Test 2 Practice : Compsci 201

Owen Astrachan

April 1, 2019

Name: _____

NetID/Login: _____

Community standard acknowledgment (signature) _____

	value	grade
NetID	1 pt.	
Problem 1	8 pts.	
Problem 2	18 pts.	
Problem 3	16 pts.	
Problem 4	12 pts.	
Problem 5	16 pts.	
Problem 6	12 pts.	
Problem 7	32 pts.	
TOTAL:	114 pts.	

This test has 20 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.

PROBLEM 1 : (*It Depends (8 points)*)

Part A (3 points)

What value is returned by the call `calculate(2043)`? What is the *runtime* complexity (big-Oh, in terms of N) of the call `calculate(N)`? Briefly justify your answers.

```
public int calculate(int n){
    int prod = 1;
    while (prod < n){
        prod *= 2;
    }
    return prod;
}
```

Part B (5 points)

Consider method `docalc` below, the call `docalc(6)` evaluates to 21.

```
public int docalc(int n){
    if (n == 0) return 0;
    return n + docalc(n-1);
}
```

Using big-Oh what is the running time of the call `docalc(n)`? Justify your answer.

Using big-Oh what is the *value returned* by the call `docalc(n)` (note: complexity of value returned, not running time: use big-Oh)

Using big-Oh what is the running time of the call `docalc(docalc(n))` *based on your answers to the previous two questions.* Justify.

Using big-Oh what is the *value returned* by the call `docalc(docalc(n))` (again, based on previous answers, justify).

PROBLEM 2 : (*Oh-Oh (22 points)*)

Part A (7 points)

Consider the method `bleem` below. The value of `bleem(10)` is 88, the value of `bleem(20)` is 360, and the value of `bleem(100)` is 9133.

A.1

Using big-Oh, what is the *runtime* complexity of the call `bleem(N)`? Justify your answer with words and labeling the code as appropriate.

A.2

Using big-Oh, what is the *value returned* by the call `bleem(N)`? Your answer should be consistent with the values given above; your answer should use O-notation, no justification needed.

A.3

Using big-Oh, what is the runtime complexity of the call `bleem(bleem(N))`? Your answer should be consistent with your answers to A.1 and A.2 above. No explanation needed.

```
public int bleem(int n) {
    int sum = 0;

    for(int k=0; k < n; k++) {

        for(int j=0; j < k; j += 1) {
            sum += 1;
        }
        for(int j=0; j < k; j += 2) {
           sum += 1;
        }
        for(int j=0; j < k; j += 3) {
            sum += 1;
        }
    }
    return sum;
}
```

Part B (6 points)

Consider the method `calc` below. The value of `calc(32)` is 160, the value of `calc(64)` is 384, and the value of `calc(1024)` is 10240.

B.1

Using big-Oh, what is the *runtime* complexity of the call `calc(N)`? Briefly justify your answer.

B.2

Using big-Oh, what is the *value returned* by the call `calc(N)`? Your answer should be consistent with the values given above; your answer should use O-notation, no justification needed.

```
public int calc(int n) {
    int sum = 0;
    int limit = n;
    while (limit > 1) {
        sum += n;
        limit = limit/2;
    }
    return sum;
}
```

Part C (9 points)

Consider the function `func` below. The value of `func(10)` is 20, the value of `func(40)` is 80, and the value of `func(90)` is 180.

C.1

What is the *exact* value of `func(1024)`? You must supply an integer answer.

C.2

Using big-Oh, what is the *runtime* complexity of the call `func(n)`? Briefly explain your answer. For full credit you should use a recurrence relation. Label the function if that's helpful for your explanation.

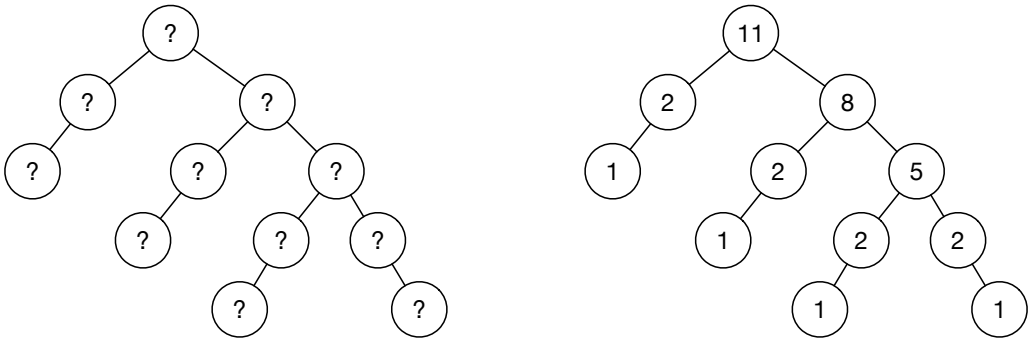
C.3

Using big-Oh, what is the *value* returned by the call `func(func(2*n))`? Your answer should be consistent with your answers to the previous questions. Briefly justify your answer.

```
public int func(int n) {
    if (n <= 0) return 0;
    return 2 + func(n-1);
}
```

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 $T(..)$

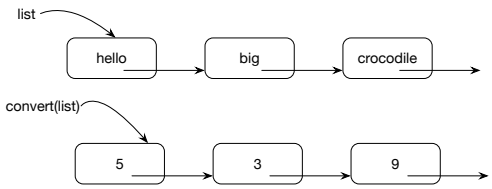
```
public TreeNode countLabelAux(TreeNode tree) {
   if (tree == null) return null;
   if (tree.left == null && tree.right == null){
        return new TreeNode(1,null,null);
   }
   TreeNode left = countLabelAux(tree.left);
   TreeNode right = countLabelAux(tree.right);
   int lcount = 0;
   int rcount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

```
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.

```
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




}
```

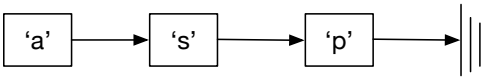
Part B: (6 points)

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.

```
public ListNode<Integer> convertRec(ListNode<String> list) {

    if (list == null) return null;

    // write code here









}
```

PROBLEM 5 : (*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.

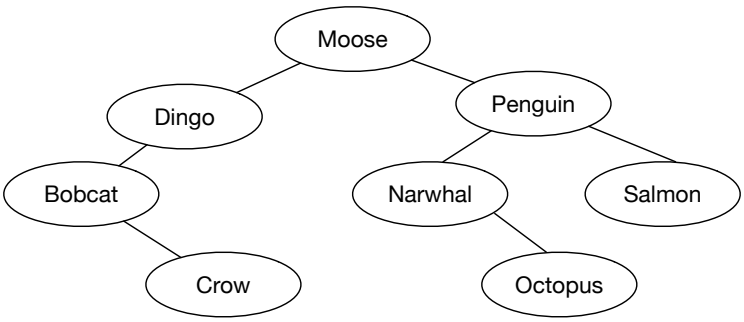
```
public ListNode str2list2(String s) {
  if (s.length() == 0) return null;

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









  return first;
}
```

PROBLEM 6 : (*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)

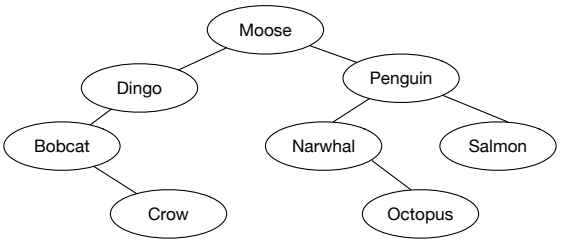
What is the post-order traversal of the tree shown?

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 7 : (*Markov, Polov (12 points)*)

In the *Markov Part 2* assignment the same method `getRandomText` below was used in both class `BaseMarkov` and in class `EfficientMarkov`. Questions about the method follow the code. Line numbers are shown, but are not part of the code.

```
0  @Override
1  public String getRandomText(int length) {
2
3  StringBuilder sb = new StringBuilder();
4        int index = myRandom.nextInt(myText.length() - myOrder + 1);
5
6        String current = myText.substring(index, index + myOrder);
7        sb.append(current);
8
9        for(int k=0; k < length-myOrder; k += 1){
10           ArrayList<String> follows = getFollows(current);
11           if (follows.size() == 0){
12               break;
13           }
14           index = myRandom.nextInt(follows.size());

15           String nextItem = follows.get(index);
16           if (nextItem.equals(PSEUDO_EOS)) {
17               break;
18           }
19           sb.append(nextItem);
20           current = current.substring(1)+ nextItem;
21       }
22       return sb.toString();
23   }
```

Part A (6 points)

One line in the code above executes more quickly for `EfficientMarkov` than for `BaseMarkov`. **What line (you can indicate the number) executes more quickly**? Briefly explain why that one line is $O(1)$ for `EfficientMarkov` and $O(T)$ for `BaseMarkov` when the training text has `T` characters.

Part B (2 points)

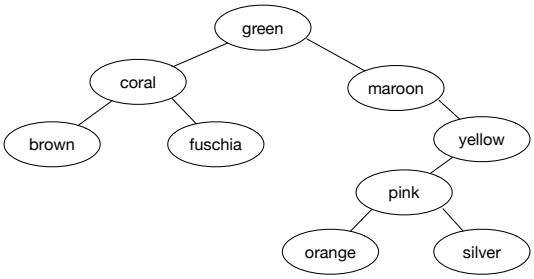
Briefly explain when `PSEUDO_EOS` can be encountered and why that sometimes results in fewer than `1000` random characters being generated by the call `getRandomText(1000)` if the training text has only 100 characters.

Part C (4 points)

The call `getRandomText(N)` on the previous page runs in $O(N)$ time to generate `N` random characters when `EfficientMarkov` is used. If `StringBuilder` is replaced by `String`, and `sb.append` is replaced by `sb.concat`, the output will be the same, but the runtime will be $O(N^2)$. Briefly explain both the $O(N)$ and the $O(N^2)$.

PROBLEM 8 : (*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.

```
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.

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.

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

    public int leafSum(TreeNode t) {
1       if (t == null) return 0;
2
3       if (t.left == null && t.right == null) return t.info;
4
5       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 $T(..)$ or $O(..)$.

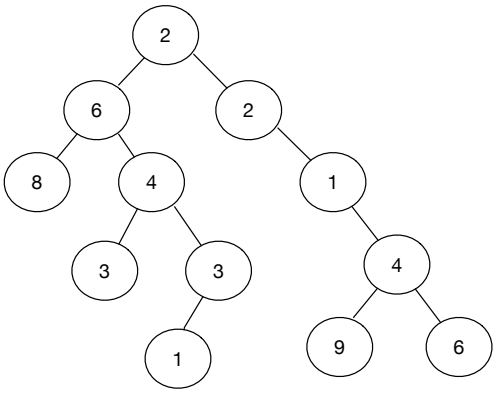
Part B (4 points)

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.

```
public int maxPath(TreeNode root) {

}
```