Indian Institute of Information Technology Design and Manufacturing,
Kancheepuram, Chennai 600 127, India
An Autonomous Institute under MHRD, Govt of India
http://www.iiitdm.ac.in
COM 209T Design and Analysis of Algorithms -Lecture Notes

Instructor
N.Sadagopan
Scribe:
P.Renjith

Order Statistics

Objective: In this lecture, we shall discuss how to find simultaneous minimum and maximum, simultaneous minimum and second minimum, and k^{th} minimum efficiently.

1. **Finding Minimum and Maximum:** A trivial approach is to scan the array twice, which incurs $(n-1)$ comparisons for minimum and $(n-2)$ comparisons for maximum, together $2n-3$ comparisons for finding simultaneous minimum and maximum.
   A non-trivial approach is to group the elements into pairs, i.e., $(a_1, a_2), (a_3, a_4), \ldots, (a_{n-1}, a_n)$. Find minimum and maximum for each pair which incurs one comparison for each pair, over all $\frac{n}{2}$ comparisons to get $\frac{n}{2}$ minimums and $\frac{n}{2}$ maximums. Perform a linear scan on $\frac{n}{2}$ minimums to get the actual minimum, this step incurs $\frac{n}{2}-1$ comparisons. Similarly, perform a linear scan on $\frac{n}{2}$ maximums to get the actual maximum, this step incurs $\frac{n}{2}-1$ comparisons. Thus, this approach incurs $\frac{n}{2}+\frac{n}{2}-1+\frac{n}{2}-1 = 3\frac{n}{2}-2$ comparisons, better than the previous approach.
   Another non-trivial approach is to find minimum and maximum using divide and conquer method. The cost given by the recurrence: $T(2)=1$, $T(n)=2T(\frac{n}{2})+2$. The additive factor '2' in the recurrence denotes the cost for updating minimum and maximum at each step.
   $T(n) = 2T(\frac{n}{2})+2$
   $T(n) = 2^2T(\frac{n}{4})+2^2+2$
   after $k-1$-steps, $T(n) = 2^{k-1}T(\frac{n}{2^{k-1}})+2^{k-1}+\ldots+2^2+2$
   Assuming $n=2^k$, $T(n) = 2^{k-1}\cdot T(2)+2^k-1-1$
   $T(n) = \frac{n}{2}+n-2 = 3\frac{n}{2}-2$.
   Thus divide and conquer approach is as good as the previous approach. Interestingly, the last two approaches are the best ever possible, i.e. any algorithm for finding minimum and maximum incurs at least $3\frac{n}{2}-2$ comparisons.

2. **Finding Minimum and Second Minimum:** Similar to the above problem, trivial approach takes $2n-3$ comparisons and divide and conquer approach takes $3\frac{n}{2}-2$ comparisons to output minimum and second minimum. It is natural to ask, can we do better than $3\frac{n}{2}-2$ comparisons. We shall now present an approach that incurs $n+\log n-2$ comparisons. Unlike minimum and maximum, minimum and second minimum are related, i.e., the candidates for second minimum are those elements that were compared with minimum at some iteration of the algorithm. The algorithm is described as follows; form pairs $(a_1, a_2), (a_3, a_4), \ldots, (a_{n-1}, a_n)$ and find a minimum for each pair. Using these local minimums, pair them and find the next set of local minimums, proceed this way to get the actual minimum. We naturally obtain a tree like structure and the height of the tree is $\log n$. Further, the actual minimum appears at every level. To get the minimum, the cost is $n-1$. Since the height of the tree is $\log n$, the minimum must have been compared with $\log n$ elements and all are candidates for the second minimum, we need $\log n - 1$ comparisons to obtain the

second minimum. Overall, $n - 1 + \log n - 1 = n + \log n - 2$ comparisons.

We shall now present an efficient algorithm to find i^{th} minimum in an array of integers. A trivial algorithm is to scan the array linearly i times which gives $O(ni)$ solution. We shall now see an $O(n)$ algorithm to find i^{th} minimum which is popularly known as i^{th} order statistics in the literature.

`Input:` A set A of n distinct numbers, an integer $i, 1 \leq i \leq n$
`Goal:` Find i^{th} order statistic. The i^{th} order statistic is the i^{th} smallest element.

To solve the above problem, another trivial method is to sort the given array in ascending order, and then find the i^{th} element. From the lower bound theory argument it is clear that any sorting incurs a cost of $\Omega(n \log n)$. A natural question is to ask for a more efficient solution to this problem. That is, is it possible to obtain the i^{th} smallest element in linear time? The solution presented here is from the text `Introduction to Algorithms, CLRS`.

Approach

- Divide the n elements into $\lfloor \frac{n}{5} \rfloor$ groups of 5 elements each, and at most one group contains n mod 5 elements.
- Find the median of each of the $\lceil \frac{n}{5} \rceil$ groups using the insertion sort. `Since there are 5 elements, the insertion sort takes at most 10 comparisons for each group which is` $O(1)$. `Overall,` $\frac{n}{5} \cdot O(1) = O(n)$
- Using the $\frac{n}{5}$ medians, recursively find the median of medians. i.e, form $\frac{n}{25}$ groups of size 5 each (each group contains 5 medians). Recursion bottoms out when the group size is one. `cost for finding the median of medians is` $T(n) = T(\frac{n}{5}) + O(n)$
- Partition the input array around the median-of-medians x such that x is the k^{th} smallest element. `Invoke quick sort PARTITION(A,x) with pivot as` x `and let` k `be the position of` x `at the end of partition. Let` L `and` R `be the left and right partition with respect to` x
- If $i = k$, then return x, otherwise recursively find the i^{th} smallest element. That is, if $i < k$, then recursively find the i^{th} smallest element in L, otherwise find the $(i-k)^{th}$ smallest element in R.

We know that partition() routine incurs $O(n)$ for each call, however, due to pruning (we either work with L or R), the size of the recursive subproblem is strictly less than n. We shall now analyze the size of the recursive subproblem using which we can estimate the run-time of order statistics. Towards this end, we consider the illustration given in Figure 1 which depicts the position of x and its relation with the other elements.
The n elements are represented by small circles, and each group of 5 elements occupies a column. The medians of the groups are whitened, and the median-of-medians x is labeled. (when finding the median of an even number of elements, we use the lower median.) Arrows go from larger elements to smaller, from which we can see that 3 out of every full group of 5 elements to the right of x are greater than x, and 3 out of every group of 5 elements to the left of x are less than x. The elements known to be greater than x appear on a shaded background.

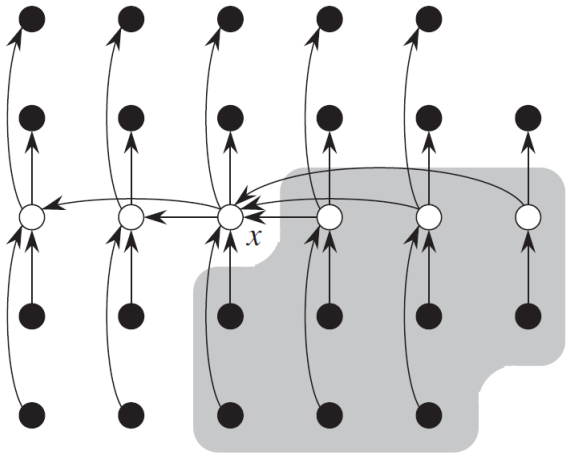

Figure 1: An illustration, Source: CLRS

To analyze the running time we shall determine a lower bound on the number of elements that are larger than the element x. Note that, at least half of the $\lceil \frac{n}{5} \rceil$ groups contribute at least 3 elements that are greater than x, except for the one group that has fewer than 5 elements if 5 does not divide n exactly, and the one group containing x itself. Discounting these two groups, it follows that the number of elements greater than x is at least $3(\frac{1}{2}\lceil \frac{n}{5} \rceil - 2) \geq \frac{3n}{10} - 6$ elements Therefore, after pruning, the recursive call is on $n - (\frac{3n}{10} - 6) = \frac{7n}{10} + 6$ elements.
Similarly, the number of elements that are smaller than x is at least $3(\frac{1}{2}\lceil \frac{n}{5} \rceil - 2) \geq \frac{3n}{10} - 6$ elements.

In either case, the recursive call is on the subproblem of size at most $\frac{7n}{10} + 6$.
The recurrence is therefore, $T(n) = T(\lceil \frac{n}{5} \rceil) + O(n) + T(\left\lceil \frac{7n}{10} + 6 \right\rceil) + O(n)$
$T(\frac{n}{5})$ in the recurrence captures the recursive subproblem of median-of-medians computation and the cost of finding $\frac{n}{5}$ medians is $O(n)$. Further, another $O(n)$ is spent by the partition() to prune unnecessary elements and the reduced subproblem size is at most $\frac{7n}{10} + 6$. After simplifying, we get,
$T(n) = T(\lceil \frac{n}{5} \rceil) + T(\left\lceil \frac{7n}{10} + 6 \right\rceil) + O(n)$
To solve this recurrence, we employ guessing strategy and guess $T(n) \leq c \cdot n$

$$\leq c(\frac{n}{5} + 1) + c(\frac{7n}{10} + 6) + dn \leq 7c + \frac{9cn}{10} + dn$$
$$\leq 7c + \frac{10cn}{10} - \frac{cn}{10} + dn \leq 7c + cn - \frac{cn}{10} + dn$$
$$\leq 7c + cn - \frac{cn}{10} + dn$$

This expression must be at most cn. i.e. what is the value of c and d such that $7c + cn - \frac{cn}{10} + dn \leq cn$
If no such c and d exist, then the guess is incorrect.

$$7c - \frac{cn}{10} + dn \leq 0 \implies c(\frac{n}{10} - 7) \geq dn \implies c(\frac{n - 70}{10}) \geq dn$$

$$c \geq \frac{10 \cdot d \cdot n}{n - 70}$$

For $n \geq 71$, there exist c satisfying the above constraint, and therefore $T(n) = O(n)$. Since the analysis works for $n \geq 71$, for problem size less than 71, to find order statistics, we employ insertion sort. Since 71 is a constant, the insertion sort takes $O(1)$ effort. A closer at the analysis tells us that for every $n \geq 140$, $\frac{n}{n-70} \leq 2$, and hence, $c \geq 20.d$. This implies that for a fixed value of d, we could fix $c = 20.d$. Also, similar to the above observation, for $n < 140$, we see that the algorithm takes constant effort and the recurrence works fine for $n \geq 140$. This completes the analysis, and we conclude

$$T(n) = \begin{cases} O(1) & \text{if } n < 140 \\ O(n) & \text{if } n \geq 140 \end{cases}$$

It is now natural to ask, what is the significance of choosing group size to be '5'. Will the above analysis work fine if it is '3' or '7'. For '7', the analysis works fine and yield $O(n)$ algorithm for order statistics whereas group size '3' leads to $O(n \log n)$ algorithm and hence '3' is not chosen in practice.

Application: Order statistics can be used as a black box in quick sort so that the worst case run-time is $\theta(n \log n)$. The classical quick sort without any additional black box, incurs $\theta(n^2)$ in the worst case as we may get a skewed partition at each iteration of the algorithm. However, if pivot is carefully chosen so that we get a balanced partition (good split) at every iteration, then we get $T(n) = 2T(\frac{n}{2}) + n$, which is $\theta(n \log n)$. Here is an approach to achieve good split at every iteration: invoke order-statistics() with $k = \frac{n}{2}$, i.e. ask for $(\frac{n}{2})^{nd}$ min and pass that to partition as the pivot element. Clearly, the output of partition guarantees a good split. Similarly, for each recursive subproblem of size m, ask for $(\frac{m}{2})^{nd}$ min using order-statistics(), subsequently, partition is done with respect to $(\frac{m}{2})^{nd}$ min. Therefore, the total cost is: $T(n) = 2T(\frac{n}{2}) + O(n) + O(n)$, which is $T(n) = \theta(n \log n)$. Note the first $O(n)$ in the expression is for order-statistics() and the second $O(n)$ is for partition().

Acknowledgements: Lecture contents presented in this module and subsequent modules are based on the following text books and most importantly, author has greatly learnt from lectures by algorithm exponents affiliated to IIT Madras/IMSc; Prof C. Pandu Rangan, Prof N.S.Narayanaswamy, Prof Venkatesh Raman, and Prof Anurag Mittal. Author sincerely acknowledges all of them. Special thanks to Teaching Assistants Mr.Renjith.P and Ms.Dhanalakshmi.S for their sincere and dedicated effort and making this scribe possible. Author has benefited a lot by teaching this course to senior undergraduate students and junior undergraduate students who have also contributed to this scribe in many ways. Author sincerely thanks all of them.

References:

1. E.Horowitz, S.Sahni, S.Rajasekaran, Fundamentals of Computer Algorithms, Galgotia Publications.
2. T.H. Cormen, C.E. Leiserson, R.L.Rivest, C.Stein, Introduction to Algorithms, PHI.
3. Sara Baase, A.V.Gelder, Computer Algorithms, Pearson.