Section 03: Recurrences, Master Theorem, Tree Method

Main Problems

1. Finding Bounds

For each of the following code blocks, what is the worst-case runtime? Give a big-Θ bound.

(a)

```
int result = 0;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        result++;
    }
}
```

The runtime for the double loop is $\sum_{i=0}^{n-1}\sum_{j=0}^{i-1} 1 = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$. This is $\Theta\left(n^2\right)$. See Section slides for more detail.

(b)

```
public IList<String> repeat(DoubleLinkedList<String> list, int n) {
    IList<String> result = new DoubleLinkedList<String>();
    for(String str : list) {
        for(int i = 0; i < n; i++) {
            result.add(str);
        }
    }
    return result;
}
```

The runtime is $\Theta\left(nm\right)$, where m is the length of the input list and n is equal to the int n parameter.
One thing to note here is that unlike many of the methods we've analyzed before, we can't quite describe the runtime of this algorithm using just a single variable: we need two, one for each loop.

(c)

```
public void foo(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 5; j < i; j++) {
            System.out.println("Hello!");
        }

        for (int j = i; j >= 0; j -= 2) {
            System.out.println("Hello!");
        }
    }
}
```

The inner loop executes about $i - 5 + i/2$ operations per loop. So we execute about

$$\sum_{i=0}^{n-1} i - 5 + i/2 = \frac{3}{2}\sum_{i=0}^{n-1} i - \sum_{i=0}^{n-1} 5 = \frac{3}{2} * \frac{(0 + n - 1) * n}{2} - 5n = \frac{3n(n-1)}{4} - 5n$$

which means the runtime is $\Theta\left(n^2\right)$.

(d)

```
public int num(int n){
    if (n < 10) {
        return n;
    } else if (n < 1000) {
        return num(n - 2);
    } else {
        return num(n / 2);
    }
}
```

The answer is $\Theta\left(\log(n)\right)$.
One thing to note is that the second case effectively has no impact on the runtime. That second case occurs only for $n < 1000$ – when discussing asymptotic analysis, we only care what happens with the runtime as n grows large.

(e)

```
public int foo(int n) {
    if (n <= 0) {
        return 3;
    }
    int x = foo(n - 1);
    System.out.println("hello");
    x += foo(n - 1);
    return x;
}
```

The answer is $\Theta(2^n)$.
In order to determine that this is exponential, let's start by considering the following recurrence:

$$T(n) = \begin{cases} 1 & \text{If } n = 0 \\ 2T(n-1) + 1 & \text{Otherwise} \end{cases}$$

While we could unfold this to get an exact closed form, we can approximate the final asymptotic behavior by taking a step back and thinking on a higher level what this is doing.
Basically, what happens is we take the work done by $T(n-1)$ and multiply it by 2. If we ignore the $+1$ constant work done in the recursive case, the net effect is that we multiply 2 approximately n times. This simplifies to 2^n.

2. Binary Search Trees

(a) Write a method `validate` to validate a BST. Although the basic algorithm can be converted to any data structure and work in any format, if it helps, you may write this method for the `IntTree` class:

```
public class IntTree {
    private IntTreeNode overallRoot;

    // constructors and other methods omitted for clarity

    private class IntTreeNode {
        public int data;
        public IntTreeNode left;
        public IntTreeNode right;

        // constructors omitted for clarity
    }
}
```

Solution:

```
public boolean validate() {
            return validate(overallRoot, Integer.MIN_VALUE, Integer.MAX_VALUE);
}

private boolean validate(IntTreeNode root, int min, int max) {
    if (root == null) {
        return true;
    } else if (root.data > max || root.data < min) {
        return false;
    } else {
        return validate(root.left, min, root.data - 1) &&
            validate (root.right, root.data + 1, max);
    }
}
```

3. Code To Recurrence

(a) Consider the following method.

```
public static int f(int N) {
    if (N <= 1) {
        return 0;
    }

    int result = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < i; j++) {
            result++;
        }
    }

    return 5 * f(N / 2) + 3 * result + 2 * f(N / 2) + f(N / 2) + f(N / 2);
}
```

Give a recurrence formula for the running time of this code. It's OK to provide a $\mathcal{O}$ for the non-recursive terms (for example if the running time is $A(n) = 4A(n/3) + 25n$, you need to get the 4 and the 3 right but you don't have to worry about getting the 25 right). Just show us how you got there.

Hint: Notice that the main loop is the exact same code as Problem 1A.

$$T(n) = \begin{cases} 1 & \text{When } n \leq 1 \\ \frac{n(n-1)}{2} + 4T(n/2) & \text{Otherwise} \end{cases}$$

We saw in Problem 1A that the runtime for the main loop is $\sum_{i=0}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$.

See Section slides for more detail.

(b) Consider the following method.

```
public static int g(n) {
    if (n <= 1) {
        return 1000;
    }
    if (g(n / 3) > 5) {
        for (int i = 0; i < n; i++) {
            System.out.println("Hello");
        }
        return 5 * g(n / 3);
    } else {
        for (int i = 0; i < n * n; i++) {
            System.out.println("World");
        }
        return 4 * g(n / 3);
    }
}
```

(i) Find a recurrence $S(n)$ modeling the worst-case runtime of `g(n)`.

$$S(n) = \begin{cases} 1 & \text{When } n \leq 1 \\ 2S(n/3) + n & \text{Otherwise} \end{cases}$$

Important: note that the if statement contains a recursive call that must be evaluated for $n > 1$.

(ii) Find a recurrence $X(n)$ modeling the *returned integer output* of `g(n)`.

$$X(n) = \begin{cases} 1000 & \text{When } n \leq 1 \\ 5T(n/3) & \text{Otherwise} \end{cases}$$

(iii) Find a recurrence $P(n)$ modeling the *printed output* of `g(n)`.

$$P(n) = 2P(n/3) + n$$

(c) Consider the following set of recursive methods.

```
public int test(int n) {
    IDictionary<Integer, Integer> dict = new AvlDictionary<>();
    populate(n, dict);
    int counter = 0;
    for (int i = 0; i < n; i++) {
        counter += dict.get(i);
    }
    return counter;
}

private void populate(int k, IDictionary<Integer, Integer> dict) {
    if (k == 0) {
        dict.put(0, k);
    } else {
        for (int i = 0; i < k; i++) {
            dict.put(i, i);
        }
        populate(k / 2, dict);
    }
}
```

(i) Write a mathematical function representing the *worst-case runtime* of `test`.

You should write two functions, one for the runtime of `test` and one for the runtime of `populate`.

The runtime of the `populate` method is:

$$P(k) = \begin{cases} \log(N) & \text{When } k = 0 \\ k\log(N) + P(k/2) & \text{Otherwise} \end{cases}$$

Here, N is the maximum possible value of n.
The runtime of the `test` method is then $R(n) = P(n) + n\log(n)$.

4. Master Theorem

For each of the recurrences below, use the Master Theorem to find the big-Θ of the closed form or explain why Master Theorem doesn't apply. (See the last page for the definition of Master Theorem.)

(a) $T(n) = \begin{cases} 18 & \text{if } n \leq 5 \\ 3T(n/4) + n^2 & \text{otherwise} \end{cases}$

This is the correct form for Master Theorem. We want to compare $\log_4(3)$ to 2. $\log_4(3)$ is between 0 and 1 (since $4^0 < 3 < 4^1$), so $\log_4(3) < 2$. We're thus in the case where the answer is $\Theta\left(n^2\right)$.

(b) $T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ 9T(n/3) + n^2 & \text{otherwise} \end{cases}$

We want to compare $\log_3(9)$ to 2. $\log_3(9)$ is 2 (since $3^2 = 9$) since the two things we're comparing are equal, we have $\Theta\left(n^2 \log n\right)$ as our final answer.

(c) $T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ \log(n)T(n/2) + n & \text{otherwise} \end{cases}$

This recurrence is not in the right form to use the Master Theorem. The coefficient of $T(n/2)$ needs to be a constant, not a function of n.

(d) $T(n) = \begin{cases} 1 & \text{if } n \leq 19 \\ 4T(n/3) + n & \text{otherwise} \end{cases}$

We want to compare $\log_3(4)$ to 1. $\log_3(4)$ is between 1 and 2 (since $3^1 < 4 < 3^2$), so $\log_3(4).1$. In this case, the Master Theorem says our result is $\Theta\left(n^{\log_3(4)}\right)$

(e) $T(n) = \begin{cases} 5 & \text{if } n \leq 24 \\ 2T(n-2) + 5n^3 & \text{otherwise} \end{cases}$

This recurrence is not in the right form to use Master Theorem. It's only applicable if we are dividing the input size, not if we're subtracting from it.

Hashing Problems

1. Hash Table Insertion!

(a) Suppose we have a hash table implemented using separate chaining. This hash table has an internal capacity of 10. Its buckets are implemented using a linked list where new elements are appended to the end. Do not worry about resizing.

Show what this hash table internally looks like after inserting the following key-value pairs in the order given using the hash function $h(x) = 4x$:

$$(1, a), (4, b), (2, c), (17, d), (12, e), (9, e), (19, f), (4, g), (8, c), (12, f)$$

(b) Consider the following scenario:

Suppose we have a hash table with an initial capacity of 12. We resize the hash table by doubling the capacity. Suppose we insert integer keys into this table using the hash function $h(x) = 4x$:

Why is this choice of hash function and initial capacity suboptimal? How can we fix it?

See slides for part (a)

Part (b) Notice that the hash function will initially always cause the keys to be hashed to at most one of three spots: 12 is evenly divided by 4.
This means that the likelihood of a key colliding with another one dramatically increases, decreasing performance.
This situation does not improve as we resize, since the hash function will continue to map to only a fourth of the available indices.
We can fix this by either picking a new hash function that's relatively prime to 12 (e.g. $h(x) = 5x$), by picking a different initial table capacity, or by resizing the table using a strategy other then doubling (such as picking the next prime that's roughly double the initial size).
See Section slides for more details.

2. More Hash Table Insertion!

For each problem, insert the given elements into the described hash table. Do not worry about resizing the internal array.

(a) Suppose we have a hash table that uses separate chaining and has an internal capacity of 12. Assume that each bucket is a linked list where new elements are added to the front of the list.

Insert the following elements in the EXACT order given using the hash function $h(x) = 4x$:

0, 4, 7, 1, 2, 3, 6, 11, 16

Solutions:

(a) To make the problem easier for ourselves, we first start by computing the hash values and initial indices:

key	hash	index (pre probing)
0	0	0
4	16	4
7	28	4
1	4	4
2	8	8
3	12	0
6	24	0
11	44	8
16	64	4

The state of the internal array will be

$6 \to 3 \to 0$	/	/	/	$16 \to 1 \to 7 \to 4$	/	/	/	$11 \to 2$	/	/	/