# 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-$\Theta$ bound.

(a)

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

(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;
}
```

(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!");
        }
    }
}
```

(d)

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

(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;
}
```

## 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
    }
}
```

## 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.

(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)`.

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

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

## 4. Master Theorem

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

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

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

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

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

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

# 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?

## 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

## Master Theorem

For recurrences in this form, where $a, b, c, e$ are constants:

$$T(n) = \begin{cases} d & \text{if } n \leq \text{some constant} \\ aT(n/b) + e \cdot n^c & \text{otherwise} \end{cases}$$

$$T(n) \text{ is } \begin{cases} \Theta(n^c) & \text{if } \log_b(a) < c \\ \Theta(n^c \log n) & \text{if } \log_b(a) = c \\ \Theta\left(n^{\log_b(a)}\right) & \text{if } \log_b(a) > c \end{cases}$$

## 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

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

### Finite geometric series

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

### Infinite geometric series

$$\sum_{i=0}^{\infty} x^i = 1 + x + x^2 + \ldots = \frac{1}{1-x}$$

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

# Useful Log Rules

## Power of a log identity

$a^{\log_b c} = c^{\log_b a}$

## Product rule

$\log_c(a * b) = \log_c a + \log_c b$

## Quotient rule

$\log_c(a/b) = \log_c a - \log_c b$

## Power rule

$\log_c(a^b) = b * \log_c a$

## Change of base formula

$\log_b a = (\log_c a)/(\log_c b)$