ПРОЕКТИРОВАНИЕ И АНАЛИЗ АЛГОРИТМОВ

1.1. ТРУДОЕМКОСТЬ АЛГОРИТМА

Определение 1.1. Под формальным описанием задачи мы понимаем такое ее описание, когда задача описывается в виде функции, на вход которой поступают формальные параметры, задающие ее входные данные.

Определение 1.2. Под размерностью задачи, которую в дальнейшем будем обозначать через l, понимают то количество информации, которое достаточно для формального описания задачи.

Теперь возникает вопрос: что такое информация и в чем она измеряется? Прежде чем дать ответ этот вопрос, напомним некоторые сведения из теории вероятности.

Пусть есть некоторое случайное событие A, которое может принимать значения из конечного множества $\{a_1, a_2, \ldots, a_n\}$ – алфавита случайной величины A (можно считать, что a_i – возможный исход некоторого эксперимента). Например, при подбрасывании монетки случайное событие A заключается в определении того, что выпало: «орел» или «решка». Для этого примера алфавит события A состоит из двух величин: {«орел», «решка»}. При подбрасывании кубика случайное событие A заключается в определении того, какое число точек выпало на верхней грани кубика. Для данного случайного события A алфавит состоит из шести величин: $\{1, 2, \ldots, 6\}$, которые соответствуют количеству выпавших на кубике точек.

Вероятность того или иного события представляет собой число в интервале $[0,1]$ (можно сказать, что вероятность отображает события на действительные числа). Вероятность 0 означает, что событие не произойдет никогда, а вероятность 1 – что оно произойдет наверняка. Пусть вероятность того, что случайная величина A примет значение, равное a_i, есть p_i:

$$p(A = a_1) = p_1 \in [0,1],$$
$$p(A = a_2) = p_2 \in [0,1],$$
$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$
$$p(A = a_n) = p_n \in [0,1].$$

Так как событие A должно обязательно произойти (один из вариантов наверняка должен быть реализован), то для суммы вероятностей справедливо следующее равенство: $\sum_{i=1}^{n} p_i = 1.$

Если предположить, что возможности реализации каждого из входов алфавита одинаковы, т. е. $p_1 = p_2 = \ldots = p_n$, то $p(A = a_i) = 1/n, \forall i = 1, \ldots, n.$

Определение 1.3. Количество информации, которой достаточно для знания того, что событие A произошло (достаточно для разрушения неопределенности об объекте), определяется по формуле

$$I(A) = -\log_b p(A),$$

где $p(A)$ – вероятность, с которой событие A произойдет.

Если в качестве основания логарифма b взять число 2, то единицей измерения информации будет бит.

Поскольку количество битов, в которых измеряется информация, является целым числом, то

$$I(A) = \lceil -\log_2 p(A) \rceil.$$

Напомним, что $\lceil x \rceil$ – наименьшее целое, большее числа x, $\lfloor x \rfloor$ – наибольшее целое, меньшее числа x:

$$x - 1 < \lfloor x \rfloor \le x \le \lceil x \rceil < x + 1.$$

Таким образом, объединяя определения 1.1, 1.2 и 1.3, можно сказать, что размерность задачи – это минимальное количество бит, которого достаточно для описания входных данных задачи.

Впервые ввести меру информации попытался Р. Хартли в 1928 г. В своих рассуждениях он исходил из интуитивной идеи о том, что сообщение, состоящее из n символов, должно нести в n раз больше информации, чем сообщение, состоящее из одного символа. Единственной функцией, которая удовлетворяет этому свойству, является логарифмическая.

Замечание 1.1. В дальнейшем мы будем предполагать (если не оговорено другое), что если на вход алгоритма поступает некоторое целое число x, то предполагается, что число x может быть выбрано из множества $\{1, \ldots, x\}$, причем любой выбор равновероятный.

Замечание 1.2. В дальнейшем мы будем предполагать (если не оговорено другое), что если на вход алгоритма поступает некоторая последовательность чисел $x_1, \ldots, x_m$, то предполагается, что все x_i могут быть выбраны из множества X, причем любой выбор равновероятный. Множество X при этом задается явно.

Пример 1.1. Пусть имеется 8 шаров с номерами от 1 до 8. Какого минимального количества информации достаточно для установления номера выбранного случайным образом шара?

Решение. Поскольку любой шар может быть выбран с равной вероятностью, то вероятность события A, заключающегося в определении номера выбранного шара, есть $p(A) = 1/8$ и $I(A) = \left\lceil -\log_2 1/8 \right\rceil = 3$. Заметим, что, действительно, трех бит достаточно, чтобы задать в памяти компьютера восемь различных чисел (т. е. распознать число от 1 до 8).

Пример 1.2. Рассмотрим различные типы вводимых чисел.

1) Предположим, что на вход задачи поступает некоторое натуральное число A, которое может быть выбрано из множества $\{1,\ldots,n\}$, причем любой выбор равновероятный. Какого минимального количества бит достаточно, чтобы определить, какое число поступило?

Решение. Поскольку натуральное число A может принимать любое значение из предложенного множества с равной вероятностью, то вероятность ввода некоторого числа A равна $1/n$. Тогда из определения информации следует, что

$$l = \left\lceil -\log_2 \frac{1}{n} \right\rceil = \lceil \log_2 n \rceil \quad (2^{l-1} < n \le 2^l).$$

Действительно, такого минимального количества бит достаточно, чтобы определить, какое конкретно число из рассматриваемого множества поступило. Например, если $n = 7$, то $l = \left\lceil -\log_2 1/7 \right\rceil = \lceil \log_2 7 \rceil = 3.$ Действительно, достаточно трех бит, чтобы можно было идентифицировать любое из семи чисел (табл. 1.1).

Таблица 1.1

1-е число	000
2-е число	001
3-е число	010
4-е число	011
5-е число	100
6-е число	101
7-е число	110

2) Предположим, что на вход задачи поступает некоторое целое число A, которое может быть выбрано из множества $\{0,1,\ldots,n\}$, причем любой

выбор является равновероятным. Какого минимального количества бит достаточно, чтобы определить, какое число поступило?

Решение. Поскольку целое число A может принимать любое значение из предложенного множества с равной вероятностью, то вероятность поступления некоторого числа A равна $1/(n+1)$. Тогда из определения информации следует, что

$$l = \left\lceil -\log_2 \frac{1}{n+1} \right\rceil = \lceil \log_2 (n+1) \rceil \quad (2^{l-1} - 1 < n \le 2^l - 1).$$

3) Предположим, что на вход задачи поступает некоторое целое число A, которое может быть выбрано из множества $\{-n,\ldots,-1,0,1,\ldots,n\}$, причем любой выбор является равновероятным. Какого минимального количества бит достаточно, чтобы определить, какое число поступило?

Решение. Поскольку целое число A может принимать любое значение из предложенного множества с равной вероятностью, то вероятность поступления числа A равна $1/(2n+1)$. Тогда из определения информации следует, что

$$l = \left\lceil -\log_2 \frac{1}{2 \cdot n + 1} \right\rceil = \lceil \log_2 (2 \cdot n + 1) \rceil \quad \left(2^{l-2} - 1 < n \le 2^{l-1} - 1\right).$$

4) Предположим, что на вход задачи поступает некоторое рациональное число A $\left(a/b\right)$. Какого минимального количества бит достаточно, чтобы определить, какое число мы выбрали?

Решение. Найдем множества возможных входных данных $S = \{(x, y) : x \in \{1,\ldots,a\}, y \in \{1,\ldots,b\}\}$, где входным числом будем считать x/y. Поскольку рациональное число A может принимать любое значение из предложенного множества S с равной вероятностью, то вероятность поступления числа A равна $1/a \cdot b$. Тогда из определения информации следует, что

$$l = \left\lceil -\log_2 \frac{1}{a \cdot b} \right\rceil = \lceil \log_2 (a \cdot b) \rceil \quad \left(2^{l-1} < a \cdot b \le 2^l\right).$$

5) Предположим, что на вход задачи поступают два целых числа x и y. Какого минимального количества бит достаточно, чтобы определить,

какие числа поступили?

Решение. Так как число x может принимать любое значение из множества $\{1,\ldots x\}$ с равной вероятностью, а число y может принимать любое значение из множества $\{1,\ldots y\}$ с равной вероятностью, то из определения информации следует, что

$$l = \lceil \log_2 x \rceil + \lceil \log_2 y \rceil.$$

6) Предположим, что на вход задачи поступает m целых чисел: $x_1,\ldots,x_m$ из множества $X = \{1,\ldots,x\}$. Какого минимального количества бит достаточно, чтобы определить, какие числа поступили?

Решение. Из определения информации следует, что

$$l = m \cdot \lceil \log_2 x \rceil.$$

7) Предположим, что на вход задачи поступает m рациональных чисел: $x_1,\ldots,x_m$ из множеста $X = \left\{ {x}/{y} : x \in \{-a,\ldots,-1,0,1,\ldots,a\},\ y \in \{1,\ldots,b\} \right\}$. Какого минимального количества бит достаточно, чтобы определить, какие числа поступили?

Решение. Из определения информации следует, что минимальное количество информации для кодирования каждого из входных чисел равно $\lceil \log_2 ((2a+1)\cdot b) \rceil$, т. е. $l = m\lceil \log_2 ((2a+1)\cdot b) \rceil$.

Объединим полученные в примере данные (табл. 1. 2).

Таблица 1.2

Данные	Формат данных	Размерность
x	целое число из множества $\{1,\ldots,x\}$	$\lceil \log_2 x \rceil$
	целое число из множества $\{0,\ldots,x\}$	$\lceil \log_2 (x+1) \rceil$
	целое число из множества $\{-x,\ldots,x\}$	$\lceil \log_2 (2\cdot x+1) \rceil$
	рациональное число $x = {a}/{b}$	$\lceil \log_2 (a\cdot b) \rceil$
$x_1,\ldots,x_m$	целые числа x_i из множества $\{1,\ldots,x\}$	$l = m\cdot\lceil \log_2 x \rceil$
	целые числа x_i из множества $\{0,\ldots,x\}$	$l = m\cdot\lceil \log_2 (x+1) \rceil$

	целые числа x_i из множества $\{-x,\ldots,x\}$	$l = m \cdot \lceil \log_2(2 \cdot x + 1) \rceil$

Если предположить, что размерность машинного слова достаточна для представления любого числа, то размерность задачи будет ограничена количеством исходных данных в ее формальном описании.

Определение 1.4. Трудоемкость алгоритма – это функция от размерности задачи, которая оценивает сверху время, требуемое для решения задачи.

Как же считать время, которое алгоритм затратит на решение задачи? Для этого необходимо определить модель вычислительного устройства, которое используется для реализации алгоритма, а также условиться, что понимать под элементарным шагом вычисления.

Мы будем использовать равнодоступную адресную машину (PAM), которая представляет собой вычислительную машину с одним сумматором, в котором команды программы не могут изменять сами себя. PAM состоит из входной ленты, с которой она может считывать данные, выходной ленты, на которую она может записывать, и памяти, которая состоит из последовательности регистров. Сама программа – это последовательность команд, она не записывается в память.

Учитывая сделанное ранее предположение о том, что для представления любого числа нам достаточно одного машинного слова, получаем, что емкостная сложность алгоритма (размерность памяти) для задания n элементов не превосходит n.

Если в качестве модели вычислений взять неветвящуюся программу и предположить, что алгоритм – это последовательность арифметических операций и все арифметические операции (аддитивные: +, –, *inc*, *dec*; мультипликативные: *, /) эквивалентны, т. е. затрачивают на свое выполнение одну единицу времени (хотя на самом деле известно, что мультипликативные операции работают дольше), то трудоемкость алгоритма определяется как функция, с количествм операций, требующимся для решения задачи, выраженное через её размерность.

В некоторых задачах удобно в качестве основной меры сложности брать число выполняемых команд разветвления. В случае сортировки, используя попарное сравнение элементов, разумно рассматривать такую модель, в которой все шаги дают разветвления, возникающие в результате сравнения двух величин. Такую программу представляют в виде двоичного дерева. Каждый узел такого дерева – один из шагов решения. Нужный выход находится в одном из листьев. Временная сложность дерева решений равна высоте дерева как функции размерности задачи, так

как обычно мы хотим измерить наибольшее число сравнений, которые приходится делать, чтобы найти нужный путь от корня к листу. Заметим, что общее число узлов в дереве может значительно превосходить его высоту. Например, дерево решений для сортировки n чисел должно содержать $n!$ листьев, при этом его высота может быть $n \cdot \log_2 n$ (рис. 1.1).

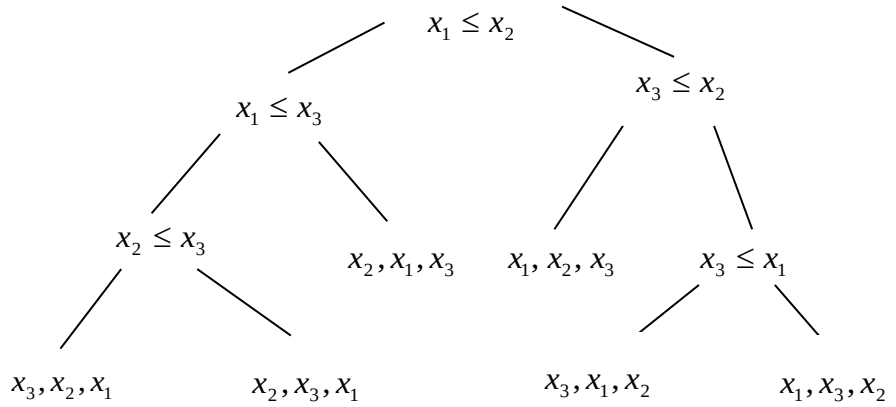

Рис. 1.1

Итак, для определения трудоемкости надо подсчитать число арифметических операций алгоритма. Однако классы входных данных существенно влияют на эту величину. Один и тот же алгоритм на одних данных работает очень быстро, а на других – медленнее. Практически мы будем искать такие входные данные, которые обеспечивают самое долгое выполнение алгоритма. Кроме того, мы будем оценивать среднюю эффективность алгоритма на всех возможных наборах данных. Различают:

- число арифметических операций в наихудшем случае;
- число арифметических операций в среднем.

Число арифметических операций в наихудшем случае: существует такой набор данных, на котором алгоритм работает медленнее всего. Этот случай важен, поскольку дает представление о максимальном времени работы алгоритма. Так, в алгоритме поиска некоторого элемента x среди n элементов массива наихудший случай, когда искомый элемент стоит на последнем месте. Такому алгоритму в наихудшем случае понадобится n сравнений. Анализ наихудшего случая дает верхние оценки для времени работы алгоритма.

Число арифметических операций в среднем на всех возможных наборах входных данных определяется следующим образом:

1. Разбиваем все входные данные на группы таким образом, чтобы время работы алгоритма для любых данных этой группы было одинаковым. Предположим, что у нас m групп.

2. Обозначим через p_i вероятность того, что входные данные принадлежат группе с номером i.

3. Обозначим через t_i время работы алгоритма на данных из группы i.

Тогда количество арифметических операций в среднем определяется по следующей формуле:

$$\sum_{i=1}^{m} p_i \cdot t_i .$$

Если данные могут попасть в любую группу с равной вероятностью, то $p_i = 1/m, \quad \forall i = 1,\ldots,m.$ Поэтому количество операций в среднем равно

$$\frac{1}{m} \cdot \sum_{i=1}^{m} t_i .$$

Проиллюстрируем это на примере поиска элемента в массиве из n чисел (предположим, что элемент обязательно присутствует и все элементы различны). Разбиение на группы будет следующим:

1-я группа: искомый элемент первый – $t_1 = 1$;

2-я группа: искомый элемент второй – $t_2 = 2$;

…….

n-я группа: искомый элемент последний – $t_n = n.$

Поскольку входные данные могут оказаться в любой из групп с равной вероятностью, то $p_i = 1/n, \quad \forall i = 1,\ldots,n$. Следовательно, число арифметических операций в среднем для поиска элемента в массиве из n чисел:

$$\sum_{i=1}^{n} p_i \cdot t_i = \frac{1}{n} \cdot \sum_{i=1}^{n} i = \frac{1}{n} \cdot (1 + 2 + \cdots + n) = \frac{n+1}{2}.$$

Точное значение количества операций не имеет для нас существенного значения. Более важна скорость роста этого числа при возрастании объема входных данных. Известно, что быстрорастущие функции с ростом аргумента доминируют над функциями с более медленным ростом. Поэтому если трудоемкость алгоритма – сумма нескольких функций, то отбрасывают все функции, кроме растущих быстрее всего. Отбрасывая эти функции, получаем функцию, которую называют порядком функции.

Так, порядок функции $f(x) = x^3 + 30 \cdot x + 7$ есть x^3 (говорят, что функция $f(x)$ растет как x^3).

Алгоритмы можно сгруппировать по скорости роста их трудоемкости в три основных класса (три асимптотики).

1. $O(f(n))$ – это класс функций, которые растут не быстрее, чем функция $f(n)$ («о большое» от f от n):

$$g(n) \in O(f(n)) \Leftrightarrow \exists n_0, c > 0 : \forall n \geq n_0, 0 \leq g(n) \leq c \cdot f(n).$$

Функция $f(n)$ дает асимптотическую верхнюю границу для функции $g(n)$ (рис. 1.2).

Этот класс нас будет особо интересовать, так как если есть два алгоритма – алгоритм 1 трудоемкостью f_1, алгоритм 2 трудоемкостью f_2 – и $f_1 \in O(f_2)$, то это говорит о том, что алгоритм 2 решает задачу не быстрее, чем алгоритм 1.

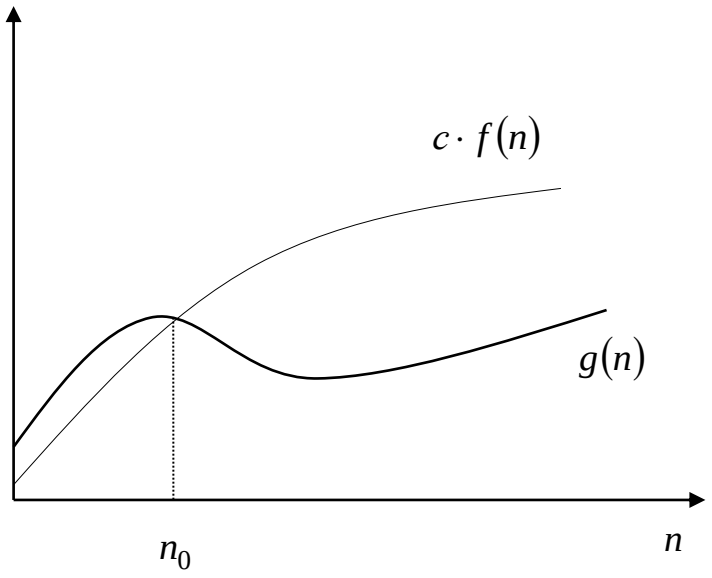

Рис. 1.2

Заметим, что в данном определении функция $f(n)$ ограничивает функцию $g(n)$ лишь для некоторой константы $c > 0$. Другой класс $o(f(n))$ («о малое» от f от n) ограничивает функцию $g(n)$ для любой константы $c > 0$:

$$g(n) \in o(f(n)) \Leftrightarrow \forall c > 0, \exists n_0 : \forall n \geq n_0, 0 \leq g(n) < c \cdot f(n),$$

т. е. верхняя граница не является асимптотически точной оценкой функции. Говорят, что функция $g(n)$ асимптотически меньше $f(n)$. Например, $3 \cdot n^2 = \mathrm{O}\left(n^2\right)$, но $3 \cdot n^2 \neq \mathrm{o}\left(n^2\right)$.

2. $\Omega(f(n))$ – это класс функций, которые растут, по крайней мере, так же быстро, как и функция $f(x)$ («омега большое» от f от n):

$$g(n) \in \Omega(f(n)) \Leftrightarrow \exists n_0, c > 0 : \forall n \geq n_0, 0 \leq c \cdot f(n) \leq g(n).$$

Функция $f(n)$ дает асимптотическую нижнюю границу для функции $g(n)$ (рис. 1.3). Поскольку нас интересует трудоемкость алгоритмов, то класс $\Omega(f(n))$ не будет интересовать, так как, например, в класс $\Omega(n^2)$ входят все функции растущие быстрее, чем n^2, например, n^3 или 2^n.

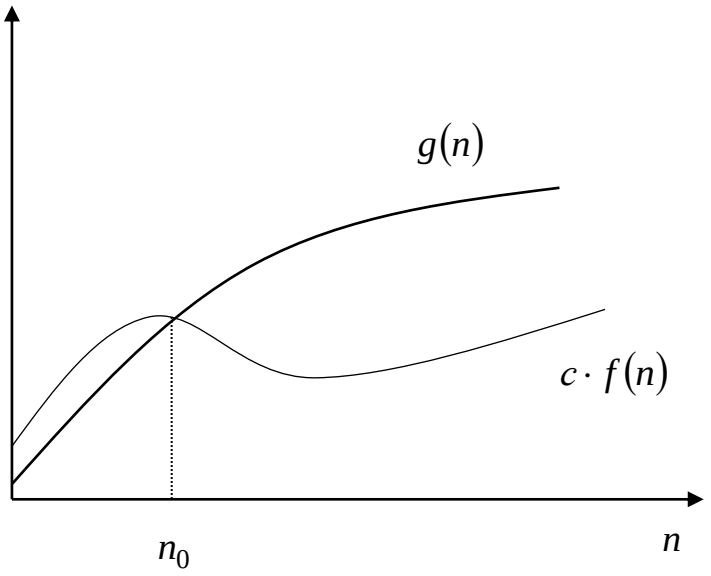

Рис. 1.3

Заметим, что в данном определении функция $f(n)$ ограничивает функцию $g(n)$ лишь для некоторой константы $c > 0$. Другой класс $\omega(f(n))$ («омега малое» от f от n) ограничивает функцию $g(n)$ для любой константы $c > 0$,

$$g(n) \in \omega(f(n)) \Leftrightarrow \forall c > 0, \exists n_0 : \forall n \geq n_0, 0 \leq c \cdot f(n) < g(n),$$

т. е. нижняя граница не является асимптотически точной оценкой функции. Например, $\frac{1}{3} \cdot n^2 = \Omega\left(n^2\right)$, но $\frac{1}{3} \cdot n^2 \neq \omega\left(n^2\right)$.

3. $\Theta(f(n))$– это класс функций, которые растут с той же скоростью, что и функция $f(x)$ («тэтта большое от f от n»):

$$g(n) \in \Theta(f(n)) \Leftrightarrow \exists n_0, c_1, c_2 > 0 : \forall n \geq n_0,\ c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n).$$

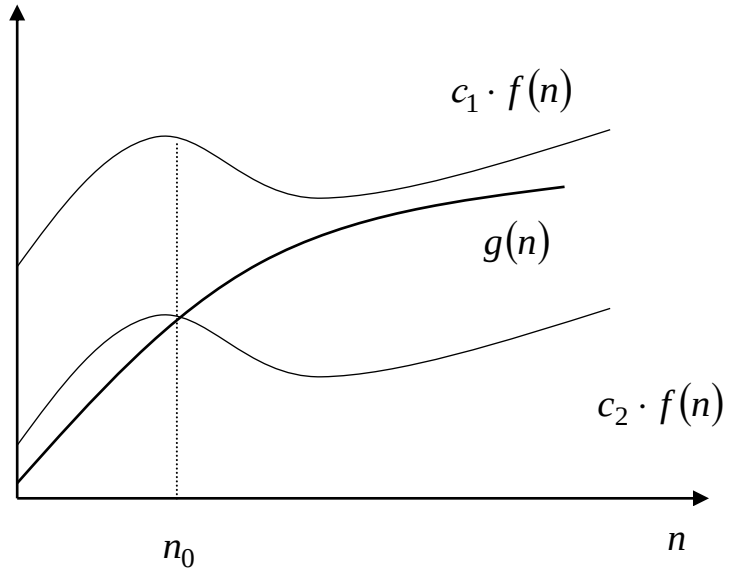

Рис. 1.4

Говорят, что функция $f(n)$ является асимптотически точной оценкой для функции $g(n)$(рис. 1.4). Это обозначение более сильное, чем $O(f(n))$ и $\Omega(f(n))$. Этот класс нас также не будет интересовать, так как нас интересуют те алгоритмы, которые решают задачу лучше, чем уже изученные.

Замечание 1.3. Каждый из рассматриваемых трех классов – множество, поэтому правильнее писать $f(n) \in \Theta(g(n))$, но мы будем также употреблять эквивалентную запись: $f(n) = \Theta(g(n))$.

Для асимптотики $O(g(n))$ справедливы следующие правила:

1) Если $f(n) = O(g(n))$ и $g(n) = O(h(n))$, то $f(n) = O(h(n))$.

2) Если $f(n) = O(k \cdot g(n))$, то $f(n) = O(g(n))$.

3) Если $f_1(n) = O(g_1(n))$ и $f_2(n) = O(g_2(n))$, то выполняется $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$.

4) Если $f_1(n) = O(g_1(n))$ и $f_2(n) = O(g_2(n))$, то выполняется $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$.

Упражнение 1.1. Сформулировать правила для асимптотик $\Omega(f(n))$ и $\Theta(f(n))$.

Напомним некоторые сведения из математики. Полиномом степени *d* от аргумента *n* называется функция $p(n)$ следующего вида:

$$p(n) = \sum_{i=0}^{d} a_i \cdot n^i,$$

где $a_0, a_1, \ldots, a_d$ – коэффициенты полинома и $a_d \neq 0$. Полином является асимптотически положительной функцией тогда и только тогда, когда $a_d > 0$.

Функция $f(n)$ полиномиально ограничена, если существует такая константа *k*, что $f(n) = \mathrm{O}(n^k)$.

Функция полилогарифмически ограничена, если существует такая константа *k*, что $f(n) = \mathrm{O}(\log^k n)$. Любая положительная полиномиальная функция возрастает быстрее, чем любая полилогарифмическая функция.

Любая показательная (экспоненциальная) функция $f(n) = a^n$, основание которой $a > 1$, возрастает быстрее любой полиномиальной функции. Экспонента – это показательная функция $f(n) = e^n$, где e – основание натурального логарифма $\approx 2,718281$.

Для факториала справедливы следующие утверждения:

$$n! = \sqrt{2 \cdot \pi \cdot n} \cdot \left(\frac{n}{e}\right)^n \cdot \left(1 + \Theta\left(\frac{1}{n}\right)\right),$$

$$n! = \mathrm{o}(n^n),$$

$$n! = \omega(2^n).$$

Как следует из последних неравенств, факториал возрастает быстрее, чем показательная функция с основанием 2, но медленнее, чем функция n^n.

Определение 1.5. Алгоритм называется полиномиальным, если его трудоемкость $T(l) = \mathrm{O}(p(l))$, где $p(l)$ – некоторый полином или полиномиально ограниченная функция.

Определение 1.6. Алгоритм называется экспоненциальным, если его трудоемкость $T(l) = \Omega(\exp(l))$, где $\exp(l)$ – некоторая экспоненциальная функция.

Упражнение 1.2. Доказать, что если $f(n) = \sum_{i=0}^{d} a_i \cdot n^i$, $a_d > 0$, то $f(n) = \Theta(n^d)$.

Замечание 1.4. В определении полиномиального алгоритма асимптотику $O(p(l))$ можно заменить асимптотикой $\Theta(p(l))$. В определении экспоненциального алгоритма это сделать нельзя, так как будет отброшен целый ряд функций, которые растут быстрее, чем экспоненциальная функция, например, $n!$, n^n (неверны неравенства $n! \le c \cdot a^n$, $n^n \le c \cdot a^n$).

Различие между полиномиальными и экспоненциальными алгоритмами заметно при решении задач большой размерности.

Таблица 1.3

Временная сложность	$n=10$	$n=20$	$n=30$
n	0,00001 с	0,00002 с	0,00003 с
n^2	0,0001 с	0,0004 с	0,0009 с
n^3	0,001 с	0,008 с	0,027 с
n^5	0,1 с	3,2 с	24,3 с
2^n	0,001 с	1 с	17,9 мин
3^n	0,059 с	58 мин	6,5 лет

Следует отметить очень быстрый рост двух приведенных экспонент (табл. 1.3). Различие между полиномиальными и экспоненциальными алгоритмами проявляется еще более убедительно, если проанализировать влияние увеличения быстродействия ЭВМ на время работы алгоритмов. Так, для функции $f=2^n$ увеличение скорости вычислений в 1000 раз приводит лишь к тому, что размерность наибольшей задачи, разрешимой за один час, возрастет только на 10, в то время как для функции $f=n^5$ эта размерность возрастет почти в 4 раза.

Таким образом, колоссальный рост скорости вычислений, вызванный появлением нынешнего поколения цифровых вычислительных машин, не уменьшает значение эффективных алгоритмов.

С другой стороны, некоторые экспоненциальные алгоритмы достаточно эффективны на практике, когда размеры решаемых задач невелики. Например, при $n=20$ функция $f=2^n$ ведет себя лучше, чем функция $f=n^5$. Кроме того, известны некоторые экспоненциальные алгоритмы, весьма хорошо зарекомендовавшие себя на практике. Дело в том, что трудоемкость определена как мера поведения алгоритма в наихудшем случае. Утверждение о том, что алгоритм имеет трудоемкость 2^n, означает, что решение по крайней мере одной задачи размерности n тре-

бует времени порядка 2^n, но на самом деле может оказаться, что для большинства других задач затраты времени значительно меньше.

Следует отметить, что большинство экспоненциальных алгоритмов – это просто варианты полного перебора.

Пример 1.3. Оределить трудоемкость алгоритма вычисления факториала: $n!=1\cdot 2\cdot\ldots\cdot n$.

Будем считать, что произведение вычисляется последовательно.

Решение

Размерность задачи	$l=\lceil \log_2 n \rceil \quad (2^{l-1} < n \le 2^l)$
Число арифметических операций	$c_1 n + c_2$
Трудоемкость алгоритма	$T(l) \ge c_3 \cdot n \ge \frac{c_3}{2}\cdot 2^l,\ (n \ge 2)$ $T(l)=\Omega(2^l)$ – алгоритм экспоненциальный

Пример 1.4. Определить временную сложность алгоритма для вычисления суммы n элементов массива: $s=a_1+a_2+\cdots+a_n$.

Суммирование производится последовательно.

Решение

Размерность задачи	$c_1\cdot n \le l \le c_2\cdot n \quad \left(\frac{1}{c_2}\cdot l \le n \le \frac{1}{c_1}\cdot l\right)$
Число арифметических операций	$c_3 n + c_4$
Трудоемкость алгоритма	$T(l) \le c\cdot n \le \frac{c}{c_1}\cdot l$ $T(l)=\mathrm{O}(l)$ – алгоритм полиномиальный

Пример 1.5. Определить временную сложность алгоритма определения простоты числа n:

for $i := 2$ to $\sqrt{n}$ *do*

  if $n \bmod i = 0$ then *exit*;

Решение

Размерность задачи	$l=\lceil \log_2 n \rceil \quad (2^{l-1} < n \le 2^l)$

Число арифметических операций	$c_1\sqrt{n}+c_2$
Трудоемкость алгоритма	$T(l)\ge c_3\sqrt{n}>c_3(2^{l-1})^{1/2}$ $T(l)=\Omega(2^{l/2})$ – алгоритм экспоненциальный

1.2. ПОНЯТИЕ РЕКУРРЕНТНОГО СООТНОШЕНИЯ

Для некоторой задачи под подзадачей мы будем понимать ту же задачу, но с меньшим числом параметров или с тем же числом параметров, но при этом хотя бы один из параметров имеет меньшее значение. Одним из основных способов решения задач является сведение к решению такого набора подзадач, чтобы из их решений было возможно получить решение исходной задачи. Найденный способ сведения исходной задачи к решению некоторых подзадач может быть записан в виде соотношений, в которых значение функции, соответствующей исходной задаче, выражается через значения функций, соответствующих подзадачам.

Определение 1.7. Соотношения, которые связывают одни и те же функции, но с различными значениями аргументов, называются рекуррентными соотношениями или рекуррентными уравнениями.

Определение 1.8. Рекуррентное уравнение будем называть правильным, если значения аргументов у любой из функций в правой части соотношения меньше значения аргументов у любой из функций в левой части соотношения; если аргументов несколько, то достаточно уменьшения одного из них.

Другими словами, правильное рекуррентное уравнение описывает функцию с использованием ее самой, но только с меньшими значениями аргументов:

$T(n)=T(n-1)+T(n-2)$ – правильное рекуррентное уравнение;

$T(n)=T(n-1)+T(n+1)$ – неправильное рекуррентное уравнение.

Определение 1.9. Правильное рекуррентное уравнение называется полным, если оно определено для всех допустимых значений аргументов.

Пример 1.6. Приведем пример полного рекуррентного уравнения:

$$\begin{cases} T(n)=T(n-1)+a_n, \quad n>1, \\ T(1)=a_1. \end{cases}$$

В дальнейшем будем предполагать, что область определения функции $T(n)$ – это множество неотрицательных целых чисел $\mathrm{N}=\{0,1,\ldots\}$ и сама

функция $T(n)$ принимает только неотрицательные целочисленные значения. Это допущение вызвано тем, что функция $T(n)$ будет нами использоваться для описания времени работы алгоритма.

Пример 1.7. Рассмотрим задачу поиска максимального элемента в массиве (предполагаем, что первый элемент максимальный; последовательно просматриваем все элементы массива, сравниваем их с текущим максимальным элементом и, в случае необходимости, корректируем текущий максимум). Обозначим через $T(n)$ количество арифметических операций, которые необходимо выполнить, чтобы найти максимальный элемент из n элементов массива. Тогда рекуррентное соотношение для количества арифметических операций описанного алгоритма будет иметь вид

$$\begin{cases} T(n) = T(n-1) + C, \quad n > 1, \\ T(1) = 1, \end{cases}$$

где $C = \mathrm{O}(1)$ – количество арифметических операций, необходимое для сравнения текущего элемента с максимальным и, в случае необходимости, корректировки текущего максимума.

Пример 1.8. Рассмотрим задачу поиска максимального и минимального элементов из n элементов массива. Предположим, что количество элементов массива является степенью числа 2. Для решения задачи будем использовать следующий алгоритм:

- делим массив на две части, затем в каждой из частей находим максимальный и минимальный элементы (для этого к каждой из частей применяем этот же алгоритм, т. е. делим часть на две части и т. д., разбиение прекращаем, когда в части остается 2 элемента);
- выбираем наибольший элемент из двух максимальных и наименьший элемент из двух минимальных.

Обозначим через $T(n)$ количество арифметических операций, которые необходимо выполнить, чтобы найти описанным алгоритмом максимальный и минимальный элементы среди n элементов массива. Тогда рекуррентное соотношение для количества арифметических операций описанного алгоритма будет иметь вид

$$\begin{cases} T(n) = 2 \cdot T\left(\dfrac{n}{2}\right) + C_1, \quad n > 2, \\ T(2) = C_2, \end{cases}$$

где $C_1 = \mathrm{O}(1)$ – количество арифметических операций, необходимое для выбора наибольшего элемента среди двух максимальных и наименьшего элемента среди двух минимальных; $C_2 = \mathrm{O}(1)$ – выбор максимального и минимального среди двух элементов массива.

В примерах 1.7 и 1.8 нами были выписаны рекуррентные соотношения для количества арифметических операций двух алгоритмов. Если мы решим эти рекуррентные соотношения, то получим функции $T(n)$, определяющие число арифметических операций алгоритмов. Выражая данные функции через размерность задачи, мы получим трудоемкости описанных алгоритмов.

Упражнение 1.3. Определить, какие из приведенных ниже рекуррентных уравнений являются полными.

1. $\begin{cases} T(n) = T(n-1) + cn,\ n \geq 2, \\ T(0) = c. \end{cases}$
2. $\begin{cases} T(n) = T(n-1) + cn,\ n \geq 2, \\ T(1) = c. \end{cases}$
3. $\begin{cases} T(n) = T(n-1) + cn,\ n \geq 1, \\ T(0) = c. \end{cases}$
4. $\begin{cases} T(n) = 2 \cdot T(n-2) + c \cdot (n-1),\ n \geq 2, \\ T(1) = C_2, \\ T(0) = C_1. \end{cases}$
5. $\begin{cases} T(n) = T(n-2) + C_1 \cdot n,\ n \geq 1, \\ T(0) = C_2. \end{cases}$

1.3. РЕШЕНИЕ РЕКУРРЕНТЫХ СООТНОШЕНИЙ

Рассмотрим основные методы решения рекуррентных уравнений.

1. Оценка решения рекуррентного уравнения. Метод подстановок.
2. Метод итераций.
3. Метод рекурсивных деревьев.

Для решения рекуррентных уравнений нам будут нужны некоторые математические навыки. Напомним, что:

• сумма арифметической прогрессии (последовательность чисел $a_1, a_2, \ldots, a_n$, в которой каждое последующее число отличается от предыдущего на одно и то же число d: $a_i = a_{i-1} + d$, $i = 2, \ldots, n$):

$$a_1 + a_2 + \ldots + a_n = \frac{a_1 + a_n}{2} \cdot n;$$

• сумма геометрической прогрессии (последовательность чисел $a_1, a_2, \ldots, a_n$, в которой каждое последующее число получается из предыдущего умножением на одно и то же число q: $a_i = q \cdot a_{i-1}, q \neq 1, i = 2, \ldots, n$):

$$q^0 + q^1 + \ldots + q^n = \frac{q^{n+1} - 1}{q - 1},$$

$$q^1 + q^2 + \ldots + q^n = \frac{q\left(q^n - 1\right)}{q - 1},$$

$$1 + q^1 + q^2 + \ldots = \frac{1}{1 - q}, \quad |q| < 1;$$

• логарифмом числа a по основанию b ($\log_b a$) называется такая степень, в которую нужно возвести b, чтобы получить a:

$$\log_b 1 = 0,$$

$$\log_b (x \cdot y) = \log_b x + \log_b y,$$

$$\log_b \frac{x}{y} = \log_b x - \log_b y,$$

$$\log_b x^y = y \cdot \log_b x,$$

$$a^{\log_b x} = x^{\log_b a},$$

$$b^{\log_b a} = a,$$

$$\log_b a = \frac{\log_x a}{\log_x b}.$$

Оценка решения рекуррентного уравнения. Метод подстановок

Рассмотрим следующее правильное рекуррентное уравнение:

$$\begin{cases} T(n) = f\left(T(\alpha_1 \cdot n), T(\alpha_2 \cdot n), \ldots, T(\alpha_l \cdot n), n\right), \quad n > 1, \\ T(1) = C, \end{cases} \tag{1. 1}$$

где f – некоторая неубывающая функция по каждой из переменных. Так как рекуррентное уравнение правильное, то $0<\alpha_i<1$ для любых значений $i=1,\ldots,l$.

Определим мажорирующую функцию (или мажоранту) $g(n)$ для функции $T(n)$ как

$$g(n)\geq T(n), \quad \forall n. \tag{1. 2}$$

Метод подстановок состоит в подборе мажоранты (верхней границы) для функции $T(n)$. При этом надо стремиться к тому, чтобы мажоранта имела наименьший возможный порядок. Сначала методом подбора определяется только ее общий вид (делается догадка о виде решения), например, $g(n)=a\cdot n^2+b$, где a и b – некоторые пока еще не определенные параметры, а затем делается попытка определить значение параметров (в нашем примере это a и b), которые не должны зависеть от n.

ТЕОРЕМА 1.1. Функция $g(n)$ является мажорантой для функции $T(n)$, если существуют ее не зависящие от n параметры, что

$$\begin{cases} g(n)\geq f\left(g(\alpha_1\cdot n), g(\alpha_2\cdot n),\ldots, g(\alpha_l\cdot n), n\right), \quad n>1, \\ g(1)\geq T(1). \end{cases} \tag{1. 3}$$

Доказательство. Для доказательства того, что функция $g(n)$ является мажорантой для функции $T(n)$, воспользуемся методом математической индукции.

Сначала проверяем истинность неравенства (1.2) для $n=1$. Если неравенство не выполняется, то мажорирующая функция выбрана слишком маленькой и необходимо ее изменить (т. е. увеличить) путем добавления некоторой константы, не зависящей от n, либо путем увеличения порядка функции. После чего повторить попытку доказательства теоремы с модифицированной функцией $g(n)$.

Если неравенство (1.2) для $n=1$ выполняется, то предположим, что оно верно и для всех $k<n$:

$$g(k)\geq T(k), \tag{1. 4}$$

и попытаемся определить, существуют ли такие неопределенные параметры функции $g(n)$ (для ранее приведенного примера это a и b), что неравенство (1.4) верно и для $k=n$.

Рассмотрим некоторую неубывающую функцию $h(x)$. Очевидно, что для этой функции из предположения (1.4) следует, что

$$h\big(g(k)\big) \geq h\big(T(k)\big).$$

Аналогично из (1.4), принимая во внимание (1.1), следует, что

$$\begin{aligned} & f\left(g(\alpha_1 \cdot n), g(\alpha_2 \cdot n), \ldots, g(\alpha_l \cdot n), n\right) \geq \\ \geq\ & f\left(T(\alpha_1 \cdot n), T(\alpha_2 \cdot n), \ldots, T(\alpha_l \cdot n), n\right) = T(n). \end{aligned} \qquad (1.5)$$

Поэтому если мы найдем такую функцию $g(n)$, что выполняется неравенство (1.3), то из неравенства (1.5) будет следовать, что $g(n) \geq T(n)$, т. е. неравенство (1.4) будет верно для $k = n$. В этом случае в силу математической индукции функция $g(n)$ является мажорантой для функции $T(n)$. В противном случае функция $g(n)$ выбрана неправильно, и необходимо выбрать другую функцию $g(n)$, возможно большего порядка, и повторить для нее описанный выше процесс проверки. Доказательство теоремы завершено.

Таким образом, метод подстановок заключается в том, что методом подбора находится такая функция $g(n)$ (как правило, на начальном этапе в качестве мажорирующей функции берут некоторый полином, надеясь на то, что алгоритм полиномиальный), при подстановке которой в исходное рекуррентное уравнение (1.1) вместо $T(n)$ получается верное неравенство (1.3). Найденная функция является мажорантой для решения исходного рекуррентного уравнения. Заметим, что мажорирующая функция должна быть наименьшего возможного порядка.

Пример 1. 9. Построить мажоранту для решения следующего рекуррентного уравнения:

$$\begin{cases} T(n) = 2T\left(\dfrac{n}{2}\right) + C_2,\ n > 1,\ C_2 > 0, \\ T(1) = C_1,\ C_1 > 0. \end{cases}$$

Решение. Попытка 1. Предположим, что $g(n) = a \cdot n$. Проверим верность неравенства (1.2) для $n = 1$:

$$T(1) \leq g(1),$$

$$C_1 \leq a,$$

$$a :\overset{\text{def}}{=} C_1.$$

По теореме 1.1, если существует не зависящий от n коэффициент a функции $g(n) = a \cdot n$, то мы найдем его из следующего неравенства:

$$g(n) \geq 2 \cdot g\left(\frac{n}{2}\right) + C_2,$$

$$a \cdot n \geq 2 \cdot \frac{a \cdot n}{2} + C_2,$$

$$0 \geq C_2.$$

Приходим к противоречию, так как по условию коэффициент $C_2 > 0$.

Попытка 2. Предположим, что $g(n) = a \cdot n + b$. Проверим верность неравенства (1.2) для $n = 1$:

$$T(1) \leq g(1),$$

$$C_1 \leq a + b,$$

$$a :\overset{\text{def}}{=} C_1 - b.$$

По теореме 1.1, если существуют не зависящие от n коэффициенты a и b функции $g(n) = a \cdot n + b$, то мы найдем их из следующего неравенства:

$$g(n) \geq 2 \cdot g\left(\frac{n}{2}\right) + C_2,$$

$$a \cdot n + b \geq 2 \cdot \left(\frac{a \cdot n}{2} + b\right) + C_2,$$

$$a \cdot n + b \geq a \cdot n + 2b + C_2,$$

$$b \leq -C_2,$$

$$b :\overset{\text{def}}{=} -C_2.$$

Таким образом, поскольку $b = -C_2$, имеем $a = C_1 - b = C_1 + C_2$, и функция $g(n) = a \cdot n + b = (C_1 + C_2) \cdot n - C_2$ является мажорантой для решения рекуррентного уравнения:

$$\begin{cases} T(n) = 2T\left(\frac{n}{2}\right) + C_2,\ n > 1,\ C_2 > 0, \\ T(1) = C_1,\ C_1 > 0. \end{cases}$$

т. е. $T(n) \leq (C_1 + C_2) \cdot n - C_2 = \mathrm{O}(n)$.

Пример 1.10. Построить мажоранту для решения следующего рекуррентного уравнения:

$$\begin{cases} T(n) = 2T\left(\frac{n}{2}\right) + C_1 \cdot n,\ n > 1,\ C_1 > 0, \\ T(1) = C_2,\ C_2 > 0. \end{cases}$$

Решение.

П о п ы т к а 1. Предположим, что $g(n) = a \cdot n$. Проверим верность неравенства (1.2) для $n = 1$:

$$T(1) \le g(1),$$
$$C_2 \le a,$$
$$a :\overset{\text{def}}{=} C_2.$$

По теореме 1.1, если существует не зависящий от n коэффициент a функции $g(n) = a \cdot n$, то мы найдем его из следующего неравенства:

$$g(n) \ge 2 \cdot g\left(\frac{n}{2}\right) + C_1 \cdot n,$$
$$a \cdot n \ge 2 \cdot \frac{a \cdot n}{2} + C_1 \cdot n,$$
$$0 \ge C_1 \cdot n \ge C_1.$$

Приходим к противоречию, так как по условию коэффициент $C_1 > 0$.

П о п ы т к а 2. Предположим, что $g(n) = a \cdot n + b$. Проверим верность неравенства (1.2) для $n = 1$:

$$T(1) \le g(1),$$
$$C_2 \le a + b,$$
$$a :\overset{\text{def}}{=} C_2 - b.$$

По теореме 1.1, если существуют не зависящие от n коэффициенты a и b функции $g(n) = a \cdot n + b$, то мы найдем их из следующего неравенства:

$$g(n) \ge 2 \cdot g\left(\frac{n}{2}\right) + C_1 \cdot n,$$
$$a \cdot n + b \ge 2 \cdot \left(\frac{a \cdot n}{2} + b\right) + C_1 \cdot n,$$

$$a \cdot n + b \geq a \cdot n + 2b + C_1 \cdot n,$$
$$-b \geq C_1 \cdot n.$$

Из последнего неравенства следует, что b всегда зависит от n. Следовательно, не существует не зависящих от n коэффициентов a и b функции $g(n) = a \cdot n + b$, удовлетворяющих неравенству (1.3).

П о п ы т к а 3. Предположим, что $g(n) = a \cdot n \cdot \log n$. Проверим верность неравенства (1.2) для $n = 1$:

$$T(1) \leq g(1),$$
$$C_2 \leq a \cdot 1 \cdot \log 1,$$
$$C_2 \leq 0.$$

Приходим к противоречию, так как по условию коэффициент $C_2 > 0$.

П о п ы т к а 4. Предположим, что $g(n) = a \cdot n \cdot \log n + b$. Проверим вер-
ность неравенства (1.2) для $n = 1$:

$$T(1) \leq g(1),$$
$$C_2 \leq a \cdot 1 \cdot \log 1 + b,$$
$$C_2 \leq b,$$
$$b :\overset{\text{def}}{=} C_2.$$

По теореме 1.1, если существуют не зависящие от n коэффициенты a и b функции $g(n) = a \cdot n \cdot \log n + b$, то мы найдем их из следующего неравенства:

$$g(n) \geq 2 \cdot g\left(\frac{n}{2}\right) + C_1 \cdot n,$$
$$a \cdot n \cdot \log n + b \geq 2 \cdot \left(\frac{a \cdot n}{2} \cdot \log \frac{n}{2} + b\right) + C_1 \cdot n,$$
$$a \cdot n \cdot \log n + b \geq a \cdot n \cdot \log \frac{n}{2} + 2b + C_1 \cdot n,$$
$$a \cdot n \cdot \log n + b \geq a \cdot n \cdot \log n - a \cdot n + 2b + C_1 \cdot n,$$
$$-b \geq -a \cdot n + C_1 \cdot n,$$
$$-b \geq n \cdot (C_1 - a),$$
$$b \leq n \cdot (a - C_1).$$

Если в последнем неравенстве положить $a - C_1 \geq b$, то для любого $n > 1$ неравенство $b \leq n \cdot (a - C_1)$ будет верно. Таким образом, полагаем

$$b \overset{\text{def}}{=} C_2,$$

$$a \overset{\text{def}}{=} b + C_1 = C_2 + C_1,$$

и функция $g(n) = a \cdot n \cdot \log n + b = (C_1 + C_2) \cdot n \cdot \log n + C_2$ является мажорантой для решения рекуррентного уравнения:

$$\begin{cases} T(n) = 2T\left(\frac{n}{2}\right) + C_1 \cdot n,\ n > 1,\ C_1 > 0, \\ T(1) = C_2,\ C_2 > 0, \end{cases}$$

т. е. $T(n) \leq (C_1 + C_2) \cdot n \cdot \log n + C_2 = \mathrm{O}(n \cdot \log n)$.

Метод итераций

Данный метод заключается в том, что рекуррентное уравнение расписывается через множество других и затем происходит суммирование полученного выражения.

Пример 1. 11. Решить рекуррентное уравнение, используя метод итераций:

$$\begin{cases} T(n) = T(n-1) + 1, \quad n \geq 1, \\ T(0) = 1. \end{cases}$$

Несложно увидеть, что данное уравнение задает количество арифметических операций алгоритма вычисления факториала: $n! = 1 \cdot 2 \cdot \ldots \cdot n$.

Решение. В правой части равенства $T(n) = T(n-1) + 1$ мы должны исключить выражение $T(n-1)$. Для этого в уравнении $T(n) = T(n-1) + 1$ заменим каждое вхождение n на $n-1$. Получим эквивалентное выражение для $T(n-1)$: $T(n-1) = T(n-2) + 1$, которое подставим в исходное уравнение:

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

Теперь необходимо исключить значение $T(n-2)$. Поступаем аналогичным образом: $T(n-2) = T(n-3) + 1$ и получаем:

$$T(n) = T(n-2) + 2 = T(n-3) + 3.$$

Несложно увидеть, что на k-м шаге исходное уравнение будет иметь следующий вид: $T(n)=T(n-k)+k$. Процесс будет завершен, когда в правой части придем к начальному условию $T(0)=1$ (это произойдет при $k=n$):

$$T(n)=T(0)+n=1+n.$$

Таким образом, точным решением рекуррентного соотношения является функция $T(n)=1+n.$

Пример 1. 12. Решить рекуррентное уравнение, используя метод итераций:

$$\begin{cases} T(n)=2\cdot T\left(\dfrac{n}{2}\right)+2, \quad n\geq 2, \\ T(1)=2. \end{cases}$$

Решение. Будем предполагать, что n – степень числа 2. Несложно увидеть, что данное уравнение задает количество арифметических операций алгоритма поиска максимального и минимального элементов из примера 1.8.

$$T(n)=2\cdot T\left(\frac{n}{2}\right)+2,$$

$$T\left(\frac{n}{2}\right)=2\cdot T\left(\frac{n}{2^2}\right)+2,$$

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

$$T\left(\frac{n}{2^2}\right)=2\cdot T\left(\frac{n}{2^3}\right)+2,$$

$$T(n)=2^3\cdot T\left(\frac{n}{2^3}\right)+2^3+2^2+2^1,$$

.....

$$T(n)=2^k\cdot T\left(\frac{n}{2^k}\right)+2^k+2^{k-1}+\ldots+2^1.$$

Процесс будет завершен, когда придем к начальному условию: $T(1)$. Это произойдет при

$$\frac{n}{2^k}=1, \quad n=2^k.$$

Тогда последняя сумма, с учетом формулы геометрической прогрессии со знаменателем $q=2$, будет иметь следующий вид:

$$T(n)=n\cdot T(1)+\frac{2\cdot\left(2^k-1\right)}{2-1}=n\cdot T(1)+2(n-1)=2\cdot n+2(n-1)=4\cdot n-2.$$

Таким образом, точным решением рекуррентного соотношения является функция $T(n)=4\cdot n-2$.

Пример 1.13. Решить рекуррентное уравнение, используя метод итераций:

$$\begin{cases} T(n)=2\cdot T\left(\frac{n}{2}\right)+5\cdot n^2, \quad n\geq 2, \\ T(1)=7. \end{cases}$$

Решение. Будем предполагать, что n – степень числа 2.

$$T(n)=2\cdot T\left(\frac{n}{2}\right)+5\cdot n^2,$$

$$T\left(\frac{n}{2}\right)=2\cdot T\left(\frac{n}{2^2}\right)+5\cdot\frac{n^2}{2^2},$$

$$T(n)=2\cdot\left(2\cdot T\left(\frac{n}{2^2}\right)+5\cdot\frac{n^2}{2^2}\right)+5\cdot n^2=2^2\cdot T\left(\frac{n}{2^2}\right)+2\cdot 5\cdot\frac{n^2}{2^2}+5\cdot n^2=$$

$$=2^2\cdot T\left(\frac{n}{2^2}\right)+5\cdot\frac{n^2}{2}+5\cdot n^2,$$

$$T\left(\frac{n}{2^2}\right)=2\cdot T\left(\frac{n}{2^3}\right)+5\cdot\frac{n^2}{2^4},$$

$$T(n)=2^2\cdot T\left(\frac{n}{2^2}\right)+2\cdot 5\cdot\frac{n^2}{2^2}+5\cdot n^2=2^2\cdot\left(2\cdot T\left(\frac{n}{2^3}\right)+5\cdot\frac{n^2}{2^4}\right)+5\cdot\frac{n^2}{2}+5\cdot n^2=$$

$$=2^3\cdot T\left(\frac{n}{2^3}\right)+2^2\cdot 5\cdot\frac{n^2}{2^4}+5\cdot\frac{n^2}{2}+5\cdot n^2=2^3\cdot T\left(\frac{n}{2^3}\right)+5\cdot\frac{n^2}{2^2}+5\cdot\frac{n^2}{2}+5\cdot n^2.$$

.....

$$T(n)=2^k\cdot T\left(\frac{n}{2^k}\right)+5\cdot\frac{n^2}{2^{k-1}}+5\cdot\frac{n^2}{2^{k-2}}+\ldots+5\cdot\frac{n^2}{2^0}=2^k\cdot T\left(\frac{n}{2^k}\right)+5\cdot n^2\cdot\sum\nolimits_{i=0}^{k-1}\frac{1}{2^i}=$$

$$= 2^k \cdot T\left(\frac{n}{2^k}\right) + 5 \cdot n^2 \cdot \frac{\left(\left(\frac{1}{2}\right)^k - 1\right)}{\frac{1}{2} - 1} = 2^k \cdot T\left(\frac{n}{2^k}\right) + 5 \cdot n^2 \cdot 2 \cdot \left(1 - \left(\frac{1}{2}\right)^k\right).$$

Последняя сумма была преобразована с учетом формулы геометрической прогрессии со знаменателем $q = 1/2$. Алгоритм закончит свою работу, когда мы придем к начальному условию: $T(1)$. Это произойдет при $n/2^k = 1$, $n = 2^k$. Таким образом, получаем точное решение рекуррентного уравнения:

$$T(n) = 2^k \cdot T\left(\frac{n}{2^k}\right) + 5 \cdot n^2 \cdot 2 \cdot \left(1 - \left(\frac{1}{2}\right)^k\right) = n \cdot T(1) + 10 \cdot n^2 \cdot \left(1 - \frac{1}{n}\right) =$$

$$= 7 \cdot n + 10 \cdot n^2 - 10 \cdot n = 10 \cdot n^2 - 3 \cdot n.$$

Метод рекурсивных деревьев

Данный метод является особым способом записи явных (не рекуррентных) функций, получаемых на шагах метода итераций. Метод рекурсивных деревьев заключается в том, что по рекуррентному уравнению итерационно строится древовидная структура, суммирование значений ключей в узлах которой осуществляется специальным образом.

Проиллюстрируем метод рекурсивных деревьев для решения следующего рекуррентного уравнения:

$$\begin{cases} T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 2, \quad n \geq 2, \\ T(1) = 2. \end{cases}$$

Будем предполагать, что n – степень числа 2, т. е. $n = 2^k$.

1. На первой итерации формируется древовидная структура следующего вида (рис. 1.5):

- в корень дерева заносится свободный член исходного рекуррентного уравнения (свободный член рекуррентного уравнения – это та его часть, которая не является рекуррентной функцией, т. е. некоторая явная функция от размера задачи), для нашего рекуррентного соотношения свободный член – 2;

• сыновьями корня являются рекуррентные функции правой части исходного рекуррентного соотношения, для нашего примера у корня с ключом 2 будет два сына: $T\left(n/2\right)$ и $T\left(n/2\right)$.

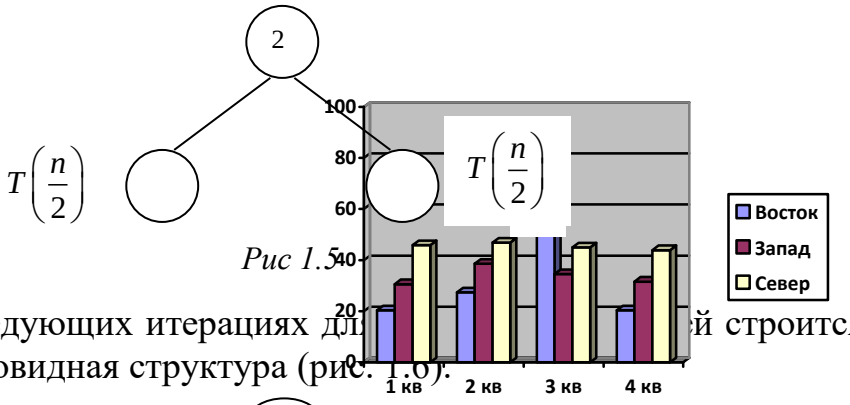

Рис 1.5

2. На последующих итерациях для [illegible]ей строится аналогичная древовидная структура (рис. 1.6).

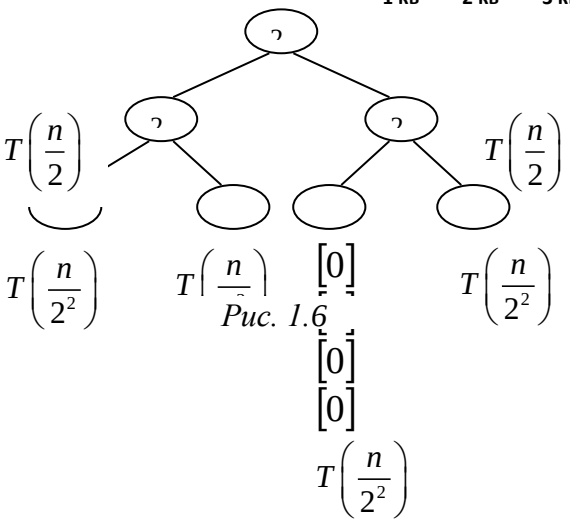

Рис. 1.6

В узел дерева, соответствующий данному сыну, заносится свободный член этого рекуррентного соотношения (= 2), а сыновьями узла полагаются рекуррентные функции из правой части этого рекуррентного соотношения: $T\left(n/2^2\right)$ и $T\left(n/2^2\right)$.

Процесс ветвления в некоторой вершине дерева заканчивается, когда данная вершина соответствует начальным данным уравнения (в нашем примере ветвление прекращается в узле, соответствующем функции $T(1)$, а ключ этого узла полагается равным 2, так как $T(1)=2$), при этом значения внутренних вершин дерева есть некоторые явные функции от размера задачи. Заметим, что висячие вершины построенной древовидной структуры не обязательно одинаково удалены от корня (в нашем примере все висячие вершины находятся на одинаковом удалении от корня).

После построения дерева суммирование значений в вершинах производится следующим образом:

1. Определяют сумму ключей в вершинах, равноудаленных от корня дерева (эти вершины находятся в дереве на одном уровне).
2. Находится максимальная сумма по уровням.
3. Итоговая сумма ограничена сверху одним из следующих значений:

- максимальной суммой, которая умножается на количество уровней;
- суммой, полученной в результате сложения сумм значений по уровням.

Заметим, что количество уровней для дерева равно $\mathrm{O}(\log n)$.

Древовидная структура приведена на рис. 1.7 для рассматриваемого примера. Сумма ключей вершин нулевого уровня – 2, первого уровня – 2^2, …, *k*-го уровня – 2^{k+1}.

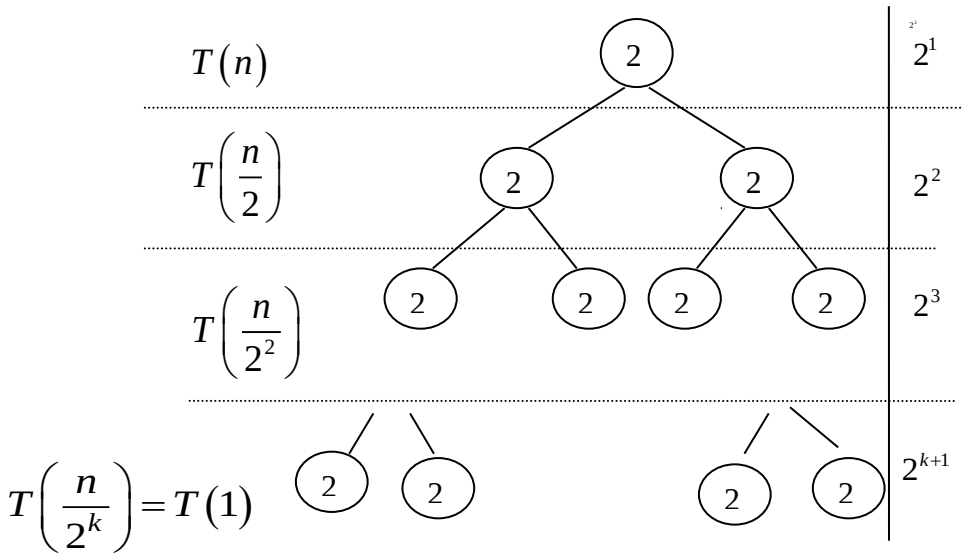

Рис.1.7

Для рекуррентного уравнения сложим суммы, полученные на уровнях:

$$T(n)=\sum_{i=1}^{k+1}2^i=\frac{2\cdot\left(2^{k+1}-1\right)}{2-1}=\left[\frac{n}{2^k}=1;\quad 2^k=n\right]=4n-2.$$

Таким образом, построено точное решение рассматриваемого рекуррентного соотношения.

Пример 1.14. Решить рекуррентное уравнение, используя метод рекурсивных деревьев:

$$\begin{cases} T(n) = 6 \cdot T\left(\frac{n}{6}\right) + 2 \cdot n + 3, \\ T(1) = 1. \end{cases}$$

Предположим, что n – степень числа 6, т. е. $n = 6^k$.

Решение. По данному рекуррентному уравнению построим древовидную структуру (рис. 1.8).

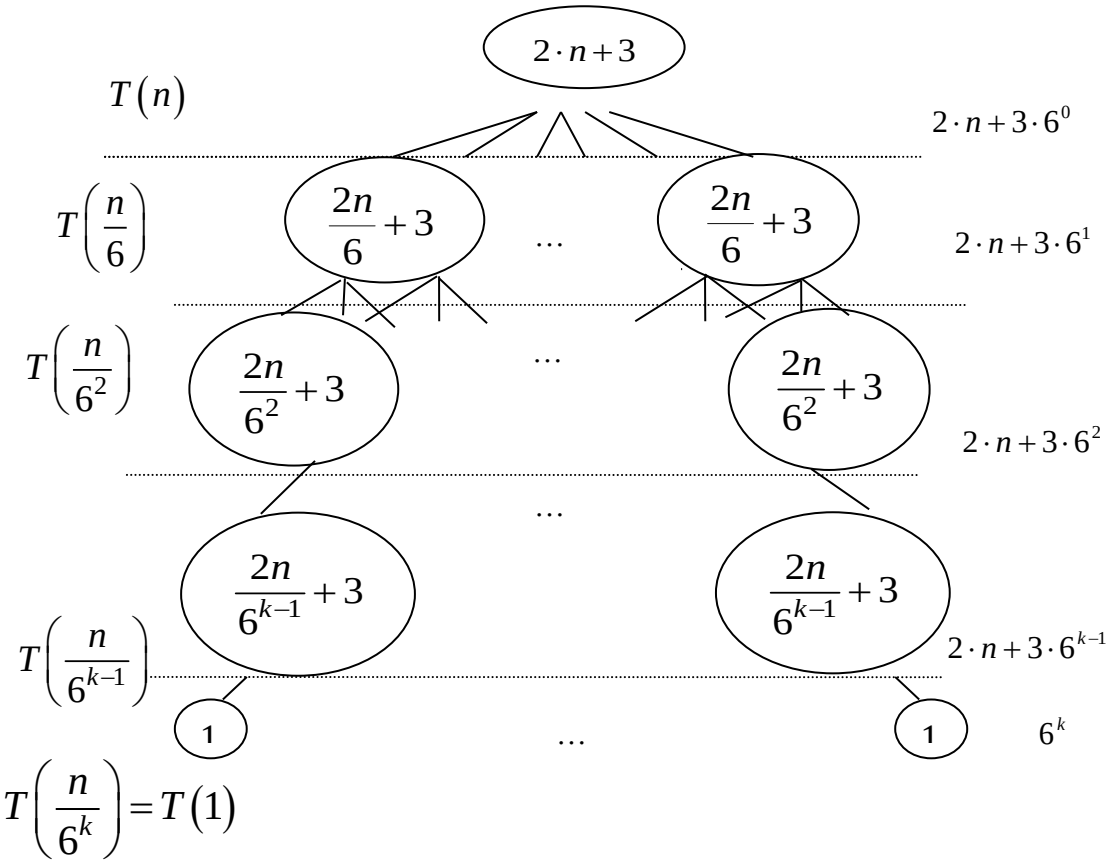

Рис. 1.8

Сумма ключей вершин нулевого уровня – $2 \cdot n + 3 \cdot 6^0$, первого уровня – $2 \cdot n + 3 \cdot 6^1$, …, $(k-1)$-го уровня – $2 \cdot n + 3 \cdot 6^{k-1}$, k-го уровня – 6^k.

Сложим суммы, полученные на уровнях, используя формулу геометрической прогрессии:

$$T(n) = 2 \cdot n \cdot k + 3 \cdot \sum_{i=0}^{k-1} 6^i + 6^k = 2 \cdot n \cdot k + 3 \cdot \frac{6^k - 1}{6 - 1} + 6^k =$$

$$= \left[. \frac{n}{6^k} = 1;\ 6^k = n;\ k = \log_6 n \right] = 2 \cdot n \cdot \log_6 n + \frac{3}{5} \cdot (n - 1) + n =$$

$$= 2 \cdot n \cdot \log_6 n + \frac{8}{5} \cdot n - \frac{3}{5}.$$

Упражнение 1.4. Решить следующие рекуррентные уравнения.

1. $\begin{cases} T(n) = 3T(n-1) - 15,\ n \geq 2, \\ T(1) = 8. \end{cases}$

2. $\begin{cases} T(n) = T(n-1) + n - 1,\ n \geq 2, \\ T(1) = 3. \end{cases}$

3. $\begin{cases} T(n) = 2T(n-2) - 15,\ n > 2, \\ T(2) = 40, \\ T(1) = 40. \end{cases}$

4. $\begin{cases} T(n) = T(n-2) + a,\ a > 0,\ n > 2, \\ T(1) = C_1, \\ T(2) = C_2. \end{cases}$

5. $\begin{cases} T(n) = 4 \cdot T(n/2) - 1,\ n = 2^k,\ k \geq 1, \\ T(4) = 5. \end{cases}$

6. $\begin{cases} T(n) = 2T(\sqrt{n}) + \log_2 n,\ n = x^{2^k},\ k \geq 1, \\ T(x) = 0. \end{cases}$

Теорема о решении рекуррентного уравнения

Рассмотрим рекуррентное соотношение, описывающее время работы алгоритма, в котором задача размером n разбивается на a вспомогательных задач размером n/b каждая, где a и b – положительные константы. Полученные в результате подзадачи решаются рекурсивным методом,

причем время их решения есть $T\left(n/b\right)$. Время, требуемое для разбиения задачи на подзадачи и объединения результатов, полученных при решении подзадач, описывается асимптотически положительной функцией $c \cdot n^k$.

ТЕОРЕМА 1.2. Пусть *a, b, c, k* – некоторые положительные константы. Тогда решение рекуррентного уравнения

$$\begin{cases} T(n) = a \cdot T\left(\dfrac{n}{b}\right) + c \cdot n^k, \\ T(1) = c \end{cases}$$

имеет вид:

$$T(n) = \mathrm{O}\left(n^{\log_b a}\right), \text{ если } a > b^k;$$

$$T(n) = \mathrm{O}\left(n^k \cdot \log n\right), \text{ если } a = b^k;$$

$$T(n) = \mathrm{O}\left(n^k\right), \text{ если } a < b^k.$$

Доказательство. Будем предполагать, что $n = b^m \left(m = \log_b n\right)$ – точная степень числа *b*. Оценка верна не только для точных степеней *b*, так как можно показать, что асимптотическая оценка не изменится, если рассматривать $\left\lfloor n/b \right\rfloor \le n/b$.

$$T(n) = a \cdot \left(a \cdot T\left(\frac{n}{b^2}\right) + c \cdot \left(\frac{n}{b}\right)^k \right) + c \cdot n^k = \ldots =$$

$$= a^m \cdot T(1) + a^{m-1} \cdot c \cdot \left(\frac{n}{b^{m-1}}\right)^k + \ldots + a \cdot c \cdot \left(\frac{n}{b}\right)^k + c \cdot n^k =$$

$$= c \cdot \sum_{i=0}^{m} a^{m-i} \cdot b^{i \cdot k} = c \cdot a^m \cdot \sum_{i=0}^{m} \left(\frac{b^k}{a}\right)^i.$$

Отметим, что

$$a^m = a^{\log_b n} = n^{\log_b a}.$$

Сумма, фигурирующая в равенстве для $T(n)$, является геометрической прогрессией, значение которой зависит от рационального числа $r = b^k/a$. Рассмотрим три возможных случая:

1. Пусть $r<1$. Учитывая, что сумма бесконечно сходящейся геометрической прогрессии со знаменателем $0<a<1$ равна константе, т. е.

$$\sum_{i=0}^{\infty} a^i = \frac{1}{1-a} \quad \text{для} \quad 0<a<1,$$

получим, что

$$\sum_{i=0}^{m} r^i = \frac{1}{1-r}$$

является константой.

Следовательно,

$$T(n) = \mathrm{O}\left(a^m\right) = \mathrm{O}\left(n^{\log_b a}\right).$$

2. Пусть $r=1$. Поскольку $r = b^k \big/ a$, то $a = b^k$. Тогда $a^m = b^{m\cdot k} = n^k$ (так как $n = b^m$). К тому же

$$\sum_{i=0}^{m} r^i = m+1 = \left(m = \log_b n\right) = \log_b n + 1.$$

Следовательно,

$$c \cdot a^m \cdot \sum_{i=0}^{m} r^i = c \cdot n^k \cdot \left(\log_b n + 1\right) = \frac{c}{\log_2 b} \cdot n^k \cdot \log_2 n + c \cdot n^k.$$

Получаем, что

$$T(n) = \mathrm{O}\left(n^k \cdot \log n\right).$$

3. Пусть $r>1$. Учитывая неравенство для суммы геометрической прогрессии

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1} \quad \text{для} \quad a>1,$$

получим, что

$$\sum_{i=0}^{m} r^i = \frac{r^{m+1}-1}{r-1} = \mathrm{O}\left(r^m\right).$$

Таким образом,

$$T(n) = \mathrm{O}\left(a^m \cdot r^m\right) = \mathrm{O}\left(a^m \cdot \left(\frac{b^k}{a}\right)^m\right) = \mathrm{O}\left(b^{k\cdot m}\right) = \mathrm{O}\left(n^k\right).$$

Доказательство теоремы завершено.

Замечание 1.5. Несложно показать, что в теореме 1.2 асимптотика $O(n)$ может быть заменена асимптотикой $\Theta(n)$.

1.4. ПРИМЕРЫ РЕКУРРЕНТНЫХ УРАВНЕНИЙ

Приведем несколько примеров построения и решения рекуррентных уравнений.

Пример 1.15. Вычислить наибольший общий делитель двух чисел *A* и *B*, используя алгоритм Евклида. Схематически определение алгоритма Евклида может быть сформулировано следующим образом: если числа равны, то наибольший общий делитель равен одному из них. В противном случае надо из большего числа вычесть меньшее число и результат запомнить на месте большего числа. После этого задача сводится к нахождению наибольшего общего делителя модифицированных чисел.

НОД(*A*, *B*) = НОД(*A*–*B*, *B*), если *A* > *B*,

НОД(*A*, *B*) = НОД(*A*, *B*–*A*), если *A* < *B*,

НОД(*A*, *B*) = *A*, если *A* = *B*.

Решение. Рассмотрим первый алгоритм решения данной задачи (рекурсивный):

```
function НОД (A, B);
begin
  if  A = B  then НОД : = A
              else   if A > B then  НОД(A–B, B);
                                    else     НОД(A, B–A);
end;
```

Если предположить, что $A > B$, то количество вычитаний числа *B* из числа *A* по крайней мере *A* div *B*. Если $B = 1$, то глубина рекурсии равна *A*. В данном случае глубина рекурсии как минимум целая часть от деления большего числа на меньшее число.

Рассмотрим второй, нерекурсивный (табличный) алгоритм решения данной задачи:

```
begin
 while A ≠ B do
    if A > B then
                A : = A – B
```

else

$B := B - A$;

НОД := A;

end;

Объем памяти нерекурсивного алгоритма равен константе (память, необходимая для представления в памяти чисел A и B).

Сопоставляя два алгоритма, мы видим, что количество арифметических операций сравнимо, а память для рекурсивного алгоритма зависит от входных данных. Конечно, есть зависимость требуемой памяти и для нерекурсивного алгоритма, но зависимость здесь совсем другого рода.

Пример 1.16. Рассмотрим задачу возведения числа 2 в некоторую степень n. Пусть $F(n) = 2^n$.

Справедливы следующие четыре рекуррентных соотношения для решения данной задачи:

1-е соотношение:

$$2^n = 2^{n-1} + 2^{n-1},$$
$$F(n) = F(n-1) + F(n-1).$$

2-е соотношение:

$$2^n = 2 \cdot 2^{n-1},$$
$$F(n) = 2 \cdot F(n-1).$$

3-е соотношение:

$$2^n = \begin{cases} 2^{n/2} \cdot 2^{n/2}, \text{ если } n - \text{четно}, \\ 2 \cdot 2^{n-1/2} \cdot 2^{n-1/2}, \text{ если } n - \text{нечетно}. \end{cases}$$

$$F(n) = \begin{cases} F\left(\frac{n}{2}\right) \cdot F\left(\frac{n}{2}\right), \text{ если } n - \text{четно}, \\ 2 \cdot F\left(\frac{n-1}{2}\right) \cdot F\left(\frac{n-1}{2}\right), \text{ если } n - \text{нечетно}. \end{cases}$$

4-ое соотношение:

$$2^n = \begin{cases} \left(2^{n/2}\right)^2, & \text{если } n - \text{четно}, \\ 2 \cdot \left(2^{n-1/2}\right)^2, & \text{если } n - \text{нечетно}. \end{cases}$$

$$F(n) = \begin{cases} \left(F\left(\frac{n}{2}\right)\right)^2, & \text{если } n - \text{четно}, \\ 2 \cdot \left(F\left(\frac{n-1}{2}\right)\right)^2, & \text{если } n - \text{нечетно}. \end{cases}$$

Каждое из приведенных соотношений приводит к своим алгоритмам (табличным и рекурсивным) решения рассматриваемой задачи. Приведем сначала рекурсивные алгоритмы для каждого из соотношений. Для каждого из алгоритмов выпишем рекуррентное уравнение для количества арифметических операций $T(n)$ и решим его. Будем также определять объем памяти M, который необходим алгоритму.

Рекурсивный алгоритм для соотношения 1

```
function F (k);
begin
  if k = 1 then F : = 2
           else  F : = F(k – 1) + F(k – 1);
end;
```

Рекуррентное соотношение для количества арифметических операций:

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

Здесь слагаемое, равное 1, означает одну операцию сложения двух вычисленных значений функции $F(n-1)$ и $F(n-1)$.

Решая данное рекуррентное уравнение, получим, что

$$T(n) = \Theta\left(2^n\right).$$

Данный алгоритм – экспоненциальный, так как

$$l = \lceil \log_2 n \rceil,$$

$$n = \Theta\left(2^l\right),$$

$$T(l) = \Omega\left(2^{2^l}\right).$$

Требуемый объем памяти равен $M = \mathrm{O}(n)$, поскольку глубина рекурсии не превосходит *n*.

Рекурсивный алгоритм для соотношения 2

```
function F (k);
begin
  if k = 1 then F : = 2
          else  F : = 2 * F(k – 1);
end;
```

Рекуррентное соотношение для количества арифметических операций:

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

Здесь слагаемое, равное 1, означает одну операцию умножения значения функции $F(n-1)$ на число 2.

Решая данное рекуррентное уравнение, получим, что

$$T(n) = \Theta(n).$$

Данный алгоритм – экспоненциальный, так как

$$l = \lceil \log_2 n \rceil,$$

$$n = \Theta\left(2^l\right),$$

$$T(l) = \Omega\left(2^l\right).$$

Требуемый объем памяти равен $M = \mathrm{O}(n)$, поскольку глубина рекурсии равна *n*.

Рекурсивный алгоритм для соотношения 3

```
function F (k);
begin
  if k = 1 then F : = 2
          else  if k – четно then F : = F (k/ 2) * F (k /2)
                                  else F : = 2 * F ((k – 1)/ 2) * F ((k – 1) /2)
end;
```

Рекуррентное соотношение для количества арифметических операций:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 2.$$

Здесь слагаемое, равное 2, означает одну операцию умножения на число 2 и одну операцию умножения значения функции $F\left((k-1)/2\right)$ на значение функции $F\left((k-1)/2\right)$. Решая данное рекуррентное уравнение, получим, что

$$T(n) = \Theta(n).$$

Данный алгоритм – экспоненциальный, так как

$$l = \lceil \log_2 n \rceil,$$

$$n = \Theta\left(2^l\right),$$

$$T(l) = \Omega\left(2^l\right).$$

Требуемый объем памяти равен $M = \mathrm{O}(\log n)$, поскольку глубина рекурсии равна $\log n$.

Рекурсивный алгоритм для соотношения 4

```
function F (k);
begin
  if k = 1 then F : = 2
        else  if k – четно then F : = SQR (F (k/ 2))
                              else F : = 2 * SQR (F ((k – 1)/ 2))
end;
```

Рекуррентное соотношение для количества арифметических операций:

$$T(n) = T\left(\frac{n}{2}\right) + 2.$$

Здесь слагаемое, равное 2, означает одну операцию умножения на число 2 и одну операцию возведения в квадрат (SQR).

Решая данное рекуррентное уравнение, получим, что

$$T(n) = \Theta(\log n).$$

Данный алгоритм – полиномиальный, так как

$$l = \lceil \log_2 n \rceil,$$
$$n = \Theta\left(2^l\right),$$
$$T(l) = \mathrm{O}(l).$$

Требуемый объем памяти равен $M = \mathrm{O}(\log n)$, поскольку глубина рекурсии равна $\log n$.

Анализируя трудоемкость и требуемый объем памяти для каждой из реализаций, получаем, что более эффективным является рекурсивный алгоритм для соотношения 4.

Приведем теперь нерекурсивные (табличные) алгоритмы для соотношений 1)–4). Для этого будем использовать структуру данных **массив** (*F*). Обозначим через $T(n)$ число арифметических операций, чтобы вычислить элемент массива $F[n]$.

Нерекурсивный алгоритм для соотношения 1

```
F [0] : = 1;
for i : = 1 to n do F [i] : = F [i – 1] + F [i – 1];
```

Рекуррентное соотношение для количества арифметических операций:

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

Здесь слагаемое, равное 1, означает одну операцию сложения величины элемента массива $F[n-1]$ с самим собой.

Решая данное рекуррентное уравнение, получим, что $T(n) = \Theta(n)$. Данный алгоритм – экспоненциальный, так как

$$l = \lceil \log_2 n \rceil,$$
$$n = \Theta\left(2^l\right),$$
$$T(l) = \Omega\left(2^l\right).$$

Требуемый объем памяти равен $M = \mathrm{O}(n)$, если использовать массив, а если использовать вместо массива одну переменную, то память $M = \mathrm{O}(1)$.

Нерекурсивный алгоритм для соотношения 2

```
F [0] : = 1;
for i : = 1 to n do F [i] : = 2 * F [i – 1];
```

Рекуррентное соотношение для количества арифметических операций:

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

Здесь слагаемое, равное 1, означает одну операцию умножения элемента массива $F[n-1]$ на число 2. Решая данное рекуррентное уравнение, получим, что $T(n) = \Theta(n)$. Данный алгоритм – экспоненциальный, так как

$$l = \lceil \log_2 n \rceil,$$

$$n = \Theta\left(2^l\right),$$

$$T(l) = \Omega\left(2^l\right).$$

Требуемый объем памяти равен $M = \mathrm{O}(n)$, если использовать массив, а если использовать вместо массива одну переменную, то память $M = \mathrm{O}(1)$.

Нерекурсивный алгоритм для соотношения 3

Данный алгоритм будет использовать структуру данных **стек**. Напомним, что трудоемкость базовых операций добавления и извлечения элемента из стека равна $\mathrm{O}(1)$.

```
 m : = n;
while m >= 1 do
  begin
    В СТЕК(m);    m : = m div 2;
  end;
f : = 1;
while стек не пуст do
   begin
     ИЗ СТЕКА(x);
     if x – четно  then f : = f * f  else  f : = 2 * f * f;
    end;
```

Рекуррентное соотношение для количества арифметических операций:

$$T(n) = T\left(\frac{n}{2}\right) + 2 .$$

Решая данное рекуррентное уравнение, получим, что $T(n)=\Theta(\log n)$. Данный алгоритм – полиномиальный, так как

$$l=\lceil \log_2 n \rceil,$$

$$n=\Theta\left(2^l\right),$$

$$T(l)=\mathrm{O}(l).$$

Требуемый объем памяти равен количеству элементов, хранящихся в стеке: $M=\mathrm{O}(\log n)$.

Нерекурсивный алгоритм для соотношения 4

```
m : = n;
while m >= 1 do
 begin
   В СТЕК(m);   m : = m div 2;
 end;
f : = 1;
while стек не пуст do
   begin
     ИЗ СТЕКА(x);
     if x – четно then f : = SQR(f )  else  f : = 2 * SQR(f );
    end;
```

Рекуррентное соотношение для количества арифметических операций:

$$T(n)=T\left(\frac{n}{2}\right)+2\,.$$

Решая данное рекуррентное уравнение, получим, что $T(n)=\Theta(\log n)$. Данный алгоритм – полиномиальный, так как

$$l=\lceil \log_2 n \rceil,$$

$$n=\Theta\left(2^l\right),$$

$$T(l)=\mathrm{O}(l).$$

Требуемый объем памяти равен количеству элементов, хранящихся в стеке: $M=\mathrm{O}(\log n)$. Объединим полученные результаты в табл. 1.4.

Таблица 1.4

Рекуррентное соотношение, задающее правило вычисления (обозначим $F(n)=2^n$)	Трудоемкость рекусивного алгоритма (размерность $l=\lceil \log_2 n \rceil$, $n=\Theta(2^l)$)	Трудоемкость нерекурсивного алгоритма
$F(n)=F(n-1)+F(n-1)$	$T(n)=2T(n-1)+1$ $T(n)=\Theta(2^n)$ $T(l)=\Omega(2^{2^l})$.	$T(n)=T(n-1)+1$ $T(n)=\Theta(n)$ $T(l)=\Omega(2^l)$.
$F(n)=2\cdot F(n-1)$	$T(n)=T(n-1)+1$ $T(n)=\Theta(n)$ $T(l)=\Omega(2^l)$.	$T(n)=T(n-1)+1$ $T(n)=\Theta(n)$ $T(l)=\Omega(2^l)$.
$F(n)=\begin{cases} F\left(\frac{n}{2}\right)F\left(\frac{n}{2}\right), \text{ если } n-\text{четно}, \\ 2F\left(\frac{n-1}{2}\right)F\left(\frac{n-1}{2}\right), \text{если } n-\text{нечетно} \end{cases}$	$T(n)=2T\left(\frac{n}{2}\right)+2$ $T(n)=\Theta(n)$ $T(l)=\Omega(2^l)$.	$T(n)=T\left(\frac{n}{2}\right)+2$ $T(n)=\Theta(\log n)$ $T(l)=\mathrm{O}(l)$.
$F(n)=\begin{cases} \left(F\left(\frac{n}{2}\right)\right)^2, \text{если} \quad n-\text{четно}, \\ 2\left(F\left(\frac{n-1}{2}\right)\right)^2, \text{если} \quad n-\text{нечетно} \end{cases}$	$T(n)=T\left(\frac{n}{2}\right)+2$ $T(n)=\Theta(\log n)$ $T(l)=\mathrm{O}(l)$.	$T(n)=T\left(\frac{n}{2}\right)+2$ $T(n)=\Theta(\log n)$ $T(l)=\mathrm{O}(l)$.

Анализируя данные из табл. 1.4, приходим к следующему утверждению.

Утверждение 1.1. Не следует использовать рекурсивный алгоритм для решения поставленной задачи, если в рекурсивной программе реализуется максимум один вызов рекурсивной функции. В этом случае рекурсивная реализация не лучше по количеству арифметических операций, чем нерекурсивная, которую можно получить с использованием цикла while.

Пример 1.17. Рассмотрим задачу построения последовательности чисел Фибоначчи.

$$\begin{cases} F(n)=F(n-1)+F(n-2), \quad n>2, \\ F(1)=F(2)=1, \end{cases}$$

где $F(i)$ – i-е число последовательности чисел Фибоначчи. Требуется оценить эффективность рекурсивного и табличного (нерекурсивного) алгоритмов решения поставленной задачи.

Решение. Рассмотрим рекурсивный алгоритм построения чисел Фибоначчи:

```
function fib(k);
begin
  if (k = 1) or (k = 2) then fib : = 1
                else fib : = fib(k – 1) + fib(k – 2);
end;
```

Пусть $T(n)$ – количество арифметических операций для вычисления n-го числа данной последовательности приведенным алгоритмом. Тогда рекуррентное соотношение для количества арифметических операций:

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

Решим это уравнение, выполнив предварительно замену переменной

$$b(n) = T(n) + 1.$$

Тогда

$$b(n) = b(n-1) + b(n-2)$$

и $b(n) = \Omega\left(2^{(n-1)/2}\right)$ (в силу свойств чисел Фибоначчи). Следовательно, $T(n) = \Omega\left(2^{n/2}\right)$. Данный алгоритм является экспоненциальным, так как его трудоемкость

$$l = \lceil \log_2 n \rceil,$$

$$n = \Theta\left(2^l\right),$$

$$T(l) = \Omega\left(2^{2^l}\right).$$

Память $M = \mathrm{O}(n)$. Заметим, что в рекурсивном алгоритме два вызова рекурсивной функции.

Рассмотрим нерекурсивный (табличный алгоритм) построения чисел Фибоначчи:

```
begin
a : = 1;
b : = 1;
 for i : = 3 to n do
```

```
    begin
      c : = a + b;
      a : = b;
      b : = c;
    end;
end.
```

Рекуррентное соотношение для количества арифметических операций равно

$$T(n) = T(n-1) + 3.$$

Решим это уравнение. Количество арифметических операций $T(n) = \mathrm{O}(n)$ и память $M = \mathrm{O}(1)$. Данный алгоритм является экспоненциальным, так как его трудоемкость

$$l = \lceil \log_2 n \rceil,$$
$$n = \Theta\left(2^l\right),$$
$$T(l) = \Omega\left(2^l\right).$$

Сравнивая нерекурсивный и рекурсивный алгоритмы построения последовательности чисел Фибоначчи, приходим к выводу, что нерекурсивный алгоритм предпочтительнее. Однако нерекурсивный алгоритм предполагает, что решаемые подзадачи помещаются в память. В этой ситуации оптимальные решения подзадач можно хранить в таблице, а возвратное соотношение вычислит решение новой подзадачи как значение в нужной клетке таблицы. Если же подзадачи, необходимые в соответствующем соотношении для счета, не помещаются в память, то нет другого выхода, как использовать рекурсивный алгоритм. В этом случае, возможно, придется решать одну и ту же подзадачу неоднократно.

Учитывая все вышесказанное, предлагаем следующие правила, которых советуем придерживаться при выборе способа (рекурсивного или нерекурсивного) программной реализации рекуррентного соотношения.

1. Записываем правило решения некоторой задачи в виде рекуррентного соотношения.
2. Объединяем одинаковые слагаемые в рекуррентном соотношении.
3. Если рекурсивный алгоритм содержит один вызов рекурсивной функции, то в силу утверждения 1.1 рекурсивный алгоритм не будет эффективнее.

4. Если рекурсивный алгоритм содержит более одного вызова рекурсивной функции, то поступаем следующим образом:

- если возникающие подзадачи на каждом шаге независимы, то каждая из них будет решаться только один раз, и поэтому рекурсивная реализация оправдана, так как не приводит к повторному решению одной и той же подзадачи;
- если возникающие подзадачи зависимы, то все зависит от того, помещаются ли все их решения в память:

✓ если помещаются, то нерекурсивный алгоритм позволяет решить каждую подзадачу только один раз, запоминая ее решение в таблице; при повторном возникновении некоторой подзадачи она уже не решается заново, а ее решение берется из таблицы; в этой ситуации рекурсивный алгоритм приводит к многократному решению некоторых подзадач;

✓ если же решения зависимых подзадач не помещаются в память, то единственный выход – рекурсивный алгоритм, конечно при условии, что он позволяет экономить память и сам может быть реализован с учетом доступной памяти (в этом случае важную роль играет глубина рекурсии).

Следуя правилам, для примера 1.16 (возведение числа 2 в степень n) предпочтителен нерекурсивный алгоритм решения поставленной задачи. Для примера 1.17 (вычисление чисел Фибоначчи) в рекуррентном соотношении, задающем правило вычисления, есть два вызова рекурсивной функции и возникающие подзадачи зависимы. Если значение n таково, что решения всех подзадач поместятся в память, то предпочтителен нерекурсивный алгоритм решения, в противном случае – рекурсивный алгоритм (либо рекурсивный алгоритм с меморизацией).

ЛИТЕРАТУРА

Ахо, А. В. Структуры данных и алгоритмы: Учеб. пособие //А. В. Ахо, Д. Э. Хопкрофт, Д. Д. Ульман : пер. с англ. М.: Вильямс, 2000. 384 с.

Волчкова, Г. П. Сборник задач по теории алгоритмов для студентов физико-математических спец. БГУ // Г. П. Волчкова, В. М. Котов, Е. П. Соболевская – Минск: БГУ, 2005. 59 с.

Волчкова, Г. П. Сборник задач по теории алгоритмов. Организация перебора вариантов и приближенные алгоритмы: для студентов спец. 1-31 03 04 «Информатика» // Г. П. Волчкова, В. М. Котов, Е. П. Соболевская – Минск: БГУ, 2008. 59 с.

Кормен, Т. Алгоритмы: построение и анализ// Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн – Москва : Вильямс, 2005. 1296 с.

Котов, В. М. Структуры данных и алгоритмы: теория и практика: Учеб. пособие // В. М. Котов, Е. П. Соболевская – Минск: БГУ, 2004. 255 с.

Котов, В. М. Разработка и анализ алгоритом: теория и практика: пособие для студентов мат. и физ. специальностей // В. М. Котов, Е. П. Соболевская – Минск: БГУ, 2009. 251 с.