Cpt S 450 Homework #3 Solutions

Please print your name!

1. (easy) Write an algorithm that selects both the maximal element and the minimal element from an array A of n elements, using only $1.5 \cdot n$ comparisons.

You have many ways to write an algorithm – this is my way of writing an algorithm using English (no code).

I scan through the array A from left to right and maintain two numbers max and min that are the maximum and the minimum of the numbers that I have scanned. During the process of scanning, I read two numbers a and b from the array at a time and keep reminding myself that I can only perform three comparisons in order to obtain the new max and the new min. (This is where the 1.5 comes from) Indeed I can achieve this using only three comparisons:

- compare a and b, withoutloss of generality, assume that the larger is a and the smaller is b;
- compare a with max – the larger one is the new max;
- compare b with min – the smaller one is the new min.

So, the total comparisons needed is roughly $1.5 \cdot n$.

2. (not so easy) The algorithm $S(A, n, i)$ selects all the j-th smallest elements (with $j \leq i$) from an array A of n elements, by using linearselect to select each of the j-th smallest elements (with $j \leq i$). Clearly, one could also implement S alternatively as $T(A, n, i)$, which first sort A (on average-case and on worst-case, the sorting takes time $O(n \log n)$ using mergesort) and then select the first i elements. Please compare the complexities of the two algorithms; i.e., For the average-case complexities, under what conditions (on the choices for i), S is better than T or vice versa.

For algorithm T, the time needed is $O(i \cdot n)$. A careful student would come up with the following (which is better). Since linearselect runs in $O(n)$, let us assume it is cn for some c. If you assume that after an element is selcted, the element is gone. Then, the total time for selecting all the j-th smallest elements with $j \leq i$ will be

$$\sum_{1 \leq j \leq i} c \cdot (n - j + 1),$$

which is $\frac{ci(2n-i+1)}{2}$; i.e., $O(i(2n - i + 1)) = O(i \cdot n)$.

For algorithm S, we first sort (takes $O(n \log n)$) and then collect (the collection of the first i elements takes time $O(i)$). Hence, the total time of S is $O(n \log n) + O(i) = O(n \log n)$.

Now, we need to figure out the conditions for $O(i \cdot n) \geq O(n \log n)$. Obviously, the cutting point is $i \geq O(\log n)$.

3. (hard) In class, we have demonstrated the worst case complexity analysis for linearselect where each group has $k = 5$ numbers. Please show the worst case complexities for $k = 3$ and $k = 7$.

For $k = 3$, total number of medians is $\frac{n}{3}$ – this is also the number of groups. How many elements that are less than the median of the medians? at least $2 \cdot \frac{1}{2} \cdot \frac{n}{3} = \frac{n}{3}$. How many elements that are greater than the median of the medians? at least $2 \cdot \frac{1}{2} \cdot \frac{n}{3} = \frac{n}{3}$. So, the worst case location of the median of the median is $n - \frac{n}{3} = \frac{2n}{3}$ (this corresponds to the $\frac{7n}{10}$ when $k = 5$ presented in class notes). Hence,

$$T_W(n) \leq \Theta(n) + T_W(\frac{n}{3}) + T_W(\frac{2n}{3}).$$

You may verify that $T(n) = O(n^\delta)$ for any δ that satisfies, $(\frac{1}{3})^\delta + (\frac{2}{3})^\delta < 1$; i.e., $\delta > 1$, such as $\delta = 1.00001$.

For $k = 7$, total number of medians is $\frac{n}{7}$ – this is also the number of groups. How many elements that are less than the median of the medians? at least $4 \cdot \frac{1}{2} \cdot \frac{n}{7} = \frac{2n}{7}$. How many elements that are greater than the median of the medians? at least $4 \cdot \frac{1}{2} \cdot \frac{n}{7} = \frac{2n}{7}$. So, the worst case location of the median of the median is $n - \frac{2n}{7} = \frac{5n}{7}$. (this corresponds to the $\frac{7n}{10}$ when $k = 5$ presented in class notes). Hence,

$$T_W(n) \leq \Theta(n) + T_W(\frac{n}{7}) + T_W(\frac{5n}{7}).$$

You may verify that $T(n) = O(n)$ works.

4. (hard) Let ilselect(A, n, i) be an algorithm that selects the i-smallest from an array A with n integers. It works as follows:

ilselect$(A, n, i)\{$

r=partition$(A, 1, n)$;

//test if $A[r]$ is the element to be selected

if $i == r$, return $A[r]$;

//test if quickselect from the low-part

if $i < r$, return quickselect$(A, 1, r-1, i)$;

//test if linearselect from the high-part

if $i > r$, return linearselect$(A, r+1, n, i-r)$;

$\}$

That is, the algorithm runs quickselect on the low-part or runs linear select on the high-part. Show the worst-case complexity and the average complexity of the algorithm.

For the worst-case, let us first fix an r. Then, for this r, the worst case complexity of the algorithm is the maximal of the following two:

- the worst case of the quickselect on the low-part (which has $r-1$ elements): $O(r^2)$;
- the worst case of the linear select on the high-part (which has $n-r$ elements): $O(n-r)$.

Therefore, the worst case complexity of the algorithm, regardless of the choice of the r is $O(\max_{1\leq r\leq n}(r^2, n-r))$, which achieves $O(n^2)$ when $r = n$.

Now, we focus on the average-case complexity, which is denoted by $T_{avg}(n)$. Notice that in here we ignore the i in $T_{avg}(n)$. That is, $T_{avg}(n)$ is measured when the input is random (in a random ordering) and the i is also random (with i being one of 1..n equally likely).

The algorithm first partitions the input array $A[1..n]$ into two parts. The low-part has $r-1$ elements, and the high-part has $n-r$ elements. Since the input is random, the r can be at any position between 1 and n equally likely (i.e., with uniform probability $\frac{1}{n}$). Let fix any such r. Now, consider the random i. It could be between 1 and n equally likely (i.e., with uniform probability $\frac{1}{n}$). That is, the i can happen to be r with probability $\frac{1}{n}$; the i can happen to fall in the low-part (whose length is $r-1$) with probability $\frac{r-1}{n}$; the i can happen to fall in the high-part (whose length is $n-r$) with probability

$\frac{n-r}{n}$. Notice also that, within a high/low part, the i is still random – that is, the i can be in any position within the part equally likely. In summary, $T_{avg}(n)$ is the summation of

- $\Theta(n)$, the cost of partition itself;
- for every $1 \leq r \leq n$, the probability $\frac{1}{n}$ (of the choice of the r) times
  - $O(1) \cdot \frac{1}{n}$ – the i happends to be r exactly;
  - $T_{avg}^{\text{quickselect}}(r-1) \cdot \frac{r-1}{n}$ – the i falls in the low-part with probability $\frac{r-1}{n}$; (so we need to run quickselect over the low-part);
  - $T_{avg}^{\text{linearselect}}(n-r) \cdot \frac{n-r}{n}$ – the i falls in the high-part with probability $\frac{n-r}{n}$; (so we need to run linearselect over the high-part).

Formally,

$$T_{avg}(n) = \Theta(n) + \frac{1}{n} \sum_{1 \leq r \leq n} (O(1) \cdot \frac{1}{n} + T_{avg}^{\text{quickselect}}(r-1) \cdot \frac{r-1}{n} + T_{avg}^{\text{linearselect}}(n-r) \cdot \frac{n-r}{n}).$$

We know that, the average case complexity for both quickselect and linearselect is linear, so we have,

$$T_{avg}(n) = \Theta(n) + \frac{1}{n} \sum_{1 \leq r \leq n} (O(1) \cdot \frac{1}{n} + O(r-1) \cdot \frac{r-1}{n} + O(n-r) \cdot \frac{n-r}{n}).$$

The above can be simplified to

$$T_{avg}(n) = \Theta(n) + \frac{2}{n} \sum_{1 \leq r \leq n} O(r-1) \cdot \frac{r-1}{n}.$$

Notice that $\sum_{1 \leq r \leq n} O((r-1)^2)$ (absue math here) will give you roughly $\frac{1}{3}n^3$. Hence, the entire $T_{avg}(n)$ can be further simplified into $T_{avg}(n) = \Theta(n)$.