Содержание

Благодарности

Мы благодарны всем, кто способствовал появлению этой книги. Руководство нашей компании A9.com активно поддерживало проект с самого начала. Билл Стейсиор предложил создать курс, легший в основу этой книги, и выбрал тему из предложенных нами вариантов. Брайан Пинкертон не только прослушал весь курс, но и всячески приветствовал идею превратить его в книгу. Мы благодарим также Мэта Маркуса, который вместе с Алексом читал похожий курс в компании Adobe в 2004–2005 годах.

На протяжении всего процесса важную роль играли другие члены группы по фундаментальным структурам данных и алгоритмам поиска. Анил Ганголли (Anil Gangolli) помогал при отборе материала для курса, Райан Эрнст (Ryan Ernst) подготовил большую часть инфраструктуры программирования, а Парамжит Оберой (Paramjit Oberoi) высказывал ценнейшие замечания на этапе написания книги. Нам доставило истинное удовольствие работать с ними, и мы признательны им за помощь.

Мы выражаем благодарность редакторам Питеру Гордону (Peter Gordon) и Грэгу Донку (Greg Doench), а также всему коллективу, собравшемуся под крышей издательства Addison-Wesley, в том числе главному редактору Джону Фуллеру, редактору по производству Мэри Кэсел Уилсон (Mary Kesel Wilson), выпускающему редактору Джилл Хоббс (Jill Hobbs), верстальщику и специалисту по LaTeX Лори Хьюз (Lori Hughes), за усилия по превращению рукописи в безупречную книгу.

Наконец, мы хотим поблагодарить наших друзей, семьи и коллег, которые прочли черновые варианты книги и поделились с нами замечаниями, исправлениями, предложениями, советами и т. д.: Гаспера Азмана (Gašper Ažman), Джона Баннинга (John Banning), Синтию Дворк (Cynthia Dwork), Германа Эпельмана (Hernan Epelman), Райана Эрнста (Ryan Ernst), Анила Ганголли (Anil Gangolli), Сьюзан Груббер (Susan Gruber), Джона Кальба (Jon Kalb), Роберта Лера (Robert Lehr), Дмитрия Лещинера (Dmitry Leshchiner), Тома Лондона (Tom London), Марка Манасси (Mark Manasse), Пола Макджонса (Paul McJones), Николаса Николова (Nicolas Nicolov), Гора Нишанова (Gor Nishanov), Парамжита Обероя (Paramjit Oberoi), Шона Пэрента (Sean Parent), Фернандо Пелличиони (Fernando Pelliccioni), Джона Рейзера (John Reiser), Роберта Роуза (Robert Rose), Стефана Варгиаса (Stefan Vargyas) и Адама Юнга (Adam Young). Благодаря им книга стала намного лучше.

Об авторах

Александр А. Степанов изучал математику в Московском государственном университете с 1967 по 1972 год. С 1972 года занимается программированием, сначала в Советском Союзе, а затем, после эмиграции в 1977 году, в США. Он принимал участие в программировании операционных систем, инструментальных средств программирования, компиляторов и библиотек. В работе по основаниям программирования ему оказывали поддержку компания Дженерал Электрик, Политехнический университет, компании Bell Labs, HP, SGI, Adobe, и – с 2009 года по сей день – A9.com, дочерняя компания Amazon, специализирующаяся на технологиях поиска. В 1995 году журнал «Dr. Dobb's Journal» присудил ему премию «За выдающиеся заслуги в программировании» за проектирование стандартной библиотеки шаблонов C++ (Standard Template Library).

Дэниэл Э. Роуз – ученый-исследователь, занимал руководящие должности в компаниях Apple, AltaVista, Xigo, Yahoo и A9.com. Круг его научных интересов охватывает технологии поиска, от низкоуровневых алгоритмов сжатия индекса до вопросов взаимодействия машины и человека в процессе поиска в веб. Роуз руководил в компании Apple группой, разработавшей систему локального поиска для компьютера Macintosh. Он обладатель докторской степени по когнитивистике и информатике, присужденной Калифорнийским университетом в Сан-Диего, а также степени бакалавра по философии, присужденной Гарвардским университетом.

От авторов

Разделение информатики и математики сильно обедняет обе науки. Лекции, положенные в основу этой книги, – предпринятая мной попытка показать, что обе деятельности – древнюю, восходящую к истокам нашей цивилизации, и самую что ни на есть современную – можно соединить.

Мне очень повезло, что мой друг Дэн Роуз, под руководством которого наша группа применяла принципы обобщенного программирования к проектированию поисковой системы, согласился преобразовать мои довольно бессвязные лекции в цельную книгу. Мы оба надеемся, что читателям понравится плод нашей совместной работы.

–А. А. С.

Книга, которую вы держите в руках, основана на заметках к курсу лекций «Алгоритмические путешествия», прочитанному Алексом Степановым в компании A9.com в 2012 году. Но в ходе нашей с Алексом работы по переложению материала курса в форму книги мы пришли к выводу, что могли бы поведать более интересную историю – об обобщенном программировании и его математических основаниях. Это побудило нас существенно изменить организацию книги и исключить целый раздел, посвященный теории множеств и математической логике, который как-то не укладывался в этот рассказ. Пришлось также добавить и удалить ряд деталей, чтобы сделать изложение более связным и доступным читателям, не имеющим основательной математической подготовки.

Алекс, в отличие от меня, получил математическое образование. Кое в чем мне было нелегко разобраться, но приложенные усилия позволили мне понять, что нуждается в дополнительном объяснении. Если иногда мы трактуем какие-то вопросы не так, как это сделал бы математик, или используем не вполне стандартную терминологию, или сознательно упрощаем изложение, то это целиком моя вина.

– Д. Э. Р.

Предисловие автора к русскому изданию

Эта книга родилась из курсов, которые я читал в Америке, но корни ее ведут к моим русским учителям. О Лобачевском и Пуанкаре я узнал от Эрнеста Борисовича Винберга, когда он нам, десятиклассникам второй школы, рассказывал про созданную Пуанкаре модель геометрии Лобачевского. Общую алгебру я полюбил, слушая лекции Александра Геннадиевича Куроша на мехмате МГУ. Он уверял, что «алгебра шлифует головы», и действительно на моей голове есть несколько до блеска отшлифованных мест. О теореме Лагранжа, да и вообще о группах я узнал от Анны Петровны Мишиной, а про классификацию простых полей – от Юрия Ивановича Манина. Пониманием важности истории математики я обязан Владимиру Игоревичу Арнольду, а об основаниях арифметики узнал из книги «Теоретическая арифметика» его отца, Игоря Владимировича Арнольда.

От моего учителя, Александра Самуиловича Завадье, идет моя любовь к грекам. Когда мне было 14 лет, он посоветовал мне читать Платона, особенно порекомендовав «Пир» в переводе Соломона Константиновича Апта. Спустя 50 лет я по-прежнему влюблен в Платона и считаю «Пир» сочинением почти божественным.

В Америке я работал с целым рядом замечательных программистов и многому от них научился, но моим настоящим учителем программирования был Александр Михайлович Гуревич, главный конструктор управляющей вычислительной машины ТА-100. У него я научился необходимости разрабатывать ортогональные и минимальные интерфейсы и десятки раз переписывать код, добиваясь красоты и оптимальности.

Я с некоторой робостью ожидаю появления этой книги в России, стране моих учителей, но надеюсь, что она донесет до молодых программистов хотя бы толику того чувства прекрасного, которым одарили меня мои наставники.

Глава 1

О чем эта книга

Невозможно познать мир, не познав математику.
Роджер Бэкон. *Большое сочинение*

Эта книга о программировании, но она отличается от большинства книг на ту же тему. Наряду с алгоритмами и кодом вы найдете в ней математические доказательства и исторические сведения о математических открытиях с античных времен до наших дней.

Если быть точным, то эта книга посвящена *обобщенному программированию*, подходу, сформировавшемуся в 1980-х годах и ставшему популярным после создания библиотеки Standard Template Library (STL) для языка C++ в 1990-х годах. Можно дать такое определение.

Определение 1.1. Обобщенным программированием называется подход к программированию, в котором упор делается на проектирование таких алгоритмов и структур данных, которые работали бы в наиболее общей ситуации без потери эффективности.

У тех, кому доводилось использовать STL, возможно, мелькнула мысль: «Как?! Вот это и есть обобщенное программирование? А как же шаблоны, характеристики итераторов и все прочее?» А это всего лишь языковые средства, обеспечивающие поддержку обобщенного программирования. И хотя, безусловно, следует знать, как использовать их эффективно, обобщенное программирование как таковое – это *отношение* к программированию, а не конкретный набор средств.

Мы считаем, что такое отношение – стремление писать код самым общим способом – должны воспринять все программисты. Компоненты хорошо написанной обобщенной программы проще повторно использовать и модифицировать, чем компоненты программы, в которой структуры данных, алгоритмы и интерфейсы отягощены ненужными предположениями о конкретном применении. Обобщенная программа оказывается одновременно проще и эффективнее.

1.1. Программирование и математика

Так откуда же проистекает обобщенное отношение к программированию и как ему научиться? Проистекает оно из математики, а точнее, из ее раздела, называемого *общей алгеброй*. Чтобы помочь вам разобраться в этом подходе, мы дадим краткое введение в общую алгебру, сосредоточившись на том, как рассуждать об объектах

в терминах абстрактных свойств операций над ними. Обычно этот предмет изучается на математических факультетах университетов, но мы полагаем, что он исключительно важен для понимания обобщенного программирования.

Вообще, многие фундаментальные идеи программирования берут начало в математике. Знания о том, как эти идеи возникли и развивались, поможет при обдумывании проекта программы. Например, «Начала» Евклида, написанные больше 2000 лет назад, и по сей день остаются одним из лучших примеров построения сложной системы из мелких, простых для понимания элементов.

Хотя существо обобщенного программирования – абстрагирование, абстракции – не возникают готовыми из ничего. Чтобы понять, как построить нечто общее, начать следует с чего-то конкретного. В частности, чтобы выявить подходящие абстракции, необходимо понять особенности конкретной предметной области.

Абстракции, рассматриваемые в общей алгебре, берут начало в конкретных результатах одного из самых старых разделов математики – *теории чисел*. Поэтому мы познакомимся с некоторыми ключевыми идеями теории чисел, которая занимается свойствами целых чисел и, в особенности делимостью.

Мыслительный процесс, вырабатывающийся при изучении математики, поможет вам усовершенствоваться в программировании. Но мы также покажем, что иногда и сами математические результаты становятся фундаментом современных программных приложений. В частности, в конце книги мы увидим, как некоторые результаты такого рода применяются в криптографических протоколах, лежащих в основе конфиденциальности в сети и электронной коммерции.

В этой книге мы постоянно переходим от математики к программированию и обратно. Важные математические идеи переплетаются с обсуждением как конкретных алгоритмов, так и методов обобщенного программирования. Некоторые алгоритмы мы упоминаем лишь вскользь, тогда как другие уточняем и обобщаем на протяжении всей книги. Две главы посвящены одной лишь математике, а две другие – исключительно программированию, но большая часть глав содержит материал, относящийся к тому и другому.

1.2. Исторические справки

Нам всегда казалось, что изучение становится проще и интереснее, если материал представлен в историческом контексте. Что происходило в описываемое время? Кем были участники событий, как они пришли к своим идеям? Была ли работа одного человека основана на результатах другого или это была попытка опровержения предшествующих результатов? Поэтому, знакомя читателя с математическими идеями, мы стараемся рассказывать об их истории и о людях, которые их выдвинули. Во многих случаях мы приводим краткие биографии математиков, сыгравших главную роль в описываемой истории. Это не исчерпывающие энциклопедические статьи, а всего лишь попытка погрузить конкретных людей в исторический контекст.

Хотя мы привержены историческому взгляду на вещи, это не означает, что книга задумана как история математики или что описанные в ней идеи представлены в

хронологическом порядке. Когда необходимо, мы свободно перемещаемся по векам и странам, но в любом случае стараемся представить все идеи на историческом фоне.

1.3. Требования к читателю

Поскольку значительная часть книги посвящена математике, у вас может сложиться впечатление, что для ее понимания нужна основательная математическая подготовка. Но хотя умение логически мыслить предполагается (впрочем, это в любом случае необходимо, чтобы стать хорошим программистом), никакие знания сверх школьной программы по алгебре и геометрии не требуются. В двух разделах показаны приложения, в которых используются элементы линейной алгебры (векторы и матрицы), но их можно без ущерба для понимания пропустить, если эти понятия вам незнакомы. В приложении A объясняются используемые обозначения.

Важная часть математики – умение строить формальное доказательство. В этой книге доказательств немало. Читать ее будет проще, если вы уже встречались с доказательствами – в школьной геометрии, в лекциях по теории автоматов в курсе информатики или математической логики. Мы описали несколько стандартных приемов доказательства – с примерами – в приложении B.

Мы предполагаем, что раз вы читаете эту книгу, то уже являетесь программистом. И в частности, достаточно хорошо знакомы с каким-нибудь типичным императивным языком программирования, например C, C++ или Java. Все наши примеры написаны на C++, но мы ожидаем, что вы сможете их понять, даже если никогда раньше не программировали на этом языке. Конструкции, уникальные для C++, объяснены в приложении C. Мы убеждены, что обсуждаемые в книге принципы применимы к программированию в целом, а не только к языку C++.

Многие рассматриваемые в этой книге вопросы обсуждаются под другим углом зрения и более формально в книге «Elements of Programming» Степанова и Макджонса. Для читателей, желающих более глубоко разобраться в теме, она станет полезным дополнением. Поэтому в некоторых местах мы отсылаем интересующихся читателей к соответствующим разделам книги «Elements of Programming».

1.4. План книги

Перед тем как переходить к деталям, полезно составить общее представление о том, чего мы собираемся достичь.

- В главе 2 излагается история древнего алгоритма умножения и рассказывается о том, как его можно улучшить.
- В главе 3 мы опишем некоторые ранние наблюдения, касающиеся свойств чисел, и рассмотрим эффективную реализацию алгоритма поиска простых чисел.
- В главе 4 мы познакомимся с алгоритмом нахождения наибольшего общего делителя (НОД), который ляжет в основу некоторых наших последующих абстракций и приложений.

- Глава 5 посвящена математическим результатам, в том числе двум теоремам, важнейшая роль которых станет ясна ближе к концу книги.
- В главе 6 приводится введение в общую алгебру, являющуюся источником самой идеи обобщенного программирования.
- В главе 7 показано, как эти математические идеи позволяют обобщить алгоритм арифметического умножения чисел на различные программные приложения.
- В главе 8 вводятся новые абстрактные математические структуры и объясняется, к каким новым приложениям они ведут.
- Глава 9 посвящена аксиоматическим системам, теориям и моделям – все это элементы обобщенного программирования.
- В главе 10 излагаются концепции обобщенного программирования и рассматриваются тонкости некоторых, на первый взгляд, простых задач.
- В главе 11 продолжается изучение ряда фундаментальных задач программирования и показывается, как можно воспользоваться теоретическими знаниями о проблеме для построения различных практических реализаций.
- В главе 12 рассматривается вопрос о том, как аппаратные ограничения могут стать стимулом для выработки нового подхода к старому алгоритму, и демонстрируются новые применения НОД.
- В главе 13 математические и алгоритмические результаты совместно используются для построения важного криптографического приложения.
- В главе 14 подытоживаются основные идеи, рассмотренные в книге.

На протяжении всей книги математика переплетается с программированием, хотя в одной-двух главах следы того или другого могут ненадолго теряться. Но каждая глава играет свою роль в следующей цепочке рассуждений, которая подводит итог книге:

> *Чтобы стать хорошим программистом, необходимо понимать принципы обобщенного программирования. Чтобы понимать принципы обобщенного программирования, нужно понимать абстракции. Чтобы понимать абстракции, нужно понимать лежащие в их основе математические идеи.*

Это и есть история, которую мы собираемся рассказать.

Глава 2

Первый алгоритм

Моисей быстро изучил арифметику и геометрию. ...Это знание он почерпнул у египтян, которые почитали математику превыше всех наук.

Филон Александрийский.
«Жизнь Моисея»

Алгоритмом называется конечная последовательность шагов по нахождению решения вычислительной задачи. Алгоритмы настолько тесно ассоциируются с программированием компьютеров, что большинство людей, знакомых с этим словом, вероятно, считает, что и сама идея алгоритма возникла в информатике. На самом же деле алгоритмы существуют уже тысячи лет. Математика изобилует алгоритмами, и некоторыми из них мы пользуемся ежедневно. Даже изучаемый в начальных классах способ сложения многозначных чисел – алгоритм.

Несмотря на долгую историю, понятие алгоритма существовало не всегда; его необходимо было изобрести. Мы не знаем, когда был изобретен первый алгоритм, но знаем, что в Древнем Египте они существовали по меньшей мере 4000 лет назад.

* * *

Древнеегипетская цивилизация сформировалась вокруг реки Нил, а сельское хозяйство в ней зависело от разливов, удобряющих почву. Проблема состояла в том, что каждый разлив Нила смывал все вешки, отмечающие границы земельных участков. Египтяне, использовавшие для измерения расстояний веревки, придумали процедуры, позволяющие справляться с записями и восстанавливать границы участков. За это отвечала особая группа жрецов, изучавших соответствующие математические методы; они назывались «натягивателями веревок», или гарпедонаптами. Позже греки назвали их *геометрами*, то есть «измерителями Земли».

К сожалению, сведений о математических знаниях египтян сохранилось немного. Лишь два относящихся к математике документа дошли до наших дней. Интересующий нас называется «Математический папирус Ринда» – по имени шотландского коллекционера XIX века, который купил его в Египте. Этот документ, написанный примерно в 1650 году до н. э. писцом по имени Ахмес, является сборником задач по арифметике и геометрии, а также включает ряд таблиц для вычислений. В числе прочего этот свиток содержит первые письменно зафиксированные алгоритмы – способы быстрого умножения и деления. Начнем с рассмотрения алгоритма быстрого умножения, который, как мы увидим ниже, и в наше время остается важной техникой вычислений.

2.1. Египетское умножение

В египетской системе счисления, как и в системах всех прочих древних цивилизаций, не использовалась позиционная нотация и отсутствовал способ для представления нуля. Поэтому умножение было чрезвычайно сложной операцией, доступной лишь немногим специально обученным людям (представьте, как бы вы стали перемножать большие числа, не имея ничего, кроме римской системы записи).

Но как мы определяем умножение? Если говорить неформально, то «сложить нечто с самим собой определенное число раз». Формально же можно выделить два случая: умножение на 1 и умножение на число, большее 1.

Умножение на 1 определяется так:

$$1a = a. \tag{2.1}$$

Далее нужно рассмотреть случай, когда мы хотим вычислить произведение уже вычисленного результата и еще одного экземпляра числа. Некоторые читатели узнают в этом процессе индукцию; позже мы опишем эту технику более формально.

$$(n + 1)a = na + a. \tag{2.2}$$

Один из способов умножить n на a состоит в том, чтобы n раз сложить a с самим собой. Однако если числа велики, то это может оказаться очень трудоемким делом, потому что необходимо $n - 1$ сложений. На C++ этот алгоритм можно записать так:

```
int multiply0(int n, int a) {
    if (n == 1) return a;
    return multiply0(n - 1, a) + a;
}
```

Строки кода соответствуют выражениям (2.1) и (2.2). Оба числа a и n должны быть положительны, а других чисел древние египтяне и не знали.

Описанный Ахмесом алгоритм – древние греки называли его «египетским умножением», а многие современные авторы «алгоритмом русского крестьянина»[1] – опирается на следующее тождество:

$$\begin{aligned} 4a &= ((a + a) + a) + a \\ &= (a + a) + (a + a). \end{aligned}$$

В основе этой оптимизации лежит правило ассоциативности сложения:

$$a + (b + c) = (a + b) + c.$$

---

[1] Многие специалисты по информатике узнали это название из книги Кнута «Искусство программирования», где говорится, что путешествующие по России XIX века видели, как крестьяне пользуются этим алгоритмом. Однако первое упоминание об этой истории встречается в изданной в 1911 году книге сэра Томаса Хита, который пишет: «Мне сообщали, что этот метод используется и сегодня (некоторые говорят, что в России, но *я не смог это проверить*)...»

Это позволяет нам вычислить сумму $a + a$ только один раз и, значит, уменьшить количество сложений.

Идея заключается в том, чтобы, повторно уменьшая вдвое n и удваивая a, вычислять сумму количества экземпляров, кратного степеням двойки. В то время алгоритмы не описывались в терминах переменных типа a и n; автор просто приводил пример и говорил: «А для других чисел поступай точно так же». Ахмес не был исключением; он продемонстрировал алгоритм, приведя следующую таблицу для умножения $n = 41$ на $a = 59$:

1	✓	59
2		118
4		236
8	✓	472
16		944
32	✓	1888

Числа в левом столбце – степени 2, каждое число в правом столбце (кроме первого) вдвое больше стоящего прямо над ним (сложение числа с самим собой – сравнительно простая операция). Числа, помеченные галочкой в среднем столбце, соответствуют единичным битам в двоичном представлении числа 41. Приведенная таблица, по существу, означает:

$$41 \times 59 = (1 \times 59) + (8 \times 59) + (32 \times 59),$$

причем каждое число в правой части можно получить удвоением 59 нужное число раз.

Алгоритм должен проверять, является n четным или нечетным, поэтому мы можем предположить, что египтяне знали об этом различии, хотя прямых доказательств у нас нет. Однако древние греки, утверждавшие, что узнали о математике от египтян, без сомнения знали о нем. Вот как они определяли[1], является число четным или нечетным (в современной нотации)[2]:

$$n = \frac{n}{2} + \frac{n}{2} \;\Rightarrow\; \text{even}(n);$$

$$n = \frac{n-1}{2} + \frac{n-1}{2} + 1 \;\Rightarrow\; \text{odd}(n).$$

Мы также воспользуемся следующим свойством:

$$\text{odd}(n) \Rightarrow \text{half}(n) = \text{half}(n - 1).$$

---

[1] Это определение встречается в датируемой I веком работе Никомаха из Герасы «Введение в арифметику», книга I, глава VII. Он пишет: «Чётным называется число, которое разделяется на два равных и не содержит единицы в середине; а нечётное число не может разделяться на два равных из-за присутствия единицы в середине».

[2] Символ ⇒ читается «влечет за собой». Сводка математических обозначений, используемых в этой книге, приведена в приложении А.

Вот как можно выразить египетский алгоритм умножения на C++:

```
int multiply1(int n, int a) {
    if (n == 1) return a;
    int result = multiply1(half(n), a + a);
    if (odd(n)) result = result + a;
    return result;
}
```

Для реализации `odd(x)` достаточно проверить младший бит x, а для реализации `half(x)` – сдвинуть x на один разряд вправо:

```
bool odd(int n) { return n & 0x1; }
int half(int n) { return n >> 1; }
```

Сколько сложений должна будет выполнить функция `multiply1`? При каждом ее вызове выполняется сложение, обозначенное знаком `+` в выражении `a + a`. Поскольку в процессе рекурсии мы уменьшаем значение n вдвое, то всего функция будет вызвана $\log n$ раз[1]. А в некоторых случаях придется еще выполнить сложение, обозначенное знаком `+` в выражении `result + a`. Таким образом, общее число сложений равно:

$$\#+(n) = \lfloor \log n \rfloor + (v(n) - 1),$$

где $v(n)$ – количество единиц в двоичном представлении n (*вес Хэмминга*). Следовательно, мы свели алгоритм со сложностью $O(n)$ к алгоритму со сложностью $O(\log n)$.

Является ли этот алгоритм оптимальным? Не всегда. Например, при умножении на 15 предыдущая формула дает результат:

$$\#+(15) = 3 + 4 - 1 = 6.$$

Но можно умножить на 15, выполнив всего 5 сложений:

```
int multiply_by_15(int a) {
    int b = (a + a) + a;  // b == 3*a
    int c = b + b;  // c == 6*a
    return (c + c) + b; // 12*a + 3*a
}
```

Такая последовательность операций называется *цепочкой сложений*. В данном случае мы нашли оптимальную цепочку сложений для умножения на 15. Тем не менее алгоритм Ахмеса достаточно хорош для большинства применений.

Упражнение 2.1. Найти оптимальные цепочки сложений для всех $n < 100$.

Возможно, читатель заметил, что вычисления можно ускорить, если обратить порядок аргументов в случае, когда первый больше второго (например, вычислить

---

[1] Всюду в этой книге под «log» понимается логарифм по основанию 2, если явно не оговорено противное.

3×15 проще, чем 15×3). Это действительно так, и египтяне об этом знали. Но мы не станем сейчас добавлять эту оптимизацию, потому что в главе 7 этот алгоритм будет обобщен на случай, когда типы аргументов необязательно совпадают и их порядок, следовательно, небезразличен.

2.2. Улучшение алгоритма

С точки зрения количества сложений наша функция `multiply1` работает хорошо, однако она выполняет $\lfloor \log n \rfloor$ рекурсивных вызовов. Поскольку вызов функции обходится дорого, мы хотим изменить программу, избавившись от накладных расходов.

При этом мы будем придерживаться принципа «часто много работы сделать проще, чем мало». Точнее, мы будем вычислять выражение

$$r + na,$$

где в r аккумулируются частичные произведения na. Иными словами, мы будем выполнять операцию *умножить и аккумулировать*, а не просто *умножить*. Этот принцип справедлив не только в программировании, но и в проектировании программного обеспечения и в математике, где часто бывает проще доказать общий результат, чем частный случай.

Вот как выглядит наша функция умножения с аккумулированием:

```
int mult_acc0(int r, int n, int a) {
    if (n == 1) return r + a;
    if (odd(n)) {
      return mult_acc0(r + a, half(n), a + a);
    } else {
      return mult_acc0(r, half(n), a + a);
    }
}
```

У этой функции есть инвариант: $r + na = r_0 + n_0 a_0$, где r_0, n_0 и a_0 – начальные значения переменных.

Мы можем еще улучшить этот код, упростив рекурсию. Отметим, что два рекурсивных вызова различаются только первым аргументом. Вместо двух рекурсивных вызовов для случаев четного и нечетного чисел мы можем просто изменить значение r перед рекурсией:

```
int mult_acc1(int r, int n, int a) {
    if (n == 1) return r + a;
    if (odd(n)) r = r + a;
      return mult_acc1(r, half(n), a + a);
}
```

Теперь наша функция обладает свойством *хвостовой рекурсии* – рекурсия происходит только при возврате значения. Скоро мы воспользуемся этим фактом.

Сделаем два наблюдения:

- n редко бывает равно 1;
- если n четно, то не имеет смысла проверять, равно ли оно 1.

Поэтому можно уменьшить количество сравнений с 1 вдвое, если проверять на нечетность сначала:

```
int mult_acc2(int r, int n, int a) {
    if (odd(n)) {
        r = r + a;
        if (n == 1) return r;
    }
    return mult_acc2(r, half(n), a + a);
}
```

Некоторые программисты думают, что оптимизирующий компилятор сам произведет такие преобразования, но это редко оказывается правдой: компилятор не способен преобразовать один алгоритм в другой.

То, что мы сейчас имеем, уже неплохо, но в конечном итоге мы хотим вообще устранить рекурсию и избавиться от накладных расходов на вызов функции. Это проще сделать, если функция обладает свойством строгой хвостовой рекурсии.

Определение 2.1. Говорят, что функция обладает свойством **строгой хвостовой рекурсии**, если во всех рекурсивных вызовах формальные параметры совпадают с соответствующими аргументами.

Чтобы добиться этого, мы просто присвоим нужные значения переменным до того, как передать их рекурсивному вызову:

```
int mult_acc3(int r, int n, int a) {
    if (odd(n)) {
        r = r + a;
        if (n == 1) return r;
    }
    n = half(n);
    a = a + a;
    return mult_acc3(r, n, a);
}
```

Теперь не составляет труда преобразовать этот код в итеративный, заменив хвостовую рекурсию циклом `while(true)`:

```
int mult_acc4(int r, int n, int a) {
    while (true) {
        if (odd(n)) {
            r = r + a;
            if (n == 1) return r;
        }
        n = half(n);
        a = a + a;
    }
}
```

Имея оптимизированную таким образом функцию умножения с аккумулированием, мы можем написать новую версию функции умножения, в которой будет вызываться вспомогательная функция умножения с аккумулированием:

```
int multiply2(int n, int a) {
    if (n == 1) return a;
    return mult_acc4(a, n - 1, a);
}
```

Отметим, что мы сэкономили на одном обращении к `mult_acc4`, сразу установив результат в `a` вместо 0.

Это хорошо во всех случаях, кроме ситуации, когда n является степенью 2. Первым делом мы вычитаем 1, а это значит, что `mult_acc4` передается число, двоичное представление которого содержит только единицы, то есть имеет место худший для алгоритма случай. Чтобы избежать этого, мы проделаем часть работы заранее, если n четно: будем делить на два (одновременно вдвое увеличивая a), пока n не станет нечетным:

```
int multiply3(int n, int a) {
    while (!odd(n)) {
        a = a + a;
        n = half(n);
    }
    if (n == 1) return a;
    return mult_acc4(a, n - 1, a);
}
```

Но теперь функция `mult_acc4` делает одну лишнюю проверку на $n = 1$, потому что при ее вызове n заведомо четно. Поэтому перед вызовом мы еще раз разделим на два второй аргумент, умножим на два третий и получим такую окончательную версию:

```
int multiply4(int n, int a) {
    while (!odd(n)) {
        a = a + a;
        n = half(n);
    }
    if (n == 1) return a;
    // even(n - 1) ⇒ n - 1 ≠ 1
    return mult_acc4(a, half(n - 1), a + a);
}
```

Переписывание кода

Как мы видели на примере преобразований алгоритма умножения, переписывание кода – важный шаг. Никто не пишет хороший код с первой попытки; чтобы найти самый общий или самый эффективный способ решения задачи, требуется много итераций. Склад ума программиста не должен быть однопроходным.

Настает момент, когда в голову закрадывается мысль: «Подумаешь, еще одна операция, какая разница?». Но, возможно, ваш код будет многократно использоваться на протяжении многих лет (на самом деле временные поделки очень часто живут дольше

всего). И кроме того, дешевая операция, которую вы сэкономили, в будущей версии программы вполне может быть заменена очень дорогостоящей.
И еще польза от борьбы за эффективность заключается в том, что по ходу дела вы начинаете глубже понимать задачу. А чем глубже вы ее понимаете, тем эффективнее реализация – это круг, но только не порочный, а благодетельный.

2.3. Заключительные мысли

В курсе элементарной алгебры студенты учатся преобразовывать выражения с целью упрощения. В наших последовательных реализациях египетского алгоритма умножения мы занимались похожим делом – преобразовывали код, стремясь сделать его более понятным и эффективным. Любой программист должен взять за правило продолжать преобразование кода, пока не получится устраивающий его результат.

Мы видели, откуда в Древнем Египте взялись математики, что привело к появлению первого известного нам алгоритма. Впоследствии мы еще вернемся к этому алгоритму и обобщим его. А пока переместимся на тысячу с лишним лет вперед и познакомимся с некоторыми математическими открытиями, совершенными в античной Греции.