

## 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  $\Theta$  as  $=$ ,  $O$  as  $\leq$ , and  $\Omega$  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  $\Theta$  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.**

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

$$\text{Geometric:} \quad 1 + 2 + 4 + \dots + Q = 2Q - 1 \in \Theta(Q)$$

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

- 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).
- Sum each layer.** Add up the work across all nodes at the same depth.
- Count the layers.** Determine the height of the tree (how many times can  $N$  be divided/reduced before hitting the base case?).
- Total it up.** Total work = (work per layer)  $\times$  (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:

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

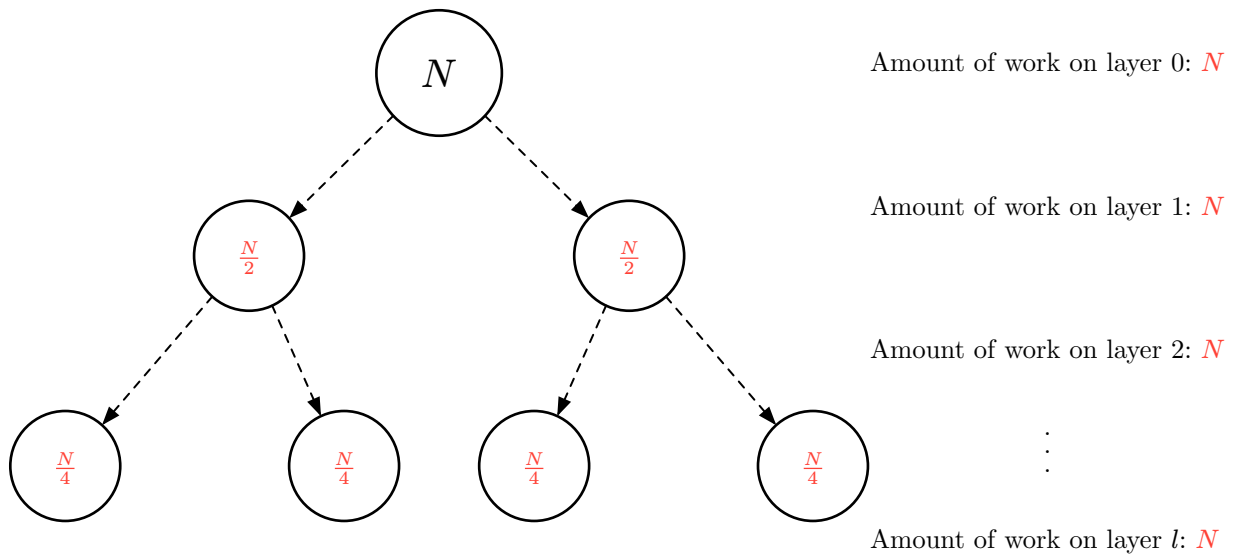
## 2 Assisted Forestry

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.



- (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?*”

$$l = \log N.$$

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

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

There are  $\log N$  layers, and, since  $N$  work is done on each layer, the total runtime is equal to  $\log N * N$ , which is in  $\Theta(N \log N)$ .

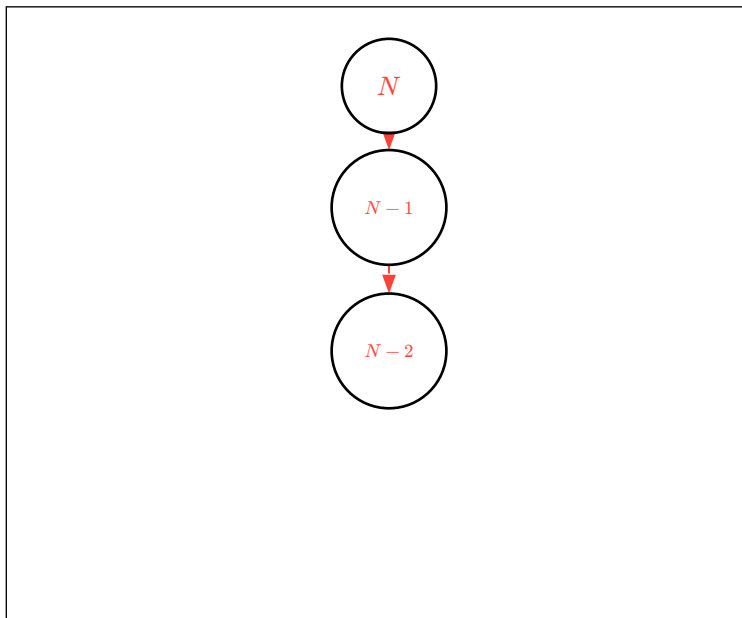
### 3 Solo Forestry

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 N + blobbier(N - 1);
}
```

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

*Hint: the rule for the amount of work per layer might be a function of  $l$ !*



Amount of work on layer 0:  $N$

Amount of work on layer 1:  $N - 1$

Amount of work on layer 2:  $N - 2$

⋮

Amount of work on layer  $l$ :  $N - l$

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

$$l = N$$

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

What is the final runtime?

*Hint: Remember the sum formulas!*

This is an arithmetic sum! Writing out the sum of the work from the topmost node to the bottom, we get

$$N + (N - 1) + (N - 2) + \dots + 1$$

$$= 1 + 2 + \dots + N$$

$$= \Theta(N^2)$$

Runtime:  $\Theta(N^2)$

## 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?

The best-case input is an even number. In this case, the top conditional will immediately cause the function to return, resulting in a  $\Omega(1)$  lower bound.

The worst-case input is an odd number. In this case, the top conditional will never cause the function to return, resulting in an  $O(N^2)$  upper bound.

Give a tight  $\Omega$  and  $O$  bound that correspond to the lower and upper bounds on this function's runtime. That is, don't just say something 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: $\Omega(1)$	Upper bound: $O(N^2)$
--------------------------	-----------------------

- (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);
}
```

$\Theta(N)$

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  $N * (\frac{1}{2})^k$ . Then we will finally reach our base case  $N \leq 1$  when we have  $N * (\frac{1}{2})^k = 1$ . 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  $N$ , 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);
    }
}
```

$\Omega(3^N), O(N * 3^N)$

For the best case, 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  $\Theta(3^{\frac{N}{2}})$  work. This gives the sum  $\Theta(3^N) + \Theta(3^{N-1}) + \dots + \Theta(3^{\frac{N}{2}}) = \Theta(3^N)$ .

For the worst case, each level has  $\Theta(3^N)$  work. Given that there are  $\sim N$  layers, summing up the tree results in  $O(N * 3^N)$  for an upper bound.

## 5 Asymptotics of Weighted Quick Union

Note: for all big  $\Omega$  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  $\Omega$  and big  $O$ , of `isConnected`?

Lower bound: $\Omega ( 1 )$	Upper bound: $O ( \log N )$
-----------------------------	-----------------------------

2. What is the runtime, in big  $\Omega$  and big  $O$ , of `connect`?

Lower bound: $\Omega ( 1 )$	Upper bound: $O ( \log N )$
-----------------------------	-----------------------------

In the best-case, if we're checking if **a** and **b** are connected, **a** is the root, and **b** is a node directly below the root. This means we only have to traverse one edge of the tree, which is constant time. In the worst-case, we have to traverse the entire height of the tree, and Weighted Quick Union gives us a worst-case height of  $\log N$ , hence the upper-bound of  $O(\log N)$ . Similar logic applies to the `connect` method.

- (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  $\Omega$  and big  $O$ ?

Lower bound: $\Omega ( N )$	Upper bound: $O ( N^2 \log N )$
-----------------------------	---------------------------------

*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`.*

Note that the if-statement terminates the method when the disjoint set becomes fully connected. The best case occurs when there is a sequence of pairs such that each `connect()` operation takes constant time and the tree becomes connected as quickly as possible. This will happen if we have a sequence  $(0, 1), (0, 2), \dots, (0, N - 1)$ , which consists of  $N - 1$  operations each taking constant time (ie. the best case for `connect` from part a). Note that long running-times occur when an element (e.g. 0) is not connected for many operations, and in the worst-case, 0 is not connected until the last  $N$  operations. This results in a tree of height  $\log N$  and requires up to  $N^2 - N + 1$  iterations.

- (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.

minimum: 1, maximum:  $N - 1$

The minimum number occurs for the sequence above, where there is only one matching size connection:  $(0, 1)$ . The maximum number is a bit more tricky, but occurs if we pairwise-connect the elements together, then pairwise connect those, and so on. An example for  $N=8$  elements is as follows:  $(0, 1), (2, 3), (4, 5), (6, 7), (0, 2), (4, 6), (0, 4)$ . In general, there are  $\frac{N}{2}$  matching-size connections of size 1,  $\frac{N}{4}$  matching-size connections of size 2, and so on, up until one matching-size connection of size  $\frac{N}{2}$ . This is the sum  $\frac{N}{2} + \frac{N}{4} + \frac{N}{8} + \dots + 2 + 1$ , which simplifies to  $N - 1$ .