# Lecture 4: Intro to Runtime Analysis

CSE 373: Data Structures and Algorithms

## Warm Up

**Slido Event #1190355**
https://app.sli.do/event/6kJCJNUzLZFsfdCCUa7zXZ

**Discuss with your neighbors:** For the following scenario select the appropriate ADT and implementation and explain why they are optimal for this situation.

**Situation:** You are writing a program to schedule jobs sent to a laser printer. The laser printer should process these jobs in the order in which the requests were received. There are busy and slow times for requests that can have large differences in the volume of jobs sent to the printer. Which ADT and what implementation would you use to store the jobs sent to the printer?

ADT options:
- List
- Stack
- Queue

Implementation options:
- array
- linked nodes

Kasey's Answer

**LinkedQueue**

This will maintain the original order of the print jobs, but allow you to easily cancel jobs stuck in the middle of the queue. This will also keep the space used by the queue at the minimum required levels despite the fact the queue will have very different lengths at different times.

## Questions?

## Announcements

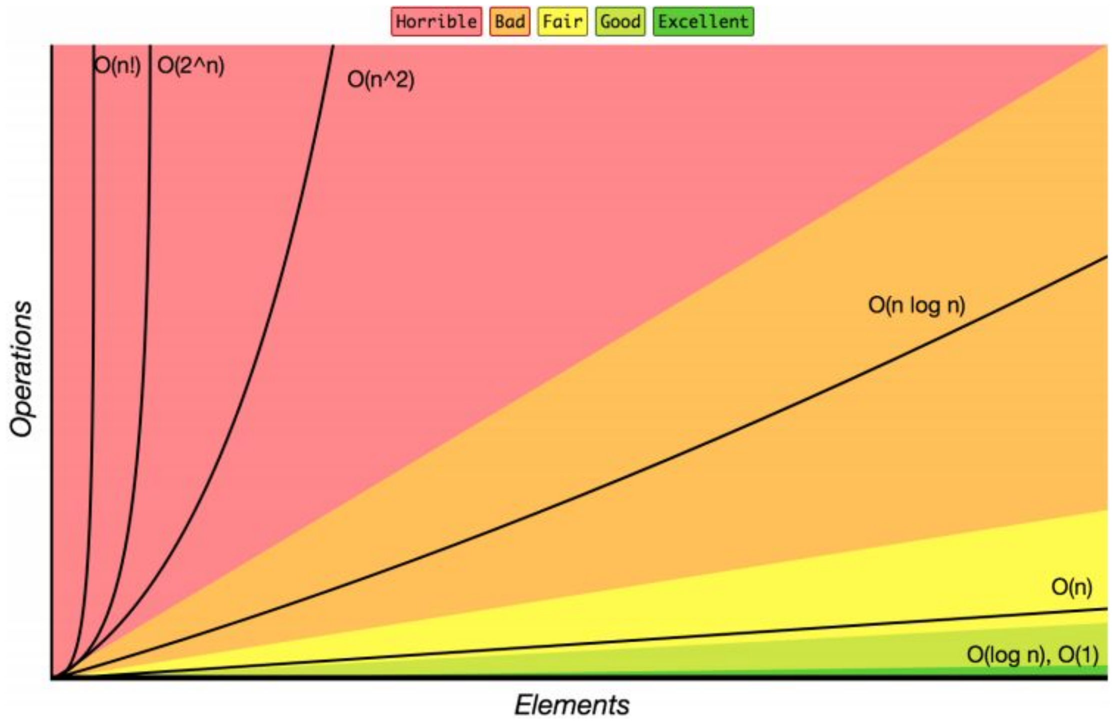
- Office hours start this week
- HW 0 – 143 Review Project
  - Group submissions
  - Due Wednesday April 6th at 11:59pm
- Exercise 0 Releasing today
  - Individual submissions
  - Due next Monday at 11:59pm

## Big O

# *Review:* Complexity Class

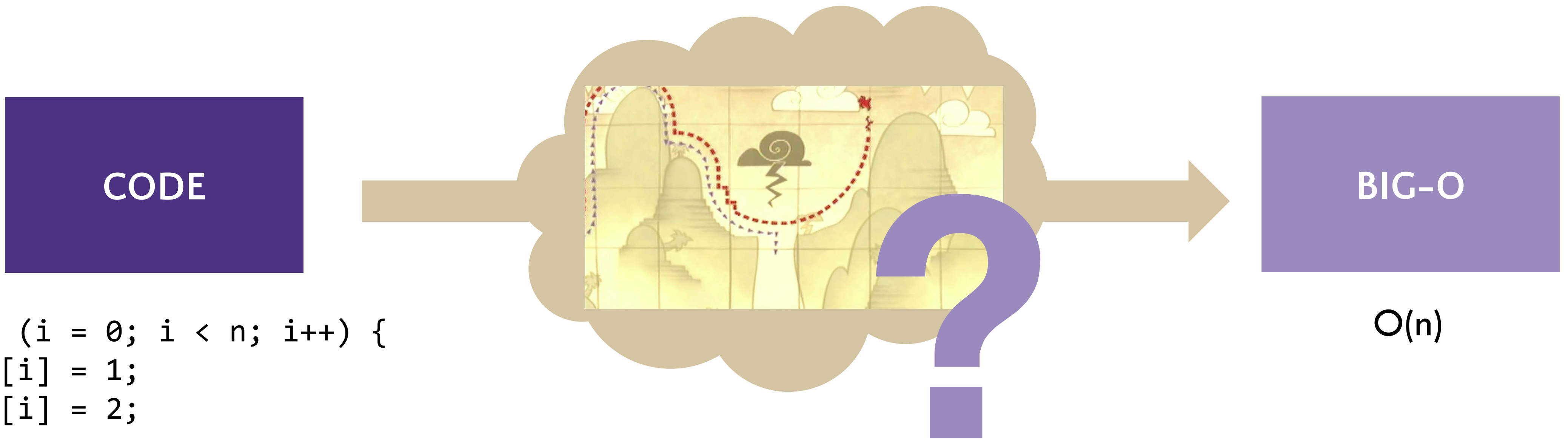
**complexity class:** A category of algorithm efficiency based on the algorithm's relationship to the input size N.

| Complexity Class | Big-O | Runtime if you double N | Example Algorithm |
|---|---|---|---|
| constant | O(1) | unchanged | Accessing an index of an array |
| logarithmic | $O(\log_2 N)$ | increases slightly | Binary search |
| linear | O(N) | doubles | Looping over an array |
| log-linear | $O(N \log_2 N)$ | slightly more than doubles | Merge sort algorithm |
| quadratic | $O(N^2)$ | quadruples | Nested loops! |
| … | … | … | … |
| exponential | $O(2^N)$ | multiplies drastically | Fibonacci with recursion |



bigocheatsheet.com

# Code to Big-O



```
for (i = 0; i < n; i++) {
  a[i] = 1;
  b[i] = 2;
}
```

O(n)

123/143 general patterns:
- O(1) constant is no loops
- O(n) is one loop
- O($n^2$) is nested loops

373:
We need a way to definitively determine Big O for all code

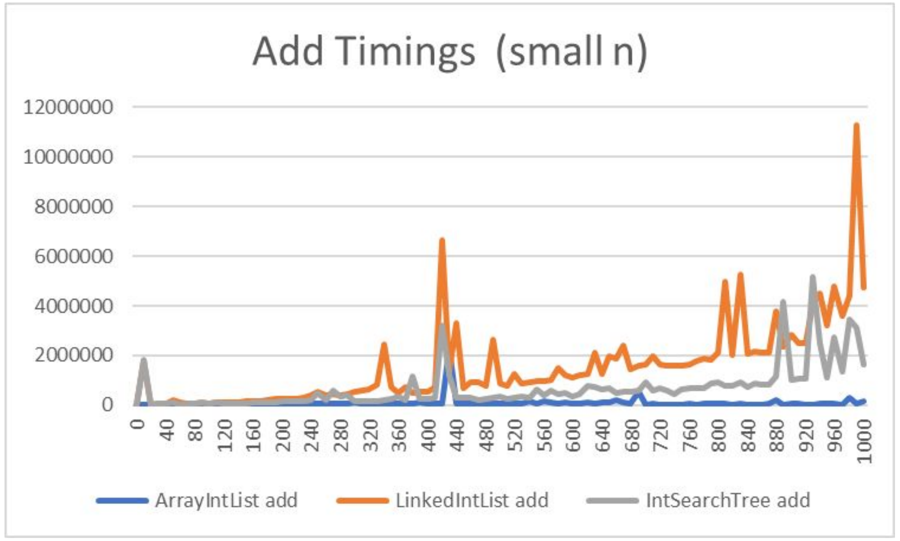
## Motivation: Why Big-O?

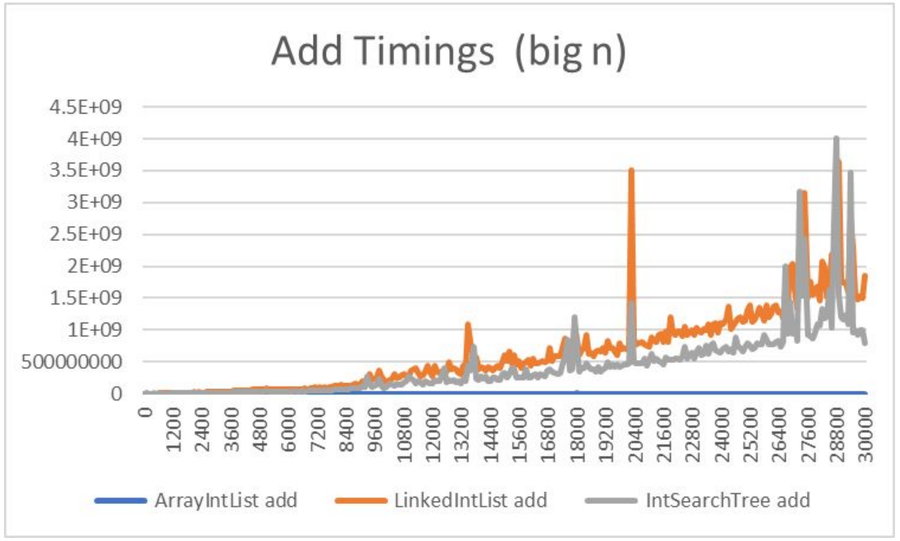
Goals of Big-O/Algorithmic Analysis:

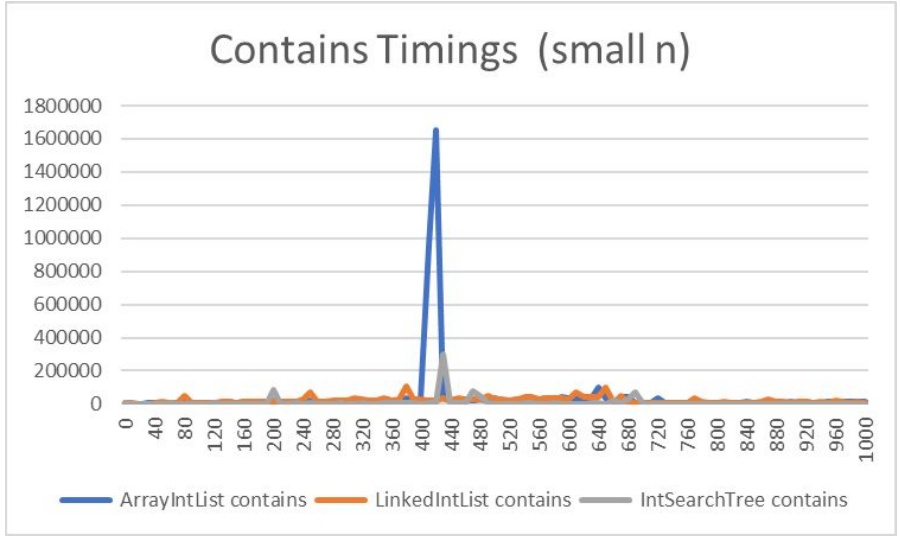
1. **Simple**
   We don't care about tiny differences in implementation, want the big picture result

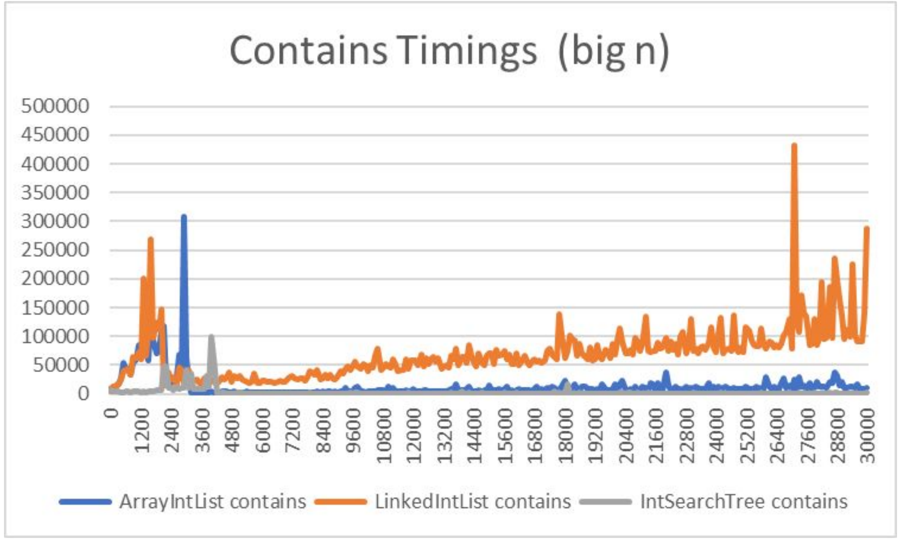
2. **Decisive**
   Produce a clear comparison indicating which code takes "longer"

# Why not time code?







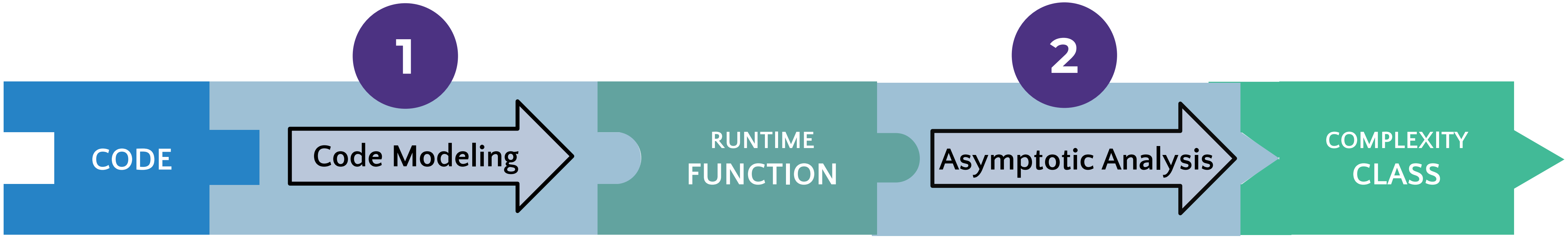


Actual time to completion can vary depending on hardware, state of computer and many other factors.

These graphs are of times to run add and contains on structures of various sizes of N and you can see inconsistencies in individual runs which can make determining the overall relationship between the code and runtime less clear.

You can find the code to run these tests on your own machine on the course website!

# Meet Algorithmic Analysis



```
for (i = 0; i < n; i++) {
  a[i] = 1;
  b[i] = 2;
}
```

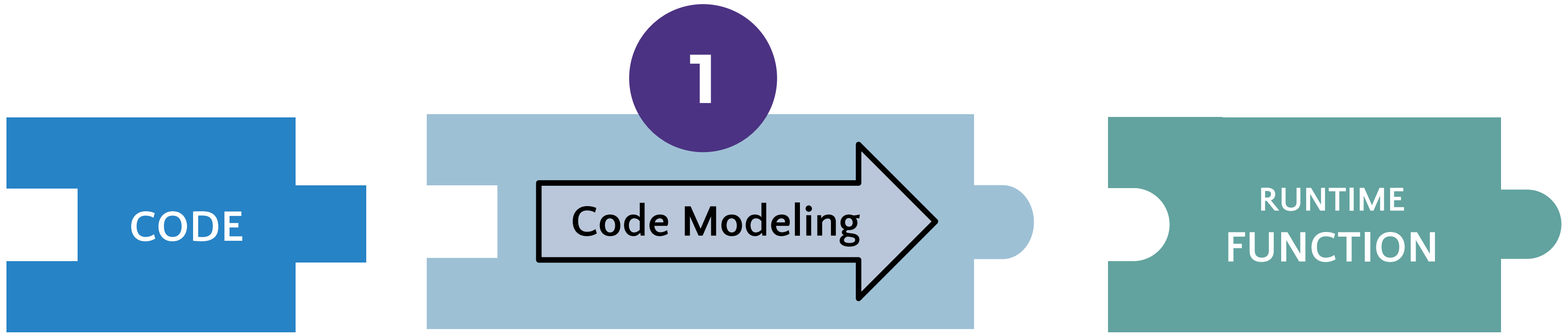
$f(n) = 2n$

$O(n)$

**Algorithmic Analysis**: The overall process of characterizing code with a complexity class, consisting of:

- **Code Modeling**: Code  Function describing code's runtime
- **Asymptotic Analysis**: Function  Complexity class describing asymptotic behavior

# Code Modeling



**Code Modeling** – the process of mathematically representing how many operations a piece of code will run in relation to the input size n.

- Convert from code to a function representing its runtime

Example:

```
for (i = 0; i < n; i++) {
  a[i] = 1;
  b[i] = 2;
}
```

- One array element update = “1” runtime count
- Loop that runs “n” times = “n” runtime count
- Loop N times(2 runtime counts inside loop)

= 2N

## What Counts?

We don’t know exact runtime of every operation, but for now let’s try simplifying assumption: all basic operations take the same time

- Basics count as “1”:
  - +, -, /, *, %, ==
  - Assignment
  - Returning
  - Variable/array access
- Function Calls
  - Total runtime in body
  - Remember: new calls a function (constructor)
- Conditionals
  - Test + time for the followed branch
    - Learn how to reason about branch later
- Loops
  - Number of iterations * total runtime in condition and body
  - For loop header operations don’t count, but while loop headers do

# Code Modeling Example 1

```
public void method1(int n) {
    int sum = 0;  +1
    int i = 0;  +1
    while (i < n) {  +1
        sum = sum + (i * 3);  +3
        i = i + 1;  +2
    }
    return sum;  +1
}
```

Loop runs n times

+6 *n

$f(n) = 6n + 3$

Practice time!

# Code Modeling Example 2

```
public void method2(int n) {
    int sum = 0;  +1
    int i = 0;  +1
    while (i < n) {  +1
        int j = 0;  +1
        while (j < n) {  +1
            if (j % 2 == 0) {  +2
                // do nothing
            }
            sum = sum + (i * 3) + j;  +4
            j = j + 1;  +2
        }
        i = i + 1;  +2
    } return sum;  +1
}
```

This inner loop runs n times

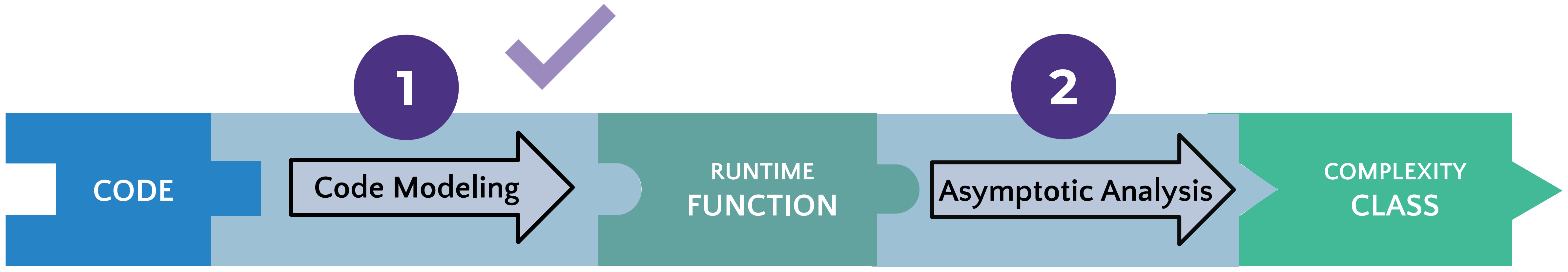
+9 *n

This outer loop runs n times

9n + 4 *n

$$f(n) = (9n+4)n + 3$$

# Where are we?



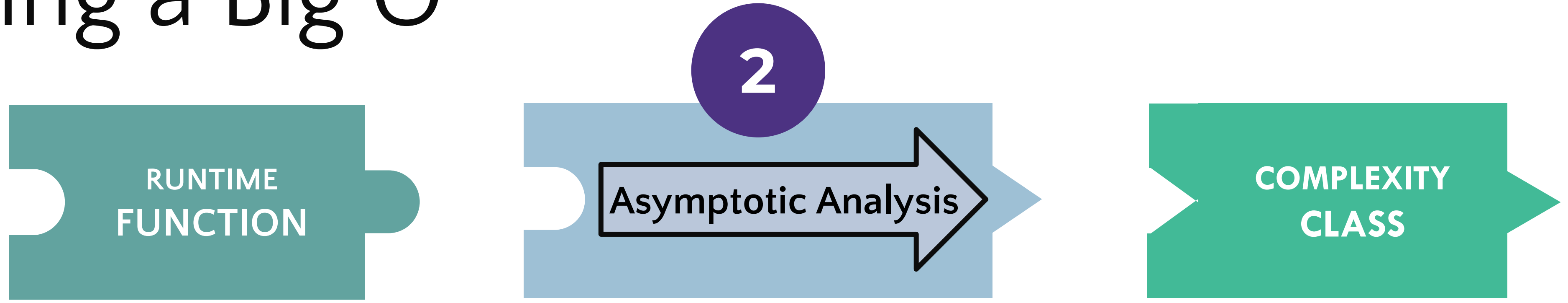
```
for (i = 0; i < n; i++) {
  a[i] = 1;
  b[i] = 2;
}
```

$f(n) = 2n$

$O(n)$

- We just turned a piece of code into a function!
    - We'll look at better alternatives for code modeling later
- Now to focus on step 2, asymptotic analysis

## Finding a Big O



We have an expression for f(n). How do we get the O() that we've been talking about?

1. Find the "dominating term" and delete all others
   a. The "dominating term" is the one that is the largest as n gets bigger. in this class, often the largest power of n.
2. Remove and constant factors

$f(n) = (9n+3)n + 3$

$= 9n^2 + 3n + 3$

$\approx 9n^2$

$\approx n^2$

$f(n)$ is $O(n^2)$

Practice time!

# Finding a Big O

We have an expression for f(n). How do we get the O() that we've been talking about?

1. Find the "dominating term" and delete all others
   a. The "dominating term" is the one that is the largest as n gets bigger. In this class, often the largest power of n.
2. Remove and constant factors

$f(n) = 6n + 3$

$= 6n + 3$

$\approx 6n$

$\approx n$

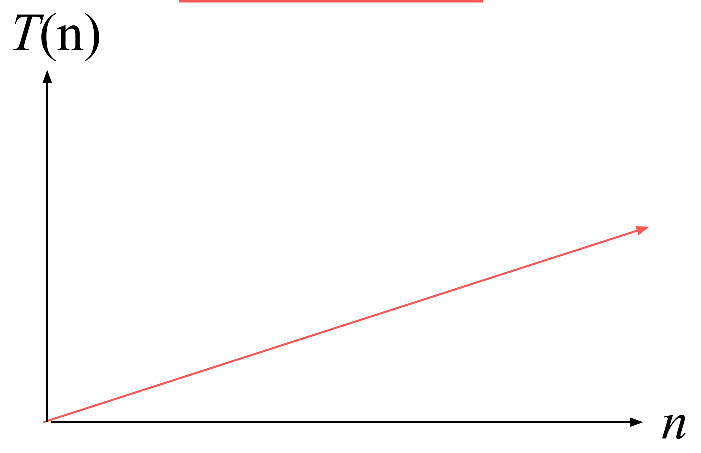
f(n) is O(n)

# What is a "dominating term"?

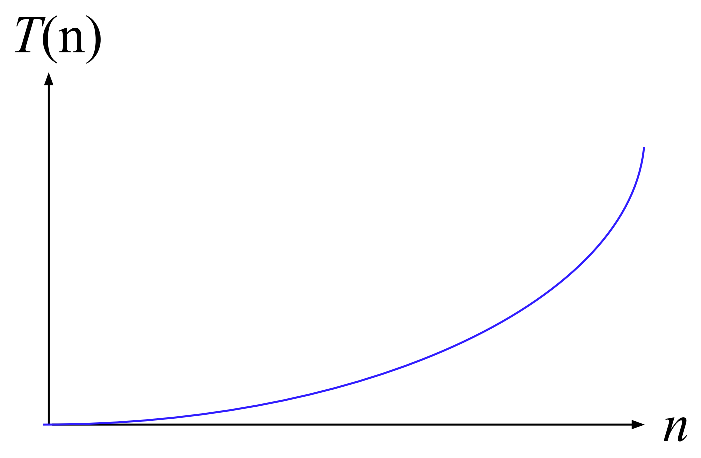
**Asymptotic Analysis**: Analysis of function behavior as its input approaches **infinity**

Dominating terms have the largest **influence** on the behavior of f(n) as they are the largest, and "dominate" the smaller terms

$f(n) = \mathbf{n} + 10$



$g(n) = \mathbf{n^\wedge 2} + n + 10$



**Practice time!**

## What is a "dominating term"?

What is the dominating term?

| | | |
|---|---|---|
| 1. | $n^2 + n$ | $n^2$ |
| 2. | $n + 1000$ | $n$ |
| 3. | $n^{100} + n^{50} + n^2 + 5$ | $n^{100}$ |
| 4. | $n^2 + 2^n$ | $2^n$ |
| 5. | $3^n + 4^n$ | $4^n$ |

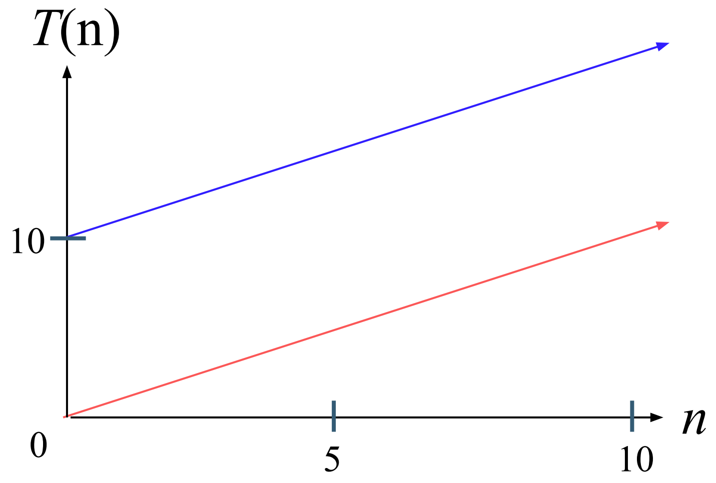
hint: ask yourself "which term is going to be the largest the bigger and bigger n is?"

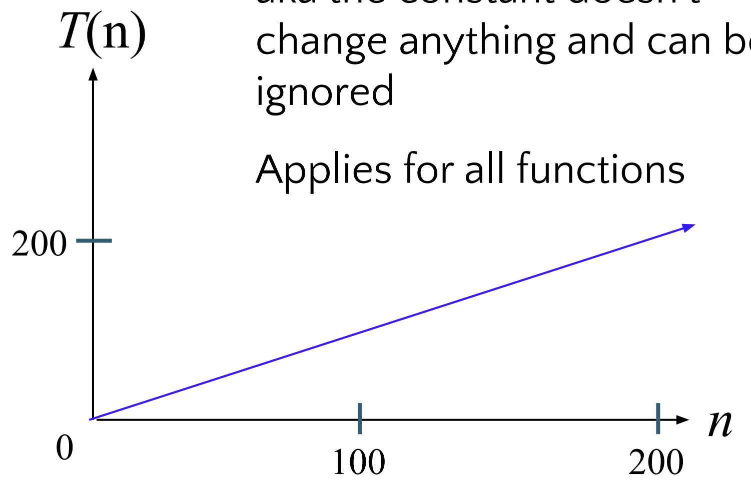
# Can we really throw away all that info?

**Asymptotic Analysis**: Analysis of function behavior as its input approaches **infinity**

Let's look at linear functions and think about the effect of constants

$f(n) = n$ $g(n) = n + 10$





At the scale of infinity, f(n) and g(n) have identical behavior, aka the constant doesn't change anything and can be ignored

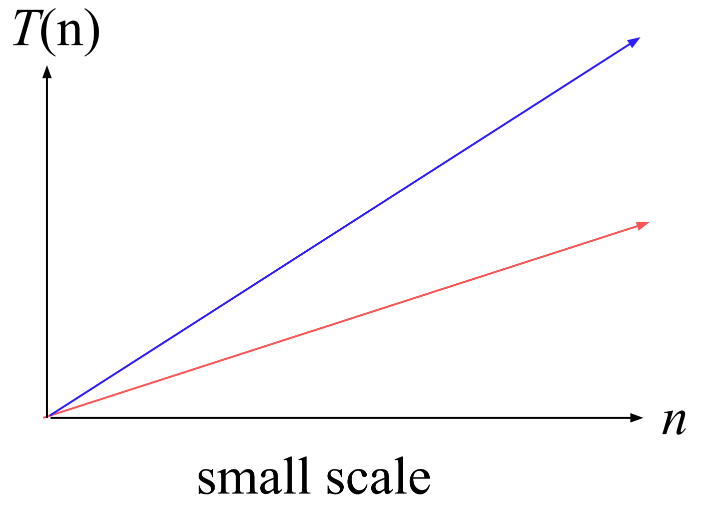
Applies for all functions

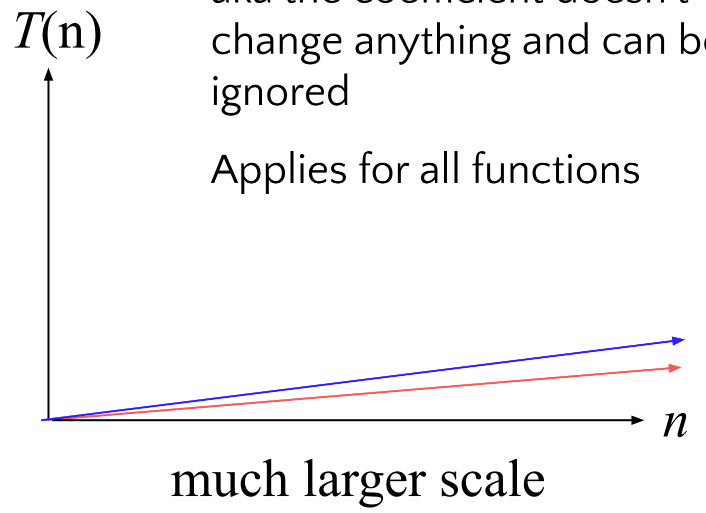
# Can we really throw away all that info?

**Asymptotic Analysis**: Analysis of function behavior as its input approaches **infinity**

Let's look at linear functions and think about the effect of coefficients

$f(n) = n$ $\qquad$ $g(n) = 2n$





At the scale of infinity, f(n) and g(n) have identical behavior, aka the coefficient doesn't change anything and can be ignored

Applies for all functions

## Can we really throw away all that info?

Big-Oh is like the “significant digits” of computer science

**Asymptotic Analysis**: Analysis of function behavior as its input approaches infinity

- We only care about what happens when n approaches infinity
- For small inputs, doesn’t really matter: all code is “fast enough”
- Since we’re dealing with infinity, constants and lower-order terms don’t meaningfully add to the final result. The highest-order term is what drives growth!

Remember our goals:

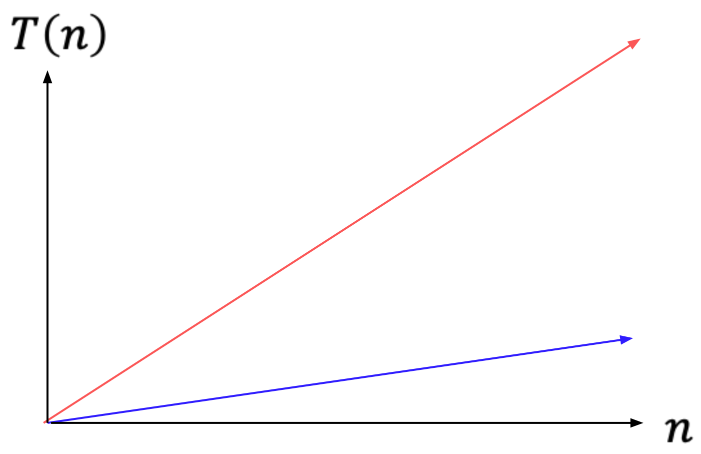
1. **Simple**
   We don’t care about tiny differences in implementation, want the big picture result

2. **Decisive**
   Produce a clear comparison indicating which code takes “longer”

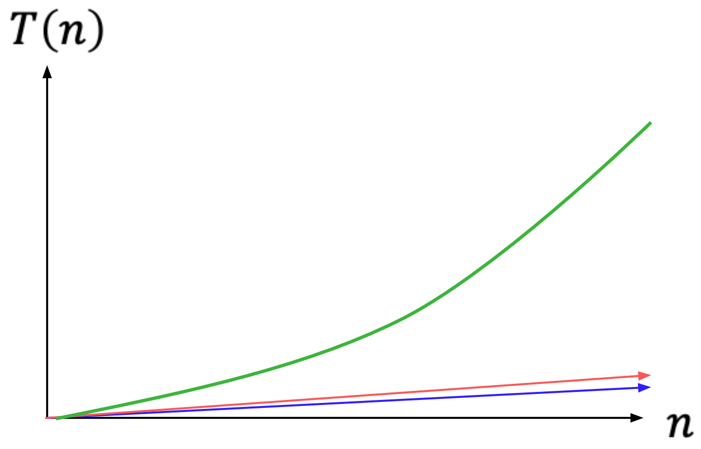
# Function growth

Imagine you have three possible algorithms to choose between.
Each has already been reduced to its mathematical model

$f(n) = n$ $\quad g(n) = 4n$ $\quad h(n) = n^2$

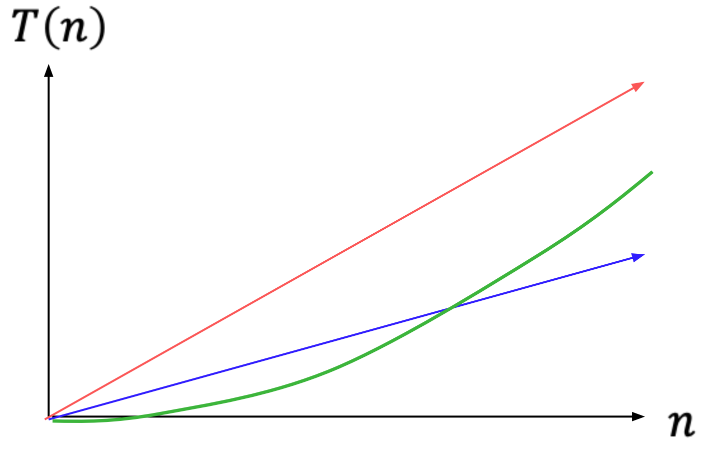


The growth rate for f(n) and g(n) looks very different for small numbers of input



…but since both are linear eventually look similar at large input sizes

whereas h(n) has a distinctly different growth rate



But for very small input values h(n) actually has a slower growth rate than either f(n) or g(n)

# *Definition:* Big-O

We wanted to find an upper bound on our algorithm's running time, but:

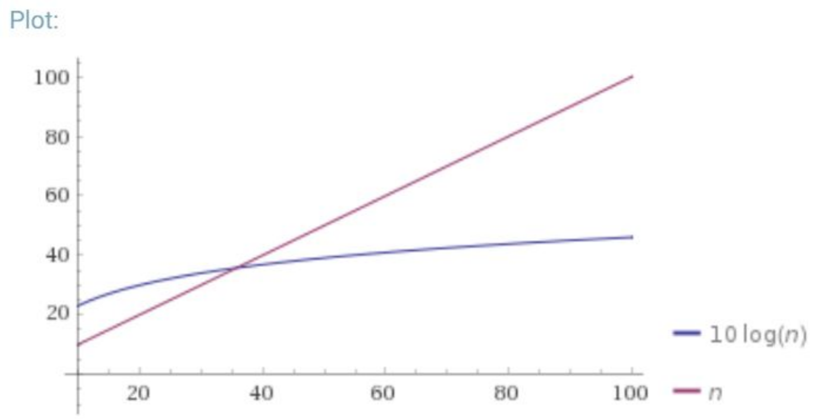
- We don't want to care about constant factors
- We only care about what happens as *n* gets larger

**Big-O**

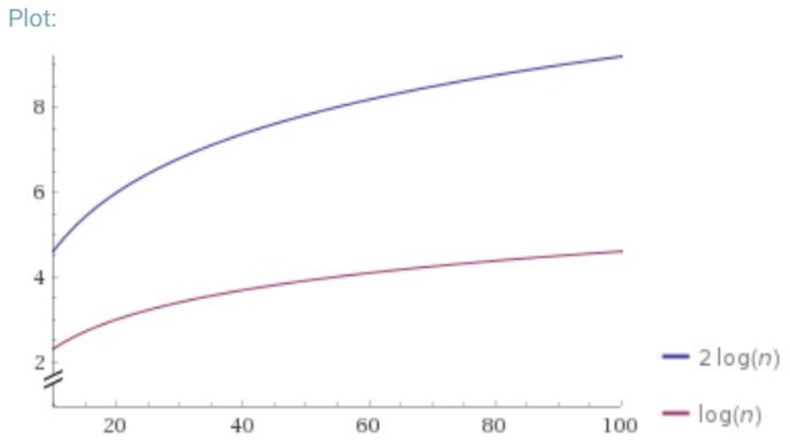
$f(n)$ is $O(g(n))$ if there exist positive constants $c$, $n_0$, such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$

We also say that $g(n)$ "dominates" $f(n)$

Why $n_0$?

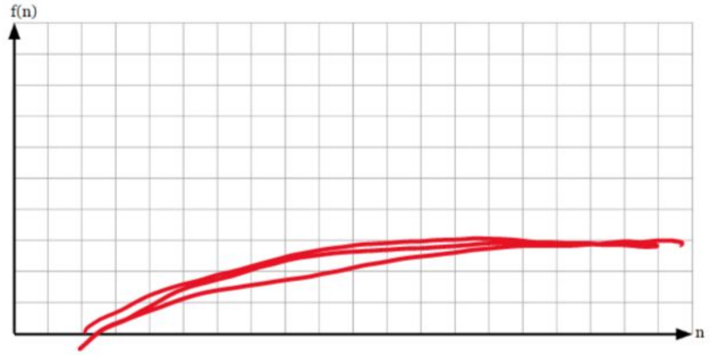


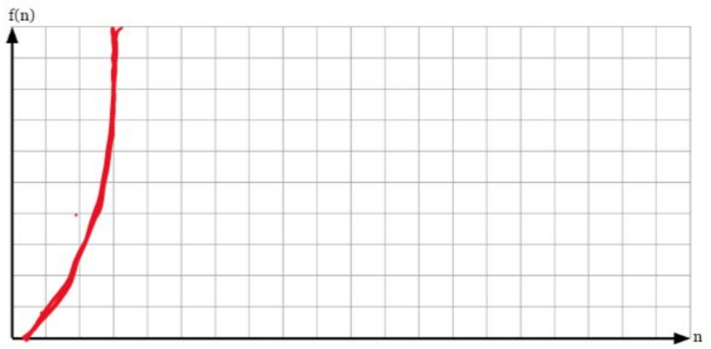
Why $c$?



# EX0: What's the Big O?

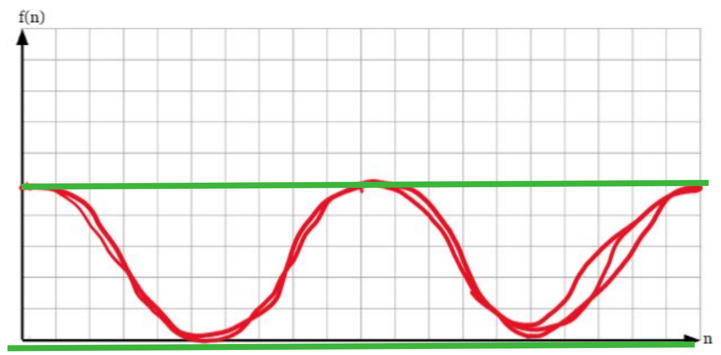
$O(\log n)$
This graph appears to follow the pattern of logarithmic growth

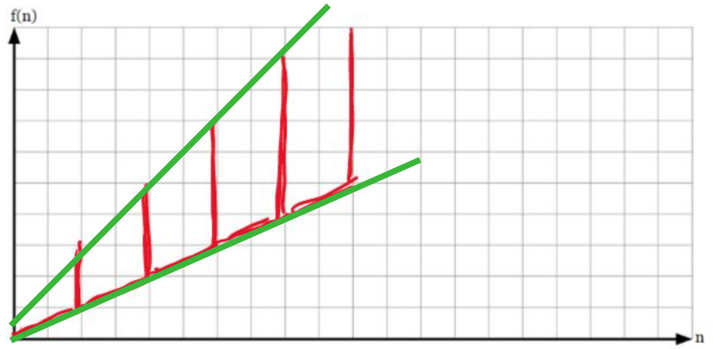




$O(n^2)$
This graph appears to follow the pattern of quadratic growth

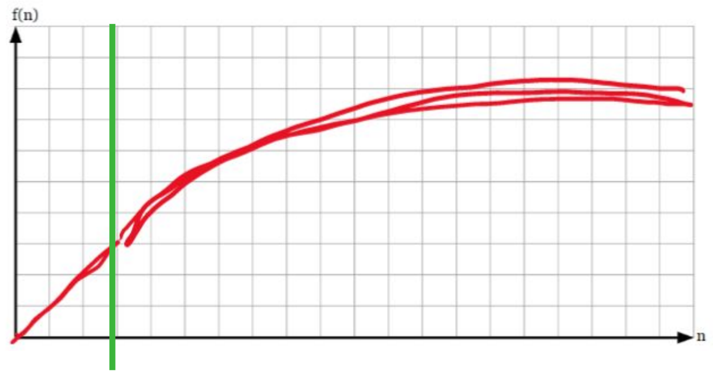
$O(1)$
Though this graph oscillates, the upper and lower bounds are constant

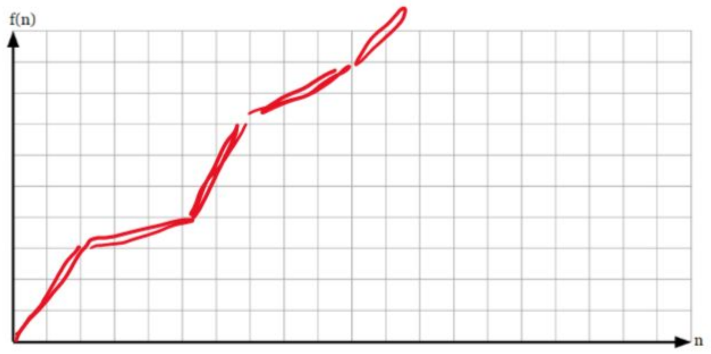




$O(n)$
Though this graph has different upper and lower bounds, they are both linear

$O(\log n)$
Though this graph has two different growth rates, we only count the one that tends to infinity





## Questions?

## That’s all!