Модель вычислений RAM

Машиной с произвольным доступом к памяти. или RAM-машиной. Согласно этой модели наш компьютер работает таким образом

1. для исполнения любой *простой* операции (+, *, -, =, if, call) требуется ровно один временной шаг;

2. циклы и подпрограммы не считаются простыми операциями, а состоят из нескольких простых операций. Нет смысла считать подпрограмму сортировки одношаговой операцией, т. к. для сортировки 1000000 элементов потребуется определенно намного больше времени, чем для сортировки десяти элементов. Время исполнения цикла или подпрограммы зависит от количества итераций или специфического характера подпрограммы;

3. каждое обращение к памяти занимает один временной шаг. Кроме этого, наш компьютер обладает неограниченным объемом оперативной памяти. Кэш и диск в модели RAM не применяются.

Анализ сложности наилучшего, наихудшего и среднего случая

С помощью RAM-модели можно подсчитать количество шагов, требуемых алгоритму для исполнения любого экземпляра задачи. Но чтобы получить общее представление о том, насколько хорошим или плохим является алгоритм, нам нужно знать, как он работает со *всеми* экземплярами задачи.

Чтобы понять, что означает наилучший, наихудший и средний случай сложности алгоритма (т. е. время его исполнения в соответствующем случае), нужно рассмотреть исполнение алгоритма на всех возможных экземплярах входных данных.

В случае задачи сортировки множество входных экземпляров состоит из всех возможных компоновок ключей n по всем возможным значениям n. Каждый входной экземпляр можно представить в виде точки графика (рис. 1), где ось x представляет размер входа задачи (для сортировки это будет количество элементов, подлежащих сортировке), а ось y — количество шагов, требуемых алгоритму для обработки данного входного экземпляра.

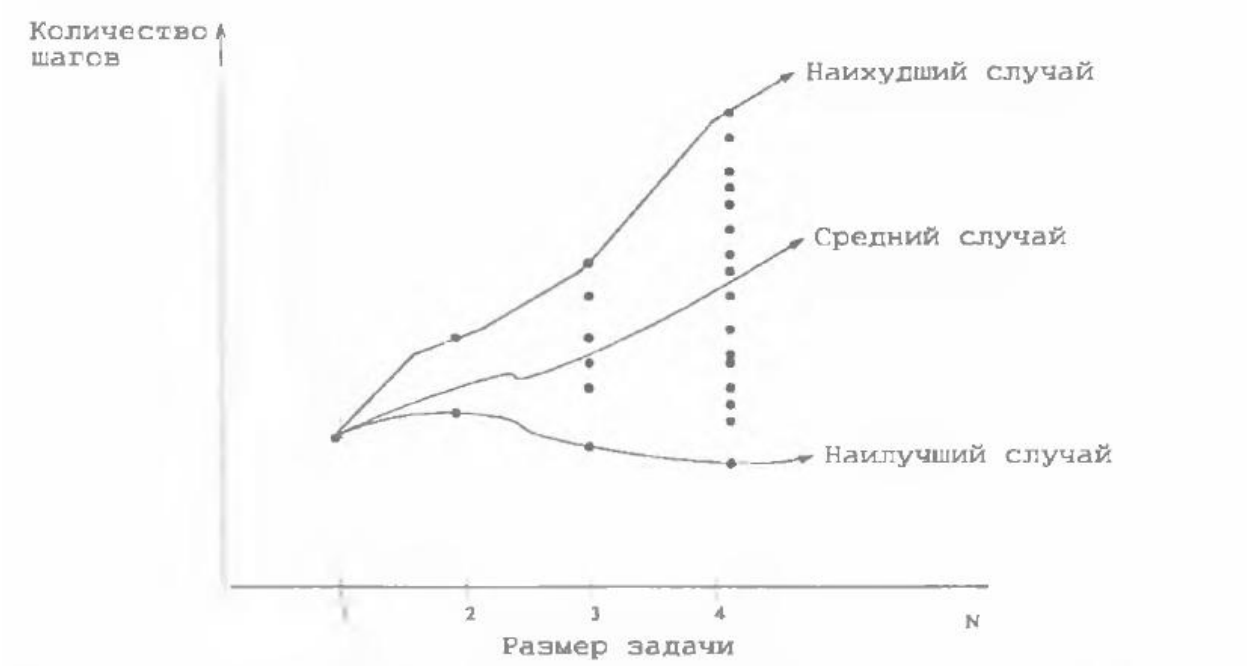

Рис. 1. Наилучший. наихудший и средний случай сложности алгоритма

Эти точки естественным обратом выстраиваются столбцами, т. к. размер входа может быть только целым числом (т. е., сортировка 10,57 элементов лишена смысла). На графике этих точек можно определить три представляющих интерес функции:

1 *сложность алгоритма в наихудшем случае*— это функция, определяемая максимальным количеством шагов, требуемых для обработки любого входного экземпляра размером *n.* Этот случай отображается кривой, проходящей через самую высшую точку каждого столбца;

2 *сложность алгоритма в наилучшем случае*— это функция, определяемая минимальным количеством шагов, требуемых для обработки любого входного экземпляра размером *n.* Этот случай отображается кривой, проходящей через самую низшую точку каждого столбца;

3. сложность алгоритма в среднем случае — это функция, определяемая средним количеством шагов, требуемых для обработки всех экземпляров размером *n.*

наиболее важной является оценка сложности алгоритма в наихудшем случае.

Важно осознавать то, что в каждом случае сложность алгоритма определяется числовой функцией, соотносящей время с размером задачи. Эти функции определены так же строго, как и любые другие числовые функции, будь то уравнение $y = x^2-2x+1$ или цена акций в зависимости от времени. Но функции временной сложности настолько трудны для понимания, что перед началом работы их нужно упростить. Для этой цели используются асимптотические обозначения, в частности обозначение "*О*-большое".

Асимптотические обозначения

Временную сложность наилучшего, наихудшего и среднего случая для любого алгоритма можно представить как числовую функцию от размеров возможных экземпляров задачи. Но работать с этими функциями очень трудно, т. к. они обладают такими свойствами:
1 *являются слишком волнистыми*. Время исполнения алгоритма, например, двоичного поиска, обычно меньше для массивов, имеющих размер $n=2^k-1$, где *к*— целое число. Эта особенность не имеет большого значения, но служит предупреждением, что *точная* функция временной сложности любого алгоритма вполне может иметь неровный график с небольшими выпуклостями и впадинами, как показано на рис. 2;

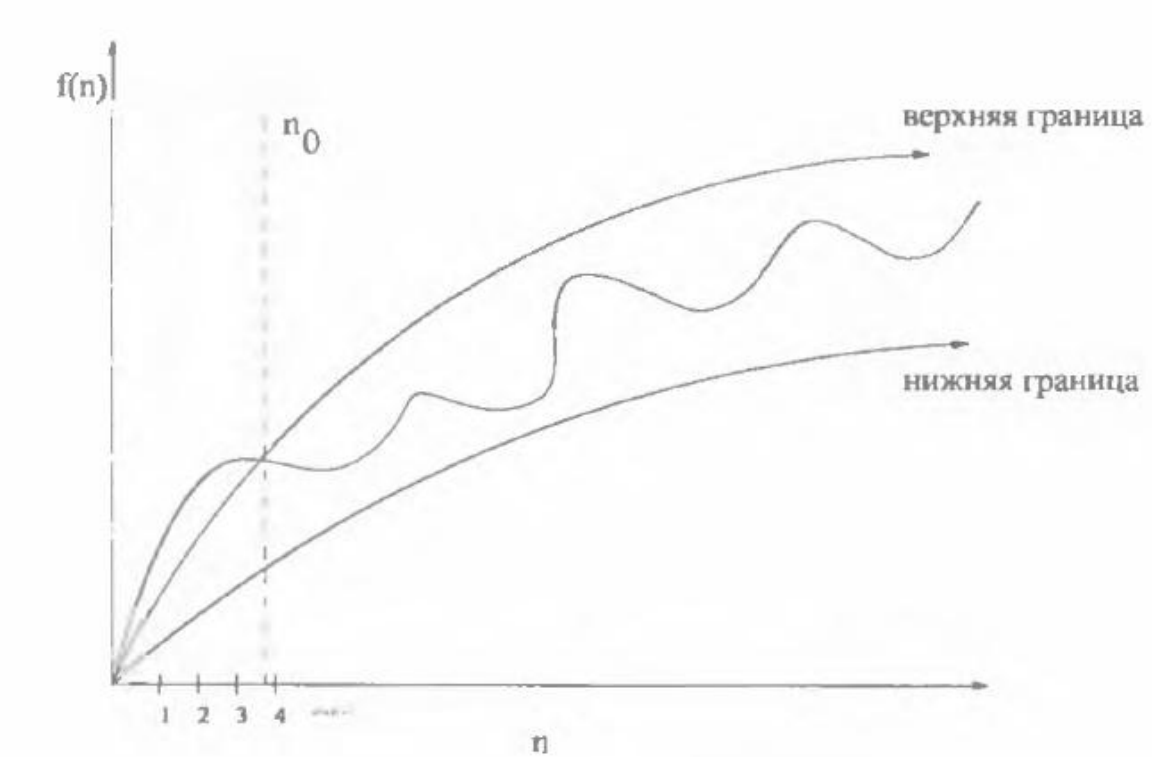

Рис. 2. Верхняя и нижняя границы. действительные для $n > n_0$, сглаживают волнистость сложных функций

требуют слишком много информации для точного определения. Чтобы сосчитать точное количество инструкций RAM-машины, исполняемых в худшем случае, нужно, чтобы алгоритм был расписан в подробностях полной компьютерной программы. Более того, точность ответа зависит от маловажных деталей кодировки (например. был ли употреблен оператор **case** вместо вложенных операторов **if).** Точный анализ наихудшего случая, например, такого:

$$T(n) = 12754n^2 + 4353n + 834\lg_2 n + 13546$$

очевидно, был бы очень трудной задачей, решение которой не предоставляет нам никакой дополнительной информации, кроме той, что "с увеличением *n* временная сложность возрастает квадратически".

Оказывается, намного легче работать с верхней и нижней границами функций временной сложности, используя для этого асимптотические обозначения (*O*-большое и Ω- большое соответственно). Асимптотические обозначения позволяют упростить анализ, поскольку игнорируют детали, которые не влияют на сравнение эффективности алгоритмов.

В частности, в асимптотических обозначениях игнорируется разница между постоянными множителями. Например, в анализе с применением асимптотического обозначения функции f(n) =2n и *g(n)=n* являются идентичными. Этот постоянный множитель, равняющийся двум, не предоставляет нам никакой информации собственно об алгоритме, т. к. в обоих случаях исполняется один и тот же алгоритм. При сравнении алгоритмов такие постоянные коэффициенты не принимаются во внимание.

Формальные определения, связанные с асимптотическими обозначениями, выглядят таким образом:

$f(n) = O(g(n))$ означает, что функция $f(n)$ *ограничена сверху* функцией $c \cdot g(n)$. Иными словами, существует такая константа c, для которой $f(n) \le c \cdot g(n)$ при достаточно большом значении n (т. е. $n \ge n_0$ для некоторой константы n_0);

$f(n) = \Omega(g(n))$ означает, что функция $f(n)$ *ограничена снизу* функцией $c \cdot g(n)$. Иными словами, существует такая константа c, для которой $f(n) \ge c \cdot g(n)$ для всех $n \ge n_0$;

$f(n) = \Theta(g(n))$ означает, что функция $f(n)$ *ограничена сверху* функцией $c_1 \cdot g(n)$, а *снизу* функцией $c_2 \cdot g(n)$ для всех $n \ge n_0$. Иными словами, существуют константы c_1 и c_2, для которых $f(n) \le c_1 \cdot g(n)$ и $f(n) \ge c_2 \cdot g(n)$. Следовательно, функция $g(n)$ дает нам хорошие ограничения для функции $f(n)$.

Графическая иллюстрация этих определений дается на рис.3.

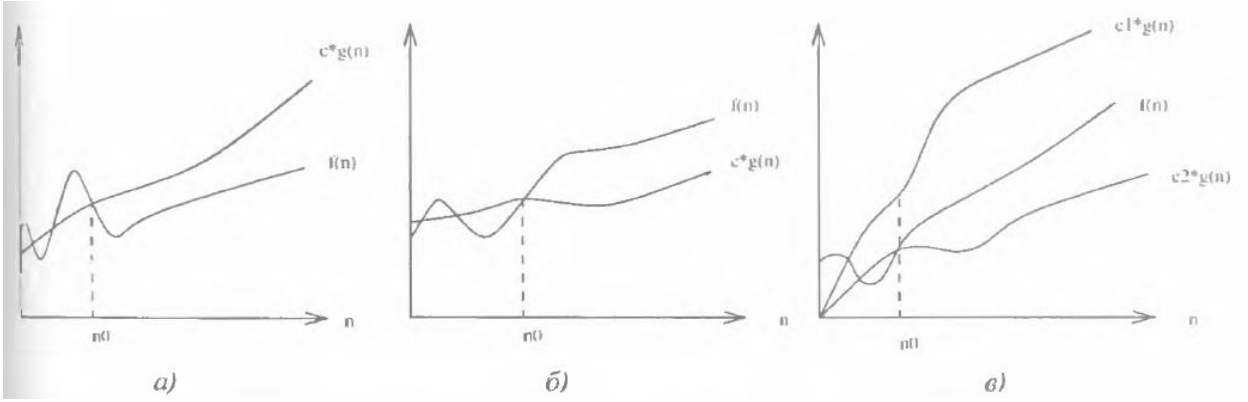

Рис. 3. Графическая иллюстрация асимптотических обозначений:

В каждом из этих определений фигурирует константа n_0, после которой эти определения всегда верны. Нас не интересуют небольшие значения *n*, т. е. значения слева от n_0 конце концов, нам безразлично, что один алгоритм

может отсортировать, скажем, шесть или восемь элементов быстрее, чем другой. Мы ищем алгоритм для быстрой сортировки 10000 или 1000000 элементов В этом отношении асимптотические обозначения позволяют нам игнорировать несущественные детали и концентрироваться на общей картине.

$3n^2 - 100n + 6 = O(n^2)$, т. к. выбрано $c = 3$ и $3n^2 > 3n^2 - 100n + 6$;

$3n^2 - 100n + 6 = O(n^3)$, т. к. выбрано $c = 1$ и $n^3 > 3n^2 - 100n + 6$ при $n > 3$;

$3n^2 - 100n + 6 \neq O(n)$, т. к. для любого значения c выбрано $cn < 3n^2$ при $n > c$;

$3n^2 - 100n + 6 = \Omega(n^2)$, т. к. выбрано $c = 2$ и $2n^2 < 3n^2 - 100n + 6$ при $n > 100$;

$3n^2 - 100n + 6 \neq \Omega(n^3)$, т. к. выбрано $c = 3$ и $3n^2 - 100n + 6 < n^3$ при $n > 3$;

$3n^2 - 100n + 6 = \Omega(n)$, т. к. для любого значения c выбрано $cn < 3n^2 - 100n + 6$ при $n > 100c$;

$3n^2 - 100n + 6 = \Theta(n^2)$, т. к. применимо как O, так и Ω;

$3n^2 - 100n + 6 \neq \Theta(n^3)$, т. к. применимо только O;

$3n^2 - 100n + 6 \neq \Theta(n)$, т. к. применимо только Ω.

Асимптотические обозначения позволяют получить приблизительное представление о равенстве функций при их сравнении. Выражение типа $n^2=O(n^3)$ может выглядеть странно, но его значение всегда можно уточнить, пересмотрев его определение в терминах верхней и нижней границ. Возможно, это обозначение будет более понятным, если в данном случае рассматривать символ равенства (=) как означающий "одна из функций, принадлежащих к множеству функций". Очевидно, что n является одной из функций, принадлежащих множеству функций $O(n)$.

Скорость роста и отношения доминирования

Используя асимптотические обозначения, мы пренебрегаем постоянными множителями, не учитывая их при вычислении функций. При таком подходе функции $f(n) = 0.001n^2$ и $g(n) = 1000n^2$ для нас одинаковы, несмотря на то, что значение функции g(n) в миллион раз больше значения функции $f(n)$ для любого n.

Причина, по которой достаточно грубого анализа, предоставляемого обозначением *O*-большое. приводится в табл. 1, в которой перечислены наиболее распространенные функции и их значения для нескольких значений n. В частности, здесь можно увидеть время исполнения $f(n)$ операций алгоритмов на быстродействующем компьютере, исполняющем каждую операцию за одну наносекунду (10^{-9} секунд). На основе представленной в таблице информации можно сделать следующие выводы:

1.время исполнения всех этих алгоритмов примерно одинаково для значений n - 10;

2. любой алгоритм с временем исполнения n! становится бесполезным для значений т >20;

3. диапазон алгоритмов с временем исполнения 2^n несколько шире, но они также становятся непрактичными для значении n> 40;

4. алгоритмы с квадратичным временем исполнения n^2 применяются при $n<$ 10 000, после чего их производительность начинает резко ухудшаться. Эти алгоритмы, скорее всего, будут бесполезны для значений $n > 1\ 000000$;
5.алгоритмы с линейным и логарифмическим временем исполнения остаются полезными при обработке миллиарда элементов:
6. в частности, алгоритм *O(lgn*) без труда обрабатывает любое вообразимое количество элементов.

Таблица 1. Скорость роста основных функций

n/*f*(*n*)	lg*n*	*n*	*n*lg*n*	n^2	2^n	*n*!
10	0.003 мкс	0.01 мкс	0,033 мкс	0.1 мкс	1 мкс	3.63 мс
20	0.004 мкс	0.02 мкс	0,086 мкс	0.4 мкс	1 мс	77.1 лет
30	0.005 мкс	0.03 мкс	0.147 мкс	0.9 мкс	1 с	$8,4 \cdot 10^{15}$ лет
40	0.005 мкс	0.04 мкс	0.213 мкс	1.6 мкс	18,3 мин	
50	0.006 мкс	0.05 мкс	0.282 мкс	2,5 мкс	13 дней	
100	0.007 мкс	0,1 мкс	0,644 мкс	10 мкс	4×10^{13} лет	
1 000	0.010 мкс	1.00 мкс	9,966 мкс	1 мс		
10 000	0.013 мкс	10 мкс	130 мкс	100 мс		
100 000	0.017 мкс	0.10 мс	1.67 мс	10 с		
1 000 000	0.020 мкс	1 мс	19.93 мс	16,7 мин		
10 000 000	0.023 мкс	0,01 с	0.23 с	1,16 дней		
100 000 000	0.027 мкс	0,10 с	2.66 с	115.7 дней		
1 000 000 000	0.030 мкс	1 с	29,90 с	31.7 лет		

Из этого можно сделать основной вывод, что, даже игнорируя постоянные множители, мы получаем превосходное общее представление о годности алгоритма для решения задачи определенного размера. Алгоритм с временем исполнения $f(n)=n^3$ секунд победит алгоритм с временем исполнения $g(n)=1000000n^2$ секунд только при $n < 1000000$. На практике такая громадная разница между постоянными множителями алгоритмов встречается намного реже, чем задачи по обработке большего объема входных данных.

Отношения доминирования

Посредством асимптотических обозначений функции разбиваются на классы, в каждом из которых сгруппированы функции с эквивалентным асимптотическим обозначением. Например, функции *f(n)*=0.34n и *g(n)=234,234n* принадлежат к одному и тому же классу, а именно к классу порядка Θ(n). Кроме того, когда функции f и *g* принадлежат к разным классам, они являются разными относительно нашего обозначения. Иными словами, справедливо либо *f(n)*=O(g(n)). либо *g(n)=O*(f(n)). но не оба равенства одновременно.

Говорят, что функция с более быстрым темпом роста *доминирует* над менее быстро растущей функцией точно так же, как более быстро растущая страна в итоге начинает доминировать над отстающей. Когда функции f и *g*

принадлежат к разным классах (т. е $f(n)=\Theta(g(n))$. говорят, что функция f *доминирует* над функцией g, когда $f(n)=O(g(n))$. Это отношение иногда обозначается как g » f.

Далее приводятся эти классы в порядке возрастания доминирования.

1.Функции-константы, f(n) = 1. Такие функции могут измерять трудоемкость сложения двух номеров, распечатывания какого-либо текста или рост таких функций, как $f(n)=\min(n,100)$. По большому счету, зависимость от параметра *n* отсутствует.

2 *Логарифмические функции, f(n)*=logn. Логарифмическая временная сложность проявляется в таких алгоритмах, как двоичный поиск. С увеличением *n* такие функции возрастают довольно медленно, но быстрее, чем функции-константы (которые вообще не возрастают).

3 *Линейные функции, f(n) = n.* Такие функции измеряют трудоемкость просмотра каждого элемента в массиве элементов один раз (или два раза, или десять раз), например, для определения наибольшего или наименьшего элемента или для вычисления среднего значения.

4 *Суперлинейные функции, f(n) = nlgn.* Эти функции возрастают лишь немного быстрее. чем линейные (см. табл.1), но достаточно быстро, чтобы составить другой класс доминирования.

5 *Квадратичные функции, f(n)* = n^2. Эти функции измеряют трудоемкость просмотра большинства или всех пар элементов в универсальном множестве из *n* элементов. Они возникают в таких алгоритмах, как сортировка вставками или сортировка методом выбора.

6 *Кубические функции, f(n) = n^3.* Эти функции возникают при перечислении всех триад элементов в универсальном множестве из *n* элементов.

7. Показательные функции, f(n) = c^n, константа c> 1. Эти функции возникают при перечислении всех подмножеств множества из *n* элементов. Как мы видели в табл. 1. экспоненциальные алгоритмы быстро становятся бесполезными с увеличением количества элементов *n.* Впрочем, не так быстро, как функции из следующего класса.

8 *Факториальные функции, f(n) = n!.* Факториальные функции определяют все перестановки *n* элементов.

Вы должны помнить следующее отношение:

$$n! \gg 2^n \gg n^3 \gg n^2 \gg n\log n \gg n \gg \log n \gg 1$$

Работа с асимптотическими обозначениями

Сложение функций

Сумма двух функций определяется доминантной функцией, а именно:

$$O(f(n)) + O(g(n)) = O(\max(f(n),(g(n)))$$

$$\Omega(f(n)) + \Omega(g(n)) = \Omega(\max(f(n),(g(n)))$$

$$\Theta(f(n)) + \Theta(g(n)) = \Theta(\max(f(n),(g(n)))$$

Это обстоятельство очень полезно при упрощении выражений, т. к. оно подразумевает, что $n^3+n^2+n+1 = O(n^3)$. По сравнению с доминантным членом все остальное является несущественным.

Умножение функций

Умножение можно рассматривать, как повторяющееся сложение. Рассмотрим умножение на любую константу $c > 0$, будь это 1,02 или 1000000. Умножение функции на константу не может повлиять на ее асимптотическое поведение, т. к. в анализе функции $c*f(n)$ с применением обозначения "О-большое" мы можем умножить ограничивающие константы на 1/c, что даст нам необходимые константы для анализа. Таким образом:

$$O(cf(n)) \rightarrow O(f(n))$$

$$\Omega(cf(n)) \rightarrow \Omega(f(n))$$

$$\Theta(cf(n)) \rightarrow \Theta(f(n))$$

$$O(f(n))*O(g(n)) \rightarrow O(f(n)*g(n))$$

$$\Omega(f(n))*\Omega(g(n)) \rightarrow \Omega(f(n)*g(n))$$

$$\Theta(f(n))*\Theta(g(n)) \rightarrow \Theta(f(n)*g(n))$$

Оценка эффективности

Проанализируем алгоритм сортировки методом выбора. При сортировке этим способом определяется наименьший неотсортированный элемент и помещается в конец отсортированной части массива. Процедура повторяется до тех пор, пока все элементы массива не будут отсортированы. Графическая иллюстрация работы алгоритма представлена на рис. 4, а соответствующий код на языке С в листинге 1.

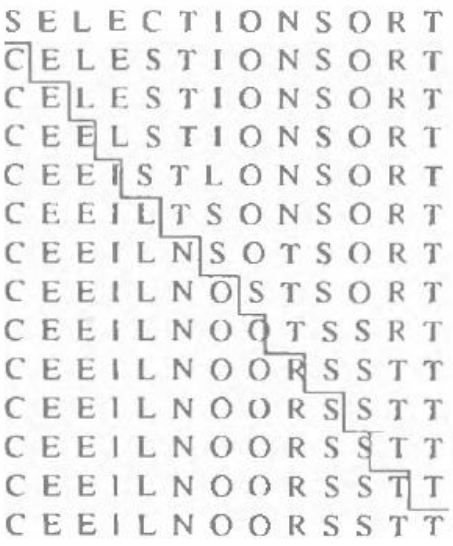

Рис. 4. Графическая иллюстрация работы алгоритма сортировки методом выбора

```
election_sort(int s[], int n)

int i,j;                    /* Счетчики */
int min;                    /* Указатель наименьшего элемента */

for (i=0; i<n; i++) {
  min=i;
  for (j=i+1; j<n; j++)
    if (s[j] < s[min]) min=j;
  swap(&s[i],&s[min]);
}
```

Внешний цикл исполняется *n* раз. Внутренний цикл исполняется *n- i - 1* раз, где *i* - счетчик внешнего цикла. Точное количество исполнений оператора if определяется следующей формулой:

$$S(n) = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-1} n - i - 1$$

Это формула сложения целых чисел в убывающем порядке, начиная с *n - 1*. т.е.:

$$S(n) = (n-1) + (n-2) + (n-3) + \ldots + 2 + 1$$

Какие выводы мы можем сделать на основе такой формулы? При работе с асимптотическими обозначениями нас интересует только *степень* выражения. Один из подходов - считать, что мы складываем *n*- 1 элементов, среднее значение которых равно приблизительно *n*/2. Таким образом мы получаем *S(n) = n(n*-1)/2.

Другим подходом будет использование верхней и нижней границ. Мы имеем не более *n* элементов, значение каждого из которых не превышает n-1. Таким образом, $S(n) < n(n - 1) = O(n^2)$. Также мы имеем n/2 элементов, чье значение больше чем *n/2.* Соответственно. $S(n) > (n/2) * (n/2) = \Omega(n^2)$. Все это говорит нам, что время исполнения равно $\Theta(n^2)$. т. е. сложность сортировки методом выбора является квадратичной.

Сортировка вставками

Основное практическое правило при асимптотическом анализе гласит, что время исполнения алгоритма в наихудшем случае получается умножением наибольшего возможного количества итераций каждого вложенного цикла. Рассмотрим, например, алгоритм сортировки вставками из листинга 1, внутренние циклы которого приведены в листинге 2.

```
for (i=1; i<n; i++)
    j=i;
    while ((j>0) && (s[j] < s[j-1])) {
        swap(&s[j] ,&s[j-1]);
        j = j-1;
    }
```

Сколько итераций осуществляет внутренний цикл while? На этот вопрос сложно дать однозначный ответ, т. к. цикл может быть остановлен досрочно, если произойдет выход за границы массива (j>o) или элемент окажется на должном месте в отсортированной части массива (s[j]<s[j-i]). Так как в анализе наихудшего случая мы ищем верхнюю границу времени исполнения, то мы игнорируем досрочное завершение и полагаем. что количество исполняемых в этом цикле итераций всегда будет i. Более того, мы можем допустить, что *всегда* исполнятся *n* итераций, т. к. i < *n.* А т. к. внешний цикл исполняется n раз, то алгоритм сортировки вставками должен быть квадратичным, т. е. O(n^2).

Такой грубый анализ методом округления всегда оказывается результативным, в том смысле, что полученная верхняя граница времени исполнения (O-большое) всегда будет правильной. Иногда этот результат может быть даже завышен в худшую сторону, т. е. в действительности время исполнения худшего случая окажется меньшим, чем результат, полученный при анализе. Тем не менее, рекомендуюn этот подход в качестве основы для простого анализа алгоритмов.

Сравнение строк

Сравнение комбинаций символов - основная операция при работе с текстовыми строками. Далее приводится алгоритм для реализации функции поиска определенного текста, которая является обязательной частью любого веб-браузера или текстового редактора.

ЗАДАЧА. Найти подстроку в строке.

Вход. Текстовая строка *t* и строка для поиска *p* (рис. 5).

Выход. Содержит ли строка t подстроку *p,* и, если содержит, в каком месте?

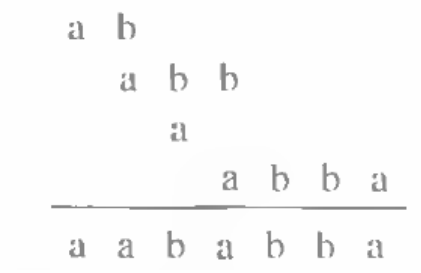

Рис.5. Пример: поиск полстроки *abba* в тексте *aababba*

Пример практического применения этого алгоритма- поиск упоминания определенной фамилии в новостях. Для данного экземпляра задачи текстом t будет статья, а строкой *p* для поиска - указанная фамилия.
Эта задача решается с помощью довольно простого алгоритма (листинг 3), который допускает, что строка *p* может начинаться в любой возможной позиции в тексте t, и выполняет проверку, действительно ли, начиная с этой позиции, текст содержит искомую строку.

```
findmatch(char *p, char *t)

int i,j;                        /* Счетчики */
int m, n;                       /* Длины строк */
m = strlen(p);
n = strlen(t);
for (i=0; i<=(n-m); i=i+1)
     j=0;
     while ((j<m) && (t[i+j]==p[j]))
          j = j+1;
     if (j == m) return(i);
}
return(-1);
```

Каким будет время исполнения этих двух вложенных циклов в наихудшем случае? Внутренний цикл *while* исполняется максимум *m* раз, а возможно, и намного меньше, если поиск заканчивается неудачей. Кроме оператора while, внешний цикл содержит еще два оператора. Внешний цикл исполняется самое большее *n-m* раз, т. к. после продвижения направо по тексту оставшийся фрагмент будет короче искомой строки. Общая временная сложность - произведение значений оценки временной сложности внешнего и вложенного циклов, что дает нам время исполнения в худшем случае O*((n-1)*(m + 2)).
При этом мы не учитываем время, потраченное на определение длины строк с помощью функции strlen. Так как мы не знаем, каким образом реализована эта функция, мы можем лишь строить предположения относительно времени ее работы. Если мы явно считаем количество символов, пока не достигнем конца строки, то отношение между временем исполнения этой операции и длиной строки будет линейным. Значит, время работы будет равно O*(n + m + (n - m)(m* + 2)).
С помощью асимптотических обозначений это выражение можно упростить. Так как *m + 2* = Θ(m), выражение " + 2" не представляет интереса, поэтому останется только *O(n + m + (n - m)m).* Выполнив умножение, получаем выражение *0(n+m + nm — m²).* которое выглядит довольно непривлекательно. Но мы знаем, что в любой представляющей интерес задаче *n> m.* т. к. невозможно, чтобы искомая строка *p* была длиннее, чем текст t, в котором выполняется ее поиск. Одним из следствий этого обстоятельства является

отношение $(n + m) < 2n = \Theta(n)$. Таким образом, формула времени исполнения для наихудшего случая упрощается дальше до $O(n + nm - m^2)$.
Еще два замечания. Обратите внимание, что $n < nm$, т. к. для любой представляющей интерес строки поиска $m > 1$. Таким образом, $n + nm = \Theta(nm)$, и мы можем опустить дополняющее n. упростив формулу анализа до $O(nm - m^2)$.
Кроме того, заметьте, что член $-m^2$ отрицательный, вследствие чего он только уменьшает значение выражения внутри скобок. Так как "О-болыиое" задает верхнюю границу. то любой отрицательный член можно удалить, не искажая оценку верхней границы. Тот факт, что $n > m$ подразумевает, что $nm > m^2$. поэтому отрицательный член недостаточно большой, чтобы аннулировать любой другой оставшийся член. Таким образом, время исполнения этого алгоритма в худшем случае можно выразить просто как $O(nm)$.

Логарифмы и их применение

Слово "логарифм" — почти анаграмма слова "алгоритм". Но интерес к логарифмам вызван не этим обстоятельством. Возможно, что кнопка калькулятора с обозначением "log"— единственное место, где вы сталкиваетесь с логарифмами в повседневной жизни. Также возможно, что вы уже не помните назначение этой кнопки. *Логарифм* — это функция, обратная показательной. То есть, выражение $b^x = y$ эквивалентно выражению $x = \log_b y$. Более того, из определения логарифма следует, что

$$b^{\log_b y} = y$$

Показательные функции возрастают чрезвычайно быстро, как может засвидетельствовать любой, кто когда-либо выплачивал долг по кредиту. Соответственно, функции, обратные показательным, т. е. логарифмы, возрастают довольно медленно. Логарифмические функции возникают в любом процессе, содержащем деление пополам. Давайте рассмотрим несколько таких примеров.

Логарифмы и двоичный поиск

Двоичный поиск является хорошим примером алгоритма с временной логарифмической сложностью $O(\log n)$. Чтобы найти определенного человека по имени p в телефонной книге, содержащей n имен, мы сравниваем имя p с выбранным именем посередине книги (т. е. с n/2-м именем. Независимо от того, находится ли имя p перед выбранным именем или после него, после этого сравнения мы можем отбросить половину всех имен в книге. Этот процесс повторяется с половиной книги, содержащей искомое имя и т. д., пока не останется всего лишь одно имя. которое и будет искомым. По определению количество таких делений равно $\log_2 n$. Таким образом, чтобы найти любое имя в телефонной книге содержащей миллион имен, достаточно выполнить всего лишь двадцать сравнений.

Идея двоичного поиска является одной из наиболее плодотворных в области разработки алгоритмов. Эта мощь становится очевидной, если мы представим, что в окружающем нас мире имеются только неотсортированные телефонные книги. Как видно из табл. **1.1,** алгоритмы с временной сложностью O(logn) можно применять для решения задач с практически неограниченным размером входных данных.

Логарифмы и деревья

Двоичное дерево высотой в один уровень может иметь две концевые вершины (листа), а дерево высотой в два уровня может иметь до четырех листов. Какова высота *h* двоичного дерева, имеющего *n* листов? Обратите внимание, что количество листов удваивается при каждом увеличении высоты дерева на один уровень. Таким образом, зависимость количества листов *n* от высоты дерева h выражается формулой $n = 2^h$, откуда следует, что $h = \log_2 n$.

Теперь перейдем к общему случаю. Рассмотрим деревья, которые имеют *d* потомков (для двоичных деревьев *d* = 2). Такое дерево высотой в один уровень может иметь *d* количество листов, а дерево высотой в два уровня может иметь d^2 количество листов. Количество листов на каждом новом уровне можно получить, умножая на *d* количество листов предыдущего уровня. Таким образом, количество листов *n* выражается формулой $n = d^h$. т. е. высота находится по формуле $h = \log_d n$; (рис. 6).

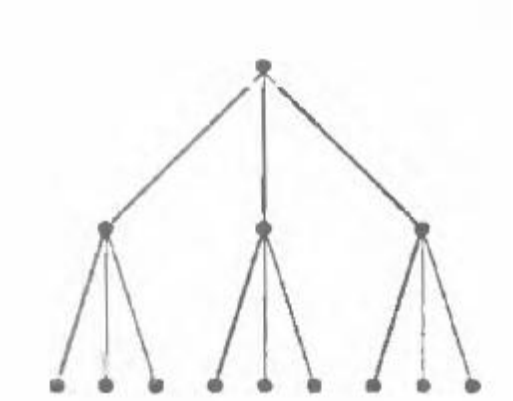

Рис. 6. Дерево высотой *h* и количеством потомков *d* лля каждого узла имеет d^h листьев.

В данном случае h = 2. *d=3*

Из вышеизложенного можно сделать вывод, что деревья небольшой высоты могут иметь очень много листьев. Это обстоятельство является причиной того, что двоичные деревья лежат в основе всех быстро обрабатываемых структур данных.

Логарифмы и биты

Положим, имеются две однобитовые комбинации (0 и I) и четыре двухбитовые комбинации (00, 01, 10 и 11). Сколько битов *w* потребуется, чтобы представить любую из *n* возможных разных комбинаций, будь то один из *n* элементов или одно из целых чисел от 1 до n?

Ключевым наблюдением здесь является то обстоятельство, что нужно иметь, по крайней мере, *n* разных битовых комбинаций длиной w. Так как количество разных битовых комбинаций удваивается с добавлением каждого бита то нам нужно, по крайней мере, *w* битов, где $2^w = n$, т. е. нам нужно $w = \log_2 n$; битов.

Логарифмы и умножение

Логарифмы имели особенно большую важность до распространения карманных калькуляторов. Применение логарифмов было самым легким способом умножения больших чисел вручную, либо с помощью логарифмической линейки, либо с использованием таблиц.

Но и сегодня логарифмы остаются полезными для выполнения операций умножения, особенно для возведения в степень. Вспомните, что

$$\log_a(xy) = \log_a(x) + \log_a(y)$$

. т. е., что логарифм произведения равен сумме логарифмов сомножителей. Прямым следствием этого является формула:

$$\log_a n^b = b \cdot \log_a n$$

Выясним, как вычислить a^b для любых *a* и *b,* используя функции exp(x) и ln(x) на карманном калькуляторе, где

$$\exp(x) = e^x \text{ и } \ln(x) = \log_e(x)$$

Мы знаем, что:

$$a^b = \exp(\ln(a^b)) = \exp(b \ln a)$$

Таким образом, задача сводится к одной операции умножения с однократным вызовом каждой из этих функций.

Быстрое возведение в степень

Допустим, что нам нужно вычислить *точное* значение *a* для достаточно большого значения *n.* Такие задачи, в основном, возникают в криптографии при проверке числа на простоту. Проблемы с точностью не позволяют нам воспользоваться ранее рассмотренной формулой возведения в степень.

Самый простой алгоритм выполняет *n*- 1 операций умножения *(a * a*x ... * *a).* Но можно указать лучший способ решения этой задачи, приняв во внимание, что

$$n = \lfloor n/2 \rfloor + \lceil n/2 \rceil$$

Если *n* четное, тогда

$$a^n = (a^{n/2})^2$$

А если *n* нечетное, то тогда

$$a^n = a(a^{\lfloor n/2 \rfloor})^2$$

В любом случае значение показателя степени было уменьшено наполовину, а вычисление сведено к. самое большее, двум операциям умножения. Таким образом, для вычисления конечного значения будет достаточно O(l*gn)* операций умножения. Псевдокод соответствующего алгоритма показан в листинге 5.

```
function power(a, n)
     if (n = 0) return(1)
     x = power (a, ⌊n/2⌋)
     if (n is even) then return(x^2)
          else return(a * x^2)
```

Этот простой алгоритм иллюстрирует важный принцип "разделяй и властвуй". Разделение задачи на (по возможности) равные подзадачи, всегда окупается. Этот принцип применим и в реальном мире. Когда значение *n* отлично от 2, то входные данные не всегда можно разделить точно пополам, но разница в один элемент между двумя половинами не вызовет никакого серьезного нарушения баланса.

Свойства логарифмов

Как мы уже видели, выражение $b^x = y$ эквивалентно выражению $x = \log_b y$. Член b называется *основанием* логарифма. Особый интерес представляют следующие основания логарифмов:

- **основание b = 2.** *Двоичный логарифм*, обычно обозначаемый как lg*x*, является логарифмом по основанию 2. Мы уже видели, что логарифмами с этим основанием выражается временная сложность алгоритмов, использующих многократное деление пополам (т. е. двоичный поиск) или умножение на два (т. е. листья деревьев). В большинстве случаев, когда речь идет о применении логарифмов в алгоритмах, подразумеваются двоичные логарифмы;

- **основание b = e.** *Натуральный логарифм*, обычно обозначаемый как lnx, является логарифмом по основанию e = 2.71828... Обратной к функции натурального логарифма является экспоненциальная функция $\exp(x) = e^x$. Суперпозиция этих функций дает нам формулу $\exp(\ln x) = x$;
- **основание b = 10.** Менее распространенными на сегодняшний день являются логарифмы по основанию 10, или *десятичные логарифмы*. До появления карманных калькуляторов логарифмы с этим основанием применялись на логарифмических линейках и таблицах алгоритмов.

Логарифм по одному основанию легко преобразовать в логарифм по другому основанию. Для этого применяется следующая формула:

$$\log_a b = \frac{\log_c b}{\log_c a}$$

Таким образом, чтобы изменить основание *a* логарифма $\log_a b$ на c. логарифм просто нужно разделить на $\log_c a$. В частности, функцию натурального логарифма можно с легкостью преобразовать в функцию десятичного логарифма и наоборот.

Из этих свойств логарифмов следуют два важных с арифметической точки зрения следствия.

♦ *Основание логарифма не оказывает значительного влияния на скорость роста функции.* Сравните следующие три значения: $\log_2(1\,000\,000) = 19.9316$. $\log_3(1000000) = 12.5754$ и $\log_{100}(1000000) = 3$. Как видите, большое изменение в основании логарифма сопровождается малыми изменениями в значении логарифма. Чтобы изменить у логарифма основание *a* на *c,* первоначальный логарифм нужно разделить на $\log_c a$. Этот коэффициент теряется в нотации "О-большое", когда *a* и с являются константами. Таким образом, игнорирование основания логарифма при анализе алгоритма обычно оправдано.

♦ *Логарифмы уменьшают значение любой функции.* Скорость роста логарифма любой полиномиальной функции определяется как O(lgn). Это вытекает из равенства:

$$\log_a n^b = b \cdot \log_a n$$

Эффективность двоичного поиска в широком диапазоне задач является прямым следствием этого свойства. Обратите внимание, что двоичный поиск в отсортированном массиве из n^2 элементов требует всего лишь вдвое больше сравнений, чем в массиве из *n* элементов.

Логарифмы уменьшают значение любой функции. С факториалами трудно выполнять какие-либо вычисления, если не пользоваться логарифмами, и

тогда формула становится еще одной причиной появления логарифмов в анализе алгоритмов.

$$n! = \prod_{i=1}^{n} i \rightarrow \log n! = \sum_{i=1}^{n} \log i = \Theta(n \log n)$$

Пределы и отношения доминирования

Отношения доминирования между функциями являются следствием теории пределов, изучаемой в курсе высшей математики. Говорят, что функция $f(n)$ доминирует над функцией $g(n)$, если $\lim_{n\to\infty} g(n)/f(n) = 0$.

Рассмотрим это определение в действии. Допустим, что $f(n) = 2n^2$ и $g(n) = n^2$. Очевидно, что $f(n) > g(n)$ для всех n, но не доминирует над ней, т. к.

$$\lim_{n\to\infty} g(n) / f(n) = \lim_{n\to\infty} n^2 / 2n^2 = \lim_{n\to\infty} 1/2 \neq 0$$

Этого следовало ожидать, т. к. обе функции принадлежат к одному и тому же классу $\Theta(n^2)$. Теперь рассмотрим функции $f(n) = n^3$ и $g(n) = n^2$. Так как

$$\lim_{n\to\infty} g(n) / f(n) = \lim_{n\to\infty} n^2 / n^3 = \lim_{n\to\infty} 1/n = 0$$

то доминирует многочлен более высокого уровня. Это справедливо для любых двух многочленов, а именно n^a доминирует над n^b, если $a > b$, т. к.

$$\lim_{n\to\infty} n^b / n^a = \lim_{n\to\infty} n^{b-a} \rightarrow 0$$

Таким образом, $n^{1,2}$ доминирует над $n^{1,1999999}$.

Перейдем к показательным функциям: $f(n) = 3^n$ и $g(n) = 2^n$. Так как

$$\lim_{n\to\infty} g(n) / f(n) = 2^n / 3^n = \lim_{n\to\infty} (2/3)^n = 0$$

значит, доминирует функция с большим основанием.

Полная картина порядка доминирования функций

$$n! \gg c^n \gg n^3 \gg n^2 \gg n^{1+\varepsilon} \gg n\log n \gg n \gg \sqrt{n} \gg \log^2 n \gg \log n \gg \log n/\log\log n \gg$$
$$\gg \log\log n \gg \alpha(n) \gg 1$$