Лекция по АиСД №2

Попов Никита

14 января 2016 г.

Задача сортировки

Вход: последовательность чисел (строго говоря, может быть что угодно с полным порядком) $a_1 \dots a_n$
Выход: $a_{i_1} \dots a_{i_n}$, где

Рассмотрим неэффективный алгоритм:

```
recursive_sort(a)                    // (a = [a_1\ldots a_n])
    n = |a|
    if n > 1 then
        recursive_sort(a[1:n-1])
        k := a_n
        for i:= n-1 downto 1 do
            if a_i > k then
                a_{i+1} = a_i
            else
                break
                a_{i+1}:= k
```

$[6,8,3,4] \to [3,6,8,4] \xrightarrow{8>4} [3,6,\ ,8] \xrightarrow{6>4} [3,\ ,6,8] \xrightarrow{3<4} [3,4,6,8]$

По сути, мы идём слева направо и каждому элементу находим место среди прошлых уже отсортированных элементов

```
insertion_sort(a)
    n = |a|
    for j := 2 to n
        k:= a_n
        for i:= j-1 downto 1 do
            if a_i > k then
                a_{i+1} = a_i
            else
                break
```

Докажем корректность алгоритма формально. Для этого найдём *инвариант*.

Инвариант: в начале каждой итерации цикла по j массив с 1 по $j-1$ индекс уже отсортирован; при этом он состоит из тех же элементов, что и раньше.

Если это условие выполняется, то после выполнения алгоритма, весь массив (с 1-го по n-ый индексы) будет отсортирован.

Докажем по индукции:

База: $j = 2$ — $a[1:1]$ отсортирован

Переход Всё до j-го отсортировано; Поставим a_j на нужное место

Насколько эффективно он работает? Понятно, что это зависит от входных данных. Ясно, что чем больше элементов, тем дольше он работает. Понятно также, что если массив уже отсортирован, то работать он будет быстрее.

$$T(n) \text{ — время работы на входе длины } n \text{ в худшем случае.} \tag{1}$$

$$\text{в среднем случае.} \tag{2}$$

$$\text{в лучшем случае.} \tag{3}$$

Однако оценка в лучшем случае, вообще говоря, бесполезна. Ведь любой алгоритм можно модифицировать так, чтобы в каком-то случае он работал очень быстро.

Асимптотический анализ: как меняется $T(n)$ при $n \to \infty$?

Для этого обычно применяют O-нотацию или Θ-нотацию.

$\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2 > 0 : \forall n \geqslant n_0 \implies 0 \leqslant c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)\}$

$f(n) = 3n^2 + 2n - 6$

$f(n) \in \Theta(n^2)$

Асимптотика — это хорошо, но на константы тоже стоит обращать внимание: для маленьких n вполне может быть, что n^3 работает быстрее, чем n.

Оценим худший случай нашего алгоритма:

$T(n) = \sum\limits_{j=2}^{n} \sum\limits_{i=j-1}^{1} \Theta(1) = \sum\limits_{j=2}^{n} \Theta(j) = \Theta(n^2)$

Средний случай:

Предположим, что все входы равновероятны. Тогда будет выполняться примерно половина сравнений.

$T(n) = \sum \Theta(\frac{j}{2}) = \theta(n^2)$

Лучший случай:

$\Theta(n)$.

Рассмотрим другой алгоритм — сортировку слиянием.

```
merge_sort(a)
    n:= |a|
    b_1 := merge_sort(a[1:\frac{n}{2}])
    b_2 := merge_sort(a[\frac{n}{2}+1:n])
return merge(b_1, b_2)
```

Рассмотрим, как может работать merge.

$[2, 5, 6, 8]; [1, 3, 7, 9]$

Будем рассматривать элементы из массивов: $2 > 1$ — первый элемент — 1; $2 < 3$ — второй элемент — 2...

Очевидно, что алгоритм корректен, а его сложность — $\Theta(n_1 + n_2)$.

Пусть худшее время для $merge\_sort$ — $T(n)$. Тогда

$$T(n) = \begin{cases} \Theta(1), n = 1 \\ 2T(n/2) + \Theta(n) \end{cases}$$

Построим дерево рекурсии:

На каждом уровне cn работы, а высота дерева — $\log_2 n$. Общее время работы — $n\Theta(1) + cn \log n$.