INDIAN STATISTICAL INSTITUTE

Mid Semestral Examination

B. Stat. - III Year, 2014-2015 (Semester - VI)

Design and Analysis of Algorithms

Date: 27.02.2015 Maximum Marks: 60 Duration: 2 hours 30 minutes

---

Note: Answer as much as you can, but the maximum you can score is 60.

This is a cheat sheet based examination. You can write whatever you want on an A4 sized sheet of paper and bring to the exam hall. Write your name and roll number on the sheet. Cheat sheets can not be shared.

(Q1) We say that a function $f(n)$ is polylogarithmically bounded if $f(n) = O(\log^k n)$ for some constant k. Now, prove that any positive polynomial function grows faster than any polylogarithmic function. A positive polynomial function is of the form n^c for any constant $c > 0$. [8]

(Ans:) For all real constants a and b with $a > 1$, we have $\lim_{n\to\infty} \frac{n^b}{a^n} = 0$. Replace n with $\log n$ and a with 2^a in the above. So, we get $\lim_{n\to\infty} \frac{\log^b n}{2^{a \log n}} = \lim_{n\to\infty} \frac{\log^b n}{n^a} = 0$. As this limit exists and is equal to zero, $\log^k n = o(n^c)$.

(Q2) Let X and Y be two arrays of n distinct integers each. X and Y are sorted in an increasing order. Now, consider $Z = X \cup Y$; you can assume that Z has $2n$ distinct elements. Find the median of Z efficiently. [8]

[Hints: Doing it in linear time is too trivial!]

(Ans:) Figure 1 should give an idea.

As to the complexity, at each step we have to look into any one of the two subproblem of size $n/2$. So, the guiding recurrence is $T(n) = T(n/2)+O(1)$, which solves to $T(n) = O(\log n)$.

(Q3) Let $\mathcal{A}$ be an array of n integers. Each $a_i \in \mathcal{A}$ lies in the range $[0, n^3 - 1]$. Design an efficient algorithm to sort $\mathcal{A}$. [10]

(Ans:) Write any $a_i \in \mathcal{A}$ in a 3-digit number system with radix n. So, the digits are $0, 1, \ldots, n-1$. So, you can write $n^3 - 1$ as $n - 1n - 1n - 1$ and its value is $n^2(n-1) + n^1(n-1) + n^0(n-1)$.

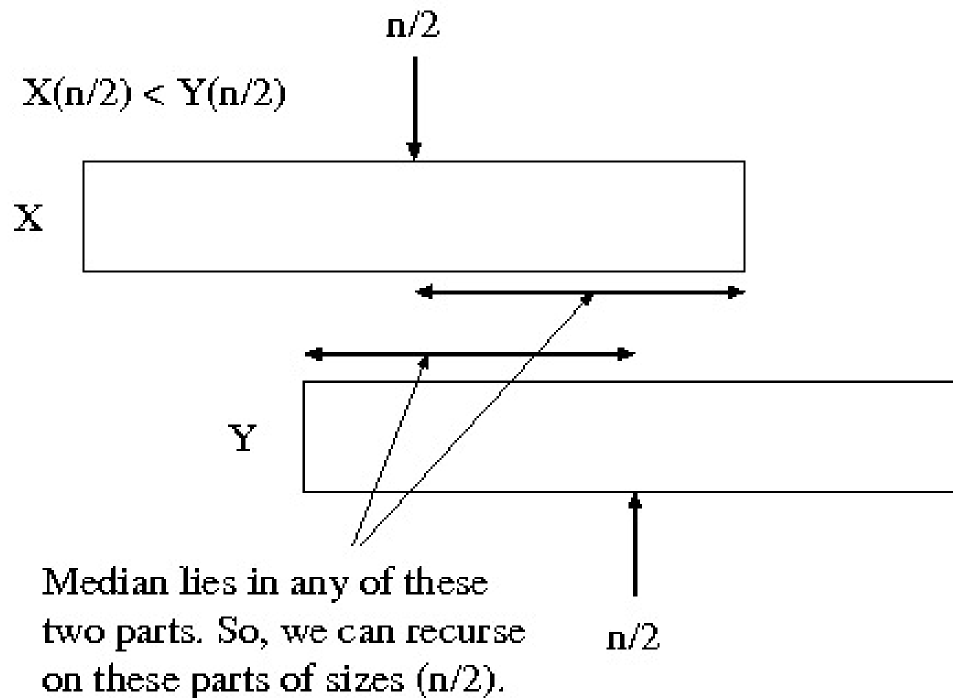

Figure 1: See the figure to understand how you recurse on a sub part to find the median.

Now, just do a counting sort on each radix taking $\Theta(n)$ time.

(Q4) Given a sequence of numbers $A = \{a_1, a_2, \ldots, a_n\}$, design an algorithm to find the length of the largest subset such that for every $i < j$, $a_i < a_j$. Find the complexity of the algorithm. An example of such a largest subset follows: $A = \{11, 17, 5, 8, 6, 4, 7, 12, 3\}$. The length of the largest subset is 4 and the set is $5, 6, 7, 12$. [10]

[Hints: Can dynamic programming be used?]

(Ans:) Sort A in $O(n \log n)$ time; let the sorted array be B. Now, run the algorithm of Longest Common Subsequence (LCS) studied in class with A and B as the two sequences; this takes $O(n^2)$ time. So, the total time taken is $O(n^2)$.

(Q5) In the *SELECTION* (median finding) algorithm studied in class, we divided the input elements into groups of 5; and found out by our analysis that the algorithm runs in linear time. Deduce what happens to the time complexity of the *SELECTION* algorithm if the input was divided into groups of 3. [10]

(Ans:) With groups of 5, we found that the median-of-medians is larger than or smaller than roughly $\frac{3}{2}\left\lfloor\frac{n}{5}\right\rfloor$ elements; and then the recursive call is on roughly $\frac{3n}{4}$ elements. This gives us a recurrence like $T(n) = T(\left\lfloor\frac{n}{5}\right\rfloor) + T(\left\lfloor\frac{3n}{4}\right\rfloor) +$

$O(n)$. The above recurrence solves to $O(n)$ because $\frac{1}{5} + \frac{3}{4} < 1$, as we saw in the class.

For groups of 3, the median-of-medians is larger than or smaller than roughly $\frac{n}{3}$ elements; and then the recursive call is on roughly $\frac{2n}{3}$ elements. This gives us a recurrence like $T(n) = T(\frac{n}{3}) + T(\frac{2n}{3}) + O(n)$. The above recurrence solves to $O(n \log n)$ because $\frac{1}{3} + \frac{2}{3} = 1$.

(Q6) Given an array $\mathcal{X}$ containing n *unique* real numbers, design and analyze an efficient algorithm that finds out a number in $\mathcal{X}$ that is neither minimum nor maximum. [4]

(Ans:) Just pick up the first three numbers $\mathcal{X}[1]$, $\mathcal{X}[2]$ and $\mathcal{X}[3]$; and from them, pick up the middlemost number. This number is surely going to be neither maximum nor minimum since of the other two numbers, one will be greater and the other lesser than this number. This is true as the elements in the array $\mathcal{X}$ are unique.

(Q7) Given n elements, the second maximum can be found trivially with $2n - 3$ comparisons in the worst case: $n - 1$ comparisons for the first maximum and $n - 2$ comparisons for the second maximum. Try to find out an efficient method (that takes less than $2n-3$ comparisons) to find the second maximum. Note that, here we are not interested in the asymptotic complexity but the exact count in the worst case. [10]

(Ans:) Think of this as a tournament. Pair up the n elements; the maximum of the two goes up. Do the same again with the $\frac{n}{2}$ elements; continue this till one gets the maximum. This procedure can be thought of as a binary tree where the root contains the maximum. The leaves are all the elements. So, the maximum must have traversed from the level of the leaf to the root. On the way, it must have beat the second maximum somewhere. To find that, traverse the tree from the root to the leaf and collect those elements; there can be at most $\log_2 n$ such elements. Among them, find the maximum; that must be the second maximum. So, it takes $(n - 1) + (\log_2 n - 1)$ comparisons, i.e., $n + \log_2 n - 2$ comparisons.

(Q8) Show that there is no comaprison-based sort whose running time is $O(n)$ for at least a constant fraction of the $n!$ inputs of length n. You can assume that the input is distinct. [10]

(Ans:) If the sort runs in linear time for m input permutations, then the height h of the portion of the decision tree, that has leaves corresponding to the m

permutations plus their ancestors, is $O(m)$. We now need to find the height of a decision tree in which each permutation appears as a reachable leaf. So, we have $2^h \geq m$, i.e. $h \geq \log m$. By the question, $m = \frac{n!}{c}$. So, from $h \geq \log m$, we have $h \geq \log \frac{n!}{c}$, i.e. $h \geq n \log n - \log c$, i.e. $h = \Omega(n \log n)$. So, h is lower bounded by $n \log n$ and thus we cannot have a comparison based sorting algorithm that runs in linear time for at least a constant fraction of the $n!$ inputs of length n.