# CS 61BL Summer 2026

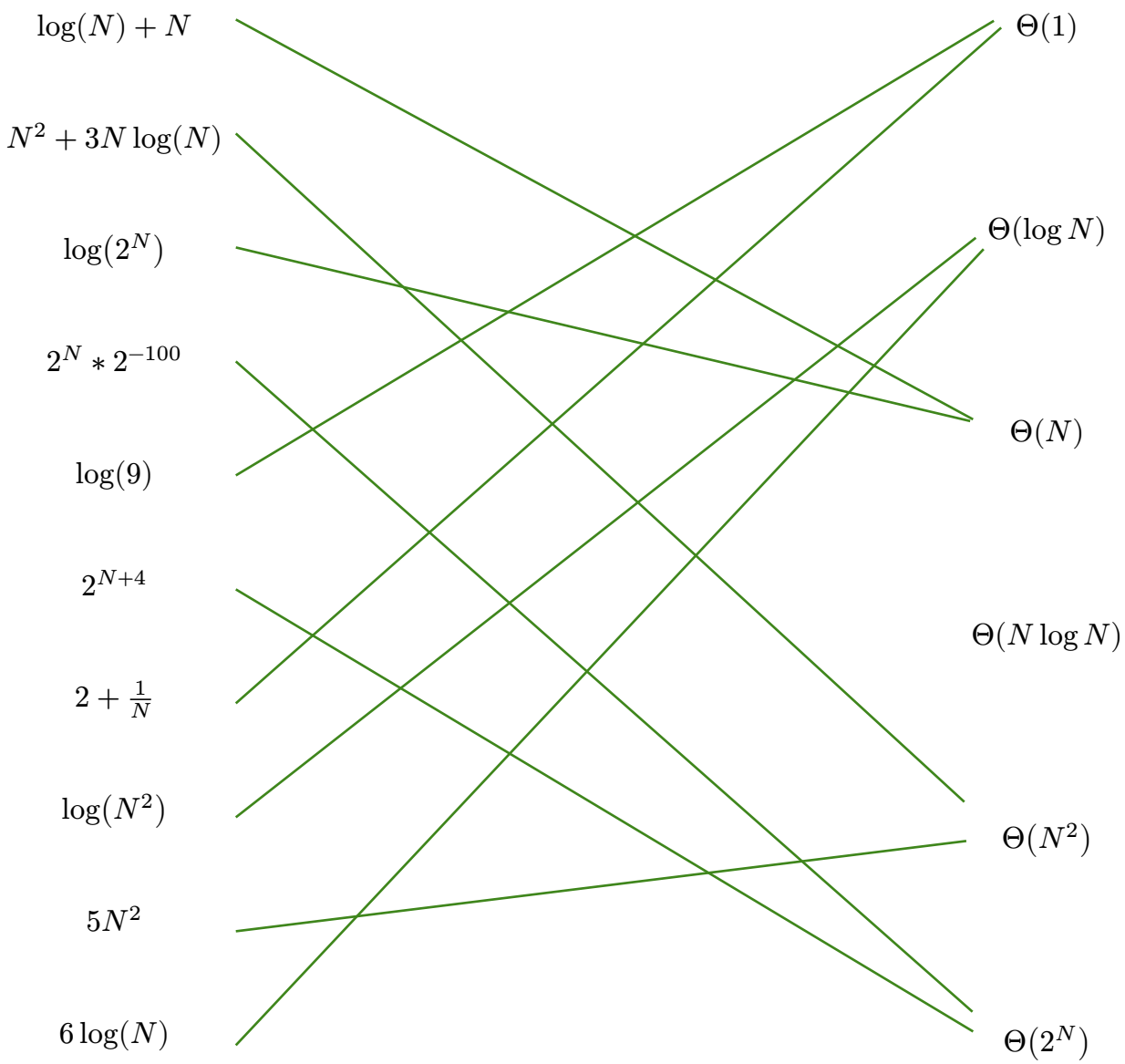
# Asymptotics and Trees

Exam Prep 01: July 10th, 2026

## 1 The Space-Time Continuum

For each growth function on the left, Theta bound it, simplify it, and draw a line to the matching Theta bound on the right.

*Recall the rules for simplifying asymptotic bounds and logarithm rules!*



# 2 Asymptotics

Give the runtime of the following functions in $\Theta$ notation. Your answer should be as simple as possible with no unnecessary leading constants or lower order terms.

*In CS 61B, we always assume* ***IO.println*** *runs in* $\Theta(1)$

```
private void f1(int N) {
  for (int i = 1; i < N; i++) {
    for (int j = 1; j < i; j++) {
        IO.println("dwang 1.0");
    }
  }
}
```

Runtime: $\Theta\ (\ N^2\ )$

The inner loop does up to $i$ work each time, and the outer loop increments $i$ each time. Summing over each loop, we get that $1 + 2 + 3 + 4 + ... + N = \Theta(N^2)$.

```
private void f2(int N) {
  for (int i = 1; i < N; i *= 2) {
    for (int j = 0; j < i; j++) {
        IO.println("dwang 2.0");
    }
  }
}
```

Runtime: $\Theta\ (\ N\ )$

The inner loop does $i$ work each time, and we double $i$ each time until reaching $N$. $1 + 2 + 4 + 8 + ... + N = \Theta(N)$

# 3 I Am Speed

For each code block below, fill in the blank(s) so that the function has the desired runtime. If the answer is impossible, just write "impossible" in the blank. Assume that `System.out.println` runs in constant time. You may use Java's `Math.pow(x, y)` to raise `x` to the power of `y`.

(a) Desired Runtime: $\Theta(N)$

```
public static void f1(int N) {
    for (int i = 1; i < N; i += 1){
        System.out.println("hi Selina");
    }
}
```

Note the solution could be `i += C`, where `C` is some constant independent of `N`. This is because even if we did for example, `i += 10`, we would do $\frac{N}{10}$ work in total, which is still $\Theta(N)$.

(b) Desired Runtime: $\Theta(\log N)$

```
public static void f2(int N) {
    for (int i = 1; i < N; i *= C) {
        System.out.println("howdy Caitlin");
    }
}
```

Here, the solution could be `i *= C`, where `C` is some constant independent of `N`. This is because even if we did for example, `i *= 5`, we would do $\log_5 N$ work in total, and in general $\log_i N$ work, which is still $O(\log n)$.

(c) Desired Runtime: $\Theta(1)$

```
public static void f3(int N) {
    for (int i = 1; i < 1000; i += 1) {
        System.out.println("hello Dawn");
    }
}
```

Again, the solution is actually just `i < C`, where `C` is some constant independent of the input `N`. Any other syntactically valid constant would work.

(d) Desired Runtime: $\Theta(2^N)$

This one is tricky! *Hint: think about the dominating term in* $1 + 2 + 4 + 8 + ... + f(N)$

```
public static void f4(int N) {
    for (int i = 1; i < Math.pow(2, N); i *= 2) {
        for (int j = 0; j < i; j += 1) {
            System.out.println("what's up Alyssa");
        }
    }
}
```

# 4 Re-cursed with Asymptotics

(a) What is the runtime of the code below in terms of $N$?

```
public static int curse(int n) {
    if (n <= 0) {
        return 0;
    } else {
        return n + curse(n - 1);
    }
}
```

Runtime: $\Theta(N)$

On each recursive call, we do a constant amount of work. We make $n$ recursive calls, because we go from $n$ to 1. Then $n$ recursive layers with 1 work at each layer is overall $\Theta(n)$ work.

(b) Can you find a runtime bound for the code below? We can assume the `System.arraycopy` method takes $\Theta(N)$ time, where $N$ is the number of elements copied. The official signature is `System.arrayCopy(Object sourceArr, int srcPos, Object dest, int destPos, int length)`. Here, `srcPos` and `destPos` are the starting points in the source and destination arrays to start copying and pasting in, respectively, and `length` is the number of elements copied.

```
public static void silly(int[] arr) {
    if (arr.length <= 1) {
        System.out.println("You won!");
        return;
    }

    int newLen = arr.length / 2;
    int[] firstHalf = new int[newLen];
    int[] secondHalf = new int[newLen];

    System.arraycopy(arr, 0, firstHalf, 0, newLen);
    System.arraycopy(arr, newLen, secondHalf, 0, newLen);

    silly(firstHalf);
    silly(secondHalf);
}
```

Runtime: $\Theta(N \log N)$

At each level, we do $N$ work, because the call to **`System.arraycopy`**. You can see that at the top level, this is $N$ work. At the next level, we make two calls that each operate on arrays of length N/2, but that total work sums up to $N$. On the level after that, in four separate recursive function frames we'll call **`System.arraycopy`** on arrays of length N/4, which again sums up to $N$ for that whole layer of recursive calls.

Now we look for the height of our recursive tree. Each time, we halve the length of N, which means that the length of the array $N$ on recursive level $k$ is roughly $N * \left(\frac{1}{2}\right)^k$. Then we will finally reach our base case $N \leq 1$ when we have $N * \left(\frac{1}{2}\right)^k = 1$. Doing some math, we see this can be transformed into $N = 2^k$, which means $k = \log_2(N)$. In other words, the number of layers in our recursive tree is $\log_2(N)$. If we have $\log_2(N)$ layers with $\Theta(N)$ work on each layer, we must have $\Theta(N \log(N))$ runtime.

(c) Given that **exponentialWork** runs in $\Theta(3^N)$ time with respect to input $N$, what is the runtime of **amy**?

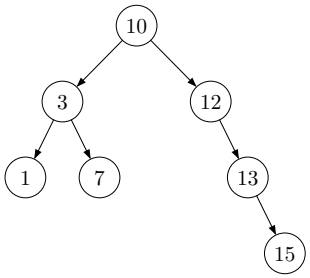
```
public void amy(int N) {
    if (N <= 1) {
        return;
    }
    amy(N - 2);
    amy(N - 2);
    amy(N - 2);
    exponentialWork(N); // Runs in Theta(3^N) time
}
```

Runtime: $\Theta\ (\ 3^N\ )$

Drawing out the recursive tree, the first level has $\Theta(3^N)$ work, the next level has $\Theta(3^{N-1})$ work, and so on until the last level which has approximately $\Theta\left(3^{\frac{N}{2}}\right)$ work. This gives the sum $\Theta(3^N) + \Theta(3^{N-1}) + ... + \Theta\left(3^{\frac{N}{2}}\right) = \Theta(3^N)$.

# 5 Numerical Trees

(a) Write the DFS pre-order, DFS in-order, DFS post-order, and BFS traversals of the following binary tree. For all traversals, process child nodes left to right.




Pre-order: $10, 3, 1, 7, 12, 13, 15$

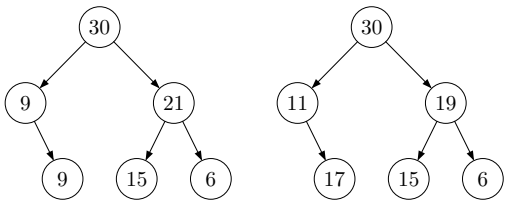
In-order: $1, 3, 7, 10, 12, 13, 15$

Post-order: $1, 7, 3, 15, 13, 12, 10$

BFS: $10, 3, 12, 1, 7, 13, 15$

**Pre-order**: 10, 3, 1, 7, 12, 13, 15
**In-order**: 1, 3, 7, 10, 12, 13, 15
**Post-order**: 1, 7, 3, 15, 13, 12, 10
**BFS**: 10, 3, 12, 1, 7, 13, 15

(b) Complete isSumTree below. In a sum tree, the value at each non-leaf node equals the sum of its children. For example, the left binary tree is a sum tree, but the right one is not.



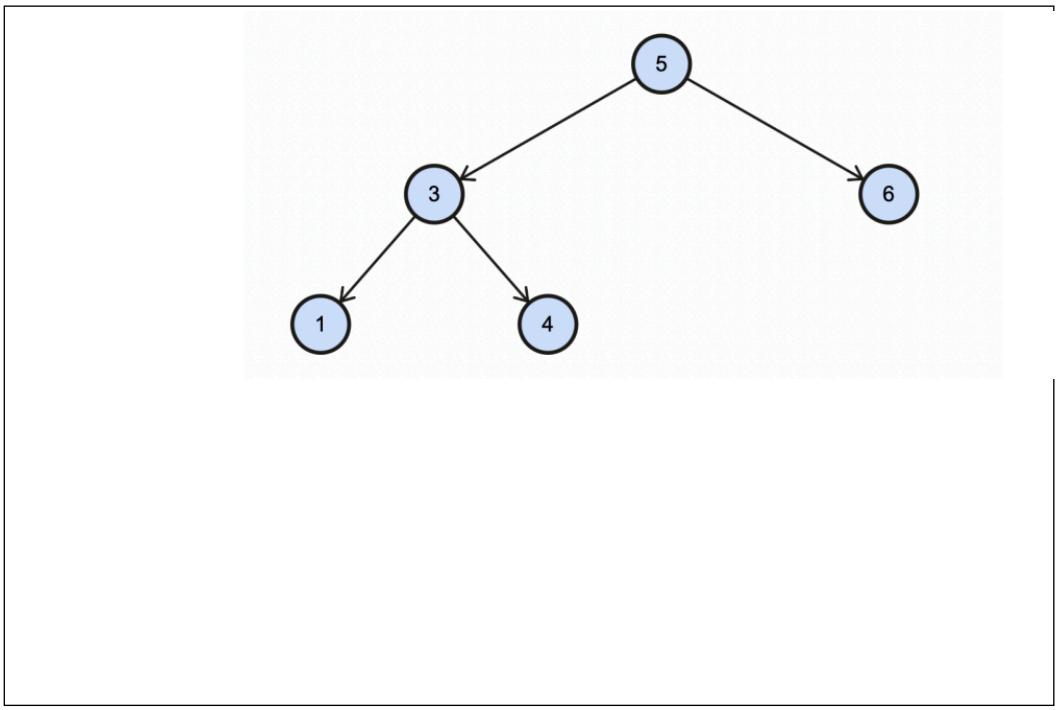

```
public class Tree {
  public int value;
  public Tree left, right;
  public Tree() {}
  public Tree(int val) { value = val; }
}

public boolean isSumTree(Tree t) {
  if (t == null || (t.left == null && t.right == null) {
    return true;
  }

  int left = 0, right = 0;
  if (t.left != null) {
    left = t.left.value;
  }
  if (t.right != null) {
    right = t.right.value;
  }

  boolean leftValid = isSumTree(t.left);
  boolean rightValid = isSumTree(t.right);

  return (t.value == left + right) && leftValid && rightValid;
}
```

# 6 Binary Search Trees

(a) We implement a binary search tree with the following methods:

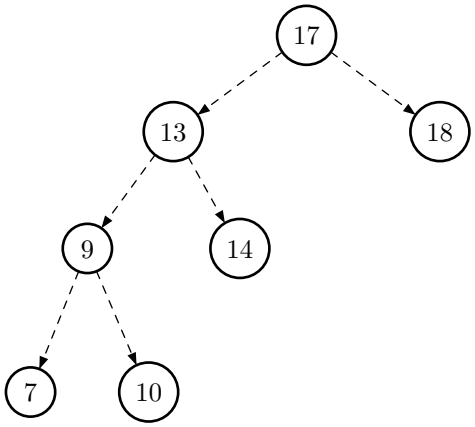
```
// Inserts an item into the binary search tree.
public void insert(T item) {
    // Implementation has been omitted
}

// Deletes an item from the binary search tree.
public void delete(T item) {
    // Implementation has been omitted
}
```

Draw the binary search tree that results from the following operations. Assume we start from an empty tree.




```
insert(5)
insert(7)
insert(10)
insert(6)
insert(3)
insert(1)
insert(4)
delete(10)
delete(7)
```

(b) Given the following binary search tree:




Suppose we delete the root node. Which node(s) can we replace 17 with as the new root node?

Upon deleting the root node, we can promote either the node containing 14 or 18 as our new root node. When choosing a new root node, as per BST properties, the new root must be greater than all nodes on the left subtree and less than all nodes on the right subtree (this is the working behind Hibbard Deletion from lecture).

(c) Suppose we create a BST by inserting the nodes $V_0, V_1, ..., V_n$, where $V_i$ is strictly smaller than $V_{\{i+1\}}$, in order. That is, we first insert $V_0$, then $V_1$, and so on. What is the runtime to find an element in this BST in the worst case, where $N$ is the number of nodes?

The runtime for a search on this BST is $\Theta(N)$. This is essentially a linked list, where in the worst case the item being searched for is at the very end of the list.