Лекция по АиСД №5

26.01.2016

1 QSort. Продолжение

Говоря об алгоритме сортировки QSort, мы рассматривали только случаи, когда все элементы различны, однако, если есть равные, алгоритм может сломаться. Составим другой алгоритм Partition, чтобы решить эту проблему. Попытаемся преобразовывать таки образом, чтобы в левой части стояли элементы строго меньшие опорного, в правой — строго большие, а в середине — равные опорному:

$< x$	$= x$	$> x$

$< i$ — $< x$
$[i, j)$ — $> x$
$[k, n]$ — $= x$

Обозначим за опорный элемент последний.
Будем проходиться по массиву от начала до конца, выставляя элементы в нужном порядке:

$< x$	$> x$	?	$= x$	x
	i	j	k	n

(? — ещё не просмотренные элементы)

Algorithm 1 Модифицированный алгоритм Partition

```
1:  function PARTITION(a)
2:      i := 1
3:      j := 1
4:      k := 1
5:      while j < k do
6:          if a[j] = a[n] then
7:              k := k − 1
8:              a[j], a[k] := a[k], a[j]
9:          else
10:             if a[j] < a[n] then
11:                 a[i], a[j] := a[j], a[i]
12:                 j := j + 1
13:                 i := i + 1
14:             else
15:                 j := j + 1
```

На выходе получаем:

$< x$	$> x$	$= x$	
	i	k	n
		j	

Заметим, что $j = k$.

Остаётся только переставить части массива:

```
1: while i < k and j ⩽ n do
2:     a[i], a[j] := a[j], a[i]
3:     i := i + 1
4:     j := j + 1
```

Самая быстрая из наших сортировок — $n \log n$. А можно ли быстрее?

На основе только сравнений — нет.

Использовать разобранные нами сортировки можно на любых сущностях, для которых определена опрерация сравнения.

Предположим теперь, что мы сортируем именно числа, причём Натуральные и не превосходящие некоторого $C = const$.

Создадим Массив b размера C, заполненный нулями. Будем проходить по исходному массиву a и на каждом шаге будем добавлять 1 к соответствующему элементу массива b ($b[a[i]] := b[a[i]] + 1$) Потом, проходя по получившемуся массиву b будем восстанавливать исходный массив уже в отсортированном виде.

Такая сортировка будет работать за $O(n)$, однако, она не универсальна.

Вернёмся к универсальным сортировкам. Возьмём $n = 3$:

картинка из тетради

Подобное дерево можно составить для любого детерменированного[1] алгоритма соритровки, зафиксировав n. Сложность алгоритма будет являться высота h дерева. Посчитаем это h:
$\#$ листьев $\geqslant n!$
$\#$ листьев $\leqslant 2^h$
$2^n \geqslant n!$
$h \geqslant \log_2 n! \geqslant \log_2 \frac{n}{2}^{\frac{n}{2}} = \frac{n}{2} \log_2 \frac{n}{2} = \Omega(n \log n)$ n элементов; не меньше $n!$ листьев.
$\Rightarrow$ сортировать любой массив сравнением меньше чем за $n \log n$ нельзя

Поиск медианы

Медиана — такой элемент массива, что не меньше половины элементов меньше неё, и не меньше половины — больше.

Для отсортированного массива размера n медиана будет находиться под номером $\dfrac{n+1}{2}$ для нечётных n и $\dfrac{n}{2}$ для чётных n. Пример: для массива (8, 1, 3, 5, 6, 9) медианой будет являться 5.

Как же найти медиану? Очевидно, что можно отсортировать и взять средний — $\Theta(n)$.

А можно ли найти медиану ли за линейное время? Можно. Напишем алгоритм, находящий элемент, стоящий на k-ом месте в массиве, получающемся из входного после сортировки. Это называется поиском k-ой порядковой статистики. Составим этот алгоритм, немного модифицировав QSort:

[1] Детерминированный алгоритм — алгоритмический процесс, который выдаёт предопределённый результат для заданных входных данных. Например, QSort, выбирающий опорный элемент случайным образом, не является детерминированным.

Algorithm 2 Поиск k-ой порядковой статистики

```
1: function SELECT(a, k)
2:     choose pivot a[p]
3:     i := PARTITION(a, p)
4:     if i := k then
5:         return a[i]
6:     if i > k then
7:         return SELECT(a[1 ... i − 1], k)
8:     else
9:         return SELECT(a[i + 1 ... n], k − i)
```

Как и в быстрой сортировке, неправильно выбранный опорный элемент портит скорость до n^2. Будем выбирать опорный элемент случайным образом.

Попробуем посчитать время работы в среднем случае.

j-подзадача размера n'.

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

Как и в QSort, в среднем мы потратим две попытки на переход к следующему j.

Максимальное j — $O(\log_{\frac{4}{3}} n)$

$$T(n) \leqslant \sum_{j=0}^{\log_{\frac{4}{3}} n} 2 \cdot c \cdot \left(\frac{3}{4}\right)^{j} n = 2cn \sum_{j=0}^{\log_{\frac{4}{3}} n} \left(\frac{3}{4}\right)^{j} \leqslant 2cn$$

Время работы алгоритма в худшем случае всё ещё $O(n^2)$. Худший случай — когда на каждом шаге мы отщеплем всего один элемент. Для достижения лучшего случая, на каждом шаге нужно выбирать в качестве опорного элемента медиану.

Медиана медиан

Попробуем несколько модифицировать наш алгоритм. Разобьём входной массив на группы по 5 элементов. Отсортируем каждую такую группу. Так как размер каждой группы зафиксирован, время сортировки не зависит от n. Зависит только количество сортировок. Возьмём медиану в каждой группе и применим алгоритм нахождения медианы к получившемуся массиву медиан. Выберем её в качестве опорного элемента.

Algorithm 3 Поиск k-ой порядковой статистики 2

```
1: function SELECT(a, k)
2:     Divide a into groups of 5
3:     Choose medians m_1, ... m_{n/5}
4:     x = SELECT([m_1, ..., m_{n/5}], n/10)
5:     choose x as pivot a[p]
6:     i := PARTITION(a, p)
7:     if i := k then
8:         return a[i]
9:     if i > k then
10:        return SELECT(a[1 ... i − 1], k)
11:    else
12:        return SELECT(a[i + 1 ... n], k − i)
```

$T(n) \leqslant cn + T\left(\frac{n}{5}\right) + T\left(\frac{7}{10}n\right)$

$T(n) \leqslant ln$ для некоторого l

$T(n) \leqslant cn + T(\frac{n}{5}) + T(\frac{7}{10}) \leqslant cn + \frac{ln}{5} + \frac{7}{10}ln$