CS 201 w21

Time and Space Complexity

Learning Objectives

After this video and practice problems, you should be able to...

1. Describe the difference between Code Modeling and Asymptotic Analysis (both components of Algorithmic Analysis)

2. Model a (simple) piece of code with a function describing its runtime

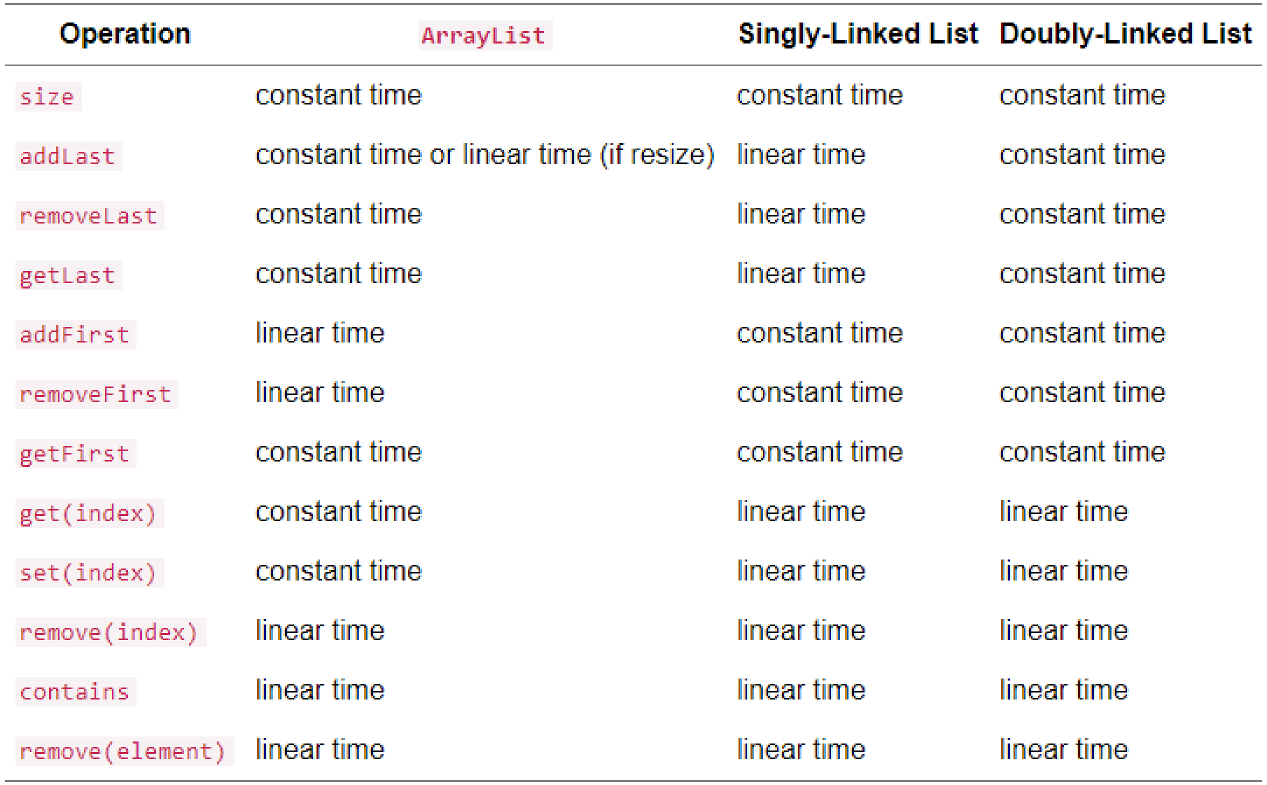
3. Explain why we can throw away constants when we compute Big-O bounds.

    - From a practical perspective and from the “definition” perspective.

Lecture Outline

- **Overview: Algorithmic Analysis** ◀
- Code Modeling
- Asymptotic Analysis
- Big-O Definition

Time and space analysis helps us differentiate between data structures



Operation	`ArrayList`	Singly-Linked List	Doubly-Linked List
`size`	constant time	constant time	constant time
`addLast`	constant time or linear time (if resize)	linear time	constant time
`removeLast`	constant time	linear time	constant time
`getLast`	constant time	linear time	constant time
`addFirst`	linear time	constant time	constant time
`removeFirst`	linear time	constant time	constant time
`getFirst`	constant time	constant time	constant time
`get(index)`	constant time	linear time	linear time
`set(index)`	constant time	linear time	linear time
`remove(index)`	linear time	linear time	linear time
`contains`	linear time	linear time	linear time
`remove(element)`	linear time	linear time	linear time

ArrayList

- Zero “overhead” per element (internal array just stores the data)
- But extra capacity is “wasted”

Linked List

- One or two extra references per element (next and previous in each node)
- But exactly as many nodes as elements (no extra capacity)

If ArrayList is managed precisely will be more space efficient, but both structures take *linear space*

- “Just change first node” vs. “Change every element” is clearly different
- To *evaluate* data structures, need to understand impact of design decisions

We need a tool to analyze code that is

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

B **Mathematically Rigorous**
Use mathematical functions as a precise, flexible basis

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

Overview: Algorithmic Analysis

1

2

CODE

Code Modeling

RUNTIME FUNCTION

Asymptotic Analysis

COMPLEXITY CLASS

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

$f(n) = 2n$

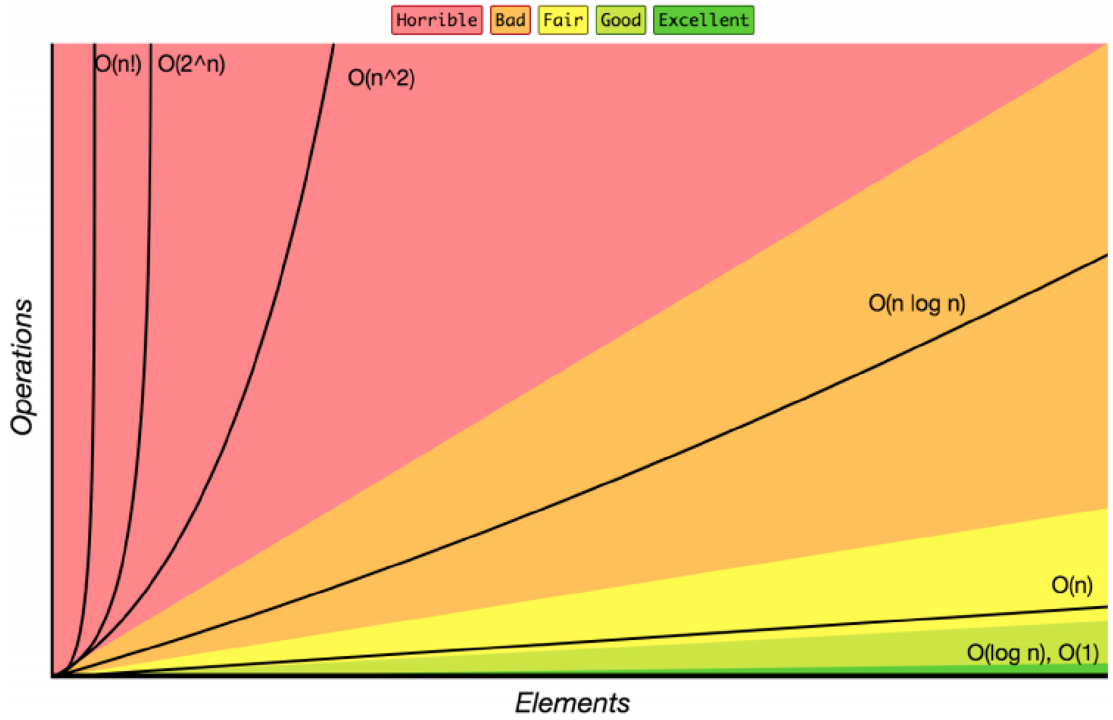
$O(n)$

- **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

What is a *Complexity Class*

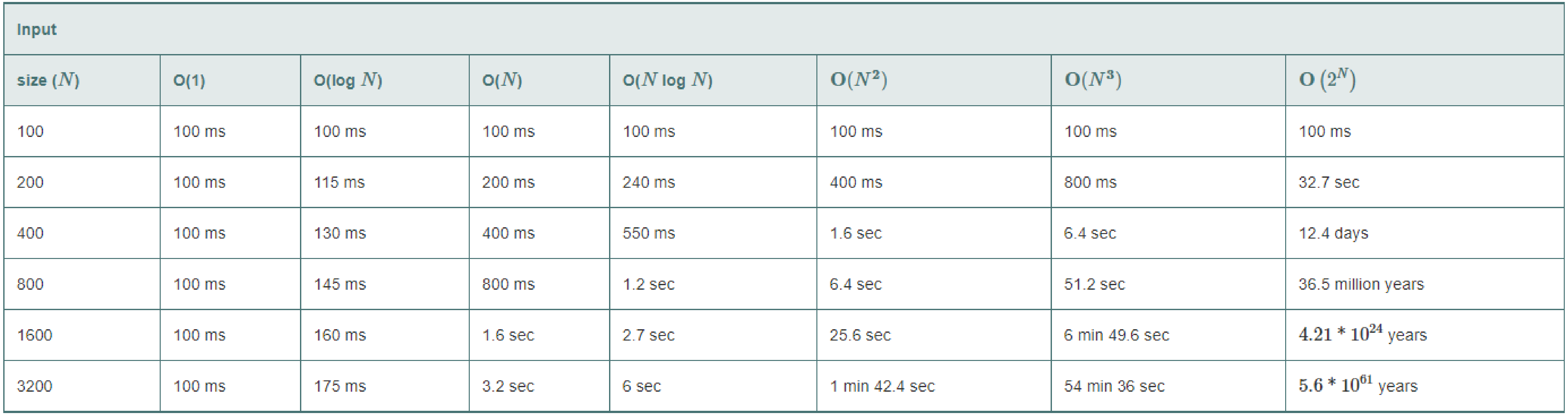
- **Complexity Class**: a category of algorithm efficiency based on the algorithm’s relationship to the input size N

Complexity Class	Big-O	Runtime if you double N
constant	**$O(1)$**	**unchanged**
logarithmic	$O(\log_2 N)$	increases slightly
linear	**$O(N)$**	**doubles**
log-linear	$O(N \log_2 N)$	slightly more than doubles
quadratic	**$O(N^2)$**	**quadruples**
…	…	…
exponential	$O(2^N)$	multiplies drastically



Does Complexity Really Matter?

Yes! The following table presents several hypothetical algorithm runtimes as an input size N grows, assuming that each algorithm required 100ms to process 100 elements. Notice that even if they all start at the same runtime for a small input size, the ones in higher complexity classes become so slow as to be impractical.

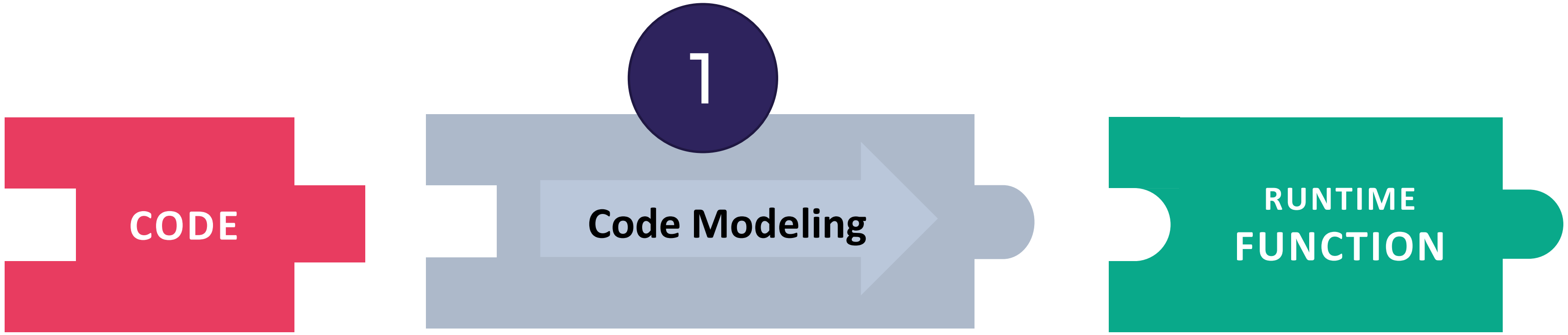


Input							
size (N)	O(1)	O(log N)	O(N)	O(N log N)	$O(N^2)$	$O(N^3)$	$O(2^N)$
100	100 ms	100 ms	100 ms	100 ms	100 ms	100 ms	100 ms
200	100 ms	115 ms	200 ms	240 ms	400 ms	800 ms	32.7 sec
400	100 ms	130 ms	400 ms	550 ms	1.6 sec	6.4 sec	12.4 days
800	100 ms	145 ms	800 ms	1.2 sec	6.4 sec	51.2 sec	36.5 million years
1600	100 ms	160 ms	1.6 sec	2.7 sec	25.6 sec	6 min 49.6 sec	$4.21 * 10^{24}$ years
3200	100 ms	175 ms	3.2 sec	6 sec	1 min 42.4 sec	54 min 36 sec	$5.6 * 10^{61}$ years

Lecture Outline

- Overview: Algorithmic Analysis
- **Code Modeling** ◀
- Asymptotic Analysis
- Big-O Definition

Code Modeling



- **Code Modeling** – the process of mathematically representing how many operations a piece of code will perform in relation to the input size n.
  - Convert from code to a function representing its runtime

What is an operation?

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

- Basics:
  - +, -, /, *, %, ==
  - Assignment
  - Returning
  - Variable/array access

- Function Calls
  - Total time from the operations in the code for that function
- Conditionals
  - Test + time for the followed branch
- Loops
  - Number of iterations * total time for the condition and code inside the loop

Code Modeling Example I

```
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
}
```

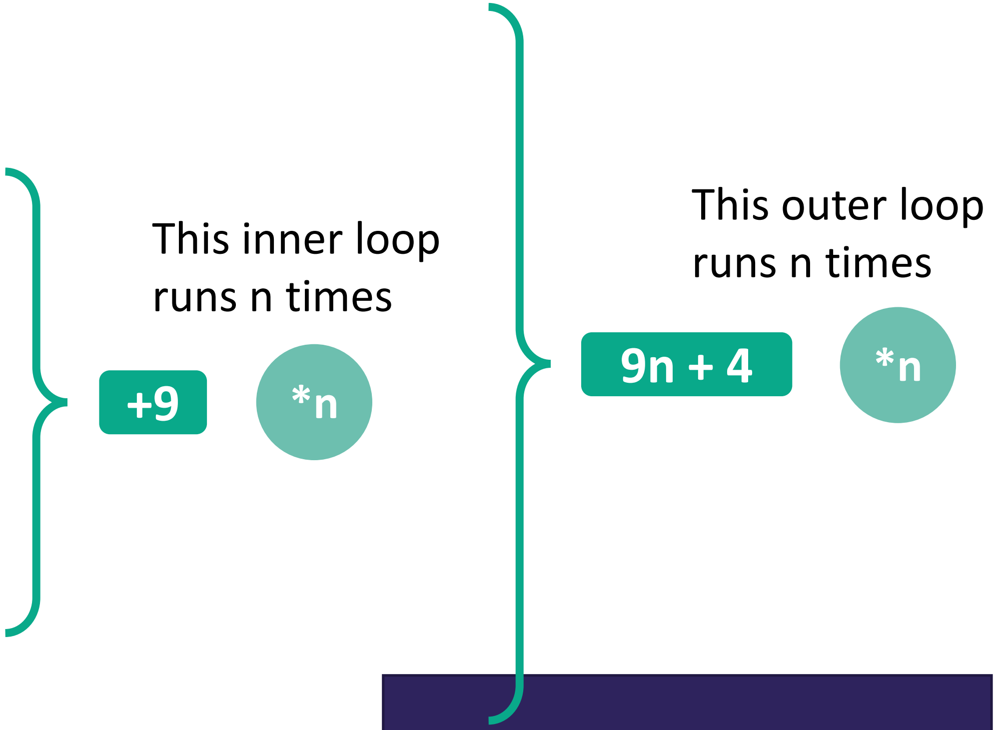
Loop runs n times

+6 *n

f(n) = 6n + 3

Code Modeling Example II

```
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

$9n + 4$

$*n$

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

Lecture Outline

- Overview: Algorithmic Analysis
- Code Modeling
- **Asymptotic Analysis**
- Big-O Definition

Where are we?

CODE

1 Code Modeling ✓

RUNTIME FUNCTION

2 Asymptotic Analysis

COMPLEXITY CLASS

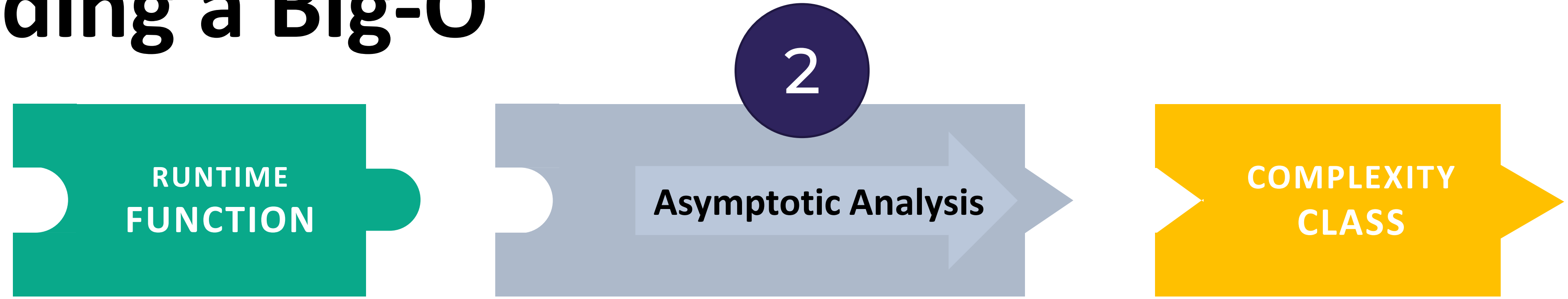
```
for (i = 0; i < n; i++) {
  a[i] = 1;
  b[i] = 2;
}
```

$f(n) = 2n$

$O(n)$

- We just turned a piece of code into a function!
- Now to focus on step 2, asymptotic analysis

Finding a Big-O



RUNTIME FUNCTION

2

Asymptotic Analysis

COMPLEXITY CLASS

- 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.
    - The "dominating" term is the one that is largest as n gets bigger. In this class, often the largest power of n.
2. Remove any constant factors.

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

$$= 9n^2 + 4n + 3$$

$$\approx 9n^2$$

$$\approx n^2$$

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

Is it okay to throw away all that info?

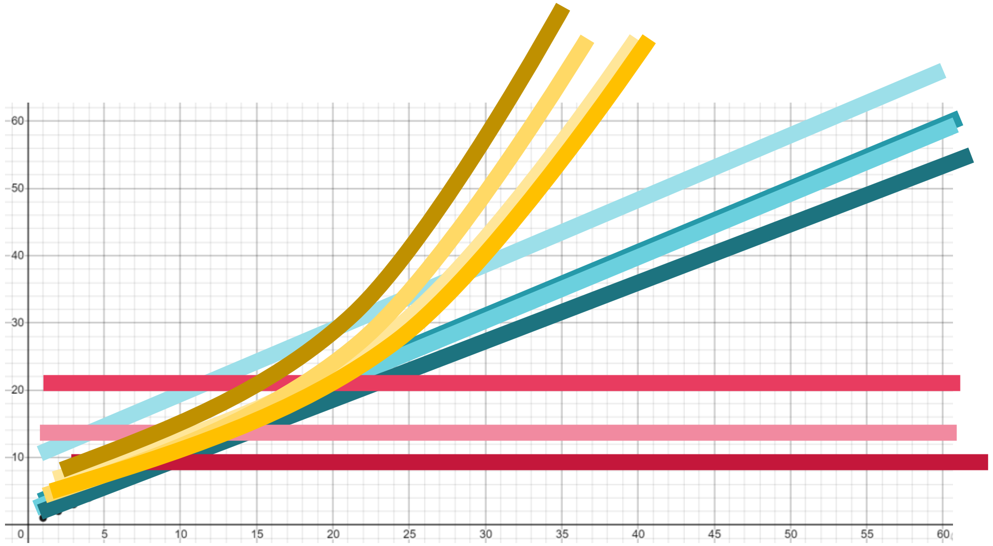
- **Asymptotic Analysis**: Analysis of function behavior as its input approaches infinity
  - We only care about what happens when n approaches infinity
  - For small inputs, doesn't really matter: all code is "fast enough"
  - Since we're dealing with infinity, constants and lower-order terms don't meaningfully add to the final result. The highest-order term is what drives growth!

Remember our goals:

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

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

No seriously, this is really okay?



- There are tiny variations in these functions (2n vs. 3n vs. 3n+1)
  - But at infinity, will be clearly grouped together
  - We care about which *group* a function belongs in

- Let’s convince ourselves this is the right thing to do:
  - https://www.desmos.com/calculator/t9qvn56yyb

Lecture Outline

- Overview: Algorithmic Analysis
- Code Modeling
- Asymptotic Analysis
- **Big-O Definition** ◀

Using Formal Definitions

- If analyzing simple or familiar functions, don’t bother with the formal definition. You *can* be comfortable using your intuition!
- If you take more CS classes (202, 252, 254) the formal definition will be important, so I wanted to mention it here

B

Mathematically Rigorous

Use mathematical functions as a precise, flexible basis

Big-O Definition

- We wanted to find an upper bound on our algorithm's running time, but
  - We only care about what happens as n gets large.
  - We don't want to care about constant factors.

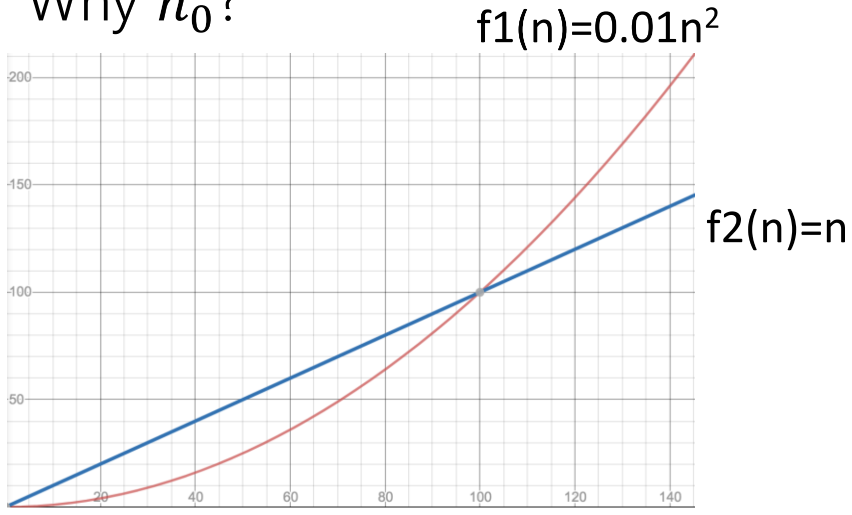
Big-O

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

$$f(n) \leq c \cdot g(n)$$

Intuition: $g(n)$ "eventually dominates" $f(n)$

Why n_0?



Why c?