Посвящается моей семье

Содержание

Предисловие

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

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

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

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

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

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

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

---

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

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

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

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

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

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

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

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

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

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

---

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

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

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

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

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

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

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

Я издаю свой варварский клич на крыше (вершине) мира!

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

---

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

[2] См. http://www.msoltys.com.

Глава 1

Предварительные условия

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

1.1. Что такое правильность?

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

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

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

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

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

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

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

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

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

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

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

1.1.1. Сложность

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

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

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

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

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

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

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

1.1.2. Деление

Что может быть проще целочисленного деления? Даны два целых числа x, y, и мы хотим найти частное и остаток от деления x на y. Например, если $x = 25$ и $y = 3$, то $q = 8$ и $r = 1$. Обратите внимание, что возвращаемые алгоритмом деления q и r обычно обозначаются соответственно как $\mathrm{div}(x, y)$ (*частное*) и $\mathrm{rem}(x, y)$ (*остаток*).

Алгоритм 1.1. Деление

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

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

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

В качестве инварианта цикла мы предлагаем следующее логическое утверждение:

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

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

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

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

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

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

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

Таким образом, мы показали полную правильность алгоритма деления.

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

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

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

1.1.3. Евклид

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

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

Алгоритм 1.2. Евклид

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

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

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

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

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

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

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

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

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

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

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

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

1.1.4. Палиндромы

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

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

Алгоритм 1.3. Палиндромы

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

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

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

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

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

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

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

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

```
s = 'palindrome'
```

тогда мы можем получить доступ к разным срезам следующим образом:

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

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

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

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

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

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

1.1.5. Дальнейшие примеры

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

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

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

Алгоритм 1.4. См. задачу 1.14

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

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

Algorithm 1.5. См. задачу 1.15

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

Рассмотрите приведеный ниже алгоритм 1.6.

Algorithm 1.6. Алгоритм Улама

Pre-condition: $a > 0$

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

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

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

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

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

1.2. Алгоритмы ранжирования

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

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

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

1.2.1. Алгоритм PageRank

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

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

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

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

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

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

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

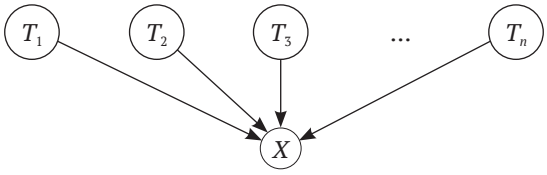


Рис. 1.1 ❖ Вычисление ранга страницы *A*

Тогда ранг страницы X можно вычислить следующим образом:

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

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

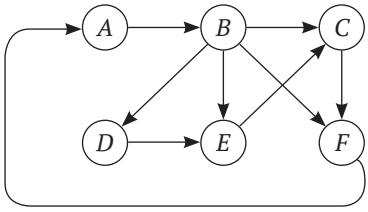
Задача с (1.4) заключается в том, что она выглядит зацикленной. Как изначально вычислить $PR(T_i)$? Алгоритм работает поэтапно, уточняя ранг каждой страницы на каждом этапе. Первоначально мы используем эталитарный подход и присваиваем каждой странице ранг $1/N$, где N – это общее число страниц в интернете.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

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

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

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

Задача 1.17. Рассмотрите следующую небольшую сеть:



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

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

1.2.2. Стабильный брачный союз

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

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

Конец ознакомительного фрагмента.
Приобрести книгу можно
в интернет-магазине
«Электронный универс»
e-Univers.ru