INDIAN STATISTICAL INSTITUTE

Class Test I

M Tech (CS) – I Year, 2016-2017 (Semester - II)

Design and Analysis of Algorithms

Date: 17.02.2017 Maximum Marks : 30 Duration : 1.5 hours

Note: The question paper is of 40 marks. Answer as much as you can, but the maximum you can score is 30. Answer a question within its allotted box.

Course: (M Tech/JRF/PLP) _____

Name: _____ **Roll Number:** _____

(Q1) You are given a sequence $\mathcal{X}$ of n real numbers. Design and analyze an efficient algorithm to find a contiguous subsequence $\mathcal{X}_{ij} = \mathcal{X}(i) \ldots \mathcal{X}(j)$ of $\mathcal{X}$ for which the sum of elements in the subsequence is maximized. (As an example, $3, -4, 9$ is a contiguous subsequence of $\{1, -2, 3, -4, 9\}$) [10]

First, note that if all numbers were positive, the entire array is the answer. A negative number throws up options. You may gain or lose if you include it. When do you gain and when do you lose? Think it over.

A quick look at the characteristics of the problem shows that it admits overlapping subproblems and the optimal substructure property. Thus, it is a fit candidate for a dynamic programming solution.

Let $S(j)$ denote the maximum sum of the elements in a contiguous subsequence ending at the j-th location of the array $\mathcal{X}$. So, finding $S(n)$ would give us the desired solution. Now,

$$S(j) = max\{S(j-1) + \mathcal{X}(j), \mathcal{X}(j)\}\ 1 \leq j \leq n;$$

Clearly, this requires $O(n)$ time.

(Q2) You are given an array A of size n. You are told that A comprises three consecutive runs – first a run of 'a's, then a run of 'b's and finally a run of 'c's. Moreover, you are provided an index i such that $A[i] = b$. Design an $O(\log n)$ time algorithm to determine the number of 'b's (i.e., length of the second run) in A. [10]

A problem that asks for an $O(\log n)$ solution should bring to your mind an application of binary search.

An index i is provided such that $A[i] = b$. The array, as it is given to you in terms of runs of 'a', 'b' and 'c', is actually a sorted array over the alphabets 'a', 'b' and 'c'. Do a binary search between 1 and i to locate the first occurrence of 'b'. How? First probe at location $\lfloor (i+1)/2 \rfloor$.

If it is an 'a', you are sure that all alphabets between $[1, \lfloor (i+1)/2 \rfloor]$ are 'a' and then you can throw this part and recurse on the part of the array between $[\lfloor (i+1)/2 \rfloor + 1, i]$. If it is a 'b', you are sure that all alphabets between $[\lfloor (i+1)/2 \rfloor + 1, i]$ are 'b' and then you can throw this part and recurse on the part of the array between $[1, \lfloor (i+1)/2 \rfloor - 1]$. In both the cases, you are throwing away half of what you started with. Continue this till you encounter the last 'a' or first 'b'.

Similarly, do on the side of 'c' to get the last 'b' or first 'c'. With the indices of the last 'a' or first 'b' and the last 'b' or first 'c', you can easily compute the length of the run of 'b'.

Recall the guiding recurrence of this recursive technique: $T(n) \leq T(n/2) + O(1)$ with $T(1) = O(1)$. This solves to $O(\log n)$.

(Q3) Your problem is to sort n distinct elements using a comparison based sort. So, your input can be any one of the $n!$ permutations of the distinct elements. Now, prove or disprove the following statement: *There can exist a comparison based sort whose running time is $O(n)$ for at least a constant fraction of the $n!$ inputs of length n.* [10]

The statement can not be true because of the following argument.

If the sort runs in linear time for m input permutations, then the height 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. Let h be the height of the decision tree. So, we have $2^h \geq m$, i.e. $h \geq \lg m$. By the question, $m = \frac{n!}{c}$, where $c > 1$ is a positive constant. So, from $h \geq \lg m$, we have $h \geq \lg \frac{n!}{c}$, i.e. $h = \Omega(n \lg n)$.

(Q4) Given an undirected graph $G = (V, E)$, design and analyze an efficient algorithm to determine if G contains a cycle of odd length. A cycle of odd length is called an odd cycle. [10]

Use either DFS or BFS.

In DFS, as you go down exploring the DFS tree, label the vertices alternately 0 and 1, i.e. if the parent is 0, mark the child as 1, and vice-versa. Now when you encounter a back edge, just check if it is between vertices of the same label. If they are between the same label, you have an odd cycle. If you do not encounter any such back edge, then there is no odd cycle.

To make the answer complete, prove the following easy statement: *There is an odd cycle in G if and only if there is a back edge between vertices of the same label.*

If you are using BFS, work with the *levels* and argue on a *cross-edge*, i.e. an edge which is not a *tree edge*.