Оглавление

От автора

I hope this Russian edition of my book will be useful to future competitive programmers in Russia. I appreciate the competitive programming culture in Russia, especially the international training camps which are known for challenging and high-quality problems. Indeed,competitive programming is a way to learn algorithms together with people from different countries. It does not matter where you come from – everybody is interested in creating efficient algorithms, which is the core of computer science.

Antti Laaksonen

Я надеюсь, что русское издание моей книги будет полезно будущим спортивным программистам России. Я ценю российскую культуру олимпиадного программирования, а в особенности международные учебные кэмпы, которые известны своими сложными и интересными задачами. Так спортивное программирование объединяет людей из разных стран в изучении алгоритмов. Не важно, откуда вы, важна ваша заинтересованность в создании эффективных алгоритмов, которые являются основой Computer Science.

Антти Лааксонен

Вступительное слово Алексея Малеева, основателя Moscow Workshops ICPC

Про спортивное программирование в России знают мало. А между тем именно российские команды ежегодно выигрывают самое престижное мировое соревнование по программированию для студентов – International Collegiate Programming Contest (ICPC). История участия студентов из российских вузов на старейшем чемпионате ICPC отсчитывается с далекого 1993 года. В 2000 году студенты Санкт-Петербургского государственного университета показали необычайный прогресс и впервые среди российских команд вырвались в чемпионы мира. С этого года российские команды завоевали 32 золотые медали. Для сравнения: студенты из Китая всего 13 раз удостаивались золота за этот период, европейские участники – 11, США – всего 6. В мире наиболее сильную подготовку демонстрируют российские, китайские команды, студенты из Азии и Польши.

Секрет успеха наших команд во многом объясняется усиленными тренировками. Но важна система, в рамках которой происходит подготовка. И здесь именно России удалось выстроить сильную тренировочную

структуру для олимпиадных программистов. Чемпионов готовят ведущие вузы страны, технические, и не только: Университет ИТМО, Московский физико-технический институт, Санкт-Петербургский государственный университет, Московский государственный университет им. Ломоносова, Уральский федеральный университет, Саратовский университет и др.

Объединившись вместе, консорциум из вышеперечисленных вузов в 2012 году запустил первый глобальный проект по подготовке студентов к чемпионату по спортивному программированию на кампусе МФТИ – **Moscow Workshops ICPC**. В формате учебно-тренировочных сборов студенты решают контексты и разбирают их с лучшими тренерами. Важная компонента **Moscow Workshops ICPC** – это интернациональность. Кэмпы открывают свои двери для молодых участников всего мира, предлагая им не только учиться, но и путешествовать, расширять свой кругозор. На каждом воркшопе участники обогащают свои знания о других странах – посещают экскурсии, знакомятся с местной культурой и людьми.

Уровень подготовки воркшопов очень высокий. Можно сказать, что те, кто прошел обучение в **Moscow Workshops ICPC** и успешно прошел отбор начальных этапов ICPC, уже представляют элиту мира программирования. За этими ребятами охотятся крупнейшие IT-компании, они получают лучшие предложения о работе.

В 2016 году по образовательной франшизе мы запустили тренировочный лагерь в Гродно (Западная Белоруссия). С тех пор мы открыли регулярно действующие площадки в Барселоне (Испания) и Коллам (Индия), а в этом году запускаем воркшоп в одном из самых восточных городов России – во Владивостоке. На данный момент в сборах уже приняли участие команды 167 университетов из 50 стран Европы, Азии, Южной и Северной Америки, Африки и Австралии. В 2016 и 2017 годах восемь из двенадцати команд, завоевавших медали в финале чемпионата ICPC, принимали участие в подготовительных сборах **Moscow Workshops ICPC**. В 2018 году их число выросло – 10 из 13 медалистов готовились на воркшопах.

Спортивное программирование – это самый перспективный интеллектуальный вид спорта, который можно назвать шахматами будущего. Уже сейчас им увлекаются лучшие умы планеты, и число участников растет год от года. Рост популярности олимпиадного программирования положительно влияет на другие сферы жизнедеятельности человека. Навыки быстрого решения сложнейших задач помогают сегодняшним студентам в будущем справляться с реальными проблемами человечества креативно и эффективно. В ходе соревнований ребята учатся справляться со сложными моральными и психологическими нагрузками. За это время у них вырабатывается навык управлять рисками, ведь, чтобы выйти в финал ICPC, нужно потратить около 5 лет. Не каждому это суждено. Стрессоустойчивость и нацеленность на результат – это те качества, которые необходимы технологическим предпринимателям для запуска собственных проектов.

Система **Moscow Workshops ICPC** также включает онлайн-курсы на платформе Coursera и онлайн-чемпионаты Opencup.ru, что в совокупности дает любому студенту из любого уголка мира, вне зависимости от принадлежности к ведущим мировым университетам, реализоваться в области алгоритмов и программирования.

Уверены, что последователей ICPC будет все больше. Всех студентов, которые горят желанием развиваться в программировании и «кодят», не замечая хода времени, мы ждем на кэмпах **Moscow Workshops ICPC** и желаем всегда стремиться к результатам, которые кажутся невозможными.

С уважением, Алексей Малеев,

директор по технологическому предпринимательству МФТИ,
основатель Moscow Workshops ICPC

Отзыв Дмитрия Гришина, основателя Mail.Ru Group

В мои студенческие годы всегда было много программирования, и я часто участвовал в различных олимпиадах, но только теперь понимаю, насколько мне тогда не хватало подобной книги. Всем, кто стремится покорить олимп международных чемпионатов по программированию, она точно придется по вкусу и поможет на этом нелегком пути достижения цели. Мы в Mail.Ru Group тоже создали несколько чемпионатов по программированию, которые за последние 7 лет стали популярны не только в России, но и далеко за рубежом – уже более 20 стран принимает участие и больше 175 тысяч человек соревновались в Russian Code Cup, VK Cup, Russian AI Cup, ML BootCamp и др. Уверен, что эта книга поможет и тем, кто собирается принять в них участие тоже.

Дмитрий Гришин,

основатель, соучредитель и председатель совета директоров Mail.Ru Group,
основатель венчурного фонда Grishin Robotics

Благодарность от редакции

Редакция издательства «ДМК-Пресс» выражает огромную благодарность Центру развития ИТ-образования при Московском физико-техническом институте и лично его руководителю Алексею Малееву и Mail.Ru Group за помощь в подготовке выпуска этой книги и, конечно, за отличную подготовку наших чемпионов по программированию.

Отзыв Нияза Нигматуллина, двукратного чемпиона мира ACM ICPC 2012 и 2013 годов

Эта книга помогает познакомиться с олимпиадным программированием. Она рассчитана больше на начинающих либо не очень опытных в этом деле читателей, чем на продвинутых. Здесь подробно описано, как проходят олимпиады, что требуется, в чем их цель. Рассказано, как нужно к ним готовиться, какие качества в себе вырабатывать и какие есть способы тренироваться. Подробно разобраны базовые темы, трюки и алгоритмы. Средние и сложные методы преподносятся без четких доказательств. Присутствует много полезных олимпиадных «трюков», которые редко встречаются в научной литературе. Большая часть популярных алгоритмов и методов, используемых в решении задач на олимпиадах, упомянута в этой книге.

К каждой теме автор приложил код на языке C++, что позволяет лучше понять описанное и увидеть способы реализации. Есть отдельная глава, в которой описываются те конструкции и библиотеки C++, которые часто используются на олимпиадах, и только те, которые необходимы: описываются они без подробностей и большой теории, а только на том уровне, чтобы читатель смог ими воспользоваться. Если читатель хочет разобраться в них подробнее, то следует почитать дополнительную информацию по языку из специальных источников. Кроме того, описанные инструменты языка C++ сравниваются на протяжении всей книги на эффективность и удобство использования на олимпиадах.

В книге раскрыт уникальный опыт участников и тренеров олимпиадного программирования.

По правилам этих соревнований в финалах нельзя участвовать более двух раз. За более чем сорокалетнюю историю этих чемпионатов всего шесть человек стали двукратными чемпионами мира, все из Санкт-Петербурга. Четверо – из Университета ИТМО: Геннадий Короткевич, Нияз Нигматуллин, Евгений Капун и Михаил Кевер, и двое – из СПбГУ: Николай Дуров и Андрей Лопатин.

Предисловие ко второму изданию

Во второе издание книги добавлено несколько новых разделов, в которых рассматриваются темы повышенного уровня: вычисление преобразования Фурье, нахождение потоков минимальной стоимости в графах и использование конечных автоматов в задачах о строках.

Я благодарен Олли Матилайнену, который прочитал большую часть нового материала и высказал много полезных замечаний и предложений.

Антти Лааксонен

Хельсинки, Финляндия
Февраль 2020

Предисловие к первому изданию

Эта книга задумана как содержательное введение в современное олимпиадное программирование. Предполагается, что читатель уже знаком с основами программирования, но опыт проектирования алгоритмов или участия в соревнованиях по программированию не обязателен. Поскольку в книге рассматривается широкий круг тем разной степени трудности, она будет полезна как начинающей, так и более искушенной аудитории.

Соревнования по программированию имеют довольно долгую историю. *Международные студенческие олимпиады по программированию* (ICPC) для студентов университетов проводились уже в 1970-х годах, а первая *Международная олимпиада по информатике* для учащихся старших классов состоялась в 1989 году. Ныне оба мероприятия стали регулярными и собирают множество участников со всего мира.

В наши дни олимпиадное программирование популярно как никогда. Важную роль в его распространении сыграл интернет. Существует активное сетевое сообщество увлеченных этим движением программистов, каждую неделю проводятся различные соревнования. Одновременно растет и трудность заданий. Технические приемы, которыми еще несколько лет назад владели лишь лучшие из лучших, стали стандартными инструментами, известными многим.

Своими корнями олимпиадное программирование уходит в научное исследование алгоритмов. Но если ученый приводит доказательство работоспособности своего алгоритма, то олимпиадный программист *реализует* алгоритм и подает его на вход системы оценивания результатов. Эта система прогоняет алгоритм через различные тесты, и если все они проходят, то решение принимается. Это важный элемент олимпиадного программирования, который позволяет *автоматически* получить убедительные аргументы в пользу правильности алгоритма. Вообще, олимпиадное программирование стало отличным способом изучения алгоритмов, т. к. от участника требуется спроектировать алгоритм, который действительно работает, а не ограничиваться наброском идей, может, правильных, а может, и нет.

У олимпиадного программирования есть еще одно достоинство – конкурсные задачи заставляют думать. В формулировках задач не бывает никакого жульничества, в отличие от многих курсов по алгоритмам, где вам предлагают решить красивую задачу, но только последнее предложение звучит, к примеру, так: «*Подсказка*: для решения задачи модифицируйте алгоритм Дейкстры». Ну, а дальше думать особенно нечего, поскольку по-

ход к решению уже известен. В олимпиадном программировании так не бывает. У вас имеется полный комплект инструментов, а уж какими из них воспользоваться, решайте сами.

Решение олимпиадных задач развивает навыки программирования и отладки. Обычно решение засчитывается, только если пройдены все тесты, поэтому программист, стремящийся к успеху, должен реализовывать алгоритм без ошибок. Это умение высоко ценится в программной инженерии, так что неудивительно, что ИТ-компании проявляют интерес к имеющим опыт олимпиадного программирования.

Чтобы стать хорошим олимпиадным программистом, нужно много времени, зато на этом пути можно и многому научиться. Можете быть уверены, что, потратив время на чтение этой книги, решение задач и участие в различных соревнованиях, вы будете лучше понимать, как устроены алгоритмы.

Буду рад вашим отзывам. Вы можете писать мне по адресу ahslaaks@cs.helsinki.fi.

Я благодарен многим людям, приславшим отзывы на черновые редакции этой книги. Они очень помогли сделать книгу лучше. Отдельное спасибо Микко Эрвасти (Mikko Ervasti), Яанне Юннила (Janne Junnila), Яанне Коккала (Janne Kokkala), Туукка Корхонену (Tuukka Korhonen), Патрику Остегярду (Patric Östergård) и Роопе Сальми (Roope Salmi), написавшим подробные рецензии на рукопись. Я также признателен Саймону Рису (Simon Rees) и Уэйну Уилеру (Wayne Wheeler) из издательства Springer за плодотворное сотрудничество в ходе подготовки книги к печати.

Антти Лааксонен

Хельсинки, Финляндия
Октябрь, 2017

Глава 1

Введение

В этой главе рассказывается, что такое олимпиадное программирование, представлен план книги и обсуждаются дополнительные образовательные ресурсы.

В разделе 1.1 описаны элементы олимпиадного программирования, перечислены некоторые популярные соревнования по программированию и даны рекомендации о том, как готовиться к соревнованиям.

В разделе 1.2 обсуждаются цели этой книги и рассматриваемые в ней темы, а также кратко излагается содержание каждой главы.

В разделе 1.3 мы познакомимся со сборником задач CSES. Решение задач параллельно чтению этой книги – отличный способ научиться олимпиадному программированию.

В разделе 1.4 обсуждаются другие книги по олимпиадному программированию и проектированию алгоритмов.

1.1. Что такое олимпиадное программирование?

Олимпиадное программирование состоит из двух частей – проектирования алгоритмов и реализации алгоритмов.

Проектирование алгоритмов. По сути своей, олимпиадное программирование – это придумывание эффективных алгоритмов решения корректно поставленных вычислительных задач. Для проектирования алгоритмов необходимы навыки в решении задач и знание математики. Зачастую решение появляется в результате сочетания хорошо известных методов и новых идей.

Важную роль в олимпиадном программировании играет математика. На самом деле четких границ между проектированием алгоритмов и математикой не существует. Эта книга не требует глубокой математической подготовки. В приложении, которое можно использовать как справочник, описаны некоторые встречающиеся в книге математические понятия и методы, например множества, математическая логика и функции.

Реализация алгоритмов. В олимпиадном программировании решение задачи оценивается путем проверки реализованного алгоритма на ряде тестов. Поэтому придумать алгоритм недостаточно, его еще нуж-

но корректно реализовать, для чего требуется умение программировать. Олимпиадное программирование сильно отличается от традиционной программной инженерии: программы короткие (несколько сотен строк – уже редкость), писать их нужно быстро, а сопровождение после соревнования не требуется.

В настоящее время на соревнованиях по программированию популярнее всего языки C++, Python и Java. Например, среди 3000 лучших участников Google Code Jam 2017 79 % писали на C++, 16 % – на Python и 8 % – на Java. Многие считают C++ самым подходящим выбором для олимпиадного программиста. К его достоинствам можно отнести очень высокую эффективность и тот факт, что в стандартной библиотеке много разнообразных структур данных и алгоритмов.

Все примеры в книге написаны на C++, в них часто используются структуры данных и алгоритмы из стандартной библиотеки. Код отвечает стандарту C++11, который разрешен в большинстве современных соревнований. Если вы еще не знаете C++, самое время начать его изучение.

1.1.1. *Соревнования по программированию*

Международная олимпиада по информатике (**IOI**) – ежегодное соревнование для старшеклассников. От каждой страны допускается команда из четырех человек. Обычно набирается около 300 участников из 80 стран.

IOI проводится в течение двух дней. В каждый день участникам предлагается решить три трудные задачи, на решение отводится пять часов. Задачи разбиты на подзадачи, за каждую из которых начисляются баллы. Хотя участники являются членами команды, соревнуются они самостоятельно.

Участники IOI отбираются на национальных олимпиадах. IOI предшествует множество региональных соревнований, например: Балтийская олимпиада по информатике (BOI), Центрально-Европейская олимпиада по информатике (CEOI) и Азиатско-Тихоокеанская олимпиада по информатике (APOI).

ICPC (Международная студенческая олимпиада по программированию) проводится ежегодно для студентов университетов. В каждой команде три участника; в отличие от IOI, студенты работают вместе, и каждой команде выделяется только один компьютер.

ICPC включает несколько этапов, лучшие команды приглашаются на финальный этап мирового первенства. Хотя в соревновании принимают участие тысячи студентов, количество участников финала ограничено[1], поэтому даже сам выход в финал считается большим достижением.

На соревновании ICPC у команды есть пять часов на решение примерно десяти алгоритмических задач. Решение засчитывается, только если прог-

[1] Точное количество участников финала от года к году меняется. В 2017 году их было 133.

рамма прошла все тесты и показала свою эффективность. В ходе соревнования участники могут видеть результаты соперников, но в начале пятого часа табло замораживается, и результаты последних прогонов не видны.

Онлайновые соревнования. Существует также много онлайновых соревнований, куда допускаются все желающие. В настоящий момент наиболее активен сайт Codeforces, который организует конкурсы почти каждую неделю. Отметим также сайты AtCoder, CodeChef, CS Academy, HackerRank и Topcoder.

Некоторые компании организуют онлайновые соревнования с очными финалами, например: Facebook Hacker Cup, Google Code Jam и Yandex.Algorithm. Разумеется, компании используют такие соревнования для подбора кадров: достойно выступить в соревновании – хороший способ доказать свои таланты в программировании.

1.1.2. *Рекомендации желающим поучаствовать*

Чтобы научиться олимпиадному программированию, нужно напряженно работать. Но способов приобрести практический опыт много, одни лучше, другие хуже.

Решая задачи, нужно иметь в виду, что не так важно *количество* решенных задач, как их *качество*. Возникает соблазн выбирать красивые и легкие задачи, пропуская те, что кажутся трудными и требующими кропотливого труда. Но чтобы отточить свои навыки, нужно отдавать предпочтение именно задачам второго типа.

Важно и то, что большинство олимпиадных задач решается с помощью простого и короткого алгоритма, самое трудное – придумать этот алгоритм. Смысл олимпиадного программирования – не в заучивании сложных и малопонятных алгоритмов, а в том, чтобы научиться решать трудные задачи, обходясь простыми средствами.

Наконец, есть люди, презирающие реализацию алгоритмов, им доставляет удовольствие проектировать алгоритмы, а реализовывать их скучно. Однако умение быстро и правильно реализовать алгоритм – важное преимущество, и этот навык поддается тренировке. Будет очень плохо, если вы потратите большую часть отведенного времени на написание кода и поиск ошибок, а не на обдумывание того, как решить задачу.

1.2. Об этой книге

Программа IOI [17] определяет, какие темы могут предлагаться на Международных олимпиадах по информатике. Именно эта программа стала отправной точкой при отборе материала. Но обсуждаются также более сложные темы, которые исключены из IOI (по состоянию на 2017 год), но могут встречаться в других соревнованиях. К ним относятся, например, максимальные потоки, игра ним и суффиксные массивы.

Многие темы олимпиадного программирования обсуждаются в стандартных учебниках по алгоритмам, но есть и отличия. Например, во многих книгах реализация алгоритмов сортировки и основных структур данных рассматривается с нуля, но такие знания на олимпиаде несущественны, потому что можно пользоваться средствами из стандартной библиотеки. С другой стороны, некоторые темы хорошо известны в олимпиадном сообществе, но редко встречаются в учебниках. Примером может служить дерево отрезков – структура данных, которая используется для решения многих задач без привлечения более хитроумных алгоритмов.

Одна из целей этой книги – *документировать* приемы олимпиадного программирования, которые обычно обсуждаются только на форумах и в блогах. Там, где возможно, даются ссылки на научную литературу по таким методам. Но сделать это можно не всегда, потому что многие методы ныне стали частью *фольклора*, и никто не знает имени первооткрывателя.

Книга организована следующим образом.

- В главе 2 рассматриваются средства языка программирования C++, а затем обсуждаются рекурсивные алгоритмы и поразрядные операции.
- Глава 3 посвящена эффективности: как создавать алгоритмы, способные быстро обрабатывать большие наборы данных.
- В главе 4 обсуждаются алгоритмы сортировки и двоичного поиска с упором на их применение в проектировании алгоритмов.
- В главе 5 дается обзор избранных структур данных в стандартной библиотеке C++: векторов, множеств и отображений.
- Глава 6 представляет собой введение в один из методов проектирования алгоритмов – динамическое программирование. Здесь же приводятся примеры задач, которые можно решить этим методом.
- В главе 7 рассматриваются элементарные алгоритмы на графах, в т. ч. поиск кратчайших путей и минимальные остовные деревья.
- В главе 8 речь пойдет о более сложных вопросах проектирования алгоритмов, в частности об алгоритмах с параллельным просмотром разрядов и амортизационном анализе.
- Тема главы 9 – эффективная обработка запросов по диапазонам массива, таких как вычисление сумм элементов и нахождение минимальных элементов.
- В главе 10 представлены специальные алгоритмы для деревьев, в т. ч. методы обработки запросов к дереву.
- В главе 11 обсуждаются разделы математики, часто встречающиеся в олимпиадном программировании.

- В главе 12 описываются дополнительные алгоритмы на графах, например поиск компонент сильной связности и вычисление максимального потока.
- Глава 13 посвящена геометрическим алгоритмам, в ней описаны методы, позволяющие удобно решать геометрические задачи.
- В главе 14 рассматриваются методы работы со строками, в т. ч. хеширование, Z-алгоритм и суффиксные массивы.
- В главе 15 обсуждаются избранные дополнительные темы, например алгоритмы, основанные на идее квадратного корня, и оптимизация динамического программирования.

1.3. Сборник задач CSES

В *сборнике задач CSES* представлены задачи для практики в олимпиадном программировании. Расположены они в порядке возрастания трудности, а в этой книге обсуждаются все методы, необходимые для их решения. Сборник задач доступен по адресу:

https://cses.fi/problemset/.

Посмотрим, как решить первую задачу из него. Она называется «Странный алгоритм» и формулируется следующим образом.

Рассмотрим алгоритм, принимающий на входе целое положительное число n. Если n четно, то алгоритм делит его на два, а если нечетно, то умножает на три и прибавляет 1. Например, для $n = 3$ получается следующая последовательность:

$$3 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1.$$

Ваша задача заключается в том, чтобы промоделировать выполнение этого алгоритма для любого заданного n.

Вход

Единственная строка, содержащая целое число n.

Выход

Печатает строку, содержащую все значения, вычисляемые алгоритмом.

Ограничения

- $1 \leqslant n \leqslant 10^6$

Пример

Вход:

3

Выход:

3 10 5 16 8 4 2 1

Эта проблема имеет отношение к знаменитой *гипотезе Коллатца*, которая утверждает, что описанный выше алгоритм завершается для любого n. До сих пор она остается недоказанной. Но в этой задаче мы знаем, что начальное значение n не превышает миллиона, что существенно упрощает проблему.

Это простая задача моделирования, не требующая глубоких размышлений. Вот как ее можно было бы решить на C++:

```
#include <iostream>

using namespace std;

int main() {
    int n;
    cin >> n;
    while (true) {
        cout << n << " ";
        if (n == 1) break;
        if (n%2 == 0) n /= 2;
        else n = n*3+1;
    }
    cout << "\n";
}
```

Сначала программа читает входное число n, затем моделирует алгоритм и печатает значение n после каждого шага. Легко проверить, что этот код правильно обрабатывает случай $n = 3$, приведенный в формулировке задачи.

Теперь время *отправить* этот код в CSES. Для каждого теста CSES сообщает, пройден он или нет, и показывает вход, ожидаемый и реальный выход.

По результатам тестирования CSES формирует следующий отчет:

test	verdict	time (s)
#1	ACCEPTED	0.06 / 1.00
#2	ACCEPTED	0.06 / 1.00
#3	ACCEPTED	0.07 / 1.00
#4	ACCEPTED	0.06 / 1.00
#5	ACCEPTED	0.06 / 1.00
#6	TIME LIMIT EXCEEDED	_ / 1.00
#7	TIME LIMIT EXCEEDED	_ / 1.00
#8	WRONG ANSWER	0.07 / 1.00
#9	TIME LIMIT EXCEEDED	_ / 1.00
#10	ACCEPTED	0.06 / 1.00

Это означает, что некоторые тесты наш код прошел (ACCEPTED), на некоторых оказался слишком медленным (TIME LIMIT EXCEEDED), а в одном случае дал неверный результат (WRONG ANSWER). Вот уж действительно неожиданно!

Первым не прошел тест для n = 138 367. Локально прогнав программу в этом случае, мы убедимся, что она действительно работает долго. На самом деле она вообще не завершается.

Причина ошибки в том, что в процессе моделирования n может оказаться слишком большим, в том числе больше максимального значения, представимого типом `int`. Чтобы решить проблему, достаточно заменить тип `int` на `long long`. Тогда получится то, что нужно:

test	verdict	time (s)
#1	ACCEPTED	0.05 / 1.00
#2	ACCEPTED	0.06 / 1.00
#3	ACCEPTED	0.07 / 1.00
#4	ACCEPTED	0.06 / 1.00
#5	ACCEPTED	0.06 / 1.00
#6	ACCEPTED	0.05 / 1.00
#7	ACCEPTED	0.06 / 1.00
#8	ACCEPTED	0.05 / 1.00
#9	ACCEPTED	0.07 / 1.00
#10	ACCEPTED	0.06 / 1.00

Как видно из этого примера, даже в очень простые алгоритмы могут вкрасться тонкие ошибки. Олимпиадное программирование учит, как писать алгоритмы, которые действительно работают.

1.4. Другие ресурсы

Помимо этой, уже есть и другие книги по олимпиадному программированию. Первой была книга Skiena, Revilla «Programming Challenges» [32], вышедшая в 2003 году. Позже вышла книга Halim, Halim «Competitive Programming 3» [16]. Обе ориентированы на читателя, не имеющего опыта олимпиадного программирования.

Книга «Looking for a Challenge?» [8] – сборник трудных задач, предлагавшихся на польских олимпиадах, – рассчитана на более подготовленного читателя. Самое интересное в ней – подробный анализ решений.

Разумеется, для подготовки к олимпиадам подойдут и общие книги по алгоритмам. Самая всеобъемлющая из них – книга Cormen, Leiserson,

Rivest, Stein «Introduction to Algorithms»[2] [7], которую также называют просто *CLRS*. Это прекрасный источник для тех, кто хочет узнать обо всех деталях алгоритма и познакомиться со строгим доказательством.

В книге Kleinberg, Tardos «Algorithm Design» [22] основное внимание уделяется технике проектирования алгоритмов и во всех деталях обсуждаются такие темы, как метод «разделяй и властвуй», жадные алгоритмы, динамическое программирование и вычисление максимального потока. Книга Skiena «The Algorithm Design Manual» [31] – практическое руководство, содержащее обширный каталог вычислительных задач и способов их решения.

[2] *Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн*. Алгоритмы. Построение и анализ. М.: Вильямс, 2016.

Глава 2

Техника программирования

В этой главе представлены некоторые средства языка C++, полезные в олимпиадном программировании, а также приведены примеры использования рекурсии и поразрядных операций.

В разделе 2.1 рассматриваются избранные вопросы C++, включая ввод и вывод, работу с числами и способы сокращения кода.

Раздел 2.2 посвящен рекурсивным алгоритмам. Сначала мы рассмотрим элегантный способ порождения всех подмножеств и перестановок множества с применением рекурсии. А затем с помощью перебора с возвратом подсчитаем, сколькими способами можно расположить n ферзей на шахматной доске $n \times n$, так чтобы они не били друг друга.

В разделе 2.3 обсуждаются основы поразрядных операций и их применение для представления подмножеств множества.

2.1. Языковые средства

Типичная олимпиадная программа на C++ устроена по такому образцу:

```
#include <bits/stdc++.h>

using namespace std;

int main() {
    // здесь находится решение
}
```

Строка `#include` в начале программы – специфика компилятора `g++`, она служит для включения всей стандартной библиотеки. Таким образом, не нужно по отдельности включать такие библиотеки, как `iostream`, `vector` и `algorithm`; все они автоматически становятся доступны.

В строке `using` объявляется, что все классы и функции из стандартной библиотеки можно использовать без указания пространства имен. Не будь `using`, нужно было бы писать, к примеру, `std::cout`, а так достаточно просто `cout`.

Для компиляции программы используется команда вида:

```
g++ -std=c++11 -O2 -Wall test.cpp -o test
```

Она порождает двоичный файл `test` из исходного файла `test.cpp`. Флаги означают, что компилятор должен следовать стандарту C++11 (`-std=c++11`), оптимизировать код (`-O2`) и выводить предупреждения о возможных проблемах (`-Wall`).

2.1.1. *Ввод и вывод*

В большинстве олимпиадных задач для ввода и вывода используются стандартные потоки. В C++ стандартный поток ввода называется `cin`, а стандартный поток вывода – `cout`. Можно также использовать C-функции, например `scanf` и `printf`.

Входными данными для программы обычно являются числа и строки, разделенные пробелами и знаками новой строки. Из потока `cin` они читаются следующим образом:

```
int a, b;
string x;
cin >> a >> b >> x;
```

Такой код работает в предположении, что элементы потока разделены хотя бы одним пробелом или знаком новой строки. Например, приведенный выше код может прочитать как входные данные:

```
123 456 monkey
```

так и входные данные:

```
123 456
monkey
```

Поток `cout` используется для вывода:

```
int a = 123, b = 456;
string x = "monkey";
cout << a << " " << b << " " << x << "\n";
```

Ввод и вывод часто оказываются узкими местами программы. Чтобы повысить эффективность ввода-вывода, можно поместить в начало программы такие строки:

```
ios::sync_with_stdio(0);
cin.tie(0);
```

Отметим, что знак новой строки `"\n"` работает быстрее, чем `endl`, потому что `endl` всегда сопровождается сбросом буфера.

C-функции `scanf` и `printf` – альтернатива стандартным потокам C++. Обычно они работают немного быстрее, но и использовать их сложнее. Вот как можно прочитать из стандартного ввода два целых числа:

```
int a, b;
scanf("%d %d", &a, &b);
```

А вот как их можно напечатать:

```
int a = 123, b = 456;
printf("%d %d\n", a, b);
```

Иногда программа должна прочитать целую входную строку, быть может, содержащую пробелы. Это можно сделать с помощью функции `getline`:

```
string s;
getline(cin, s);
```

Если объем данных заранее неизвестен, то полезен такой цикл:

```
while (cin >> x) {
    // код
}
```

Этот цикл читает из стандартного ввода элементы один за другим, пока входные данные не закончатся.

В некоторых олимпиадных системах для ввода и вывода используются файлы. В таком случае есть простое решение: писать код так, будто работаешь со стандартными потоками, но в начало программы добавить такие строки:

```
freopen("input.txt", "r", stdin);
freopen("output.txt", "w", stdout);
```

После этого программа будет читать данные из файла «input.txt», а записывать в файл «output.txt».

2.1.2. *Работа с числами*

Целые числа. Из целых типов в олимпиадном программировании чаще всего используется `int` – 32-разрядный тип[1], принимающий значения из диапазона $-2^{31}...2^{31}-1$ (приблизительно $-2 \cdot 10^9...2 \cdot 10^9$). Если типа `int` недостаточно, то можно взять 64-разрядный тип `long long`. Диапазон его значений $-2^{63}...2^{63}-1$ (приблизительно $-9 \cdot 10^{18}...9 \cdot 10^{18}$).

Ниже определена переменная типа `long long`:

[1] На самом деле стандарт C++ не определяет точные размеры числовых типов, а границы диапазонов зависят от платформы и компилятора. В этом разделе указаны размеры, с которыми вы, вероятнее всего, встретитесь в современных системах.

```
long long x = 123456789123456789LL;
```

Суффикс LL означает, что число имеет тип long long.

Типичная ошибка при использовании типа long long возникает, когда где-то в программе встречается еще и тип int. Например, в следующем коде есть тонкая ошибка:

```
int a = 123456789;
long long b = a*a;
cout << b << "\n"; // -1757895751
```

Хотя переменная b имеет тип long long, оба сомножителя в выражении a*a имеют тип int, поэтому тип результата тоже int. Из-за этого значение b оказывается неверным. Проблему можно решить, изменив тип a на long long или изменив само выражение на (long long)a*a.

Обычно олимпиадные задачи формулируются так, что типа long long достаточно. Но все же полезно знать, что компилятор g++ поддерживает также 128-разрядный тип __int128_t с диапазоном значений $-2^{127}...2^{127}-1$ (приблизительно $-10^{38}...10^{38}$). Однако этот тип доступен не во всех олимпиадных системах.

Арифметика по модулю. Иногда ответом является очень большое число, но достаточно вывести результат «по модулю m», т. е. остаток от деления на m (например, «7 по модулю 10^9»). Идея в том, что даже когда истинный ответ очень велик, типов int и long long все равно достаточно.

Остаток x от деления на m обозначается $x \bmod m$. Например, $17 \bmod 5 = 2$, поскольку $17 = 3 \cdot 5 + 2$. Важные свойства остатков выражаются следующими формулами:

$$(a + b) \bmod m = (a \bmod m + b \bmod m) \bmod m;$$
$$(a - b) \bmod m = (a \bmod m - b \bmod m) \bmod m;$$
$$(a \cdot b) \bmod m = (a \bmod m \cdot b \bmod m) \bmod m.$$

Это значит, что можно брать остаток после каждой операции, поэтому числа никогда не станут слишком большими.

Например, следующий код вычисляет $n!$ (n факториал) по модулю m:

```
long long x = 1;
for (int i = 1; i <= n; i++) {
    x = (x*i)%m;
}
cout << x << "\n";
```

Обычно мы хотим, чтобы остаток находился в диапазоне $0...m-1$. Но в C++ и в других языках остаток от деления отрицательного числа равен

нулю или отрицателен. Чтобы не возникали отрицательные остатки, можно поступить следующим образом: сначала вычислить остаток, а если он отрицателен, прибавить m:

```
x = x%m;
if (x < 0) x += m;
```

Но это стоит делать, только если в программе встречается операция вычитания, так что остаток может стать отрицательным.

Числа с плавающей точкой. В большинстве олимпиадных задач целых чисел достаточно, но иногда возникает потребность в числах с плавающей точкой. В C++ наиболее полезен 64-разрядный тип `double`, а в компиляторе `g++` имеется расширение – 80-разрядный тип `long double`. Чаще всего типа `double` достаточно, но `long double` дает более высокую точность.

Требуемая точность ответа обычно указывается в формулировке задачи. Проще всего для вывода ответа использовать функцию `printf` и указать количество десятичных цифр в форматной строке. Например, следующий код печатает значение x с 9 цифрами после запятой:

```
printf("%.9f\n", x);
```

С использованием чисел с плавающей точкой связана одна сложность: некоторые числа невозможно точно представить в таком формате, поэтому неизбежны ошибки округления. Например, в следующем коде получается значение x, немного меньшее 1, тогда как правильное значение равно в точности 1.

```
double x = 0.3*3+0.1;
printf("%.20f\n", x); // 0.99999999999999988898
```

Числа с плавающей точкой рискованно сравнивать с помощью оператора ==, потому что иногда равные значения оказываются различны из-за ошибок округления. Более правильно считать, что два числа равны, если разность между ними меньше ε, где ε мало. Например, в следующем коде $\varepsilon = 10^{-9}$:

```
if (abs(a-b) < 1e-9) {
    // a и b равны
}
```

Отметим, что хотя числа с плавающей точкой, вообще говоря, не точны, не слишком большие целые числа представляются точно. Так, тип `double` позволяет точно представить все целые числа, по абсолютной величине не большие 2^{53}.

2.1.3. *Сокращение кода*

Имена типов. Ключевое слово typedef позволяет сопоставить типу данных короткое имя. Например, имя long long слишком длинное, поэтому можно определить для него короткий псевдоним ll:

```
typedef long long ll;
```

После этого код

```
long long a = 123456789;
long long b = 987654321;
cout << a*b << "\n";
```

можно немного сократить:

```
ll a = 123456789;
ll b = 987654321;
cout << a*b << "\n";
```

Ключевое слово typedef применимо и к более сложным типам. Например, ниже мы сопоставляем вектору целых чисел имя vi, а паре двух целых чисел – тип pi.

```
typedef vector<int> vi;
typedef pair<int,int> pi;
```

Макросы. Еще один способ сократить код – макросы. Макрос говорит, что определенные строки кода следует подменить до компиляции. В C++ макросы определяются с помощью ключевого слова #define.

Например, мы можем определить следующие макросы:

```
#define F first
#define S second
#define PB push_back
#define MP make_pair
```

После чего код

```
v.push_back(make_pair(y1,x1));
v.push_back(make_pair(y2,x2));
int d = v[i].first+v[i].second;
```

можно сократить до:

```
v.PB(MP(y1,x1));
v.PB(MP(y2,x2));
int d = v[i].F+v[i].S;
```

У макроса могут быть параметры, что позволяет сокращать циклы и другие структуры. Например, можно определить такой макрос:

```
#define REP(i,a,b) for (int i = a; i <= b; i++)
```

После этого код

```
for (int i = 1; i <= n; i++) {
    search(i);
}
```

можно сократить следующим образом:

```
REP(i,1,n) {
    search(i);
}
```

2.2. Рекурсивные алгоритмы

Благодаря *рекурсии* часто удается элегантно реализовать алгоритм. В этом разделе мы обсудим рекурсивные алгоритмы, которые систематически перебирают потенциальные решения задачи. Сначала рассмотрим порождение подмножеств и перестановок множества, а затем обсудим более общую технику перебора с возвратом.

2.2.1. *Порождение подмножеств*

В качестве первого применения рекурсии рассмотрим порождение всех подмножеств множества из n элементов. Например, подмножествами {1, 2, 3} являются 0, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3} и {1, 2, 3}. Следующая рекурсивная функция `search` генерирует подмножества. Функция манипулирует вектором

```
vector<int> subset;
```

который содержит элементы подмножества. Чтобы начать поиск, следует вызвать функцию с параметром 1.

```
void search(int k) {
    if (k == n+1) {
        // обработать подмножество
    } else {
        // включить k в подмножество
        subset.push_back(k);
        search(k+1);
        subset.pop_back();
        // не включать k в подмножество
```

```
        search(k+1);
    }
}
```

При вызове с параметром k функция `search` решает, следует ли включать элемент k в множество или нет, но в обоих случаях вызывает себя с параметром $k + 1$. Если оказывается, что $k = n + 1$, то функция понимает, что все элементы обработаны и подмножество сгенерировано.

На рис. 2.1 показано порождение подмножеств при $n = 3$. При каждом вызове функции выбирается либо верхняя ветвь (k включается в подмножество), либо нижняя (k не включается в подмножество).

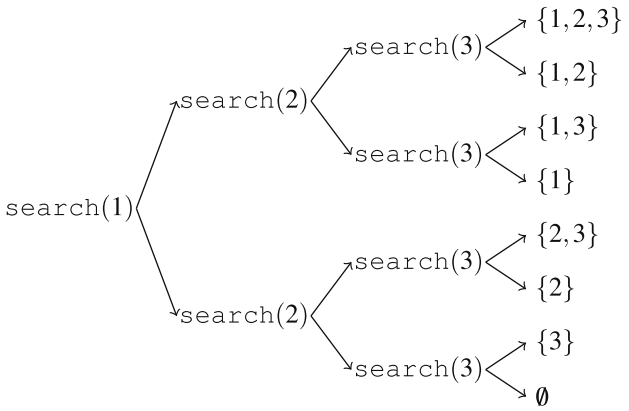

Рис. 2.1. Дерево рекурсии при порождении подмножеств множества {1, 2, 3}

2.2.2. *Порождение перестановок*

Теперь рассмотрим задачу о порождении всех перестановок множества из n элементов. Например, перестановками {1, 2, 3} будут (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2) и (3, 2, 1). И снова для решения можно применить рекурсию. Следующая функция манипулирует вектором

```
vector<int> permutation;
```

который содержит перестановку, и массивом

```
bool chosen[n+1];
```

который для каждого элемента показывает, включен он уже в перестановку или нет. Поиск начинается, когда эта функция вызывается без параметров.

```
void search() {
    if (permutation.size() == n) {
        // обработать перестановку
```

```
    } else {
        for (int i = 1; i <= n; i++) {
            if (chosen[i]) continue;
            chosen[i] = true;
            permutation.push_back(i);
            search();
            chosen[i] = false;
            permutation.pop_back();
        }
    }
}
```

При каждом вызове функция добавляет новый элемент в вектор `permutation` и запоминает в массиве `chosen`, что он был добавлен. Если размер `permutation` оказывается равен размеру множества, то генерируется перестановка.

Отметим, что в стандартной библиотеке C++ имеется функция `next_permutation`, которую также можно использовать для порождения перестановок. Этой функции передается перестановка, а она порождает следующую за ней в лексикографическом порядке. Следующая программа перебирает все перестановки множества {1, 2, ..., *n*}:

```
for (int i = 1; i <= n; i++) {
    permutation.push_back(i);
}
do {
    // обработать перестановку
} while (next_permutation(permutation.begin(), permutation.end()));
```

2.2.3. *Перебор с возвратом*

Алгоритм *перебора с возвратом* начинает работу с пустого решения и шаг за шагом расширяет его. Рекурсивно перебираются все возможные способы построения решения.

В качестве примера рассмотрим задачу о вычислении количества способов расставить *n* ферзей на доске *n*×*n*, так чтобы никакие два не били друг друга. Так, на рис. 2.2 показано два возможных решения для *n* = 4.

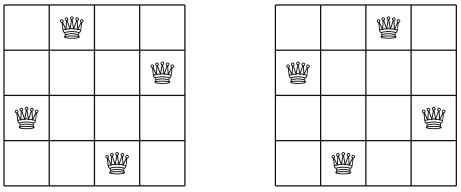

Рис. 2.2. Возможные способы расстановки 4 ферзей на доске 4×4

Задачу можно решить методом перебора с возвратом, рассматривая горизонтали одну за другой. Точнее, на каждой горизонтали можно разместить ровно одного ферзя, так чтобы он не оказался под боем ранее поставленных ферзей. Очередное решение будет найдено, когда на доску поставлены все n ферзей.

Например, на рис. 2.3 показано несколько частичных решений, сгенерированных алгоритмом перебора с возвратом при $n = 4$. Первые три расстановки на нижнем уровне недопустимы, поскольку ферзи бьют друг друга. Но четвертая расстановка допустима, и ее можно дополнить до решения, поставив на доску еще двух ферзей. Сделать это можно единственным способом.

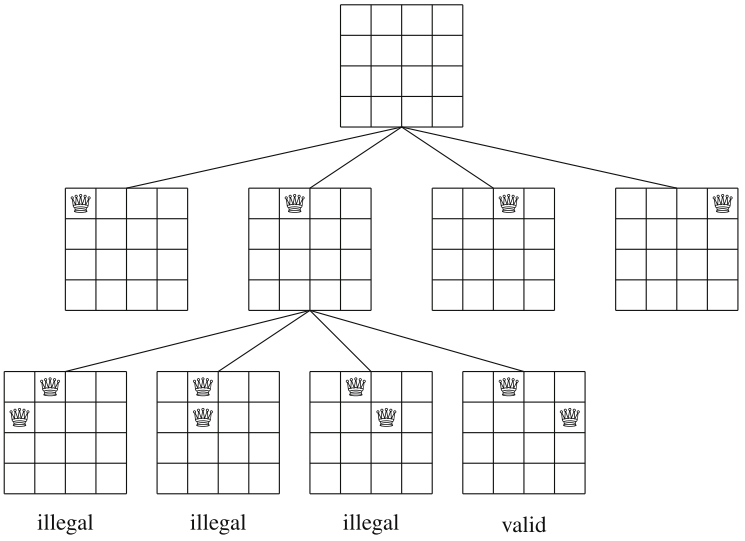

Рис. 2.3. Частичные решения задачи о расстановке ферзей, полученные методом перебора с возвратом

Алгоритм можно реализовать следующим образом:

```
void search(int y) {
   if (y == n) {
      count++;
      return;
   }
   for (int x = 0; x < n; x++) {
      if (col[x] || diag1[x+y] || diag2[x-y+n-1]) continue;
      col[x] = diag1[x+y] = diag2[x-y+n-1] = 1;
      search(y+1);
      col[x] = diag1[x+y] = diag2[x-y+n-1] = 0;
   }
}
```

Поиск начинается вызовом `search(0)`. Размер доски равен *n*, функция подсчитывает количество решений в переменной `count`. В коде предполагается, что горизонтали и вертикали доски пронумерованы от 0 до *n* − 1. Будучи вызвана с параметром *y*, функция `search` помещает ферзя на горизонталь *y* и вызывает себя с параметром *y* + 1. Когда *y* = *n*, решение найдено, поэтому значение `count` увеличивается на 1.

В массиве `col` запоминаются вертикали, уже занятые ферзями, а в массивах `diag1` и `diag2` запоминаются диагонали. Запрещается ставить ферзя на вертикаль или диагональ, в которой уже находится другой ферзь. На рис. 2.4 показана нумерация вертикалей и диагоналей для доски 4×4.

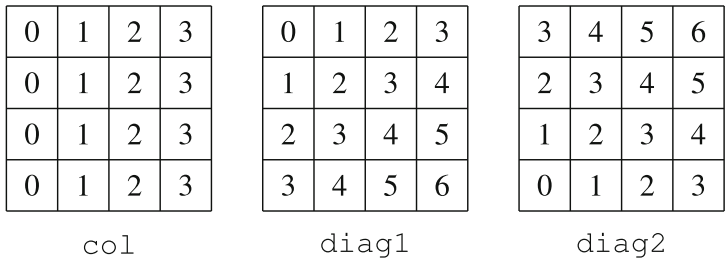

Рис. 2.4. Нумерация массивов при подсчете комбинаций на доске 4×4

Показанный выше алгоритм перебора с возвратом говорит, что существует 92 способа расставить 8 ферзей на доске 8×8. С ростом *n* поиск быстро замедляется, поскольку число решений возрастает экспоненциально. Так, на современном компьютере требуется около минуты, чтобы вычислить количество расстановок 16 ферзей на доске 16×16 – 14 772 512.

На самом деле до сих пор неизвестен эффективный способ подсчета числа расстановок ферзей для больших *n*. В настоящее время наибольшее *n*, для которого результат известен, равно 27: в этом случае существует 234 907 967 154 122 528 расстановок. Это было установлено в 2016 году группой исследователей, использовавших для решения кластер компьютеров [29].

2.3. Поразрядные операции

В программировании *n*-разрядное целое число хранится в виде двоичного числа, содержащего *n* бит. Например, тип `int` в C++ 32-разрядный, т. е. любое число типа `int` содержит 32 бита. Так, двоичное представление числа 43 типа `int` имеет вид

$$00000000000000000000000000101011.$$

Биты в этом представлении нумеруются справа налево. Преобразование двоичного представления $b_k \ldots b_2 b_1 b_0$ в десятичное число производится по формуле

$$b_k 2^k + \ldots + b_2 2^2 + b_1 2^1 + b_0 2^0.$$

Например:

$$1 \cdot 2^5 + 1 \cdot 2^3 + 1 \cdot 2^1 + 1 \cdot 2^0 = 43.$$

Двоичное представление числа может быть *со знаком* и *без знака*. Обычно используется представление со знаком, позволяющее представить положительные и отрицательные числа. n-разрядная переменная со знаком может содержать любое целое число в диапазоне от -2^{n-1} до $2^{n-1} - 1$. Например, тип `int` в C++ знаковый, поэтому переменная типа `int` может содержать любое целое число от -2^{31} до $2^{31} - 1$.

Первый разряд в представлении со знаком содержит знак числа (0 для неотрицательных чисел, 1 – для отрицательных), а остальные $n - 1$ разрядов – абсолютную величину числа. Используется *дополнительный код*, т. е. для получения противоположного числа нужно сначала инвертировать все его биты, а затем прибавить к результату единицу. Например, двоичное представление числа –43 типа `int` имеет вид

11111111111111111111111111010101.

Представление без знака позволяет представить только неотрицательные числа, но верхняя граница диапазона больше. n-разрядная переменная без знака может содержать любое целое число от 0 до $2^n - 1$. Например, в C++ переменная типа `unsigned int` может содержать любое целое число от 0 до $2^{32} - 1$.

Между обоими представлениям существует связь: число со знаком $-x$ равно числу без знака $2^n - x$. К примеру, следующая программа показывает, что число со знаком $x = -43$ равно числу без знака $y = 2^{32} - 43$:

```
int x = -43;
unsigned int y = x;
cout << x << "\n"; // -43
cout << y << "\n"; // 4294967253
```

Если число больше верхней границы допустимого диапазона, то возникает переполнение. В представлении со знаком число, следующее за $2^{n-1} - 1$, равно -2^{n-1}, а в представлении без знака за $2^n - 1$ следует 0. Рассмотрим следующий код:

```
int x = 2147483647
cout << x << "\n"; // 2147483647
x++;
cout << x << "\n"; // -2147483648
```

Первоначально x принимает значение $2^{31} - 1$. Это наибольшее значение, которое можно сохранить в переменной типа `int`, поэтому следующее за $2^{31} - 1$ значение равно -2^{31}.

2.3.1. *Поразрядные операции*

Операция И. Результатом операции *И* x & y является число, двоичное представление которого содержит единицы в тех позициях, на которых в представлениях x и y находятся единицы. Например, 22 & 26 = 18, поскольку

```
   10110 (22)
&  11010 (26)
=  10010 (18)
```

С помощью операции И можно проверить, является ли число x четным, т. к. x & 1 = 0, если x четно, и x & 1 = 1, если x нечетно. Вообще, x нацело делится на 2^k, если x & $(2^k - 1) = 0$.

Операция ИЛИ. Результатом операции *ИЛИ* x | y является число, двоичное представление которого содержит единицы в тех позициях, на которых хотя бы в одном из представлений x и y находятся единицы. Например, 22 | 26 = 30, поскольку

```
   10110 (22)
|  11010 (26)
=  11110 (30)
```

Операция ИСКЛЮЧАЮЩЕЕ ИЛИ. Результатом операции *ИСКЛЮЧАЮЩЕЕ ИЛИ* x ^ y является число, двоичное представление которого содержит единицы в тех позициях, на которых ровно в одном из представлений x и y находятся единицы. Например, 22 ^ 26 = 12, поскольку

```
   10110 (22)
^  11010 (26)
=  01100 (12)
```

Операция НЕ. Результатом операции *НЕ* ~x является число, в двоичном представлении которого все биты x инвертированы. Справедлива формула ~x = $-x$ – 1, например ~29 = −30. Результат операции *НЕ* на битовом уровне зависит от длины двоичного представления, поскольку инвертируются все биты. Например, в случае 32-разрядных чисел типа `int` имеем:

```
 x =  29  00000000000000000000000000011101
~x = -30  11111111111111111111111111100010
```

Поразрядный сдвиг. Операция поразрядного сдвига влево x << k дописывает в конец числа k нулей, а операция поразрядного сдвига вправо

x >> k удаляет k последних бит. Например, 14 << 2 = 56, поскольку двоичные представления 14 и 56 равны соответственно 1110 и 111000. Аналогично 49 >> 3 = 6, потому что 49 и 6 в двоичном виде равны соответственно 110001 и 110. Отметим, что операция x << k соответствует умножению x на 2^k, а x >> k – делению x на 2^k с последующим округлением с недостатком до целого.

Битовые маски. *Битовой маской* называется число вида 1 << k, содержащее в позиции k единицу, а во всех остальных позициях – нули. Такую маску можно использовать для выделения одиночных битов. В частности, k-й бит числа равен единице тогда и только тогда, когда x & (1 << k) не равно нулю. В следующем фрагменте печатается двоичное представление числа x типа `int`:

```
for (int k = 31; k >= 0; k--) {
   if (x&(1<<k)) cout << "1";
   else cout << "0";
}
```

Аналогичным образом можно модифицировать отдельные биты числа. Выражение x | (1 << k) устанавливает k-й бит x в единицу, выражение x & ~(1 << k) сбрасывает k-й бит x в нуль, а выражение x ^ (1 << k) инвертирует k-й бит x. Далее выражение x & (x − 1) сбрасывает последний единичный бит x в нуль, а выражение x & −x сбрасывает в нуль все единичные биты, кроме последнего. Выражение x | (x − 1) инвертирует все биты после последнего единичного. Наконец, положительное число x является степенью двойки тогда и только тогда, когда x & (x − 1) = 0.

При работе с битовыми масками нужно помнить, что `1<<k` всегда имеет тип `int`. Самый простой способ создать битовую маску типа `long long` – написать `1LL<<k`.

Дополнительные функции. Компилятор g++ предлагает также следующие функции для подсчета битов:

- `__builtin_clz`(x): количество нулей в начале двоичного представления;
- `__builtin_ctz`(x): количество нулей в конце двоичного представления;
- `__builtin_popcount`(x): количество единиц в двоичном представлении;
- `__builtin_parity`(x): четность количества единиц в двоичном представлении.

Эти функции используются следующим образом:

```
int x = 5328; // 00000000000000000001010011010000
cout << __builtin_clz(x) << "\n"; // 19
```

```
cout << __builtin_ctz(x) << "\n"; // 4
cout << __builtin_popcount(x) << "\n"; // 5
cout << __builtin_parity(x) << "\n"; // 1
```

Отметим, что данные функции поддерживают только числа типа `int`, но есть также их варианты для типа `long long`, их имена заканчиваются суффиксом `ll`.

2.3.2. *Представление множеств*

Любое подмножество множества {0, 1, 2, ..., n − 1} можно представить n-разрядным целым числом, в котором единичные биты соответствуют элементам, принадлежащим подмножеству. Такой способ представления эффективен, поскольку для каждого элемента требуется всего один бит памяти, а теоретико-множественные операции можно реализовать с помощью поразрядных операций.

Например, поскольку тип `int` 32-разрядный, числом типа `int` можно представить любое подмножество множества {0, 1, 2, ..., 31}. Двоичное представление множества {1, 3, 4, 8} имеет вид

00000000000000000000000100011010,

что соответствует числу $2^8 + 2^4 + 2^3 + 2^1 = 282$.

В показанном ниже коде объявлена переменная x типа `int`, которая может содержать подмножество множества {0, 1, 2, ..., 31}. Затем в это подмножество добавляются элементы 1, 3, 4, 8 и печатается его размер.

```
int x = 0;
x |= (1<<1);
x |= (1<<3);
x |= (1<<4);
x |= (1<<8);
cout << __builtin_popcount(x) << "\n"; // 4
```

Далее печатаются все элементы, принадлежащие множеству:

```
for (int i = 0; i < 32; i++) {
   if (x&(1<<i)) cout << i << " ";
}
// выводится: 1 3 4 8
```

Операции над множествами. В табл. 2.1 показано, как реализовать теоретико-множественные операции с помощью поразрядных. Так, следующий код сначала конструирует множества x = {1, 3, 4, 8} и y = {3, 6, 8, 9}, а затем множество $z = x \cup y$ = {1, 3, 4, 6, 8, 9}:

```
int x = (1<<1)|(1<<3)|(1<<4)|(1<<8);
int y = (1<<3)|(1<<6)|(1<<8)|(1<<9);
int z = x|y;
cout << __builtin_popcount(z) << "\n"; // 6
```

Таблица 2.1. Реализация теоретико-множественных операций с помощью поразрядных

Операция	Теоретико-множественная запись	Поразрядная запись
Пересечение	$a \cap b$	a & b
Объединение	$a \cup b$	$a \mid b$
Дополнение	$\bar{a}$	$\sim a$
Разность	$a \setminus b$	a & ($\sim b$)

Следующий код перебирает подмножества множества $\{0, 1, \ldots, n - 1\}$:

```
for (int b = 0; b < (1<<n); b++) {
   // обработать подмножество b
}
```

А этот код перебирает только подмножества, содержащие ровно k элементов:

```
for (int b = 0; b < (1<<n); b++) {
   if (__builtin_popcount(b) == k) {
      // обработать подмножество b
}
}
```

Наконец, следующий код перебирает все подмножества множества x:

```
int b = 0;
do {
   // обработать подмножество b
} while (b=(b-x)&x);
```

Битовые множества в C++. В стандартной библиотеке C++ имеется структура `bitset`, соответствующая массиву, все элементы которого равны 0 или 1. Например, следующий код создает битовое множество из 10 элементов:

```
bitset<10> s;
s[1] = 1;
s[3] = 1;
s[4] = 1;
s[7] = 1;
```

```
cout << s[4] << "\n"; // 1
cout << s[5] << "\n"; // 0
```

Функция count возвращает количество единичных битов в битовом множестве:

```
cout << s.count() << "\n"; // 4
```

К битовым множествам также применимы поразрядные операции:

```
bitset<10> a, b;
// ...
bitset<10> c = a&b;
bitset<10> d = a|b;
bitset<10> e
```