Major Exam COL 106. 23rd Nov, Sem I, 2024-25.

Saturday November 23rd, 2024. 8 am to 10 am.

There are 7 questions and all are compulsory. Maximum marks: 35

Note: You can assume without explanation any algorithm (or result) discussed in class. Each algorithm must be clearly stated along with justification of correctness and running time. You must **not** use examples in correctness proofs.

1. **(4 marks)** You are given a directed graph $G = (V, E)$. A vertex v is called nice if there is a directed path from v to each of the other vertices in G (there could be multiple nice vertices in a graph). Suppose we run DFS on G and let w be the vertex with the highest finish time (recall that the finish time of a vertex u in DFS is the time at which the procedure DFS(u) ends, i.e., u is popped from the stack). Prove that if G has a nice vertex, then w is a nice vertex.

2. **(6 marks)** Let $G = (V, E)$ be a directed graph where each edge has positive length. Given a vertices s and t give an $O((m + n) \log n)$ time algorithm to check whether there is a unique shortest path from s to t. In other words, if $d(s, t)$ denotes the shortest distance from s to t, the algorithm should output TRUE if there is exactly one path from s to t of length $d(s, t)$, otherwise it should output FALSE. The graph is given in the adjacency list representation. Here m denotes the number of edge and n denotes the number of vertices in the graph.

3. **(4 marks)** Suppose we are given n numbers in an array where only t values are "out of order", i.e., if we remove these t values, then the rest of the array is sorted in ascending order. Give an $O(tn)$ time algorithm to sort the array in ascending order. Note that the algorithm does not know the identity of these t values.

4. **(6 marks)** You are given two sorted arrays of length n and m of respectively. Give an algorithm takes as input a number k, and outputs the k^{th} smallest number in the union of the two arrays. Running time should be $O(\log n + \log m)$. Note that k can be any number between 1 and n, and the proposed algorithm should have the desired time complexity independent of the value of k. Specifically, a solution which has a linear time dependence on k is not allowed.

5. **(5 marks)** Consider a binary tree where every internal node has exactly 2 children. For each node v, let $w(v)$ denote the number of external nodes (i.e., leaves) below the subtree rooted at v. Note that if v is an external node, then $w(v)$=1. We call such a tree "2-balanced" if for every internal node v with children v_L and v_R, the ratio $w(v_L)/w(v_R)$ lies in the range $[1/2, 2]$. Show that any 2-balanced tree with n nodes has height $O(\log n)$.
6. **(5 marks)** You are maintaining a skip list on n keys in the ascending order. Show how you can modify this data-structure by adding one more field at each node such that in addition to the originally supported operations of (i) Search for a key, (ii) Insert a key, (iii) Delete a key, another operation can be supported which is: given a number k, return the k^{th} smallest key. Each of these operations should take expected $O(\log n)$ time.

7. **(a) (2 marks)** Recall that in the 2-dimensional range tree data-structure on n points, each internal node in the range tree stores an associated tree (which is a balanced tree that uses y coordinates of the points below the node). Now suppose that only the nodes with depth 0, 2, 4, 6, ... have an associated tree structure – the rest of the nodes do not store any associated tree (the root is at depth 0). A range query is specified by a rectangle and the output is supposed to be the number of input points inside the rectangle. Show how to answer a range query with this modified range tree data-structure.

   (b) (3 marks) Now suppose only the nodes at depth 0, j, $2j$, $3j$... store an associated tree data-structure (here j is some integer which may not be constant). What will be the space requirement of this data-structure and how much will be the query time to answer a range query? Express your answer using $O()$ notation in terms of n and j. Justify your answer.

Solutions:

1. Suppose G has a nice vertex v and let w be as defined in the question. We know that w has the highest finish time. We define an interval for each vertex which starts at the time we visit the vertex and ends at the finish time of the vertex. Then we know that these intervals are nested. Since w has the highest finish time, two possibilities arise:
(i) Intervals for v and w are disjoint and the one for v lies to the left of w: But then we know that there cannot be a path from v to w (as discussed in class). This contradicts the fact that v is nice.
(ii) Interval for v is nested inside the interval for w: In this case, there is a path from w to v. Since there is a path from v to every other vertex, it follows that there is a path from w to every other vertex as well. Thus, w is a nice vertex.

2. We run Dijkstra's algorithm starting from the vertex s . With every vertex v, we maintain a binary variable $U[v]$ which is 1 iff there is a unique path from s to v among the vertices explored so far. The binary variable $U[v]$ is initialized to 1 for all vertices. As in Dijkstra's algorithm, we maintain a variable $D[v]$ with each vertex and a set S of explored vertices.

   The algorithm proceeds as in Dijkstra's algorithm, except for one change. When we remove the vertex $v \notin S$ with the minimum $D[v]$ value, we perform the following steps for each edge (v, w), $w \notin S$:

```
(*)  If $D[v] + l(v,w) < D[w]$
       Update $D[w] = D[v] + l(v,w)$, $U[w] = U[v]$
(**) Else if $D[v] + l(v,w) = D[w]$
       Update $U[w] = 0$
```

   The running time is clearly same as that of Dijkstra's algorithm and hence $O((m + n) \log n)$.

   Proof of correctness. The algorithm maintains the following invariants. Given a set S of explored vertices (i) $D[v]$ is the length of the shortest path from s to v where every vertex other than v in the shortest path belongs to S (this invariant is same as in class), and (ii) $U[v]$ is 1 iff there is a unique shortest path from s to v of length $D[v]$ which only involves the vertices in S. Invariant (i) follows from class. We show the second invariant by induction on the number of steps. Suppose step (*) is executed. Then before v was added to S, the shortest path so far to w has length more than $D[v] + l(v,w)$. Thus, after adding v to S and updating $D[w]$, any path of length $D[w]$ from s to w using vertices in S must have the last edge (v,w). Hence, there is a unique shortest path of length $D[w]$ to w iff the same holds for v. The argument for (**) is clear: we have found two paths (one using v and one without v) of length $D[w]$ to w.

(Alternate Solution)

Find the shortest paths $d(s, v)$ from s all vertices v and $d(v, t)$ from all vertices v to t. Let $P := (s = v_0, v_1, ..., v_k = t)$ be a shortest path from s to t. Check the following for each vertex v_i, $i < k$, on this path: if there is a neighbour w of v_i other than v_{i+1} such that $d(v_i, t) = l(v_i, v_{i+1}) + d(v_{i+1}, t)$, then declare FALSE. If no such case happens, declare TRUE.

Proof: There are two parts to it. If there is a unique shortest path, then the algorithm will declare TRUE. If there are two shortest paths then the algorithm will declare FALSE. Consider the first part. It is clear that if there is a unique shortest path, then the above-mentioned condition cannot hold– otherwise there are two shortest paths to t. Now, we prove the converse. Suppose there is another shortest path Q from s to t. Consider the first edge in P which is not present in Q - let this edge be (v_i, v_{i+1}). Let w be the next vertex after v_i in Q. Then the above condition holds for these vertices. Thus the algorithm declares FALSE.

3. We run insertion sort algorithm, where in the i^{th} iteration, we compare the element $A[i]$ with $A[i - 1]$, $A[i - 2]$.... till find an element smaller than $A[i]$. Correctness follows from the correctness of insertion sort (done in class). We now argue about the running time. Let S be the set of t elements which are out of order (the algorithm does not know the set S). There are two parts to the argument (i) When $A[i]$ happens to be in S, this iteration may take $O(n)$ time. Hence, the total time for such cases is $O(tn)$, (ii) When $A[i] \notin S$, we will stop this iteration the moment we compare $A[i]$ with an element $A[j]$ not lying in S (because the relative order of $A[j]$ and $A[i]$ is preserved in the original array and hence, $A[j] < A[i]$). Thus, this iteration will take at most $O(t)$ steps. Overall, the total time such such cases is $O(tn)$.

4. Our algorithm will be recursive and follow the divide and conquer method as in binary search. Let A and B be the two arrays. In $O(1)$ step, we shall divide one of the two arrays in half. It follows that the overall running time will be $O(\log n + \log m)$. Suppose we want to find the k^{th} smallest number. We compare the middle elements of the two arrays : (i) If $A[n/2] < B[m/2]$: The first $n/2$ elements of A and the first $m/2$ elements of B are less than or equal to $B[m/2]$. Thus, if $k \leq (n + m)/2$, we know that the desired element cannot be in the second half of B (it could still be in the second half of A) $-$ otherwise there will be at least $(m + n)/2$ elements smaller than it. Thus we can recurse on A and the second half of B with the same value of k. Similarly, if $k > (n + m)/2$, we know that the desired element cannot be in the first half of A. Thus we recurse on the second half of A and all of B, but update k to $k - n/2$.
(ii) If $A[n/2] > B[m/2]$: This case is similar to above.

As in binary search, we maintain two indices for each array A and B, and move one of these to the middle of the current array in each iteration.

5. Let $h(n)$ be the **maximum** height of a 2-balanced tree on **at most** n external nodes . By definition, $h(n)$ is a monotonically ascending function of n. We write a recurrence for $h(n)$. Clearly, $h(1) = 1$. Suppose T is a tree on $n > 1$ external nodes with height $h(n)$. Let r be the root of T and r_L, r_R be its left and right children. Let T_L and T_R be the sub-trees rooted at r_L and r_R respectively. We claim that both of these trees have at most $2n/3$ external nodes . Suppose not. Assume without loss of generality that T_L has more than $2n/3$ external nodes, and hence, T_R has at most $n/3$ external nodes. But then, the ratio $w(r_L)/w(r_R) > 2$, which is a contradiction. Therefore, we get the recurrence. It is easy to verify that its solution is $T(n) = O(\log n)$.
6. For a key i, let $pos(i)$ denote its position in the bottom level of the skip list (i.e., $pos(i)$ denotes the number of keys less than or equal to i). We cannot maintain $pos(i)$ because updating it after each insert or delete will take too much time. However for any node v (at any level) storing a key i, we store the following information: let w be the node after v at this level and suppose w stores the key j. Then v stores a variable $x(v) = pos(j) - pos(i)$, i.e., the number of keys between i and j (including these two keys). Let us see how each of the operations can be performed:
(i) Search: no change is needed here.
(ii) Insert: Again it proceeds as before, but when we insert a new key i, then each node v storing this key needs to update the $x()$ value of its predecessor in the list at which v resides (and update $x(v)$ as well). This can be done because when we search for key i, we knows the position of all the keys that it encounters during the search process (it can calculate these using the $x()$ values stored in the nodes). This will take $O(1)$ time at each level, resulting in expected $O(\log n)$ time.
(iii) Delete: The details of delete are as in a skip list. As in the case of insert, when we remove a node v, we need to update the the $x()$ value of the predecessor.
(iv) k^{th} smallest: As in the case of search, we start from the top left. As we go down to a node v, we also maintain the position of the key stored in it. Thus, we know whether to go to the next node in this level or go down. The running time analysis is exactly as in the case of insert.

7. As done in class, we shall first find at most $2 \log n$ nodes in the primary tree (based on the x-coordinate) would like to report the number of leaf nodes below

each of these nodes which satisfy that y-coordinate criteria. Let S denote the set of these $2\ \log n$ (or less) nodes in the primary tree. For any node $v\ \in S$, if v lies at an even depth, we know that we are maintaining the auxiliary tree data-structure for v, and hence, we can search in this tree . But if the depth of v is odd, then we need to look at both the children of v, and perform search in the auxiliary tree stored at these two nodes. Thus, the running time remains $O(\log^2 n)$.

In the second case, the argument is similar. Let S be as above. When we want to find all the leaf nodes below v that satisfy the y-coordinate criteria, we need to go down to the descendants of v which are storing the auxiliary trees and check in each one of these. Thus if v is at depth $i\ mod\ j$, then we need to look at descendants of v a depth $j - i$ below it. There could be 2^{j-i} such descendants, and we need to check in the auxiliary tree in each one of these. The set S will have at most $2 \log\ n/j$ nodes which are at depth $i\ mod\ j$. Hence, the running time will be:

$\sum_{i<j} |S|/j \cdot 2^{j-i} \cdot \log\ n = O(2^j \log^2 n/j)$. The space complexity will be $O(n\ \log\ n/j)$, because we are storing the auxiliary data-structure at every j^{th} level only.