Сегодня поговорим про сложность алгоритмов.

Для того, чтобы выбрать подходящий для решения задачи алгоритм, нужно уметь оценивать скорость его работы, объём требуемой памяти и т.д. Следует помнить, что не всегда самый быстрый алгоритм является самым лучшим — обычно машинное время сейчас стоит дешевле времени программиста, поэтому часто имеет смысл выбрать не самый быстрый/потребляющий мало памяти алгоритм, а тот, который быстрее и проще написать. Зато если предполагается, что программа будет выполняться часто, стоит использовать эффективный алгоритм, даже если он более сложный. В некоторых случаях эффективные, но сложные алгоритмы могут быть нежелательными, если готовые программы будут поддерживать лица, не участвующие в написании этих программ.

Сложность бывает не только вычислительная, но и емкостная — сколько дополнительной памяти требует программа. Причем часто бывает так, что одна переводится в другую. Например, если предварительно нафигачить в памяти здоровенную таблицу со значениями некоторой сложно вычислимой функции, то эти значения вообще можно будет получать за константное время. Но с емкостной сложностью более-менее понятно, что считать, так что поговорим про вычислительную.

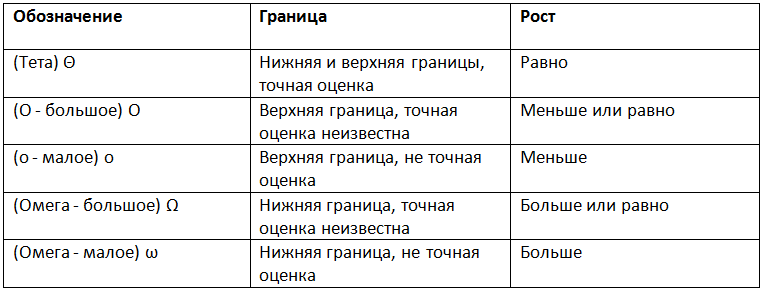
Само измерение времени выполнения алгоритма — не такая простая задача. Во-первых, его нельзя мерять в секундах, минутах и т.д. — время зависит от конкретной машины, на которой выполняется программа, реализующая алгоритм, от компилятора и т.д. Во-вторых, время зависит от входных данных (от них самих или их количества). Поэтому время выполнения обычно считают в неких условных единицах — в элементарных шагах алгоритма, или в количестве операций некой абстрактной машины, например, машины Тьюринга (ну или у Кнута была некая идеальная машина, которую он использовал для оценки времени работы). При этом время выполнения обычно считают как функцию объёма входных данных (например, для алгоритмов сортировки это число элементов в сортируемом массиве, для численных алгоритмов — длина двоичного представления числа и т.п.). Поскольку работа алгоритма зависит и от самих входных данных, можно говорить о времени выполнения в наилучшем, среднем и наихудшем случае. Например, некоторые алгоритмы сортировки работают на уже отсортированном массиве за число операций, линейно зависящее от размера массива.

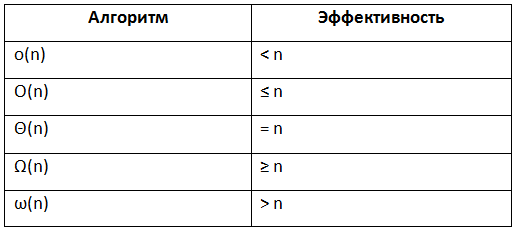
Собственно, точная оценка времени выполнения алгоритма обычно является сложной математической задачей и никому в реальной жизни нафиг не нужна. Обычно используются асимптотическая сложность — приблизительная оценка скорости роста функции времени выполнения в зависимости от размера входных данных, для этого используется O-символика (собственно, с символами O вы ещё не раз встретитесь на матане).

 <=> 

Т.е. f ограничена сверху функцией g с точностью до постоянного множителя.

Например, f(n) = O(n2) — f(n) растёт не быстрее, чем n2 с любой константой. Это имеет самое прямое отношение к алгоритмам, поскольку степень роста функции времени выполнения показывает, какого размера задачи этим алгоритмом имеет смысл решать. Например, алгоритм, имеющий степень роста О(n2) лучше алгоритма, имеющего степень роста O(n3), но не всегда. Алгоритмы с экспоненциальной трудоёмкостью лучше на компах не реализовывать вовсе. Ещё бывают полезны символы омега (то же самое, только Сg <=f) и тета (когда и О, и омега). Также бывают о-малое и омега-малое, но это уже к матанщикам, нам такие детали ни к чему.





Феерический пример из википедии: «пропылесосить ковер» требует время, линейно зависящее от его площади (Θ(*A*)), то есть на ковер, площадь которого больше в два раза, уйдет в два раза больше времени. Соответственно, при увеличении площади ковра в сто тысяч раз, объем работы увеличивается строго пропорционально в сто тысяч раз, и т. п.

Собственно, вычисление трудоёмкости задачи использует следующие правила: если P1 и P2 выполняются за времена T1(n) и T2(n), имеющие порядок O(f(n)) и O(g(n)), то последовательно выполненные эти фрагменты выполняются за время порядка O(max(f(n), g(n))). В частности, из этого следует, что O(n2 + n) = O(n2). Произведение двух функций имеет порядок произведения - если T1(n) и T2(n) имеют порядок роста O(f(n)) и O(g(n)), то T1(n)T2(n) имеет порядок O(f(n)g(n)) - это полезно при анализе циклов.

Например, рассмотрим пузырёк:

*for (int i=0; i < n; i++)*

*for (int j=n; j > i; j--)*

*if (a[j-1] > a[j])*

*swap(a[j-1], a[j]);*

swap() не зависит от размера входного массива и выполняется за O(1). if выполняется за O(1), if и его содержимое - за O(1 + 1) = O(1), хотя мы и не знаем, выполнится содержимое или нет (мы ищем время выполнения в худшем случае). Время выполнения внутреннего цикла - сумма времён выполнения его содержимого, n - i раз, так что O((n - i)\*1) = O(n - i).

Внешний цикл - сумма по i от 1 до (n - 1) n - i = n(n-1)/2 = n2/2 + n/2, итого O(n2). Бывают сортировки с временем выполнения O(n log n), это qsort, который вам надо было реализовать дома в прошлой работе, или heapsort, который в текущей. Быстрее сортировок, не использующих информацию о числах в массиве, не бывает. Бывает сортировка за O(n) - поразрядная, но она использует знания о числах. Для сравнения, натуральный логарифм от миллиона — чуть меньше 14, двоичный от ста тысяч — примерно 17.

Программы с рекурсивными процедурами оценивать несколько интереснее: нужно получить рекуррентное соотношение времён выполнения процедуры. Например, факториал:

*int recFactorial(int a)*

*{*

*if (a <= 1)*

*return 1;*

*else*

*return a \* recFactorial(a - 1);*

*}*

T(n) = c + T(n-1) при n > 1, и d, при n <= 1.

T(n) = c + T(n-1) = 2c + T(n-2) = ... = i\*c + T(n-i) = ... = (n-1)\*c + T(1) = (n-1)\*c+d.

Следовательно, T(n) имеет порядок O(n).

Ещё интересным примером является задача вычисления n-го числа Фибоначчи:

F\_n = F\_n-2 + F\_n-1, F\_0 = 1, F\_1 = 1

F = 1, 1, 2, 3, 5, 8, 13, 21, ...

Рекурсивное решение имеет трудоёмкость 2n:

*long fibonacci(int n)*

*{*

*if (n == 0 || n == 1)*

*return 1;*

*else*

*return fib(n - 2) + fib(n - 1);*

*}*

Итеративное - O(n):

*long fibonacciIterative(int n)*

*{*

*int prev = 1;*

*int curr = 1;*

*for (int i = 2; i <= n; ++i)*

*{*

*int temp = prev + curr;*

*prev = curr;*

*curr = temp;*

*}*

*return curr;*

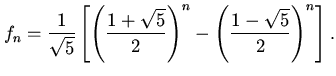
*}*

А ещё можно считать Фибоначчи так:

F\_n+1 F\_n = |1 1| ^n

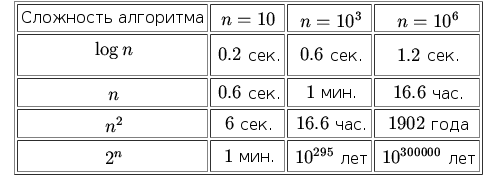
F\_n F\_n-1 |1 0|

Доказательство по индукции, оставим его в качестве упражнения желающим. Еще можно применить яростный матан прямиком из ада и получить формулу для выражения чисел Фибоначчи через золотое сечение (формула Бине):

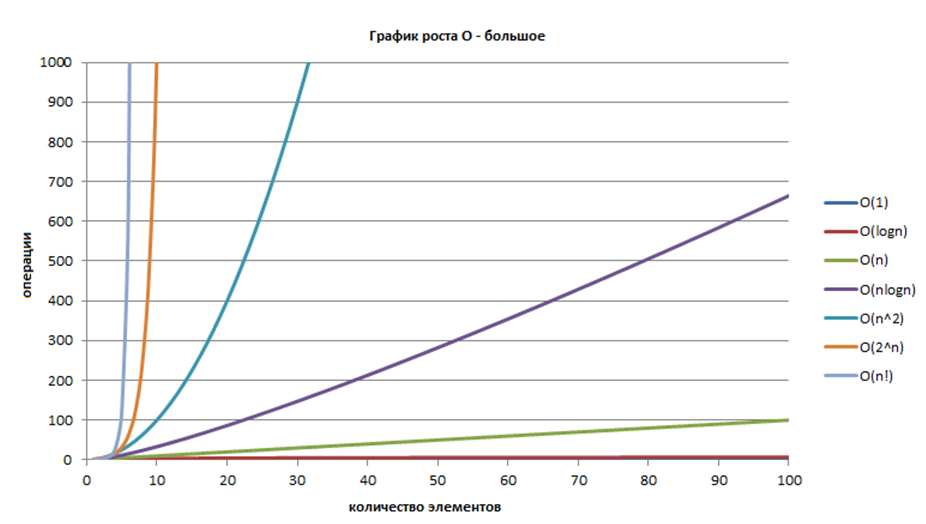


Хоть тут будет и логарифмическая сложность, однако, тут видимо будет более 9000 ошибок округления, так что применение этой формулы для программистов довольно сомнительно.

Рассмотрим четыре алгоритма решения одной и той же задачи, имеющие логарифмическую, линейную, квадратичную и экспоненциальную сложности соответственно. Предположим, что второй из этих алгоритмов требует для своего выполнения на некотором компьютере при значении параметра n=10^3 ровно одну минуту времени. Тогда времена выполнения всех этих четырех алгоритмов на том же компьютере при различных значениях параметра будут примерно такими:



Еще одна показательная картинка:



В следующей таблице приведено сравнение алгоритмов сортировки.



Более интересные примеры (ответы приводятся, доказательство оставляю в качестве упражнения).

for (int i = 1; i\*i <= N; i = i\*4)

sum++;

Тело цикла будет выполнено log4(N1/2) раз. Таким образом, сложность O(log n).

int sum = 0;

for (int i = 1; i <= N; i++)

for (int j = 1; j <= N; j += i)

sum++;

Внутренняя часть циклов выполнится N + N/2 + N/3 + N/4 + ... + 1 ~ N ln N раз. (Используется разложение 1 + 1/2 + 1/3 + ... + 1/N ~ ln N). Собственно, сложность получается O(n\*log(n)).

int sum = 0;

for (int i = 1; i\*i <= N; i = i\*4)

for (int j = 0; j < i; j++)

sum++;

Тело внутреннего цикла выполнится 1 + 4 + 16 + 64 + ... + sqrt(N) ~ 4/3 sqrt(N) раз. Сложность — O(sqrt(n)).