INDIAN STATISTICAL INSTITUTE

Mid Semestral Examination

M. Tech (CS) - I Year, 2018-2019 (Semester - II)

Design and Analysis of Algorithms

Date: 22.02.2019 Maximum Marks: 100 Duration: 3.0 Hours

Note:

The model solution is only for Group-B questions as you have encountered Group-A questions in class.

Group A

(QA1) Show that, on the assumption that all permutations of a sequence of n elements are equally likely to appear as input, any decision tree that sorts n elements has an expected depth of at least $\log n!$. [10]

(QA2) Let P be a set of n points in $\mathbb{R}^2$. Design and analyze an algorithm to compute the closest pair of points in P in $O(n \log n)$ time. [5+5=10]

(QA3) Let $G = (V, E)$ be a directed graph, $v_0 \in V$ is a source vertex and $w : e \mapsto \mathbb{R}^+$, be a function that maps each edge e of G to a positive real weight.

(i) Describe Dijkstra's shortest path algorithm to find for each vertex $v \in V$, the least weighted path from v_0 to $v \in V$ in G. Deduce the time complexity of the algorithm.

(ii) Give the proof of correctness of the above algorithm.

[(3+2)+5=10]

(QA4) Given a sequence of matrices $A_1, \ldots, A_n$ and dimensions $p_0, \ldots, p_n$ where A_i is of dimension $p_{i-1} \times p_i$, design an algorithm that determines the multiplication order that minimizes the number of operations (i.e., multiplications), along with the minimum number of operations required. Explain your steps properly, and find the space and time complexities. [10]

(QA5) You are given an alphabet set and a set of frequencies for the letters.

(i) Define an optimal prefix code for the above input.

(ii) Design and analyze an algorithm to compute such an optimal prefix code that minimizes the average number of bits per letter.

(iii) Prove the correctness of your algorithm.

[2+4+4=10]

[P.T.O.]

Group B

(QB1) Let $\mathcal{A} = \{a_1, \ldots, a_n\}$ be a sequence of n distinct numbers. We say that two indices $i < j$ form an *inversion* if $a_i > a_j$. Design and analyze an efficient algorithm to find out the number of inversions in $\mathcal{A}$. [7+3=10]

(Ans:) Trivially, the problem can be solved in $O(n^2)$ time. We aim for an $O(n \log n)$ time algorithm using divide and conquer. So, we look for a recurrence like $T(n) \leq 2T(\frac{n}{2}) + O(n)$.

Our idea is like merge sort. We divide $\mathcal{A}$ into two equal halves — the left half goes to array A and the right half to array B; so any index in array B is greater than any index in array A. We sort A and B in an ancreasing fashion and count the number of inversions in A and B recursively. Now onward to the conquer part. We merge the sorted lists A and B into a sorted list C and count the number of inversions in them; let them be n_A and n_B, respectively. We merge A and B into C using pointers marching over A and B. Let the pointer on A be at $A[i]$ and the pointer on B be at $B[j]$. If $A[i] < B[j]$, then $A[i]$ gets copied to C and i is increased by 1. If $B[j] < A[i]$, then $B[j]$ is copied to C and j is increased by 1. Till this point, it is the same as the steps taken in merge sort. Now, when $B[j] < A[i]$, inversions have taken place not only with $A[i]$ (remember $j > i$) but also with all of $A[i+1]$ down to end of the array A. Keep doing this merge and count procedure till both arrays A and B are exhausted. Surely, this needs $O(n)$ time and we have our sweet little recurrence $T(n) \leq 2T(\frac{n}{2}) + O(n)$ and we are done!

(QB2) Let $\mathcal{A}$ be an array of n integers. Each $a_i \in \mathcal{A}$ lies in the range $[0, n^4 - 1]$. Design an efficient algorithm to sort $\mathcal{A}$. [10]

[Hints: Sorting in $O(n \log n)$ time using any known algorithm will not fetch any credit. Try using any linear time sorting!]

(Ans:) Write any $a_i \in \mathcal{A}$ in a 4-digit number system with radix n. In radix n, the digits are $0, 1, \ldots, n-1$. So you can write $n^4 - 1$ as $n-1\ n-1\ n-1\ n-1$ and its value is $n^3(n-1) + n^2(n-1) + n^1(n-1) + n^0(n-1)$.

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

(QB3) 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]

(Ans:) 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. $h = \Omega(n \lg n)$.

(QB4) In the selection algorithm studied in the class, we worked with 'groups of 5' and deduced that the algorithm runs in $O(n)$ time. Find out what happens, if we work with 'groups of 3' and 'groups of 7'. [5+5=10]

(Ans:) Do a careful analysis on the same lines of groups of 5 that we did in class. For groups of 3, the recurrence will be $T(n) = T(n/3) + T(2n/3) + \Theta(n)$, that solves to $O(n \log n)$. For groups of 7, the recurrence will be $T(n) = T(n/7) + T(5n/7) + \Theta(n)$, that solves to $O(n)$.

(QB5) (i) $X = \{x_1, x_2, \ldots, x_n\}$ is an array of n distinct integers and x is an integer. Design an efficient algorithm to determine whether there are two elements in X whose sum is exactly x. Find the time complexity of the algorithm you designed.

(ii) $X = \{x_1, x_2, \ldots, x_n\}$ is an array of n distinct integers. Design an efficient algorithm to determine whether there are three elements in X whose sum is exactly 0. Find the time complexity of the algorithm you designed.

[4+6=10]

(Ans (i):) First sort the array X in $O(n \log n)$ time. Now start two pointers from the left and right extreme of the sorted array X and move them inwards, i.e., the left pointer moves right and the right pointer moves left. Look at the sum of the two numbers pointed to by the two pointers. If the sum is equal to k, we are done. If the sum is less than k, then move the right pointer left; and if the sum is more than k, then move the left pointer right. Continue till the pointers meet. This process takes $O(n)$ time. So, the total time taken is $O(n \log n)$.

(Ans (ii):) This problem is known as the 3SUM problem. First sort X in $O(n \log n)$ time. Now if $\exists a$, b, c such that $a + b + c = 0$, then $a + b = -c$, i.e., two numbers that sum to a number and we are back to the setting of Problem (i). So, for the first $n - 2$ entries in the sorted X, we run the $O(n)$ time algorithm as above. So, in all we have $O(n^2)$ time complexity. There are better algorithms than this for the 3SUM problem though!

(QB6) (i) A *binary heap* is an *almost-complete* binary tree with each node satisfying the *heap property:* If v and par(v) are a node and its parent, respectively, then the key of the item stored in par(v) is not greater than the key of the item stored in v. A binary heap supports the following operations: (i) deleting the minimum element in $O(\log_2 n)$ time, and (ii) inserting/shifting an element up the heap in $O(\log_2 n)$ time. A d-ary heap is a generalization of a binary heap in which each internal node in the almost-complete d-ary rooted tree has at most d children instead of 2, where $d > 2$ can be arbitrary.

Design and analyze efficient algorithms for the following operations in a d-ary heap. Explicitly mention the time complexity.

- deleting the minimum element
- inserting/shifting an element up the heap

(ii) Let $G = (V, E)$, with $|V| = n$ and $|E| = m$, be a directed graph as in the setting of (QA3). Let $m \geq n^{1+\epsilon}$, for some $\epsilon > 0$ that is not too small, i.e. the graph is dense. Show that one can obtain a time complexity of $O(\frac{m}{\epsilon})$ for Dijkstra's algorithm if a d-ary heap, with a suitable choice of d as a function of m and n, is used.

[(5+5=10)]

(Ans (i):) The basic idea for a d-ary heap is that the same volume of nodes in the binary tree is redistributed with a lesser height and greater breadth. Each node of the d-ary heap now has at most d children. This makes the height of the d-ary heap to be $O(\log_d n)$. Inserting/shifting an element up the heap takes time same as the height. Thus, we have lot of savings for a d-ary heap (takes $O(\log_d n)$ time) as compared to 2-ary heap (takes $O(\log_2 n)$ time) for inserting/shifting an element up the heap. But this comes at a cost. While deleting an element and finding the correct place for the element that was swapped with the root, we need to search for all the children of a node; find the correct one; go to that level and recurse down the height of the tree. Thus, deleting the minimum element takes $O(d \log_d n)$ time.

(Ans (ii):) There are $O(n)$ minimum extraction operations from the heap, taking a total of $O(nd \log_d n)$ time. Each edge is relaxed at most once, and after the relaxation, a node may shift up the heap in $O(\log_d n)$ time. So, the total relaxation may take $O(m \log_d n)$ time. Thus, the total time taken is $O(nd \log_d n + m \log_d n)$.

Now, set $d = \lceil 2 + m/n \rceil$. With this choice, $nd = O(m)$. Thus, the total time taken is $O(m \log_d n)$. Now, we do a little algebraic manipulation,

$$\begin{aligned} O(m \log_d n) &= O(m \log_{\lceil 2+n^\epsilon \rceil} n) \\ &= O\left(m \frac{\log n}{\log n^\epsilon}\right) \\ &= O\left(m \frac{\log n}{\epsilon \log n}\right) \\ &= O\left(\frac{m}{\epsilon}\right) \end{aligned}$$

(QB7) Let $G = (V, E)$ be a directed acyclic graph with weight $w(u, v)$ on edge $(u, v) \in E$. Design and analyze an efficient algorithm to find the average path length from a source vertex $s \in V$ to a destination vertex $t \in V$. The average path length is defined as the total weight of all paths from s to t divided by the total number of distinct paths. [10]

[Hints: It would be easy to give a dynamic programming solution. The graph G being a directed acyclic graph admits a topological ordering on its set of vertices.]

(Ans:) If it is a dynamic programming technique we are looking for, then the crucial step is to locate the subproblems; and use their optimal solutions to generate further optimal solutions.

So, we do as following. Let $\mathsf{path}[x]$ be the number of distinct paths from x to t and $\mathsf{path\_sum}[x]$ be the sum of length of all paths from x to t. Now, consider the node/vertex y that is a neighbor of x and was visited first after x in the path from x to t. Now, note that by the definition of the number of paths $\mathsf{path}[x]$, there are $\mathsf{path}[y]$ distinct paths to take from y onwards to t. Now, we can formulate $\mathsf{path}[x]$ in terms of $\mathsf{path}[y]$ as follows:

$$\mathsf{path}[x] = \sum_{y:(x,y)\in E} \mathsf{path}[y].$$

Similarly, the corresponding paths from x to t are each $w(x, y)$ longer. Thus, the sum of path lengths are as follows:

$$\mathsf{path\_sum}[x] = \sum_{y:(x,y)\in E} (\mathsf{path}[y] \cdot w(x, y) + \mathsf{path\_sum}[y]).$$

As we are using subproblems, we have to fix the base cases. Obviously, $\mathsf{path}[t] = 1$ and $\mathsf{path\_sum}[t] = 0$.

Now, notice the dynamic programming recurrence. The recurrence indicates an ordering on the way we visit the vertices of the graph. Note that the graph under consideration is a *DAG*(directed acyclic graph). So, we can do a topological sorting on the $|V|$ vertices of G. Let the indices of the vertices after the topological ordering be $1, 2, \ldots, |V|$ so that for all edges $(u, v) \in E$, $u < v$. Now, start from $|V|$ and go down to 1 filling the tables for $\mathsf{path}[x]$ and $\mathsf{path\_sum}[x]$. We go on filling the table with zeros from $|V|$ down onwards till we reach the vertex t in the topological order. Then, we fill in the entry for t using the base cases of the recursion. Then, from t onwards down to 1, we use the recurrence equations developed above to fill in the tables for values of $\mathsf{path}[x]$ and $\mathsf{path\_sum}[x]$. Once the values are found out for the vertex s, then we can return the ratio.

As to the complexity, topological sorting takes $\Theta(|V| + |E|)$. For filling in the table, constant number of arithmetic operations per entry is to be done. Thus, the overall complexity is $\Theta(|V| + |E|)$.