Основные алгоритмы, комментарии по четвертому листку (Сортировки I)

Шибаев Иннокентий

March 2, 2021

1 По поводу теории

1.1 QuickSort

Алгоритм *QuickSort* основан на процедуре *Partition*, которая разделяет массив на два используя какой-то элемент массива в качестве опорного. Делать это можно различными способами, есть метод с двумя итераторами идущими с начала где в качестве опорного используется обычно последний элемент (то что разбиралась на лекции), после данной процедуры опорный элемент становится на место которое бы он занимал в отсортированном массиве. Этот вариант называется разбиением Ломуто. Другой вариант, разбиение Хоара это вариант с двумя указателями идущими с разных концов массива и меняющих местами элементы упорядоченные неправильно относительно опорного. Приведем алгоритм разбиения Ломуто

Алгоритм 1: Алгоритм разбиения Ломуто (Lomuto partition)

Input : Массив A; индексы $left, right:\ 1 \leqslant left \leqslant right \leqslant |A|$ – левый и правый концы отрезка в этом массиве

Output: Позицию разделителя $p:\ left \leqslant p \leqslant right$, меняет массив A так что если $left \leqslant i < p$ то $A[i] \leqslant A[p]$, а если $p < i \leqslant right$ то $A[p] < A[i]$

Function *Partition(A,left,right)* **:**
- $pivot := A[right]$
- $i := left$
- **for** j := *left to right* **do**
  - **if** $A[j] < pivot$ **then**
    - *swap* $A[i]$ *with* $A[j]$
    - $i := i + 1$
- *swap* $A[i]$ *with* $A[right]$
- **return** i

Тогда алгоритм быстрой сортировки *QuickSort* задается через алгоритм *Partition* следующим образом

Алгоритм 2: Алгоритм быстрой сортировки ($QuickSort$)

Input : Массив A; индексы $left, right$: $1 \leqslant left \leqslant right \leqslant |A|$ – левый и правый концы отрезка массива который надо отсортировать

Output: Отсортированный массив A

Function *QSort(A,left,right)* **:**
  if *left < right* **then**
    $p = Partition(A, left, right)$
    $QSort(A, left, p-1)$
    $QSort(A, p+1, right)$

В худшем случае этот алгоритм работает за $O(n^2)$. Однако можно рассматривать скорость работы "в среднем" (фактически усреднить время работы по всем расстановкам) и вот такая оценка "в среднем" будет иметь вид $O(n \log n)$, как у $MergeSort$.

1.2 Алгоритм поиска k-й порядковой статистики

Сам алгоритм обсуждался на лекции, вкратце (в том смысле что мы не расписываем здесь отдельно разделение на пятерки, поиск медиан в них и т.д.) он приведен ниже

Алгоритм 3: Алгоритм поиска k-й порядковой статистики

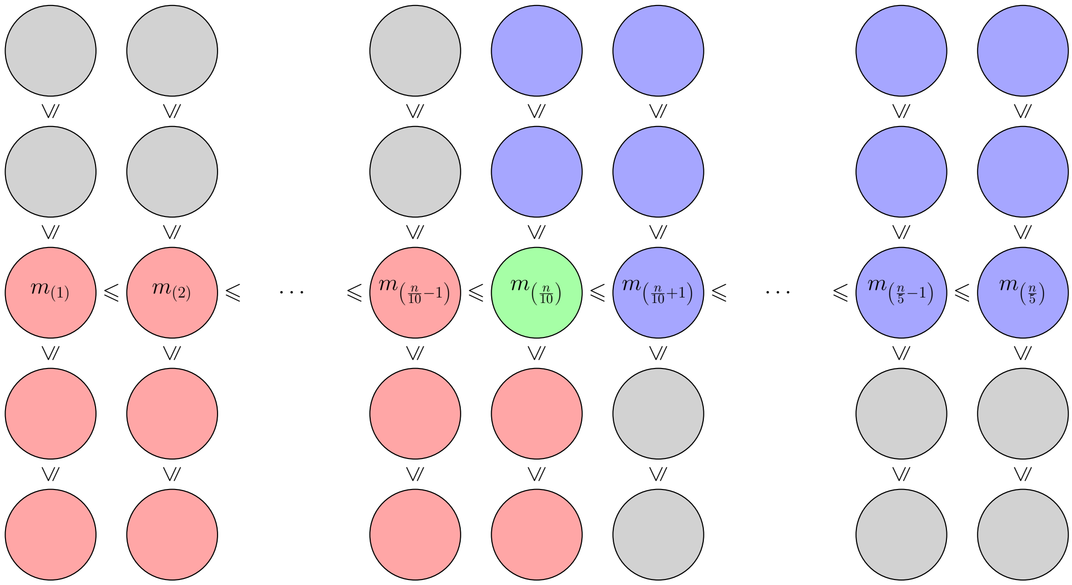
Input : Массив A; индексы k : $1 \leqslant k \leqslant |A|$ – номер искомой статистики

Output: k-я порядковая статистика в массиве A

Function *kth_ element(A,k)* **:**
  $n := |A|$
  if n < *5* **then**
    Отсортировать элементы A (за константу)
    return *A[k]*
  Разделить массив на $\frac{n}{5}$ пятерок
  Для i-й пятерки найти медиану m_i
  Сделать массив $A' = [m_1, \ldots, m_{n/5}]$
  Найти медиану медиан, т.е. $\frac{n}{10}$-ю порядковую статистику массива A' медиан $m_{(n/10)} = kth\_element\left(A', \frac{n}{10}\right)$
  Применить $Partition$ по медиане медиан, т.е. найти ее в массиве A, переставить ее в конец массива A (swap двух элементов) и вызвать $p = Partition(A, 1, n)$
  if $k < p$ **then**
    $L = [A[1], \ldots, A[p-1]]$
    return *kth_ element(L, k)*
  else if $k == p$ **then**
    return *A[p]*
  else
    $R = [A[p+1], \ldots, A[n]]$
    return *kth_ element(R,k-p)*

Для этого алгоритма рекуррента имеет вид $T(n) \leqslant T\left(\frac{n}{5}\right) + \max\{T(L), T(R)\} + cn$. Первое слагаемое в ней берется из вызова алгоритма для поиска медианы медиан в массиве медиан пятерок (который в 5 раз меньше исходного). Второе слагаемое это оценка сверху – мы не знаем в какой именно подмассив пойдет алгоритм, это зависит от k поэтому мы оцениваем самым сложным из них. Остальные шаги в алгоритме либо линейны либо требуют константное число операций, поэтому последнее слагаемое имеет вид cn.

Чтобы оценить асимптотику алгоритма осталось разобраться со слагаемым $\max\{T(L), T(R)\}$. Для этого рассмотрим множество пятерок (считая что n делится на 10 для простоты) уже упорядоченных внутри и дополнительно упорядочим их по возрастанию медиан:



Красным здесь отмечены вершины которые заведомо меньше либо равны медианы медиан (зеленой). Аналогично, синим отмечены вершины которые больше либо равны ей. Как можно понять, множество L элементов что пойдут в левую часть массива, тех элементов что меньше либо равны медиане медиан **не** не включает в себя синие элементы. Оно точно включает красные, про серые же мы ничего сказать не можем – они несравнимы с зеленым (нельзя перейти по неравенствам в них из зеленой вершины). Таким образом выполняются два соотношения

$$\frac{3n}{10} \leqslant |L| \leqslant \frac{7n}{10}; \ \frac{3n}{10} \leqslant |R| \leqslant \frac{7n}{10}$$

так как мы в формуле должны оценить максимум из двух нас интересует оценка сверху, таким образом

$$\max\{T(L), T(R)\} \leqslant T\left(\frac{7n}{10}\right).$$

На самом деле мы уже пару не очень аккуратных переходов сделали, во-первых при подсчете красных и синих элементов надо быть аккуратнее, во-вторых в последнем неравенстве мы как бы пользуемся монотонностью, а она не очевидна, можно это все решить оценив сверху как $T\left(\frac{7.5n}{10}\right)$ к примеру но мы не будем на этом заострять внимание так как на результат это не повлияет. Так или иначе, мы переходим к рекурренте вида

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

Заметим, что $f(n) = cn$ – линейная, и на каждом уровне у нас суммарный "размер массива" уменяшается в геометрической прогрессии т.к. $\frac{1}{5} + \frac{7}{10} = \frac{9}{10} < 1$, поэтому возникает предположение что $T(n) = O(n)$. Давайте проверим это:

$$T(n) \leqslant T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + cn \leqslant d\frac{n}{5} + d\frac{7n}{10} + cn =$$
$$= \left(\frac{9d}{10} + c\right) n \overset{\text{Берем } d = 10c}{=} 10cn = dn$$

что и требовалось доказать. Тем самым (с учетом того что массив надо считать как минимум, т.е. что $T(n) = \Omega(n)$ получаем, наконец, $T(n) = \Theta(n)$.

2 По поводу задач

Задача 2.1 (Задача 6 из листка 4)**.** Постройте итеративную версию алгоритма MergeSort.

Решение. Первое что выдает Google по поводу итеративной реализации MergeSort.

Но можно обсудить и словами. $MergeSort$ делит массив на все меньшие куски, после чего собирает их обратно. Если мы хотим это сделать итеративно то нам достаточно перебирать размер $size$ сливаемых массивов (как степень двойки пока $size < n$), смотреть на пары последовательных кусочков массива этого размера, и вызывать внешнюю функцию $Merge(A, l, p, r)$ которая берет массив A, смотрит в нем на два куска $[l, p)$ и $[p, r)$, создает новый массив A' размера $r - l$, сливает туда два этих куска и заменяет им отрезок $[l, r)$ в массиве A.

Задача 2.2 (Задача 8 из листка 4)**.** Докажите, что любую сортировку сравнениями можно сделать устойчивой сохранив асимптотическое время работы.

Решение. Устойчивость сортировки, по определению, означает что если до сортировки было выполнено $A[i] = A[j]$ и $i < j$ то после сортировки, когда $i \to t', j \to j'$ выполнено $i' < j'$, т.е. порядок одинаковых элементов сохраняется.

Так давайте сделаем их неодинаковыми!

Будем работать не с исходными элементами, а с новым массивом $B = [(A[1], 1), (A[2], 2), \ldots, (A[n], n)]$. И определим сравнение двух пар из этого массива следующим образом

$$(l_1, l_2) < (r_1, r_2) := \begin{cases} l_1 < r_1, & l_1 \neq r_1 \\ l_2 < r_2, & else \end{cases}$$

при этом равенства здесь получаться не будет, т.к. как минимум по второму параметру все элементы различны. Осталось запустить сортировку, использующую данную процедуру вместо сравнения. Элементы у которых равны первые значения окажутся отсортированными по второму – по индексу в исходном массиве, что нам и нужно.