Test 2 Practice : Compsci 201

Owen Astrachan

November 9, 2019

Name: ________________________________

NetID/Login: ____________

Community standard acknowledgment (signature) ________________________________

<table>
<thead>
<tr>
<th></th>
<th>value</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>NetID</td>
<td>1 pt.</td>
<td></td>
</tr>
<tr>
<td>Problem 1</td>
<td>7 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 2</td>
<td>16 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 3</td>
<td>12 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 4</td>
<td>12 pts.</td>
<td></td>
</tr>
</tbody>
</table>

This test has 22 pages, be sure your test has them all. Write your NetID clearly on each page of this test (worth 1 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.

You may consult your six (6) note sheets and no other resources. You may not use any computers, calculators, cell phones, or other human beings. Any note sheets must be turned in with your test.
Common Recurrences and their solutions.

<table>
<thead>
<tr>
<th>label</th>
<th>recurrence</th>
<th>solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>T(n) = T(n/2) + O(1)</td>
<td>O(\log n)</td>
</tr>
<tr>
<td>B</td>
<td>T(n) = T(n/2) + O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>C</td>
<td>T(n) = 2T(n/2) + O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>D</td>
<td>T(n) = 2T(n/2) + O(n)</td>
<td>O(n \log n)</td>
</tr>
<tr>
<td>E</td>
<td>T(n) = T(n-1) + O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>F</td>
<td>T(n) = T(n-1) + O(n)</td>
<td>O(n^2)</td>
</tr>
<tr>
<td>G</td>
<td>T(n) = 2T(n-1) + O(1)</td>
<td>O(2^n)</td>
</tr>
</tbody>
</table>

TreeNode and ListNode classes as used on this test. In some problems the type of the info field may change from int to String and vice versa.
PROBLEM 1:  (*Summary Motion (7 points)*)

The method `sum` below correctly returns the sum of all the nodes in its linked-list parameter.

```java
public int sum(ListNode list) {
    if (list == null) return 0;
    return list.info + sum(list.next);
}
```

**Part A (3 points)**

What is the recurrence relation for this method when `list` has `n` nodes? Use `T(n)` as the time for `sum` to execute on an `n`-node list. What is the solution to this recurrence? You don’t have to explain/justify, just write the recurrence and its solution.

\[ T(n) = \]

**Part B (4 points)**

Write an iterative solution, one in which no recursion is used. For every list the values of `sum(list)` and `summer(list)` should be the same.

```java
public int summer(ListNode list) {
    // iterative solution
}
```
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(...) \text{ or } T(...) \text{ when } \text{countLabel} \text{ is called with an } N \text{ node tree, so } T(N) \text{ is the time for countLabel to execute with an } N \text{ node tree.}

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

```
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 \( O(\ldots) \) or \( T(\ldots) \) when \( \text{countLabel} \) is called with an \( N \) node tree, so \( T(N) \) is the time for \( \text{countLabel} \) to execute with an \( N \) node tree.

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

Label each line with \( O(\ldots) \) or \( T(\ldots) \)

```java
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 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 $O(\cdot\cdot\cdot)$ or $T(\cdot\cdot\cdot)$ 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.

```java
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);
}
```

Label each line with $O(\cdot\cdot\cdot)$ or $T(\cdot\cdot\cdot)$

Be sure to write the recurrence relation and its solution.
PROBLEM 3: (Integer Overflow (12 points))

The ListNode class at the beginning of this test allows linked-list nodes to hold any types. For example, the code below creates a linked list of three strings as shown at the top of the diagram to the right.

```java
ListNode<String> list =
    new ListNode<>("hello",
    new ListNode<>("big",
    new ListNode<>("crocodile",null)));
```

Write the method convert, which is started below. The method creates a new linked-list storing integer values such that the value in the $i$th node returned is the length of the string of the $i$th node that’s a parameter to convert. The list in the diagram shown on the bottom above illustrates this.

**Part A: (6 points)**

Complete the method convert below. **You must write the method iteratively**, that is with a loop.

```java
public ListNode<Integer> convert(ListNode<String> list){
    ListNode<Integer> first = new ListNode<>(list.info.length(),null);
    list = list.next;
    ListNode<Integer> last = first;

    // write code below
}
```
Part B: (6 points)

Complete the method convertRec below. You must use write the method recursively, that is without a loop and without any collections or arrays. The method convertRec returns the same list that convert returns when the parameters to the methods are the same.

```java
public ListNode<Integer> convertRec(ListNode<String> list) {
    if (list == null) return null;

    // write code here
}
```
**PROBLEM 4 : (Frayed Knot (12 points))**

The `ListNode` class has an `info` field of type `String`. The call `str2list("asp")` for the method `str2list` below returns the linked list shown.

```
0    public ListNode str2list(String s) {
1        if (s.length() == 0) return null;
2        return new ListNode(s.substring(0,1),
3                        str2list(s.substring(1)));
        }
```

**Part A (2 points)**

The recursive call (line 3) and its execution assigns a value to three next fields as shown in the diagram of the linked list. What code/where is the explicit assignment to the `.next` field of a node?

**Part B (2 points)**

The base case returns the value `null`. In a sentence or two explain how the recursive call (line 3) is closer to the base case each time a recursive call is made.
Part C (8 points)

Write an iterative version (no recursion) of this method by completing the code below. You must use a loop and you must maintain the invariant that last points the last node of the linked list being created in the loop.

```java
public ListNode str2list2(String s) {
    if (s.length() == 0) return null;

    ListNode first = new ListNode(s.substring(0,1),null); // first letter in first node
    ListNode last = first;

    return first;
}
```
PROBLEM 5:  (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.

<table>
<thead>
<tr>
<th>inorder</th>
<th>preorder</th>
<th>postorder</th>
</tr>
</thead>
<tbody>
<tr>
<td>void inOrder(TreeNode t) { if (t != null) { inOrder(t.left); System.out.println(t.info); inOrder(t.right); } }</td>
<td>void preOrder(TreeNode t) { if (t != null) { System.out.println(t.info); preOrder(t.left); preOrder(t.right); } }</td>
<td>void postOrder(TreeNode t) { if (t != null) { postOrder(t.left); postOrder(t.right); System.out.println(t.info); } }</td>
</tr>
</tbody>
</table>

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 \texttt{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: \textit{bear, cougar, elephant, badger}.

```
Moose
  Dingo
  Bobcat
  Crow
Penguin
  Narwhal
  Salmon
  Octopus
```

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 6:  \textit{(Needle in a Haystack? (23 points))}

The method \texttt{SearchTree} below correctly returns \texttt{true} if its parameter \texttt{root} is a binary search tree, and \texttt{false} if it is not a binary search tree.

```java
29*    public int max(TreeNode root) {
30        if (root == null) return Integer.MIN_VALUE;
31        return Math.max(root.info,
32                           Math.max(max(root.left), max(root.right)));
33    }
34
35*    public int min(TreeNode root) {
36        if (root == null) return Integer.MAX_VALUE;
37        return Math.min(root.info,
38                           Math.min(min(root.left), min(root.right)));
39    }
40
41*    public boolean searchTree(TreeNode root) {
42        if (root == null) return true;
43        if (! searchTree(root.left)) return false;
44        if (! searchTree(root.right)) return false;
45        if (root.info <= max(root.left)) return false;
46        if (root.info >= min(root.right)) return false;
47        return true;
48    }
```

Consider each tree rooted at \texttt{F} below.

- The tree on the left is \textit{not a search tree} because although the left-subtree of the root is a search tree, and the right-subtree of the root is a search tree, one of the nodes, \texttt{M}, in the left-subtree is greater than the root.

- The tree in the middle \textit{is a search tree} because for each node, the subtree rooted at that node is a search tree and each node is greater than all values in its right subtree and less than or equal to all values in its left subtree.

- The tree on the right is \textit{not a search tree} because the subtree rooted at \texttt{T} is not a search tree since \texttt{V} is not less than or equal to \texttt{T}.

\textbf{Part A (7 points)}

Methods \texttt{min} and \texttt{max} shown above each have a recurrence of \(M(n) = 2M(n/2) + O(1)\).

Let \(T(n)\) be the time for \texttt{SearchTree} to run when the tree parameter has \(n\) nodes. \textit{Label each line of the method \texttt{SearchTree} with an expression using \(T\), or with a big-Oh expression. Assume trees are roughly balanced. Using these labels complete the recurrence:}

\(T(n) = \) 

What is the solution to this recurrence?
Part B (8 points)
In this problem TreeNode.info has type int, so stores an integer value.
A full tree has $2^{n-1}$ nodes at level $n$ where the root is at level one, its children at level two, and in general the children of a node at level $k$ are at level $k + 1$. The nodes at level $n$ are numbered starting with $2^{n-1}$ and incrementing by one, from left-to-right, until the last node on level $n$ which is $2^n - 1$.

In the trees below, the tree on the left is a full-tree with two levels and the tree on the right is a full tree with four levels. We call these 2-level and 4-level full trees, respectively.

B.1
What is the total number of nodes in a 4-level tree, i.e., the tree on the right above?

B.2
What is the total number of nodes in an $n$-level full tree?

B.3
What is the number of non-leaf nodes in a 3-level full tree?

B.3
What is the number of non-leaf nodes in a $n$-level full tree?
Part C (8 points)

The method `createFull` below creates and returns a full tree with the specified number of levels so that the call `createFull(2)` returns the tree on the left and the call `createFull(4)` returns the tree on the right.

The method uses a private helper method that you should complete using at most 10 lines of code.

```java
public TreeNode createFull(int levels) {
    return fullHelper(1, levels);
}

/**
* Return a full-tree with the specified number of levels
* in which the root contains nodeValue
* @param nodeValue is the value of the root of the tree returned
* @param levels is the number of levels in the full tree returned
* @return a full tree
* /
private TreeNode fullHelper(int nodeValue, int levels) {
    if (levels == 1) {
        return new TreeNode(nodeValue);
    }
    return null;
}
```
PROBLEM 7: (RYOGVIB (12 points))

The tree shown below is a binary search tree. The in-order traversal of the tree will be a list of all the values in the tree in alphabetical order.

Part A (6 points)

The code for a pre-order and post-order traversal are shown below.

```java
public void preOrder(TreeNode root) {
    if (root != null) {
        System.out.println(root.info);
        preOrder(root.left);
        preOrder(root.right);
    }
}

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

What is the pre-order traversal of the tree (values printed)?

What is the post-order traversal of the tree (values printed)?

If the recursive calls in method `preOrder` are swapped, so that the right subtree call is made first, what values are printed?

Part B (6 points)

Insert the words "red", "white", and "blue" in that order, in the tree above so that it remains a search tree. Label the values by drawing on the tree.
Write method `divide` which alters a linked list and returns the altered list so that all nodes with an odd index appear in the same order as in the original list followed by all nodes with an even index in the same order. Odd and even nodes are determined by the index of the node in the original list with the first node as node number 1, the next node as node number 2 and so on.

For example, the original list is shown at the top of the diagram and the returned list at the bottom.

Your code should not create any new nodes, but instead alter next fields. Your code should run in $O(n)$ time where $n$ is the number of nodes in the original list.

**You can write the code in any way.**

You may find the following ideas useful in writing a solution.

- Check if `list` is null or has one node, in which case it can be returned without changing any pointers. This ensures that the code you write deals with lists of two or more nodes and thus contains both an odd node and an even node.

- Create a variable `boolean odd = true` before the loop you write. In the loop you write use the statement `odd = !odd` to change the value from `true` to `false` and vice versa.

One idea that will work and that you’re strongly encouraged to use is to maintain pointers to the first and last nodes of two lists: an odd-list and an even-list as shown in the diagram. As the original list is traversed, nodes are alternately added to the end of the odd-list or even-list.

If you write the code to maintain four such pointers, the last statements you might write are:

```java
oddLast.next = evenFirst;
evenLast.next = null;
return oddFirst;
```

You’ll need to initialize these four variables before the loop and access some of them in the body of the loop that you write.

Complete the code on the next page.
public ListNode divide(ListNode list) {

}
PROBLEM 9:  (I think That I (16 points))

In class we went over the two methods below. Method `height` returns the height of a binary tree, the longest root-to-leaf path. Method `leafSum` returns the sum of the values in all leaves of a tree (assuming integer values are stored in each node). Line numbers shown are not part of the methods.

```java
int height(Tree root) {
    1    if (root == null) return 0;
    2    return 1 + Math.max(height(root.left),
    3        height(root.right));
}

public int leafSum(TreeNode t) {
    1    if (t == null) return 0;
    2    if (t.left == null && t.right == null) return t.info;
    3    return leafSum(t.left) + leafSum(t.right);
}
```

**Part A (4 points)**
Assume trees are roughly balanced. The methods above have the same recurrence relation. **What is this recurrence relation?** Briefly explain why the same recurrence holds for each method by labeling each line in the methods above with an expression involving $T(\cdot)$ or $O(\cdot)$.

**Part B (4 points)**
If the line labeled 3 is removed from `leafSum` the method returns the same value for every non-empty tree, i.e., `leafSum(tree)` returns the same value for every tree. What value is returned? Briefly justify your answer.
Part C (8 points)
In answering this question assume all values in a tree are positive.
Write a method `maxPath` that returns the maximal value of all root-to-leaf paths in a binary tree. In the tree shown here the root-to-leaf paths sum to 16, 15, 16, 18, and 15 since the paths are 2-6-8, 2-6-4-3, 2-6-4-3-1, 2-2-1-4-9, and 2-2-1-4-6. The method should return 18.

In writing you method you may **not use any instance variables**.

In writing your method you must consider the base case of an empty tree in which the maximal value must be zero since there are no paths.

Using recursion, the maximal value for the root of a tree can be determined by the maximal values of its subtrees.

```java
public int maxPath(TreeNode root) {

```