

1. Алгоритмы и их эффективность

1.1. Задача, алгоритм, программа

Процесс *решения задачи* (problem solving) на компьютере включает в себя нижеследующие этапы.

1. Постановка задачи.
2. Разработка алгоритма решения задачи.
3. Доказательство корректности алгоритма и анализ его эффективности.
4. Реализация алгоритма на языке программирования.
5. Выполнение программы для получения требуемого результата.

В *постановке задачи* (problem statement) приводится точная формулировка условий задачи с описанием ее *входных* (input) и *выходных* (output) данных.

В процессе *разработки алгоритма* (algorithm design) формулируется пошаговая процедура получения из входных данных выходных. Дадим определение понятию алгоритм.

Алгоритм (algorithm) – это конечная последовательность инструкций *исполнителю*, в результате выполнения которых обеспечивается получение из входных данных требуемого выходного результата (решение задачи).

Алгоритм записывается на формальном языке исполнителя, что исключает неоднозначность толкования предназначенных ему предписаний. Запись алгоритма на формальном языке исполнителя называется *программой* (program, рис. 1.1).

Говорят, что алгоритм *корректен* (correct), если он для любых корректных значений входных данных выдает корректные выходные данные. Здесь под корректностью данных понимается их соответствие условиям решаемой задачи.

После того как разработан алгоритм решения задачи осуществляется его *реализация* (implementation) на одном из языков программирования, например: C, C++, Java, C#, Go, Python, JavaScript и др. Язык программирования выступает в роли языка исполнителя, а исполнителем является интерпретатор или процессор, в набор инструкций которого компилятор

транслирует программу.

В ходе разработки программы алгоритм может претерпевать изменения, связанные с учетом архитектуры целевой системы. Например, в алгоритм могут вноситься изменения, обеспечивающие эффективное использование кеш-памяти процессора, и др.

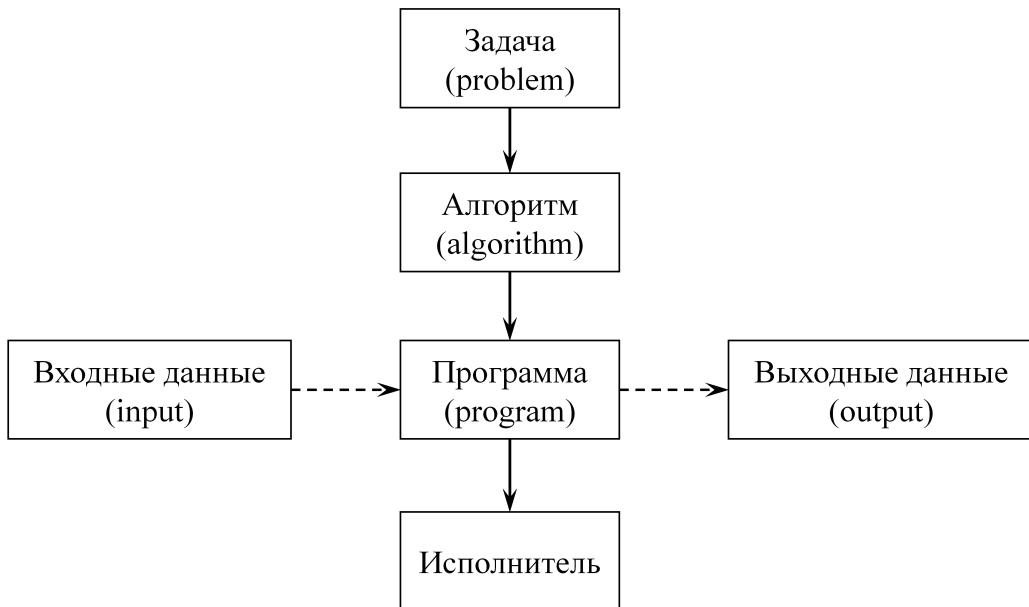


Рис. 1.1. Процесс решения задачи: задача, алгоритм, программа.

Рассмотрим основные свойства, которыми должен обладать алгоритм.

1. *Дискретность* – алгоритм представляется как последовательность инструкций исполнителя. Каждая инструкция выполняется только после того, как закончилось выполнение предыдущей команды.
2. *Конечность* (результативность, финитность) – алгоритм должен заканчиваться после выполнения конечного числа инструкций.
3. *Детерминированность* – каждый шаг алгоритма должен быть однозначно определен – записан на формальном языке исполнителя. Детерминированность обеспечивает совпадение результатов, получаемых при многократном выполнении алгоритма, на одном и том же наборе входных данных.
4. *Массовость* – алгоритм решения задачи должен быть применим для некоторого класса задач, различающихся лишь значениями входных данных.

Пример. Поиск максимального элемента массива. Рассмотрим задачу поиска в массиве номера элемента с максимальным значением. Формальная постановка задачи выглядит следующим образом:

- **вход:** последовательность из n чисел (a_1, a_2, \dots, a_n) ;
- **выход:** номер i элемента a_i , имеющего наибольшее значение:

$$a_i \geq a_j, \quad \forall j \in \{1, 2, \dots, n\}. \quad (1.1)$$

Если на вход подается последовательность $(14, 20, 34, 6, 85, 232, 177)$, то решением задачи является номер 6 (элемент со значением 232). Такой конкретный набор входных данных называется *экземпляром задачи* (problem instance).

Ниже приведен псевдокод линейного алгоритма Max решения рассматриваемой задачи о поиске номера максимального элемента. На вход алгоритма поступает массив $a[1..n]$ из n элементов.

Алгоритм 1.1. Поиск максимального элемента массива

```

1 function Max( $a[1..n]$ )
2    $maxi = 1$ 
3   for  $i = 2$  to  $n$  do
4     if  $a[i] > a[maxi]$  then
5        $maxi = i$ 
6     end if
7   end for
8   return  $maxi$ 
9 end function
```

Алгоритм Max является *дискретным*, так как записан на диалекте императивного языка программирования Алгол-60. *Конечность* алгоритма обеспечивается, тем, что он всегда заканчивает свою работу после проштотра $n - 1$ элементов массива a . В процессе своей работы алгоритм не использует датчик псевдослучайных чисел для принятия решений о ходе дальнейшего выполнения, это обеспечивает его *детерминированность*. *Массовость* алгоритма обусловлена тем, что он решает задачу поиска номера максимального элемента для произвольного массива $a[1..n]$.

1.2. Показатели эффективности алгоритмов

Алгоритмы, разработанные для решения одной и той же задачи, могут значительно различаться по эффективности. Поэтому для характеристики их качества вводят показатели эффективности. Рассмотрим наиболее распространенные из них.

1. Количество операций – *временная эффективность* (time efficiency), показывает насколько быстро работает алгоритм.
2. Объем потребляемой памяти – *пространственная эффективность* (space efficiency), отражает максимальное количество памяти, требуемой для выполнения алгоритма.

Существуют и другие показатели, которые имеет смысл рассматривать, если они в значительной степени влияют на процесс решения задачи. Например, для алгоритмов, работающих с данными на внешних носителях информации (жесткие диски, сетевые хранилища), целесообразно учитывать количество обращений к внешней памяти, а в алгоритмах, использующих сетевые каналы связи, важно принимать во внимание количество переданных сообщений (сетевых пакетов).

Введение показателей эффективности позволяет проводить анализ алгоритмов с целью сравнения их между собой и оценивания потребности того или иного алгоритма в вычислительных ресурсах: процессорном времени, памяти, пропускной способности сети.

1.3. Подсчет числа операций алгоритма

Для большинства алгоритмов количество выполняемых ими операций напрямую зависит от размера входных данных. Чем больше входных данных, тем дольше работает алгоритм. Так, время выполнения алгоритма Мах поиска максимального элемента (алгоритм 1.1) зависит от длины n массива.

Количество операций алгоритма можно выразить как функцию от одного или нескольких параметров, связанных с размером входных данных. Рассмотрим несколько примеров:

- алгоритм сортировки выбором – количество операций алгоритма (время его работы) зависит от числа n элементов в массиве;
- алгоритм умножения двух матриц – время выполнения зависит от количества строк m и столбцов n в матрицах;
- алгоритм Дейкстры поиска кратчайшего пути в графе – время выполнения зависит от числа n вершин и m ребер в графе.

RAM-машина. Для подсчета числа операций, выполняемых алгоритмом, необходимо формально описать систему команд некоторого исполнителя. В качестве такого исполнителя будем использовать модель однопроцессорной вычислительной машины с произвольным доступом к памяти (Random Access Machine — RAM) [1, 2]. Условимся, что машина обладает неограниченной памятью и функционирует по следующим правилам:

- для выполнения арифметических и логических операций ($+, -, *, /, \%$) требуется один временной шаг (такт процессора);
- каждое обращение к ячейке в оперативной памяти для чтения или записи занимает один временной шаг;
- выполнение условного перехода (*if-then-else*) требует вычисления логического выражения и выполнения одной из ветвей *if-then-else*;
- выполнение цикла (*for*, *while*, *do*) подразумевает выполнение всех его

итераций, в свою очередь, выполнение каждой итерации требует вычисления условия завершения цикла и выполнение его тела.

Пример. Суммирование элементов массива. Вычислим количество операций алгоритма SUMARRAY, реализующего вычисление суммы n элементов массива.

Алгоритм 1.2. Суммирование элементов массива

```

1 function SUMARRAY( $a[1..n]$ )
2    $sum = 0$ 
3   for  $i = 1$  to  $n$  do
4      $sum = sum + a[i]$ 
5   end for
6   return  $sum$ 
7 end function
```

Время работы алгоритма SUMARRAY зависит только от размера n массива. В строке 2 выполняется одна операция записи в память. Далее, перед выполнением каждой из n итераций цикла происходят проверка условия его окончания $i = n$ и переход на строку 4 или 6. На каждой итерации в строке 4 выполняется четыре операции: чтение из памяти значений sum и $a[i]$, их сложение и запись результата в память. В конце алгоритма выполняется возврат результирующего значения – одна операция. Таким образом, количество операций $T(n)$, выполняемых алгоритмом SUMARRAY, есть

$$T(n) = 4n + 2.$$

Далее мы убедимся, что такой точный анализ числа операций алгоритма во многих случаях не требуется. Достаточно ограничиться подсчетом лишь тех операций, суммарное количество которых зависит от размера входных данных. Так, в алгоритме SUMARRAY строки 2 и 6 не имеют значимого влияния на итоговое время выполнения, которое фактически определяется только операциями в строке 4.

При анализе вычислительной сложности алгоритмов мы будем игнорировать операции, связанные с проверкой условия окончания цикла *for* и автоматическим увеличением его счетчика.

Пример. Линейный поиск. Существует большое количество алгоритмов, время выполнения которых зависит не только от размера входных данных, но и от их значений. В качестве примера рассмотрим алгоритм LINEARSEARCH линейного поиска заданного значения x в массиве из n элементов.

Количество операций алгоритма LINEARSEARCH может существенно различаться для одного и того же размера n входных данных. Рассмотрим три возможных случая.

Алгоритм 1.3. Линейный поиск

```

1 function LINEARSEARCH( $a[1..n]$ ,  $x$ )
2   for  $i = 1$  to  $n$  do
3     if  $a[i] = x$  then
4       return  $i$ 
5     end if
6   end for
7   return  $-1$ 
8 end function

```

Лучший случай (best case) – экземпляр задачи (набор входных данных), на котором алгоритм выполняет *наименьшее* число операций. В нашем примере – входной массив, первый элемент которого содержит искомое значение x . В этой ситуации требуется выполнить

$$T_{Best}(n) = 3$$

операции: проверка условия окончания цикла, условие в цикле (строка 3) и возврат найденного значения (строка 4). Таким образом, время работы алгоритма в лучшем случае – теоретическая *нижняя граница* времени его работы.

Худший случай (worst case) – экземпляр задачи, на котором алгоритм выполняет наибольшее число операций. Для рассматриваемого алгоритма – массив, в котором отсутствует искомый элемент или он расположен в последней ячейке. В этой ситуации требуется выполнить

$$T_{Worst}(n) = 2n + 1$$

операций: n раз проверить условие окончания цикла и условие в нем, затем вернуть значение -1 . Время работы алгоритма в худшем случае – теоретическая *верхняя граница* времени его работы.

Средний случай (average case) – «средний» экземпляр задачи, набор «усредненных» входных данных. В среднем случае оценивается математическое ожидание количества операций, выполняемых алгоритмом. Стоит заметить, что не всегда очевидно, какие входные данные считать «усредненными» для задачи. Часто делается предположение, что все наборы данных поступают на вход алгоритма с одинаковой вероятностью.

Вернемся к нашему примеру и проведем анализ его эффективности для среднего случая. Обозначим вероятность успешного поиска элемента в массиве через $p \in [0, 1]$. Тогда вероятность отсутствия значения x в массиве равна $1 - p$. Будем считать, что искомый элемент с одинаковой вероятностью p/n может находиться в любой из n ячеек массива.

Если искомый элемент x находится в ячейке 1, то для его поиска тре-

буется выполнить 3 операции (проверить условие окончания цикла, условие в цикле и вернуть значение 1); если элемент находится в ячейке 2, то требуется 5 операций, и т.д. В общем случае, если искомый элемент x расположен в ячейке i , то это требует выполнения $2i + 1$ операций.

Запишем математическое ожидание (среднее значение) числа операций, выполняемых алгоритмом. По определению, математическое ожидание есть сумма произведений значения дискретной случайной величины на вероятность принятия случайной величиной этого значения. Наша случайная величина – число операций, выполняемых алгоритмом. Как мы условились выше, она может принимать с одинаковой вероятностью p/n следующие значения: 3, 5, …, $(2i + 1)$, …, $2n + 1$. Поэтому

$$T_{Average}(n) = 3\frac{p}{n} + 5\frac{p}{n} + \cdots + (2i + 1)\frac{p}{n} + \cdots + (2n + 1)\frac{p}{n}.$$

В нашей оценке мы должны учесть тот факт, что искомое значение x с вероятностью $1 - p$ может отсутствовать в массиве. Тогда выражение примет следующий вид:

$$T_{Average}(n) = \frac{p}{n} [3 + 5 + \cdots + (2i + 1) + \cdots + (2n + 1)] + (1 - p)(2n + 1).$$

Нетрудно заметить, что в квадратных скобках записана сумма членов арифметической прогрессии (см. приложение). Вычислим ее:

$$\begin{aligned} T_{Average}(n) &= \frac{p}{n} [n^2 + 2n] + (1 - p)(2n + 1) = \\ &= p(n + 2) + (1 - p)(2n + 1). \end{aligned}$$

Из полученной формулы можно сделать следующий вывод: если искомый элемент присутствует в массиве ($p = 1$), то в среднем требуется выполнить $n + 2$ операции для его нахождения.

Анализ эффективности алгоритмов для среднего случая – более трудная задача, чем анализ их поведения в худшем и лучшем случаях. Однако для многих алгоритмов именно оценки эффективности для среднего случая дают реальную картину относительно их практической применимости.

Далее при анализе алгоритмов мы будем уделять основное внимание времени работы алгоритмов в худшем случае – максимальному времени работы на всех наборах входных данных и, по возможности, строить оценки эффективности для среднего случая.

В качестве синонимов понятию *количество операций* алгоритма мы будем использовать термины *время выполнения* алгоритма (execution time) и *вычислительная сложность* алгоритма (computational complexity).

1.4. Скорость роста функций

Пусть, мы имеем два алгоритма решения одной и той же задачи. В соответствии с описанной выше схемой построены функции $T_1(n)$ и $T_2(n)$ зависимости числа операций алгоритмов от размера их входных данных для лучшего, среднего либо худшего случая. Определимся, что

$$T_1(n) = 90n^2 + 201n + 2000,$$

$$T_2(n) = 2n^3 + 3.$$

Возникает вопрос: какой из алгоритмов предпочтительнее использовать на практике?

Время выполнения обоих алгоритмов возрастет с увеличением размера n входных данных. На рис. 1.2 приведены графики функций $T_1(n)$ и $T_2(n)$, а в табл. 1.1 – их значения. На данных небольшого размера второй алгоритм выполняет меньше операций чем первый, следовательно, в этом случае предпочтение надо отдать второму алгоритму. На больших значениях n второй алгоритм начинает заметно уступать первому, здесь стоит выбрать первый алгоритм.

В общем случае мы можем найти такое значение n_0 , при котором происходит пересечение функций $T_1(n)$ и $T_2(n)$, и, зная конкретный размер n входных данных и n_0 , отдавать предпочтение тому или иному алгоритму.

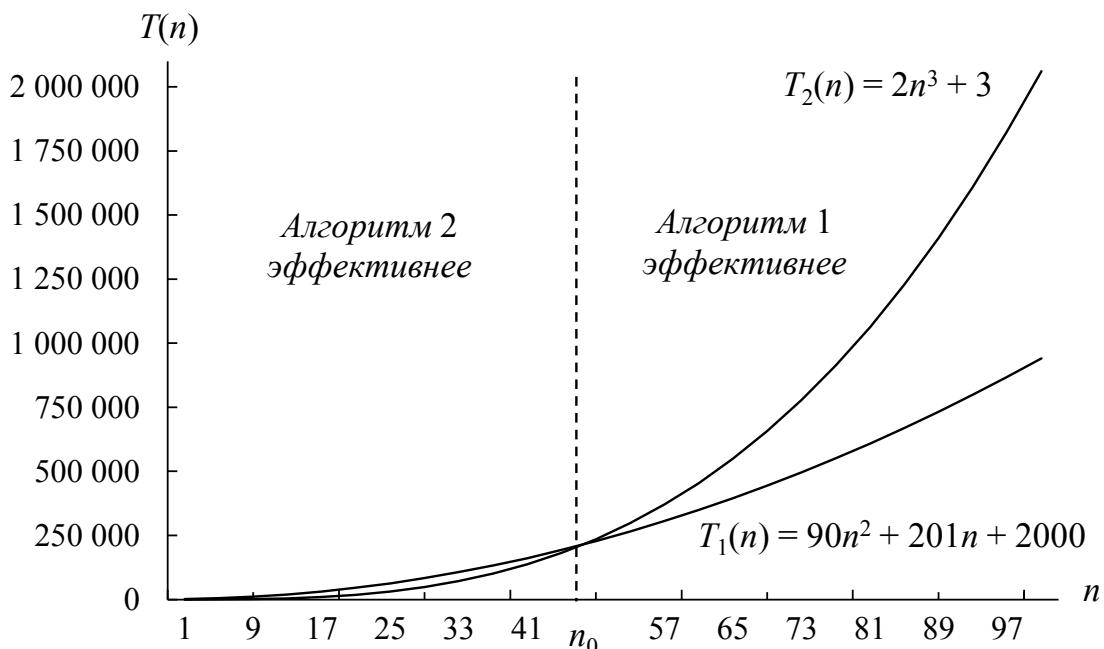


Рис. 1.2. Зависимость числа операций, выполняемых алгоритмами, от размера n входных данных.

В большинстве случаев при малых размерах n входных данных разница во времени выполнения алгоритмов, решающих одну и ту же задачу, как правило, незначительна. Если априори известно, что на вход будут

Таблица 1.1. Количество операций, выполняемых алгоритмами

n	$T_1(n)$	$T_2(n)$	Сравнение алгоритмов
10	13 010	2 003	Первый алгоритм в 6.5 раза медленнее второго: $T_1(n)/T_2(n) = 6.5$
20	42 020	16 003	Первый в 2.6 раза медленнее второго
30	89 030	54 003	Первый в 1.6 раза медленнее второго
40	154 040	128 003	Первый в 1.2 раза медленнее второго
50	237 050	250 003	Первый в 1.1 раза быстрее второго
60	338 060	432 003	Первый в 1.3 раза быстрее второго
70	457 070	686 003	Первый в 1.5 раза быстрее второго
80	594 080	1 024 003	Первый в 1.7 раза быстрее второго
90	749 090	1 458 003	Первый в 1.9 раза быстрее второго
100	922 100	2 000 003	Первый в 2.2 раза быстрее второго
1 000	9 0203 000	2 000 000 003	Первый в 22 раза быстрее второго
10 000	9 002 012 000	2 000 000 000 003	Первый в 222 раз быстрее второго

поступать данные небольших размеров, то вопрос о выборе эффективного алгоритма не является первостепенным: можно использовать самый «простой» алгоритм – понятный, легко реализуемый на языке программирования или доступный в стандартной библиотеке языка. *Даже неэффективный алгоритм при работе с входными данными небольшого размера завершается за допустимое время.*

Вопросы, связанные с пространственной и времененной эффективностью алгоритмов, приобретают смысл при больших размерах входных данных. В том числе, когда заранее не известно, какого размера экземпляры задач нам предстоит решать. Следовательно, в первую очередь нас будет интересовать вопрос о том, как быстро растет число операций $T(n)$, выполняемых алгоритмом, при увеличении размера n входных данных.

Для сравнения функций по тому, насколько быстро они изменяют свои значения с увеличением значений их аргументов, вводят понятие *скорости роста функций*.

Скорость роста (rate of growth) или *порядок роста* (order of growth) функции $T(n)$ определяются ее старшим, доминирующим членом. При больших значениях n членами меньшего порядка можно пренебречь. В нашем примере функция $T_1(n) = 90n^2 + 201n + 2000$ растет как n^2 , а функ-

ция $T_2(n) = 2n^3 + 3$ как n^3 . Порядок роста функции n^3 больше порядка роста функции n^2 , так как n^3 принимает большие значения, чем n^2 .

Один алгоритм рассматривается как более эффективный по сравнению с другим (в худшем, среднем либо лучшем случае), если число его операций имеет более низкий порядок роста.

В табл. 1.2 показана скорость роста некоторых функций, которые часто встречаются в оценках времени работы алгоритмов. Медленнее всего растут функции $\lg n$ и $\log_2 n$. Алгоритмы с подобной логарифмической зависимостью числа операций от размера входных данных работают практически мгновенно. Например, бинарный поиск элемента в упорядоченном массиве или поиск узла в АВЛ-дереве или красно-черном дереве. Быстрее всех (из приведенных в табл. 1.2) растут функции 2^n и $n!$. Алгоритмы с такой сложностью на практике мало применимы, так как даже на небольших размерах входных данных требуют выполнения колоссального числа операций. Примерами могут служить некоторые оптимизационные алгоритмы, реализующие полный перебор множества допустимых решений задачи. Если решение представляется в виде перестановки из n чисел, то количество перебираемых алгоритмом вариантов равно числу перестановок порядка n , а именно $n!$.

Таблица 1.2. Скорость роста функций

$\lg n$	$\log_2 n$	n	$n \log_2 n$	n^2	2^n	$n!$
0	0	1	0	1	2	1
0.3	1	2	2	4	4	2
0.5	1.6	3	5	9	8	6
0.6	2.0	4	8	16	16	24
0.7	2.3	5	12	25	32	120
0.78	2.6	6	16	36	64	720
0.85	2.8	7	20	49	128	5 040
0.90	3	8	24	64	256	40 320
0.95	3.2	9	29	81	512	362 880
1	3.3	10	33	100	1024	3 628 800
3	10	1 000	9 966	1 000 000		
4	13.3	10 000	132 877	100 000 000		
5	16.6	100 000	1 660 964	10 000 000 000		
6	19.9	1 000 000	19 931 569	1 000 000 000 000		

Для сравнения и классификации скоростей роста функций, выражющих время работы алгоритмов, нам необходим специальный математиче-

ский аппарат. Он должен давать возможность делать выводы о том, что два алгоритма при больших n работают с одинаковым временем или один алгоритм эффективнее другого. Ответы на эти вопросы дает *асимптотический анализ* (asymptotic analysis), который позволяет оценивать скорость роста функций $T(n)$ при стремлении размера входных данных к бесконечности (при $n \rightarrow \infty$).

1.5. Асимптотические обозначения

Как было показано ранее, время выполнения алгоритма в худшем, среднем и лучшем случаях можно представить, как функцию $T(n)$ от размера его входных данных. Однако анализировать эффективность алгоритмов по таким функциям достаточно трудно по ряду причин. Как правило, функция $T(n)$ времени выполнения алгоритма имеет большое количество локальных экстремумов – неровный график с выпуклостями и впадинами (рис. 1.3, а). Например, возможна ситуация, когда время выполнения алгоритма на входных массивах с нечетной длиной будет больше времени выполнения на массивах с четной длиной. В этом случае график $T(n)$ будет иметь пилообразный вид, как на рис. 1.3, б. Поэтому намного проще работать с верхней и нижней границами (оценками) времени выполнения алгоритма. Так для примера на рис. 1.3, б вместо пилообразной функции $T(n)$ можно использовать ее верхнюю границу $5n + 15$ или другую.

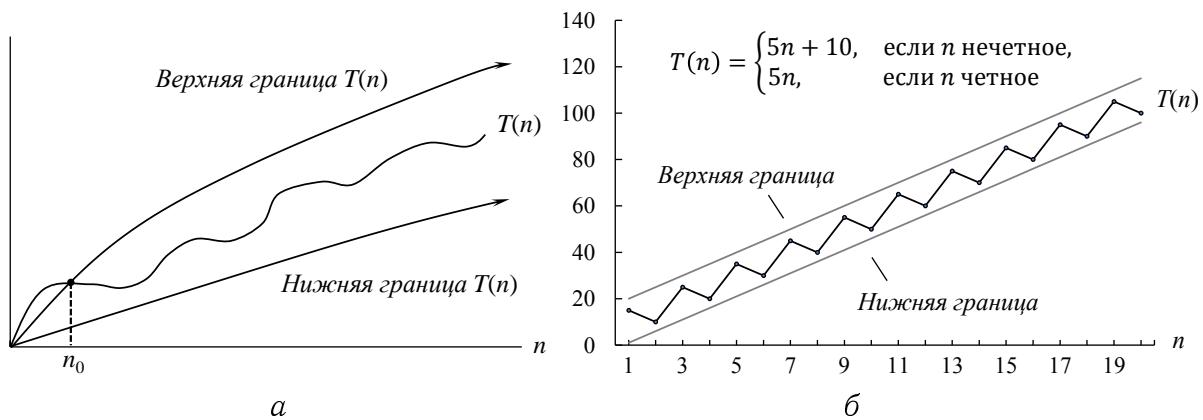


Рис. 1.3. Пример возможных верхней и нижней границ времени $T(n)$ выполнения алгоритмов.

Для указания границ функций $T(n)$ в теории *вычислительной сложности алгоритмов* (computational complexity theory) используют асимптотические обозначения: O (о большое), Ω (омега большое), Θ (тета большое), а также o (о малое) и ω (омега малое). Далее будем считать, что областью определения функций $f(n)$ и $g(n)$, которые выражают число операций алгоритма, является множество неотрицательных целых чисел

($n \in \{0, 1, 2, \dots\}$). Функции $f(n)$ и $g(n)$ являются *асимптотически неотрицательными* – при больших значениях n они принимают значения, большие или равные нулю. Применительно к анализу сложности алгоритмов последнее требование выполняется всегда, так функции выражают число операций алгоритма или объем потребляемой им памяти, которые не могут быть отрицательными.

1.5.1. O -обозначение

O -обозначение используют, если необходимо указать *асимптотическую верхнюю границу* (asymptotic upper bound) для функции $f(n)$, числа операций алгоритма. Говорят, что функция $f(n)$ принадлежит множеству функций $O(g(n))$, что записывается как $f(n) \in O(g(n))$, если существуют положительная константа $c > 0$ и $n_0 \in \{0, 1, 2, \dots\}$, такие что для любых $n \geq n_0$

$$0 \leq f(n) \leq cg(n).$$

Формально множество $O(g(n))$ определяется следующим образом:

$$O(g(n)) = \left\{ f(n): \begin{array}{l} \exists c > 0, n_0 \in \{0, 1, 2, \dots\}, \text{ такие, что} \\ 0 \leq f(n) \leq cg(n), \quad \forall n \geq n_0. \end{array} \right\} \quad (1.2)$$

Иначе говоря, $O(g(n))$ – это множество всех функций, значения которых при больших n *не превышают* значение $cg(n)$. Обычно факт принадлежности функции $f(n)$ множеству $O(g(n))$ записывают как $f(n) = O(g(n))$. Читается как « f от n есть о большое от g от n ».

Для доказательства, $f(n) = O(g(n))$, требуется найти константы $c > 0$ и $n_0 \in \{0, 1, 2, \dots\}$, которые обеспечивают выполнение неравенств (1.2). Если доказано, что $f(n) = O(g(n))$, то говорят: «функция f асимптотически ограничена сверху функцией g с точностью до постоянного множителя». Таким образом, произведение $cg(n)$ является *асимптотической оценкой сверху* времени $f(n)$ работы алгоритма.

Пример. Докажем, что $2n - 10 = O(n)$. Для этого требуется найти соответствующие константы $c > 0$ и $n_0 \in \{0, 1, 2, \dots\}$. Доказательство: возьмем $c = 2$, $n_0 = 5$. Эти значения обеспечивают выполнение неравенства $0 \leq 2n - 10 \leq 2n$ для любых $n \geq 5$. На рис. 1.4 видно, что прямая $2n$ проходит выше прямой $2n - 10$. Можно было взять и другие значения c и n_0 , главное – это то, что мы показали их существование.

Аналогично можно доказать корректность следующих утверждений.

1. $3n^2 + 100n + 8 = O(n^2)$. Для доказательства возьмем $c = 4$, $n_0 = 101$, при любых $n \geq 101$ справедливо неравенство $0 \leq 3n^2 + 100n + 8 \leq 4n^2$.
2. $3n^2 + 100n + 8 = O(n^3)$. Возьмем $c = 1$, $n_0 = 12$, для любых $n \geq 12$ справедливо $0 \leq 3n^2 + 100n + 8 \leq n^3$.

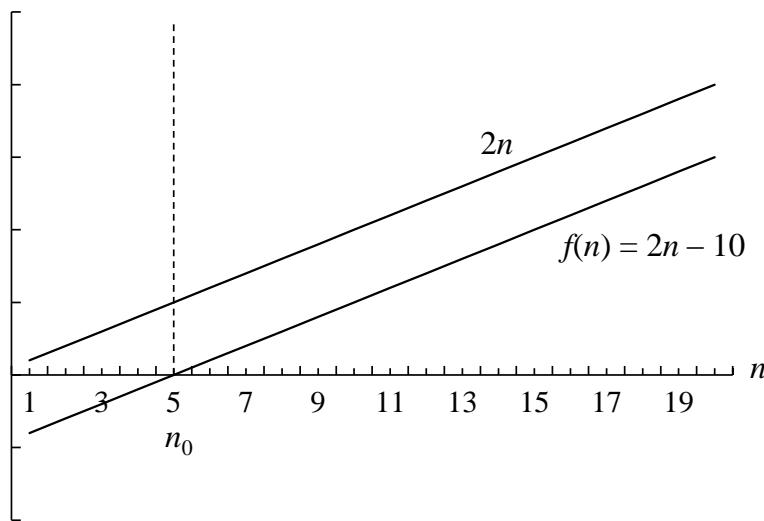


Рис. 1.4. Иллюстрация принадлежности функции $2n - 10$ множеству $O(n)$.

3. $0.000001n^3 \neq O(n^2)$, так как не существует констант $c > 0$ и n_0 , которые обеспечивают выполнение неравенств (1.2). Для любых $c > 0$ и $n \geq c/0.000001$ имеет место неравенство $0.000001n^3 > cn^2$.

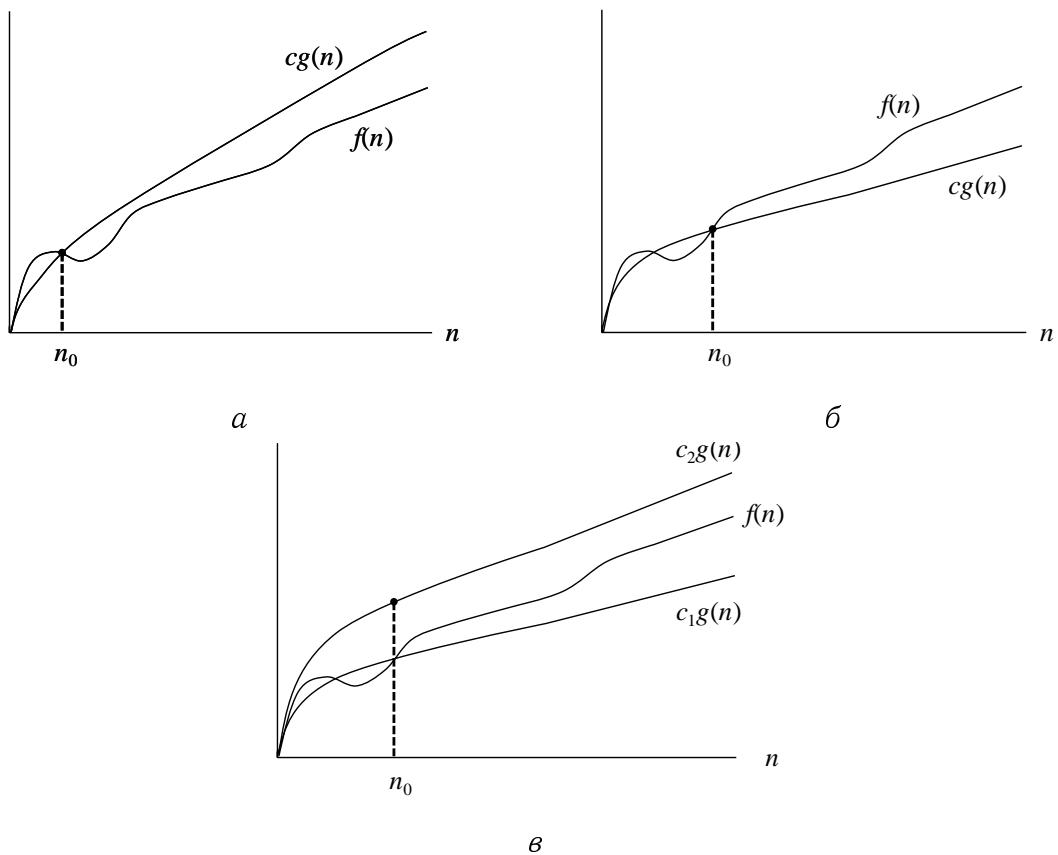


Рис. 1.5. Иллюстрация асимптотических обозначений:
 $a - f(n) = O(g(n)); \beta - f(n) = \Omega(g(n)); \theta - f(n) = \Theta(g(n)).$

На рис. 1.5 приведены иллюстрации основных асимптотических обозначений.

1.5.2. Ω -обозначение

Ω -обозначение используется для записи *асимптотической нижней границы* (asymptotic lower bound) для функции $f(n)$. Говорят, функция $f(n)$ принадлежит множеству $\Omega(g(n))$ (записывается как $f(n) \in \Omega(g(n))$), если существуют константа $c > 0$ и $n_0 \in \{0, 1, 2, \dots\}$, такие что для любых $n \geq n_0$

$$0 \leq cg(n) \leq f(n).$$

Читается как « f от n есть омега большое от g от n ». Формально множество $\Omega(g(n))$ определяется следующим образом

$$\Omega(g(n)) = \left\{ f(n) : \begin{array}{l} \exists c > 0, n_0 \in \{0, 1, 2, \dots\}, \text{ такие, что} \\ 0 \leq cg(n) \leq f(n), \quad \forall n \geq n_0. \end{array} \right\} \quad (1.3)$$

Другими словами, $\Omega(g(n))$ – это множество всех функций, значения которых при больших n не меньше значения $cg(n)$. Для доказательства принадлежности функции $f(n)$ множеству $\Omega(g(n))$ требуется найти константы $c > 0$ и $n_0 \in \{0, 1, 2, \dots\}$, которые обеспечивают выполнение неравенств (1.3). Если доказано, что $f(n) = \Omega(g(n))$, то говорят: «функция f асимптотически ограничена снизу функцией g с точностью до постоянного множителя». Таким образом, произведение $cg(n)$ является *асимптотической оценкой снизу* времени $f(n)$ работы алгоритма.

Пример. Докажем справедливость отношения $n^3 + 5 = \Omega(n^2)$. Следуя определению, нам необходимо показать существование констант $c > 0$ и $n_0 \in \{0, 1, 2, \dots\}$, при которых $n^2 \leq 4n^3 + 5$ для любых $n \geq n_0$. Для доказательства достаточно взять $c = 1$ и $n_0 = 0$.

1.5.3. Θ -обозначение

Θ -обозначение позволяет записать *асимптотически точную оценку* (asymptotic tight bound) для функции $f(n)$. Функция $f(n)$ принадлежит множеству функций $\Theta(g(n))$, записывается как $f(n) \in \Theta(g(n))$, если существуют положительные константы $c_1 > 0, c_2 > 0$ и $n_0 \in \{0, 1, 2, \dots\}$, такие что для любых $n \geq n_0$

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n).$$

Читается как « f от n есть тета большое от g от n ». Множество $\Theta(g(n))$ определяется следующим образом

$$\Theta(g(n)) = \left\{ f(n) : \begin{array}{l} \exists c_1 > 0, c_2 > 0, n_0 \in \{0, 1, 2, \dots\}, \text{ такие, что} \\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \quad \forall n \geq n_0. \end{array} \right\} \quad (1.4)$$

Для доказательства того, что $f(n) = \Theta(g(n))$, требуется найти константы $c_1 > 0$, $c_2 > 0$ и $n_0 \in \{0, 1, 2, \dots\}$, которые обеспечивают выполнение неравенств (1.4). Если доказано, что $f(n) = \Theta(g(n))$, то говорят: «функция f асимптотически ограничена снизу и сверху функцией g с точностью до постоянного множителя».

Обозначение Θ более сильное, чем O и Ω . Используя обозначения теории множеств, можно записать $\Theta(g(n)) \subseteq O(g(n))$.

Утверждение. Для любых двух функций $f(n)$ и $g(n)$ отношение $f(n) = \Theta(g(n))$ выполняется тогда и только тогда, когда $f(n) = O(g(n))$ и $f(n) = \Omega(g(n))$.

Из этого утверждения следует: если известно, что $f(n) = \Theta(g(n))$, то это гарантирует $f(n) = O(g(n))$ и $f(n) = \Omega(g(n))$. С другой стороны, если мы покажем, что для некоторой функции $f(n)$ одновременно выполняются $f(n) = O(g(n))$ и $f(n) = \Omega(g(n))$, то тем самым мы докажем справедливость $f(n) = \Theta(g(n))$. Рассмотренное утверждение дает нам двухшаговую процедуру для построения асимптотически точных оценок для функций.

Пример. Докажем корректность утверждения $2n^2 + 3n = \Theta(n^2)$. По определению требуется найти константы $c_1 > 0$, $c_2 > 0$ и $n_0 \in \{0, 1, 2, \dots\}$, которые обеспечивают выполнение неравенств

$$0 \leq c_1 n^2 \leq 2n^2 + 3n \leq c_2 n^2.$$

Сперва покажем, что наша функция ограничена сверху. Пусть $c_2 = 3$. Неравенство $2n^2 + 3n \leq 3n^2$ выполняется для любых $n \geq 3$ – это точка пересечения функций $2n^2 + 3n$ и $3n^2$. Таким образом $n_0 = 3$ и $2n^2 + 3n = O(n^2)$. Осталось показать выполнение неравенств $0 \leq c_1 n^2 \leq 2n^2 + 3n$. Они справедливы при $c_1 = 2$ и $n_0 = 0$, следовательно, $2n^2 + 3n = \Omega(n^2)$. Далее, нам нужно выбрать n_0 , при котором выполняются как условия ограничения функции снизу, так и ограничения сверху. Поэтому выберем n_0 как максимальное значение из найденных n_0 , это обеспечит выполнение обоих условий: $n_0 = \max(0, 3) = 3$. Мы нашли константы $c_1 = 2$, $c_2 = 3$ и $n_0 = 3$, тем самым доказали справедливость исходного утверждения $2n^2 + 3n = \Theta(n^2)$.

1.5.4. o -обозначение

Обозначение o используют для указания того, что верхняя граница функции не является асимптотически точной. Например, запись

$$100n = O(n^2)$$

содержит в себе корректную асимптотическую верхнюю границу функции $100n$, но не обеспечивает нас верной информацией об асимптотически точ-

ной верхней границе функции. Дадим определение o -обозначения:

$$o(g(n)) = \left\{ f(n) : \begin{array}{l} \forall c > 0, \exists n_0 \in \{0, 1, 2, \dots\}, \text{ такие, что} \\ 0 \leq f(n) < cg(n), \quad \forall n \geq n_0. \end{array} \right\} \quad (1.5)$$

Таким образом $o(g(n))$ – это множество всех функций, значения которых при больших n *меньше* значения $c g(n)$ для *любых* c . Факт принадлежности функции $f(n)$ множеству $o(g(n))$ записывают как $f(n) = o(g(n))$. Читается как « f от n есть о малое от g от n ». Важным отличием от O -обозначения является ограничение сверху функции $f(n)$ значением $c g(n)$ для *любых* (*всех*) положительных констант c . Например, $100n = o(n^2)$, в то же время $4n^2 + 2n \neq o(n^2)$.

1.5.5. ω -обозначение

Обозначение ω подобно обозначению Ω . Дадим ему формальное определение:

$$\omega(g(n)) = \left\{ f(n) : \begin{array}{l} \forall c > 0, \exists n_0 \in \{0, 1, 2, \dots\}, \text{ такие, что} \\ 0 \leq cg(n) < f(n), \quad \forall n \geq n_0. \end{array} \right\} \quad (1.6)$$

Иначе говоря, $\omega(g(n))$ – множество всех функций, значения которых при больших n *больше* значения $c g(n)$ для *любых* c . Факт принадлежности функции $f(n)$ множеству $\omega(g(n))$ записывают как $f(n) = \omega(g(n))$. Читается как « f от n есть омега малое от g от n ». Важным отличием от Ω -обозначения является ограничение снизу функции $f(n)$ значением $c g(n)$ для любых положительных констант c . Например, справедливо отношение $n^2 = \omega(n)$, но $24n^2 \neq \omega(n^2)$.

1.5.6. Случай функции нескольких переменных

Асимптотические обозначения можно обобщить и на случай функций нескольких переменных. Например, для функции двух переменных будем иметь следующее определение O -обозначения:

$$O(g(n, m)) = \left\{ f(n, m) : \begin{array}{l} \exists c > 0, n_0 \in \{0, 1, 2, \dots\}, m_0 \in \{0, 1, 2, \dots\}, \text{ такие, что} \\ 0 \leq f(n, m) \leq cg(n, m), \quad \forall n \geq n_0, \forall m \geq m_0. \end{array} \right\}$$

Аналогично можно ввести определения остальных асимптотических обозначений для функций нескольких переменных.

1.5.7. Сравнение порядка роста функций через предел

Пусть, как и ранее, мы имеем две функции $f(n)$ и $g(n)$. Для ответа на вопрос, какая из функций имеет более высокий порядок роста, достаточно вычислить предел их отношения при $n \rightarrow \infty$ [2, 3]

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \begin{cases} 0, & \text{если } f(n) = o(g(n)), \\ c, & \text{если } f(n) = \Theta(g(n)), \\ \infty, & \text{если } f(n) = \omega(g(n)). \end{cases} \quad (1.7)$$

Если предел равен 0, то $f(n) = o(g(n))$ и $f(n) = O(g(n))$. Второй случай: $f(n) = \Theta(g(n))$, следовательно, $f(n) = O(g(n))$ и $f(n) = \Omega(g(n))$. Из третьего случая следует $f(n) = \omega(g(n))$ и $f(n) = \Omega(g(n))$.

Пример. Сравним порядки (скорости) роста функций $f(n) = n(n-1)/2$ и $g(n) = n^2$.

$$\lim_{n \rightarrow \infty} \frac{n(n-1)/2}{n^2} = \frac{1}{2} \lim_{n \rightarrow \infty} \left(1 - \frac{1}{n}\right) = \frac{1}{2}.$$

Предел равен положительной константе, тогда по (1.7) обе функции имеют одинаковый порядок роста $n(n-1)/2 = \Theta(n^2)$.

Пример. Сравним порядки роста функции $f(n) = 2n + 1$ и функции $g(n) = \log_a n$ при $a > 1$.

$$\lim_{n \rightarrow \infty} \frac{2n + 1}{\log_a n} = \lim_{n \rightarrow \infty} \frac{2}{\frac{1}{n \ln a}} = \lim_{n \rightarrow \infty} 2n \ln a = \infty.$$

Для вычисления последнего предела использовано *правило Лопиталя* – перешли к рассмотрению предела отношения производных функций $f(n)$ и $g(n)$. Предел равен бесконечности. Это означает, что функция $f(n)$ имеет больший порядок роста, чем $g(n)$. Иначе говоря, $2n+1 = \omega(\log n)$. Обратите внимание, что здесь намеренно опущено основание a логарифма, так как при использовании совместно с асимптотическими обозначениями оно не имеет смысла. По свойствам логарифмов изменение основания приводит к умножению логарифма на константу, а по свойствам асимптотических обозначений константы могут быть отброшены:

$$\log_a n = \frac{1}{\log_c a} \log_c n.$$

1.5.8. Свойства асимптотических отношений

Факт принадлежности функции $f(n)$ некоторому множеству мы записывали через знак равенства, например, $f(n) = O(g(n))$. Это обозначение удобно на практике, но нужно помнить, что смысл его в том, что функция,

стоящая слева от знака равенства, принадлежит множеству, указанному справа.

$$f(n) \in O(g(n)).$$

Говоря более точно, это не равенство в обычном понимании, а несимметричное *бинарное отношение* между функциями $f(n)$ и $g(n)$. Например, запись $f(n) = O(g(n))$ является корректной, в то же время выражение $O(g(n)) = f(n)$ лишено смысла.

Если асимптотическое обозначение O , Ω или Θ присутствует в формуле, то вместо него можно мысленно подставить любую функцию из этого множества. Например, в выражении $3n^2 + 5n + 34 = 3n^2 + \Theta(n)$, вместо $\Theta(n)$ можно подставить любую функцию с линейным порядком роста.

Асимптотические бинарные отношения O , Ω и Θ обладают многими свойствами бинарных отношений между действительными числами (для определенности обозначим числа через f и g):

- $f(n) = O(g(n))$ соответствует $f \leq g$;
- $f(n) = \Omega(g(n))$ соответствует $f \geq g$;
- $f(n) = \Theta(g(n))$ соответствует $f = g$;
- $f(n) = o(g(n))$ соответствует $f < g$;
- $f(n) = \omega(g(n))$ соответствует $f > g$.

Транзитивность

- Если $f(n) = O(g(n))$ и $g(n) = O(h(n))$, то $f(n) = O(h(n))$;
- Если $f(n) = \Omega(g(n))$ и $g(n) = \Omega(h(n))$, то $f(n) = \Omega(h(n))$;
- Если $f(n) = \Theta(g(n))$ и $g(n) = \Theta(h(n))$, то $f(n) = \Theta(h(n))$;
- Если $f(n) = o(g(n))$ и $g(n) = o(h(n))$, то $f(n) = o(h(n))$;
- Если $f(n) = \omega(g(n))$ и $g(n) = \omega(h(n))$, то $f(n) = \omega(h(n))$.

Рефлексивность

- $f(n) = O(f(n))$;
- $f(n) = \Omega(f(n))$;
- $f(n) = \Theta(f(n))$.

Симметричность

- $f(n) = \Theta(g(n))$ тогда и только тогда, когда $g(n) = \Theta(f(n))$.

Перестановочная симметрия

- $f(n) = O(g(n))$ тогда и только тогда, когда $g(n) = \Omega(f(n))$;
- $f(n) = o(g(n))$ тогда и только тогда, когда $g(n) = \omega(f(n))$.

Рассмотрим свойства O -обозначения, которые удобно использовать на практике:

- **произведение** – если $f_1 = O(g_1)$ и $f_2 = O(g_2)$, то $f_1 \cdot f_2 \in O(g_1 \cdot g_2)$;

- **сложение** – если $f_1 = O(g_1)$ и $f_2 = O(g_2)$, то $f_1 + f_2 \in O(|g_1| + |g_2|)$;
- **умножение на константу** – если $k > 0$ и $f = O(g)$, то $kf \in O(g)$.

При использовании асимптотических обозначений константы в функции $T(n)$ игнорируются.

Из свойств следует: для любого полинома $p(n)$ степени $k \in \{0, 1, 2, \dots\}$ при $a_k > 0$ справедливо

$$p(n) = a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0 = \Theta(n^k).$$

1.6. Этапы асимптотического анализа

Асимптотический анализ сложности алгоритма заключается в построении асимптотических оценок его вычислительной сложности (computational complexity) или объема требуемой ему памяти (space complexity) в худшем, среднем или лучшем случае. Рассмотрим основные шаги асимптотического анализа.

1. Определяются параметры, от которых зависит время выполнения алгоритма или объем требуемой ему памяти: устанавливается, сколько аргументов будет у функции $T(n)$. Например, $T(n)$, $T(n, m)$, $T(n, m, q)$.

2. Для худшего, среднего или лучшего случая вычисляется количество операций алгоритма или объем требуемой ему памяти, что записывается как функция от параметров, установленных на предыдущем шаге. Например, $T(n) = 3n^2 + 32n + 5$. В первую очередь вычисляют число операций алгоритма для худшего случая. Если на практике возникновение этого случая маловероятно, то подсчитывается число операций алгоритма для среднего случая. На данном этапе можно ограничиться подсчетом числа только *базовых операций* (basic operations) – наиболее важных операций, от которых зависит время выполнения алгоритма. Например, можно не учитывать константное число операций чтения значений переменных из памяти и операций присваивания.

3. К построенной функции $T(n)$ применяются асимптотические обозначения для классификации порядка ее роста. При помощи Θ -обозначения строится асимптотически точная оценка для функции $T(n)$. В случае затруднения построения такой оценки, ограничиваются O -обозначением – асимптотической верхней границей функции $T(n)$.

На практике в процессе асимптотического анализа алгоритмов встречается небольшой набор классов сложности, которые приведены в табл. 1.3 (в порядке возрастания скорости роста).

Таблица 1.3. Классы сложности алгоритмов

Обозначение класса сложности	Название класса	Пример
$O(1)$	Константная сложность	Алгоритм определения четности целого числа. Время выполнения таких алгоритмов (или объем потребляемой памяти) не зависит от размера входных данных
$O(\log n)$	Логарифмическая сложность	Алгоритм бинарного поиска в упорядоченном массиве. Такие алгоритмы на каждом шаге обрабатывают лишь часть входного набора данных
$O(n)$	Линейная сложность	Алгоритм поиска минимального элемента в неупорядоченном массиве. Алгоритмы с такой сложностью выполняют просмотр всего набора входных данных
$O(n \log n)$	Линейно-логарифмическая сложность	Алгоритм сортировки слиянием. Такая сложность характерна для алгоритмов, разработанных с использованием метода декомпозиции («разделяй и властвуй»)
$O(n^2)$	Квадратичная сложность	Алгоритм сортировка выбором
$O(n^3)$	Кубическая сложность	Алгоритм умножения квадратных матриц по определению
$O(2^n)$	Экспоненциальная сложность	Алгоритмы, обрабатывающие все подмножества некоторого множества из n элементов
$O(n!)$	Факториальная сложность	Оптимизационные алгоритмы, реализующие полный перебор множества допустимых решений задачи

Пример. Факториал числа. Проведем асимптотический анализ вычислительной сложности алгоритма вычисления факториала целого числа n . По определению факториал числа n есть

$$n! = 1 \cdot 2 \cdot \dots \cdot n.$$

Алгоритм 1.4. Вычисление факториала числа

```

1 function FACTORIAL( $n$ )
2    $fact = 1$ 
3   for  $i = 2$  to  $n$  do
4      $fact = fact \cdot i$ 
5   end for
6   return  $fact$ 
7 end function
```

Шаг 1. Время выполнения алгоритма FACTORIAL зависит только от значения n , следовательно, функция $T(n)$ будет иметь один аргумент.

Шаг 2. Вычислим количество $T(n)$ операций алгоритма. Худший, средний и лучший случаи для этого алгоритма совпадают. Одна операция уходит на инициализацию переменной $fact$, далее на каждой из $n - 1$ итерации цикла выполняется две операции (умножение и присваивание), после чего происходит возврат значения из функции – еще одна операция. Следовательно,

$$T(n) = 1 + 2(n - 1) + 1 = 2n.$$

Шаг 3. Получим асимптотически точную оценку вычислительной сложности алгоритма. Докажем, что $2n = \Theta(n)$. Возьмем $c_1 = 1$, $c_2 = 2$ и $n_0 = 0$, эти константы обеспечивает выполнение неравенства

$$0 \leq n \leq 2n \leq 2n, \quad \text{при } n \geq 0.$$

Результатом асимптотического анализа вычислительной сложности алгоритма FACTORIAL является асимптотически точная оценка $\Theta(n)$, которая говорит о том, что алгоритм имеет линейную вычислительную сложность.

Теперь выполним асимптотический анализ сложности по памяти алгоритма. Вычислим количество $M(n)$ ячеек памяти RAM-машины, которое требуется алгоритму. В строке 1 инициализируется переменная $fact$ – это требует одной ячейки памяти, для переменной i также требуется одна ячейка памяти. Окончательно получаем

$$M(n) = 2 = \Theta(1).$$

Алгоритм FACTORIAL имеет константную сложность по памяти – он не создает в памяти структур данных, размер которых зависит от n .

Пример. Сортировка выбором. Проведем асимптотический анализ вычислительной сложности алгоритма сортировки выбором (selection sort). Подробное описание алгоритма приведено в разделе 3.4.

Алгоритм 1.5. Сортировка выбором

```

1 function SELECTIONSORT( $A[1..n]$ )
2   for  $i = 1$  to  $n - 1$  do
3      $min = i$ 
4     for  $j = i + 1$  to  $n$  do
5       if  $a[j] < a[min]$  then
6          $min = j$ 
7       end if
8     end for
9     if  $min \neq i$  then
10       $temp = a[i]$ 
11       $a[i] = a[min]$ 
12       $a[min] = temp$ 
13    end if
14  end for
15 end function
```

Шаг 1. Время выполнения алгоритма SELECTIONSORT зависит от размера n массива и значений элементов в нем.

Шаг 2. Вычислим количество $T(n)$ операций алгоритма, выполняемых им в худшем случае. Это подразумевает выполнение максимального числа операций, что достигается при выполнении на всех итерациях циклов условий в строках 5 и 9. Худший случай для алгоритма SELECTIONSORT – это массив, элементы которого упорядочены по убыванию – в порядке, обратном направлению сортировки. В таком случае внешний цикл *for* выполняется $n - 1$ раз. На каждой итерации этого цикла одна операция приходится на запись значения i в переменную min (строка 3), что в сумме дает $n - 1$ операцию. На строки 9–13 приходится четыре операции (проверка условия и три присваивания), что дает еще $4(n - 1)$ операций. Тогда

$$T(n) = (n - 1) + 4(n - 1) + T_{InnerLoop}(n),$$

где функция $T_{InnerLoop}(n)$ отражает количество операций, приходящихся на внутренний цикл, за все время выполнения алгоритма.

Вычислим, сколько операций приходится на внутренний цикл в строках 4–8. Рассмотрим процесс выполнение внешнего цикла:

- при $i = 1$ внутренний цикл выполняется $n - 1$ раз (переменная j принимает значения от 2 до n);
- при $i = 2$ внутренний цикл выполняется $n - 2$ раза (переменная j

принимает значения от 3 до n);

- ...
- при $i = n - 2$ внутренний цикл выполняется 2 раза (переменная j принимает значения от $n - 1$ до n);
- при $i = n - 1$ внутренний цикл выполняется 1 раз (переменная j принимает значения от n до n).

Учитывая, что на каждой итерации внутреннего цикла выполняется две операции (проверка условия и присваивание (строки 4, 5)), получаем

$$T_{InnerLoop}(n) = 2(n - 1 + n - 2 + \dots + 1),$$

$$T(n) = (n - 1) + 4(n - 1) + 2[n - 1 + n - 2 + \dots + 1].$$

В квадратных скобках записана сумма членов арифметической прогрессии с разностью 1. Вычислим ее

$$T(n) = (n - 1) + 4(n - 1) + 2 \left[\frac{n^2 - n}{2} \right] = n^2 + 4n - 5.$$

Шаг 3. Получим асимптотически точную оценку вычислительной сложности алгоритма `SELECTIONSORT`. Докажем справедливость

$$T(n) = n^2 + 4n - 5 = \Theta(n^2).$$

Следуя определению асимптотических обозначений, возьмем константы $c_1 = 1$, $c_2 = 2$ и $n_0 = 2$. Они обеспечивают выполнение неравенств

$$0 \leq n^2 \leq n^2 + 4n - 5 \leq 2n^2, \quad \text{для любых } n \geq 2.$$

Таким образом, мы показали, что алгоритм сортировки выбором в худшем случае имеет *квадратичную вычислительную сложность*, и эта оценка является асимптотически точной.

Действуя аналогично, можно оценить вычислительную сложность алгоритма сортировки выбором для лучшего случая, когда входной массив уже упорядочен в направлении сортировки. Условия внутри циклов выполнятся не будут, и число операций алгоритма будет равно

$$T_{Best}(n) = (n - 1) + (n - 1) + [n - 1 + n - 2 + \dots + 1] = \Theta(n^2).$$

Сложность по памяти алгоритма сортировки выбором является константной, так как в процессе своей работы алгоритму требуется лишь четыре дополнительные ячейки под переменные i , j , min и $temp$ (входной массив $a[1..n]$ не учитывается).

1.7. Упражнения

1. Определите параметры, от которых зависит время работы перечисленных ниже алгоритмов:

- алгоритм последовательного суммирования элементов массива;
- алгоритм сложения по определению двух квадратных матриц;
- алгоритм Евклида нахождения наибольшего общего делителя двух целых чисел.

2. Разместите следующие функции по неубыванию скорости их роста

$$n, \quad \log_2 n, \quad \sqrt{n}, \quad \log_{10} n, \quad n^{1/3}, \quad \log_2(\log_2 n), \quad \frac{1}{2^n}, \quad (\log_2 n)^2, \quad 2^n.$$

3. Имеются вычислитель (процессор), который выполняет $3 \cdot 10^9$ операций в секунду, и три алгоритма с числом операций

$$T_1(n) = 400n \log_2 n + 1024, \quad T_2(n) = 2n^2 + 8n + 4, \quad T_3(n) = n! + 4.$$

Определите сколько времени (дней, часов, минут, секунд) будет выполняться каждый алгоритм при $n = 100$.

4. Используя определения асимптотических обозначений O и Θ , докажите справедливость следующих отношений:

- $n^3 - 2n^2 - 100n + 1 = O(n^3)$;
- $n^2 = O(2^n)$;
- $2^{n+1} = \Theta(2^n)$.

5. Сравните порядки роста функций через предел (1.7)

- $f(n) = 2^{2n}$, $g(n) = 2^n$;
- $f(n) = \ln n$, $g(n) = n$;
- $f(n) = 2^n$, $g(n) = n^{100}$ ^{*};
- $f(n) = \log_2 n$, $g(n) = \sqrt{n}$;
- $f(n) = n!$, $g(n) = 2^n$ [†].

6. Ниже приведен псевдокод функции LOOPS_A. Вычислите количество $T(n)$ операций в ней.

7. Ниже приведен псевдокод функции LOOPS_B. Вычислите количество $T(n)$ операций в ней.

^{*}Подсказка: примените правило Лопиталя 100 раз.

[†]Подсказка: используйте формулу Стирлинга.

```
1 function LOOPSA(n)
2     x = 1
3     for i = 1 to n do
4         for j = i + 1 to n do
5             for k = 1 to n do
6                 x = x + 1
7             end for
8         end for
9     end for
10    return x
11 end function
```

```
1 function LOOPSB(n)
2     x = 1
3     for i = 1 to n do
4         for j = i + 1 to n do
5             for k = 1 to j do
6                 x = x + 1
7             end for
8         end for
9     end for
10    return x
11 end function
```
