Quiz 1 SOLUTIONS

You will have 2 hours to complete this exam. No notes or other resources are allowed. Unless otherwise specified, full credit will only be given to a correct answer which is described clearly and concisely.

Do not discuss this exam with anyone who has not yet taken it.

Problem	Points	Grade	Initials
Name	1		
1	14		
2	10		
3	20		
4	15		
5	10		
6	30		
Total	100		

Name: [1 point] _____

R01	R02	R03	R04	R05	R06	R07
WF10	WF11	WF12	WF1	WF2	WF3	WF3
Shaunak	Shaunak	Alan	Jeff	Rafael	Henrique	Dragos

Problem 1. True/False [14 points]

Answer True or False. You do not need to explain your answer.

(a) If $f(n) = \Omega(g(n))$ and $g(n) = \Theta(h(n))$, then $f(n) = \Omega(h(n))$.
True. This follows directly from the definitions of Ω and Θ.

(b) If $f(n) = o(g(n))$, then $\log(f(n)) = o(\log(g(n)))$.
False. For example, $f(n) = n$ and $g(n) = n^2$.

(c) If a student cheats by copying another's code, then defining functions in a different order will help him escape a plagiarism detection algorithm which uses the notion of document distance from lecture.
False. Document distance compares word frequencies, not order.

(d) If a student cheats by copying another's code, then renaming variables and functions will help him escape a plagiarism detection algorithm which uses the notion of document distance from lecture.
True. Renaming variables and functions will change the word frequency statistics.

(e) The following algorithm finds a peak in an array in $O(\log n)$ time: if the array only has one element, return it. Otherwise, pick the two adjacent elements at the middle and recurse on the side of the array containing the larger of the two.
True.

(f) If you use chaining to resolve collisions, you never have to resize the hash table, as long as you are willing to take a hit in performance.
True.

(g) Merge sort can be implemented so that it uses $O(n)$ space and $O(n \log n)$ time.
True.

Problem 2. Recurrences [10 points]

(a) Consider the following argument:

> Consider the recurrence $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n^2)$.
>
> At the top level of recursion, $\Theta(n^2)$ work is done. At the next level of recursion, there are 2 subproblems, each of which requires $\Theta(\left(\frac{n}{2}\right)^2)$ work, for a total of $\frac{1}{2}\Theta(n^2) = \Theta(n^2)$ work. At the next level of recursion, there are 4 subproblems, each of which requires $\Theta(\left(\frac{n}{4}\right)^2)$ work, for a total of $\frac{1}{4}\Theta(n^2) = \Theta(n^2)$ work. This continues for $\log_2(n)$ levels. Since each level requires $\Theta(n^2)$ work, the total runtime is $T(n) = \Theta(n^2 \log n)$.

i. [3 points] Briefly explain what is wrong with this argument.
Solution: The coefficients of the n^2 term decrease geometrically, and $n^2 + n^2/2 + n^2/4 + \cdots$ converges to $\Theta(n^2)$. In this analysis, they lost the exponential decrease in the constant factors by converting to Θ notation at each step.

ii. [2 points] What is the real solution to this recurrence?
Solution: $T(n) = \Theta(n^2)$. This can be found by the above argument, by the Master theorem, or by several other methods.

(b) [5 points] Find the asymptotic growth (Θ notation) of $T(n)$, given that it obeys the follow recurrence. (Hint: Think about the number of leaves in the recursion tree.)

$$T(n) = T(n/3) + T(2n/3) + \Theta(1)$$

Solution: $T(n) = \Theta(n)$.

There are several possible approaches to this problem.

Approach 1. The total running time is proportional to the total number of nodes in the recurrence tree. (Note: It is not true that the running time of every level of the tree is $\Theta(1)$, since the number of nodes in a level is not constant.) Furthermore, since every node in the tree has out-degree 0 or 2, the total number of nodes in the tree is at most twice the number of leaves. Therefore, it suffices to obtain a bound on the number

of leaves in the tree. We can prove that the recurrence tree has n nodes by induction: The base cases are clear, and the result follows from the fact that the number of leaves in the tree for $T(n)$ is the sum of the number of leaves in the tree for $T(n/3)$ and $T(2n/3)$.

Note: A less rigorous argument similar to the above approach imagines the behavior of $T(n)$ as a sequence of steps partitioning an array of size n. At every step, we divide one of the pieces into two pieces (one of $2/3$ the size and one of $1/3$ the size) and we recurse on all of these pieces until a piece has size 1 (in which case it is a leaf). Therefore, the total number of leaves is n. Furthermore, the total number of time we needed to break apart a segment is $\Theta(n)$, and we needed to do $\Theta(1)$ work for each of these partition operations. Since each leaf only requires $\Theta(1)$ work, we conclude that the total amount of work necessary is $\Theta(n)$.

Approach 2. We can directly prove that $T(n) = \Theta(n)$ by induction. For example, to prove that $T(n) = O(n)$, we know that there is a constant d such that

$$T(n) \leq T(n/3) + T(2n/3) + d$$

for all n. We now claim that there is some constant c such that

$$T(n) \leq cn - d$$

for all n. Indeed, we can choose a c such that this holds for the base cases $n = 1$ and $n = 2$. We now proceed by induction. Suppose that $T(n) \leq cn - d$ for all $n < N$. We now observe that

$$T(N) \leq T(N/3) + T(2N/3) + d \leq c(N/3) - d + c(2N/3) - d + d = cN - d$$

and the bound $T(n) = O(n)$ is proven by induction. A similar argument can be used to prove that $T(n) = \Omega(n)$.

Problem 3. Short Answer [20 points]

(a) [2 points] If a hash table using open addressing has a load factor of α, under the UHA assumption, how many tries does it take to insert a key, in expectation?

Solution: $1/(1-\alpha)$. When the load factor is α, each attempted insertion under UHA has success probability $1-\alpha$. Therefore, the expected number of attempts is $1/(1-\alpha)$.

(b) [6 points] Consider the following $O(n^3)$ algorithm for finding a peak on an $(n \times n \times n)$ cube of integers C:

- Create an $(n \times n)$ square S, where

$$S(i,j) = \max_{k=1,\ldots,n} C[i][j][k]$$

- Use the $O(n)$ peak-finding algorithm from class to find a peak in S.
- Return the peak found.

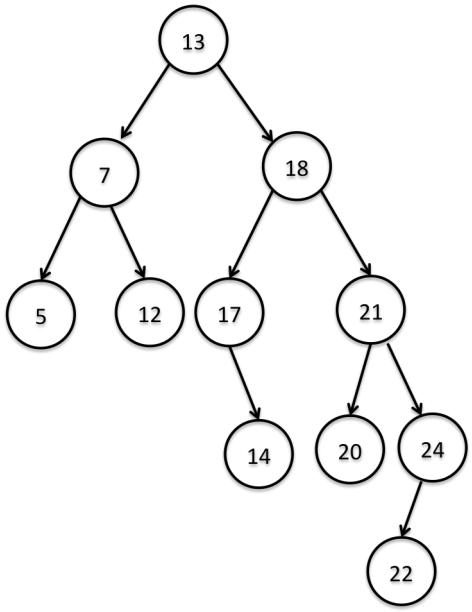
Explain how to modify this algorithm to obtain an $O(n^2)$ algorithm for finding a peak in an $(n \times n \times n)$ cube.

Solution: We can perform the $O(n)$ algorithm for an $n \times n$ square as before, but we do not compute every entry of the square S in advance. Instead, every time the $2d$ algorithm queries a cell in S, we do a linear scan over the appropriate n entries in C and use the max of these n entries for the value of the cell in S. By lazily computing the entries in S only when needed for the $2d$ algorithm, we slow down the $2d$ algorithm by a factor of n. Therefore, the overall running time of our algorithm is $O(n^2)$.

(c) The next two problems are about the AVL algorithm for balancing a tree.

i. [2 points] Change one value in the following tree to make it satisfy the binary search tree property. (You may have to use a non-integer value.)

Solution: Either change the 14 to something between 17 and 18 or change the 17 to something between 13 and 14.



ii. [3 points] Perform a single rotation on your answer from the previous part to make it satisfy the AVL invariant.

Solution: Perform a left-rotate on the root node (the node with value 13) on the tree given as the solution to the previous part. (The resulting tree depends on the answer given to the part above.)

(d) Consider using a hash table for integer values and resolving collisions by open addressing. At attempt i, you hash key k to the slot $f(k) + i \cdot g(k) \pmod{11}$, where $f(x) = x \pmod{11}$ and $g(x) = x^2 + 1 \pmod{11}$. (The first attempt is attempt 0.)

The table is initially empty (every slot has value NIL), and we perform the following operations in order:

- Insert 3
- Insert 14
- Insert 90
- Insert 2
- Delete 14

i. [5 points] What are the contents of the table after the above sequence operations? You should fill in the contents in **EVERY** slot in the hash table with either DELETED, NIL, or an INTEGER VALUE.

0	NIL
1	2
2	DELETED
3	3
4	NIL
5	NIL
6	NIL
7	90
8	NIL
9	NIL
10	NIL

Solution: See right column in the above table. Note that this problem is faster if we take the values $\mod 11$ at intermediate steps in the computation. For example, to determine the hash value of 90, we instead determine the hash value of $90 \mod 11 = 2$.

ii. [2 points] At this point, if we search for 13, what is the sequence of slots that we check before returning that it does not exist in the table?
Solution: 2, 7, 1, 6

Problem 4. BST Algorithms [15 points]

The following problems have you fill in the code for binary search tree algorithms. It is assumed that each node of a tree has a value, `val`, and pointers to its `left`, `right`, and `parent` nodes, which may be `None`. All node values are distinct.

You do not need to justify your answer.

(a) i. [5 points] Fill in the missing lines in the following code for `next`, which returns the next (successor) node in the binary search tree containing `node`, or `None` if there is none:

```
def next(node):
  if node.right is not None:
    cur_node = node.right
    while cur_node.left is not None:
      cur_node = cur_node.left
    return cur_node
  else:
    prev_node = node
    cur_node = node.parent
    while cur_node is not None and prev_node == ___________:
      prev_node = cur_node
      cur_node = prev_node.parent
    return cur_node
```

solution

```
cur_node.right
```

ii. [2 points] If the tree containing `node` is balanced and contains n nodes, what is the asymptotic worst case runtime of this algorithm?

Solution: $O(\log n)$. Depending on the case, the algorithm either goes from `node` down the tree or from `node` up the tree. In either case, the depth of the while loop is bounded by the height of the tree. Since the tree is balanced, the depth is $O(\log n)$.

(b) i. [5 points] Fill in the missing lines in the following code for `verify`, which verifies whether the tree rooted at `node` satisfies the BST property. The `next` function is that from above. You may assume that the input is a valid tree structure rooted at `node`, and that the input `node` has no parent.

```
def verify(node):
  prev_node = node
  while prev_node.left is not None:
    prev_node = prev_node.left
  cur_node = next(prev_node)
  while cur_node is not None:
    if _____________________________:
      return False
    prev_node = cur_node
    cur_node = next(prev_node)
  return True
```

solution:

```
prev_node.val > cur_node.val
```

ii. [3 points] If the tree rooted at `node` is balanced and contains n nodes, and if `node` does not have a parent, what is the asymptotic worst case runtime of this algorithm?

Solution: $\Theta(n)$. In an execution of the algorithm which doesn't abort and return false, we traverse each edge twice. Since the number of edges in a tree is one less than the number of nodes, we conclude that the worst case runtime of the algorithm is $\Theta(n)$.

Problem 5. Counting Close Pairs [10 points]

A pair (a, b) of integers is a *d-close pair* if $|a - b| \leq d$. Describe an algorithm to efficiently compute the number of d-close pairs in a list of integers. That is, given a list A and an integer d, your algorithm should find the number of ordered pairs (i, j) such that $i < j$ and $|A[i] - A[j]| \leq d$.

Describe and analyze your algorithm and give a brief argument why it's correct. Be sure to explicitly state your algorithm's asymptotic running time.

Solution: First, sort A, and let B be the sorted array. (this takes $O(n \log n)$ time)

Now, we will proceed by using a two-finger pass on B:

1. Let ctr be the counter of the number of d-close pairs. Initially $ctr = 0$.
2. Let i and j be two pointers to the entries of B. Initially $i = 1$ and $j = 2$.
3. If $j < n$ go to the next step. Else, go to step 6.
4. If $B[j] - B[i] \leq d$ then $j+ = 1$. Go back to step 3.
5. Else if $B[j] - B[i] > d$ then $ctr \leftarrow ctr + j - i - 1$ and $i \leftarrow i + 1$. (if $i == j$ after we increase i then increment j by 1 as well). Go back to step 3.
6. (We only get here when $j == n$.)
   (a) If $i == n$, return ctr.
   (b) Else if $B[n] - B[i] > d$: then $i \leftarrow i + 1$ and repeat this step.
   (c) Else (if $B[n] - B[i] \leq d$): then $ctr \leftarrow ctr + \binom{n-i}{2}$ and return ctr.

Since this two-finger pass takes linear time (since all its steps take constant time and we will execute each of them $O(n)$ times), our total running time is $O(n \log n)$.

Proof of correctness: The algorithm counts each d-close pair (i, j) once- when we pass the "left finger" through i. If they are d-close, then the first index of B that will violate the condition for index i (and hence update the counter) will be an index greater than j, and therefore we will always count that pair. We count these pairs only once because we account for this pair only after passing the left finger through i. It is easy to show furthermore that we do not count any pairs which are not close.

Rubric: 1 pt for the trivial solution ($O(n^2)$)

If your algorithm was worse than $O(n^2)$ then in general you got a 0. However, if you had the right ideas (sorting to compare the elements, but then when comparing you did something wrong) then you got a maximum of 3-4 points, depending on your mistake.

If your algorithm runs in $o(n^2)$ then:

- description and correctness of your algorithm: 7 points (if all correct)
- analysis of runtime: 3 points

We took off 2 points for the lack of clarity, even if your solution was correct.

Also, we took off 1 point if you counted twice the number of pairs (since this will not be the correct output), and we took off 4 points if you kept track of the wrong count (people counted the complement of the d-close pairs, for instance) but you had the right ideas.

Problem 6. Team Selection [30 points]

As captain of the school football team, you are trying to select a team from n candidate players. Your goal is to choose the best team based on your ratings of the players.

(a) [10 points] Suppose that each of these players i has two ratings: their speed, a_i, and their strength, b_i. All a_i and b_i are distinct. A player i is *majorized* by a player j if player j is faster and stronger than player i - that is, if $a_i < a_j$ and $b_i < b_j$.

Give an efficient algorithm for determining the list of players who are not majorized by any other player, given the list of the n pairs $(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)$.

Solution: We claim that we can find the non-majorized players in $O(n \log n)$ time. Sort the players by decreasing order of a value and then iterate over the pairs, keeping track of the maximum b value seen so far - call it b_{max}. At each step, if the next (a_i, b_i) pair has a b value greater than b_{max}, append it to the answer list and update b_{max}. The final answer list consists of the non-majorized players.

Proof of correctness: a player is majorized if and only if some player has a larger a value and a larger b. When we consider a player, we are comparing its b value precisely to the maximum b value among players with larger a's, so we add a player to the list if and only if he is not majorized.

Proof of runtime: sorting the players by a value takes $O(n \log n)$ time. The iteration takes $O(n)$ time, so the overall runtime is $O(n \log n)$.

Rubric: A complete response should describe a correct algorithm that runs in time $O(n \log n)$ with an argument for correctness and a running time analysis. 3 points were deducted for lack of analysis or major errors, and 2 points for minor errors. 1 point was deducted for sufficient lack of clarity. The obvious but correct solution that checks all pairs of athletes to eliminate majorized players in n^2 time was awarded the symbolic credit of 1 point. Slower solutions than the obvious one were not given credit. Algorithms with hints of the correct ideas were awarded symbolic credit depending on quality/correctness. Making wrong statements, though, decreased the score to 0

(b) [20 points] After holding tryouts, you are now able to rate players on their skill c_i as well as their strength and speed. Now, a player i is majorized by a player j if player j is faster, stronger, and more skillful - that is, if $a_i < a_j$, $b_i < b_j$, and $c_i < c_j$. All a_i, b_i and c_i are distinct.

Again, give an algorithm for determining the list of players who are not majorized by any other player, given the list of the n triples $(a_1, b_1, c_1), (a_2, b_2, c_2), \ldots, (a_n, b_n, c_n)$.

Solution: Again, we claim that there is an $O(n \log n)$ solution. Sort the players by decreasing order of a_i, and then iterate over the triples, inserting them into a binary search tree keyed by b_i. This tree is augmented with information about the c_i: in particular, each node keeps track of the maximum c_i of any triple stored in its subtree. It is not hard to verify that one can maintain this augmentation. When inserting a player, whenever we branch to the left, we check that his c value is greater than that of the node where we branched, and the maximum c_i on that node's right subtree. If this fails at any point of insertion, we abort the insertion (and delete this player from the list). At the end, we perform an in-order traversal to create a list of the players in the tree. We claim that the resulting list corresponds to the non-majorized players.

Proof of correctness: Essentially, the augmentation lets us determine the maximum c_i of any node with $b_i > x$, for any x. Thus we abort insertion for player i if and only if there is some player j, with $a_j > a_i$ (because of the order of insertion), $b_j > b_i$ (because of the subtrees/nodes we are checking), and $c_j > c_i$ (because we are checking the max in those subtrees/nodes)

Proof of run-time: The sort takes $O(n \log n)$ time. We then iterate over n items, doing $O(\log n)$ inserts (even with rebalances and augmentation maintenance). Lastly, we do a $O(n)$ traversal. Thus the overall run-time is $O(n \log n)$.

Alternate Solution: Again, we claim that there is an $O(n \log n)$ solution. As in the first part, we will sort players by decreasing order of a value and then iterate over the list, determining if each player is majorized or not. At all times, we will maintain two data structures: a list of non-majorized players found so far, and the subset of that list that is not majorized when we only take b and c values into account. We call this subset the bc frontier. Because this frontier is not majorized by b and c value, if we sort it by increasing b value, it will also be sorted by decreasing c value. We keep the frontier in a BST sorted by increasing b.

Now, given this information for the first $i - 1$ triples, we need to determine if the next triple (a_i, b_i, c_i) is majorized or not. This triple has a smaller a value than any of the first $i - 1$ triples, so it is not majorized if and only if it is not majorized by the triples on the bc frontier. We can check this condition by searching in the BST for the node on the frontier with the smallest b value greater than b_i. If this node has a larger c value than c_i, then (a_i, b_i, c_i) is dominated. Otherwise, we add this triple to the answer list and to the bc frontier. This node may dominate some nodes on the frontier by b and c value - we delete those nodes from the tree.

Proof of correctness: for each triple, we check that it is not majorized by any other triple by checking it against the bc frontier of all triples with a greater a value.

Proof of run-time: sorting the triples by a value takes $O(n \log n)$ time. The iteration takes $O(n \log n)$ time total, because for each triple, we spend $O(\log n)$ time searching the frontier to check the condition, up to $O(\log n)$ time if we insert the triple into the frontier, and up to $O(\log n)$ time if we delete it.

Rubric: As in part (a), but with everything doubled.

Test ends here.