Section 03: Asymptotic Analysis

Section Problems

1. Binary Search Trees

(a) Write a method `validate` to validate a BST containing no duplicates. 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
    }
}
```

(b) Suppose we want to implement a method `findNode(V value)` that searches a binary search tree with n unique nodes for a given value.

(i) We want to analyze the runtime of our `findNode` method in the best possible case and the worst possible case. What does our tree look like in the best possible case? In the worst possible case? *Draw* two examples of binary search trees with up to 5 nodes each that would result in the best-case and worst-case runtimes of `findNode`.

(ii) What is the worst case big-Θ runtime for `findNode`?

(iii) What is the best case big-Θ runtime for `findNode`?

2. TreeMap implemented as a Binary Search Tree

Consider the following method, which is a part of a Binary Search Tree implementation of a TreeMap class.

```
public V find(K key) {
    return find(this.root, key);
}

private V find(Node<K, V> current, K key) {
    if (current == null) {
        return null;
    }
    if (current.key.equals(key)) {
        return current.value;
    }
    if (current.key.compareTo(key) > 0) {
        return find(current.left, key);
    } else {
        return find(current.right, key);
    }
}
```

(a) We want to analyze the runtime of our `find(x)` method in the best possible case and the worst possible case. What does our tree look like in the best possible case? In the worst possible case?

(b) Write a recurrence to represent the worst-case runtime for `find(x)` in terms of n, the number of elements contained within our tree. Then, provide a Θ bound.

(c) Assuming we have an optimally structured tree, write a recurrence for the runtime of `find(x)` (again in terms of n). Then, provide a Θ bound.

3. Code Analysis

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

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

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

(c)
```
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;
}
```

(d)
```
public boolean isPrime(int n) {
    int toTest = 2;
    while (toTest < n) {
        if (n % toTest == 0) {
            return false;
        } else {
            toTest++;
        }
     }
     return true;
}
```

4. Tree method walk-through

Consider the following recurrence: $A(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ 3A(n/6) + n & \text{otherwise} \end{cases}$

We want to find an *exact* closed form of this equation by using the tree method. Suppose we draw out the total work done by this method as a tree, as discussed in lecture. Let n be the initial input to A.

(a) What is the size of the input at level i (as in class, call the root level 0)?

(b) What is the number of nodes at level i?

Note: let $i = 0$ indicate the level corresponding to the root node. So, when $i = 0$, your expression should be equal to 1.

(c) What is the total work at the i^{th} **recursive** level?

(d) What is the last level of the tree?

(e) What is the work done in the base case?

(f) Combine your answers from previous parts to get an expression for the total work.

(g) Simplify to a closed form.

Note: you do not need to simplify your answer, once you found the closed form. Hint: You should use the finite geometric series identity somewhere while finding a closed form.

5. More tree method recurrences

For each of the following recurrences, find their closed form using the tree method. It may be a useful guide to use the steps from section 4 of this handout to help you with all the parts of solving a recurrence problem fully.

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

(b) $S(q) = \begin{cases} 1 & \text{if } q = 1 \\ 2S(q-1) + 1 & \text{otherwise} \end{cases}$

(c) $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 8T(n/2) + 4n^2 & \text{otherwise} \end{cases}$

Useful summation identities

Splitting a sum

$$\sum_{i=a}^{b}(x+y) = \sum_{i=a}^{b} x + \sum_{i=a}^{b} y$$

Adjusting summation bounds

$$\sum_{i=a}^{b} f(x) = \sum_{i=0}^{b} f(x) - \sum_{i=0}^{a-1} f(x)$$

Factoring out a constant

$$\sum_{i=a}^{b} cf(i) = c\sum_{i=a}^{b} f(i)$$

Summation of a constant

$$\sum_{i=0}^{n-1} c = \underbrace{c + c + \ldots + c}_{n \text{ times}} = cn$$

Note: this rule is a special case of the rule on the left

Gauss's identity

$$\sum_{i=0}^{n-1} i = 0 + 1 + \ldots + n - 1 = \frac{n(n-1)}{2}$$

Sum of squares

$$\sum_{i=0}^{n-1} i^2 = \frac{n(n-1)(2n-1)}{6}$$

Finite geometric series

$$\sum_{i=0}^{n-1} x^i = \frac{x^n - 1}{x - 1}$$

Infinite geometric series

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$$

Note: applicable only when $-1 < x < 1$

Useful logarithm identities

Note: we assume here that in all cases, variables are non-zero.

Log of a product

$$\log_b(x \cdot y) = \log_b(x) + \log_b(y)$$

Log of a fraction

$$\log_b\left(\frac{x}{y}\right) = \log_b(x) - \log_b(y)$$

Log of a power

$$\log_b(x^y) = y \cdot \log_b(x)$$

Power of a log

$$x^{\log_b(y)} = y^{\log_b(x)}$$

Change of base

$$\log_b(x) = \frac{\log_d(x)}{\log_d(b)}$$