# Сложность алгоритмов: краткое введение

А. Г. Фенстер, http://info.fenster.name

6 февраля 2009 г.

В программировании зачастую бывает важно не просто придумать алгоритм решения некоторой задачи, а ещё и оценить время его работы на различных входных данных. Решение задачи на олимпиаде обычно должно на самом большом тесте работать не больше одной-двух секунд, что часто сильно ограничивает диапазон алгоритмов, которые в этом решении можно применить. Чтобы продемонстрировать, насколько могут отличаться два алгоритма, решающие одну и ту же задачу, рассмотрим классический пример: сортировку элементов массива по возрастанию двумя способами: методом выбора и алгоритмом пирамидальной сортировки, которые будут описаны ниже. В таблице приведено время работы обоих алгоритмов в зависимости от размера массива, при замере использовался процессор с тактовой частотой 2 ГГц.

| Количество элементов | 10 000 | 50 000 | 100 000 | 1 000 000 | 10 000 000 |
|---|---|---|---|---|---|
| Метод выбора | 0 с | 9 с | 34 с | – | – |
| Пирамидальная | 0 с | 0 с | 0 с | 1 с | 13 с |

Этот пример показывает, что если небольшие массивы (до десяти тысяч элементов) ещё можно сортировать простым алгоритмом, то на бóльших массивах необходимо реализовывать более сложные для написания, но быстрее работающие методы. Сейчас наша цель — научиться сравнивать различные алгоритмы по времени их работы.

Рассмотрим для примера процесс поиска минимального элемента в массиве размера $N$[1]. Для этого необходимо пройти в цикле по всем элементам массива и сравнить каждый элемент с минимальным из найденных. Сложно подсчитать точное количество операций, которое будет при

[1]Мы будем нумеровать элементы массива с 0, т. е. массив размера $N$ содержит 0-й, 1-й, …, $(N-1)$-й элементы.

этом выполнено процессором, но понятно, что оно будет прямо пропорционально числу $N$. Говоря более строго, начиная с некоторого $N$ число операций будет не больше, чем $c \cdot N$, где $c$ — положительное число, не зависящее от $N$. В таких случаях мы будем говорить, что алгоритм *выполняет порядка $N$ операций* и будем писать, что время работы алгоритма — $O(N)$ (читается «о-большое от $N$»).

Рассмотрим другой пример. Пусть $A$ — наш массив размера $N$ и нам необходимо подсчитать количество таких пар $(i, j)$, что $i < j$, но $A_i > A_j$ (любую такую пару мы будем называть *инверсией*, в упорядоченном по возрастанию массиве инверсий нет). Для решения задачи мы напишем два цикла: первый `for` будет перебирать все возможные $i$ от 0 до $N-2$, второй (вложенный) `for` переберёт все $j$ от $i+1$ до $N-1$ и проверит пару $(i, j)$. Сколько раз выполнится эта проверка? Для $i = 0$ внутренний цикл отработает $N-1$ раз, для $i = 1$ — $N-2$ раза, и так далее. При $i = N-2$ он сделает всего одну проверку. Считаем общее количество проверок:

$$(N-1) + (N-2) + \ldots + 1 = \frac{1 + (N-1)}{2} \cdot (N-1) = \frac{1}{2}N^2 - \frac{1}{2}N$$

(суммирование произведено по формуле суммы арифметической прогрессии).

При больших $N$ член $\frac{1}{2}N^2$ будет заметно больше, чем $\frac{1}{2}N$, поэтому вторым членом можно в принципе пренебречь. Заметим, что можно найти такое число $c > 0$, что начиная с некоторого $N$ количество операций не будет превышать $c \cdot N^2$, т. е. данный алгоритм выполняет порядка $N^2$ операций, его время работы — $O(N^2)$.

Алгоритмы со временем работы $O(N)$ мы будем называть *линейными*, со временем работы $O(N^2)$ — *квадратичными*, и так далее. Ясно, что начиная с какого-то $N$ линейный алгоритм всегда будет работать быстрее квадратичного, а квадратичный — быстрее кубического. При этом нам не так важно точное количество операций, которое будет сделано, важно лишь, как оно зависит от числа $N$ — размера задачи. Такой подход даёт нам способ сравнивать время работы различных алгоритмов, чем мы и будем далее часто заниматься.