Алгоритмы и структуры данных, лекция 2

14.01.2016

Задача сортировки

Вход: последовательность чисел (строго говоря, может быть что угодно с полным порядком) $(a_1, a_2, \ldots, a_n)$.

Выход: $(a_{i_1}, a_{i_2}, \ldots, a_{i_n})$, где $a_{i_k} \leq a_{i_{k+1}}$. Другими словами, на выходе получается отсортированная по возрастанию последовательность.

Рассмотрим неэффективный алгоритм:

Algorithm 1 Неэффективный алгоритм сортировки

```
1:  function RECURSIVE_SORT(a)                          ▷ a = (a_1, a_2, ..., a_n)
2:      n := |a|
3:      if n > 1 then
4:          RECURSIVE_SORT(a[1 : n − 1])
5:          k := a_n
6:          for i := n − 1 downto 1 do
7:              if a_i > k then
8:                  a_{i+1} := a_i
9:              else
10:                 break
11:             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.

Теперь рассмотрим алгоритм *сортировки вставками*.

Algorithm 2 Алгоритм сортировки вставками

```
1:  function RECURSIVE_SORT(a)                          ▷ a = (a_1, a_2, ..., a_n)
2:      n := |a|
3:      for j := 2 to n do
4:          k := a_j
5:          for i := j − 1 downto 1 do
6:              if a_i > k then
7:                  a_{i+1} = a_i
8:              else
9:                  break
10:             a_{i+1} := k
```

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.

Доказательство. По индукции:

База: $j = 2$ — $a[1:1]$ отсортирован

Переход Всё до j-го отсортировано; Поставим a_j на нужное место. Тогда полученный массив также будет отсортирован. □

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.

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

$$\text{в среднем случае.} \tag{2}$$

$$\text{в лучшем случае.} \tag{3}$$

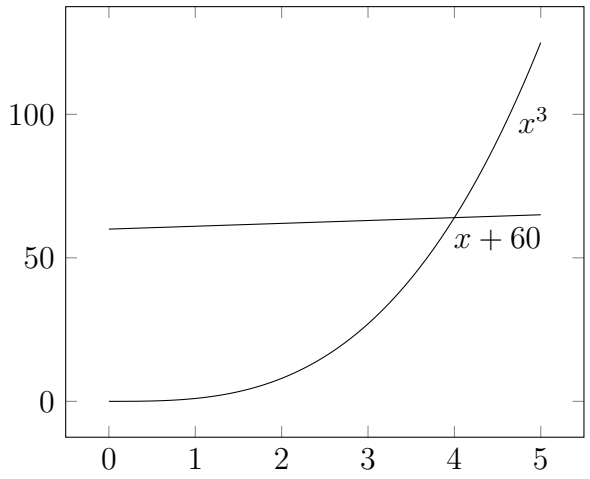
Однако оценка в лучшем случае, вообще говоря, бесполезна. Ведь любой алгоритм можно модифицировать так, чтобы в каком-то случае он работал очень быстро.

Асимптотический анализ: как меняется $T(n)$ при $n \to \infty$? Для исследования этого обычно применяют O-нотацию или Θ-нотацию.

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

Например, пусть задана функция $f(n) = 3n^2 + 2n - 6$. Тогда $f(n) \in \Theta(n^2)$.

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 (когда на каждом шагу приходится совершать максимальное число перемещений): $T(n) = \sum\limits_{j=2}^{n} \sum\limits_{i=j-1}^{1} \Theta(1) = \sum\limits_{j=2}^{n} \Theta(j) = \Theta(n^2)$

Средний случай: Предположим, что все входы равновероятны. Тогда будет выполняться примерно половина сравнений и $T(n) = \sum \Theta(\frac{j}{2}) = \Theta(n^2)$

Лучший случай — это случай, когда массив уже отсортирован. Тогда $T(n) = \Theta(n)$.

Рассмотрим другой алгоритм — *сортировку слиянием.*

Algorithm 3 Алгоритм сортировки слиянием

```
1: function MERGE_SORT(a)                                   ▷ a = (a_1, a_2, ..., a_n)
2:     n := |a|
3:     if n > 1 then
4:         b_1 := MERGE_SORT(a[1 : n/2])
5:         b_2 := MERGE_SORT(a[n/2 + 1 : n])
6:         a := MERGE(b_1, b_2)                  ▷ сливаем два отсортированных массива в один
7:     return a
```

Рассмотрим, как может работать $\text{MERGE}(b_1, b_2)$ на примере. Пусть даны массивы $b_1 := [2, 5, 6, 8]$ и $b_2 := [1, 3, 7, 9]$.

Будем сливать элементы из массивов в результирующий массив b, сравнивая поочерёдно минимальные элементы, которые ещё не вошли в результирующий массив.

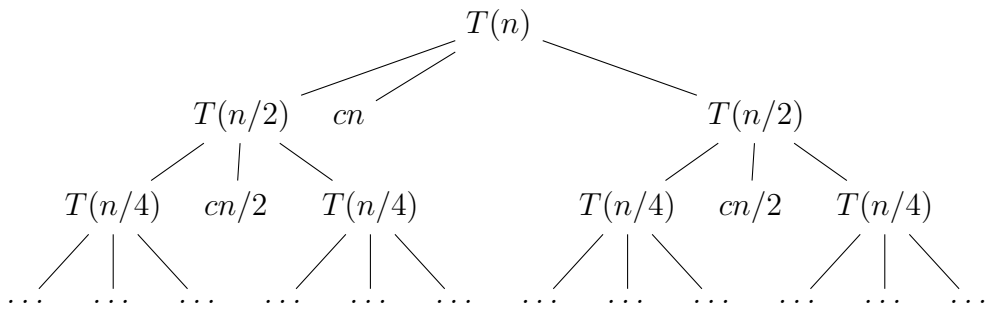
- $2 > 1$. Тогда $b[1] := b_2[1] = 1$ (будем считать, что нумерация идёт с единицы).
- $2 < 3$. Тогда $b[2] := b_1[1] = 2$.
- Аналогично продолжаем для всех остальных элементов массивов.

Очевидно, что алгоритм корректен, а его сложность — линейная, так как мы один раз проходим по массивам, то есть $\Theta(n)$.

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 — $n\Theta(1) + cn \log n = \Theta(n \log n)$.