CS 61B
Spring 2026

Asymptotics II
Discussion 07: March 2, 2026

1 Asymptotics Summary

Asymptotic Notation Recap. Given a function $R(N)$:

- $R(N) \in \Theta(f(N))$: This means $R(N)$ grows at the **same rate** as $f(N)$.
- $R(N) \in O(f(N))$: This means $R(N)$ grows **no faster than** $f(N)$. Used as an **upper bound**.
- $R(N) \in \Omega(f(N))$: This means $R(N)$ grows **no slower than** $f(N)$. Used as a **lower bound**.

Informally, we can think of Θ as $=$, O as $\leq$, and Ω as $\geq$. Suppose $R(N) = N^3 + 2N + \cos(N)$, then all of the following are true:

- $R(N) \in \Theta(N^3)$
- $R(N) \in O(N^3)$
- $R(N) \in O(N^{1271})$
- $R(N) \in \Omega(N^2)$
- $R(N) \in \Omega(1)$

If the asymptotic runtime depends on the input, we **cannot** give a Θ bound for the runtime of the code. For example, suppose we want to characterize the runtime $R(N)$ of a function `contains` that searches an unsorted array to see if some item is present, where N is the length of the array. All four of the following statements are true:

- The worst case runtime of `contains` is $\Theta(N)$.
- The runtime of `contains` is $O(N)$.
- The best case runtime of `contains` is $\Theta(1)$.
- The runtime of `contains` is $\Omega(1)$.

Useful Sum Formulas.

Arithmetic: $$1 + 2 + 3 + \cdots + Q = \frac{Q(Q+1)}{2} \in \Theta(Q^2)$$

Geometric: $$1 + 2 + +8 + \cdots + Q = 2Q - 1 \in \Theta(Q)$$

Analyzing Recursive Functions (Tree Method). To find the runtime of a recursive function:

(a) **Draw the call tree.** Each node represents one call; label it with the non-recursive work done in that call (e.g. the cost of loops, helper calls).
(b) **Sum each layer.** Add up the work across all nodes at the same depth.
(c) **Count the layers.** Determine the height of the tree (how many times can N be divided/reduced before hitting the base case?).
(d) **Total it up.** Total work = (work per layer) × (number of layers), or sum the per-layer costs if they vary.

Note, non-61B sources are likely to write the geometric sum as shown below. If you like this equivalent formulation better, it's fine to use this instead:

Geometric: $$1 + 2 + 4 + \cdots + 2^k = 2^{k+1} - 1 \in \Theta(2^k)$$

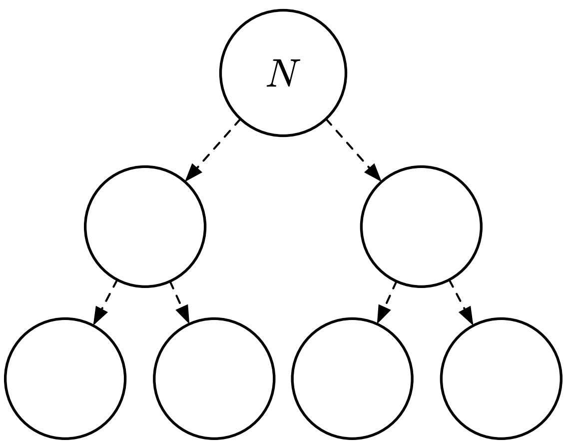
2 Assisted Forestry

When analyzing the runtime of recursive functions, it's almost always a good idea to map out the recursive calls using a visual diagram. Consider the function below, where **`blocky(int M)`** is a static method that runs in $\Theta(M)$

```
public static void blobby(int N) {
  if (N <= 1) { return; }
  blocky(N);
  blobby(N / 2);
  blobby(N / 2);
}
```

It looks a bit complicated to analyze! Let's break it down.

(a) Fill out **each node** of the recursive call tree below with the amount of work done relative to N on each respective call of **`blobby`**. The first call has been filled in for you. Then, sum up the work across each layer, and try to predict a rule for the amount of work per layer.



Amount of work on layer 0: ____

Amount of work on layer 1: ____

Amount of work on layer 2: ____

⋮

Amount of work on layer l: ____

(b) Now, we need to figure out how many layers the tree has. This is equivalent to asking the question *"how many times can N be divided by 2 before it hits the base case of 1?"*

$l =$ ____. *Hint: l is a function of N.*

(c) Now, add up the work on all l layers.

There are ____ layers, and, since ____ work is done on each layer,

the total runtime is equal to ____ $*$ ____, which is in $\Theta\left(\underline{\qquad}\right)$.

3 Solo Forestry

Drawing tree diagrams is applicable to a wide variety of asymptotics problems! Let's try another. Once again, `blocky(int M)` is a static method that runs in $\Theta(M)$.

```
public static int blobbier(int N) {
  if (N <= 0) { return 0; }
  blocky(N);
  return 17 + blobbier(N - 1);
}
```

(a) Draw a recursive call tree, starting with the initial call. Then, analyze the amount of work across each layer.

Hint: How much work is done in layer 0: 1, l, or N?

Amount of work on layer 0: ________

Amount of work on layer 1: ________

Amount of work on layer 2: ________

⋮

Amount of work on layer l: ________

(b) Now, we need to figure out how many layers does the tree have?

$l =$ ________

(c) Now, add up the work on all l layers, and theta-bound it.

What is the final runtime?

Hint: Remember the sum formulas.

Runtime: Θ (________)

4 The Re-Cursed Swamp

Sometimes, it isn't possible to give a theta bound for an entire function. In these cases, it's best to analyze the best and worst-case inputs and evaluate the runtime on those to produce a "tightest" upper and lower bound.

(a) Consider the function below...

```
public static int curse(int N) {
    if (N % 2 = 0 || N <= 0) {
        return 0;
    } else {
      for (int i = 0; i < N; i++) {
        System.out.println("You have been cursed!");
      }
      return curse(N - 2);
    }
}
```

What type of input will result in the best-case runtime? What type of input will result in the worst-case runtime?

Give a tight Ω and O bound that correspond to the lower and upper bounds on this function's runtime. That is, don't just say sometihng like $O(2^N)$ which is technically true, but useless.

Feel free to use the tree-drawing technique from questions 1 and 2!

Lower bound: Ω (________)	Upper bound: O (________)

(b) Give the tightest runtime bound(s) for the function 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) {
      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);
}
```

(c) Given that **exponentialWork** runs in $\Theta(3^N)$ time with respect to input N, give the tightest runtime bound(s) for **yellowWood**.

Hint: This one is hard! Drawing trees will be of utmost importance. If you suspect that the runtime cannot be theta-bounded, drawing one for the best case and one for the worst case can be a good idea.

```
public void yellowWood(int N) {
    if (Math.random() > 0.9) {
      twoPathsDiverge(N, 2);
    } else {
      twoPathsDiverge(N, 1);
    }
}

private void twoPathsDiverge(int N, int j) {
  if (N <= 1) {
      return;
  }
  exponentialWork(N);
  for (int i = 0; i < 3; i++) {
    twoPathsDiverge(N - j, j);
  }
}
```

5 Asymptotics of Weighted Quick Union

Note: for all big Ω and big O bounds, give the *tightest* bound possible.

(a) Suppose we have a Weighted Quick Union (WQU) without path compression with N elements.

1. What is the runtime, in big Ω and big O, of `isConnected`?

Lower bound: Ω (________)	Upper bound: O (________)

2. What is the runtime, in big Ω and big O, of `connect`?

Lower bound: Ω (________)	Upper bound: O (________)

(b) Suppose we add the method `addToWQU` to a WQU without path compression. The method takes in a list of `elements` and `connects` them in a random order, stopping when all elements are connected. Assume that all the `elements` are disconnected before the method call.

```
void addToWQU(int[] elements) {
    int[][] pairs = pairs(elements);
    for (int[] pair: pairs) {
        if (size() == elements.length) {
            return;
        }
        connect(pair[0], pair[1]);
    }
}
```

The `pairs` method takes in a list of `elements` and generates all possible pairs of elements in a random order. For example, `pairs([1, 2, 3])` might return `[[1, 3], [2, 3], [1, 2]]` or `[[1, 2], [1, 3], [2, 3]]`.

The `size` method calculates the size of the largest component in the WQU.

Assume that `pairs` and `size` run in constant time.

What is the runtime of `addToWQU` in big Ω and big O?

Lower bound: Ω (________)	Upper bound: O (________)

Hint: Consider the number of calls to `connect` *in the best case and worst case. Then, consider the best/ worst case time complexity for one call to* `connect`*.*

(c) Let us define a **matching size connection** as `connecting` two components in a WQU of equal size. For instance, suppose we have two trees, one with values 1 and 2, and another with the values 3 and 4. Calling `connect(1, 4)` is a matching size connection since both trees have 2 elements.

What is the **minimum** and **maximum** number of matching size connections that can occur after executing `addToWQU`? Assume `N`, i.e. `elements.length`, is a power of two. Your answers should be exact.