# Asymptotics

## Topical Review Session 04

CS 61B // Spring 2021

# Agenda

Asymptotics

❑ Importance of Asymptotics
❑ Introduction to Asymptotics
❑ Orders of Growth
❑ Symbols and Representation
❑ Examples

# Importance of Asymptotics

# Why do we care about runtime?

- **Runtimes** are how we measure how well something we code performs.
- When writing code for practical application, we care how long it takes to run.
- When measuring asymptotics, we see how runtime scales with the size of our input.

*Example:* if Bing takes 5 minutes every time you search something but Google only takes 5 seconds to get the same results for the same input, we would want to use Google over Bing.

# Examples

For each scenario, pick which data structure would be better: ArrayList or SLList (not Deque)

1. If we will be values stored across the entire data structure (so not just the first or last element) more often than we are adding values to the data structure.

2. If we used a method `insert(T val, int i)` where it insert `val` at location `i`.

# Examples

For each scenario, pick which data structure would be better: ArrayList or SLList (not Deque)

1. If we will be getting values stored across the entire data structure (so not just the first or last element) more often than we are adding values to the data structure.
   We would pick the ArrayList since we are mostly getting values instead of inserting and ArrayList's get method is faster than the LinkedList get method.

2. If we used a method `insert(T val, int i)` where it insert `val` at location `i`.
   We would pick LinkedList over ArrayList this time since LinkedList has a faster insertion time for randomly inserting elements across the list, whereas an ArrayList would have to move elements over to accommodate for the element we are inserting.

# Introduction to Asymptotics

# Basic Runtimes

```
public void constantTimeOperation(int a, int b) { // This takes 1 operation
     int x = a + b;
}
public void constantTimeOperation2(String s) {    // This takes 1 operation
     System.out.println(s);
}
public void linearTimeOperation(int n) {          // This takes 1+1+...+1=n operations
     for(int i = 0; i < n, i++) {
          System.out.println(i);
     }
}
public void quadraticTimeOperation(int n) {       // This takes n+n+...+n=n^2 operations
     for(int i = 0; i < n, i++) {
          for(int j = 0; j < n, j++) {
               System.out.println(i+j);
          }
     }
}
```

# Notation

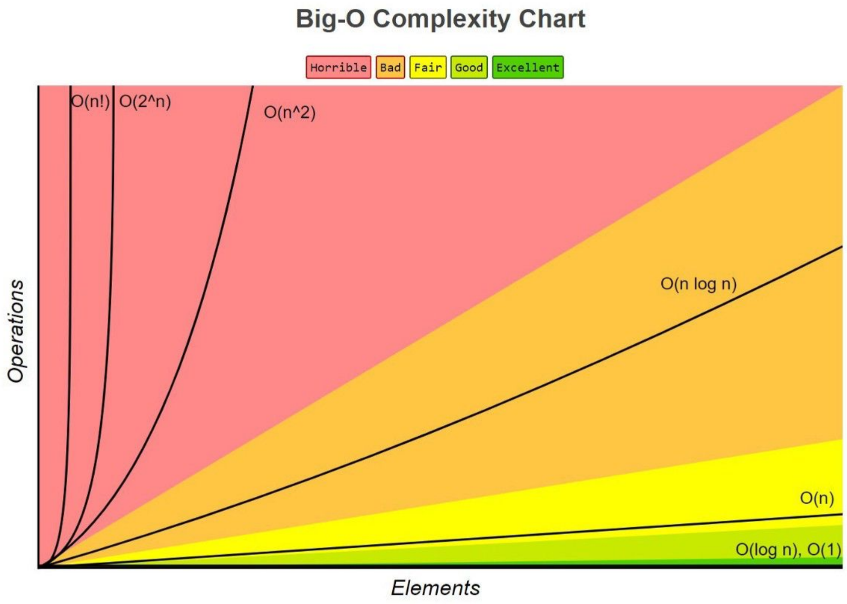
- We treat numeric computations, print statements, and other basic operations as **constant time operations**. We base our asymptotic analysis on how many of these we do.
- We refer to all runtimes in terms of algebraic formulas, usually with respect to some variable **n**. Depending on the situation given, this variable can refer to a number of things
    - Size of an array
    - A specific input
    - Etc.

```java
public void printList(int[] arr) {
    int N = arr.length;
    for(int i = 0; i < N, i++) {
        System.out.println(arr[i]);
    }
}
// This takes N operations, where N is the
length of the array

public void recursiveCountDown(int N) {
    System.out.println(N);
    if (N > 0) {
        recursiveCountDown(N - 1);
    }
}
// This also takes N operations, where N is
some int we pass in
```

# Simplifying Runtimes

- Asymptotics are a measure of how well a system scales - not a measure of performance for an individual case
- What matters is the proportion of how long it takes to run as we change the input size from maybe 10 to 1,000 to 1,000,000
- So, when we discuss runtime, the metric that matters is relative orders of magnitude
- When computing runtimes, always put it in terms of the leading coefficient of **n**



# Simplifying Runtimes

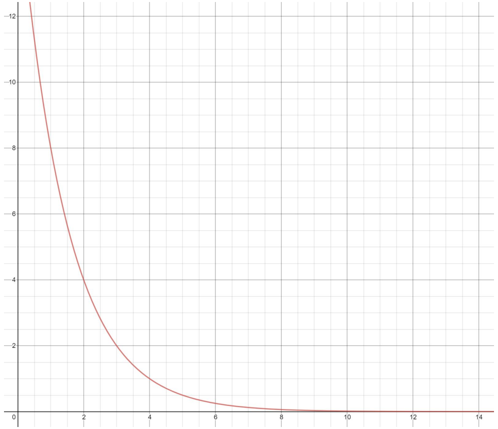
- For example, the runtime **O(5n+23)=O(1,000n + 6)=O(n)** and all of which are better than **n^2**

| Runtime | n=1 | n=10 | n=100 | n=10,000 | n=1,000,000 |
|---|---|---|---|---|---|
| **1** | 1 | 1 | 1 | 1 | 1 |
| **n** | 1 | 10 | 100 | 10,000 | 1,000,000 |
| **5n+23** | 28 | 73 | 523 | 50,023 | 5,000,023 |
| **1000n+6** | 1,006 | 10,006 | 100,006 | 10,000,006 | 1,000,000,006 |
| **n^2** | 1 | 100 | 10,000 | 100,000,000 | 1,000,000,000,000 |

# Logarithmic Runtimes

- All algorithms with a logarithmic runtime are treated equally because of change of base. (The math behind this isn’t in scope)
- We see this runtime is logarithmic because if we let `n = 2`$^x$, we see that x decreases by 1 every iteration and `x = log(n)`.

```
public void recursiveHalving(int n) {
     if (n > 0) {
          recursiveHalving(n/2);
     }
} //This takes log(N) operations
```



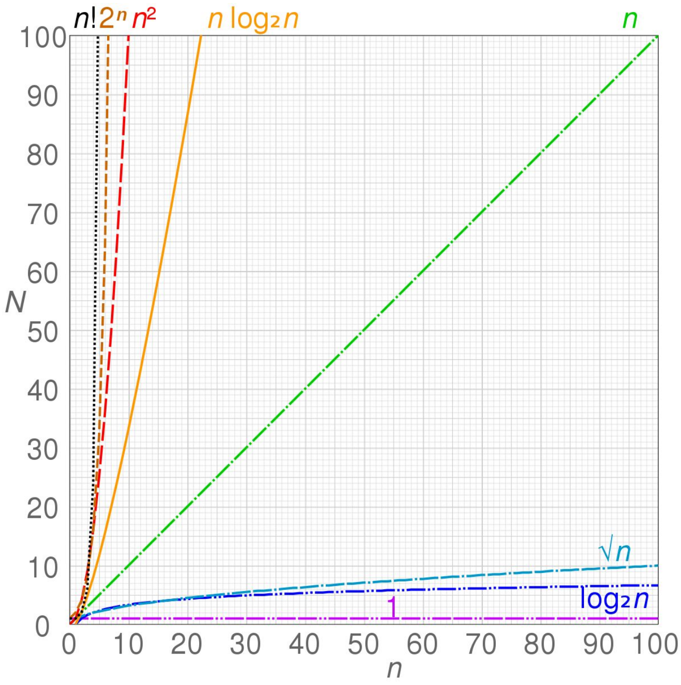
# Orders of Growth

# What Happens As **N** Gets Big?

Meet Time Complexity:

- We know that constants and coefficients don’t matter–so what does?
- Answer: Order of Growth
- What do terms like “constant,” “log,” “linear”, “quadratic” and “cubic” really mean?
    - We can use them to describe how the number of operations scale as N gets big
    - Or, to describe the *order of growth* of the number of operations!
- Ok, so how do we simplify our expressions and choose the one that describes our runtimes?

# A Useful Visual



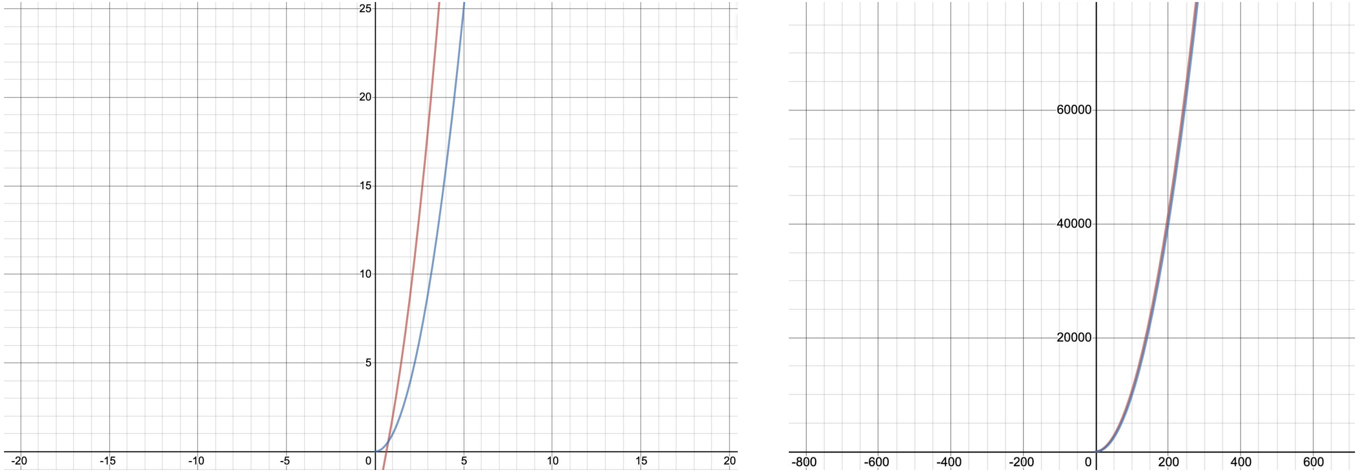
No matter the coefficient, these time complexities will always maintain their relations.

*Note*: you **don't** need to know this name ->

| Name | Example | Big-Θ |
|---|---|---|
| constant | 1000 | Θ(1) |
| logarithmic | 200 log n + 2 | Θ(log n) |
| linear | 150n + log n | Θ(n) |
| linearithmic | 50 n log n + n | Θ(n log n) |
| quadratic | $32n^2$ + n log n | Θ($n^2$) |
| cubic | $21n^3 + n^2$ | Θ($n^3$) |
| exponential | $10 \cdot 2^n + n^3$ | Θ($2^n$) |
| factorial | $2n! + 100 \cdot 2^n$ | Θ(n!) |

# Why Can We Ignore Lower-Order Terms?

- The short answer: they are insignificant as N gets big
- The higher-order term dominates the growth of the function asymptotically, so we can simplify our analysis
- Take for example $T(n) = n^2 + n \log n + n + \log n$. As $T(n) \in \Theta(n^2)$, let's graph T(n) and $f(n) = n^2$
  - T(n) is in red and f(n) is in blue
  - Initially, red looks like it is bigger than blue, but after N is sufficiently, large, there is no discernible difference!



# Symbols and Representation

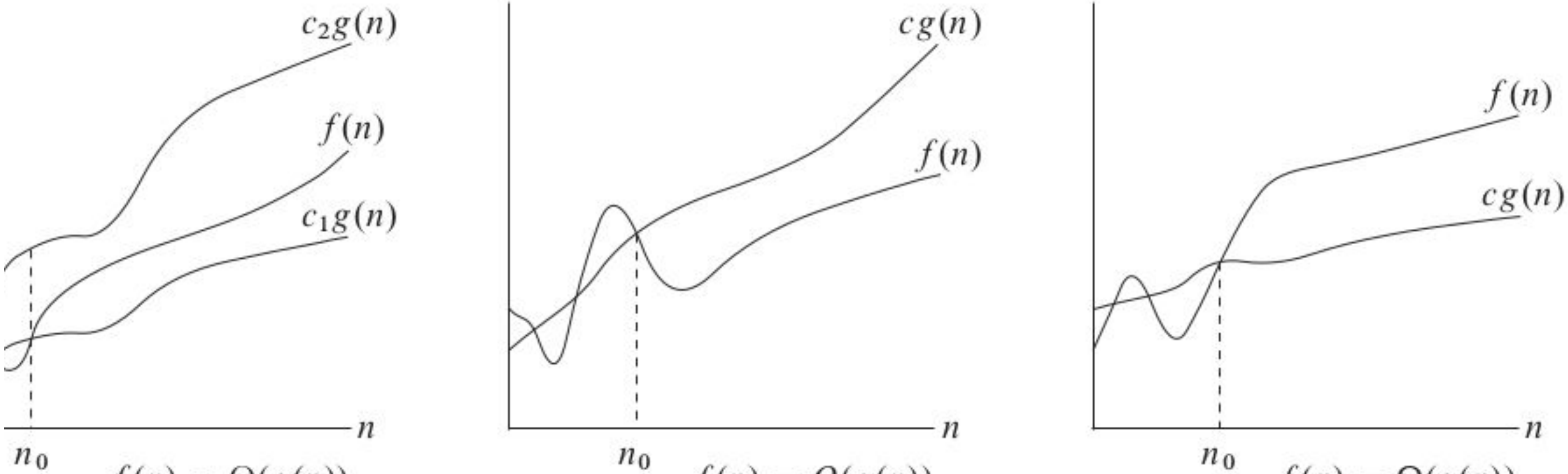
# What Is Big-Theta Notation?

- A very useful tool for describing runtime
- After we find an expression that represents our runtime, we can remove all lower order terms as well as all multiplicative and additive constants, as we saw on the last slide!
- So, something like $2n! + 100 \cdot 2^n + n^3 + 1000000n$ is just $\Theta(n!)$
  - If $T(n) = 2n! + 100 \cdot 2^n + n^3 + 1000000n$, we can say $T(n) \in \Theta(n!)$
- Essentially, "$\Theta$" is used to show the order of growth!

# Now For Big-*O* and Big-Omega

- We saw that Big-Theta perfectly describes the order of growth of a function, almost an analog of “=”, so let’s find some analogues for ≤ and ≥
- Before we talk about Big-O and Big-Omega, let’s give some intuition:
  - Big-**Θ** : f(n) = T(n)
  - Big-*O* : f(n) ≥ T(n)
  - Big-**Ω**: f(n) ≤ T(n)
- Given T(n), we can upper and lower bound its runtime using Big-*O* and Big-**Ω**!
- What does this tell us? Big-**Θ** only exists only when the tightest Big-*O* is equal to the tightest Big-**Ω**!

# Here Is A Useful Visual:



$f(n) = \Theta(g(n))$

$f(n) = O(g(n))$

$f(n) = \Omega(g(n))$

# How To Use These, & Best vs. Worst Case

- Because Big-*O* and Big-**Ω** are upper and lower bounds, they can be very loose and still be “correct”
    - e.g. if T(n) = n log n, it is correct to say T(n) ∈ O(n!) and T(n) ∈ **Ω**(1)
- We’d like to be more descriptive than this, so we should always search for the tightest bounds.
- For some runtime analysis, we may not be able to determine a T(n), which makes finding its Order Of Growth difficult, we can find Big-*O* and Big-**Ω** to show Big-**Θ**!
- **Note:** while it might seem that Big-*O* and Big-**Ω** point towards a good way for representing best and worst case runtime for some function, this is not necessarily true. We can still use Big-**Θ** in some situations.
- On the following slide is a contrived example that illustrates the usefulness of the three!

# Using Notation To Describe A Procedure

```
/**
 * randomCoin() has a 50% chance of returning 0
 */
public void f(int[] arr) {
     int N = arr.length;
     if (N % 2 == 0) {
          if (randomCoin() == 0) {
               thetaN(arr); //this runs in Θ(n) time
          } else {
               thetaN2(arr); //this runs in Θ(n^2) time
          }
     } else {
          thetaN5(arr); //this runs in Θ(n^5) time
     }

}
```

What is the best and worst case runtime?

# Using Notation To Describe A Procedure

```
/**
 * randomCoin() has a 50% chance of returning 0
 */
public void f(int[] arr) {
      int N = arr.length;
      if (N % 2 == 0) {
            if (randomCoin() == 0) {
                  thetaN(arr); //this runs in Θ(n) time
            } else {
                  thetaN2(arr); //this runs in Θ(n^2) time
            }
      } else {
            thetaN5(arr); //this runs in Θ(n^5) time
      }

}
```

Worst case and best case is based on some aspect of the input that is independent of its size, N. In this case, it depends on if N is even or odd

- The Best Case is when n is even and randomCoin() returns 0:
    - We say, in the best case, that the runtime is $\Theta(n)$
- The Worst Case is when n is odd, and notice, we ***can*** use Big-$\Theta$ here.
    - We say, in the worst case, that the runtime $\Theta(n^5)$
- We cannot use $\Theta$ notation when assessing the general runtime of the function

# Examples

# Example 1

```
public void f1(int N) {
    for (int i = 1; i < N; i *= 2) {
        System.out.println(“Hello World.”); //Assume this line runs in Θ(1) time
    }
}
```

What is the Big Θ runtime of this function?

# Example 1

```
public void f1(int N) {
    for (int i = 1; i < N; i *= 2) { // This loop runs log(N) times
        System.out.println("Hello World."); // Assume this line runs in Θ(1) time
    } // i = 1, 2, 4, ..., N = $2^{0}$, $2^{1}$, $2^{2}$, ..., $2^{log(N)}$
} // So, we get Θ(log n) * Θ(1) = Θ(log N)
```

What is the Big Θ runtime of this function? Θ(log N)

# Example 2

```
public void oddOrEven(int N) {
     if (N % 2 == 0) { // Expression returns true if N is even, line runs in Θ(1)
time
           System.out.println(“Input is even!”);
     } else {
          f1(N); // Function from previous example, runs in Θ(log N) time
          System.out.println(“Input is odd!”);
     }
}
```

What is the best & worst case runtime of this function?

# Example 2

```
public void oddOrEven(int N) {
    if (N % 2 == 0) { // Expression returns true if N is even, line runs in Θ(1)
time
          System.out.println("Input is even!"); // Best case, we stop here right
    away
    } else { // The size of N has no bearing on what happens to this if statement
         f1(N); // Function from previous example, runs in Θ(log N) time
         System.out.println("Input is odd!"); // Worst case, we do the above first
    }
}
```

What is the best & worst case runtime of this function? Best: Θ(1), Worst: Θ(log N)

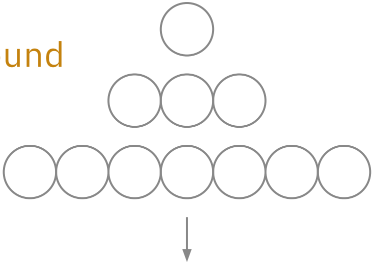
# Example 3

```
public static void spaceJam(int N) { // This is a hard problem
   if (N <= 1) {
       return;
   }
   for (int i = 0; i < N; i += 1) {
       spacejam(N - 1);
   }
}
```

What is the best & worst case runtime of this function?

# Example 3

```
public static void spaceJam(int N) { // This is a hard problem
    if (N <= 1) { // This is dependent on N, so best case == worst case
        return;
    }
    for (int i = 0; i < N; i += 1) { // Each iteration of this function calls N more
        spacejam(N - 1);
    }
} // Summing is going to be hard, so lets upper bound
  // (N!/i!) = O(N!) --> O(N!) * N --> O(N * N!)
```



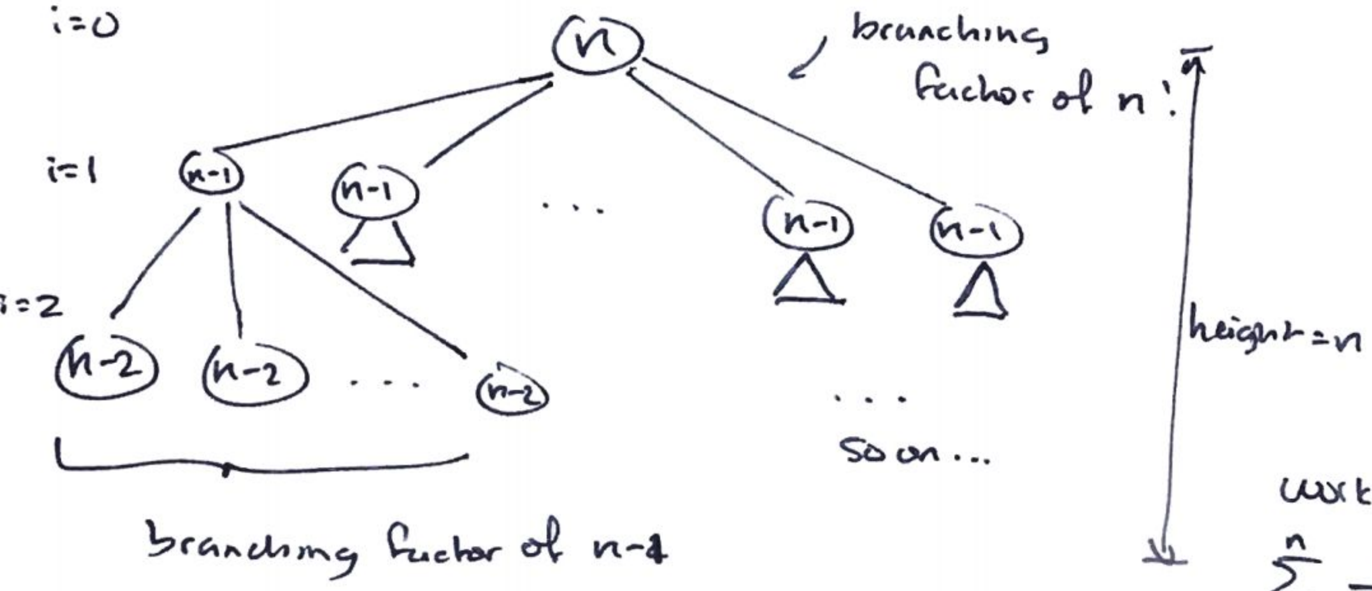
1

N

N * (N-1)

N * (N-1) * ... * (N-i)

What is the best & worst case runtime of this function? Best: O(N*N!), Worst: O(N*N!)



$$\text{height} = n$$

$$\frac{\text{nodes}}{\text{layer}} = \frac{n!}{(n-i)!}$$

$$\frac{\text{work}}{\text{node}} = O(n)$$

work ...

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

$$\leq \sum_{i=0}^{n} n! = n \cdot n!$$

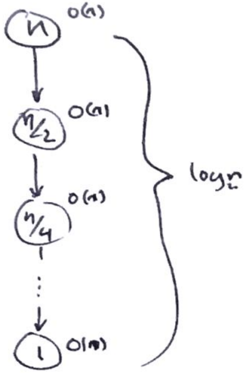
$$\in O(n \cdot n!)$$

## EXTRA PRACTICE!

# Example 4

```
public void andSlam(int N) {
   if (N > 0) { // Again, depends on N so best case == worst case
       for (int i = 0; i < N; i += 1) {
           System.out.println("datboi.jpg"); // Assume this line is constant time
       }
       andslam(N / 2); // N = n, n/2, n/4, n/8, ... 1
   }                  // Zeno’s Paradox! The above sums to 2N
}                     // Also, sum(N) = N * (1 - 0.5^(log N))/(1 - 0.5) = N * 1
```

What is the best & worst case runtime of this function? Best: Θ(N), Worst: Θ(N)



[1] Height of tree

→ howmany times can you divide n by 2 until you get n=1. Let h be height.

$$\frac{n}{2^h} = 1 \to n = 2^h \to h = \log_2 n$$

[2] Branching Factor

→ Note each time andslam is called, it makes 1 recursive call on $n/2$.

→ # nodes per layer = 1.

[3] Amount of work each node does.

→ linear relative to input size. so $O(n)$.

Now running time ~~can be~~ of entire recursive procedure can be calcula by summing over entire recursive tree.

$$\text{running time} = \overset{\text{sum over}}{\#\text{ layers}} \cdot \left(\frac{\#\text{ nodes}}{\#\text{ layer}}\right) \cdot \left(\frac{\text{amount work}}{\$1\text{ node}}\right)$$

$$= \underbrace{\sum_{i=0}^{\log n}}_{\text{layers}} \overset{\#\text{nodes / layer}}{(1)} \cdot \underbrace{\left(\frac{n}{2^i}\right)}_{\text{work / node}}$$

$$= \sum_{i=0}^{\log n} \frac{n}{2^i} = n \sum_{i=0}^{\log n} \frac{1}{2^i} \leq 2n \in \Theta(n)$$