Лекция 2 от 14.01.2016

Задача сортировки. Сортировка вставками

На сегодняшней лекции мы будем рассматривать классическую задачу мира алгоритмов, встречающуюся на практике повсеместно — *задачу сортировки*. Поставим её формально.

Вход: последовательность из n чисел $[a_1, a_2, \ldots, a_n]$.

Выход: последовательность из n чисел $[a'_1, a'_2, \ldots, a'_n]$ такая, что $a'_1 \leqslant a'_2 \leqslant \ldots \leqslant a'_n$.

Как её решить? Один из самых простых алгоритмов сортировки устроен рекурсивно. Пусть на вход подаётся массив из n элементов и последний элемент в нём равен k. Тогда вызываем этот же алгоритм от первых $n-1$ элемента, после чего делаем следующее: до тех пор, пока элемент, стоящий перед k, больше k, меняем их местами. В псевдокоде это будет записано так:

Algorithm 1 Неэффективный алгоритм сортировки

```
1: function RECURSIVE-SORT(A)
2:     if A.length > 1 then
3:         RECURSIVE-SORT(A[1 : A.length − 1])
4:         key := A[j]
5:         i := j − 1
6:         while i > 0 and A[i] > key do
7:             A[i + 1] := A[i]
8:             i := i − 1
9:         A[i + 1] := key
```

Покажем, как работает этот алгоритм, на примере массива $[6, 8, 3, 4]$:

- Сначала вызовем этот алгоритм от массива $[6, 8, 3]$. Легко проверить, что в итоге получится массив $[3, 6, 8]$.
- Далее сравниваем 4 с стоящим впереди элементом и меняем их местами, если необходимо:

$$[3, 6, 8, 4] \mapsto [3, 6, 4, 8] \mapsto [3, 4, 6, 8]$$

В итоге получилась отсортированная последовательность. По сути, алгоритм вставляет элементы на нужные места в уже отсортированный массив.

Теперь рассмотрим алгоритм *сортировки вставками*:

Algorithm 2 Алгоритм сортировки вставками

```
1: function INSERTION-SORT(A)
2:     for j := 2 to A.length do          ▷ вставка A[j] в отсортированный массив A[1..j − 1]
3:         key := A[j]
4:         i := j − 1
5:         while i > 0 and A[i] > key do
6:             A[i + 1] := A[i]
7:             i := i − 1
8:         A[i + 1] := key
```

Нетрудно заметить, что он делает то же самое, что и описанный ранее алгоритм, только без рекурсивных вызовов.

Возникает логичный вопрос — а можем ли мы сказать, что этот алгоритм делает именно то, что нам нужно? Другими словами, *корректен* ли он? Оказывается, что да. Докажем это.

Для этого стоит рассмотреть так называемый *инвариант цикла* — нечто, что не изменяется при переходе на следующую итерацию цикла. В данном случае он будет таков: на j-й итерации цикла первые $i-1$ элементов массива будут упорядочены.

Далее нужно рассмотреть три свойства инварианта цикла:

- **Инициализация:** Инвариант цикла выполняется на первой итерации;
- **Переход:** Инвариант сохраняется при переходе на следующую итерацию;
- **Завершение:** При окончании цикла инвариант даёт какое-то ценное свойство полученного массива.

Можно сказать, что первые два свойства доказывают корректность инварианта. Приступим:

- **Инициализация:** На первой итерации цикла элемент вставляется в массив из одного элемента. Но он, очевидно, отсортирован. Тогда инвариант цикла выполняется для первой итерации.
- **Переход:** Пусть массив $[a_1, \ldots, a_{j-1}]$ уже отсортирован. В теле цикла делается следующее: до тех пор, пока a_j меньше стоящего перед ним элемента, они меняются местами. По сути, элемент просто вставляется на нужное место. Тогда инвариант выполняется при переходе на следующую итерацию. Следовательно, инвариант корректен.
- **Завершение:** Цикл завершается тогда, когда $j = n+1$. Заметим, что это будет итерация под номером n. Но тогда (согласно инварианту цикла) первые n элементов массива, то есть весь массив, будут упорядочены. Тогда в конце массив окажется отсортированным. Тем самым мы доказали, что алгоритм корректен.

Хорошо, он корректен. Но насколько быстро он работает? Достаточно очевидны три факта:

- Скорость работы зависит от входных данных;
- Чем больше элементов, тем дольше будет работать алгоритм;
- Если алгоритм уже отсортирован, то он будет работать быстрее.

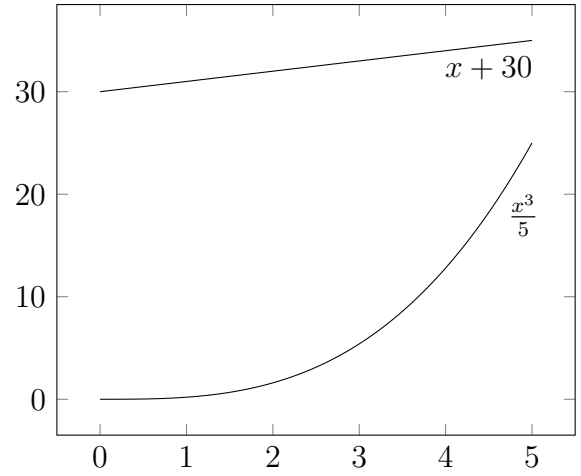
Обычно время работы рассматривают в трёх случаях — в худшем, в лучшем и в среднем. Хотя оценка в лучшем случае не несёт особо ценной информации, так как всегда можно модифицировать алгоритм так, что на некоторых наборах данных он будет работать очень быстро.

Для оценки времени работы введём асимптотическую оценку Θ:

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

Рассмотрим следующий пример. Пусть $f(n) = 3n^2 + 2n + 6$. Тогда $f(n) \in \Theta(n^2)$. Хотя правильней писать так, но на практике все пишут, что $f(n) = \Theta(n^2)$.

Стоит обратить внимание, что асимтотическая оценка показывает рост функции. Но не стоит забывать про константы — может выйти так, что для достаточно маленьких n функция из $\Theta(n^3)$ работает быстрее, чем функция из $\Theta(n)$:



Вернёмся к оценке времени работы алгоритма. Пусть его скорость работы равна $T(n)$.

Рассмотрим худший случай. Когда он достигается? Когда приходится делать максимальное число перестановок. Тогда на i-й итерации совершается $\Theta(i)$ операций. Следовательно,

$$T(n) = \sum_{k=0}^{n} \Theta(k) = \Theta(n^2).$$

В среднем случае все исходы равновероятны. Тогда в среднем будет выполняться половина всех возможных перестановок. А это $\Theta(n^2)$.

В лучшем случае не совершается ни одной перестановки. Это означает, что массив уже отсортирован. Тогда алгоритм работает за $\Theta(n)$ (так как он просто проходится по массиву).

Сортировка слиянием

Рассмотрим другой алгоритм — *сортировку слиянием*.

Algorithm 3 Алгоритм сортировки слиянием

```
1: function MERGE-SORT(A)
2:     n := A.length
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 $\textsc{Merge}(B_1, B_2)$ 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.

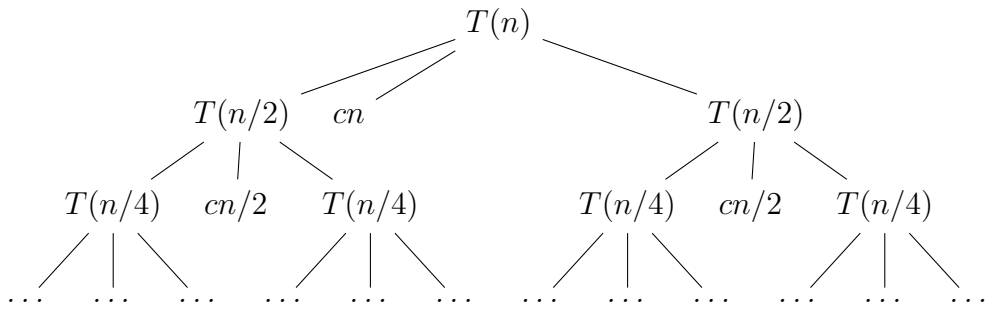
- $2 > 1$. Тогда $b[1] := b_2[1] = 1$ (будем считать, что нумерация идёт с единицы).
- $2 < 3$. Тогда $b[2] := b_1[1] = 2$.
- Аналогично продолжаем для всех остальных элементов массивов.

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

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

Построим дерево рекурсии:



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