Practice Quiz 2 (Solutions)

Problem 1. 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 Professors Indor and Tidyk 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.
Many solutions were concerned only with the running time of Goldemaine's algorithm, which is $O(E \log V)$, the same as Kruskal's or Prim's (using a binary heap). They concluded that this algorithm offers no advantages, but this is not the whole story: it might be easier to implement, or more memory-efficient, or have a smaller hidden constant in the big-O running time. The key issue here is correctness.

Problem 2. 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. Many solutions argued that identical trees produce identical output lists, but this is the *converse* of the given statement (and is true).

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 $\Omega(n \log n)$.

Many solutions said that each insertion requires $\Omega(\log n)$ time, because $\log n$ is the height of the tree. There are two errors in this argument: first, the height of the tree isn't *always* $\Omega(\log n)$; during the first few insertions it is constant. Also, this answer doesn't rule out other, possibly more clever, ways of constructing binary search trees (which might not insert elements one-by-one). A correct proof must prove that *every* conceivable algorithm runs in at least the stated time.

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 A maximum spanning tree (i.e., a spanning tree that maximizes the sum of the weights) can be constructed in $O(E \log V)$ time.

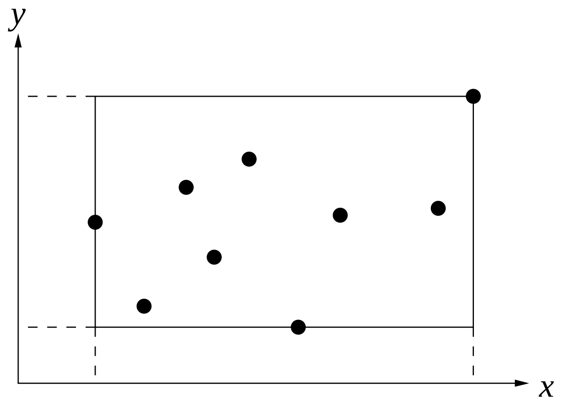
Solution: True. Negate each cost and find a minimum cost spanning tree, by any algorithm shown in the book or in class. (Prim's or Kruskal's) These algorithms make no assumption that the costs are positive, and finding the minimum of the negated costs clearly finds the maximum of the original costs.

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.

Problem 3. Bounding Boxes

An important notion in computer graphics is the **bounding box** of a set of objects, which is the smallest rectangle (where each side is parallel to the x- or y-axis) such that all the objects are contained in the rectangle (including its border). Note that a bounding box can be described by the two x-coordinates of its two vertical edges, and the two y-coordinates of its two horizontal edges. For example, the figure below shows a collection of points and their bounding box, which can be represented by the indicated pairs of x- and y-coordinates.



In this problem we are interested in designing a dynamic set of points in the plane which supports the standard INSERT and SEARCH operations. It must also support an operation B-BOX(x_0, x_1), which computes the bounding box of *only* those points in the set whose x-coordinates are between x_0 and x_1 (inclusive). That is, it should return a pair (y_0, y_1) containing the y-coordinates of the bounding box's horizontal edges.

(a) What data structure would you augment to implement this data structure, and what auxiliary data would you keep? Explain why the running times of INSERT and SEARCH are $O(\log n)$, where n is the number of points in the set.

Solution: We use a 2-3-4 tree, ordered by x-coordinate, with y-coordinates breaking ties (that is, the node with smaller y-coordinate appears first – without this feature, SEARCH cannot be implemented efficiently). In addition, we augment each node with the minimum and maximum y-coordinate, taken over all nodes its entire subtree. This information can be maintained without affecting the $O(\log n)$ running time of the tree operations, because it is only a function of the node's own y-coordinate and the auxiliary data of the node's children (Theorem 14.1 of CLRS).

We note one addition to the operation of SEARCH: when searching for a point (x, y), if a node has the desired x-coordinate but a different y-coordinate, then we use the y-coordinate to decide which child to recurse on.

(b) Explain how to implement B-BOX(x_0, x_1) so that it runs in $O(\log n)$ time. You may assume that x_0 and x_1 are the x-coordinates of some points in the set.

Solution: The algorithm for B-BOX(x_0, x_1) is as follows: search for the "first" (i.e., leftmost) node n_0 of a point having x-coordinate x_0, and the "last" node n_1 of a point having x-coordinate x_1. Then consider the paths from n_0 and n_1 to the root; at some node these paths converge into one. We will only care about the parts of the paths which are distinct from each other. Whenever the n_0-path goes "right," i.e., from u to v, where u is a child of v and v has another child to the right of u, we flag subtrees of the parent that are to the right of u. Whenever the n_1-path goes "left," we perform a symmetric operation. The minimum value y_0 returned is the minimum of all auxiliary minima for each flagged subtree. The maximum value y_1 is computed similarly, but with the auxiliary maxima. One subtlety is that the above flagging procedure needs to be modified to avoid marking nodes in the paths from n_0 and n_1 to the root; this is an issue only for the nodes at the level below the first common parent.

The desired running time holds because we flag $O(\log n)$ different nodes and subtrees, since the paths have length $O(\log n)$. The algorithm is correct because, as we have argued before, all nodes greater than n_0 (respectively, less than n_1) are covered exactly by the flagged subtrees. The intersection of these two sets is covered exactly by the distinct parts of the paths.

Problem 4. 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 $v_1, v_2, \ldots, v_n, m_1, m_2, \ldots, m_n$, 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)
  ▷ fill in n × V array for M; base case first
  for v ← 1 to V
     M[0, v] ← ∞
  ▷ now fill rest of table
  for i ← 1 to n
     for v ← 1 to V
          ▷ compute first term of recurrence
          if v − v_i ≤ 0
             a ← m_i
          else
             a ← m_i + M[i − 1, v − v_i]
          ▷ fill table entry i, v
          M[i, v] ← 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).

Some solutions kept in each cell an entire length-n bit vector of which problems should be done for the fastest time. This is inefficient, because it requires $\Theta(n^2 V)$ space (n bits for each cell), and therefore $\Theta(n^2 V)$ time to fill in the table.

(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.

Some solutions claimed that this problem is exactly the fractional knapsack problem, but this is not strictly true. In that problem, we try to maximize the value of what is put in a fixed-size sack, while in this problem we try to minimize the size of a sack needed to carry a fixed value. However, the proofs of the greedy-choice property are very similar in both cases.