INDIAN STATISTICAL INSTITUTE

Mid Semestral Examination

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

Design and Analysis of Algorithms

Date : 21.02.2017 Maximum Marks : 60 Duration : 3.0 Hours

Note: This is a two page question paper and it is of 75 marks. Answer as much as you can, but the maximum you can score is 60.

(Q1) Solve the recurrence relation

$$T(n) = \begin{cases} 0 & \text{if } n = 2; \\ 2T(\lfloor\sqrt{n}\rfloor) + 1 & \text{if } n > 2. \end{cases}$$

[10]

(Ans:) Let $m = \log_2 n$. Therefore, $T(2^m) = 2T(2^{m/2}) + 1$. We can now rename $S(m) = T(2^m)$. New recurrence becomes $S(m) = 2S(m/2) + 1$. Solving, we get $S(m) = O(m)$. Therefore, $T(n) = T(2^m) = S(m) = O(m) = O(\log n)$.

(Q2) We want to use a binary counter to count from 0 to n. You can think of a binary counter as an array X storing values 0 and 1; and any number is represented as a string of 0s and 1s. As an example, 11 (eleven) is represented as $\boxed{1011}$; to count 12 (twelve) using this counter, we have to change the configuration of the binary counter from $\boxed{1011}$ to $\boxed{1100}$. So in this example, we have to change 3 bit positions.

Now, the size or number of bits k of the binary counter to count numbers upto n is $\lfloor\log_2 n\rfloor + 1$. To increment the counter by 1, we can use the following routine (the routine is given for your help):

Method *Increment Counter*
Input: A counter $X[0, \ldots, k-1]$ of size $k = (\lfloor\log_2 n\rfloor + 1)$ to count from 0 to n with any number x_i between 0 to $n-1$.
Output: The number $x_i + 1$ stored in the counter X.

1. $j \leftarrow 0$;
2. **while** ($X[j] = 1$)
3. $\quad X[j] \leftarrow 0$; $j \leftarrow j + 1$;
4. **endwhile**
5. $X[j] \leftarrow 1$;

So, in the worst case we have to increment $\Theta(k) = \Theta(\log n)$ bits for incrementing by one. As, we have n numbers in all, the time complexity is $\Theta(nk) = \Theta(n \log n)$.

Do you think this is a tight time complexity analysis of the method? If not, then try improving the time complexity by doing a tighter analysis of the same method. [5]

[Hints: Write the binary representation of consecutive numbers and then see.]

(Ans:) Instead of looking into the number of flips of $\lfloor \log n \rfloor + 1$ bits for each increment, we look into each bit separately. As an example, see the binary representations of numbers from 0 to 7: $\boxed{000}$ $\rightarrow$ $\boxed{001}$ $\rightarrow$ $\boxed{010}$ $\rightarrow$ $\boxed{011}$ $\rightarrow$ $\boxed{100}$ $\rightarrow$ $\boxed{101}$ $\rightarrow$ $\boxed{110}$ $\rightarrow$ $\boxed{111}$. The ***LSB*** (i.e. $X[0]$) flips n times, $X[1]$ flips $\lfloor \frac{n}{2} \rfloor$ times, $X[2]$ flips $\lfloor \frac{n}{2^2} \rfloor$ times, ..., $X[j]$ flips $\lfloor \frac{n}{2^j} \rfloor$ times and so on. So, the total number of flips is $\sum_{j=0}^{\lfloor \log n \rfloor} \frac{n}{2^j} < n \sum_{j=0}^{\infty} \frac{1}{2^j} = 2n$. So, a tighter analysis will give a time complexity of $\Theta(n)$.

(Q3) Let $A[1 \ldots n]$ be an array of n distinct numbers. A is unimodal, i.e., for some i, $1 \leq i \leq n$, $A[1] < \ldots < A[i]$ and $A[i] > A[i+1] > \ldots > A[n]$. Design and analyze an efficient algorithm to find i. [5]

(Ans:) Probe at the middle of the array, i.e. $A[\frac{n}{2}]$ and compare it with $A[\frac{n}{2} - 1]$ and $A[\frac{n}{2} + 1]$. Now, consider the following cases:

(Case 1) ($A[\frac{n}{2} - 1] < A[\frac{n}{2}] < A[\frac{n}{2} + 1]$): The mode will be between $A[\frac{n}{2} + 1]$ and $A[n]$ and we can recurse on the part between $\frac{n}{2} + 1$ and n.

(Case 2) ($A[\frac{n}{2} - 1] > A[\frac{n}{2}] > A[\frac{n}{2} + 1]$): The mode will be between $A[1]$ and $A[\frac{n}{2} - 1]$ and we can recurse on the part between 1 and $\frac{n}{2} - 1$.

(Case 3) (otherwise:) So, here $A[\frac{n}{2}]$ is greater than both $A[\frac{n}{2} - 1]$ and $A[\frac{n}{2} + 1]$ and hence $A[\frac{n}{2}]$ is the peak element.

In all of the above (exclusive as well as exhaustive cases), we perform at most three comparisons and reduce the problem to a size of at most half of the original problem. Thus, we have an $O(\log n)$ time algorithm.

(Q4) 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 these operations in a d-ary heap. Explicitly mention the time complexity.

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

[(5+5=10)]

(Ans:) 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.

(Q5) 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]

[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 3-digit number system with radix n. In radix n, the digits are $0, 1, \ldots, n-1$. So you can write $n^3 - 1$ as $n - 1n - 1n - 1$ and its value is $n^2(n-1) + n^1(n-1) + n^0(n-1)$.

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

(Q6) Consider the problem of constructing a binary search tree from a list $\mathcal{L}$ of n unsorted elements. Find out a non-trivial lower bound of this problem. [10]

(Ans:) Let $f(n)$ be the worst case running time of constructing a binary search tree. We want to lower bound $f(n)$. Now, we know that an inorder traversal on a binary search tree of the list $\mathcal{L}$ gives its sorted order. Also, this inorder traversal takes $\Theta(n)$ time. This means that in linear time we can convert the binary search tree on $\mathcal{L}$ to its sorted order. We know that the lower bound on comparison based sorting is $\Omega(n \log n)$. Therefore,

$$f(n) + \Theta(n) = \Omega(n \log n).$$

So,

$$f(n) = \Omega(n \log n).$$

(Q7) 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. A topological ordering on the set of vertices is a linear ordering of its vertices such that if the directed edge $(u, v) \in E$, then u appears before v in the ordering. Now, look at the ordering obtained in the topological ordering for finding the average length. Does it admit an overlapping subproblem? Let $s \rightsquigarrow x \rightarrow y \rightsquigarrow t$ be any path from s to t where y is the immediate neighbour of x. Can you formulate the number of distinct paths as well as their weight from x to t if you know the the number of distinct paths as well as their weight from y to t?]

(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 $\Theta(|V| + |E|)$.

(Q8) Let $G = (V, E)$ be a directed graph in which $E = E_1 \cup E_2$ and $E_1 \cap E_2 = \phi$. The edges in E_1 have weights greater than or equal to zero and the edges in E_2 have weights less than zero. Now to find out the single source shortest path, we do the following. Let w_e be the minimum weight of all edges in E_2. Surely $w_e < 0$. Now we add $|w_e|$ to all the edge weights to make them non-negative. Now we run Dijkstra's shortest path algorithm on this graph with the new edge weights (i.e., all edge weights increased by $|w_e|$) to get the desired shortest path.

Can we do so? Prove or disprove the above statement. [5]

(Ans:) This is a **wrong** statement and cannot be proved.

If we add the same amount of weight to each edge to make it positive, and then run the Dijkstra's algorithm, it does not find the shortest path in the original graph. This is so because paths that use more edges (which could have been negative edges) will be rejected in favour of paths having more weight using fewer edges in the new graph. It is easy to construct such an example.

(Q9) (a) Let $G = (V, E)$ be a connected graph with the weight on each edge being a real number. The maximum spanning tree is a spanning tree whose sum of edge weights is the maximum. Design and analyze an efficient algorithm to compute the maximum spanning tree. Prove that your algorithm is correct.

(Ans:) Either in Prim's of Kruskal's algorithm, instead of selecting the minimum weight edge, select the maximum. The rest remains the same. The proof will also be the same.

You can also alter the edge weight w_x of any edge x of G by changing w_x to $\max_{e \in E} w_e - w_x$ and then run any minimum spanning tree algorithm.

(b) Show that if all the edge weights of the graph are unique, the maximum spanning tree is also unique.

(Ans:) A model solution for minimum spanning tree is given. You can surely convert the proof for maximum spanning tree!

Assume there are n vertices in G. So, there are $n-1$ edges in the minimum spanning tree(MST). First, observe that the minimum weight edge will be in any MST.

Now, for a contradiction, assume that there are two minimum cost spanning trees T_1 and T_2. Let the sequence of the edges in an ascending sorted order be $e_{min}, e^1_{T_1}, e^2_{T_1}, \ldots, e^i_{T_1}, \ldots, e^{n-1}_{T_1}$ in T_1 and the sequence of the edges in ascending sorted order in T_2 be $e_{min}, e^1_{T_2}, e^2_{T_2}, \ldots, e^i_{T_2}, \ldots, e^{n-1}_{T_2}$. Let $e^1_{T_1} = e^1_{T_2}$, $e^2_{T_1} = e^2_{T_2}$, $\ldots$ Let the first edge where the edge sequences differ be $e^i_{T_1}$ and $e^i_{T_2}$, i.e. $e^i_{T_1} \neq e^i_{T_2}$. Assume $e^i_{T_1} > e^i_{T_2}$. Observe that because of the ascending sorted edge sequence $e^i_{T_2}$ cannot belong to $\underbrace{e^{i+1}_{T_1}, \ldots, e^{n-1}_{T_1}}$. The sum of the rest of the edges in T_1 should be lesser (strictly lesser because of uniqueness of edge weights) than the sum of the rest of the edges in T_2, i.e. $\sum_{k=i+1}^{n-1} e^k_{T_1} < \sum_{k=i+1}^{n-1} e^k_{T_2}$. This happens because the sum of the edge weights in T_1 and T_2 are the same. Now, consider the edges $e^{i+1}_{T_1}, \ldots, e^{n-1}_{T_1}$. They form a spanning tree of $n-i-1$ vertices. Also, consider the edges $e_{min}, e^1_{T_1}, e^2_{T_1}, \ldots, e^{i-1}_{T_1}$. They form a spanning tree of $i-1$ vertices. Finally consider the following edge sequence comprising the two above *disjoint* spanning trees joined by the edge $e^i_{T_2}$ not belonging to any of these two spanning trees, $\underbrace{e_{min}, e^1_{T_1}, e^2_{T_1}, \ldots, e^{i-1}_{T_1}}\, e^i_{T_2}\, \underbrace{e^{i+1}_{T_1}, \ldots, e^{n-1}_{T_1}}$. This new edge sequence surely is a spanning tree, and note that its total weight is less than T_1. Thus, T_1 is not an MST. Hence, we have a contradiction and the proof.

[5+5=10]