Quiz 1

- Do not open this quiz booklet until directed to do so. Read all the instructions on this page.
- When the quiz begins, write your name on the top of every page of this quiz booklet.
- You have 120 minutes to earn a maximum of 120 points. Do not spend too much time on any one problem. Skim them all first, and attack them in the order that allows you to make the most progress.
- **You are allowed one double-sided letter-sized sheet with your own notes**. No calculators, cell phones, or other programmable or communication devices are permitted.
- Write your solutions in the space provided. Pages will be scanned and separated for grading. If you need more space, write "Continued on S1" (or S2, S3, S4) and continue your solution on the referenced scratch page at the end of the exam.
- Do not waste time and paper rederiving facts that we have studied in lecture, recitation, or problem sets. Simply cite them.
- When writing an algorithm, a **clear** description in English will suffice. Pseudo-code is not required. Be sure to argue that your **algorithm is correct**, and analyze the **asymptotic running time of your algorithm**. Even if your algorithm does not meet a requested bound, you **may** receive partial credit for inefficient solutions that are correct.
- **Pay close attention to the instructions for each problem**. Depending on the problem, partial credit may be awarded for incomplete answers.

Problem	Parts	Points
0: Information	2	2
1: Tree Exploration	4	25
2: Sum 99	1	20
4: 2D Binary Search	2	20
3: Kerberos Confusion	1	20
5: Winter Commute	3	33
Total		120

Name: _____

School Email: _____

Problem 0. [2 points] **Information** (2 parts)

(a) [1 point] Write your name and email address on the cover page.
Solution: OK!

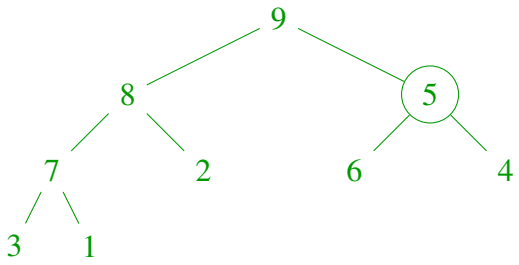
(b) [1 point] Write your name at the top of each page.
Solution: OK!

Problem 1. [25 points] **Tree Exploration** (4 parts)

(a) [5 points] The following integer array would comprise a binary max-heap, except that one integer key **does not satisfy** the max-heap property. Draw the binary tree associated with the array and circle the integer that violates the max-heap property.

```
[9,8,5,7,2,6,4,3,1]
```

Solution:

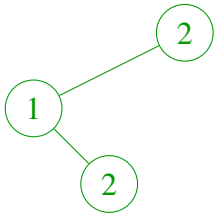


Rubric:

- 3 points for drawing of tree
- 2 points for circling key 5

(b) [5 points] Given a binary search tree containing duplicate keys, where two or more nodes contain the same key v, will **every** node containing key v be the child or parent of another node containing v? If yes, prove the claim. If no, provide a counter example, i.e., draw a binary search tree for which the claim does not hold.

Solution: No, a node containing key v might not be adjacent to another node containing key v. Below is a minimal counter example:

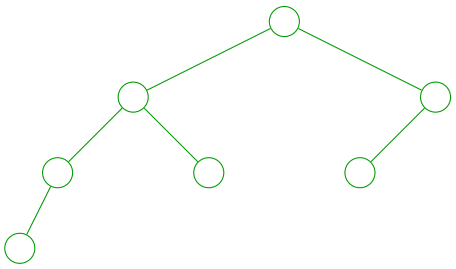


Rubric:

- 2 points for stating that claim is false
- 3 points for a counter example

(c) [5 points] Consider the set T of all binary trees containing exactly 7 nodes which also satisfy the **AVL Property**. What is the maximum height of any tree in T? Draw a tree from T that achieves maximum height. (Recall that the **height** of a tree is the number of **edges** contained in a longest root-to-leaf path.)

Solution: No binary tree from T may have height 4: a binary tree on 7 nodes of height 4 must have a root-to-leaf path containing 5 nodes, containing one of the root's children having subtree height 3. The root's **other child**'s subtree may contain at most the remaining 2 nodes, allowing maximum height ≤ 1, so the root can not satisfy the AVL property because heights 3 and ≤ 1 must differ by more than one. A tree achieving **maximum height 3** is shown below.



Rubric:

- 2 points for max height three (no proof needed)
- 3 points for drawing of binary tree having asserted max height
- Partial credit may be awarded

(d) [10 points] Given two nodes from a **complete** binary tree on n nodes (i.e. full except for the lowest level), describe an $O(\log n)$ time algorithm to return the node farthest from the root whose subtree contains both nodes. State whether your algorithm achieves a **worst-case**, **expected**, and/or **amortized** bound.

Solution: For each query node, construct a list of ancestors by walking up the tree following parent pointers, appending a reference to each ancestor in order to the end of a linked list or dynamic array. The number of ancestors for any node in a complete binary tree is $O(\log n)$, so each list can be constructed in $O(\log n)$ time. The last node in each list will be the root of the tree as the root is the highest ancestor of all nodes in the tree. Scan backwards from the root simultaneously in both lists to find the longest suffix present in both lists. By definition, the left-most node in the common suffix will be the lowest common ancestor of the two query nodes. The scan process at most $O(\log n)$ nodes in each list, so the algorithm runs in **worst-case** $O(\log n)$ time.

An alternative expected $O(\log n)$ algorithm stores the ancestors of one node in a hash table, and then walks up to the root from the other, checking whether an ancestor of the second node exists in the ancestor hash table of the first.

Rubric:

- 6 points for a correct algorithm
- Partial credit may be awarded
- 2 points running time analysis
- 2 points for stating **worst-case**

Problem 2. [20 points] **Sum 99** (1 part)

Given an array of n **distinct** integers, we would like to identify three integers from the array that sum to 99, if such a triple exists. If the array contains multiple such triples, a correct algorithm may return any such triple. For example, for array `[11, 6, 45, 93, -5, 50, 43]`, a correct algorithm will return either `[11, 45, 43]`, `[11, 93, -5]`, or `[6, 50, 43]` (triples may be returned in arbitrary order). Describe an $O(n^2)$ time algorithm to identify a triple of elements that sum to 99, if one exists. State whether your algorithm achieves a **worst-case**, **expected**, and/or **amortized** bound.

Solution: Create a hash table to store the integers, and insert each into the table one-by-one. Then, iterate through the $n(n-1)$ distinct pairs of elements from the array, and for each pair (a, b), check if $c = 99 - a - b$ exists in the hash table. If it does not exist, we have not found a triple that sums to 99. Otherwise, if the integer c exists in the hash table, it corresponds to an integer c from the original input. Check to see if $a = c$ or $b = c$. If it does not, return the triple (a, b, c). Otherwise, the triple is not distinct, so move on to process the next pair. (Many students forgot to check for duplicates elements in a triple). For pair of the input, we perform one lookup operation and one insert operation on the hash table, each taking $O(1)$ time in expectation. Thus, the algorithm runs in expected $O(n^2)$ time.

A common incorrect algorithm was to hash all the triples. Not only does this not answer the problem, it takes $O(n^3)$ time. If you have $O(n^3)$, simply compute the sum of each triple and compare it to 99; no need to use hashing.

A harder worst-case $O(n^2)$ solution involves first sorting the array in $O(n \log n)$ time, and then carefully checking all possible pairs in worst-case $O(n^2)$ time. Worst-case is not required for this problem.

Rubric:

- 14 points for a correct algorithm
- Partial credit may be awarded
- 2 points for checking that triples are distinct
- 2 points for running time analysis
- 2 points for stating correct running-time qualifiers
- Maximum 18 points if counting or radix sort are used, making unjustified assumptions on range of integer inputs.

Problem 3. [20 points] **2D Binary Search** (2 parts)

A two-dimensional $n \times n$ matrix A is **two-dimensionally sorted** if every row and column is sorted, i.e., each element $A[i][j]$ is at least as large as any item in the same row to the left, or in the same column above it in the matrix. For example, the matrix below is two-dimensionally sorted. In this problem, **you may assume that all elements in the matrix are unique**.

```
A = [[ 1,  4,  7, 11, 15],
     [ 2,  5,  8, 12, 19],
     [ 3,  6,  9, 16, 22],
     [10, 13, 14, 17, 24],
     [18, 21, 23, 26, 30]]
```

(a) [10 points] Given an $n \times n$ square matrix that is two-dimensionally sorted, a brute force algorithm can search for an element in the matrix in $O(n^2)$ time. Describe an algorithm to solve two-dimensional search, asymptotically faster than $O(n^2)$, but asymptotically slower than $O(n)$. State whether your algorithm achieves a **worst-case**, **expected**, and/or **amortized** bound. (You will be asked for an $O(n)$ algorithm in the next part. We would like a **different** algorithm that improves on brute force.)

Solution: Multiple solutions to this. Easiest is probably repeated binary search: for each sorted row, binary search for the item. There are n rows and each binary search takes $O(\log n)$ time, so repeated binary search takes **worst-case** $O(n \log n)$ time.

Another solution is to pick a middle element along the diagonal, and determine if it is larger or smaller than the item. If it is the item, we are done. If it is larger, the item cannot be in the bottom right quadrant, and if it is smaller, the item cannot be in the top left quadrant. In either case, recursively search the other three quadrants. Let $T(n)$ represent the worst-case time for this algorithm to search for an item in an $n \times n$ matrix. Then the recurrence relation is given by:

$$T(n) = 3 \cdot T(n/2) + O(1),$$

which by the master theorem yields **worst-case** $O(n^{\log_2 3})$ time, where $1 < \log_2 3 < 2$.

Rubric:

- 7 points for a correct algorithm
- Partial credit may be awarded
- 2 points for running time analysis
- 1 point for stating and achieving a **worst-case** bound

(b) [10 points] One can perform two-dimensional search in linear time by starting a search from a corner. Give an $O(n)$ time algorithm for two-dimensional search. State whether your algorithm achieves a **worst-case**, **expected**, and/or **amortized** bound.

Solution: Start with the top right element. If the element is smaller than the element you are looking for, you can eliminate the entire first row and move one position down. If the element is larger, you can eliminate the entire last column and move one position left. Obviously, if the element is the top right element, you can stop.

The neat thing about the above strategy is that it either eliminates a row or a column in each step. So you will find the element you are searching for in at most $2n$ steps for a $n \times n$ matrix, or you will determine that it does not exist, so the algorithm runs in **worst-case** linear time.

A common mistake was to start with the minimum element $A[0][0]$ and move along the diagonal. Find k such that $A[k][k] < val < A[k+1][k+1]$. If the $<$ is a $\leq$ we are done. Then, search $A[k+1][0:k]$ and $A[0:k][k+1]$ for val. This is a maximum of $3n$ comparisons. Unfortunately, this strategy fails for the example below, when searching for `8`.

```
[[1, 2,  3,  4 ],
 [2, 3,  4,  5 ],
 [6, 7,  9, 10 ],
 [8, 9, 10, 11 ]]
```

While the same approach does not seem to work when starting from the minimum or maximum element, a different linear time algorithm starts with the minimum element, moves rightward until you find an element greater than the element you are searching for, and then potentially move downward and then leftward iteratively to find the element. This requires $3n$ moves in the worst case. In the example above, you go all the way to `4`, and then start moving downward to `11` and move left to find `8`.

Rubric:

- 7 points for a correct linear time algorithm
- Partial credit may be awarded
- 2 points for running time analysis
- 1 point for stating and achieving a **worst-case** bound

Problem 4. [20 points] **Kerberos Confusion** (1 part)

MIT administrators have trouble distinguishing between similar-looking Kerberos usernames. A Kerberos username is a sequence of at most eight alpha-numeric characters. The administrators would like to determine how many Kerberos usernames are **similar** to another: two Kerberos usernames are similar if one can be transformed to the other by replacing a single character. For example, username `fun6006` is similar to usernames `fun6046` and `fuu6006`, but not username `6006fun`. Given a list of n unique Kerberos usernames, design a **worst-case** $O(n)$ algorithm to find every username from the list that is similar to another username from the list. You may receive significant partial credit for an **expected** $O(n)$ time algorithm.

Solution: We assume the number of possible characters k to be small compared to the number of Kerberos usernames, $k \ll n$. A Kerberos username will be similar to another when it is different only by a single character. Suppose a similar pair differs at character i. If we sort the usernames character-by-character, but prioritize character i last below all others, similar usernames differing at character i will be adjacent to each other in the sort as all other characters in a similar pair are the same by definition. We can perform such a sort in **worst-case** $O(n + k)$ time, by applying radix sort with a counting sort subroutine on the letters, prioritizing character i last. Then scan through the sorted list comparing adjacent usernames. If two adjacent usernames are identical except at character i, then they are similar. To keep track of whether usernames have been found to be similar, append the character 0 to the end of each username. If a scan finds a pair that are similar, change their appended character to 1. We repeat this process eight times, each time prioritizing character $i \in \{0, \ldots, 7\}$ last. After all eight radix sorts and scans have been complete, loop through the sorted list in linear time and append to an output list any username for which a 1 is appended. We perform a constant number of radix sorts and linear scans, so the entire algorithm takes **worst-case** linear time.

Note that algorithms relying on a constant factor 36^8 lost points. Because there are 36^8 possible Kerberos usernames, $n \leq 36^8$, so although a constant, 36^8 is larger than the size of any valid input. While this is an abuse of asymptotic notation because our problem size is bounded, it is still useful to think of algorithms that achieve 'linear' performance, even for large but bounded n.

Rubric:

- 12 points for correct radix sort (max 9 points if counting sort/hashing)
- Partial credit may be awarded:
- 4 points for keeping track of similar (free if hashing)
- 2 points for running time analysis
- 2 point for stating and achieving a **worst-case** bound (not hashing)
- (Hashing solutions can receive max 15 points)

Problem 5. [33 points] **Winter Commute** (3 parts)

The recent snowy weather in Cambridge is causing trouble for students living on Massachusetts Avenue who bike to MIT. Biking in a bike lane containing too much snow can be dangerous: a student could be injured, or even worse, they might miss class!

A group of enterprising students embed sensors at one foot intervals along Mass. Ave. to measure the height of snow at their location along the bike path. The collected data is upload to an online database every few hours. If the height of snow in the Mass. Ave. bike lane is too large, as measured by any sensor between a biker's house and MIT, then the biker will not be able to go to school. Each student living along Mass. Ave. in Cambridge (only north of MIT) wants to be able to query the current maximum height of snow in the bike path between MIT and their home, located a known number of feet from MIT along Mass. Ave.

(a) [5 points] Design a data structure to process and store new data from the n sensors into the online database in worst-case $O(n)$ time, that supports student queries in worst-case $O(1)$ time.

Solution: When a new batch of sensor data is received, store the sensor data in an array A, with the measurement from sensor located i feet from MIT at array index i. If the data does not come sorted by distance from MIT, sort the measurements by distance in worst-case linear time using counting sort, since the range of distances is $O(n)$. Then, construct another array B, where the value stored at array index i represents the maximum measurement from the prefix $A[:i]$. Each index $B[i]$ can be computed from $i = 0$ to $i = n$ in $O(1)$ time via the relation $B[i] = \max(B[i-1], A[i])$, the maximum of prefix $A[:i]$ is either the maximum of prefix $A[:i-1]$ or the value stored at $A[i]$, so this data structure can be produced in worst-case linear time. To query a student living j feet from MIT, return the value stored at $B[j]$ in worst-case constant time.

Rubric:

- 3 points for a correct database description
- Partial credit may be awarded
- 1 point for description and running time analysis of preprocessing
- 1 point for description and running time analysis of query

(b) [5 points] It turns out that the database does not update frequently enough for many students, as snow plows periodically remove (and sometimes add!) snow to the bike lane. The enterprising students outfit the snow plows with trackers that can send to the database, updates of the following form: h inches of snow have been uniformly added or removed from the bike lane at and between distances a and b, measured in feet from MIT along Mass. Ave. Adapt your data structure from part (a) to update snow estimates based on a snow plow update in worst-case $O(n)$ time.

Solution: Store and update the same data structure as part (a). For a query (h, a, b), add h to each $A[i]$ for i from a to b. Then update B starting from $B[a]$ by using the same process as part (a), i.e., $B[i] = \max(B[i-1], A[i])$, for the new values stored in A. Because at most constant work is done per item of A and B, this update also runs in worst-case linear time.

Rubric: for part (b)

- 3.5 points for description of update
- 1.5 points for running time analysis

Rubric: for part (c)

- 2 points for BST augmentation, e.g., subtree maximums
- 1 point for construction of BST in $O(n)$ time
- 6 points for a correct query algorithm
- Partial credit may be awarded
- 1 point for query running time analysis

Rubric: for part (d)

- 2 points for BST augmentation, e.g., stored subtree modifiers
- 7 points for a correct update algorithm
- Partial credit may be awarded
- 2 points for update running time analysis
- 2 points for adaptation of query algorithm

Unfortunately, there are so many sensors in the network that performing snow plow updates in linear time is much too slow. You want to redesign your data structure to:

1. process and store new sensor data in $O(n)$ time,
2. respond to a student query in $O(\log n)$ time, and
3. process a snow plow update in $O(\log n)$ time.

(c) [10 points] Describe how to augment a binary search tree to process new sensor data in worst-case $O(n)$ time, and respond to student queries in worst-case $O(\log n)$ time. Hint: use an augmentation that will make supporting updates in the next part easier. For this part, a rigorous proof of correctness is not required.

Solution: Store data from the n sensors in a **complete binary search tree** on n nodes, ordered by sensor distance from MIT. To build the tree, construct an complete tree on n nodes, and insert sorted measurements one-by-one via an in-order traversal in worst-case $O(n)$ time. Each node v stores information about one sensor, including sensor distance $v.d$ in feet along Mass. Ave., and snow height $v.h$. In addition, augment each node to store $v.m$, the **maximum height** of any sensor in v's subtree. $v.m$ can be computed in constant time per node from the stored maximums of v's children, $v.m = \max(v.h, v.left.m, v.right.m)$; so $v.m$ can be computed for all nodes in worst-case $O(n)$ by computing from the leaves to the root. Let $M(v, a)$ denote the maximum height of any snow measurement within a feet of MIT also contained in node v's subtree. For leaves, $M(v, a) = v.m = v.h$. For all other nodes, there are two cases:

1. If $a < v.d$, the student's house is in v's left subtree. Thus $M(v, a) = M(v.left, a)$, so we recursively compute once on $v.left$.
2. Otherwise, the student's house is at node v or in v's right subtree. Thus $M(v, a) = \max(v.left.m, v.h, M(v.right, a))$, so we recursively compute once on $v.right$.

Each recursive step takes constant time to compute, and makes a single recursive call on a lower node. So to answer a query from a student who lives a feet from MIT, we can compute and return $M(v, a)$ for root v in $O(\log n)$ time, since the height of a complete binary tree is $O(\log n)$.

(d) [10 points] Describe how to augment your data structure from part (c) to support snow-plow updates in worst-case $O(\log n)$ time, in addition to the other operations. For this part, a rigorous proof of correctness is not required.

Solution: A snow-plow update may affect the heights of $\Omega(n)$ sensors, so we cannot even address all of them in $O(\log n)$ time. Still, we need to affect all the heights within the range. Instead of $v.h$ corresponding directly to current the sensor height stored at node v, we will compute the current sensor height using additional information stored at each node. Augment each node v with a subtree modifier $v.x$. We maintain that for any node v, the current snow height of its sensor is $v.h + \sum_{u \in \pi_v} u.x$, where π_v is the path of nodes from the root to v. This property holds for the tree described in part (c) if we initialize all $v.x$ to zero.

To perform a snow-plow update adding height h to the range (a, b), we perform a procedure analogous to a range query. Start a BST search from the root to find the highest node p whose sensor location is in range (a, b), and add h to $p.x$. For p's left child v, perform the following recursive update (and symmetrically on the right).

1. If v's measurement is in range, then by BST property so are all nodes in v's right subtree. If v's parent is not in range, add h to $v.x$. Then recursively update $v.left$.
2. If v's measurement is not in range, then by BST property neither is v's left subtree. If v's parent is in range, subtract h from $v.x$. Then recursively update $v.right$.

At the end of this procedure, the sum of update values along the path from the root to any node v will have: increased by h if v is in range, or not changed if v is not in range. Proof by induction on the ancestors of v. The claim is true for the root as its update value will be modified only when the root is p. Now assume that the sum of update values from the root to the parent q of v is correct. There are four cases based on whether v and q are in range, each with a similar argument; only one is provided here: if v is in range and q is out of range, then $\sum_{u \in \pi_q} u.x = 0$, so adding h to $v.x$ in case (1) ensures that $\sum_{u \in \pi_v} u.x = h$.

After reaching the bottom of the tree in p's left and right subtrees, walk back up the tree updating $v.m = \max(v.h, v.left.x + v.left.m, v.right.x + v.right.m)$. The maximum value within v's subtree will then be given by $v.m + \sum_{u \in \pi_v} u.x$, rather than by $v.m$ directly. This procedure traverses forwards and backwards across three chains of constant time recursive calls, one from the root to node p, and once in each subtree of p. Since the tree has logarithmic height, this procedure takes $O(\log n)$ time.

Lastly, to support the student query algorithm, we modify the query from (c) to compute $M(v, a)$ using subtree modifiers.

1. If $a < v.d$, $M(v, a) = v.x + M(v.left, a)$.
2. Otherwise, $M(v, a) = v.x + \max(v.left.m + v.left.x, v.h, M(v.right, a))$.

F18 Instructor Note: This is an unusually tough problem for a 6.006 quiz.

SCRATCH PAPER 1. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to refer to the scratch paper next to the question itself.

SCRATCH PAPER 2. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to refer to the scratch paper next to the question itself.

SCRATCH PAPER 3. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to refer to the scratch paper next to the question itself.

SCRATCH PAPER 4. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to refer to the scratch paper next to the question itself.