*Math 363: Theory of Algorithms*
*Instructor: Sreekar M. Shastry*
*Solutions to the Midterm Examination*
*2012-03-02-Fri, 1530-1700*

- There are 6 problems. Each problem is worth 5 points. The maximum score is 30 points.

(1) Let us recall the following definitions.

A *queue* is a set from which we extract elements in first-in-first-out (FIFO) order: we select elements in the same order in which they were added. The procedure that adds an element is `Enqueue(Q,x)` and the procedure that removes an element is `Dequeue(Q)`.

A *stack* is a set from which we extract elements in last-in-first-out (LIFO) order: we select the most recently added element. The procedure that adds an element is `Push(S,x)` and the procedure that removes an element is `Pop(S)`.

A *max-priority queue* is a data structure that maintains a set of elements $P$, where each element $v \in P$ has an associated value $\texttt{Key}(v)$ that denotes the priority of the element $v$; higher keys represent higher priorities. Priority queues support the following procedures: `Insert(P,x)` adds the element $x$ to $P$, `Maximum(P)` returns the element of $P$ with the largest key, `ExtractMax(P)` removes and returns the element of $P$ with the largest key (it returns $-\infty$ if $P$ is empty), `IncreaseKey(P,x,k)` increases the value of element $x$'s key to the new value $k$, which is assumed to be at least as large as $x$'s current key value. Likewise one can define a *min-priority queue* with procedures `Insert, Minimum, ExctractMin, DecreaseKey`.

(a) Show how to implement a queue with a priority queue.
(b) Show how to implement a stack with a priority queue.
(For both (a) and (b), you may choose which type of priority queue to use.)

*Solution.* (a) We implement a queue by means of a min-priority queue $P$. Namely, we define `Enqueue(Q,x)` := `Insert(P,x)` and `Dequeue(Q)` := `ExctractMin(P)`. In greater detail, given the $i$th element $x_i$, we insert it into the min-priority queue with key value $i$:

$$\texttt{Key}(x_i) = i.$$

Then the first call to `Dequeue(Q)` returns $x_1$, the second call returns $x_2$ and so on, which demonstrates the required FIFO property.

(b) We implement a stack by means of a max-priority queue $P$. We define `Push(S,x)` := `Insert(P,x)` and `Pop(Q)` := `ExctractMax(P)`. In detail, given the $i$th element $x_i$, we insert it into the max-priority queue with key value $i$:

$$\texttt{Key}(x_i) = i.$$

Suppose that the stack has $N$ elements at the time we issue a call to pop. Then the first call to `Pop(Q)` returns $x_N$; if no other elements have been pushed onto the stack, the second call returns $x_{N-1}$ and so on. On the other hand, if some new element has been pushed onto the stack after the first call to `Pop`, the next call to `Pop` will return the most recently pushed element. This demonstrates the required LIFO property.

Let us remark on the following distinction between (a) and (b).

In the case of the FIFO queue, we have the following invariant: at all times the set of key values present in the min-priority queue will be a consective block of integers of the form $k, k+1, k+2, \ldots, k+m$ for some positive integers $k$ and $m$.

In the case of a LIFO stack, at all times we are only guaranteed that the set of key values in the max-priority queue is a strictly increasing set of integers $k, k+i_1, k+i_2, \ldots k+i_m$.

(2) We call a graph $G = (V, E)$ a *near-tree* if it is connected and has at most $n + 8$ edges, where $n := |V|$. Give an algorithm with running time $O(n)$ that takes a near-tree with edge costs and returns a minimum spanning tree of $G$. You may assume that all of the edge costs are distinct.

*Solution.* The crucial subroutine is the following procedure DetectCycleByDepthFirstTraversal which we will call simply $F$. It is a modification of the depth first traversal algorithm. We maintain and update an associative array $P$ which contains the partial DFS tree. (We remark that even though $G$ is an undirected graph, once we specify a root vertex, the DFS tree is a directed graph.) Thus $P[t] = s$ means that node $t$ has node $s$ as parent on the DFS tree.

$F$ takes as input a graph $G$, a node $s$, and the node which is the parent of $s$ in the depth first tree $P$. Note that $P$ is constructed by $F$ and grows with each subsequent call to $F$.

$F$ returns the first nontrivial, non-tree edge $(u, v)$, else it returns NIL to indicate that the graph $G$ is already a tree. By nontrivial non-tree edge, we mean an edge that completes a nontrivial cycle in the undirected graph $G$; a trivial cycle in an undirected graph is one of the form $s \to t \to s$.

```
1: Let P be a global variable, the associative array representing the DFS tree
2: s is the root node if and only if P[s] = NIL
3: procedure F(G, s)
4:     Mark s as explored
5:     for each edge (s, t) incident to s do
6:         if t is marked explored and t ≠ P[s] then
7:             return (s, t) // this edge completes a nontrivial cycle in G
8:         end if
9:         if t is not marked explored then
10:            Let P[t] := s
11:            F(G, t, P[t])
12:        end if
13:    end for
14:    return NIL
15: end procedure
```

Now, with the above subroutine in hand, we may describe the idea of the algorithm to find an MST in a near-tree. We use $F$ to search for a cycle. If there is no cycle, then the graph is a tree and we are done. Otherwise, we delete the heaviest edge from the cycle. We do this 9 times.

```
1: procedure NearTreeMST(G)
2:     Choose an initial node s
3:     for i = 1, 2, ..., 9 do
4:         Let (u, v) = F(G, s)
5:         if (u, v) = NIL then
6:             return // G is already a tree
7:         end if
8:         Let w = LeastCommonAncestor(u, v)
9:         Let e1 be the heaviest edge on the path from u to w
10:        Let e2 be the heaviest edge on the path from v to w
11:        Delete from G the heaviest of e1, e2, (u, v) // to use the Cycle Property of MSTs
12:    end for
13: end procedure
```

To analyze the complexity, write $n := |V|, m := |E|$, so that we have $n - 1 \leq m \leq n + 8$ by connectedness and the near-tree property. The running time of $F$ is bounded by that of DFS which is $O(n + m) = O(n)$ on a near-tree. LeastCommonAncestor takes $O(n)$ time (see page 96 of KT). Finding the heaviest edge on a path takes $O(n)$ time, thus each of lines 9 and 10 takes $O(n)$. The total running time of this algorithm is thus $O(9n) = O(n)$, as required.

(3) Dr. Foo drives his car from Pune to Goa. When full, his car's gas tank holds enough fuel to enable him to drive for $n$ kilometers. His gps navigation system tells him in advance the distances between the gas stations on his route. The doctor wishes to make as few gas stops as possible along the way. Give an efficient algorithm by which Dr. Foo may determine at which gas stations to stop so as to achieve his goal of minimizing the total number of gas stops. Prove that your algorithm is correct and determine its running time.

*Solution.* The optimal strategy goes as follows. Starting with a full tank of gas, Foo should go to the farthest gas station at position $P_1$ from Goa which is at most $n$ miles from Pune. Starting from that point, he should go to the farthest gas station $P_2$ which is within $n$ miles of position $P_1$. And so on.

Put differently, at each gas station, Foo should ask whether he can make it to the next gas station without stopping at this one. If he can, skip this one. This description shows that the algorithm is $O(m)$, where $m$ is the total number os stations.

(i) Now, consider an optimal solution with $s$ stations which stops first at the $k$th station. Then we claim that the rest of the optimal solution must be an optimal solution to the subproblem of the remaining $m-k$ stations. If this were not the case, we could find a solution to the problem on $m-k$ stations which stopped at $< s-1$ stations and we could use this to construct an optimal solution on $m$ stations with $< s$ stops, contradicting our supposition.

(ii) Moreover, any optimal solution must choose as the first station the farthest station (the $k$th station, say) which is within $n$ miles of Pune. For if not and we had chosen the $j$th station with $j < k$, then upon leaving the $j$th station we could get no further than if we stopped at the $k$th station. Thus stopping at station $j < k$ results in a solution which can do no better than stopping at the $k$th station.

Taken together, (i) and (ii) prove that our algorithm is correct.

(4) We are given two sets $A$ and $B$, each consisting of $n$ positive integers. You may reorder the sets in any manner you choose. After reordering, write $a_i$ for the $i$th element of $A$ and $b_i$ for the $i$th element of $B$. The payoff obtained from the chosen orderings is

$$\prod_{i=1}^{n} a_i^{b_i}.$$

(a) Give an algorithm which maximizes the payoff.
(b) Prove the correctness of your answer to part (a) and determine its running time.

*Solution.* Sort $A$ and $B$ into decreasing order.

To prove that this yields the optimal solution, we argue as follows. Let $i < j$ and consider $a_i^{b_i}$ and $a_j^{b_j}$. Let us show that $a_i^{b_i} a_j^{b_j} \geq a_i^{b_j} a_j^{b_i}$. Since we have sorted $A$ and $B$ into decreasing order, $a_i \geq a_j$ and $b_i \geq b_j$. Since $a_i, a_j > 0$ and $b_i - b_j \geq 0$, we have $a_i^{b_i - b_j} \geq a_j^{b_i - b_j}$. Multiplying both sides by $a_i^{b_j}$ yields $a_i^{b_i} a_j^{b_j} \geq a_i^{b_j} a_j^{b_i}$.

Since multiplication of real numbers is commutative, sorting $A$ and $B$ into increasing order also would have worked. What the above argument actually demonstrated was that we must choose compatible orderings on $A$ and $B$ in the sense that for both sets, for all $i$, the $i$th element must be the $i$th element from the top or that for all $i$, the $i$th element is the $i$th element from the bottom.

The running time is the running time the sorting algorithms we know (e.g. mergesort): $O(n \log n)$.

(5) This problem is about Huffman coding.
(a) Explain the relationship between prefix codes and binary trees.
(b) Show that the binary tree corresponding to the optimal prefix code must be full.

*Solution.* (a) This is in section 4.8 of KT.
(b) This is proposition 4.28 on page 168 of KT.

(6) This problem is about Dijkstra's algorithm for computing the *length* of the shortest path from a given root to any other vertex.
   (a) State Dijkstra's algorithm.
   (b) Give a simple example of a directed graph with some edges having negative length for which Dijkstra's algorithm produces an incorrect answer.

   *Solution.* (a) This is in KT on page 138.
   (b) In the following figure, we start from s. Dijkstra's algorithm tells us that the distance from s to y is 1, but going the long way around shows that the distance is 0.