# Test 2 : Compsci 201

Owen Astrachan

November 2, 2018

Name: Aaron Aardvark

NetID/Login: aca000

Community standard acknowledgment (signature) _____

| | value | grade |
|---|---|---|
| NetID | 1 pt. | |
| Problem 1 | 22 pts. | |
| Problem 2 | 12 pts. | |
| Problem 3 | 12 pts. | |
| Problem 4 | 12 pts. | |
| Problem 5 | 16 pts. | |
| TOTAL: | 75 pts. | |

This test has 12 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.

| 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 {
    String info;
    TreeNode left;
    TreeNode right;

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

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

**PROBLEM 1 :** *(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.

$O(n^2)$ $1+2+\cdots n \Rightarrow n^2$ for each inner loop

**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.

$O(n^2)$

**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.

bleem($n^2$) $\Rightarrow$ $n^4$

```
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;
}
```

O(k) → $n^2$ w/ outer

O(k) →

O(k) →

**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.

O(log N)

n, n/2, n/4 ... 1

$\log_2 n$

**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.

n+n+..+n

O(n log n)

```
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.

2,048

**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.

$T(n) = T(n-1) + O(1) \Rightarrow O(n)$

**C.3**

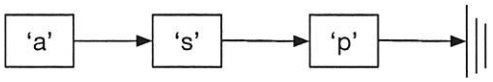
Using big-Oh, what is the *value* returned by the call `func(func(2*n))`? Your answer should be consistent with your answers to the previous questions. Briefly justify your answer.

func(2n) ⇒ 4n
func(4n) ⇒ 8n ⇒ O(n)

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

**PROBLEM 2 :** *(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?

ListNode constructor
next = link

**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.

.substring(1) length is 1 less ⇒ each time closer to zero

**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;
```

for(int k=1, k<s.length(); k++){

last.next = new ListNode(s.substring(k,k+1), null);

last=last.next;

}

```
  return first;
}
```

**PROBLEM 3 :** (*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.

line 10
Efficient, one call to myMap.get hashmap O(1)

Base, scan entire text looking for current O(T)

**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.

last k/myOrder characters have no follow so PSEUDO-EOS follows them.

Can be chosen on line 15 if PSEUDO in value associated with current

Then break before loop finishes

**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 $O(N^2)$.

StringBuilder.append(B)

complexity is O(B) the one character?

O(1) ⇒ N times ⇒ O(N)

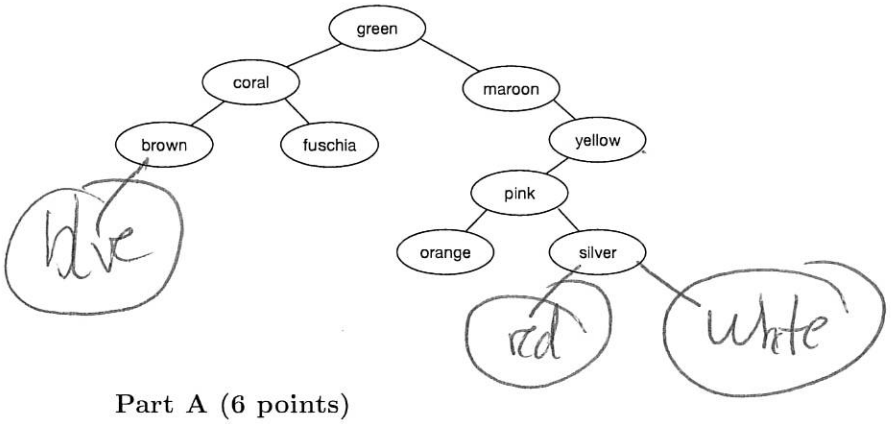
String.concat(B)

complexity is O(A+B)

a ⇒ 0, 1, 2, 3, ..., N-1 ⇒ O(N²)

**PROBLEM 4 :** ***(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)?

green, coral, brown, fuschia, maroon, yellow, pink, orange, silver

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

brown, fuschia, coral, orange, silver, pink, yellow, maroon, green

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

green, maroon, yellow, pink, silver, orange, coral, fuschia, brown

**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.

see above

**PROBLEM 5 :** ***(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);
}
```

O(1)
T(n/2)
T(n/2)

O(1)
O(1)
T(n/2), T(n/2)

**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 $O(..)$.

$T(n) = 2T(n/2) + O(1)$

**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.

zero returned

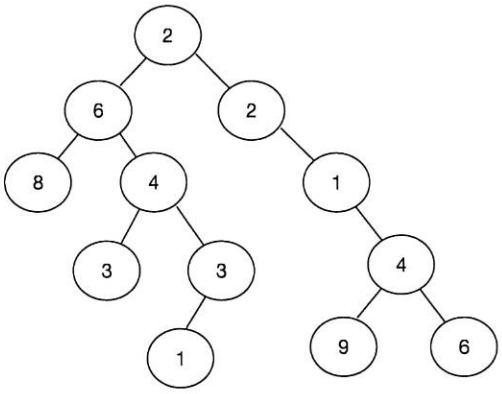
base case $\Rightarrow 0$

all others? Add 0+0 $\Rightarrow 0$

**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 maxSum(TreeNode root) {
    if (root == null) rtn 0;
    int rmax = maxSum(root.right);
    int lmax = maxSum(root.left);
    rtn root.info + Math.max(lmax, rmax);
}
```