Section 05: Midterm Review

1. Asymptotic Analysis

(a) **Applying definitions**

For each of the following, choose a c and n_0 which show $f(n) \in \mathcal{O}(g(n))$. Explain why your values of c and n_0 work.

(i) $f(n) = 5000n^2 + 6n\sqrt{n}$ and $g(n) = n^3$

(ii) $f(n) = n(4 + \log(n))$ and $g(n) = n^2$

(iii) $f(n) = 2^n$ and $g(n) = 3^n$

(b) **Runtime Analysis**

For each of the following, give a Θ-bound for runtime of the algorithm/operation:

(i) Best-case **get** in a binary search tree of size n.

(ii) Best-case **put** in a hash table with size n and a current $\lambda = 1$ if the collision resolution is:

- Separate chaining
- Linear Probing

(iii) Pop a value off a stack containing n elements implemented as an array.

(iv) Finding the minimum value in a BST of size n.

(v) Finding the minimum value in a AVL tree of size n.

(vi) Print out values in AVL tree of size n.

(vii) Iterating through and printing every element in an array list using a for loop and the `get(i)` method.

(viii) Pop on a stack containing n elements implemented as a singly-linked list.

(ix) Inserting a value into an AVL tree of size n, where the value you are inserting is smaller than any other values currently in the tree.

2. Eyeballing Big-Θ bounds

For each of the following code blocks, what is the worst-case runtime? Give a big-Θ bound. You do not need to justify your answer.

(a)
```
void f1(int n) {
    int i = 1;
    int j;
    while(i < n*n*n*n) {
        j = n;
        while (j > 1) {
            j -= 1;
        }
        i += n;
    }
}
```

(b)
```
int f2(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            System.out.println("j = " + j);
        }
        for (int k = 0; k < i; k++) {
            System.out.println("k = " + k);
            for (int m = 0; m < 100000; m++) {
                System.out.println("m = " + m);
            }
        }
    }
}
```

(c)
```
int f3(n) {
    count = 0;
    if (n < 1000) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                for (int k = 0; k < i; k++) {
                    count++;
                }
            }
        }
    } else {
        for (int i = 0; i < n; i++) {
            count++;
        }
    }
    return count;
}
```

(d)
```
void f4(int n) {
    IList<Integer> arr = new DoubleLinkedList<>();
    for (int i = 0; i < n; i++) {
        if (list.size() > 20) {
            list.remove(0);
        }
        list.add(i);
    }
    for (int i = 0; i < list.size; i++) {
        System.out.println(list.get(i));
    }
}
```

3. Recurrences and Summations

Find an expression for the total work of the following expressions using the Tree Method, then simplify that summation to a closed form. If possible, check that the big-$\mathcal{O}$ is correct with Master Theorem.

Hint: Just as a reminder, here are the steps you should go through for **any** Tree Method Problem:

i. Draw the recurrence tree.

ii. What is the size of the **input** to each node at level i? What is the amount of **work** done by each node at the i-th *recursive* level? As in class, we call the root level $i = 0$. This means that at $i = 0$, your expression for the input should equal n.

iii. What is the total number of nodes at level i? As in class, we call the root level $i = 0$. This means that at $i = 0$, your expression for the total number of nodes should equal 1.

iv. What is the total work done across the i-th *recursive* level?

v. What value of i does the last level of the tree occur at?

vi. What is the total work done across the base case level of the tree (i.e. the last level)?

vii. Combine your answers from previous parts to get an expression for the total work.

(a) $T(n) = \begin{cases} T(n-1) + n^2 & \text{if } n > 19 \\ 57 & \text{otherwise} \end{cases}$

(b) $T(n) = \begin{cases} T(n/2) + n^2 & \text{if } n \geq 4 \\ 5 & \text{otherwise} \end{cases}$

(c) $T(n) = \begin{cases} 2T(n/3) + 5n & \text{if } n > 1 \\ 9 & \text{otherwise} \end{cases}$

4. Modeling

Consider the following method. Let n be the integer value of the `n` parameter, and let m be the length of `DoubleLinkedList`. You may assume that $n > 7$.

```
public int mystery(int n, DoubleLinkedList<Integer> list) {
    if (n < 7) {
        System.out.println("???");
        int out = 0;
        for (int i = 0; i < n; i++) {
            out += i;
        }
        return out;
    } else {
        System.out.println("???");
        System.out.println("???");
        out = 0;
        for (int i : list) {
            out += 1;
            for (int j = 0; j < list.size(); j++) {
                System.out.println(list.get(j));
            }
        }
        return out + 2 * mystery(n - 4, list) + 3 * mystery(n / 2, list);
    }
}
```

Note: your answer to all three questions should be a recurrence, possibly involving a summation. You do not need to find a closed form.

(a) Construct a mathematical function modeling the *approximate* worst-case runtime of this method in terms of n and m.

(b) Construct a mathematical function modeling the *exact* integer output of this method in terms of n and m.

(c) Construct a mathematical function modeling the *exact* number of lines printed out in terms of n and m.

5. AVL/BST

(a) Insert $\{94, 33, 50, 76, 96, 67, 56, 65, 83, 34\}$ into an initially empty AVL tree.

(b) Insert $\{6, 5, 4, 3, 2, 1, 10, 9, 8, 6, 7\}$ into an initially empty AVL tree.

(c) Suppose you insert 7 elements into a BST. What are the possible heights after the insertions? (Think about different orders of inserting elements).

(d) If you insert 7 elements into an AVL tree, what are the possible heights of the tree?

(e) More generally, what is the minimum number of nodes in an AVL tree of height 4? Draw an instance of such an AVL tree.

(f) Describe an implementation of `delete()` for a BST.

(g) Adapt `delete()` for a BST to work for an AVL tree. Don't worry if there are some edge cases that don't quite work yet, the next part will help with those.

(h) Make a more robust (not buggy) `delete()` for an AVL tree, and try to be as (**hint**) lazy as possible.

6. Hash tables

(a) What is the difference between primary clustering and secondary clustering in hash tables?

(b) Suppose we implement a hash table using double hashing. Is it possible for this hash table to have clustering?

(c) Suppose you know your hash table needs to store keys where each key's hash code is always a multiple of two. In that case, which resizing strategy should you use?

(d) Consider the following key-value pairs.

(6, a), (29, b), (41, d). (34, e), (10, f), (64, g), (50, h)

Suppose each key has a hash function $h(k) = 2k$. So, the key 6 would have a hash code of 12. Insert each key-value pair into the following hash tables and draw what their internal state looks like:

(i) A hash table that uses separate chaining. The table has an internal capacity of 10. Assume each bucket is a linked list, where new pairs are appended to the end. Do not worry about resizing.

(ii) A hash table that uses linear probing, with internal capacity 10. Do not worry about resizing.

(iii) A hash table that uses quadratic probing, with internal capacity 10. Do not worry about resizing.

(e) Consider the three hash tables in the previous question. What are the load factors of each hash table?

7. Debugging

Suppose we are trying to implement an algorithm `isTree(Node node)` that detects whether some binary tree is a valid one or not, and returns `true` if it is, and `false` if it isn't. In particular, we want to confirm that the tree does not contain any *cycles* – there are no nodes where the `left` and `right` fields point to their parents.

Assume that the node passed into the method is supposed to be the root node of the tree.

(a) List at least four different test cases for each problem. For each test case, be sure to specify what the input is (drawing the tree, if necessary), and what the expected output is (assuming the algorithm is implemented correctly).

(b) Here is one (buggy) implementation in Java. List every bug you can find with this algorithm.

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

    // We are simplifying what the equals method is supposed to look like in Java.
    // This method is technically invalid, but you may assume it's
    // implemented correctly.
    public boolean equals(Node n) {
        if (n == null) {
            return false;
```

```
        }
        return this.data == n.data;
    }

    public int hashCode() {
        // Pick a random number to ensure good distribution in hash table
        return Random.randInt();
    }
}

boolean isTree(Node node) {
    ISet<Node> set = new ChainedHashSet<>();
    return isTreeHelper(node, set)
}

boolean isTreeHelper(Node node, ISet<Node> set) {
    ISet<Node> set = new ChainedHashSet<>();
    if (set.containsKey(node)) {
        return false;
    } else {
        set.add(node);
        return isTreeHelper(node.left, set) || isTreeHelper(node.right, set);
    }
}
```

8. Design Decisions

There isn't a design decisions problem in this handout, but you should review the questions from previous weeks. Some design questions in previous sections:

Section 1: Problems 2 and 3
Section 4: Problems 8 and 10