МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА»

Гайдель А. В.

Основы информатики

Учебное пособие

Самара 2016

УДК 004
ББК 73

Рецензенты: д. т. н., доцент Храмов Александр Григорьевич,
д. т. н., доцент Куприянов Александр Викторович

Гайдель А. В. Основы информатики: учеб. пособие / А. В. Гайдель. – Самара: Издательство СГАУ, 2016. – 218 с.

Представленные материалы включают теоретические и практические основы, необходимые для изучения информатики и программирования в технических вузах. Излагаются основы систем счисления, теории сложности алгоритмов, а также примеры простейших алгоритмов и программ. Большинство примеров программ приводится на языке C++.

Учебное пособие предназначено для студентов специальностей «Прикладная математика и информатика» и «Прикладная математика и физика» в качестве дополнительных материалов для подготовки по курсам «Основы информатики» и «Информатика».

ВВЕДЕНИЕ

Как можно догадаться из названия, *информатика* – это наука об информации: о способах её хранения, обработки и передачи. Также информатика изучает методы автоматического сбора информации, способы её оценки и возможность использования информации для принятия решений. Как правило, речь идёт о применении для указанных процессов компьютерных технологий, а также других, в том числе абстрактных, вычислительных машин и сетей.

Информация, как наиболее общий объект исследования информатики, сама по себе является довольно сложным понятием, едва допускающим формализацию и изучение. Существует множество разных по степени научности и официальности определений этого понятия, приводить которые не имеет смысла, поскольку в любом случае понимание информации присутствует у людей на интуитивном уровне. Для определённости терминологического аппарата можно сказать, что *информация* – это сведения о действительности, которые мыслящий субъект получает извне. Однако, как видно, это в большей степени философское определение, слабо помогающее на практике.

Информацию следует отличать от *данных*, которые представляют собой информацию, записанную на материальный носитель. Данные имеют конкретное представление и форму, доступную для интерпретации многими субъектами. Носителем может выступать, в том числе, электромагнитное поле, звуковые волны и прочее.

Своими основами информатика упирается в различные математические теории: формальную логику, дискретную математику, теорию чисел, теорию вероятностей и даже функциональный анализ. Тем не менее, информатика – это прикладная наука. Предметами её изучения в числе прочих являются языки программирования, базы данных, телекоммуникации, а также машинное обучение и искусственный интеллект.

Информатика – сравнительно молодая наука. Конечно, можно относить к информатике многие исследования и изобретения для устного счёта, созданные ещё до нашей эры, но полноценный вклад в жизнь общества эта наука внесла лишь с появлением первых вычислительных машин в XX веке нашей эры. Во второй половине XX века в русском языке появилось и само слово «информатика», первоначально означавшее теорию научно-информационной деятельности. Современное фундаментальное значение это слово получило лишь к 90-м годам XX века.

Если говорить о терминологии, то нельзя не упомянуть некоторую нестыковку с англоязычными понятиями. Сегодня в России словом «информатика» называется то же, что в английском языке носит название «Computer Science» (дословно, «компьютерная наука»). Слово «Informatics» в английском языке тоже существует, но используется несколько реже и включает в большей степени математическую статистику, теорию информации и смежные дисциплины.

Вклад информатики в развитие человеческой цивилизации сложно переоценить. Именно благодаря успехам этой науки современное общество существует в единой информационной среде, где каждый человек имеет свободный доступ к знаниям и может общаться с другими людьми, не выходя из дома. Современные вычислительные машины получили способность обрабатывать большие объёмы данных в короткие сроки, что дало сильнейший толчок к развитию многим другим наукам, в том числе медицине, биологии, химии, физике, гидрометеорологии, экологии. Большая часть развлечений и работы в современном обществе связаны с информационными технологиями и стали возможны благодаря достижениям в информатике.

Сама по себе информатика представляет собой фундаментальную науку, изучающую очень широкую область знаний. В силу этого можно выделить множество относительно обособленных разделов информатики, отличающихся более конкретными объектами исследования. Прежде всего, информатику

можно условно разделить по уровню абстракции на теоретическую и прикладную. Теоретическая информатика изучает достаточно абстрактные объекты, такие как алгоритмы и структуры данных. Прикладная информатика ориентирована на информационные технологии, программирование и базы данных.

К теоретическим разделам информатики можно отнести следующие.

1. *Теория алгоритмов* изучает формальные алгоритмы, их свойства и модели представления. *Теория вычислимости* – это раздел теории алгоритмов, изучающий проблемы построения алгоритмов, вычисляющих определённые функции.
2. *Теория формальных грамматик* изучает способы построения языков, описывающих алгоритмы и вычислительные процессы.
3. *Теория информации* (англ. *Information Theory*) изучает методы сжатия, кодирования и обеспечения целостности информации при её хранении и передаче.
4. *Теория распознавания образов* изучает методы классификации объектов, заданных некоторыми наборами признаков.
5. *Интеллектуальный анализ данных* (англ. *Data Mining*) изучает способы извлечения полезной информации из данных и алгоритмы принятия решений на основе этой информации.

К прикладным разделам информатики можно отнести следующие.

1. *Биоинформатика* исследует модели биологических систем, прежде всего геномов животных и человека.
2. *Геоинформатика* занимается проектированием и созданием географических информационных систем, систем обработки данных дистанционного зондирования Земли.
3. *Криптография* – это наука о защите конфиденциальности данных.
4. *Программирование* как наука о разработке, тестировании, отладке и поддержке компьютерных программ.

5. *Теория распределённых вычислений* или *теория параллельных вычислений* изучает методы организации вычислительных процессов одновременно на нескольких вычислительных устройствах или узлах.
6. *Компьютерная лингвистика* моделирует и исследует естественные языки.
7. *Машинное обучение* (англ. *Machine Learning*) строит информационные системы, допускающие обучение на некоторых данных с целью повышения эффективности их работы.
8. *Машинное зрение* (англ. *Machine Vision*) проектирует устройства, воспринимающие визуальные данные в процессе своей работы и изменяющие поведение в соответствии с ними.
9. *Компьютерная графика* синтезирует и обрабатывает визуальную информацию в пригодном для восприятия человеком формате.
10. *Теория баз данных* включает теоретические основы проектирования и оптимизации баз данных и систем управления базами данных.

Понятно, что представленное разделение весьма условно. Во-первых, деление на теоретические и прикладные разделы и без того прикладной науки во многом субъективно. В каждом разделе можно в свою очередь выделить теоретические основы и приложения для решения конкретных задач. Во-вторых, многие разделы тесно связаны с другими, так что во многих случаях границы между ними размыты. В-третьих, представленный список далеко не полный, поскольку у информатики существует ещё огромное множество приложений, каждое из которых также имеет под собой некоторые теоретические основания.

В данном пособии речь не пойдёт ни об одном из названных разделов информатики. Главной целью этой книги является обучение основным знаниям и навыкам, необходимым для последующего более глубокого усвоения специализированных курсов. В пособие вошли материалы, описывающие способы хранения основных типов и структур данных в памяти компьютера, а также основы построения, анализа и программной реализации алгоритмов.

Большая часть приведённой информации сопровождается поясняющими иллюстрациями и примерами. В конце каждого раздела имеются упражнения для самостоятельного решения. Некоторые из них не могут быть решены с использованием только информации, приведённой в этой книге, и даны в качестве пищи для размышлений на будущее.

Целью этой книги не является обучение особенностям конкретного языка программирования. Для определённости большинство примеров исходного кода программ приведено на языках C и C++, но предполагается, что синтаксис и структуру этих языков читатель изучает на других курсах или осваивает самостоятельно. Многие приёмы, характерные для программирования в целом, тем не менее, детально разбираются.

1 Целочисленная арифметика

1.1 Понятие целых чисел

Чтобы в полной мере осознать, как вычислительные машины оперируют с целыми числами, требуется чётко представлять себе, что вообще собой представляют целые числа. Конечно, многие люди интуитивно понимают, что такое целые числа и чем они отличаются от дробных чисел, однако чтобы быть уверенным в одинаковом понимании этого термина автором и всеми читателями, требуется ввести более формальное математическое определение.

Для определения целых чисел необходимо сначала определить менее абстрактные натуральные числа. Это известные всем с начальной школы числа, издавна используемые людьми для счёта предметов. Например, числа 1, 2, 3 и так далее являются натуральными.

Множество натуральных чисел обозначается $\mathbf{N}$. Их природа также интуитивно понятна, но для формального определения этого множества используются *аксиомы Пеано*. Итак, $\mathbf{N}$ называется множеством *натуральных чисел*, если для его элементов определена операция $s(n):\mathbf{N}\to\mathbf{N}$, принимающая значение следующего за n натурального числа, и справедливы следующие аксиомы.

1. $1\in\mathbf{N}$ (единица является натуральным числом).
2. $\forall n\in\mathbf{N}: s(n)\in\mathbf{N}$ (за любым натуральным числом следует натуральное число).
3. $\forall n\in\mathbf{N}: s(n)\neq 1$ (единица не следует ни за каким натуральным числом).
4. $\forall n,m\in\mathbf{N}:\big(s(n)=s(m)\big)\Rightarrow\big(n=m\big)$ (каждое натуральное число следует только за одним натуральным числом).
5. $\forall P(x):\mathbf{N}\to\{0;1\}: P(1)\wedge\big(\forall n\in\mathbf{N}\mathbf{:}\,P(n)\Rightarrow P\big(s(n)\big)\big)\Rightarrow\big(\forall n\in\mathbf{N}\mathbf{:}\,P(n)\big)$

   (справедлив *принцип математической индукции*: если некоторое

утверждение $P(n)$ справедливо для единицы и из его справедливости для произвольного n следует его справедливость для следующего за n натурального числа, то это означает, что утверждение $P(n)$ имеет место вообще для всех натуральных чисел n).

Любое множество с операцией $s(n)$, удовлетворяющее аксиомам Пеано, является множеством натуральных чисел, не зависимо от того, как обозначаются числа в нём.

Принцип математической индукции может показаться запутанным для аксиомы, тем не менее, он не следует ни из каких других более простых аксиом. Он часто используется для доказательства утверждений во многих областях математики.

Пример 1

Задача. Доказать, что сумма кубов трёх последовательных натуральных чисел делится на 9.

Решение. Утверждение $P(n)$ заключается в том, что $n^3+(n+1)^3+(n+2)^3$ делится на 9, то есть $P(n)=\left\{n^3+(n+1)^3+(n+2)^3 \vdots 9\right\}$.

Для $n=1$ имеем $1^3+2^3+3^3=36$. Очевидно, 36 делится на 9, ведь $36=9\cdot 4$. Таким образом, $P(1)$ – истина.

Допустим, $P(n)$ справедливо для некоторого натурального n, то есть существует натуральное число m, такое что $n^3+(n+1)^3+(n+2)^3=9m$. Рассмотрим $P(n+1)$:

$$(n+1)^3+(n+2)^3+(n+3)^3=(n+1)^3+(n+2)^3+n^3+9n^2+27n+27=$$
$$=9m+9n^2+27n+27=9\left(m+n^2+3n+3\right).$$

Число $\left(m+n^2+3n+3\right)$ также натуральное, так что $(n+1)^3+(n+2)^3+(n+3)^3$ тоже делится на 9.

В итоге было доказано, что верно $P(1)$, и из $P(n)$ следует $P(n+1)$. Согласно принципу математической индукции, это означает, что $P(n)$ справедливо вообще для любых натуральных чисел, так что сумма кубов любых трёх последовательных натуральных чисел делится на 9.

Что и требовалось доказать.

Арифметические операции, как и отношения эквивалентности и порядка, на множестве натуральных чисел вводятся через операцию $s(n)$ взятия следующего натурального числа, фигурирующую в аксиомах Пеано. Например, *сложение* натуральных чисел определяется как

$$n+m=\underbrace{s\big(s\big(\ldots s\big(n\big)\ldots\big)\big)}_{m},$$

а *умножение* –

$$n\cdot m=\underbrace{n+n+\ldots+n}_{m}.$$

Натуральные числа *равны*, если следуют за одним и тем же натуральным числом. Единица не следует ни за каким натуральным числом, но понятно, что $1=1$. Натуральное число n *не превосходит* натуральное число m, если $n=m$, или если следующее за n натуральное число не превосходит m.

Вооружившись этими определениями, легко доказать следующие основные свойства операций с натуральными числами.

1. *Ассоциативность* сложения:

$$\forall n,m,k\in\mathbf{N}:\big(n+m\big)+k=n+\big(m+k\big).$$

2. *Коммутативность* сложения:

$$\forall n,m\in\mathbf{N}:n+m=m+n.$$

3. *Ассоциативность* умножения:

$$\forall n,m,k\in\mathbf{N}:\big(n\cdot m\big)\cdot k=n\cdot\big(m\cdot k\big).$$

4. Единица нейтральна по умножению:

$$\forall n\in\mathbf{N}:n\cdot 1=n.$$

5. *Коммутативность* умножения:

$\forall n,m \in \mathbf{N} : n \cdot m = m \cdot n$.

6. *Дистрибутивность* умножения относительно сложения:

$\forall n,m,k \in \mathbf{N} : (n+m) \cdot k = n \cdot k + m \cdot k$.

Все остальные свойства арифметических операций с натуральными числами могут быть получены из этих свойств. Операции вычитания и деления очевидным образом определяются через операции сложения и умножения. Также можно определить возведение в степень и многое другое.

Нельзя не отметить некоторые нестыковки с терминологией, связанной с определением натуральных чисел. В некоторой литературе 0 также считается натуральным числом. Аксиомы Пеано в этом случае можно переписать естественным образом, также у операции сложения, как и у операции умножения, появляется нейтральный элемент. Существуют разные соображения, какое из определений лучше, но для разрешения путаницы в последнее время вместо термина «натуральные числа» стали употреблять однозначный термин «целые положительные числа». Множество целых положительных чисел, очевидно, совпадает с множеством натуральных чисел и обозначается $\mathbf{Z}_+$. Для него также справедливы аксиомы Пеано и оно может быть определено через них. В терминологии, в которой $\mathbf{N} = \mathbf{Z}_+$, множество натуральных чисел с нулём называется *расширенным натуральным рядом* и обозначается $\mathbf{N}_0 = \mathbf{N} \cup \{0\}$.

Потребность в отрицательных числах очевидным образом следует из невозможности вычитать бóльшие натуральные числа из меньших натуральных чисел. Например, выражение $2-3+4=3$ не имеет смысла в натуральных числах, поскольку число $(2-3)$ не является натуральным. Для устранения этого недостатка было введено множество целых чисел, которое обозначается $\mathbf{Z}$.

Множество *целых чисел* определяется, как *замыкание* множества натуральных чисел относительно операции вычитания. То есть множество

целых чисел – это множество минимальной мощности, такое что оно включает в себя множество натуральных чисел, так что операции сложения и умножения натуральных чисел сохраняются, кроме того эти операции удовлетворяют основным свойствам, представленным в таблице **1**.

Таблица 1 – Свойства арифметических операций с целыми числами

$\forall n,m,k \in \mathbf{Z}$	**Сложение**	**Умножение**
Замкнутость	$n+m \in \mathbf{Z}$	$n \cdot m \in \mathbf{Z}$
Ассоциативность	$(n+m)+k = n+(m+k)$	$(n \cdot m) \cdot k = n \cdot (m \cdot k)$
Нейтральный элемент	$n+0 = n$	$n \cdot 1 = n$
Обратный элемент	$\exists(-n) \in \mathbf{Z} : n+(-n) = 0$	существует только у 1 и -1
Коммутативность	$n+m = m+n$	$n \cdot m = m \cdot n$
Дистрибутивность	$(n+m) \cdot k = n \cdot k + m \cdot k$	

Основным отличием свойств арифметических операций с целыми числами от свойств арифметических операций с натуральными числами является наличие обратного элемента по сложению, то есть просто противоположного числа. Для положительного целого числа n противоположным является соответствующее отрицательное число $(-n)$, а для отрицательного – соответствующее положительное.

Напоследок ещё раз отметим, что любое множество, отвечающее определению целых чисел, является множеством целых чисел, независимо от того, как обозначать его элементы. Иными словами, целые числа существуют сами по себе, не зависимо от способа их визуального представления.

1.2 Системы счисления

Как было отмечено выше, числа определяются через аксиомы, определяющие операции над ними, так что числа существуют сами по себе, не зависимо от представления. Это некие идеи, не имеющие формы по умолчанию. Однако чтобы использовать информацию о числах в общении друг с другом людям нужно некоторым образом их обозначать. Именно для этого и

используются различные системы счисления, речь о которых пойдёт в этом разделе.

Система счисления – это способ представления чисел в виде знаков. В этой главе речь идёт о целых числах, так что системы счисления тоже рассматриваются только для них. Кроме прочего, обычно система счисления позволяет записывать числа в виде последовательности символов, называемых *цифрами*.

Система счисления называется *позиционной*, если каждое целое число $x \in \mathbf{Z}$ представляется в ней в виде последовательности *цифр* $\{a_k\}_{k=0}^{\infty}$, так что

$$x = \sum_{k=0}^{\infty} a_k b_k , \qquad (1)$$

где $\forall k \in \mathbf{N}_0 : a_k \in \mathbf{Z} \cap [0; Q]$, причём начиная с некоторого k все цифры равны нулю (можно дописать любое количество незначащих лидирующих нулей к любому целому числу), то есть

$$\exists K \in \mathbf{N}_0 \forall k \in \mathbf{N}_0 : k > K \Rightarrow a_k = 0 .$$

Здесь под Q понимается наибольшая цифра в данной системе счисления. Количество цифр в подавляющем большинстве систем счисления конечно. При записи числа цифры указываются в порядке уменьшения разрядов, то есть сначала указывается цифра при самом старшем разряде, затем следующая за ней, и так далее. Последней цифрой указывается a_0.

Последовательность $\{b_k\}_{k=0}^{\infty}$ называется *последовательностью разрядов*. Она не зависит от самого числа x и полностью характеризует позиционную систему счисления. Количество цифр выбирается из тех соображений, чтобы каждое целое число x имело представление в данной системе счисления, причём по возможности только одно.

Позиционная система счисления называется q-ичной, если её разряды представляют собой степени числа q:

$b_k = q^k$.

Само число q называется *основанием системы счисления*. Можно доказать, что для однозначного представления каждого целого числа в q-ичной системе счисления требуется q различных цифр от 0 до $(q-1)$ включительно, то есть

$$\forall k \in \mathbf{N}_0 : a_k \in \mathbf{Z} \cap [0; q-1].$$

Традиционной и привычной всем со школы является *десятичная система счисления*, в которой $q = 10$. По умолчанию считается, что все числа записаны именно в ней. Например, число 123 в этой системе счисления записывается именно так, потому что $123 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$.

Вычислительным машинам проще представлять числа в *двоичной системе счисления*, поскольку каждому разряду такой системы счисления соответствует всего две возможных цифры: 0 либо 1. Конечно, записи чисел в такой системе счисления в целом длиннее, чем в десятичной системе счисления. Для явного обозначения того факта, что число записано в определённой позиционной системе счисления, принято использовать её основание в качестве нижнего индекса. Например,

$$1111011_2 = 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 =$$

$$= 64_{10} + 32_{10} + 16_{10} + 8_{10} + 2_{10} + 1_{10} = 123_{10}.$$

Похожим образом можно легко вручную перевести число из любой q-ичной системы счисления в привычную десятичную систему счисления. Для этого достаточно просто по формуле (1) перемножить соответствующие цифры на соответствующие степени числа q. Например, важной для информатики системой счисления является шестнадцатеричная система, в которой $q = 16$, а для обозначения цифр от 10 до 15 используются большие буквы латинского алфавита от A до F:

$$7\mathrm{B}_{16} = 7 \cdot 16^1 + 11 \cdot 16^0 = 112_{10} + 11_{10} = 123_{10}.$$

Вручную перевести число из произвольной q-ичной системы счисления в десятичную несколько сложнее. Для этого нужно сформулировать не совсем тривиальное утверждение, оперирующее таким понятием, как остаток от деления.

Легко показать, что для любых двух целых чисел $n \in \mathbf{Z}$ и $m \in \mathbf{Z}$ найдутся два целых числа $k \in \mathbf{Z}$ и $r \in \mathbf{Z} \cap \left[0; |m| - 1\right]$, такие что

$$n = k \cdot m + r, \tag{2}$$

причём эти числа k и r можно выбрать единственным образом. В этом случае n называется *делимым, m – делителем, k – частным* или *целой частью от деления* n на m, а r *– остатком* от деления n на m.

Утверждение. Последняя цифра целого числа x в q-ичной системе счисления равна остатку от деления x на q.

Доказательство. Допустим, число x записывается в q-ичной системе счисления последовательностью цифр $\{a_k\}_{k=0}^{\infty}$, тогда в соответствии с формулой (1)

$$x = \sum_{k=0}^{\infty} a_k q^k = \sum_{k=1}^{\infty} a_k q^k + a_0 = q \sum_{k=1}^{\infty} a_k q^{k-1} + a_0 = q \sum_{k=0}^{\infty} a_{k+1} q^k + a_0 = qy + a_0.$$

Видно, что при делении числа x на q получается число y, состоящее из тех же цифр, что и x, но без последней цифры, соответствующей младшему разряду. В остатке как раз и остаётся эта последняя цифра в соответствии с определением остатка от деления по формуле (2).

Что и требовалось доказать.

Этот факт и используется для перевода числа из десятичной системы счисления в q-ичную. Общий алгоритм выглядит следующим образом.

Входные данные: целое число x, основание результирующей системы счисления q.

Выходные данные: цифры числа x в системе счисления q.

1. Найти a_0 как остаток от деления x на q. Это и есть последняя цифра числа x в q-ичной системе счисления.
2. Найти число y, равное целой части от деления x на q. Это число x без последней цифры.
3. Если $y \neq 0$, то перевести его в q-ичную систему счисления этим же алгоритмом, записать все его цифры.
4. Дописать цифру a_0.

Пример 2

Задача. Вручную перевести число 123 в двоичную систему счисления.

Решение. Будем делить текущее число пополам и выписывать остатки от деления.

$$123 = 2 \cdot 61 + 1,$$
$$61 = 2 \cdot 30 + 1,$$
$$30 = 2 \cdot 15 + 0,$$
$$15 = 2 \cdot 7 + 1,$$
$$7 = 2 \cdot 3 + 1,$$
$$3 = 2 \cdot 1 + 1,$$
$$1 = 2 \cdot 0 + 1.$$

Выписываем остатки в обратном порядке и получаем ответ.

Ответ: $123_{10} = 1111011_2$.

Пример 3

Задача. Вручную перевести число 123 в шестнадцатеричную систему счисления.

Решение. Будем делить текущее число 16 и выписывать остатки от деления.

$$123 = 16 \cdot 7 + 11,$$
$$7 = 16 \cdot 0 + 7.$$

Выписываем остатки в обратном порядке и получаем ответ.

Ответ: $123_{10} = 7B_{16}$.

Системы счисления, не являющиеся позиционными, называются *непозиционными системами счисления*. Классическим примером непозиционной системы счисления является римская система счисления, в которой числа представлялись, как сумма или разность своих цифр. В общем случае в непозиционных системах счисления числа могут представляться в виде рисунков, графов или даже верёвочных узлов.

Напоследок, как обычно, нужно отметить, что терминология, связанная с системами счисления, также не лишена определённой путаницы. В некоторой литературе позиционными системами счисления называются только q-ичные системы счисления. Если же в системе счисления разряды не являются степенями одного и того же числа q, то такие системы счисления называются *смешанными системами счисления*. Классическим примером смешанной системы счисления является фибоначчева система счисления, в которой разрядами служат числа Фибоначчи, затронутые в последующих разделах этой книги.

1.3 Представление целых чисел в памяти компьютера

Компьютерная память – это среда для хранения данных, позволяющая вычислительным процессам многократно получать доступ к одним и тем же данным в разные промежутки времени. Существует множество запоминающих устройств, отличающихся способами хранения, порядком доступа и прочими характеристиками. В первую очередь интересна оперативная память, поскольку состояние программ хранится именно в ней, причём не стоит сразу детально разбираться в физических принципах её работы.

С точки зрения программиста память компьютера состоит из множества последовательно расположенных ячеек, в каждой из которых может храниться одно из двух возможных значений: 0 или 1. Эти элементарные ячейки памяти называются *битами*. Таких ячеек довольно много, так что нумеровать каждую

из них было бы неразумно: вместо этого номера имеют только блоки памяти, содержащие одинаковое количество бит. Эти минимальные блоки памяти, имеющие номера, называются *байтами*, а сами номера – *адресами* этих байтов. В современных компьютерах каждый байт состоит из восьми бит.

В программах большая часть состояния хранится в виде *переменных* – именованных участков памяти, хранящих данные с определённым способом организации доступа к ним по имени. Чтобы организовать работу с данными, хранящимися в определённых переменных, программа хранит информацию о типе данных этих переменных. *Тип данных* определяет множество значений и операций с этими значениями. Подавляющее большинство языков программирования поддерживают довольно большое количество типов данных.

Большинство типов данных определяют, сколько в точности байт будет занимать область памяти, в которой хранятся данные этого типа. Для хранения переменных такого типа выделяется столько байт памяти с последовательными адресами. Нужно отметить, что если два байта памяти имеют последовательные адреса, то физически они тоже, скорее всего, располагаются рядом и получить доступ к каждому из них по очереди довольно легко.

Как уже стало понятно, целые числа хранятся в памяти компьютера в двоичном коде по одной цифре на каждый занятый бит памяти. Само собой, речь идёт о последовательно расположенных в памяти битах. Также очевидно, что для определённого типа данных, отвечающего за хранение целых чисел, должно быть заранее определено, сколько байт памяти он занимает. Таким образом, для неотрицательных целых чисел становится довольно очевидно, как их хранить.

Допустим, имеется тип данных для хранения неотрицательных целых чисел, занимающий n байт памяти. Это означает, что с помощью него можно хранить $8n$ цифр в двоичной системе счисления. Количество различных чисел, которые можно хранить в таком участке памяти, равно количеству способов разместить нули и единицы в $8n$ позициях, то есть 2^{8n}. Для чисел, состоящих

из меньшего количества разрядов, в старших разрядах просто хранятся незначащие лидирующие нули. Учитывая, что наименьшее число, которое мы можем хранить таким способом – это ноль (в этом случае все биты содержат нули), самое большое число, которое можно хранить таким способом – это $\left(2^{8n}-1\right)$. Числа, большие этого, требуют большего количества байт для хранения.

Например, число 123 можно хранить с помощью однобайтового целочисленного типа данных, как 01111011. Одного байта хватает, чтобы хранить 256 различных чисел. Ноль будет храниться, как 00000000, а самое большое хранимое таким способом число 255 – как 11111111.

Такие целочисленные типы данных, позволяющие хранить только неотрицательные целые числа, называются *беззнаковыми*. Они существуют во многих языках программирования. Их недостаток в том, что они не позволяют хранить отрицательные целые числа, хотя на практике последние, как правило, бывают нужны.

Для того чтобы отличать положительные целые числа от отрицательных, необходимо хранить знак, но всё равно на это нельзя использовать дополнительный бит сверх заранее отведённого под тип данных количества байт. Допустим, снова имеется n байт памяти на хранение целых чисел со знаком, причём первый бит будет отвечать за знак: числа, начинающиеся с нуля – положительные, а с единицы – отрицательные. Однако если хранить отрицательное целое число x, как единицу, после которой помещается двоичная запись положительного числа $|x|$, то наблюдаются две проблемы:

1) операции сложения и вычитания реализуются довольно сложно, поскольку приходится разбирать отдельные случаи для положительных и отрицательных чисел;

2) при такой записи присутствуют два нуля: положительный и отрицательный (например, для одного байта нулями являются последовательности 00000000 и 10000000).

Для решения этих проблем отрицательные целые числа хранят по-другому. Для n-байтового целочисленного типа данных будем хранить отрицательное целое число x, как $\left(2^{8n}+x\right)$. Такой способ хранения отрицательных целых чисел называется *дополнительным кодом*. Например, для $n=1$ число -1 будет храниться так же, как $256-1=255$, то есть в виде 11111111. Число -2 – как $256-2=254$, то есть в виде 11111110, и так далее.

При таком хранении существует всего один ноль, который хранится путём заполнения всех битов нулями. Хранить можно по-прежнему только 2^{8n} различных целых чисел, наименьшее из которых $\left(-2^{8n-1}\right)$, которое хранится как $2^{8n}-2^{8n-1}=2^{8n-1}$, то есть в виде одной единицы, за которой следуют нули. Наибольшее хранимое число при этом равно $\left(2^{8n-1}-1\right)$, которое хранится в виде одного нуля, за которым следуют единицы.

Таблица 2 – Пример хранения четырёхбитного целого числа со знаком

Число	**Двоичный код**	2^4+x	**Код числа**
–8	–1000	8	1000
–7	–111	9	1001
–6	–110	10	1010
–5	–101	11	1011
–4	–100	12	1100
–3	–11	13	1101
–2	–10	14	1110
–1	–1	15	1111
0	0		0000
1	1		0001
2	10		0010
3	11		0011
4	100		0100
5	101		0101
6	110		0110
7	111		0111

В таблице 2 приведены примеры хранения всех допустимых целых чисел для гипотетического знакового целочисленного типа данных, занимающего 4 бита, то есть половину байта (это называется *ниббл* (англ. *nibble*)). Конечно,

такого типа данных не существует практически ни на одном реальном вычислительном устройстве, да и пользы от него немного, но по таблице можно проследить закономерности в структуре дополнительного кода целого числа.

Кроме того, при таком хранении операция вычитания целых чисел со знаком может быть заменена на операцию сложения с противоположным числом. Например, чтобы вычесть из числа 3 число 5 нужно к представлению числа 3 (0011) прибавить представление числа -5 (1011). Складываем столбиком и получаем $0011+1011=1110$, что является дополнительным кодом числа -2 (см. таблицу 2), как и должно быть.

Перевести число из обычного представления в дополнительный код и обратно довольно легко и вручную. Определим инверсию битов в n-байтовом целом числе x, как операцию, меняющую все нули в $8n$ младших разрядах этого числа на единицы, а все единицы – на нули.

Утверждение. Для того чтобы получить n-байтовый дополнительный код отрицательного целого числа x, нужно инвертировать $8n$ младших битов положительного целого числа $(-x)$ и прибавить к получившемуся числу единицу.

Доказательство. Очевидно, что если прибавить к некоторому n-байтовому целому числу $(-x)$ число $\hat{x}$, полученное инверсией $8n$ младших битов в числе $(-x)$, то получится число, состоящее из $8n$ единиц, то есть число $\left(2^{8n}-1\right)$. Это объясняется тем, что в каждом разряде складываются 0 и 1. Таким образом, $(-x)+\hat{x}=2^{8n}-1$, откуда $2^{8n}+x=\hat{x}+1$.

Что и требовалось доказать.

Утверждение. Для того чтобы получить отрицательное целое число x из его n-байтового дополнительного кода c, нужно инвертировать $8n$ битов дополнительного кода c, прибавить к получившемуся числу единицу и взять противоположное число.

Доказательство. Аналогично предыдущему доказательству, если прибавить к некоторому дополнительному коду $c = 2^{8n} + x$ число $\hat{c}$, полученное инверсией $8n$ битов дополнительного кода c, то получится число, состоящее из $8n$ единиц, то есть число $\left(2^{8n} - 1\right)$. Таким образом, $2^{8n} + x + \hat{c} = 2^{8n} - 1$, откуда $x = -\left(\hat{c} + 1\right)$.

Что и требовалось доказать.

Главное, что нужно заметить, – не обязательно запоминать отдельно методы перевода числа в дополнительный код и дополнительного кода в число, поскольку оба перехода делаются абсолютно одинаково. По сути, для перевода в обе стороны нужно: а) инвертировать биты, б) прибавить единицу. Конечно, при переводе из дополнительного кода будет получено соответствующее положительное число.

Пример 4

Задача. Как число $\left(-123\right)$ хранится в памяти компьютера в виде однобайтового целого числа со знаком?

Решение. В предыдущих примерах мы уже выяснили, что $123_{10} = 01111011_2$. Инвертируем все биты этого числа и прибавим к нему единицу: 10000100+1=10000101. Это и есть дополнительный код числа $\left(-123\right)$.

Ответ: 10000101.

Пример 5

Задача. Какое число хранится в памяти компьютера в виде однобайтового числа со знаком как 10000101?

Решение. Инвертируем биты в коде 10000101 и прибавим единицу. Получим 01111010+1=01111011.

Известно, что $1111011_2 = 2^6 + 2^5 + 2^4 + 2^3 + 2^1 + 1 = 123_{10}$. Значит, искомое число – $\left(-123\right)$.

Ответ: -123.

В таблице 3 представлены стандартные целочисленные типы данных, поддерживаемые большинством вычислительных машин и реализованные в большинстве языков программирования. Для каждого из них приведён объём занимаемой памяти в байтах и в битах, информация о том, знаковый этот тип или беззнаковый, название этого типа данных в языке C, количество различных значений, принимаемых переменными этого типа, минимальное и максимальное возможное значение, а также примерное количество обеспечиваемых десятичных знаков.

Таблица 3 – Целочисленные типы данных

Байт	**Бит**	**Знак**	**Название в C**	**Кол-во**	**min**	**max**	**Цифр**
1	8	со знаком	`signed char`	256	−127	128	≈ 3
1	8	без знака	`unsigned char`	256	0	255	≈ 3
2	16	со знаком	`short`	65536	−32768	32767	≈ 5
2	16	без знака	`unsigned short`	65536	0	65535	≈ 5
4	32	со знаком	`int`	$\approx 4 \cdot 10^9$	$\approx -2 \cdot 10^9$	$\approx 2 \cdot 10^9$	≈ 10
4	32	без знака	`unsigned int`	$\approx 4 \cdot 10^9$	0	$\approx 4 \cdot 10^9$	≈ 10
8	64	со знаком	`long long`	$\approx 2 \cdot 10^{19}$	$\approx -9 \cdot 10^{18}$	$\approx 9 \cdot 10^{18}$	≈ 19
8	64	без знака	`unsigned long long`	$\approx 2 \cdot 10^{19}$	0	$\approx 2 \cdot 10^{19}$	≈ 20

Во-первых, нужно отметить, что на самом деле в стандарте языка C размеры большинства приведённых типов данных не определены, а лишь указано, что размеры более коротких типов не должны превышать размеры более длинных. Например, `int` не должен занимать больше, чем `long`, `short` не должен занимать больше, чем `int`, и так далее. Кроме того, не определено, должен ли обычный тип `char` быть знаковым или беззнаковым по умолчанию. На деле, как правило, эти типы данных занимают именно указанные в таблице 3 объёмы памяти, а тип `char`, как это ни удивительно, по умолчанию хранит числа со знаком.

Ещё одно замечение – в языке C присутствует также тип `long`, который не указан в таблице, поскольку в большинстве современных компиляторов он совпадает с типом `int`. В реальности на 64-битных машинах даже тип данных `int` может компилироваться как 64-битный со знаком. При написании

программы на C точные предельные значения для указанных типов данных можно получить, подключив заголовочный файл `<limits.h>`.

На практике следует с осторожностью применять беззнаковые целочисленные типы данных, поскольку при вычитании переменных таких типов можно легко выйти за границы допустимых для этого типа значений. По той же причине не стоит использовать короткие 16-битные целые числа. В обычной ситуации следует использовать тип данных `int` (32-битный со знаком), если же требуется хранить числа, превышающие миллиард, то тип `long long` (64-битный со знаком). Для хранения целых чисел, не умещающихся в 64-битные типы данных, используется специальный приём, называемый *длинной арифметикой*.

1.4 Арифметические операции с целыми числами

Понятно, что кроме хранения целых чисел вычислительные процессы обычно производят их обработку, выполняя с ними некоторые операции. Большая часть этих операций взята из математики и ранее уже была описана для математического понятия целых чисел. Основные бинарные *арифметические операции* с целыми числами, поддерживаемые большинством вычислительных устройств, приведены в таблице *4*. Для каждой из них приведено обозначение этой операции в языке C и пример её выполнения над некоторыми целыми операндами.

Таблица 4 – Арифметические операции с целыми числами

Название	**Обозначение в C**	**Пример операции**	**Результат**
Сложение	`+`	`7 + 3`	`10`
Вычитание	`-`	`7 - 3`	`4`
Умножение	`*`	`7 * 3`	`21`
Деление нацело	`/`	`7 / 3`	`2`
Остаток от деления	`%`	`7 % 3`	`1`

С первыми тремя операциями, приведёнными в таблице 4, всё более-менее понятно, кроме одной проблемы, о которой стоит упомянуть отдельно. Результат выполнения этих операций с некоторыми числами какого-нибудь

определённого типа данных может лежать за границами допустимых значений этого типа данных и просто не влезет в область памяти такого размера. Такая ситуация называется *целочисленным переполнением*. Она чаще возникает при выполнении операции умножения целых чисел, но также может возникнуть и при сложении или вычитании.

При возникновении целочисленного переполнения в результирующую область памяти просто записываются младшие биты результата, при этом не учитывается интерпретация старшего бита в качестве знака или отбрасывание старших битов, не попавших в область памяти нужного размера. Понятно, что в итоге в этой области памяти будет храниться неправильный результат операции. Поэтому надо понимать, что хотя свойства арифметических операций сложения и умножения целых чисел, приведённые в таблице 1, и выполняются для операций, указанных в таблице 4, они работают, только если не произошло целочисленного переполнения.

Пример 6

Например, однобайтовое число со знаком может хранить числа от -128 до $+127$ включительно. Если в этом типе данных умножить 32 на 16 (каждое из них можно хранить в переменной такого типа), то получится $32 \cdot 16 = 2^5 \cdot 2^4 = 2^9 = 512 = 1000000000_2$, но $512 > 127$ и не умещается в 1 байт памяти. В результате запишутся только 8 младших бит, то есть 8 нулей. Таким образом, в результате такого умножения получится ноль.

Пример 7

Другой пример: сложение чисел $100 + 100$ в рамках одного байта со знаком. Получится $200 = 11001000_2$. Это число умещается в 1 байт, но лидирующая единица в старшем разряде означает, что записано отрицательное число, хранящееся в дополнительном коде. Инвертируем биты и прибавляем единицу, чтобы выяснить, что это за число:

$$110111+1=111000=2^5 + 2^4 + 2^3 = 32 + 16 + 8 = 56_{10}.$$

Таким образом, если сложить два однобайтовых числа 100 со знаком и записать результат в один байт со знаком, то получится число (-56).

Целочисленное переполнение – одна из основных ошибок при работе с целыми числами. Чтобы избегать её, необходимо всегда помнить про неё в процессе написания программы. При выполнении каждой операции с целыми числами следует задумываться, не может ли результат её выполнения выйти за рамки допустимых для типа данных значений. В случае обнаружения такой возможности, простым решением проблемы является использование переменных более широкого типа данных (например, `long long` вместо `int`), либо приведение каких-либо проблемных операндов к более широкому типу данных. Если же результат операции не умещается в 64 бита, то опять же следует хранить вместо одного целого числа несколько и использовать длинную арифметику.

Иногда помогает вычисление операций в другом порядке. Например, при вычислении $(100 \cdot 100) / 100$ результат первой операции не умещается в 1 байт, а при вычислении $100 \cdot (100 / 100)$ будет получен корректный результат. Этот пример также в некотором роде иллюстрирует невыполнение свойства ассоциативности операции умножения для целочисленного типа данных в ситуации, когда происходит целочисленное переполнение.

Операция деления нацело, указанная четвёртой в таблице 4, тоже имеет одну неприятную особенность. Если оба операнда имеют один знак, то она, как и положено, вычисляет целую часть от деления, определяемую соотношением (2). В этом случае с точки зрения математики эту операцию можно записать $\lfloor n / m \rfloor$, где оператор $\lfloor x \rfloor$ означает целую часть от числа x, то есть наибольшее целое число, не превышающее x.

Однако если ровно один из операндов отрицательный, то она вычисляет результат деления нацело соответствующих положительных целых чисел, и

ставит перед результатом знак минус, то есть в целом можно записать результат работы этой операции над операндами n и m как

$$\operatorname{sgn}\left(\frac{n}{m}\right)\left\lfloor\left|\frac{n}{m}\right|\right\rfloor,$$

где оператор $\operatorname{sgn}(x)$ определяется как

$$\operatorname{sgn}(x)=\begin{cases}-1, & x<0;\\ 0, & x=0;\\ 1, & x>0.\end{cases}$$

Нужно понимать, что это не то же самое, что просто $\lfloor n/m\rfloor$, поскольку в общем случае $\lfloor -n/m\rfloor \neq -\lfloor n/m\rfloor$. Например, $\lfloor -1/2\rfloor = -1 \neq -\lfloor 1/2\rfloor = 0$. На практике же в большинстве случаев нужно именно разделить с округлением вниз, поэтому при выполнении операции деления рекомендуется внимательно относиться к знакам операндов. Следует задумываться о том, какой именно вид деления нужен в случае, когда один из операндов отрицательный, и обработать эту ситуацию отдельно, если нужно именно деление с округлением вниз. Лучше вообще стараться не выполнять деление, если один из операндов может быть отрицательным.

При выполнении операции деления целого числа n на целое число m распространена ошибка, при которой m неожиданно оказывается равной нулю. Такая ситуация известна, как *деление на ноль*. В математике эта операция не определена, поскольку лишена всякого смысла, но в программировании в силу особенностей хранения целых чисел в памяти компьютера, такая ситуация может возникнуть непосредственно в ходе работы программы. В этом случае программа немедленно завершает свою работу с ошибкой, о чём сообщает операционной системе ненулевым кодом возврата. Чтобы избежать этой ошибки, следует выполнять перед делением проверку, заключающуюся в сравнении знаменателя с нулём. Как правило, нулевой знаменатель означает какой-нибудь тривиальный случай, который должен быть разобран отдельно.

1.5 Арифметика остатков по модулю

Операция взятия остатка от деления и её свойства заслуживают отдельного детального рассмотрения. Для неотрицательных операндов она вычисляет остаток от деления в том виде, в котором он определяется соотношением (2). В математике для операции взятия остатка от деления $n \geq 0$ на $m > 0$ не определено специального обозначения, но из того же соотношения (2) можно заключить, что остаток равен

$$n - m\left\lfloor \frac{n}{m} \right\rfloor.$$

Вместо обозначения для операции остатка от деления в математике есть обозначение для так называемого *сравнения по модулю*. Говорят, что целое число a *сравнимо* с целым числом b по натуральному модулю m, если $(a-b)$ делится на m без остатка. Этот факт обозначается

$$a \equiv b \pmod{m}.$$

Другое определение: целое число a *сравнимо* с целым числом b по модулю m, если они дают одинаковые остатки от деления на m.

Утверждение. Приведённые выше определения эквивалентны.

Доказательство. Пусть a даёт остаток r_a от деления на m, а b даёт остаток r_b от деления на m. В соответствии с соотношением (2) это значит, что найдутся два целых числа k_a и k_b, что $a = k_a m + r_a$ и $b = k_b m + r_b$. Кроме того,

$$a - b = k_a m + r_a - k_b m - r_b = (k_a - k_b)m + (r_a - r_b).$$

Допустим, $(a-b)$ делится на m без остатка. Это значит, что $(k_a - k_b)m + (r_a - r_b)$ делится на m, а это значит, что каждое слагаемое должно делиться на m. Понятно, что $(k_a - k_b)m$ делится на m, но и $(r_a - r_b)$ тоже должно делиться на m, то есть если разность целых чисел делится на m, то и разность их остатков тоже должна делиться на m. Каждый из остатков r_a и r_b лежит в границах $0 \leq r_a, r_b \leq m-1$, значит их разность $-m+1 \leq r_a - r_b \leq m-1$, но

единственное целое число из этого промежутка, которое делится на m, – это 0. Выходит, $r_a - r_b = 0$, то есть $r_a = r_b$.

Верно и обратное. Если $r_a = r_b$, то $a - b = (k_a - k_b)m + (r_a - r_b) = (k_a - k_b)m$, то есть, очевидно, делится на m, поскольку $(k_a - k_b)$ – целое число.

Что и требовалось доказать.

Например, $7 \equiv 19 \pmod 3$, поскольку $7 - 19 = -12$, а (-12) делится на 3. С другой стороны, и 7, и 19 дают остаток 1 от деления на 3. Другой пример: неверно, что $14 \equiv 6 \pmod 5$, поскольку $14 - 6 = 8$, а 8 не делится на 5. С другой стороны, 14 даёт остаток 4 от деления на 5, а 6 даёт остаток 1 от деления на 5.

Сравнимость по модулю m – это бинарное отношение, в которое два целых числа a и b могут вступать, а могут не вступать. Это отношение удовлетворяет следующим трём основным свойствам отношений.

1. *Рефлексивность*:

$\forall a \in \mathbf{Z} : a \equiv a \pmod m$.

Доказательство. Очевидно, $a - a = 0$, а 0 делится на m без остатка.

Что и требовалось доказать.

2. *Симметричность*:

$\forall a,b \in \mathbf{Z} : a \equiv b \pmod m \Rightarrow b \equiv a \pmod m$.

Доказательство. Если $a \equiv b \pmod m$, значит, $(a - b)$ делится на m без остатка. В соответствии с соотношением (2) это означает, что найдётся целое число k, такое что $a - b = km$. Но тогда $b - a = -km$, что тоже делится на m без остатка, а это значит, что $b \equiv a \pmod m$.

Что и требовалось доказать.

3. *Транзитивность*:

$\forall a,b,c \in \mathbf{Z} : a \equiv b \pmod m \wedge b \equiv c \pmod m \Rightarrow a \equiv c \pmod m$.

Доказательство. Разделим $(a - c)$ на m:

$$\frac{a-c}{m}=\frac{a-b+b-c}{m}=\frac{a-b}{m}+\frac{b-c}{m}. \qquad (3)$$

Если $a\equiv b\pmod m$ и $b\equiv c\pmod m$, то и $(a-b)$, и $(b-c)$ делятся на m без остатка. Но тогда правая часть соотношения (3) представляет собой целое число, поскольку она является суммой двух целых чисел. Значит, слева тоже стоит целое число, то есть $(a-c)$ нацело делится на m, откуда можно заключить, что $a\equiv c\pmod m$.

Что и требовалось доказать.

Видно, что тем же самым трём основным свойствам отвечает отношение равенства двух целых чисел. Это означает, что сравнение по модулю задаёт *отношение эквивалентности* на множестве целых чисел. Согласно этому отношению все целые числа, дающие один и тот же остаток от деления на m, в определённом смысле эквивалентны друг другу.

В информатике наиболее интересны свойства сравнения по модулю, связанные с операциями сложения и умножения целых чисел по модулю.

Утверждение. Для любых целых чисел a_1, b_1, a_2, b_2 и для любого натурального модуля m если $a_1\equiv a_2\pmod m$ и $b_1\equiv b_2\pmod m$, то справедливы два соотношения:

1) $a_1+b_1\equiv a_2+b_2\pmod m$,

2) $a_1\cdot b_1\equiv a_2\cdot b_2\pmod m$.

Доказательство. По определению если $a_1\equiv b_1\pmod m$ и $a_2\equiv b_2\pmod m$, то (a_1-b_1) и (a_2-b_2) делятся на m без остатка.

1) Для операции сложения можно записать

$$\frac{a_1+b_1-a_2-b_2}{m}=\frac{(a_1-a_2)+(b_1-b_2)}{m}=\frac{a_1-a_2}{m}+\frac{b_1-b_2}{m}.$$

Справа стоит целое число, поскольку оба слагаемых целые. Это значит, что слева тоже стоит целое число, и $a_1+b_1\equiv a_2+b_2\pmod m$.

2) Для операции умножения можно записать

$$\frac{a_1 b_1 - a_2 b_2}{m} = \frac{a_1 b_1 - a_1 b_2 + a_1 b_2 - a_2 b_2}{m} = \frac{a_1 (b_1 - b_2) + b_2 (a_1 - a_2)}{m} =$$

$$= a_1 \frac{b_1 - b_2}{m} + b_2 \frac{a_1 - a_2}{m}.$$

Аналогично прошлому случаю справа стоит целое число, значит, и слева тоже, а значит, $a_1 \cdot b_1 \equiv a_2 \cdot b_2 \pmod{m}$.

Что и требовалось доказать.

Это утверждение означает, что левые и правые части сравнений по одному и тому же модулю можно складывать друг с другом и умножать друг на друга. На практике это означает, что если требуется вычислить какое-то выражение в целых числах, но в качестве ответа требуется предъявить не сам результат вычисления, а остаток от деления этого результата на какое-либо натуральное число m, то вместо этого можно брать остаток от деления на m после каждой операции сложения или умножения. Во многих случаях это помогает избежать целочисленного переполнения.

Свойства сложения и умножения сравнений для операции взятия остатка от деления в языках программирования можно записать на языке C следующим образом:

1) `(a + b) % m == (a % m + b % m) % m`;

2) `(a * b) % m == ((a % m) * (b % m)) % m`.

Пример 8

Задача. Напишите участок исходного кода программы, вычисляющий факториал целого числа n $(1 \le n \le 10^9)$ по заданному целому модулю m $(1 \le m \le 10^9)$.

Решение. Факториал натурального числа n обозначается $n!$ и определяется, как

$$n! = \prod_{k=1}^{n} k. \qquad (4)$$

Факториал является быстрорастущей функцией и для $n > 20$ уже не помещается в 64-битный целочисленный тип данных. Тем не менее, остаток от деления $n!$ на m таким недостатком не обладает. Чтобы вычислить ответ корректно, нужно вычислять остаток от деления на m после каждого умножения в соответствии со свойствами операции умножения сравнений.

Несмотря на то, что каждый множитель не превышает $(2^{31}-1)$, умножения всё равно придётся выполнять в рамках 64-битного типа, поскольку произведения двух чисел, не превышающих, $(2^{31}-1)$ могут превышать $(2^{31}-1)$, но не могут превышать $(2^{63}-1)$. Исходный код функции на языке C приведён ниже.

Листинг 1. Вычисление факториала по модулю

```
int fact(int n, int m)
{
    long long ans = 1;
    for (int i = 2; i <= n; ++i)
    {
        ans = ans * i % m;
    }
    return (int)ans;
}
```

Пример 9

Задача. Напишите участок исходного кода программы, вычисляющий сумму цифр целого числа n $(0 \le n \le 10^9)$.

Листинг 2. Вычисление суммы цифр натурального числа

```
int sum_of_digits(int n)
{
    int ans = 0;
    while (n > 0)
    {
        ans += n % 10;
        n /= 10;
    }
    return ans;
}
```

Решение. В соответствии с основным утверждением из раздела 1.2 последнюю цифру натурального числа можно определить, как остаток от деления этого числа на 10, а это же число без последней цифры – как результат деления этого числа 10 нацело. Будем делить на 10 до тех пор, пока не получится ноль, и складывать остатки. Исходный код на языке C приведён выше.

Пример 10

Задача. Напишите исходный код программы, принимающей на вход два целых числа n и q $(1 \le n \le 10^9,\ 2 \le q \le 10)$, записанных в десятичной системе счисления, и выводящей запись числа n в q-ичной системе счисления.

Решение. Снова утверждение из раздела 1.2 даёт алгоритм решения этой задачи. Последняя цифра натурального числа n в q-ичной системе счисления – это остаток от деления n на q, а результат целочисленного деления n на q – это то же число без последней цифры. Будем делить на q, пока не получим ноль, и выписывать в обратном порядке остатки от деления. Ниже приведён исходный код программы на языке C++.

Листинг 3. Перевод числа в заданную систему счисления

```cpp
#include <iostream>
#include <vector>

int main()
{
    int n, q;
    std::cin >> n >> q;
    std::vector<int> ans;
    while (n > 0)
    {
        ans.push_back(n % q);
        n /= q;
    }
    for (int i = (int)ans.size() - 1; i >= 0; --i)
    {
        std::cout << ans[i];
    }
    std::cout << std::endl;
    return 0;
}
```

Напоследок, как и в прошлом разделе, нужно заметить, что операцию взятия остатка от деления следует выполнять, только когда оба операнда положительны. Если второй операнд равен нулю, то во время выполнения программы возникнет ошибка деления на ноль. В остальном знак результата этой операции равен знаку первого операнда, то есть делимого. Знак второго операнда игнорируется. Часто отрицательный модуль свидетельствует об ошибках его вычисления, допущенных ранее, так что имеет смысл проверить этот случай до выполнения операции.

1.6 Битовые операции с целыми числами

Битовые операции (или *побитовые операции*) – это аналоги логических операций, применяемые над соответствующими битами участков памяти, в которых хранятся целые числа. Эти операции определены для целочисленных типов данных и выполняют привычные булевы операции над каждым разрядом соответственно. В таблице 5 приведены основные битовые операции, поддерживаемые большинством вычислительных устройств. Для каждой из них

приведено обозначение этой операции в языке С и пример выполнения этой операции с некоторыми целыми операндами.

Битовая инверсия уже рассматривалась ранее. Эта унарная операция инвертирует каждый бит в участке памяти, в котором хранится целое число, то есть заменяет 0 на 1 и 1 на 0. Ранее уже было показано, что дополнительный код числа $(-x)$ – это битовая инверсия числа x (обозначим её $\hat{x}$), увеличенная на единицу, то есть $-x=\hat{x}+1$, откуда $\hat{x}=-x-1$. То есть битовая инверсия неотрицательного целого числа x – это число, на единицу меньшее числа $(-x)$.

Таблица 5 – Битовые операции с целыми числами

Название	**Обозначение в С**	**Пример операции**	**Результат**
Битовая инверсия	~	`~5`	`-6`
Конъюнкция	&	`10 & 12`	`8`
Дизъюнкция	|	`10 | 12`	`14`
Сложение по модулю 2	^	`10 ^ 12`	`6`
Битовый сдвиг влево	<<	`11 << 2`	`44`
Битовый сдвиг вправо	>>	`11 >> 2`	`2`

Битовую конъюнкцию (битовое И) и битовую дизъюнкцию (битовое ИЛИ) не следует путать с соответствующими логическими операциями. Поскольку логический тип данных во многих языках программирования реализуется как целочисленный, логические операции для этого типа считают число 0 ложью, а остальные целые числа истиной, а вовсе не выполняют соответствующие операции над каждым битом. В языке С логические операции определены для всех целочисленных типов данных, что позволяет короче записывать некоторые логические условия, но влечёт ошибки, связанные с путаницей между логическими и битовыми операциями, а также ошибочное использование оператора присваивания = вместо оператора сравнения ==.

Примеры выполнения операций битовой конъюнкции и битовой дизъюнкции, приведённые в таблице 5, показывают результаты выполнения этих операций над операндами $10_{10}=1010_2$ и $12_{10}=1100_2$. Если применять к каждому разряду этих чисел операцию конъюнкции, то получается $8_{10}=1000_2$,

если применять операцию дизъюнкции, то получается $14_{10} = 1110_2$. Этот пример похож на таблицы значений булевых операций конъюнкции и дизъюнкции.

Таблица 6 – Свойства битовых операций с целыми числами

Свойство	Битовая дизъюнкция	Битовая конъюнкция
Ассоциативность	`((a | b) | c) == (a | (b | c))`	`((a & b) & c) == (a & (b & c))`
Нейтральный элемент	`(a | 0) == a`	`(a & -1) == a`
Коммутативность	`(a | b) == (b | a)`	`(a & b) == (b & a)`
Идемпотентность	`(a | a) == a`	`(a & a) == a`
Дистрибутивность	`(a | b & c) == ((a | b) & (a | c))`	`(a & (b | c)) == (a & b | a & c)`

В таблице 6 приведены некоторые алгебраические свойства операций битовой дизъюнкции и битовой конъюнкции. В отличие от свойств арифметических операций сложения и умножения, приведённых в таблице 1, у битовых операций дизъюнкции и конъюнкции для большинства операндов нет обратных элементов, но соблюдается свойство идемпотентности, которое не наблюдается у арифметических операций.

Эффективность использования битовых операций заключается в том, что большинство вычислительных устройств выполняет их значительно быстрее, чем арифметические. Операции умножения и деления выполняются дольше, чем операции сложения и вычитания, а битовые операции выполняются быстрее, чем операции сложения и вычитания. Это следует учитывать при написании исходного кода программ.

Пример 11

Задача. Напишите участок исходного кода программы, проверяющий, является ли заданное целое число нечётным.

Решение. Операция взятия остатка от деления, как и операция деления, одна из самых медленных арифметических операций. Вместо проверки остатка от деления на 2 предлагается проверить младший бит целого числа, ведь в соответствии с утверждением из раздела 1.2 оно и представляет собой остаток от деления на 2. Ниже приведён исходный код программы на языке C++.

Листинг 4. Проверка числа на нечётность

```
bool is_odd(int x)
{
    return x & 1;
}
```

Побитовое сложение по модулю два (англ. *XOR* от *Exclusive OR*) напоминает обычную операцию сложения столбиком без переноса единицы в старший разряд. В математике эта операция чаще всего обозначается символом $\oplus$ и обладает интересными свойствами.

1. *Ассоциативность*:

   $\forall a,b,c \in \mathbf{N}_0 : (a \oplus b) \oplus c = a \oplus (b \oplus c)$.

2. Ноль является *нейтральным элементом*:

   $\forall a \in \mathbf{N}_0 : a \oplus 0 = a$.

3. *Обратным элементом* к целому числу x является само число x:

   $\forall a \in \mathbf{N}_0 : a \oplus a = 0$.

4. *Коммутативность*:

   $\forall a,b \in \mathbf{N}_0 : a \oplus b = b \oplus a$.

Свойства записаны для математического аналога этой операции на неотрицательных целых числах, но понятно, что для целочисленных типов данных эти свойства сохраняются, в том числе для отрицательных чисел в дополнительном коде. Пример вычисления этой операции, показанный в таблице 5:

$10_{10} \oplus 12_{10} = 1010_2 \oplus 1100_2 = 0110_2 = 6_{10}$.

Операция побитового сложения по модулю два имеет множество интересных приложений на практике.

Пример 12

Задача. Напишите участок исходного кода программы, меняющий местами значения двух переменных целочисленного типа.

Решение 1. Классическим решением этой задачи является использование дополнительной переменной. Сначала значение первой переменной помещается в дополнительную переменную, затем значение второй переменной

помещается в первую, и, наконец, значение дополнительной переменной помещается во вторую. Этот способ работает не только для целочисленных типов данных, но и для переменных любого типа. Пример исходного кода на языке C приведён ниже.

Листинг 5. Обмен значений переменных через третью переменную

```
int t = a;
a = b;
b = t;
```

Решение 2. Другое решение основано на арифметических операциях сложения и вычитания. Сначала полагается $a' = a + b$, затем

$$b' = a' - b = a + b - b = a,$$

и наконец,

$$a'' = a' - b' = a + b - a = b.$$

Как видно, дополнительных переменных при этом можно не заводить, а присваивать каждый раз в ту же самую переменную.

Листинг 6. Обмен значений переменных через арифметику

```
a = a + b;
b = a - b;
a = a - b;
```

Решение 3. Наиболее эффективное решение основано на операции побитового сложения по модулю 2. Сначала полагается $a' = a \oplus b$, затем

$$b' = a' \oplus b = a \oplus (b \oplus b) = a,$$

и наконец,

$$a'' = a' \oplus b' = a \oplus b \oplus a = b \oplus (a \oplus a) = b.$$

Видно, что дополнительные переменные снова не требуются.

Листинг 7. Обмен значений переменных через XOR

```
a = a ^ b;
b = a ^ b;
a = a ^ b;
```

Операции *битового сдвига* выполняют сдвиг двоичной записи левого операнда на количество позиций, равное второму операнду. Все единицы, вышедшие за границы области памяти, содержащей целое число данного типа,

исчезают. Новые биты для неотрицательного левого операнда заполняются нулями. Например, для однобайтового целого числа со знаком операция `11 << 2` означает сдвиг числа $11_{10} = 00001011_2$ на 2 разряда влево, в результате чего получается число $00101100_2 = 44_{10}$. В той же ситуации операция `11 >> 2` означает сдвиг этого числа на 2 разряда вправо, в результате чего получается число $00000010_2 = 2_{10}$. В последнем случае видно, как две единицы, вышедшие за границы числа, потерялись при сдвиге. Эти операции имеют простую арифметическую интерпретацию.

Утверждение. Битовый сдвиг двоичного представления целого неотрицательного числа на один разряд влево соответствует умножению этого числа на 2, а битовый сдвиг двоичного представления целого неотрицательного числа на один разряд вправо соответствует целочисленному делению этого числа на 2.

Доказательство. Рассмотрим целое неотрицательное число x, записываемое в двоичной системе счисления цифрами $\{a_k\}_{k=0}^{\infty}$, то есть

$$x = \sum_{k=0}^{\infty} a_k 2^k .$$

При сдвиге влево на один бит получим число $x_{<<}$, записанное цифрами $a'_k = a_{k-1}$, при том что $a'_0 = 0$. Имеем

$$x_{<<} = \sum_{k=0}^{\infty} a'_k 2^k = \sum_{k=1}^{\infty} a'_k 2^k + 0 = \sum_{k=1}^{\infty} a_{k-1} 2^k = 2\sum_{k=1}^{\infty} a_{k-1} 2^{k-1} = 2\sum_{k=0}^{\infty} a_k 2^k = 2x.$$

При сдвиге вправо на один бит получим число $x_{>>}$, записанное цифрами $a''_k = a_{k+1}$. В этом случае можно записать

$$x = \sum_{k=0}^{\infty} a_k 2^k = \sum_{k=1}^{\infty} a_k 2^k + a_0 = 2\sum_{k=1}^{\infty} a_k 2^{k-1} + a_0 = 2\sum_{k=0}^{\infty} a_{k+1} 2^k + a_0 = 2x_{>>} + a_0 .$$

Таким образом, в соответствии с соотношением (2), $x_{>>}$ является целой частью от деления x на 2, а a_0 – остатком от деления.

Что и требовалось доказать.

Очевидным следствием из этого утверждения является тот факт, что для неотрицательных целых чисел операция `x << y` соответствует операции $x \cdot 2^y$, а операция `x >> y` – операции $\lfloor x / 2^y \rfloor$. Разумеется, первое справедливо, только если не возникает целочисленного переполнения, иначе единичные биты, вышедшие за границы области памяти, занимаемой переменной этого типа, просто потеряются. Таким образом, если требуется умножить или разделить целое число на степень двойки, то не требуется ни вычислять эту степень, ни выполнять долгие операции умножения или деления. Вместо этого можно просто воспользоваться соответствующей операцией битового сдвига: она работает быстро.

Упражнения для самостоятельной работы

1. Докажите основные свойства арифметических операций с натуральными числами.

2. Докажите формулы для сумм первых n членов арифметической и геометрической прогрессий.

3. Докажите, что среднее арифметическое n произвольных натуральных чисел не меньше их среднего геометрического.

4. Профессор О. П. Рометчивый доказал по индукции, что все люди одного роста. Пусть утверждение $P(n)$ заключается в том, что любые n человек одного роста. Очевидно, любой один человек одного роста сам с собой, так что $P(1)$ справедливо. Допустим, что $P(n)$ справедливо и любые n человек одного роста. Верно ли тогда, что любые $(n+1)$ человек тоже одного роста? Занумеруем всех людей и рассмотрим произвольное множество из $(n+1)$ человека: $A = \{a_1, a_2, \ldots, a_n, a_{n+1}\}$. Для любой пары людей a_i и a_j из множества A найдётся его подмножество из n человек, включающее обоих этих людей (в этом подмножестве просто должен отсутствовать любой другой человек, кроме a_i и a_j), значит, любая пара людей из этого множества одного роста. Это

означает, что вообще все люди из множества A одного роста, а это в силу произвольности выбора множества A означает, что согласно принципу математической индукции вообще все люди одного роста. Найдите ошибку в этом доказательстве.

5. В системах счисления с какими основаниями выполняется равенство $2+2=4$?

6. Докажите, что для однозначного представления каждого целого числа в q-ичной системе счисления требуется ровно q различных цифр от 0 до $(q-1)$ включительно.

7. Докажите, что для любых двух целых чисел $n \in \mathbf{Z}$ и $m \in \mathbf{Z}$ найдутся два целых числа $k \in \mathbf{Z}$ и $r \in \mathbf{Z} \cap \left[0; |m|-1\right]$, такие что $n = k \cdot m + r$, причём эти числа k и r можно выбрать единственным образом.

8. Найдите натуральные основания p и q, для которых $1234_p = 365_q$.

9. Приведите пример записи числа, состоящей только из цифр 0 и 1, такой что соответствующее число в любой позиционной системе счисления с произвольным основанием q не является простым. Можете ли вы привести пример такой записи, заканчивающейся на цифру 1?

10. Существует ли запись числа, состоящая только из цифр 0 и 1, такая что соответствующие этой записи числа во всех позиционных системах счисления с произвольным основанием q являются простыми?

11. В унарной системе счисления основание $q=1$ и всего одна цифра. Как вы думаете, какая это цифра, и как записываются числа в унарной системе счисления?

12. Как число (-1000) хранится в памяти компьютера в виде двухбайтового целого числа со знаком?

13. Какое число хранится в памяти компьютера в виде двухбайтового числа со знаком как 1111110000011000?

14. Допустим, в переменной `n` типа `int` (32 бита со знаком) хранится наименьшее целое число, которое может храниться в переменной такого типа. Что будет храниться в ней после выполнения операции `n = -n`?

15. Какой ответ выдаст функция из листинга 1 при $n \geq m$? Если в этой функции убрать операцию взятия остатка от деления на m, то какой ответ она будет выдавать при $m \geq 34$?

16. Напишите исходный код программы, которая считывает три целых числа a, b и c ($-10^9 \leq a,b,c \leq +10^9$) и выводит наибольшее из них.

17. Год является високосным, если его номер делится на 4. Но есть исключение: если его номер делится на 100, то такой год високосным не является. Но и из этого исключения есть исключение: если его номер делится на 400, то, не смотря на всё, он является високосным. Напишите исходный код программы, которая считывает единственное целое число y ($1 \leq y \leq 10^9$) и выводит «`LEAP`», если y-й год високосный, либо «`NOT LEAP`» в противном случае.

18. Напишите исходный код программы, которая считывает два целых числа a и b ($1 \leq a,b \leq 10^9$) и выводит результат деления a на b с округлением вверх до ближайшего целого. То есть программа должна выводить наименьшее целое число, такое что a / b не превышает его.

19. Напишите исходный код программы, которая считывает единственное целое число n ($1 \leq n \leq 10^9$) и выводит сумму первых n нечётных натуральных чисел.

20. Напишите исходный код программы, которая считывает четыре целых числа x_1, y_1, x_2 и y_2 ($-10^9 \leq x_1,y_1,x_2,y_2 \leq +10^9$) и выводит «`YES`», если два ферзя, стоящих на бесконечной доске в клетках с такими координатами, бьют друг друга, либо «`NO`» в противном случае. Ферзи бьют друг друга, если стоят на одной горизонтали, на одной вертикали или на одной диагонали.

21. Напишите исходный код программы, которая считывает единственное целое число x $(1 \le x \le 10^9)$ и выводит число, образованное из x выписыванием его десятичных цифр в обратном порядке.

22. Напишите исходный код программы, которая считывает единственное целое число x $(1 \le x \le 10^9)$ и выводит количество единиц в его двоичной записи.

23. Напишите исходный код программы, которая считывает единственное целое число x $(1 \le x \le 10^9)$ и выводит наименьшую степень двойки, такую что это число не превышает её.

24. Напишите исходный код программы, которая считывает два целых числа x и y $(1 \le x, y \le 10^9)$ и выводит наименьшее количество бит, которое нужно инвертировать в первом из них, чтобы получить второе.

25. Напишите исходный код программы, которая считывает три строки s_a, s_b и s_c, каждая из которых состоит не более чем из 9 цифр, и выводит наименьшее основание системы счисления q $(2 \le q \le 10)$, в которой для чисел a, b и c, соответствующих записям s_a, s_b и s_c, выполняется равенство $a + b = c$, либо указывает, что подходящего основания нет.

2 Числа с плавающей запятой

2.1 Понятие вещественных чисел

Как видно из таблицы 1, у целых чисел не хватает одного полезного свойства: наличия обратного элемента по умножению. *Обратным элементом* к x по операции $\circ$ называется такой элемент x^{-1}, что $x \circ x^{-1} = 1$, где 1 – это *нейтральный элемент* по этой операции, то есть такой что $x \circ 1 = x$. Попытка замкнуть множество целых чисел по операциям умножения и деления приводит к возникновению *рациональных чисел*, то есть множества несократимых дробей. Примерами рациональных чисел являются $1/2$ и $2/3$. Напомним, что целые числа были получены замыканием натуральных чисел по операциям сложения и вычитания.

Останавливаться на свойствах рациональных чисел особого смысла не имеет: большая часть из них подробно изучаются в школьном курсе математики. Большинство вычислительных устройств не поддерживают рациональные числа на аппаратном уровне: в этом нет никакой необходимости, ведь их можно реализовать в программе путём хранения числителя и знаменателя в виде двух целых чисел. Единственная сложность – после каждой арифметической операции следует делить и числитель, и знаменатель на их наибольший общий делитель, чтобы поддерживать их взаимную простоту.

Из курса математического анализа известно, что любое геометрическое расстояние может быть сколь угодно точно приближено последовательностью рациональных чисел. Тем не менее, и множества рациональных чисел также недостаточно, поскольку существуют отрезки на плоскости, не измеримые никаким рациональным числом. Например, таким отрезком является гипотенуза прямоугольного треугольника, катеты которого равны единице. В этом смысле множество рациональных чисел не является полным: в нём есть фундаментальные последовательности, не имеющие предела в этом множестве.

Пополнение пространства рациональных чисел и приводит к возникновению вещественных чисел.

Для курса информатики хватает школьного определения вещественных чисел, как множества бесконечных десятичных дробей. Определим множество *вещественных чисел* (или *действительных чисел*), как множество рядов вида

$$x = \sum_{k=-\infty}^{\infty} a_k 10^k ,$$

таких что

1) $\exists K \in \mathbf{Z}\ \forall k \in \mathbf{Z} : k > K \Rightarrow a_k = 0$ (начиная с какого-то разряда все цифры в старших разрядах равны нулю),

2) $\forall K \in \mathbf{Z}\ \exists k < K : a_K = 9 \Rightarrow a_k \neq 9$ (вещественное число не может оканчиваться на бесконечную последовательность девяток).

Каждое вещественное число однозначно определяется бесконечной в обе стороны последовательностью своих десятичных цифр $\{a_k\}_{k=-\infty}^{+\infty}$. Сравнения вещественных чисел реализуются поразрядно от старших разрядов к младшим. Операции сложения и умножения можно определить, как пределы последовательностей результатов этих операций с конечными десятичными дробями меньшего размера.

Первое свойство из определения ограничивает вещественные числа, запрещает бесконечно длинные. Второе свойство более специфическое. Дело в том, что бесконечные десятичные дроби, оканчивающиеся на бесконечную последовательность девяток, дублируют другие конечные десятичные дроби, оканчивающиеся на бесконечную последовательность нулей. Например, если допустить, что число $0,(9)$ является вещественным, то $0,(9) = 1$, потому что

$$0,(9) = 0,(3) \cdot 3 = \frac{1}{3} \cdot 3 = 1.$$

Чтобы множество вещественных чисел содержало только различные элементы, такие бесконечные дроби, оканчивающиеся на девятки, просто не включают в него.

2.2 Вещественные числа в двоичной системе счисления

Может показаться странным, что вещественные числа, которые сами по себе должны быть оторваны от их письменного представления, определяются именно через такое представление – в виде десятичных дробей. На самом деле, существуют и другие определения, как конструктивные, так и аксиоматические. Даже приведённое определение не обязательно должно быть основано именно на десятичной системе счисления. Само собой, вещественные числа, как и целые, могут быть представлены в любой позиционной системе счисления с основанием q в виде

$$x = \sum_{k=-\infty}^{\infty} a_k q^k ,$$

причём

1) $\exists K \in \mathbf{Z}\ \forall k \in \mathbf{Z} : k > K \Rightarrow a_k = 0$ (начиная с какого-то разряда все цифры в старших разрядах равны нулю),

2) $\forall K \in \mathbf{Z}\ \exists k < K : a_K = q-1 \Rightarrow a_k \neq q-1$ (вещественное число, записанное в q-ичной системе счисления, не может оканчиваться на бесконечную последовательность цифр $(q-1)$).

Более того, само определение вещественных чисел, приведённое ранее для десятичных дробей, может быть переписано для любой системы счисления, причём все такие определения эквивалентны. В информатике, как обычно, наибольший интерес представляет запись вещественных чисел в двоичной системе счисления. Из ранее сказанного ясно, что вещественные числа в двоичной системе счисления представляются в виде

$$x = \sum_{k=-\infty}^{\infty} a_k 2^k ,$$

причём

1) $\exists K \in \mathbf{Z}\ \forall k \in \mathbf{Z} : k > K \Rightarrow a_k = 0$ (начиная с какого-то разряда все цифры в старших разрядах равны нулю),

2) $\forall K \in \mathbf{Z}\ \exists k < K : a_K = 1 \Rightarrow a_k \neq 1$ (вещественное число, записанное в двоичной системе счисления, не может оканчиваться на бесконечную последовательность единиц).

Пример 13

Некоторые примеры вещественных чисел, записанных в двоичной системе счисления:

$11{,}11_2 = 1 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} = 3{,}75_{10}$,

$1{,}01_2 = 1 \cdot 2^0 + 1 \cdot 2^{-2} = 1{,}25_{10}$,

$101{,}101_2 = 1 \cdot 2^2 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-3} = 5{,}625_{10}$,

$0{,}000\left(1100\right)_2 = 0{,}1_{10}$.

Последнее свойство в определении опять же вызвано тем, что записи вещественных чисел в двоичной системе счисления, оканчивающихся на бесконечную последовательность единиц, обозначают те же числа, что и другие записи, оканчивающиеся на бесконечную последовательность нулей. Например, $0{,}\left(1\right) = 1$. Этот факт легко доказать: пусть $x = 0{,}\left(1\right)$, тогда $2x = 1{,}\left(1\right)$. Вычтем из второго равенства первое, получим $x = 1{,}\left(1\right) - 0{,}\left(1\right) = 1$.

Чтобы вручную перевести вещественное число из двоичной системы в десятичную, можно просто сложить степени двойки, при которых в записи стоят единицы, как показано в примере 13. Некоторые из этих степеней могут быть отрицательными, а позицию нулевого разряда показывает запятая. Понятно, что легко посчитать ответ только для конечных дробей небольшой длины.

Чтобы вручную перевести вещественное число из десятичной системы в двоичную, нужно отдельно перевести целую и дробную часть. Целая часть – это просто целое число, алгоритм перевода которого уже был описан ранее. Чтобы перевести дробную часть, будем последовательно умножать её на 2 и выписывать целые части получившегося числа, пока дробная часть не кончится.

Пример 14

Задача. Переведите число 12,1875 в двоичную систему счисления.

Решение. Перевод числа 12 не составляет труда, ведь

$12_{10} = 8 + 4 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^1 = 1100_2$.

Будем умножать 0,1875 на 2, выписывать целые части и отбрасывать их:

$0{,}1875 \cdot 2 = 0{,}375$ (целая часть равна 0),

$0{,}375 \cdot 2 = 0{,}75$ (целая часть равна 0),

$0{,}75 \cdot 2 = 1{,}5$ (целая часть равна 1, отбрасываем её, получаем $1{,}5 - 1 = 0{,}5$),

$0{,}5 \cdot 2 = 1$ (целая часть равна 1, дробной части нет, заканчиваем алгоритм).

Ответ: $12{,}1875_{10} = 1100{,}0011_2$.

Корректность такого алгоритма объясняется тем, что когда целая часть нулевая, она нулевая в любой системе счисления. После умножения на 2 в двоичной системе счисления целая часть становится равна цифре, которая до этого была первой после запятой, и в других системах счисления это тоже верно. После отбрасывания этой цифры снова имеем число с нулевой целой частью и переводим уже его.

Понятно, что в случае бесконечно большой дробной части этот алгоритм никогда не остановится, но так можно получить сколь угодно много цифр после запятой, пока точность не будет достаточной. Рациональные числа представляют собой периодические дроби, так что их можно перевести в другую систему счисления целиком, если найти длину периода, однако её поиск представляет собой отдельную задачу.

2.3 Числа с фиксированной запятой

Понятно, что память компьютера ограничена, и хранить в ней бесконечные вещественные числа невозможно. Логичным решением этой проблемы является хранение чисел с некоторой точностью. Тогда некоторые вещественные числа, для записи которых требуется лишь небольшое количество цифр, будут храниться точно.

Первым, что приходит в голову, является хранение заданного количества двоичных цифр записи числа до и после запятой. Тип данных, предназначенный для этого, характеризуется общим количеством бит n, которое занимает переменная такого типа, и количеством бит $0 \le m \le n$, которое отводится под хранение дробной части числа. При $m = 0$ получается обычный целочисленный тип данных. Также один бит может отводиться под знак.

Числа, хранящиеся в рамках такого типа данных, называются *числами с фиксированной запятой* (англ. *fixed-point numbers*). Если под хранение положительного вещественного числа x отводится n бит, из которых m заняты под хранение дробной части, то вместо него будет храниться целое число

$$\hat{x} = \lfloor x \cdot 2^m \rfloor.$$

В случае если исходное число отрицательное, можно хранить число $\hat{x}$ в дополнительном коде, либо просто отвести один бит под хранение знака. Например, число $5{,}625$ будет храниться в виде 8-битного числа с фиксированной запятой, в котором 4 бита отводится под дробную часть, в виде 01011010, потому что $5{,}625_{10} = 0101{,}1010_2$.

Арифметические операции для чисел с фиксированной запятой реализуются как соответствующие арифметические операции для целых чисел с последующим битовым сдвигом на необходимое количество разрядов. При сложении и вычитании сдвиг не нужен: эти операции полностью совпадают с соответствующими операциями над целыми числами. Большинство свойств, характерных для операций над целыми числами также сохраняется.

Хранение дробных чисел в виде чисел с фиксированной запятой обладает следующими преимуществами.

1. Скорость выполнения операций не слишком превышает скорость выполнения соответствующих операций с целыми числами.

2. Возможность хранить величины, естественным образом представимые в формате с фиксированной запятой. Например, количество денег в рублях не может содержать больше двух десятичных цифр после запятой.
3. Предсказуемое поведение на любых платформах вне зависимости от особенностей реализации, что также свойственно целым числам.
4. Числа хранятся равномерно на всём диапазоне хранения. В отрезках одинаковой длины хранится одинаковое количество чисел.

Тем не менее, у такого подхода существуют критические недостатки.

1. Узкий диапазон хранимых значений. Например, 8-битное число с фиксированной запятой, в котором 4 бита заняты под дробную часть и 1 под знак, может хранить лишь числа из отрезка $[-8;+7{,}9375]$ с шагом $0{,}0625$.
2. Опасность переполнения в старших разрядах, основанная на целочисленном переполнении. При получении слишком больших по модулю результатов старшие разряды, не подлежащие хранению, просто теряются.

На практике числа с фиксированной запятой применяются крайне редко. Для них не предусмотрены специальные типы данных в большинстве языков программирования, а также большинство электронно-вычислительных машин не поддерживают их на аппаратном уровне. Это связано, во-первых, с указанными выше недостатками, а во-вторых, в этом нет необходимости: их можно реализовать в программе с использованием обычных целочисленных типов данных. Тем не менее, некоторые встроенные системы, а также некоторые языки программирования, поддерживают числа с фиксированной запятой.

2.4 Хранение чисел с плавающей запятой в памяти компьютера

Естественной альтернативой числам с фиксированной запятой является идея хранения положения запятой в числе отдельно. Однако тип данных предполагает наличие общего ограничения на число бит, так что предлагается часть участка памяти, отведённого на хранение числа, занять под хранение

положения запятой в его двоичной записи. Такая форма хранения чисел называется *числами с плавающей запятой* (англ. *floating point numbers*).

Сразу нужно указать на небольшую путаницу в переводе, поскольку *floating point* дословно означает «плавающая точка». Это связано с тем, что в зарубежной литературе дробная часть десятичной дроби отделяется от целой части точкой, а не запятой, как это принято в отечественной литературе. Из-за этого существует так же вариант названия чисел с плавающей запятой «числа с плавающей точкой», который считается менее корректным, но всё равно используется в некоторой литературе.

Участок памяти, отведённый под хранение числа с плавающей запятой, разбивается на три части:

1. Знак – 1 бит, предназначенный для хранения знака числа (плюс или минус).
2. Мантисса – участок памяти, предназначенный для хранения определённого количества старших разрядов двоичной записи числа.
3. Порядок – участок памяти, хранящий целое число, обозначающее показатель степени двойки, на которую нужно умножать хранимое число.

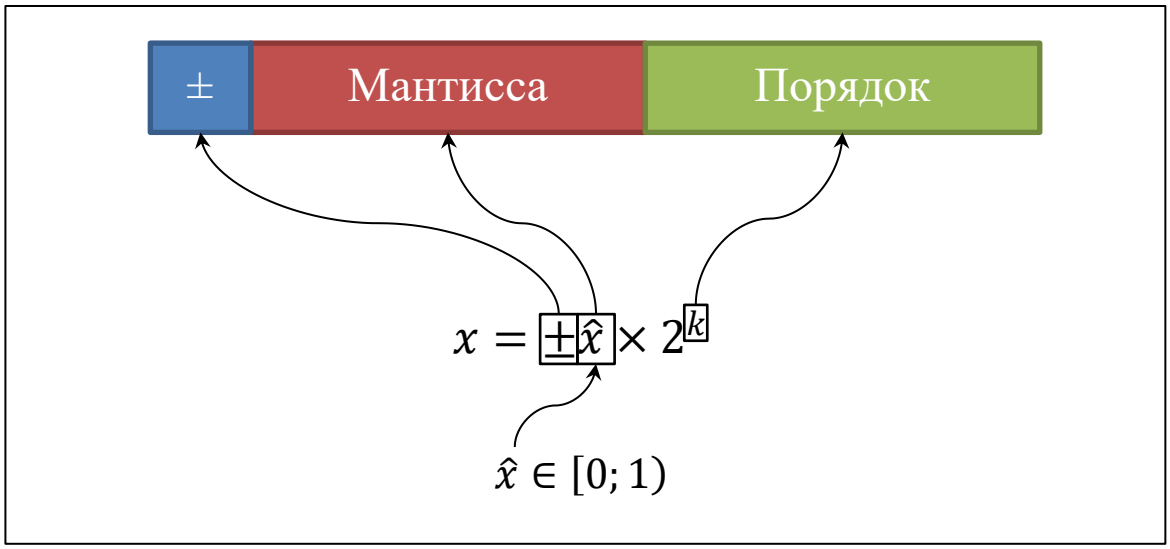

Рисунок 1 – Представление числа с плавающей запятой в памяти компьютера

На рисунке 1 схематично показано, каким образом вещественное число x хранится в памяти компьютера. Любое вещественное число $x \in \mathbf{R}$, кроме нуля, может быть представлено в виде

$$x = (-1)^s \cdot \hat{x} \cdot 2^k, \qquad (5)$$

где $s \in \{0;1\}$ – знак числа, $\hat{x} \in [0,5;1)$ – значащие цифры числа, $k \in \mathbf{Z}$ – порядок числа. По сути $\hat{x}$ – это вещественное число, двоичная запись которого начинается на 0,1. Заданное количество следующих старших разрядов этой записи и хранятся в мантиссе.

Пример 15

Задача. Представить число 12,1875 в виде (5).

Решение. Ранее уже было показано, что $12{,}1875_{10} = 1100{,}0011_2$. Это положительное число, запятая находится на четвёртой позиции после первой значащей цифры, так что

$$1100{,}0011_2 = (-1)^0 \cdot 0{,}11000011_2 \cdot 2^4.$$

В десятичной форме $0{,}11000011_2 = 2^{-1} + 2^{-2} + 2^{-7} + 2^{-8} = 0{,}76171875_{10}$, но именно цифры 1000011 будут храниться в мантиссе числа с плавающей запятой.

Ответ: $12{,}1875 = (-1)^0 \cdot 0{,}76171875 \cdot 2^4$.

Допустим, имеется всего 8 бит, из которых первый отводится под знак, следующие 3 – под мантиссу, а последние 4 – под порядок. Тогда число 12,1875 будет записываться как такое 8-битное число с плавающей запятой в виде 01000100. Если разделить вертикальными линиями три части числа, то получится 0|100|0100. Знак числа положительный, что обозначается первой цифрой 0. Само число в двоичной форме $1100{,}0011_2$, но первая единица не хранится, поскольку в двоичной записи всего две цифры – 0 и 1, так что первая значащая цифра всегда 1. Хранятся только три старших цифры, кроме первой единицы. Порядок числа $4_{10} = 100_2$, так что порядок записывается в виде 0100. Отметим, что если бы порядок был отрицательный, он был бы записан в дополнительном коде. По сути, хранится не число $12{,}1875_{10} = 1100{,}0011_2$, а

число $(-1)^0 \cdot 0{,}1100_2 \cdot 2^4 = 12_{10}$. Дробная часть находится в младших разрядах, и на её хранение не хватило точности.

Первое, что следует отметить, – если не хранить первую значащую единицу, то нет возможности хранить ноль, потому то в записи нуля, по сути, вообще нет ни одной значащей цифры. Конечно, можно хранить все значащие цифры, включая первую единицу, а ноль записывать в виде мантиссы, состоящей из одних нулей, но тогда одинаковые вещественные числа могут быть представлены по-разному, поскольку, например, $0{,}0100_2 \cdot 2^3 = 0{,}0010_2 \cdot 2^4$. Альтернативным решением проблемы является соглашение о том, что одна из легальных записей числа с плавающей запятой будет считаться нулём, а не тем числом, которое она представляет на самом деле.

Как видно, хранить числа с плавающей запятой в памяти компьютера можно по-разному. Число способов хранения довольно велико, особенно учитывая произвольное задание числа бит, отводимых под хранение мантиссы и порядка. Во избежание путаницы существует стандарт IEEE 754, который определяет конкретные способы хранения чисел с плавающей запятой, которые могут поддерживаться аппаратными средствами. Большинство вычислительных машин и языков программирования придерживаются этого стандарта, однако в полной мере, по всей вероятности, он не поддерживается ничем.

Таблица 7 – Типы данных для чисел с плавающей запятой

Бит	Байт	Название в С	Десятичных знаков точности
32	4	`float`	7
64	8	`double`	15
80	10	`long double`	19

В таблице 7 приведены наиболее распространённые на практике типы данных для чисел с плавающей запятой. Для каждого из них указано общее количество бит и байт, которое занимает в памяти одно значение такого типа, название этого типа в языке С, а также примерное количество десятичных знаков точности, обеспечиваемое этим типом данных. Примерное это значение потому, что на самом деле обеспечивается целое количество двоичных знаков

точности (размер мантиссы). В таблице для удобства оно переведено в количество десятичных знаков. Можно считать, что указанный тип данных хранит только заданное количество первых значащих десятичных разрядов числа, а остальную часть отбрасывает.

Стоит заметить, что последний расширенный тип данных, представляющий собой 10-байтовое число с плавающей запятой, поддерживается не всеми вычислительными устройствами. Изначально это был специальный тип данных, реализованный только в сопроцессорах Intel. На сегодняшний день в некоторых встроенных системах этот тип данных также может не поддерживаться. В подавляющем большинстве случаев на практике для работы с дробными числами следует использовать тип данных `double`.

Как было сказано ранее, ноль является в некотором роде специальным значением. Под него отведено специальное значение мантиссы и порядка, но не знака. Поэтому в действительности среди чисел с плавающей запятой существует два нуля: положительный ноль и *отрицательный ноль* в зависимости от значения знакового бита. Это два различных числа с плавающей запятой, оба из которых соответствуют одному и тому же вещественному числу. Согласно стандарту IEEE 754 операторы сравнения положительного и отрицательного нуля должны полагать их равными, однако на практике это правило выполняется далеко не во всех программных и аппаратных средах. Следует быть осторожней при работе с этими специальными значениями.

Стандарт также определяет три других специальных значения для чисел с плавающей запятой: положительная бесконечность, отрицательная бесконечность и не число (англ. *not a number*, NaN). Способы получения этих значений и свойства операций с ними будут рассмотрены в следующем разделе. Нужно отметить лишь, что следует учитывать возможность их возникновения и проявлять осторожность при работе с ними.

Преимущества хранения чисел с плавающей запятой по сравнению с числами с фиксированной запятой состоят в следующем.

1. Широкий диапазон хранимых значений за счёт возможности хранения чисел с большим значением порядка.
2. Возможность более точного хранения чисел, близких к нулю, за счёт возможности хранения чисел с отрицательным значением порядка.

Для записи чисел с плавающей запятой в исходном коде программ на большинстве языков программирования используются литералы с плавающей запятой. *Литерал* – это синтаксическая конструкция в исходном коде программы, представляющая собой конкретное значение. Например, `0`, `13` и `-1` – целочисленные литералы.

В литералах чисел с плавающей запятой в качестве разделителя целой и дробной части используется точка. Перед самим числом может стоять знак плюс или минус. По умолчанию полагается плюс. Нулевую часть можно не записывать. Число может быть записано в экспоненциальной форме. Для этого после числа записывается символ `e` (регистр не важен) и значение десятичного порядка. Так запись вида `±nEm` означает число $\pm n \cdot 10^m$. В таблице 8 приведены некоторые примеры литералов для чисел с плавающей запятой и значения вещественных чисел, которым они соответствуют. Следует помнить, что каким бы большим или маленьким по модулю ни было число с плавающей запятой, оно хранит лишь небольшое конечное количество цифр в старших разрядах.

Таблица 8 – Примеры литералов для чисел с плавающей запятой

Литерал	**Значение**
`12.34`	12,34
`.1`	0,1
`1.0e100`	10^{100}
`-1.0`	–1
`1.2e-5`	0,000012

2.5 Операции над числами с плавающей запятой

Числа с плавающей запятой поддерживают стандартные арифметические операции, такие как сложение, вычитание, умножение и деление, так же как и целые числа. Большинство вычислительных устройств поддерживают также некоторые элементарные функции, такие как модуль, корень, синус, косинус, арктангенс, экспонента, логарифм и т. д. В языке С эти операции объявлены в заголовочном файле `math.h`. Также поддерживаются операции сравнения. Тем не менее, работа с числами с плавающей запятой значительно отличается от работы с целыми числами.

Понятно, что каждое число с плавающей запятой соответствует некоторому вещественному числу. Однако, чисел с плавающей запятой, в отличие от вещественных чисел, конечное число. Основная особенность, о которой следует помнить при работе с такими числами, – после каждой операции вместо реального результата выполнения этой операции над соответствующими вещественными числами получается наиболее близкое к этому результату число с плавающей запятой. Сам результат операции при этом может вообще не представляться в виде числа с плавающей запятой, например, в силу бесконечного количества цифр. Скажем, результат деления числа 1 на число 3 – бесконечная периодическая дробь. Только часть цифр этой дроби будут храниться в качестве числа с плавающей запятой на практике.

Этот факт следует учитывать при работе с числами с плавающей запятой. Следует понимать, что после каждой операции с этими числами получается неточный результат. В случае с целыми числами неправильный ответ может получиться, только если возникает целочисленное переполнение, а в случае с числами с плавающей запятой, ошибка возникает практически после каждой операции. Кроме того, однажды возникшая погрешность имеет обыкновение накапливаться, поскольку в следующие операции передаётся не точный результат операции, а уже искажённый вариант, который ещё больше искажается в дальнейшем.

Первое и главное заключение, которое можно сделать из этого факта, – числа с плавающей запятой нельзя сравнивать на равенство. Допустим, в результате вычислений получилось два числа с плавающей запятой и теперь требуется проверить, равны ли они. Проблема заключается в том, что реальные вещественные числа, которые могли получиться в результате этих операций, могут быть равны, но числа с плавающей запятой получены с некоторой погрешностью, так что они могут оказаться не равными. Кроме того, существует опасность сравнения отрицательного и положительного нулей, бесконечностей, не чисел, а также некоторые другие.

В листинге 8 на языке C++ приведён известный пример ошибки, которая может возникать при сравнении двух равных на первый взгляд чисел с плавающей запятой. На некоторых платформах указанный код может выводить сообщение о том, что косинусы единицы не равны между собой. Это связано со сложной особенностью хранения чисел с плавающей запятой в регистрах сопроцессора.

Листинг 8. Ошибка сравнения чисел с плавающей запятой

```
#include <iostream>
#include <cmath>

void foo(double x, double y)
{
    if (std::cos(x) != std::cos(y))
    {
        std::cout << "WTF???" << std::endl;
    }
}

int main()
{
    foo(1.0, 1.0);
    return 0;
}
```

Чтобы избежать подобных ошибок следует взять за правило никогда не сравнивать числа с плавающей запятой на точное равенство или точное неравенство. Вместо этого следует вычитать из одного числа другое и сравнивать модуль их разности с некоторым достаточно маленьким числом.

Пример правильного сравнения чисел с плавающей запятой на языке C++ приведён в листинге 9.

Листинг 9. Сравнение чисел с плавающей запятой

```
const double EPS = 1.0E-6;

bool is_equal(double x, double y)
{
    return abs(x - y) < EPS;
}
```

Само маленькое число `EPS` следует выбирать с осторожностью. С одной стороны оно должно быть достаточно маленькое, чтобы действительно различные числа ошибочно не полагались одинаковыми. То есть оно должно быть меньше, чем возможная абсолютная погрешность вычислений для этой задачи. С другой стороны, если сделать его слишком маленьким, то одинаковые числа могут ошибочно полагаться различными из-за погрешности их вычисления, как если бы они просто сравнивались на точное равенство.

При желании можно достаточно точно оценить возможную погрешность вычислений для конкретной задачи. Этим занимается теория погрешностей, которая является разделом направления прикладной математики, называемого «численные методы». Это богатая теория, даже введение в которую лежит далеко за рамками курса информатики. Следует понимать, что некоторые выражения сами по себе не могут быть посчитаны точно, поскольку даже небольшое отклонение в исходных данных для них влечёт существенные изменения результата вычисления. Такие операторы называются *вычислительно неустойчивыми*.

Отдельным недостатком такого способа сравнения является тот факт, что оно не задаёт на множестве чисел с плавающей запятой отношение эквивалентности, поскольку не является транзитивным. Очевидно, что для любых трёх вещественных чисел x, y и z, таких что $x = y$ и $y = z$ выполняется также равенство $x = z$. Для операции сравнения, определяемой в листинге 9 для чисел с плавающей запятой, это условие не выполняется, что

влечёт некоторые проблемы с доказательствами корректности построенных алгоритмов.

Что касается переполнения, характерного для операций с целыми числами, то его аналог существует и для чисел с плавающей запятой. Хотя на практике он менее распространён и менее опасен, чем в случае с целыми числами, всё равно следует про него помнить. *Переполнение числа с плавающей запятой* происходит, когда происходит целочисленное переполнение значения порядка этого числа. Это может означать, что порядок стал слишком большой по модулю положительный или отрицательный.

В случае переполнения порядка в большую сторону в большинстве случаев не произойдёт никакой ошибки, программа продолжит свою работу, а значением результата такой операции станет бесконечность: положительная бесконечность, если переполнилось положительное число, либо отрицательная бесконечность, если переполнилось отрицательное число. В случае переполнения в обратную сторону, если порядок стал слишком маленьким, результатом будет ноль, который в зависимости от знака исходного числа также может быть положительным или отрицательным.

Так в ходе работы программы переменные могут принимать бесконечные значения. Другая ситуация, в которой это может произойти – деление числа с плавающей запятой на ноль. В большинстве случаев, опять же, программа не упадёт с ошибкой, а вместо этого результатом такой операции будет бесконечность. Положительная бесконечность больше любого другого числа с плавающей запятой, а отрицательная – меньше. Операции сложения и вычитания с обычными числами с плавающей запятой не изменяют бесконечности. Следует проявлять осторожность и помнить о возможности возникновения таких значений при написании программы.

Даже самые распространённые операции сложения и вычитания могут представлять опасность в смысле возникновения погрешностей. Если складывать большое число с плавающей запятой и маленькое, то разрядов на

хранение результата может не хватить, и исходное число вообще не изменится. Например, при сложении чисел 1 и 10^{17} в рамках типа данных `double`, результатом будет 10^{17}, поскольку на хранение единицы потребовалось бы 17 десятичных разрядов, а тип `double`, поддерживает только 15, как можно видеть из таблицы 7.

С этой особенностью операций сложения и вычитания связано понятие машинного эпсилон. *Машинный эпсилон* – это наибольшее число с плавающей запятой, которое при прибавление к единице даёт в результате единицу. Сравнение чисел порядка единицы друг с другом невозможно с точностью, меньше машинного эпсилон. Кроме того, к числам порядка единицы нельзя прибавлять числа порядка, меньшего, чем у машинного эпсилон. Понятно, что значение машинного эпсилон зависит от типа данных и от его аппаратной реализации. Оно может быть определено с помощью простой программы, пример которой на языке С приведён в листинге 10.

Листинг 10. Вычисление машинного эпсилон

```
double machine_eps()
{
    double eps = 1.0;
    while (1.0 + eps != 1.0)
    {
        eps /= 2.0;
    }
    return eps;
}
```

Указанная особенность также может приводить к проблемам при вычислении суммы большого количества чисел с плавающей запятой. Например, пусть требуется сложить число 10^{17} и 10^6 единиц в рамках типа данных `double`. Если прибавлять к числу 10^{17} единицу 10^6 раз, то каждый раз никаких изменений происходить не будет, как было показано выше, и результат будет отличаться от правильного результата на 10^6. Если же сначала сложить 10^6 единиц, а затем прибавить к числу 10^{17} число 10^6, то разрядов на хранение такой суммы хватит, и результат будет храниться точно. Поэтому существует

правило, что при суммировании большого количества чисел с плавающей запятой, сначала следует суммировать маленькие числа, а затем большие. Это может быть важно, например, при вычислении суммы ряда.

2.6 Не число

Не число (NaN) – это одно из специальных значений, которое могут принимать числа с плавающей запятой, представляющее собой объект, не являющийся по своей сути числом. Оно заслуживает отдельного рассмотрения, поскольку часто может возникать из-за некоторых ошибок в программировании вычислений с плавающей запятой, тогда как многие программисты вообще не подозревают о его существовании. В большинстве случаев появление не числа, как и возникновение бесконечности, не сопровождается никакими ошибками в ходе работы программы, хотя в некоторых случаях вместо этого программа может завершать работу с ошибкой. По этой причине сознательной работы с не числами рекомендуется избегать, и, напротив, следить за тем, чтобы они не могли возникнуть в ходе работы программы.

Не число может возникнуть по трём основным причинам.

1. Выполнение арифметической операции, одним из операндов которой является не число.
2. Выполнение арифметической операции, аналог предела для которой представляет собой неопределённость.
   2.1. Деление нуля на ноль (как положительного, так и отрицательного).
   2.2. Деление бесконечности на бесконечность (как положительной, так и отрицательной).
   2.3. Сложение положительной и отрицательной бесконечностей.
   2.4. Ноль в степени ноль.
   2.5. Единица в степени бесконечность.
   2.6. Бесконечность в степени ноль.
3. Элементарные функции, приводящие к мнимым результатам.
   3.1. Корень из отрицательного числа.

3.2. Логарифм отрицательного числа.

3.3. Арксинус или арккосинус числа, по модулю превышающего единицу.

Всех перечисленных ситуаций следует избегать в ходе написания программы, выполняя соответствующие проверки перед выполнением операции, способной привести к возникновению не числа. Как правило, эти проверки необходимы и по смыслу решаемой задачи, поскольку позволяют выявить тривиальные случаи, для которых ход решения может отличаться от обычного.

Не число обладает интересными свойствами.

1. Все арифметические операции с не числами возвращают не числа.
2. Из предыдущего свойства есть исключение. Единица в степени не число равна единице.
3. Операция сравнения числа на равенство с нечислом возвращает ложь. Даже в случае сравнения не числа на равенство с самим собой.

Теперь становится ясно, в чём состоит опасность возникновения не чисел в программе. Во-первых, при возникновении одного не числа, многие другие переменные в результате операций с ним также станут не числами. Во-вторых, все операции сравнения начнут возвращать ложь. В-третьих, не числа нарушают рефлексивность отношения эквивалентности, поскольку они не равны сами себе.

При желании можно проверить, хранится ли в данной переменной не число. Распространённой ошибкой является сравнение этой переменной с не числом: в любом случае такая операция вернёт ложь. Вместо этого нужно сравнить переменную с самой собой: если она не равна самой себе, то она хранит не число.

Листинг 11. Проверка, хранит ли переменная не число

```
bool is_NaN(double x)
{
    return x != x;
}
```

2.7 Ошибки при вычислениях с плавающей запятой

Суммируя описанные выше свойства и особенности чисел с плавающей запятой, можно сформулировать следующие проблемы возникающие в процессе написания программ с их использованием.

1. После каждой операции с числами с плавающей запятой происходит некоторая ошибка вычисления: результатом является лишь близкое к настоящему результату число.
2. Нельзя сравнивать числа с плавающей запятой на точное равенство – можно лишь сравнивать модуль разности с некоторым маленьким числом, выбор которого представляет собой отдельную проблему.
3. Нельзя складывать отличающиеся на много порядков числа: такая операция просто оставляет без изменения число большого порядка.
4. В результате операций с числами с плавающей запятой могут возникать специальные значения, такие как бесконечности и не числа, которые могут повлечь аномальное поведение некоторых последующих операций.

Все эти проблемы следует учитывать при написании исходного кода программ. Чтобы не совершать ошибок, можно придерживаться простых правил.

1. Вообще не использовать числа с плавающей запятой, если задачу можно решить в целых числах.
2. Никогда не сравнивать два числа с плавающей запятой на точное равенство. Вместо этого можно сравнивать модуль их разности с маленьким числом эпсилон.
3. Перед сложением большого количества чисел с плавающей запятой их следует упорядочить по возрастанию их модулей.
4. Каждое выражение, содержащее числа с плавающей запятой, следует записывать так, чтобы минимизировать возможную погрешность вычислений.

5. Если входные данные задачи не являются целыми числами, но представлены с заданным количеством знаков после запятой, то можно умножить их на достаточно большое число, чтобы они стали целыми, решить задачу в целых числах, а затем скорректировать результат с учётом масштабирования входных данных.

Пример 16

Задача. Напишите исходный код программы, принимающей на вход три целых числа a, b и c ($-1000 \le a,b,c \le +1000$), и выводящей все вещественные корни уравнения $ax^2 + bx + c = 0$, либо сообщающей, что их бесконечно много.

Листинг 12. Решение квадратного уравнения

```
#include <iostream>
#include <cmath>

int main()
{
    int a, b, c;
    std::cin >> a >> b >> c;
    if (a == 0)
    {
        if (b == 0)
        {
           if (c == 0)
           {
               std::cout << "Infinitely many solutions" << std::endl;
           }
        }
        else
        {
            std::cout << (double)(-c) / b << std::endl;
        }
    }
    else
    {
        int d = b * b - 4 * a * c;
        if (d == 0)
        {
            std::cout << (double)(-b) / (2 * a) << std::endl;
        }
        else if (d > 0)
        {
            std::cout << (-b - std::sqrt(d)) / (2 * a) << std::endl;
            std::cout << (-b + std::sqrt(d)) / (2 * a) << std::endl;
        }
    }
    return 0;
}
```

Решение. Входные данные представляют собой целые числа. Не следует приводить их к виду чисел с плавающей запятой, пока не потребуется

вычислить ответ. Например, дискриминант такого квадратного уравнения представляет собой целое число и может сравниваться с нулём в целых числах, чтобы избежать возможных ошибок сравнения. Кроме того, для успешного решения требуется проверить немало тривиальных случаев, без учёта которых может возникнуть деление на ноль. В листинге 12 приведён исходный код программы на языке C++.

Упражнения для самостоятельной работы

1. Докажите в рамках рассмотренного определения вещественных чисел, что между любыми двумя вещественными числами существует рациональное, а между любыми двумя рациональными – вещественное.

2. Переведите вещественное число $101011{,}101011_2$ в десятичную систему счисления.

3. Переведите вещественное число $43{,}671875$ в двоичную систему счисления.

4. В системе счисления с каким основанием верно равенство $12{,}12 + 23{,}23 = 40{,}40$?

5. Допустим, некто решил записывать вещественные числа в двоичной системе счисления, используя все десять цифр, характерных для десятичной системы (основанием системы счисления остаётся число 2). В чём могут выражаться преимущества и недостатки такого подхода? Приведите примеры двух различных записей одного и того же вещественного числа из интервала $(0;1)$ в такой системе счисления.

6. Найдите натуральные основания p и q, для которых $12{,}12_p = 5{,}32_q$?

7. Докажите, что любое вещественное число, кроме нуля, представимо в форме (5), причём единственным образом.

8. Напишите исходный код программы, принимающей на вход целочисленные координаты двух точек на плоскости (x_1, y_1) и (x_2, y_2) ($-1000 \le x_1, y_1, x_2, y_2 \le +1000$), и выводящей евклидово расстояние между ними.

9. Напишите исходный код программы, принимающей на вход координаты точки на плоскости в полярной системе координат: расстояние от начала координат r и полярный угол θ, который радиус-вектор точки образует с осью абсцисс, ($0 < r \le 1000$, $0 \le \theta < 2\pi$), и выводящей координаты этой точки в декартовой системе координат.

10. Напишите исходный код программы, принимающей на вход целочисленные координаты центров двух окружностей (x_1, y_1) и (x_2, y_2), а также их радиусы r_1 и r_2 ($-1000 \le x_i, y_i \le +1000$, $0 < r_i \le 1000$), и выводящей расстояние между этими окружностями. Расстояние между окружностями – это наименьшее расстояние между парой точек, одна из которых лежит на первой окружности, а вторая – на второй.

11. Напишите исходный код программы, принимающей на вход координаты заданного конечного количества точек на плоскости $\{(x_i, y_i)\}_{i=1}^{n}$ ($-1000 \le x_i, y_i \le +1000$), и выводящей сообщение «YES», если все эти точки лежат на одной примой, либо «NO» в противном случае.

12. Напишите исходный код программы, принимающей на вход коэффициенты в уравнениях заданного конечного количества прямых вида $ax + by + c = 0$ на плоскости: $\{(a_i, b_i, c_i)\}_{i=1}^{n}$ ($-1000 \le a_i, b_i, c_i \le +1000$, $a_i^2 + b_i^2 > 0$), и выводящей количество различных точек пересечения этих прямых.

13. Напишите исходный код программы, принимающей на вход коэффициенты в уравнениях заданного конечного количества прямых вида $ax + by + c = 0$ на плоскости: $\{(a_i, b_i, c_i)\}_{i=1}^{n}$ ($-1000 \le a_i, b_i, c_i \le +1000$, $a_i^2 + b_i^2 > 0$), и выводящей количество областей, на которое эти прямые разбивают плоскость.

14. Напишите исходный код программы, принимающей на вход целочисленные размеры двух прямоугольных конвертов (длину a_i и ширину b_i) ($-1000 \le a_1, b_1, a_2, b_2 \le +1000$), и выводящей либо сообщение «YES», если один

из этих конвертов можно, не складывая и не сминая, положить в другой, либо «NO» в противном случае.

3 Построение и анализ алгоритмов

3.1 Понятие и свойства алгоритмов

В предыдущих разделах уже приводились некоторые алгоритмы, например, для перевода чисел из одной системы счисления в другую. Пришло время разобраться, что вообще такое алгоритмы, и как они строятся. Следует внимательно относиться к терминологии, чтобы не путать понятия «алгоритм» и «программа».

Понятие алгоритма имеет богатую историю. Предположительно, оно появилось в IX веке, как имя автора книги о позиционной десятичной системе счисления. Позже понятие претерпевало серьёзные изменения, особенно связанные с появлением вычислительной техники. Современное определение в отечественной научной школе связано с работами советского математика Андрея Маркова.

Алгоритм – это набор точных предписаний, описывающих действия исполнителя, необходимые для достижения конечного результата. В нашем случае исполнителем является электронная вычислительная машина, так что алгоритм, по сути, содержит набор элементарных инструкций для неё, позволяющий получить результат из заданных входных данных. Часто в определении алгоритма используют слово «последовательность» вместо слова «набор», что не вполне корректно, поскольку чёткая последовательность выполнения инструкций в некоторых алгоритмах может быть не определена. Например, это относится к существенно параллельным алгоритмам.

Программа – это данные, определяющие действия, выполняемые вычислительным устройством с другими данными. Часто понятия «алгоритм» и «программа» путают. В действительности алгоритм существует в виде некоторой идеи, он не имеет материальной формы, в то время как программа – это конкретные данные, хранящиеся на конкретном материальном носителе, определяющие действия над другими данными.

Приведённое определение алгоритма является неформальным, поэтому не может быть подвергнуто изучению математическими методами. Существуют также формальные определения понятия «алгоритм» в виде машины Тьюринга, нормального алгоритма Маркова, рекурсивных функций и некоторые другие. Они позволяют строить формальные доказательства вычислимости некоторых алгоритмов. Раздел математики, занимающийся изучением формальных моделей алгоритмов и их свойств, называется *теорией алгоритмов*. В рамках курса информатики введение в эту теорию рассматриваться не будет.

Традиционно выделяют свойства алгоритмов, которым в той или иной мере отвечают все алгоритмы. Ниже приведены наиболее распространённые из них.

1. *Дискретность*. Алгоритм должен разбиваться на счётное количество элементарных инструкций.
2. *Универсальность*. Алгоритм должен решать весь класс задач, для которого предназначен, для всех допустимых входных данных.
3. *Понятность*. Алгоритм может включать только инструкции, понятные исполнителю.
4. *Результативность*. Алгоритм должен предоставлять определённый результат после своего завершения.
5. *Детерминированность*. В каждый момент времени в процессе исполнения алгоритма следующий шаг алгоритма должен определяться текущим его состоянием.
6. *Конечность*. Алгоритм должен завершаться за конечное количество шагов.

Эти свойства ещё менее формальны, чем само определение алгоритма, и практически не несут никакого смысла и не помогают в понимании понятия алгоритма. Кроме того, существуют стохастические алгоритмы, которые не являются детерминированными и могут работать бесконечно долго. Для научного исследования следует использовать формальные определения алгоритмов и их формальные свойства.

Пример 17

Задача. Крестьянину требуется перевезти через реку волка, козу и капусту. У него имеется лодка, в которой кроме самого крестьянина может поместиться лишь один другой объект: либо волк, либо коза, либо капуста. Он может поместить в лодку один из этих объектов, который находится на том же берегу, что и он сам, и перевезти его на другой берег, либо переправиться на другой берег в одиночку. Проблема в том, что если оставить козу и капусту на одном берегу без присмотра, то коза съест капусту, а если оставить волка и козу, то волк съест козу. Ни одной из этих ситуаций нельзя допускать. Постройте алгоритм, позволяющий перевезти все три объекта на другой берег.

Решение. Запишем алгоритм в виде последовательности элементарных действий.

1. Погрузить в лодку козу и отвезти её на другой берег. Волк капусту не съест.
2. Переправиться обратно в одиночку.
3. Погрузить в лодку волка и перевезти его на другой берег.
4. Погрузить в лодку козу и отвезти её обратно, иначе волк её съест.
5. Погрузить в лодку капусту и перевезти её на другой берег.
6. Переправиться обратно в одиночку. Волк капусту не съест.
7. Погрузить в лодку козу и перевезти её на другой берег. На другом берегу одновременно окажутся и волк, и коза, и капуста.

3.2 Вычислительная сложность алгоритма

Прежде чем приступать к реализации разработанного алгоритма в большинстве случаев требуется оценить его эффективность, чтобы не тратить время на заведомо бесполезную работу. В некоторых ситуациях, программная реализация придуманного алгоритма решения задачи может работать годы или даже века на реальных входных данных, так что оценка эффективности алгоритма и при необходимости разработка более эффективного алгоритма решения могут значительно сэкономить время. Во многих случаях анализ эффективности алгоритма не представляет труда и легко может быть выполнен

в уме, хотя изредка, напротив, такой анализ может представлять определённую сложность.

Главной характеристикой, влияющей на эффективность алгоритма, по праву можно считать время исполнения его программных реализаций. Однако это время может зависеть от способа реализации, от языка программирования, от компилятора, от платформы, на которой будет исполняться программа. Поэтому вместо времени в секундах в качестве оценки эффективности алгоритма используют количество элементарных операций, которое будет выполнено потенциальным вычислительным устройством. Понятно, что реальное время исполнения программы практически монотонно зависит от числа операций.

Можно заметить, что число операций, выполняемых по алгоритму, естественным образом зависит от входных данных. Чаще всего важен размер входных данных: подано ли на вход алгоритму 10 чисел или 10000 чисел, однако в некоторых случаях важны также конкретные значения. Зависимость числа элементарных операций в заданном алгоритме от входных данных называется *вычислительной сложностью* этого алгоритма. Именно эта характеристика и является основным показателем эффективности алгоритмов. Умение быстро оценивать вычислительную сложность разработанного алгоритма очень важно при написании программ.

Пример 18

Задача. Дано натуральное число n. Требуется вывести все пары различных натуральных чисел, не превышающих n. Каждую пару следует вывести ровно один раз. Пары, отличающиеся порядком чисел, считать одинаковыми. Разработайте алгоритм решения этой задачи и точно оцените его вычислительную сложность.

Решение. Решение этой задачи не представляет труда. Просто переберём все пары в некотором порядке и выведем результат. В качестве иллюстрации

разработанного алгоритма приведём исходный код его программной реализации на C++.

Листинг 13. Вывод различных пар натуральных чисел

```
#include <iostream>

int main()
{
    int n;
    std::cin >> n;
    for (int i = 1; i <= n; ++i)
    {
        for (int j = i + 1; j <= n; ++j)
        {
            std::cout << i << ' ' << j << std::endl;
        }
    }
    return 0;
}
```

Оценим количество элементарных операций $T(n)$, выполняемых алгоритмом, программная реализация которого приведена в листинге 13, в зависимости от заданного на входе числа $n \in \mathbf{N}$. Будем считать элементарными операциями все арифметические операции, а также считывание, вывод, присваивание.

Сначала выполняется одно считывание, затем одно присваивание. Далее внешний цикл выполняется n раз: сначала для $i=1$, затем для $i=2$, и так далее. В последний раз он выполняется для $i=n$. Отметим, что при $i=n+1$ всё равно осуществляется последняя проверка условия входа во внешний цикл.

Для заданного $i \in \{1,2,\ldots,n\}$ всегда происходит одно присваивание и одно сложение, после чего внутренний цикл выполняется $(n-i)$ раз. Внутри цикла каждый раз происходит 4 вывода. В итоге имеем следующую оценку вычислительной сложности:

$$T(n)=3+\sum_{i=1}^{n}\left(5+6(n-i)\right)=3+5n+6\sum_{i=1}^{n}(n-i)=3+5n+6\sum_{i=1}^{n-1}i=$$

$$= 3 + 5n + 6\frac{1+n-1}{2}(n-1) = 3 + 5n + 3n(n-1) = 3n^2 + 2n + 3. \quad (6)$$

На рисунке 2 поясняется большинство констант, используемых при выводе вычислительной сложности (6). Ещё раз стоит заметить, что условие выхода из цикла выполняется на единицу больше раз, чем его тело.

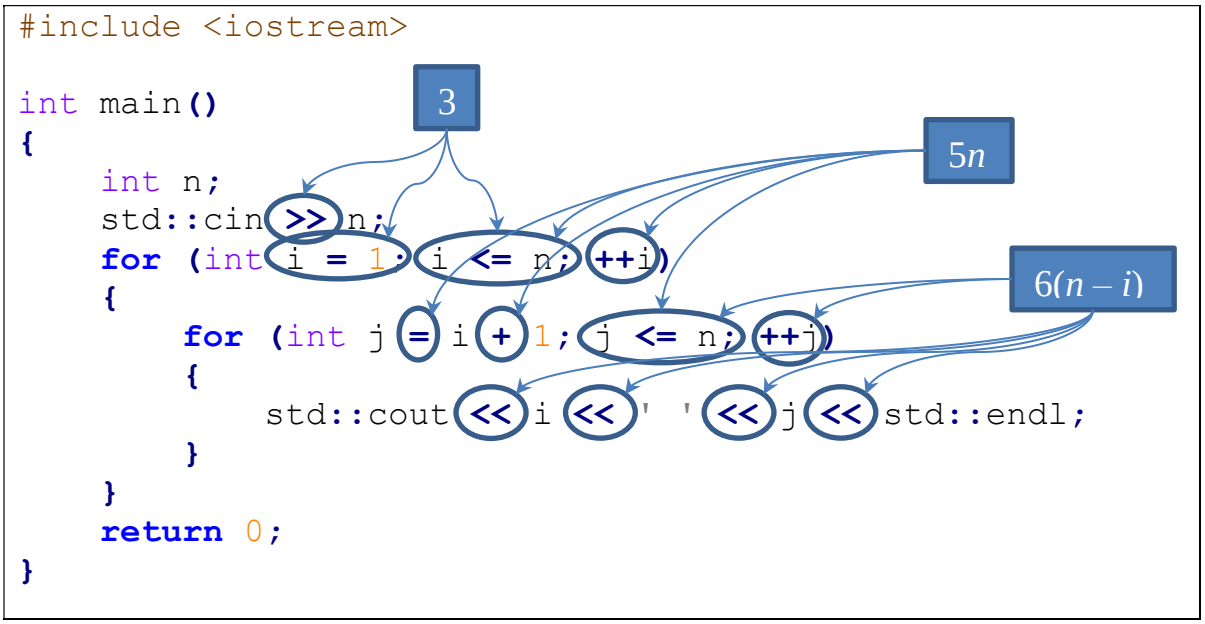

Рисунок 2 – Анализ вычислительной сложности алгоритма

Ответ: $T(n) = 3n^2 + 2n + 3$.

В действительности можно поспорить с тем, что вычислительная сложность (6) оценена точно. Это связано, прежде всего, с определением, что является элементарной операцией. При реальном исполнении программы из листинга 13 количество операций, выполняемых вычислительным устройством, может существенно отличаться от оценки (6). Даже директива `#include<iostream>` включает в себя выполнение большого количества операций, содержащихся в исходном коде файла `iostream`. Чтение и запись также являются сложными операциями.

Кроме того, на практике даже элементарные арифметические операции выполняются разное время. Так известно, что умножение и деление выполняются медленнее, чем сложение и вычитание. Поэтому в некоторых случаях зависимость числа этих операций от входных данных оценивается

отдельно. Зависимость числа операций сложения и вычитания от входных данных называется *аддитивной сложностью* алгоритма, а зависимость числа операций умножения и деления от входных данных называется *мультипликативной сложностью* алгоритма.

Для алгоритма, проиллюстрированного в листинге 13, мультипликативная сложность $T_{\times}(n)=0$, поскольку не выполняется ни одной операции умножения или деления, а аддитивная сложность

$$T_{+}(n)=\sum_{i=1}^{n}(2+n-i)=2n+\sum_{i=1}^{n}(n-i)=2n+\sum_{i=1}^{n-1}i=2n+\frac{1+n-1}{2}(n-1)=$$

$$=2n+\frac{n(n-1)}{2}=2n+\frac{n^2}{2}-\frac{n}{2}=\frac{n^2}{2}+\frac{3n}{2}.$$

Часто, особенно в случаях большого разнообразия входных данных, требуется оценить вычислительную сложность алгоритма вне зависимости от вариативности некоторых из входных данных. Для этого можно рассмотреть все возможные значения части входных данных и оценить предельные значения вычислительной сложности. *Вычислительная сложность в лучшем случае* – это наименьшая вычислительная сложность алгоритма, которую можно получить при варьировании входных данных. *Вычислительная сложность в худшем случае* – это наибольшая вычислительная сложность алгоритма, которую можно получить при варьировании входных данных. Также можно определить *среднюю вычислительную сложность* – это средняя по всем вариантам входных данных вычислительная сложность алгоритма.

Допустим, для алгоритма, проиллюстрированного в листинге 13, входное число n может принимать значения от 1 до 1000 включительно. Тогда, очевидно, что вычислительная сложность в лучшем случае

$$T_{\min}=T(1)=3+2+3=8,$$

вычислительная сложность в худшем случае

$$T_{\max}=T(1000)=3000000+2000+3=3002003,$$

а средняя вычислительная сложность

$$\overline{T} = \frac{1}{1000}\sum_{n=1}^{1000} T(n) = \frac{1}{1000}\sum_{n=1}^{1000}\left(3n^2 + 2n + 3\right) = \frac{1}{1000}\left(3\sum_{n=1}^{1000} n^2 + 2\sum_{n=1}^{1000} n + 3000\right) =$$

$$= \frac{1}{1000}\left(3\frac{2\cdot 10^9 + 3\cdot 10^6 + 1000}{6} + 2000\frac{1+1000}{2} + 3000\right) =$$

$$= \frac{1}{1000}\left(1000\cdot\frac{2\cdot 10^6 + 3000 + 1}{2} + 1000\cdot 1001 + 3000\right) =$$

$$= \frac{2\cdot 10^6 + 3000 + 1}{2} + 1001 + 3 = 10^6 + 1500 + 0{,}5 + 1004 = 1002504{,}5\,.$$

Замечание 1. После оценки вычислительной сложности в лучшем случае, в худшем случае и в среднем получившаяся оценка перестаёт зависеть от варьируемых входных данных. В примере оценка (6) и так зависела только от n, так что после оценки вычислительной сложности в худшем случае, в лучшем случае и в среднем, она вообще перестала представлять собой функцию и окончательно потеряла смысл. Обычно оценку в лучшем случае, в худшем случае и в среднем проводят для больших наборов входных данных, добиваясь, чтобы вычислительная сложность зависела только от размера входных данных, а не от их конкретных значений.

Замечание 2. Чтобы вычислить среднюю вычислительную сложность алгоритма не достаточно просто посчитать среднее из вычислительных сложностей в лучшем и в худшем случае. Вместо этого нужно искать среднее по всем вариациям входных данных. Например, для оценки (6)

$$\frac{T_{\min} + T_{\max}}{2} = \frac{8 + 3002003}{2} = 1501005{,}5 \neq 1002504{,}5 = \overline{T}\,.$$

3.3 Асимптотическая оценка вычислительной сложности

Как можно заметить, точная оценка вычислительной сложности алгоритмов – довольно сложное и неблагодарное занятие. Причин этому сразу несколько.

1. Различные элементарные операции могут выполняться реальными вычислительными устройствами разное время.

2. Различные программные реализации одного и того же алгоритма могут иметь различное количество операций.
3. Один и тот же исходный код программной реализации алгоритма может породить разное количество инфраструктурных операций при компиляции различными компиляторами под различные платформы.
4. Со временем вычислительные устройства становятся более производительными, так что в несколько раз более медленные алгоритмы могут начать работать за то же время, что и в настоящий момент работают более быстрые алгоритмы.
5. На практике интерес представляет только общий характер зависимости количества операций от входных данных, а не мелкие детали.
6. В большинстве случаев при увеличении размера входных данных и их значений на вычислительную сложность оказывают влияние лишь некоторые быстро растущие слагаемые.
7. Для вычисления точной зависимости количества операций от входных данных требуется много времени и стараний. В некоторых случаях может оказаться быстрее реализовать алгоритм и посмотреть, сколько работает программа.

Чтобы решить эти проблемы, предлагается оценивать вычислительную сложность не точно, а приблизительно, пренебрегая незначительными деталями. Главное при таком оценивании – понять, насколько быстро вообще растёт эта функция, а для этого нужно выделить только самую быстро растущую её компоненту. Постоянные множители также не важны, поскольку длительность отдельных элементарных операций может варьироваться.

Например, для алгоритма, продемонстрированного в листинге 13, была получена вычислительная сложность $T(n) = 3n^2 + 2n + 3$. Понятно, что когда n уже достаточно большое, то на рост функции $T(n)$ влияет только слагаемое $3n^2$. Слагаемое $2n$ растёт значительно медленнее, а слагаемое 3 вообще не растёт с ростом n. Сама константа 3 перед n^2 также не играет особой роли:

одни вычислительные устройства сами по себе могут быть в три раза быстрее других, так что на эффективность алгоритма это не влияет. Получается, что основной результат исследования эффективности алгоритма заключается в том, что его вычислительная сложность $T(n)$ растёт как n^2.

В математике для формальной записи подобных результатов существует так называемая O-нотация. Она служит для сравнения асимптотического поведения функций. В нашем случае она нужна для изучения поведения функции вычислительной сложности алгоритма с ростом входных данных.

Пусть имеется две функции $f(n)$ и $g(n)$. Обозначим

$$f(n) = O(g(n)) \Leftrightarrow \exists C \in \mathbf{R}\ \exists N \in \mathbf{N}\ \forall n > N : |f(n)| < C|g(n)|,$$

то есть будем говорить, что $f(n)$ есть $O(g(n))$ («о» большое от $g(n)$), если найдётся такая константа, что, начиная с некоторого аргумента N, функция $f(n)$ ограничена сверху функцией $g(n)$, помноженной на эту константу. Смысл этой конструкции в том, что функция $g(n)$ может быть значительно проще функции $f(n)$, так что для больших значений n можно ограничиться исследованием функции $g(n)$ вместо $f(n)$.

Сразу нужно оговориться, что модуль в этом определении не несёт смысла, поскольку число операций не может быть отрицательным. Он остался от общего математического определения, подходящего для любых функций. Кроме того, поскольку в нашем случае исследуемые функции определены на множестве натуральных чисел, существует лишь конечное множество значений аргумента, не превышающих N, поэтому определение, указанное выше, для нашего частного случая можно переписать проще:

$$f(n) = O(g(n)) \Leftrightarrow \exists C \in \mathbf{R}\ \forall n \in \mathbf{N} : f(n) < Cg(n),$$

то есть в нашем случае найдётся такая константа, что вообще для всех значений аргумента функция $f(n)$ будет ограничена сверху функцией $g(n)$, помноженной на эту константу.

Для примера алгоритма из листинга 13 можно записать $T(n)=O(n^2)$, поскольку существует константа $C=10$, такая что для всех $n\in\mathbf{N}$ верно, что $3n^2+2n+3<Cn^2$. Это верно, потому что для $n=1$ левая часть равна 8, а правая уже 10, а в дальнейшем правая часть растёт быстрее, чем левая, так что продолжает доминировать и дальше. Как видно, для получения этой оценки мы отбросили все медленные слагаемые и постоянные множители.

При этом для того чтобы получить оценку $T(n)=O(n^2)$, не нужно перед этим получать точную оценку вычислительной сложности в виде $T(n)=3n^2+2n+3$. Вместо этого можно просто посмотреть на листинг 13 и заключить, что узким местом являются два вложенных цикла, причём внешний работает n итераций, а внутренний – в среднем $n/2$ итераций, поэтому вычислительная сложность равна $O(n^2)$. Асимптотическую оценку вычислительной сложности иногда коротко называют *асимптотикой* алгоритма.

Нужно понимать, что для функции $T(n)=3n^2+2n+3$ верным также является утверждение $T(n)=O(n^3)$, поскольку O – это оценка сверху, то есть найдётся константа C, для которой $3n^2+2n+3<Cn^3$. Однако такая оценка является более грубой и не так ценна, как первая. Особенность оценки $T(n)=O(n^2)$ заключается в том, что верно и обратное, то есть $n^2=O(3n^2+2n+3)$. Это очевидно выполняется даже для $C=1$. Для n^3 это уже не верно.

Таким образом, оценка $T(n)=O(n^2)$ наилучшая в том смысле, что она симметрична, то есть $n^2=O(T(n))$. Для обозначения такого типа асимптотических оценок имеется специальная конструкция

$$f(n)=\Theta(g(n))\Leftrightarrow f(n)=O(g(n))\wedge g(n)=O(f(n)).$$

Это означает, что функция $f(n)$ асимптотически ограничена функцией $g(n)$ не только сверху, но и снизу. Обычно когда говорят об асимптотических оценках вычислительной сложности алгоритмов, имеют в виду именно такие точные оценки, хотя пишут всё равно $f(n)=O(g(n))$, подразумевая, что это не просто оценка сверху, а наилучшая из таких оценок. Однако более точным было бы написать для алгоритма из листинга 13 $T(n)=\Theta(n^2)$.

Основные свойства асимптотических оценок $f(n)=O(g(n))$ следующие.

1. *Рефлексивность*:

$$\forall f(n): f(n)=O(f(n)).$$

2. *Транзитивность*:

$$\forall f(n),g(n),h(n): f(n)=O(g(n)) \wedge g(n)=O(h(n)) \Rightarrow f(n)=O(h(n)).$$

Оценка $f(n)=\Theta(g(n))$ также обладает симметричностью:

$$\forall f(n),g(n): f(n)=\Theta(g(n)) \Rightarrow g(n)=\Theta(f(n)).$$

Доказательства этих свойств выглядят в высшей мере очевидными.

Кроме асимптотической оценки вычислительной сложности алгоритма, можно также провести асимптотическую оценку потребляемой памяти. Для этого нужно оценить, как количество памяти в битах, выделенное программной реализации алгоритма, зависит от входных данных. Для листинга 13 асимптотическая оценка потребляемой памяти составляет $M(n)=O(1)$, поскольку всегда используются только три целочисленных переменных. Правда, нужно отметить, что при этом мы не считали размер вывода, который может накапливаться в буфере в оперативной памяти. Позже будут рассмотрены алгоритмы, которые выделяют разное количество памяти в зависимости от входных данных.

3.4 Примеры анализа вычислительной сложности алгоритмов

Приёмы анализа вычислительной сложности алгоритма будут активно использоваться в дальнейшем повествовании. Следует понимать, какие вычислительные сложности являются хорошими, а какие плохими. В таблице 9 в порядке возрастания сложности приведены некоторые наиболее характерные асимптотики алгоритмов. Самая плохая из них – экспоненциальная. Если алгоритм имеет вычислительную сложность $O\left(2^n\right)$, то при увеличении входных данных n на единицу время работы алгоритма возрастает в два раза.

Таблица 9 – Примеры вычислительных сложностей алгоритмов

№	Вычислительная сложность	Название
1	$O(1)$	Константная
2	$O(\log n)$	Логарифмическая
3	$O(n)$	Линейная
4	$O(n\log n)$	Полилогарифмическая
5	$O\left(n^2\right)$	Квадратичная
6	$O\left(n^3\right)$	Кубическая
7	$O\left(2^n\right)$	Экспоненциальная

Все вычислительные сложности, ограниченные сверху многочленами от входных данных, называются *полиномиальными*. В большинстве своём алгоритмы с такими вычислительными сложностями пригодны для практического использования, в отличие от экспоненциальных алгоритмов. Чтобы классифицировать задачи по вычислительной сложности, их делят на классы сложности.

Пусть имеется задача $f(x): X \to Y$, где X – это множество входных данных, Y – это множество выходных данных, а сама задача состоит в вычислении функции $f(x)$. К сожалению, сама функция на практике не известна, так что разработка алгоритма, решающего задачу, является отдельной проблемой. Будем говорить, что задача $f(x): X \to Y$ относится к *классу сложности NP*, если существует алгоритм $A(x, y): X \times Y \to \{0;1\}$, который для

заданного набора входных данных x и результата y за полиномиальное время определяет, верно ли, что $f(x)=y$, то есть проверяет, является ли y правильным ответом на задачу для входных данных x. Примером такой задачи является задача поиска целочисленного решения у заданной системы линейных неравенств.

Кроме того, определим *класс сложности P*, как класс задач, для которых существует полиномиальный алгоритм $A(x): X \to Y$, который за полиномиальное время вычисляет правильный ответ на задачу для входных данных x. Примером такой задачи является задача умножения двух матриц.

На самом деле, существуют более формальные определения этих классов задач, которые не приводятся в этой работе. Понятно, что если задача относится к классу сложности P, то она также относится и к классу сложности NP, поскольку алгоритм для её решения можно использовать в алгоритме проверки решения: решить задачу и сравнить ответы. Обратное утверждение уже не столь очевидно. Вопрос о равенстве классов P и NP является одной из ключевых нерешённых проблем в информатике.

Пример 19

Задача. Дано натуральное число n, требуется разработать алгоритм проверки, является ли это число простым, а также написать исходный код его программной реализации. Натуральное число является простым, если имеет ровно два различных натуральных делителя. Оценить вычислительную сложность реализованного алгоритма.

Решение. Понятно, что делители натурального числа n не превышают само число n. Предлагается перебрать все возможные натуральные числа от 1 до n включительно и посчитать, сколько из них являются делителями числа n. После этого можно по определению проверить, является ли число n простым. Пример реализации такого алгоритма в виде функции на языке C++ приведён в листинге 14.

Как нетрудно заметить, вычислительная сложность приведённого алгоритма составляет $O(n)$, потому что тело единственного цикла выполняется n раз. Асимптотическая оценка затраченной памяти составляет $O(1)$, поскольку заводятся только две целочисленные переменные. Программная реализация, приведённая в листинге 14, может с лёгкостью успеть проверить число $n \le 10^7$ на простоту менее чем за секунду на обычном домашнем компьютере, однако для $n > 10^8$ уже нельзя быть настолько уверенным в её производительности, поскольку деление с остатком – довольно медленная операция.

Листинг 14. Наивная проверка числа на простоту

```
bool is_prime(int n)
{
    int count = 0;
    for (int i = 1; i <= n; ++i)
    {
        if (n % i == 0)
        {
            ++count;
        }
    }
    return count == 2;
}
```

Можно ли построить более эффективный алгоритм решения этой задачи? Для этого нужно задуматься, не выполняет ли приведённый алгоритм лишней работы, которую можно не делать. Первое наблюдение состоит в том, что любое число делится на само себя и на единицу. Проверять это не имеет смысла: важно, есть ли другие делители. Единственное исключение – единица. Она делится только на саму себя и на единицу, но при этом имеет всего один делитель, так что не является простым числом.

Второе замечание – натуральное число k не может являться делителем натурального числа n, если

$$\frac{n}{2} < k < n,$$

потому что в таком случае результат деления

$$1 < \frac{n}{k} < 2,$$

тогда как должен быть целым. Конечно, можно в листинге 14 изменить условие входа в цикл с `i <= n` на `(i << 1) <= n`, тогда тело цикла будет выполняться в два раза меньшее количество раз, но вычислительная сложность такого алгоритма по-прежнему будет составлять $O(n)$, так что существенного выигрыша в производительности такое улучшение не даёт.

Можно пойти дальше и заметить, что если у натурального числа n есть делитель $k > \sqrt{n}$, то у него есть и другой делитель $m < \sqrt{n}$, причём $km = n$. Этот факт довольно очевиден: m – это просто результат деления n на k, и если он больше $\sqrt{n}$, как и k, то $km > \sqrt{n} \cdot \sqrt{n} = n$, что сразу же даёт противоречие. Это означает, что нет необходимости проверять, являются ли делителями натурального числа n числа $k > \sqrt{n}$, если ранее уже выяснилось, что у числа n нет делителей среди натуральных чисел $2 \le k \le \sqrt{n}$.

Пример реализации такого алгоритма в виде функции на языке C++ приведён в листинге 15. Сразу стоит обратить внимание на отличие, заключающееся в отдельной проверке единицы, как тривиального случая, для которого основной алгоритм сработал бы неправильно. Кроме того, условие входа в цикл сформулировано в виде $i^2 \le n$, что как раз и означает $i \le \sqrt{n}$, но вычисляется без погрешности в целых числах.

Самое главное, что вычислительная сложность этого алгоритма составляет $O\left(\sqrt{n}\right)$, поскольку тело цикла выполняется не более $\sqrt{n}$ раз. Этот алгоритм существенно эффективнее предыдущего. Его программная реализация на обычном домашнем компьютере способна проверить на простоту любое положительное число типа `int` менее чем за секунду, если считать, что это 32-битный целочисленный тип. Более того, она способна менее чем за

секунду проверить число $n \le 10^{14}$, если тип данных заменить на 64-битный, поскольку $\sqrt{10^{14}} = 10^7$.

Листинг 15. Ускоренная проверка числа на простоту

```
bool is_prime(int n)
{
    if (n == 1)
    {
        return false;
    }
    for (int i = 2; i * i <= n; ++i)
    {
        if (n % i == 0)
        {
            return false;
        }
    }
    return true;
}
```

Существуют ли ещё более эффективные алгоритмы решения этой задачи? На самом деле, да, существуют. Они называются *тестами простоты*, некоторые из них вероятностные, некоторые предназначены только для чисел определённого вида, но так или иначе по большей части они довольно сложны, используют непростую математическую основу и их описание выходит за рамки этого курса. Тесты простоты активно используются на практике в криптографии в различных алгоритмах шифрования.

Напоследок приведём несколько простых *правил сравнения вычислительной сложности* алгоритмов.

1. Любой многочлен растёт асимптотически быстрее любого логарифма в любой степени:

   $$\forall a > 1 \, \forall b > 0 \, \forall c > 0 : \left(\log_a n\right)^b = O\left(n^c\right).$$

2. Любая показательная функция с основанием большим единицы растёт асимптотически быстрее любого многочлена:

   $$\forall a > 0 \, \forall b > 1 : n^a = O\left(b^n\right).$$

3. Все логарифмы растут асимптотически одинаково, то есть

$$\forall a>1\,\forall b>1: \log_a n = \Theta\left(\log_b n\right),$$

поэтому иногда под знаком O основание логарифма не пишут, например, $T(n)=O(\log n)$.

4. Из двух показательных функций быстрее растёт та, у которой основание больше:

$$\forall a>1\,\forall b>1: a<b \Rightarrow a^n = O\left(b^n\right).$$

Упражнения для самостоятельной работы

1. Имеется три стержня, на первый из которых нанизаны n дисков разного размера, так чтобы среди любых двух дисков размер нижнего больше размера верхнего. За одно действие можно снять верхний диск с любого стержня и переложить его на любой другой, если на последнем вообще нет стержней, либо если все диски на этом стержне большего размера, чем перекладываемый. Требуется переместить все диски на третий стержень. Постройте алгоритм решения этой задачи. Оцените его вычислительную сложность.

2. У вас имеется алгоритм с точной вычислительной сложностью $T(n)$. Приведите достаточно наглядную асимптотическую оценку его вычислительной сложности.

а) $T(n)=3n^2+2n+1$;

б) $T(n)=3+2n+n^2$;

в) $T(n)=12345$;

г) $T(n)=1+2^n$;

д) $T(n)=1+\left(\frac{1}{2}\right)^n$;

е) $T(n)=\frac{n^2+n+1}{n+2}$.

3. Пусть имеются два алгоритма решения одной и той же задачи. Первый имеет вычислительную сложность $f(n)$, а второй – $g(n)$. Какой из них лучше подходит для больших значений n?

а) $f(n)=2n+100,\ g(n)=100500;$

б) $f(n)=n^2+1000,\ g(n)=1000n+1;$

в) $f(n)=\sqrt{n},\ g(n)=(\log_2 n)^2;$

г) $f(n)=n\sqrt{n},\ g(n)=\left(\sqrt[3]{n}\right)^5;$

д) $f(n)=n\log_2 n,\ g(n)=\log_2 n^5;$

е) $f(n)=n^{1,1},\ g(n)=\sqrt{n}(\log n)^5;$

ж) $f(n)=2^{2n},\ g(n)=3^n;$

з) $f(n)=\sqrt{3^n},\ g(n)=2^n;$

и) $f(n)=2^n\sqrt{n},\ g(n)=3^n\log_2 n;$

к) $f(n)=n2^n,\ g(n)=n^n;$

л) $f(n)=n^2 2^n,\ g(n)=n!;$

м) $f(n)=(n!)^2,\ g(n)=n^n;$

н) $f(n)=\log_2\log_2 n,\ g(n)=\dfrac{n}{(\log_2 n)^3};$

о) $f(n)=n,\ g(n)=(\log_2 n)^{\log_2 n}.$

4. Докажите свойства асимптотических оценок из раздела 3.3.

5. Докажите правила сравнения вычислительной сложности алгоритмов из раздела 3.4.

6. Докажите, что $\max\{n,m\}=\Theta(n+m).$

4 Массив – фундаментальная структура данных

4.1 Понятие массива

Прежде чем вести разговор о массивах, следует в очередной раз остановиться на терминологии. Следует различать понятия «тип данных» и «структура данных». *Тип данных*, как уже было сказано ранее, – это множество возможных значений и действий с ними. *Структура данных* – это способ хранения данных в памяти и организации доступа к ним. Таким образом, тип данных – это более абстрактная и в то же время более мелкая единица классификации данных. Он определяет только значения и операции с ними, не определяя конкретный способ их хранения и доступа к ним.

Массив – это набор занумерованных элементов одного типа, расположенных в памяти друг за другом. В некоторых случаях для решения задачи требуется завести много переменных, их число даже может зависеть от входных данных. В таких ситуациях и используются массивы. По сути это способ объединить заданное количество переменных одного типа в одну.

Как уже упоминалось в предыдущих разделах, с точки зрения программиста оперативная память выглядит, как множество занумерованных ячеек памяти размером в один байт. Номера этих ячеек называются их адресами. С некоторыми оговорками можно считать, что ячейки с соседними адресами физически тоже находятся рядом, так что получить доступ к ячейкам с последовательными адресами чуть быстрее, чем к ячейкам, физически расположенным далеко друг от друга.

Не смотря на это, можно считать, что оперативная память является памятью *с произвольным доступом* (англ. *Random Access*), то есть по адресу ячейки памяти можно мгновенно получить доступ к содержимому этой ячейки. Элементы массива хранятся в памяти последовательно друг за другом, причём каждый элемент сам по себе занимает несколько байт в зависимости от типа данных элементов массива. Поэтому если известен адрес первого элемента

массива a и размер каждого элемента t, то можно вычислить адрес k-го элемента массива как $a_k = a + kt$, считая, что для первого элемента $k = 0$. В связи с этим элементы массива нумеруются с нуля.

Отсюда вытекает главная особенность массива – возможность получать доступ к элементу по его номеру за $O(1)$, как если бы это действительно были отдельные переменные. Однажды задав размер массива, можно быстро получать значение элемента с конкретным номером, а также изменять это значение на другое. Эта особенность массивов используется для построения других более сложных структур данных на основе массивов. Именно поэтому массив считается одной из фундаментальных структур данных: она низкоуровневая и может использоваться для построения более сложных высокоуровневых структур.

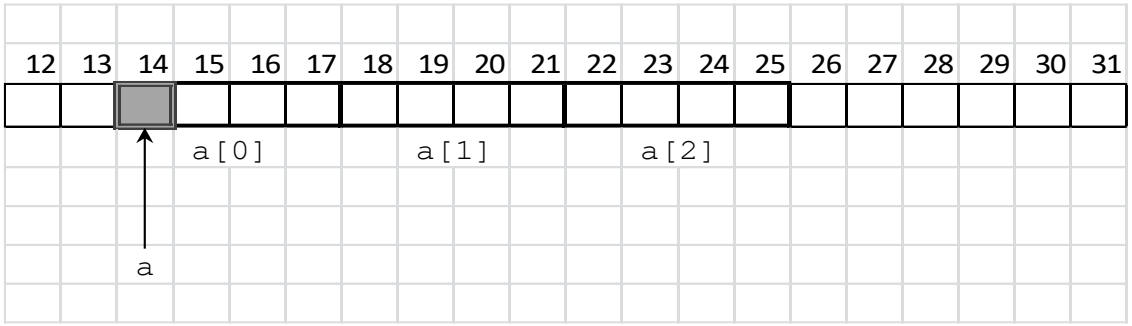

Рисунок 3 – Пример хранения массива в памяти компьютера

На рисунке 3 представлен пример хранения массива `a` из трёх целых чисел типа `int` (32 бита со знаком) в ячейках памяти с адресами с 14 по 25 включительно. Адрес 14 нулевого элемента массива, выделенный серым, будет храниться в переменной `a`. Это, по сути, и есть массив, как таковой. Первый элемент (на самом деле второй, если считать с единицы) хранится в ячейках с адресами 18-21, поскольку $14 + 1 \cdot 4 = 18$, а второй – в ячейках с адресами 22-25, поскольку $14 + 2 \cdot 4 = 22$. Вся остальная память вокруг может быть занята чем угодно другим, в том числе вообще не принадлежать программе, для которой был выделен этот массив.

На практике массивы используются, например, для хранения и обработки аналогов математических векторов и матриц. На их основе строятся структуры данных различной сложности: от линейных списков, стеков и очередей до хэш-таблиц и сбалансированных деревьев поиска. Большинство языков программирования поддерживают различные типы данных для массивов, а большинство вычислительных устройств имеют возможность существенной оптимизации работы с массивами на аппаратном уровне.

4.2 Реализация массивов в языках программирования

За выделение памяти для запущенной программы и под массивы, и под переменные других типов отвечает операционная система. Оперативная память сама по себе также не однородная и делится на сегменты, из которых пока что важно отметить два: статическую память (стек) и динамическую память (кучу). Не вдаваясь в подробности, можно считать, что *статическая память* – это также часть оперативной памяти, доступ к которой оптимизируется операционной системой для ускорения доступа к ней. Под оптимизацией, прежде всего, понимается кэширование на разных уровнях для ускорения последующей загрузки данных в регистры процессора. *Динамической* же памятью можно считать всю остальную часть оперативной памяти, в которой можно заводить переменные. Доступ к ней не оптимизируется так активно.

Хранение данных в статической памяти имеет очевидное преимущество в скорости доступа к этим данным. Недостатком же является особенность, из-за которой программа должна в момент запуска указать операционной системе необходимый ей объём стека и не может изменить его в дальнейшем. Таким образом, компилятор должен на момент компиляции иметь возможность оценить объём памяти, который требуется занять под стек. Если в случае с обычными переменными целого типа или типа с плавающей запятой это не составляет труда, то в случае с массивами приводит к ограничению, заключающемуся в том, что размер массива, хранящегося на стеке, должен

быть задан явно в исходном коде программы, и не должен меняться от запуска к запуску.

Итак, обычные переменные по умолчанию хранятся на стеке. Доступ к ним можно считать быстрым. Массивы же могут храниться на стеке, а могут в динамической памяти, в зависимости от способа их объявления и инициализации. Массивы, которые хранятся в статической памяти, называются *статическими массивами*, а массивы, которые хранятся в динамической памяти, – *динамическими массивами*. Некоторые языки программирования позволяют программисту выбирать различные виды массивов, в то время, как некоторые языки реализуют только один из них. В качестве примера языка с широкими возможностями по работе с различными типами массивов предлагается разобрать некоторые примеры исходного кода на языках C и C++.

В языке C статические массивы объявляются с помощью синтаксической конструкции вида

```
тип идентификатор[размер];,
```

которая создаёт переменную `идентификатор`, являющуюся массивом из `размер` элементов типа `тип`, например, инструкция `int a[100];` создаёт массив `a` из ста целых чисел. При этом размером может быть литерал, либо константа. Нельзя просто считать размер массива и создать статический массив такого размера.

Доступ к элементам статических и любых других массивов в языке C осуществляется путём написания индекса элемента в квадратных скобках после названия переменной. Например, `a[k]` – это `k`-ый элемент массива `a`. С элементами массива можно работать так же, как с обычными переменными такого же типа: использовать их в выражениях, присваивать им значения и т. д.

Важной особенностью языка C является отсутствие контроля выхода за границы массива. Например, если обратиться к элементу `a[200]`, при том, что в массиве `a` всего сто элементов, то произойдёт обращение к участку памяти, который в действительности лежит за пределами этого массива. В этом участке

памяти может лежать какой-то мусор или даже значение любой другой переменной. В этом случае в выражение просто будет передано это значение. Кроме того, этот участок памяти может принадлежать другой программе. В этом случае по умолчанию операционная система остановит выполнение программы, которая пытается получить доступ к не принадлежащим ей данным.

Следует помнить, что в массиве из n элементов, которые занумерованы с нуля, последний элемент имеет номер $(n-1)$. Обращение к n-му элементу такого массива будет являться выходом за его границы, что является одной из грубейших ошибок при программировании на языках C и C++, поскольку поведение программы в этом случае не определено. Нужно изо всех сил стараться избегать таких ошибок.

Листинг 16. Ввод и вывод статического массива в C

```c
#include <stdio.h>

#define N 10

int main()
{
    int a[N];
    for (int i = 0; i < N; ++i)
    {
        scanf("%d", &a[i]);
    }
    for (int i = 0; i < N; ++i)
    {
        printf("%d\n", a[i]);
    }
    return 0;
}
```

В действительности переменная, обозначающая массив, в языке C является обычным указателем на значение типа, который имеют элементы этого массива. Она хранит адрес нулевого элемента массива. Запись вида `a[k]` в этом смысле эквивалентна записи `*(a + k)`, то есть указатель `a` сдвигается вправо `k` раз на количество байт, которое занимает его тип, после чего

разыменовывается. Более того, в силу коммутативности сложения эта запись также эквивалентна записи `k[a]`, хотя последняя выглядит гораздо менее наглядно и поэтому не следует её использовать, если только нет желания запутать исходный код.

В листинге 16 приведён пример считывания и вывода статического массива из десяти целых чисел в языке C. Учитывая, что с массивом можно работать, как с указателем, вместо `&a[i]` при считывании можно передавать просто `(a + i)`. Обратите внимание, что массив состоит из 10 элементов, что известно до компиляции, и в циклах номера элементов меняются с 0 до 9 включительно.

В языке C нет как таковой реализации динамических массивов в виде отдельного типа данных, однако выделение динамической памяти есть. Базовый способ выделения памяти основан на использовании функции

```
void* malloc(size_t size);,
```

которая принимает единственный аргумент – размер участка динамической памяти байтах, и возвращает нетипизированный указатель на первый байт выделенного участка памяти. Поскольку работа с массивами в этом языке не отличается от работы с указателями, эту функцию можно использовать для выделения массива. Например, чтобы выделить массив из n целых чисел, можно использовать конструкцию

```
int* a = (int*)malloc(n * sizeof(int));.
```

Вначале объявляется переменная `a`, представляющая собой указатель на целое число. В функцию `malloc` передаётся необходимое количество байт, равное произведению числа элементов массива на размер одного такого элемента. Результат работы функции следует явно привести к указателю на целое число, поскольку формально он не типизированный. Так можно выделить динамический массив заданного размера, а не только константного, как в случае со статическими массивами.

Теперь необходимо поговорить об освобождении памяти. Всякая выделенная память в программе должна быть освобождена после использования, чтобы другие приложения смогли её использовать. В случае обычного объявления переменных и статических массивов выделенная память освобождается автоматически, когда соответствующая переменная покидает область своего действия, так что программисту можно не заботиться об этом. Если же память выделяется явно с помощью функции `malloc`, то она должна быть так же явно освобождена с помощью функции `free`. Эта функция принимает указатель на первый байт памяти и освобождает эту память.

Две типичные ошибки при работе с памятью – утечка памяти и висячие указатели. *Утечка памяти* – это ситуация, когда выделенная динамическая память не освобождается явно, указатель на неё выходит за свою область действия, и возможность освободить эту память безвозвратно теряется. В этом случае программа в процессе работы потребляет всё больше и больше памяти со временем, что приводит к необходимости её закрытия после длительной работы. Память, выделенная программой, указатель на которую уже потерялся, так что полезная работа с этой памятью стала невозможной, называется *мусором*.

Висячий указатель – это указатель на область памяти, не принадлежащей программе. Он может появиться в результате выхода за границы области памяти при вычислении адреса, а также после освобождения памяти по этому адресу. Разыменование такого указателя приводит к неопределённым последствиям, вплоть до завершения программы с ошибкой, так что следует старательно его избегать. Также запрещено разыменовывать нулевые указатели, то есть указатели, хранящие в качестве адреса число ноль.

В листинге 17 приведён пример считывания и вывода динамического массива на языке C. Обратите внимание, размер массива также читается из стандартного потока ввода, после чего создаётся массив такого размера. В

остальном работа с таким массивом аналогична работе со статическими массивами. В конце выделенная память явным образом освобождается.

Листинг 17. Ввод и вывод динамического массива в C

```c
#include <stdio.h>

int main() {
    int n;
    scanf("%d", &n);
    int* a = (int*)malloc(n * sizeof(int));
    for (int i = 0; i < n; ++i)
    {
        scanf("%d", &a[i]);
    }
    for (int i = 0; i < n; ++i)
    {
        printf("%d\n", a[i]);
    }
    free(a);
    return 0;
}
```

Конечно, можно заявить, что освобождение памяти в этой программе не столь необходимо, ведь после этого программа сразу же завершается и вся память, занятая под неё, освобождается естественным образом. Тем не менее, настоятельно рекомендуется всегда следить, чтобы память, однажды выделенная вручную, непременно освобождалась. Привычка всегда явно освобождать выделенную память позволяет не допускать утечек памяти при написании реальных приложений на языках с ручным управлением памятью.

Особенность динамических массивов, выделенных с помощью функции `malloc`, состоит в возможности последующего изменения их размера с помощью команды

```
void* realloc (void* ptr, size_t new_size);,
```

которая принимает указатель `ptr` на уже выделенную область памяти, а также новый размер, который она должна иметь, а возвращает новый указатель на область памяти нового размера. В большинстве случаев это тот же самый указатель, но в некоторых случаях, когда размер требуется увеличить, а память

за пределами области уже занята, необходимая память будет выделена в другом месте, содержимое старого блока памяти будет скопировано туда, и ненужный старый блок памяти будет освобождён. Так можно при необходимости менять размер динамических массивов в ходе программы, но следует делать это с осторожностью, поскольку увеличение размера массива из n элементов в случае необходимости копировать блок памяти может занимать $O(n)$ времени.

В языке C++ указанные выше два способа работы с массивами также работают, поскольку язык C++, по сути, является расширением языка C. Кроме того, в этом языке существует другой способ создания динамических массивов с помощью синтаксической конструкции

`new тип[размер];`,

которая возвращает указатель на вновь созданный массив из `размер` элементов типа `тип`. Работа с такими массивами аналогична работе с динамическими массивами в языке C, но освобождение памяти под них выполняется с помощью оператора `delete[]`, после которого записывается указатель на такой массив.

Листинг 18. Ввод и вывод динамического массива в C++

```
#include <iostream>

int main()
{
    int n;
    std::cin >> n;
    int* a = new int[n];
    for (int i = 0; i < n; ++i)
    {
        std::cin >> a[i];
    }
    for (int i = 0; i < n; ++i)
    {
        std::cout << a[i] << std::endl;
    }
    delete[] a;
    return 0;
}
```

В листинге 18 приведён пример считывания и записи динамического массива на языке C++. Как видно, такой способ создания динамических массивов выглядит короче и нагляднее, чем в языке C. Недостатком является отсутствие прямой возможности изменять размер такого массива, несмотря на то, что он динамический. Обратите внимание, что в этом случае также следует заботиться об освобождении выделенной памяти.

Кроме того, нельзя не упомянуть, что в библиотеке STL, которая входит в стандартную библиотеку языка C++, в файле `vector` определён шаблонный класс `vector`, агрегирующий в себе динамический массив. В случае хорошего понимания объектно-ориентированного программирования, шаблонов и в частности библиотеки STL рекомендуется использовать именно этот класс вместо непосредственной работы с массивами. Пример работы с этим классом приведён в листинге 19.

Листинг 19. Ввод и вывод массива с использованием STL

```
#include <iostream>
#include <vector>

int main()
{
    int n;
    std::cin >> n;
    std::vector<int> a(n);
    for (int i = 0; i < n; ++i)
    {
        std::cin >> a[i];
    }
    for (int i = 0; i < n; ++i)
    {
        std::cout << a[i] << std::endl;
    }
    return 0;
}
```

Оператор «`std::vector<int> a(n);`» создаёт переменную `a` типа `std::vector<int>`, представляющую собой динамический массив из `n` целых чисел. Освобождать память, занятую под этот массив, нет

необходимости: переменная `a` сама по себе была создана в автоматической памяти. Когда она выйдет за область действия, сработает деструктор класса `vector` и очистит память автоматически. Как видно, в остальном работа с объектами этого класса похожа на работу с массивами.

Преимущества использования класса `vector` перед непосредственной работой с массивами следующие.

1. Нет необходимости ручного контроля памяти, если объект был создан на стеке.
2. STL представляет широкую функциональность для работы с объектами класса `vector`, содержащуюся как в его методах, так и в других модулях библиотеки STL.
3. Хранит размер в самом объекте, так что нет необходимости хранить его отдельно.

4.3 Основы работы с массивами

Из раздела 4.2 уже должно быть ясно, что ввод и вывод массива из n элементов примитивных типов, таких как целые числа, занимает $O(n)$ операций. Кроме того, сам по себе массив из n таких элементов занимает $O(n)$ бит памяти. Тем не менее, одно обращение к элементу такого массива занимает $O(1)$ времени, поскольку адрес этого элемента в памяти может быть вычислен по простой формуле, а сама оперативная память поддерживает произвольный доступ.

Рассмотрим примеры некоторых других операций, которые могут быть выполнены с массивами. Самая простая из них – свёртка массива с какой-нибудь операцией. *Свёртка* массива a из n элементов с операцией $\circ$ в случае $n=1$ представляет собой этот единственный элемент, а в случае $n>1$ это операция $\circ$, выполненная над первым элементом массива и свёрткой массива из оставшихся $(n-1)$ элементов. Не следует путать свёртку массива (англ. *fold* или *reduce*) со свёрткой сигналов (англ. *convolution*).

В листинге 20 приведён исходный код функции на языке C, вычисляющей сумму элементов массива из целых чисел. Сам массив передаётся в функцию в виде указателя на начало. Также необходимо передать количество элементов массива, поскольку сам по себе указатель не хранит такой информации. Этот способ передачи массива в функцию подходит для статических и динамических массивов в C и в C++.

Листинг 20. Сумма элементов массива

```
int sum(int* a, int n)
{
    int ans = 0;
    for (int i = 0; i < n; ++i)
    {
        ans += a[i];
    }
    return ans;
}
```

В случае использования класса `vector` следовало бы передавать его по константной ссылке, например, так: `int sum(const vector<int>& a)`. Передавать размер массива в этом случае не имело бы смысла, поскольку его можно было бы получить как `a.size()`. Передавать вектор по значению без символа `&` неправильно, поскольку приведёт к копированию всего вектора целиком за $O(n)$ операций, которые можно было бы не делать.

Как видно из листинга 20, сумма элементов массива может быть вычислена за $O(n)$ операций и с использованием $O(1)$ дополнительной памяти. То же самое верно для свёртки массива с любой операцией, которая сама по себе выполняется за $O(1)$ времени и требует $O(1)$ дополнительной памяти. Следует также отметить, что сумма элементов типа `int` сама по себе может выходить за границы этого типа, так что, возможно, было бы правильнее возвращать из функции значение типа `long long`.

Как было отмечено ранее, замена значения одного из элементов массива на другое может быть выполнена за $O(1)$. Часто требуется не заменить

значение элемента массива, а вставить элемент между другими элементами или удалить элемент массива со сдвигом остальных элементов. В случае со статическими массивами этого сделать не удастся, поскольку такая операция предполагает изменение общего количества элементов массива, тогда как статические массивы всегда имеют заданное количество элементов. Некоторым решением этой проблемы может служить хранение большого статического массива из такого количества элементов, чтобы их заведомо хватило на все операции вставки и удаления. При этом отдельно нужно хранить количество элементов в таком массиве, которое в данный момент действительно используется. Недостаток такого подхода в использовании лишней памяти.

Также нельзя менять число элементов у динамических массивов в языке C++, созданных с помощью оператора `new`. Однако эти операции могут быть реализованы для динамических массивов в языке C, память под которые была выделена с помощью функции `malloc`. Рассмотрим реализацию операций вставки и удаления для таких массивов.

Листинг 21. Вставка элемента в массив

```
void insert(int** a, int n, int index, int value)
{
    ++n;
    int* b = (int*)realloc(*a, n * sizeof(int));
    for (int i = n - 1; i > index; --i)
    {
        b[i] = b[i - 1];
    }
    b[index] = value;
    *a = b;
}
```

В листинге 21 приведена функция на языке C для вставки элемента `value` в массив на позицию `index`. Она работает только для массивов, память под которые была выделена функциями `malloc`, `calloc` или `realloc`. Массив передаётся даже не по указателю, а по указателю на указатель, поскольку указатель на начало массива сам по себе может измениться после выполнения функции `realloc`. После изменения реального размера массива

элементы сдвигаются на один к концу массива, начиная с последнего и заканчивая элементом на позиции `index`. Наконец, на позицию `index` устанавливается элемент `value`, и новый указатель на массив устанавливается вместо старого.

Вычислительная сложность алгоритма вставки элемента в массив составляет $O(n)$, поскольку хотя бы даже функция realloc может выполняться $O(n)$ времени, если массив будет копироваться в другую область памяти. Но даже если исключить такую возможность, всё равно, усредняя вычислительную сложность по позиции `index`, мы получим оценку $O(n)$, поскольку в среднем будет выполняться операция вставки в середину массива за $(n/2)$ операций сдвига.

Вставка элемента в объект класса `vector` может осуществляться через встроенный метод `insert`, принимающий итератор на позицию, перед которой следует вставлять элемент, и сам элемент. Вычислительная сложность такой вставки также составляет $O(n)$. Однако нужно отметить, что у класса `vector` существует также метод `push_back`, который вставляет элемент в конец массива. В силу особенностей реализации этот метод работает за время $O(1)$, хотя выделение памяти всё равно может произойти. Это достигается за счёт увеличения реального размера выделенного участка памяти не на один элемент каждый раз, а в два раза при необходимости.

Листинг 22. Удаление элемента из массива

```
void erase(int** a, int n, int index)
{
    --n;
    int* b = *a;
    for (int i = index; i < n; ++i)
    {
        b[i] = b[i + 1];
    }
    *a = (int*)realloc(b, n * sizeof(int));
}
```

В листинге 22 приведён исходный код функции на языке C для удаления элемента с номером `index` из массива. Массив, как и в случае вставки элемента, передаётся по указателю на указатель, поскольку сам указатель на начало массива потенциально может измениться после попытки изменить его размер. Сам алгоритм состоит в сдвиге элементов на один к началу массива, начиная с элемента с номером `index`, и далее до конца массива. В конце реальный размер массива уменьшается на единицу.

Нетрудно заметить, что вычислительная сложность алгоритма удаления элемента составляет $O(n)$. Это связано даже не с функцией `realloc`, которая предположительно должна работать за $O(1)$, поскольку размер необходимого блока памяти лишь уменьшается, а со сдвигом элементов. Если усреднить вычислительную сложность по всем значениям `index`, то получится, что в среднем будет выполняться $(n/2)$ операций сдвига, что и приводит к такой оценке вычислительной сложности. Правда, можно надеяться, что удаление последнего элемента всегда будет протекать за $O(1)$, хотя никто и не гарантирует, что функция `realloc` не станет заново выделять память и копировать все элементы.

Класс вектор опять же поддерживает эту операцию в виде метода `erase`, принимающего итератор на элемент, который следует удалить. Этот метод также работает за время $O(n)$. Кроме того, у класса `vector` существует метод `pop_back`, который удаляет последний элемент. Этот метод работает за $O(1)$, поскольку в большинстве случаев вообще не выполняет никаких операций, кроме уменьшения переменной, отвечающей за количество элементов массива.

Нужно отметить, что в приведённых в этом разделе листингах в функциях нет проверки на корректность аргументов. Если передать значения индексов, выходящие за границы массива, или висячий указатель, то функция будет работать неправильно. В данном случае, ответственность за передачу корректных аргументов ложится на программиста, использующего функцию. В

реальных приложениях хорошим решением является выполнение такой проверки внутри функции, чтобы, по крайней мере, устранить неопределённое поведение функции в случае передачи некорректных аргументов.

4.4 Поиск в массиве

Задача поиска в широком смысле является одной из глобальных задач науки. Она состоит в поиске элемента x из некоторого множества X, которое удовлетворяет некоторому свойству, заданному предикатом $P(x): X \to \{0;1\}$. Для различных частных случаев эта задача может решаться самыми разными способами.

В нашем случае, под *поиском элемента* x *в массиве* a из n элементов понимается поиск индекса $k \in [0;n-1] \cap \mathbf{Z}$, такого что $a[k] = x$. Элемент x может вообще отсутствовать в массиве, и в этом случае требуется сообщить об этом некоторым образом. На практике эта задача возникает довольно часто.

Решение этой задачи не представляет сложности. Нужно перебрать все элементы массива с начала до конца и для каждого из них проверить, не равен ли он x. Как только будет найден нужный элемент, можно заканчивать выполнение алгоритма и сообщать ответ. Если все элементы проверены, и ни один из них не равен x, то следует сообщить об этом. Этот алгоритм называется *линейным поиском*.

В листинге 23 приведён пример исходного кода на языке C, реализующего функцию линейного поиска в массиве. Массив передаётся по указателю, после него передаётся количество элементов в этом массиве и элемент, который требуется найти. Такой способ передачи, как ранее уже было отмечено, подходит для статических и динамических массивов в C и в C++. В цикле проверяются все элементы массива, и возвращается первый попавшийся элемент, совпадающий с `x`. В случае отсутствия в массиве такого элемента, вместо индекса возвращается `-1`. Это легально, поскольку такого индекса в массиве быть не может.

Листинг 23. Линейный поиск в массиве

```
int find(int* a, int n, int x)
{
    for (int i = 0; i < n; ++i)
    {
        if (a[i] == x)
        {
            return i;
        }
    }
    return -1;
}
```

Вычислительная сложность алгоритма линейного поиска, как легко заметить, составляет $O(n)$, поскольку даже если элемент присутствует в массиве, он может с равной возможностью находиться на любой позиции, так что алгоритм сделает в среднем $(n/2)$ итераций. Тем более, если быть чуть более пессимистичным, то можно предположить, что в большинстве случаев выполняется поиск элемента, которого вообще нет в массиве, так что массив каждый раз будет просматриваться целиком. Существенно более эффективного алгоритма решения задачи поиска в обычном массиве не существует.

В случае использования класса `vector` для поиска элемента в объекте такого класса можно использовать встроенную функцию `find`, объявленную в файле `algorithm`. Она принимает итераторы на начало и конец вектора, а также значение, которое требуется отыскать, а возвращает итератор на искомый элемент. В случае отсутствия искомого элемента, функция возвращает итератор на конец, который был ей передан.

Встроенную функцию можно использовать и для обычных массивов, поскольку сами указатели являются итераторами в некотором смысле, но это помогает не во всех ситуациях. Иногда требуется искать не конкретный элемент, а элемент, отвечающий более сложному свойству. Как правило, более эффективного алгоритма, чем просто проверить все элементы, всё равно

придумать не удастся, однако в некоторых случаях потребуются более сложные проверки.

Рассмотрим задачу поиска максимального элемента массива. Элемент x называется максимальным элементом массива a из n элементов, если все остальные элементы не превышают его, то есть $\forall k \in [0;n-1] \cap \mathbf{Z} : a[k] \le x$. Пусть задан массив целых чисел, и требуется найти в нём значение максимального элемента.

В листинге 24 представлена функция на языке C для поиска максимального элемента в массиве. Изначально максимальным элементом полагается первый элемент, а затем все остальные элементы сравниваются с текущим максимальным элементом, и если некоторый элемент больше текущего максимального, то текущим максимальным элементом полагается именно он. После просмотра всех элементов текущий максимальный элемент и является ответом.

Почему такой алгоритм вообще работает корректно? Дело в том, что если текущий элемент сравнивается с настоящим глобальным максимумом в массиве, то после этого текущий максимум обязательно станет равным ему, и после этого уже не поменяется, потому что все остальные элементы не могут превышать максимальный элемент. Конечно, такой алгоритм подходит только для непустых массивов. Поиск максимума в пустом массиве лишён смысла.

Листинг 24. Поиск максимума в массиве

```
int max(int* a, int n)
{
    int ans = a[0];
    for (int i = 1; i < n; ++i)
    {
        if (a[i] > ans)
        {
            ans = a[i];
        }
    }
    return ans;
}
```

Вычислительная сложность алгоритма поиска максимума в массиве составляет $O(n)$, причём этот алгоритм всегда выполняет одинаковое количество итераций, проверяя все элементы массива, в отличие от алгоритма линейного поиска из листинга 23. Иногда вместо первого элемента изначально текущий максимум инициализируют некоторым маленьким числом, меньшим, чем любой из элементов массива, но выбор этого числа представляет собой отдельную задачу, так что рекомендуется использовать элемент массива. Также стоит отметить, что алгоритм использует $O(1)$ дополнительной памяти.

4.5 Многомерные массивы

Рассмотренные ранее массивы были предназначены для хранения элементов одного типа последовательно друг за другом. Структуры данных, которые позволяют хранить данные в таком виде, называются *линейными структурами данных*. В частности, рассмотренные массивы называются линейными массивами или *одномерными массивами*.

В некоторых случаях требуется хранить не последовательность элементов, а таблицу из элементов, многослойную таблицу, а также более сложно организованные структуры. Для этого могут использоваться многомерные массивы. Строго говоря, m-мерный массив элементов из множества X – это функция $a(i_1, i_2, \ldots, i_m): D_1 \times D_2 \times \ldots \times D_m \to X$, где $D_i = [0; n_i - 1] \cap \mathbf{Z}$. Иными словами, элементы этого массива имеют несколько индексов.

Можно представить себе многомерный массив как массив массивов. Например, массив, элементами которого являются массивы целых чисел, – это двумерный массив. Иногда двумерный массив называют *матрицей* по аналогии с аналогичным математическим объектом. Массив, элементами которого являются матрицы, – это трёхмерный массив. Массив, элементами которого являются m-мерные массивы, представляет собой $(m+1)$-мерный массив.

На практике в различных языках программирования существует несколько способов работы с многомерными массивами. Всегда есть вариант, когда такой массив на самом деле является массивом из массивов более мелкой размерности. При этом если эти массивы являются указателями, то, по сути, создаётся одномерный массив указателей на другие массивы меньшей размерности, которые могут храниться в памяти как угодно. Однако более эффективно хранить элементы многомерного массива рядом, как если бы это был длинный одномерный массив. Предлагается рассмотреть несколько примеров создания двумерного массива из $n \times m$ элементов на языках C и C++, заполненных нулями.

В листинге 25 приведён исходный код на языке C, в котором создаётся и заполняется нулями двумерный статический массив. В этом случае все $n \times m$ его элементов действительно располагаются в памяти последовательно друг за другом, так что два цикла, заполняющие этот массив нулями, будут работать очень быстро: они последовательно заполняют одинаковыми значениями блок памяти на стеке. Ещё раз следует напомнить, что `n` и `m` в этом случае должны быть константами, известными на момент компиляции, поскольку массив статический.

Листинг 25. Статический двумерный массив

```
int a[n][m];
for (int i = 0; i < n; ++i)
{
    for (int j = 0; j < m; ++j)
    {
        a[i][j] = 0;
    }
}
```

Вычислительная сложность создания такого массива и заполнения его нулями, тем не менее, составляет $O(nm)$. Это связано с тем, что для заполнения такого массива нулями используются два вложенных цикла. В программировании обычно первый индекс матрицы обозначается `i` и изменяется от `0` до `n-1`, а второй индекс называется `j` и изменяется от `0` до

m-1. Также стоит заметить, что освобождать память, занятую под такой массив, не требуется, поскольку он создан на стеке. Как только переменная a выйдет за область действия, память будет освобождена автоматически.

Листинг 26. Динамический двумерный массив в C

```
int** a = (int**)malloc(n * sizeof(int*));
for (int i = 0; i < n; ++i)
{
    a[i] = (int*)malloc(m * sizeof(int));
}
for (int i = 0; i < n; ++i)
{
    for (int j = 0; j < m; ++j)
    {
        a[i][j] = 0;
    }
}
```

В листинге 26 представлен аналогичный исходный код создания и заполнения нулями динамического массива на языке C. Он представляет собой массив указателей, то есть в реальности массивы, являющиеся его элементами, не хранятся друг за другом, а могут лежать в памяти разрозненно, поэтому работа с таким массивом чуть медленнее. Зато можно задавать такой массив ровно нужного не константного размера, чтобы не тратить память попусту.

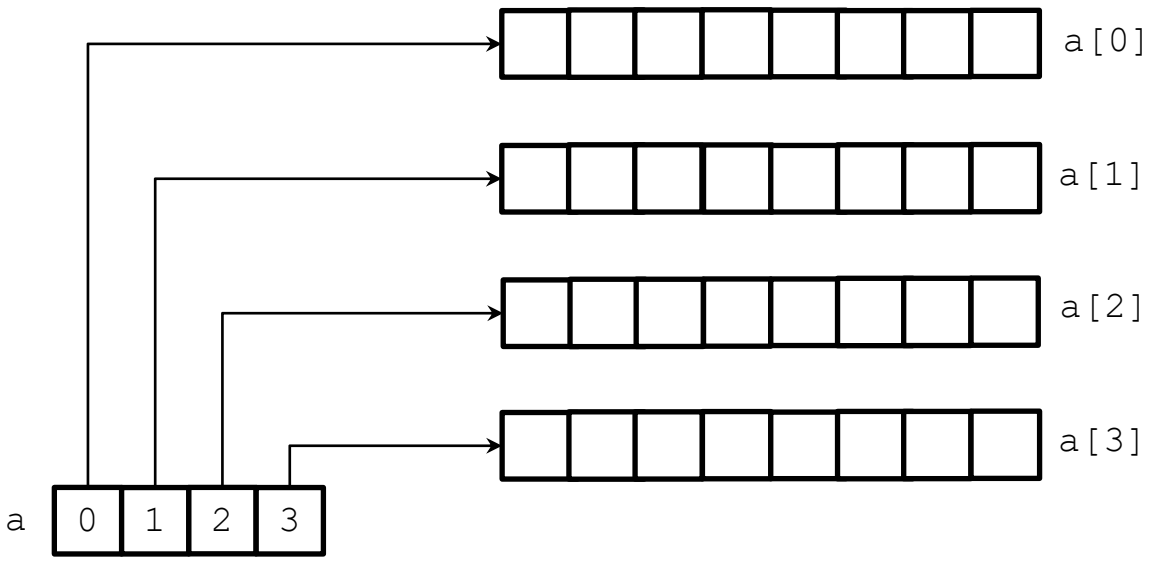

Рисунок 4 – Хранение массива массивов в памяти

Здесь после выделения памяти под массив из n указателей, в каждый его элемент проставляется указатель на только выделенный массив из m целых чисел. Наглядная схема хранения массива из четырёх указателей на массивы в памяти компьютера представлена на рисунке 4. Следует отметить, что отдельные внутренние одномерные массивы в этом случае могут иметь произвольный размер. Также имеет смысл упомянуть, что в данном случае вместо функции malloc более эффективно было бы использовать функцию calloc, которая отдельно принимает количество элементов и размер каждого элемента, и заполняет выделенную память нулями автоматически.

Листинг 27. Освобождение памяти под динамический массив в C

```
for (int i = 0; i < n; ++i)
{
    free(a[i]);
}
free(a);
```

Нельзя не заметить, что освобождать память, выделенную таким образом, необходимо с особой осторожностью. Нужно пройти по массиву указателей и сначала освободить память под все одномерные массивы, после чего освободить память непосредственно под массив указателей. Пример операций на языке C, освобождающих память под двумерный массив, выделенный в листинге 26, приведён в листинге 27.

Листинг 28. Динамический двумерный массив в C++

```
int** a = new int*[n];
for (int i = 0; i < n; ++i)
{
    a[i] = new int[m];
}
for (int i = 0; i < n; ++i)
{
    for (int j = 0; j < m; ++j)
    {
        a[i][j] = 0;
    }
}
```

В листинге 28 приведён пример создания и заполнения двумерного динамического массива на языке C++. Использование оператора new позволяет сделать выделение динамической памяти значительно более наглядным, чем в языке C, но созданные таким образом массивы не могут изменять свой размер в процессе работы программы. В остальном отличий практически не наблюдается. Массивы снова располагаются в памяти, как показано на рисунке 4.

Выделенную таким образом память нужно не забывать освобождать. Для этого опять же применяется похожая процедура, но использующая оператор `delete[]`. Пример исходного кода на языке C++, освобождающего память таким образом, приведён в листинге 29.

Листинг 29. Освобождение памяти под динамический массив в C++

```
for (int i = 0; i < n; ++i)
{
    delete[] a[i];
}
delete[] a;
```

Также в C++ можно создать такой двумерный динамический массив, заполненный нулями, используя класс `vector`, с помощью всего лишь одной инструкции:

```
std::vector< std::vector<int> > a(n, std::vector<int>(m, 0)).
```

Это возможно, поскольку класс vector имеет конструктор, принимающий количество элементов и значение, которым следует инициализировать каждый элемент. В данном случае, во внешний конструктор передаётся вектор из `m` нулей, который копируется в каждый из элементов внешнего вектора векторов. Конечно, создание такого двумерного массива по-прежнему занимает $O(nm)$ операций.

Упражнения для самостоятельной работы

1. Напишите исходный код программы, принимающей на вход конечную последовательность из n целых чисел $\{a_i\}_{i=1}^{n}$ $(1 \le n \le 10^5, -10^9 \le a_i \le +10^9)$, и

выводящей максимальный по модулю элемент этой последовательности. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

2. Напишите исходный код программы, принимающей на вход конечную последовательность из n целых чисел $\{a_i\}_{i=1}^{n}$ $(1 \le n \le 10^5, -10^9 \le a_i \le +10^9)$, и выводящей максимальный и минимальный элементы этой последовательности. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

3. Напишите исходный код программы, принимающей на вход конечную последовательность из n целых чисел $\{a_i\}_{i=1}^{n}$ $(1 \le n \le 10^5, -10^9 \le a_i \le +10^9)$, и выводящей номер элемента этой последовательности, у которого надо поменять знак, чтобы сумма всех элементов получившейся последовательности была максимальна. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

4. Напишите исходный код программы, принимающей на вход конечную последовательность из n различных целых чисел $\{a_i\}_{i=1}^{n}$ $(1 \le n \le 10^5$ $1 \le a_i \le n+1)$, и выводящей, какое из натуральных чисел от 1 до $n+1$ отсутствует в этой последовательности. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

5. Напишите исходный код программы, принимающей на вход конечную последовательность из n целых чисел $\{a_i\}_{i=1}^{n}$ $(1 \le n \le 10^5, -10^9 \le a_i \le +10^9)$, и выводящей уникальный элемент этой последовательности. Гарантируется, что количество элементов n нечётное, а каждый элемент, кроме уникального элемента, встречается в последовательности ровно два раза. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

6. Напишите исходный код программы, принимающей на вход конечную последовательность из n целых чисел $\{a_i\}_{i=1}^{n}$ $(1 \le n \le 10^5, -10^9 \le a_i \le +10^9)$, и выводящей те же самые числа, но в обратном порядке. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

7. Напишите исходный код программы, принимающей на вход конечную последовательность из n целых чисел $\{a_i\}_{i=1}^{n}$ $(1 \le n \le 10^5, -10^9 \le a_i \le +10^9)$, и выводящей количество минимальных элементов в этой последовательности. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

8. Напишите исходный код программы, принимающей на вход конечную последовательность из n целых чисел $\{a_i\}_{i=1}^{n}$ $(1 \le n \le 10^5, -10^9 \le a_i \le +10^9)$, и выводящей количество нечётных чисел этой последовательности, которые больше наибольшего из её чётных чисел. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

9. Напишите исходный код программы, принимающей на вход конечную последовательность из n целых чисел $\{a_i\}_{i=1}^{n}$ $(1 \le n \le 10^5, -10^9 \le a_i \le +10^9)$, и выводящей количество чисел этой последовательности, которые больше среднего арифметического всех её элементов. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

10. Напишите исходный код программы, принимающей на вход конечную последовательность из n целых чисел $\{a_i\}_{i=1}^{n}$ $(1 \le n \le 10^5, 1 \le a_i \le 10^9)$, и выводящей количество элементов этой последовательности на которые делится наибольший из её элементов. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

11. Напишите исходный код программы, принимающей на вход конечную последовательность из n целых чисел $\{a_i\}_{i=1}^{n}$ $(2 \le n \le 10^5,$

$-10^9 \le a_i \le +10^9$), и выводящей второй по величине элемент этой последовательности, либо сообщающей, что такой элемент отсутствует. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

12. Напишите исходный код программы, принимающей на вход конечную последовательность из n целых чисел $\{a_i\}_{i=1}^{n}$ ($1 \le n \le 10^5$, $-10^9 \le a_i \le +10^9$), и выводящей номера всех максимальных элементов этой последовательности. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

13. Напишите исходный код программы, принимающей на вход конечную последовательность из n целых чисел $\{a_i\}_{i=1}^{n}$ ($1 \le n \le 10^5$, $-10^9 \le a_i \le +10^9$), и выводящей границы самого длинного участка массива, состоящего из одинаковых элементов. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

14. Напишите исходный код программы, принимающей на вход две конечных последовательности из n целых чисел: $\{a_i\}_{i=1}^{n}$ и $\{b_i\}_{i=1}^{n}$ ($1 \le n \le 10^5$, $-10^9 \le a_i, b_i \le +10^9$), и выводящей последовательность элементов, у которой на каждой позиции стоит максимальный из соответствующих элементов этих двух последовательностей. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

15. Напишите исходный код программы, принимающей на вход матрицу A размером $n \times m$, состоящую из целых чисел ($1 \le n, m \le 10^3$, $-10^9 \le A_{i,j} \le +10^9$), и выводящей номер строки и номер столбца матрицы, в которых находится максимальный элемент. В случае нескольких правильных ответов можно вывести любой из них. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

16. Напишите исходный код программы, принимающей на вход матрицу A размером $n \times m$, состоящую из целых чисел ($1 \le n, m \le 10^3$, $-10^9 \le A_{i,j} \le +10^9$), и выводящей сообщение «YES», если эта матрица симметрична относительно главной диагонали, а в противном случае выводящей сообщение «NO». Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

17. Напишите исходный код программы, принимающей на вход матрицу A размером $n \times m$, состоящую из целых чисел ($1 \le n, m \le 10^3$, $-10^9 \le A_{i,j} \le +10^9$), и выводящей транспонированную матрицу A^T. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

18. Напишите исходный код программы, принимающей на вход матрицу A размером $n \times m$, состоящую из целых чисел ($1 \le n, m \le 10^3$, $-10^9 \le A_{i,j} \le +10^9$), и выводящей сообщение «YES», если матрица не изменится при отражении ни по вертикали, ни по горизонтали, а в противном случае выводящей сообщение «NO». Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

19. Напишите исходный код программы, принимающей на вход матрицу A размером $n \times m$, состоящую из целых чисел ($1 \le n, m \le 10^3$, $-10^9 \le A_{i,j} \le +10^9$), и выводящей количество элементов матрицы, строго больших всех своих соседей по четырёхсвязной области. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

20. Напишите исходный код программы, принимающей на вход матрицу A размером $n \times m$, состоящую из целых чисел ($1 \le n, m \le 10^3$, $-10^9 \le A_{i,j} \le +10^9$), и выводящей количество элементов матрицы, для которых строка и столбец, в которых они расположены, целиком состоят из одинаковых

элементов. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

21. Напишите исходный код программы, принимающей на вход матрицу A размером $n \times m$, состоящую из целых чисел ($1 \le n, m \le 10^3$, $-10^9 \le A_{i,j} \le +10^9$), и выводящей количество элементов матрицы, наименьших в своей строке и наибольших в своём столбце. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

22. Напишите исходный код программы, принимающей на вход матрицу A размером $n \times m$, состоящую из дробных чисел не более чем с тремя знаками после запятой ($1 \le n, m \le 10^3, -10^3 \le A_{i,j} \le +10^3$), и выводящей ранг этой матрицы. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

23. Напишите исходный код программы, принимающей на вход две матрицы A и B размерами $n_1 \times m_1$ и $n_2 \times m_2$ соответственно, состоящие из дробных чисел не более чем с тремя знаками после запятой ($1 \le n_i, m_i \le 10^3$, $-10^3 \le A_{i,j}, B_{i,j} \le +10^3$), и выводящей произведение этих матриц, либо сообщающей, что их произведение не существует. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

5 Рекурсия

5.1 Понятие рекурсии в математике и в программировании

Рекурсия – это очень широкое понятие, использующееся во многих областях науки, прежде всего, в математике и в информатике. В широком смысле оно означает самоподобие, то есть ситуацию, при которой объект является частью самого себя. Рекурсия встречается в творчестве некоторых писателей и художников, в некоторых акронимах, на некоторых изображениях, в числе которых герб Российской Федерации.

В основном рекурсия используется для определения различных объектов. *Рекурсивное определение* – это определение, включающее в себя само определяемое понятие. Например, можно рекурсивно определить понятие человек, как существо, рождённое женщиной. Можно заметить, что чтобы это определение имело смысл, нужно также определить хотя бы одну женщину не рекурсивно.

В математике многие понятия определяются рекурсивно. Примером рекурсивного определения в математике может служить определение натуральных чисел через аксиомы Пеано, приведённое в разделе 1.1. В этом определении натуральными числами считаются единица и каждое число, следующее за натуральным числом. Другими известными примерами рекурсивных определений в математике являются фракталы, цепные дроби, деревья, некоторые комбинаторные объекты и т. д. В теории алгоритмов одной из моделей алгоритма как такового являются так называемые *примитивно рекурсивные функции*, также в некотором смысле определяемые рекурсивно.

Рассмотрим простой классический пример рекурсивного определения факториала. Факториал целого неотрицательного числа n – это единица для $n=0$, либо произведение числа n и факториала числа $(n-1)$ для всех остальных целых неотрицательных чисел. Если обозначить факториал целого

неотрицательного числа n за $n!$, то указанное определение можно переписать в виде

$$n! = \begin{cases} 1, & n = 0; \\ n \cdot (n-1)!, & n \in \mathbf{N}. \end{cases} \tag{7}$$

Это рекурсивное определение, поскольку оно содержит понятие факториала в самом себе. Тем не менее, оно однозначно определяет понятие факториала для всех целых неотрицательных чисел.

Уже можно было заметить, что все осмысленные рекурсивные определения отличает наличие в определении частного случая, определяемого не рекурсивно. Эта часть рекурсивных определений называется *базой рекурсии*. Например, в определении факториала (7) базой рекурсии является частный случай $n = 0$. Если база рекурсии отсутствует, либо задана неправильно, то рекурсивное определение лишается смысла, и часть объектов остаётся без определения.

Иногда для рекурсивных определений существуют эквивалентные им не рекурсивные определения. Например, факториал целого неотрицательного числа n можно определить, как произведение (4) всех натуральных чисел, не превышающих n, как это было сделано в примере 8. Доказать, что построенное определение верно, обычно можно по индукции.

Пример 20

Задача. Доказать, что рекурсивное и не рекурсивное определения факториала эквивалентны.

Доказательство. Для начала в обоих определениях постулируется, что $0! = 1$. Допустим, что

$$k! = \prod_{j=1}^{k} j .$$

Тогда для $(k+1)!$ можно записать

$$(k+1)! = (k+1) \cdot k! = (k+1) \prod_{j=1}^{k} j = \prod_{j=1}^{k+1} j .$$

Таким образом, определения совпадают для тривиального случая $n=0$, и если они совпадают для некоторого k, то совпадают и для $(k+1)$. Согласно принципу математической индукции, это означает, что определения эквивалентны для всех целых неотрицательных n.

Что и требовалось доказать.

Рекурсивная функция в математике, как нетрудно догадаться, – это функция, заданная рекурсивно, то есть содержащая сама себя в своём определении. Например, факториал, определённый в виде (7) можно считать рекурсивной функцией. Другим известным примером рекурсивной функции в математике является функция Аккермана, определяемая как

$$A(m,n)=\begin{cases} n+1, & m=0; \\ A(m-1,1), & m>0 \wedge n=0; \\ A(m-1,A(m,n-1)), & m>0 \wedge n>0. \end{cases} \quad (8)$$

Это крайне быстрорастущая функция, не являющаяся вычислимой с помощью примитивно рекурсивных функций, которые упоминались выше.

В языках программирования также есть аналоги рекурсивных функций из математики. В программировании *рекурсивная функция* – это функция, вызывающая сама себя в процессе своей работы. Аналогичным образом определяются *рекурсивные процедуры*.

В листинге 30 приводится пример рекурсивной функции на языке C, решающей задачу из примера 8, то есть вычисляющей значение остатка от деления факториала числа n на число m. Видно, что в начале функции традиционно располагается база рекурсии – условие, которое нужно проверить в первую очередь, чтобы не зациклиться. Далее выполняется вычисление непосредственно по формуле (7) с учётом взятия остатка от деления после каждого умножения. Небольшой тонкостью является необходимость приведения типов с целью не допустить целочисленного переполнения при умножении.

Листинг 30. Реализация рекурсивного алгоритма вычисления факториала по модулю

```
int fact(int n, int m)
{
    if (n == 0)
    {
        return 1;
    }
    return (int)((long long)n * fact(n - 1, m) % m);
}
```

Чтобы говорить о том, как происходит рекурсивный вызов функции, следует сначала обсудить, что вообще происходит с вычислительным устройством в случае вызова функции. Значения локальных переменных вместе с адресом выполняемой в данный момент инструкции называются *фреймом активации* или *стековым кадром* (англ. *activation frame* или *stack frame*). Когда происходит вызов функции, фрейм активации сохраняется на стек, после чего происходит выполнение непосредственно тела функции. После того, как функция завершила свою работу, фрейм активации загружается, восстанавливаются значения локальных переменных и программа продолжает выполняться с инструкции, на которую указывает адрес возврата, предусмотрительно сохранённый во фрейме активации вместе со значениями переменных.

Как видно, в листинге 30 функция вызывает себя прямо посреди вычисления некоторого выражения. При этом происходит следующее.

1. Фрейм активации сохраняется на стек.
2. Исполняется та же самая функция, но с другими значениями параметров и другими локальными переменными. Возможно, на этом этапе произойдут ещё рекурсивные вызовы по той же схеме.
3. Фрейм активации загружается, исполнение возвращается во внешнюю функцию, и исполнение продолжается. В данном случае операция деления с остатком будет выполнена после рекурсивного вызова.

Набор фреймов активации, сохранённых на стеке в данный момент, называется *стеком вызовов* (англ. *call stack*). Он содержит функции, работающие в данный момент, в том порядке, в котором каждая из них вызывала следующую функцию. Следует обратить внимание, что в первую очередь исполнение будет передано в ту функцию, которая была помещена в стек вызовов последней.

Например, пусть была вызвана функция `fact(5, 1000)` из листинга 30. Поскольку $n = 5 \neq 0$, текущие параметры сохранятся на стек, и будет вызвана функция `fact(4, 1000)`, после чего, поскольку $n = 4 \neq 0$ будет вызвана функция `fact(3, 1000)`. Так будет продолжаться, пока, наконец, не будет вызвана функция `fact(0, 1000)`. Когда программа зайдёт в эту функцию, чтобы проверить базу рекурсии, стек вызовов будет иметь вид

```
fact(1, 1000)
fact(2, 1000)
fact(3, 1000)
fact(4, 1000)
fact(5, 1000)
```

Ниже в стеке может быть функция, которая собственно и вызвала функцию `fact(5, 1000)`. В самом низу стека будет лежать функция `main` – в неё управление будет передано в последнюю очередь.

После этого сначала в функцию `fact(1, 1000)` будет передана единица, вернувшаяся из функции `fact(0, 1000)`. Она будет умножена на $n = 1$ и результат вернётся в функцию `fact(2, 1000)`. Там результат будет умножен на $n = 2$ и передан обратно в `fact(3, 1000)`, которая когда-то вызвала `fact(2, 1000)`. Так стек вызовов раскручивается обратно. Наконец, финальный результат 120 будет возвращён из функции `fact(5, 1000)`. В этом примере факт взятия остатка от деления на 1000 можно не учитывать, поскольку все получающиеся результаты умножения не превосходят 1000, но фактически оно выполняется в каждом вызове функции.

Если база рекурсии была построена неправильно, программа может уйти в глубокую рекурсию, делая всё новые и новые вызовы, каждый раз сохраняя фреймы активации на стек. Количество вхождений рекурсивной функции в стек вызовов называется *глубиной рекурсии*. Нужно понимать, что каждый новый сохранённый фрейм активации занимает память на стеке. Поэтому в случае бесконечной рекурсии, как правило, стек быстро переполняется, и программа аварийным образом завершает свою работу с ошибкой, называемой *переполнением стека* (англ. *stack overflow*). Такая же ошибка возникает, если попытаться создать на стеке огромный статический массив.

Алгоритмы сами по себе также могут быть рекурсивными и не рекурсивными. Например, алгоритм вычисления факториала, основанный формуле (7), рекурсивный, поскольку отсылает к вычислению факториала на единицу меньшего числа по тому же самому алгоритму. Рекурсивные алгоритмы реализуются в виде рекурсивных функций. В частности, листинг 30 представляет собой рекурсивную функцию, реализующую такой рекурсивный алгоритм.

В общем случае реализовать рекурсивный алгоритм без использования рекурсивных функций практически невозможно. Понятно, что можно самостоятельно сохранять аналоги фреймов активации в массив, являющийся аналогом стека вызовов, но фактически это будет просто программная реализация аппаратного механизма вызова рекурсивных функций.

Ясно, что для вычисления факториала существует и не рекурсивный алгоритм, основанный на формуле (4). Алгоритм заключается в изначальном присвоении некоторой переменной a значения 1 и в последующем последовательном умножении переменной a на числа $k \in [1;n] \cap \mathbf{Z}$. Такой алгоритм называется итеративным, поскольку заключается в итерировании по значениям переменной k. Одна из возможных программных реализаций такого алгоритма была приведена в листинге 1. Как видно, эта реализация представляет собой не рекурсивную функцию.

Однако любой итеративный алгоритм может быть представлен в виде рекурсивной функции, не содержащей циклов. В листинге 31 приведена рекурсивная функция на языке C, которая реализует не рекурсивный алгоритм вычисления факториала по модулю. Она принимает три аргумента, последний из которых – это аналог переменной `ans` из листинга 1, в которой накапливается ответ. Эту функцию для заданных переменных `n` и `m` следует вызывать так: `fact(n, m, 1)`. В этом случае, единственным отличием от реализации из листинга 1 будет тот факт, что умножение на числа из множества $[1;n] \cap \mathbf{Z}$ выполняется в обратном порядке, а не в прямом, но и это при желании можно исправить.

Листинг 31. Вычисление факториала по модулю с помощью хвостовой рекурсии

```
int fact(int n, int m, int ans)
{
    if (n == 0)
    {
        return ans;
    }
    return fact(n - 1, m, (int)((long long)ans * n % m));
}
```

Важным отличием рекурсивной функции из листинга 31 от рекурсивной функции из листинга 30 является отсутствие каких-либо операторов после рекурсивного вызова: полученный ответ сразу же возвращается из функции. В листинге 30 после рекурсивного вызова выполняется вычисление выражения: умножение и деление с остатком. Ситуация, при которой единственный рекурсивный вызов является последним оператором в функции, называется *хвостовой рекурсией*.

Любой итеративный алгоритм может быть представлен в виде хвостовой рекурсии без циклов, и, напротив, любая хвостовая рекурсия может быть переписана в виде цикла без рекурсивных вызовов. Здесь нужно понимать, что, несмотря на то, что в листингах 1 и 31 реализован один и тот же алгоритм, цикл работает эффективнее рекурсии, поскольку на сам рекурсивный вызов функции

дополнительно тратится время и память: сохраняется и загружается фрейм активации. Поэтому некоторые компиляторы автоматически оптимизируют хвостовую рекурсию в цикл при компиляции программы, хотя к большинству компиляторов языков C и C++ это и не относится. Если компилятор умеет оптимизировать хвостовую рекурсию, то циклы в таком языке программирования, по сути, представляют собой *синтаксический сахар*, то есть являются лишними синтаксическими конструкциями, без которых можно обойтись.

Надо сказать, не всегда рекурсивная функция вызывает себя непосредственно в своём исходном коде. Может быть так, что функция вызывает другую функцию, а уже та функция вызывает её саму. Ситуация, в которой функция вызывает саму себя по имени непосредственно в своём же коде, называется *простой рекурсией*. Рекурсия, не являющаяся простой, называется *сложной рекурсией*. Эти определения легко можно переформулировать и в отношении рекурсивных функций в математике, рекурсивных алгоритмов и рекурсивных определений.

Напоследок можно заметить, что при наличии затруднений в понимании рекурсии, сведения об этом понятии могут быть найдены в разделе 5.1 настоящего пособия.

5.2 Оценка вычислительной сложности рекурсивных алгоритмов

Для некоторых задач рекурсивные алгоритмы решения приходят в голову гораздо быстрее итеративных, проще реализуются и выглядят более изящно. Однако прежде чем приступать к реализации алгоритма, неплохо бы оценить его вычислительную сложность, и тут с рекурсивными алгоритмами могут возникнуть определённые проблемы. Если для итеративных алгоритмов обычно достаточно просто прикинуть количество итераций разных вложенных циклов, то для рекурсивных алгоритмов всё может зайти дальше, чем хотелось бы.

В некоторых простых случаях, тем не менее, оценка вычислительной сложности рекурсивных алгоритмов не представляет труда. Например, для

алгоритма, реализованного в листинге 30, очевидно, что в случае вызова функции `fact(n, m)` для некоторого целого неотрицательного числа n будут произведены также рекурсивные вызовы для всех целых неотрицательных чисел, не превышающих n, причём для каждого такого числа будет сделан ровно один рекурсивный вызов. Это связано с тем, что после каждого вызова в следующем вызове n уменьшается на единицу, пока не достигнет нуля. При этом очевидно, что каждый вызов без учёта других рекурсивных вызовов выполняет $O(1)$ операций. Таким образом, имеется $(n+1)$ вызов, каждый из которых выполняет $O(1)$ операций, то есть общая вычислительная сложность алгоритма составляет $O(n)$.

Полностью аналогичные рассуждения работают и для рекурсивной реализации итеративного алгоритма из листинга 31. К тому же, очевидно, что этот алгоритм имеет такую же вычислительную сложность, как и итеративный алгоритм, реализованный в листинге 1. Вообще для хвостовой рекурсии всегда можно просто оценить вычислительную сложность, глядя на итеративную реализацию этого алгоритма. Следует отметить, что вычислительная сложность приведённых алгоритмов нахождения факториала по модулю m не зависит от m.

А вот асимптотическая оценка потребляемой памяти имеет отношение по большей части к реализации алгоритма, а не к самому абстрактному алгоритму. Как уже было отмечено, алгоритм, реализованный в листинге 30, выполняет $(n+1)$ рекурсивный вызов самого себя. При каждом вызове сохраняется фрейм активации, потребляющий $O(1)$ памяти на стеке. Таким образом, функция, записанная в листинге 30, потребляет $O(n)$ памяти. То же самое касается и функции из листинга 31. И последний факт интересен тем, что итеративная реализация вычисления факториала из листинга 1 потребляла $O(1)$ памяти, поскольку нуждалась лишь в нескольких дополнительных переменных, ничего не сохраняя на стек при каждой итерации. Это ещё один недостаток

рекурсивных реализаций перед итеративными реализациями: они потребляют память при каждом рекурсивном вызове, что может даже привести к асимптотическому ухудшению объёма потребляемой памяти.

К сожалению, не всегда оценка вычислительной сложности рекурсивных алгоритмов даётся так легко. Далее рассмотрим другой простой классический пример рекурсивных вычислений.

Пример 21

Задача. Требуется разработать алгоритм, который для двух целых чисел n и m вычисляет остаток от деления n-го числа Фибоначчи на m, а также написать исходный код его программной реализации. Первые два числа Фибоначчи равны 1, а каждое следующее вычисляется, как сумма двух предыдущих. Оценить вычислительную сложность реализованного алгоритма.

Решение. Первое, что приходит в голову, – это решить представленную задачу с помощью рекурсивного алгоритма, вытекающего непосредственно из определения чисел Фибоначчи: если нас просят найти первое или второе число, то мы уже знаем ответ, а иначе просто вычислим предыдущие два числа с помощью этого же алгоритма и посчитаем их сумму. Рекурсивная формула для вычисления чисел Фибоначчи может быть записана в виде

$$F_n = \begin{cases} 1, & n \in \{1;2\}; \\ F_{n-1} + F_{n-2}, & n \in \mathbf{N} \setminus \{1;2\}. \end{cases} \tag{9}$$

Рекурсивная функция на языке C, реализующая такой алгоритм вычисления чисел Фибоначчи по модулю, представлена в листинге 32.

Листинг 32. Рекурсивная функция для вычисления чисел Фибоначчи

```
int fib(int n, int m)
{
    if (n <= 2)
    {
        return 1;
    }
    return (int)(((long long)fib(n - 1, m) + fib(n - 2, m)) % m);
}
```

Здесь опять во избежание целочисленного переполнения выполняются приведения типов, в остальном же решение не имеет каких-либо тонких

деталей реализации. Как видно, в данной функции не один, а целых два рекурсивных вызова. Из раздела 1.5 уже известно, что выполнять деление с остатком после каждого сложения корректно.

Попытаемся оценить вычислительную сложность $T(n)$ такого алгоритма. Для простоты будем учитывать только самую первую операцию сравнения переменной `n` с числом `2`. Две последующие операции сложения и деления с остатком не должны существенно изменять асимптотику. Понятно, что при таком раскладе $T(1)=T(2)=1$. Что касается других $T(n)$, их сложность определяется сложностью $T(n-1)$ и $T(n-2)$, поскольку сначала выполняется один рекурсивный вызов, а затем другой. Опять же для простоты допустим, что никакие другие операции при этом не выполняются, поскольку они не должны повлиять на асимптотику. Тогда можно записать следующие соотношения для вычислительной сложности $T(n)$:

$$T(n)=\begin{cases}1, & n\in\{1;2\};\\ T(n-1)+T(n-2), & n\in\mathbf{N}\setminus\{1;2\}.\end{cases}$$

Видно, что это рекуррентное соотношение совпадает с рекуррентным соотношением (9) для самих чисел Фибоначчи, то есть $T(n)=F_n$. Иными словами, чтобы вычислить n-е число Фибоначчи, этому алгоритму требуется примерно столько операций, какое это число само по величине. Много это или мало? Это однородное линейное рекуррентное соотношение с постоянными коэффициентами, которое легко можно решить точно. Получающийся ответ:

$$F_n=\frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n-\left(\frac{1-\sqrt{5}}{2}\right)^n\right). \qquad (10)$$

Поскольку $1<\left(1+\sqrt{5}\right)/2<2$, а $-1<\left(1-\sqrt{5}\right)/2<0$, только первое слагаемое влияет на асимптотику, то есть

$$T(n)=O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right).$$

Число $\left(1+\sqrt{5}\right)/2$ в математике имеет особое значение и называется *золотым сечением*.

Нужно ли объяснять, что это экспоненциальная вычислительная сложность, поэтому такой алгоритм менее эффективен, чем любой полиномиальный алгоритм? На первый взгляд, числа Фибоначчи растут довольно медленно. Первые десять чисел Фибоначчи: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55. На самом же деле, это впечатление обманчиво: числа Фибоначчи растут крайне быстро, так что уже $F_{40} > 10^8$. Это значит, что поиск 40-го числа Фибоначчи с помощью функции, реализованной в листинге 32, скорее всего, не успеет отработать за секунду на большинстве современных домашних компьютеров.

Между тем асимптотическая оценка затраченной памяти для функции из листинга 32 выглядит не настолько печально. Поскольку в стеке вызовов никогда не может находиться более n функций одновременно (каждая функция вызывает сама себя от аргумента, меньшего на единицу), можно заключить, что такая функция потребляет $O(n)$ памяти. Когда рекурсивный вызов заканчивает работу, затраченная память очищается и начинает работать второй рекурсивный вызов.

Как решить эту задачу более эффективно? Ну, во-первых, прямое вычисление числа Фибоначчи по формуле (10) с использованием чисел с плавающей запятой и определённой для них операции возведения в степень даёт алгоритм с вычислительной сложностью $O(1)$, использующий $O(1)$ памяти. Казалось бы, ничего лучше просто не может быть. Однако в силу ошибок, возникающих при вычислении чисел с плавающей запятой, такой способ не годится на практике: пятнадцати знаков точности, обеспечиваемых типом `double`, хватит на хранение от силы 70 первых чисел Фибоначчи. Если нас просят вывести остаток от деления, то неточные числа Фибоначчи нас не интересуют.

На самом деле, довольно очевиден итеративный алгоритм решения этой задачи. Просто будем вычислять числа Фибоначчи последовательно, начиная с третьего, используя для этого формулу (9). Хранить все вычисленные таким образом числа нет необходимости: достаточно хранить только два последних. Когда будет посчитано число Фибоначчи с необходимым номером, остановимся и выведем ответ. Пример реализации такого алгоритма в виде функции на языке C приведён в листинге 33.

Листинг 33. Итеративная функция, вычисляющая числа Фибоначчи

```
int fib(int n, int m)
{
    long long a = 1;
    long long b = 1;
    for (int i = 3; i <= n; ++i)
    {
        long long c = (a + b) % m;
        a = b;
        b = c;
    }
    return (int)b;
}
```

Для простоты записи вычисления выполняются в рамках 64-битного целочисленного типа данных, хотя после каждого деления с остатком результат снова может быть помещён в 32-битный тип. Вместо этого приведение типа выполняется один раз в самом конце. Заметно, что вычислительная сложность такого алгоритма снова не зависит от переменной `m`.

Несложно понять, что вычислительная сложность алгоритма, реализованного в листинге 33, составляет $O(n)$: цикл делает $(n-2)$ итерации, в каждой из которых выполняется $O(1)$ операций. Это значительно более эффективно, чем в алгоритме из листинга 32, поскольку на обычном домашнем компьютере можно менее чем за секунду вычислить число Фибоначчи с номером $n=10^7$. Кроме того, приятно, что функция из листинга 33 использует $O(1)$ памяти: в процессе её работы заводятся всего 4 переменные.

Понятно, что итеративный алгоритм, реализованный в листинге 33, допускает и рекурсивную реализацию, использующую хвостовую рекурсию. Рекурсивная функция на языке C, реализующая этот итеративный алгоритм, приведена в листинге 34. Здесь переменные `a`, `b` и `i` являются параметрами функции и меняются так же, как раньше менялись при итерациях цикла. Предполагается, что эта функция должна вызваться в форме `fib(n, m, 1, 1, 3)` по аналогии с тем, как эти переменные были проинициализированы в листинге 33.

Листинг 34. Вычисление чисел Фибоначчи по модулю с помощью хвостовой рекурсии

```
int fib(int n, int m, int a, int b, int i)
{
    if (i > n)
    {
        return b;
    }
    return fib(n, m, b, (int)(((long long)a + b) % m), i + 1);
}
```

Функция из листинга 34 является рекурсивной, так же как и функция из листинга 32, но отличается намного большей эффективностью, поскольку её вычислительная сложность составляет $O(n)$. Этот факт объясняется тем, что она работает точно так же, как и цикл из листинга 33. Тем не менее, в отличие от цикла эта функция потребляет $O(n)$ памяти, поскольку выполняет $O(n)$ рекурсивных вызовов подряд, каждый из которых приводит к сохранению фрейма активации на стеке. Как видно, не рекурсивная реализация снова выглядит более эффективной.

Существует и более эффективный и надёжный алгоритм вычисления чисел Фибоначчи, подходящий, кроме того, вообще для любых однородных линейных рекуррентных соотношений. С помощью него можно решить представленную задачу фактически для любых натуральных чисел n и m, представимых в 64-битном целочисленном типе данных, менее чем за секунду на всех современных персональных компьютерах. Более того, при эффективной

реализации длинной арифметики можно решать эту задачу и для намного больших значений n и m. Однако в данном курсе этот алгоритм рассматриваться не будет.

В итоге общий порядок оценки вычислительной сложности рекурсивных алгоритмов состоит в следующем.

1. Составить рекуррентное соотношение для функции вычислительной сложности алгоритма, установив её зависимость от вычислительной сложности этого же алгоритма с другими начальными данными.
2. Определить начальные условия для рекуррентного соотношения, рассмотрев вычислительную сложность алгоритма для некоторых тривиальных значений аргумента, соответствующих базе рекурсии.
3. Решить рекуррентное соотношение с данным набором начальных условий и асимптотически оценить решение.

К сожалению, не все рекуррентные соотношения допускают решения в элементарных функциях, а для тех, которые допускают, решение всё равно может представлять значительную сложность. Иногда при введении некоторых допущений удаётся привести рекуррентное соотношение к виду однородного линейного рекуррентного соотношения, решение которого не представляет труда. В других случаях приходится прибегать к более сложным хитростям и оценкам.

5.3 Мемоизация и рекурсия с сохранением

Уже должно быть понятно, что в большинстве случаев вместо рекурсивного алгоритма лучше реализовать итеративный алгоритм. Однако в некоторых случаях разработка итеративного алгоритма весьма затруднительна, а то и вовсе фактически невозможна. Существует приём, позволяющий существенно ускорить работу программы и в этом случае.

Давайте попробуем разобраться, в чём заключается проблема рекурсивной функции из листинга 32, из-за которой она настолько менее эффективна, чем рекурсивная функция из листинга 34. Дело в том, что чтобы

вычислить, например, пятое число Фибоначчи, эта функция рекурсивно вычисляет третье и четвёртое числа Фибоначчи, а для того, чтобы вычислить четвёртое число Фибоначчи, она рекурсивно вычисляет второе и третье. Но рекурсивные вызовы ничего не знают друг про друга, поэтому даже в приведённом выше рассуждении делается два вызова для вычисления третьего числа Фибоначчи, каждый из которых порождает ещё по два одинаковых рекурсивных вызова. Схема этих рекурсивных вызовов изображена на рисунке 5. Можно себе представить, сколько лишних вызовов делается при попытке посчитать, например, десятое число Фибоначчи.

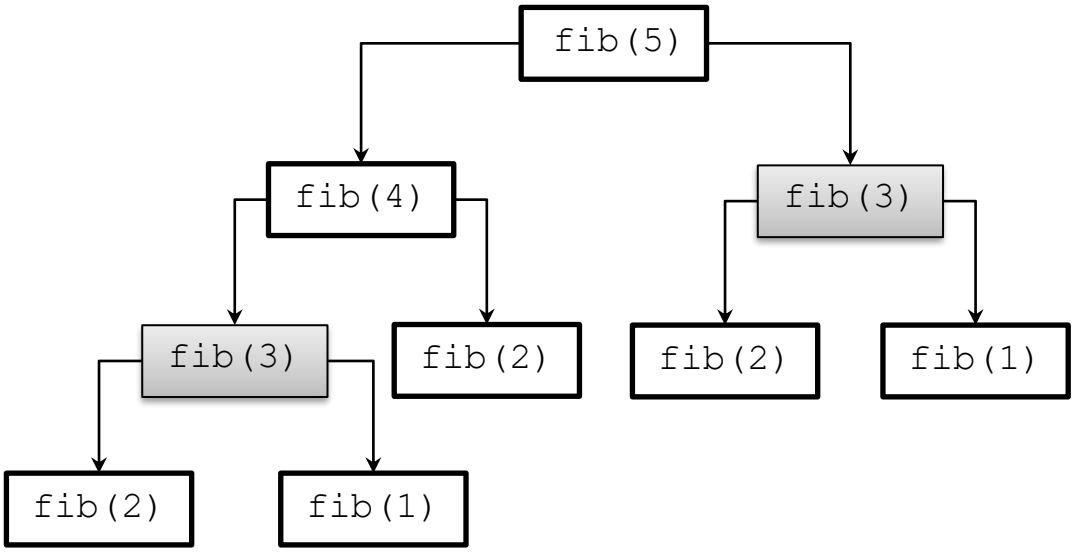

Рисунок 5 – Вызовы рекурсивной функции, вычисляющей числа Фибоначчи

Чтобы решить эту проблему предлагается весьма простой подход: не считать несколько раз то, что ранее было уже посчитано. Вместо этого можно запоминать результат работы функции для определённых значений аргументов, и впоследствии просто возвращать этот результат, если он имеется в памяти. Такой приём в программировании называется *мемоизацией* (англ. *memoization*). Не важно, сохраняются ли результаты в обычный глобальный массив или в более сложные структуры данных, а также сохраняются ли результаты абсолютно всех вызовов функции, или же только для некоторых значений аргументов. Мемоизация сама по себе является довольно общим подходом.

Речь даже не идёт о том, чтобы сохранять таким образом только результаты работы рекурсивных функций: не рекурсивные функции тоже могут часто вызываться от одного и того же набора аргументов.

Общая схема мемоизации состоит в следующем.

1. Если функция уже вызывалась от такого набора аргументов, просто вернуть сохранённый результат её работы.
2. Иначе вычислить значение функции, сохранить его и вернуть результат.

Понятно, что использование мемоизации имеет определённые ограничения и на практике используется в довольно редких случаях. Основные ограничения приведены ниже.

1. Функция не должна иметь побочных эффектов, то есть для заданного набора аргументов функция всегда должна возвращать одинаковый результат.
2. Вычислительная сложность получения сохранённого в памяти результата работы функции должна быть существенно ниже, чем вычислительная сложность самой функции.
3. Выделенной программе памяти должно хватать, чтобы хранить результаты работы функции для всех аргументов, для которых эти результаты планируется сохранять.

Но даже если все эти условия выполняются, нужно понимать, что мемоизация потребляет память на хранение результатов работы функции. Следует всегда внимательно оценивать, насколько большой выигрыш по производительности и насколько большой проигрыш по использованию памяти будет получен в результате использования этого приёма. Подобная ситуация, в которой увеличение потребления памяти приводит к ускорению работы программы и наоборот, в западной литературе носит название *space-time trade-off*.

Применение идеи мемоизации для оптимизации рекурсивных функций называется *рекурсией с сохранением*. При использовании этого приёма можно быть уверенным, что функция не будет много раз рекурсивно вызываться от

одного и того же набора аргументов. При этом можно не задумываться о разработке альтернативного итеративного алгоритма для исправления ситуации: рекурсию с сохранением можно применить непосредственно к рекурсивной реализации рекурсивного алгоритма.

Листинг 35. Рекурсия с сохранением для вычисления чисел Фибоначчи

```c
#define LIM 1000000

int mem[LIM];

int fib(int n, int m)
{
    if (n < LIM && mem[n] != -1)
    {
        return mem[n];
    }
    int ans;
    if (n <= 2)
    {
        ans = 1;
    }
    else
    {
        ans = (int)(((long long)fib(n - 1, m) + fib(n - 2, m)) % m);
    }
    if (n < LIM)
    {
        mem[n] = ans;
    }
    return ans;
}
```

В листинге 35 на языке C показан пример использования рекурсии с сохранением для функции из листинга 32. Результаты работы функции сохраняются в глобальный статический массив `mem`, состоящий из миллиона чисел. Предполагается, что перед вызовом функции `fib` массив `mem` должен быть заполнен значениями `-1`. В момент вызова функции сразу же проверяется, есть ли для заданного аргумента уже сохранённый ответ, и в случае наличия ответа, он немедленно возвращается из функции. В противном случае ответ вычисляется ровно точно так же, как и в листинге 32. Фактически это та же самая функция. Далее если аргумент один из первого миллиона, то результат сохраняется в массив, после чего результат возвращается из функции. Следует

заметить, что в данном случае мемоизация осуществлялась только для первого аргумента функции, тогда как второй просто не менял своё значение.

Понятно, что если требуется вычислить значение такой функции для некоторого n, и имеется достаточно памяти для заведения массива из n элементов, то вычислительная сложность такого алгоритма, использующего рекурсию с сохранением, составит $O(n)$. Это связано с тем, что рекурсивные вызовы для всех натуральных значений первого аргумента, не превосходящих n, фактически осуществляются только по одному разу. Кроме того, эта функция использует $O(n)$ памяти на все сохранения и рекурсивные вызовы, что не хуже, чем другие рекурсивные аналоги. По использованию памяти её превосходит только не рекурсивная реализация из листинга 33.

Упражнения для самостоятельной работы

1. Докажите, что функция Аккермана (8) определена для любых целых неотрицательных чисел n и m.

2. Имеются три алгоритма, вычислительные сложности которых соответственно $O(n!)$, $O(n^n)$ и $O(A(n,n))$, где $A(m,n)$ – это функция Аккермана (8). Какой из них наименее эффективен? Приведите пример вычислительной сложности ещё менее эффективного алгоритма?

3. Имеются три алгоритма, вычислительные сложности которых соответственно $O(\sqrt{n})$, $O(\log n)$ и $O(\alpha(n))$, где $\alpha(n) = \min\{k \in \mathbf{N}_0 : A(k,k) \geq n\}$ – это функция, обратная функции Аккермана $A(n,n)$. Какой из них наиболее эффективен? Можете ли вы привести вычислительную сложность ещё более эффективного алгоритма?

4. Конструктивно покажите, что любой итеративный алгоритм может быть представлен в виде хвостовой рекурсии без циклов, и, напротив, любая хвостовая рекурсия может быть переписана в виде цикла без рекурсивных вызовов.

5. Решите линейное рекуррентное соотношение $x(n)=x(n-1)+x(n-2)$ в общем виде. Какие асимптотические оценки можно построить для функции $x(n)$ и как они могут зависеть от начальных условий?

6. Решите линейное рекуррентное соотношение $x(n)=x(n-1)+x(n-2)+1$ в общем виде. Понятно, что это соотношение лучше описывает вычислительную сложность алгоритма, реализованного в листинге 32. Отличается ли асимптотическая оценка вычислительной сложности этого алгоритма при использовании такого рекуррентного соотношения?

7. Рекурсивный алгоритм выполняет $T(n)$ операций, где n – размер входных данных. Для пустых входных данных он выполняет только одну операцию, проверяющую этот факт, то есть $T(0)=1$. Оцените вычислительную сложность этого алгоритма.

а) $T(n)=T(n-1)+1$

б) $T(n)=T(n-1)+n$

в) $T(n)=T\left(\left\lfloor \frac{n}{2} \right\rfloor\right)+1$

г) $T(n)=T\left(\left\lfloor \frac{n}{2} \right\rfloor\right)+1$

д) $T(n)=2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right)+1$

е) $T(n)=2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right)+n$

ж) $T(n)=T\left(\left\lfloor \sqrt{n} \right\rfloor\right)+1$

8. Имеется исходный код функции на языке C:

```
int foo(int a, int b)
{
    if (b == 0)
    {
        return a;
    }
    if (b > 0)
    {
        return foo(a + 1, b - 1);
    }
    else
    {
        return foo(a - 1, b + 1);
    }
}
```

Какую задачу решает эта функция? Оцените её вычислительную сложность. Приведите более эффективный алгоритм решения этой же задачи.

9. Докажите, что для чисел Фибоначчи F_n справедливо, что

$$\sum_{k=1}^{n} F_n = F_{n+2} - 1.$$

10. Вычислите предел

$$\lim_{n\to\infty} \frac{F_{n+1}}{F_n},$$

где F_n – это n-е число Фибоначчи.

11. Докажите, что для чисел Фибоначчи F_n справедливо, что

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}.$$

Постройте алгоритм вычисления чисел Фибоначчи, основанный на этом факте, и напишите его программную реализацию. Асимптотически оцените его вычислительную сложность и объём потребляемой памяти.

12. Докажите, что последовательность последних цифр соответствующих чисел Фибоначчи является периодической. Какова длина её периода?

13. Найдите все числа Фибоначчи, являющиеся точными квадратами натуральных чисел.

14. Напишите рекурсивную функцию для поиска максимального элемента массива. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

15. Напишите рекурсивную функцию без циклов и массивов, считывающую последовательность чисел и выводящую эту же последовательность задом наперёд. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

16. Напишите рекурсивную функцию для возведения одного натурального числа в степень другого натурального числа по модулю третьего натурального числа. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

17. Напишите рекурсивную функцию, проверяющую, является ли заданное натуральное число простым. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

18. Напишите исходный код программы, принимающей на вход три целых числа m, n и p ($1 \le m, n, p \le 10^9$), и выводящей остаток от деления функции Аккермана $A(m,n)$ на p. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти. Для каких m и n ваша программа успеет отработать быстрее, чем за секунду?

19. Напишите исходный код программы, принимающей на вход единственное целое число n ($1 \le n \le 10$), и выводящей все различные последовательности из нулей и единиц длины n. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

20. Напишите исходный код программы, принимающей на вход единственное целое число n ($1 \le n \le 10$), и выводящей все различные подмножества множества из первых n натуральных чисел. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

21. Напишите исходный код программы, принимающей на вход единственное целое число n $(1 \le n \le 10)$, и выводящей все различные перестановки из первых n натуральных чисел. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

22. Напишите исходный код программы, принимающей на вход конечную последовательность натуральных чисел $\{a_i\}_{i=1}^{n}$ $(1 \le n \le 10,$ $1 \le a_i \le 10^9)$, и выводящей, перед какими из этих чисел нужно поставить знак минус, чтобы в сумме получился ноль, либо сообщающей, что решение отсутствует. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

6 Сортировка массива

6.1 Постановка задачи сортировки массива

На практике часто требуется упорядочивать различные объекты по определённым критериям: при работе с электронными таблицами, при поиске записей в базе данных, при графическом отображении событий, произошедших в разное время, и т. д. Число объектов, которые необходимо упорядочить, при этом может достигать миллиона и более. Понятно, что для успешной реализации процедуры упорядочивания необходим эффективный алгоритм. Но прежде чем говорить об алгоритме, необходимо формально поставить задачу.

Пусть имеется конечная последовательность $\{a_i\}_{i=1}^{n}$ из $n \in \mathbf{N}$ элементов множества A, и на множестве A задано *отношение линейного порядка* $\leq$, то есть выполняются четыре условия.

1. Рефлексивность:

$$\forall a \in A : a \leq a .$$

2. Антисимметричность:

$$\forall a_1, a_2 \in A : a_1 \leq a_2 \wedge a_2 \leq a_1 \Rightarrow a_1 = a_2 .$$

3. Транзитивность:

$$\forall a_1, a_2, a_3 \in A : a_1 \leq a_2 \wedge a_2 \leq a_3 \Rightarrow a_1 \leq a_3 .$$

4. Полнота:

$$\forall a_1, a_2 \in A : a_1 \leq a_2 \vee a_2 \leq a_1 .$$

В случае, если на множестве задано отношение линейного порядка, такое множество называется *линейно упорядоченным множеством*.

Отношение строгого порядка $<$ может быть определено через отношение линейного порядка $\leq$:

$$a < b \Leftrightarrow a \leq b \wedge \overline{b \leq a} .$$

Иными словами, один элемент меньше другого элемента, если он не больше его, и при этом они не равны.

Назовём массив a из $n \in \mathbf{N}$ элементов линейно упорядоченного множества A *отсортированным по неубыванию*, если для любых двух его элементов элемент с меньшим номером не больше, чем элемент с большим номером, то есть

$$\forall i, j \in [1; n] \cap \mathbf{Z} : i < j \Rightarrow a_i \le a_j .$$

Назовём массив a из $n \in \mathbf{N}$ элементов линейно упорядоченного множества A *отсортированным по невозрастанию*, если для любых двух его элементов элемент с большим номером не больше, чем элемент с меньшим номером, то есть

$$\forall i, j \in [1; n] \cap \mathbf{Z} : i < j \Rightarrow a_j \le a_i .$$

Назовём массив a из $n \in \mathbf{N}$ элементов линейно упорядоченного множества A *отсортированным строго по возрастанию*, если для любых двух его элементов элемент с меньшим номером строго меньше, чем элемент с большим номером, то есть

$$\forall i, j \in [1; n] \cap \mathbf{Z} : i < j \Rightarrow a_i < a_j .$$

Назовём массив a из $n \in \mathbf{N}$ элементов линейно упорядоченного множества A *отсортированным строго по убыванию*, если для любых двух его элементов элемент с большим номером не больше, чем элемент с меньшим номером, то есть

$$\forall i, j \in [1; n] \cap \mathbf{Z} : i < j \Rightarrow a_j < a_i .$$

Например, массив $(-4; -1; 0; 7; 10)$ является упорядоченным строго по возрастанию. Массив $(10; 7; 4; -2; -3)$ является упорядоченным строго по убыванию. Массив $(-1; 4; 4; 8; 11; 13; 13)$ является упорядоченным по неубыванию. Массив $(1; 1; 1; -3; -7; -10)$ является упорядоченным по невозрастанию. Массив $(-2; 5; 3; 7; 10)$ не является упорядоченным ни по неубыванию, ни по невозрастанию. Массив $(0; 0; 0; 0; 0; 0)$ является упорядоченным и по неубыванию, и по невозрастанию. Понятно, что чтобы

массив являлся упорядоченным строго по возрастанию или строго по убыванию, в нём любые два элемента с разными номерами должны быть различны.

Перестановкой p из $n \in \mathbf{N}$ первых натуральных чисел называется биекция множества $[1;n] \cap \mathbf{Z}$ в себя, то есть $p : [1;n] \cap \mathbf{Z} \to [1;n] \cap \mathbf{Z}$ и выполняются два свойства.

1. Инъективность:

$$\forall i, j \in [1;n] \cap \mathbf{Z} : i \neq j \Rightarrow p(i) \neq p(j).$$

2. Сюръективность:

$$\forall j \in [1;n] \cap \mathbf{Z}\ \exists i \in [1;n] \cap \mathbf{Z} : p(i) = j.$$

В общем случае, решить *задачу сортировки массива* a из $n \in \mathbf{N}$ элементов линейно упорядоченного множества A по неубыванию означает найти перестановку p из n первых натуральных чисел, такую что при применении этой перестановки к индексам элементов этого массива, массив становится отсортированным по неубыванию, то есть такую что

$$\forall i, j \in [1;n] \cap \mathbf{Z} : i < j \Rightarrow a_{p(i)} \leq a_{p(j)}.$$

Аналогичным образом можно определить, в чём состоят задачи сортировки по невозрастанию, строго по возрастанию и строго по убыванию. Нужно отметить, что для задач сортировки строго по возрастанию или строго по убыванию ответ может не существовать.

На самом деле, на практике обычно требуется не сама перестановка p, а именно отсортированный с помощью неё массив a', такой что

$$\{a'_i\}_{i=1}^{n} = \left\{a_{p(i)}\right\}_{i=1}^{n}.$$

Например, отсортируем массив $(8; 5; 3; 5; 1; 6; 3; 6; 5; 4)$ по неубыванию. Нетрудно сделать это вручную: получится массив $(1; 3; 3; 4; 5; 5; 5; 6; 6; 8)$. Если отсортировать этот массив по невозрастанию, то получится $(8; 6; 6; 5; 5; 5; 4; 3; 3; 1)$.

Задача сортировки – не самая сложная задача в информатике, но она сложнее, чем может показаться на первый взгляд. Ведь в ней требуется найти перестановку из n элементов, а общее количество таких перестановок $n!$, так что перебирать их все экспоненциально долго. При этом проверить, является ли заданный массив уже отсортированном в некотором порядке, можно с помощью несложного алгоритма за $O(n)$ операций. В данной главе не ставится цель продемонстрировать эффективный алгоритм решения этой задачи, используемый на практике. Вместо этого для начала предлагается разобрать несколько простых алгоритмов и проанализировать их вычислительную сложность.

6.2 Сортировка пузырьком

Для определённости предлагается всегда сортировать массив по неубыванию. Понятно, что можно получить алгоритм сортировки по невозрастанию из алгоритма сортировки по неубыванию, если заменить для элементов массива отношение порядка $\leq$ на $\geq$. Кроме того, в некоторых случаях проще бывает отсортировать массив по неубыванию, после чего на месте перевернуть его задом наперёд за $O(n)$ операций.

Для начала заметим, что если для каждой пары соседних объектов выполняется условие сортировки, то оно выполняется и вообще для всего массива. Верно и обратное. Иными словами,

$$\left(\forall i \in [1; n-1] \cap \mathbf{Z} : a_i \leq a_{i+1}\right) \Leftrightarrow \left(\forall i, j \in [1; n] \cap \mathbf{Z} : i < j \Rightarrow a_i \leq a_j\right).$$

Это так в силу транзитивности отношения порядка, о которой говорилось ранее.

Этот факт намекает на следующий алгоритм: найти пару соседних элементов, стоящих в неправильном порядке, и поменять их местами. Пройдёмся по массиву с начала до конца и проверим так каждую пару соседних элементов: сначала первую пару, затем вторую, и так далее. Например, для

массива $(2; 8; 4; 5; 6; 9; 1; 3; 7)$ после одного такого прохода получится массив $(2; 4; 5; 6; 8; 1; 3; 7; 9)$. Обмены происходили следующим образом:

1) (**2**; **8**; 4; 5; 6; 9; 1; 3; 7),

2) (2; **4**; **8**; 5; 6; 9; 1; 3; 7),

3) (2; 4; **5**; **8**; 6; 9; 1; 3; 7),

4) (2; 4; 5; **6**; **8**; 9; 1; 3; 7),

5) (2; 4; 5; 6; **8**; **9**; 1; 3; 7),

6) (2; 4; 5; 6; 8; **1**; **9**; 3; 7),

7) (2; 4; 5; 6; 8; 1; **3**; **9**; 7),

8) (2; 4; 5; 6; 8; 1; 3; **7**; **9**).

Жирным шрифтом выделены числа, которые сравнивались.

На один такой проход по массиву из n элементов требуется $(n-1)$ сравнение. Конечно, после такого прохода массив всё ещё далёк от того, чтобы стать отсортированным. Но кое-что всё же можно утверждать. А именно, после одного такого прохода максимальный элемент массива всегда оказывается на последней позиции. Это верно, потому что когда максимальный элемент сравнивается с предыдущим, он не поменяется с ним местами, а все последующие сравнения со следующими меньшими элементами вызовут обмен местами, так что максимальный элемент будет продвигаться всё дальше и дальше по массиву. Этот процесс можно наблюдать в примере выше.

Таким образом, после одного такого прохода на последнем месте стоит правильный элемент. Больше нет смысла сравнивать с ним предыдущий элемент. Можно считать, что неотсортированными остались только первые $(n-1)$ элементов массива. Чтобы отсортировать их, можно выполнить ещё один такой же проход, но только для неотсортированной части массива, после чего она ещё уменьшится. Понятно, что после $(n-1)$ прохода останется один элемент, который сам по себе уже отсортирован.

Итак, окончательный алгоритм сортировки пузырьком состоит в следующем. Нужно выполнить $(n-1)$ проход по неотсортированной части массива, каждый раз сравнивая соседние элементы и меняя их местами, если они стоят в неправильном порядке. После каждого прохода неотсортированная часть уменьшается, поскольку очередной максимальный элемент оказывается в конце массива. Этот алгоритм называется *сортировкой пузырьком* (англ. *bubble sort*), потому что максимальные элементы поднимаются к концу массива, подобно тому, как пузыри поднимаются к поверхности жидкости.

В листинге 36 показан пример реализации сортировки пузырьком на языке C для массива целых чисел. Массив передаётся по указателю, после указателя также передаётся количество элементов в массиве. Обмен элементов массива местами производится через третью переменную.

Листинг 36. Сортировка пузырьком

```
void bubble_sort(int* a, int n)
{
    for (int i = 1; i < n; ++i)
    {
        for (int j = 0; j < n - i; ++j)
        {
            if (a[j] > a[j + 1])
            {
                int t = a[j];
                a[j] = a[j + 1];
                a[j + 1] = t;
            }
        }
    }
}
```

Для определённости при анализе вычислительной сложности подобных алгоритмов сортировки предлагается оценивать только количество сравнений элементов массива. Очевидно, остальных операций при таком подходе не может быть асимптотически больше. Внешний цикл выполняется $(n-1)$ раз, его тело делает проход по неотсортированной части массива. В первый раз для $i=1$ выполняется $(n-1)$ сравнение, далее с каждым проходом выполняется на

одно сравнение меньше. В последний раз при $i = n - 1$ выполняется всего одно сравнение. Если обозначить $T(n)$ – количество сравнений элементов массива длины n в алгоритме сортировки пузырьком, то можно записать, что

$$T(n) = \sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i = \frac{1+n-1}{2}(n-1) = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2). \qquad (11)$$

Таким образом, можно констатировать, что вычислительная сложность алгоритма сортировки пузырьком составляет $O(n^2)$ и не зависит от самого массива. Можно после каждого прохода проверять, изменился ли вообще массив, и останавливать алгоритм, если никаких изменений не произошло. Тогда, по крайней мере, в лучшем случае, когда входной массив уже отсортирован, вычислительная сложность составит $O(n)$, но вычислительная сложность в худшем случае и в среднем от этого не изменится.

Конечно, этот алгоритм потребляет $O(n)$ памяти, чтобы хранить массив, но нужно отметить, что он выполняет сортировку на месте и потребляет при этом $O(1)$ дополнительной памяти. То есть кроме входного массива заводятся всего лишь три переменные. Тот факт, что алгоритм выполняет сортировку *на месте* (англ. *in-place*) означает, что он изменяет входные данные, а не возвращает новые.

Пример 22

Задача. Отсортировать массив $(3; 2; 6; 1; 5; 4)$ по неубыванию с помощью сортировки пузырьком. Отдельно продемонстрировать каждое произведённое сравнение и состояние массива на каждом шаге работы алгоритма.

Решение. Массив состоит из шести элементов, так что алгоритм сортировки пузырьком сделает $6 \cdot 5/2 = 15$ сравнений. Ниже приведены все состояния массива до и после сравнения, а жирным шрифтом выделены сравниваемые элементы. В квадратных скобках указана ещё не отсортированная часть массива.

1) ([**3**; **2**; 6; 1; 5; 4]) → ([**2**; **3**; 6; 1; 5; 4])

2) ([2; **3**; **6**; 1; 5; 4]) → ([2; **3**; **6**; 1; 5; 4])

3) ([2; 3; **6**; **1**; 5; 4]) → ([2; 3; **1**; **6**; 5; 4])

4) ([2; 3; 1; **6**; **5**; 4]) → ([2; 3; 1; **5**; **6**; 4])

5) ([2; 3; 1; 5; **6**; **4**]) → ([2; 3; 1; 5; **4**; **6**])

6) ([**2**; **3**; 1; 5; 4]; 6) → ([**2**; **3**; 1; 5; 4]; 6)

7) ([2; **3**; **1**; 5; 4]; 6) → ([2; **1**; **3**; 5; 4]; 6)

8) ([2; 1; **3**; **5**; 4]; 6) → ([2; 1; **3**; **5**; 4]; 6)

9) ([2; 1; 3; **5**; **4**]; 6) → ([2; 1; 3; **4**; **5**]; 6)

10) ([**2**; **1**; 3; 4]; 5; 6) → ([**1**; **2**; 3; 4]; 5; 6)

11) ([1; **2**; **3**; 4]; 5; 6) → ([1; **2**; **3**; 4]; 5; 6)

12) ([1; 2; **3**; **4**]; 5; 6) → ([1; 2; **3**; **4**]; 5; 6)

13) ([**1**; **2**; 3]; 4; 5; 6) → ([**1**; **2**; 3]; 4; 5; 6)

14) ([1; **2**; **3**]; 4; 5; 6) → ([1; **2**; **3**]; 4; 5; 6)

15) ([**1**; **2**]; 3; 4; 5; 6) → ([**1**; **2**]; 3; 4; 5; 6)

Как видно, массив стал полностью отсортированным уже после 10-го шага, то есть треть шагов алгоритма была выполнена впустую. Небольшая модификация алгоритма могла бы решить эту проблему.

6.3 Сортировка вставками

Хотя алгоритм сортировки пузырьком известен, как самый простой для понимания алгоритм сортировки массива, вряд ли человек стал бы пользоваться именно им, если бы ему потребовалось отсортировать что-нибудь вручную. На практике при упорядочивании предметов люди обычно интуитивно используют алгоритм, называемый *сортировкой вставками* (англ. *insertion sort*). Идея этого алгоритма заключается в поддерживании отсортированной части массива и последовательных вставках очередного элемента в эту отсортированную часть. Классическим примером использования этого алгоритма на практике является процедура сортировки игральных карт в руке.

Пусть отсортированная часть массива всегда располагается в начале массива. Изначально эта часть состоит из одного первого элемента: один элемент всегда отсортирован. На каждом шаге алгоритма первый элемент из неотсортированной части вставляется на своё место в отсортированной части массива, так чтобы для отсортированной части поддерживалось условие сортировки. Так с каждым шагом отсортированная часть увеличивается на один элемент. Вставка происходит по алгоритму, подобный которому был реализован в листинге 21, то есть элементы, следующие после заданной позиции, сдвигаются вправо.

В листинге 37 приведена реализация алгоритма сортировки вставками на языке C. Как и в листинге 36, массив целых чисел передаётся по указателю, после чего передаётся количество элементов в нём. Переменная `i` внешнего цикла отвечает за количество элементов в отсортированной части массива. Внутренний цикл нужен для сдвига элементов массива право, пока не освободится место для очередного элемента `t`. В теле цикла переменная `j` уменьшается на единицу в том же операторе, в котором происходит сдвиг очередного элемента массива.

Листинг 37. Сортировка вставками

```
void insertion_sort(int* a, int n)
{
    for (int i = 1; i < n; ++i)
    {
        int t = a[i];
        int j = i - 1;
        while (j >= 0 && a[j] > t)
        {
            a[j + 1] = a[j--];
        }
        a[j + 1] = t;
    }
}
```

Понятно, что в худшем случае, когда массив изначально отсортирован по убыванию, внутренний цикл всегда будет помещать очередной элемент в начало массива, и все остальные элементы придётся сдвигать. На первом шаге

нужно будет сдвинуть один элемент, на втором – два, и так далее. При каждом сдвиге выполняется и сравнение очередного элемента массива: на i-ой итерации внешнего цикла отсортированная часть массива содержит i элементов, и с каждым из них будет произведено сравнение. Всего внешний цикл вызывается $(n-1)$ раз. Как видно, для худшего случая рассуждения о количестве операций сравнения, выполняемых этим алгоритмом, полностью совпадают с рассуждениями о количестве операций (11), выполняемых алгоритмом сортировки пузырьком. Таким образом, вычислительная сложность этого алгоритма в худшем случае составляет $O(n^2)$.

В лучшем случае массив уже отсортирован, так что внутренний цикл вообще ни разу не выполнится, поскольку в условии этого цикла проверятся, не стоят ли соседние элементы массива в неправильном порядке. В этом случае, произойдёт всего $(n-1)$ такое сравнение, так что вычислительная сложность алгоритма сортировки вставками в лучшем случае составляет $O(n)$, что довольно приятно. Однако в среднем всё равно очередной элемент может с равной возможностью оказаться на любой позиции отсортированной части массива, так что вычислительная сложность в этом случае составляет $O(n^2)$.

Что касается потребления памяти, здесь алгоритм сортировки вставками мало отличается от алгоритма сортировки пузырьком. В целом, конечно, он потребляет $O(n)$ памяти, чтобы хранить входной массив, но сортировка выполняется на месте, и, что важно, при этом используется $O(1)$ дополнительной памяти на хранение нескольких переменных. Правда, в этом случае одна из переменных временно хранит элемент массива, но от неё при желании можно отказаться.

Пример 23

Задача. Отсортировать массив $(3; 2; 6; 1; 5; 4)$ по неубыванию с помощью сортировки вставками. Отдельно продемонстрировать каждую

вставку элемента на определённую позицию и состояние массива на каждом шаге работы алгоритма.

Решение. На рисунке 7 демонстрируется работа каждого шага внешнего цикла алгоритма сортировки вставками. Прямоугольником выделена уже отсортированная часть массива. Стрелкой показано, как очередной элемент из неотсортированной части перемещается на своё место в отсортированную часть массива.

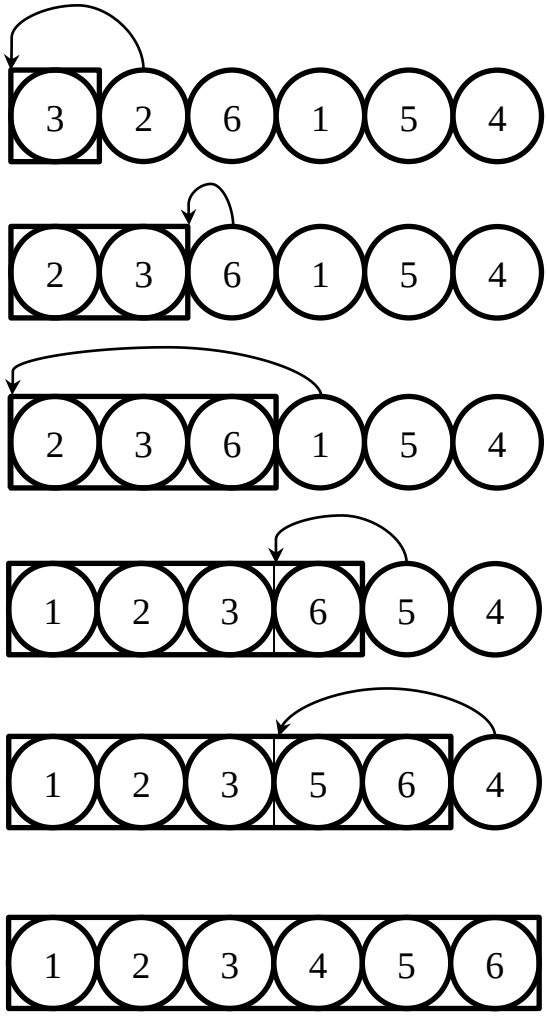

Рисунок 6 – Сортировка вставками

Напоследок стоит отметить преимущества алгоритма сортировки вставками перед алгоритмом сортировки пузырьком.

1. Лучшее время работы в лучшем случае.
2. Высокая эффективность для уже частично отсортированных массивов.

3. Способность сортировать массив по мере его считывания по одному элементу, иными словами для работы очередного шага алгоритма нет необходимости знать весь массив целиком. Алгоритм, отличающийся возможностью обрабатывать данные по мере их чтения, в западной литературе носит название *online algorithm*.

6.4 Сортировка выбором

Ещё один простой алгоритм сортировки, который часто приходит в голову не знакомым ни с какими алгоритмами сортировки программистам, основан на алгоритме поиска минимального или максимального элемента массива. Если найти в массиве, например, минимальный элемент, то его позиция в отсортированном массиве очевидна: он должен стоять первым. После этого, первый элемент можно считать отсортированным, и приступить к поиску минимального элемента в оставшейся части массива. Алгоритм сортировки массива, состоящий в последовательном выборе минимального элемента в неотсортированной части массива и перемещении его в начало этой неотсортированной части, называется сортировкой выбором (англ. *selection sort*).

Общая схема работы алгоритма сортировки выбором очевидна. Пусть i – это количество элементов в отсортированной части массива. Изначально $i = 0$. Чтобы отсортировать массив нужно проделать следующее.

1. Найти минимальный элемент среди элементов с индексами $[i; n-1] \cap \mathbf{Z}$.
2. Поменять его местами с i-ым элементом.
3. Увеличить i на единицу и отсортировать оставшуюся часть массива тем же самым алгоритмом.

Как и в алгоритме сортировки вставками, после каждого шага алгоритма отсортированная часть массива будет увеличиваться на 1 элемент. Когда останется неотсортированным последний элемент, алгоритм можно остановить: этот элемент максимальный и стоит на своей позиции.

Пример программной реализации алгоритма сортировки выбором приведён в листинге 38 в виде исходного кода функции на языке C. Эта функция принимает указатель на массив целых чисел и количество элементов в этом массиве. Внутренний цикл отвечает за поиск индекса минимального элемента, а не его значения, поскольку после этого необходимо произвести обмен, однако алгоритм поиска индекса минимального элемента мало отличается от алгоритма поиска значения максимального элемента, который был приведён в листинге 24.

Листинг 38. Сортировка выбором

```
void selection_sort(int* a, int n)
{
    for (int i = 0; i < n - 1; ++i)
    {
        int mini = i;
        for (int j = i + 1; j < n; ++j)
        {
            if (a[j] < a[mini])
            {
                mini = j;
            }
        }
        int t = a[mini];
        a[mini] = a[i];
        a[i] = t;
    }
}
```

Видно, что количество итераций циклов не зависит от значений элементов массива: оно зависит только от количества элементов n. Внешний цикл, как и в других рассмотренных алгоритмах сортировки, выполняет $(n-1)$ итерацию. Внутренний цикл для $i=0$ выполняет $(n-1)$ сравнение, для $i=1$ – $(n-2)$ сравнения, и так далее, пока на последней итерации для $i=n-1$ он не выполнит одно сравнение. Эта схема опять же ничем не отличается от количества сравнений, выполняемых двумя другими рассмотренными ранее алгоритмами, так что для неё справедлива оценка вычислительной сложности

(11). Итак, можно констатировать, что вычислительная сложность алгоритма сортировки выбором составляет $O\left(n^2\right)$, причём она эта оценка справедлива и в лучшем случае, и в худшем случае, и в среднем.

Однако одно преимущество у этого алгоритма всё же есть. Видно, что для доступа к массиву он использует только две высокоуровневые операции: выяснить, верно ли что i-ый элемент массива меньше j-го, и поменять местами i-ый элемент массива с j-ым. При этом, хотя он и выполняет $O\left(n^2\right)$ сравнений, он обходится выполнением лишь $O\left(n\right)$ обменов. Из листинга 38 видно, что он проделывает ровно $\left(n-1\right)$ обмен. Причём, иногда он делает обмены элемента с самим собой, от чего можно избавиться, добавив ещё одну проверку. Не то чтобы операция обмена двух элементов массива местами была столь вычислительно сложной, чтобы всеми силами стремиться минимизировать количество таких операций, но надо признать, что эта операция выполняется дольше, чем операция сравнения двух элементов массива друг с другом.

Что касается использования памяти, то здесь картина довольно знакомая. Алгоритм сортировки выбором использует $O\left(n\right)$ памяти, чтобы хранить входной массив, но он выполняет сортировку на месте, используя при этом $O\left(1\right)$ дополнительной памяти. Как видно, приведённые алгоритмы сортировки хоть и не слишком хороши, но оптимальны, по крайней мере, по использованию памяти.

Пример 24

Задача. Отсортировать массив $\left(3;\,2;\,6;\,1;\,5;\,4\right)$ по неубыванию с помощью сортировки выбором. Отдельно продемонстрировать каждый обмен элементов массива местами и состояние массива на каждом шаге работы алгоритма.

Решение. На рисунке 6 демонстрируется работа каждого шага внешнего цикла алгоритма сортировки выбором. Прямоугольником выделена уже отсортированная часть массива. Стрелкой показано, какие элементы меняются местами. Серым цветом выделен минимальный элемент в неотсортированной части массива. Когда 5 из 6 первых элементов отсортированы правильно, то, очевидно, и последний элемент тоже стоит на своё месте.

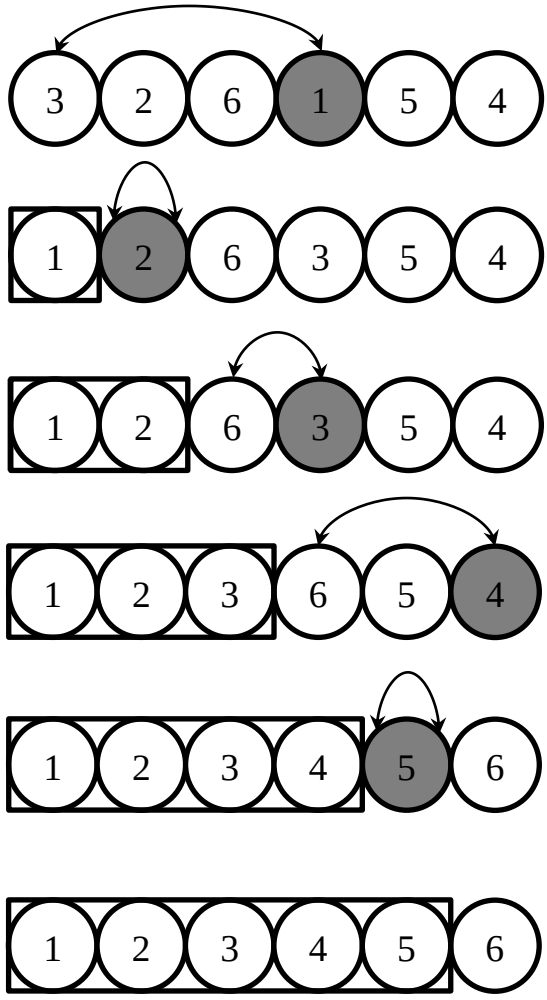

Рисунок 7 – Сортировка выбором

Упражнения для самостоятельной работы

1. Отсортируйте массив $(5; 4; 2; 0; 1; 6; 6; 4; 1; 6)$ по неубыванию, а затем по невозрастанию.

2. Модифицируйте алгоритм сортировки пузырьком, реализованный в листинге 36, чтобы он останавливался, если при очередном проходе по массиву

не было сделано ни одного обмена. Оцените вычислительную сложность полученного алгоритма в лучшем случае, в худшем случае и в среднем. Какие у него преимущества и недостатки по сравнению с оригиналом?

3. Модифицируйте алгоритм сортировки вставками, реализованный в листинге 37, чтобы он использовал при обращении к массиву только две высокоуровневые операции: выяснить, верно ли что i-ый элемент массива не больше, чем j-ый, и поменять местами i-ый элемент массива с j-ым. Оцените вычислительную сложность полученного алгоритма в лучшем случае, в худшем случае и в среднем. Какие у него преимущества и недостатки по сравнению с оригиналом?

4. Модифицируйте алгоритм сортировки выбором, реализованный в листинге 38, чтобы он не выполнял лишних обменов элементов массива местами. Докажите, что полученный алгоритм вообще выполняет наименьшее количество обменов, необходимых для сортировки любого входного массива.

5. Сортировка перемешиванием (англ. *cocktail sort, shaker sort*) имеет два отличия от пузырьковой сортировки: во-первых, она попеременно проходит массив слева направо и справа налево, а не каждый раз слева направо, во-вторых, она не проходит те участки на концах массива, которые ранее уже выглядели отсортированными. Напишите исходный код функции, реализующей сортировку перемешиванием. Асимптотически оцените вычислительную сложность этого алгоритма и объём используемой памяти.

6. Сортировка расчёской (англ. *comb sort*) отличается от пузырьковой сортировки тем, что сначала производится сравнение элементов на некотором расстоянии, большем единицы, потом на каждом шаге расстояние уменьшается и последний проход делается как в сортировке пузырьком. Напишите исходный код функции, реализующей сортировку расчёской. Асимптотически оцените вычислительную сложность этого алгоритма и объём используемой памяти. Докажите, что этот алгоритм действительно сортирует массив.

7. На каждой итерации алгоритма глупой сортировки (англ. *stupid sort*) в массиве линейным поиском обнаруживается пара соседних элементов, стоящих в неправильном порядке, после чего эти элементы меняются местами. Напишите исходный код функции, реализующей глупую сортировку. Асимптотически оцените вычислительную сложность этого алгоритма и объём используемой памяти.

8. Гномья сортировка (англ. *gnome sort*) отличается от сортировки вставками тем, что вместо вставки очередного элемента в отсортированную часть массива происходит серия обменов этого элемента с предыдущими элементами, подобно алгоритму пузырьковой сортировки. При этом используется всего один цикл вместо двух вложенных циклов. Напишите исходный код функции, реализующей гномью сортировку. Асимптотически оцените вычислительную сложность этого алгоритма и объём используемой памяти.

9. Сортировка Шелла (англ. *Shellsort*) отличается от сортировки вставками тем, что сначала сравниваются и сортируются между собой значения, стоящие один от другого на некотором расстоянии, с каждой итерацией расстояние уменьшается, и для расстояния 1 выполняется сортировка вставками. Напишите исходный код функции, реализующей сортировку Шелла.

10. Предложите настолько неэффективный алгоритм сортировки массива, насколько сможете, и оцените его вычислительную сложность. Чем хуже будет вычислительная сложность этого алгоритма, тем лучше, но он, тем не менее, должен правильно работать для любого массива из любого количества элементов.

11. Напишите исходный код программы, принимающей на вход конечную последовательность из n целых чисел $\{a_i\}_{i=1}^{n}$ ($1 \le n \le 10^5$, $-10^9 \le a_i \le +10^9$), и выводящей «`YES`», если эта последовательность является упорядоченной по неубыванию, либо «`NO`» в противном случае.

Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

12. Напишите исходный код программы, принимающей на вход конечную последовательность из n целых чисел $\{a_i\}_{i=1}^{n}$ ($1 \le n \le 10^3$, $-10^9 \le a_i \le +10^9$), и выводящей ту же последовательность, отсортированную по невозрастанию. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

13. Напишите исходный код программы, принимающей на вход конечную последовательность из n целых чисел $\{a_i\}_{i=1}^{n}$ ($1 \le n \le 10^5$, $-10^9 \le a_i \le +10^9$), и выводящей границы самого длинного отсортированного по неубыванию участка массива. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

7 Быстрые алгоритмы сортировки

7.1 Сортировка слиянием

Приведённые в главе 6 алгоритмы никогда не используются на практике, поскольку их вычислительная сложность в среднем $O\left(n^2\right)$. Они были приведены лишь с целью демонстрации, как одну и ту же задачу можно решить разными алгоритмами. С их помощью можно отсортировать массив из тысячи элементов на обычном домашнем компьютере менее чем за секунду. На практике же массивы могут быть гораздо длиннее, а задача сортировки возникает довольно часто, так что не удивительно, что были разработаны быстрые алгоритмы сортировки, вычислительная сложность которых значительно ниже квадрата. Они сложнее для понимания, но зато работают существенно быстрее.

Для начала рассмотрим следующую задачу. Допустим, есть массив a из n элементов линейно упорядоченного множества A, причём в нём участок из m первых элементов уже отсортирован, а также уже отсортирован участок с m-го до последнего элемента. То есть массив состоит из двух уже заранее отсортированных участков. Требуется отсортировать этот массив.

Пример такого массива: $\left(3; 6; 7; 10; -1; 0; 4; 12\right)$. Здесь $n=8$, $m=4$, то есть первая и вторая половины уже отсортированы. Понятно, что отсортировать такой массив проще, чем в общем случае, когда никакой дополнительной информации не имеется. Например, первым в отсортированном массиве должен идти один из элементов -1 и 3, потому что остальные, очевидно, ещё больше их. После того, как стало ясно, что первым должен идти элемент -1, имеет смысл сравнить 3 со следующим элементом из второй половины: с 0. Когда все элементы из одной половины закончатся, нужно просто дописать все элементы из оставшейся половины.

Общая идея алгоритма слияния двух частей массива схематично проиллюстрирована на рисунке 8. Допустим, нужно отсортировать участок массива с l-го элемента включительно до r-го элемента исключительно. Заведём две переменные, отвечающие за индексы элементов массива: $i = l$ для левой половины и $j = m$ для второй. На каждом шаге работы алгоритма будем сравнивать i-ый и j-ый элементы массива друг с другом, меньший из них записывать в другой массив и увеличивать соответствующую переменную на единицу, пока одна из переменных не выйдет за границы своей части массива. После этого просто допишем в конец результирующего массива все элементы из оставшейся части. Такой подход в программировании, когда алгоритм строится на одновременном проходе двумя индексами и сдвигании нужного из них, называется *двумя указателями*.

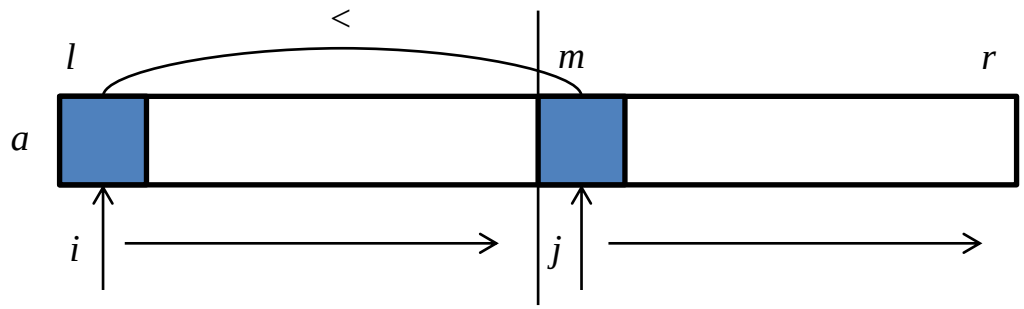

Рисунок 8 – Слияние двух частей массива

Пример реализации слияния двух отсортированных частей массива на языке C приведён в листинге 39. Приведённая функция принимает указатель на массив, элемент, с которого начинается первый отсортированный участок, элемент, с которого начинается второй отсортированный участок, и правая граница второго отсортированного участка исключительно (то есть на самом деле последний элемент второго отсортированного участка имеет на единицу меньший индекс). Вспомогательный массив `b` выделяется с помощью функции `malloc`, хотя можно было использовать оператор `new` или просто объект класса `vector`. Для итерирования по вспомогательному массиву используется третий индекс `k`. Второй и третий циклы нужны, чтобы дописать оставшиеся

элементы во вспомогательный массив: реально выполняется только один из них. Последний цикл переписывает результат слияния обратно в исходный массив. В конце функции выделенная память освобождается.

Листинг 39. Слияние двух частей массива

```
void merge(int* a, int left, int mid, int right)
{
    int* b = (int*)malloc((right - left) * sizeof(int));
    int i = left;
    int j = mid;
    int k = 0;
    while (i < mid && j < right)
    {
        if (a[i] <= a[j])
        {
            b[k++] = a[i++];
        }
        else
        {
            b[k++] = a[j++];
        }
    }
    while (i < mid)
    {
        b[k++] = a[i++];
    }
    while (j < right)
    {
        b[k++] = a[j++];
    }
    for (k = left; k < right; ++k)
    {
        a[k] = b[k - left];
    }
    free(b);
}
```

Для оценки времени работы алгоритма слияния нужно заметить, что три первых цикла проходят весь участок массива ровно один раз, а последний цикл – ещё раз. Из второго и третьего цикла реально выполняется только один, поскольку для выхода из первого цикла одно из условий входа во второй и в третий циклы должно было нарушиться. Таким образом, можно заключить, что

общая вычислительная сложность алгоритма слияния составляет $O(r-l)$, то есть линейно зависит от количества элементов на всём заданном участке массива. Если слияние выполняется на всём массиве целиком, то вычислительная сложность такой процедуры составляет $O(n)$.

Нужно отметить, что также эта функция требует $O(r-l)$ дополнительной памяти на хранение вспомогательного массива `b`. В худшем случае, придётся создавать массив, по размеру совпадающий с исходным массивом, то есть будет потрачено $O(n)$ памяти. Это ключевое неприятное отличие от ранее рассмотренных алгоритмов.

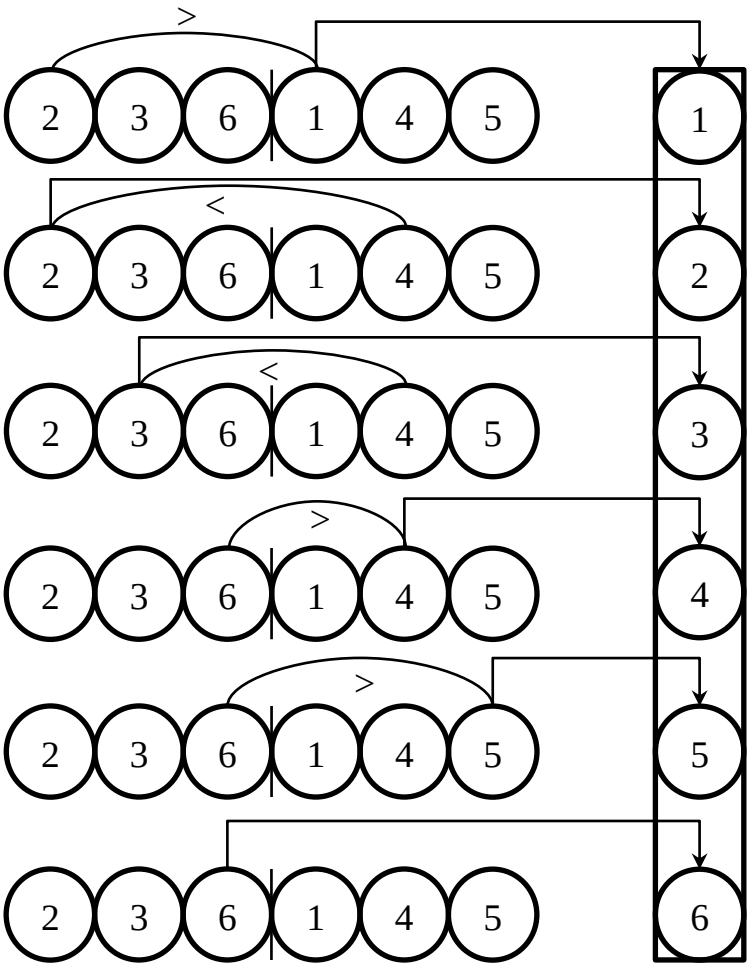

Рисунок 9 – Пример слияния двух частей массива

Пример 25

Задача. Отсортировать массив $(2; 3; 6; 1; 4; 5)$ путём слияния двух уже отсортированных половин. Отдельно графически продемонстрировать каждый шаг работы алгоритма.

Решение. На рисунке 9 демонстрируется работа каждого шага алгоритма слияния. Дугой показано, какие элементы сравниваются, а стрелкой показано, как меньший элемент переписывается во вспомогательный массив. На последнем шаге никакие элементы не сравниваются: просто правый индекс вышел за пределы правой части массива, и оставшийся элемент из левой части просто дописывается в конец вспомогательного массива.

Конечно, алгоритм слияния двух отсортированных участков массива сам по себе ещё не является алгоритмом сортировки. Он оставляет без ответа один важный вопрос: как вообще получилось так, что две части массива стали отсортированными? Однако если учесть, что получив участок массива, состоящий из двух отсортированных участков, мы сможем за линейное время отсортировать этот участок, можно синтезировать быстрый алгоритм сортировки массива методом «разделяй и властвуй».

Метод «разделяй и властвуй» (англ. *divide and conquer*) – это подход к синтезу быстрых алгоритмов, заключающийся в разбиение исходной задачи на несколько подзадач меньшего размера, в рекурсивном решении этих задач тем же алгоритмом, и в последующем объединении результатов решения для получения решения исходной задачи. Этот подход позволяет синтезировать алгоритмы с низкой вычислительной сложностью. Проблемой обычно является лишь разработка быстрого алгоритма объединения результатов решения подзадач, но в нашем случае выше уже был разработан такой алгоритм. Метод «разделяй и властвуй» кроме алгоритмов быстрой сортировки массива позволяет синтезировать алгоритмы умножения длинных чисел, быстрых ортогональных преобразований, а также некоторые сложные алгоритмы вычислительной геометрии.

Итак, алгоритм сортировки слиянием (англ. *merge sort*) заключается в следующем.

1. Если сортируемый участок массива состоит из одного элемента, то он и так отсортирован.
2. Если в сортируемом участке больше одного элемента, разбить сортируемый участок на две примерно равные части.
3. Отсортировать левую часть этим же алгоритмом.
4. Отсортировать правую часть этим же алгоритмом.
5. Выполнить слияние двух отсортированных участков массива.

Листинг 40. Сортировка слиянием

```
void merge_sort(int* a, int left, int right)
{
    if (right - left <= 1)
    {
        return;
    }
    int mid = ((left + right) >> 1);
    merge_sort(a, left, mid);
    merge_sort(a, mid, right);
    merge(a, left, mid, right);
}
```

Таким образом, сам алгоритм сортировки слиянием при наличии самой процедуры слияния выглядит довольно простым. Его программная реализация в виде функции на языке C приведена в листинге 40. Эта функция принимает указатель на массив из целых чисел, а также левую границу сортируемого участка включительно и правую границу исключительно, как и в самой процедуре слиянием. Такой подход к выбору границ позволяет проще записывать некоторые формулы в программе. База рекурсии проверяет, что сортируемый участок состоит из одного элемента. Средний элемент выбирается, как середина сортируемого участка, но вместо деления пополам выполняется битовый сдвиг вправо. В конце вызывается функция слияния из листинга 39.

Для того чтобы оценить вычислительную сложность этого алгоритма, нужно составить рекуррентное соотношение для неё. Пусть алгоритм сортировки слиянием выполняет $T(n)$ операций сравнения элементов массива, где n – это длина сортируемого участка массива. Предполагается, что первый вызов функции из листинга 40 выполняется для всего массива, то есть в виде `merge_sort(a, 0, n)`. Сам алгоритм делает два рекурсивных вызова для двух одинаковых по длине половин сортируемого участка, после чего выполняет слияние. Допустим, что сама процедура слияния выполняет n операций сравнения элементов (больше она точно не может выполнять), тогда можно записать

$$T(n) = 2T\left(\frac{n}{2}\right) + n. \tag{12}$$

Если же $n = 1$, то элементы массива не сравниваются вообще, то есть

$$T(1) = 0. \tag{13}$$

К сожалению, это не однородное линейное рекуррентное соотношение с постоянными коэффициентами, так что его решение не столь очевидно. Придётся немного подумать о том, сколько операций сравнения выполняется для разных длин, и сколько рекурсивных вызовов делается. Понятно, что для каждой заданной длины сортируемого участка, в конечном счёте, выполняется столько вызовов, что сливать приходится весь массив целиком. Причём всего таких размеров участков столько, сколько раз нужно n поделить пополам, прежде чем получится 1, то есть $\log_2 n$. Поскольку полное слияние всего массива занимает $O(n)$ операций, то можно предположить, что

$$T(n) = Cn\log_2 n, \tag{14}$$

где C – некоторая константа, значение которой неважно для асимптотической оценки.

К сожалению, оценка (14) не была явно выведена из рекуррентного соотношения (12), а была получена из эвристических соображений. Чтобы

показать, что она действительно правильная, предлагается подставить её в соотношение (14) и проверить, что уравнение сходится. Во-первых,

$$T(1) = C \cdot 1 \cdot \log_2 1 = C \cdot 0 = 0,$$

то есть начальное условие (13) подходит. Во-вторых,

$$Cn\log_2 n = 2C\frac{n}{2}\log_2 \frac{n}{2} + n,$$

$$C\log_2 n = C(\log_2 n - \log_2 2) + 1,$$

$$C\log_2 n = C\log_2 n - C + 1,$$

$$C = 1.$$

Таким образом, точным и единственным решением рекуррентного соотношения (12) с начальным условием (13) является

$$T(n) = n\log_2 n. \qquad (15)$$

Это означает, что реальная вычислительная сложность алгоритма сортировки слиянием составляет $O(n\log n)$. Это, естественно, намного лучше, чем $O(n^2)$ поскольку позволяет сортировать массивы длиной до 10^7 элементов на домашнем компьютере менее чем за секунду, поскольку $\log_2 10^7 < 25$.

Единственным недостатком этого алгоритма, из-за которого его не используют на практике, является требование $O(n)$ дополнительной памяти для хранения вспомогательных массивов. Не то чтобы сохранение памяти было настолько приоритетной задачей, просто существуют быстрые алгоритмы сортировки массивов, не использующие дополнительной памяти, так что их использование приоритетно.

Нужно отметить, что одновременно может быть создан только один вспомогательный массив, поскольку вызов функции слияния происходит после рекурсивных вызовов, а память в ней очищается. Можно избавиться от лишних операций выделения и освобождения памяти, выделив глобальный вспомогательный массив один раз и используя его в рекурсивных вызовах. Более того, при желании можно вообще избавиться от использования

дополнительной памяти и выполнять сортировку слияниями на месте, но при этом алгоритм существенно усложняется.

7.2 Быстрая сортировка Хоара

Существует более простой в реализации алгоритм быстрой сортировки массива, который к тому же не выделяет явным образом дополнительную память, хотя и расходует стек на рекурсивные вызовы. Этот алгоритм, по всей видимости, является исторически первым из когда-либо созданных человеком быстрых алгоритмов сортировки, поэтому он называется просто *быстрой сортировкой* (англ. *quicksort*). Он был разработан британским учёным сэром Чарльзом Энтони Ричардом Хоаром, также известным как Tony Hoare, в его бытность студентом Московского государственного университета. В отличие от приведённых ранее алгоритмов, этот алгоритм довольно популярен и часто применяется на практике, поскольку действительно эффективен для многих входных данных, хотя и не лишён определённых недостатков.

Быстрая сортировка Хоара также основана на подходе «разделяй и властвуй», но разделение задачи на подзадачи происходит не по индексу, как при сортировке слияниями, а по значениям элементов. Как обычно, участок из одного элемента считается отсортированным и служит базой рекурсии. На участке из двух и более элементов один из элементов выбирается в качестве *опорного* элемента (англ. *pivot*), после чего элементы на этом участке переставляются таким образом, чтобы слева от опорного элемента стояли элементы, не большие его, а справа – не меньшие его. После этого остаётся только отсортировать части массива слева и справа от опорного элемента тем же алгоритмом. Нужно понимать, что в процессе перестановки элементов для достижения указанного условия, сам опорный элемент тоже может перемещаться.

Осталось только придумать, как быстро переставить элементы так, чтобы слева от опорного элемента оказались элементы не большие его, а справа – не меньшие. На самом деле, это можно сделать за линейное время с помощью

двух указателей, как и в случае с сортировкой слиянием, но теперь указатели будут идти навстречу друг другу. Предлагается просто идти по массиву с двух сторон и менять элементы местами, если левый не меньше опорного, а правый – не больше.

Листинг 41. Быстрая сортировка Хоара

```
void quick_sort(int* a, int left, int right)
{
    if (left >= right)
    {
        return;
    }
    int i = left;
    int j = right;
    int pivot = a[left];
    while (i <= j)
    {
        if (a[i] < pivot)
        {
            ++i;
        }
        else if (a[j] > pivot)
        {
            --j;
        }
        else
        {
            int t = a[i];
            a[i] = a[j];
            a[j] = t;
            ++i;
            --j;
        }
    }
    quick_sort(a, left, j);
    quick_sort(a, i, right);
}
```

В листинге 41 приведена функция на языке C, реализующая быструю сортировку Хоара. Она принимает указатель на массив и границы участка массива, который требуется отсортировать. На сей раз обе границы включительно, то есть это индексы первого и последнего элементов

сортируемого участка. Если очередной элемент массива равен опорному, то обмен местами всё равно происходит. Это нужно, чтобы оба индекса гарантированно сдвинулись с изначальной позиции, поскольку хотя бы один опорный элемент в массиве точно присутствует. Цикл выполняется, пока левый индекс не окажется справа от правого. После этого выполняются два рекурсивных вызова от соответствующих участков массива.

На рисунке 10 схематично показано, как происходит проход по массиву в быстрой сортировке: представлены состояния массива до и после прохода. Дугой показано, какие элементы сравниваются между собой. После прохода, в некотором месте в массиве оказывается опорный элемент, показанный синим на нижнем изображении. Слева от него изображена зелёная область, в которой элементы не превосходят опорный элемент, а справа изображена красная область, в которой элементы не меньше опорного элемента. В итоге выполняется два рекурсивных вызова как раз для зелёной и для красной областей.

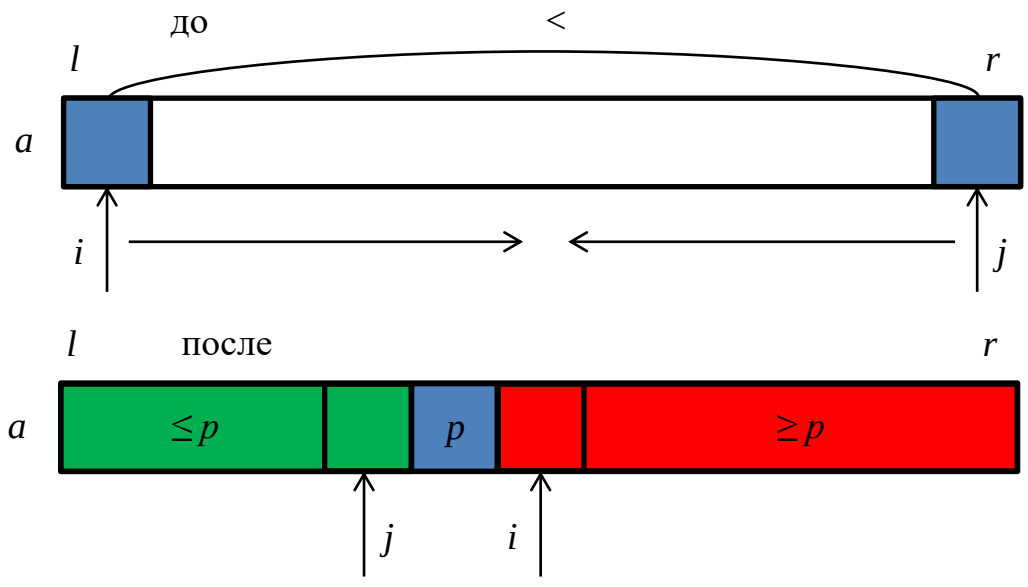

Рисунок 10 – Быстрая сортировка Хоара

Понятно, что проход перед рекурсивными вызовами выполняется за $O(n)$ операций, поскольку при этом осуществляется ровно один проход по массиву.

Однако определение общей вычислительной сложности алгоритма быстрой сортировки представляет определённую сложность. Всё зависит от того, как участки массива делятся на части. В лучшем случае при каждом рекурсивном вызове участок массива будет делиться ровно пополам, и тогда для оценки вычислительной сложности справедливы все рассуждения из раздела 7.1. Таким образом, вычислительная сложность этого алгоритма в лучшем случае составляет $O(n\log n)$, поскольку для неё справедлива оценка (15).

К сожалению, в худшем случае всё не так складно. Допустим, при каждом рекурсивном вызове опорным элементом оказывается минимальный элемент массива, тогда левая часть всегда будет пустой, а правая на один элемент короче исходного сортируемого участка. Так каждый рекурсивный вызов будет вызывать ту же функцию для участка на единицу меньшей длины, что приводит к $O(n)$ рекурсивным вызовам, в каждом из которых выполняется $O(n)$ операций. То есть в данном случае для вычислительной сложности справедливы выкладки (11). Таким образом, вычислительная сложность алгоритма быстрой сортировки Хоара в худшем случае составляет $O(n^2)$, что не лучше, чем у сортировки пузырьком. Правда, отличие состоит в том, что вычислительная сложность быстрой сортировки Хоара в среднем составляет $O(n\log n)$ и на большинстве реальных данных этот алгоритм работает очень быстро.

Оценка использованной памяти похожа на оценку вычислительной сложности. Во-первых, алгоритм тратит $O(n)$ памяти на хранение массива. В лучшем случае участки массива делятся примерно пополам, так что глубина рекурсии, как в случае сортировки слиянием, составляет $O(\log n)$, и каждый вызов сохраняет фрейм активации на стек. Получается, что в лучшем случае алгоритм быстрой сортировки Хоара использует $O(n)$ памяти на хранение входных данных и $O(\log n)$ дополнительной памяти. В худшем случае делается

последовательных $O(n)$ рекурсивных вызовов, так что в худшем случае алгоритм быстрой сортировки Хоара использует $O(n)$ памяти на хранение входных данных и $O(n)$ дополнительной памяти, что не лучше, чем при сортировке слияниями. Хорошая новость состоит в том, что в среднем этот алгоритм всё же затрачивает $O(\log n)$ дополнительной памяти.

В итоге алгоритм быстрой сортировки Хоара обладает следующими достоинствами.

1. Простота понимания и реализации.
2. Возможность реализации не только для массивов, но и для других слабых линейных структурах данных.
3. Одна из самых высоких производительностей на случайных данных.

Тем не менее, алгоритм быстрой сортировки Хоара не лишён одного очень важного недостатка. На массивах, сконструированных специальным образом, вычислительная сложность этого алгоритма составляет $O(n^2)$, что не лучше, чем у большинства неэффективных алгоритмов сортировки. Например, для реализации из листинга 41 для этого достаточно передать в функцию уже отсортированный массив из n элементов. Тогда при каждом вызове левая часть будет пустой, а правая – на единицу меньшего размера.

Видно, что это связано с выбором опорного элемента. В листинге 41 в качестве опорного элемента просто выбирается первый элемент на сортируемом участке. Поэтому уже отсортированный массив является худшим случаем. Другие варианты выбора опорного элемента приведены ниже.

1. Последний элемент.
2. Средний элемент по индексу.
3. Средний элемент по значению из первых трёх элементов.
4. Средний элемент по значению из первых пяти элементов.
5. Среднее арифметическое между первым и последним элементом.
6. Среднее арифметическое между первыми тремя элементами.

7. Среднее арифметическое всех элементов на участке.
8. Случайный элемент.

Как видно, в некоторых случаях можно выбирать и элемент, которого нет в сортируемом участке, но тогда может потребоваться существенная модификация приведённой в листинге 41 реализации алгоритма. В некотором случае выбор опорного элемента может сам по себе представлять сложную задачу. Нужно помнить, что нельзя тратить на него больше $O(n)$ операций.

Для всех приведённых способов выбора опорного элемента всё равно можно предложить такой способ конструирования массивов, что вычислительная сложность алгоритма быстрой сортировки Хоара для таких массивов будет $O(n^2)$. Это очень серьёзный недостаток алгоритма, из-за которого вместо него используют другие алгоритмы, не имеющие такого недостатка. Даже алгоритм сортировки слиянием всегда работает за $O(n\log n)$ операций не зависимо от значений элементов в массиве.

Для решения этой проблемы могут применяться следующие подходы.

1. Случайное перемешивание элементов массива перед сортировкой.
2. Отслеживание глубины рекурсии и переход на другой алгоритм сортировки при достижении определённой глубины рекурсии.

Пример 26

Задача. Отсортировать массив (4; 8; 7; 9; 3; 1; 6; 5; 2) с помощью алгоритма быстрой сортировки Хоара. Отдельно графически продемонстрировать каждый шаг работы алгоритма.

Решение. На рисунке 11 демонстрируется работа каждого шага алгоритма быстрой сортировки Хоара. Стрелками показано, какие элементы меняются местами. Серым цветом выделены опорные элементы. Прямоугольными рамками выделены участки массива, для которых выполняются рекурсивные вызовы. Работа разных рекурсивных вызовов показана одновременно, хотя фактически они выполняются последовательно.

Как можно заметить, на небольших массивах существенное преимущество этого алгоритма в скорости работы не слишком ощущается.

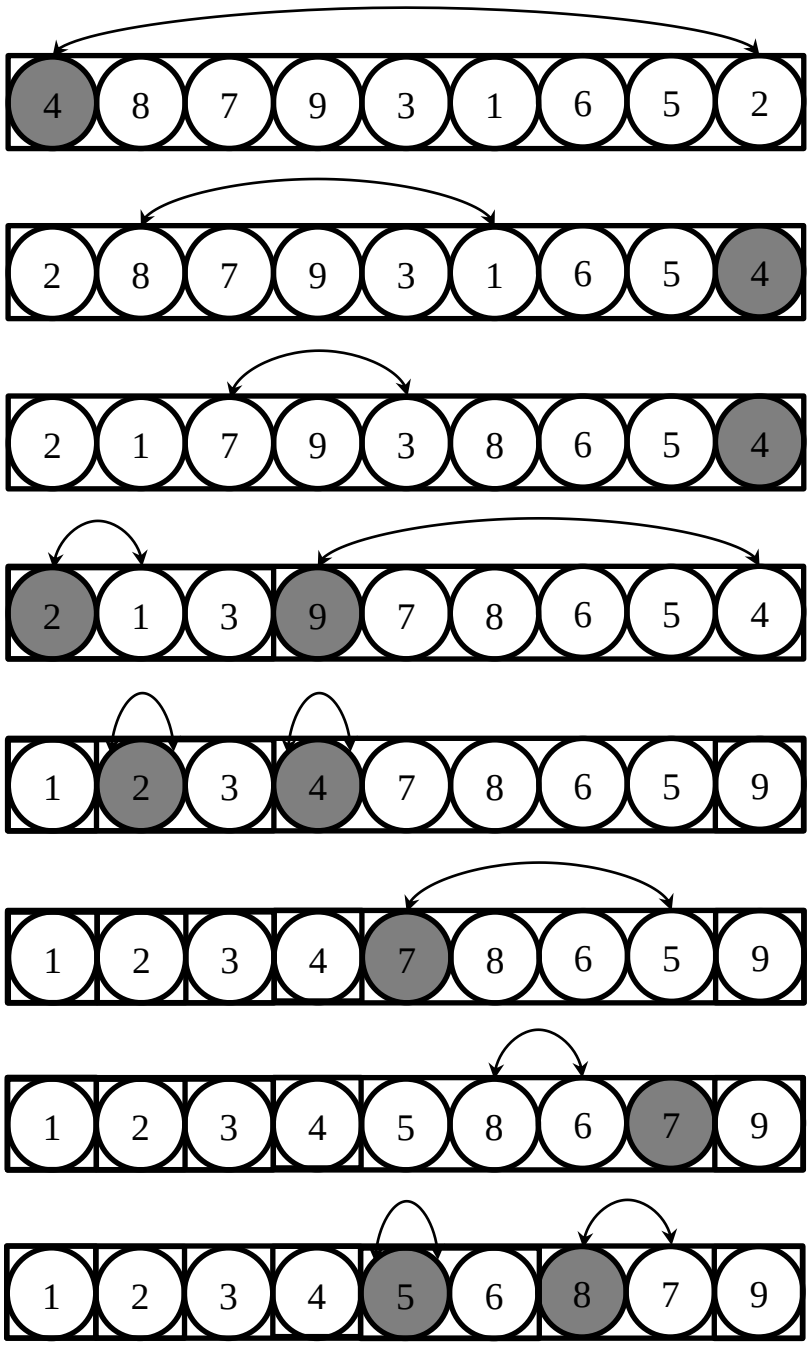

Рисунок 11 – Пример быстрой сортировки Хоара

7.3 Наилучшая асимптотика для алгоритмов сортировки

В данной главе было показано, как использование подхода «разделяй и властвуй» позволяет построить быстрые алгоритмы сортировки массива, вычислительная сложность которых значительно ниже, чем у их эвристических

аналогов, которые приходят в голову в первую очередь. Вычислительная сложность быстрых алгоритмов сортировки составляет $O(n\log n)$, в то время как вычислительная сложность обычных менее эффективных алгоритмов сортировки составляет $O(n^2)$. Возникает закономерный вопрос: можно ли ещё снизить эту вычислительную сложность и насколько вообще низкой может быть вычислительная сложность алгоритма сортировки массива. Может ли, скажем, отсортировать массив за $O(n)$ операций? Например, алгоритм сортировки вставками из листинга 37 в лучшем случае работает именно столько. Можно ли добиться, чтобы в среднем алгоритм также работал за линию?

На все эти вопросы есть вполне определённый ответ. Простые теоретические выкладки позволяют решить эту проблему и отыскать наилучшую вычислительную сложность, которую может иметь алгоритм сортировки массива. Правда при этом подразумевается лишь алгоритм определённого вида.

Как можно было заметить, все представленные алгоритмы для доступа к массиву используют две высокоуровневые операции: выяснить, верно ли что i-ый элемент массива не больше, чем j-ый, и поменять местами i-ый элемент массива с j-ым. Допустим, что это единственные две операции, которые допустимо выполнять с массивом при его сортировке. Понятно, что операций сравнения всегда требуется больше, чем операций обмена: если знать правильный порядок элементов, то переставить их в нужном порядке можно за $O(n)$ операций перестановки, как это делалось в алгоритме сортировки выбором из листинга 38. Предлагается оценить минимальное количество операций сравнения элементов массива, необходимое, чтобы отсортировать массив из n элементов.

Для небольших длин массивов можно оценить минимальное количество операций точно. В таблице 10 показано наименьшее количество сравнений

$\hat{T}(n)$, необходимых для сортировки массива из n элементов в худшем случае. Кажется, что эта функция растёт довольно медленно. Сначала она вообще напоминает последовательность нечётных чисел с нулём. Но насколько медленно она растёт? К сожалению, отыскание значения функции $\hat{T}(n)$ для заданного n само по себе представляет собой вычислительно сложную задачу.

Таблица 10 – Наименьшее количество сравнений для сортировки массива

n	1	2	3	4	5	6
$\hat{T}(n)$	0	1	3	5	7	10

Теорема о вычислительной сложности наиболее эффективного алгоритма сортировки. Количество сравнений элементов массива в алгоритме сортировки, который использует для доступа к массиву только операции сравнения и обмена, не может быть меньше, чем $\Theta(n\log n)$.

Здесь впервые была явно использована Θ-нотация вместо O-нотации, поскольку $O(n\log n)$ – это оценка сверху. Использовать оценку сверху в качестве оценки снизу было бы крайне некорректно: не ясно, что означает фраза «вычислительная сложность не может быть меньше, чем $O(n\log n)$», ведь функция $T(n)=n$ также является $O(n\log n)$, и при этом она меньше.

Доказательство. Обозначим за $\hat{T}(n)$ минимальное количество сравнений, необходимое для сортировки массива из n элементов. Для начала нужно заметить, что существует $2^{\hat{T}(n)}$ различных результатов $\hat{T}(n)$ сравнений, при том что n элементов могут быть упорядочены $n!$ способами. Если $2^{\hat{T}(n)} < n!$, то, даже зная результат всех $\hat{T}(n)$ сравнений, мы не имеем возможности различить, которая из перестановок имеется в виду, поскольку согласно принципу Дирихле нескольким перестановкам соответствует один и тот же результат всех $\hat{T}(n)$ сравнений. Понятно, что если алгоритм должен сортировать массив, то такая ситуация для него недопустима, то есть

$$2^{\hat{T}(n)} \geq n!.$$

Отсюда можно заключить, что

$\hat{T}(n) \geq \log_2 n!$.

Остаётся только получить хорошую оценку снизу для $\log_2 n!$.

Конечно, существует формула Стирлинга, согласно которой

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + O\left(\frac{1}{n}\right)\right) = \Theta\left(\sqrt{n}\left(\frac{n}{e}\right)^n\right),$$

откуда сразу же можно получить правильный результат, но доказательно этой формулы достаточно нетривиально. Допустим, что читателю она не известна.

Вместо этого, имея в виду, что логарифм – это монотонно возрастающая функция, можно записать

$$\hat{T}(n) \geq \log_2 n! \geq \ln n! = \ln \prod_{k=1}^{n} k = \sum_{k=1}^{n} \ln k = \ln 1 + \sum_{k=2}^{n} \ln k \geq \int_1^n \ln x \, \mathrm{d}x =$$

$$= \left(x \ln x - x\right)\Big|_1^n = n \ln n - n - \ln 1 + 1 = n \ln n - n + 1.$$

Понятно, что для больших значений n только слагаемое $n \ln n$ вносит определяющий вклад, так что можно записать, что

$\hat{T}(n) \geq \Theta(n \log n)$.

Известно, что существуют алгоритмы сортировки, работающие за $O(n \log n)$, поэтому

$\hat{T}(n) = \Theta(n \log n)$.

Что и требовалось доказать.

Таким образом, алгоритмы сортировки слиянием из листинга 40 и быстрой сортировки Хоара из листинга 41 достигают наилучшей возможной вычислительной сложности для алгоритма сортировки такого типа. Разработать алгоритм, который сортировал бы массив ещё эффективнее, в общем случае невозможно. Минимальное количество сравнений, необходимое, чтобы отсортировать массив из n элементов растёт, как $n \log n$.

В заключение хотелось бы отметить, что на практике практически никогда не приходится самостоятельно реализовывать сортировку массива. В подавляющем большинстве языков программирования эта сортировка уже реализована в одной из функций стандартной библиотеки. В языке C для этого определена функция `qsort`, объявленная в файле `stdlib.h`. В языке C++ в файле `algorithm` определена функция `sort`, принимающая итераторы на начало и конец сортируемого объекта. При написании программ на C++ для сортировки массивов и других линейных структур данных рекомендуется использовать именно эту функцию.

7.4 Сортировка подсчётом за линейное время

В разделе 7.3 было показано, что массив нельзя отсортировать быстрее, чем за $O(n\log n)$ операций в общем случае. В этом разделе предлагается сделать всего одно небольшое допущение, которое позволяет построить линейный алгоритм сортировки массива. Это не противоречит теоретическому результату из раздела 7.3, поскольку для доступа к массиву здесь используются не только операции сравнения и обмена, кроме того имеется дополнительная априорная информация о массиве.

Для начала предлагается задуматься над следующей задачей: дан массив из n целых чисел, каждое из которых может быть либо нулём, либо единицей. Требуется отсортировать этот массив. Понятно, что никто не будет использовать для этого ни один из приведённых выше алгоритмов сортировки. Вместо этого достаточно просто посчитать количество нулей и количество единиц, после чего записать в этот массив столько нулей и столько единиц, сколько в нём было, но чтобы сначала шли нули, а после них – единицы. Такой алгоритм сортирует массив из n нулей и единиц за $O(n)$ операций.

Неплохой результат, но что если в массиве всё же содержатся не только нули и единицы. Допустим, в массиве из n элементов могут содержаться лишь первые m неотрицательных целых чисел. То есть если считать массивом

конечную последовательность $\{a_i\}_{i=1}^{n}$, то все её элементы отвечают условию $a_i \in [0; m-1] \cap \mathbf{Z}$. Это уже довольно типичная ситуация, распространённая на практике. Однако в этой ситуации можно использовать тот же подход, что использовался для сортировки массива из нулей и единиц.

Листинг 42. Сортировка подсчётом за линейное время

```
void counting_sort(int* a, int n, int m)
{
    int* c = (int*)malloc(m * sizeof(int));
    for (int j = 0; j < m; ++j)
    {
        c[j] = 0;
    }
    for (int i = 0; i < n; ++i)
    {
        ++c[a[i]];
    }
    int i = 0;
    for (int j = 0; j < m; ++j)
    {
        for (int k = 0; k < c[j]; ++k)
        {
            a[i++] = j;
        }
    }
    free(c);
}
```

Заводится отдельный массив $\{c_j\}_{j=0}^{m-1}$, такой что c_j – это количество элементов равных j в массиве a. Изначально полагается, что все c_j равны нулю. Далее предлагается пройтись по массиву a и для каждого $i \in [1; n] \cap \mathbf{Z}$ увеличить на единицу элемент c_{a_i}. После этого все c_j станут хранить необходимые величины, то есть

$$c_j = \sum_{i=1}^{n} \delta_{a_i, j},$$

где $\delta_{i,j}$ – это символ Кронекера, то есть

$$\delta_{i,j} = \begin{cases} 1, & i = j; \\ 0, & i \neq j. \end{cases}$$

Наконец нужно последовательно для всех $j \in [0; m-1] \cap \mathbf{Z}$ записать c_j раз в массив a число j.

В листинге 42 приведён исходный код функции на языке C, реализующей сортировку подсчётом. Эта функция принимает указатель на массив, количество элементов в нём и количество различных значений, которые могут принимать элементы массива. Память под дополнительный массив, в котором хранится количество элементов с заданными значениями, выделяется с помощью функции `malloc`. После выделения памяти, элементы этого массива явно заполняются нулями. В конце память явным образом освобождается.

Функция содержит три цикла. Первый, очевидно, выполняет m итераций, второй – n итераций. Последний цикл несколько интереснее. Внешний цикл выполняет m итераций, но общее количество всех итераций внутреннего цикла составляет n, поскольку

$$\sum_{j=0}^{m-1} c_j = n,$$

ведь элементы c_j увеличиваются на единицу ровно n раз. Таким образом, общая вычислительная сложность алгоритма составляет $O(n+m)$, так как каждый оператор выполняется не больше, чем $\max\{n, m\}$ раз.

Эта функция затрачивает $O(n)$ памяти на хранение входного массива и $O(m)$ дополнительной памяти на хранение вспомогательного массива. Таким образом, общей расход памяти составляет $O(n+m)$. В некоторых случаях, при этом происходит явный перерасход памяти, например, если некоторые значения из множества $[0; m-1] \cap \mathbf{Z}$ никогда не принимаются.

Конечно, это довольно специфическая сортировка, которая имеет очевидные ограничения на использование: элементы массива не должны

принимать широкий диапазон значений. Однако она позволяет отсортировать массив из 10^6 элементов, каждый из которых может принимать значения из множества $\left[0;10^6\right] \cap \mathbf{Z}$, менее чем за секунду на обычном домашнем компьютере. Этот алгоритм может быть эффективнее обычных быстрых сортировок, работающих за $O(n\log n)$, для случаев, когда элементы массива принимают значения из небольшого диапазона. Нужно учитывать, что хотя вычислительная сложность алгоритма сортировки подсчётом и зависит от n линейно, она, кроме этого, также линейно зависит и от величины самих элементов массива.

Пример 27

Задача. Построить массив подсчёта c, получающийся в функции из листинга 42 для массива (0; 3; 6; 1; 8; 4; 5; 7; 0; 6; 7; 4; 1; 1; 6; 8; 3; 7; 6; 9).

Решение. Очевидно, элементы массива принимают целые значения, не меньшие нуля и не большие девяти. Запишем массив подсчёта c из 10 элементов, посчитав для каждого элемента массива, сколько раз он в нём встречается. Получим массив (2; 3; 0; 2; 2; 1; 4; 3; 2; 1). То есть в исходном массиве содержатся 2 нуля, 3 единицы, 0 двоек и так далее.

Ответ: (2; 3; 0; 2; 2; 1; 4; 3; 2; 1).

Упражнения для самостоятельной работы

1. Отсортируйте массив (73; 24; 34; 31; 30; 28; 98; 28; 57; 49; 40; 52; 94; 23; 38; 23; 66; 10; 73; 50; 34; 81; 27; 25; 15; 94; 36; 13; 35; 61; 28; 28) по неубыванию.

2. Профессор О. П. Рометчивый построил алгоритм отыскания суммы всех элементов массива на основе метода «разделяй и властвуй». Если участок массива, на котором нужно найти сумму, состоит из одного элемента, то эта сумма равна этому элементу. Если же на этом участке больше одного элемента, то можно разбить его пополам, посчитать суммы на каждой половине тем же самым алгоритмом, после чего общую сумму вычислить, как сумму этих двух

сумм. Он утверждает, что поскольку для массива из n элементов глубина рекурсии составляет $O(\log n)$, а пересчёт результата после рекурсивных вызовов и вовсе выполняется за $O(1)$, то общая вычислительная сложность приведённого алгоритма составляет $O(\log n)$, что эффективнее, чем просто вычислять сумму элементов в цикле. Где ошибка в этих рассуждениях, и какова на самом деле вычислительная сложность приведённого алгоритма?

3. Разработайте и реализуйте не рекурсивную версию алгоритма сортировки слиянием. Оцените вычислительную сложность полученного алгоритма в лучшем случае, в худшем случае и в среднем. Какие у него преимущества и недостатки по сравнению с оригиналом?

4. Модифицируйте алгоритм сортировки слиянием, так чтобы он использовал $O(1)$ дополнительной памяти. Оцените вычислительную сложность полученного алгоритма в лучшем случае, в худшем случае и в среднем. Какие у него преимущества и недостатки по сравнению с оригиналом?

5. Сортировка естественным слиянием отличается от обычной сортировки слиянием тем, что использует в качестве элементарных отсортированных участков массива не участки из одного элемента, а реальные отсортированные участки исходного массива. Предварительно эти участки находятся за $O(n)$ и используются в алгоритме. Напишите исходный код программы, реализующей сортировку естественным слиянием. Асимптотически оцените вычислительную сложность этого алгоритма и объём используемой памяти.

6. Идея блочных алгоритмов сортировки состоит в разбиении исходного массива из n элементов примерно на $\sqrt{n}$ блоков по $\sqrt{n}$ элементов в каждом. Каждый блок отдельно сортируется любым алгоритмом, например, пузырьком, а затем отсортированные блоки последовательно сливаются друг с другом с помощью алгоритма, реализованного в листинге 39. Напишите исходный код программы, реализующей блочную сортировку. Асимптотически оцените вычислительную сложность этого алгоритма и объём используемой памяти.

7. Докажите, что вычислительная сложность алгоритма быстрой сортировки Хоара в среднем составляет $O(n\log n)$.

8. Реализуйте алгоритм быстрой сортировки Хоара, в котором выбор опорного элемента осуществляется, как среднего по значению из трёх первых элементов на сортируемом участке. Асимптотически оцените вычислительную сложность этого алгоритма и объём используемой памяти. Какие у него преимущества и недостатки по сравнению с оригиналом? Предложите алгоритм построения массива из n элементов, для которого полученный алгоритм сортировки будет работать за $O(n^2)$.

9. Реализуйте алгоритм быстрой сортировки Хоара, в котором выбор опорного элемента осуществляется, как среднего по индексу на сортируемом участке. Асимптотически оцените вычислительную сложность этого алгоритма и объём используемой памяти. Какие у него преимущества и недостатки по сравнению с оригиналом? Предложите алгоритм построения массива из n элементов, для которого полученный алгоритм сортировки будет работать за $O(n^2)$.

10. Модифицируйте алгоритм сортировки подсчётом, так чтобы он использовал работал для массива, состоящего из элементов, принимающих значения из множества $[a;b]\cap\mathbf{Z}$. Асимптотически оцените вычислительную сложность полученного алгоритма и объём потребляемой памяти.

11. Блуждающая сортировка (stooge sort) работает рекурсивно, подобно сортировке слияниями. Сначала она меняет местами первый и последний элемент текущего участка, если они стоят в неправильном порядке, а затем делает три рекурсивных вызова самой себя: для первых двух третей участка, для последних двух третей участка и снова для первых двух третей участка. Напишите исходный код программы, реализующей блуждающую сортировку. Асимптотически оцените вычислительную сложность этого алгоритма и объём используемой памяти.

12. При поразрядной сортировке (*radix sort*) сравниваются отдельные разряды чисел в массиве. Сначала в начало массива перемещаются все числа с нулями в старшем разряде, а с единицей остаются в конце. Затем числа с нулями и единицами в старшем разряде отдельно сортируются по второму разряду и так далее. Напишите исходный код программы, реализующей поразрядную сортировку. Асимптотически оцените вычислительную сложность этого алгоритма и объём используемой памяти.

13. Напишите исходный код программы, принимающей на вход конечную последовательность из n целых чисел $\{a_i\}_{i=1}^{n}$ ($1 \le n \le 10^5$, $-10^9 \le a_i \le +10^9$), и выводящей все пары элементов этой последовательности, дающих в сумме ноль. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

14. Напишите исходный код программы, принимающей на вход две конечных последовательности из n целых чисел: $\{a_i\}_{i=1}^{n}$ и $\{b_i\}_{i=1}^{n}$ ($1 \le n \le 10^5$, $-10^9 \le a_i, b_i \le +10^9$), и выводящей «`YES`», если элементы первой последовательности можно переставить местами так, чтобы получить элементы второй последовательности, либо «`NO`» в противном случае. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

15. Напишите исходный код программы, принимающей на вход две конечных последовательности из n целых чисел: $\{a_i\}_{i=1}^{n}$ и $\{b_i\}_{i=1}^{n}$ ($1 \le n \le 10^5$, $-10^9 \le a_i < b_i \le +10^9$), где $[a_i; b_i]$ – это отрезки на прямой, и выводящей точку на этой прямой, покрытую наибольшим количеством отрезков. В случае наличия нескольких правильных ответов можно вывести любой из них. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

8 Строковые алгоритмы

8.1 Хранение символов в памяти компьютера

Если задуматься, существует только два типа данных, которые действительно отличаются по способу хранения: целые числа, описанные в главе 1, и числа с плавающей запятой, описанные в главе 2. Все остальные типы данных, включая указатели, символы и логические значения, основаны на этих двух типах, в особенности на целых числах. Уже упоминалось, что указатель – это просто целое число, обозначающее адрес некоторого байта в памяти. Логические переменные в C++ также занимают один байт и являются целыми числами: в языке C вообще не было логического типа, а вместо него использовались целые числа.

С самого начала развития компьютерной техники и до сих пор текстовый интерфейс является одним из основных способов взаимодействия пользователя с компьютером. Даже в современных операционных системах, снабжённых богатым графическим интерфейсом, остаётся возможность взаимодействовать с ресурсами компьютера через текстовую консоль, причём некоторыми возможностями операционной системы можно воспользоваться только через неё. Это относится даже к семейству операционных систем Windows, которые изначально были ориентированы на оконный графический интерфейс.

В этой связи операционные системы до сих пор поддерживают текстовое взаимодействие со всеми запущенными программами. При запуске программы из консоли в неё можно передать аргументы командной строки в виде последовательности строк, разделённых пробелами. После запуска программы, она может читать текстовые данные из консоли и выводить текст в консоль. Базовые возможности ввода-вывода, поддерживаемые стандартными библиотеками большинства языков программирования, включая C и C++, как раз заключаются в получении аргументов командной строки, считывании

текстовых данных из стандартного потока ввода и запись текстовых данных в стандартный поток вывода.

Логично, что раз всё взаимодействие с пользователем исторически было построено на текстовых данных, с самого начала программы поддерживали возможность работы с символами, включающими как буквы и цифры, так и знаки пунктуации и некоторые более экзотические вещи. Кроме печатных символов, подразумевающих некоторое графическое представление, исторически были определены управляющие символы, которые не имеют формы, но предназначены для управления отображением текстовых данных и для некоторых других целей. Символы стали простейшим структурным элементом на пути слияния естественного языка, на котором говорят люди, и машинного языка, представляющего собой инструкции, исполняемые компьютером.

Итак, в информатике *символы* (англ. *characters*) – это единицы информации, соответствующие графическим единицам письменной речи. Главное, что нужно понимать про хранение символов в памяти компьютера, – все они хранятся как обычные целые числа, представляющие собой коды символов. Графическое представление символов обычно не хранится вместе с кодами символов, а хранится отдельно. Это представление используется только в момент отображения символа с заданным кодом на дисплее или на печати.

Отображение, задающее соответствие между кодами символов и их графической интерпретацией, называется *набором символов* (англ. *character set*) или *кодировкой символов* (англ. *character encoding*). Исторически сложилось, что существует множество различных кодировок символов, в которых одним и тем же кодам соответствуют разные символы. При отображении символа с заданным кодом на экране используется некоторая заданная кодировка. Разработчик программы должен сам заботиться, чтобы символы выводились именно в той кодировке, в которой они хранятся по его замыслу, и выполнять необходимые преобразования между кодировками. Распространённые ошибки,

при которых символы хранятся в одной кодировке, а выводятся в другой, приводят к образованию нечитаемого текста.

Одна из самых распространённых до сегодняшнего дня кодировок символов появилась в 60-х годах XX века на заре развития электронных вычислительных машин. Она носит название ASCII (*American Standard Code for Information Interchange*) и определяет коды для наиболее распространённых печатных и управляющих символов. Среди символов, входящих в ASCII, содержатся следующие символы.

1. Управляющие символы.
2. Арабские цифры.
3. Буквы латинского алфавита.
4. Распространённые математические символы.
5. Распространённые знаки препинания.
6. Некоторые другие символы.

Таблица 11 – ASCII

	.0	.1	.2	.3	.4	.5	.6	.7	.8	.9	.A	.B	.C	.D	.E	.F
0.	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1.	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2.		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3.	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4.	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5.	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6.	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7.	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

В таблице 11 представлен набор символов ASCII. Номера символов записаны в шестнадцатеричной системе счисления: строка обозначает старший разряд кода, а столбец – младший разряд. Как видно, первые 32 символа управляющие, далее следует пробел, некоторые знаки препинания и арифметические операции, затем цифры, ещё несколько символов, затем заглавные буквы, после них ещё символы, затем строчные буквы, и, наконец, ещё несколько символов.

Всего в таблице содержатся 128 символов, то есть для их кодирования достаточно 7 бит. Однако в минимальной адресуемой единице памяти 8 бит, так что даже если кодировать символы одним байтом, всё равно можно закодировать в два раза больше символов, чем содержится в таблице ASCII. На этом факте и основано наличие большого количества разнообразных кодировок. В большинстве кодировок коды с 0 по 127 включительно означают те же символы, что и в таблице ASCII, а остальные 128 символов отличаются.

Например, как видно, в таблице ASCII нет символов кириллицы, с помощью которых преимущественно написана эта книга. Существуют кодировки, в которых символы кодируются одним байтом и для букв кириллицы предусмотрены коды, большие 127. Так сложилось, что такие кодировки отличаются в разных операционных системах: в Windows для этого предназначена кодировка Windows-1251, также известная как CP-1251, в UNIX-системах для этого была разработана кодировка KOI8-R, также имеется десяток других менее распространённых кодировок кириллицы, связанных с DOS, ISO, Macintosh и т. д.

Можно догадаться, что существует огромное множество таких кодировок для других языковых групп по всему миру, отличающихся в разных операционных системах. Ясно также, что для некоторых языков, таких как китайский и японский, все символы алфавита вообще нельзя закодировать одним байтом информации, так что для них необходимо несколько кодировок для разных групп символов. Наличие такого огромного числа кодировок приводит к проблемам разработки даже оконных приложений с версиями для множества языков. Наиболее сложная ситуация обстоит с браузерами: пользователь может открывать любые страницы в Интернете, текст на которых может быть закодирован в любой из существующих кодировок, при том что иногда кодировка даже не указывается в коде HTML-страницы.

Продвижение в решении проблемы наличия большого количества кодировок привело к появлению крайне популярной на сегодняшний день

кодировки *Юникод* (англ. *Unicode*). Идея, положенная в основу этой кодировки, состоит в кодировании отдельных символов не одним, а большим количеством байт. За счёт этого можно в единой кодировке указать коды для всех известных человечеству символов, и при этом останется ещё много незанятых кодов для будущих символов.

Традиционно считается, что каждый символ в Юникоде кодируется двумя байтами, а сам его код обозначается, соответственно, четырьмя шестнадцатеричными цифрами. Тем не менее, из-за особенностей кодирования общее количество различных символов, которые можно закодировать в Юникоде, составляет $2^{20} + 2^{16} - 2^{11} = 1112064$. Кроме того, существует несколько *форм представления Юникода* (англ. *Unicode transformation format, UTF*), самые распространённые из которых приведены ниже.

1. UTF-8. Для представления каждого символа тратится от одного до шести байт: на распространённые символы меньше, а редкие и несуществующие – больше. Кодирует все символы из ASCII-таблицы их обычными восьмибитными кодами.
2. UTF-16. Для представления большинства символов тратится два байта. Для некоторых редких символов тратится больше, поскольку общее количество символов в Юникоде превышает 2^{16}.
3. UTF-32. Для представления всех символов тратится ровно четыре байта. Позволяет записывать все символы Юникода, и ещё остаются несуществующие коды.

Для большинства ситуаций рекомендуется использовать формат UTF-8. Он полностью совместим с ASCII-таблицей и позволяет кодировать тексты достаточно компактно, на практике затрачивая наименьшее количество памяти. На деле использование Юникода при работе с символами в программировании также может представлять некоторые сложности, несмотря на широкую поддержку этой кодировки многими языками программирования, однако описание подробностей работы с Юникодом не входит в задачи данного курса.

Предлагается рассмотреть только работу с восьмибитными символами. В языках C и C++ для хранения символов предназначен тип данных `char`, который хранит целое число размером в один байт. В разделе 1.3 уже упоминалось, что это обычный целочисленный тип данных, причём в стандарте языка даже не определяется, должен ли этот тип данных хранить целое число со знаком или без знака. Можно рассчитывать, что если переменная типа `char` хранит число от 0 до 127 включительно, то это код символа из таблицы ASCII.

Символьные литералы в языках C и C++, а также во многих других C-подобных языках записываются в одиночных кавычках. Например, `'a'` соответствует латинской букве «a», а `'?'` – вопросительному знаку. Нужно помнить, что эти литералы в действительности представляют собой целые числа. С ними можно выполнять арифметические операции, операции сравнения, использовать их в качестве индексов в массивах.

Для обозначения управляющих символов в виде литералов используются так называемые *управляющие последовательности* (англ. *escape sequence*), начинающиеся с обратного слеша. В таблице 12 приведены наиболее распространённые управляющие последовательности. Например, для записи литерала символа перевода строки используется конструкция `'\n'`. Пустой символ с нулевым кодом предназначен для обозначения отсутствия символа. Это не просто непечатный символ, а, по сути, вообще не символ, а признак отсутствия символа.

Таблица 12 – Некоторые управляющие последовательности

Код	**Управляющая последовательность**	**Название символа**
00	`\0`	Пустой символ
09	`\t`	Табуляция
10	`\n`	Перевод строки
13	`\r`	Возврат каретки

Здесь нельзя не упомянуть об одной проблеме, связанной с переводом строки в текстовых файлах. Дело в том, что традиционно в операционных системах семейства Windows перевод строки в текстовых потоках обозначается парой символов перевода строки и возврата каретки в этом порядке (коды 10 и

13), а в UNIX-системах для перевода строки используется единственный символ перевода строки (только код 10). Из-за этого могут возникать некоторые проблемы при обработке текстовых файлов: в зависимости от того, в какой операционной системе был создан файл, перевод строки может обозначаться по-разному, и это нужно иметь в виду при написании программ. Конечно, многие языки программирования и библиотеки предлагают определённые решения этой проблемы.

На деле не нужно запоминать коды большинства символов, даже если они входят в таблицу ASCII. В исходном коде программ обычно есть возможность использовать непосредственно символы в качестве их кодов, как это и реализовано в C-подобных языках. На них определено обычное для чисел отношение порядка, так что можно сравнивать переменные с символьными литералами, чтобы определить в каком интервале находится символ. Например, в листинге 43 приведён исходный код функции на языке C, которая определяет, является ли заданный символ цифрой. Поскольку цифры в таблице ASCII расположены последовательно, можно просто проверить два условия, вместо сравнения символа с каждой из цифр.

Листинг 43. Проверка, является ли символ цифрой

```
int isdigit(char c)
{
    return '0' <= c && c <= '9';
}
```

Аналогично можно проверить, является ли символ буквой латинского алфавита. В листинге 44 приведён исходный код функции на языке C для подобной проверки. Здесь проверяется принадлежность символа к одному из отрезков маленьких или больших букв латинского алфавита. Нужно понимать, что хотя эти буквы в таблице ASCII и расположены по алфавиту, между большими и маленькими буквами есть ещё несколько других символов. Причём большие буквы имеют меньшие коды, чем маленькие, но этот факт можно не запоминать, поскольку приведённый способ проверки это не учитывает. И в

этой функции, в предыдущей возвращается `int`, потому что в чистом C долгое время не было логического типа данных. Результаты этих функций, тем не менее, можно использовать в качестве логических условий.

Листинг 44. Проверка, является ли символ латинской буквой

```
int isalpha(char c)
{
    return 'a' <= c && c <= 'z' || 'A' <= c && c <= 'Z';
}
```

На самом деле, реализовывать эти функции в том виде, в котором они указаны в листингах 43 и 44, нет необходимости. Эти функции именно с такими названиями реализованы в стандартных библиотеках и в языке C, и в языке C++. Также имеются полезные функции `isspace` для проверки, является ли символ пробельным, `isprint` для проверки, является ли символ печатным, и `isascii` для проверки, входит ли код символа в таблицу 11.

Также с символами можно выполнять арифметические операции, определённые для целых чисел. Например, в листинге 45 приведён исходный код функции на языке C для перевода символа, представляющего собой цифру, в численное значение этой цифры. Для этого из кода символа вычитается код цифры «0». Таким образом, символ `'0'` перейдёт в число 0, `'1'` перейдёт в 1 и т. д.

Листинг 45. Перевод кода цифры в её значение

```
int digit_to_int(char c)
{
    return c - '0';
}
```

8.2 Хранение строк в памяти компьютера

Ясно, что для хранения текстовых данных используются не отдельные символы, а целые последовательности символов, называемые *строками* (англ. *string*). Строки хранятся просто в виде массивов символов. Однако в общем случае массив – это просто указатель на область памяти. Для строк очень часто нужно точно знать количество элементов и иметь возможность точно

определять конец строки, так что в разных языках реализованы различные подходы к деталям в способах хранения строк.

1. Хранение длины строки в самом массиве символов. Например, можно первые 4 байта массива отвести под хранение текущего количества символов. Используется, например, в языке Pascal.
2. Хранение нулевого символа в конце строки. Всё, что содержится в массиве после нулевого символа, считается мусором. Используется, например, в языке C.
3. Комбинирование первых двух способов. В этом случае в начале массива хранится количество символов, а в конце – нулевой символ. Используется, например, в языке Оберон.
4. Хранение массива и количества символов в единой структуре или объекте. Используется в большинстве современных языков программирования, поддерживающих ООП, например, в Java.
5. Хранение не в виде массива, а в виде динамического связного списка. Используется в некоторых функциональных языках, например, в Haskell.
6. Отсутствие реализации хранения строк и работы со строками. В этом случае, программист сам должен хранить строки в массивах, как ему удобно, и сам должен реализовывать все операции с ними, в зависимости от выбранного им способа хранения. Это характерно для ранних этапов развития некоторых языков программирования, а также для эзотерических языков, например, Brainfuck.

В языке C нет как такового типа данных для строк. Строки хранятся, как массивы символов типа `char*`. Считается, что строка начинается с самого начала массива, а заканчивается там, где в массиве располагается символ с нулевым кодом. Такие строки называются *нуль-терминированными строками* (англ. *null-terminated string*). Все строковые литералы как в C, так и в C++ являются именно такими константными указателями на символ. Например, литерал вроде `"Hello, World!"` имеет тип `const char*`.

В листинге 46 приведён простой пример исходного кода программы на языке C, которая вводит некоторую строку и тут же выводит её на экран. Для хранения строки используется статический массив фиксированного размера. Если количество символов в считываемой строке превысит `SIZE`, то программа выйдет за границы выделенной памяти при считывании, и её поведение в этом случае не определено. Также нужно понимать, что функция `scanf` считывает один токен, то есть последовательность символов, отделённую пробельными символами или переводами строк. Для считывания строки целиком до перевода строки в языке C следует использовать функцию `gets`.

Листинг 46. Чтение и вывод строки в C

```c
#include <stdio.h>

const int SIZE = 1000000;

int main()
{
    char s[SIZE];
    scanf("%s", s);
    printf("%s\n", s);
    return 0;
}
```

В языке C++ поддерживается возможность работы со строками, имеющаяся в языке C, к тому же строковые литералы также являются нуль-терминированными строками. Однако в стандартной библиотеке языка C++ имеется класс `std::string`, предназначенный для хранения строк и работы с ними. Объекты этого класса также хранят внутри обычную нуль-терминированную строку, но они также хранят и количество символов, что позволяет более прозрачно работать со строками. При написании программ на C++ рекомендуется использовать класс `string` вместо непосредственной работы с массивами символов, как в языке C.

В листинге 47 приведён простой пример исходного кода программы на языке C++, которая считывает и выводит строку. Видно, что здесь отсутствует проблема с переполнением буфера, характерная для языка C, кроме того работа

со строкой выглядит более наглядной. Для вывода перевода строки используется `std::endl`. Здесь также считывается только один токен, как при использовании `scanf` в C. Для считывания строки до перевода строки в C++ используется функция `getline`.

Листинг 47. Чтение и вывод строки в C++

```
#include <iostream>
#include <string>

int main()
{
    std::string s;
    std::cin >> s;
    std::cout << s << std::endl;
    return 0;
}
```

Чтобы узнать длину строки в языке C используется функция `strlen`, определённая в файле `string.h`. Она работает примерно так, как написано в листинге 48. Во-первых, её небезопасно использовать, если в массиве вообще нет нулевого символа, поскольку в этом случае произойдёт выход за выделенную область памяти. Во-вторых, её вычислительная сложность для строки из n символов составляет $O(n)$, поскольку она, по сути, производит линейный поиск нулевого символа в массиве, подобный алгоритму из листинга 23. В C++ количество символов в строке `s`, являющейся объектом класса `string`, можно выяснить с помощью метода `s.size()`, который работает за $O(1)$, поскольку количество элементов хранится в самом объекте.

Листинг 48. Определение длины строки в C

```
int strlen(const char* s)
{
    int i = 0;
    while (s[i] != '\0')
    {
        ++i;
    }
    return i;
}
```

В математике тоже иногда рассматривается множество строк S над некоторым алфавитом A, представляющее собой множество конечных последовательностей различной длины, состоящих из элементов множества A. Длину строки $s \in S$ обычно обозначают $|s|$, потому что отвечает некоторым свойствам нормы векторов, в частности $\forall s \in S : |s| \geq 0$, и кроме того $\forall s \in S : |s| = 0 \Rightarrow s = ""$. Последнее означает, что только у пустой строки длина равна нулю.

Нужно также отметить, что в разных языках способы хранения строк можно разделить на две категории.

1. Изменяемые (англ. *mutable*) строки. Отдельные символы в строках можно изменять. В C и C++ все строки изменяемые.
2. Неизменяемые (англ. *immutable*) строки. Такие строки не допускают никаких модификаций и изменений символов. Любое изменение должно создавать другую строку. За счёт этого удаётся более быстро выполнять некоторые операции, например, взятие подстроки. Такой способ хранения строк используется, например, в языке Java.

8.3 Работа с файлами

Всё это время в качестве основных источников ввода и вывода данных в программе рассматривалась текстовая консоль. Это действительно самый популярный способ ввода и вывода, по крайней мере, он точно был таким раньше. Второй по популярности способ ввода и вывода данных заключается в чтении из файлов и записей в файлы. Подавляющее большинство прикладных и системных программ работают с файлами для сохранения данных, обеспечения персистентности, например, для хранения пользовательских настроек между запусками программы.

Файл (англ. *file*) – это внешний ресурс, представляющий собой именованную область данных. Как правило, имеется в виду область данных на некотором внешнем накопителе, например, на жёстком диске, хотя в общем

случае файлы могут иметь самую разную природу. Фактически файлы представляют собой мельчайшие единицы данных, хранящиеся операционными системами. Именно операционные системы организуют доступ к файлам.

Файл является внешним ресурсом в том смысле, что доступ к нему не предоставляется непосредственно исполнителем кода программы, а запрашивается у операционной системы, то есть у среды, в которой программа исполняется. Другими примерами внешних ресурсов, предназначенных для ввода и вывода, являются сокеты, устройства, соединения с базой данных и т. д. В UNIX-системах взаимодействие с большинством внешних ресурсов реализовано именно через чтение и запись файлов.

Основной особенностью ресурсов подобного рода является тот факт, что они общие для всех одновременно исполняемых программ. В этом смысле даже динамическая память является внешним ресурсом: её выделение запрашивается у операционной системы, а после использования операционной системе сообщается об освобождении памяти. Эта особенность присуща работе со всеми внешними ресурсами: нужно запросить доступ к ним, произвести чтение или запись, после чего явным образом освободить их, чтобы другие программы также могли ими воспользоваться.

Конкретный способ организации хранения файлов в операционной системе называется *файловой системой* (англ. *file system*). Различные файловые системы задают способы именования файлов, наборы дополнительных файловых атрибутов, организацию безопасного доступа к файлам и прочее. Если речь идёт о хранении файлов на жёстких дисках, то наиболее популярными файловыми системами являются NTFS для операционных систем Windows и ext2 в операционных системах на базе ядра Linux.

Имена файлов в разных файловых системах могут формироваться по-разному. Наиболее популярной является иерархическая структура, в которой каждый файл помещён в некоторый каталог, у которого также имеется имя. Каталоги также могут лежать в каталогах. Полное или абсолютное имя файла в

таких файловых системах включает имена всех каталогов, в которых лежит этот файл и все родительские каталоги. Непосредственно же имя файла, не включающее каталог, в котором он находится, называется *относительынм именем файла*. В конце имени файла после точки обычно указывают расширение, которое означает, каким приложением следует открывать этот файл. Например, «`C:\Windows\System32\notepad.exe`» – это полное имя файла в операционных системах Windows, «`/bin/cat`» – это полное имя файла в UNIX-подобных операционных системах, а «`notepad.exe`» и «`cat`» – это их соответствующие относительные имена.

При реализации работы с файлами в исходном коде программ главное помнить об общей схеме работы с любыми внешними ресурсами, уже описанной выше.

1. Открыть файл. При этом операционная система предоставляет программе специальный *файловый дескриптор* (англ. *file descriptor*), который может в дальнейшем использоваться для операций ввода и вывода. В то же время доступ к этому файлу ограничивается для других программ, так что они не могут в полной мере использовать этот файл.
2. Произвести с файлом операции чтения или записи. При этом не следует тратить время на другие операции, ведь доступ к файлу всё это время ограничен для других программ, которые могут ожидать возможности поработать с ним.
3. Закрыть файл. При этом явным образом программа сообщает операционной системе, что работа с файлом закончена, после чего выданный ей файловый дескриптор лишается смысла, а файл вновь становится доступен для других программ. Как и при работе с памятью, частой ошибкой является отсутствие явной операции закрытия файла, что приводит к блокировке файла, так что большинство операций с ним становится невозможным.

При чтении из файлов и при записи в файлы используются те же приёмы, как и при чтении из консоли и при записи в консоль, только используется

файловый дескриптор, чтобы указать, какой именно файл имеется в виду. Различные языки программирования реализуют работу с файлами немного по-разному, но общая схема всегда походит на три пункта, приведённые выше. Иногда для работы с файлами используется концепция *потоков данных* (англ. *stream*), представляющих собой абстрактные объекты, допускающие чтение и запись. Подробное рассмотрение возможностей этой концепции выходит за рамки этой книги.

Например, в языке C для представления потоков используется тип данных `FILE`, объявленный в файле `stdio.h`, причём работа обычно осуществляется с указателями на значения этого типа. Стандартные потоки чтения и записи в этом языке называются `stdin` и `stdout`. Для открытия файла используется функция `fopen`, принимающая имя файла и режим доступа к нему в виде строк. Для чтения используется режим доступа `"r"`, а для записи – режим доступа `"w"`. Для чтения отдельных символов можно использовать, например, функцию `getc`, для чтения строк до перевода строки – функцию `fgets`, а для форматированного ввода – функцию `fscanf`. Для вывода отдельного символа можно использовать, например, функцию `putc`, для вывода строки – функцию `fputs`, для форматированного вывода – функцию `fprintf`. После завершения работы с файлом необходимо закрыть его с помощью функции `fclose`.

В листинге 49 приведён исходный код программы на языке C, которая посимвольно читает содержимое файла `input.txt` и записывает прочитанное в файл `output.txt`. Чтение осуществляется в статический массив, так что при наличии в файле больше `LIMIT` символов произойдёт выход за его границы, и поведение программы в этом случае не определено. Файлы открываются по своему относительному имени, так что поиск файлов с таким именем будет производиться в рабочей директории программы. После открытия файла проверяется, что указатель, который вернула функция `fopen`, не нулевой, поскольку в противном случае это означает, что файл открыть не

удалось. При чтении происходит одновременное присваивание результата в переменную c и проверка, что этот результат не является признаком конца файла. Такая конструкция возможна, потому что оператор присваивания возвращает ссылку на объект, в который производилось присваивание. Также нужно отметить, что функция getc в действительности возвращает значение типа int. После чтения в конец строки явным образом помещается нулевой символ, чтобы вывод этой строки можно было осуществить с помощью функции fputs.

Листинг 49. Файловый ввод и вывод в C

```c
#include <stdio.h>

#define LIMIT 1000000

int main()
{
    char s[LIMIT];
    s[0] = '\0';
    FILE* fin = fopen("input.txt", "r");
    if (fin)
    {
        int i = 0;
        int c;
        while ((c = getc(fin)) != EOF)
        {
            s[i++] = (char)c;
        }
        s[i] = '\0';
        fclose(fin);
    }
    FILE* fout = fopen("output.txt", "w");
    if (fout)
    {
        fputs(s, fout);
        fclose(fout);
    }
    return 0;
}
```

В языке C++ для чтения файлов можно использовать потоки класса ifstream, а для записи – потоки класса ofstream. Оба класса объявлены в

файле fstream. Для открытия файла достаточно просто создать объект нужного класса, передав в конструктор имя файла. Для чтения отдельных символов из потока класса ifstream можно использовать, например, метод get, для чтения строк до перевода строки – функцию getline, а для форматированного ввода – операцию >>. Для вывода отдельного символа в поток класса ofstream можно использовать, например, метод put, для вывода строки и для форматированного вывода – операцию <<. После завершения работы с файлом необходимо закрыть поток с помощью метода close.

Листинг 50. Файловый ввод и вывод в C++

```
#include <fstream>
#include <string>

int main()
{
    std::ifstream fin("input.txt");
    std::string s("");
    if (fin.is_open())
    {
        char c;
        while (fin.get(c))
        {
            s += c;
        }
        fin.close();
    }
    std::ofstream fout("output.txt");
    if (fout.is_open())
    {
        fout << s;
        fout.close();
    }
    return 0;
}
```

В листинге 50 приводится исходный код программы на языке C++, которая посимвольно считывает содержимое файла input.txt и сразу же записывает его в файл output.txt, подобно тому, как это делает программа

из листинга 49. Чтение осуществляется в строку класса `string`, так что память динамически выделяется при необходимости. Для проверки, что файл открылся корректно, используется метод `is_open`. Метод `get` возвращает сам поток, который приводится к логическому условию проверкой на конец файла. Для дописывания символа в конец строки используется переопределённая операция +=.

8.4 Конкатенация строк

Основная бинарная операция, определённая на множестве строк, – это конкатенация. *Конкатенация* (англ. *concatenation*) – это операция склеивания двух строк друг с другом, так чтобы в результирующей строке сначала шли символы первой строки, а затем – второй. Например, результатом конкатенации строк «конь» и «як» является строка «коньяк».

В математике конкатенация строк s_1 и s_2 обозначается $s_1 \cdot s_2$, как умножение чисел. Если обозначить $s(k)$ k-ый символ строки s, то конкатенация $s = s_1 \cdot s_2$ определяется так, что

$$s(k) = \begin{cases} s_1(k), & k \le |s_1|; \\ s_2(k - |s_1|), & k > |s_1|. \end{cases}$$

Эта операция отвечает следующим очевидным свойствам.

1. Ассоциативность:

   $\forall s_1, s_2, s_3 \in S : (s_1 \cdot s_2) \cdot s_3 = s_1 \cdot (s_2 \cdot s_3)$.

2. Пустая строка является нейтральным элементом:

   $\forall s \in S : s \cdot "" = s$.

3. Длина конкатенации двух строк равна сумме длин этих строк:

   $\forall s_1, s_2 \in S : |s_1 \cdot s_2| = |s_1| + |s_2|$.

На практике конкатенация часто используется, например, для конструирования сообщений, отображающихся в пользовательском интерфейсе программы. В случае, если язык программирования поддерживает изменяемые строки, то обычно есть возможность дописывать одну строку непосредственно

к другой, в противном случае конкатенация двух строк всегда порождает новую строку.

В языке C для конкатенации строк предназначена функция `strcat`, определённая в файле `string.h`. Она работает примерно как похожая функция, описанная в листинге 51. Эта функция принимает два указателя на нуль-терминированные строки и дописывает содержимое второй строки к первой. Для этого она сначала находит конец первой строки, а затем дописывает к ней вторую, пока она не закончится. В последний момент в конец первой строки дописывается нулевой символ, чтобы обозначить, где она заканчивается. Эта функция не проверяет, что для записи всех символов в конец первой строки вообще хватает выделенной области памяти, так что её использование в этом смысле не в полной мере безопасно. На рисунке 12 схематично изображена работа алгоритма из листинга 51 для конкатенации строк «`Hello`» и «`World`».

Листинг 51. Конкатенация строк

```
void strcat(char* s1, const char* s2)
{
    int i = 0;
    while (s1[i] != '\0')
    {
        ++i;
    }
    int j = 0;
    while (s2[j] != '\0')
    {
        s1[i++] = s2[j++];
    }
    s1[i] = '\0';
}
```

Видно, что первый цикл работает столько итераций, сколько символов в первой строке, а второй работает столько итераций, сколько символов во второй строке. Таким образом, общая вычислительная сложность алгоритма конкатенации двух строк, имеющих длины n и m символов соответственно, составляет $O(n+m)$, поскольку циклы идут последовательно один после

другого. Очевидно, этот алгоритм тратит $O(n+m)$ памяти на хранение входных строк, но вторая строка просто дописывается к первой, так что используется только $O(1)$ дополнительной памяти.

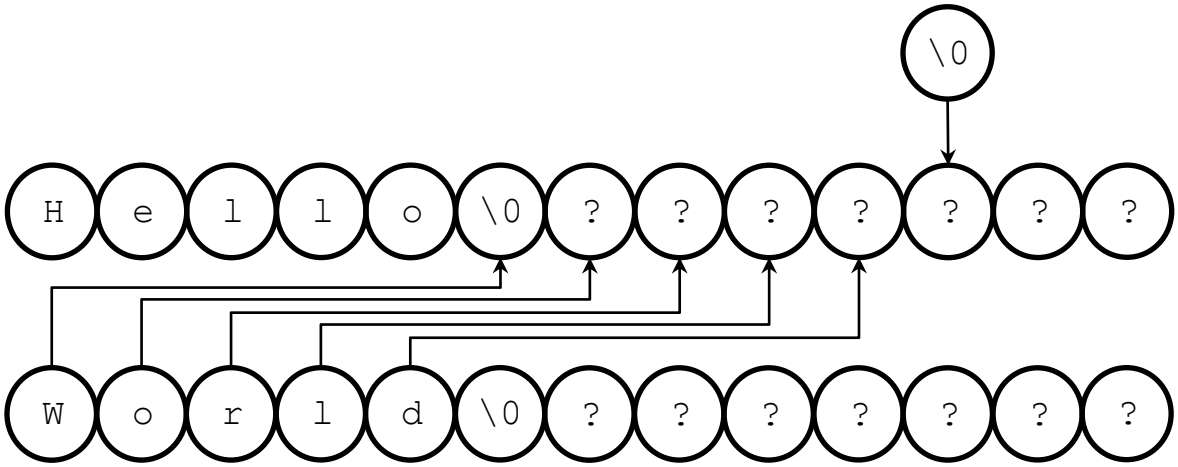

Рисунок 12 – Пример конкатенации

В языке C++ для объектов класса `string` перегружена операция `+`, так что конкатенация строк `s1` и `s2` записывается просто как `s1 + s2`. При этом строки `s1` и `s2` сами по себе не меняются, а происходит создание новой строки, являющейся результатом конкатенации строк `s1` и `s2`. Чтобы изменить строку `s1`, дописав к ней строку `s2`, как это делается в листинге 51, для объектов класса `string` в C++ достаточно написать `s1 += s2`. Понятно, что в любом случае вычислительная сложность алгоритма конкатенации остаётся $O(n+m)$.

8.5 Лексикографический порядок

На множестве строк, как и на множестве чисел, можно ввести отношение линейного порядка, отвечающее свойствам, приведённым в разделе 6.1. Этот порядок широко известен просто как алфавитный порядок, в котором слова располагаются в различных словарях и списках. Не вдаваясь в подробности, можно описать этот порядок следующим образом: слова сравниваются сначала по первой букве, при равенстве первых букв – по второй и так далее.

Рассмотрим множество S конечных последовательностей различной длины, элементы которых принадлежат некоторому линейно упорядоченному

множеству A. Обозначим k-ый элемент последовательности $s \in S$ за $s(k)$, а подпоследовательность с i-го по j-ый элемент за $s(i:j)$. *Лексикографический порядок* (англ. *lexicographical order*) задаёт линейный порядок на таком множестве последовательностей с помощью следующих правил.

1. Пустая последовательность предшествует любой другой:

   $\forall s \in S : s \neq \{\ \} \Rightarrow \{\ \} < s$.

2. Из двух последовательностей предшествует та, у которой первый элемент меньше:

   $\forall s_1, s_2 \in S : s_1(1) < s_2(1) \Rightarrow s_1 < s_2$.

3. Из двух последовательностей с одинаковыми первыми элементами предшествует та, у которой предшествует подпоследовательность без первого элемента:

   $\forall s_1, s_2 \in S : s_1(1) = s_2(1) \wedge s_1(2:|s_1|) < s_2(2:|s_2|) \Rightarrow s_1 < s_2$.

Это в определённом смысле рекурсивное определение, поскольку третье правило содержит в себе отсылку к лексикографическому порядку последовательностей меньшей длины. Если считать множество A алфавитом, состоящим из символов, а множество S – множеством строк, то введённое выше определение можно считать определением лексикографического порядка для строк. Как видно, для сравнения двух строк в лексикографическом порядке, нужно просто сравнивать их соответствующие символы, пока они не закончатся или не станут различаться, после чего меньшей будет та строка, у которой символы закончились, или у которой первый отличающийся символ меньше.

На практике множество символов – это множество их кодов в некоторой кодировке, так что с заданием порядка на них проблем не возникает. Порядок символов просто соответствует порядку хранимых вместо них целых чисел. Так для программной реализации сравнения строк в лексикографическом порядке легко построить несложный итеративный алгоритм.

Листинг 52. Сравнение строк в лексикографическом порядке

```
int strcmp(const char* s1, const char* s2)
{
    int i = 0;
    while (s1[i] != '\0' && s2[i] != '\0')
    {
        if (s1[i] < s2[i])
        {
            return -1;
        }
        if (s1[i] > s2[i])
        {
            return +1;
        }
        ++i;
    }
    if (s1[i] == '\0' && s2[i] == '\0')
    {
        return 0;
    }
    if (s1[i] == '\0')
    {
        return -1;
    }
    else
    {
        return +1;
    }
}
```

В языке C для сравнения строк предназначена функция `strcmp`, которая объявлена в файле `string.h`. Основной принцип её работы можно понять из реализации похожей функции, описанной в листинге 52. Эта функция принимает два указателя на нуль-терминированные строки и возвращает отрицательное целое число, если первая строка предшествует второй в лексикографическом порядке, положительное целое число, если вторая строка предшествует первой, либо ноль, если строки равны. Для этого она последовательно сравнивает очередные соответствующие символы двух строк, пока одна из строк не закончится. Если очередные символы отличаются, то она сразу же возвращает результат сравнения. Если символы в одной из строк

закончились, то нужно просто выяснить, в какой именно, и не закончились ли при этом символы и в другой строке тоже.

Видно, что единственный цикл в этом алгоритме работает, пока одна из строк не закончится. Таким образом, общая вычислительная сложность лексикографического сравнения двух строк, имеющих длины n и m символов соответственно, составляет $O\left(\min\{n,m\}\right)$. То есть если обе строки имеют длину n символов, то вычислительная сложность растёт линейно и составляет $O(n)$, если же одна из строк пустая, то сравнение её с любой другой строкой занимает $O(1)$. Здесь же нужно отметить, что хотя реализация этого алгоритма и тратит $O(n+m)$ памяти на хранение сравниваемых строк, она никак их не меняет и требует для работы $O(1)$ дополнительной памяти.

В языке C сравнивать строки следует только с помощью функций вроде `strcmp`. Использование для этого операций сравнения, таких как < или >, является серьёзной ошибкой, ведь они сравнивают указатели, как целые числа, а не строки, на которые они указывают. То есть даже для проверки равенства двух строк `s1` и `s2` в языке C следует писать условие `strcmp(s1, s2) == 0` вместо `s1 == s2`, поскольку в последнем случае проверяется, что эти строки реально расположены в памяти в одном и том же месте.

В языке C++ для объектов класса `string` переопределены операции сравнения, так что в дополнительных функциях нет необходимости. Для проверки того, что строка `s1` предшествует строке `s2`, можно просто записать `s1 < s2`. Также в C++ нормально сравнивать такие строки на равенство с помощью оператора ==. Конечно, при работе с указателями на обычные нуль-терминированные строки в C++ следует использовать те же способы сравнения, что и в C. При этом следует помнить, что обычные строковые литералы вроде `"Hello, World!"` являются именно указателями на массивы символов, а не объектами класса `string`.

Пример 28

Задача. Отсортировать строки «throwable», «thought», «throw», «hound» в лексикографическом порядке.

Решение. Сравним сначала первые буквы. Буква «h» стоит в алфавите раньше «t», поэтому строка «hound» однозначно предшествует всем остальным. У остальных трёх строк первые две буквы совпадают, но у слова «thought» третья буква «o», которая предшествует третьей букве в двух оставшихся словах, так что вторым должно стоять слово «thought». Из слов «throw» и «throwable» раньше должно стоять слово «throw», потому что оно короче, хотя все его буквы и совпадают с соответствующими буквами слова «throwable». Таким образом, правильный порядок: «hound», «thought», «throw», «throwable».

Ответ: «hound», «thought», «throw», «throwable».

8.6 Поиск подстроки в строке

На практике очень часто возникает необходимость поиска некоторой строки в текстовых данных. Иногда нужно просто проверить, содержит ли текст некоторое слово, а иногда даже точно установить, в каком месте это слово содержится. Многие системы управления базами данных, поисковики в Интернете и обыкновенные текстовые редакторы активно используют в своей работе эту процедуру.

Формально эту задачу можно поставить следующим образом. Пусть имеется две строки $s_1, s_2 \in S$. Задача поиска подстроки $s_2 \in S$ в строке $s_1 \in S$ состоит в нахождении натурального числа k, такого что $s_1\left(k : k + \left|s_2\right| - 1\right) = s_2$, либо в сообщении, что такое число отсутствует. В общем случае при наличии нескольких подходящих натуральных чисел k, нас интересует любое из них.

Распространённый алгоритм решения этой задачи подобен линейному поиску в массиве, описанному в листинге 23, с тем отличием, что проверяется не один элемент, а целая последовательность из нескольких элементов. Для решения задачи нужно просто пройтись по первой строке и для каждого символа первой строки проверить, не начинается ли с него подстрока,

совпадающая со второй строкой. Для проверки совпадения подстроки со строкой придётся пройтись по второй строке и в случае совпадения всех символов сразу сообщить ответ.

В листинге 53 приведён исходный код функции на языке C для поиска подстроки в строке. Эта функция принимает два указателя на нуль-терминированные строки и возвращает индекс символа в первой строке, с которого начинается подстрока, совпадающая со второй строкой, либо возвращает –1, если вторая строка не содержится в первой в качестве подстроки. Индекс `i` в этой функции пробегает по всей длине первой строки, и для каждого заданного `i` индекс `j` пробегает по всей длине второй строки, сравнивая соответствующие символы. Если исполнение функции дошло до конца, значит, подстрока так и не была обнаружена.

Листинг 53. Поиск подстроки в строке

```
int find_substring(const char* s1, const char* s2)
{
    for (int i = 0; s1[i] != '\0'; ++i)
    {
        int j = 0;
        while (s2[j] != '\0' && s1[i + j] == s2[j])
        {
            ++j;
        }
        if (s2[j] == '\0')
        {
            return i;
        }
    }
    return -1;
}
```

Поскольку для каждого символа из первой строки происходит последовательная проверка всех символов из второй строки, то общее количество операций составляет произведение количеств этих символов. Таким образом, общая вычислительная сложность такого алгоритма поиска подстроки длины m в строке длины n составляет $O(nm)$. То есть такой алгоритм

позволяет лишь искать короткие строки в длинных строках, но не длинные строки в длинных строках. Конечно, на практике вложенный цикл редко работает много итераций, но всегда можно построить строку длины 10^6 символов, так что для поиска подстроки длиной 10^5 символов в ней потребуется больше одной секунды на обычном домашнем компьютере.

С другой стороны, указанная реализация использует $O(1)$ дополнительной памяти, что считается хорошей особенностью. Конечно, общий объём использованной памяти составляет $O(n+m)$, поскольку всё ещё необходимо хранить входные строки. Как видно, неэффективность вычислительной сложности данного алгоритма компенсируется эффективностью использования памяти.

На самом деле, реализовывать функцию из листинга 53 на практике нет необходимости. В языке C существует функция `strstr`, объявленная в файле `string.h`, которая работает примерно как функция из листинга 53, но в отличие от неё возвращает не индекс символа, с которого начинается подстрока, а указатель на этот символ в первой строке. В случае отсутствия такой подстроки возвращается нулевой указатель.

В языке C++ у объектов класса `string` есть метод `find`, который тоже работает похожим образом. Он возвращает индекс, как и функция из листинга 53, но этот индекс имеет тип `size_t`, который не хранит знак, так что в случае отсутствия подстроки возвращается специальное значение `string::npos`. Все эти функции реализуют тот же самый алгоритм, который реализован в листинге 53, так что они работают за $O(nm)$.

В действительности, такое время работы для алгоритма подобного рода крайне неудовлетворительно. Существуют более эффективные алгоритмы решения задачи поиска подстроки в строке, вычислительная сложность которых доходит до $O(n+m)$. Правда они используют около $O(n+m)$ дополнительной памяти в дополнение памяти под хранение входных строк.

Использование дополнительной памяти – это основная причина, по которой эти алгоритмы не реализованы в стандартных библиотеках большинства языков программирования. Детальное рассмотрение этих алгоритмов выходит также и за рамки этой книги.

Упражнения для самостоятельной работы

1. Напишите исходный код программы, принимающей на вход непустую строку из строчных латинских букв длиной до 10^5 символов, и выводящей сообщение «`YES`», если эта строка является палиндромом, либо «`NO`» в противном случае. Строка является палиндромом, если совпадает с соответствующей перевёрнутой задом наперёд строкой. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

2. Напишите исходный код программы, принимающей на вход непустую строку из строчных латинских букв длиной до 10^5 символов, и выводящей ту же строку, перевёрнутую задом наперёд. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

3. Напишите исходный код программы, принимающей на вход две непустые строки из строчных латинских букв одинаковой длины до 10^5 символов, и выводящей расстояние Хэмминга между этими строками. Расстояние Хэмминга между двумя строками – это количество соответствующих позиций, в которых символы этих строк различаются. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

4. Напишите исходный код программы, принимающей на вход непустую строку из строчных латинских букв и цифр длиной до 10^5 символов, и выводящей сумму цифр, встречающихся в этой строке. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

5. Напишите исходный код программы, принимающей на вход непустую строку из строчных латинских букв длиной до 10^5 символов, и выводящей ту же строку, зашифрованную шифром Цезаря. При шифровании шифром Цезаря каждая латинская буква заменяется на циклически сдвинутую букву в латинском алфавите на три позиции. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

6. Напишите исходный код программы, принимающей на вход непустую строку из строчных латинских букв длиной до 10^5 символов, и выводящей количество различных символов, содержащихся в этой строке. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

7. Напишите исходный код программы, принимающей на вход две непустые строки из строчных латинских букв длиной до 10^5 символов, и выводящей сообщение «`YES`», если символы в первой строке можно поменять местами так, чтобы получилась вторая строка, либо «`NO`» в противном случае. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

8. Напишите исходный код программы, принимающей на вход название файла, и выводящей самое длинное слово в этом файле. Слова состоят из латинских букв и отделяются друг от друга символами, не являющимися буквами. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

9. Напишите исходный код программы, принимающей на вход название файла, и выводящей последние 10 строк этого файла. Строки отделяются друг от друга переводами строк. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

10. Напишите исходный код программы, принимающей на вход название файла, и вставляющей в начало каждой строки этого файла номер этой строки и знак табуляции. Строки отделяются друг от друга переводами строк.

Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

11. Напишите исходный код программы, принимающей на вход название файла и две строки из строчных латинских букв длиной до 10^5 символов каждая, и выводящей наименьшее количество символов, разделяющее эти строки в этом файле. Асимптотически оцените вычислительную сложность вашего алгоритма и объём используемой памяти.

12. Напишите Куайн – программу, выводящую на экран свой собственный исходный код. Эта программа не должна считывать что-либо ни из консоли, ни из файла.

ЗАКЛЮЧЕНИЕ

В книге были детально описаны способы хранения большинства базовых типов данных в памяти компьютера и подходы к разработке простых алгоритмов обработки этих данных. Из основных типов данных были рассмотрены целые числа, числа с плавающей запятой и символы. Также подробно была рассмотрена работа с массивами данных, поскольку массив является фундаментальной структурой данных, на базе которой строятся многие более сложные структуры.

Из множества рассмотренных алгоритмов много внимания было уделено алгоритмам сортировки с целью демонстрации того, как одну и ту же несложную задачу можно решить огромным количеством способов, отличающихся по эффективности. Также для каждого рассмотренного алгоритма приведён подробный анализ его эффективности в плане использования процессорного времени и памяти. Это сделано, чтобы показать больше примеров оценки вычислительной сложности алгоритмов, в результате чего читатель мог бы легко и быстро оценивать вычислительную сложность собственных алгоритмов ещё до их непосредственной программной реализации.

Можно заметить, что в совокупности изложенный в книге материал не является стандартным для курса информатики. Здесь опускается много абстрактных тем вроде истории развития этой науки, понятийного аппарата, основ теории информации, а вместо этого приводится много практических сведений о разработке алгоритмов и о написании программ. Причина этому – крайняя полезность этих умений на практике как при изучении других курсов, связанных с информационными технологиями, так и при работе непосредственно по специальности, причём не важно, какого рода приложения или скрипты пишет программист.

Основной целью этого курса является сделать читателя более дружелюбным к компьютеру, чтобы он мог решать всевозможные возникающие в повседневной жизни вычислительные задачи на электронных устройствах разного рода. Нужно понимать, что компьютер в отличие от человека чётко исполняет любые данные ему команды, так что является крайне надёжным инструментом для решения множества задач. Кроме того, написание программ доставляет многим людям массу удовольствия, поскольку представляет собой определённый вызов и при этом является творческим занятием, приводящим к созданию чего-то качественно нового.

СПИСОК ЛИТЕРАТУРЫ

1. Александреску, А. Современное проектирование на C++: Обобщенное программирование и прикладные шаблоны проектирования / А. Александреску. – СПб.: Вильямс, 2008. – 336 с.

2. Вирт, Н. Алгоритмы + структуры данных = программы / Н. Вирт. – М.: Мир, 1985. – 406 с.

3. Дасгупта, С. Алгоритмы / С. Дасгупта, Х. Пападимитриу, У. Вазирани. – М.: МЦНМО, 2014. – 320 с.

4. Каймин, В. А. Информатика: Учебник / В. А. Каймин. – М.: ИНФРА-М, 2001. – 272 с.

5. Кнут, Д. Искусство программирования, том 1. Основные алгоритмы / Д. Кнут. – М.: Вильямс, 2006. – 720 с.

6. Кнут, Д. Искусство программирования, том 2. Получисленные алгоритмы / Д. Кнут. – М.: Вильямс, 2007. – 832 с.

7. Кнут, Д. Искусство программирования, том 3. Сортировка и поиск / Д. Кнут. – М.: Вильямс, 2007. – 824 с.

8. Кормен, Т. Х. Алгоритмы: построение и анализ / Т. Х. Кормен, Ч. И. Лейзерсон, Р. Л. Ривест, К. Штайн. – М.: Вильямс, 2005. – 1296 с.

9. Королев, Л. Информатика. Введение в компьютерные науки / Л. Королев, А. И. Миков. – М.: Высшая школа, 2003. – 342 с.

10. Липпман, С. Б. Основы программирования на C++ / С. Б. Липпман. – М.: Вильямс, 2002. – 256 с.

11. Макарова, Н. В. Информатика: Учебник для вузов / Н. В. Макарова, В. Б. Волков. – СПб.: Питер, 2011. – 576 с.

12. Майерс, С. Эффективное использование C++. 35 новых способов улучшить стиль программирования / С. Майерс. – СПб.: Питер, 2006. – 224 с.

13. Майерс, С. Эффективное использование C++. 50 рекомендаций по улучшению ваших программ и проектов / С. Майерс. – СПб.: Питер, 2006. – 240 с.

14. Окулов, С. М. Основы программирования / С. М. Окулов. – М.: БИНОМ. Лаборатория знаний, 2004. – 424 с.

15. Окулов, С. М. Программирование в алгоритмах / С. М. Окулов. – М.: БИНОМ. Лаборатория знаний, 2004. – 341 с.

16. Павловская, Т. А. C/C++. Программирование на языке высокого уровня / Т. А. Павловская. – СПб.: Питер, 2003. – 461 с.

17. Савельев, А. Я. Основы информатики: Учеб. для вузов / А. Я. Савельев. – М.: Изд-во МГТУ им. Н. Э. Баумана, 2001. – 328 с.

18. Саттер, Г. Новые сложные задачи на C++ / Г. Саттер. – М.: Вильямс, 2015. – 272 с.

19. Саттер, Г. Решение сложных задач на C++ / Г. Саттер. – М.: Вильямс, 2015. – 400 с.

20. Саттер, Г. Стандарты программирования на C++ / Г. Саттер, А. Александреску. – М.: Вильямс, 2015. – 224 с.

21. Седжвик, Р. Алгоритмы на C++. Фундаментальные алгоритмы и структуры данных / Р. Седжвик. – М.: Вильямс, 2011. – 1056 с.

22. Страуструп, Б. Дизайн и эволюция C++ / Б. Страуструп. – М.: ДМК Пресс, 2014. – 446 с.

23. Страуструп, Б. Язык программирования C++ / Б. Страуструп. – СПб.: Невский Диалект, 2001. – 1099 с.

24. Халперн, П. Стандартная библиотека C++ на примерах / П. Халперн. – М.: Вильямс, 2001. – 336 с.

25. Шень, А. Программирование: теоремы и задачи / А. Шень. – М.: МЦНМО, 2004. – 296 с.

26. Шилдт, Г. C++: базовый курс / Г. Шилдт. – М.: Вильямс, 2012. – 624 с.

27. Шилдт, Г. Теория и практика C++ / Г. Шилдт. – СПб.: BHV, 1996. – 416 с.

28. Шилдт, Г. Самоучитель C++ / Г. Шилдт. – СПб.: БХВ-Петербург, 2003. – 688 с.

29. Эккель, Б. Философия C++. Введение в стандартный C++ / Б. Эккель. – СПб.: Питер, 2004. – 572 с.

30. Эккель, Б. Философия C++. Практическое программирование / Б. Эккель. – СПб.: Питер, 2004. – 608 с.

31. Элджер, Дж. C++: Библиотека программиста / Дж. Элджер. – СПб.: Питер, 1999. – 320 с.

СОДЕРЖАНИЕ