Лекция 3 от 19.01.2016

Нотация

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

Θ — *асимптотическое* $=$. Например, $2n = \Theta(n)$. По определению, $c_1 n \leqslant 2n \leqslant c_2 n$. Тогда $c_1 = 1, c_2 = 2$.

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

O — *асимптотическое* $\leqslant$. Например, по этому определению $n = O(n \log n)$, так как при достаточно больших n_0 $\log n > 1$. Тогда $c_2 = 1$.

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

Ω — *асимптотическое* $\geqslant$. Например, $n \log n = \Omega(n \log n)$ и $n \log n = \Omega(n)$. В обоих случаях подходит $c_1 = 1$.

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

o — *асимптотическое* $<$. Например, $n = o(n \log n)$. Покажем это. Пусть $n < c_2 n \log n \iff 1 < c_2 \log n \iff n > 2^{1/c_2}$. Тогда $n_0 = [2^{1/c_2} + 1]$

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

ω — *асимптотическое* $>$. Например, нельзя сказать, что $n \log n = \omega(n \log n)$. Но можно сказать, что $n \log n = \omega(n)$.

Когда мы пишем такую нотацию, мы подразумеваем функции, а не числа. Если же указывать функции явно, то это можно сделать с помощью λ-нотации:

$$\lambda n.n \in o(\lambda n.n \log_2 n)$$

Примечание: данная нотация очень похожа на лямбда-функции в Python:

$$\texttt{lambda x: x * x} \iff \lambda x.x^2$$

Заметим, что в логарифмах можно свободно менять основание: $\log_c n = \frac{\log_2 n}{\log_2 c}$. Именно поэтому не пишут основание логарифма.

Разделяй и властвуй. Быстрая сортировка

Ход действий при алгоритме **"разделяй и властвуй"**:

1. Разбить задачу на подзадачи.

2. Каждую подзадачу решить рекурсивно.

3. Объединяем решения подзадач некоторым образом.

Этот алгоритм даст решение общей задачи.

Вернёмся к *сортировке слиянием.* Алгоритм состоит из трёх шагов:

1. Раделить массив напополам — $\Theta(1)$
2. Рекурсивно решить подзадачи — $2T(\frac{n}{2})$
3. Слияние уже отсортированных массивов — $\Theta(n)$

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \implies \Theta(n \log n)$$

Задача та же — отсортировать массив.

Воспользуемся методом “Разделяй и властвуй”. Разобьем по-другому:

Выберем в массиве опорный элемент x (как угодно). Выбор важен, от него может много зависить. Пройдем по всем элементам и запишем те элементы, что меньше x до него, а те, что больше — после.

Две подзадачи: сортировка двух подмассивов.

Третий шаг — соединить их.

Algorithm 1 Разбитие массива на подмассивы

function PARTITION(a, p, q) $\triangleright$ a — массив, p и q — индексы начала и конца соотвественно
    $i := p$
    for $j := p + 1$ to q **do**
        if $a[j] < a[p]$ **then**
            $i := i + 1$
            SWAP$(a[i], a[j])$
    return i

Рассмотрим работу алгоритма на примере массива $\{6, 3, 8, 7, 5,1\}$:

1. $j = 1$. Так как $6 > 3$, то запускается тело цикла. Тогда $i = 1$ и 3 остаётся на месте.
2. $j = 2$. Так как $6 < 8$, то ничего не изменяется.
3. $j = 3$. Так как $6 < 7$, то ничего не изменяется.
4. $j = 4$. Так как $6 > 5$, то запускается тело цикла. Тогда $i = 2$ и числа 5 и 8 меняются местами.
$$\begin{array}{|c|c|c|c|c|c|}\hline 6 & 3 & \mathbf{8} & 7 & \mathbf{5} & 1 \\ \hline\end{array} \longrightarrow \begin{array}{|c|c|c|c|c|c|}\hline 6 & 3 & \mathbf{5} & 7 & \mathbf{8} & 1 \\ \hline\end{array}$$
5. $j = 5$. Так как $6 > 1$, то запускается тело цикла. Тогда $i = 3$ и 7 и 1 меняются местами.
$$\begin{array}{|c|c|c|c|c|c|}\hline 6 & 3 & 5 & \mathbf{7} & 8 & \mathbf{1} \\ \hline\end{array} \longrightarrow \begin{array}{|c|c|c|c|c|c|}\hline 6 & 3 & 5 & \mathbf{1} & 8 & \mathbf{7} \\ \hline\end{array}$$
6. Последний шаг — переставить опорный элемент на место i:
$$\begin{array}{|c|c|c|c|c|c|}\hline \mathbf{6} & 3 & 5 & \mathbf{1} & 8 & 7 \\ \hline\end{array} \longrightarrow \begin{array}{|c|c|c|c|c|c|}\hline \mathbf{1} & 3 & 5 & \mathbf{6} & 8 & 7 \\ \hline\end{array}$$

Теперь рассмотрим скорость работы алгоритма.

1. Разбить задачу на подзадачи — $\Theta(n)$
2. Рекурсивно решить подзадачи. Пусть индекс опорного элемента равен r. Тогда на выполнение уйдёт $T(r - 1) + T(n - r)$.
3. Объединить решения задач в одно глобальное — 0 (уже сделано).

Тогда скорость работы алгоритма задаётся следующим рекуррентным соотношением:

$$T(n) = T(r-1) + T(n-r) + \Theta(n)$$

Рассмотрим возможные случаи:

1. **Оптимальный вариант** — r всегда посередине:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \implies T(n) = \Theta(n \log n)$$

2. **Худший случай** — r всегда минимален/максимален (массив уже «почти» отсортирован):

$$T(n) = T(n-1) + \Theta(n) \implies T(n) = \Theta(n^2)$$

3. **Средний случай.** Пусть каждый раз обе части не меньше четверти.

   Подзадачей типа j называется задача такая, что размер входного массива n' соответсвтует следующему неравенству:

$$n\left(\frac{3}{4}\right)^{j+1} < n' \leqslant n\left(\frac{3}{4}\right)^{j}$$

   Не считая рекурсии, на каждую подзадачу типа j уходит $O\left(\left(\frac{3}{4}\right)^j n\right)$.

   Стоит заметить, что подзадачи типа j не пересекаются по разбиению. При этом из них получаются подзадачи типа не меньше $j+1$.

   При этом количество подзадач типа j не больше $\left(\frac{4}{3}\right)^{j+1}$. Отсюда получаем, что на все подзадачи типа j нужно $O\left(\left(\frac{4}{3}\right)^{j+1}\left(\frac{3}{4}\right)^j n\right) = O(n)$.

   Так как максимальный тип подзадачи можно ограничить сверху $\log_{4/3} n$, то оценка работы в среднем случае равна $O(n \log n)$. При условии, что «везёт» всегда.

Как обеспечить везение?

Мы хотим, чтобы опорный элемент был близок к середине (в отсортированном массиве). Условно, в пределах средних двух четвертей. Если выбирать наугад, вероятность 50%.

Предположим, что мы выбираем случайный элемент. Распределим все прочие, и если одна из частей меньшь четверти, забудем про этот элемент и выберем другой. Повторим, пока не получим хороший элемент. В среднем на это уйдёт две попытки $\left(\frac{1}{P}\right)$. На сложности алгоритма это не сказывается никак, т.к. меняется только константа. Зато теперь так не только в лучшем, но и в среднем случае.