Test 2 Midterm: Compsci 201

Owen Astrachan
Jeff Forbes

November 8, 2017

Name: _____

NetID/Login: _____

Honor code acknowledgment (signature) _____

	value	grade
Problem 1	18 pts.	
Problem 2	12 pts.	
Problem 3	16 pts.	
Problem 4	16 pts.	
Problem 5	6 pts.	
TOTAL:	68 pts.	

This test has 15 pages, be sure your test has them all. Do NOT spend too much time on one question — remember that this class lasts 75 minutes.

That means you should spend no more than *1 minute per point*. Put your NetID *clearly* on each page of this test (worth 1 extra point).

In writing code you do not need to worry about specifying the proper `import statements`. Don't worry about getting function or method names exactly right. Assume that all libraries and packages we've discussed are imported in any code you write. You can write any helper methods you would like in solving the problems. You should show your work on any analysis questions.

The is one blank page at the end of the test. Make a note on the appropriate problem if you use the extra sheet.

Common Recurrences and their solutions.

label	recurrence	solution
A	`T(n) = T(n/2) + O(1)`	$O(\log n)$
B	`T(n) = T(n/2) + O(n)`	$O(n)$
C	`T(n) = 2T(n/2) + O(1)`	$O(n)$
D	`T(n) = 2T(n/2) + O(n)`	$O(n \log n)$
E	`T(n) = T(n-1) + O(1)`	$O(n)$
F	`T(n) = T(n-1) + O(n)`	$O(n^2)$
G	`T(n) = 2T(n-1) + O(1)`	$O(2^n)$

TreeNode and ListNode classes as used on this test.

```
public class TreeNode {
    int info;
    TreeNode left;
    TreeNode right;

    TreeNode(int x){
        info = x;
    }
    TreeNode(int x, TreeNode lNode,
                    TreeNode rNode){
        info = x;
        left = lNode;
        right = rNode;
    }
}
```

```
public class ListNode {
    int info;
    ListNode next;
    ListNode (int val) {
        info = val;
    }
    ListNode(int val,
             ListNode link){
        info = val;
        next = link;
    }
}
```

PROBLEM 1 : (*Say You'll Remember Me (18 points)*)

The `ArrayList` class and the `LinkedList` class both implement the `java.util.List` interface. The code below removes elements from a `List` until the list is empty – this code has a runtime of $O(n)$ for both `ArrayList` and `LinkedList` lists that contain n elements. Recall that `ArrayList` objects have an internal array to store elements whereas `LinkedList` objects have an internal doubly-linked list to store elements.

```
public double removeLast(List<String> list) {
    double start = System.nanoTime();
    while (list.size() != 0) {
        list.remove(list.size()-1);
    }
    double end = System.nanoTime();
    return (end - start) / 1e9;
}
```

Part 1.1 (2 points)

Which element is removed from the list by the call `list.remove(list.size()-1)`?

Part 1.2 (6 points)

Suppose a similar method `removeFirst` is implemented — identical except the statement in the body of the loop is `list.remove(0)` — the method will have a runtime of $O(n)$ for one of `ArrayList` and `LinkedList` objects, but not for both. **Briefly justify both answers below.**

What is the big-Oh runtime complexity of `removeFirst(list)` when list is an `ArrayList` and why?

What is the big-Oh runtime complexity of `removeFirst(list)` when list is n `LinkedList` and why?

Part 1.3 (4 points)

The method `total` below correctly calculates the sum of all values in a `List<Integer>` so that `total(list)` will work when `list` is an `ArrayList` and when it is a `LinkedList`. Explain why this method has a runtime of $O(n)$ to calculate the sum of all values in an `ArrayList` and a runtime of $O(n^2)$ to calculate the sum of all values in a `LinkedList`.

```
public int total(List<Integer> list)  {
    int sum = 0;
    for(int k=0; k < list.size(); k++){
        sum += list.get(k);
    }
    return sum;
}
```

Why $O(n)$ for an n-element `ArrayList`

Why $O(n^2)$ for an n-element `LinkedList`

Part 1.4 (4 points)

The method `itotal` below correctly calculates the sum of all values in a `List<Integer>` so that `itotal(list)` will work when `list` is an `ArrayList` and when it is a `LinkedList`. The big-Oh runtime complexity is the same for both types of list when `itotal` is called with a list of n elements. What is the big-Oh complexity. **Briefly justify your answer.**.

```
public int itotal(List<Integer> list)  {
    int sum = 0;
    for(int val : list) {
        sum += val;
    }
    return sum;
}
```

What is the big-Oh runtime of `itotal` for both `ArrayList` and `LinkedList` (it's the same expression)

Why is this the complexity for both types of `List`?

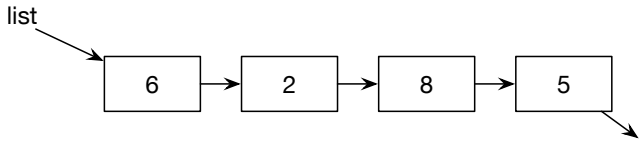
Part 1.5 (2 points)

Explain briefly why the `LinkedList` class implements the `java.util.Queue` interface, but the `ArrayList` class does not? In your answer reference the runtime of the `java.util.List.removeFirst` method.

PROBLEM 2 : (*Listing to Starboard (12 points)*)

Part 2.1 (6 points)

Write the method `lessThan` that returns the number of nodes in its list parameter that are *less than* parameter `ceiling`. For example, the call `lessThan(list,9)` returns 4, the call `lessThan(list,1)` returns 0, and the call `lessThan(list,6)` returns 2 since only the nodes with `.info` fields 2 and 5 are less than 6.




```
public int lessThan(ListNode list, int ceiling) {

}
```

Part 2.2 (3 points)

The code below is a version of a correct/all-green solution to the list APT *DoubleList* — the code creates n new nodes when parameter `list` is an n-node list so that the list [1,2,3,4] is changed to [1,1,2,2,3,3,4,4]. More generally one new node is added after every node, the new node stores the same value as the node before it.

```
 public ListNode bigify(ListNode list) {
     ListNode first = list;
     while (list != null) {
         list.next = new ListNode(list.info,list.next);
         list = list.next.next;
     }
     return first;
}
```

What is the big-Oh complexity of this code for an n-node list? *Briefly justify your answer.*

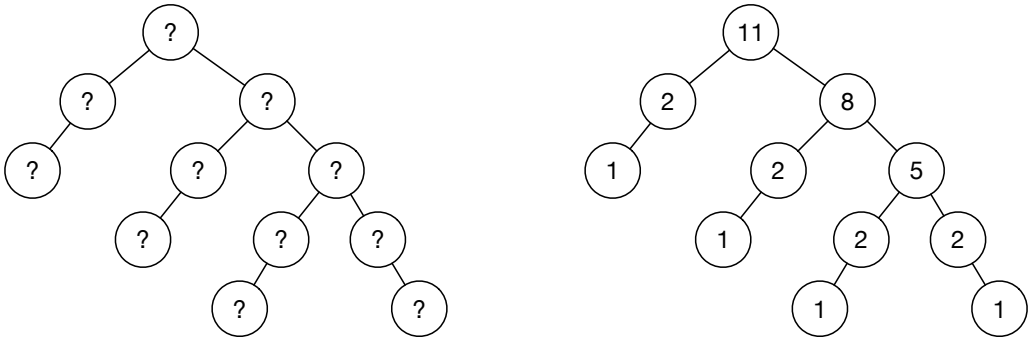
Part 2.3 (3 points)

Suppose that the only change to the code in `bigify` is that the statement `list = list.next.next;` is replaced by the `list = list.next;` statement. With this change the method above generates a *time limit exceeded* message with the APT tester for every list that has one or more elements.

Explain why the method no longer terminates.

PROBLEM 3 : (*Orings, Othings (16 points)*)

In this problem you'll reason about two methods that each return a tree with the same shape as a parameter `tree`. In the returned tree each node's `.info` field is equal to the number of nodes of the tree with that node as root. For example, given the tree below on the left, the returned tree is shown on the right. In the tree shown below the returned tree's root has the value 11, where as the tree's right subtree has the value 8 as its root.



Part 3.1 (2 points)

What value is stored in every leaf of the returned tree?

Part 3.2 (2 points)

If the root is at level zero, the trees above have deepest leaves at level 4. By definition in a *complete* tree there are are 2^k nodes at level k for every level $k = 0, 1, \ldots$; and there a total of $2^{n+1} - 1$ nodes in a complete tree whose deepest leaves are at level n. What value is stored in the root of the tree returned if the tree parameter is a complete tree with deepest nodes at level 9? You must supply an exact, numerical answer.

Part 3.3 (4 points)

The method `countLabel` shown below correctly returns a tree with the same shape.

Label each line below with an expression involving $O(..)$ or $T(..)$ when `countLabel` is called with an N node tree, so $T(N)$ is the time for `countLabel` to execute with an N node tree.

On this page label lines for the average case. Be sure to label each line *of method* `countLabel` with either an $O(...)$ expression or a $T(...)$ expression for **the average case when trees are roughly balanced**. $T(N)$ **is the time for** `countLabel` **to run.**

You don't need to label code in the method `count`. You do need to label the call of `count` in `countLabel`.

Label each line with $O(..)$ or $T(..)$

```
public TreeNode countLabel(TreeNode tree) {
    if (tree == null) return null;

    TreeNode left = countLabel(tree.left);
    TreeNode right = countLabel(tree.right);
    int size = count(tree);
    return new TreeNode(size,left,right);
}

public int count(TreeNode tree) {
    if (tree == null) return 0;
    return 1 + count(tree.left) + count(tree.right);
}
```

Be sure to write the recurrence relation and its solution.

Part 3.4 (4 points)

Label each line below with an expression involving $O(..)$ or $T(..)$ when `countLabel` is called with an N node tree, so $T(N)$ is the time for `countLabel` to execute with an N node tree.

On this page label lines for the worst case. Be sure to label each line *of method* ***`countLabel`*** with either an $O(...)$ expression or a $T(...)$ expression for the worst case when trees are completely unbalanced, e.g., all nodes in the right subtree. $T(N)$ **is the time for `countLabel` to run.** You don't need to label code in the method `count`. You do need to label the call of `count` in `countLabel`.

Label each line with $O(..)$ or $T(..)$

```
public TreeNode countLabel(TreeNode tree) {
    if (tree == null) return null;

    TreeNode left = countLabel(tree.left);
    TreeNode right = countLabel(tree.right);
    int size = count(tree);
    return new TreeNode(size,left,right);
}

public int count(TreeNode tree) {
    if (tree == null) return 0;
    return 1 + count(tree.left) + count(tree.right);
}
```

Be sure to write the recurrence relation and its solution.

Part 3.5 (4 points)

The method `countLabelAux` correctly returns a tree with the same shape. You are to determine the *average case* case complexity of this method. You must develop a recurrence relation for `countLabelAux` — the average case is when trees are roughly balanced.

Label each line below with an expression involving $O(..)$ or $T(..)$ when `countLabel` is called with an N node tree, so $T(N)$ is the time for `countLabelAux` to execute with an N node tree.

This is for the average case, when trees are roughly balanced.

Label each line with $O(..)$ or $T(..)$

```
public TreeNode countLabelAux(TreeNode tree) {
   if (tree == null) return null;
   if (tree.left == null && tree.right == null){
        return new TreeNode(1,null,null);
   }
   TreeNode left = countLabelAux(tree.left);
   TreeNode right = countLabelAux(tree.right);
   int lcount = 0;
   int rcount = 0;
   if (left != null)  lcount = left.info;
   if (right != null) rcount = right.info;
   return new TreeNode(1 + lcount + rcount,left,right);
}
```

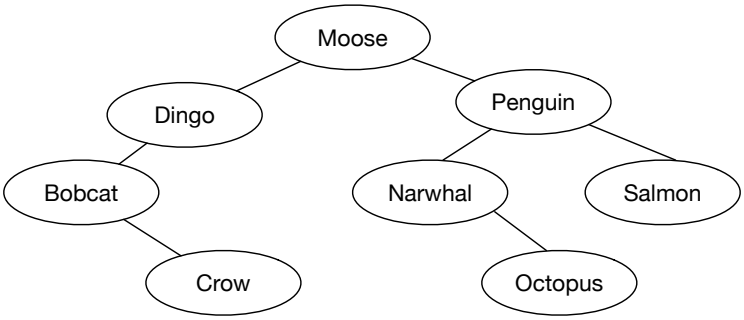
Be sure to write the recurrence relation and its solution.

PROBLEM 4 : (*Benedict Arnold and Trees (16 points)*)

Consider the binary search tree shown below. You'll be asked several questions about trees using this tree as an example. Strings are inserted into the tree in *natural* or *lexicographical* order.

In answering the questions you can use these words, or any other words you choose

> anteater, badger, bear, cougar, dog, elephant, ferret, fox, giraffe, hippo, jaguar, kangaroo, koala, leopard, llama, meerkat, mole, mouse, mule, newt, orangutan, ostrich, otter, panda, pelican, tiger, walrus, yak, zebra



See below for code for the *inorder*, *preorder*, and *postorder* traversals of a tree.

inorder	preorder	psotorder

`void inOrder(TreeNode t) {`
`  if (t != null) {`
`  inOrder(t.left);`
`  System.out.println(t.info);`
`  inOrder(t.right);`
`  }`
`}`
	`void preOrder(TreeNode t) {`
	`  if (t != null) {`
	`  System.out.println(t.info);`
	`  preOrder(t.left);`
	`  preOrder(t.right);`
	`  }`
	`}`
		`void postOrder(TreeNode t) {`
		`  if (t != null) {`
		`  postOrder(t.left);`
		`  postOrder(t.right);`
		`  System.out.println(t.info);`
		`  }`
		`}`

The *inorder* traversal of the tree is *bobcat, crow, dingo, moose, narwhal, octopus, penguin, salmon.*

Part 4.1 (3 points)

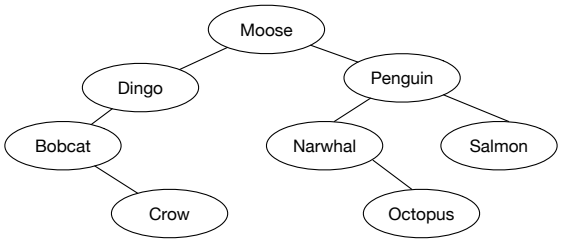
What is the post-order traversal of the tree shown?

Part 4.2 (3 points)

If the recursive calls in method `inOrder` are swapped, so that the right subtree of `t` is visited, then `t.info` printed, then the left subtree visited, then what is the order of nodes visited for the tree shown with this modified search?

Part 4.3 (3 points)

In the drawing below, label where these strings are inserted, in the order shown: *bear*, *cougar*, *elephant*, *badger*.



Part 4.4 (3 points)

The height of the tree shown is four since the longest root-to-leaf path has four nodes. List three strings/animals such that if they are inserted into the search tree in the order you list them the height of the resulting tree will be seven.

Part 4.5 (2 points)

If a *sorted list* of n strings is inserted one-at-a-time into an initially empty search tree that does no balancing after each insertion the complexity of the n insertions will be greater than $O(n)$. *What is the complexity of the n insertions from a sorted list and why?*

Part 4.6 (2 points)

If a *sorted list* of n strings is inserted one-at-a-time into a `java.util.TreeSet` (which internally uses a balanced Red-Black tree) the complexity of the n insertions will be greater than $O(n)$. *What is the complexity of the n insertions from a sorted list and why?*

PROBLEM 5 : (*Good to the Last Drop (6 points)*)

The percolation assignment called for estimating the *percolation threshold* on an $N \times N$ grid of cells/sites. The classes `PercolationDFS` and `PercolationDFSFast` were similar in estimating the threshold. Assume that in the worst case both have a runtime of $O(N^4)$ for estimating the threshold on an $N \times N$ grid.

Part 5.1 (2 points)

Code in the class `PercolationDFSFast` was able to call methods `inBounds` and `dfs` from the `PercolationDFS` class. Briefly, what concepts/Java mechanisms allowed these methods from another class to be called from code in `PercolationDFSFast`? It's possible to answer this with one word.

Part 5.2 (2 points)

Although the complexity for both these classes is $O(N^4)$, the runtimes are drastically different. For example, estimating the threshold for a 200×200 grid for `PercolationDFS` takes about 60 seconds whereas it takes less than 1 second for `PercolationDFSFast`.

Explain why the times can be so different for the two classes although both classes are $O(N^4)$. Your answer **does not** need to reference any methods or code in either class. This question is asking about two methods that have the same runtime of $O(N^4)$ but have very different runtimes empirically.

Part 5.3 (2 points)

In the `PercolationUF` class, an `IUnionFind` object is used to determine when percolation occurs. Code you wrote in the `PercolationUF` class calls `IUnionFind.connected` and `IUnionFind.union` methods, but does not call the `IUnionFind.find` method. The `connected(p,q)` method returns true if and only if p and q are in the same set. The call `union(p,q)` merges the sets containing p and q.

When/where is the `IUnionFind.find` method called? Be brief.