CS21003 Algorithms – I, Autumn 2012–13

End-semester Test

Maximum marks: 55 Time: 20-Nov-2012 (AN) Duration: 3 hours

Roll no: _____ **Name:** _____

[Write your answers in the question paper itself. Be brief and precise. Answer all questions.]

1. A point P in the two-dimensional plane is said to *dominate* another point Q if their x- and y-coordinates satisfy the conditions: $x(P) \geqslant x(Q)$ and $y(P) \geqslant y(Q)$. A *maximal point* in a collection C of n points $P_1, P_2, \ldots, P_n$ is a point P_i which is not dominated by any other point P_j in the collection.

(a) Demonstrate by an example that a collection may have multiple maximal points. **(5)**

Solution All the points in the collection $(1, n), (2, n-1), (3, n-2), \ldots, (n, 1)$ are maximal.

(b) Propose a worst-case $\mathrm{O}(n \log n)$-time algorithm to compute *all* the maximal points in a given collection C of n points. For simplicity, assume that the points in C do not have equal x- or y-coordinates. **(5)**

Solution The steps of the algorithm are described below.

1. Sort the input points in the decreasing order of their y-coordinates. Rename this sorted list as $P_1, P_2, \ldots, P_n$.
2. Print "P_1 is a maximal point".
3. Set $M = P_1$.
4. For $i = 2, 3, 4, \ldots, n$ (in that order), repeat: {
5. If $(x(P_i) \geqslant x(M))$, then: {
6. Print "P_i is a maximal point".
7. Set $M = P_i$.
8. } /* End of *if* */
9. } /* End of *for* */

Step 1 of this algorithm takes $\mathrm{O}(n \log n)$ time. The remaining part of the algorithm takes $\Theta(n)$ time.

(c) All the maximal points in C constitute a subcollection C_1 called the *first maximal layer* of C. All the maximal points in $C \setminus C_1$ constitute another subcollection C_2 called the *second maximal layer* of C. In general, if $C_1, C_2, \ldots, C_k$ are the first k maximal layers of C, then the $(k+1)$*-st maximal layer* of C is the set C_{k+1} of all maximal points in $C \setminus \bigcup_{i=1}^{k} C_i$. Using an appropriate reduction algorithm, prove that no algorithm can compute the maximal layers of a collection C of n points in $o(n \log n)$ time in the worst case. **(5)**

Solution We reduce the problem of sorting to the problem of computing maximal layers. Let $A = [a_1, a_2, \ldots, a_n]$ be an array (of integers or floating-point numbers), that we want to sort. We assume that the elements a_i are distinct from one another. We generate n points $P_1, P_2, P_3, \ldots, P_n$, where $P_i = (-a_i, -a_i)$ for all i. We run an algorithm H for computing the maximal layers of the collection of these n points. The smallest a_i gives the first maximal layer consisting of the only point P_i. If a_j is the second smallest element in A, then the second maximal layer consists of the only point P_j, and so on. In particular, if H outputs $P_{i_1}, P_{i_2}, \ldots, P_{i_n}$, then the sorted version of A is $a_{i_1}, a_{i_2}, \ldots, a_{i_n}$, where $a_{i_j} = -x(P_{i_j})$.

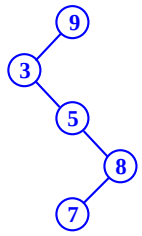
Clearly, the reduction algorithm runs in $\Theta(n)$ time. Therefore, if H runs in $o(n \log n)$ time in the worst case, we get a worst-case $o(n \log n)$-time algorithm for sorting A. On the contrary, we know that any (comparison-based) sorting algorithm must take $\Omega(n \log n)$ time in the worst case.

Roll no: _____ **Name:** _____

2. Let us insert the (distinct) integers $a_1, a_2, \ldots, a_n$ in that sequence in an initially empty binary search tree. We use the standard insertion procedure without height balancing. We know that if the integers $a_1, a_2, \ldots, a_n$ appear in sorted order (increasing or decreasing), the resulting binary search tree is of maximum possible height $n-1$, and the insertion of all the n items takes $\Theta(n^2)$ running time.

(a) Demonstrate by an example that even an unsorted sequence may result in a binary search tree having the maximum possible height. Take $n = 5$. **(5)**

Solution Insertion of the integers $9, 3, 5, 8, 7$ in an initially empty binary search tree gives the adjacent tree of height $5 - 1 = 4$.



(b) You are given a sequence of integers $a_1, a_2, \ldots, a_n$ in an array. You need to decide whether inserting these integers in that sequence leads to a height of $n-1$ of the binary search tree. Propose a worst-case $O(n)$-time algorithm to solve the problem. Note that if you actually build the tree, you end up in a $\Theta(n^2)$ running time in the worst case. **(5)**

Solution In order that we get a chain of length $n-1$, we need each non-leaf node to have only one child. This limits the allowed values of a_i given that $a_1, a_2, \ldots, a_{i-1}$ have already resulted in a binary search tree of height $i-2$. For example, if $a_2 < a_1$, then a_2 is inserted as the left child of the root a_1. But then, the root cannot have a right child, that is, none of $a_3, a_4, \ldots, a_n$ can be larger than a_1. The following pseudocode implements these observations.

1. If $(n \leqslant 2)$, return *true*.
2. Initialize $lower\_limit = -\infty$ and $upper\_limit = +\infty$.
3. For $i = 2, 3, 4, \ldots, n$, repeat: {
4. If $(a_i < lower\_limit)$ or $(a_i > upper\_limit)$, return *false*.
5. If $(a_i < a_{i-1})$, set $upper\_limit = a_{i-1}$.
6. If $(a_i > a_{i-1})$, set $lower\_limit = a_{i-1}$.
7. } /* End of *for* */
8. Return *true*.

3. (a) Design a worst-case $O(n)$-time algorithm to generate a random permutation of $0, 1, 2, \ldots, n-1$. **(5)**

Solution The following algorithm assumes that a random integer in the range $0, 1, 2, \ldots, m-1$ can be computed in $O(1)$ time for any positive integer m.

1. Initialize an array P of size n as $P[i] = i$ for all $i = 0, 1, 2, \ldots, n-1$.
2. For $i = 0, 1, 2, \ldots, n-2$, repeat: {
3. Let j be a random integer in the range $0, 1, 2, \ldots, n-i-1$.
4. Swap $A[i]$ with $A[i+j]$.
5. } /* End of *for* */
6. Return P.

(b) Now, your task is to generate a random undirected *connected* graph with n vertices and m edges (n and m are supplied to you, we must have $m \geqslant n-1$). Propose an efficient algorithm for solving this problem. Analyze the worst-case running time of your algorithm. **(5)**

Solution Let us name the vertices as $0, 1, 2, \ldots, n-1$. We first compute a random spanning tree on these vertices. We then randomly add the remaining $m-n+1$ edges. In order that a random edge is not repeatedly chosen for addition to the graph, we use a strategy similar to Part (a).

1. Initialize the edge set E to be empty.
2. Compute in P a random permutation of $0, 1, 2, \ldots, n-1$ (use Part (a)).
3. For $i = 1, 2, 3, \ldots, n-1$, repeat: {
4. Compute a random integer j in the range $0, 1, 2, \ldots, i-1$.
5. Add the edge $(P[i], P[j])$ to E.
6. } /* End of *for* */
7. Store in a list L all possible edges between n vertices.
8. Delete from L the edges that are already added to E.
9. Let the size of L be s (in fact, $s = \frac{n(n-1)}{2} - (n-1) = \frac{(n-1)(n-2)}{2}$ at this moment).
10. While ($|E| < m$), repeat: {
11. Choose a random integer j in the range $0, 1, 2, \ldots, s-1$.
12. Add the j-th edge in the list L to E.
13. Delete the j-th edge from L (and decrease the size of L by 1).
14. } /* End of *while* */
15. Return E.

By Part (a), a random permutation P can be computed in $\Theta(n)$ time. The *for* loop (Lines 3–6) computes a random spanning tree in $\Theta(n)$ time. Subsequently, we initialize the list L (Lines 7–9) in $\Theta(n^2)$ time. The *while* loop is executed $m-n+1$ times. Each iteration of the loop can be done in $\Theta(1)$ time (for example, if L is implemented as an array as in Part (a)). Since $m = O(n^2)$, the running time of the above algorithm is $\Theta(n^2)$.

Roll no: _____ **Name:** _____

4. Let $G = (V, E)$ be a connected undirected graph with each edge $(u, v) \in E$ carrying a cost $c(u, v)$.

(a) A *maximum spanning tree* of G is a spanning tree T of G such that the sum of the costs of the edges in T is as large as possible. Modify Kruskal's algorithm to compute a maximum spanning tree of G. Write only the modifications (not the entire Kruskal algorithm). What is the running time of your algorithm? **(5)**

Solution One possibility is to convert the given graph G to another weighted graph G' on the same vertex and edge sets as G but with the edge costs redefined as $c'(u, v) = -c(u, v)$. We run the original Kruskal algorithm on G'.

Another possibility is to sort the edges in E in the decreasing order of their costs, and run the original Kruskal algorithm on this edge list. That means that Kruskal's algorithm will now attempt to add edges in the decreasing sequence of their costs.

Both the above variants perform essentially the same task. In other words, apart from ties, both the variants perform exactly the same sequence of edge-addition attempts. Clearly, a maximum spanning tree in G is a minimum spanning tree in G'.

The input can be modified (to get G') in $O(|E|)$ time. The second variant changes the sorting order from increasing to decreasing. Therefore, the running time of either variant is $O(|E| \log |V|)$.

(b) A *bottleneck edge* in a spanning tree T of G is an edge in T with the largest cost among the edges in T. A *maximum bottleneck spanning tree* of G is a spanning tree of G such that the bottleneck edge in T is of cost as large as possible. Propose a worst-case $O(|E|)$-time algorithm to compute a maximum bottleneck spanning tree of G. **(5)**

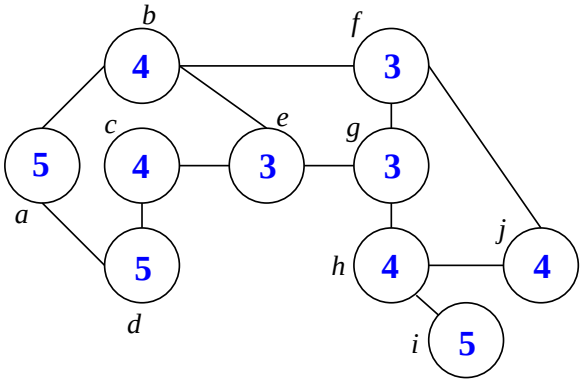
Solution Let c^* be the maximum cost of an edge in E. Any maximum bottleneck spanning tree will contain an edge e of cost c^*.

If we run the modified Kruskal algorithm of Part (a), the maximum spanning tree will contain an edge e of the maximum cost c^*. Indeed, such an edge is the first to be added to the spanning tree. However, this takes $O(|E| \log |V|)$ running time.

In order to achieve $O(|E|)$ running time, we may generate any spanning tree of G, containing an edge e of the maximum cost c^*. In particular, we may run a BFS or DFS traversal on G from one of the endpoints of e. We also make sure that e is an edge in the BFS or DFS tree generated by our traversal. Since such a traversal takes $O(|E| + |V|)$ time, $|E| \geqslant |V| - 1$, and the maximum cost c^* and an edge e of this cost can be located in $O(|E|)$ time, this traversal-based algorithm runs in $O(|E|)$ time.

5. Let $G = (V, E)$ be an undirected graph with a positive cost $c(u, v)$ associated with each edge $(u, v) \in E$. For $u, v \in V$, let $d(u, v)$ be the cost of a shortest u, v path. The *eccentricity* of a vertex $u \in V$ is defined as $\epsilon(u) = \max_{v \in V} d(u, v)$. The set of vertices in G having the minimum eccentricity is called the *center* of G, denoted $C(G)$.

(a) Find the eccentricities of all the vertices in the following undirected graph. Assume that each edge has cost 1 (so the distance $d(u, v)$ is the *length* of the shortest u, v path). Write the eccentricities inside the circles representing the vertices. Identify the center of G. **(5)**



$C(G) =$ $\{e, f, g\}$

(b) Propose an efficient algorithm to compute the center of G. What is the running time of your algorithm? **(5)**

Solution Run the Floyd-Warshall algorithm on G. The final matrix $D^{(n-1)}$ computed by this algorithm reveals the center of G. More precisely, the eccentricity of vertex i is the maximum entry in the i-th row of $D^{(n-1)}$. Then, we locate the minimum eccentricity, and report all vertices with this minimum eccentricity.

The Floyd-Warshall algorithm takes $\Theta(n^3)$ time. The eccentricities of all vertices can be computed in $\Theta(n^2)$ time. The minimum eccentricity can be computed and the vertices with this minimum eccentricity can be identified in $\Theta(n)$ time. Thus, the overall running time of this algorithm is $\Theta(n^3)$.

Note that if G is not connected, the eccentricity of each vertex is ∞, and $C(G)$ is the entire vertex set of G. The above algorithm will work for this case too. Nevertheless it makes sense to talk about the center of connected graphs only.

Roll no: _____ **Name:** _____

For rough work and leftover answers