого используется массив **liquid\_needed** ❷. Комбинация из **adj\_list** и **liquid\_needed** обеспечивает всю необходимую нам информацию о тестовом примере.

## Проверка пригодности решения

На следующем этапе надо определить, является ли заданное количество жидкости допустимым решением. Это очень важный шаг, потому что, когда у нас появится функция для проверки допустимости значения, мы сможем использовать ее для постепенного сужения пространства поиска до момента обнаружения оптимального решения. Для этой функции мы напишем следующий заголовок:

```

int can_feed(int node, double liquid,
             edge *adj_list[], int liquid_needed[])

```

Здесь **node** является корневым узлом дерева, **liquid** указывает количество жидкости, заливаемой в корень, **adj\_list** представляет список смежности для дерева, а **liquid\_needed** отражает объем жидкости, необходимый каждому муравью. Если значения **liquid** окажется достаточно, будет возвращаться 1, в противном случае 0.

В задачах второй главы мы разрабатывали рекурсивные функции для деревьев. Давайте теперь подумаем, можно ли применить такую функцию здесь.

Вспомним, что для использования рекурсии требуется сформулировать базовый случай, который можно решить без рекурсии. К счастью, у нас такой имеется.

Если дерево представлено одним концевым узлом, то можно сразу определить, достаточно ли количества `liquid`. Если `liquid` больше либо равно необходимому муравью количеству жидкости, то перед нами допустимое решение. В противном случае решение не допустимое.

Определить, является ли узел концевым, можно проверкой соответствующего значения в `liquid_needed`: если это `-1`, значит, он не концевой (можно также проверить, является ли связный список смежности пустым). Вот что получается:

```
if (liquid_needed[node] != -1)
    return liquid >= liquid_needed[node];
```

А теперь рассмотрим рекурсивный алгоритм. Представьте, что корневой узел некоторого дерева имеет  $p$  нисходящих трубок (а значит, и  $p$  детей). Дано количество заливаемой в корень жидкости. Используя процентные значения трубок, можно определить объем жидкости, которая поступает через каждую из них. При этом с помощью статусов супертрубок можно определить количество жидкости на выходе из каждой трубки. Если до низа трубки доходит достаточный объем питья, то заливаемой в корень жидкости хватает и нужно вернуть `1`. Если до низа хотя бы одной из трубок доходит недостаточный объем жидкости, то возвращается `0`. Это предполагает, что нужно сделать  $p$  рекурсивных вызовов, по одному для каждой нисходящей из корня трубки. Сделаем мы это в цикле, использующем список смежности для каждой такой трубки.

Код для этой функции представлен в листинге 6.2.

**Листинг 6.2.** Проверка, достаточно ли заливаемого количества жидкости

```
int can_feed(int node, double liquid,
            edge *adj_list[], int liquid_needed[]) {
    edge *e;
    int ok;
    double down_pipe;
    if (liquid_needed[node] != -1)
        return liquid >= liquid_needed[node];
    e = adj_list[node];
    ok = 1; ❶
    while (e && ok) {
        down_pipe = liquid * e->percentage / 100;
        if (e->superpipe)
            down_pipe = down_pipe * down_pipe; ❷
        if (!can_feed(e->to_node, down_pipe, adj_list, liquid_needed))
            ok = 0; ❸
        e = e->next;
    }
    return ok;
}
```

Переменная `ok` отслеживает, является ли `liquid` допустимым решением для дерева. Пока `ok` равна 1, решение считается допустимым. Инициализируется `ok` как 1 ❶ и устанавливается равной 0, если количество жидкости, прошедшей через одну из трубок, окажется недостаточным ❷. Если к концу функции `ok` по-прежнему будет 1, то все трубки провели достаточное количество жидкости и мы заключаем, что решение `liquid` допустимо.

Количество жидкости, поступающее в каждую трубку, определяется с помощью ее процентного значения. Затем, если трубка имеет суперрежим, это значение возводится в квадрат ❸... но стоп! В условии задачи сказано, что Боби должен решить, нужно ли использовать суперрежим каждой супертрубки. Однако здесь мы без вариантов возводим количество жидкости в квадрат, всегда используя суперрежим.

Причина в том, что возведение в квадрат увеличивает значения: сравните 2 и  $2^2 = 4$ , 3 и  $3^2 = 9$ , и т. д. Поскольку нам нужно знать, является ли данное количество жидкости допустимым, и штрафа за использование супертрубки нет, можно предполагать максимальный объем жидкости. Возможно, есть вариант обойтись и без использования суперрежима какой-то супертрубки, но никто не просит нас экономить. Не беспокойтесь о том, что возведение в квадрат уменьшает положительные числа меньше единицы, например 0,5 ( $0,5^2 = 0,25$ ). Действительно, в подобных случаях активировать супертрубку мы бы не стали. Поскольку каждому муравью нужно не менее 1 литра жидкости, значит, если в одном из узлов мы получим 0,5 литра, то независимо от наших действий муравьи в поддереве этого узла останутся голодными и мы вернем 0.

Теперь посмотрим, насколько полезна функция `can_feed`, и для этого продолжим работу, сделанную в разделе «Новая форма задачи с деревом». Там мы показали, что 10 литров недостаточно для рассматривавшегося примера. Закомментируйте вызов `solve` в нижней части листинга 6.1 (эту функцию `solve` мы вскоре напомним) и добавьте вызов `can_feed` для проверки объема 10 литров:

```
printf("%d\n", can_feed(1, 10, adj_list, liquid_needed));
```

В результате должен вернуться 0, означающий, что 10 литров мало. В том же разделе мы продемонстрировали, что 20 литров будет достаточно. Измените вызов `can_feed` для проверки 20 литров:

```
printf("%d\n", can_feed(1, 20, adj_list, liquid_needed));
```

В результате должна вернуться 1, означающая, что 20 литров достаточно.

Теперь нам известно, что 10 недостаточно, а 20 хватает. Давайте сужать этот диапазон далее. Попробуйте 15, на что должен вернуться 0. Значит, 15 маловато. Теперь оптимальный ответ больше 15, но не больше 20.

Попробуйте 18 — в этом случае жидкости окажется достаточно. А как насчет 17? Нет, 17 уже мало, как и 17,5 или 17,9, значит, оптимальное решение — 18.

Этих результатов узконаправленного поиска достаточно, теперь можно их систематизировать.

## Поиск решения

Из условия задачи известно, что оптимальное значение не больше 2 миллиардов. Следовательно, решение находится в огромном пространстве поиска. Наша цель сократить это пространство максимально быстро, не тратя времени на пустые догадки.

Промахнуться легко. Например, если начать с предположения 10, а оптимальное решение окажется равным двум миллиардам, тогда, по сути, это предположение следует считать промахом, исключаяющим из области поиска только значения от 0 до 10. Но 10 окажется отличным вариантом, если ответ будет равен, например, 8, потому что тогда первый же шаг сразу сократит диапазон до 0 — 10, и на поиск решения много времени не уйдет. Тем не менее совершение таких выстрелов в небо себя не оправдывает, поскольку редкое везение не перекроет наиболее вероятный случай, когда такая догадка практически бесполезна. Именно поэтому, когда вас попросят отгадать число между 0 и 1000, вы вряд ли начнете с 10. Конечно же, если на такое предположение вам скажут «меньше», то вы очень обрадуетесь, но если ответом будет «больше», как скорее всего и получится, то можно считать, что первую попытку вы потратили зря.

Чтобы гарантировать получение с каждой догадкой максимальной информации, мы всегда будем загадывать середину диапазона. Для этого нужно будет вести две переменные, `low` и `high`, которые будут хранить нижнюю и верхнюю границы текущего диапазона соответственно. Затем мы будем вычислять середину этого диапазона, `mid`, проверять допустимость `mid`, по результату обновляя `low` и `high`. Вся эта стратегия прописана в листинге 6.3.

### Листинг 6.3. Поиск оптимального решения

```
#define HIGHEST 2000000000

void solve(edge *adj_list[], int liquid_needed[]) {
    double low, high, mid;
    low = 0;
    high = HIGHEST;
    while (high - low > 0.00001) { ❶
        mid = (low + high) / 2; ❷
        if (can_feed(1, mid, adj_list, liquid_needed)) ❸
            high = mid;
        else
```

```
    low = mid;  
}  
printf("%.4lf\n", high); ❹  
}
```

Важно инициализировать `low` и `high` так, чтобы в их диапазоне гарантированно содержалось оптимальное решение. Значение `low` будет постоянно меньшим или равным оптимальному решению, а `high` — большим или равным оптимальному решению. Начнем со значения `low`, равного 0. Поскольку каждому муравью требуется не менее 1 литра, 0 литров будет определено меньше или равно оптимальному решению. Для `high` же стартовое значение установим как два миллиарда, поскольку условие задачи гарантирует, что значение оптимального решения не может превышать этого числа.

Условие цикла `while` ведет к минимизации диапазона между `low` и `high` ❶. Нам нужна точность выше четырех цифр после запятой, отсюда появляются четыре 0 после десятичной точки в 0.00001.

Первым делом в цикле вычисляется середина диапазона. Для этого мы берем среднее от `low` и `high`, сохраняя результат в `mid` ❷.

Теперь пора проверить `mid` литров на допустимость с помощью `can_feed` ❸. Если `mid` окажется допустимым числом, значит, угадывание значений выше `mid` бесполезно. Следовательно, мы устанавливаем `high = mid`, уменьшая максимум диапазона до `mid`.

Если же `mid` окажется недопустимым вариантом, то бесполезно угадывать меньшие значения. Следовательно, мы устанавливаем `low = mid`, повышая минимум диапазона до `mid`.

По завершении цикла `low` и `high` окажутся очень близки. Я вывожу в конце `high` ❹, но вывод `low` также вполне допустим.

Такая техника, в которой мы последовательно сокращаем диапазон поиска в два раза, пока он не станет очень мал, называется *двоичным поиском*. Это удивительно мощный алгоритм, что будет доказано в оставшихся разделах этой главы. Он также очень быстр и способен с легкостью обрабатывать диапазоны из миллиардов и триллионов значений.

Отправьте это решение на проверку, и продолжим. О двоичном поиске предстоит еще многое узнать.

## Двоичный поиск

«Кормление муравьев» относится к типу задач, для решения которых нет метода лучше двоичного поиска. Для них характерны две особенности:

- **Особенность 1: сложность выбора метода решения, но простота проверки допустимости ответа.** Для некоторых задач сложно определить алгоритм поиска оптимального решения. К счастью, во многих таких случаях значительно проще определять, является ли предлагаемое решение допустимым. Такова и была ситуация в задаче про муравьев: мы не знали, как найти оптимальное решение, но при этом видели, как определять, является ли некоторый объем жидкости допустимым.
- **Особенность 2: четкое разделение решений на допустимые и недопустимые.** Нам нужно, чтобы задача позволяла провести четкую черту, разделяющую допустимые и недопустимые решения. В задаче про муравьев недостаточные объемы жидкости были недопустимы, а избыточные — допустимы. То есть когда мы рассматривали значения объема, то могли установить для них диапазоны недопустимых и допустимых значений. После нахождения первого допустимого значения мы определяли диапазон, не содержащий недопустимых значений. Предположим, что выполняем проверку для 20 литров и выясняем, что этого мало. Следовательно, мы все еще не достигли допустимой части пространства поиска и нужно пробовать большие значения. Если же 20 литров оказывается достаточным, значит, мы находимся в допустимой части пространства и нужно пробовать меньшие значения.
- Если задача не отвечает второму требованию, то двоичный поиск будет бесполезен. Например, есть задачи, где меньшие значения являются недопустимыми, средние допустимыми, а большие снова недопустимыми. Однако вполне нормально, если область поиска будет переходить от допустимых значений к недопустимым, а не наоборот. Следующая задача как раз является примером такого случая.

## **Время выполнения двоичного поиска**

Причина высокой эффективности двоичного поиска состоит в том, что за одну итерацию он добивается огромного прогресса. Представьте, что мы ищем оптимальное решение для диапазона в два миллиарда. Одна итерация двоичного поиска уже отбрасывает половину этого диапазона, оставляя один миллиард. Только представьте: с помощью всего одного условия `if` и одного обновления переменной `mid` мы продвигаемся на миллиард единиц к цели! Если двоичный поиск предпримет  $p$  итераций для поиска по диапазону в один миллиард, то для поиска в диапазоне два миллиарда ему потребуется  $p + 1$  итераций. По сравнению с шириной диапазона количество необходимых итераций растет очень медленно.

Говоря точнее, число итераций, предпринимаемых алгоритмом для сокращения диапазона с  $n$  до 1, примерно равно числу делений  $n$  пополам, чтобы получить 1. Возьмем, к примеру, диапазон шириной 8. После одной итерации он сократится

до 4. После двух итераций — до 2. Спустя три итерации получится уже диапазон шириной 1. Более того, если нас не интересует точность выше 1, то на этом все, итого три итерации.

Существует математическая функция, называемая *двоичный логарифм* или *логарифм по основанию 2*, которая, получая значение  $n$ , сообщает, сколько раз нужно поделить  $n$  на 2, чтобы получить 1 или меньше. Записывается она как  $\log_2 n$  или, когда из контекста очевидно, что в основании два, просто как  $\log n$ . Например,  $\log 28 = 3$ , а  $\log 216 = 4$ .  $\log 22\,000\,000\,000 = 30,9$ , значит, на сокращение этого диапазона до 1 уходит примерно 31 итерация.

Двоичный поиск является примером алгоритма с *логарифмическим временем*, которое записывается как  $O(\log m)$  (обычно вместо  $m$  используется  $n$ , но ниже в этом разделе символом  $n$  будет обозначена другая величина). Эта формула позволяет определить число итераций, необходимое, чтобы сократить ширину диапазона поиска с  $m$  до 1. Однако в задаче «Кормление муравьев» нам требовалось пойти дальше и получить точность до пятой цифры после запятой. Каким должно быть  $m$  в этом случае?

Пора окончательно разобраться, как мы использовали в первой задаче двоичный поиск: мы выполняли больше чем  $\log_2 2\,000\,000\,000$  итераций алгоритма, потому что не останавливались, когда ширина диапазона достигала 1. Остановка происходила только при достижении точности выше четырех знаков после запятой. При добавлении пяти нулей количество совершаемых итераций равно  $\log_2 200\,000\,000\,000\,000$ , который округляется до 48. Всего 48 итераций необходимо для получения решения с точностью выше четырех цифр после запятой из огромнейшего диапазона в триллионы значений. Вот он каков, двоичный поиск!

В дереве из  $n$  узлов функция `can_feed` в задаче «Кормление муравьев» (листинг 6.2) выполняется в линейном времени, то есть продолжительность ее выполнения пропорциональна  $n$ . Мы вызываем эту функцию  $\log_2(m \times 10^4)$  раз, где  $m$  — ширина диапазона (два миллиарда в тестовых примерах). Это пропорционально  $\log m$  работы. Тогда в общей сложности выполняется  $n$  работы  $\log m$  раз. То есть время можно записать как  $O(n \log m)$ . Оно не совсем линейно из-за множителя  $\log m$ , но двоичный поиск все равно очень быстр.

## Определение допустимости

В алгоритмах двоичного поиска мне больше всего нравится то, что для определения допустимости значения зачастую нужно использовать другой метод. То есть снаружи у нас двоичный поиск, а внутри что-то еще, проверяющее каждое значение на допустимость. Причем это «что-то еще» может быть чем угодно. В «Кормлении муравьев» это был поиск по дереву, в следующей задаче это будет жадный



алгоритм, а в третьей задаче главы и вовсе динамическое программирование. И хотя здесь я его использовать не буду, но скажу, что в некоторых задачах для определения допустимости решения задействуется графовый алгоритм. Таким образом, в следующих задачах вновь надо будет применять материал, пройденный в предыдущих главах.

Для определения допустимости решений нередко требуется проявить изрядную креативность (к счастью, не столь изрядную, как для нахождения оптимальных решений).

### **Поиск по упорядоченному массиву**

Если двоичный поиск был вам знаком до прочтения этой главы, то наверняка в контексте поиска по упорядоченному массиву. В типичном сценарии дается массив  $a$  и значение  $v$ . Задача — найти наименьший индекс элемента в  $a$ , который больше или равен  $v$ . К примеру, если дан массив  $\{-5, -1, 15, 31, 78\}$ , а  $v$  равно 26, то поиск вернет индекс 3, потому что соответствующее ему значение (31) является первым, которое больше или равно 26.

Почему здесь сработает двоичный поиск? Проверим две особенности этой задачи:

- **Особенность 1.** Без двоичного поиска нахождение оптимального значения подразумевало бы долгое сканирование всего массива. Следовательно, добиться оптимальности сложно, но при этом легко определить допустимость: если я дам вам индекс  $i$ , то вы сходу скажете, больше соответствующий элемент или равен  $v$ , просто сравнив их.
- **Особенность 2.** Любые элементы меньше  $v$  идут раньше любых элементов, которые больше или равны  $v$ , так как  $a$  упорядочен. Это значит, что недопустимые значения будут идти до допустимых.

Верно, что двоичный поиск можно использовать для нахождения подходящего индекса массива в логарифмическом времени, но задачу про муравьев мы решали также с помощью двоичного поиска, а никакого массива задано не было. Не стоит ограничивать использование двоичного поиска только случаями, когда анализу подлежит массив. Этот алгоритм намного более гибок.

## **Задача 2. Прыжки вдоль реки**

А теперь рассмотрим задачу, в которой для определения допустимости решений потребуется жадный алгоритм.

Это задача с платформы POJ под номером 3258.

### Условие

Дана река длиной  $L$ , вдоль которой лежат камни. Один камень лежит в точке 0 (начало реки), другой в точке  $L$  (конец реки), а между ними находится еще  $n$  камней. Например, вдоль реки длиной 12 камни могут располагаться в следующих местах: 0, 5, 8 и 12.

Корова начинает с первого камня (локация 0), перепрыгивает оттуда на второй камень, потом со второго на третий и так далее, пока не достигает последнего камня в точке  $L$ . Минимальная длина прыжка равна минимальному расстоянию между любыми двумя соседними камнями. В примере выше минимальное расстояние будет 3 (между точками 5 и 8).

Фермеру Джону наскучили короткие прыжки коровы, и он хочет максимально увеличить их длину. Камни в точках 0 или  $L$  он убрать не может, но может удалить  $m$  других камней.

Предположим, что в приведенном выше примере Джон может убрать один камень. В этом случае перед ним стоит выбор — убрать камень из точки 5 или 6. Если он уберет его из точки 5, то минимальное расстояние прыжка станет 4 (из 8 в 12). Однако если он уберет камень из точки 8, то получит минимальный прыжок большей длины (из 0 в 5).

Задача — максимизация минимальной длины прыжка, которой может добиться фермер, убрав  $m$  камней.

### Входные данные

Входные данные состоят из одного тестового примера, содержащего следующие строки:

- Три целых числа:  $L$  — протяженность реки;  $n$  — количество камней, не включая начальный и конечный камни;  $m$  — количество камней, которые Джон может убрать. Значение  $L$  может находиться в диапазоне от 1 до 1 000 000 000,  $n$  — от 0 до 50 000, а  $m$  — между 0 и  $n$ .
- $n$  строк, каждая из которых указывает координату камня. При этом два камня в одном месте оказаться не могут.

### Выходные данные

Требуется вывести максимально достижимое минимальное расстояние прыжка. Для примера выше выводом будет 5.

Время на решение тестового примера — две секунды.

## Жадная идея

Когда в главе 3 мы решали задачу «Экономные покупатели», я представил идею жадного алгоритма. Такой алгоритм выглядит подходящим для данной ситуации без привязки к последствиям его использования. Жадный алгоритм зачастую реализуется просто, нужно только установить правило, на основе которого он будет совершать выбор. К примеру, при решении задачи «Экономные покупатели» алгоритм выбирал вариант с наименьшей ценой за яблоко. Тогда жадный алгоритм выдавал неверные результаты, и этот урок стоит запомнить: сформулировать жадный алгоритм несложно, но не всегда легко определить правильное условие.

Я не выделил отдельную главу под жадные алгоритмы по двум причинам. Во-первых, они не настолько широко применимы, как, например, динамическое программирование. Во-вторых, если они и подходят, то обычно по уникальным, характерным для конкретной задачи причинам. За многие годы практики я неоднократно обманывался кажущейся привлекательностью жадных алгоритмов, но на деле они оказывались абсолютно ошибочными. Чтобы отличить реально подходящий алгоритм от кажущегося подходящим, требуется тщательное доказательство корректности.

И все же жадные алгоритмы скрытно пробрались в главу 5 под видом алгоритма Дейкстры. Как правило, специалисты относят этот алгоритм именно к жадным. Когда он объявляет о нахождении кратчайшего пути к узлу, то уже не возвращается к этому решению. Алгоритм окончательно утверждает найденное значение, и дальнейшее исследование пространства поиска не влияет на этот результат.

Сейчас нас вновь ждет встреча с жадными алгоритмами. Когда я впервые столкнулся с задачей «Прыжки вдоль реки», то сразу интуитивно почувствовал, что ее можно решить с помощью жадного алгоритма. Интересно, а вы сможете сформулировать его интуитивно, как это сделал я? Вот жадное правило: найти два камня, расположенных ближе всех друг к другу, убрать тот, что ближе к другому своему соседу, и повторить процесс.

Вернемся к примеру из описания задачи:

12 2 1  
5  
8

Камни расположены в точках 0, 5, 8 и 12. Убрать можно один. Самые близкие друг к другу камни расположены в 5 и 8, значит, жадное правило приведет к удалению одного из них. Камень в точке 8 находится от своего соседа справа на расстоянии 4. Камень в точке 5 находится от своего соседа слева на расстоянии 5. Согласно жадному правилу, удаляется камень из точки 8. В этом примере правило работает корректно.

Теперь посмотрим, как сработает алгоритм в примере побольше. Предположим, что длина реки равна 12 и можно удалить два камня. Вот тестовый пример:

```
12 4 2
1
3
8
9
```

Камни расположены в точках 0, 1, 3, 8, 9 и 12. Как поступит жадный алгоритм? Наилближайшие друг к другу камни находятся в точках 0 и 1, а также 8 и 9. Придется выбрать одну пару, пусть ею будет 0 и 1. Поскольку удаление камня из точки 0 не допускается, следует убрать его соседа из 1. После этого остались камни в 0, 3, 8, 9 и 12.

Теперь ближайшие камни находятся в точках 8 и 9. Расстояние между 9 и 12 меньше, чем между 8 и 3, значит, камень удаляется из точки 9. Остаются камни 0, 3, 8 и 12. Минимальное расстояние прыжка здесь равно 3, и это значение является верным ответом. Жадный алгоритм вновь побеждает.

Ведь так? Нужно просто продолжать исключать наименьшие расстояния между камнями. Разве можно добиться лучшего? Жадный алгоритм прекрасен.

К сожалению, он оказывается неверен. Советую вам попробовать самостоятельно придумать контрпример, прежде чем прочитаете следующий абзац.

Вот один из таких контрпримеров:

```
12 4 2
2
4
5
8
```

Камни располагаются в точках 0, 2, 4, 5, 8 и 12. Допускается убрать два. Жадный алгоритм определяет ближайшими камни в точках 4 и 5. Из них он удаляет камень 4, поскольку расстояние между 4 и 2 меньше, чем между 5 и 8. Остаются 0, 2, 5, 8 и 12.

Теперь жадное правило определяет ближайшими камни в точках 0 и 2. Удалять стартовый камень нельзя, значит, удаляется камень из 2. Остаются 0, 5, 8 и 12. Минимальная длина прыжка здесь — 3. Однако в данном случае алгоритм с задачей не справился, потому что можно добиться, чтобы это расстояние равнялось 4. Для получения такого значения нужно удалить камни в точках 2 и 5. Так мы получим 0, 4, 8 и 12.

Что же пошло не так? Удалив камень в локации 4 на первом ходе, жадный алгоритм создал два прыжка с длиной три. На втором ходе он может убрать только один

из этих двух, что никак не позволит получить итоговое минимальное расстояние больше трех.

Мне не известен жадный алгоритм, который бы корректно решал эту задачу. Как и в кормлении муравьев, здесь сложно придумать правильный метод решения, но, к счастью, этого и не требуется.

### **Проверка допустимости**

В разделе «Двоичный поиск» я предложил два ориентира, указывающих на возможность решения с помощью двоичного поиска: большая простота проверки допустимости, чем определения оптимального решения; четкий переход в области поиска между допустимыми и недопустимыми решениями. Посмотрим, как в этом отношении выглядит задача «Прыжки вдоль реки».

Вместо поиска оптимального решения мы попробуем ответить на вопрос: можно ли добиться длины прыжка не менее  $d$ ? Если нам удастся получить ответ, то можно будет использовать двоичный поиск для нахождения наибольшего допустимого значения  $d$ .

Вот тестовый пример, которым закончился предыдущий раздел:

12 4 2  
2  
4  
5  
8

Камни расположены в точках 0, 2, 4, 5, 8 и 12. Убрать можно два. Вопрос: минимум сколько нужно убрать камней, чтобы длина прыжка получилась не менее 6? Разберем пример слева направо и проверим.

Камень в точке 0 трогать нельзя — так гласит условие задачи. Тогда, очевидно, у нас не остается выбора в отношении камня в точке 2: его необходимо убрать. Если этого не сделать, то минимальное расстояние между двумя камнями будет меньше 6. Итак, после этого остались камни в точках 0, 4, 5, 8 и 12. Теперь рассмотрим камень в точке 4 — нужно ли его оставить или убрать? В нашем случае его необходимо также убрать. Если этого не сделать, то минимальное расстояние между камнями составит 4, что меньше нужных нам 6. После его удаления остаются камни в точках 0, 5, 8 и 12. Камень в точке 5 тоже надо убрать, потому что расстояние от него до 0 равно всего лишь 5. Это третье удаление, после которого остаются камни в точках 0, 8 и 12. Теперь нужно удалить камень из точки 8. Он достаточно удален от 0, но недостаточно от 12. Итого четыре удаления, и в конечном итоге остаются камни в точках 0 и 12.

Итак, чтобы получить минимальное расстояние прыжка не меньше 6, требуется четыре удаления, но разрешено удалить всего два камня. В таком случае 6 не является допустимым решением. Оно слишком велико. Будет ли допустимым решением 3? То есть можно ли добиться минимального расстояния прыжка 3, удалив всего два камня? Посмотрим.

Камень в точке 0 остается, а камень в точке 2 нужно убрать. Это первое удаление, оставляющее камни в позициях 0, 4, 5, 8 и 12. Камень в точке 4 может остаться: он удален от 0 более чем на 3. А вот камень в точке 5 нужно убрать, потому что он слишком близок к камню в точке 4. Это второе удаление, после которого остаются заняты точки 0, 4, 8 и 12. Камень в точке 8 не мешает: он достаточно удален от камней в 4 и 12. Значит, здесь мы закончили: для получения минимального расстояния прыжка 3 потребовалось два удаления. Таким образом, 3 является допустимым решением.

Здесь для проверки допустимости решения мы возвращаемся к жадному алгоритму. Правило получается следующее: поочередно рассматривать каждый камень и удалять его, если он оказывается слишком близок к предыдущему оставленному камню. Для предпоследнего камня надо выполнить проверку также относительно его правого соседа. Затем подсчитывается число удаленных камней. Этот результат сообщит нам, является ли предложенное минимальное расстояние прыжка допустимым с учетом количества камней, которое разрешено удалить (уточню, что это формулировка жадного алгоритма для проверки допустимости заданного расстояния прыжка, а не поиска оптимального решения). Реализация этого алгоритма представлена в листинге 6.4.

#### Листинг 6.4. Проверка допустимости расстояния прыжка

```
int can_make_min_distance(int distance, int rocks[], int num_rocks,
                          int num_remove, int length) {
    int i;
    int removed = 0, prev_rock_location = 0, cur_rock_location;
    if (length < distance)
        return 0;
    for (i = 0; i < num_rocks; i++) {
        cur_rock_location = rocks[i];
        if (cur_rock_location - prev_rock_location < distance) ❶
            removed++;
        else
            prev_rock_location = cur_rock_location;
    }
    if (length - prev_rock_location < distance) ❷
        removed++;
    return removed <= num_remove;
}
```

У этой функции пять параметров:

- **distance** — минимальное расстояние прыжка, которое проверяется.
- **rocks** — массив, содержащий координату каждого камня, кроме расположенных в начале и конце реки.
- **num\_rocks** — количество камней в массиве **rocks**.
- **num\_remove** — количество камней, которые можно убрать.
- **length** — длина реки.

Эта функция возвращает 1 (true), если **distance** оказывается допустимым решением, и 0, если нет.

Переменная **prev\_rock\_location** отслеживает координату последнего нетронутого камня. Внутри цикла **for** переменная **cur\_rock\_location** содержит координату камня, рассматриваемого в данный момент. Далее идет проверка, определяющая, нужно ли оставить или удалить текущий камень ❶. Если он находится слишком близко к предыдущему, то мы его удаляем и увеличиваем количество удалений на один. В противном случае текущий камень остается, и соответствующим образом обновляется **prev\_rock\_location**.

По завершении цикла мы получаем количество камней, которое нужно удалить. Хотя не совсем. Нам все еще нужно проверить, не находится ли предпоследний камень слишком близко к концу реки ❷. Если да, то он удаляется (не беспокойтесь о возможном удалении камня в точке 0. Если действительно удалить все камни, то **prev\_rock\_location** окажется равным 0. Однако **length - 0 < distance** не может быть верно. В противном случае сработала бы инструкция **if** еще в начале функции).

Итак, условие расстояния прыжка между камнями соблюдено, причем мы удалили заданное число камней. Можно ли добиться еще лучшего результата? Жадный алгоритм прекрасен... но круг опять замыкается. Последний раз, а именно в разделе «Жадная идея», этот алгоритм оказался неверен. Не стоит вестись на пару успешных примеров, где все, казалось бы, работает. Не позволяйте мне вас заболтать и заставить поверить, что все в порядке. Поэтому, прежде чем продолжать, необходимо доказать верность этого жадного алгоритма. Я покажу, что он удаляет минимальное число камней, необходимое для достижения длины прыжка не менее  $d$ . Я предположу, что  $d$  может быть не больше длины реки. В противном случае жадный алгоритм тут же верно определяет, что длина прыжка  $d$  недопустима.

Для каждого камня при движении слева направо алгоритм решает, оставить его или убрать. Нашей целью будет показать, что он шаг в шаг соответствует оптимальному

решению. Когда жадный алгоритм решает удалить камень, мы покажем, что в оптимальном решении этот камень тоже удаляется. Когда он решает оставить камень, мы покажем, что в оптимальном решении он также остается. Если жадный алгоритм делает в точности то, что и в оптимальном решении, то получаемый результат должен быть верен.

В этом примере для каждого камня будут четыре возможных случая: жадное и оптимальное решения удаляют камень; жадное и оптимальное решения оставляют камень; жадное решение удаляет, а оптимальное сохраняет; жадное сохраняет, а оптимальное удаляет. Нужно показать, что третий и четвертый случаи невозможны.

Прежде чем перейти к этим вариантам, рассмотрим удаление двух камней из точек 0, 2, 4, 5, 8 и 12. Если спрашивается, можно ли достичь длины прыжка не менее 3, то мы видели, что жадный алгоритм удаляет камни в точках 2 и 5, оставляя 0, 4, 8 и 12. Значит, можно ожидать, что в оптимальном решении будут удалены те же два камня. Несмотря на то что это оптимально, другим возможным решением будет удалить камни в точках 2 и 4, оставив камни в 0, 5, 8 и 12. Это другой способ получить расстояние не менее 3 удалением двух камней, который столь же хорош, что и предложенный жадным алгоритмом. Вместо того чтобы сопоставлять с конкретным оптимальным решением, мы будем сравнивать только результат. Нам не важно, с каким именно вариантом решения в итоге совпадет жадный алгоритм, поскольку все оптимальные решения равны с точки зрения своей верности.

Итак, у нас есть некое оптимальное решение  $S$ , которое нужно сопоставить с жадным. Жадное начинает выполнение, и в течение какого-то времени расхождений нет: оно делает то же, что и  $S$ . Как минимум жадное правильным образом поступает с камнем в точке 0: он должен остаться в любом случае.

Жадный алгоритм просматривает камни слева направо, совершая верные действия, то есть оставляя или удаляя камни в соответствии с  $S$ . И тут вдруг их действия в отношении одного из камней расходятся. Мы рассматриваем *первый* камень, в отношении которого жадное и  $S$  решения не сошлись.

**Жадное решение удаляет камень, а оптимальное оставляет.** Жадный алгоритм удаляет камень, только если тот слишком близок к другому камню. Если он удаляет камень, потому что тот находится на расстоянии менее  $d$  от камня слева, то в  $S$  он тоже должен быть удален. А раз это первое разногласие, значит,  $S$  содержит слева в точности те же камни, что и жадное решение. Тогда если в  $S$  камень не удалялся, то на расстоянии менее  $d$  в этом решении должны находиться два камня. Однако этого произойти не может, так как  $S$  является оптимальным (и обязательно допустимым) решением, в котором все расстояния между камнями равны как минимум  $d$ . Значит,  $S$  действительно удаляет камень, соглашаясь с жадным вариантом. По той же



логике, если жадное решение удалит камень из-за его излишней близости к концу реки, то и в  $S$  этот камень тоже должен быть удален.

**Жадное решение оставляет камень, а оптимальное удаляет.** В данном случае нам не удастся пошагово сопоставить действия жадного решения и  $S$ , но можно сформулировать новое оптимальное решение  $U$ , которое оставит этот камень. Пусть  $r$  будет камнем, который жадное решение оставляет, а  $S$  удаляет. Получается новый набор камней  $T$ , в котором находятся те же камни, что и в  $S$ , плюс камень  $r$ . Следовательно, в  $T$  удалено на один камень меньше, чем в  $S$ . Из-за этого  $T$  не может быть допустимым решением. Если бы оно таковым было, то оказалось бы лучше, чем  $S$ , опровергая оптимальность  $S$ . Поскольку единственное отличие между  $S$  и  $T$  в том, что  $T$  содержит камень  $r$ , именно наличие этого камня и обуславливает недопустимость  $T$ . Следовательно, в  $T$  камень  $r$  должен находиться от расположенного справа камня  $r_2$  на расстоянии меньше  $d$ .

Известно, что  $r_2$  не может быть камнем в конце реки, потому что тогда жадный алгоритм не сохранил бы  $r$  (так как  $r$  оказался бы слишком близок к концу реки). Значит,  $r_2$  — это камень, который можно удалить.

Теперь представим еще один набор камней  $U$ , в котором есть те же камни, что и в  $T$ , за исключением  $r_2$ . Можно сказать, что в  $U$  есть то же число камней, что и в  $S$ : мы добавили в  $S$  камень  $r$  для получения  $T$  и удалили камень  $r_2$  для получения  $U$ . Кроме того, в  $U$  нет камней, разделенных расстоянием меньше  $d$ , потому что в этот набор не входит нарушающий порядок  $r_2$ . То есть  $U$ , как и  $S$ , оказывается оптимальным решением. Решающим фактором является то, что  $U$  содержит камень  $r$ . Значит, жадный алгоритм соглашается с оптимальным решением  $U$  о включении  $r$ .

Прежде чем продолжать, давайте выполним проверку на допустимость. Ниже представлен тест для примера, который использовался в этом разделе:

```
int main(void) {
    int rocks[4] = {2, 4, 5, 8};
    printf("%d\n", can_make_min_distance(6, rocks, 4, 2, 12));
    return 0;
}
```

Код выше спрашивает, можно ли достичь минимальной длины прыжка в шесть единиц, удалив два камня. Ответ «нет», значит, в выводе должен отобразиться 0 (ложно). Измените первый параметр с 6 на 3, и теперь вопрос ставится о возможности получения минимальной длины в три единицы. Запустите программу, и на этот раз в выводе отобразится 1 (верно).

Превосходно: теперь у нас есть способ проверки допустимости. Время задействовать двоичный поиск для нахождения оптимального решения.

## Поиск решения

Для использования двоичного поиска мы адаптируем код из листинга 6.3. В «Кормлении муравьев» требовалось достичь точности выше четырех знаков после запятой. Здесь же нам нужно оптимизировать количество камней, выражаемое целочисленным значением. Так что вычисления будут останавливаться при целых значениях `high` и `low`.

### Листинг 6.5. Поиск оптимального решения (с ошибкой!)

```
void solve(int rocks[], int num_rocks, //ошибка!
           int num_remove, int length) {
    int low, high, mid;
    low = 0;
    high = length;
    while (high - low > 1) {
        mid = (low + high) / 2;
        if (can_make_min_distance(mid, rocks, num_rocks, num_remove, length)) ❶
            low = mid; ❷
        else
            high = mid; ❸
    }
    printf("%d\n", high);
}
```

В каждой итерации мы вычисляем среднюю точку диапазона `mid` и используем вспомогательную функцию для проверки ее допустимости ❶.

Если `mid` оказывается допустимой, то все значения меньше `mid` также допустимы и `low` обновляется, отсекая нижнюю половину диапазона ❷. Обратите внимание, что в листинге 6.3 допустимыми являются все значения больше `mid` (в случае ее допустимости), в результате чего отсекается верхняя половина диапазона, а не нижняя.

Если же `mid` оказывается недопустимым, то все значения больше него также оказываются недопустимыми, в связи с чем обновляется `high`, отсекая верхнюю половину диапазона ❸.

К сожалению, этот вариант двоичного поиска ошибочен. Чтобы понять, почему, проверьте его на следующем тестовом примере:

```
12 4 2
2
4
5
8
```

На выходе должно вернуться 5, но оптимальное решение равно 4.

О! Я знаю, что нужно сделать. Мы изменим вызов `printf`, чтобы выводить не `high`, а `low`. При завершении цикла `low` будет на единицу меньше, чем `high`, что приведет к выводу не 5, а 4. Обновленный код приведен в листинге 6.6.

**Листинг 6.6.** Поиск оптимального решения (все еще с ошибкой!)

```
void solve(int rocks[], int num_rocks, //ошибка!
           int num_remove, int length) {
    int low, high, mid;
    low = 0;
    high = length;
    while (high - low > 1) {
        mid = (low + high) / 2;
        if (can_make_min_distance(mid, rocks, num_rocks, num_remove, length))
            low = mid;
        else
            high = mid;
    }
    printf("%d\n", low);
}
```

Мы успешно подстроили код под проблематичный тестовый пример, но теперь ошибку вызовет другой набор исходных данных:

```
12 0 0
```

Это валидный пример, хотя и немного странный: протяженность реки равна 12, а камней между первым и последним камнем нет. Минимальное расстояние прыжка в этом случае равно 12, но предложенный нами алгоритм двоичного поиска возвращает здесь 11. Очередной промах на единицу.

Двоичный поиск знаменит сложностью правильной реализации. Нужно ли использовать `>` или `>=`? Должно ли это быть значение `mid` или `mid+1`? Интересно ли нас `low + high` или `low + high + 1`? Если вы продолжите работу с задачами двоичного поиска, то в конечном итоге во всем этом разберетесь. Лично мне не известны другие алгоритмы с таким же потенциалом к возникновению ошибок.

Постараемся быть внимательнее при следующей попытке. Рассмотрим вариант, в котором известно, что `low` и все меньшие значения являются допустимыми, а `high` и все большие значения — недопустимыми. Подобное утверждение называется *инвариантом* и по ходу выполнения кода всегда сохраняет истинность.

Когда цикл завершится, `low` будет на один меньше `high`. Если нам удастся поддерживать инвариант, то мы будем знать, что `low` является допустимым значением, а значения больше `low` таковыми быть не могут: `high` идет следом, и, согласно инварианту, его значение является недопустимым. Значит, `low` будет максимально допустимым, и вывести нужно именно `low`.

Однако при всем этом предполагается, что мы можем сделать инвариант, который будет истинным в начале кода, и сохранить истинность до его завершения.

Начнем с кода над циклом. Он *не* обязательно делает инвариант истинным:

```
low = 0;
high = length;
```

Является ли нулевое значение `low` допустимым? Конечно! Минимальное расстояние прыжка, равное нулю, всегда достижимо. Является ли `high` недопустимым? Да, оно может таковым быть, но что, если после удаления разрешенного количества камней нам удастся пропрыгать вдоль всей реки? Тогда значение `length` окажется допустимым и наш инвариант сломается. Вот инициализация получше:

```
low = 0;
high = length + 1;
```

Теперь значение `high` точно недопустимо: нельзя прыгнуть на длину `length + 1`, когда протяженность всей реки равна `length`.

Далее нужно выяснить, что делать с этими двумя возможностями в цикле. Если `mid` окажется допустимо, то можно установить `low = mid`. Такой инвариант в порядке, потому что `low` и все, что левее, является допустимым. Если же `mid` недопустимо, то можно установить `high = mid`. Такой инвариант тоже окажется в порядке, потому что `high` и все правее него значения являются недопустимыми. Таким образом, в обоих случаях истинность инварианта сохраняется.

Теперь мы видим, что никакой элемент кода не может повлиять на инвариант, а значит, можно спокойно выводить `low` по завершении цикла. Верный код дан в листинге 6.7.

#### Листинг 6.7. Поиск оптимального решения

```
void solve(int rocks[], int num_rocks,
           int num_remove, int length) {
    int low, high, mid;
    low = 0;
    high = length + 1;
    while (high - low > 1) {
        mid = (low + high) / 2;
        if (can_make_min_distance(mid, rocks, num_rocks, num_remove, length))
            low = mid;
        else
            high = mid;
    }
    printf("%d\n", low);
}
```

## Считывание входных данных

Мы практически достигли цели. Осталось только считать входные данные и вызвать `solve`. В листинге 6.8 приведен соответствующий код.

**Листинг 6.8.** Функция `main` для считывания входных данных

```
#define MAX_ROCKS 50000

int compare(const void *v1, const void *v2) {
    int num1 = *(const int *)v1;
    int num2 = *(const int *)v2;
    return num1 - num2;
}

int main(void) {
    static int rocks[MAX_ROCKS];
    int length, num_rocks, num_remove, i;
    scanf("%d%d%d", &length, &num_rocks, &num_remove);
    for (i = 0; i < num_rocks; i++)
        scanf("%d", &rocks[i]);
    qsort(rocks, num_rocks, sizeof(int), compare); ❶
    solve(rocks, num_rocks, num_remove, length);
    return 0;
}
```

Мы анализировали эту задачу, рассматривая координаты камней слева направо, то есть от наименьшего значения к наибольшему. Однако во входных данных они могут быть представлены в любом порядке, так как в условии задачи последовательность не гарантируется.

Как-то давно, в главе 2, мы упорядочивали узлы с помощью `qsort`, когда решали задачу «Расстояние до потомка». Упорядочивание камней — сравнительно простой процесс. Функция сравнения `compare` получает указатели на два целых числа и возвращает результат вычитания второго из первого. Если первое число меньше второго, то получается отрицательный результат, если они равны — 0, а если первое больше второго — положительное значение. Для упорядочивания камней совместно с функцией сравнения используется `qsort` ❶. Затем вызывается `solve` с массивом упорядоченных камней.

Если отправить это решение на проверку, то все тестовые примеры будут успешно пройдены.

## Задача 3. Качество жизни

До сих пор в этой главе для проверки допустимости мы использовали два подхода: рекурсивный обход дерева и жадный алгоритм. Теперь же мы рассмотрим пример,

в котором для эффективной проверки допустимости используются идеи динамического программирования (глава 3).

Это первая задача книги, в которой не происходит считывание из `stdin` или запись в `stdout`. Мы напишем функцию с именем, указанным судебским сайтом, а вместо стандартного потока ввода используем массив, переданный тем же ресурсом. Вместо `stdout` будет возвращаться значение из нашей функции. Это все довольно удобно, так как нам не придется использовать `scanf` и `printf`.

В то же время это будет нашим первым испытанием на уровне чемпионата мира по программированию (IOI 2010).

Перед нами задача с платформы DMOJ под номером `ioi10p3`.

### Условие

Город состоит из прямоугольной сетки кварталов. Каждый квартал обозначается координатами строки и столбца. Всего есть  $r$  строк, пронумерованных от 0 до  $r - 1$  сверху вниз, и  $c$  столбцов, пронумерованных от 0 до  $c - 1$  слева направо.

Каждому квадрату присвоен индивидуальный *ранг качества* в диапазоне от 1 до  $rc$ . К примеру, если дано семь строк и семь столбцов, то рангами кварталов будут числа между 1 и 49. Пример схемы города дан в табл. 6.1.

**Таблица 6.1.** Пример города

	0	1	2	3	4	5	6
0	48	16	15	45	40	28	8
1	20	11	36	19	24	6	33
2	22	39	30	7	9	1	18
3	14	35	2	13	31	12	46
4	32	37	21	3	41	23	29
5	42	49	38	10	17	47	5
6	43	4	34	25	26	27	44

*Средний ранг качества* прямоугольника — это ранг качества, относительно которого другие ранги делятся на две равные половины — с меньшими и большими значениями. К примеру, рассмотрим прямоугольник размером пять строк на три столбца ( $5 \times 3$ ) в верхнем левом углу табл. 6.1. Он содержит 15 рангов качества: 48, 16, 15, 20, 11, 36, 22, 39, 30, 14, 35, 2, 32, 37 и 21. Средним рангом здесь выступает 22, потому что семь чисел меньше 22, а другие семь — больше.

Нам будут даны целые числа  $h$  и  $w$ , указывающие высоту (количество строк) и ширину (количество столбцов) прямоугольников. Задача — определить минимальный средний ранг качества любого прямоугольника с  $h$  строк и  $w$  столбцов.

Предположим, что  $h$  равна 5, а  $w$  равна 3. Тогда для города в табл. 6.1 средним рангом качества будет определено значение 13. Прямоугольник со средним рангом качества 13 — этот тот, у которого верхний левый угол образует клетка (1, 3), а правый нижний угол — (5, 5).

### Входные данные

Из стандартного ввода считывать ничего не требуется. Все необходимое поступит от судейского ресурса в виде параметров функции. Вот сама функция:

```
int rectangle(int r, int c, int h, int w, int q[3001][3001])
```

Здесь  $r$  и  $c$  — количество строк и столбцов в схеме соответственно. Аналогичным образом  $h$  и  $w$  — количество строк и столбцов в рассматриваемых прямоугольниках соответственно.  $h$  будет не больше  $r$ , а  $w$  не больше  $c$ . Также гарантируется, что  $h$  и  $w$  являются нечетными (произведение двух нечетных чисел — нечетное число, поэтому и  $hw$ , количество кварталов в рассматриваемом прямоугольнике, будет нечетным. Средний ранг качества в этом случае можно определить точно. Если же у нас будет четное количество рангов качества, например четыре ранга — 2, 6, 4 и 5, то среднее значение придется выбирать между 4 и 5. Автор задачи избавил нас от этого выбора).

Заключительный параметр  $q$  указывает ранг качества кварталов. К примеру,  $q[2][3]$  определяет значение ранга квартала в строке 2, ряду 3. Обратите внимание, что максимальное количество и строк, и столбцов в городе — 3001.

### Выходные данные

Мы не будем ничего выводить в `stdout`. Вместо этого из только что описанной функции `rectangle` мы вернем минимальный средний ранг качества.

Время на решение тестового примера — 10 секунд.

## Упорядочивание прямоугольников

Сложно определить эффективное решение без использования двоичного поиска, но в текущем подразделе мы все равно попробуем сделать это. Заодно попрактикуемся в переборе всех возможных вариантов прямоугольников. А потом перейдем к двоичному поиску.

Для начала нам потребуются пара констант и определение типа:

```
#define MAX_ROWS 3001
#define MAX_COLS 3001

typedef int board[MAX_ROWS][MAX_COLS];
```

Во многом аналогично главе 4 мы будем использовать двумерный массив `board`.

Предположим, что даны координаты верхнего левого и правого нижнего углов прямоугольника и нужно определить средний ранг качества его кварталов. Как это сделать?

Здесь поможет упорядочивание. Нужно просто отсортировать ранги качества от меньших к большим и затем выбрать элемент со средним индексом. Рассмотрим, к примеру, все те же 15 рангов качества: 48, 16, 15, 20, 11, 36, 22, 39, 30, 14, 35, 2, 32, 37 и 21. Если их упорядочить, то получится: 2, 11, 14, 15, 16, 20, 21, 22, 30, 32, 35, 36, 37, 39 и 48. Всего здесь 15 рангов, значит, нужно взять восьмой (22), который и будет средним.

Есть и более быстрые алгоритмы для непосредственного нахождения среднего значения без необходимости упорядочивания. Сортировка — алгоритм, которому для нахождения среднего значения требуется  $O(n \log n)$  времени. Существует и более сложный алгоритм для нахождения среднего со временем  $O(n)$ . Но здесь мы не станем его использовать. Реализуемый в этом подразделе процесс будет настолько медленным, что никакое улучшение алгоритма поиска среднего значения все равно не поможет.

В листинге 6.9 приведен код для нахождения среднего в заданном прямоугольнике.

### Листинг 6.9. Поиск среднего ранга заданного прямоугольника

```
int compare(const void *v1, const void *v2) {
    int num1 = *(const int *)v1;
    int num2 = *(const int *)v2;
    return num1 - num2;
}

int median(int top_row, int left_col, int bottom_row, int right_col,
           board q) {
    static int cur_rectangle[MAX_ROWS * MAX_COLS];
```



```

int i, j, num_cur_rectangle;
num_cur_rectangle = 0;
for (i = top_row; i <= bottom_row; i++)
    for (j = left_col; j <= right_col; j++) {
        cur_rectangle[num_cur_rectangle] = q[i][j];
        num_cur_rectangle++;
    }
qsort(cur_rectangle, num_cur_rectangle, sizeof(int), compare); ❶
return cur_rectangle[num_cur_rectangle / 2];
}

```

Первые четыре параметра `median` очерчивают прямоугольник, указывая координаты его левого верхнего и правого нижнего углов. Заключительный параметр `q` содержит ранги качества. Для хранения рангов качества прямоугольника используется одномерный массив `cur_rectangle`. Вложенные циклы `for` проходят через каждую ячейку (квартал) прямоугольника и добавляют соответствующий ранг качества в `cur_rectangle`. После сбора всех рангов можно отправлять их в `qsort` ❶. Далее нам будет в точности известно, где находится среднее значение — в середине массива, — и мы просто его возвращаем.

Вооружившись этой функцией, можно перебирать все варианты прямоугольника, и в конце определить тот, чей средний ранг качества окажется наименьшим. Код дан в листинге 6.10.

**Листинг 6.10.** Поиск наименьшего среднего значения среди возможных прямоугольников

```

int rectangle(int r, int c, int h, int w, board q) {
    int top_row, left_col, bottom_row, right_col;
    int best = r * c + 1; ❶
    int result;
    for (top_row = 0; top_row < r - h + 1; top_row++)
        for (left_col = 0; left_col < c - w + 1; left_col++) {
            bottom_row = top_row + h - 1; ❷
            right_col = left_col + w - 1; ❸
            result = median(top_row, left_col, bottom_row, right_col, q); ❹
            if (result < best)
                best = result;
        }
    return best;
}

```

В переменной `best` сохраняется наилучшее (то есть наименьшее) среднее значение, найденное до сих пор. Эту переменную мы изначально устанавливаем с большим значением, которое превосходит среднее любого возможного прямоугольника ❶. Невозможно, чтобы среднее прямоугольника равнялось  $r * c + 1$ , это бы означало, что половина его рангов качества больше  $r * c$ . Но, согласно условию задачи, *никакие* ранги качества не могут быть больше  $r * c$ . Вложенные циклы `for`

рассматривают все возможные координаты верхнего левого угла прямоугольника. Так мы получаем верхнюю строку и левый столбец, но для вызова `median` нам также нужна нижняя строка и правый столбец. Чтобы вычислить координату нижней строки, мы берем номер верхней строки, прибавляем  $h$  (количество строк в рассматриваемых прямоугольниках), а затем вычитаем 1 ❷. Здесь очень просто допустить промах на единицу, но вычитание единицы необходимо. Если верхняя строка имеет номер 4, а  $h$  равна 2, то нам нужно, чтобы номер нижней строки был равен  $4 + 2 - 1 = 5$ . Если же сделать этот номер равным  $4 + 2 = 6$ , то у нас получится прямоугольник с тремя строками вместо требуемых двух. Аналогичным образом вычисляется номер правого столбца ❸. Получив все четыре координаты, мы вызываем `median` для вычисления среднего значения ранга прямоугольника ❹. Если в итоге обнаруживается лучшее значение, то оставшаяся часть кода обновляет `best`.

На этом текущее решение закончено. Здесь нет функции `main`, потому что судья вызывает `rectangle` напрямую. Но ее отсутствие означает, что протестировать код на собственном компьютере не получится. Для тестирования можно все же ввести функцию `main`, но затем удалить ее перед отправкой решения на проверку.

Вот пример `main`:

```
int main(void) {
    static board q = {{48, 16, 15, 45, 40, 28, 8},
                      {20, 11, 36, 19, 24, 6, 33},
                      {22, 39, 30, 7, 9, 1, 18},
                      {14, 35, 2, 13, 31, 12, 46},
                      {32, 37, 21, 3, 41, 23, 29},
                      {42, 49, 38, 10, 17, 47, 5},
                      {43, 4, 34, 25, 26, 27, 44}};

    int result = rectangle(7, 7, 5, 3, q);
    printf("%d\n", result);
    return 0;
}
```

Можете отправлять наше решение, исключив из него функцию `main`. Оно может пройти несколько тестовых примеров, но превысит временной лимит в остальных.

Чтобы понять, почему наш код столь медленный, рассмотрим пример, в котором  $r$  и  $s$  представлены числом  $m$ . Для демонстрации наихудшего случая мы возьмем  $h$  и  $w$ , равные  $m/2$  (нам не нужно, чтобы прямоугольники получались излишне большими, иначе их будет немного. Нам также не нужно, чтобы они были слишком маленькими, так как тогда каждый будет легко обработать). Самая медленная часть функции `median` — вызов `qsort`. Она получает массив с  $m/2 \times m/2 = m^2/4$  значений. Для обработки массива из  $n$  значений `qsort` совершает  $n \log n$  шагов. Замена  $n$  на  $m^2/4$  дает оценку времени  $(m^2/4) \log(m^2/4) = O(m^2 \log m)$ . Итак, выполнение происходит

уже медленнее, чем в квадратичном случае, — а мы ведь вычислили среднее значение только для одного прямоугольника. Функция `rectangle` вызывает `median` всего  $m^2/4$  раз, значит, общее время выполнения составляет  $O(m^4 \log m)$ . При таком числе операций данное решение подходит только для примеров с очень малым количеством данных.

В нашем решении присутствуют два узких места. Первое — это упорядочивание каждого прямоугольника. Второе — это создание массива `cur_rectangle` с нуля для каждого прямоугольника. Использование двоичного поиска решает первую проблему, а динамическое программирование — вторую.

## Двоичный поиск

Почему нам стоит рассчитывать на то, что двоичный поиск обеспечит в данном случае прирост скорости? Во-первых, в предыдущем подразделе я показал, что прямой поиск очень затратен. Подход, который опирался на сортировку, оказался медленнее алгоритма  $m^4$ . Во-вторых, это еще один пример задачи, где сначала идут все недопустимые решения, а за ними уже допустимые. Предположим, что нет такого прямоугольника, чей средний ранг качества имеет значение менее 6. Тогда не будет смысла искать прямоугольники со средним качеством 5, 4 или любым меньшим значением. И наоборот, предположим, я скажу, что есть прямоугольник со средним рангом качества 5 или меньше. Теперь не будет смысла искать прямоугольники со средним рангом качества 6, 7 или любым другим, чье значение выше 5.

Так задается область двоичного поиска.

В задаче с прыжками вдоль реки небольшие значения были допустимыми, а большие недопустимыми. Здесь же у нас обратный случай: небольшие значения оказываются недопустимыми, а большие допустимыми. Следовательно, нам придется изменить инвариант, сменив расположение допустимых и недопустимых частей пространства решений.

И вот какой инвариант мы используем: `low` и все, что меньше, будет недопустимо; `high` и все, что больше, — допустимо. Таким образом, по завершении нужно возвращать `high`, поскольку таково будет наименьшее допустимое значение. В остальном код из листинга 6.11 очень похож на код из листинга 6.7.

### Листинг 6.11. Поиск оптимального решения

```
int rectangle(int r, int c, int h, int w, board q) {
    int low, high, mid;
    low = 0;
    high = r * c + 1;
```

```

while (high - low > 1) {
    mid = (low + high) / 2;
    if (can_make_quality(mid, r, c, h, w, q))
        high = mid;
    else
        low = mid;
}
return high;
}

```

Для завершения нужно проверить допустимость с помощью `can_make_quality`.

## Проверка допустимости

Вот функция проверки допустимости:

```
int can_make_quality(int quality, int r, int c, int h, int w, board q)
```

В разделе «Упорядочивание прямоугольников» нас обременяла необходимость вычисления среднего ранга качества каждого прямоугольника. Теперь же такой проблемы не будет: нам достаточно определять, не превосходит ли среднее значение некоторого прямоугольника пороговое значение ранга `quality`.

Это более простая задача, для которой этап сортировки необязателен. Конкретные значения нам больше не важны, принципиально только соотношение между каждым значением и `quality`. Поэтому мы заменим все меньшие или равные `quality` значения на  $-1$ , а все большие — на  $1$ . Затем мы сложим эти  $-1$  и  $1$  для заданного прямоугольника. Если в итоге значений  $-1$  получится столько же или больше, чем значений  $1$  (то есть относительно `quality` число меньших значений будет превосходить количество больших), то их сумма будет нулевой или отрицательной и мы сделаем вывод, что средний ранг качества данного прямоугольника равен `quality` или меньше него.

Рассмотрим пример, в котором снова возьмем 15 рангов для прямоугольника  $5 \times 3$  из верхнего левого угла табл. 6.1: 48, 16, 15, 20, 11, 36, 22, 39, 30, 14, 35, 2, 32, 37 и 21. Сравним средний ранг качества этого прямоугольника с 16. Для этого возьмем каждое значение и, если оно меньше или равно 16, заменим его на  $-1$  ❶, а если больше 16, то на  $1$ . Получатся следующие значения: 1,  $-1$ ,  $-1$ , 1,  $-1$ , 1, 1, 1, 1,  $-1$ , 1,  $-1$ , 1, 1 и 1. Их сложение даст 5. Это означает, что значений, превосходящих 16, на пять больше, чем меньших или равных этому числу. Это, в свою очередь, говорит о том, что среднее значение 16 или меньше для данного прямоугольника невозможно. Если бы мы хотели узнать, является ли допустимым среднее значение 30, то также узнали бы об этом, заменив числа на  $-1$  и  $1$ : 1,  $-1$ ,  $-1$ ,  $-1$ ,  $-1$ , 1,  $-1$ , 1,  $-1$ ,  $-1$ , 1,  $-1$ , 1,

1 и  $-1$ . Сложение этих значений даст  $-3$ . Значит, 30 является допустимым средним. Важно отметить, что решение о допустимости/недопустимости принимается без предварительного упорядочивания.

Нам нужно перебрать каждый прямоугольник, проверяя, содержит ли он средний ранг качества, равный `quality` или меньше него. Этот процесс реализуется в коде листинга 6.12.

**Листинг 6.12.** Проверка допустимости `quality`

```
int can_make_quality(int quality, int r, int c, int h, int w, board q) {  
    static int zero_one[MAX_ROWS][MAX_COLS]; ❶  
    int i, j;  
    int top_row, left_col, bottom_row, right_col;  
    int total;  
  
    for (i = 0; i < r; i++)  
        for (j = 0; j < c; j++)  
            if (q[i][j] <= quality) ❷  
                zero_one[i][j] = -1;  
            else  
                zero_one[i][j] = 1;  
  
    for (top_row = 0; top_row < r - h + 1; top_row++)  
        for (left_col = 0; left_col < c - w + 1; left_col++) {  
            bottom_row = top_row + h - 1;  
            right_col = left_col + w - 1;  
            total = 0;  
            for (i = top_row; i <= bottom_row; i++)  
                for (j = left_col; j <= right_col; j++)  
                    total = total + zero_one[i][j]; ❸  
            if (total <= 0)  
                return 1;  
        }  
    return 0;  
}
```

Нельзя просто заместить массив `q` значениями  $-1$  и  $1$ , потому что тогда мы не сможем использовать исходные ранги качества для последующей проверки других значений `quality`. Следовательно, здесь для хранения  $-1$  и  $1$  создается новый массив ❶. Обратите внимание, что он заполняется на основании того, является ли каждое значение меньшим или равным ( $-1$ ) либо большим ( $1$ ), чем пороговый параметр `quality` ❷.

Далее мы проходим по каждому прямоугольнику так же, как делали в листинге 6.10. При этом суммируются все его  $-1$  и  $1$  ❸, и возвращается  $1$  (верно), если средний ранг качества оказывается достаточно мал.

Вот мы и избавились от сортировки — искусно, не так ли? Этот прием очень важен для решения нашей задачи, но недостаточен. Если посчитать вложенные циклы, то окажется, что у нас их целых четыре.

В конце раздела «Упорядочивание прямоугольников» мы обнаружили, что решение без двоичного поиска оказалось очень медленным  $O(m^4 \log m)$ , где  $m$  — это количество строк или столбцов в таблице. Здесь же оценка скорости проверки допустимости все еще равна  $m^4$ . Умножьте ее на логарифмический множитель, характеризующий двоичный поиск, и возникнет вопрос: а добились ли мы хоть какого-нибудь прогресса?

Но мы добились! Просто он скрыт за слишком большим числом вложенных циклов, производящих чересчур много вычислений. Оставшуюся часть пути нам поможет пройти динамическое программирование.

### Ускоренная проверка допустимости

Предположим, что дана таблица 6.1 и требуется выяснить, содержит ли какой-либо из прямоугольников  $5 \times 3$  средний ранг качества 16 или менее. Изменив все значения, которые меньше или равны 16, на  $-1$ , а все значения больше 16 — на 1, получим табл. 6.2.

**Таблица 6.2.** Город после замены рангов качества

	0	1	2	3	4	5	6
0	1	-1	-1	1	1	1	-1
1	1	-1	1	1	1	-1	1
2	1	1	1	-1	-1	-1	1
3	-1	1	-1	-1	1	-1	1
4	1	1	1	-1	1	1	1
5	1	1	1	-1	1	1	-1
6	1	-1	1	1	1	1	1

Можно начать с суммирования элементов прямоугольника  $5 \times 3$  с координатами верхнего левого угла  $(0, 0)$ . Как мы видели в разделе «Проверка допустимости»,

сумма значений этого прямоугольника равна 5. Далее вычислим сумму элементов прямоугольника  $5 \times 3$  с координатами верхнего левого угла  $(0, 1)$ . В предыдущем подразделе мы бы в данном случае сложили все 15 чисел. Однако тогда мы не сможем воспользоваться результатами вычислений суммы первого прямоугольника. Действительно, во втором прямоугольнике есть 10 общих значений с первым. Нам нужно избежать подобного дублирования работы как для этого, так и для всех остальных прямоугольников.

Исключение повторения работы здесь подразумевает эффективное выполнение так называемого *запроса суммы двумерного диапазона*. Одномерный случай реализуется по тем же принципам, но проще, поэтому сперва мы вкратце изучим его, а потом уже вернемся к завершению задачи «Качество жизни» (половина главы 7 будет посвящена запросам по диапазону, так что, как говорится, «оставьтесь с нами!»).

### Запрос суммы одномерного диапазона

Рассмотрим одномерный массив:

Индекс	0	1	2	3	4	5	6
Значение	6	2	15	9	12	4	11

Если требуется найти сумму элементов массива от индекса 2 до индекса 5, то можно напрямую суммировать значения в этом диапазоне:  $15 + 9 + 12 + 4 = 40$ . Это не очень быстро, и будет совсем печально, если потребуется найти сумму всего массива. Однако если требуется ответить всего на несколько таких запросов, то этот вариант приемлем, и можно каждый раз суммировать соответствующие значения.

Теперь представим, что мы получаем сотни или даже тысячи подобных запросов. Тогда есть смысл единожды проделать предварительную работу, которая позволит отвечать на них быстрее.

Рассмотрим запрос от индекса 2 до 5. Что, если предварительно найти сумму значений от индекса 0 до 5? Она равна 48. Это не нужный нам ответ, которым является 40, однако здесь удобен тот факт, что 48 находится от него недалеко. Ошибочно это значение лишь потому, что включает индексы 0 и 1, которые теперь нужно исключить. Это можно быстро сделать при условии нахождения их суммы, которая равна 8. Если вычесть 8 из 48, то получится нужная сумма 40.

В таком случае нам нужен новый массив, тот, где индекс  $i$  содержит сумму всех значений от индекса 0 до  $i$ . Этот новый массив добавляется в строку *частичной суммы* в следующей таблице:

<b>Индекс</b>	0	1	2	3	4	5	6
<b>Значение</b>	6	2	15	9	12	4	11
<b>Частичная сумма</b>	6	8	23	32	44	48	59

Теперь можно быстро отвечать на любой запрос, используя массив частичных сумм: для вычисления суммы диапазона от индекса  $a$  до  $b$  нужно взять значение в индексе  $b$  и вычесть из него значение в индексе  $a - 1$ . Для диапазона от 2 до 5 получится  $48 - 8 = 40$ , а от 1 до 6 получится  $59 - 6 = 53$ . Теперь на получение ответов независимо от величины диапазона требуется одинаковое количество времени, причем для этого достаточно было сделать всего один предварительный проход по массиву.

### Запрос суммы двумерного диапазона

Вернемся в двумерный мир наших рангов качества. Суммирование элементов каждого прямоугольника — слишком медленный процесс, поэтому мы расширим освоенный на одномерном примере прием до двух измерений. В частности, требуется создать новый массив, в котором индекс  $(i, j)$  является суммой элементов прямоугольника, координаты верхнего левого угла которого —  $(0, 0)$ , а нижнего правого —  $(i, j)$ .

Еще раз взглянем на табл. 6.2.

	<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>
<b>0</b>	1	-1	-1	1	1	1	-1
<b>1</b>	1	-1	1	1	1	-1	1
<b>2</b>	1	1	1	-1	-1	-1	1
<b>3</b>	-1	1	-1	-1	1	-1	1
<b>4</b>	1	1	1	-1	1	1	1
<b>5</b>	1	1	1	-1	1	1	-1
<b>6</b>	1	-1	1	1	1	1	1



Соответствующий ей массив частичных сумм представлен в табл. 6.3 (здесь может показаться несколько странным называть его массивом частичных сумм, но таким образом мы сохраним согласованность терминологии с одномерным случаем).

**Таблица 6.3.** Массив для запросов по двумерному диапазону

	0	1	2	3	4	5	6
0	1	0	-1	0	1	2	1
1	2	0	0	2	4	4	4
2	3	2	3	4	5	4	5
3	2	2	2	2	4	2	4
4	3	4	5	4	7	6	9
5	4	6	8	6	10	10	12
6	5	6	9	8	13	14	17

Для начала убедимся, что правильно понимаем предоставляемые этим массивом данные, а потом уже перейдем к рассмотрению его построения. Значение в строке 4 и столбце 2 указывает сумму чисел в прямоугольнике, координаты верхнего левого угла которого —  $(0, 0)$ , а правого нижнего —  $(4, 2)$ . В разделе «Проверка допустимости» мы видели, что эта сумма равна 5, и, действительно, массив содержит именно такое значение.

Как вычислить значение для ячейки  $(4, 2)$  на основе других ранее вычисленных значений? Нужно начать со значения этой ячейки в табл. 6.2 и прибавить все числа выше, а также левее нее. Это можно сделать, грамотно используя массив из табл. 6.3, как показано в табл. 6.4.

Необходимо начать с 1, затем охватить ячейки, включающие  $x$  (выше), а также ячейки, включающие  $y$  (слева), после чего вычислить сумму. Сумма ячеек, включающих  $x$ , представлена элементом  $(3, 2)$ . Сумма ячеек, включающих  $y$ , аналогичным образом представлена элементом  $(4, 1)$ . Однако их сложение приводит к двойному подсчету ячеек  $xy$  (расположенных выше и левее). Но это не проблема, потому

что элемент (3, 1) содержит сумму именно этих ячеек и компенсировать двойной подсчет можно его вычитанием. Таким образом, получается  $1 + 2 + 4 - 2 = 5$ , что нам и требовалось. При условии продвижения сверху вниз и слева направо можно строить этот массив с помощью всего двух операций сложения и одной операции вычитания для каждой ячейки.

**Таблица 6.4.** Проработка быстрого вычисления заданной суммы

	0	1	2	3	4	5	6
0	ху	ху	х				
1	ху	ху	х				
2	ху	ху	х				
3	ху	ху	х				
4	у	у	1				
5							
6							

Теперь мы знаем, как создавать массив, подобный табл. 6.3, но что это дает? Это позволяет быстро вычислять сумму элементов любого прямоугольника. Предположим, что нам нужна сумма для прямоугольника, координаты левого верхнего угла которого — (1, 3), а правого нижнего — (5, 5). Нельзя просто использовать значение 10 в ячейке (5, 5) табл. 6.3. Помимо суммы ячеек нужного прямоугольника оно включает также элементы, находящиеся выше и слева. Однако так же, как и в одномерном случае, можно вычислить необходимое значение, чтобы оно включало элементы только интересующего нас прямоугольника. Данная операция отражена в табл. 6.5, где ячейки нужного прямоугольника отмечены звездочками.

На этот раз нужно вычесть ячейки, включающие  $x$ , и ячейки, включающие  $y$ . Сумму ячеек  $x$  содержит ячейка (0, 5), а ячеек  $y$  — (5, 2). Но вычитанием обеих этих сумм мы дважды вычтем ячейки  $ху$ , поэтому нужно прибавить значение ячейки (0, 2). Таким образом, получается  $10 - 2 - 8 + (-1) = -1$ , что и является суммой ячеек нужного прямоугольника.

**Таблица 6.5.** Проработка быстрого вычисления суммы элементов прямоугольника

	0	1	2	3	4	5	6
0	ху	ху	ху	х	х	х	
1	у	у	у	*	*	*	
2	у	у	у	*	*	*	
3	у	у	у	*	*	*	
4	у	у	у	*	*	*	
5	у	у	у	*	*	*	
6							

Вот выражение для этого вычисления:

```
sum[bottom_row][right_col] - sum[top_row-1][right_col] -
sum[bottom_row][left_col-1] + sum[top_row-1][left_col-1]
```

Оно и будет использовано в коде, представленном ниже.

### Код суммы двумерного диапазона

Теперь можно объединить все воедино: идею с  $-1$  и  $1$ , построение массива частичных сумм и его использование для быстрого получения сумм прямоугольников. Соответствующий код дан в листинге 6.13.

**Листинг 6.13.** Быстрая проверка допустимости quality

```
int can_make_quality(int quality, int r, int c, int h, int w, board q) {
    static int zero_one[MAX_ROWS][MAX_COLS];
    static int sum[MAX_ROWS + 1][MAX_COLS + 1];
    int i, j;
    int top_row, left_col, bottom_row, right_col;
    int total;

    for (i = 0; i < r; i++) ❶
        for (j = 0; j < c; j++)
            if (q[i][j] <= quality)
                zero_one[i][j] = -1;
```

```

else
    zero_one[i][j] = 1;

for (i = 0; i <= c; i++)
    sum[0][i] = 0;
for (i = 0; i <= r; i++)
    sum[i][0] = 0;
for (i = 1; i <= r; i++) ❶
    for (j = 1; j <= c; j++)
        sum[i][j] = zero_one[i-1][j-1] + sum[i-1][j] +
                    sum[i][j-1] - sum[i-1][j-1] ;

for (top_row = 1; top_row <= r - h + 1; top_row++) ❷
    for (left_col = 1; left_col <= c - w + 1; left_col++) {
        bottom_row = top_row + h - 1;
        right_col = left_col + w - 1;
        total = sum[bottom_row][right_col] - sum[top_row-1][right_col] -
                sum[bottom_row][left_col-1] + sum[top_row-1][left_col-1];
        if (total <= 0)
            return 1;
    }
return 0;
}

```

Вначале создается массив `zero_one` ❶, в точности как это делалось в листинге 6.12. Вторым шагом идет создание массива частичных сумм `sum` ❷. Мы будем использовать индексы, начинающиеся с 1, а не с 0, чтобы не пришлось беспокоиться о возможном выходе за границы массива при обработке ячеек в строке 0 или столбце 0. На третьем шаге массив частичных сумм используется для быстрого вычисления суммы каждого прямоугольника ❸. Обратите внимание, что любой прямоугольник здесь суммируется за одинаковое время. На втором шаге мы выполнили предварительную обработку, которая в дальнейшем позволит узнать сумму для любого прямоугольника, не тратя времени на сложение всех его элементов.

По сравнению с листингом 6.12 мы удалили из циклов `for` два уровня вложения. Следовательно, у нас получился алгоритм  $O(m^2 \log m)$ , который достаточно быстр, чтобы пройти все тестовые примеры. Попробуйте! А потом советую немного передохнуть, так как до конца главы нас ждет еще одна серьезная задача.

## Задача 4. Двери пещеры

Еще одна задача с IOI? Давайте решим ее! Ее особенность состоит в том, что двоичный поиск будет использоваться не для нахождения оптимального решения, а для быстрого выделения нужного элемента. Как и в задаче «Качество жизни», мы не будем выполнять считывание из `stdin` и запись в `stdout`. Вместо этого мы отправим ответ через вызов функций, определенных судьейским ресурсом. По ходу

чтения условия задачи попробуйте разобраться, почему двоичный поиск может использоваться и в этом случае.

Рассмотрим задачу с платформы DMOJ под номером `ioi13p4`.

## Условие

Вы стоите перед входом в длинную узкую пещеру, через которую необходимо пробраться и достичь выхода. На пути нужно пройти через  $n$  дверей: первая — это дверь 0, вторая — дверь 1 и т. д.

Каждая дверь может быть открыта либо закрыта. Через открытые двери можно пройти, через закрытые — нельзя. Таким образом, если двери 0 и 1 открыты, а дверь 2 закрыта, то можно пройти только до двери 2, причем вы не видите состояния следующих за ней дверей.

У входа в пещеру установлена панель с  $n$  рычагов. Как и двери, рычаги пронумерованы, начиная с 0. Каждый из них может быть в верхнем (0) или нижнем (1) положении. При этом каждый рычаг связан с одной дверью, определяя, закрыта она или открыта. Если рычаг установлен в правильное положение, то соответствующая дверь открыта. В противном случае эта дверь закрыта. Вам не известны ни связь рычагов с дверьми, ни правильные положения рычагов. К примеру, возможно, что рычаг 0 связан с дверью 5 и для ее открывания должен находиться в нижнем положении, а рычаг 1 связан с дверью 0 и для ее открывания должен находиться в верхнем положении.

Можно устанавливать рычаги в желаемое положение, а затем проходить в пещеру для определения первой закрытой двери и возвращаться. При этом запаса сил вам хватит максимум на 70 000 таких циклов. Цель — определить верное положение (0 или 1) каждого рычага и связанную с ним дверь.

Нужно написать следующую функцию:

```
void exploreCave(int n)
```

Здесь  $n$  является количеством дверей и рычагов (от 1 до 5000). Для реализации этой функции нужно будет вызвать две функции, предоставленные судьей. Они будут описаны далее.

## Входные данные

Из стандартного ввода мы ничего не считываем. Единственный способ узнать исходные данные — это вызвать предоставленную судьей функцию `tryCombination`. Вот ее сигнатура:

```
int tryCombination(int switch_positions[])
```

Параметр `switch_positions` является массивом с длиной  $n$ , указывающим положение (0 или 1) каждого рычага. То есть `switch_positions[0]` определяет положение рычага 0, `switch_positions[1]` — положение рычага 1 и т. д. Функция `tryCombination` симулирует ситуацию, в которой мы устанавливаем рычаги в `switch_positions` и проходим по пещере до первой закрытой двери. Она возвращает номер первой закрытой двери либо -1, если все двери открыты.

### Выходные данные

В стандартный вывод запись не производится. Вместо этого мы отправляем ответ через вызов функции `answer`, также предоставленной судьейским ресурсом. Вот ее сигнатура:

```
void answer(int switch_positions[], int door_for_switch[])
```

У нас есть только одна попытка: при вызове `answer` наша программа уничтожается, так что правильный ответ следует отправлять с первого раза. Параметр `switch_positions` — массив, содержащий предлагаемые нами положения рычагов в том же формате, что и `tryCombination`. Параметр `door_for_switch` содержит предлагаемые нами связи между рычагами и дверьми: `door_for_switch[0]` указывает дверь, связанную с рычагом 0, `door_for_switch[1]` указывает дверь, связанную с рычагом 1, и т. д.

В данном случае ограничено не время, а количество вызовов `tryCombination`, которых допускается сделать на более 70 000. В случае превышения этого значения программа завершается.

### Решение подзадачи

Автор задачи разделил ее на пять *подзадач*. Пятая подзадача — это задача в общем виде, как я ее здесь представил. Другие подзадачи имеют дополнительные ограничения, упрощая решение.

Мне нравится, когда авторы задач используют подзадачи, особенно если у меня возникают сложности с поиском решения. В таких случаях можно поочередно разделяться с каждой подзадачей, постепенно дорабатывая решение, пока не будет решена вся задача. Более того, если я не могу решить всю задачу, то все равно получаю баллы за подзадачи, с которыми справился.

Первая подзадача состоит в решении варианта, где каждый рычаг  $i$  связан с дверью под номером  $i$ . Это значит, что рычаг 0 связан с дверью 0, рычаг 1 с дверью 1 и т. д. Нам же нужно определить верное положение (0 или 1) для каждого рычага.

Не беспокойтесь, мы не бросим задачу, пока полностью ее не решим. Но начнем с подзадачи 1, чтобы иметь возможность вызывать судейские функции `tryCombination` и `answer`, а уже потом приступим к другим частям основной задачи.

У нас нет доступа к судейским функциям, поэтому не будет возможности компилировать или выполнять код локально (если вы хотите протестировать свою работу локально, то можете запросить в Google «IOI 2013 tasks» и найти тестовые данные с шаблонами для задачи Cave, но для следования текущему материалу этого не требуется). При желании проверить проделанную работу можно отправить код на судейском сайте и узнать оценку. Код для подзадачи 1 приведен в листинге 6.14.

**Листинг 6.14.** Решение подзадачи 1

```
void exploreCave(int n) {
    int switch_positions[n], door_for_switch[n];
    int i, result;
    for (i = 0; i < n; i++) {
        switch_positions[i] = 0; ❶
        door_for_switch[i] = i; ❷
    }

    for (i = 0; i < n; i++) {
        result = tryCombination(switch_positions); ❸
        if (result == i) // дверь i закрыта
            switch_positions[i] = 1; ❹
    }
    answer(switch_positions, door_for_switch); ❺
}
```

Вначале положение каждого рычага устанавливается на 0 ❶, и дверь `i` связывается с рычагом `i` ❷. Положения рычагов мы будем обновлять при необходимости, а вот связь рычаг–дверь в данном случае (согласно ограничениям подзадачи) трогать не потребуется.

Второй цикл `for` перебирает каждый рычаг. Его задача — определить, должен ли текущий рычаг остаться в положении 0 или переключиться в положение 1. Рассмотрим первую итерацию, когда `i` равно 0. Вначале происходит вызов `tryCombination` ❸, которая возвращает номер первой закрытой двери. Если возвращается 0, то рычаг 0 установлен неправильно. Если этот рычаг будет установлен верно, то дверь 0 окажется открыта и `tryCombination` вернет ненулевой номер двери. Если дверь 0 оказывается закрыта, мы изменяем положение рычага 0 с 0 на 1 ❹. В результате дверь 0 открывается, и можно продвигаться к двери 1.

На следующем шаге `i` становится равной 1 и снова вызывается `tryCombination`. Результат 0 в данном случае точно не вернется, потому что код уже проделал работу,

гарантирующую открывание двери 0. Если возвращается 1, это означает, что дверь 1 закрыта и нужно изменить положение рычага 1 с 0 на 1.

Обобщая, можно сказать, что в начале новой итерации цикла все двери до  $i$  (включая  $i - 1$ ) открыты. Если дверь  $i$  закрыта, то мы изменяем положение рычага  $i$  с 0 на 1. Если дверь  $i$  уже открыта, значит, рычаг  $i$  установлен верно.

Закончив второй цикл `for`, мы выяснили верное положение каждого рычага. Этот ответ мы передаем судейскому ресурсу через вызов функции `answer` ⑤.

Я предлагаю вам отправить текущий код на проверку, чтобы убедиться в правильности вызова `tryCombination` и `answer`. После этого мы продолжим.

## **Использование линейного поиска**

Хорошо, что помимо знакомства с задачей мы выполнили первую подзадачу. Дело в том, что наше решение выстраивается на определенной стратегии. Суть этой стратегии проста: выяснить, как открыть каждую дверь, и больше к ней не возвращаться.

В решении первой подзадачи мы начинаем с двери 0, добившись ее открытия. После этого ее рычаг больше не трогаем. Разобравшись с дверью 0, мы занялись открыванием двери 1. Закончив с ней, к ее рычагу мы также больше не возвращаемся. Теперь двери 0 и 1 позади, а на очереди дверь 2. Мы продолжаем идти этим путем, поочередно разбираясь с каждой дверью, пока все они не окажутся открытыми.

В подзадаче 1 мы точно знали, какая дверь с каким рычагом связана. Для определения данного соответствия не требовалось выполнять поиск, но для решения всей задачи поиск нам понадобится, так как неизвестно, какой рычаг за какую дверь отвечает. Мы начнем с закрывания двери 0, после чего займемся перебором рычагов, чтобы открыть ее. Для этого будем последовательно изменять положение рычагов, проверяя, открылась или нет дверь 0. Если нет, то рычаг выбран ошибочно. Если же дверь 0 открылась, значит, связанный с ней рычаг найден. С этого момента дверь 0 остается открытой, и тот же процесс повторяется для двери 1: вначале она закрывается, а затем перебираются оставшиеся рычаги в поиске того, который ее откроет.

Начнем с нового кода `exploreCave`, приведенного в листинге 6.15. Он краток, потому что перекладывает поиск на вспомогательную функцию.

### **Листинг 6.15. Функция `main`**

```
void exploreCave(int n) {  
    int switch_positions[n], door_for_switch[n];
```



```
int i;
for (i = 0; i < n; i++)
    door_for_switch[i] = -1; ❶

for (i = 0; i < n; i++)
    set_a_switch(i, switch_positions, door_for_switch, n); ❷
answer(switch_positions, door_for_switch);
}
```

Как и при решении подзадачи 1, каждый элемент `switch_positions` имеет значение 0 или 1, указывая положение каждого рычага. Массив `door_for_switch` указывает, какая дверь с каким рычагом соединена. Изначально каждый элемент `door_for_switch` инициализируется как -1 ❶, поскольку все связи дверей с рычагами неизвестны. Когда становится известна связанная с рычагом `i` дверь, `door_for_switch[i]` обновляется.

А вот контрольный вопрос: если `door_for_switch[5]` равна 8, то что это значит? Значит ли это, что рычаг 5 связан с дверью 8 или что дверь 5 связана с рычагом 8? Первый вариант! Запомните это, прежде чем продолжать.

Для каждой двери `i` мы вызываем вспомогательную функцию `set_a_switch` ❷. Ее задача — выполнить поиск по рычагам и найти связанный с дверью `i`. Она также определяет, должен рычаг находиться в положении 0 или 1.

Код для `set_a_switch` дан в листинге 6.16.

**Листинг 6.16.** Нахождение и установка рычага для текущей двери с помощью линейного поиска

```
void set_a_switch(int door, int switch_positions[],
                  int door_for_switch[], int n) {
    int i, result;
    int found = 0;

    for (i = 0; i < n; i++)
        if (door_for_switch[i] == -1)
            switch_positions[i] = 0; ❶

    result = tryCombination(switch_positions);
    if (result != door) {
        for (i = 0; i < n; i++)
            if (door_for_switch[i] == -1)
                switch_positions[i] = 1; ❷
    }

    i = 0;
    while (!found) {
        if (door_for_switch[i] == -1)
            switch_positions[i] = 1 - switch_positions[i]; ❸
    }
}
```

```
    result = tryCombination(switch_positions);
    if (result != door) ❹
        found = 1;
    else
        i++;
}
door_for_switch[i] = door;
}
```

Параметр `door` определяет, какую дверь нужно обработать следующей.

Код начинается с перебора рычагов. Положение всех рычагов устанавливается как 0 ❶, за исключением тех, которые еще не связаны ни с какой дверью (если рычаг уже связан с дверью, его положение больше никогда не должно меняться).

Установив все еще не связанные рычаги в положение 0, мы определяем, открыта или нет текущая дверь. Если открыта, то нужно ее закрыть, чтобы потом путем поочередного изменения положения рычагов определить, какой из них ее откроет. Для закрывания двери мы устанавливаем все несвязанные рычаги в положение 1 ❷. Такой прием срабатывает, потому что, когда все рычаги находились в положении 0, дверь была открыта. Один из этих рычагов связан с этой дверью, значит, при изменении их положения она точно закроется.

Закрыв дверь, можно переходить к поиску связанного с ней рычага. Для каждого рычага, еще не связанного ни с какой дверью, положение *переключается* с 0 на 1 или с 1 на 0 ❸. Обратите внимание, что положение изменяется вычитанием из 1: если до этого оно было 1, то теперь становится 0; если же до этого оно было 0, то теперь становится 1. Затем выполняется проверка нового состояния двери. Если она открыта ❹, значит, связанный рычаг найден. Если же она все еще закрыта, значит, рычаг оказался неверный и цикл продолжается.

В `set_a_switch` выполняется линейный поиск по всем оставшимся рычагам. Всего может быть дано до 5000 рычагов, значит, для одной двери может потребоваться до 5000 вызовов `tryCombination`. Условие разрешает вызвать `tryCombination` не более 70 000 раз. Если нам не повезет и на первую дверь уйдет 5000 вызовов, на вторую — 4999, на третью — 4998 и т. д., то в рамках лимита обработать удастся всего около 14 дверей. Но этого мало, так как их число может достигать 5000. На этом с линейным поиском придется закончить.

## Использование двоичного поиска

Числа 5000 (максимальное количество дверей) и 70 000 (максимум попыток подбора) тонко намекают на необходимость применения для решения стратегии двоичного поиска. Обратите внимание, что  $\log_2 5000$  округляется до 13. Если мы найдем

способ использовать двоичный поиск, то он сможет подбирать верный рычаг для текущей двери всего за 13 шагов, но никак не за 5000. Если на каждую дверь нам потребуется по 13 шагов, то при 5000 дверей получится  $13 \times 5000 = 65\,000$  шагов в общем. Таким образом, мы впишемся в лимит 70 000.

Как же здесь использовать двоичный поиск? Процесс должен позволить отбрасывать на каждом шаге половину диапазона рычагов. Поразмышляйте на эту тему, прежде чем продолжать чтение.

Поясню идею на примере. Предположим, что у нас восемь дверей и восемь рычагов, а также что дверь 0 в текущий момент закрыта. Если мы переключим рычаг 0 и дверь 0 не откроется, то из этого мы поймем немного: только то, что рычаг 0 не связан с дверью 0 (все равно что сказать «1» при угадывании загаданного кем-то числа между 1 и 1000). Более удачной идеей будет переключить половину рычагов, так что давайте переключим рычаги 0, 1, 2 и 3. Независимо от того, как это повлияет на дверь 0, мы узнаем уже очень многое. Если дверь по-прежнему останется закрыта, то рычаги с 0 по 3 к ней отношения не имеют и можно сосредоточиться на оставшейся половине рычагов с 4 по 7. Если же дверь 0 откроется, то станет ясно, что с ней связан один из рычагов с 0 по 3 и нужно более плотно сосредоточиться именно на них. За один шаг мы сразу отсекаем половину диапазона. Продолжая таким же путем дальнейший поиск, мы в конечном итоге определим рычаг, связанный с дверью 0 (и его положение).

Предположим, что проделываем весь этот путь, снова и снова уменьшая диапазон рычагов в два раза, пока не останется всего один рычаг. Допустим, что связанным с дверью 0 оказался рычаг 6. Тогда мы установим его в положение, при котором дверь 0 будет открыта. В этом положении он и останется. Когда далее мы перейдем к обработке двери 1 или любой другой двери впоследствии, то предусмотрительно не станем менять положение рычага 6.

Теперь я могу представить решение для этой задачи, основанное на двоичном поиске. Новый код `set_a_switch` приведен в листинге 6.17. Функция `exploreCave` осталась той же, что и в листинге 6.15.

**Листинг 6.17.** Нахождение и установка рычага для текущей двери с помощью двоичного поиска

```
void set_a_switch(int door, int switch_positions[],
                 int door_for_switch[], int n) {
    int i, result;
    int low = 0, high = n-1, mid;

    for (i = 0; i < n; i++)
        if (door_for_switch[i] == -1)
            switch_positions[i] = 0;
```

```

result = tryCombination(switch_positions);
if (result != door) {
    for (i = 0; i < n; i++)
        if (door_for_switch[i] == -1)
            switch_positions[i] = 1;
}

while (low != high) { ❶
    mid = (low + high) / 2;
    for (i = low; i <= mid; i++)
        if (door_for_switch[i] == -1)
            switch_positions[i] = 1 - switch_positions[i];
    result = tryCombination(switch_positions); ❷
    if (result != door) {
        high = mid;
        for (i = low; i <= mid; i++)
            if (door_for_switch[i] == -1)
                switch_positions[i] = 1 - switch_positions[i];
    }
    else
        low = mid + 1;
}
door_for_switch[low] = door;
switch_positions[low] = 1 - switch_positions[low]; ❸
}

```

По сравнению с листингом 6.16 единственным отличием является замена линейного поиска двоичным. При каждой проверке условия двоичного поиска ❶ мы делаем так, чтобы текущая дверь была закрыта. В частности, когда `low` и `high` равны и цикл завершается, дверь остается закрытой. Тогда останется лишь изменить положение рычага `low`, чтобы ее открыть.

Теперь рассмотрим сам двоичный поиск. В каждой итерации мы вычисляем среднюю точку `mid`, затем изменяем положение первой половины рычагов (но только тех, которые еще не связаны с дверью). Какой эффект это оказало на текущую дверь ❷? Здесь есть две возможности:

- **Дверь открылась.** Теперь нам известно, что искомый рычаг находится между `low` и `mid`, значит, все рычаги больше `mid` отбрасываются. Мы также возвращаем каждый рычаг между `low` и `mid` обратно в положение, предшествовавшее текущей итерации. В итоге дверь снова закрывается, и можно переходить к очередной итерации.
- **Дверь осталась закрыта.** Значит, искомый рычаг находится между `mid + 1` и `high`, поэтому мы отбрасываем все рычаги, начиная с `mid` и менее. На этом все! В данном случае никакие рычаги не переключаются, потому что дверь по-прежнему закрыта, что нам и нужно.

По завершении двоичного поиска `low` и `high` будут равны, указывая на рычаг, связанный с текущей дверью. Текущая дверь в этот момент будет все еще закрыта, и мы переключим найденный рычаг для ее открытия ③.

Больше здесь подвохов нет: мы получили чистое, быстрое решение, основанное на двоичном поиске. Отправьте его на проверку, и оно успешно пройдет все тесты.

## Выводы

Иногда найти оптимальное решение намного сложнее, чем проверить допустимость одного из его вариантов. Сколько жидкости нужно залить в дерево? Неизвестно. Будет ли достаточно 10 литров? А вот это уже можно проверить.

При правильных условиях двоичный поиск способен преобразовать сложную задачу оптимизации в намного более простую задачу проверки допустимости. Иногда это похоже на жульничество. Добавление двоичного поиска означает всего лишь необходимость учитывать дополнительный логарифмический фактор. Но этот фактор практически не требует ресурсов, зато взамен мы получаем более простую задачу.

Я не утверждаю, что двоичный поиск является единственным способом решения задач текущей главы. Предполагаю, что есть и более быстрый способ накормить муравьев. Некоторые задачи, которые можно решить с помощью двоичного поиска, также можно решить и через динамическое программирование. Но я утверждаю, что двоичный поиск часто может предложить решения, которые окажутся конкурентноспособными и более простыми, чем их альтернативы. Если вам интересно, вернитесь к каждой задаче главы еще раз, но теперь подумайте, как можно их решить без двоичного поиска. Хотя скажу честно: если вы видите задачу, в которой можно использовать двоичный поиск, то не стоит сомневаться.

## Примечания

Задача «Кормление муравьев» входила в четвертый раунд Хорватского открытого соревнования по информатике 2014 года (2014 Croatian Open Competition in Informatics). Задача «Прыжки вдоль реки» предлагалась участникам серебряного дивизиона олимпиады по информатике США 2006 года (2006 USA Computing Olympiad). Задача «Качество жизни» взята из программы Международной олимпиады по информатике 2010 года (2010 International Olympiad in Informatics). Задача «Двери пещеры» взята из программы Международной олимпиады по информатике 2013 года (2013 International Olympiad in Informatics).

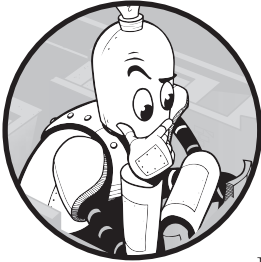
Двоичный поиск является одной из реализаций техники построения алгоритмов под названием *разделяй и властвуй* (divide and conquer, D&C). Алгоритмы D&C решают одну или более независимых подзадач, после чего совмещают их решения для поиска решения исходной задачи (двоичный поиск решает только одну подзадачу, соответствующую той части входных данных, о которой известно, что она содержит решение, в то время как другие алгоритм D&C обычно решают две или более подзадач). Чтобы больше узнать об этом семействе алгоритмов и некоторых задачах, которые они эффективно решают, прочитайте книгу *Algorithms Illuminated (Part 1): The Basics*<sup>1</sup> Тима Рафгардена (Tim Roughgarden).

---

<sup>1</sup> Рафгарден Т. Совершенный алгоритм. Основы. СПб.: Питер.

# 7

## Кучи и деревья отрезков



Использование структур данных позволяет грамотно их организовать, ускорив тем самым определенные операции. К примеру, в главе 1 мы узнали о хеш-таблицах, которые ускоряют поиск заданного элемента в коллекции.

В данной главе мы узнаем о двух новых структурах данных: кучах и деревьях отрезков. Куча полезна, когда требуется найти максимальный или минимальный элемент. Деревья отрезков используются, когда нужно выполнить запросы к частям массива. Первая задача продемонстрирует, как превратить медленные вычисления максимума в быстрые операции с кучей. Вторая и третья задачи покажут, как с помощью деревьев отрезков добиться того же эффекта в случае более обобщенных запросов к массивам.

### Задача 1. Акция в супермаркете

Рассмотрим задачу с платформы SPOJ с номером PRO.

#### Условие

В супермаркете каждый покупатель выбирает товары, которые хочет купить, после чего проходит через кассу, чтобы расплатиться. После оплаты покупатель получает чек, где указана общая стоимость покупки. К примеру, если человек наберет товаров на 18 долларов, то на чеке отразится именно такая сумма. При этом цены отдельных товаров нас не интересуют.

В супермаркете проходит акция, которая будет длиться  $n$  дней. В ходе акции каждый чек помещается в лотерейную корзину. В конце каждого дня из этой корзины извлекаются два чека: один с максимальной суммой  $x$  и второй с минимальной суммой  $y$ . Покупатель, которому принадлежит максимальный чек, получает приз в размере  $x - y$  долларов (неважно, как супермаркет вычисляет владельца чека). После этого чеки  $x$  и  $y$  безвозвратно исключаются из розыгрыша, но все остальные чеки остаются в корзине и участвуют в последующих розыгрышах.

Гарантируется, что в конце каждого дня в лотерейной корзине будет находиться не менее двух чеков.

Наша задача — вычислить общую призовую сумму, которая будет выдана супермаркетом за время акции.

### Входные данные

Входные данные содержат один тестовый пример, состоящий из следующих строк:

- целого числа  $n$ , указывающего продолжительность акции в днях. Значение  $n$  лежит в диапазоне от 1 до 5000;
- $n$  строк, по одной для каждого дня акции. Каждая такая строка начинается с целого числа  $k$ , указывающего число чеков в соответствующий день. Затем в этой же строке идут  $k$  целых чисел, представляющих суммы чеков. Значение  $k$  лежит в диапазоне от 0 до 100 000. Сумма каждого чека — не более 1 000 000.

Общее количество чеков, полученных за все время акции, не может превышать 1 000 000.

### Выходные данные

Требуется вывести общую сумму призовых денег, выданных супермаркетом.

Время на решение тестового примера — одна секунда.

### Решение 1. Максимум и минимум в массиве

С чего следует начать? Я бы советовал в первый же день скупить в супермаркете все товары, кроме одного леденца, а потом вернуться и купить тот самый леденец. Но как это выразить на языке алгоритмов?

Для многих задач в этой книге сложно определить подходящий алгоритм, не говоря уже об эффективном. Для текущей же задачи выбор подходящего алгоритма не



выглядит сложным. Определение суммы приза каждого дня предполагает простой поиск по лотерейной корзине максимальной и минимальной сумм. Этот путь выглядит и весьма эффективным.

Рассмотрим тестовый пример:

```
2
16 6 63 16 82 25 2 43 5 17 10 56 85 38 15 32 91
1 57
```

В конце первого дня до удаления чеков у нас есть 16 сумм:

```
6 63 16 82 25 2 43 5 17 10 56 85 38 15 32 91
```

Максимальный чек — 91, а минимальный — 2. Эти два чека извлекаются, в результате чего выплачивается приз размером  $91 - 2 = 89$  долларов. Вот что остается после их извлечения:

```
6 63 16 82 25 43 5 17 10 56 85 38 15 32
```

Далее мы переходим во второй день и добавляем 57:

```
6 63 16 82 25 43 5 17 10 56 85 38 15 32 57
```

Теперь максимальным чеком будет 85, а минимальным 5, значит, будет выплачено  $85 - 5 = 80$  долларов. В прошедшей двухдневной акции общая сумма выплат составила  $89 + 80 = 169$ .

Одна из возможных идей решения предполагает сохранение чеков в массиве. Для извлечения чека можно его просто удалить, как мы только что и сделали. Но в результате такого действия следующие за удаляемым чеки будут смещаться влево, заполняя освободившиеся места. Поэтому проще будет сохранить чеки в их текущих местах и связать с каждым флаг `used`. Если `used` равен 0, то чек еще не использован. Значение же `used`, равное 1, будет указывать, что данный чек уже использовался и логически удален (то есть мы его игнорируем).

Вот пара макросов и структура `receipt`:

```
#define MAX_RECEIPTS 1000000
#define MAX_COST 1000000

typedef struct receipt {
    int cost;
    int used;
} receipt;
```

Для нахождения и удаления максимальных и минимальных чеков потребуется вспомогательная функция. Ее код приведен в листинге 7.1.

**Листинг 7.1.** Поиск и удаление максимальных и минимальных чеков

```

int extract_max(receipt receipts[], int num_receipts) {
    int max, max_index, i;
    max = -1; ❶
    for (i = 0; i < num_receipts; i++)
        if (!receipts[i].used && receipts[i].cost > max) { ❷
            max_index = i;
            max = receipts[i].cost;
        }
    receipts[max_index].used = 1; ❸
    return max;
}

int extract_min(receipt receipts[], int num_receipts) {
    int min, min_index, i;
    min = MAX_COST + 1; ❹
    for (i = 0; i < num_receipts; i++)
        if (!receipts[i].used && receipts[i].cost < min) { ❺
            min_index = i;
            min = receipts[i].cost;
        }
    receipts[min_index].used = 1; ❻
    return min;
}

```

Операцию, которая удаляет и возвращает максимальное значение, принято называть *extract-max*. Аналогичным образом операция, удаляющая и возвращающая минимальное значение, называется *extract-min*.

Эти функции очень похожи. *extract\_max* устанавливает *max* равной -1 ❶, то есть меньше суммы любого чека. Когда она находит «реальную» сумму чека, *max* устанавливается равным этой сумме, после чего отслеживается только наибольшая сумма чека, найденная на текущий момент. По той же логике *extract\_min* инициализирует *min* со значением выше, чем любая возможная сумма чека ❹. Обратите внимание на то, что в каждой функции рассматриваются только те чеки, чье значение *used* равно 0 ❷ ❺, а также на то, что каждая функция устанавливает значение *used* каждого рассмотренного чека равным 1 ❸ ❻.

После этих вспомогательных функций осталось написать функцию *main* для считывания входных данных и решения задачи. Интересный нюанс состоит в том, что считывание входных данных и решение задачи выполняются поочередно: мы считываем небольшую часть данных (чеки первого дня), вычисляем призовую сумму этого дня, считываем еще часть данных (чеки второго дня), вычисляем призовую выплату и т. д. Все это реализовано в листинге 7.2.

**Листинг 7.2.** Функция *main* для считывания входных данных и решения задачи

```

int main(void) {
    static struct receipt receipts[MAX_RECEIPTS];

```

```
int num_days, num_receipts_today;
int num_receipts = 0;
long long total_prizes = 0; ❶
int i, j, max, min;
scanf("%d", &num_days);

for (i = 0; i < num_days; i++) {
    scanf("%d", &num_receipts_today);
    for (j = 0; j < num_receipts_today; j++) {
        scanf("%d", &receipts[num_receipts].cost);
        receipts[num_receipts].used = 0;
        num_receipts++;
    }
    max = extract_max(receipts, num_receipts);
    min = extract_min(receipts, num_receipts);
    total_prizes += max - min;
}
printf("%lld\n", total_prizes);
return 0;
}
```

Особенность этого кода — тип переменной `total_prizes` ❶. Дело в том, что возможностей `int` или `long int` будет недостаточно. `long int` может содержать значения до четырех миллиардов, а общая сумма призовых денег может составлять  $5000 \times 1\,000\,000$ , что равняется пяти миллиардам. В свою очередь, `long long int` может содержать числа, выражающие миллиарды, триллионы и намного большие значения, то есть в данном случае такой тип нас точно устроит.

Внешний цикл `for` выполняется по разу для каждого дня, а внутренний цикл `for` считывает каждый чек этого дня. После считывания всех чеков дня мы извлекаем максимальный и минимальный, обновляя общую сумму призовой выплаты.

Это завершенное решение задачи. Оно корректно выводит для нашего примера сумму 169. Но, к сожалению, оно окажется слишком медленным и вызовет ошибку «Time-Limit Exceeded».

Неэффективность этого решения можно показать, рассмотрев наиболее сложный для расчета случай. Предположим, что акция продолжается 5000 дней и что в каждый из первых 10 дней мы получаем по 100 000 чеков. На одиннадцатый день у нас в массиве будет миллион чеков. Поиск максимального и минимального значений происходит путем линейного сканирования массива. Однако поскольку мы удаляем только по два чека в день, то до конца акции их количество в массиве будет близко к миллиону. Итак, мы будем 5000 раз выполнять около миллиона шагов для нахождения максимума и еще столько же шагов для нахождения минимума. Итого получается  $5000 \times 2\,000\,000 = 10$  миллиардов шагов! Учитывая жесткость лимита времени на решение, уложиться в него никак не получится. Здесь нам потребуется возможность ускорить вычисления максимумов и минимумов.

Давайте сразу отбросим сортировку как возможный вариант оптимизации. Если мы будем поддерживать массив чеков упорядоченным, то поиск и удаление максимумов будут занимать одинаковое и постоянное время, поскольку максимум будет находиться в крайней правой позиции. Столько же времени займет поиск минимума, однако его удаление потребует других затрат, которые линейно зависят от размера массива, так как придется сдвигать оставшиеся элементы влево. Необходимость упорядочивания также существенно затрудняет процесс добавления чека, поскольку мы не можем просто поместить его в конец массива, а нужно будет находить подходящую позицию. Нет, упорядочивание здесь не подойдет. Правильный же ответ — кучи.

## Мах-куча

Начнем с рассмотрения способа быстрого поиска и извлечения максимального элемента массива. Это решит только половину задачи, так как нужно еще проделать то же самое для минимального, но и до него мы доберемся.

### Поиск максимума

Взгляните на дерево на рис. 7.1. Оно содержит 13 узлов, соответствующих следующим 13 чекам (первые 13 чеков из рассматривавшегося примера): 6, 63, 16, 82, 25, 2, 43, 5, 17, 10, 56, 85 и 38.

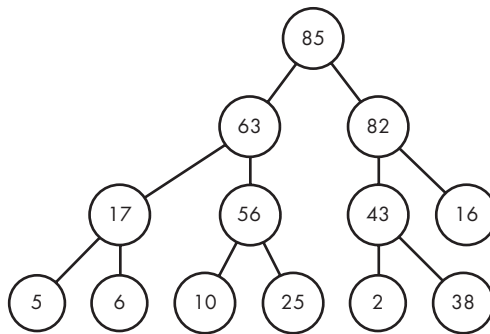


Рис. 7.1. Мах-куча

Вопрос на скорость мышления: какова максимальная сумма чека в дереве? Ответ — 85, корневой узел дерева. Если бы мы были уверены, что максимальный элемент дерева всегда будет расположен в его корне, то можно было бы просто возвращать элемент из корня, не прибегая к обходу дерева. Поэтому нашей задачей будет поддерживать дерево таким образом, чтобы чек с максимальной

суммой всегда располагался именно в корне. Придется быть очень внимательными, потому что нас будут сбивать два вида событий, способных нарушить порядок дерева:

- **Появление нового чека.** Нужно выяснить, как реорганизовать дерево для включения этого чека. Если новый чек оказывается самым крупным в дереве, то мы будем помещать его в корень.
- **Извлечение чека.** Нужно выяснить, как реорганизовать дерево, чтобы максимальный из оставшихся элементов дерева оказывался в корне.

Конечно же, все эти добавления и извлечения нужно делать быстро. В частности, необходимо превзойти в скорости алгоритм с линейным временем, так как именно такой малоэффективный способ сканирования нас сюда и привел.

### Что такое *тах-куча*?

На рис. 7.1 показан пример *тах-кучи*. Такое дерево позволяет быстро найти максимальный элемент.

Мах-куча обладает двумя важными свойствами. Во-первых, это *законченное* двоичное дерево, то есть каждый его уровень завершен (в нем нет недостающих узлов), за исключением, возможно, нижнего уровня, чьи узлы заполняются слева направо. Обратите внимание, что на рис. 7.1 каждый уровень закончен. Нижний при этом завершен не до конца, но это нормально, потому что его узлы заполняются слева направо (не путайте законченные двоичные деревья с полными двоичными деревьями из главы 2). Тот факт, что тах-куча является законченным двоичным деревом, не помогает нам непосредственно найти максимум, добавить элемент или извлечь максимум, но, как мы увидим, дает возможность с молниеносной скоростью создавать кучи.

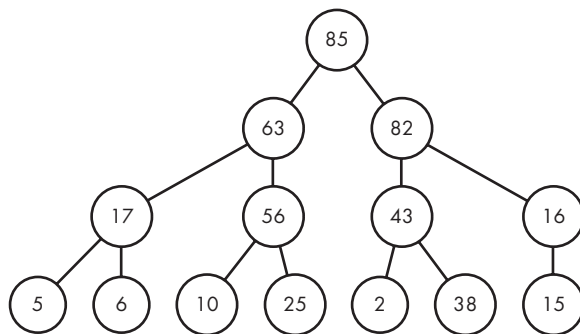
Во-вторых, значение каждого узла больше или равно каждому из значений его дочерних узлов (на рис. 7.1 все числа различны, значит, значение родителя будет строго больше, чем значения его детей). Это свойство называется *порядком тах-кучи*.

Рассмотрим узел со значением 56 на рис. 7.1. Как и говорилось, 56 больше, чем значения его детей (10 и 25). Это свойство верно в любой части дерева, именно поэтому максимальное число должно находиться в корне, ведь у каждого другого узла есть родительский узел с превосходящим его значением.

### Добавление в тах-кучу

При поступлении нового чека он добавляется в тах-кучу, но сделать это нужно правильно, не нарушив свойства порядка.

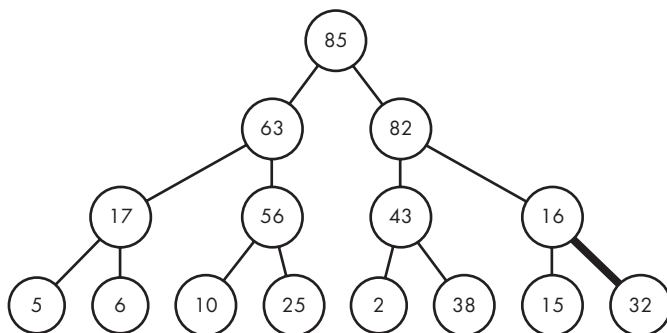
Добавим в кучу на рис. 7.1 значение 15. Есть всего одно место, куда можно поместить этот элемент, не нарушая свойства завершенности: в нижний уровень, справа от 38 (рис. 7.2).



**Рис. 7.2.** Мах-куча после добавления 15

Это определенно законченное двоичное дерево, но сохраняется ли свойство порядка? Да! Родителем 15 является 16, который его старше, что нам и нужно. Значит, дополнительно ничего делать не требуется.

Рассмотрим пример посложнее. Вставим в кучу на рис. 7.2 узел 32 (рис. 7.3).

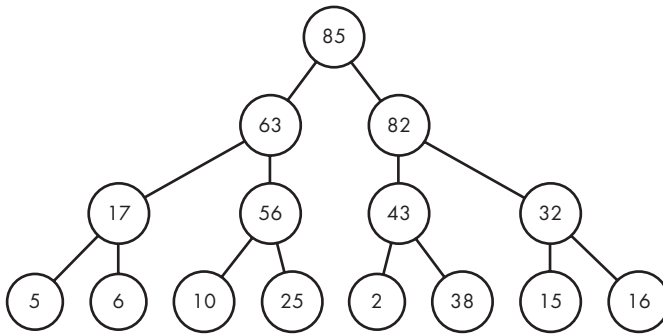


**Рис. 7.3.** Мах-куча после добавления 32

Здесь возникла проблема. Добавление 32 нарушило свойство порядка мах-кучи, потому что родитель (16) оказался младше потомка (32). Это можно исправить, поменяв их местами, как показано на рис. 7.4.

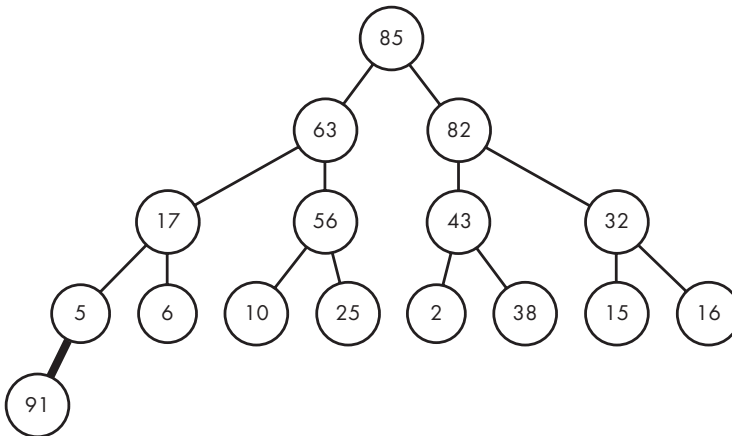
Итак, порядок восстановлен: теперь узел 32 больше, чем оба его потомка: он старше, чем 16, потому что именно для этого мы и произвели обмен, а также старше 15,

потому что 15 ранее был потомком 16. Как правило, выполнение такого обмена гарантирует сохранение свойства порядка двоичной кучи между новым узлом и его потомками.



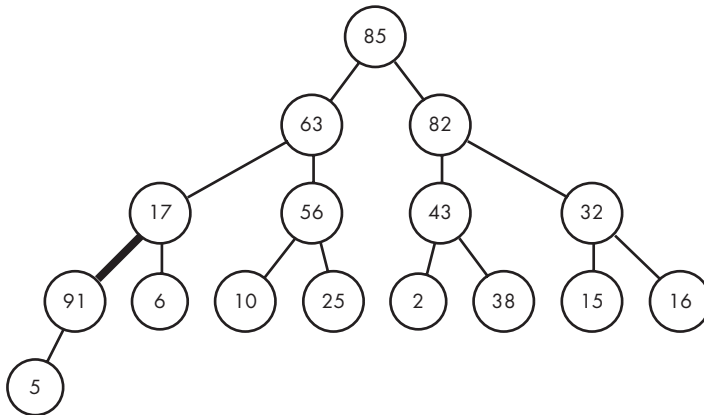
**Рис. 7.4.** Мах-куча с восстановленным свойством порядка

Итак, мы снова получили правильную мах-кучу, для чего было достаточно сделать один обмен. Но может потребоваться и больше таких операций, что демонстрируется на примере добавления 91. Результат показан на рис. 7.5.



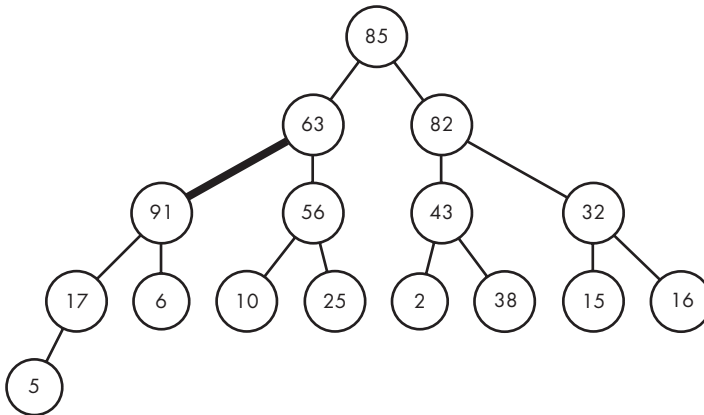
**Рис. 7.5.** Мах-куча после добавления 91

Теперь потребовалось создать новый уровень в нижней части дерева, так как предыдущий уже был заполнен. Однако нельзя оставлять 91 в качестве потомка 5, потому что это нарушит свойство порядка. Обмен это исправит, но не до конца. Смотрим рис. 7.6.



**Рис. 7.6.** Мах-куча с поднятым выше узлом 91

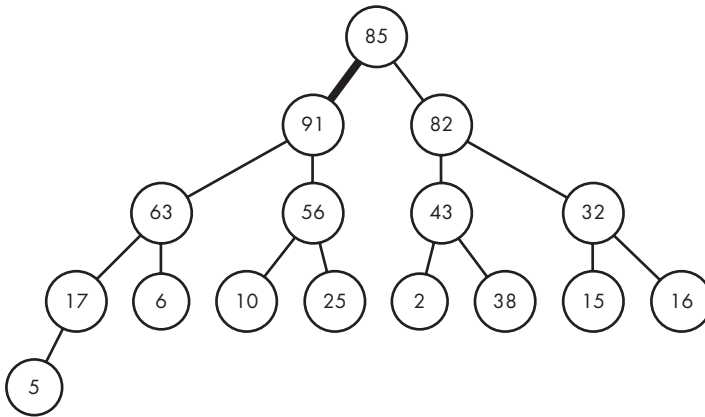
Проблему связи между узлами 5 и 91 мы исправили, но теперь узел 91 несогласован с узлом 17. Проблему можно исправить очередной перестановкой (рис. 7.7).



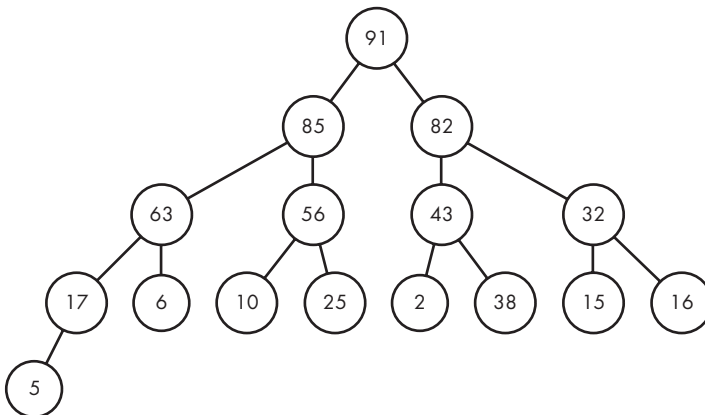
**Рис. 7.7.** Мах-куча после очередного подъема узла 91

И все же порядок мах-кучи все еще не восстановлен: теперь неправильно расположены узлы 63 и 91. При этом нарушение сдвигается все выше и выше по дереву, постепенно приближаясь к его корню. В худшем случае мы закончим на том, что поменяем местами новый и корневой узлы. Именно это и происходит, потому что 91 оказывается наибольшим элементом дерева. Для восстановления порядка требуются еще два обмена: первый показан на рис. 7.8, второй — на рис. 7.9.





**Рис. 7.8.** Мах-куча после очередного подъема узла 91



**Рис. 7.9.** Мах-куча после восстановления свойства порядка

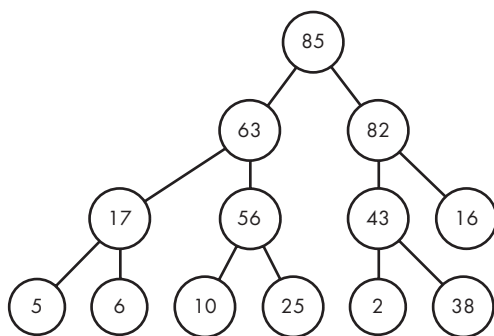
Наконец мы снова получили правильную мах-кучу. Потребовалось выполнить четыре операции обмена для значения, которое продвигалось с нижнего уровня до корня дерева. Ну а добавление значений, которым не приходится проделывать весь путь до корня, как вы уже видели, происходит еще быстрее.

### Извлечение из мах-кучи

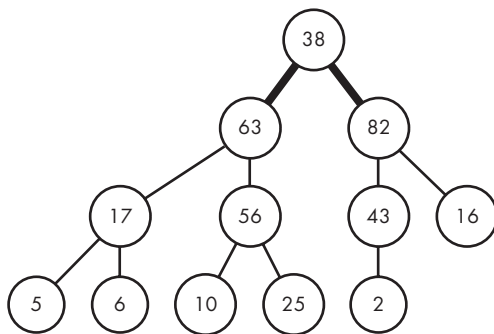
В конце каждого дня акции нам нужно будет извлекать из мах-кучи максимальный чек. При этом нужно быть внимательными и корректировать дерево. Вы увидите,

что извлечение противоположно процессу добавления узла, поскольку он будет не подниматься, а спускаться вниз.

Начнем также с рис. 7.1:



Извлечение максимума ведет к удалению 85 из корня, куда нужно будет передвинуть новый элемент. В противном случае наше дерево разрушится. Единственный узел, который можно использовать, не нарушая свойство законченности дерева, это правый крайний узел нижнего уровня. Это значит, что можно поменять местами 85 и 38 (рис. 7.10).

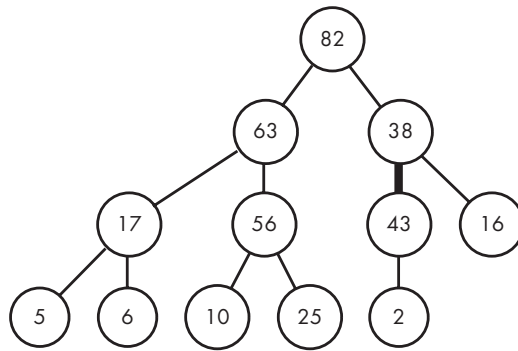


**Рис. 7.10.** Мах-куча после извлечения 85

Мы только что взяли небольшое значение из нижнего уровня дерева и перекинули его на самый верх. Как правило, это ведет к нарушению свойства порядка мах-кучи. В данном случае так и есть, потому что 38 меньше, чем 63 и 82.

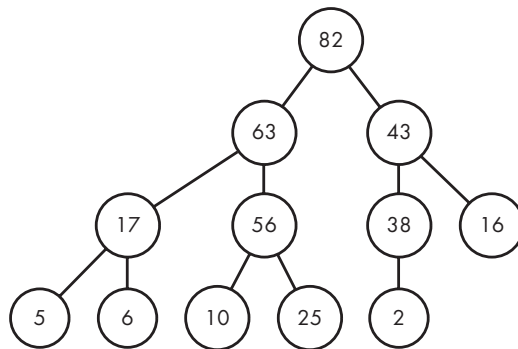
Для восстановления порядка мы вновь используем технику обмена. Однако в отличие от добавления, при извлечении у нас есть выбор. Будем ли мы менять

местами 38 и 63 или же 38 и 82? Обмен 38 и 63 не решает проблему, так как тогда 82 окажется потомком 63. А вот обмен 38 и 82 окажется уже правильным действием. Обычно нас интересует обмен с наиболее крупным потомком, что позволяет восстановить свойство порядка. На рис. 7.11 показан результат обмена местами 38 и 82.



**Рис. 7.11.** Мах-куча после смещения 38 ниже

Но на этом дело еще не закончено, так как порядок по-прежнему нарушен, теперь между 38 и 43. Тем не менее нарушение постепенно смещается вниз. В худшем случае мы получим мах-кучу, в которой 38 достигнет низа дерева. А пока поменяем местами 38 и 43 (рис. 7.12).



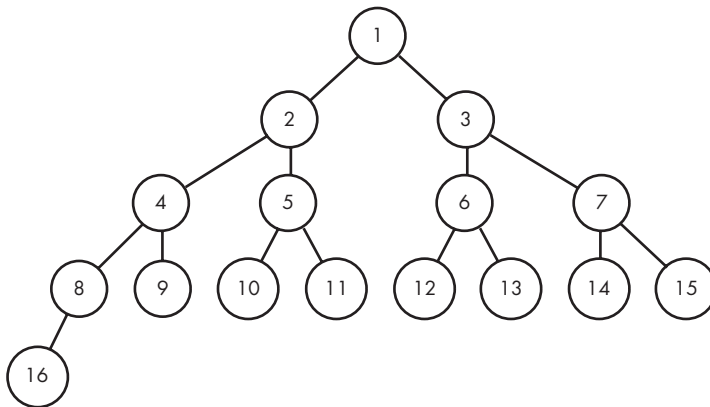
**Рис. 7.12.** Мах-куча после очередного смещения 38 вниз

Теперь 38 разместилось в подходящем для него месте и порядок мах-кучи восстановлен.

### Высота мах-кучи

Как добавление, так и извлечение узла подразумевает не более одного обмена на уровень: добавление требует обменов вверх по дереву, а извлечение — вниз. Быстры ли эти операции? Это зависит от высоты мах-кучи: если она невелика, то и операций будет мало. Поэтому нам необходимо определить связь между количеством элементов в мах-куче и ее высотой.

Взгляните на рис. 7.13, где изображено законченное двоичное дерево из 16 узлов.



**Рис. 7.13.** Законченное двоичное дерево из 16 узлов

Я пронумеровал узлы сверху вниз и слева направо, поэтому корневым является узел 1, двумя его ближайшими потомками — узлы 2 и 3, их детьми — узлы 4, 5, 6 и 7 и т. д. Можно заметить, что значение крайнего левого узла на каждом уровне удваивается: корень равен 1, следом идет 2, затем 4, потом 8 и, наконец, 16. То есть для создания очередного уровня дерева необходимо удваивать число его узлов. Это подобно двоичному поиску, где удваивание количества элементов ведет к дополнительной итерации цикла. Значит, высота завершеного двоичного дерева и, следовательно, высота мах-кучи имеют логарифмическую связь с количеством элементов в дереве.

Победа! Как и двоичный поиск, добавление в мах-кучу характеризуется оценкой времени  $O(\log n)$ , где  $n$  — количество элементов в дереве. Извлечение из мах-кучи — это тоже  $O(\log n)$ . Таким образом, нам удалось уйти от медленного линейного  $O(n)$ .

### Мах-кучи как массивы

Мах-куча является разновидностью двоичного дерева — модели, которая нам уже знакома. Вспомним, как мы решали задачу про Хэллоуин в главе 2. Тогда мы

использовали структуру `node` с указателями на левого и правого потомков. Этого достаточно для извлечения значения из `тах-кучи`, поскольку его продвижение вниз требует обращения только к дочерним узлам. А вот для добавления значений необходимо обращаться к родительским узлам, значит, потребуется указатель на родителя:

```
typedef struct node {  
    ... fields for receipts  
    struct node *left, *right, *parent;  
} node;
```

Не забудьте инициализировать указатель на родителя для каждого нового узла, а также установить левый и правый указатели на потомков как `NULL` при удалении соответствующего узла из `тах-кучи`.

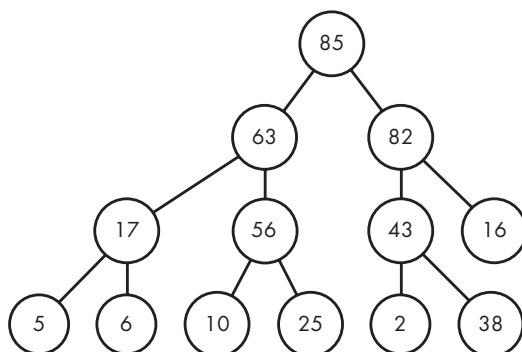
Хотя нет, забудьте это все — мы сможем обойтись вообще без указателей на детей и родителей. Вернемся снова к рис. 7.13, где узлы пронумерованы сверху вниз и слева направо. Родителем узла 16 является узел 8. Родителем узла 12 — узел 6. Родитель узла 7 — это 3. Какова связь между номерами узла и его родителя? Они отличаются в два раза.  $16/2 = 8$ ,  $12/2 = 6$ ,  $7/2 = 3$ . Правда, последняя операция на деле даст 3,5, но мы просто отбросим дробную часть.

Для продвижения вверх по дереву мы производим целочисленное деление на 2. Посмотрим, что произойдет, если обратить процесс и вместо деления умножить на два.  $8 \times 2 = 16$ , значит, умножение на 2 переносит нас от 8 к его левому потомку. Но большинство узлов имеют двух потомков, а нам может потребоваться переходить из узла к его правому потомку. Это легко: мы просто добавляем к номеру левого потомка 1. Например, можно перейти от 6 к его левому потомку через  $6 \times 2 = 12$ , а к правому — путем  $6 \times 2 + 1 = 13$  (связь между 13/2 и 6 подтверждает, что можно спокойно отбрасывать дробные 0,5 для перехода от потомка к его родителю).

Эти связи между узлами работают только потому, что `тах-кучи` являются законченными двоичными деревьями. Двоичное дерево может иметь и беспорядочную структуру, где в одном месте присутствует длинная цепочка узлов, а в другом — короткая. По такому дереву уже не получится перемещаться через умножение и деление на два, пока мы не отсортируем узлы. Если дерево изначально окажется сильно разбалансированным, то на это могут уйти огромные ресурсы.

Если сохранить `тах-кучу` в массиве — сначала корень, затем его потомков, затем их потомков и т. д., тогда индекс узла в массиве будет соответствовать номеру этого узла. При этом, чтобы нумерация совпала с приведенной на рис. 7.13, потребуется начать индексирование не с 0, а с 1 (можно начать и с индекса 0, но тогда связи между узлами будут иными: родитель  $i$ -го узла будет иметь индекс  $(i - 1)/2$ , а потомки — индексы  $2i + 1$  и  $2i + 2$ ).

Снова обратимся к рис. 7.1 с кучей из 13 сумм чеков:



Вот соответствующий массив:

Индекс	1	2	3	4	5	6	7	8	9	10	11	12	13
Значение	85	63	82	17	56	43	16	5	6	10	25	2	38

Индекс 6 в массиве содержит значение 43. Какой у 43 левый потомок? Просто вычислите индекс:  $6 \times 2 = 12$ , в массиве это 2. Какой у 43 правый потомок? Рассчитываем индекс:  $6 \times 2 + 1 = 13$ , здесь у нас 38. А кто родитель 43? Проверяем индекс:  $6/2 = 3$ , значит, 82. Какой бы узел нас в данный момент не интересовал, с помощью массива можно перейти к его потомку или родителю путем минимальных математических расчетов.

## Реализация max-кучи

Каждый элемент создаваемых куч будет хранить индекс чека и его сумму. Именно эти два значения нам потребуются при его извлечении.

Вот структура:

```

typedef struct heap_element {
    int receipt_index;
    int cost;
} heap_element;
  
```

Теперь мы готовы реализовать max-кучу. Две ключевые операции здесь — это добавление в кучу нового чека и извлечение из нее максимального. Начнем с добавления, прописанного в листинге 7.3.

**Листинг 7.3.** Добавление узла в max-кучу

```
void max_heap_insert(heap_element heap[], int *num_heap,
                    int receipt_index, int cost) {
    int i;
    heap_element temp;
    (*num_heap)++; ❶
    heap[*num_heap] = (heap_element){receipt_index, cost}; ❷
    i = *num_heap; ❸
    while (i > 1 && heap[i].cost > heap[i / 2].cost) { ❹
        temp = heap[i];
        heap[i] = heap[i / 2];
        heap[i / 2] = temp;
        i = i / 2; ❺
    }
}
```

Функция `max_heap_insert` получает четыре параметра. Первые два относятся к куче: `heap` — это массив, содержащий max-кучу, а `num_heap` — указатель на количество элементов в куче. `num_heap` является указателем, потому что нам потребуется увеличивать количество элементов на один и учитывать это в вызывающей инструкции. Еще два параметра относятся к новому чеку: `receipt_index` — индекс добавляемого чека, `cost` — связанная с ним сумма.

Код начинается с увеличения количества элементов в куче на один ❶, после чего новый чек сохраняется в новой ячейке ❷. Переменная `i` отслеживает индекс только что добавленного элемента ❸.

Нет гарантии, что после этого у нас по-прежнему будет max-куча. Внедренный элемент может оказаться больше своего родителя, и тогда придется выполнить операции обмена. За это отвечает цикл `while` ❹.

Для продолжения цикла `while` должны быть выполнены два условия. Во-первых, нужно, чтобы было `i > 1`, потому что в ином случае `i` равно 1 и родителя не имеет (напомним, что куча начинается с индекса 1, а не 0). Во-вторых, нужно, чтобы сумма чека была больше, чем у его родителя. Тело цикла `while` выполняет обмен, затем перемещает нас из текущего узла к его родителю ❺. Здесь мы снова встречаем схему деления на 2 для продвижения вверх по дереву. Простой, понятный и правильный код — самый лучший.

Теперь перейдем к извлечению элементов из max-кучи. В листинге 7.4 дан соответствующий код.

**Листинг 7.4.** Извлечение максимума из max-кучи

```
heap_element max_heap_extract(heap_element heap[], int *num_heap) {
    heap_element remove, temp;
    int i, child;
```

```

remove = heap[1]; ❶
heap[1] = heap[*num_heap]; ❷
(*num_heap)--; ❸
i = 1; ❹
while (i * 2 <= *num_heap) { ❺
    child = i * 2; ❻
    if (child < *num_heap && heap[child + 1].cost > heap[child].cost)
        child++; ❼
    if (heap[child].cost > heap[i].cost) { ❽
        temp = heap[i];
        heap[i] = heap[child];
        heap[child] = temp;
        i = child; ❾
    } else
        break;
}
return remove;
}

```

Код начинается с сохранения находящегося в корне чека, который будет извлекаться ❶. Затем корень заменяется нижним крайним правым узлом ❷, и количество элементов в куче уменьшается на один ❸. Этот новый корневой узел может не вписаться в свойство порядка, поэтому мы используем переменную *i* для отслеживания его позиции в куче ❹.

Далее, так же как в листинге 7.3, используется цикл `while`, который будет выполнять необходимые обмены. На этот раз условие цикла `while` ❺ говорит, что в куче должен присутствовать левый потомок узла *i*, а если его нет, то потомки у узла *i* отсутствуют и нарушение свойства порядка невозможно.

Внутри цикла `child` устанавливается как левый потомок ❻. Затем, если правый потомок существует, то его сумма сравнивается с суммой левого потомка. Если сумма справа больше, то `child` устанавливается как правый потомок ❼. Теперь `child` является потомком с наибольшим значением, и мы проверяем, не вызывает ли он нарушение порядка ❽. Если да, то выполняется обмен. Затем мы переходим вниз по дереву ❾, чтобы проверить возможное нарушение порядка.

Обратите внимание, что если узел и его наибольший потомок уже упорядочены верно, то цикл прерывается оператором `break`, поскольку больше нарушений в дереве быть не может.

Последнее, что делает функция — возвращает чек с максимальной суммой. Далее мы сможем использовать этот чек для определения приза дня, а также обеспечим невозможность его повторного рассмотрения. Но перед этим нам еще нужно познакомиться с `min`-кучами, чтобы можно было извлекать минимум.



## Min-кучи

*Min-куча* позволит нам быстро добавлять новый чек и извлекать чек с минимальной суммой.

### Определение и операции

На самом деле вам уже известно практически все необходимое о *min-кучах*, потому что они почти идентичны *max-кучам*.

*Min-куча* — это законченное двоичное дерево. Ее высота логарифмически связана с количеством элементов. Мы сможем сохранять ее в массиве точно так же, как делали с *max-кучей*. Для нахождения родительского узла используем деление на 2. Левого потомка находим путем умножения на 2, а правого — путем умножения на 2 и прибавления 1. Все как и было описано выше.

Единственное новшество — это *свойство порядка min-кучи*: значение узла меньше либо равно значению каждого из его дочерних узлов. Таким образом, в корне теперь оказывается наименьшее значение, а не наибольшее. Именно это нам и нужно, чтобы ускорить извлечение минимальных значений.

Снова рассмотрим суммы 13 чеков: 6, 63, 16, 82, 25, 2, 43, 5, 17, 10, 56, 85 и 38. На рис. 7.14 показана *min-куча* для этих данных.

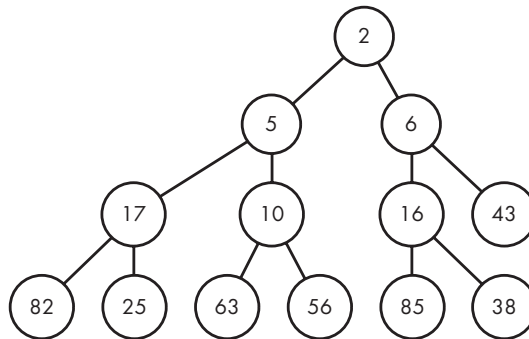


Рис. 7.14. Min-куча

Операции добавления элемента в *min-кучу* и извлечения минимального значения аналогичны соответствующим операциям в *max-кучах*.

Для добавления создается новый узел справа от всех узлов нижнего уровня либо, если этот уровень заполнен, начинается новый уровень. Затем узел меняется

местами с узлами выше, пока не будет достигнут корень или его значение не окажется большим либо равным значению его родителя.

При извлечении минимума корень заменяется нижним правым узлом, после чего происходит его обмен с расположенными ниже узлами дерева, пока он либо не станет концевым, либо не окажется меньше своих детей или равен им.

### Реализация min-кучи

Реализация min-кучи будет копией кода max-кучи за исключением имен функций и изменения сравнений с  $>$  на  $<$ . В листинге 7.5 приведен код для добавления элементов.

#### Листинг 7.5. Добавление в min-кучу

```
void min_heap_insert(heap_element heap[], int *num_heap,
                    int receipt_index, int cost) {
    int i;
    heap_element temp;
    (*num_heap)++;
    heap[*num_heap] = (heap_element){receipt_index, cost};
    i = *num_heap;
    while (i > 1 && heap[i].cost < heap[i / 2].cost) {
        temp = heap[i];
        heap[i] = heap[i / 2];
        heap[i / 2] = temp;
        i = i / 2;
    }
}
```

В листинге 7.6 дан код для извлечения минимального значения.

#### Листинг 7.6. Извлечение минимума из min-кучи

```
heap_element min_heap_extract(heap_element heap[], int *num_heap) {
    heap_element remove, temp;
    int i, child;
    remove = heap[1];
    heap[1] = heap[*num_heap];
    (*num_heap)--;
    i = 1;
    while (i * 2 <= *num_heap) {
        child = i * 2;
        if (child < *num_heap && heap[child + 1].cost < heap[child].cost)
            child++;
        if (heap[child].cost < heap[i].cost) {
            temp = heap[i];
            heap[i] = heap[child];
            heap[child] = temp;
        }
    }
    return remove;
}
```

```

        i = child;
    } else
        break;
}
return remove;
}

```

Можно было бы написать более общие функции `heap_insert` и `heap_extraction`, получающие функцию сравнения в качестве параметра (во многом аналогично `qsort`). Но код понятнее без этого, так что оставим все как есть.

## Решение 2. Кучи

Вооружившись `max`- и `min`-кучами, можно приступить ко второму этапу решения задачи.

Теперь нам нужна функция `main`, считывающая входные данные и использующая кучи для быстрого добавления и извлечения чеков. Ее код приведен в листинге 7.7. В нем вы встретите два цикла `while` и, возможно, удивитесь: что они тут делают?

**Листинг 7.7.** Функция `main` для решения задачи с помощью куч

```

int main(void) {
    static int used[MAX_RECEIPTS] = {0}; ❶
    static heap_element min_heap[MAX_RECEIPTS + 1]; ❷
    static heap_element max_heap[MAX_RECEIPTS + 1];
    int num_days, receipt_index_today;
    int receipt_index = 0;
    long long total_prizes = 0;
    int i, j, cost;
    int min_num_heap = 0, max_num_heap = 0;
    heap_element min_element, max_element;
    scanf("%d", &num_days);

    for (i = 0; i < num_days; i++) {
        scanf("%d", &receipt_index_today);
        for (j = 0; j < receipt_index_today; j++) {
            scanf("%d", &cost);
            max_heap_insert(max_heap, &max_num_heap, receipt_index, cost); ❸
            min_heap_insert(min_heap, &min_num_heap, receipt_index, cost); ❹
            receipt_index++;
        }

        max_element = max_heap_extract(max_heap, &max_num_heap); ❺
        while (used[max_element.receipt_index])
            max_element = max_heap_extract(max_heap, &max_num_heap);
        used[max_element.receipt_index] = 1;
    }
}

```

```
    min_element = min_heap_extract(min_heap, &min_num_heap); ❸
    while (used[min_element.receipt_index])
        min_element = min_heap_extract(min_heap, &min_num_heap);
    used[min_element.receipt_index] = 1;
    total_prizes += max_element.cost - min_element.cost;
}
printf("%lld\n", total_prizes);
return 0;
}
```

В массиве `used` ❶ для каждого чека будет сохраняться 1, если он был использован, и 0, если нет. Мах-куча ❷ и min-куча на один элемент больше массива `used`, так как индекс 0 в кучах мы не используем.

Индекс каждого чека добавляется и в мах-кучу ❸, и в min-кучу ❹. Затем происходит извлечение чека из мах-кучи ❺, а также из min-кучи ❻. Здесь-то и задействуются циклы `while`, перебирающие массивы до нахождения еще не использованного чека. Позвольте, я объясню подробнее.

При извлечении чека из мах-кучи было бы здорово также извлечь его из min-кучи и наоборот, чтобы обе они всегда содержали одинаковые чеки. На деле этого не происходит. Но, поскольку каждый такой чек помечается как использованный, мы его отбрасываем и больше не обрабатываем. Именно это и делают циклы `while`: игнорируют чеки, ранее обработанные одной из куч.

Разобраться поможет следующий пример:

```
2
2 6 7
2 9 10
```

Призовая сумма здесь составит  $7 - 6 = 1$  в первый день и  $10 - 9 = 1$  во второй, итого 2. После считывания два чека первого дня оказываются в каждой из куч (здесь и ниже первой приведена мах-куча, второй — min-куча):

receipt_index	cost
1	7
0	6

receipt_index	cost
0	6
1	7

Затем выполняются процедуры извлечения, в ходе которых из каждой кучи удаляется по одному чеку. Вот что остается:

receipt_index	cost
0	6

receipt_index	cost
1	7

Чек 0 по-прежнему присутствует в max-куче, а чек 1 — в min-куче. Однако оба они были использованы, значит, повторно их обрабатывать не нужно.

Теперь рассмотрим второй день. В каждую кучу добавляются чеки 2 и 3, после чего кучи выглядят так:

receipt_index	cost
3	10
0	6
2	9

receipt_index	cost
1	7
2	9
3	10

Из max-кучи извлекается чек 3. Отлично. Но из min-кучи вроде бы должен быть извлечен чек 1. В отсутствие отбрасывающего его цикла `while` может возникнуть проблема, потому что чек 1 уже обрабатывался.

В конце каждого дня один или оба цикла `while` могут выполнить множество итераций. Если бы это продолжалось день за днем, то у нас возникли бы сложности с быстродействием программы. Тем не менее заметьте, что чек можно удалить из кучи не более одного раза. Если в куче находится  $r$  чеков, то и извлечений из кучи может быть не более  $r$ , независимо от того, произойдут они в один день или будут разделены по нескольким.

Пришло время отправить решение на проверку. В отличие от решения 1, которое впустую растрачивало время на медленные поиски, его версия на основе куч пройдет все тестовые примеры за установленное время.

## Кучи

Если у вас есть поток входящих данных и периодически требуется определять максимальное или минимальное значения, то вам помогут кучи. Мах-куча используется для извлечения и обработки максимума, а min-куча — для минимума.

Куча может применяться и для задач с *приоритетной очередью*. В такой очереди каждый элемент имеет приоритет, определяющий его важность. Иногда более высокий приоритет определяется большими числами, в этом случае нужно использовать мах-кучу, в других ситуациях — меньшими числами, тогда используется min-куча. Конечно, если нас интересуют элементы как с максимальным, так и минимальным приоритетом, то можно использовать две кучи, как мы делали при решении задачи с акцией супермаркета.

## Два дополнительных варианта применения

По опыту могу сказать, что min-кучи применяются чаще, чем мах-кучи. Поэтому давайте рассмотрим два дополнительных примера именно с их использованием.

### Сортировка кучей

Речь пойдет об очень известном алгоритме *сортировки кучей*, который мы можем реализовать на базе уже имеющихся знаний. Все, что потребуется, — это добавить все значения в min-кучу, а затем поочередно извлечь минимальные, получив их в порядке возрастания от меньшего к большему. Для этого потребуется буквально четыре строки кода (листинг 7.8).

#### Листинг 7.8. Сортировка кучей

```
#define N 10

int main(void) {
    static int values[N] = {96, 61, 36, 74, 45, 60, 47, 6, 95, 93};
    static int min_heap[N + 1];
    int i, min_num_heap = 0;

    //Сортировка кучей. 4 строки!
    for (i = 0; i < N; i++)
        min_heap_insert(min_heap, &min_num_heap, values[i]);
}
```

```
for (i = 0; i < N; i++)
    values[i] = min_heap_extract(min_heap, &min_num_heap);

for (i = 0; i < N; i++)
    printf("%d ", values[i]);
printf("\n");
return 0;
}
```

Здесь мы добавляем в кучу целые числа, следовательно, нужно изменить `min_heap_insert` и `min_heap_extract` на использование и сравнение чисел, а не структур `heap_element`.

Сортировка кучей выполняет  $n$  добавлений и  $n$  извлечений. Каждое из этих действий куча реализует за время  $\log n$ , значит, она является алгоритмом с оценкой времени  $O(n \log n)$ . Таким образом, в наихудшем случае время выполнения будет таким же, как у быстрееших алгоритмов сортировки (q в функции Си `qsort`, вероятно, происходит от *quicksort*, алгоритма сортировки, который на практике работает быстрее, чем сортировка кучей).

## Алгоритм Дейкстры

Алгоритм Дейкстры (глава 5) затрачивает много времени на нахождение очередного узла для обработки, поскольку делает это путем перебора расстояний до узлов в поиске наименьшего. Для ускорения этого алгоритма можно использовать min-кучу. Соответствующая реализация продемонстрирована в приложении Б.

## Выбор структуры данных

Структура данных имеет принципиальное значение всего для нескольких видов операций. Нет такой универсальной структуры, которая бы все ускоряла, поэтому выбор подходящей для нужной задачи остается за вами.

Вспомним главу 1, где мы использовали хеш-таблицу. Можно ли применить ее для решения задачи «Акция в супермаркете»? Нет! Хеш-таблица хорошо подойдет для ускорения поиска конкретного элемента. Какие снежинки могут быть похожи на снежинку  $s$ ? Находится ли слово  $s$  в списке слов? Именно такие запросы адресуются хеш-таблице. Однако хеширование не поможет ответить на вопрос: каков минимальный элемент массива? Придется выполнять поиск по хеш-таблице, что окажется ничуть не быстрее поиска по стандартному массиву.

Итак, выбор структуры данных, подходящей для решаемой задачи, остается за вами. Для нахождения минимального элемента массива такой структурой будет min-куча.

Как и любую структуру данных общего назначения, кучи можно использовать для решения очень разных задач, но сама структура кучи всегда является такой, как вы ее изучили здесь. Поэтому вместо того, чтобы решать очередную задачу с помощью кучи, мы перейдем к такой, где нам потребуется уже другая структура — *дерево отрезков*. Как и кучи, деревья отрезков ускоряют только небольшое число операций. Но при этом впечатляет количество задач, в которых нас будет интересовать именно такое ускорение.

## Задача 2. Построение декартовых деревьев

В этой задаче мы построим *декартово дерево*, проще называемое «дуча» (от слов «дерево» и «куча»)¹. Дуча — это гибкая структура данных, способная решать разнообразнейшие поисковые задачи. Я настоятельно рекомендую впоследствии познакомиться с ней поближе. Здесь же нашей целью будет лишь ее построение, но не использование. Естественно, я предоставлю вам все необходимые для этого сведения.

Перед нами задача с платформы POJ под номером 1785.

### Условие

Дуча — это двоичное дерево, где каждый узел имеет и метку, и значение приоритета. На рис. 7.15 показан пример дучи, в которой заглавные буквы являются метками, а положительные числа — приоритетами. Метку и приоритет каждого узла я разделил слешем. К примеру, корневой узел имеет метку С и приоритет 58.

Дуча должна обладать двумя свойствами: одним свойством меток и одним — приоритетов.

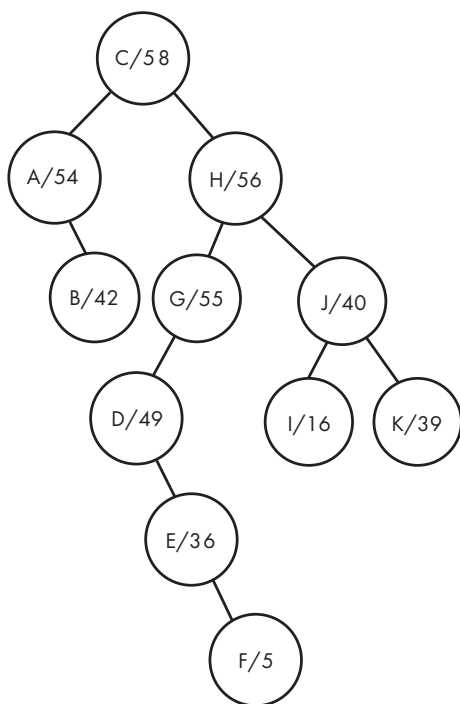
Сначала поговорим о метках. Для любого узла  $x$  все метки в его левом поддереве должны быть меньше метки  $x$ , а в правом поддереве — больше  $x$ . Это свойство *двоичного дерева поиска* (BST).

Можете убедиться, что у дучи на рис. 7.15 это свойство меток выполняется. В случае использования буквенных меток их старшинство определяется алфавитным порядком. Возьмем в качестве примера корень с меткой С. Обе метки в его левом поддереве меньше С, и все метки в его правом поддереве больше С. В качестве другого примера рассмотрим узел с меткой G. Все метки в его левом поддереве — D, E и F — меньше G. А что насчет меток в правом поддереве? Здесь их нет, значит, и проверять нечего.

Теперь поговорим о приоритетах. Для любого узла  $x$  приоритеты его детей будут меньше, чем приоритет  $x$ . Да ведь это же в точности свойство *max-кучи*!

¹ В английском варианте «treap» — от слов «tree» (дерево) и «heap» (куча). — *Примеч. пер.*



**Рис. 7.15.** Декартово дерево

Снова взглянем на корень, чей приоритет равен 58. Его дети должны иметь меньшие значения — так и есть, у них приоритеты 54 и 56. А что насчет узла G с приоритетом 55? Нужно, чтобы его дитя обладало меньшим значением, — и здесь все верно, так как его приоритет равен 49.

Таким образом, дуча — двоичное дерево, чьи метки удовлетворяют свойству BST, а приоритеты удовлетворяют свойству max-кучи. Обратите внимание, что требование к форме отсутствует, поэтому дуча может обладать любой структурой. Также нет необходимости в законченности, как в случае с кучами.

В текущей задаче нам предоставляются значения метки и приоритета для каждого узла, на основе которых требуется собрать и вывести декартово дерево.

### Входные данные

Входные данные могут содержать любое число тестовых примеров. Каждая строка начинается с целого числа  $n$  из диапазона от 0 до 50 000. Если  $n$  равно 0, значит, далее тестовых примеров для обработки нет.

Если же  $n$  больше нуля, то оно указывает количество узлов в тестовом примере. За  $n$  идут разделенные пробелом записи, по одной для каждого узла. Каждая запись имеет форму  $L/P$ , где  $L$  является значением метки, а  $P$  — приоритета узла. Метки — это строки из букв; приоритеты — положительные целые числа. Все метки и приоритеты уникальны.

Вот пример входных данных, из которых выстраивается дуча на рис. 7.15:

```
11 A/54 I/16 K/39 E/36 B/42 G/55 D/49 H/56 C/58 J/40 F/5
0
```

### Выходные данные

Для каждого тестового примера следует вывести формулу дучи в одну строку. Требуемый формат дучи:

```
(<left_subtreap><L>/<P><right_subtreap>)
```

Здесь `<left_subtreap>` является левым декартовым поддеревом, `<L>` — меткой корня, `<P>` — приоритетом корня, а `<right_subtreap>` — правым декартовым поддеревом. Декартовы поддеревья выводятся в том же формате.

Приведенным выше входным данным должен соответствовать следующий вывод:

```
((A/54(B/42))C/58(((D/49(E/36(F/5)))G/55)H/56((I/16)J/40(K/39))))
```

Время на решение тестовых примеров — две секунды.

### Рекурсивный вывод декартовых поддеревьев

Давайте еще раз рассмотрим образцы узлов и подумаем, как из них можно создать дучу. Вот эти узлы:

```
A/54 I/16 K/39 E/36 B/42 G/55 D/49 H/56 C/58 J/40 F/5
```

Вспомним, что приоритеты декартового дерева должны подчиняться свойству *пах-кучи*. В частности, это означает, что узел с максимальным приоритетом должен быть корневым. Так как условия задачи гарантируют, что все приоритеты имеют разную величину, то только один из них может быть максимальным. Значит, с этим мы разобрались: корневым должен быть узел `C/58`.

Теперь для каждого узла из оставшихся необходимо решить, должен ли он идти в левое или правое поддерево узла `C`. Все приоритеты этих узлов меньше 58, значит, с их помощью разделение на левое и правое поддеревья сделать не получится, а вот свойство *BST* в этом поможет. Это свойство сообщает, что метки в левом поддереве

должны быть меньше, чем С, а метки в правом поддереве больше С. Следовательно, можно разделить оставшиеся узлы на две группы: для левого поддерева и для правого:

A/54 B/42  
I/16 K/39 E/36 G/55 D/49 H/56 J/40 F/5

Таким образом в левом декартовом поддереве окажутся узлы А и В, а в правом узлы I, K, E, G и т. д.

Мы разделили изначальную задачу на две меньшие подзадачи в точности того же типа. От нас требовалось создать дучу для 11 узлов. Мы уменьшили эту задачу до создания дучи для двух узлов и дучи для восьми узлов. Сделать это можно рекурсивно.

Давайте определим конкретные правила, которые будем использовать. В качестве базового случая можно взять дучу из нуля узлов, для которой вывод не потребуется. Для рекурсивного случая мы определим в качестве корневого узел с наивысшим приоритетом, а затем разделим оставшиеся узлы на имеющие меньшие и большие метки относительно метки корня. Далее мы выводим открывающую скобку, рекурсивно выводим дучу для меньших меток, затем корневой узел, затем дучу для больших меток и в завершение закрывающую скобку.

Для нашего образца входных данных мы выведем открывающую скобку, а затем левое декартово поддерево:

(A/54(B/42))

Далее пойдет корневой узел:

C/58

А за ним правое декартово поддерево:

((D/49(E/36(F/5)))G/55)H/56((I/16)J/40(K/39))

Завершится строка закрывающей скобкой.

## **Сортировка по меткам**

Прежде чем перейти к коду, хочу поделиться с вами еще одной идеей. Исходя из вышеизложенного, можно предположить, что нам нужно создать новый массив с узлами, содержащими меньшие метки, для передачи в первый рекурсивный вызов и массив с узлами, имеющими большие метки, для передачи во второй рекурсивный вызов. К счастью, можно этого избежать, на начальном этапе упорядочив узлы по

меткам от меньших к большим. Затем мы просто сообщим каждому рекурсивному вызову начальный и конечный индексы массива, за который он отвечает.

К примеру, если упорядочить те же входные данные по меткам, то получится следующее:

```
A/54 B/42 C/58 D/49 E/36 F/5 G/55 H/56 I/16 J/40 K/39
```

Затем укажем первому рекурсивному вызову создать декартово поддереву для первых двух узлов, а второму рекурсивному вызову — создать декартово поддереву для последующих восьми узлов.

## Решение 1. Рекурсия

Вот константы и структура:

```
#define MAX_NODES 50000
#define LABEL_LENGTH 16

typedef struct treap_node {
    char * label;
    int priority;
} treap_node;
```

Длина меток нам не известна, поэтому я выбрал начальный размер 16. Вы увидите, что для считывания каждой метки вызывается функция `read_label`. Если длина 16 окажется недостаточной, то функция будет увеличивать выделяемую память, пока метка в нее не уместится (возможно, это перебор, поскольку в тестовых примерах используются короткие метки длиной до пяти букв, но лучше перестраховаться, чем потом сожалеть).

## Функция `main`

Рассмотрим функцию `main`, приведенную в листинге 7.9. В ней задействуется вспомогательная функция `read_label`, которую я только что упомянул, и `compare` для сравнения узлов дучи, а также происходит вызов `solve` для вывода дучи. Все это мы вскоре разберем.

### Листинг 7.9. Функция `main` для считывания входных данных и решения задачи

```
int main(void) {
    static treap_node treap_nodes[MAX_NODES];
    int num_nodes, i;
    scanf("%d ", &num_nodes);
    while (num_nodes > 0) {
        for (i = 0; i < num_nodes; i++) {
            treap_nodes[i].label = read_label(LABEL_LENGTH);
        }
    }
}
```

```

        scanf("%d ", &treap_nodes[i].priority);
    }
    qsort(treap_nodes, num_nodes, sizeof(treap_node), compare);
    solve(treap_nodes, 0, num_nodes - 1);
    printf("\n");
    scanf("%d ", &num_nodes);
}
return 0;
}

```

Будьте осторожны со `scanf` в программе, считывающей смесь чисел и строк. Здесь каждое число из входных данных сопровождается пробелом, и нам не нужно, чтобы эти символы пробелов предворяли следующие за ними метки. Для считывания и отбрасывания этих пробелов я добавил пробел после каждого спецификатора формата `%d` `scanf`.

## Вспомогательные функции

С помощью `scanf` мы считываем приоритеты, но не метки. Метки считываются функцией `read_label` из листинга 7.10. По сути, такую же функцию мы использовали ранее дважды, в последний раз в листинге 4.16. На этот раз единственное отличие состоит в том, что мы прекращаем считывание у символа `/`, отделяющего метку от приоритета ❶.

### Листинг 7.10. Считывание метки

```

/*based on https://stackoverflow.com/questions/16870485 */ char *read_label(int
size) {
    char *str;
    int ch;
    int len = 0;
    str = malloc(size);
    if (str == NULL) {
        fprintf(stderr, "malloc error\n");
        exit(1);
    }
    while ((ch = getchar()) != EOF && (ch != '/')) { ❶
        str[len++] = ch;
        if (len == size) {
            size = size * 2;
            str = realloc(str, size);
            if (str == NULL) {
                fprintf(stderr, "realloc error\n");
                exit(1);
            }
        }
    }
    str[len] = '\0';
    return str;
}

```

Как обычно, `qsort` требуется функция сравнения. В листинге 7.11 приведена функция, которая сравнивает узлы по их меткам.

**Листинг 7.11.** Функция сравнения для сортировки

```
int compare(const void *v1, const void *v2) {
    const treap_node *n1 = v1;
    const treap_node *n2 = v2;
    return strcmp(n1->label, n2->label);
}
```

Функция `strcmp` отлично справляется с задачей, возвращая отрицательное целое число, если первая строка по алфавиту меньше второй, 0, если строки равны, и положительное число, если первая строка больше второй.

## Вывод декартова дерева

Прежде чем перейти к функции `solve`, нам нужна вспомогательная функция для возвращения индекса узла с максимальным приоритетом. Она приведена в листинге 7.12. Здесь используется медленный линейный поиск от индекса `left` до индекса `right` (есть повод обеспокоиться).

**Листинг 7.12.** Нахождение максимального приоритета

```
int max_priority_index(treap_node treap_nodes[], int left, int right) {
    int i;
    int max_index = left;
    for (i = left + 1; i <= right; i++)
        if (treap_nodes[i].priority > treap_nodes[max_index].priority)
            max_index = i;
    return max_index;
}
```

Вот теперь можно выводить дучу. Функция `solve` дана в листинге 7.13.

**Листинг 7.13.** Решение задачи

```
void solve(treap_node treap_nodes[], int left, int right) {
    int root_index;
    treap_node root;
    if (left > right) ❶
        return;
    root_index = max_priority_index(treap_nodes, left, right); ❷
    root = treap_nodes[root_index];
    printf("(");
    solve(treap_nodes, left, root_index - 1); ❸
    printf("%s/%d", root.label, root.priority);
    solve(treap_nodes, root_index + 1, right); ❹
    printf(")");
}
```

Эта функция получает три параметра: массив узлов дучи, а также индексы `left` и `right`, определяющие диапазон узлов, в котором должна строиться дуча. Начальный вызов из `main` передаст 0 для `left` и `num_nodes - 1` для `right`, чтобы в построении дучи были задействованы все узлы.

Базовый случай для этой рекурсивной функции — отсутствие узлов в дуче ❶. В этом случае происходит возвращение без вывода чего-либо. Нет узлов, нет и вывода.

В противном случае из числа узлов с индексами между `left` и `right` мы находим индекс узла с максимальным приоритетом ❷. Это корень дучи, который разделяет задачу на две части: вывод дучи для узлов с меньшими метками и вывод дучи для узлов с большими метками. Каждая из этих задач решается отдельным рекурсивным вызовом ❸ ❹.

Вот оно, наше первое решение. Я бы сказал — неплохое. В действительности оно делает правильно две важные вещи. Во-первых, рационально сортирует узлы, чтобы каждому вызову `solve` требовались только индексы `left` и `right`. Во-вторых, использует рекурсивный вызов, сокращая работу по выводу дучи.

Однако при отправке этого кода на проверку вы увидите, что все стопорится из-за линейного поиска узла с максимальным приоритетом (листинг 7.12). Что с ним не так? Какой вид дучи провоцирует его низкую производительность? Об этом мы и поговорим далее.

## Запросы максимума на отрезке

В разделе «Запрос суммы одномерного диапазона» главы 6 рассматривалась задача нахождения суммы на отрезке массива. В ней требовалось найти в массиве *а* сумму всех элементов от `a[left]` до `a[right]`, где `left` и `right` — левый и правый индексы интервала.

Здесь же требуется решить похожую задачу, известную как *запрос максимума на отрезке* (RMQ, range maximum query). В массиве *а* надо найти индекс *максимального* элемента в интервале от `a[left]` до `a[right]` (в некоторых задачах вместо индекса бывает достаточно получить значение максимального элемента, но нам требуется именно его индекс).

В решении 1 мы предложили реализацию RMQ из листинга 7.12. Его код перебирает элементы от `left` до `right`, проверяя, есть ли индекс, узел которого имеет более высокий приоритет, чем найденный до сих пор. Эта функция вызывается для каждого декартова поддерева, и каждый вызов запускает линейный поиск по активному отрезку массива. Если бы большинство линейных поисков проводились по небольшим отрезкам массивов, то мы бы вполне обошлись этим алгоритмом. Однако решение проверяется, в том числе на входных данных, которые

обуславливают множество итераций поиска по большим отрезкам массива. Вот пример такого списка узлов:

A/1 B/2 C/3 D/4 E/5 F/6 G/7

Мы сканируем все семь узлов, находя в качестве узла с максимальным приоритетом G/7. Затем рекурсивно выводим дучу для узлов с меньшими метками, а также для узлов с большими метками. При этом первый рекурсивный вызов получает все узлы, кроме G/7, а второй рекурсивный вызов выполняется для нуля узлов. Первый вызов получает следующее:

A/1 B/2 C/3 D/4 E/5 F/6

Теперь нужно выполнить еще одно сканирование этих шести элементов для определения узла с наивысшим приоритетом. Им оказывается F/6, который устанавливается корнем этого декартова поддерева, после чего производятся еще два рекурсивных вызова, первый из которых нагружается всеми оставшимися узлами, что ведет к очередному ресурсоемкому сканированию массива. Этот паттерн дорогостоящих сканирований массивов продолжается, пока есть необработанные узлы.

Таким образом, в худшем варианте задачи для  $n$  узлов первый RMQ совершает  $n$  шагов, второй —  $n - 1$  шагов, и т. д. Итого получится  $1 + 2 + 3 + \dots + n$  шагов. Эта формула выглядит как  $n(n + 1)/2$ . В главе 1 мы видели схожую формулу в разделе «Выявление проблемы», и можно аналогичным образом заключить, что здесь продельвается работа  $O(n^2)$  (квадратичная).

А вот еще один способ убедиться, что мы проделываем  $O(n^2)$  работы. Отбросьте  $n/2$  наименьших выражений и сосредоточьтесь только на оставшихся  $n/2$  больших выражениях (предположим, что  $n$  — четное число, чтобы  $n/2$  было целым). Тогда мы получаем формулу  $n + (n - 1) + (n - 2) + \dots + (n/2 + 1)$ . Здесь мы имеем  $n/2$  выражений, каждое из которых больше  $n/2$ , значит, их минимальная сумма составит  $(n/2)(n/2) = n^2/4$ . Это квадратичный вариант!

Следовательно, линейный поиск в решении задачи RMQ неэффективен. В разделе «Запрос суммы одномерного диапазона» главы 6 мы использовали для ускорения запроса массив частичных сумм. Освежите эту тему в памяти, чтобы ответить на вопрос: «Можно ли использовать эту технику также и для решения RMQ?»

К сожалению, нет (вообще-то к счастью, потому что это дает мне возможность рассказать вам об одной из моих любимых структур данных). Для суммирования элементов от индекса 2 до индекса 5 можно найти частичную сумму для индекса 5 и вычесть частичную сумму для индекса 1. Таким образом, вычитание отменит излишнее сложение: частичная сумма для индекса 5 содержит частичную сумму для индекса 1, значит, можно просто вычесть последнюю. Увы, тем же путем



нельзя упростить нахождение максимума. Если максимум среди элементов до индекса 5 будет равен 10, а максимум среди элементов до индекса 1 — тоже 10, то каков тогда максимум элементов от индекса 2 до индекса 5? Да кто его знает. Это может быть какой угодно элемент с индексом 2, 3, 4 или 5. Крупный элемент в начале интервала не дает последующим элементам оказать влияние на значения максимума. Когда этот элемент отбрасывается, мы утрачиваем ориентир. Сравните это с массивом частичных сумм, где каждый элемент оказывает влияние на общее значение.

Давайте в качестве последней отчаянной попытки обратимся к куче. Можно ли использовать мах-кучу для решения RMQ? И снова ответ отрицательный. Мах-куча дает нам максимальный элемент во всей куче без возможности выявления его в каком-либо интервале.

Настало время для чего-то нового.

## **Деревья отрезков**

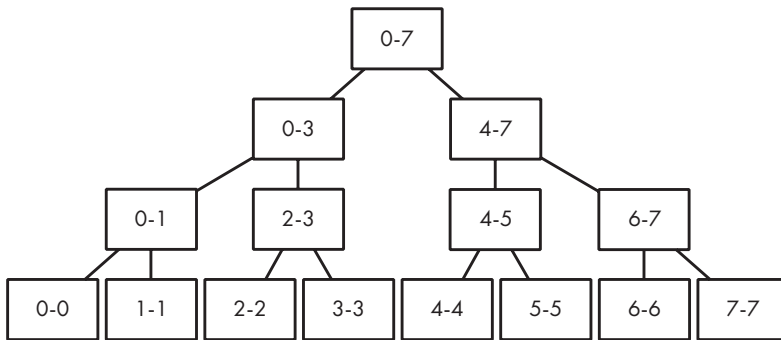
Оставим пока тему дучей в стороне и вернемся к ней позже, когда найдем лучший способ реализации RMQ.

*Дерево отрезков* — это полное двоичное дерево, где каждый узел связан с конкретным отрезком лежащего в его основе массива. Каждый узел хранит ответ на запрос об его отрезке. Например, для запросов максимума каждый узел хранит индекс максимального элемента в своем отрезке. Но деревья отрезков можно использовать и для других запросов. Отрезки упорядочиваются так, чтобы объединение нескольких из них позволяло ответить на любой запрос.

### **Отрезки**

Корневой узел дерева отрезков содержит информацию для запроса обо всем массиве. Значит, если нам потребуется выполнить RMQ для всего массива, то можно будет сделать это в один шаг, просто обратившись к корню. Для других запросов нам придется использовать другие узлы. У корневого узла есть два потомка: левый отвечает на запрос о первой половине массива, а правый — на запрос о второй половине. Каждый из этих узлов имеет двух потомков, которые продолжают дальнейшее разделение на отрезки, и так далее, пока не будут достигнуты отрезки, состоящие из всего одного элемента.

На рис. 7.16 показано дерево отрезков для запросов к массиву из восьми элементов. Название каждого узла содержит его левую и правую крайние точки. Сейчас в дереве отрезков еще нет информации для RMQ, пока просто сосредоточимся на самих отрезках.

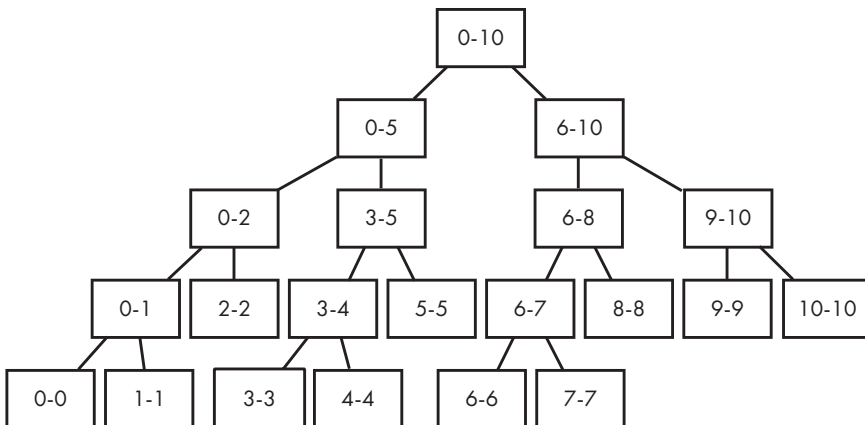


**Рис. 7.16.** Дерево отрезков для массива из восьми элементов

Обратите внимание, что размеры отрезков при спуске на каждый очередной уровень делятся пополам. К примеру, корневой отрезок охватывает восемь элементов, каждый из его детей — уже по четыре, их дети, в свою очередь, — по 2 и т. д. Как и в куче, высота дерева отрезков равна  $\log n$ , где  $n$  — это количество элементов в массиве. Мы сможем отвечать на любой запрос, проделывая одинаковое количество работы для каждого уровня, то есть получим оценку времени запроса  $O(\log n)$ .

На рис. 7.16 представлено законченное двоичное дерево. В процессе изучения куч мы узнали, как работать с такими деревьями — сохранять их в массиве. Это существенно упрощает поиск детей родителя.

Ниже представлено еще одно дерево отрезков, но уже с небольшим сюрпризом (рис. 7.17).



**Рис. 7.17.** Дерево отрезков для массива из 11 элементов

Это незаконченное двоичное дерево, потому что нижний уровень полностью не заполнен. К примеру, у узла 2–2 нет детей, несмотря на то что у узла 3–4 они есть.

Тем не менее все здесь хорошо. Мы можем сохранить такое дерево отрезков в массиве. Мы также можем умножать индекс узла на 2 для получения его левого потомка и умножать на 2 с последующим прибавлением 1 для получения индекса правого потомка. При этом в массиве произойдет лишь небольшой перерасход памяти. Например, ниже представлен порядок элементов массива для дерева с рис. 7.17, где \* обозначает неиспользуемый элемент:

```
0-10
0-5 6-10
0-2 3-5 6-8 9-10
0-1 2-2 3-4 5-5 6-7 8-8 9-9 10-10
0-0 1-1 * * 3-3 4-4 * * 6-6 7-7 * * * * *
```

Однако наличие неиспользуемых ячеек несколько усложняет определение количества элементов в массиве, созданном для дерева отрезков.

Я буду называть информацию из входных данных задачи «внутренним массивом». К примеру, в «Построении декартовых деревьев» внутренний массив — это массив приоритетов.

Если количество элементов внутреннего массива  $n$  равно степени 2 (4, 8, 16 и т. п.), то нам подойдет дерево отрезков и соответствующий массив с числом элементов  $2n$ . Например, подсчитайте узлы на рис. 7.16 — их там 15, что меньше, чем  $8 \times 2 = 16$  ( $2n$  достаточно, потому что сумма всех степеней 2 меньше  $n$  равна  $n - 1$ , например  $4 + 2 + 1 = 7$ , что на 1 меньше 8).

Если же  $n$  внутреннего массива не будет являться степенью 2, то дерева и массива с числом элементов  $2n$  окажется недостаточно. Чтобы убедиться в этом, взгляните на рис. 7.17: для хранения узлов дерева потребуется массив из 26 элементов (больше, чем  $2 \times 11 = 22$ ).

Чем больше элементов нужно охватить в дереве отрезков, тем больше должен быть соответствующий массив, но каков его требуемый размер? Предположим, что у нас есть внутренний массив из  $n$  элементов, для которого нужно создать дерево отрезков. Я утверждаю, что дереву отрезков достаточен массив из  $4n$  элементов.

Пусть  $m$  будет наименьшей степенью 2, которая больше либо равна  $n$ . К примеру, если  $n$  равно 11, то  $m$  равно 16. Следовательно, дерево отрезков для  $m$  элементов можно сохранить в массиве с  $2m$  элементов. Поскольку  $m \geq n$ , массива с  $2m$  элементов будет достаточно и для хранения дерева для  $n$  элементов.

К счастью,  $m$  может быть максимум вдвое больше  $n$  (худший случай относится к значениям  $n$ , которые идут сразу за степенями 2, например, если  $n$  равно 9, то

$m$  равно 16). Следовательно, если нам нужен массив из  $2m$  элементов и  $m$  не может быть больше  $2n$ , то  $2m$  будет не больше  $2 \times 2n = 4n$ .

### Инициализация отрезков

В каждом узле дерева отрезков мы сохраняем три единицы информации: левый индекс его отрезка, правый индекс его отрезка и индекс максимального отрезка в интервале. В этом разделе мы инициализируем первые два из этих параметров.

Вот структура, которую мы используем для узла дерева отрезков:

```
typedef struct segtree_node {
    int left, right;
    int max_index;
} segtree_node;
```

Для инициализации левого и правого параметров каждого узла напомним функцию:

```
void init_segtree(segtree_node segtree[], int node,
                 int left, int right)
```

Предположим, что `segtree` — это массив достаточного размера для дерева отрезков. Параметр `node` является индексом корня дерева отрезков; `left` и `right` — это левый и правый индексы соответственно. Начальный вызов `init_segtree` будет выглядеть так:

```
init_segtree(segtree, 1, 0, num_elements - 1);
```

Здесь `num_elements` представляет количество элементов во внутреннем массиве (к примеру, количество узлов в дуге).

Для инициализации `init_segtree` можно использовать рекурсию. Если `left` и `right` равны, то перед нами отрезок из одного элемента и деление производить не нужно. В противном случае мы имеем рекурсию и должны разделить отрезок на два. Код приведен в листинге 7.14.

#### Листинг 7.14. Инициализация отрезков дерева

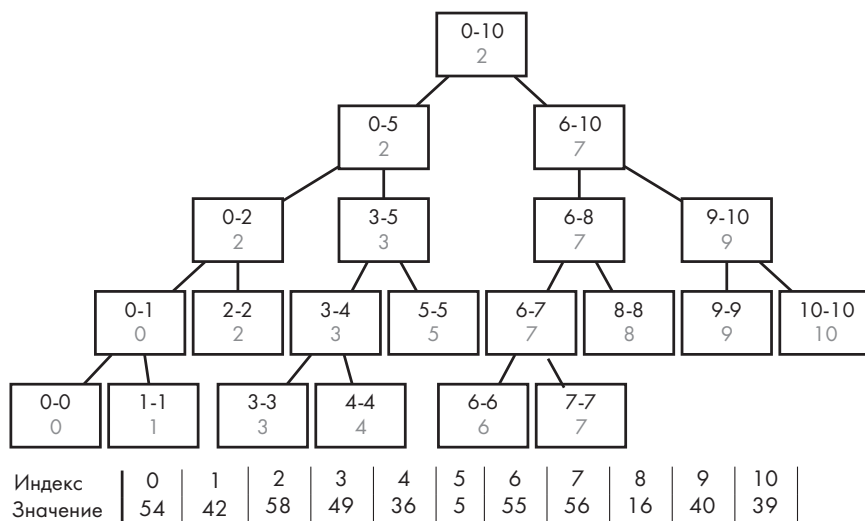
```
void init_segtree(segtree_node segtree[], int node,
                 int left, int right) {
    int mid;
    segtree[node].left = left;
    segtree[node].right = right;
    if (left == right) ❶
        return;
    mid = (left + right) / 2; ❷
    init_segtree(segtree, node * 2, left, mid); ❸
    init_segtree(segtree, node * 2 + 1, mid + 1, right); ❹
}
```

Сначала идет сохранение значений `left` и `right` в узле. Затем следует проверка базового случая ❶ и возврат из функции, если создания дочерних узлов не требуется.

Если же дочерние узлы нужны, то вычисляется средняя точка текущего диапазона ❷. Затем нужно создать левое дерево отрезков для индексов от `left` до `mid` и правое дерево отрезков для индексов от `mid + 1` до `right`. Это осуществляется двумя рекурсивными вызовами: одним для левого ❸ и вторым для правого ❹ отрезков. Обратите внимание, что с помощью `node * 2` мы перемещаемся к левому потомку, а с помощью `node * 2 + 1` — к правому.

### Заполнение дерева отрезков

После инициализации дерева отрезков надо добавлять в каждый узел индекс максимального элемента его отрезка. Рассмотрим пример, для которого нам потребуются и дерево отрезков, и массив, на котором оно будет основано. В качестве первого используем дерево отрезков, приведенное на рис. 7.17, а в качестве второго — массив из 11 элементов из раздела «Сортировка по меткам». На рис. 7.18 показано заполненное дерево отрезков. Индекс максимума указан в каждом узле дерева ниже его крайних точек.



**Рис. 7.18.** Дерево отрезков и массив приоритетов

Произведем пару быстрых проверок. Возьмем узел 0–0 в нижней части дерева. Это отрезок из одного элемента с индексом 0, значит, единственный вариант индекса максимума в нем будет тоже 0. Похоже на базовый случай.

А теперь рассмотрим узел 6–10. В нем указано, что между индексами 6 и 10 индексом максимального элемента является 7. В индексе 7 хранится число 56, и вы можете убедиться, что это наибольший элемент на отрезке. Чтобы вычислить это быстро, можно использовать индексы максимумов, сохраненные в дочерних узлах отрезка 6–10: левый потомок сообщает, что для отрезка 6–8 нужным индексом является 7, а правый указывает, что для отрезка 9–10 нужный индекс — 9. Тогда для 6–10 у нас есть всего два варианта: индексы 7 или 9. Похоже на рекурсивный случай!

Все верно: мы будем заполнять дерево с помощью рекурсии во многом аналогично тому, как делали для инициализации его отрезков. Код приведен в листинге 7.15.

**Листинг 7.15. Добавление максимумов**

```
int fill_segtree(segtree_node segtree[], int node,
               treap_node treap_nodes[]) {
    int left_max, right_max;

    if (segtree[node].left == segtree[node].right) { ❶
        segtree[node].max_index = segtree[node].left;
        return segtree[node].max_index; ❷
    }

    left_max = fill_segtree(segtree, node * 2, treap_nodes); ❸
    right_max = fill_segtree(segtree, node * 2 + 1, treap_nodes); ❹

    if (treap_nodes[left_max].priority > treap_nodes[right_max].priority) ❺
        segtree[node].max_index = left_max;
    else
        segtree[node].max_index = right_max;
    return segtree[node].max_index; ❻
}
```

Параметр `segtree` является массивом, в котором хранится дерево отрезков. Мы предполагаем, что он уже инициализирован в листинге 7.15. Параметр `node` — индекс корня дерева отрезков, а `treap_nodes` — массив узлов дучи. Мы задействуем здесь узлы дучи, чтобы иметь возможность обращаться к их приоритетам, но более здесь с ними ничего не связано. Вы можете легко заменить эти узлы на любой подходящий для решения задачи компонент.

Приведенная функция возвращает индекс максимального элемента для корневого узла дерева отрезков. Код начинается с проверки базового случая: состоит ли узел из всего одного элемента ❶. Если да, то индекс максимума для этого узла — его левый индекс (можно было бы использовать и правый, так как они одинаковы). Затем мы возвращаем этот индекс максимума ❷.

Если же перед нами не базовый случай, то выполняется поиск отрезка, охватывающего более одного индекса. Мы совершаем рекурсивный вызов к левому поддереву ❸. Этот вызов выясняет значение `max_index` для каждого узла в этом

поддереве и возвращает значение `max_index` корня этого поддерева. Затем то же самое проделывается для правого поддерева ④. Далее возвращенные рекурсивными вызовами индексы сравниваются ⑤ и выбирается тот, чей приоритет выше. На основе этого устанавливается `max_index` текущего узла. В заключение нужно вернуть этот индекс максимума ⑥.

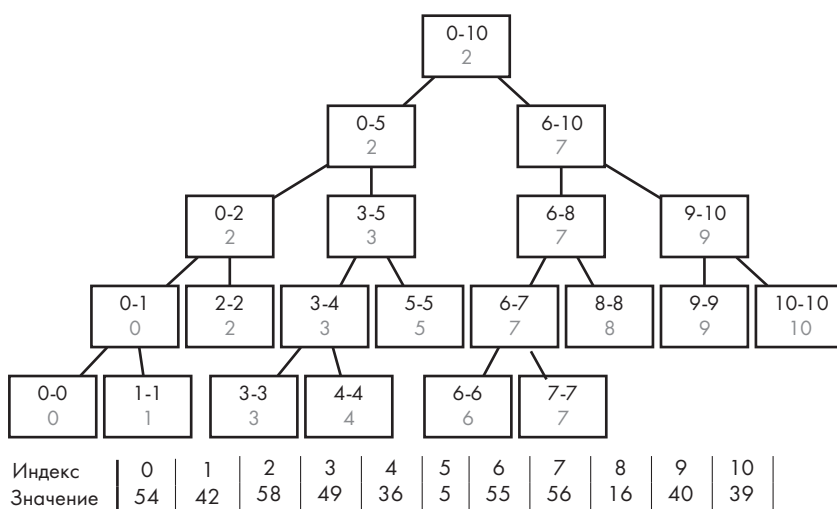
Заполнение дерева подобным образом происходит в линейном времени: для каждого узла мы выполняем постоянное количество работы по нахождению индекса его максимального элемента.

### Запрос к дереву отрезков

Подведем итог. Мы зашли в тупик, стараясь решить задачу «Построение декартовых деревьев», потому что не умели быстро находить максимумы в диапазонах. В результате мы потратили много времени на разработку деревьев отрезков, решая, как выбирать эти отрезки, какой размер должен иметь массив для дерева и как сохранять индекс максимального элемента для каждого узла.

Конечно же, вся эта возня с деревом отрезков окажется напрасной, если мы не добьемся ускорения обработки запросов. И вот настал долгожданный момент, когда труд должен принести результат: реализацию быстрых запросов при помощи деревьев отрезков. Не беспокойтесь, для этого потребуется немногим больше, чем реализация того же типа рекурсии, что мы использовали для деревьев отрезков.

Чтобы лучше разобраться, сделаем несколько запросов к дереву, представленному на рис. 7.18.



Для первого запроса возьмем интервал 6–10. Он охватывает только часть диапазона корневого отрезка 0–10, значит, возвращение индекса максимума из корня не даст точного результата. Вместо этого мы запросим у каждого дочернего узла корня индекс его максимального элемента и выберем из них индекс наибольшего. Левый потомок корня охватывает отрезок 0–5, который не пересекается с интервалом 6–10. Значит, левый рекурсивный вызов ничего нам не дает. Однако правый потомок корня охватывает именно отрезок 6–10. Рекурсивный вызов к этому потомку вернет индекс 7, который и нужно будет вернуть в качестве общего ответа.

Для второго запроса возьмем интервал 3–8. Снова запросим у каждого дочернего узла корня индекс максимального значения в его отрезке, но на этот раз ответят уже оба потомка, потому что диапазон 3–8 пересекается и с 0–5, и с 6–10. Рекурсивный вызов к левому потомку вернет 3, а рекурсивный вызов к правому вернет 7. Тогда в корне мы только сравним элементы с индексами 3 и 7. Последний окажется больше, значит, он и будет нашим ответом.

Обычно я не разбираю процесс рекурсии, но здесь сделаю исключение, потому что это важно. Давайте углубимся в рекурсивный вызов к левому поддереву. Мы делаем запрос по 3–8, а диапазон узла охватывает элементы 0–5. Левым потомком 0–5 является 0–2. У 0–2 нет общих индексов с интересующим нас диапазоном 3–8, следовательно, он исключается. В результате остается узел 3–5. Важно, что 3–5 полностью входит в 3–8, следовательно, здесь мы остановимся и вернем из рекурсивного вызова к 3–5 индекс 3.

Запрос узла дерева отрезков попадает в один из трех случаев, которые мы разбирали здесь на примерах. Случай 1: у узла нет общих индексов с запрашиваемым диапазоном. Случай 2: отрезок узла полностью находится в интервале запроса. Случай 3: отрезок узла содержит часть интервала запроса, но также содержит индексы, выходящие за его границы.

Прежде чем переходить к коду, я советую приостановиться, чтобы самостоятельно проработать еще пару примеров запросов. В частности, попробуйте запрос по диапазону 4–9. Вы заметите, что он требует прохождения двух длинных путей вниз по дереву. Это один из худших случаев: у корня дерева происходит разделение на два узла, затем по двум путям совершается обход до самого низа.

Убедитесь на дополнительных примерах, возможно в более крупных деревьях отрезков, что эти пути далее не разделяются. В итоге несмотря на то, что запрос к дереву отрезков продлевает немного больше работы, чем операция с кучей, обходя иногда два пути вместо одного, он все равно обращается к небольшому количеству узлов на каждом уровне, затрачивая на выполнение  $O(\log n)$  времени.



Код запроса к дереву отрезков дан в листинге 7.16.

**Листинг 7.16.** Запрос к дереву отрезков

```
int query_segtree(segtree_node segtree[], int node,
                  treap_node treap_nodes[], int left, int right) {
    int left_max, right_max;

    if (right < segtree[node].left || left > segtree[node].right) ❶
        return -1;

    if (left <= segtree[node].left && segtree[node].right <= right) ❷
        return segtree[node].max_index;

    left_max = query_segtree(segtree, node * 2, ❸
                             treap_nodes, left, right);
    right_max = query_segtree(segtree, node * 2 + 1, ❹
                              treap_nodes, left, right);

    if (left_max == -1)
        return right_max;
    if (right_max == -1)
        return left_max;
    if (treap_nodes[left_max].priority > treap_nodes[right_max].priority) ❺
        return left_max;
    return right_max;
}
```

Параметры функции аналогичны используемым в листинге 7.15, за исключением того что мы добавили индексы запроса `left` и `right`. Код обрабатывает каждый из трех случаев по очереди.

В случае 1 узел не имеет общих элементов с запросом. Он возникает, когда диапазон запроса заканчивается до начала отрезка узла либо когда диапазон запроса начинается после завершения отрезка узла ❶. В этом случае мы возвращаем `-1`, указывая, что этот узел не содержит индекс максимума.

В случае 2 отрезок узла полностью входит в диапазон запроса ❷. Следовательно, мы возвращаем индекс максимума отрезка этого узла.

Остается случай 3, когда отрезок узла частично пересекается с диапазоном запроса. Здесь мы выполняем два рекурсивных вызова: один для получения индекса максимума от левого потомка ❸ и второй для получения индекса максимума правого потомка ❹. Если один из них вернет `-1`, то мы возвращаем ответ другого. Если они оба вернут допустимые индексы, то выбирается тот, чей элемент больше ❺.

## Решение 2. Дерево отрезков

Наконец, осталось доработать наше первое решение (в частности, функцию `main` в листинге 7.9 и функцию `solve` в листинге 7.13) с использованием деревьев отрезков. Для этого потребуется немного: мы просто сделаем соответствующие вызовы к уже написанным функциям деревьев отрезков.

В листинге 7.17 содержится новая функция `main`.

**Листинг 7.17.** Функция `main` после добавления деревьев отрезков

```
int main(void) {
    static treap_node treap_nodes[MAX_NODES];
    static segtree_node segtree[MAX_NODES * 4 + 1]; ❶
    int num_nodes, i;
    scanf("%d ", &num_nodes);
    while (num_nodes > 0) {
        for (i = 0; i < num_nodes; i++) {
            treap_nodes[i].label = read_label(LABEL_LENGTH);
            scanf("%d ", &treap_nodes[i].priority);
        }
        qsort(treap_nodes, num_nodes, sizeof(treap_node), compare);
        init_segtree(segtree, 1, 0, num_nodes - 1); ❷
        fill_segtree(segtree, 1, treap_nodes); ❸
        solve(treap_nodes, 0, num_nodes - 1, segtree); ❹
        printf("\n");
        scanf("%d ", &num_nodes);
    }
    return 0;
}
```

Единственное, что добавилось, — это объявление дерева отрезков ❶, вызов для инициализации отрезков этого дерева ❷, вызов для вычисления индекса максимума каждого узла дерева ❸, а также новый аргумент для передачи дерева отрезков в функцию `solve` ❹.

Функция `solve` приведена в листинге 7.18.

**Листинг 7.18.** Решение задачи с помощью деревьев отрезков

```
void solve(treap_node treap_nodes[], int left, int right,
          segtree_node segtree[]) {
    int root_index;
    treap_node root;
    if (left > right)
        return;
    root_index = query_segtree(segtree, 1, treap_nodes, left, right); ❶
    root = treap_nodes[root_index];
    printf("(");
    solve(treap_nodes, left, root_index - 1, segtree);
```

```
printf("%s/%d", root.label, root.priority);  
solve(treap_nodes, root_index + 1, right, segtree);  
printf(" ");  
}
```

Здесь сделано всего одно существенное изменение: вызов `query_subtree` для реализации RMQ ❶.

Да уж, в этот раз нам пришлось изрядно поработать. Решение с деревом отрезков должно пройти все тестовые проверки в установленных временных рамках. И все же оно того стоило, потому что деревья отрезков находят применение в быстрых решениях очень широкого спектра задач.

## Деревья отрезков

Деревья отрезков могут использоваться под разными именами, включая «деревья сегментов», «турнирные деревья», «деревья порядковой статистики» и «деревья запросов диапазона». Кроме того, «дерево отрезков» используется и для обозначения совершенно другой структуры данных, чем та, которую мы здесь изучали. Возможно, выбрав определенную терминологию, я, сам того не зная, присоединился к определенному сегменту сообщества программистов.

И все же, как бы их ни называли, деревья отрезков обязательны к освоению, если вы изучаете алгоритмы или интересуетесь спортивным программированием. В массиве из  $n$  элементов можно создать дерево отрезков за время  $O(n)$  и выполнить запрос по диапазону за время  $O(\log n)$ .

В задаче «Построение декартовых деревьев» мы использовали деревья отрезков для нахождения максимума в интервале, но они применяются и для других запросов. Если для ответа на запрос требуется быстро совместить ответы на два подзапроса, то дерево отрезков будет удачным выбором. А как насчет запроса минимума по диапазону? С помощью дерева отрезков вы можете получить минимальный (а не максимальный) из ответов потомков. А что насчет запроса суммы элементов в интервале? С помощью деревьев отрезков вы можете легко получить сумму на основе ответов потомков.

Вам, вероятно, интересно, могут ли применяться деревья отрезков в случаях, когда элементы внутреннего массива изменяются в ходе выполнения программы. К примеру, в «Построении декартовых деревьев» узел дучи никогда не менялся, поэтому дерево отрезков не могло рассинхронизироваться с данными, хранящимися в массиве.

Действительно, многие деревья отрезков формируют массив, предполагающий запросы, но не изменение. Однако у таких деревьев есть и дополнительное свойство,

позволяющее использовать их даже при возможном изменении внутреннего массива. В задаче 3 вы увидите, как это происходит, а также узнаете о новом виде запросов, который мы еще не встречали.

### Задача 3. Сумма двух элементов

На этот раз задача не будет иметь контекста — это просто задача для деревьев отрезков. Как вы увидите, нам понадобится поддерживать обновления массива, и интересующий нас запрос будет отличаться от запроса максимума по диапазону.

Рассмотрим задачу с платформы SPOJ с номером KGSS.

#### Условие

Дана последовательность целых чисел  $a[1], a[2], \dots, a[n]$ , где каждое число не меньше 0 (последовательность можно представить как массив, начинающийся с индекса 1, а не 0).

Возможны два типа операций для этой последовательности:

- **Обновление.** Получая числа  $x$  и  $y$ , изменять значение  $a[x]$  на  $y$ .
- **Запрос.** Получая числа  $x$  и  $y$ , возвращать максимальную сумму двух элементов в диапазоне от  $a[x]$  до  $a[y]$ .

#### Входные данные

Входные данные содержат один тестовый пример, состоящий из следующих строк:

- $n$  — количество элементов в последовательности. Значение  $n$  лежит в диапазоне между 2 и 100 000.
- $n$  целых чисел, каждое из которых указывает один элемент последовательности по порядку от  $a[1]$  до  $a[n]$ . Значение каждого числа больше или равно 0.
- $q$  — количество операций над последовательностью. Значение  $q$  находится в интервале между 0 и 100 000.
- $q$  строк, каждая из которых указывает одну операцию обновления или запроса для выполнения над последовательностью.

В последних  $q$  строках используются следующие обозначения:

- Обновление обозначается буквой U, после которой через пробелы приводятся целые числа  $x$  и  $y$ . К примеру, U 1 4 означает, что  $a[1]$  нужно изменить

с текущего значения на 4. Значение  $x$  находится между 1 и  $n$ ;  $y$  больше или равно 0. Данная операция не производит вывода.

- Операция запроса обозначается буквой **Q**, после которой через пробелы приводятся целые числа  $x$  и  $y$ . Таким образом указывается, что нужно вывести максимальную сумму двух элементов в диапазоне от  $a[x]$  до  $a[y]$ . К примеру, **Q 1 4** требует получить максимальную сумму двух элементов в диапазоне от  $a[1]$  до  $a[4]$ . Значения  $x$  и  $y$  находятся в интервале от 1 до  $n$ , при этом  $x$  меньше  $y$ .

### Выходные данные

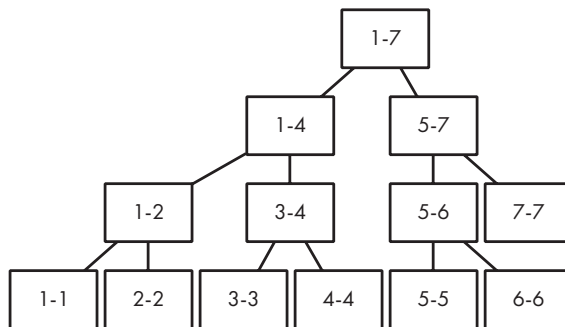
Следует выводить результат каждой операции запроса, по одному на строку.

Время на решение тестового примера — одна секунда.

### Заполнение дерева отрезков

В задаче «Построение декартовых деревьев» нам требовалось получать от дерева отрезков индексы, которые мы использовали в описании рекурсии и для разделения узлов дучи. На этот раз сохранять индексы в дереве отрезков бессмысленно, так как нас интересует сумма элементов, а не их индексы.

Мы инициализируем отрезки дерева так же, как делали это в разделе «Инициализация отрезков». Теперь отрезки нам нужны, чтобы начать охватывать массив с индекса 1, а не 0. В остальном же здесь все по-старому. На рис. 7.19 показано дерево отрезков, поддерживающее массив из семи элементов. Для соответствия условиям задачи используются индексы с 1 до 7, а не с 0 до 6.



**Рис. 7.19.** Дерево отрезков для массива из семи элементов

Теперь подумаем о том, как заполнить каждый узел максимальной суммой двух элементов его отрезка. Предположим, что мы уже нашли такую сумму для узлов 1–2

и 3–4. Нам нужно узнать максимальную сумму двух элементов для узла 1–4. Как это сделать?

Было очень удобно, когда мы решали RMQ, потому что максимум для узла — это просто максимум среди его детей. К примеру, если максимальное значение в левом поддереве равно 10, а в правом поддереве 6, то максимумом для их родительского узла будет 10. Здесь нет никаких сюрпризов. Но при использовании дерева отрезков с «максимальной суммой двух элементов» все не так просто.

Предположим, что у нас есть четыре последовательных элемента: 10, 8, 6 и 15. Максимальная сумма двух элементов в отрезке 1–2 равна 18, а максимальная сумма двух элементов в отрезке 3–4 равна 21. В таком случае ответом для отрезка 1–4 будет 18 или 21? Ни то ни другое. Ответом будет  $10 + 15 = 25$ . Но мы не можем получить сумму 25, если все, что нам известно, — это 18 слева и 21 справа. Нужно, чтобы левый и правый дочерние узлы сообщили нам больше сведений об их отрезке, а не просто выдали максимальную сумму двух элементов.

Для большей ясности скажу, что иногда достаточно получения просто максимальной суммы двух элементов от каждого потомка. Рассмотрим такую последовательность: 10, 8, 6 и 4. Максимальная сумма двух элементов отрезка 1–2 равна 18, а максимальная сумма двух элементов отрезка 3–4 равна 10. В этом случае максимальной суммой для 1–4 будет 18, которую мы получили от дочернего отрезка 1–2, но здесь нам просто повезло.

Для получения максимальной суммы двух элементов отрезка возможны всего три варианта (если у потомка узла нет валидной максимальной суммы, то вариантов еще меньше):

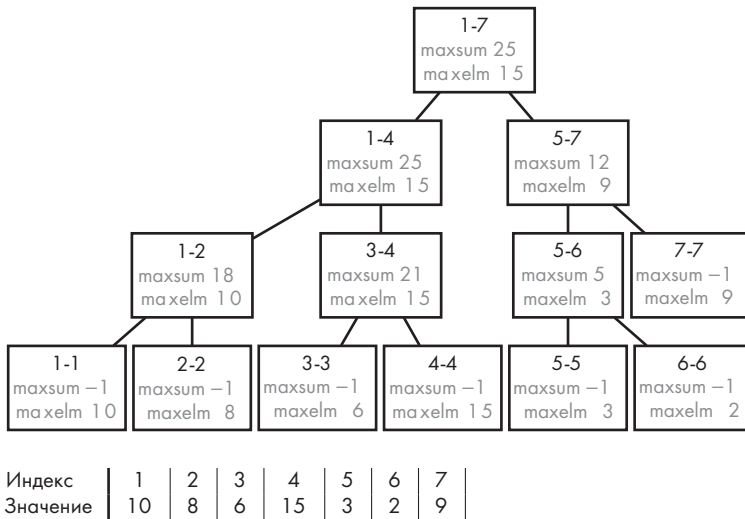
- **Вариант 1.** Максимальная сумма находится в левом потомке. Это подобно тому удачному примеру, который мы только что разобрали. В таком случае ответ приходит от левого потомка.
- **Вариант 2.** Максимальная сумма находится в правом потомке. Это другой удачный случай, когда ответ предоставляет правый дочерний узел.
- **Вариант 3.** Максимальная сумма включает один элемент из левого потомка и один из правого. Вот здесь уже нужно поработать дополнительно, потому что ответом не будет один из максимумов дочерних узлов. В этом случае от них понадобится дополнительная информация.

Если максимальная сумма двух элементов для отрезка складывается из одного элемента слева и одного элемента справа, то этими элементами должны быть максимальный слева и максимальный справа. Вернемся к последовательности 10, 8, 6 и 15. В получении максимальной суммы здесь задействуются элемент 10 слева и 15 справа. Обратите внимание, что это наибольшие элементы в левом и правом

отрезках соответственно. Невозможно добиться лучшего результата, используя иные элементы слева или справа.

Теперь мы видим, что именно узлы отрезков должны нам сообщать. Помимо максимальной суммы двух элементов нас также интересует значение максимального элемента. Обладая этими данными о дочерних отрезках, мы можем рассчитать сумму для родительского отрезка.

На рис. 7.20 показан пример дерева отрезков, построенного для массива. Обратите внимание, что каждый узел содержит и `maxsum` (максимальную сумму двух элементов) и `maxelm` (максимальный элемент).



**Рис. 7.20.** Дерево отрезков и соответствующий массив

Вычисление максимального элемента для каждого узла — это задача `RMQ`, которую мы решали при построении декартова дерева. Остается получить максимальную сумму для каждого узла. Для начала мы установим максимальную сумму для узлов с отрезками из одного элемента, например 1–1, 2–2 и т. п., равной особому значению (–1). Поскольку на этих отрезках только один элемент, –1 предупреждает, что максимальная сумма родителя не может быть максимальной суммой этого потомка.

Максимальная сумма для каждого другого узла устанавливается на основе максимальной суммы его потомков. Рассмотрим узел 1–7. Для определения его максимальной суммы возможны три варианта: можно взять максимальную сумму 25 слева, максимальную сумму 12 справа либо максимальный элемент 15 слева

и максимальный элемент 9 справа, получив  $15 + 9 = 24$ . Из всех вариантов наибольшим числом оказывается 25, значит, его мы и выбираем.

Мы предусматриваем особый случай фиктивных значений максимальной суммы  $-1$ , подчеркивая, что их нельзя использовать в качестве вариантов при определении максимальной суммы родительского узла. Обратите на это внимание в коде, который будет приведен далее.

Для узлов дерева отрезков мы используем следующую структуру:

```
typedef struct segtree_node {
    int left, right;
    int max_sum, max_element;
} segtree_node;
```

Еще одну структуру мы задействуем для результатов вызова функций `fill_segtree` и `query_segtree`:

```
typedef struct node_info {
    int max_sum, max_element;
} node_info;
```

`node_info` позволяет нам возвращать и максимальную сумму, и максимальный элемент. Возвращения одного целого числа без структуры было бы недостаточно.

Код для вычисления максимальной суммы и максимального элемента для каждого отрезка дан в листинге 7.19.

**Листинг 7.19.** Добавление максимальной суммы и максимального элемента

```
int max(int v1, int v2) {
    if (v1 > v2)
        return v1;
    else
        return v2;
}

node_info fill_segtree(segtree_node segtree[], int node,
                      int seq[]) {
    node_info left_info, right_info;

    if (segtree[node].left == segtree[node].right) { ❶
        segtree[node].max_sum = -1;
        segtree[node].max_element = seq[segtree[node].left];
        return (node_info){segtree[node].max_sum, segtree[node].max_element}; ❷
    }

    left_info = fill_segtree(segtree, node * 2, seq); ❸
    right_info = fill_segtree(segtree, node * 2 + 1, seq);
```



```
segtree[node].max_element = max(left_info.max_element, ❹  
                                right_info.max_element);  
  
if (left_info.max_sum == -1 && right_info.max_sum == -1) ❺  
    segtree[node].max_sum = left_info.max_element + ❻  
                            right_info.max_element;  
  
else if (left_info.max_sum == -1) ❼  
    segtree[node].max_sum = max(left_info.max_element +  
                                right_info.max_element,  
                                right_info.max_sum);  
  
else if (right_info.max_sum == -1) ❸  
    segtree[node].max_sum = max(left_info.max_element +  
                                right_info.max_element,  
                                left_info.max_sum);  
  
else  
    segtree[node].max_sum = max(left_info.max_element + ❸  
                                right_info.max_element,  
                                max(left_info.max_sum, right_info.max_sum));  
return (node_info){segtree[node].max_sum, segtree[node].max_element};  
}
```

Если отрезок содержит всего один элемент — это базовый случай рекурсии ❶. Тогда максимальной сумме назначается особое значение -1, которое указывает, что здесь нет валидной суммы двух элементов, а в качестве максимального устанавливается единственный элемент отрезка. Затем мы возвращаем максимальную сумму и максимальный элемент ❷.

В противном случае мы имеем рекурсию. Информация левого отрезка сохраняется в переменной `left_info`, а правого — в `right_info`. Каждая из этих переменных инициализируется посредством рекурсивного вызова ❸.

Как и говорилось, максимальный элемент отрезка — это наибольший элемент из максимального левого и максимального правого элементов ❹.

Теперь рассмотрим максимальную сумму двух элементов. Если ни один из детей не содержит максимальную сумму ❺, то мы понимаем, что их отрезки содержат всего по одному элементу. Следовательно, отрезок этого родителя содержит только два элемента, сложение которых станет единственным вариантом получения суммы ❻.

А что мы делаем, если левый дочерний узел содержит всего один элемент, а правый — больше одного элемента ❼? Что ж, теперь перед нами два варианта получения максимальной суммы родителя. Первый — сложить максимальные элементы каждой стороны. Второй — взять максимальную сумму из правого отрезка. Для выбора лучшего варианта мы используем функцию `max`. Случай, когда правый



```
    return ret_info;
}

else {
    ret_info.max_sum = max(left_info.max_element +
                          right_info.max_element,
                          max(left_info.max_sum, right_info.max_sum));
    return ret_info;
}
}
```

Структура этого кода аналогична структуре кода RMQ из листинга 7.16. Если отрезок узла не имеет общих элементов с диапазоном запроса, мы возвращаем -1 как для максимальной суммы, так и максимального элемента ❶. Такое значение означает отсутствие валидной информации.

Если отрезок узла полностью находится в диапазоне запроса ❷, то мы возвращаем значения максимальной суммы и максимального элемента.

И, наконец, если отрезок узла частично пересекается с диапазоном запроса, то мы следуем той же логике, что и при выводе информации об отрезке в листинге 7.19.

## Обновление дерева отрезков

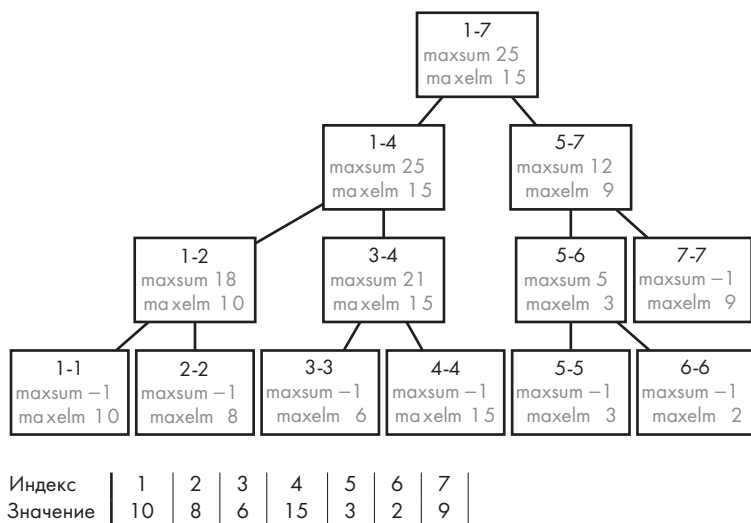
При обновлении элемента массива необходимо для согласованности также подстроить и дерево отрезков. В противном случае запросы к дереву будут использовать устаревшие элементы массива, давая результаты, которые не будут соответствовать актуальным данным.

Один из способов — это начать строить дерево с нуля и игнорировать находящуюся в отрезках информацию. Для этого можно повторно выполнять код листинга 7.19 при каждом обновлении элементов массива. Это определенно будет возвращать дерево отрезков в актуальное состояние, поэтому оно будет верным.

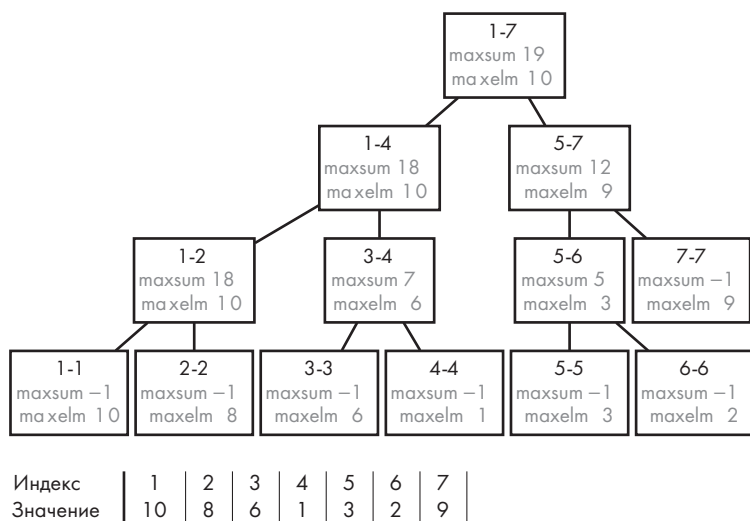
Однако такое решение не станет эффективным. Повторное создание дерева отрезков занимает  $O(n)$  времени. Поток из  $q$  операций обновления может существенно снизить производительность даже при отсутствии запросов. Если  $n$  работы будет проделано  $q$  раз, производительность составит  $O(nq)$ . Это особенно печально, если вспомнить, что обновления являются операциями над массивом, которые занимают постоянное время. Мы не можем позволить себе менять постоянное время на линейное, но *можем* перейти на логарифмическое, потому что последнее очень близко к первому.

Чтобы избежать работы в линейном времени, нужно понять, что при изменении элемента массива соответствующего обновления требует лишь небольшое число узлов в дереве отрезков. Разбор всего дерева ради одного обновления нецелесообразен.

Поясню на примере. Перед нами снова рис. 7.20:



Теперь представьте, что следующая операция — это  $U\ 4\ 1$ , то есть элемент с индексом 4 меняет свое значение с 15 на 1. Обновленные дерево отрезков и массив показаны на рис. 7.21.



**Рис. 7.21.** Дерево отрезков и его внутренний массив после обновления массива

Обратите внимание, что изменились всего три узла. Узел 4–4 должен измениться безусловно, потому что изменился единственный элемент в его отрезке. Однако влияние этого изменения не может распространиться слишком далеко и скажется только на предках 4–4. Действительно, в этом единственными изменившимися узлами являются три предка: 3–4, 1–4 и 1–7. Тогда в худшем случае мы можем двигаться из концевого узла дерева к корню, обновляя попутные узлы. Поскольку высота дерева оценивается как  $O(\log n)$ , этот путь содержит  $O(\log n)$  узлов.

Если мы не тратим время на рекурсию по неизменившимся частям дерева отрезков, то получаем процедуру обновления за время  $O(\log n)$ . Код приведен в листинге 7.21.

**Листинг 7.21.** Обновление дерева отрезков

```
node_info update_segtree(segtree_node segtree[], int node,
                        int seq[], int index) {
    segtree_node left_node, right_node;
    node_info left_info, right_info;

    if (segtree[node].left == segtree[node].right) { ❶
        segtree[node].max_element = seq[index];
        return (node_info) {segtree[node].max_sum, segtree[node].max_element};
    }

    left_node = segtree[node * 2];
    right_node = segtree[node * 2 + 1];
    if (index <= left_node.right) { ❷
        left_info = update_segtree(segtree, node * 2, seq, index); ❸
        right_info = (node_info){right_node.max_sum, right_node.max_element}; ❹
    } else {
        right_info = update_segtree(segtree, node * 2 + 1, seq, index);
        left_info = (node_info){left_node.max_sum, left_node.max_element};
    }

    segtree[node].max_element = max(left_info.max_element,
                                    right_info.max_element);

    if (left_info.max_sum == -1 && right_info.max_sum == -1)
        segtree[node].max_sum = left_info.max_element +
                                right_info.max_element;

    else if (left_info.max_sum == -1)
        segtree[node].max_sum = max(left_info.max_element +
                                    right_info.max_element,
                                    right_info.max_sum);

    else if (right_info.max_sum == -1)
```

```
segtree[node].max_sum = max(left_info.max_element +
                           right_info.max_element,
                           left_info.max_sum);

else
    segtree[node].max_sum = max(left_info.max_element +
                              right_info.max_element,
                              max(left_info.max_sum, right_info.max_sum));
return (node_info) {segtree[node].max_sum, segtree[node].max_element};
}
```

Эта функция будет вызываться *после* обновления элемента массива с индексом `index`. Вызов этой функции гарантирует, что `node` является корнем дерева отрезков, чей отрезок содержит элемент `index`.

Базовый случай — отрезок содержит всего один элемент **❶**. Поскольку мы не совершаем рекурсивный вызов, если `index` не находится в отрезке узла, нам известно, что этот отрезок содержит именно нужный индекс. Таким образом мы обновляем `max_element` узла на значение, хранящееся в `seq[index]`. Мы не обновляем `max_sum`, имеющую значение `-1`, потому что этот отрезок по-прежнему содержит всего один элемент.

Теперь предположим, что наш случай — не базовый. У нас есть узел, и мы знаем, что его элемент `index` был обновлен. Тогда нет причины совершать *два* рекурсивных вызова, поскольку только один из потомков узла может содержать обновленный элемент. Если `index` находится в левом потомке, мы сделаем рекурсивный вызов к левому потомку для обновления левого поддерева. Если же `index` окажется в правом потомке, мы рекурсивно вызовем правого потомка для обновления правого поддерева.

Чтобы определить, в каком потомке находится `index`, мы сравниваем его с большим индексом левого потомка. Если `index` входит в интервал левого потомка **❷**, то нужно сделать рекурсивный вызов слева. В противном случае нужно выполнить его справа.

Рассмотрим случай рекурсивного вызова для левой стороны **❸**. Ветка `else`, где выполняется рекурсивный вызов к правой стороне, аналогична. Мы делаем рекурсивный вызов, который обновляет левое поддерево и возвращает нам информацию для обновленного отрезка. Для правого поддерева мы оставляем старые значения **❹** — обновления здесь не происходит.

Остальная часть кода соответствует листингу 7.19.

## Функция `main`

Теперь мы готовы использовать для решения задачи доработанное дерево отрезков. Код функции `main` приведен в листинге 7.22.

**Листинг 7.22.** Функция main для считывания входных данных и решения задачи

```
#define MAX_SEQ 100000

int main(void) {
    static int seq[MAX_SEQ + 1];
    static segtree_node segtree[MAX_SEQ * 4 + 1];
    int num_seq, num_ops, i, op, x, y;
    char c;
    scanf("%d", &num_seq);
    for (i = 1; i <= num_seq; i++)
        scanf("%d", &seq[i]);
    init_segtree(segtree, 1, 1, num_seq);
    fill_segtree(segtree, 1, seq);
    scanf("%d", &num_ops);
    for (op = 0; op < num_ops; op++) {
        scanf(" %c%d%d ", &c, &x, &y);
        if (c == 'U') { ❶
            seq[x] = y;
            update_segtree(segtree, 1, seq, x);
        } else { ❷
            printf("%d\n", query_segtree(segtree, 1, seq, x, y).max_sum);
        }
    }
    return 0;
}
```

Единственное, что здесь стоит отметить, — это логика обработки операций. Если следующая операция будет обновлением ❶, то мы отвечаем обновлением элемента массива, а затем обновлением дерева отрезков. В противном случае операция является запросом ❷, и мы отвечаем запросом к дереву отрезков.

Время отправлять код на проверку. Судей точно порадует столь быстрое решение, основанное на дереве отрезков.

## Выводы

В этой главе мы научились реализовывать и использовать кучи, а также деревья отрезков. Как и любая другая полезная структура данных, они поддерживают относительно малое количество высокоэффективных операций. Очень редко бывает так, что структура данных сама по себе решает всю задачу. Чаще всего у вас уже есть алгоритм с неплохой скоростью, и структура данных помогает его ускорить. К примеру, наша реализация алгоритма Дейкстры в главе 5 уже работает хорошо, но если добавить к ней min-кучу, то результат окажется еще лучше.

Всякий раз, когда вам приходится повторять однотипную операцию, следует искать возможность оптимизировать алгоритм путем выбора подходящей структуры данных. Ищете заданные элементы в массиве? Помогут хеш-таблицы. Нужно найти

максимум или минимум? К вашим услугам кучи. Запрашиваете отрезки массива? Воспользуйтесь деревьями отрезков. А что насчет определения, находятся ли два элемента в одном наборе? На этот вопрос ответит следующая глава!

## Примечания

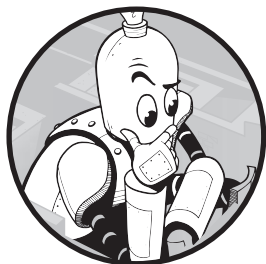
Задача «Акция в супермаркете» входила в программу 3-го этапа Польской олимпиады по информатике 2000 года (2000 Polish Olympiad in Informatics). Задача «Построение декартовых деревьев» предлагалась участникам соревнования Ульмского университета 2004 года (2004 Ulm University Local Contest). Задача «Сумма двух элементов» взята из программы онлайн-конкурса по программированию Курукшетры 2009 года (2009 Kurukshetra Online Programming Contest).

Для лучшего ознакомления с деревьями отрезков и многими другим структурами данных рекомендую посмотреть серию видеосюжетов *Algorithms Live!* в блоге Мэтта Фонтейна (Matt Fontaine) (<http://algorithms-live.blogspot.com>). Видео Мэтта о деревьях отрезков подсказало мне идею явно сохранять индексы отрезка `left` и `right` в каждом узле. В большинстве случаев в коде, который вам встретится, эти индексы будут передаваться в качестве параметров функции.



# 8

## Система непересекающихся множеств



В главах 4 и 5 для решения задач на графах мы использовали список смежности, применяя к нему необходимый алгоритм. Эта структура данных работает независимо от вида задачи на графах. Однако, ограничив спектр решаемых задач, можно спроектировать еще более эффективную структуру данных. Если такое ограничение будет незначительным, то более эффективного способа решения, чем список смежности, мы не получим. Если же спектр чрезмерно заузить, то специальную структуру данных никто не станет использовать, так как она будет подходить только под один вид задач. А вот если ограничение диапазона задач будет умеренным, то мы получим полезную структуру данных, такую как система непересекающихся множеств. Ей и будет посвящена текущая глава. Такая структура подходит для решения только некоторых видов задач на графах, но дает значительный выигрыш в скорости по сравнению с универсальными структурами.

Отслеживание сообществ в социальных сетях, поддержание групп друзей и недоброжелателей, а также распределение элементов по заданным ящикам — все это относится к задачам на графах. Важно то, что они являются именно теми задачами, которые можно решить очень быстро за счет применения системы непересекающихся множеств.

## Задача 1. Социальная сеть

Рассмотрим задачу с платформы SPOJ под номером SOCNETC.

### Условие

Требуется написать программу, отслеживающую связи людей в социальной сети.

Имеется  $n$  людей, которым присвоены уникальные номера: 1, 2, ...,  $n$ . Сообществом считается группа людей, связанных дружбой как напрямую, так и через друзей, друзей этих друзей и т. д. К примеру, если человек 1 и человек 4 — друзья, а человек 4 при этом дружит с человеком 5, то все трое являются членами одного сообщества.

Изначально каждый человек является единственным членом своего сообщества. Далее по мере знакомства с другими людьми его сообщество разрастается.

Программа должна поддерживать три операции:

- **Совмещение (Add).** Зафиксировать, что двое людей стали друзьями. Если оказывается, что ранее они состояли в разных сообществах, то теперь эти сообщества объединяются.
- **Проверка (Examine).** Сообщить, находятся ли заданные люди в одном сообществе.
- **Запрос размера (Size).** Сообщить количество людей в сообществе, в котором состоит заданный человек.

Дополнительный параметр  $m$  указывает максимально возможное число людей в сообществе. Необходимо игнорировать операцию совмещения, если в случае ее выполнения число людей в сообществе превышает  $m$ .

### Входные данные

Входные данные содержат один тестовый пример, состоящий из следующих строк:

- целое число  $n$ , указывающее количество людей в социальной сети, а также число  $m$ , указывающее максимальное число людей в сообществе. Значения  $n$  и  $m$  могут находиться в интервале от 1 до 100 000;
- целое число  $q$  — количество операций. Значение  $q$  лежит в интервале от 1 до 200 000;
- $q$  строк, по одной для каждой операции.

Формат строк операций:

- Совмещение:  $A\ x\ y$ , где  $x$  и  $y$  — номера людей.
- Проверка:  $E\ x\ y$ , где  $x$  и  $y$  — номера людей.
- Запрос размера:  $S\ x$ , где  $x$  — номер человека.

### Выходные данные

Выводятся только результаты операций проверки и запроса размера. Каждый результат должен выводиться на отдельную строку.

- Для операции проверки выводится **Yes**, если два человека состоят в одном сообществе, и **No** в противном случае.
- Для запроса размера выводится количество людей в сообществе.

Время на решение тестового примера — 0,5 секунды.

### Моделирование в виде графа

В главах 4 и 5 мы практиковались в решении задач через исследование графов. Мы выясняли, что использовать в качестве узлов, а что в качестве ребер, после чего применяли BFS или алгоритм Дейкстры.

Аналогичным образом можно смоделировать в виде графа и социальную сеть. Узлы — это люди в сети. Узлы друзей соединяют ребра. Граф будет ненаправленный, потому что дружба между двумя людьми — взаимная.

Ключевое отличие от задач из глав 4 и 5 состоит в том, что граф социальной сети будет динамическим. При каждой операции совмещения между людьми, еще не являющимися друзьями, в него будет добавляться новое ребро. Сравните этот подход с задачей «Перевод книги» из главы 4. Тогда нам изначально были известны все языки и переводчики, что позволяло построить граф единожды и после его не обновлять.

Давайте рассмотрим тестовый пример, чтобы наблюдать за ростом графа и выполнением операций совмещения, проверки и запроса размера:

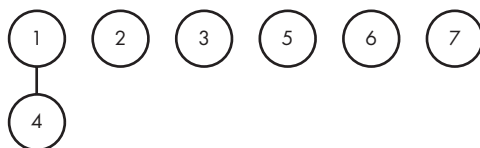
```
7 6
11
A 1 4
A 4 5
A 3 6
E 1 5
E 2 5
```

A 1 5  
A 2 5  
A 4 3  
S 4  
A 7 6  
S 4

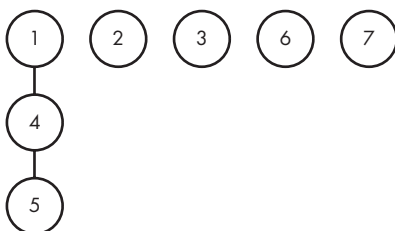
Мы начинаем с семи человек при отсутствии дружеских связей:



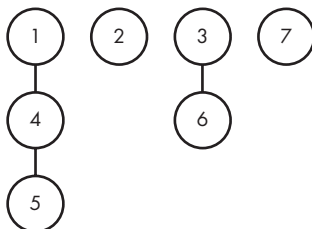
Операция A 1 4 делает друзьями людей 1 и 4, следовательно, эти узлы соединяются ребром:



Операция A 4 5 то же самое проводит с людьми 4 и 5:



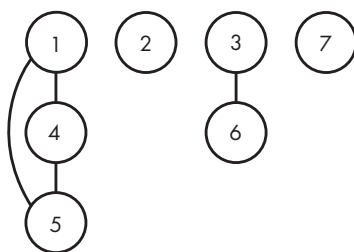
После A 3 6 получаем:



Следующая операция  $E\ 1\ 5$  спрашивает, находятся ли люди 1 и 5 в одном сообществе. Граф дает на это ответ: если от узла 1 до узла 5 (или от 5 к 1) существует путь, то они находятся в одном сообществе; в противном случае нет. В данной ситуации путь от узла 1 через узел 4 к узлу 5 позволяет ответить положительно.

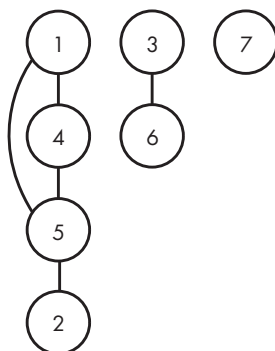
Следующая операция —  $E\ 2\ 5$ . Между узлами 2 и 5 пути нет, значит, эти двое людей не состоят в одном сообществе.

Операция  $A\ 1\ 5$  добавляет ребро между узлами 1 и 5 (обратите внимание, что мы чередуем операции, изменяющие граф, с операциями, которые его опрашивают):

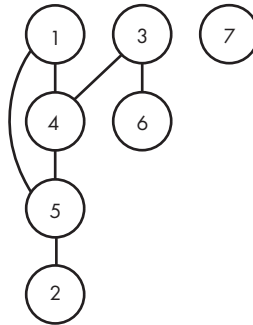


Добавление этого ребра сформировало цикл, потому что добавило связь между двумя людьми, которые уже находились в одном сообществе. Следовательно, новое ребро не оказывает влияния на количество сообществ или их размер. Можно было бы им пренебречь, но здесь я решил добавить все допустимые дружеские связи.

Операция  $A\ 2\ 5$  объединяет два сообщества:



Далее идет A 4 3, которая также объединяет два сообщества:



Теперь мы встречаем первый запрос размера: S 4. Сколько людей состоит в сообществе 4-го человека? Расчет сводится к определению количества узлов, достижимых из узла 4. Получается шесть узлов, при этом единственным недостижимым остается узел 7. Значит, ответ 6.

Далее рассмотрим A 7 6, добавляющую ребро между узлами 7 и 6... А вот сейчас интересно! Это ребро сформирует сообщество, охватывающее семь человек, но условия ограничивают допустимый размер сообществ шестью людьми. Значит, данную операцию совмещения нужно игнорировать.

В результате ответ на заключительный запрос S 4 будет тем же, что и ранее: 6.

Этот пример показывает, что необходимо для выполнения каждой из трех операций. Для совмещения мы добавляем в граф новое ребро при условии, что оно не создаст сообщество, превышающее установленный размер. При проверке определяем, достижим ли один узел из другого. Для этого можно использовать BFS. При запросе размера выясняем количество узлов, достижимых из заданного узла. И здесь тоже можно задействовать BFS!

## Решение 1. BFS

Реализуем это графовое решение в два шага. Сначала я покажу функцию `main`, обрабатывающую операции и постепенно создающую граф, а затем приведу код BFS.

### Функция `main`

Для начала нам понадобятся константа и структура:

```
#define MAX_PEOPLE 1000000
typedef struct edge {
```

```

    int to_person;
    struct edge *next;
} edge;

```

Функция `main` приведена в листинге 8.1. Она считывает входные данные и выполняет операции, пошагово создавая и опрашивая граф.

**Листинг 8.1.** Функция `main` для обработки операций

```

int main(void) {
    static edge *adj_list[MAX_PEOPLE + 1] = {NULL};
    static int min_moves[MAX_PEOPLE + 1];
    int num_people, num_community, num_ops, i;
    char op;
    int person1, person2;
    edge *e;
    int size1, size2, same_community;
    scanf("%d%d", &num_people, &num_community);
    scanf("%d", &num_ops);

    for (i = 0; i < num_ops; i++) {
        scanf(" %c", &op);

        if (op == 'A') { ❶
            scanf("%d%d", &person1, &person2);
            find_distances(adj_list, person1, num_people, min_moves); ❷
            size1 = size(num_people, min_moves); ❸
            same_community = 0;
            if (min_moves[person2] != -1) ❹
                same_community = 1;
            find_distances(adj_list, person2, num_people, min_moves); ❺
            size2 = size(num_people, min_moves); ❻
            if (same_community || size1 + size2 <= num_community) { ❼
                e = malloc(sizeof(edge));
                if (e == NULL) {
                    fprintf(stderr, "malloc error\n");
                    exit(1);
                }
                e->to_person = person2;
                e->next = adj_list[person1];
                adj_list[person1] = e;
                e = malloc(sizeof(edge));
                if (e == NULL) {
                    fprintf(stderr, "malloc error\n");
                    exit(1);
                }
                e->to_person = person1;
                e->next = adj_list[person2];
                adj_list[person2] = e;
            }
        }
    }
}

```

```

else if (op == 'E') { ❸
    scanf("%d%d", &person1, &person2);
    find_distances(adj_list, person1, num_people, min_moves);
    if (min_moves[person2] != -1)
        printf("Yes\n");
    else
        printf("No\n");
}

else { ❹
    scanf("%d", &person1);
    find_distances(adj_list, person1, num_people, min_moves);
    printf("%d\n", size(num_people, min_moves));
}
}
return 0;
}

```

Так же как в «Переводe книги» из главы 4 и задачах главы 5, здесь используется представление графа в виде списка смежности.

Разберем, как код обрабатывает каждую из трех операций, начиная с совмещения ❶. Мы вызываем вспомогательную функцию `find_distances` ❷, которая реализует BFS — заполняет `min_moves` значениями кратчайших путей в графе от `person1` до каждого человека, используя для недостижимых людей значение -1. Затем идет вызов вспомогательной функции `size` ❸, которая на основе информации о расстоянии из `min_moves` определяет размер сообщества `person1`. Далее мы выясняем, находятся ли `person1` и `person2` в одном сообществе: если `person2` достижим от `person1`, значит, результат проверки положительный ❹. Эта информация нужна, чтобы определить необходимость добавления ребра: если люди уже состоят в одном сообществе, то ребро можно смело добавлять, не опасаясь, что появится сообщество, выходящее за допустимые рамки максимального количества людей.

Определив размер сообщества `person1`, мы то же самое проделываем для сообщества `person2`: сначала вызываем BFS ❺, после чего вычисляем размер ❻.

Теперь, если новых сообществ нет либо размер нового сообщества не превышает допустимой величины ❼, мы добавляем в граф ребро. По факту мы добавляем даже два ребра, потому что граф у нас ненаправленный.

Остальные операции проще. Для операции проверки ❸ мы выполняем BFS и проверяем, достижим ли `person2` от `person1`. При запросе размера ❹ мы выполняем BFS, а затем подсчитываем количество узлов, достижимых от `person1`.



## Код BFS

Нужный нам код BFS очень похож на тот, что мы писали для решения задачи «Перевод книги», только без учета стоимостей перевода. Смотрим листинг 8.2.

### Листинг 8.2. Вычисление минимального расстояния между людьми с помощью BFS

```
void add_position(int from_person, int to_person,
                 int new_positions[], int *num_new_positions,
                 int min_moves[]) {
    if (min_moves[to_person] == -1) {
        min_moves[to_person] = 1 + min_moves[from_person];
        new_positions[*num_new_positions] = to_person;
        (*num_new_positions)++;
    }
}

void find_distances(edge *adj_list[], int person, int num_people,
                  int min_moves[]) {
    static int cur_positions[MAX_PEOPLE + 1], new_positions[MAX_PEOPLE + 1];
    int num_cur_positions, num_new_positions;
    int i, from_person;
    edge *e;
    for (i = 1; i <= num_people; i++)
        min_moves[i] = -1;
    min_moves[person] = 0;
    cur_positions[0] = person;
    num_cur_positions = 1;

    while (num_cur_positions > 0) {
        num_new_positions = 0;
        for (i = 0; i < num_cur_positions; i++) {
            from_person = cur_positions[i];
            e = adj_list[from_person];

            while (e) {
                add_position(from_person, e->to_person,
                           new_positions, &num_new_positions, min_moves);
                e = e->next;
            }
        }

        num_cur_positions = num_new_positions;
        for (i = 0; i < num_cur_positions; i++)
            cur_positions[i] = new_positions[i];
    }
}
```

### Размер сообщества

Осталось написать лишь небольшую вспомогательную функцию `size`, которая будет возвращать количество людей в сообществе заданного человека. Функция приведена в листинге 8.3.

#### Листинг 8.3. Размер сообщества

```
int size(int num_people, int min_moves[]) {
    int i, total = 0;
    for (i = 1; i <= num_people; i++)
        if (min_moves[i] != -1)
            total++;
    return total;
}
```

Здесь предполагается, что массив `min_moves` уже заполнен функцией `find_distances`. Каждый человек, чье значение в `min_moves` не равно -1, является достижимым. Для суммирования всех достижимых людей используется переменная `total`.

Вот мы и реализовали решение на основе графа. Для каждой из  $q$  операций выполняется BFS. В худшем случае каждая операция добавляет в граф одно ребро, значит, каждый вызов BFS проделывает работу, пропорциональную максимальному значению  $q$ . Следовательно, мы получили алгоритм  $O(q^2)$ , иначе говоря, квадратичный.

В главе 4 было сказано, что важно не выполнять BFS слишком много раз. Предпочтительно совершать всего один его вызов. В принципе, допустимо сделать даже несколько вызовов. Например, при решении задачи «Погоня за пешкой» (с. 160) нам сошло с рук выполнение вызова BFS для каждой позиции пешки. Тот же принцип работал в алгоритме Дейкстры в главе 5, когда мы совершали наименьшее возможное число вызовов. И в данной ситуации совершение нескольких вызовов нам вроде бы не мешает. Так, мы решили задачу «Мышиный лабиринт» (с. 208), выполняя около 100 вызовов алгоритма Дейкстры. С излишне расточительным применением поиска по графу мы до этого момента не сталкивались.

Однако если вы отправите текущее решение на проверку, то получите ошибку «Time-limit Exceeded». Причем мы очень сильно выходим за установленные временные рамки. Я проверял на своем ноутбуке пример со 100 000 людей в социальной сети и 200 000 операций (общее число операций в равных долях разделено между тремя их видами). При таких входных данных наше решение на графе выполняется примерно две минуты. Поэтому вам предстоит знакомство с еще одной структурой данных — системой непересекающихся множеств, которая с тем же примером справится в 300 раз быстрее. Я не шучу! Это буквально монстр эффективности.

## Система непересекающихся множеств

Есть две причины, по которым решение с использованием BFS на графе для данной задачи не подходит. Во-первых, оно находит много лишних данных, определяя кратчайшие пути между людьми. К примеру, этот поиск сообщает нам, что наименьшее расстояние между людьми 1 и 5 равно двум, но кого это интересует? Все, что нам нужно знать, — находятся ли эти двое людей в одном сообществе. При этом нас не интересует ни цепочка их друзей, ни то, как они оказались в одном сообществе.

Во-вторых, это решение сохраняет слишком мало промежуточных результатов. Честно говоря, оно вообще ничего не сохраняет: BFS начинает каждый раз с нуля. Но только подумайте, насколько это затратно. К примеру, операция совмещения добавляет в граф всего одно ребро. Сообщества от этого не могут сильно измениться в сравнении с их предыдущим состоянием. BFS же, не используя полученную ранее информацию, при каждой операции повторно обрабатывает весь граф.

Следовательно, нужно придумать такую структуру данных, которая не сохраняет информацию о кратчайших путях и при возникновении новой дружеской связи продвигает только необходимую небольшую работу.

## Операции

Операция совмещения должна выполнять только объединение двух сообществ в одно: она ничего не делает, если итоговое сообщество превысит лимит людей или если два человека уже находятся в одном сообществе. Такой вид операции в мире алгоритмов называется *объединением*, и ее результатом является множество. По своей сути объединение заменяет два набора данных одним более обширным, в котором содержатся все их элементы.

Операция проверки сообщает нам, находятся ли два человека в одном сообществе. Чтобы выполнить ее, можно назначить одного человека из каждого сообщества как *представителя*. К примеру, сообщество, состоящее из людей 1, 4 и 5, может иметь в качестве представителя человека 4, а в сообществе, состоящем из людей 3 и 6, можно назначить представителем человека 3. Находятся ли люди 1 и 5 в одном сообществе? Да, потому что они имеют одинакового представителя (4). Находятся ли в одном сообществе люди 4 и 6? Нет, потому что у них разные представители (4 и 3). Определение представителя сообщества называется *поиском*. Мы можем реализовать операцию проверки с помощью двух поисков: найти представителя сообщества первого человека, затем найти представителя сообщества второго человека и сравнить их.

Итак, «совмещение» — это «объединение», а «проверка» — это «поиск» в получаемых в результате объединений множествах. На этих двух принципах и строится система непересекающихся множеств.

Если мы реализуем объединение и поиск, то сможем с легкостью отвечать и на запрос размера. Все, что мы будем делать, — это сохранять размер каждого сообщества, не забывая актуализировать данные при выполнении объединения. Таким образом, мы сможем отвечать на каждый запрос, возвращая размер соответствующего сообщества.

### Подход на основе массива

Мы можем попробовать использовать массив `community_of`, указывающий представителя для каждого сообщества. К примеру, если люди 1, 2, 4 и 5 находятся в одном сообществе, в другом сообществе состоят люди 3 и 6, а в третьем находится только человек 7, то массив будет выглядеть так:

Индекс	1	2	3	4	5	6	7
Значение	5	5	6	5	5	6	7

Для сообщества из одного человека выбирать представителя не приходится — им назначается его единственный участник. В сообществе из нескольких человек представителем может стать любой.

Используя эту схему, можно реализовать поиск за постоянное время. Все, что нужно, — это найти представителя интересующего человека:

```
int find(int person, int community_of[]) {  
    return community_of[person];  
}
```

Лучше уже не получится.

К сожалению, эта схема рухнет при выполнении объединения, поскольку придется изменять все записи о представителе одного из сообществ на представителя другого сообщества. Выглядеть это будет так:

```
void union_communities(int person1, int person2,  
                       int community_of[], int num_people) {  
    int community1, community2, i;  
    community1 = find(person1, community_of);  
    community2 = find(person2, community_of);  
    for (i = 1; i <= num_people; i++)
```

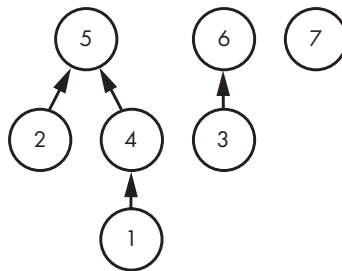
```
if (community_of[i] == community1)
    community_of[i] = community2;
}
```

Здесь я игнорирую ограничение размера сообществ нашей социальной сети, чтобы не отвлекаться от главной проблемы. В этом коде с помощью `find` устанавливаются `community1` и `community2` для представителей сообществ `person1` и `person2` соответственно. Затем цикл перебирает всех людей, перенося из `community1` в `community2`. В результате `community1` исчезает, поглощаемое `community2`.

Если вы сделаете и отправите полноценное решение на основе приведенного здесь кода, то увидите, что оно по-прежнему получает ошибку «Time-Limit Exceeded». Нам необходим более удачный способ объединения двух сообществ, чем перебор всех людей.

### Подход на основе дерева

Наиболее эффективная структура данных системы непересекающихся множеств основывается на деревьях. Каждый набор выражается как отдельное дерево, корень которого служит его представителем. Я поясню, как это работает, с помощью примера на рис. 8.1.



**Рис 8.1.** Структура непересекающихся множеств, основанная на дереве

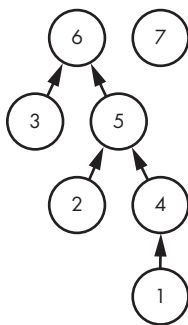
У нас имеются три дерева, то есть три отдельных сообщества: одно состоит из людей 1, 2, 4 и 5; второе — из людей 3 и 6; третье — из человека 7. Корни этих деревьев — люди 5, 6 и 7 — выступают как представители своих сообществ.

Ребра со стрелками указывают от потомка к родителю. Ранее такого в книге не встречалось. Сейчас же это подчеркивает особенность способа, которым мы будем обходить деревья. Когда я буду объяснять, как выполняются операции поиска и объединения, вы увидите, что по дереву необходимо перемещаться именно вверх и никогда вниз.

Начнем с поиска. Получив запрос о человеке, нам нужно вернуть его представителя. Это можно сделать, перемещаясь вверх по соответствующему дереву до его корня. Например, найдем на рис. 8.1 представителя человека 1. Узел 1 не является корневым, значит, мы переходим к его родителю — узлу 4, но он тоже не корневой, и мы переходим теперь уже к его родителю. Здесь мы оказываемся в корневом узле 5, на чем и останавливаемся — представителем человека 1 выступает человек 5.

Сравните эти переходы по дереву с тем, что нам приходилось делать при использовании массива. Вместо обычного поиска представителя в один шаг нам нужно перемещаться вверх по дереву до достижения его корня. Звучит неважно: что, если дерево окажется очень высоким? Однако вскоре станет ясно, что это опасение безосновательно, поскольку у нас будет возможность контролировать высоту дерева.

А теперь поговорим об объединении. Допустим, что нам нужно объединить два дерева. С точки зрения корректности не имеет значения, как именно мы их совместим. Однако, как только что было отмечено в контексте поиска, желательно сохранять дерево невысоким. Если мы поместим одно дерево в нижнюю часть другого, то без необходимости увеличим высоту итогового дерева. Чтобы этого избежать, мы поместим первое дерево прямо под корень второго. На рис. 8.2 я объединил деревья с корнями 5 и 6.



**Рис. 8.2.** Структура непересекающихся множеств после объединения

Корнем совмещенного дерева я решил сделать узел 6, но можно было выбрать и узел 5 (вот вам задачка: почему 5 окажется более удачным выбором? Ответ мы поймем, когда будем обсуждать оптимизацию системы непересекающихся множеств).

Теперь у нас есть все необходимое для построения решения задачи «Социальная сеть» на основе непересекающихся множеств.

## Решение 2. Система непересекающихся множеств

Вооруженные знаниями о кучах и деревьях отрезков из главы 7, вы не удивитесь тому, что мы будем сохранять структуру непересекающихся множеств в массиве.

Деревья непересекающихся множеств не обязательно являются двоичными, так как их узлы могут иметь любое число детей. Поэтому не получится перемещаться по таким деревьям путем умножения и деления на 2, как мы делали в главе 7. Но единственным маршрутом, который нам нужно поддерживать, является путь от потомка к его родителю. Для этого понадобится массив, сопоставляющий заданный узел с его родителем. Реализовать это можно с помощью массива `parent`, где `parent[i]` — родитель узла `i`.

Вспомним рис. 8.1, на котором изображены три сообщества: из людей 1, 2, 4 и 5; из людей 3 и 6; из человека 7. Вот массив `parent`, соответствующий рисунку:

Индекс	1	2	3	4	5	6	7
Значение	4	5	6	5	5	6	7

Кто представляет сообщество человека 1? По индексу 1 хранится значение 4, которое сообщает нам, что родителем 1 является 4. По индексу 4 хранится 5, указывая, что родителем 4 является 5. А в индексе 5 находится значение 5, означая, что 5 — это... родитель 5? Конечно же нет! Если `parent[i]` имеет то же значение, что и `i`, значит, мы достигли корня дерева (другой типичный прием для определения корня состоит в использовании значения -1, поскольку его нельзя спутать с допустимым индексом массива. Я не буду здесь прибегать к этому приему, но вы вполне можете столкнуться с ним в будущем).

### Функция `main`

Теперь мы готовы к написанию кода. Начнем с функции `main`, приведенной в листинге 8.4. Она намного короче, чем в листинге 8.1. Как правило, код для структуры непересекающихся множеств компактен.

#### Листинг 8.4. Функция `main` для обработки операций

```
int main(void) {
    static int parent[MAX_PEOPLE + 1], size[MAX_PEOPLE + 1]; ❶
    int num_people, num_community, num_ops, i;
    char op;
    int person1, person2;
    scanf("%d%d", &num_people, &num_community);
```

```

for (i = 1; i <= num_people; i++) { ❷
    parent[i] = i;
    size[i] = 1;
}
scanf("%d", &num_ops);

for (i = 0; i < num_ops; i++) {
    scanf(" %c", &op);

    if (op == 'A') {
        scanf("%d%d", &person1, &person2);
        union_communities(person1, person2, parent, size, num_community); ❸
    }

    else if (op == 'E') {
        scanf("%d%d", &person1, &person2);
        if (find(person1, parent) == find(person2, parent)) ❹
            printf("Yes\n");
        else
            printf("No\n");
    }

    else {
        scanf("%d", &person1);
        printf("%d\n", size[find(person1, parent)]); ❺
    }
}
return 0;
}

```

Помимо уже описанного массива `parent` имеется также массив `size` ❶. `size[i]` сообщает количество людей в сообществе представителя `i`. Никогда не ищите размер сообщества через человека, не являющегося представителем. Если задействовать не представителя, то мы не сможем обновить значение `size`.

Для инициализации `parent` и `size` используется цикл `for` ❷. Для `parent` мы позволяем каждому человеку быть своим собственным представителем, что соответствует присутствию каждого человека в его собственном множестве. Так как в начале каждое множество состоит из всего одного человека, мы устанавливаем каждое значение `size` равным 1.

Для операции совмещения мы вызываем вспомогательную функцию `union_communities` ❸. Она объединяет сообщества `person1` и `person2` с учетом ограничения размера `num_community`. Ее код будет приведен ниже.

Для операции проверки совершаются два вызова `find` ❹. Если они возвращают одно и то же значение, значит, люди находятся в одном сообществе; в противном случае — нет.



Для запроса размера мы используем массив `size`, в котором ищем представителя множества интересующего нас человека ⑤.

Далее я представлю функции `find` и `union_communities`, которые завершат общий вариант реализации.

### Функция `find`

Функция `find` получает в качестве параметра номер человека и возвращает его представителя. Смотрим листинг 8.5.

#### Листинг 8.5. Функция `find`

```
int find(int person, int parent[]) {
    int community = person;
    while (parent[community] != community)
        community = parent[community];
    return community;
}
```

Цикл `while` продвигается вверх по дереву, пока не находит корень. Этот корневой человек является представителем сообщества, и мы возвращаем его номер.

### Функция `union`

Функция `union_communities` получает номера двоих людей (помимо массива `parent`, массива `size` и ограничения `num_community`) и объединяет их сообщества. Я бы назвал эту функцию `union`, но это недопустимо, так как слово `union` в Си зарезервировано. Код приведен в листинге 8.6.

#### Листинг 8.6. Функция `union_communities`

```
void union_communities(int person1, int person2, int parent[],
                       int size[], int num_community) {
    int community1, community2;
    community1 = find(person1, parent); ①
    community2 = find(person2, parent); ②
    if (community1 != community2 && size[community1] +
        size[community2] <= num_community) {
        parent[community1] = community2; ③
        size[community2] = size[community2] + size[community1]; ④
    }
}
```

Код начинается с поиска представителя сообщества каждого из двух людей ① ②. Для объединения должны быть выполнены два условия: во-первых, сообщества должны быть разными; во-вторых, сумма размеров этих сообществ не должна

превышать допустимое значение. Если эти условия выполняются, то происходит объединение.

Я предпочел включить `community1` в `community2`. То есть `community1` исчезнет, будучи поглощенным `community2`. Чтобы это произошло, необходимо соответствующим образом изменить `parent` и `size`.

До этого объединения `community1` было корнем сообщества, но теперь нам нужно, чтобы его родителем стало `community2` ③. Любой человек, чьим представителем было `community1`, теперь получит в качестве представителя `community2`.

В массиве `size` необходимо отразить, что `community2` содержит всех людей, которые в нем были ранее, плюс всех людей, унаследованных из `community1`. Итак, новый размер рассчитывается как сумма того, что было до объединения, и размера `community1` ④.

Вот и все. Можете смело отправлять это решение на проверку. Теперь оно должно завершиться в установленное время и пройти все тесты.

...Хотя в глубине души я надеялся, что оно не впишется во временной лимит, потому что в запасе имеются две возможности оптимизации структуры непересекающихся множеств, которым я очень хочу вас научить.

А почему бы нам их не реализовать? Для данной задачи это может считаться излишним, но они обеспечивают настолько потрясающий прирост скорости, что мы будем применять их в других задачах главы, забыв о возможных проблемах с временными ограничениями.

## Оптимизация 1. Объединение по размеру

Наше решение уже работает быстро, но тестовые примеры можно подстроить так, что оно провалится. Вот пример худшего сценария:

```
7 7
7
A 1 2
A 2 3
A 3 4
A 4 5
A 5 6
A 6 7
E 1 2
```

Сообщества 1 и 2 сливаются, после чего итоговое сообщество сливается с сообществом 3, новое итоговое сообщество сливается с сообществом 4 и т. д. После шести операций объединения у нас получается дерево, показанное на рис. 8.3.

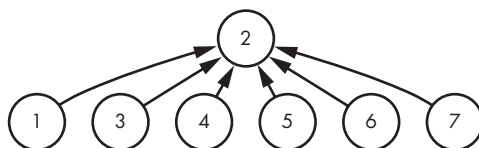


**Рис. 8.3.** Трудный случай структуры непересекающихся множеств на основе дерева

Получилась длинная цепочка узлов, так что операции поиска и объединения могут потребовать обхода всей этой цепи. К примеру,  $E\ 1\ 2$  вызовет поиск для людей 1 и 2, каждый из которых пройдет почти все узлы. Конечно же, цепочка из семи узлов невелика, но мы можем по той же схеме получать цепи большей длины. Таким образом, можно прийти к тому, что операции поиска и объединения будут занимать квадратичное время  $O(q^2)$ , где  $q$  — общее количество операций. Это означает, что в худшем случае основанное на дереве решение обеспечит такую же скорость, как и решение с BFS. На практике, конечно, оно будет быстрее, потому что большинство тестовых примеров не приводят к созданию длинных цепочек узлов... но некоторые смогут создать трудности!

Подождите-ка... Почему мы позволяем запугивать себя такими страшными деревьями? Нас же не волнует, как именно выглядит структура непересекающихся множеств. В частности, при объединении у нас есть возможность выбрать, кого назначить представителем нового объединенного сообщества. Вместо того чтобы всегда включать первое сообщество во второе, нам следует исходить из того, какой вариант создаст лучшее дерево. Сравните несусразицу на рис. 8.3 с идеальной структурой на рис. 8.4.

Можно сделать корнем человека 2, и все остальные узлы будут удалены от него всего на одно ребро. Теперь и объединение, и поиск можно выполнить очень быстро.



**Рис. 8.4.** Оптимизированная структура непересекающихся множеств на основе дерева

Каким образом наш код создаст структуру, представленную на рис. 8.4, вместо приведенной на рис. 8.3? Эта оптимизация называется *объединением по размеру*. Всякий раз, когда вы собираетесь объединить два сообщества, включайте сообщество с меньшим числом людей в сообщество с большим.

В рассматриваемом тестовом примере мы начинаем с A 1 2. В обоих этих сообществах по одному человеку, значит, неважно, какое из них мы сохраним, и пусть это будет сообщество 2. Теперь в сообществе 2 находится 2 человека: тот, что был изначально, и тот, что перешел из сообщества 1. Для выполнения A 2 3 мы сравниваем размеры сообществ 2 (два человека) и 3 (один человек). Далее мы сохраняем сообщество 2, потому что оно больше сообщества 3. Теперь в сообществе 2 состоят 3 человека. А что насчет A 3 4? Это слияние приводит к добавлению в сообщество 2 еще одного человека. Далее мы продолжаем, поглощая одно за другим сообщества из одного человека.

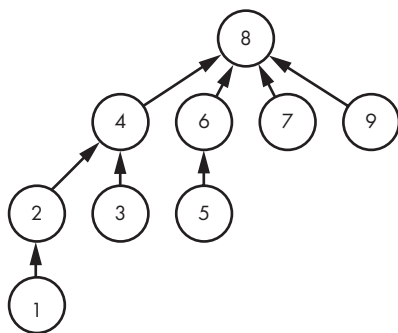
Объединение по размеру позволяет эффективно решать худшие тестовые примеры, но при этом все же остаются такие, которые требуют дополнительной работы для достижения корня. Вот пример:

```

9 9
9
A 1 2
A 3 4
A 5 6
A 7 8
A 8 9
A 2 4
A 6 8
A 4 8
E 1 5
  
```

Объединение по размеру дает структуру на рис. 8.5.

Несмотря на то что некоторые узлы правильно располагаются под корнем, есть и удаленные от него (хуже всех расположен узел 1). При этом дерево по-прежнему достаточно сбалансировано и определено лучше, чем длинная цепь узлов, которую мы наблюдали до оптимизации объединением по размеру.



**Рис. 8.5.** Трудный случай объединения по размеру

Далее я покажу, что максимальная высота дерева при использовании объединения по размеру составляет  $O(\log n)$ , где  $n$  — это общее количество узлов. Это означает, что и поиск, и объединение занимают  $O(\log n)$  времени, потому что поиск является простым проходом вверх по дереву, а объединение заключается в проведении двух таких поисков плюс изменение родителя.

Давайте выберем произвольный узел  $x$  и прикинем, сколько раз может увеличиваться количество ребер между  $x$  и его корнем. Когда сообщество  $x$  поглощает другое сообщество, количество ребер между  $x$  и его корнем не меняется, потому что корень остается прежним. Однако когда сообщество  $x$  поглощается другим сообществом, число ребер между  $x$  и его новым корнем становится на единицу больше пути к его бывшему корню. Следовательно, определение верхнего предела количества ребер между  $x$  и его корнем сводится к расчету максимального числа поглощений сообщества  $x$  другими сообществами.

Предположим, что размер сообщества  $x$  равен четырем. Может ли оно быть поглощено сообществом с размером два? Нет, поскольку мы используем объединение по размеру. Таким образом, сообщество  $x$  может быть поглощено другим сообществом только при условии, что второе сообщество будет иметь размер не меньше четырех. Значит, после поглощения мы получаем сообщество размером не менее  $4 + 4 = 4 \times 2 = 8$ . То есть размер сообщества  $x$  после поглощения другим сообществом по меньшей мере удваивается.

Если начальный размер  $x$  — единица, то после первого поглощения он находится в сообществе размером не менее двух. Это сообщество снова поглощается, и теперь его размер — не менее четырех. Очередное поглощение приводит к его увеличению до минимум восьми. Такое удваивание не может продолжаться бесконечно. Оно должно остановиться по крайней мере тогда, когда сообщество  $x$  будет содержать всех  $n$  людей. Начиная с одного, сколько раз можно удвоить размер сообщества до

достижения  $n$ ? Ответ —  $\log n$ , поэтому количество ребер между любым узлом и его корнем ограничивается величиной  $\log n$ .

Использование объединения по размеру сокращает время до логарифмического. И что еще лучше, для такой оптимизации не потребуется много нового кода. В нашем решении мы уже рассчитываем размеры сообществ, поэтому можно просто использовать эти данные для определения поглощаемого сообщества. Новый код дан в листинге 8.7. Если сравнить его с листингом 8.6, то очевидно, что мы почти не поменяли решение.

**Листинг 8.7.** Функция `union_communities`, использующая объединение по размеру

```
void union_communities(int person1, int person2, int parent[],
                      int size[], int num_community) {
    int community1, community2, temp;
    community1 = find(person1, parent);
    community2 = find(person2, parent);
    if (community1 != community2 && size[community1] +
        size[community2] <= num_community) {
        if (size[community1] > size[community2]) { ❶
            temp = community1;
            community1 = community2;
            community2 = temp;
        }
        parent[community1] = community2; ❷
        size[community2] = size[community2] + size[community1];
    }
}
```

По умолчанию `community2` поглощает `community1`. Это верное действие, если `community2` больше или равно `community1`. Если же `community1` будет больше `community2` ❶, то мы меняем их местами, чтобы изменить их роли. После этого `community2` гарантированно будет иметь больший размер, и мы сможем выполнить слияние `community1` с `community2` ❷.

## Оптимизация 2. Сжатие пути

Вернемся к тестовому примеру, в котором получалась структура на рис. 8.5, только на этот раз построим дерево, а затем будем выполнять одну и ту же операцию проверки:

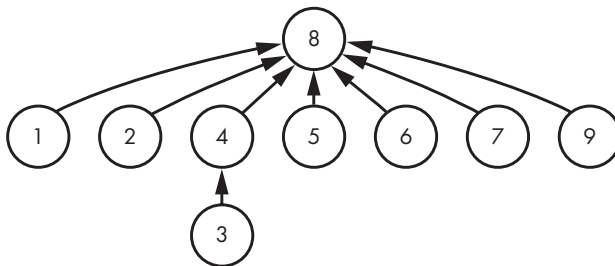
```
9 9
13
A 1 2
A 3 4
A 5 6
A 7 8
A 8 9
```

A 2 4  
A 6 8  
A 4 8  
E 1 5  
E 1 5  
E 1 5  
E 1 5  
E 1 5

Операция E 1 5 — медленная, так как каждый раз требует длительного прохода к корню. Например, для поиска представителя человека 1 мы переходим от узла 1 к 2, 4 и 8. Теперь мы знаем, что представителем узла 1 является узел 8. Свой обход надо проделать и для человека 5, причем эта информация теряется, потому что мы ее не сохраняем. Каждая операция E 1 5 вынуждает нас переделывать заново всю работу для поиска людей 1 и 5, чтобы еще раз узнать то, что уже было установлено ранее.

Давайте еще раз воспользуемся возможностью контроля структуры дерева. Вспомним, что конкретная форма дерева значения не имеет: важно лишь, что люди в одном сообществе представлены в одном дереве. Таким образом, как только мы нашли корень сообщества, можно переместить человека, для которого выполнялся поиск, на место дочернего узла найденного корня. Более того, предков этого человека можно сразу переместить туда же под корень.

Снова рассмотрим рис. 8.5 и предположим, что следующей операцией будет E 1 5. Если бы мы использовали только оптимизацию объединения по размеру, то эта операция проверки (как и любая операция проверки) не изменила бы структуру дерева. А теперь посмотрите, что происходит, если использовать оптимизацию *сжатия пути*, как показано на рис. 8.6.



**Рис. 8.6.** Пример сжатия пути

Эффектно, не так ли? Поиск для узла 1 ведет к тому, что узлы 1 и 2 становятся дочерними узлами корня. Поиск для узла 5 ведет к тому, что он также становится дочерним узлом корня. Процесс сжатия берет каждый узел на своем пути и делает

его дочерним узлом корня. Следовательно, последующий поиск для любого из этих узлов станет чрезвычайно быстрым.

Для реализации сжатия пути в функции `find` можно сделать два обхода от интересующего узла к корню. Первый обход обнаруживает сам корень. Этот обход выполняет любая функция `find`. Второй же обход делает каждый узел на своем пути дочерним узлом корня. В листинге 8.8 приводится соответствующий код. Сравните его с листингом 8.5.

**Листинг 8.8.** Функция `find` с реализацией сжатия пути

```
int find(int person, int parent[]) {  
    int community = person, temp;  
    while (parent[community] != community) ❶  
        community = parent[community];  
    while (parent[person] != community) { ❷  
        temp = parent[person];  
        parent[person] = community;  
        person = temp;  
    }  
    return community;  
}
```

Действие функции разделено на два этапа. Первый этап выполняется первым циклом `while` ❶, который приводит к тому, что в переменной `community` сохраняется представитель (корень) сообщества. На следующем этапе второй цикл `while` ❷ повторно обходит путь от `person` до дочерней позиции корня дерева, устанавливая корень в качестве предка `parent` каждого узла. В переменной `temp` сохраняется старый родитель текущего узла. Таким образом, мы можем переместиться к старому родителю текущего узла даже после того, как сделали его корнем дерева.

При использовании объединения по размеру совместно со сжатием пути все еще возможен вариант, когда небольшое число операций объединения или поиска будут занимать  $O(\log n)$  времени. Однако для большого числа объединений и поисков среднее время одной операции будет приближаться к постоянной величине. Время выполнения такого алгоритма описывается *обратной функцией Аккермана*, которая растет очень-очень медленно. Я не буду давать ее определение или показывать динамику роста, но мне хочется, чтобы вы поняли, какого результата мы добились.

Логарифмическая функция растет медленно, значит, начнем с нее. Двоичный логарифм от огромного числа имеет очень малое значение. К примеру, двоичный логарифм миллиарда равен примерно 30. Однако при использовании достаточно большого значения  $n$  вы можете добиться любой величины  $\log n$ .



Обратная функция Аккермана также непостоянна, но, в отличие от логарифмической функции, на практике вы никогда не получите даже значения около 30. Можете сделать  $n$  настолько большим, насколько захотите, пусть даже самым большим числом, какое может представить ваш компьютер, на что обратная функция Аккермана вернет вам значение не более 4. То есть вы можете рассматривать систему непересекающихся множеств с объединением по размеру и сжатием пути как алгоритм, выполняющий в среднем не более четырех шагов за операцию.

## Система непересекающихся множеств

Непересекающиеся множества кардинально ускоряют решение задач на графах, чьими основными операциями выступают объединение и поиск. Эта система не поможет в задачах, которые мы решали в главах 4 и 5, где нужно было вычислить расстояние между узлами. Если же система непересекающихся множеств оказывается применима в подобных задачах, то списки смежности и поиск по графу становятся излишними и слишком медленными.

### Три требования к связям

Система непересекающихся множеств эффективна для коллекций объектов, где каждый объект изначально расположен в собственном множестве. Объекты, находящиеся в одном множестве, всегда являются равнозначными с точки зрения конкретной решаемой задачи. К примеру, в задаче «Социальная сеть» люди, состоящие в одном множестве (сообществе), имеют равнозначные дружеские связи.

Система непересекающихся множеств требует, чтобы связи между объектами удовлетворяли трем критериям. Во-первых, объекты должны быть связаны с самими собой. В смысле дружеских отношений в социальной сети это означает, что каждый человек является собственным другом. Связь, которая отвечает этому критерию, называется *рефлексивной*.

Во-вторых, связь должна быть ненаправленной: не может быть так, чтобы  $x$  был другом  $y$ , но при этом  $y$  не был другом  $x$ . Связь, отвечающая этому критерию, называется *симметричной*.

В-третьих, связь должна быть каскадной: если  $x$  является другом  $y$ , а  $y$  является другом  $z$ , тогда  $x$  тоже будет другом  $z$ . Связь, отвечающая этому критерию, называется *транзитивной*.

Если один из этих критериев не выполняется, то операция объединения работать не будет. Предположим, что у нас есть дружеская связь, где отсутствует

транзитивность. Если мы узнаем, что  $x$  является другом  $y$ , то не будем знать, являются ли друзья  $x$  друзьями  $y$ . В результате люди могут попасть в одно множество, не будучи при этом друзьями.

Связь, отвечающая требованиям рефлексивности, симметрии и транзитивности, называется *отношением эквивалентности*.

## Применение системы непересекающихся множеств

Когда стоит вопрос о том, можно ли применить систему непересекающихся множеств, спросите себя: какую связь между объектами нужно будет поддерживать? Будет ли она рефлексивной, симметричной и транзитивной? Если да, и при этом основные операции можно описать как поиск или объединение, то эту структуру можно рассмотреть в качестве стратегии решения.

В основе каждой задачи с системой непересекающихся множеств лежит задача на графе, которую можно смоделировать, хотя и менее эффективно, с помощью списков смежности и поиска по графу. В отличие от того, что мы делали в задаче «Социальная сеть», в оставшихся задачах главы мы не будем прокладывать через графы хитроумные маршруты.

## Оптимизации

Мы рассмотрели два варианта оптимизации системы непересекающихся множеств: объединение по размеру и сжатие пути. Они обеспечивают защиту в случае трудных тестовых примеров и, как правило, увеличивают производительность независимо от того, какой пример решается. Каждая из этих оптимизаций требует всего нескольких строк кода, поэтому я советую применять их везде, где возможно.

Но «везде, где возможно» не следует путать со «всегда». К сожалению, есть и такие задачи для систем непересекающихся множеств, в которых эти оптимизации не подойдут. Мне еще не попадались задачи, где проблематичным являлось бы сжатие пути, но иногда необходимо помнить порядок, в котором множества объединяются. В таких случаях нельзя менять местами корни деревьев, используя объединение по размеру. Так, в задаче 3 вы увидите пример, в котором мы не сможем использовать эту оптимизацию.

## Задача 2. Друзья и враги

В задаче «Социальная сеть» мы рассматривали только одну разновидность операции совмещения:  $x$  и  $y$  являются друзьями;  $x$  и  $y$  ходят в одну школу;  $x$  и  $y$  живут в одном городе... Но возможны и другие виды совмещения информации. Например,

$x$  и  $y$  не являются друзьями. Хмм... вот это интересный вариант, который сообщает не о том, что  $x$  и  $y$  находятся в одном множестве, а о том, что они *не* находятся в одном множестве. Как в этом случае будет работать система непересекающихся множеств? Скоро вы это узнаете.

Рассмотрим задачу с платформы UV под номером 10158.

## Условие

Две страны находятся в состоянии войны. Вам разрешили присутствовать на мирных переговорах, где можно слушать, как пары людей беседуют друг с другом. На этих переговорах присутствует  $n$  людей, пронумерованных с 0 до  $n-1$ . Сначала вы не знаете о том, какие из пар являются дружескими (состоят из граждан одной страны), а какие враждебными (состоят из граждан воюющих стран). Ваша задача — записывать информацию о том, какие пары являются друзьями, а какие врагами, и отвечать на запросы согласно имеющимся на момент их получения данным.

Здесь потребуется поддержка четырех операций:

- **SetFriends** — записать, что два человека являются друзьями.
- **SetEnemies** — записать, что два человека являются врагами.
- **AreFriends** — сообщить, известно ли вам наверняка, что запрошенные люди — друзья.
- **AreEnemies** — сообщить, известно ли вам наверняка, что запрошенные люди — враги.

Дружба — это эквивалент связи: она рефлексивна ( $x$  является другом  $x$ ), симметрична (если  $x$  является другом  $y$ , то  $y$  является другом  $x$ ) и транзитивна (если  $x$  является другом  $y$ , а  $y$  дружит с  $z$ , то  $x$  также является другом  $z$ ).

Враждебность же симметрична (если  $x$  является врагом  $y$ , тогда  $y$  является врагом  $x$ ), но при этом не рефлексивна и не транзитивна.

Есть еще один важный нюанс о структуре друзей и врагов. Предположим, что у  $x$  есть друзья и враги, и у  $y$  тоже есть друзья и враги, после чего нам сообщают, что  $x$  и  $y$  враждуют. Что мы выяснили? В первую очередь, мы узнали, что  $x$  и  $y$  — враги, но это еще не все. Мы также можем заключить, что враги  $x$  являются друзьями всех людей во множестве  $y$  (предположим, что Алиса и Боб — враги, а Дэвид и Ева — друзья. Далее нам сообщают, что Алиса враждует с Дэвидом. Тогда можно сделать вывод, что Боб дружит с Дэвидом и Евой). Аналогичным образом можно заключить, что враги  $y$  являются друзьями всех участников множества  $x$ . Смысл изложенного можно выразить одним известным афоризмом: враг моего врага — мой друг.

### Входные данные

Входные данные содержат один тестовый пример, состоящий из следующих строк:

- $n$  — общее количество участников переговоров, число меньше 10 000.
- По одной строке для каждой операции.
- Три целых числа, первое из которых — 0, обозначающее конец тестового примера.

Каждая строка операции имеет одинаковый формат с кодом операции и номерами людей ( $x$  и  $y$ ):

- SetFriends: 1  $\times$   $y$ .
- SetEnemies: 2  $\times$   $y$ .
- AreFriends: 3  $\times$   $y$ .
- AreEnemies: 4  $\times$   $y$ .

### Выходные данные

Если операция требует вывода результата, то он выполняется на отдельную строку:

- В случае успеха операции SetFriends и SetEnemies не производят вывода. Если же такая операция противоречит уже известной информации, то выводится -1, а сама операция игнорируется.
- Операция AreFriends должна вывести 1, если два рассматриваемых человека являются друзьями, и 0 в противном случае.
- Операция AreEnemies должна вывести 1, если два рассматриваемых человека являются врагами, и 0 в противном случае.

Время на решение тестового примера — три секунды.

### Аугментация: враги

Если бы требовалось работать только с операциями SetFriends и AreFriends, то можно было бы напрямую применить систему непересекающихся множеств, как при решении задачи «Социальная сеть». Для каждой группы друзей тогда можно хранить по одному множеству. Аналогично операции совмещения, SetFriends будет реализовываться как объединение двух множеств друзей в более крупное. И аналогично операции проверки AreFriends будет реализовываться как поиск для каждого из двух людей для определения, находятся ли они в одном множестве.

Я уверен, что вы сможете прямо сейчас решить ограниченную задачу для этих двух операций без моей помощи. Я же сосредоточусь на объяснении техники встраивания в это решение операций `SetEnemies` и `AreEnemies`.

### Аугментация структуры непересекающихся множеств

*Аугментация* структуры данных означает ее дополнение информацией для поддержки новых или более быстрых операций. Примером аугментации является сохранение размера каждого множества в их непересекающейся структуре: можно, конечно, реализовать структуру данных и без этого, но так вы сможете быстро сообщать размеры множеств и выполнять объединение по размеру.

Аугментацию целесообразно использовать, когда существующая структура данных делает *почти* все, что вам нужно. Главное — определить подходящий вид аугментации, который добавит именно нужную функциональность без ощутимого замедления других операций.

У нас есть структура непересекающихся множеств, поддерживающая `SetFriends` и `AreFriends`. Она сохраняет данные о родителе каждого узла, а также размер каждого множества. Эту структуру данных мы аугментируем, дополнив операциями `SetEnemies` и `AreEnemies`. Причем это практически не замедлит операции `SetFriends` и `AreFriends`.

Предположим, нам сообщили, что  $x$  и  $y$  враги. Из условия задачи известно, что нужно объединить множество  $x$  с врагами  $y$ , а множество  $y$  — с врагами  $x$ . С кем враждует  $y$ , а с кем  $x$ ? При использовании стандартной структуры непересекающихся множеств мы этого не знаем. Именно поэтому ее нужно аугментировать.

Помимо родителя каждого узла и размера каждого множества, мы будем также отслеживать врагов каждого множества. Этих врагов мы будем сохранять в массиве `enemy_of`. Предположим, что  $s$  является представителем некоего множества. Если у этого множества нет врагов, тогда мы определяем для хранения в `enemy_of[s]` особое значение, которое нельзя спутать с человеком. Если же у этого множества есть один или более врагов, тогда `enemy_of[s]` будет давать нам одного из них.

Все верно: *одного* из них, а не *всех*. Достаточно знания одного врага каждого множества, потому что с его помощью можно найти представителя соответствующего множества.

Для подготовки к последующей реализации сначала проработаем два тестовых примера. Примите во внимание, что я покажу общую концепцию, которая в некоторых деталях может расходиться с программной реализацией. В частности, я не буду

демонстрировать объединение по размеру или сжатие пути, но с целью повышения производительности мы эти оптимизации добавим в код.

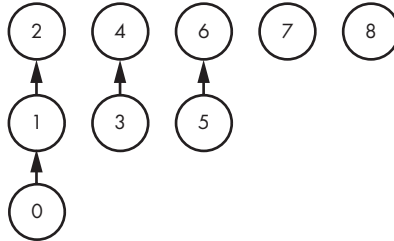
### Тестовый пример 1

Напомним, что операция `SetFriends` представлена кодом 1, а `SetEnemies` кодом 2.

Вот первый тестовый пример:

```
9
1 0 1
1 1 2
1 3 4
1 5 6
2 1 7 ❶
2 5 8 ❷
1 2 5 ❸
0 0 0
```

Первые четыре операции — `SetFriends`. Врагов пока ни у кого нет, значит, эти операции работают так же, как операция совмещения в задаче «Социальная сеть». На рис. 8.7 показано состояние структуры данных после их выполнения.



**Рис. 8.7.** Структура данных после четырех операций `SetFriends`

Далее идет первая операция `SetEnemies` ❶, которая указывает, что люди 1 и 7 являются врагами. Это означает, что все во множестве 1 стали врагами всех во множестве 7. Чтобы отобразить эту информацию в нашей структуре данных, мы добавляем связи между корнями этих двух множеств: связь от 2 (корень множества 1) к 7 и связь от 7 (корень множества 7) к 1 (вы могли подумать, что вместо последней должна быть связь от 7 к 2; все верно, она также вполне допустима). Результат выполненной операции показан на рис. 8.8. На этом и последующих рисунках связи врагов отмечены пунктирными линиями. В нашей реализации связи врагов будут храниться в форме вышеупомянутого массива `enemy_of`.

Следующей операцией будет `SetEnemies` между людьми 5 и 8 ❷. Выполнение этой операции приводит к схеме на рис. 8.9.

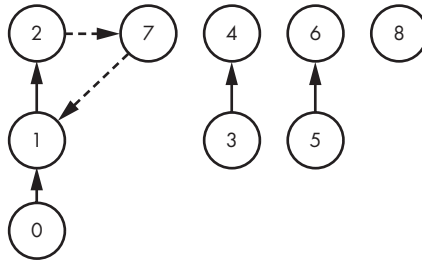


Рис. 8.8. Структура данных после операции SetEnemies

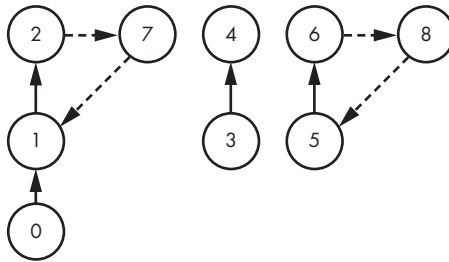


Рис. 8.9. Структура данных после еще одной операции SetEnemies

А теперь случай посерьезнее: заключительная операция ③, которая сообщает, что люди 2 и 5 — друзья. Таким образом, она объединяет множества 2 и 5 в одно множество друзей. Неожиданно здесь то, что мы также должны объединить два вражеских множества, поскольку свои враги есть и у множества человека 2, и у множества человека 5. Если мы знаем, что два человека принадлежат одной стране, то все множества их врагов должны относиться к другой стране. Результат выполнения этих двух операций объединения показан на рис. 8.10.

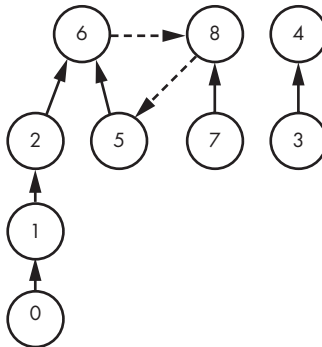


Рис. 8.10. Структура данных после заключительной операции SetFriends

Я не стал рисовать вражеские связи от человека 2 к человеку 7 и от человека 7 к человеку 1, так как мы поддерживаем такие связи только из корневых узлов. Как только узел перестает быть корневым, для обнаружения врагов мы его больше не используем.

Этот тестовый пример помог нам понять, что, во-первых, информация об одном враге множества сохраняется в корне этого множества; во-вторых, операция `SetFriends` требует двух объединений, а не одного. А что мы делаем, если у множества уже есть враг и это множество включается в операцию `SetEnemies`? В этом поможет разобраться второй тестовый пример.

### Тестовый пример 2

Этот тестовый пример будет отличаться от предыдущего только последней операцией:

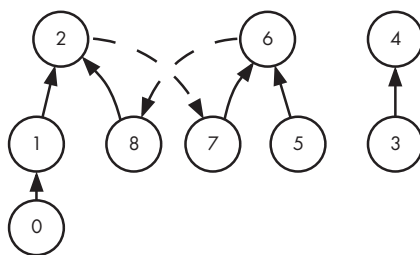
```

9
1 0 1
1 1 2
1 3 4
1 5 6
2 1 7
2 5 8
2 2 5 ❶
0 0 0

```

До последней операции структура данных выглядит, как показано на рис. 8.9. В этом примере последней операцией является `SetEnemies`, а не `SetFriends`. Множество человека 2 уже имеет врага, а теперь получает еще и новых противников из множества человека 5. Это значит, что нам нужно объединить врагов человека 2 с множеством человека 5. Аналогичным образом множество человека 5 уже имеет врага, а теперь получило новых врагов из множества человека 2. Следовательно, нужно также объединить врагов человека 5 с множеством человека 2.

Результат этих двух объединений показан на рис. 8.11.



**Рис. 8.11.** Структура данных после заключительной операции `SetEnemies`



Основу мы заложили, можно приступать к реализации.

## Функция *main*

Начнем с функции `main`, которая дана в листинге 8.9. Она считывает входные данные и вызывает вспомогательную функцию для каждой из поддерживаемых четырех операций.

**Листинг 8.9.** Функция `main` для обработки операций

```
#define MAX_PEOPLE 9999

int main(void) {
    static int parent[MAX_PEOPLE], size[MAX_PEOPLE];
    static int enemy_of[MAX_PEOPLE];
    int num_people, i;
    int op, person1, person2;
    scanf("%d", &num_people);
    for (i = 0; i < num_people; i++) {
        parent[i] = i;
        size[i] = 1;
        enemy_of[i] = -1; ❶
    }
    scanf("%d%d%d", &op, &person1, &person2);

    while (op != 0) {
        if (op == 1) ❷
            if (are_enemies(person1, person2, parent, enemy_of))
                printf("-1\n");
            else
                set_friends(person1, person2, parent, size, enemy_of);

        else if (op == 2) ❸
            if (are_friends(person1, person2, parent))
                printf("-1\n");
            else
                set_enemies(person1, person2, parent, size, enemy_of);

        else if (op == 3) ❹
            if (are_friends(person1, person2, parent))
                printf("1\n");
            else
                printf("0\n");

        else if (op == 4) ❺
            if (are_enemies(person1, person2, parent, enemy_of))
                printf("1\n");
            else
                printf("0\n");
    }
}
```

```

        scanf("%d%d", &op, &person1, &person2);
    }
    return 0;
}

```

Обратите внимание, что в процессе инициализации мы также устанавливаем каждое значение `enemy_of` равным -1 ❶. Это особое значение, обозначающее отсутствие врага.

Для реализации `SetFriends` ❷ сначала проверяем, являются ли два человека врагами. Если да, то выводим -1. Если нет, то вызываем вспомогательную функцию `set_friends`. Реализация `SetEnemies` ❸ происходит по тому же шаблону. Для `AreFriends` ❹ и `AreEnemies` ❺ мы вызываем вспомогательную функцию, определяющую, выполняется ли условие, и выводим 1 или 0.

## Поиск и объединение

Здесь я представлю функции поиска и объединения. Они будут вызываться вспомогательными функциями `SetFriends`, `SetEnemies`, `AreFriends` и `AreEnemies`. Функция поиска дана в листинге 8.10, а объединения — в листинге 8.11. Вот тут мы и применим сжатие пути при выполнении поиска, а также объединение по размеру при объединении.

### Листинг 8.10. Функция `find`

```

int find(int person, int parent[]) {
    int set = person, temp;
    while (parent[set] != set)
        set = parent[set];
    while (parent[person] != set) {
        temp = parent[person];
        parent[person] = set;
        person = temp;
    }
    return set;
}

```

### Листинг 8.11. Функция `union_sets`

```

int union_sets(int person1, int person2, int parent[],
               int size[]) {
    int set1, set2, temp;
    set1 = find(person1, parent);
    set2 = find(person2, parent);
    if (set1 != set2) {
        if (size[set1] > size[set2]) {
            temp = set1;
            set1 = set2;
            set2 = temp;
        }
    }
}

```

```

    }
    parent[set1] = set2;
    size[set2] = size[set2] + size[set1];
}
return set2; ❶
}

```

Функция объединения имеет одну особенность, не представленную ранее: она возвращает представителя получающегося множества ❶. Далее мы перейдем к операции SetFriends и вы увидите, что мы используем это возвращаемое значение.

## SetFriends и SetEnemies

Операция SetFriends реализована в листинге 8.12.

**Листинг 8.12.** Записываем, что два человека являются друзьями

```

void set_friends(int person1, int person2, int parent[],
                int size[], int enemy_of[]) {
    int set1, set2, bigger_set, other_set;
    set1 = find(person1, parent); ❶
    set2 = find(person2, parent); ❷
    bigger_set = union_sets(person1, person2, parent, size); ❸
    if (enemy_of[set1] != -1 && enemy_of[set2] != -1) ❹
        union_sets(enemy_of[set1], enemy_of[set2], parent, size); ❺
    if (bigger_set == set1) ❻
        other_set = set2;
    else
        other_set = set1;
    if (enemy_of[bigger_set] == -1) ❼
        enemy_of[bigger_set] = enemy_of[other_set];
}

```

Код начинается с определения представителей: `set1` представляет `person1` ❶, а `set2` — `person2` ❷. Поскольку эти два множества людей теперь должны быть друзьями, мы объединяем их в одно крупное множество ❸. Возвращаемое из `union_sets` значение мы сохраняем в `bigger_set`; вскоре мы его используем.

Мы объединили множества людей `person1` и `person2`, но это еще не все. Если вспомнить тестовый пример 1, то нам также может потребоваться объединить и некоторых врагов. В частности, если `set1` имеет врагов и `set2` тоже имеет врагов, то нужно объединить этих врагов в одно множество. Именно это делает наш код: если у обоих множеств есть враги ❹, мы объединяем множества этих врагов ❺.

Заманчиво представить, что на этом можно закончить решение. Мы выполнили объединения друзей и врагов. А представьте, что у `set1` есть враги, а у `set2` их нет. Следовательно, значение `enemy_of` у представителя `set2` будет -1. Теперь, возможно, `set1` будет включен в `set2`, так как `set2` больше. Если в этом месте мы закончим,

ничего дополнительно не предприняв, то `set2` не сможет найти своих врагов. Значение `enemy_of` для представителя `set2` по-прежнему равно `-1`, но это ошибка, потому что теперь у `set2` враги *есть*.

Вот как мы обработаем этот случай в коде. У нас уже есть `bigger_set`, указывающий, какое множество — `set1` или `set2` — получилось после объединения `set1` и `set2`. С помощью инструкции `if-else` мы устанавливаем `other_set` как другое множество ⑥: если `bigger_set` является `set1`, то `other_set` будет `set2` и наоборот. Тогда, если у `bigger_set` врагов нет ⑦, мы перекопируем вражескую связь из `other_set`. В результате `bigger_set` гарантированно сможет найти своих врагов, если `set1` или `set2` либо оба этих множества таковых имели.

Теперь настал черед операции `SetEnemies`, которая приведена в листинге 8.13.

**Листинг 8.13.** Записываем, что два человека являются врагами

```
void set_enemies(int person1, int person2, int parent[],
                int size[], int enemy_of[]) {
    int set1, set2, enemy;
    set1 = find(person1, parent);
    set2 = find(person2, parent);
    enemy = enemy_of[set1]; ①
    if (enemy == -1)
        enemy_of[set1] = person2; ②
    else
        union_sets(enemy, person2, parent, size); ③
    enemy = enemy_of[set2]; ④
    if (enemy == -1)
        enemy_of[set2] = person1;
    else
        union_sets(enemy, person1, parent, size);
}
```

Мы снова начинаем с поиска представителя каждого множества, сохраняя их в `set1` и `set2` соответственно. Далее выполняется поиск врага `set1` ①. Если у `set1` врагов нет, тогда мы устанавливаем в качестве его врага `person2` ②. Если у `set1` враг есть, тогда мы попадаем в ситуацию второго тестового примера. Мы объединяем врагов `set1` с множеством `person2` ③, гарантируя, что `person2` и все его друзья станут врагами `person1`.

Так мы разобрались с `set1`. Теперь аналогичное нужно проделать для `set2` ④, установив его врагом `person1`, если у него еще нет врагов, либо объединив его врагов с множеством `person1`.

Важно отметить, что эта функция поддерживает симметрию связей между врагами: если по `person1` можно найти его врага — `person2`, тогда по `person2` можно найти `person1`. Рассмотрим операцию `set_enemies` для `person1` и `person2`. Если у `person1`

врагов нет, то его врагом становится `person2`. Если же у `person1` враги есть, то их множество увеличивается за счет добавления `person2`. Симметрично, если у `person2` врагов нет, то его врагом становится `person1`. Если же у `person2` враги есть, то их множество увеличивается, включая `person1`.

## ***AreFriends и AreEnemies***

Операция `AreFriends` сводится к проверке, находятся ли два человека в одном множестве или, что равнозначно, имеют ли они одного представителя. Это можно реализовать через два вызова поиска, как показано в листинге 8.14.

**Листинг 8.14.** Проверка, являются ли два человека друзьями

```
int are_friends(int person1, int person2, int parent[]) {  
    return find(person1, parent) == find(person2, parent);  
}
```

Осталась всего одна операция! `AreEnemies` можно реализовать путем проверки, находится ли один человек в множестве врагов другого человека. Код дан в листинге 8.15.

**Листинг 8.15.** Проверка, являются ли два человека врагами

```
int are_enemies(int person1, int person2, int parent[],  
                int enemy_of[]) {  
    int set1, enemy;  
    set1 = find(person1, parent);  
    enemy = enemy_of[set1];  
    return (enemy != -1) && ❶  
           (find(enemy, parent) == find(person2, parent));  
}
```

Для того чтобы `person2` был врагом `person1`, должны выполняться два условия ❶. Во-первых, `person1` должен иметь врага. Во-вторых, `person2` должен находиться во множестве его врагов.

Стоп. А не нужно ли также проверить, является ли `person1` врагом `person2`? Нет, этого не требуется, потому что связь между врагами симметрична. Если `person2` не является врагом `person1`, то нет смысла проверять, является ли `person1` врагом `person2`.

Вот и все! Мы успешно аугментировали чистую структуру непересекающихся множеств, включив в нее информацию о друзьях и врагах. Если отправите этот код на проверку, то он пройдет все тесты. А что насчет лимита времени выполнения? При использовании объединения по размеру и сжатия пути мы к нему даже не приблизимся.

### Задача 3. Уборка комнаты

В задачах «Социальная сеть» и «Друзья и враги» мы добились ускорения работы за счет использования и объединения по размеру и сжатия пути. Но в следующей задаче корни множеств нельзя будет менять для выполнения объединения по размеру. Предлагаю вам в процессе чтения условия подумать о причинах такого ограничения.

Рассмотрим задачу с платформы Kattis с кодовым номером `ladice`.

#### Условие

У Мирко есть  $n$  предметов, разбросанных по комнате, а также комод с  $d$  пустых ящиков. Предметы пронумерованы от 1 до  $n$ ; ящики пронумерованы от 1 до  $d$ . Каждый ящик может вместить не более одного предмета. Цель Мирко — поочередно брать по одному предмету и по возможности укладывать его в ящик. Если же это окажется невозможным, то предмет выбрасывается.

Для каждого предмета существуют всего два ящика, куда его можно положить: А и В (смысл этого условия в организованности, ведь мы не хотим положить конфеты с Хэллоуина вместе с муравьями). Например, для предмета 3 может подойти ящик А под номером 7 и ящик В под номером 5.

Для определения судьбы каждого предмета мы будем использовать следующие правила:

1. Если ящик А пуст, поместить предмет в ящик А и остановиться.
2. Если ящик В пуст, поместить предмет в ящик В и остановиться.
3. Если ящик А занят, переместить находящийся в нем предмет в другой подходящий этому предмету ящик. Если тот ящик тоже окажется занят, переместить находящийся там предмет в другой подходящий ему ящик и т. д. Если этот процесс завершится, поместить предмет в ящик А и остановиться.
4. Если ящик В занят, переместить находящийся в нем предмет в другой подходящий ящик. Если тот ящик тоже занят, переместить находящийся в нем предмет в другой подходящий ящик и т. д. Если этот процесс завершится, переместить предмет в ящик В и остановиться.
5. Если предмет не удастся поместить в ящик согласно описанным четырем правилам, то выбросить его.

Выполнение правил 3 и 4 может привести к поочередному перекладыванию предметов в другие подходящие им ящики.

## Входные данные

Входные данные содержат один тестовый пример, состоящий из следующих строк:

- Количество предметов  $n$  и количество ящиков комода  $d$ . Значения обоих параметров могут лежать в интервале от 1 до 300 000.
- $n$  строк, по одной для каждого предмета. Каждая строка содержит два целых числа  $a$  и  $b$ , указывающих ящики  $A$  и  $B$  соответственно. При этом  $a$  не равно  $b$ .

## Выходные данные

Результат для каждого предмета выводится в отдельную строку. Если предмет удалось поместить в ящик, следует вывести слово LADICA, если нет — SMECE (эти выражения взяты из оригинального условия задачи: *ladica* переводится с хорватского как «ящик», а *smese* — «мусор»).

Время на решение тестового примера — одна секунда.

## Равнозначные ящики

Разберем один интересный сценарий. Мы кладем новый предмет в ящик 1, который, к сожалению, оказывается занят. Вторым ящиком для находящегося в первом предмета является ящик 2. Тогда мы пробуем переложить этот предмет в ящик 2, который оказывается занят. Вторым ящиком для лежащего в ящике 2 предмета является ящик 6, но и он тоже оказывается полон. Мы перекладываем находящийся в нем предмет во второй подходящий для этого ящик — 4. Ура, ящик 4 оказывается пуст, и мы на этом останавливаемся.

В ходе всего этого процесса, завершившегося заполнением ящика 4, мы переложили три уже имевшихся предмета: из ящика 1 в 2, из 2 в 6 и из 6 в 4. Однако конкретно эти перемещения для нас значения не имеют. Нам лишь нужно знать, что ящик 4 в конечном итоге оказался заполнен.

До добавления нового предмета ящики 1, 2, 6 и 4 схожи тем, что, если попробовать положить в любой из них предмет, это приведет к заполнению ящика 4. В этом смысле данные ящики можно назвать равнозначными. К примеру, если положить предмет непосредственно в ящик 4, тогда он заполнится сразу. Если же положить предмет в ящик 6, то находящийся в нем предмет переместится в ящик 4. Этот паттерн сохраняется, если поместить предмет в ящик 2 или, как мы видели в начале, в ящик 1. Ящик 4 является пустым, в связи с чем может завершить цепочку операций. Если представить структуру непересекающихся множеств, то станет очевидным, что он будет представителем своего множества. Таким образом, представителями множеств всегда будут пустые ящики.

Чтобы все это конкретизировать, мы проработаем два тестовых примера. В первом результатом всегда будет LADICA: мы сможем положить каждый предмет в ящик. Во втором же примере мы встретим результат SMECE: некоторым предметам ящика не найдется.

### Тестовый пример 1

Рассмотрим тестовый пример:

```

6 7
1 2
2 6
6 4
5 3
5 7
2 5

```

Здесь у нас имеется семь ящиков, каждый из которых изначально пуст и находится в собственном множестве. Я буду располагать каждое множество на отдельной строке и выделять его представителя курсивом:

```

1
2
3
4
5
6
7

```

Самое время освежить в памяти правила из условия задачи. Для первого предмета ящиком А является 1, а ящиком В — 2. Ящик 1 пуст, значит, этот предмет помещается в него (на основе правила 1). Кроме этого, ящики 1 и 2 попадают в одно множество, так как если в ящик 1 или 2 положить предмет, то заполненным в итоге окажется ящик 2:

```

1 2
3
4
5
6
7

```



Обратите внимание, что представителем нового множества стал ящик 2. Использовать в качестве его представителя ящик 1 было бы неправильным: это бы ошибочно указывало, что ящик 1 пуст. Именно поэтому мы не будем применять здесь объединение по размеру: оно может привести к выбору неверного корня в качестве представителя получающегося множества.

Далее рассмотрим второй предмет: 2 6. Ящик 2 пуст, значит, мы кладем этот предмет в него (снова работает правило 1). Теперь размещение предмета в ящиках 1, 2 или 6 приведет к заполнению ящика 6, значит, мы объединяем ящики 1 и 2 с ящиком 6:

1 2 6

3

4

5

7

Ящик 6 пуст, то есть непосредственное размещение в нем предмета заполняет его за один ход. Если же предмет сначала кладется в ящик 2, то лежащий там предмет перемещается в ящик 6 с тем же результатом. Размещение предмета в ящике 1 приводит к перекладыванию имеющегося предмета из ящика 1 в ящик 2, а предмета, лежащего в ящике 2, — в ящик 6, то есть последний и в этом случае заполняется. Это объясняет необходимость объединения всех этих ящиков в одно множество с ящиком 6 в качестве представителя.

Следующим идет предмет 6 4. Здесь мы снова используем правило 1:

1 2 6 4

3

5

7

Далее — предмет 5 3. Опять же, проблем не возникает и применяется правило 1:

1 2 6 4

5 3

7

До этого момента нам удавалось успешно обрабатывать каждый предмет, используя правило 1. Следующий же предмет 5 7 демонстрирует иной случай. Правило 1 здесь неприменимо, потому что ящик 5 уже заполнен. Тем не менее применяется правило 2, так как ящик 7 пуст. Следовательно, рассматриваемый предмет кладется

в него. Пустым ящиком в объединенном множестве является 3, значит, он и будет его представителем:

1 2 6 4

5 7 3

Осталось обработать еще один предмет: 2 5. Подойдет ли здесь правило 1? Нет, потому что ящик 2 заполнен. Подойдет ли правило 2? Опять нет, так как ящик 5 тоже заполнен. Применимо ли правило 3? Да! Оно подходит, потому что во множестве ящика 2 есть пустой ящик (4). Как же нужно действовать?

В этом случае множество ящика 2 должно быть объединено с множеством ящика 5:

1 2 6 4 5 7 3

Объясню, почему это работает. Предмет 2 5 помещается в ящик 2, а уложенные ранее предметы перекладываются из ящика 2 в ящик 6 и из ящика 6 в ящик 4. Ящик 4 теперь оказывается заполнен, следовательно, представителем множества больше быть не может. По факту единственным пустым ящиком оказывается 3, и мы рассчитываем, что он сможет выступить представителем множества. Ящики 5, 7 и 3 точно должны находиться в одном множестве, так как размещение предмета в любом из них в конечном итоге заполняет ящик 3 (до появления предмета 2 5 они уже находились в одном множестве).

Остается объяснить, почему ящики 1, 2, 6 и 4 также должны находиться в множестве ящика 3. С ящиком 2 все понятно: размещение предмета в нем приводит к перекладыванию лежащего там предмета в ящик 5. Ящик 5 находится во множестве ящика 3, и дальнейшие действия нам понятны: в конечном итоге ящик 3 заполняется.

С ящиком 1 тоже вопросов нет: размещение предмета в нем приводит к перекладыванию уже находящегося там предмета в ящик 2, и отсюда можно использовать доказательство из предыдущего абзаца, что ящик 3 будет заполнен. Та же логика применяется к ящикам 6 и 4. Например, если поместить предмет в ящик 4, а затем «отменить» действия, произошедшие при заполнении ящика 2, то находящийся в ящике 4 предмет вернется в ящик 6, а находящийся там предмет переместится в ящик 2, и мы вновь окажемся в случае из предыдущего абзаца.

В этом тестовом примере каждый предмет размещается в ящике, значит, верным выводом будет:

LADICA  
LADICA  
LADICA  
LADICA  
LADICA  
LADICA

Попробуем сформулировать общий принцип. Предположим, что мы обрабатываем предмет  $x$   $y$  и этот предмет оказывается во множестве  $x$ . Тогда мы объединяем множества  $x$  и  $y$ , сохраняя в качестве представителя получающегося множества  $y$ .

Почему это верно? Подумайте, что произойдет, когда вы попытаете поместить предмет в объединенное множество, состоящее из бывших множеств  $x$  и  $y$ . Помещение его в один из ящиков множества  $y$  приведет к заполнению его представителем  $y$ , потому что во множество  $y$  мы не вмешивались. Если же поместить его в ящик из множества  $x$ , то заполнен окажется также его представитель  $y$ , потому что мы переместим находящийся в  $x$  предмет в  $y$ , оказавшись в том же случае размещения предмета в ящике множества  $y$ . Допустим еще один вариант: новый предмет помещается в ящик  $z$  (отличающийся от  $x$ ) множества  $x$ . От  $z$  к  $x$  ведет цепочка ящиков. Перемещая предметы по этой цепочке, мы в итоге заполним ящик  $x$ , что также впоследствии приведет к заполнению представителя  $y$ .

Что, если мы обрабатываем предмет  $x$   $y$ , и он в итоге оказывается во множестве  $y$ ? Тогда два множества меняются ролями. В частности, представителем объединенного множества становится представитель множества  $x$ .

## Тестовый пример 2

Теперь посмотрим, как возникает результат SMECE. Вот второй тестовый пример:

```

7 7
1 2
2 6
6 4
1 4
2 4
1 7
7 6

```

Результат обработки первых трех предметов — LADICA, и мы приходим к знакомому состоянию:

```

1 2 6 4
3
5
7

```

Следующий — предмет 1 4. Здесь мы впервые встречаем ситуацию, когда ящики А и В находятся в *одном* множестве. Следовательно, нет пустого ящика для данного множества. На основе правила 2 предмет заполняет ящик 4 (то есть получает

отметку LADICA), но не дает множества для объединения. Ящики 1, 2, 6 и 4 входят в новый вид состояния, когда становится невозможным успешно поместить предмет в любой из них. Попытка это сделать приведет к бесконечному циклу перемещений предметов. К примеру, попробуйте положить предмет в ящик 1. Лежащий в нем предмет мы перенесем в ящик 2, предмет из ящика 2 переложим в ящик 6, предмет из ящика 6 — в ящик 4, предмет из ящика 4 — в ящик 1, и так далее, пока не кончатся страницы книги.

Мы будем отмечать подобное состояние, устанавливая в качестве представителя такого множества 0:

1 2 6 4 0

3

5

7

Теперь мы оказываемся в опасной близости от SMECE. Если нам встретится очередной предмет, чьи ящики окажутся в этом множестве, то поместить его будет уже некуда. Смотрим на следующий предмет: 2 4. Можно ли поместить его в ящик 2? Нет, ящик заполнен. Может, в ящик 4? Нет, он тоже полон. Найдём ли мы пустой ящик, если проследуем по их цепочке от ящика 2? Нет. Есть ли цепочка ящиков, ведущая от ящика 4 до пустого? Тоже нет. Мы получаем результат SMECE.

Идем дальше: предмет 1 7. Его мы обработаем по правилу 2. В результате будет выполнено объединение (и сделана отметка LADICA), но будьте внимательны, потому что мы снова получили множество без пустого ящика:

1 2 6 4 7 0

3

5

Последним идет предмет 7 6, и снова отметка SMECE, так как ни одно из четырех правил не приведет к положительному результату: ящики 7 и 6 находятся в одном множестве, в котором нет пустого ящика.

Выводом для этого тестового примера будет:

LADICA  
LADICA  
LADICA  
LADICA  
SMECE  
LADICA  
SMECE

В рассмотренных тестовых примерах я не показал только использование правила 4. Рекомендую вам самостоятельно поработать с ним, прежде чем продолжать. В частности, можете убедиться, что при каждом его применении представителем объединенного множества будет 0.

Ну а теперь черед реализации в коде!

## Функция *main*

Я начну с функции *main*, считывающей каждый предмет из входных данных и выполняющей его обработку. Код приведен в листинге 8.16.

**Листинг 8.16.** Функция *main* для обработки предметов

```
#define MAX_DRAWERS 300000

int main(void) {
    static int parent[MAX_DRAWERS + 1];
    int num_items, num_drawers, i;
    int drawer_a, drawer_b;
    scanf("%d%d", &num_items, &num_drawers);
    parent[0] = 0; ❶
    for (i = 1; i <= num_drawers; i++)
        parent[i] = i;

    for (i = 1; i <= num_items; i++) {
        scanf("%d%d", &drawer_a, &drawer_b);

        if (find(drawer_a, parent) == drawer_a) ❷
            union_sets(drawer_a, drawer_b, parent); ❸

        else if (find(drawer_b, parent) == drawer_b) ❹
            union_sets(drawer_b, drawer_a, parent); ❺

        else if (find(drawer_a, parent) > 0) ❻
            union_sets(drawer_a, drawer_b, parent); ❼

        else if (find(drawer_b, parent) > 0) ❽
            union_sets(drawer_b, drawer_a, parent); ❾

        else
            printf("SMECE\n");
    }
    return 0;
}
```

Как обычно, массив *parent* сохраняет родителя каждого узла в структуре непересекающихся множеств. Предметы пронумерованы начиная с 1, значит, можно смело использовать представителя 0 для множеств ящиков, в которые нельзя положить

новый предмет. Мы определяем для 0 представителя 0 ❶, указывая, что это множество в начале является пустым.

А теперь взглянем на правила. Каждое из первых четырех правил реализуется одним вызовом `find` и одним вызовом `union`. Если ни одно из этих правил не применимо, то мы оказываемся в случае `SMECE`. Разберем каждое из правил по очереди.

Для применения правила 1 нам нужно знать, пуст ли `drawer_a`. Напомню, что каждое множество ящиков (за исключением «нулевого») содержит ровно один пустой ящик, и этот пустой ящик является представителем своего множества. Функция `find` возвращает представителя заданного множества. Сопоставляя эти два факта, мы видим, что `find` возвращает `drawer_a` именно тогда, когда он пуст ❷.

Если мы используем правило 1, то нужно объединить множества `drawer_a` и `drawer_b`. Для этого вызывается `union_sets` ❸. Но помните, что в качестве представителя объединенного множества нужно взять представителя `drawer_b`, потому что `drawer_a` теперь заполнен и в его множестве пустых ящиков не осталось. Для этого мы используем `union_sets`, которая не выполняет объединение по размеру. Благодаря этому гарантируется, что представитель второго передаваемого параметра (в рассматриваемом случае — `drawer_b`) станет представителем объединенного множества. Эта же функция отвечает за вывод сообщения `LADICA`.

Для применения правила 2 нам нужно знать, пуст ли `drawer_b`. Проверяется это также с помощью `find` ❹, после чего, если правило оказывается применимо, выполняется объединение ❺. На этот раз мы вызываем `union_sets` с ящиками в обратном порядке, чтобы представителем объединенного множества стал представитель `drawer_a`.

Для применения правила 3 нам нужно знать, есть ли в множестве `drawer_a` пустой ящик. В множестве всегда есть пустой ящик, если его представитель не установлен как 0. Для проверки этого условия используется `find` ❻: если `find` возвращает любого представителя, кроме 0, то в множестве есть пустой ящик. Если правило 3 применимо, то выполняется объединение ❼. В следующем подразделе вы увидите, как `union_sets` выполняет перенос в множество 0.

Наконец, для применения правила 4 требуется знать, есть ли пустые ящики в множестве `drawer_b`. Здесь используется та же логика, что и для правила 3: с помощью `find` проверяется наличие пустого ящика в множестве ❸; если таковой имеется, выполняется объединение ❾.

## Поиск и объединение

Функция поиска приведена в листинге 8.17, где также реализовано сжатие пути. И оно здесь весьма кстати, так как я отправлял решение без этой оптимизации и получил ошибку `Time_limit Exceeded`.

**Листинг 8.17.** Функция `find`

```
int find(int drawer, int parent[]) {
    int set = drawer, temp;
    while (parent[set] != set)
        set = parent[set];
    while (parent[drawer] != set) {
        temp = parent[drawer];
        parent[drawer] = set;
        drawer = temp;
    }
    return set;
}
```

Функция объединения приведена в листинге 8.18.

**Листинг 8.18.** Функция `union_sets`

```
void union_sets(int drawer1, int drawer2, int parent[]) {
    int set1, set2;
    set1 = find(drawer1, parent);
    set2 = find(drawer2, parent);
    parent[set1] = set2; ❶
    if (set1 == set2) ❷
        parent[set2] = 0; ❸
    printf("LADICA\n");
}
```

Как и было анонсировано, объединение по размеру здесь не используется: в качестве нового множества всегда используется множество элемента `drawer2`, а именно `set2` ❶.

Кроме этого, всякий раз при размещении предмета, чьи ящики находятся в одном множестве ❷, мы устанавливаем представителя получающегося множества как 0 ❸. При всех последующих вызовах `find` для любого элемента этого множества будет возвращаться 0, указывая, что предметы в него больше поместить нельзя.

Вот оно: 50-строчное решение на базе системы непересекающихся множеств для одной из самых сложных задач книги. Можете отправлять его на проверку!

## Выводы

В этой главе мы научились эффективно использовать систему непересекающихся множеств. Из всех структур данных, изученных нами в книге, меня особенно поражают примеры применения именно этой структуры. «Вы серьезно? Это задача для системы непересекающихся множеств?» — такие мысли зачастую посещают меня. Возможно, и у вас возникали сомнения, когда мы решали задачи «Друзья и враги» и «Уборка в комнате». Как бы то ни было, в будущем вы наверняка

встретитесь с задачами, которые сперва покажутся совсем непохожими на продемонстрированные здесь, но на деле будут решаться именно с помощью системы непересекающихся множеств.

Широкие возможности применения и высокая скорость сочетаются с небольшим объемом кода: всего-то несколько строк для объединения и еще несколько для поиска. Кроме того, после освоения представления деревьев через массивы этот код уже не кажется излишне хитроумным. И для сопутствующих оптимизаций — объединения по размеру и сжатия пути — требуется лишь немного кода.

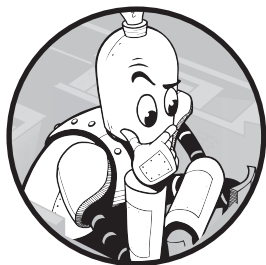
Думаю, что знакомство с одной из наиболее популярных структур данных является удачным завершением для книги.

## Примечания

Задача «Уборка в комнате» взята из программы 5-го тура Хорватского открытого соревнования по информатике 2013 года (2013 Croatian Open Competition in Informatics). Идею использования «представителя 0» я позаимствовал с сайта COCI ([http://hsin.hr/coci/archive/2013\\_2014](http://hsin.hr/coci/archive/2013_2014)).



## Послесловие



Цель этой книги — научить вас правильному пониманию структур данных и алгоритмов, а также их грамотному проектированию. На этом пути мы изучили множество устоявшихся концепций компьютерной науки. Хеш-таблицы освобождают от ресурсоемких линейных поисков.

Деревья организуют иерархические данные. Рекурсия справляется с задачами, которые можно разделить на подзадачи. Мемоизация и динамическое программирование позволяют сохранить быстроту выполнения рекурсии в случаях, когда подзадачи пересекаются. Графы — обобщенные структуры, включающие в себя деревья. Поиск в ширину и алгоритм Дейкстры находят кратчайшие пути в графах, причем, поскольку графы являются обобщенными структурами, в качестве «путей» могут быть представлены разные данные. Двоичный поиск преобразует решение в проверку возможных ответов. Кучи ускоряют поиск минимальных и максимальных элементов. Деревья отрезков делают то же самое для других видов запросов. Система непересекающихся множеств ускоряет решение задач на графах с равнозначными множествами узлов. Неплохой получился список, и я надеюсь, что вы довольны полученными знаниями. Также хочется полагать, что вы стали лучше понимать, чем эти структуры данных и алгоритмы полезны, почему они так эффективно работают и чего мы можем добиться благодаря умению их проектировать.

Я написал эту книгу, чтобы увлечь вас анализом и проектированием структур данных и алгоритмов. Для этого я использовал задачи по программированию, которые, надеюсь, показались вам достаточно интересными (чтобы появилось

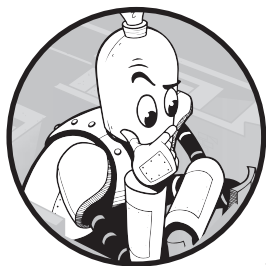
желание решить их) и достаточно трудными (чтобы для решения приходилось осваивать новые методы). Может быть, вас мотивировали сами задачи, а может, способы, которыми специалисты по информатике решают их. Возможно, вам уже не терпится заняться собственными задачами. Как бы то ни было, я рассчитываю, что помог вам выработать навыки и мотивацию для преследования важной цели.

Приятная особенность задач по программированию, подобных рассмотренным в книге, состоит в том, что они терпеливо ждут от нас своего решения. Они не меняются и не торопят нас. Когда мы застреваем на пути к решению, можно отвлечься, изучить что-то новое и вернуться позже, чтобы попробовать еще раз. Реальные задачи точно не будут содержать тщательно структурированные входные и выходные данные. Некоторые из их параметров могут даже меняться со временем. Мы сами должны выяснить, будут ли эти задачи столь же терпеливо дожидаться своего решения.

Я написал эту книгу с целью научить вас новому. Спасибо вам за доверие и время, потраченное на чтение того, чем я хотел поделиться.

# А

## Время выполнения алгоритма



В каждой решенной нами в книге задаче установлен временной лимит, в который должно уложиться выполнение программы. Если программа превысит этот лимит, вернется ошибка «Time-Limit Exceeded». Ограничение времени задается для того, чтобы отсеивать алгоритмически незрелые решения. Автор задачи держит в уме некоторые модели решений и устанавливает лимит времени в качестве фильтра, определяющего, в достаточной ли степени мы освоили эти идеи решений. Таким образом, наши программы должны получаться не только корректными, но еще и быстрыми.

### Оценка скорости выполнения... и не только

В большинстве книг по алгоритмам при обсуждении времени выполнения лимиты не устанавливаются. Почему тогда здесь временным лимитам уделено столько внимания? Основная причина в том, что подобные временные ориентиры помогают нам выработать интуитивное понимание эффективности наших программ. Мы можем запустить программу и замерить, насколько быстро она справляется. Если она не укладывается в установленный лимит времени, то мы понимаем, что требуется оптимизировать ее код или найти совершенно иной подход. Нам неизвестно, какой компьютер используют судьи, но выполнение программы на нашем собственном все равно оказывается информативным. Предположим, что мы запускаем программу на своем ноутбуке и решение одного из наименьших

тестовых примеров занимает 30 секунд. Если ограничение для этой задачи установлено в три секунды, то можно уверенно заключить, что наша программа недостаточно быстра.

Однако учитывать надо еще и следующее:

- **Время зависит от компьютера.** Как я только что отметил, оценка времени выполнения программы сообщает о том, как долго она выполняется на определенном компьютере. Это достаточно специфическая информация, которая дает нам довольно слабое представление о том, как программа будет работать на другой машине. В процессе изучения книги вы могли заметить, что время выполнения кода раз от раза отличается даже на одном компьютере. Вы можете выполнить программу на тестовом примере и выяснить, что на это уходит ровно 3 секунды, а затем еще раз выполнить ее на том же примере, и затрачено будет уже 2,5 или 3,5 секунды. Причина этого отличия в том, что распределением ресурсов машины руководит операционная система, перебрасывая их на разные задачи по необходимости. Принимаемые операционной системой решения по распределению ресурсов влияют и на время выполнения программ.
- **Время зависит от тестового примера.** Оценка скорости выполнения программы на тестовом примере сообщает лишь о том, как быстро программа справляется именно с этим набором данных. Предположим, что для выполнения небольшого примера ей требуется 3 секунды. Но тут нужно учитывать, что с небольшими примерами способно справиться любое разумное решение. Если я попрошу вас упорядочить несколько чисел, то вы сможете это быстро сделать с помощью первой же пришедшей в голову идеи. Все интересное начинается при работе с большими тестовыми примерами. Именно на них и проверяется эффективность алгоритма. Сколько времени вашей программе потребуется для обработки большого или даже огромного набора данных? Мы не знаем. Потребуется сначала проверить ее на таком наборе. И даже этого мало, так как бывают и очень специфичные тестовые примеры, которые существенно затрудняют достижение высокого быстродействия. В итоге легко впасть в заблуждение, считая свою программу более быстрой, чем она есть на самом деле.
- **Для проверки требуется реализация программы.** Нельзя замерить время выполнения того, что не реализовано. Предположим, что в ходе размышления над задачей у нас возникает идея по ее решению. Быстрое ли оно? Можно, конечно, реализовать его и проверить на практике, но было бы здорово оценить быстродействие наперед. Вы бы не стали реализовывать программу, заранее зная, что она получится ошибочной. То же касается и заведомого понимания низкой скорости работы программы.

- **Оценка скорости не объясняет причин медленной работы.** Если мы узнаем, что наша программа слишком медленная, то возникает задача написать более быструю версию. Однако простая оценка скорости никак не помогает понять причины медленного выполнения. Более того, если мы сможем придумать возможное улучшение, то, чтобы увидеть, решает ли оно проблему, потребуется снова его реализовать.
- **Время выполнения сложно обсуждать.** Есть много причин, по которым сложно использовать время выполнения для обсуждения алгоритмов. Оцените информативность фразы: «Моя написанная на Си программа выполняется за две секунды на купленном в прошлом году компьютере, на тестовом примере с восемью курицами и четырьмя яйцами. А как ваша?».

К счастью, специалисты computer science придумали метод оценки, который подходит там, где не годится замер времени. Такая оценка не зависит от комплектации компьютера, тестового примера и конкретной реализации программы. Она указывает, почему медленная программа работает медленно, и при этом легко поддается обсуждению. Называется она *нотацией «О-большое»*, и далее мы рассмотрим ее подробнее.

## Нотация «О-большое»

Причина привлекательности нотации «О-большое» состоит в том, что она определяет каждый алгоритм в один из классов эффективности. Достаточно знать класс эффективности, чтобы иметь представление о важных особенностях всех входящих в него алгоритмов. Здесь мы рассмотрим три таких класса: линейное время, постоянное время и квадратичное время.

### Линейное время

Предположим, что нам дан массив целых чисел в возрастающем порядке и нужно вернуть максимальное из них. К примеру, дан такой массив:

```
[1, 3, 8, 10, 21]
```

Нужно вернуть 21. Одно из решений — отслеживать максимальное значение. Всякий раз, находя превосходящее его число, мы будем обновлять максимум. Эта идея реализована в листинге А.1.

#### Листинг А.1. Поиск максимума в массиве возрастающих чисел

```
int find_max(int nums[], int n) {  
    int i, max;  
    max = nums[0];
```

```
for (i = 0; i < n; i++)
    if (nums[i] > max)
        max = nums[i];
return max;
}
```

Код устанавливает `max` на значение элемента с индексом 0 в массиве `nums`, а затем перебирает этот массив в поиске более крупных чисел. Не обращайте внимания на то, что вначале цикл сравнивает `max` с самим собой: это всего одна итерация ненужной работы.

Давайте представим количество выполняемой алгоритмом работы в виде функции от размера массива. Предположим, что в массиве пять элементов: что делает программа? Она выполняет одно присваивание переменной перед циклом, затем совершает 5 итераций в цикле и возвращает результат. Если в массиве 10 элементов, то программа делает то же самое, но в этот раз совершает не 5, а 10 итераций. А как насчет миллиона элементов? В этом случае программа выполняет миллион итераций. Теперь мы видим, что затраты на присваивание перед циклом и возвращение после цикла меркнут по сравнению с объемом работы, которую проделывает сам цикл. Важнее всего здесь именно количество итераций цикла, особенно при увеличении объема входных данных.

Если в массиве содержится  $n$  элементов, то цикл выполняет  $n$  итераций. В терминах «О-большого» мы говорим, что это алгоритм  $O(n)$ . Понимать это надо так: для массива из  $n$  элементов алгоритму требуется время, пропорциональное  $n$ . Алгоритм  $O(n)$  называется *алгоритмом с линейным временем*, потому что между размером задачи и временем ее выполнения есть линейная связь. Если этот размер удвоить, то удвоится и время выполнения. К примеру, если для обработки массива из двух миллионов элементов требуется одна секунда, то можно ожидать, что для массива из четырех миллионов потребуется две секунды.

Обратите внимание, что нам не пришлось выполнять код для того, чтобы это понять. Нам даже не потребовалось писать код (я его написал только для пояснения алгоритма). Если говорится, что алгоритм —  $O(n)$ , то это указывает на фундаментальную связь между размером задачи и временем ее выполнения. При этом неважно, какой компьютер используется или какой тестовый пример рассматривается.

## Постоянное время

Нам известно о наших массивах еще кое-что, что мы не использовали: их числа расположены по возрастанию. Таким образом, наибольшее значение будет всегда находиться в конце массива. Тогда можно просто напрямую возвращать это значение, вместо того чтобы находить его в результате поиска по всему массиву. Эта новая идея отражена в листинге А.2.

**Листинг А.2.** Поиск максимума в массиве из возрастающих чисел

```
int find_max(int nums[], int n) {  
    return nums[n - 1];  
}
```

Сколько работы проделывает этот алгоритм в зависимости от размера массива? Размер массива больше не имеет значения. Алгоритм обращается к последнему элементу, `nums[n-1]`, и возвращает его. При этом уже неважно, будет в массиве 5 элементов или миллион. В терминах «О-большого» такой алгоритм записывается как  $O(1)$  и называется *алгоритмом с постоянным временем*, потому что объем его работы постоянен и с ростом размера исходных данных не увеличивается.

Это лучший вид алгоритма. Независимо от размера массива, можно всегда ожидать одинаковое время выполнения, что однозначно лучше, чем выполнение в линейном времени, когда алгоритм с ростом задачи замедляется. Жаль только, что алгоритмы с линейным временем позволяют решить не так много интересных задач. К примеру, если дан массив с произвольным порядком элементов, а не возрастающим, то такой алгоритм уже не подойдет. В этом случае никак не получится находить максимум, обращаясь каждый раз к фиксированному элементу.

## Дополнительный пример

Взгляните на листинг А.3 и ответьте: это алгоритм  $O(n)$ ,  $O(1)$  или какой-то другой? Обратите внимание, что я отбросил определения функции и переменной, чтобы не было соблазна скомпилировать и выполнить этот код.

**Листинг А.3.** Что это за алгоритм?

```
total = 0;  
for (i = 0; i < n; i++)  
    total = total + nums[i];  
for (i = 0; i < n; i++)  
    total = total + nums[i];
```

Предположим, что в массиве `nums` находится  $n$  элементов. Первый цикл выполняется  $n$  раз, и второй тоже выполняется  $n$  раз. Всего получается  $2n$  итераций. Естественным будет предположить, что это алгоритм  $O(2n)$ . Но, соглашаясь, что технически так и есть, специалисты computer science проигнорировали бы 2 и просто написали  $O(n)$ .

Это может показаться странным, поскольку данный алгоритм вдвое медленнее приведенного в листинге А.1, но и тот и другой мы называем  $O(n)$ . Причина кроется в балансе между простотой и выразительностью оценки. Если оставить 2, то мы окажемся более точны, но снизим очевидность того, что это алгоритм с линейным

временем. Будь то  $2n$ , или  $3n$ , или любое другое количество  $n$ , сохраняется фундаментальный принцип линейного роста времени выполнения.

## Квадратичное время

Алгоритмы с линейным временем (на практике очень быстрые) и алгоритмы с постоянным временем (которые еще быстрее) мы рассмотрели. Теперь взглянем на вариант алгоритма, который медленнее линейного. Его код приведен в листинге А.4.

### Листинг А.4. Алгоритм с квадратичным временем

```
total = 0;
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        total = total + nums[j];
```

Заметьте, что, в отличие от листинга А.3, циклы теперь выполняются не последовательно, а вложены. Каждая итерация внешнего цикла вызывает  $n$  итераций внутреннего цикла, в результате чего `total` обновляется  $n^2$  раз (первая итерация внешнего цикла стоит  $n$  работы, вторая —  $n$  работы, третья — тоже  $n$  работы и т. д. Всего получается  $n+n+n+\dots+n$ , где количество прибавлений  $n$  равно  $n$ ).

Применительно к оценке «О-большое» мы говорим, что это алгоритм  $O(n^2)$ . Называется он *алгоритмом с квадратичным временем*, потому что квадратичность в математике подразумевает возведение в степень 2.

А теперь разберемся, почему алгоритмы с квадратичным временем медленнее алгоритмов с линейным временем. Предположим, что дан алгоритм с квадратичным временем, который выполняется за  $n^2$  шагов. Для задачи размером 5 потребуется  $5^2 = 25$  шагов. Для задачи размером 10 уже  $10^2 = 100$  шагов, а для задачи размером 20 получится  $20^2 = 400$  шагов. Обратите внимание, что происходит: при удваивании размера задачи необходимое на выполнение время увеличивается вчетверо. Это намного хуже, чем в случае с линейным временем, когда удваивание размера задачи увеличивало время ее выполнения всего в два раза.

Вас не должно удивлять, что алгоритм, совершающий  $2n^2$  шагов,  $3n^2$  шагов и т. д. также классифицируется как алгоритм с квадратичным временем. Оценка «О-большое» скрывает множители перед  $n^2$  так же, как скрывает множители перед  $n$  в алгоритмах с линейным временем.

А что, если у нас будет алгоритм, выполняющийся за  $2n^2 + 6n$  шагов? Он тоже будет считаться квадратичным. Здесь мы берем квадратичное время выполнения  $2n^2$  и прибавляем к нему линейное время  $6n$ . В результате по-прежнему получается алгоритм с квадратичным временем, так как учетверение в квадратичной части быстро превосходит удвоение в линейной.



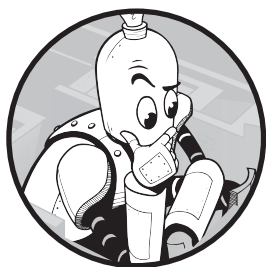
### **«О-большое» в книге**

Про оценку «О-большое» можно сказать еще очень многое. Она является формальной математической основой, используемой в computer science для строго научного анализа времени выполнения алгоритмов. Помимо трех представленных классов эффективности, есть и другие, некоторые из них я упоминаю в книге. Если вы заинтересованы в дальнейшем развитии, то в этой области можно еще многому учиться, но для наших целей достаточно и того, что я вам представил.

«О-большое», как правило, появляется в книге по мере надобности. Мы можем изначально рассматривать некое решение задачи только для того, чтобы в итоге получить ошибку «Time-Limit Exceeded». В таких случаях нам нужно понять, что пошло не так. И первым шагом в этом анализе будет оценка роста времени выполнения как функции от размера задачи. Анализ «О-большого» не только подтверждает, что медленный алгоритм выполняется медленно, но зачастую также раскрывает в нем конкретные узкие места. Далее эту информацию можно использовать для выработки более эффективного решения.

# Б

## Потому что не могу удержаться



В это приложение я включил дополнительные материалы, связанные с некоторыми из рассматриваемых в книге задач. Можно считать это приложение необязательным: оно не относится к ключевому материалу, необходимому для изучения структур данных и алгоритмов. Однако если у вас есть желание получше разобраться в задачах, то данное приложение пригодится.

### Уникальные снежинки: неявные связанные списки

Зачастую в процессе компиляции мы не знаем, сколько памяти потребуется программе. Если вы когда-нибудь задавались вопросом: «Насколько большим нужно сделать этот массив?» или «Будет ли размер этого массива достаточен?», значит, вы уже знакомы с недостаточной гибкостью массивов в Си. Нам необходимо изначально задать размер массива, хотя мы можем его не знать, пока не начнем заполнять этот массив. Во многих подобных случаях связанные списки уверенно справляются с проблемой. Когда нам требуется дополнительная память для новых данных, мы просто вызываем в среде выполнения `malloc`, добавляя в связный список узел.

В первой задаче главы 1 «Уникальные снежинки» мы использовали связный список для соединения снежинок, находящихся в одной корзине. Для каждой считываемой снежинки мы задействовали `malloc`, чтобы выделить память именно под нее одну. Если считывалось 5000 снежинок, то производилось 5000 вызовов `malloc`. Время, необходимое для этих вызовов, может суммироваться.

Подождите-ка! Я только что сказал, что связанные списки пригождаются, когда нам неизвестно, сколько памяти потребуется. Но в «Уникальных снежинках» это было известно! Не в точности, конечно, но по крайней мере мы знали *максимум*, который потребуется: он определялся максимально возможным количеством снежинок, которое было ограничено 100 000.

Здесь возникают вопросы. Почему мы все равно используем `malloc`? Есть ли способ избежать этого? По правде говоря, у меня есть решение для «Уникальных снежинок», в котором не используется `malloc`, и при этом оно вдвое быстрее.

Ключевая идея состоит в предварительном выделении массива с максимальным числом узлов (100 000). Назовем этот массив `nodes`. Он хранит узлы всех (теперь неявных) связанных списков. Каждый элемент `nodes` — это целое число, представляющее индекс следующего узла в его списке узлов. Давайте наглядно разберем это на примере массива `nodes`:

```
[-1, 0, -1, 1, 2, 4, 5]
```

Предположим, мы знаем, что один из списков начинается в индексе 6. В этом индексе хранится значение 5, которое сообщает, что индекс 5 представляет следующий узел в списке. Аналогичным образом индекс 5 сообщает, что следующим узлом списка является индекс 4. Индекс 4 указывает, что очередным узлом будет индекс 2. А что насчет индекса 2, где хранится значение -1? Значение -1 будет использоваться в качестве NULL: оно указывает, что следующего элемента нет. Таким образом, мы раскрыли список из элементов с индексами 6, 5, 4 и 2.

В этом массиве есть еще один непустой список. Предположим, нам известно, что этот список начинается с индекса 3. Индекс 3 сообщает, что следующий за ним узел находится в индексе 1. Индекс 1, в свою очередь, показывает, что очередным узлом списка является индекс 0. На этом все — в индексе 0 хранится -1, значит, список здесь заканчивается. В этом случае мы раскрыли список из элементов с индексами 3, 1 и 0.

Если в каком-то индексе находится значение -1, значит, на нем список заканчивается. В противном случае он указывает на индекс следующего элемента этого списка.

Обратите внимание, что `nodes` ничего не говорит о том, где списки начинаются. Нам пришлось предположить, что начала списков располагаются в индексах 6 и 3. Но как мы могли это узнать? С помощью другого массива — `heads`, который сообщает индекс первого узла (вершины) списка. Другие элементы в `heads` имеют значение -1.

В решении без `malloc` используются всего три массива: `nodes`, `heads` и `snowflakes`. Массив `snowflakes` хранит снежинки, позволяя нам находить каждую из них по индексам в `nodes` и `heads`. Вот эти три массива:

```
static int snowflakes[SIZE][6];
static int heads[SIZE];
static int nodes[SIZE];
```

Чтобы перейти от связанных списков к используемым здесь неявным, потребуется изменить только две функции из четырех: `identify_identical` и `main`. Эти изменения коснутся синтаксиса, но не сути: `identify_identical` по-прежнему будет выполнять попарные сравнения всех снежинок в списке, а `main` — все так же считывать снежинки и создавать эти списки.

Новый вариант `identify_identical` приведен в листинге Б.1 — сравните его с тем, что был в листинге 1.12.

**Листинг Б.1. Определение идентичных снежинок в неявных связанных списках**

```
void identify_identical(int snowflakes[][6], int heads[],
                       int nodes[]) {
    int i, node1, node2;
    for (i = 0; i < SIZE; i++) {
        node1 = heads[i];
        while (node1 != -1) {
            node2 = nodes[node1]; ❶
            while (node2 != -1) {
                if (are_identical(snowflakes[node1], snowflakes[node2])) {
                    printf("Twin snowflakes found.\n");
                    return;
                }
                node2 = nodes[node2]; ❷
            }
            node1 = nodes[node1]; ❸
        }
    }
    printf("No two snowflakes are alike.\n");
}
```

Внутри цикла `for` в качестве первого узла текущего списка устанавливается `node1`. Если этот список пуст, то внешний цикл `while` для этого узла выполняться не будет. Если же он не пуст, то при помощи массива `nodes` узел `node2` устанавливается как идущий за `node1` ❶. Вместо кода связанного списка вроде `node2 = node2->next` мы снова находим следующий узел с помощью массива `nodes` ❷ ❸.

Новая функция `main` приведена в листинге Б.2.

**Листинг Б.2. Функция `main` для неявных связанных списков**

```
int main(void) {
    static int snowflakes[SIZE][6];
    static int heads[SIZE];
    static int nodes[SIZE];
    int n;
```

```
int i, j, snowflake_code;
for (i = 0; i < SIZE; i++) {
    heads[i] = -1;
    nodes[i] = -1;
}
scanf("%d", &n);
for (i = 0; i < n; i++) {
    for (j = 0; j < 6; j++)
        scanf("%d", &snowflakes[i][j]);
    snowflake_code = code(snowflakes[i]);
    nodes[i] = heads[snowflake_code]; ❶
    heads[snowflake_code] = i; ❷
}
identify_identical(snowflakes, heads, nodes);
return 0;
}
```

Предположим, мы только что считали снежинку и сохранили ее в строке `i` массива `snowflakes`. Нам нужно, чтобы эта снежинка стала вершиной соответствующего списка. Для этого мы сохраняем старую вершину в индексе `i` массива `nodes` ❶, а затем устанавливаем в качестве новой снежинку `i` ❷.

Сравните это решение с решением из главы 1 на основе связного списка. Какое бы вы предпочли? Сложнее ли вам было понять решение без `malloc`? Отправьте оба варианта на проверку. Как думаете, повышение скорости стоило дополнительной работы?

## Стрость к бургерам: реконструкция решения

В главе 3 мы решали три задачи — «Стрость к бургерам», «Экономные покупатели» и «Хоккейное соперничество», — в которых требовалось минимизировать или максимизировать значения ответов. В «Страсти к бургерам» мы давали ответы в форме `2 2`, это означало, что Гомер съедает два бургера и потом две минуты пьет пиво. В «Экономных покупателях» мы минимизировали сумму денег на покупку яблок. Там наш ответ выглядел как `Buy 3 for $3.00`. В «Хоккейном соперничестве» мы максимизировали количество голов в принципиальных матчах и в ответе выводили их число, например `20`.

Однако обратите внимание, что во всех случаях мы выводим лишь *значение* оптимального решения, не приводя само это решение. Мы не указываем, какие бургеры нужно есть, как покупать яблоки или какие из игр принципиальные.

Подавляющее большинство задач оптимизации в спортивном программировании требуют вывести только значение ответа, на чем и был сделан фокус в главе 3. Хотя мы вполне можем использовать мемоизацию и динамическое программирование для вывода оптимального решения.

Давайте посмотрим, как это делается на примере «Страсти к бургерам», и возьмем для этого следующий тестовый пример:

```
4 9 15
```

Для него мы выведем не только результат оптимального решения, но и само это решение:

```
2 2
Eat a 4-minute burger
Eat a 9-minute burger
```

В первой строке выведен ответ. Остальные две объясняют оптимальное решение, подтверждая, что ответ 2 2 действительно достигим.

Подобный вывод оптимального решения называется *реконструкцией* или *воссозданием* решения. Оба этих термина предполагают, что у нас уже есть все необходимые элементы, которые нужно совместить для пояснения оптимального решения. Все верно: нужные нам данные хранятся в массивах `memo` и `dp`. В данном случае мы используем `dp`. При желании точно таким же образом можно использовать массив `memo`.

Напишем такую функцию:

```
void reconstruct(int m, int n, int dp[], int minutes)
```

Напомню, что в задаче даются  $m$ -минутные и  $n$ -минутные бургеры. Параметры  $m$  и  $n$  отражают именно эти величины и задаются в исходных данных. Массив `dp` создается алгоритмом динамического программирования (листинг 3.8). Наконец, параметр `minutes` отражает время, затраченное на поедание бургеров. Бургеры, которые нужно съесть в оптимальном решении, будут выводиться по одному на строку.

Какой последний бургер должен съесть Гомер в оптимальном решении? Если бы мы решали задачу с нуля, то ответ был бы неизвестен. Пришлось бы смотреть, что происходит при выборе в качестве последнего сначала  $m$ -минутного, а затем  $n$ -минутного бургера. Действительно, так мы и делали при решении этой задачи в главе 3. Но сейчас-то в нашем распоряжении есть массив `dp`. Он и сообщит, какой из двух вариантов лучше.

Вот основная идея: рассмотреть `dp[minutes - m]` и `dp[minutes - n]`. Нам доступны оба этих значения, потому что массив `dp` уже построен. Исходя из того, какое значение окажется больше, мы решим, какой бургер должен быть последним. То есть если больше окажется `dp[minutes - m]`, то последним будет бургер  $m$ . Если же больше окажется `dp[minutes - n]`, значит, последним должен быть бургер  $n$  (если `dp[minutes - m]` и `dp[minutes - n]` равны, тогда можно выбрать в качестве последнего бургера любой из двух).

Это рассуждение аналогично тому, что используется в листинге 3.8 для построения массива `dp`. Там мы выбирали максимум из `first` и `second`. Здесь же мы разбираем, какой из этих выборов сделал алгоритм динамического программирования.

После определения последнего бургера мы отнимаем время, затраченное на его съедение, и повторяем процесс до тех пор, пока не достигнем нуля минут, когда реконструкция завершится. В листинге Б.3 приведена соответствующая функция.

**Листинг Б.3. Реконструкция решения**

```
void reconstruct(int m, int n, int dp[], int minutes) {
    int first, second;
    while (minutes > 0) {
        first = -1;
        second = -1;
        if (minutes >= m)
            first = dp[minutes - m];
        if (minutes >= n)
            second = dp[minutes - n];
        if (first >= second) {
            printf("Eat a %d-minute burger\n", m);
            minutes = minutes - m;
        } else {
            printf("Eat a %d-minute burger\n", n);
            minutes = minutes - n;
        }
    }
}
```

Эта функция должна дважды вызываться в двух местах листинга 3.8, по разу после каждого вызова `printf`. Первый вызов:

```
reconstruct(m, n, dp, t);
```

Второй вызов:

```
reconstruct(m, n, dp, i);
```

Я рекомендую вам реконструировать, следуя тому же принципу, оптимальные решения задач «Экономные покупатели» и «Хоккейное соперничество».

## Погоня за пешкой: кодирование ходов

В задаче «Погоня за пешкой» главы 4 мы разработали алгоритм BFS для поиска количества ходов, необходимых коню, чтобы попасть в каждую клетку из его стартовой позиции. Коню доступно восемь возможных ходов, каждый из которых мы

прописали в коде (см. листинг 4.1). К примеру, чтобы конь исследовал переход на одну клетку вверх и на две вправо, мы прописали следующее:

```
add_position(from_row, from_col, from_row + 1, from_col + 2,
            num_rows, num_cols, new_positions,
            &num_new_positions, min_moves);
```

А вот код для перехода на одну клетку вверх и на две влево:

```
add_position(from_row, from_col, from_row + 1, from_col - 2,
            num_rows, num_cols, new_positions,
            &num_new_positions, min_moves);
```

У нас получилось повторение кода: единственное, что поменялось, — это знак «плюс» на знак «минус». Действительно, все восемь ходов кодируются очень похожим образом путем манипуляций с плюсами, минусами, а также единицами и двойками. Такой подход весьма уязвим для ошибок.

К счастью, есть хорошая техника, чтобы избежать подобного повторения. Она применима ко многим задачам, в которых требуется исследовать неявный граф, имеющий несколько измерений (например, рядов и столбцов).

Вот восемь возможных ходов коня в том виде, в котором я представил их в главе 4:

- вверх на 1, вправо на 2;
- вверх на 1, влево на 2;
- вниз на 1, вправо на 2;
- вниз на 1, влево на 2;
- вверх на 2, вправо на 1;
- вверх на 2, влево на 1;
- вниз на 2, вправо на 1;
- вниз на 2, влево на 1.

Начнем с рядов и запишем, как каждый ход изменяет номер ряда. Первый ход увеличивает его номер на один, второй — тоже. Третий и четвертый ходы уменьшают номер ряда на один. Пятый и шестой увеличивают его номер на два, а седьмой и восьмой уменьшают также на два. Вот массив этих чисел:

```
int row_dif[8] = {1, 1, -1, -1, 2, 2, -2, -2};
```

Он назван `row_dif`, потому что указывает разницу между текущим рядом и рядом после совершения хода.



Теперь то же самое сделаем для столбцов. Первый ход увеличивает номер столбца на два, второй ход уменьшает его на два, и т. д. В виде массива эти разницы в столбцах будут выглядеть так:

```
int col_dif[8] = {2, -2, 2, -2, 1, -1, 1, -1};
```

Эти два массива описывают влияние, оказываемое каждым ходом на текущие номера рядов и столбцов. Например, элементы `row_dif[0]` и `col_dif[0]` сообщают, что первый ход увеличивает номер ряда на один и столбца на два, значения в `row_dif[1]` и `col_dif[1]` указывают, что второй ход увеличивает номер ряда на один и уменьшает номер столбца на два, и т. д.

Теперь, вместо того чтобы вводить практически идентичные вызовы `add_position`, можно использовать цикл из восьми итераций, прописав в нем всего один вызов `add_position`. Вот как это делается при помощи новой целочисленной переменной `m`, используемой для перебора ходов:

```
for (m = 0; m < 8; m++)
    add_position(from_row, from_col,
                from_row + row_dif[m], from_col + col_dif[m],
                num_rows, num_cols, new_positions,
                &num_new_positions, min_moves);
```

Уже лучше! Обновите код для задачи «Погоня за пешкой» из главы 4 и отправьте его на проверку. Он также пройдет все тесты, но заметного изменения скорости ждать не стоит. В данном случае достижение заключается в том, что вы избавились от большого количества повторяющегося кода.

Здесь у нас было всего восемь ходов, поэтому мне удалось обойтись в главе 4 без этого приема. Однако если бы в задаче было намного больше возможных ходов, то постоянное повторение вызова `add_position` точно бы не дало допустимого решения. Представленный же здесь вариант прекрасно масштабируется.

## Алгоритм Дейкстры и использование куч

В главе 5 мы изучили алгоритм Дейкстры для нахождения кратчайшего пути во взвешенных графах. Время выполнения алгоритма Дейкстры составляло  $O(n^2)$ , где  $n$  — количество вершин графа. Алгоритм Дейкстры затрачивает на поиск минимумов много времени, поскольку в каждой итерации ему нужно найти вершину, расстояние до которой минимально среди всех еще необработанных вершин.

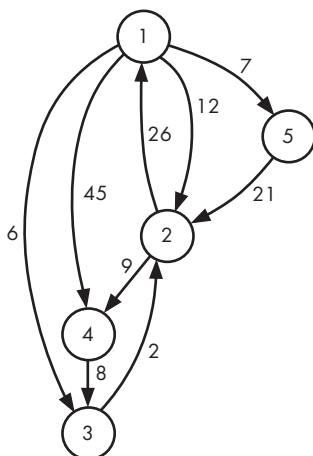
Затем в главе 7 мы познакомились с `max`- и `min`-кучами. `Max`-куча в нахождении кратчайшего пути не поможет, а вот `min`-куча вполне подойдет, так как ее задача

состоит в быстром поиске минимума. Получается, что этот вид кучи можно задействовать для ускорения алгоритма Дейкстры. Этот союз был создан самими небесами компьютерной науки.

Min-куча будет содержать все найденные вершины, которые еще не были обработаны. Она также может содержать некоторые найденные вершины, которые *были* обработаны. Это нормально: как и в решении задачи «Акция в супермаркете» (глава 7, «Решение 2. Кучи»), мы просто проигнорируем все обработанные вершины, попавшие в min-кучу.

### Мышиный лабиринт: отслеживание с помощью куч

Давайте доработаем наше решение задачи «Мышиный лабиринт» (глава 5, задача 1), задействовав в нем min-кучу. Вот граф, который мы тогда использовали (см. рис. 5.1):



В разделе «Кратчайшие пути во взвешенных графах» главы 5 я прослеживал работу алгоритма Дейкстры, начиная с вершины 1. На этот раз мы сделаем то же самое, но уже для min-кучи. Каждый элемент кучи будет состоять из вершины и времени, необходимого для достижения этой вершины. Мы увидим, что один элемент может встречаться неоднократно. Однако, так как это min-куча, мы сможем обработать каждую вершину, используя только ее минимальное время.

В каждом последующем «снимке» min-кучи я выстроил ряды в том же порядке, в каком они будут храниться в массиве кучи.

Вначале куча будет содержать только вершину 1 со временем 0. Для других вершин нет информации о времени. Следовательно, мы получим следующее.

Min-куча	
вершина	время
1	0

Состояние		
вершина	обработана	min_time
1	false	0
2	false	
3	false	
4	false	
5	false	

Извлечение из min-кучи дает нам ее единственный элемент, вершину 1. Затем вершину 1 мы используем для обновления кратчайших путей до вершин 2, 3, 4 и 5, помещая эти вершины в min-кучу. Теперь получается такое состояние:

Min-куча	
вершина	время
3	6
2	12
5	7
4	45

Состояние		
вершина	обработана	min_time
1	true	0
2	false	12
3	false	6
4	false	45
5	false	7

Следующей из min-кучи извлекается вершина 3, которая сообщает кратчайший путь до вершины 2. Тогда мы снова добавляем вершину 2 в кучу, но уже с более коротким путем. Вот что получилось:

Min-куча	
вершина	время
5	7
2	8
4	45
2	12

Состояние		
вершина	обработана	min_time
1	true	0
2	false	8
3	true	6
4	false	45
5	false	7

Следующая вершина — 5. Она не дает новых кратчайших путей, значит, в кучу добавлять нечего.

Min-куча	
вершина	время
2	8
2	12
4	45

Состояние		
вершина	обработана	min_time
1	true	0
2	false	8
3	true	6
4	false	45
5	true	7

Следующей в минимальной куче идет вершина 2 — та, в которой указано время 8, не 12. Она ведет к обновлению кратчайшего пути до вершины 4, а значит, и к очередному вхождению этой вершины в min-кучу. Вот результат:

Min-куча	
вершина	время
2	12
4	45
4	17

Состояние		
вершина	обработана	min_time
1	true	0
2	true	8
3	true	6
4	false	17
5	true	7

Следующей из кучи извлекается вершина 2. Опять! Вершина 2 уже обрабатывалась, значит, мы просто извлекаем ее из кучи, ничего не предпринимая. Вот что остается:

Min-куча	
вершина	время
4	17
4	45

Состояние		
вершина	обработана	min_time
1	true	0
2	true	8
3	true	6
4	false	17
5	true	7

Две версии вершины 4 будут извлекаться из min-кучи по очереди. Первая вершина 4 не даст обновлений кратчайших путей, так как все остальные вершины обработаны, но установит саму вершину 4 как обработанную. Следовательно, второй ее вариант уже будет пропущен.

В большинстве учебников при реализации алгоритма Дейкстры на основе кучи предполагается, что есть возможность сократить путь до вершины в куче. Таким образом, вершину в куче можно будет обновить, и отпадет необходимость в нескольких ее вхождениях. Однако кучи, которые мы разработали в главе 7, не поддерживают подобную операцию «сокращения». Но можете быть уверены, что выполняемые нами добавления дополнительных вершин вместо обновления предполагают точно такой же худший случай времени выполнения. А какой он, кстати?

Пусть  $n$  — количество вершин в графе, а  $m$  — количество ребер. Мы обрабатываем каждое ребро  $u \rightarrow v$  максимум один раз — при извлечении  $u$  из кучи. Каждое ребро может вести не более чем к одному добавлению в кучу, значит, добавляется не более  $m$  элементов. Тогда размер максимально возможной кучи равен  $m$ . Извлечь можно только то, что было добавлено, значит операций извлечения будет не более  $m$ . Таким образом, всего получится  $2m$  операций над кучей, каждая из которых занимает не более  $\log m$  времени. Следовательно, здесь у нас алгоритм  $O(m \log m)$ .

Сравните это с реализацией  $O(n^2)$  из главы 5. Очевидно, что вариант на основе кучи выигрывает, если число ребер мало относительно  $n^2$ . К примеру, если всего есть  $n$  ребер, то реализация на основе кучи будет иметь быстродействие  $O(n \log n)$  и существенно опередит по времени выполнения реализацию  $O(n^2)$  из главы 5. Если же количество ребер будет большим, то преимущество новой реализации снизится. Например, если ребер будет  $n^2$ , тогда решение на основе кучи будет иметь быстродействие  $O(n^2 \log n)$ , что сопоставимо с  $O(n^2)$ , но немного медленнее. Если наперед не известно, будет в графе много или мало ребер, то безопаснее всего использовать кучу: единственной возможной проблемой станет фактор  $\log n$  в графах с большим количеством ребер, но это небольшая цена за существенный прирост производительности в графах с меньшим числом ребер.

## Мышиный лабиринт: реализация с кучами

Для элементов кучи мы используем следующую структуру:

```
typedef struct heap_element {
    int cell;
    int time;
} heap_element;
```

Я не стану повторять здесь код добавления min-кучи (листинг 7.5) и код извлечения (листинг 7.6). Единственное изменение состоит в сравнении `time` вместо `cost`. Оставляю эту доработку вам.

Функция `main` — та же, что была в главе 5, листинг 5.1. Все, что нужно сделать, — это заменить `find_time` (листинг 5.2), чтобы использовать min-кучу вместо линейных поисков. Соответствующий код дан в листинге Б.4.

**Листинг Б.4.** Нахождение кратчайшего пути к выходу с помощью алгоритма Дейкстры и куч

```
int find_time(edge *adj_list[], int num_cells,
              int from_cell, int exit_cell) {
    static int done[MAX_CELLS + 1];
    static int min_times[MAX_CELLS + 1];
    static heap_element min_heap[MAX_CELLS * MAX_CELLS + 1]; ❶
    int i;
    int min_time, min_time_index, old_time;
    edge *e;
    int num_min_heap = 0;
    for (i = 1; i <= num_cells; i++) {
        done[i] = 0;
        min_times[i] = -1;
    }
    min_times[from_cell] = 0;
    min_heap_insert(min_heap, &num_min_heap, from_cell, 0);

    while (num_min_heap > 0) { ❷
        min_time_index = min_heap_extract(min_heap, &num_min_heap).cell;
        if (done[min_time_index])
            continue; ❸
        min_time = min_times[min_time_index];
        done[min_time_index] = 1;

        e = adj_list[min_time_index];
        while (e) { ❹
            old_time = min_times[e->to_cell];
            if (old_time == -1 || old_time > min_time + e->length) {
                min_times[e->to_cell] = min_time + e->length;
                min_heap_insert(min_heap, &num_min_heap, ❺
                               e->to_cell, min_time + e->length);
            }
            e = e->next;
        }
    }
    return min_times[exit_cell];
}
```

Каждая ячейка может привести к добавлению в min-кучу не более `MAX-CELLS` элементов. Тогда, чтобы обезопасить себя от переполнения кучи, мы выделим

пространство для `MAX-CELLS*MAX_CELL`s элементов плюс один, поскольку индексация здесь начинается с 1, а не с 0 ❶.

Основной цикл `while` продолжается до тех пор, пока в `min`-куче остаются вершины ❷. Если извлекаемая вершина оказывается обработана, то в этой итерации ничего не происходит ❸. В противном случае мы обрабатываем исходящие ребра как обычно ❹, добавляя вершины в `min`-кучу при нахождении более коротких путей ❺.

## Сжатие сжатия пути

В разделе «Оптимизация 2. Сжатие пути» главы 8 вы познакомились со сжатием пути, оптимизацией в структуре непересекающихся множеств на основе дерева. Ее код для задачи «Социальная сеть» приведен в листинге 8.8. Но в том виде — с двумя циклами `while` — на практике вы этот код вряд ли встретите.

Обычно я стараюсь приводить прозрачный по смыслу код и надеюсь, что в этой книге иных примеров вам не встречалось. Однако здесь сделаю исключение, потому что вы можете столкнуться с особо краткой однострочной реализацией сжатия пути. Представлена она в листинге Б.5.

### Листинг Б.5. Сжатие пути на практике

```
int find(int p, int parent[]) {  
    return p == parent[p] ? p : (parent[p] = find(parent[p], parent));  
}
```

Я изменил `person` на `p`, чтобы вместить код на одну строку (раз читаемость и без того утрачена, то почему бы нет?).

Здесь производится несколько действий с использованием тернарного оператора `? :` (если), результата присваивания оператора `=` и даже рекурсии. Все это мы разделим на три шага.

## Шаг 1. Больше никаких тернарных «если»

Оператор `? :` — это форма инструкции `if-else`, которая возвращает значение. Программисты используют его, когда хотят сэкономить место и втиснуть всю инструкцию `if-else` в одну строку.

Вот краткий пример:

```
return x >= 10 ? "big" : "small";
```

Если `x` больше или равно 10, то возвращается `big`; в противном случае возвращается `small`.

Оператор `?` : называется *тернарным* (ternary — триада), потому что требует трех операндов: первый — это логическое выражение, чью верность мы проверяем; второй — результат верности логического выражения; третий, соответственно, результат его ложности.

Давайте перепишем листинг Б.5 с использованием стандартной инструкции `if-else` вместо тернарного `if`:

```
int find(int p, int parent[]) {
    if (p == parent[p])
        return p;
    else
        return parent[p] = find(parent[p], parent);
}
```

Уже лучше. Теперь стало очевидным, что код имеет два пути: один, если `p` уже является корнем, и второй, если `p` корнем не является.

## Шаг 2. Более понятный оператор присваивания

Что, по-вашему, делает этот фрагмент кода?

```
int x;
printf("%d\n", x = 5);
```

Он выводит 5! Вы знаете, что `x = 5` присваивает переменной `x` значение 5, но это также и выражение, выводящее значение 5. Все верно: `=` присваивает значение, но также и возвращает значение, хранящееся в этой переменной. Именно поэтому мы можем присвоить нескольким переменным одно значение:

```
a = b = c = 5;
```

В коде сжатия пути у нас на одной строке используются инструкции возвращения и присваивания. Эта строка и присваивает значение `parent[p]`, и возвращает это значение. Разделим два этих действия:

```
int find(int p, int parent[]) {
    int community;
    if (p == parent[p])
        return p;
    else {
        community = find(parent[p], parent);
        parent[p] = community;
        return community;
    }
}
```



Мы явно находим представителя для  $p$ , присваиваем ему  $\text{parent}[p]$ , а затем возвращаем этого представителя.

### **Шаг 3. Понятная рекурсия**

Теперь выделим для рекурсии отдельную строку:

```
community = find(parent[p], parent);
```

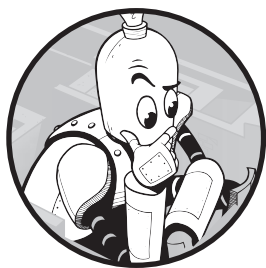
Функция `find` выполняет сжатие пути от своего аргумента к корню и возвращает корень дерева. Следовательно, этот рекурсивный вызов выполняет сжатие пути от родителя  $p$  до корня и возвращает корень дерева. Таким образом, обрабатывается все сжатие пути, кроме самого  $p$ . Нам нужно задать родителя  $p$  как корень дерева, что мы делаем следующим образом:

```
parent[p] = community;
```

Вот мы и доказали, что однострочный код сжатия пути тоже работает.

# В

## Сводка по задачам



Я признателен всем, кто, не жалея своего времени, делится опытом и помогает людям обучаться, занимаясь спортивным программированием. Я старался установить автора каждой использованной в книге задачи, а также названия соревнований, в программу которых входили эти задачи. Если у вас есть дополнительная информация об этих задачах, пожалуйста, сообщите мне. Обновления я размещу на сайте книги.

В приведенной ниже таблице используются следующие сокращения:

- CCC — Canadian Computing Competition (Канадское соревнование по программированию);
- CCO — Canadian Computing Olympiad (Канадская олимпиада по программированию);
- COCI — Croatian Open Competition in Informatics (Хорватское открытое соревнование по информатике);
- ECNA — ACM East Central North America Regional Programming Contest (Региональное соревнование по программированию восточной части центрального региона Северной Америки, организованное ассоциацией ACM);
- IOI — International Olympiad in Informatics (Международная олимпиада по информатике);
- POI — Polish Olympiad in Informatics (Польская олимпиада по информатике);

- SAPO — South African Programming Olympiad (Южно-Африканская олимпиада по программированию);
- SWERC — ACM Southwestern Europe Regional Contest (Соревнование юго-западного региона Европы, организованное ассоциацией ACM);
- USACO — USA Computing Olympiad (Олимпиада по программированию США).

Глава	Задача	Оригинальное название	Соревнование / автор задачи
Введение	Очереди за продуктами	Food Lines	Кевин Ван (Kewin Wan)
1	Уникальные снежинки	Snowflakes	2007 CCO / Ондрей Лхотак (Ondrej Lhoták)
1	Сложносоставные слова	Compound Words	Гордон В. Кормак (Gordon V. Cormack)
1	Проверка правописания	Spelling Check	Михаил Мирзаянов (Mikhail Mirzayanov), Наталья Бондаренко (Natalia Bondarenko)
2	Трофеи Хэллоуина	Trick or Tree'ing	2012 DWITE / Амlesh Джаякумар (Amlesh Jayakumar)
2	Расстояние до потомка	Countdown	2005 ECNA / Джон Бономо (John Bonomo), Тодд Фейл (Todd Feil), Шон Маккаллох (Sean McCulloch), Роберт Роос (Robert Roos)
3	Страсть к бургерам	Homer Simpson	Садрул Хабиб Чоудхури (Sadrul Habib Chowdhury)
3	Экономные покупатели	Lowest Price in Town	МАК Ян Кей (Yan Kei), Сабур Захид (Sabur Zaheed)
3	Хоккейное соперничество	Geese vs. Hawks	2018 CCO / Трой Васига (Troy Vasiga), Энди Хуанг (Andy Huang)
3	Учебный план	Marks Distribution	Бахлул Хайдер (Bahlul Haider), Танвир Асан (Tanveer Ahsan)
4	Погоня за пешкой	A Knightly Pursuit	1999 CCC
4	Лазание по канату	Reach for the Top	2018 Woburn Challenge / Джейкоб Плахта (Jacob Plachta)

Глава	Задача	Оригинальное название	Соревнование / автор задачи
4	Перевод книги	Lost in Translation	2016 ECNA / Джон Бономо (John Bonomo), Том Векслер (Tom Wexler), Шон Маккаллох (Sean McCulloch), Дэвид Поэшль (David Poeschl)
5	Мышиный лабиринт	Mice and Maze	2001 SWERC
5	Дорога к бабушке	Visiting Grandma	2008 SAPO / Гарри Виггинс (Harry Wiggins), Киган Каррутерс-Смит (Keegan Carruthers-Smith)
6	Кормление муравьев	Mravi	2014 COCI / Антонио Юрич (Antonio Juric)
6	Прыжки вдоль реки	River Hopscotch	2006 USACO / Ричард Хо (Richard Ho)
6	Качество жизни	Quality of Living	2010 IOI / Крис Чен (Chris Chen)
6	Двери пещеры	Cave	2013 IOI / Амори Пули (Amaury Pouly) и Артур Шаргеро (Arthur Charguéraud) по замыслу Курта Мальхорна (Kurt Mehlhorn)
7	Акция в супер-маркете	Promotion	2000 POI / Томаш Вален (Tomasz Walen)
7	Построение декартовых деревьев	Binary Search. Heap Construction	2004 Ulm / Уолтер Гуттман (Walter Guttman)
7	Сумма двух элементов	Maximum Sum	2009 Kurukshetra Online Programming Contest / Свранапракаш Удайкаумар (Swarnaprakash Udayakumar)
8	Социальная сеть	Social Network Community	Пратик Агарвал (Prateek Agarwal)
8	Друзья и враги	War	Петко Минков (Petko Minkov)
8	Уборка в комнате	Ladice	2013 COCI / Лука Калинович (Luka Kalinovic), Густав Матула (Gustav Matula)

Права на задачи с соревнований CCC и CCO принадлежат Центру образования по математике и вычислительной технике (СЕМС) при Университете Ватерлоо.

*Даниэль Зингаро*  
**Алгоритмы на практике**

*Перевел с английского Д. Брайт*

Руководитель дивизиона	<i>Ю. Сергиенко</i>
Ведущий редактор	<i>Е. Строганова</i>
Литературный редактор	<i>Д. Гудилин</i>
Художественный редактор	<i>В. Мостипан</i>
Корректоры	<i>С. Беляева, М. Лауконен</i>
Верстка	<i>Л. Егорова</i>

Изготовлено в России. Изготовитель: ООО «Прогресс книга».  
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,  
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 01.2023. Наименование: книжная продукция. Срок годности: не ограничен.

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

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 28.11.22. Формат 70×100/16. Бумага офсетная. Усл. п. л. 34,830. Тираж 1000. Заказ 0000.

*Брэдфорд Такфилд*

## **АЛГОРИТМЫ НЕФОРМАЛЬНО. ИНСТРУКЦИЯ ДЛЯ НАЧИНАЮЩИХ ПИТОНИСТОВ**



Алгоритмы — это не только задачи поиска, сортировки или оптимизации, они помогут вам поймать бейсбольный мяч, проникнуть в «механику» машинного обучения и искусственного интеллекта и выйти за границы возможного.

Вы узнаете нюансы реализации многих самых популярных алгоритмов современности, познакомитесь с их реализацией на Python 3, а также научитесь измерять и оптимизировать их производительность.

**КУПИТЬ**

*Ришал Харбанс*

## **ГРОКАЕМ АЛГОРИТМЫ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА**



Искусственный интеллект — часть нашей повседневной жизни. Мы встречаемся с его проявлениями, когда занимаемся шопингом в интернет-магазинах, получаем рекомендации «вам может понравиться этот фильм», узнаем медицинские диагнозы...

Чтобы уверенно ориентироваться в новом мире, необходимо понимать алгоритмы, лежащие в основе ИИ.

«Грокаем алгоритмы искусственного интеллекта» объясняет фундаментальные концепции ИИ с помощью иллюстраций и примеров из жизни. Все, что вам понадобится, — это знание алгебры на уровне старших классов школы, и вы с легкостью будете решать задачи, позволяющие обнаружить банковских мошенников, создавать шедевры живописи и управлять движением беспилотных автомобилей.

**КУПИТЬ**

*Кори Альтхофф*

## **COMPUTER SCIENCE ДЛЯ ПРОГРАММИСТА-САМОУЧКИ. ВСЕ ЧТО НУЖНО ЗНАТЬ О СТРУКТУРАХ ДАННЫХ И АЛГОРИТМАХ**



Книги Кори Альтхоффа вдохновили сотни тысяч людей на самостоятельное изучение программирования.

Чтобы стать профи в программировании, не обязательно иметь диплом в области computer science, и личный опыт Кори подтверждает это: он стал разработчиком ПО в eBay и добился этого самостоятельно.

Познакомьтесь с наиболее важными темами computer science, в которых должен разбираться каждый программист-самоучка, мечтающий о выдающейся карьере, — это структуры данных и алгоритмы. «Computer Science для программиста-самоучки» поможет вам пройти техническое интервью, без которого нельзя получить работу в «айти».

Книга написана для абсолютных новичков, поэтому у вас не должно возникнуть трудностей, даже если ранее вы ничего не слышали о computer science.

**КУПИТЬ**