Содержание

Предисловие ... 10

Глава 1. Предварительные условия ... 13

1.1. Что такое правильность? ... 13
- 1.1.1. Сложность ... 14
- 1.1.2. Деление ... 15
- 1.1.3. Евклид ... 17
- 1.1.4. Палиндромы ... 18
- 1.1.5. Дальнейшие примеры ... 20

1.2. Алгоритмы ранжирования ... 21
- 1.2.1. Алгоритм PageRank ... 22
- 1.2.2. Стабильный брачный союз ... 24
- 1.2.3. Попарные сравнения ... 27

1.3. Ответы к избранным задачам ... 29
1.4. Примечания ... 35

Глава 2. Жадный алгоритм ... 37

2.1. Остовные деревья минимальной стоимости ... 37
2.2. Задания с предельными сроками и прибылями ... 44
2.3. Дальнейшие примеры и задачи ... 47
- 2.3.1. Отсчитывание сдачи ... 47
- 2.3.2. Паросочетание с максимальным весом ... 48
- 2.3.3. Кратчайший путь ... 48
- 2.3.4. Коды Хаффмана ... 51

2.4. Ответы к избранным задачам ... 53
2.5. Примечания ... 59

Глава 3. Разделяй и властвуй ... 61

3.1. Сортировка слиянием ... 61
3.2. Умножение двоичных чисел ... 63
3.3. Алгоритм Савича ... 65
3.4. Дальнейшие примеры и задачи ... 67
- 3.4.1. Расширенный алгоритм Евклида ... 67
- 3.4.2. Быстрая сортировка ... 68
- 3.4.3. Команда git bisect ... 68

3.5. Ответы к избранным задачам ... 69
3.6. Примечания ... 70

Глава 4. Динамическое программирование ... 72

4.1. Задача о наибольшей монотонной подпоследовательности ... 72
4.2. Задача кратчайшего пути для всех пар ... 73

4.2.1. Алгоритм Беллмана–Форда 75
4.3. Простая задача о рюкзаке 75
4.3.1. Задача о рассредоточенном рюкзаке 78
4.3.2. Общая задача о рюкзаке 79
4.4. Задача выбора мероприятий 79
4.5. Задания с указанием предельных сроков, длительностей и прибылей 82
4.6. Дальнейшие примеры и задачи 83
4.6.1. Задача суммирования сплошной подпоследовательности 83
4.6.2. Перетасовка 84
4.7. Ответы к избранным задачам 86
4.8. Примечания 90

Глава 5. Онлайновые алгоритмы 92
5.1. Задача доступа к списку 92
5.2. Замещение страниц 97
5.2.1. Замещение страниц по требованию 98
5.2.2. Первым вошел/первым вышел (FIFO) 100
5.2.3. Наименее недавно использованная страница (LRU) 100
5.2.4. Маркировочные алгоритмы 103
5.2.5. Сброс при заполнении (FWF) 105
5.2.6. Наибольшее расстояние вперед (LFD) 105
5.3. Ответы к избранным задачам 109
5.4. Примечания 112

Глава 6. Рандомизированные алгоритмы 113
6.1. Идеальное паросочетание 114
6.2. Сопоставление с образцом 117
6.3. Проверка простоты 119
6.4. Шифрование с публичным ключом 122
6.4.1. Обмен ключами Диффи–Хеллмана 122
6.4.2. Криптосистема Эль-Гамаля 124
6.4.3. Криптосистема RSA 127
6.5. Дальнейшие задачи 128
6.6. Ответы к избранным задачам 129
6.7. Примечания 134

Глава 7. Алгоритмы в линейной алгебре 138
7.1. Введение 138
7.2. Гауссово исключение 138
7.2.1. Формальные доказательства правильности над $\mathbb{Z}_2$ 141
7.3. Алгоритм Грама-Шмидта 144
7.4. Гауссова редукция решетки 144
7.5. Вычисление характеристического многочлена 145
7.5.1. Алгоритм Чанки 145
7.5.2. Алгоритм Берковица 146
7.5.3. Доказательство свойств характеристического многочлена 147

7.6. Ответы к избранным задачам 152
7.7. Примечания 154

Глава 8. Вычислительные основы 155
8.1. Введение 155
8.2. Алфавиты, строки и язык 155
8.3. Регулярные языки 156
8.3.1. Детерминированный конечный автомат 156
8.3.2. Недетерминированные конечные автоматы 159
8.3.3. Регулярные выражения 162
8.3.4. Алгебраические законы для регулярных выражений 165
8.3.5. Свойства замыкания в регулярных языках 166
8.3.6. Сложность преобразований и принятия решений 167
8.3.7. Эквивалентность и минимизация автоматов 167
8.3.8. Нерегулярные языки 169
8.3.9. Автоматы на членах 171
8.4. Контекстно-свободные языки 172
8.4.1. Контекстно-свободные грамматики 172
8.4.2. Магазинные автоматы 174
8.4.3. Нормальная форма Хомского 176
8.4.4. Алгоритм CYK 178
8.4.5. Лемма о накачке для контекстно-свободных языков 179
8.4.6. Дальнейшие замечания по нормальной форме Хомского 180
8.4.7. Другие грамматики 181
8.5. Машины Тьюринга 181
8.5.1. Недетерминированные машины Тьюринга 182
8.5.2. Варианты кодирования 184
8.5.3. Разрешимость 184
8.5.4. Тезис Черча–Тьюринга 185
8.5.5. Неразрешимость 186
8.5.6. Редукции 188
8.5.7. Теорема Райса 189
8.5.8. Задача соответствий Поста 189
8.5.9. Неразрешимые свойства контекстно-свободных языков 194
8.6. Ответы к избранным задачам 195
8.7. Примечания 205

Глава 9. Математическая основа 208
9.1. Индукция и инвариантность 208
9.1.1. Индукция 208
9.1.2. Инвариантность 211
9.2. Теория чисел 212
9.2.1. Простые числа 213
9.2.2. Модулярная арифметика 213
9.2.3. Теория групп 214
9.2.4. Приложения теории групп к теории чисел 216

9.3. Отношения 217
9.3.1. Замыкание 218
9.3.2. Отношение эквивалентности 220
9.3.3. Частичные порядки 221
9.3.4. Решетки 223
9.3.5. Теория неподвижных точек 224
9.3.6. Рекурсия и неподвижные точки 227
9.4. Логика 229
9.4.1. Пропозициональная логика 230
9.4.2. Первопорядковая логика 235
9.4.3. Арифметика Пеано 239
9.4.4. Формальная верификация 239
9.5. Ответы к избранным задачам 242
9.6. Примечания 261

Библиография 263

Тематический указатель 269

Предисловие

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

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

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

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

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

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

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

---

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

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

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

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

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

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

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

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

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

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

---

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

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

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

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

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

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

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

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

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

---

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

[2] См. http://www.msoltys.com.

Глава 1

Предварительные условия

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

1.1. Что такое правильность?

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

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

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

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

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

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

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

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

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

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

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

1.1.1. Сложность

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

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

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

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

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

Например, $an^2 + bn + c = \Theta(n^2)$, где $a > 0$. Для того чтобы это увидеть, обратите внимание, что $an^2 + bn + c \leqslant (a + |b| + |c|)n^2$ для всех $n \in \mathbb{N}$, и поэтому $an^2 + bn + c = O(n^2)$, где мы взяли абсолютное значение b, c, потому что они могут быть отрицательными. С другой стороны, $an^2 + bn + c = a((n + c_1)^2 - c^2)$, где $c_1 = b/2a$ и $c_2 = (b^2 - 4ac)/4a^2$, благодаря чему мы можем найти c_3 и n_0, вследствие чего для всех $n \geqslant n_0$, $c_3n^2 \leqslant a((n + c_1)^2 - c_2)$, и поэтому $an^2 + bn + c = \Omega(n^2)$.

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

1.1.2. Деление

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

Алгоритм 1.1. Деление

Предусловие: $x \geqslant 0 \wedge y > 0 \wedge x, y \in \mathbb{N}$
1: $q \leftarrow 0$
2: $r \leftarrow x$
3: **while** $y \leqslant r$ **do**
4: $\quad r \leftarrow r - y$
5: $\quad q \leftarrow q + 1$
6: **end while**
7: **return** q, r
Постусловие: $x = (q \cdot y) + r \wedge 0 \leqslant r < y$

В качестве инварианта цикла мы предлагаем следующее логическое утверждение:

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

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

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

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

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

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

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

Таким образом, мы показали полную правильность алгоритма деления.

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

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

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

1.1.3. Евклид

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

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

Алгоритм 1.2. Евклид

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

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

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

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

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

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

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

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

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

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

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

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

1.1.4. Палиндромы

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

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

Алгоритм 1.3. Палиндромы

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

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

```
3:        if (A[i] ≠ A[n − i − 1]) then
4:            return F
5:        end if
6: i ← i + 1
7: end while
8: return T
```
Постусловие: вернуть T тогда и только тогда, когда A является палиндромом

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

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

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

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

```
s = 'palindrome'
```

тогда мы можем получить доступ к разным срезам следующим образом:

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

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

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

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

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

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

1.1.5. Дальнейшие примеры

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

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

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

Алгоритм 1.4. См. задачу 1.14

1: $x \leftarrow m$; $y \leftarrow n$; $z \leftarrow 0$
2: **while** ($x \neq 0$) **do**
3: **if** (rem(x, 2) = 1) **then**
4: $z \leftarrow z + y$
5: **end if**
6: $x \leftarrow$ div(x, 2)
7: $y \leftarrow y \cdot 2$
8: **end while**
9: **return** z

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

Algorithm 1.5. См. задачу 1.15

1: **while** ($a > 0$) **do**
2: **if** ($a < b$) **then**
3: $(a, b) \leftarrow (2a, b - a)$
4: **else**
5: $(a, b) \leftarrow (a - b, 2b)$
6: **end if**
7: **end while**

Рассмотрите приведеный ниже алгоритм 1.6.

Algorithm 1.6. Алгоритм Улама

Pre-condition: $a > 0$
$x \leftarrow a$
while последние три значения x не равны 4, 2, 1 **do**

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

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

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

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

1.2. Алгоритмы ранжирования

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

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

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

1.2.1. Алгоритм PageRank

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

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

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

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

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

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

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

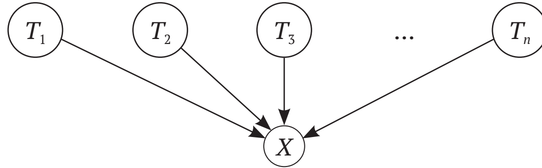


Рис. 1.1 ❖ Вычисление ранга страницы *A*

Тогда ранг страницы X можно вычислить следующим образом:

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

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

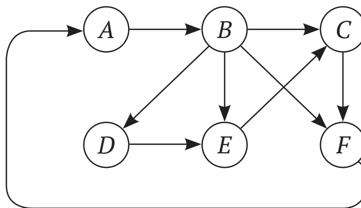
Задача с (1.4) заключается в том, что она выглядит зацикленной. Как изначально вычислить PR(T_i)? Алгоритм работает поэтапно, уточняя ранг каждой страницы на каждом этапе. Первоначально мы используем эгалитарный подход и присваиваем каждой странице ранг $1/N$, где N – это общее число страниц в интернете.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

Разумеется, интернет представляет собой обширную коллекцию разнородных документов, и (1.4) является слишком простой формулой, чтобы выразить ею абсолютно все, – поиск в Google намного сложнее. Например, не все исходящие ссылки обрабатываются одинаково: ссылка более крупным шрифтом или выде-

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

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

Задача 1.17. Рассмотрите следующую небольшую сеть:



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

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

1.2.2. Стабильный брачный союз

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

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

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

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

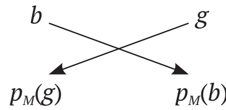


Рис. 1.2 ❖ Блокирующая пара: b и g предпочитают друг друга своим партнерам $p_M(b)$ и $p_M(g)$

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

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

В конце этапа s будем считать, что мы произвели паросочетание

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

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

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

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

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

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

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

Алгоритм 1.7. Гейл-Шепли

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

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

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

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

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

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

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

Задача 1.23. Реализуйте алгоритм 1.7.

1.2.3. Попарные сравнения

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

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

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

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

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

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

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

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

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

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

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

1.3. Ответы к избранным задачам

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

Задача 1.2. Очевидно, что

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Алгоритм 1.8. Расширенный алгоритм Евклида

Предусловие: $m > 0, n > 0$
1: $a \leftarrow 0$; $x \leftarrow 1$; $b \leftarrow 1$; $y \leftarrow 0$; $c \leftarrow m$; $d \leftarrow n$
2: **loop**
3: $q \leftarrow \text{div}(c, d)$
4: $r \leftarrow \text{rem}(c, d)$
5: **if** $r = 0$ **then**
6: stop
7: **end if**
8: $c \leftarrow d$; $d \leftarrow r$; $t \leftarrow x$; $x \leftarrow a$; $a \leftarrow t - qa$; $t \leftarrow y$; $y \leftarrow b$; $b \leftarrow t - qb$
9: **end loop**
Постусловие: $am + bn = d = \gcd(m, n)$

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

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

Базовый случай:

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

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

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

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

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

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

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

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

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

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

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

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

Задача 1.13. Решение дано алгоритмом 1.9.

Алгоритм 1.9. Степени числа 2

Предусловие: $n \geqslant 1$

$x \leftarrow n$
while $(x > 1)$ **do**
  if $(2|x)$ **then**
    $x \leftarrow x/2$
  else
    остановиться и вернуть «нет»
  end if
end while
return «да»

Постусловие: «да» $\Leftrightarrow$ n является степенью числа 2

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

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

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

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

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

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

Результат приведен на рис. 1.3.

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

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

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

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

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

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

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

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

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

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

1.4. Примечания

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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