Quiz 2

- Do not open this quiz booklet until you are directed to do so. Read all the instructions first.
- **For this quiz, you *need not* provide rigorous proofs of correctness. Instead, give informal arguments for why you believe your algorithms are correct. Pseudocode is only required when explicitly indicated, but you may include it if it clarifies your answers.**
- When the quiz begins, write your name on every page of this quiz booklet.
- The quiz contains 4 multi-part problems. You have 80 minutes to earn 80 points.
- This quiz booklet contains 11 pages, including this one. Two extra sheets of scratch paper are attached. Please detach them before turning in your quiz.
- This quiz is closed book. You may use one handwritten A4 or $8\frac{1}{2}'' \times 11''$ crib sheet. No calculators or programmable devices are permitted.
- Write your solutions in the space provided. If you need more space, write on the back of the sheet containing the problem. Do not put part of the answer to one problem on the back of the sheet for another problem, since the pages may be separated for grading.
- Do not waste time and paper rederiving facts that we have studied. It is sufficient to cite known results.
- Do not spend too much time on any one problem. Read them all through first, and attack them in the order that allows you to make the most progress.
- Show your work, as partial credit will be given. You will be graded not only on the correctness of your answer, but also on the clarity with which you express it. Be neat.
- Good luck!

Problem	Points	Grade	Initials
Name (all pages)	1		
1	20		
2	18		
3	17		
4	24		
Total	80		

Name: Solutions _____

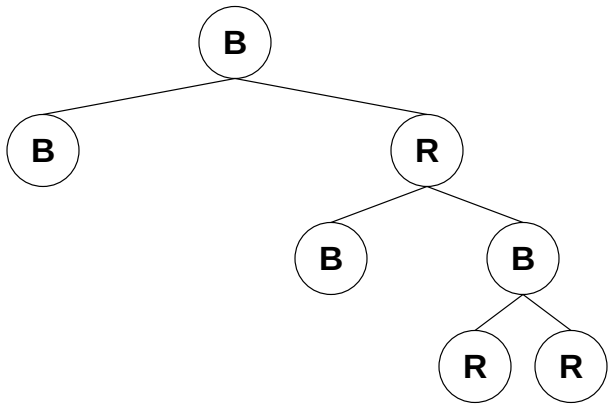
Circle the name of your recitation instructor:

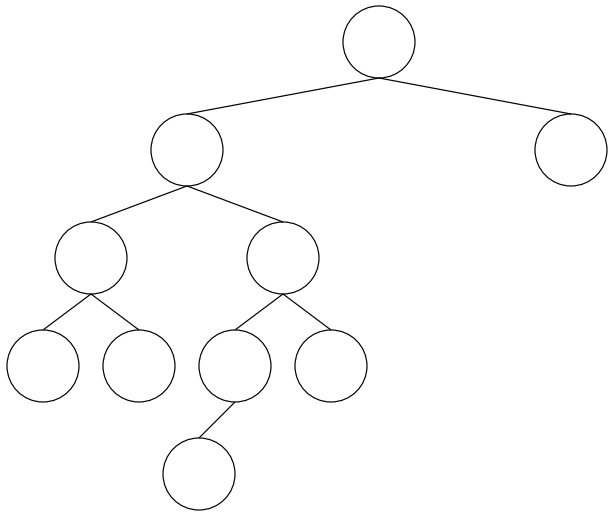
Moses Jen Steve

Problem 1. Short Answer [20 points]

Give *brief*, but complete, answers to the following questions.

(a) In each of the trees below, label the nodes of the tree to make it a red-black tree, or prove that no such labelling exists. Use the following convention: label the black nodes with a 'B,' and the red nodes with an 'R'.





Solution: The second tree cannot be colored, because it has a root-to-leaf path of 5 nodes, and one of 2 nodes. The latter path can have at most 2 black nodes. On the other hand, since the root node must be colored black, and there cannot be two red nodes in a row, the path with 5 nodes must have at least 3 black nodes. This would be a violation of the black-height property.

(b) Name some similarities and differences between the *Divide and Conquer* and *Dynamic Programming* paradigms. For each paradigm, name a problem to which the paradigm is applied.

Solution: Similarities: both techniques analyze a problem recursively, and solve subproblems to arrive at a final answer. Differences: DP takes advantage of overlapping subproblems for efficiency, exploits the "optimal substructure" of problems, and usually fills in a table of values of an objective function in optimization problems.
Divide-and-conquer is applied to many problems: sorting, matrix multiplication, search in a sorted array, etc. Dynamic programming is applied to several as well: matrix-chain multiplication (note: *not* matrix multiplication), longest common subsequence, knapsack problems, etc.
Some solutions said that divide-and-conquer re-solves overlapping subproblems, which isn't usually the case. It is more accurate to say that divide-and-conquer is more appropriate when subproblems tend not to overlap.

(c) For each of the dynamic set operations below, fill in the letter (from the table on the right) corresponding to its running time (in terms of the number of elements n in the set). Choose the *strongest* running times that apply. *Note:* the SUCCESSOR(x) operation is given a pointer to the element x in the set, and does not have to first find the element using SEARCH.

Structure	**Operation**	**Time**
Treap	SEARCH	D
	SUCCESSOR	D
Binary Search Tree	SEARCH	H
	SUCCESSOR	H
Red-Black Tree	SEARCH	C
	SUCCESSOR	C
Skip List	SEARCH	D
	SUCCESSOR	A
van Emde Boas (with $u = n^c$, constant c)	SEARCH	F
	SUCCESSOR	F

A	$O(1)$ worst-case
B	$O(1)$ expected
C	$O(\log n)$ worst-case
D	$O(\log n)$ w/ high prob.
E	$O(\log n)$ expected
F	$O(\log \log n)$ worst-case
G	$O(\log \log n)$ w/ high prob.
H	$O(n)$ worst-case

(d) Recall the following facts, proved in lecture: 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 Professor Goldemaine 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 [18 points]

Circle **T** or **F** for each of the following statements, and briefly explain why. The better your argument, the higher your grade, but be brief. Your justification is worth more points than your true-or-false designation.

(a) 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).

(b) 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.

(c) T F It is possible to sort n distinct integers from the range $1 \ldots n^{\log n}$ in $O(n \log \log n)$ time.

Solution: True. Insert all the elements into a van Emde Boas structure, then output the elements in sorted order by starting with the minimum and making $n - 1$ SUCCESSOR calls. Each INSERT and SUCCESSOR call takes $O(\log \log(n^{\log n})) = O(\log(\log n)^2) = O(2 \log \log n) = O(\log \log n)$ time, for the desired total running time.
It is not correct to use RADIX-SORT with base n, because then each number has $d = \log_n n^{\log n} = \log n$ digits, for a running time of $\Theta(nd) = \Theta(n \log n)$.

(d) T F The expected amount of space used by a skip list on n elements is $O(n)$.

Solution: True. Each element appears in $2 = O(1)$ rows in expectation, and each occurrence requires $O(1)$ space for the element and pointers. By linearity of expectation, the space taken by all elements is $O(n)$ in expectation. Alternatively, we expect each level to have half as many elements as the one below it, for space $O(n + n/2 + n/4 + \cdots) = O(2n) = O(n)$ by geometric summation.
It is not correct to answer "false" because the skip list has $O(\log n)$ levels (w/ high prob.), and each level has $O(n)$ elements, for a total of $O(n \log n)$ space. While this bound is true, it is not the tightest possible (i.e., it is too loose of an approximation of the actual space used).

(e) T F The height of a randomly built binary search tree is $O(\log n)$ with high probability; therefore in randomized quicksort, every element is involved in $O(\log n)$ comparisons with high probability.

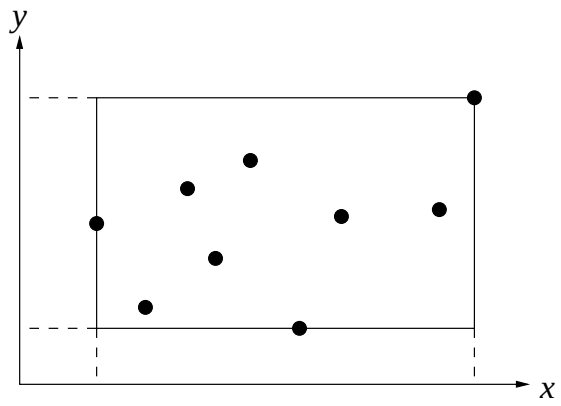
Solution: False. In randomized quicksort, the first pivot is compared to *every* other element, and there are $n - 1 = \Omega(n)$ of them. Therefore every element has at least a $1/n$ chance of being involved in $\Omega(n)$ comparisons.
Alternately, the root of the randomly-built binary search tree is compared to every other element. (More generally, an element is compared with every element in its subtree, as well as all its ancestors.) These comparisons are the same ones done by randomized quicksort.
Many solutions regurgitated the fact that there is a connection between the comparisons performed by quicksort and those performed in building a random BST. This is true, but irrelevant. The *height* of a node in the BST has nothing to do with the number of elements it is compared to in the corresponding quicksort.

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

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

Problem 3. Bounding Boxes [17 points]

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) [7 points] 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 red-black 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 whether to recurse on the left or right child.

(b) [10 points] 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 a left child to its parent), we flag the parent and its right subtree. Whenever the n_1-path goes "left," we flag the parent and its left subtree. The minimum value y_0 returned is the minimum of all the flagged parents' y-coordinates (including n_0), and the auxiliary minimums for each flagged right subtree. The maximum value y_1 is computed similarly, but with the auxiliary maximums for each flagged parent and subtree.

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 parents and their right (respectively, left) subtrees. The intersection of these two sets is covered exactly by the distinct parts of the paths.

Problem 4. Test-Taking Strategies [24 points]

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) [8 points] 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) [5 points] 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) [4 points] 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) [7 points] 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.

SCRATCH PAPER — Please detach this page before handing in your quiz.

SCRATCH PAPER — Please detach this page before handing in your quiz.