Оленчикова Т.Ю., Сартасова М.Ю.

Методические указании к самостоятельной работе по теме «Вычислительная сложность алгоритмов»

Оглавление

1 Оценки сложности алгоритмов

Даже если теоретически доказано существование алгоритма для решения задачи, то его практическая реализация может натолкнуться на проблемы нехватки ресурсов компьютера для получения решения за приемлемое время. Оценки сложности алгоритмов введены для того, чтобы можно было сравнивать эффективность решения одной и той же задачи разными алгоритмами, а также для оценки затрат на реализацию данного алгоритма.

Сложность алгоритмов оценивается двумя параметрами:

1) T(n) – *временная сложность* решения задачи с размерностью исходных данных длины n; позволяет оценить время решения задачи;
2) M(n) – *пространственная (емкостная) сложность*, это количество дополнительной памяти (помимо памяти для исходных данных), необходимой для решения задачи размерности n; оценивает затраты памяти.

Принято в качестве оценок использовать асимптотическое поведение функций T(n) и M(n) при достаточно больших n. Параметр n – размерность задачи.

Временная эффективность T(n) измеряется путем подсчета числа основных операций, выполняемых в алгоритме. Пространственная эффективность M(n) оценивается в виде количества дополнительных единиц памяти, требуемых для работы алгоритма.

Оценка временной сложности алгоритма T(n) определяется как *порядок роста* (order of growth) количества основных операций с точностью до постоянного множителя. Для обозначения степени роста функций применяется O-символика.

Основными, или *базовыми операциями*, называются операции, которые вносят наибольший вклад в общее время выполнения алгоритма. Как правило, составить список основных операций алгоритма совсем нетрудно. Обычно в него включают наиболее длительные по времени операции, выполняемые во внутреннем цикле алгоритма. Например, в большинстве алгоритмов сортировки используется метод сравнения двух элементов (ключей) списка, который сортируется. Для подобного типа алгоритмов основной является операция сравнения ключей. основные операции: умножение и сложение. На большинстве компьютеров команда умножения двух целых чисел выполняется намного дольше, чем сложение. Поэтому она является безусловным кандидатом на включение в список основных операций.

Оценивая примерное время работы алгоритма, важно понимать, чем мы пренебрегаем, а за чем следим (чтобы не пропустить главное и не копаться в несущественных деталях). Распространённый подход – оценивать время работы алгоритма как число элементарных шагов алгоритма (как функцию от размера входа n) с точностью до ограниченных множителей. При этом ошибка в два раза или даже в тысячу раз считается допустимой, в то время как разница между n и n^2 считается существенной.

При таком подходе нет надобности разбираться в длительности элементарных операций, хотя в реальных процессорах разные команды разлагаются в последовательность микрокоманд разной длины и занимают разное время. Важно только, что время любой операции ограничено некоторой константой. Другое следствие того же подхода: если, скажем, алгоритм выполняет $5n^3 + 4n + 3$ элементарных операций на входе размера n, мы можем отбросить слагаемые $4n$ и 3 (которые при больших n существенно меньше, чем $5n^3$). Более того, мы можем отбросить и множитель 5 в старшем слагаемом (через несколько лет компьютеры станут в пять раз быстрее) и сказать, что время работы алгоритма есть $O(n^3)$.

Объясним смысл этих O-обозначений.

Определение 1. Говорят, что функция $f(n)$ принадлежит множеству $O(g(n))$, что записывается как $f(n) \in O(g(n))$, если существует положительная константа c и натуральное число n_0 такое, что $f(n) \le c \cdot g(n)$ для всех $n \ge n_0$.

Определение 2. Говорят, что функция $f(n)$ принадлежит множеству $\Omega(g(n))$, что записывается как $f(n) \in \Omega(g(n))$, если существует положительная константа c и натуральное число n_0 такое, что $f(n) \ge c \cdot g(n)$ для всех $n \ge n_0$.

Определение 3. Говорят, что функция $f(n)$ принадлежит множеству $\Theta(g(n))$, что записывается как $f(n) \in \Theta(g(n))$, если существуют положительные константы c_1 и c_2, а также натуральное число n_0 такое, что $c_2 \cdot g(n) \le f(n) \le c_1 \cdot g(n)$ для всех $n \ge n_0$.

Запись $f \in O(g)$ можно читать как «$f \le g$ с точностью до константы». При этом, скажем, $10n \in O(n)$.

O-обозначения позволяют сравнивать поведение функций при больших n .

Например, пусть один алгоритм требует $f_1(n) = n^2$ шагов, а второй – $f_2(n) = 2n + 20$. Какой из них лучше (рис. 1)? Это, конечно, зависит от значения n: первый лучше второго при $n < 5$, но сильно хуже при большом n .

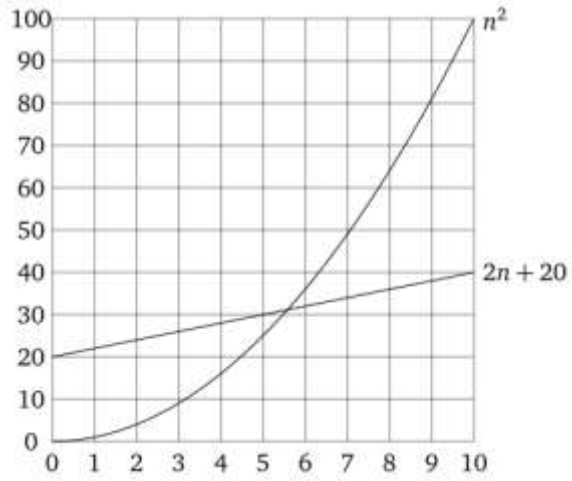


Рис. 1. Сравнение квадратичной и линейной функций

В О-обозначениях для данного примера можно сказать, что $f_2 \in O(f_1)$, но $f_1 \notin O(f_2)$. В самом деле, отношение

$$\frac{2n+20}{n^2} = \frac{2}{n} + \frac{20}{n^2} \le 22$$

ограничено, а

$$\frac{n^2}{2n+20} \ge \frac{n^2}{22n} = \frac{n}{22}$$

– нет.

Обозначение $O(\cdot)$ можно считать аналогом $\le$. Аналоги для $\ge$ и $=$ такие:

$f \in \Omega(g)$ (f растет не медленнее g, с точностью до константы) означает $g \in O(f)$.
$f \in \Theta(g)$ (f и g имеют одинаковый порядок роста) означает что $f \in O(g)$ и $g \in O(f)$.
$f \in O(g)$ (f растет не быстрее g, с точностью до константы); $g \in \Omega(f)$.

На рис. 2 приведены графики, иллюстрирующие эти обозначения.

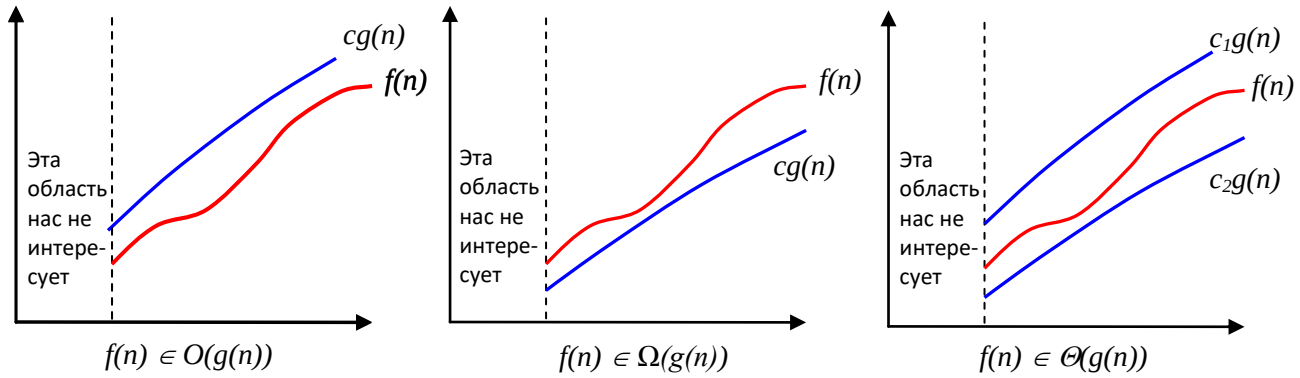


Рис.2. Иллюстрация О-обозначений

Можно (и нужно) упрощать оценки, пренебрегая несущественными (при больших n) слагаемыми. Например, можно заменить $3n^2+4n+5$ на n^2. Приведем несколько правил такого рода упрощения:

1) постоянные множители можно опускать; например $14n^2$ заменяем на n^2:
2) n^a растет быстрее n^b для $a>b$; например, в присутствии слагаемого n^2 можно пренебречь слагаемым n;
3) любая экспонента растет быстрее любого полинома; например, 3^n растет быстрее n^5;
4) любой полином растет быстрее любого логарифма; например $\sqrt{n}$ растет быстрее $(\log n)^3$, n^2 растет быстрее $n \log n$.

Вот несколько примеров – все приведенные ниже утверждения справедливы:
$n \in O(n^2)$; $100n+2 \in O(n^2)$; $n(n-1)/2 \in O(n^2)$; $n^3 \notin O(n^2)$; $0.0001n^3 \notin O(n^2)$, $n^3 \in \Omega(n^2)$; $n(n-1)/2 \in \Omega(n^2)$; $100n+5 \in \Omega(n^2)$; $an^2+bn+c \in \Theta(n^2)$ при $a>0$, $n^2/6+\sin(n) \in \Theta(n^2)$; $n^2+\log n \in \Theta(n^2)$.

Чтобы определить, какая из двух функций $f(n)$ или $g(n)$ растет быстрее удобно пользоваться пределами при $n \to \infty$. Если:

a) $\lim_{n\to\infty}\frac{f(n)}{g(n)}$= const>0, то *f(n)* и *g(n)* имеют одинаковый порядок роста, обозначение: $f \in \Theta(g)$; допустимо обозначение : $f \in O(g)$ и $f \in \Omega(g)$;

b) $\lim_{n\to\infty}\frac{f(n)}{g(n)}$=0, то *f(n)* растет медленнее *g(n)*, и *f(n)* можно опускать в слагаемых; обозначение: $f \in O(g)$; $g \notin O(f)$, ; $g \in \Omega(f)$,

c) $\lim_{n\to\infty}\frac{f(n)}{g(n)} = \infty$, то *f(n)* растет быстрее *g(n)*, и *g(n)* можно опускать в слагаемых; обозначение: $f \in \Omega(g)$; $g \in O(f)$;

Пример 1. Сравним порядки роста функций $log_2 n$ и $\sqrt{n}$.

Решение. Найдем предел:

$$\lim_{n\to\infty}\frac{log_2 n}{\sqrt{n}} = \lim_{n\to\infty}\frac{(log_2 n)'}{(\sqrt{n})'} = \lim_{n\to\infty}\frac{(log_2 e)/n}{1/(2\sqrt{n})} = 2\,log_2 e\lim_{n\to\infty}\frac{\sqrt{n}}{n} = 0$$

Ответ. $log_2 n \in O(\sqrt{n})$, т.е. функция $log_2 n$ растет медленнее $\sqrt{n}$; справедливо также $\sqrt{n} \in \theta(log_2 n)$.

Пример 2. Сравним порядки роста функций $n!$ и 2^n.

Решение. Воспользуемся формулой Стирлинга:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \text{ для достаточно больших } n.$$

Тогда

$$\lim_{n\to\infty}\frac{n!}{2^n} = \lim_{n\to\infty}\frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{2^n} = \lim_{n\to\infty}\sqrt{2\pi n}\frac{n^n}{2^n e^n} = \lim_{n\to\infty}\sqrt{2\pi n}\left(\frac{n}{2e}\right)^n = \infty.$$

Ответ. Функция $n!$ растет быстрее, чем 2^n, т.е. $2^n \in O(n!)$

Задание 1. ***Порядок роста функций***

Используя определения множеств O, Ω и Θ, определите, истинны или ложны перечисленные ниже утверждения.

1) $\frac{n(n+1)}{2} \in O(n^3)$;
2) $\frac{n(n+1)}{2} \in O(n^2)$;
3) $\frac{n(n+1)}{2} \in \Theta(n^3)$;
4) $\frac{n(n+1)}{2} \in \Omega(n)$.

Для каждой из приведенных ниже функций укажите класс O($g(n)$), к которому относится функция. (При ответе используйте максимально простую функцию $g(n)$).

5) $(n^2+1)^{10}$;
6) $\sqrt{10n^2+7n+3}$;
7) $2n\lg(n+2)^2 + (n+2)^2\lg(\frac{n}{2})$;

8) $2^{n+1}+3^{n-1}$;

Расположите перечисленные ниже функции в соответствии с их порядком роста (от самого меньшего к самому большему).

9) $(n\text{-}2)!$, $5\ lg(n+100)^{10}$, 2^{2n}, $0.001n^4+3n^3+1$, $ln^2 n$, $\sqrt[3]{n}$, 3^n
10) Докажите, что любой многочлен вида $p(n)=a_k n^k+a_{k-1}n^{k-1}+\ldots+a_0$, где $a_k>0$, принадлежит множеству $\Theta(n^k)$
11) Докажите, что порядок роста показательной функции a^n зависит от значения ее неотрицательного основания a.

Используя определения рассматриваемых множеств, докажите или опровергните, приведя соответствующий контрпример, следующие утверждения:

12) Если $f(n) \in O(g(n))$, то $g(n) \in \Omega(f(n))$;
13) $\Omega(a\ g(n))= \Omega(g(n))$, где $a>0$;
14) $\Theta(g(n))=O(g(n))\cap\Omega(g(n))$.

Для каждой из приведенных ниже пар функций определите, соответствует ли первая функция порядку роста второй функции (с точностью до постоянного множителя), меньше его или больше. Ответ дайте, используя O-обозначения.

15) $f(n)=n\ (n+1)$, $g(n)= 2000\ n^2$;
16) $f(n)=100\ n^2$, $g(n)= 0.01\ n^3$;
17) $f(n)=log_2\ n$, $g(n)= ln\ n$;
18) $f(n)=(log_2\ n)^2$, $g(n)= log_2\ n^2$;
19) $f(n)=2^{n-1}$, $g(n)= 2^n$;
20) $f(n)=(n-1)!$, $g(n)= n!$;

Для решения системы из n линейных уравнений, где n — произвольное целое число, используется классический алгоритм последовательного исключения Гаусса, в котором выполняется $n^3/3$ умножений, являющихся основной операцией алгоритма:

21) определите, во сколько раз дольше будет работать алгоритм Гаусса для решения системы из 1000 уравнений по сравнению со временем решения системы из 500 уравнений.
22) предположим, вы собираетесь приобрести компьютер, быстродействие которого превосходит в 1000 раз быстродействие того компьютера, на котором вы сейчас работаете. Определите, на сколько порядков большую систему уравнений сможет решить за одно и то же время новый компьютер по сравнению со старым.

2 Основные классы эффективности

Временную эффективность большого количества алгоритмов можно отнести всего к нескольким классам. Эти классы, их названия, а также некоторые пояснения, приведены в табл. 1 в соответствии с возрастанием их порядка роста. В столбце (5) приведена оценка времени работы алгоритма для задачи размерности $n=10^6$

при скорости вычислений 10^6 операций/сек. В столбце (6) – максимальная размерность задачи n на том же компьютере при ограничении времени работы алгоритма 30 сек.

Таблица 1

Основные асимптотические классы эффективности

Класс	Название	Типы задач	Число операций при $n=10^6$	Время работы при скорости 10^6 опер/сек	Максим. n если время работы 30 сек
(1)	(2)	(3)	(4)	(5)	(6)
1	Константа O(1)	В этот класс попадает очень небольшое количество алгоритмов. Обычно при бесконечном увеличении размера входных данных время выполнения алгоритма также стремится к бесконечности	1	1 мкс	Не ограничено
log n	Логарифмическая O(log n)	Обычно такая зависимость появляется в результате сокращения размера задачи на постоянное значение на каждом шаге итерации. В логарифмическом алгоритме невозможна работа со всеми входными данными, в этом случае время выполнения любого алгоритма будет находиться по меньшей мере в линейной зависимости от размера входных данных n. Например, бинарный поиск в сортированном массиве.	$\log_2(10^6)$	20 мкс	$2^{30\,000\,000}$
n	Линейная O(n)	Алгоритмы последовательного перебора всех исходных данных	10^6	1 сек	$3\cdot10^7$
n log n	O(n log n)	Задача разбивается на подзадачи меньшей размерности. Затем решение получается путем объединения решений подзадач. Например, сортировка слиянием, быстрая сортировка, быстрое преобразование Фурье	$20\cdot10^6$	20 сек	$1{,}5\cdot10^6$
n^2	Квадратичная O(n^2)	Алгоритмы, содержащие два встроенных. Например, сортировка «пузырьком» и целый ряд операций, выполняемых над матрицами размером n x n	10^{12}	11,6 дней	$5{,}5\cdot10^3$
n^3	Кубическая O(n^3)	Алгоритмы, содержащие три встроенных цикла. Например, перемножение матриц размером n x n	10^{18}	32 000 лет	$3{,}1\cdot10^2$
a^n	Экспоненциальная O(a^n)	Алгоритмы, выполняющие обработку всех подмножеств некоторого множества, состоящего из n элементов. Например, симплекс- метод для решения задач линейного программирования, ал-	10^{301030}, $a=2$	Во много раз больше времени су-	25

		горитмы ветвей и границ		щество-вания Вселен-ной	
$n!$	Фактори-альные $O(n!)$	Алгоритмы, выполняющие обработ-ку всех перестановок некоторого множества, состоящего из n элемен-тов	$10^{5565709}$		11

В общем случае, алгоритмы с временной сложностью $O(a^n)$ называются *полиномиальными.*

Алгоритмы, временная сложность которых больше полиномиальной, называются *экспоненциальными*. К этому же классу обычно относят и факториальные алгоритмы. Большинство экспоненциальных алгоритмов - это просто варианты полного перебора, в то время как полиномиальные алгоритмы обычно можно построить лишь тогда, когда удается более глубоко проникнуть в суть решаемой задачи.

Деление на полиномиальные и экспоненциальные алгоритмы является отправным пунктом в определении труднорешаемых задач и теории NP-полных задач. Имеется широко распространенное соглашение, согласно которому задача не считается "хорошо решаемой" до тех пор, пока для нее не получен полиномиальный алгоритм.

3 Эффективность алгоритма в разных случаях

Существует большое количество алгоритмов, время выполнения которых зависит не только от размера входных данных, но также и от особенностей конкретных входных данных.

Например, рассмотрим задачу последовательного поиска. Она решается с помощью довольно простого алгоритма, который выполняет поиск заданного элемента (ключа поиска К) в списке, состоящем из n элементов. Работа алгоритма завершается, либо когда заданный ключ найден, либо когда весь список исчерпан. Очевидно, что время работы этого алгоритма может отличаться в очень широких пределах для одного и того же списка размера n. В наихудшем случае, т.е. когда в списке нет искомого элемента либо когда искомый элемент расположен в списке последним, в алгоритме будет выполнено наибольшее количество операций сравнения ключа со всеми n элементами списка, тогда $T(n) \in O(n)$. В наилучшем случае, совпадение произойдет на первом элементе и $T(n) \in O(1)$. Для такого рода алгоритмов вводят оценки эффективности:

1) наихудшем случае – O_{max};
2) в наилучшем случае – O_{min};
3) для типовых или случайно заданных входных данных (в среднем) – O_{avg}.

Анализ эффективности алгоритма для наихудшего случая позволяет получить очень важную информацию о быстродействии алгоритма в целом, поскольку в данном случае речь идет о максимально возможном времени его выполнения.

Анализ эффективности алгоритма для наилучшего случая не так важен, как для наихудшего случая, хотя его нельзя назвать совсем уж бесполезным, учитывая тот факт, что для некоторых алгоритмов их высокое быстродействие для наилучшего случая сохраняется и для случаев близких к наилучшему.

Оценка в среднем дает более достоверную картину, но представляет гораздо более трудную задачу.

Пример 3. Рассмотрим задачу последовательного поиска заданного элемента в массиве целых четырехбайтовых чисел. Алгоритм приведен ниже. Работа алгоритма заканчивается, либо, когда заданный ключ найден, либо, когда просмотрены все элементы массива.

```
int index=SequentialSearch(int *A, int N, int K) {
// входные данные: массив целых чисел A[0..N-1] и ключ поиска K
// выходные данные: индекс первого элемента массива A, который равен K,
// либо -1, если заданный элемент не найден
        int i=0;
        while ((i<N) && (A[i]!=K)) i++;
        if (i<N) return i;
        else return -1;
}
```

Сделаем два стандартных предположения:

а) вероятность успешного поиска равна p, где $0 \le p \le 1$;

б) вероятность первого совпадения ключа с i-м элементом списка одинакова для любого i.

Теперь можно найти среднее количество операций сравнения с ключом $T_{avg}(N)$. Если совпадение сключом найдено, то вероятность того, что это произошло именно на i-ом элементе списка, равна p/N для любого i; при этом количество операций сравнения равно i. Если совпадение с ключом не найдено, то кличество операций сравнения равно N, а вероятность этого события равна $(1-p)$. Поэтому

$$T_{avg}(N)=\left[1 \cdot \frac{p}{N} + 2 \cdot \frac{p}{N} + \cdots + i \cdot \frac{p}{N} + \cdots + N \cdot \frac{p}{N}\right] + N(1-p) =$$

$$= \frac{p}{N}[1 + 2 + \cdots + N] + N(1-p) = \frac{p(N+1)}{2} + N(1-p).$$

Проанализируем полученное выражение. Если $p=1$ (искомый элемент точно присутствует в списке), среднее количество операций сравнения будет равно $(N+1)/2$. Если $p=0$ (искомого элемента в списке нет), среднее количество операций сравнения будет равно N. В любом случае $T_{avg}(N) \in \Theta(N)$.

4 Математический анализ не рекурсивных алгоритмов

1. Выберите параметр (или параметры), по которому будет оцениваться размер входных данных алгоритма.

2. Определите основную операцию алгоритма. (Как правило, она находится в наиболее глубоко вложенном внутреннем цикле алгоритма.)

3. Проверьте, зависит ли число выполняемых основных операций только от размера входных данных. Если оно зависит и от других факторов, рассмотрите при необходимости, как меняется эффективность алгоритма для наихудшего, среднего и наилучшего случаев.

4. Запишите сумму, выражающую количество выполняемых основных операций алгоритма.

5. Используя стандартные формулы и правила суммирования, упростите полученную формулу для количества основных операций алгоритма. Если это невозможно, определите хотя бы их порядок роста.

Пример 4. Рассмотрим задачу поиска наибольшего элемента в массиве из *n* целых чисел. Приведем алгоритм решения задачи:

```
double MaxElem(double* A, int N)  {
      //входные данные: массив вещественных чисел  A[0..N-1]
      //выходные данные: значение наибольшего элемента массива A
      double MaxVal=A[0];
      for (int i=1; i<N; i++) {
         if(A[i]>MaxVal) MaxVal=A[i];
      }
      return MaxVal;
}
```

В этом алгоритме размер входных данных нужно оценивать по количеству элементов в массиве, т.е. числом *N*. Операции, выполняемые чаще всего, находятся во внутреннем цикле for алгоритма. Таких операций две: сравнение (A[i]>MaxVal) и присваивание (MaxVal=A[i]). Какую из них считать базовой? Поскольку сравнение выполняется на каждом шаге цикла, а присваивание – нет, логично считать, что основной операцией алгоритма является операция сравнения. Так как для любого массива размером *N* количество операций сравнения в рассматриваемом алгоритме постоянно, не имеет смысла отдельно рассматривать эффективность алгоритма для худшего, типичного и лучшего случаев.

Определим $T(N)$ количество базовых операций, выполняемых в алгоритме. За один цикл в алгоритме выполняется одна операция сравнения. Число повторений цикла – N-1 (i изменяется от 1 до N-1), поэтому получаем:

$$T(N) = \sum_{i=1}^{N-1} 1 = N\text{-}1 \in \Theta(n).$$

Пример 5. Рассмотрим задачу проверки единственности элементов. Другими словами, нужно убедиться, что все элементы массива различны.

Алгоритм решения задачи:

```
bool  UniqueElem(double* A, int N)  {
      //входные данные: массив вещественных чисел  A[0..N-1]
      //выходные данные: значение true, если все элементы массива A
      //различны, и false в противном случае
      for (int i=0; i<N-1; i++) {
         for(int j=i+1; j<N; j++)  {
                if(A[i]=A[j]) return false;
         }
      }
      return true;
```

}

Как в предыдущем примере, размер входных данных нужно оценивать по количеству элементов в массиве, т.е. числом *N*. Поскольку в наиболее глубоко вложенном внутреннем цикле алгоритма выполнятся только одна операция сравнения двух элементов, ее и будем считать основной операцией этого алгоритма. Обратите внимание, что количество операций сравнения будет зависеть не только от общего числа *N* элементов в массиве, но и от того, есть ли в массиве одинаковые элементы, и если есть, то на каких позициях они расположены. Ограничимся рассмотрением наихудшего случая.

а) Наихудший случай входных данных (т.е. когда цикл выполняется от начала до конца, а не завершается досрочно, может возникнуть при обработке массивов двух типов: а) в которых нет одинаковых элементов; б) в которых два одинаковых элемента находятся рядом и расположены в самом конце массива. В обоих случаях при каждом повторе внутреннего цикла в нашем алгоритме выполняется одна операция сравнения. При этом переменная внутреннего цикла *j* последовательно принимает значения от *i* до *N*-2. Внешний цикл (по *i*) выполняется *N*-1 раз. Получаем:

$$T_{max}(N) = \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} 1 = \sum_{i=0}^{N-2} \big((n-1) - (i+1) + 1\big) =$$

$$= \sum_{i=0}^{N-2} (N-1) - \sum_{i=0}^{N-2} i \;=\; (N-1) \sum_{i=0}^{N-2} 1 - \frac{(N-2)(N-1)}{2} = \frac{(N-1)N}{2} \approx \frac{1}{2} N^2.$$

То есть

$$T_{max}(N) \in \Theta(n^2).$$

Пример 6. Данный алгоритм позволяет определить количество разрядов, которое будет иметь положительное десятичное число в двоичном представлении.

```
int  Binary (int N) {
// Входные данные: целое положительное число N
// Выходные данные: колич-во разрядов в двоичном представлении числа N
      int count = 1;
      while (N > 1) {
         count = count + 1;
         N=N div 2;
      }
      return count;
}
```

Заметим, что в этом алгоритме операция сравнения ($N > 1$), которая выполняется чаще всего, не находится в его внутреннем цикле while. Однако она определяет, будет ли выполняться все тело цикла. Поскольку количество выполняемых операций сравнения всегда на единицу больше, чем общее число повторов выполнения цикла, выбор основной операции алгоритма не имеет большого значения. Будем считать число выполнений цикла while. Хуже то, что while – не счетчик. Для определения числа повторений цикла while заметим, что на каждом шаге цикла значение переменной N уменьшается примерно вдвое. Точная формула:

$$T(N) = \lfloor \log_2 N \rfloor + 1 \in \Theta(log_2 N),$$

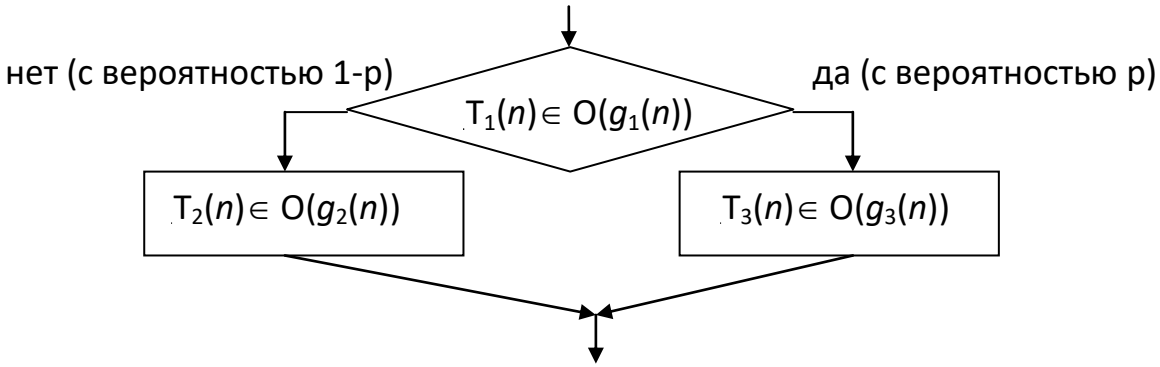
здесь операция $\lfloor \cdot \rfloor$ – обозначает округление до меньшего целого.

Обычно алгоритм состоит из **последовательно выполняемых блоков**. Чтобы оценить временную сложность такого алгоритма, суммируют сложности выполнения каждого блока, результат упрощают, используя свойства О-обозначений, а именно, общая эффективность алгоритмов, состоящих из нескольких блоков зависит от наименее эффективного блока, который имеет наибольший порядок роста:

$T_1(n) \in O(g_1(n))$
$T_2(n) \in O(g_2(n))$
…
$T_M(n) \in O(g_M(n))$

$$T_1(n)+\ldots+T_M(n) \in O(\max\{ g_1(n), \ldots , g_M(n)\})$$

Рассмотрим алгоритм **с условным переходом**:



Логическое выражение вычисляется всегда. Положим, что ветвь «да» выполняется с вероятностью p, а ветвь «нет» - соответственно с вероятностью 1-p. Тогда,

$$T(n)= T_1(n)+p\cdot T_3(n)+(1-p)\cdot T_2(n)$$

Пренебрегаем постоянными коэффициентами p и 1-p и снова получаем:

$$T(n) \in O(\max\{ g_1(n), g_2(n), g_3(n)\})$$

Если алгоритм содержит **циклы**, то следует различать циклы по счетчику и циклы по условию. Допустим, что цикл выполняется m раз

Для цикла по счетчику, вычислительная сложность счетчика $T_1(m) \in O(m)$. Вычислительная сложность тела цикла: $T_2(n) \in O(g_2(n))$. Тогда вычислительная сложность всего цикла равна:

$$T(n) = T_1(m)+ m\cdot T_2(n) \in O(\max\{m, m\cdot g_2(n)\} = O(m\cdot g_2(n))$$

Аналогично считаем для вложенных циклов.

Для циклов с выходом по условию полезно получить оценки в наилучшем, наихудшем случаях и в среднем (см. пример 5).

При необходимости более точного анализа отдельно определяют оценки числа операций а) сложения и вычитания, б) умножения, в)деления. Примем длительность операций сложения и вычитания за 1 $T_+(n) \in O(1)$, тогда, соответственно, длительность операции умножения $T_*(n) \in O(n)$ и деления $T_/(n) \in O(n^2)$, где n – число двоичных разрядов чисел, участвующих в операциях.

Задание 2. ***Анализ нерекурсивных алгоритмов***

1) Эмпирическая дисперсия выборки, состоящей из n элементов x_1, x_2, …, x_N вычисляется по формуле:

а)
$$s^2 = \frac{\sum_{i=1}^{N}(x_i - \bar{x})^2}{N-1}, \text{где } \bar{x} = \frac{\sum_{i=1}^{N} x_i}{N}$$

или

б)
$$s^2 = \frac{\sum_{i=1}^{N} x_i^2 - \left(\sum_{i=1}^{N} x_i\right)^2 / N}{N-1}$$

Найдите и сравните количество операций деления, умножения, сложения и вычитания (последние две операции обычно объединяются и считаются как одна), которые необходимо выполнить для вычисления эмпирической дисперсии по каждой из приведенных выше формул. Каким способом считать дисперсию быстрее?

Задачи 2)-4). Для приведенных ниже алгоритмов ответьте на вопросы:

a) Что вычисляется с помощью этого алгоритма?
b) Назовите основную операцию алгоритма.
c) К какому классу эффективности относится алгоритм?
d) Усовершенствуйте этот алгоритм или предложите другой алгоритм и оцените его класс эффективности. Если вам не удалось, попытайтесь доказать, что это невозможно.

2)
```
int Mystery(int N) {
    // входные данные: целое положительное число N
    int S=0;
    for (int i=1; i<=N; i++) {
        S=S+i*i;
    }
    return S;
}
```

3)
```
double Secret(double* A, int N) {
    //входные данные: массив N вещественных чисел A[0..N-1]
    double minval=A[0];
    double maxval=A[0];
    for (int i=1; n<N; i++) {
        if (A[i]<minval) minval=A[i];
        if (A[i]> maxval) maxval=A[i];
    }
    return maxval-minval;
}
```

4)
```
bool Enigma(double *A[], int N) {
//входные данные: матрица действительных чисел A[0..N-1,0..N-1]
    for (int i=0; i<N-1; i++) {
    for (int j=i+i; j<N; j++) {
```

```
            if (A[i,j]!=A[j,i]) return false;
        }
        }
        return true;
    }
```

5) Улучшите реализацию приведенного ниже алгоритма умножения матриц за счет уменьшения количества выполняемых операций сложения. Как это отразится на эффективности работы алгоритма? (Подсказка: посмотрите алгоритм Штрассена)

```
void  MatrixMultiplication(double *A[], double *B[], double *C[], int N) {
    //умножение двух квадратных матриц размером N×N
    //входные данные: две квадратные матрицы размером N×N A и B
    //выходные данные: матрица C=AB
    for (int i=0; i<N; i++) {
            for (int j=0; j<N; j++) {
                    C[i,j]=0.;
                    for (int k=0; k<N; k++)  {
                            C[i,j]=C[i,j]+A[i,k]*B[k,j];
                    }
            }
    }
}
```

6) Для приведенного ниже алгоритма ответьте на вопросы:
    a) Определите класс временной эффективности этого алгоритма.
    b) Какое место в этом коде сразу бросается в глаза из-за своей неэффективности? Как можно переписать этот алгоритм, чтобы повысить скорость работы алгоритма?

```
void  GE(double *A[], int N) {
    //входные данные: матрица A[0…N-1, 0,N] вещественных чисел
    //размером (N×N+1)
    for (int i=0; i<N-1; i++) {
            for (int j=i+1; j<N; j++) {
                    for (int k=i; k<N; k++)  {
                            A[j,k]= A[j,k]-A[i,k]*A[j,i]/A[i,i];
                    }
            }
    }
}
```

5 Математический анализ рекурсивных алгоритмов

Любой алгоритм можно переписать без рекурсии, но рекурсия позволяет существенно упростить код программы. Рассмотрим, как можно оценить эффективность рекурсивного алгоритма.

Пример 7. Возьмем самый простой рекурсивный алгоритм вычисления факториала N!. Имеем рекуррентные уравнения (1):

$$F(N) = N! = F(N-1)*N; \quad F(0) = 0! = 1. \tag{1}$$

Рекурсивная программа вычисления F(N) имеет вид:

```
long Fact(int N) {
        if (N==0) return 1;
        return Fact(N-1)*N;
}
```

Для простоты будем считать, что размер входных данных алгоритма указывает само число N, а не количество битов в его двоичном представлении.

Основной операцией этого алгоритма является умножение, количество выполнений которой мы обозначим через T(N).

Рекуррентное соотношение и начальные условия для количества операций умножения T(N) рассматриваемого нами алгоритма выглядят так:

$$T(N) = T(N-1) + 1, \text{ для } N>0; \quad T(0) = 0. \tag{2}$$

В формулах (2): T(N-1) – число операций для вычисления F(N-1) и добавляется 1 – умножение F(N-1) на N. T(0)=0, так как при N=0 операция умножения не выполняется.

Уравнения (2) называются *рекуррентными.*

Решать рекуррентные уравнения (2) будем *методом обратных подстановок*.

T(N) = T(N-1) + 1 =	подставляем T(N-1) = T(N-2) + 1
=(T(N-2) +1) +1= T(N-2)+2=	подставляем T(N-2) = T(N-3) + 1
=(T(N-3) +1) +2= T(N-3)+3=	…
= …=	
=T(N-i)+ i	выявили закономерность, ее хорошо бы обосновать по индукции,

Применяем начальные условия, при i=N:

$$T(N) = T(N-i)+i=T(0)+N=N \in \Theta(N)$$

Пример 8. Рассмотрим рекурсивный вариант алгоритма определения количества разрядов в двоичном представлении числа N (пример 5)

```
int BinRec (int N) {
// Входные данные: целое положительное число N
// Выходные данные: колич-во разрядов в двоичном представлении числа N
    if (N==1) return 1;
```

```
    return BinRec(N div 2)+1;
}
```

Давайте построим рекуррентное уравнение и зададим начальные условия для количества операций сложения T(N), выполняемых в этом алгоритме.

Если N>1, то количество операций сложения при вычислении функции BinRec(N div 2), равно $T(\lfloor N/2 \rfloor)$ плюс одна операция сложения. При N=1 операции сложения нет, и T(1)=0. Таким образом, рекуррентное уравнение имеет вид:

$$T(N) = T(\lfloor N/2 \rfloor) + 1, \text{ для } N>1; \quad T(1) = 0. \tag{3}$$

Наличие в параметре функции BinRec выражения $\lfloor N/2 \rfloor$ приводит к тому, что в методе обратных подстановок сложно использовать значения параметра N, которые не являются степенью 2. Поэтому стандартным подходом при решении такого рекуррентного отношения является поиск решения только для $N = 2^k$ с последующим применением теоремы, называемой правилом сглаживания. В ней утверждается, что сделанную оценку порядка роста функции для $N = 2^k$ можно с очень высокой степенью приближенности считать правильной для всех значений N. Пусть $N = 2^k$, тогда рекуррентные выражения (3) примут вид:

$$T(2^k) = T(2^{k-1}) + 1, \text{ для } k>0; \quad T(2^0) = 0. \tag{4}$$

Теперь можно применить метод обратной подстановки:

$T(2^k) = T(2^{k-1}) + 1 =$	подставляем $T(2^{k-1}) = T(2^{k-2}) + 1$
$=(T(2^{k-2}) +1) +1= T(2^{k-2})+2=$	подставляем $T(2^{k-2}) = T(2^{k-3}) + 1$
$=(T(2^{k-3}) +1) +2= T(2^{k-3})+3=$	…
$= \ldots=$	
$=T(2^{k-i})+ i$	выявили закономерность,

Применяем начальные условия, при i=k:

$$T(N) = T(2^{k-i})+i=T(2^0)+k=k.$$

Окончательно, так как $N=2^k$, то $k=\log_2 N$. Поэтому

$$T(N)=\log_2 N \in \Theta(\log N).$$

Общий план анализа эффективности рекурсивных алгоритмов.

1. Выберите параметр (или параметры), по которому будет оцениваться размер входных данных алгоритма.

2. Определите основную операцию алгоритма.

3. Проверьте, зависит ли число выполняемых основных операций только от размера входных данных. Если оно зависит и от других факторов, рассмотрите при необходимости, как меняется эффективность алгоритма для наихудшего, среднего и наилучшего случаев.

4. Составьте рекуррентное уравнение, выражающее количество выполняемых основных операций алгоритма, и укажите соответствующие начальные условия.

5. Найдите решение рекуррентного уравнения или, если это невозможно, определите хотя бы его порядок роста.

Если в рекурсивном алгоритме выполняется более одного вызова его самого, то для анализа такого алгоритма полезно построить дерево его рекурсивных вызовов. Узлы такого дерева будут соответствовать рекурсивным вызовам.

Следующий пример иллюстрирует с какими проблемами можно столкнуться, используя рекурсию.

Пример 9. Числа Фибоначчи

Рассмотрим знаменитую последовательность целых чисел:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

Ее можно описать с помощью рекуррентных соотношений

$$F(N) = F(N-1) + F(N-2), \text{ для } N>1; \quad F(0)=0,\ F(1)=1. \tag{5}$$

Известна явная формула значения N-го элемента F(N) последовательности чисел Фибоначчи:

$$F(N) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^N - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^N \tag{6}$$

При взгляде на эту формулу сразу видно, что функция F(N) имеет экспоненциальный порядок роста, т.е. $F(N) \in \Theta(\psi^N)$, где $\psi = \left(\frac{1+\sqrt{5}}{2}\right)$.

Проанализируем несколько алгоритмов вычисления чисел Фибоначчи.

Алгоритм1. Прямое использование рекурсии.

```
long  Fib1(int N) {
// вычисление числа Фибоначчи с применением рекурсивного алгорита
// входные данные: целое неотрицательное число N;
// выходные данные: N-ый элемент последовательности чисел Фибоначчи.
        if (N<=1) return N;
        return F(N-1)+F(N-2);
}
```

Основной операцией этого алгоритма является сложение. Обозначим через T(N) количество операций сложения при вычислении функции F(N). Справедливы следующие рекуррентные уравнения:

$$T(N) = T(N-1) + T(N-2) + 1, \quad \text{для } N > 1; \quad T(0) = 0,\ T(1) = 0 \tag{7}$$

Решение этой рекурсии приведено в [1] (стр. 123-124) и дает ответ:

$T(N)=F(N+1) - 1 \in \Theta(\psi^N)$, где $\psi = \left(\frac{1+\sqrt{5}}{2}\right)$ – экспоненциальная сложность

Чтобы понять причину низкой эффективности алгоритма, достаточно взглянуть на его дерево рекурсивных вызовов, построенное по результатам выполнения алгоритма. Пример подобного дерева, построенного для N = 5 приведен на рис. 3.

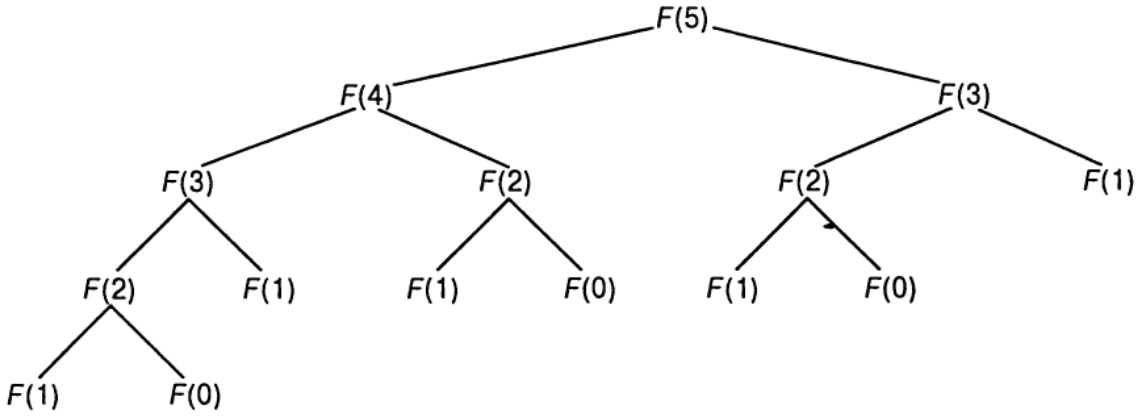


Рис. 3. Дерево рекурсивных вызовов, построенное для алгоритма Fib1 вычисления последовательности чисел Фибоначчи при N = 5

Обратите внимание, что в этом алгоритме функция с одним и тем же значением аргумента вызывается по нескольку раз. Поэтому очевидно, что такой алгоритм будет крайне неэффективен.

Получится намного более эффективный алгоритм, если числа Фибоначчи вычислять последовательно одно за другим в цикле, как в алгоритме Fib2.

Алгоритм2. Итеративный алгоритм.

```
long  Fib2(int N) {
// вычисление числа Фибоначчи с применением итераций
// входные данные: целое неотрицательное число N;
// выходные данные: N-ый элемент последовательности чисел Фибоначчи.
        long F[2];
        F[0] = 0;  F[1] = 1;
        for (int i=2; i<=N; i++) {
                F[i & 1]=F[0] + F[1];
        }
        return F[N & 1];
}
```

Очевидно, что в этом алгоритме выполняется N – 1 операция сложения. Следовательно, алгоритм является линейным, если рассматривать зависимость времени его работы от N.

Третий способ вычисления N-го элемента последовательности чисел Фибоначчи заключается в использовании формулы (6). Очевидно, что эффективность этого алгоритма будет определяться эффективностью алгоритма, используемого для вычисления ψ^N. Если это делается путем простого перемножения ψ на себя N – 1 раз, то алгоритм будет принадлежать множеству $\Theta(N)$.

Наконец, для вычисления N-го элемента последовательности чисел Фибоначчи существует еще один алгоритм со временем работы $T(N) \in \Theta(\log N)$, в котором используются только целые числа. Он основан на равенстве

$$\begin{bmatrix} F(N-1) & F(N) \\ F(N) & F(N+1) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^N, \text{ для } N \geq 1$$

И использует эффективные способы возведения матриц в степень.

Вывод из проведенного анализа: *рекурсивные алгоритмы следует использовать очень осторожно, поскольку часто за их внешней компактностью скрывается крайняя неэффективность.*

Задание 3. ***Анализ рекурсивных алгоритмов***

Задачи 1)-5). Найдите решение для следующих рекуррентных отношений:

1) $x(n) = x(n-1) + 5$, для $n>1$; $x(1)=0$

2) $x(n) = 3x(n-1) + 5$, для $n>1$; $x(1)=4$

3) $x(n) = x(n-1) + n$, для $n>1$; $x(0)=0$

4) $x(n) = x(n/2) + n$, для $n>1$; $x(1)=1$ (найдите решение для $n=2^k$)

5) $x(n)=x(n/3)+1$, для $n>1$; $x(1)=1$ (найдите решение для $n=3^k$)

6) Проанализируйте рекурсивный алгоритм вычисления суммы кубов первых N чисел $S(N)=\sum_{i=1}^{N} i^3$.

```
long S(int N) {
 // входные данные: целое положительное число N
 // выходные данные; сумма кубов первых N целых чисел
        if(N==1) return 1;
        return S(N-1)+N*N*N;
}
```

a) Постройте рекуррентное уравнение для количества выполнений основной операции алгоритма и найдите его решение.
b) Сравните этот алгоритм с простым нерекурсивным алгоритмом.

7) Дан рекурсивный алгоритм

```
long Q(long N) {
        if (N==1) return 1;
        return Q(N-1)+2*N-1;
}
```

a) Постройте рекуррентное уравнение для значения этой функции, найдите его решение и определите, что вычисляется с помощью этого алгоритма.
b) Постройте рекуррентное уравнение для количества умножений, выполняемых в алгоритме, и найдите его решение.
c) Постройте рекуррентное уравнение для количества сложений/вычитаний, выполняемых в алгоритме, и найдите его решение.

8) Дан рекурсивный алгоритм

```
double Min1(double *A, int N) {
// входные данные: массив A[0..N-1] вещественных чисел
        if(N==1) return A[0];
        double temp=Min1(A,N-1);
        if (temp<=A[N-1]) return temp;
```

```
        return A[N-1];
    }
```

a) Что вычисляется с помощью этого алгоритма?
b) Постройте рекуррентное уравнение для количества выполнений основной операции алгоритма и найдите его решение.

9) Дан рекурсивный алгоритм, в котором выполняется разделение массива на две части. Первый раз он вызывается так: double r=Min2(A,0,N-1):

```
double Min2(double *A, int NLeft, int NRight)  {
    if (NLeft == NRight) return A[NLeft];
    double t1=Min2(NLeft,(NLeft+NRight) div 2);
    double t2= Min2(,(NLeft+NRight) div 2+1,NRight);
    if (t1<=t2) return t1;
    else return t2;
}
```

a) Что вычисляется с помощью этого алгоритма?
b) Постройте рекуррентное уравнение для количества выполнений основной операции алгоритма и найдите его решение.
c) Какой из алгоритмов — Min1 или Min2 — быстрее? Можете ли вы предложить алгоритм для решения этой же задачи, который был бы эффективнее двух рассмотренных выше алгоритмов?

10) Определитель матрицы A размером N × N, обозначаемый как det A, может быть построен следующим образом.

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1N} \\ a_{21} & \cdots & a_{2N} \\ \vdots & \ddots & \vdots \\ a_{N1} & \cdots & a_{NN} \end{bmatrix}$$

При N=1 det A= a_{11}. При N>1 его можно вычислить по приведенной ниже рекурсивной формуле:

$$\det A = \sum_{j=1}^{N} s_j a_{1j} \det A_j,$$

В этой формуле элемент s_j равен +1, если j нечетное, и -1, если j четное; a_{1j} – элемент первой строки и j-го столбца; A_j – детерминант матрицы размером (N-1)×(N-1), полученной из исходной матрицы A путем вычеркивания первой строки и j-го столбца.

a) Постройте рекуррентное уравнение для количества операций умножения, выполняемых при вычислении детерминанта.
b) Не решая это рекуррентное уравнение, что вы можете сказать о его порядке роста в сравнении с факториалом N!?

Резюме

1. Различают два вида эффективности алгоритмов — временную и протранственную. ***Временная эффективность*** указывает, насколько быстро выполняется алгоритм; ***пространственная эффективность*** показывает, как много дополнительной памяти требуется алгоритму для работы.
2. Временная эффективность алгоритма в основном определяется как функция от размера входных данных, путем подсчета количества выполнения базовых операций. ***Базовая операция*** — это операция, которая вносит основной вклад в общее время выполнения алгоритма. Обычно это операция с наибольшим временем работы в наиболее глубоко вложенном цикле.
3. Для ряда алгоритмов время работы существенно отличается для разных входных данных одного и того же размера, что приводит к эффективности для ***наихудшего, наилучшего*** и ***среднего случаев.***
4. Анализ временной эффективности алгоритма основывается на порядке роста времени работы алгоритма при росте размера его входных данных до бесконечности.
5. Для указания и сравнения асимптотических порядков роста функций, выражающих эффективность алгоритмов, используются O-, Ω- и Θ-обозначения, которые соответственно определяют: оценку роста функции сверху, снизу и совпадение порядков роста функций с точностью до коэффициента
6. Эффективность подавляющего большинства алгоритмов подразделяется на несколько классов — ***константную, логарифмическую, линейную, n*** log ***n, квадратичную, кубическую*** и ***экспоненциальную.***
7. Главным инструментом анализа временной эффективности нерекурсивного алгоритма является построение выражения для суммы количества выполнений его основной операции и выяснение порядка ее роста.
8. Главным инструментом анализа временной эффективности рекурсивного алгоритма является построение рекуррентного соотношения для количества выполнений его основной операции и выяснение порядка его роста.
9. Лаконичность рекурсивного алгоритма может скрывать его неэффекивность.
10. ***Числа Фибоначчи*** представляют собой важную последовательность целых чисел, в которой каждый элемент рааен сумме двух своих предшественников. Имеется несколько алгоритмов вычисления чисел Фибоначчи с существенно различной эффективностью.

Контрольные вопросы

https://studfiles.net/preview/2213464/

Принято считать, что алгоритмы с полиномиальной сложностью являются «быстрыми», в то время как алгоритмы, сложность которых больше полиномиальной, — «медленными». С этой точки зрения алгоритмы с экспоненциальной сложностью являются медленными. Однако, это предположение не совсем точное. Дело в том, что время работы алгоритма зависит от значения *n* (размерности задачи) и сопутствующих констант скрытых в О-нотации. В некоторых случаях для малых значений *n* полиномиальное вре-

мя может превосходить экспоненциальное. Однако, для больши́х значений *n* время работы алгоритма с экспоненциальной сложностью существенно больше.

Литература

1. Левитин, А.В. Алгоритмы: введение в разработку и анализ /А.В.Левитин : пер. с англ., – М.: Вильямс, 2006. – С. 75-128.
2. Дасгупта, С. Алгоритмы / С. Дасгупта, Х. Пападимитриу, У. Вазирани: пер. с англ. под ред. А. ,Шеня. – М.: МЦНМО, 2014. – С.10-12.