Solutions to Practice Problems

Problem -1. True or False, and Justify

Circle **T** or **F** for each of the following statements to indicate whether the statement is true or false, respectively. If the statement is correct, briefly state why. If the statement is wrong, explain why. Your justification is worth more points than your true-or-false designation.

T F To determine if two binary search trees are identical trees, one could perform an inorder tree walk on both and compare the output lists.

Solution: False. Take the two search trees on the elements $\{1, 2\}$: both trees produce the list $1, 2$, but the trees are different.

T F Constructing a binary search tree on n elements takes $\Omega(n \log n)$ time in the worst case (in the comparison model).

Solution: True. Suppose we could always construct a binary search tree in $o(n \log n)$ time; then we could sort in $o(n \log n)$ time by putting the elements in a BST and performing an $O(n)$-time tree walk. This contradicts our comparison-based sorting lower bound of $\Omega(n \log n)$.

T F A greedy algorithm for a problem can never give an optimal solution on all inputs.

Solution: False. Prim's algorithm for minimum spanning trees is a counter-example: it greedily picks edges to cross cuts, but it gives an optimal solution on all inputs.

T F Suppose we have computed a minimum spanning tree of a graph and its weight. If we make a new graph by doubling the weight of every edge in the original graph, we still need $\Omega(E)$ time to compute the cost of the MST of the new graph.

Solution: False. Consider sorting the edges by weight. Doubling the weights does not change the sorted order. But this means that Kruskal's and Prim's algorithm will do the same thing, so the MST is unchanged. Therefore the weight of the new tree is simply twice the weight of the old tree and can be computed in constant time if the original MST and weight are known.

T F 2-3-4 trees are a special case of B-trees.

Solution: True. They are the case when $t = 2$.

T F You have n distinct elements stored in an augmented red-black tree with an extra field per node containing the number of elements in the subtree rooted at that node (including itself). The rank of an element is the number of elements with value less than or equal to it (including itself). The best algorithm for finding the rank of a given element runs in linear time.

Solution: We can augment a red-black tree to support order-statistic queries in $O(\log n)$ time. (For each node, keep track of the number of nodes in it's subtree. This information is easily maintained on rotation, insert, and delete, and is good enough to search for an order-statistic.)

T F Let $G = (V, E)$ be a connected, undirected graph with edge-weight function $w : E$ to reals. Let $u \in V$ be an arbitrary vertex, and let $(u, v) \in E$ be the least-weight edge incident on u; that is, $w(u, v) = \min \{w(u, v') : (u, v') \in E\}$. (u, v) belongs to some minimum spanning tree of G.

Solution: True. Consider any MST. Assume it doesn't contain (u, v), or else we're done. Since it is a spanning tree, there must be a path from u to v. Let (u, x) be the first edge on this path. Delete (u, x) from the tree and add (u, v). Since (u, v) is a least weight-edge from u, we did not increase the cost, so as long as we still have a spanning tree, we still have a minimum spanning tree. Do we still have a spanning tree? We broke the tree in two by deleting an edge. So did we reconnect it, or add an edge in a useless place? Well, there is only one path between a given two nodes in a tree, so u and v were separated when we deleted e. Thus our added edge does in fact give us back a spanning tree.

T F It is possible to do an amortized analysis of a heap (plain old stored-in-an-array-discussed-at-the-time-of-heapsort heap) to argue that the amortized time for insert is $O(\log n)$ (where n is the number of items in the heap at the time of insertion) and the amortized time for extract-max is $O(1)$.

Solution: Define a potential function $\Phi = c \sum_{i=1}^{n} \log i$ when there are n items in the heap. The amortized cost of insertion is $O(\log n)$, $O(\log n)$ for the actual insert and $c \log n$ for the increase in potential. Assuming we pick c large enough, the amortized cost of extract-max is $O(1)$, $O(\log n)$ actual cost minus the $c \log n$ decrease in potential. The starting potential is 0 and after that it is non-negative, so that's it.

Problem -2. Minimum and Maximum Spanning Trees

(a) It can be shown that in any minimum spanning tree (of a connected, weighted graph), if we remove an edge (u, v), then the two remaining trees are each MSTs on their respective sets of nodes, and the edge (u, v) is a least-weight edge crossing between those two sets.

These facts inspire the 6.046 Staff to suggest the following algorithm for finding an MST on a graph $G = (V, E)$: split the nodes arbitrarily into two (nearly) equal-sized sets, and recursively find MSTs on those sets. Then connect the two trees with a least-cost edge (which is found by iterating over E).

Would you want to use this algorithm? Why or why not?

Solution: This algorithm is actually *not correct*, as can be seen by the counterexample below. The facts we recalled are essentially irrelevant; it is their *converse* that we would need to prove correctness. Specifically, it *is* true that every MST is the combination of two sub-MSTs (by a light edge), but it is *not* true that every combination of two sub-MSTs (by a light edge) is an MST. In other words, it is not safe to divide the vertices arbitrarily.

A concrete counterexample is the following: vertices A, B, C, D are connected in a cycle, where $w(A, B) = 1$, $w(B, C) = 10$, $w(C, D) = 1$, and $w(D, A) = 10$. A minimum spanning tree consists of the first three edges and has weight 12. However, the algorithm might divide the vertices into sets $\{B, C\}$ and $\{A, D\}$. The MST of each subgraph has weight 10, and a light edge crossing between the two sets has weight 1, for a total weight of 21. This is not an MST.

(b) Consider an undirected, connected, weighted graph G. Suppose you want to find the *maximum* spanning tree. Give an algorithm to find the maximum spanning tree.

Solution: You can make a copy of G called G' in which the weight of every edge is replaced with its negation (i.e. $w(u, v)$ becomes $-w(u, v)$) and compute the minimum spanning tree $MST(G')$ of the resulting graph using Kruskal's algorithm (or any other MST algorithm). The resulting tree corresponds to a maximum weight spanning tree for the original graph G. Note that if it were not a maximum weight spanning tree, then the actual maximum weight spanning tree of G corresponds to a minimum spanning tree with less weight than $MST(G')$, which is a contradiction.

Problem -3. Finding Shortest Paths

We would like solve, as efficiently as possible, the single source shortest paths problem in each of the following graphs. (Recall that in this problem, we must find a shortest path from a designated source vertex to every vertex in the graph.) For each graph, state which algorithm (from among those we have studied in class and section) would be best to use, and give its running time. If you don't know the name of the algorithm, give a *brief* description of it. If the running time depends on which data structures are used in the implementation, choose an arbitrary data structure.

(a) A weighted directed acyclic graph.

Solution: First Topological Sort and then dynamic programming (Refer to DAG-SHORTEST-PATHS on Page 536 in CLR) - $O(V+E)$

(b) A weighted directed graph where all edge weights are non-negative; the graph contains a directed cycle.

Solution: Dijkstra' Algorithm with Fibonacci Heaps - $O(E \lg V)$

(c) A weighted directed graph in which some, but not all, of the edges have negative weights; the graph contains a directed cycle.

Solution: Bellman-Ford's Algorithm - $O(VE)$

Problem -4. Roots of a graph

A **root** of a directed graph $G = (V, E)$ is a node r such that every other node $v \in V$ is reachable from r.

(a) [3 points] Give an example of a graph which does not have a root.

Solution:
Refer Figure 1

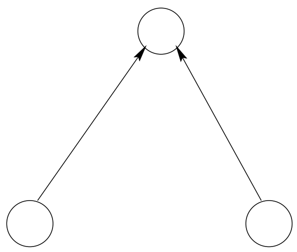

Figure 1: Problem 4-(a)

(b) [10 points] Prove the following claim.

Consider the forest constructed by running depth-first-search (DFS) on the graph G. Let T be the last tree constructed by DFS in this forest and let r be the root of this tree (i.e, DFS constructed T starting from the node r.) If the graph G has a root, then r is a root of G.

Solution: None of the nodes in the trees other than T of the forest constructed by DFS reach r (or r would not be the root of a new tree). If any node in T is a root of the graph, so is r. Thus proved.

(c) [7 points] Using the result proved in **(b)**, give an $O(|V|+|E|)$-algorithm which when given a graph G, finds a root of the graph if one exists, and outputs NIL otherwise.

Solution: From above, we know that if G has a root, then the root of the last tree output by DFS must be a root. Thus, we first run DFS and then check whether the last root r reaches out to all nodes. This can be done by doing DFS again starting with r and checking if the forest has only one root. Clearly, this algorithm takes $O(|V|+|E|)$ time.

```
FIND-ONE-ROOT(G)
    for each vertex u ∈ V[G]
        do color[u] ← WHITE
    for each vertex u ∈ V[G]
        do if color[u] = WHITE
              then last-root ← u            ▷ store current root
                   DFS-VISIT(u)
    r ← last-root
    TEST-ROOT(r)
```

We test whether r is a root by recoloring all nodes white, calling DFS-VISIT(r) and then verifying that every node in G is black. This will show that every node is reachable from r.

(d) [5 points] Suppose you are a given a root r of a graph G. Give an $O(|V| + |E|)$-algorithm to find all the roots of the graph.

(Hint: u is a root of G iff r is reachable from u).

Solution: We now find all other roots as follows. A node u is a root iff u can reach r. This is because if u is a root then it reaches r and conversely, if u reaches r then it reaches every other node as well. We can find all nodes that reach r by reversing the edges of G and then starting a DFS from r in the reversed graph.

```
FIND-ALL-ROOTS(G)
    REVERSE-GRAPH(G)
    for each vertex u ∈ V[G]
        do color[u] ← WHITE
    DFS-VISIT(r)
    for each vertex u ∈ V[G]
        do if color[u] = BLACK
              then OUTPUT(u)                ▷ u is a root
```

It is easy to see that the procedure REVERSE-GRAPH(G) takes $O(V+E)$ time. Therefore, the algorithm to find all roots takes $O(V + E)$ as well.

Problem -5. Test-Taking Strategies

Consider (if you haven't already!) a quiz with n questions. For each $i = 1, \ldots, n$, question i has integral point value $v_i > 0$ and requires $m_i > 0$ minutes to solve. Suppose further that no partial credit is awarded (unlike this quiz).

Your goal is to come up with an algorithm which, given v_1, v_2, …, v_n, m_1, m_2, …, m_n, and V, computes the minimum number of minutes required to earn at least V points on the quiz. For example, you might use this algorithm to determine how quickly you can get an A on the quiz.

(a) Let $M(i, v)$ denote the minimum number of minutes needed to earn v points when you are restricted to selecting from questions 1 through i. Give a recurrence expression for $M(i, v)$.

We shall do the base cases for you: for all i, and $v \leq 0$, $M(i, v) = 0$; for $v > 0$, $M(0, v) = \infty$.

Solution: Because there is no partial credit, we can only choose to either do, or not do, problem i. If we do the problem, it costs m_i minutes, and we should choose an optimal way to earn the remaining $v - v_i$ points from among the other problems. If we don't do the problem, we must choose an optimal way to earn v points from the remaining problems. The faster choice is optimal. This yields the recurrence:

$$M(i, v) = \min \{m_i + M(i-1, v-v_i), M(i-1, v)\}$$

(b) Give pseudocode for an $O(nV)$-time dynamic programming algorithm to compute the minimum number of minutes required to earn V points on the quiz.

Solution:

FASTEST$(\{v_i\}, \{m_i\}, V)$
  $\triangleright$ fill in $n \times V$ array for M; base case first
  for $v \leftarrow 1$ **to** V
    $M[0, v] \leftarrow \infty$
  $\triangleright$ now fill rest of table
  for $i \leftarrow 1$ **to** n
    for $v \leftarrow 1$ **to** V
        $\triangleright$ compute first term of recurrence
        if $v - v_i \leq 0$
          $a \leftarrow m_i$
        else
          $a \leftarrow m_i + M[i-1, v-v_i]$
        $\triangleright$ fill table entry i, v
        $M[i, v] \leftarrow \min\{a, M[i-1, v]\}$
  return $M[n, V]$

Filling in each cell takes $O(1)$ time, for a total running time of $O(nV)$. The entry $M[n, V]$ is, by definition, the minimum number of minutes needed to earn V points, when all n problems are available to be answered. This is the quantity we desire.

(c) Explain how to extend your solution from the previous part to output a list S of the questions to solve, such that $V \leq \sum_{i \in S} v_i$ and $\sum_{i \in S} m_i$ is minimized.

Solution: In each cell of the table containing value $M(i, v)$, we keep an additional bit corresponding to whether the minimum was $m_i + M(i-1, v-v_1)$ or $M(i-1, v)$, i.e. whether it is fastest to do problem i or not. After the table has been filled out, we start from the cell for $M(n, V)$ and trace backwards in the following way: if the bit for $M(i, v)$ is set, we include i in S and go to the cell for $M(i-1, v-v_i)$; otherwise we exclude i from S and go to the cell for $M(i-1, v)$. The process terminates when we reach the cell for $M(i, v')$ for some $v' \leq 0$.

It is also possible to do this without keeping the extra bit (just by looking at both possibilities and tracing back to the smallest one).

(d) Suppose partial credit is given, so that the number of points you receive on a question is proportional to the number of minutes you spend working on it. That is, you earn v_i/m_i points per minute on question i (up to a total of v_i points), and you can work for fractions of minutes. Give an $O(n \log n)$-time algorithm to determine which questions to solve (and how much time to devote to them) in order to receive V points the fastest.

Solution: A greedy approach works here (as it does in the fractional knapsack problem, although the problems are not identical). Specifically, we sort the problems in descending value of v_i/m_i (i.e., decreasing "rate," or "bang for the buck"). We then iterate over the list, choosing to do each corresponding problem until V points are accumulated (so only the last problem chosen might be left partially completed). The running time is dominated by the sort, which is $O(n \log n)$. Correctness follows by the following argument: suppose an optimal choice of problems only did part of problem j and some of problem k, where $v_j/m_j > v_k/m_k$. Then for some of the points gained on problem k, there is a faster way to gain them from problem j (because j hasn't been completed). This contradicts the optimality of the choice that we assumed. Therefore it is safe to greedily choose to do as much of a remaining problem having highest "rate" as needed.