

CS 171 Midterm 2

Tyler Angert

TOTAL POINTS

89 / 100

QUESTION 1

Complexity True False 18 pts

1.1 1.a 4.5 / 4.5

- 0 Correct
- 4.5 Incorrect choice and explanation
- 0 Click here to replace this description.

1.2 1.b 4.5 / 4.5

- 0 Correct
- 2 Incorrect choice, but explanation has some merit
- 4.5 Incorrect choice and explanation

1.3 1.c 4.5 / 4.5

- 0 Correct
- 2 Correct choice, but explanation is incorrect
- 4.5 Incorrect

☞ $O(N \lg N)$

1.4 1.f 4.5 / 4.5

- 0 Correct
- 2 Correct choice, but incorrect explanation
- 4.5 Incorrect

QUESTION 2

Complexity Code 18 pts

2.1 2.a 6 / 6

- 0 Correct
- 6 Incorrect
- 3 No explanation

2.2 2.b 6 / 6

- 0 Correct
- 6 Incorrect
- 3 No explanation

2.3 2.c 0 / 6

- 0 Correct
- 6 Incorrect

- 3 Correct without any explanation

QUESTION 3

EMD Debug 10 pts

3.1 3.a 5 / 5

- + 5 correct
- + 0 Incorrect

3.2 3.b 5 / 5

- + 5 Correct
- + 0 Incorrect

QUESTION 4

4 Debug 6 / 6

- 0 Correct
- 3 Half right
- 6 Incorrect

QUESTION 5

5 Mergesort 6 / 6

- 0 Correct
- 2 Missed a step
- 4 Missed several steps
- 6 Incorrect/Dpes not follow mergesort
- 0.5 Minor flaw

QUESTION 6

6 Replicate Elements 12 / 14

- 0 Correct
- 2 syntax errors
- 14 Incorrect
- 5 Minor problem
- 9 It is not completely correct
- 14 I could not read
- 2 Replace numbers than K times

QUESTION 7

Binary Search Tree 28 pts

7.1 max 14 / 14

- 0 Correct
- 3 Minor problems
- 7 Will not work, but has some merit
- 10 Major problems
- 14 No answer, or wrong.

7.2 countOdds 11 / 14

- 0 Correct
 - 3 Minor Problems
 - 7 Does not work, but has some merit
 - 10 Major problems
 - 14 No answer / Wrong answer
- 🗨 Forgot about the value of the root itself

CS 171 - Spring 2016

Midterm exam #2

Full Name: Tyler Angert OPUS ID: tangert Section: ☐ 000 (Vigfusson)
☒ 001 (Fossati)
☐ 002 (Wildani)

My answers to this exam are my own work. I understand that this exam is governed by the Emory Honor Code.

Signature: 

Grading:

| 1a | 1b | 1c | 1d | 1e | 1f | 2a | 2b | 2c | 3a | 3b | 4 | 5 | 6 | 7a | 7b | Tot |
|----|----|----|----|----|----|----|----|----|----|----|---|---|----|----|----|-----|
| 3 | 3 | 3 | 3 | 3 | 3 | 6 | 6 | 6 | 5 | 5 | 6 | 6 | 14 | 14 | 14 | 100 |
| | | | | | | | | | | | | | | | | |

1. For each of the following statements, circle whether it is true or false, and briefly explain why. N is the number of elements in a data structure.

1.a) Pushing an element to the top of a stack takes $O(N)$ time.

True False Explanation:

$O(1)$, There's always space on top!

1.b) Swapping the top with the bottom of a stack takes $O(N)$ time.

True False Explanation:

Have to iterate through the whole stack to get the values switched.

1.c) Inserting N elements in a binary search tree takes $O(N)$ time.

True False Explanation:

BST insertion is $O(\log(N))$.

~~1.d) Finding the smallest element in a min heap takes $O(\log N)$ time.~~

True False Explanation:

~~1.e) Finding the largest element in a min heap takes $O(\log N)$ time.~~

True False Explanation:

1.f) Inserting 5 elements in a hash table takes $O(\log N)$ time.

True False Explanation:

$O(1)$. Hash table insertion is constant.

2. What is the run-time complexity in big-O notation of the following blocks of code? Justify your answer. Here are some potentially useful formulas:

$$1 + 2 + 3 + \dots + n = n * (n+1) / 2$$

$$1 + x + x^2 + x^3 + \dots + x^n = (x^{n+1} - 1) / (x - 1)$$

2.a)

```
Stack<Integer> s = new Stack<Integer>();
for (int i = N; i > 0; i--) {
    s.push(i);  $\Rightarrow N$  times
    while (!s.isEmpty()) {
        s.pop();  $\Rightarrow N$  times
    }
}
```

$O(N) + O(N) = O(N)$

$O(N)$. For each value N , an element is pushed onto the stack. While the stack still has elements, that very element is popped as well. Both of these operations occur N times.

2.b)

```
public static int f(int k) {
    if (k <= 0)
        return 0;
    else
        return 1 + f(k / 2) + 1;
}
```

$f(2)$
 \downarrow
 $f(\frac{2}{2})$
 \downarrow
 $f(\frac{1}{2})$
 \downarrow

2^k

$f(2 * 4 * 8 * N)$; // this is the call in the main method

$O(\log N)$. When this is called, each recursive step requires half the operations of the previous.

2.c)

```
BinarySearchTree bst = new BinarySearchTree();
for (int i = 0; i < N; i++) {
    bst.add(i);
}
```

$O(N \log N)$. Insertion into a BST is $\log(N)$.

Because this operation occurs N times, the total time-complexity is $O(N \log N)$.

3. Consider the following solution for add() in Project 5 (Emory Movie Database). It works in many cases, but there are some cases for which this code does not work properly.

```
// Add the key and a value to your RangeMap. (For EMD, this would be the
// name of the movie (key) and its description (value), respectively.) If
// there is a duplicate key, the old entry should be overwritten with the
// new value.
```

```
public void add(K key, V value) {
    if (root == null) {
        root = new Node();
        root.kv = new KVPair<K, V>(key, value);
    } else {
        add(root, key, value);
    }
}


// recursive helper for add
private void add(Node n, K key, V value) {
    if (key.compareTo(n.kv.key) < 0) {
        if (n.left == null) {
            n.left = new Node();
            n.left.kv = new KVPair<K, V>(key, value);
        } else {
            add(n.left, key, value);
        }
    } else if (key.compareTo(n.kv.key) > 0) {
        if (n.right == null) {
            n.right = new Node();
            n.right.kv = new KVPair<K, V>(key, value);
        } else {
            add(n.right, key, value);
        }
    }
}
```

3.a) What are the cases for which this code will not work correctly?

when the keys are compared and they are equal.
(aka, the entries are the same)

3.b) How would you fix it?

add this if statement:


else if { key.compareTo(n.kv.key) == 0 } {
 n = new Node();
 n.kv = new KVPair<K, V>(key, value);
}

This overwrites the existing⁴ KV pair and replaces it with a new one.

4. Consider the following Java code.

```
\ public static int sumOdds(Stack<Integer> s) {  
    2 if (s.isEmpty()) {  
        3 return 0;  
    4 } else if (s.peek() % 2 == 1) {  
        5 return s.pop() + sumOdds(s);  
    6 } else {  
        7 return sumOdds(s);  
    8 }  
9 }
```

This method is supposed to remove all elements from a stack, and return the sum of its odd elements. It will compile and run, but will give you an error. Explain the error and fix it.

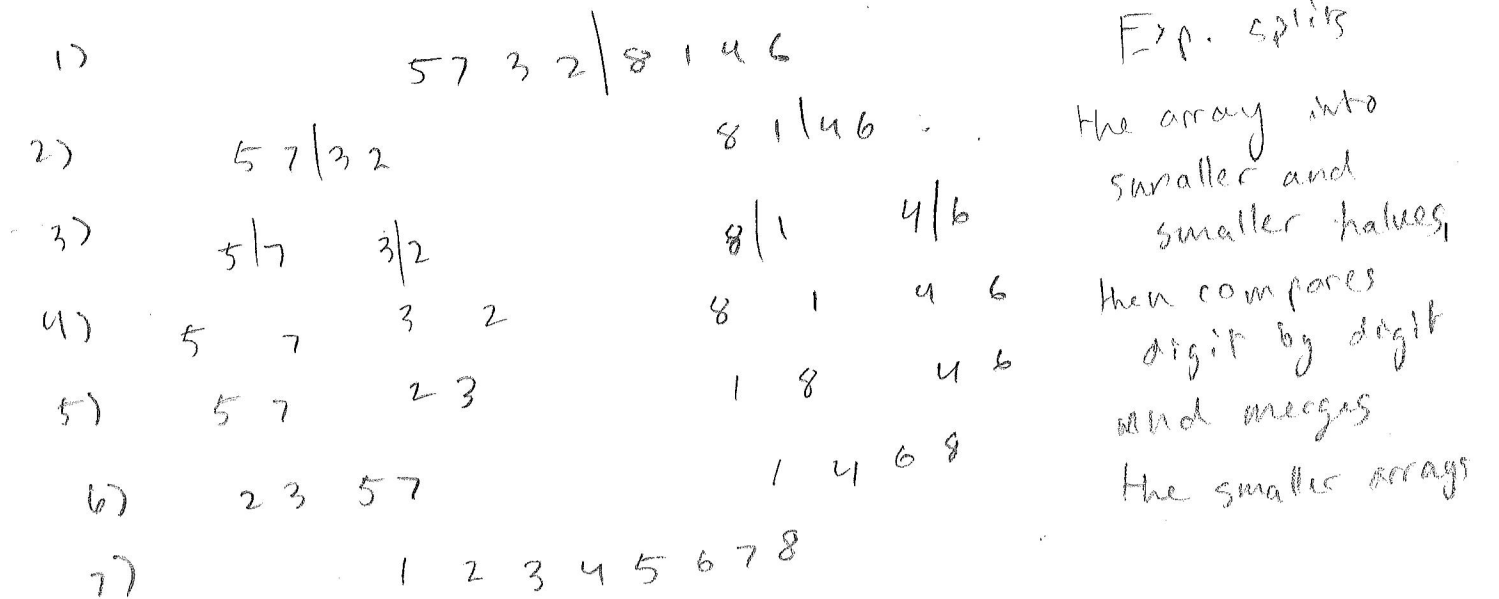
This does not remove all elements. It only removes the odd ones! Currently, what this does is return the sum of odd elements but leaves us with a stack of just even elements.

To fix:

On line 7, change to:

s.pop() + return sumOdds(s);

5. Sort the list of numbers 5, 7, 3, 2, 8, 1, 4, 6 using **mergesort**. Show all the steps.



6. Write a method `replicateElements` that takes a Queue of integers `q` and an integer `k`. The method replicates all the elements in the queue `k` times. For example, if `q` contains [1, 2, 3], after the call `replicateElements(q, 3)` it will contain [1, 1, 1, 2, 2, 2, 3, 3, 3].

Hint: you can use the following methods in a Queue:

`add(int e)` adds an element to the end of the queue

`int remove()` removes an element from the front of the queue and returns it

`int size()` returns the number of elements in the queue

$$3 \cdot 3 = 9$$

$$2 \cdot 2 = 4$$

$$1 \cdot 1 = 1$$

```
public static void replicateElements(Queue<Integer> q, int k) {
```

```
    int originalSize = q.size();
```

```
    for (int i = 0; i < originalSize; i++) { // for each element in queue
```

```
        int n = remove();
```

```
        for (int j = 0; j < k; j++) { // copies the number k times
```

```
            add(n);
```

```
        }
```

What this is doing:

1) removes the 1st element and assigns that to a number `n`.

2) goes through a loop of size `k` and copies `n` `k` times to the front.

3) goes to the next number in the queue and repeats.

7. Complete the methods `max()` and `countOdds()` in the binary search tree class below. Hint: you can write additional private recursive helper methods if you need so.

```
class Node {
    int data;
    Node left;
    Node right;
}
```

```
public class BinarySearchTree {
```

```
    Node root;
```

```
    // (7.a) Returns the largest value in the tree.
    // If the tree is empty, returns 0.
    public int max() {
```

```
        if (root == null) return 0;
```

```
        else {
```

```
            while (root.right != null) {
```

```
                root = root.right;
```

```
            }
```

```
        } return root.data;
```

```
    // (7.b) Returns the number of odd values in the tree
    public int countOdds() {
```

```
        if (root == null) return 0;
```

```
        int odds = 0;
```

```
        Node rootCopy = new Node();
```

```
        rootCopy = root;
```

```
        while (root.right != null) {
```

```
            root = root.right;
```

```
            if (root.data % 2 == 1) odds++;
```

```
        }
```

```
        root = rootCopy;
```

```
        while (root.left != null) {
```

```
            root = root.left;
```

```
            if (root.data % 2 == 1) odds++;
```

```
        } return odds;
```

7

Recursive???

What this does:

1. Establishes an odd count
2. creates a copy of the root for future use.

3. Iterates through all left elements and increments odds if found

4. resets root

5. Repeats step 3 but with right side

6. Returns odd count

Problem: doesn't hit every child/leaf.

