LEC 05

CSE 373

O/Ω/Θ, Case Analysis

BEFORE WE START

Review: Which of the following functions are in $O(n^2)$?

$f(n) = 30n^3 + 10$
$g(n) = 10000000$
$h(n) = 2n^2 + 5n + 20$

pollev.com/uwcse373

Instructor Aaron Johnston

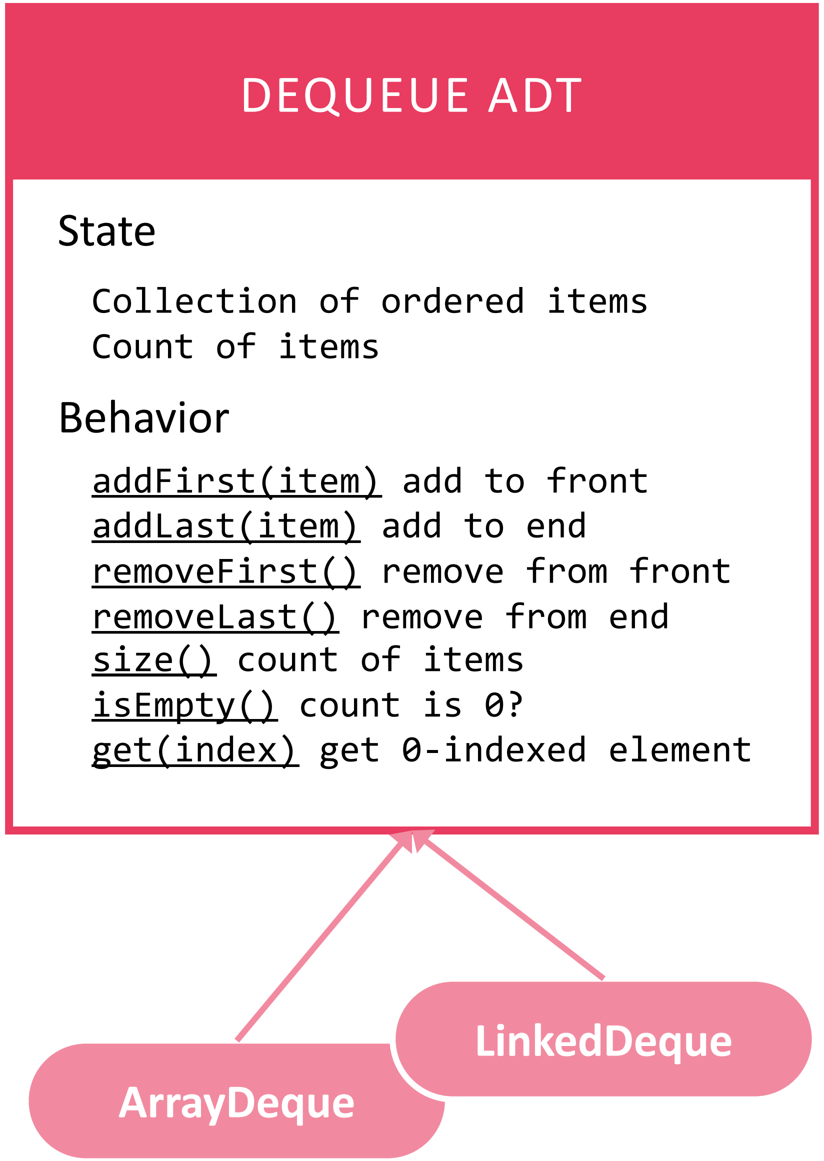
TAs Timothy Akintilo, Brian Chan, Joyce Elauria, Eric Fan, Farrell Fileas, Leona Kazi, Keanu Vestil, Howard Xiao, Siddharth Vaidyanathan

Announcements

- Project 0 (CSE 143 Review) due **TONIGHT 11:59pm**
- Project 1 (Deques) comes out this evening, due next Wednesday 7/8 11:59pm PDT
  - Remember to read the partner set-up instructions!
- Friday (July 3^{rd}) is a holiday: Independence Day (observed)
  - No lecture or office hours (we'll still check Piazza)
- Exercise 1 (written, individual) released Friday, due next Friday 7/10 11:59pm PDT
- We found the culprit who forgot to publish last lecture's recording promptly
  - (It was me)

P1: Deques

- Deque ADT: a double-ended queue
  - Add/remove from both ends, get in middle
- This project builds on ADTs vs. Data Structure Implementations, Queues, and a little bit of Asymptotic Analysis
  - Practice the techniques and analysis covered in LEC 02 & LEC 03!
- 3 components:
  - Debug ArrayDeque implementation
  - Implement LinkedDeque
  - Run experiments



P1: Sentinel Nodes

Tired of running into these?

java.lang.NullPointerException

java.lang.NullPointerException

java.lang.NullPointerException

java.lang.NullPointerException

Find yourself writing case after case in your linked node code?

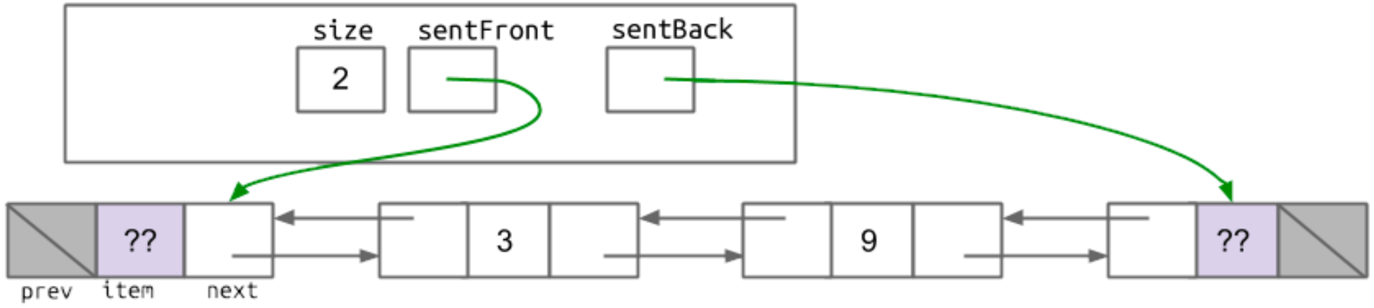
```
if (a.front != null && b.front != null) {
if (a.front != null && b.front == null) {
if (a.front == null && b.front != null) {
if (a.front == null && b.front == null) {
```

Introducing
Sentinel Nodes

- Reduce code complexity & bugs
- Tradeoff: a tiny amount of extra storage space for more reliable, easier-to-develop code

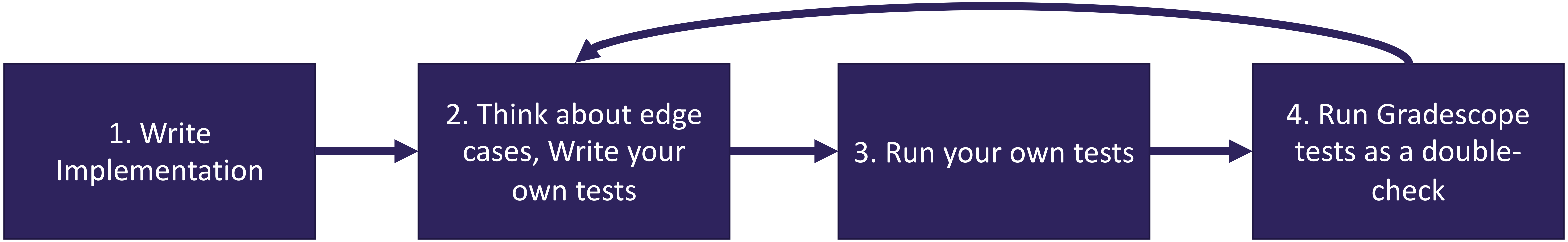
Client View: [3, 9]

Implementation:



P1: Gradescope & Testing

- From this project onward, we'll have some Gradescope-only tests
  - Run & give feedback when you submit, but only give a general name!
- The practice of reasoning about your code and writing your own tests is crucial
  - Use Gradescope tests as a double-check that your tests are thorough
  - **To debug Gradescope failures, your first step should be writing your own test case**
- You can submit as many times as you want on Gradescope (we'll only grade the last active submission)
  - If you're submitting a lot (more than ~6 times/hr) it will ask you to wait a bit
  - Intention is not to get in your way: to give server a break, and guess/check is not usually an effective way to learn the concepts in these assignments



P1: Working with a Partner

- P1 Instructions talk about collaborating with your partner
  - Adding each other to your GitLab repos
- Recommendations for partner work:
  - Pair programming! Talk through and write the code together
    - Two heads are better than one, especially when spotting edge cases ☺
  - Meet in real-time! Consider screen-sharing via Zoom
  - Be kind! Collaborating in our online quarter can be especially difficult, so please be patient and understanding – partner projects are usually an awesome experience if we're all respectful
- We expect you to understand the full projects, not just half
  - Please don't just split the projects in half and only do part
  - Please don't come to OH and say "my partner wrote this code, I don't understand it, can you help me debug it?"

Learning Objectives

After this lecture, you should be able to...

1. Differentiate between Big-Oh, Big-Omega, and Big-Theta
2. Come up with Big-Oh, Big-Omega, and Big-Theta bounds for a given function
3. Perform Case Analysis to identify the Best Case and Worst Case for a given piece of code
4. Describe the difference between Case Analysis and Asymptotic Analysis

Lecture Outline

- Big-O, Big-Omega, Big-Theta
- Case Study: Linear Search
- A New Tool: Case Analysis

Review Algorithmic Analysis Roadmap


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

$f(n) = 2n$

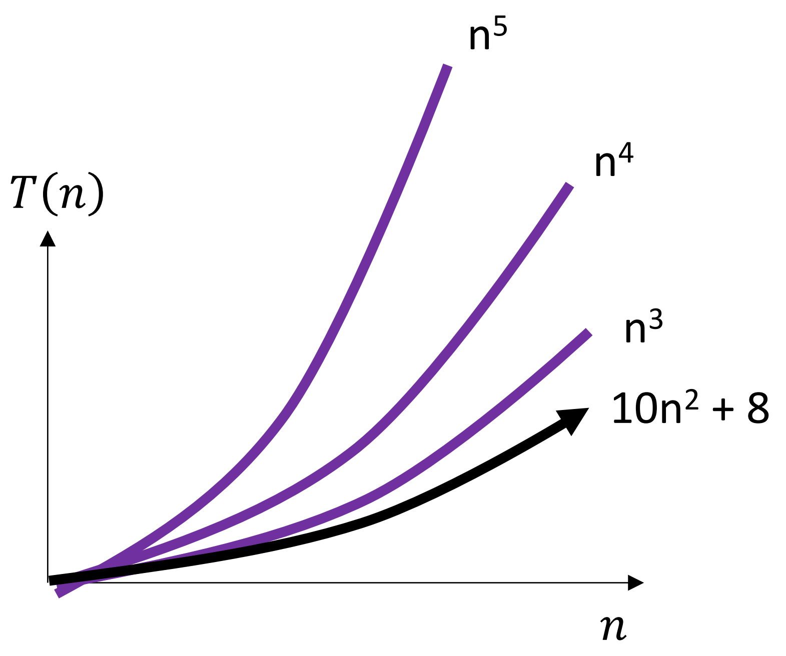
$O(n)$

- **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

Review Asymptotic Analysis

- Given a function that models some piece of code, characterize that function's growth rate asymptotically (as n approaches infinity)
  - We usually think of n as the "size of the input", so **we typically only care about non-negative integers**

$$f(n) = 10n^2 + 8$$



- Big-Oh is an upper bound on that function's growth rate
  - Constants and smaller terms ignored
  - We prefer a tight bound (e.g. n^2), **but doesn't have to be – also in $O(n^3)$**

Review Big-Oh Definition

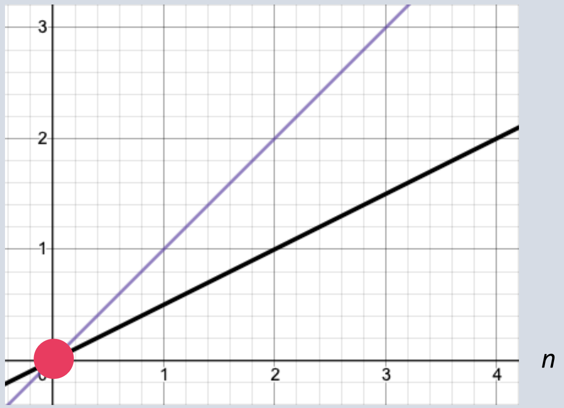
- Intuitively, f(n) is O(g(n)) if it's smaller than a constant factor of g(n), *asymptotically*
- To prove that, all we need is:
  - (**c**): What is the constant factor?
  - (**n_0**): From what point onward is f(n) smaller?

Big-Oh

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

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

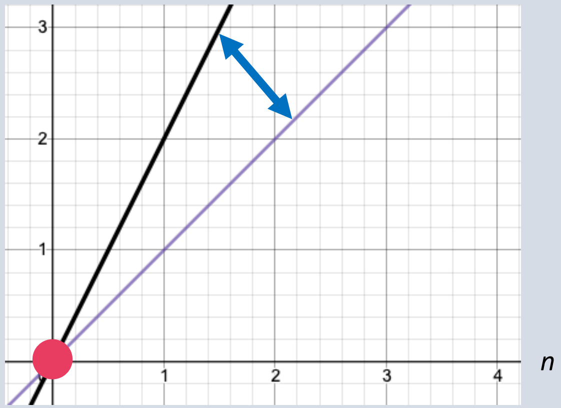
f(n) = 0.5n is O(g(n) = n)



Proof: **c=0** **n_0=0**

0.5n always ≤ n!
Straightforward O(n).

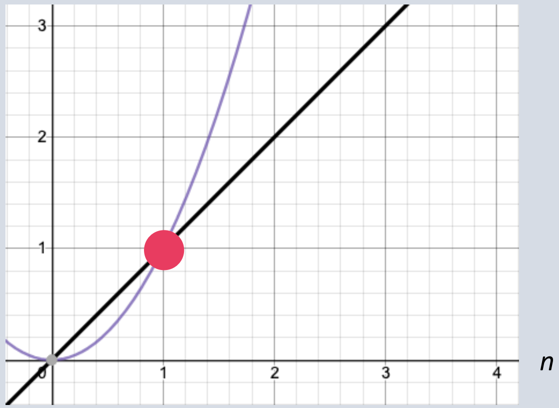
f(n) = 2n is O(g(n) = n)



Proof: **c=2** **n_0=0**

Just need to use constant factor c=2 so 2n ≤ c · n

f(n) = n is O(g(n) = n^2)

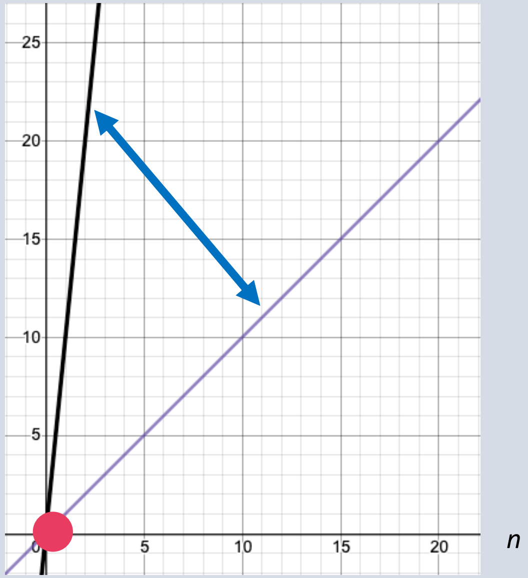


Proof: **c=0** **n_0=1**

n ≤ n^2, but only after n=1. Choose that as n_0.

Review What about Multiple Terms?

f(n) = 10n is O(g(n) = n)

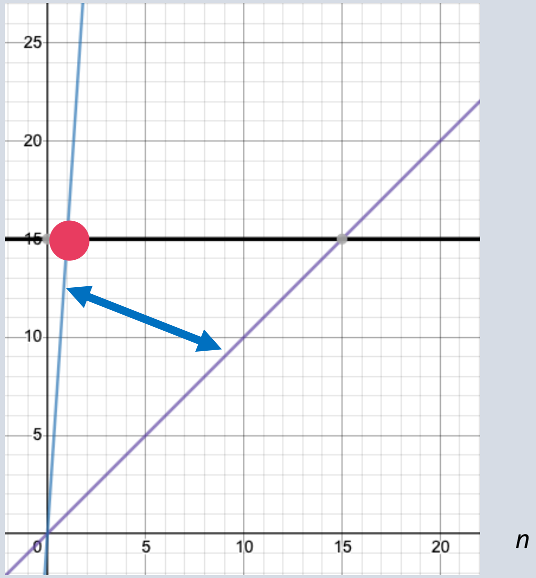


Proof: c=10 n_0=0

Just need to use constant factor c=10 so 10n ≤ c · n

\+

f(n) = 15 is O(g(n) = n)

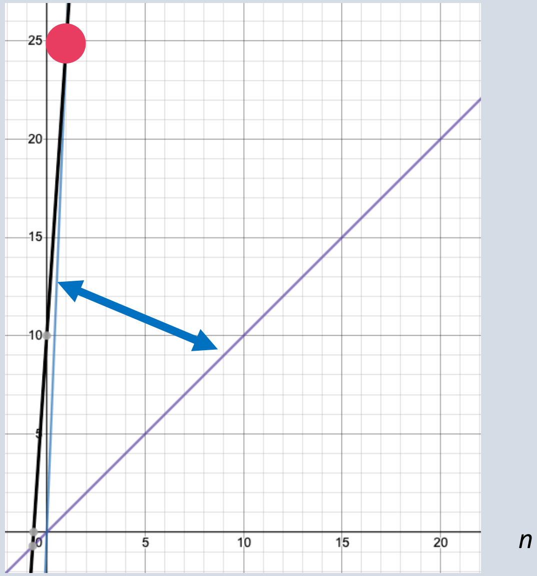


Proof: c=15 n_0=1

One way: use c=15, then 15 ≤ 15n becomes true starting at n=1

=

f(n) = 15n+10 is O(g(n) = n)



Proof: c=25 n_0=1

Just add up the proofs we came up with for individual terms!

Uncharted Waters: Prime Checking

```
boolean isPrime(int n) {
    int toTest = 2;          +1
    while(toTest < n) {      +1
        if (n % toTest == 0) {   +2
            return false;    +1
        } else {
            toTest += 1;     +2
        }
    }
    return true;        +1
}
```

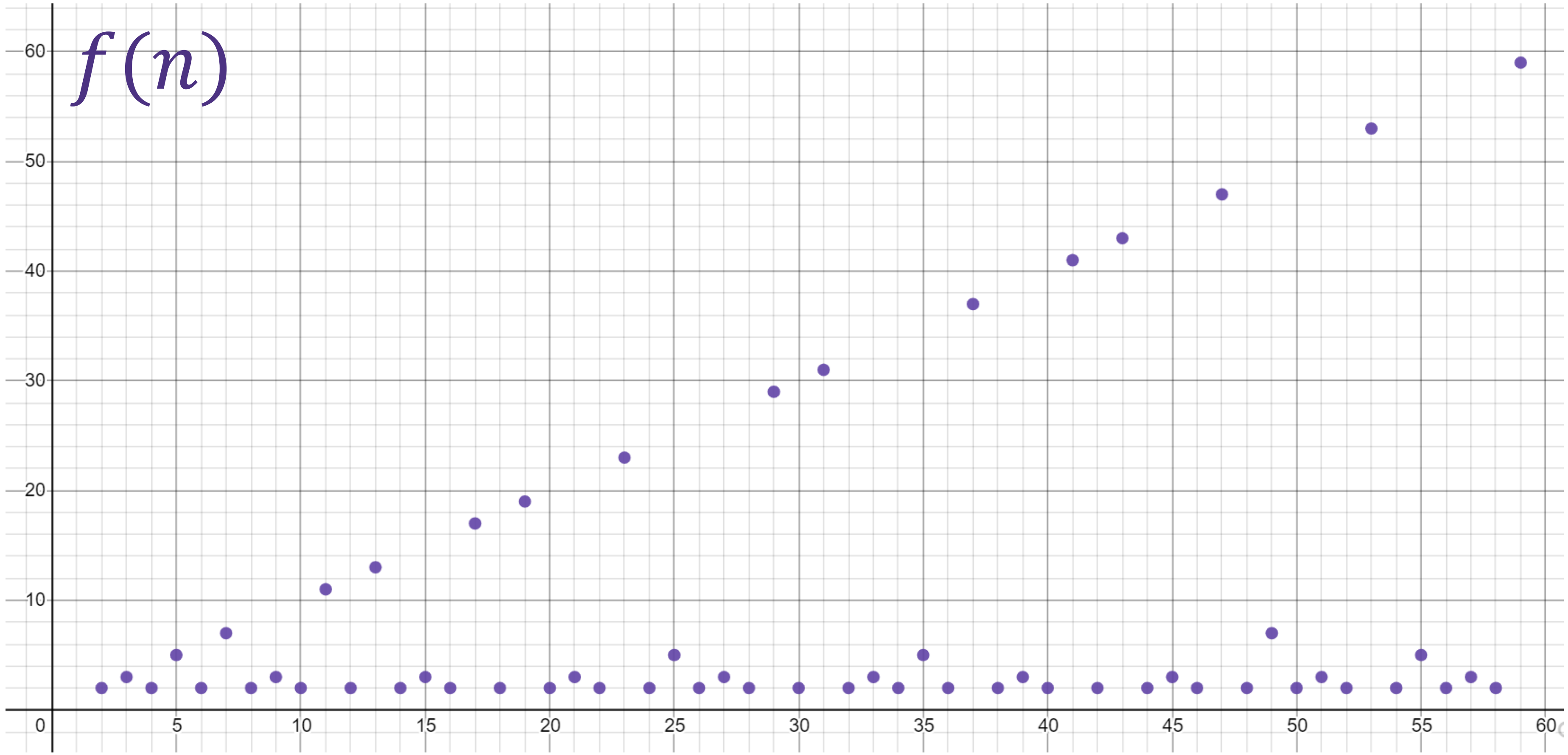
~+5

*?

- Find a model $f(n)$ for the running time of this code on input n → What's the Big-O?
  - We know how to count the operations
  - But how many times does this loop run?

- Sometimes it can stop early
- Sometimes it needs to run n times

Prime Checking Runtime

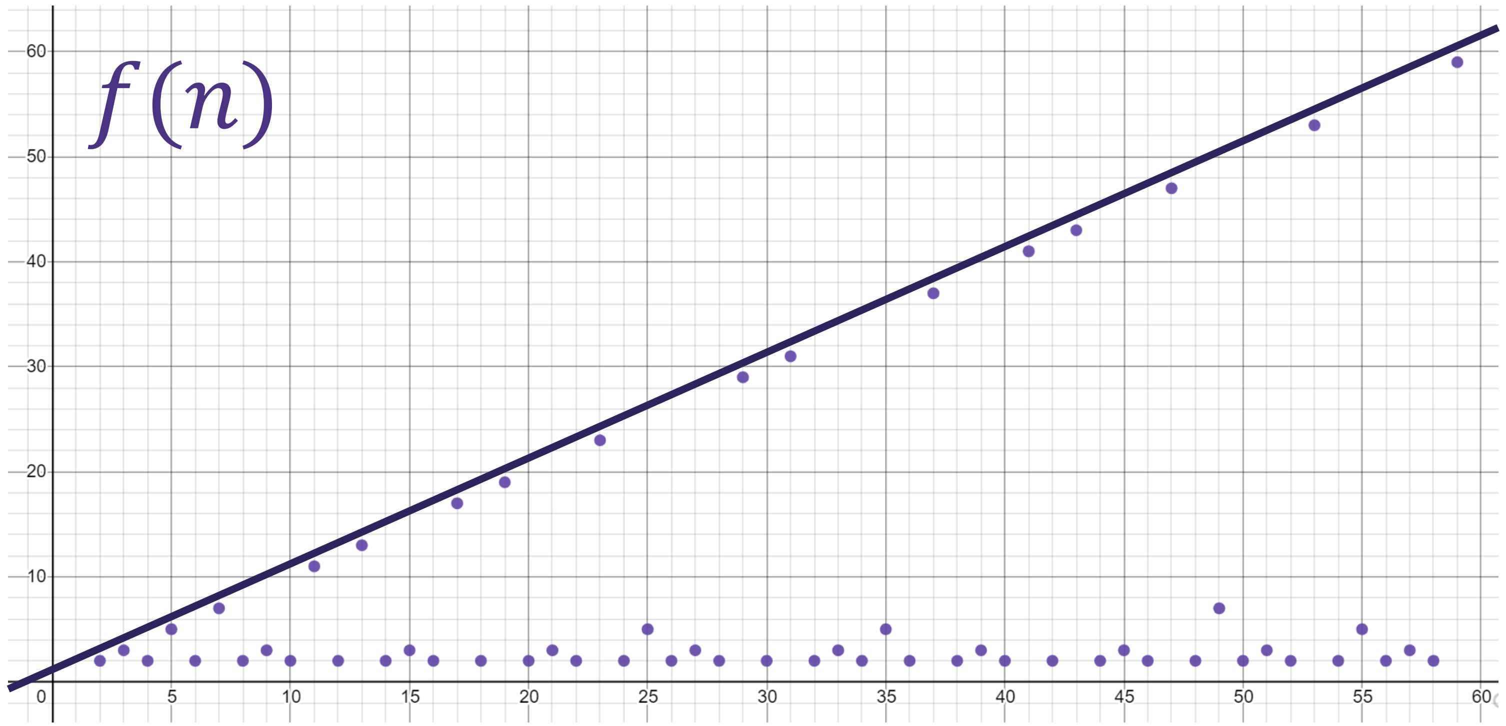


Is the runtime O(1) or O(n)?

More than half the time we need 3 or fewer iterations. Is it O(1)?

But we can always come up with another value of n to make it take n iterations. So O(n)?

This is why we have definitions!



Big-O

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

Using our definitions, we see it's O(n) and not O(1)

Is the runtime O(n)?
Can you find constants c and n_0?

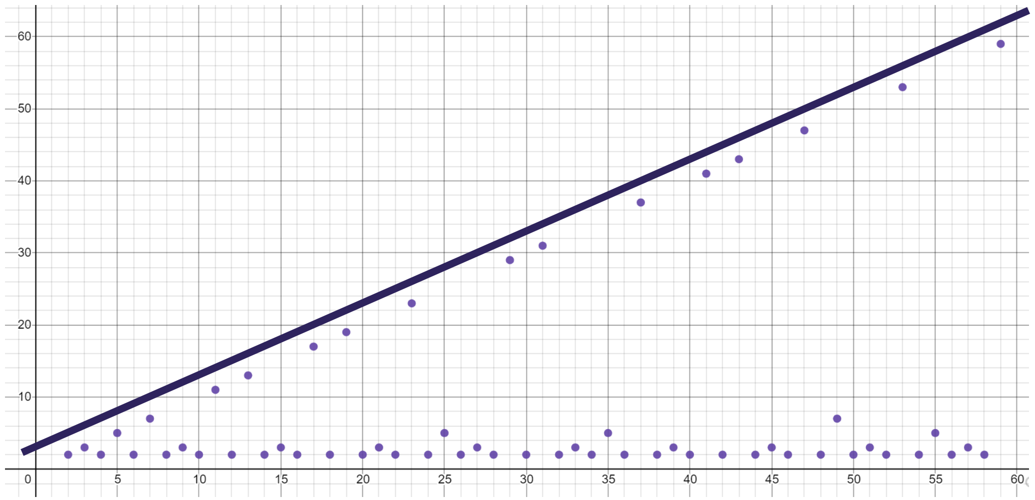
How about $c = 1$ and $n_0 = 5$,
$f(n)$ =smallest divisor of $n \leq 1 \cdot n$ for $n \geq 5$

Is the runtime O(1)?
Can you find constants c and n_0?

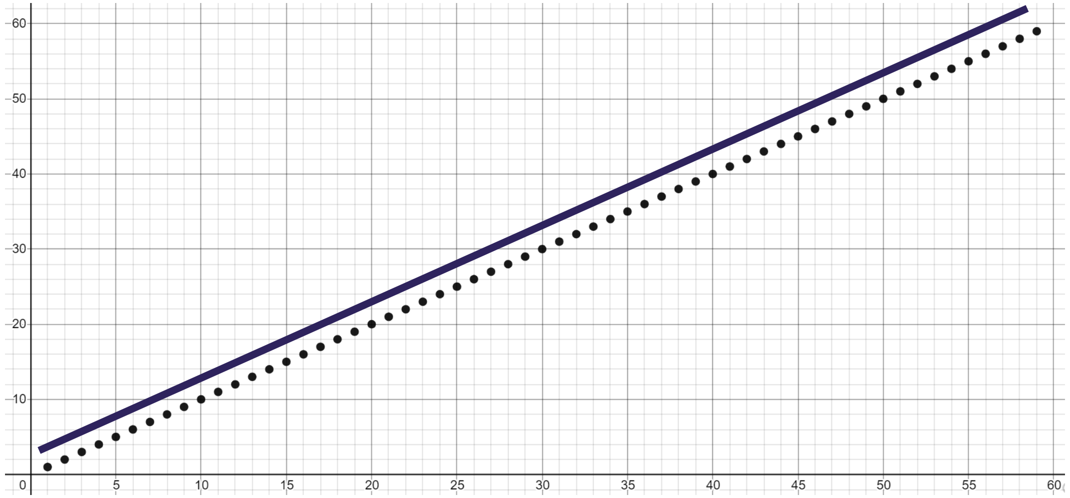
No! Choose your value of c. I can find a prime number k bigger than c.
And $f(k) = k > c \cdot 1$ so the definition isn't met!

Big-Oh isn't everything

- Our prime finding code is O(n) as a tight bound. But so is printing all the elements of a list (a basic for loop).



$O(n)$



$O(n)$

Your experience running these two pieces of code is going to be very different.
It's disappointing that the Big-Ohs are the same – that's not very precise!
Could we have some way of pointing out the list code always takes AT LEAST n operations?

Big-Ω [Omega]

Big-Omega

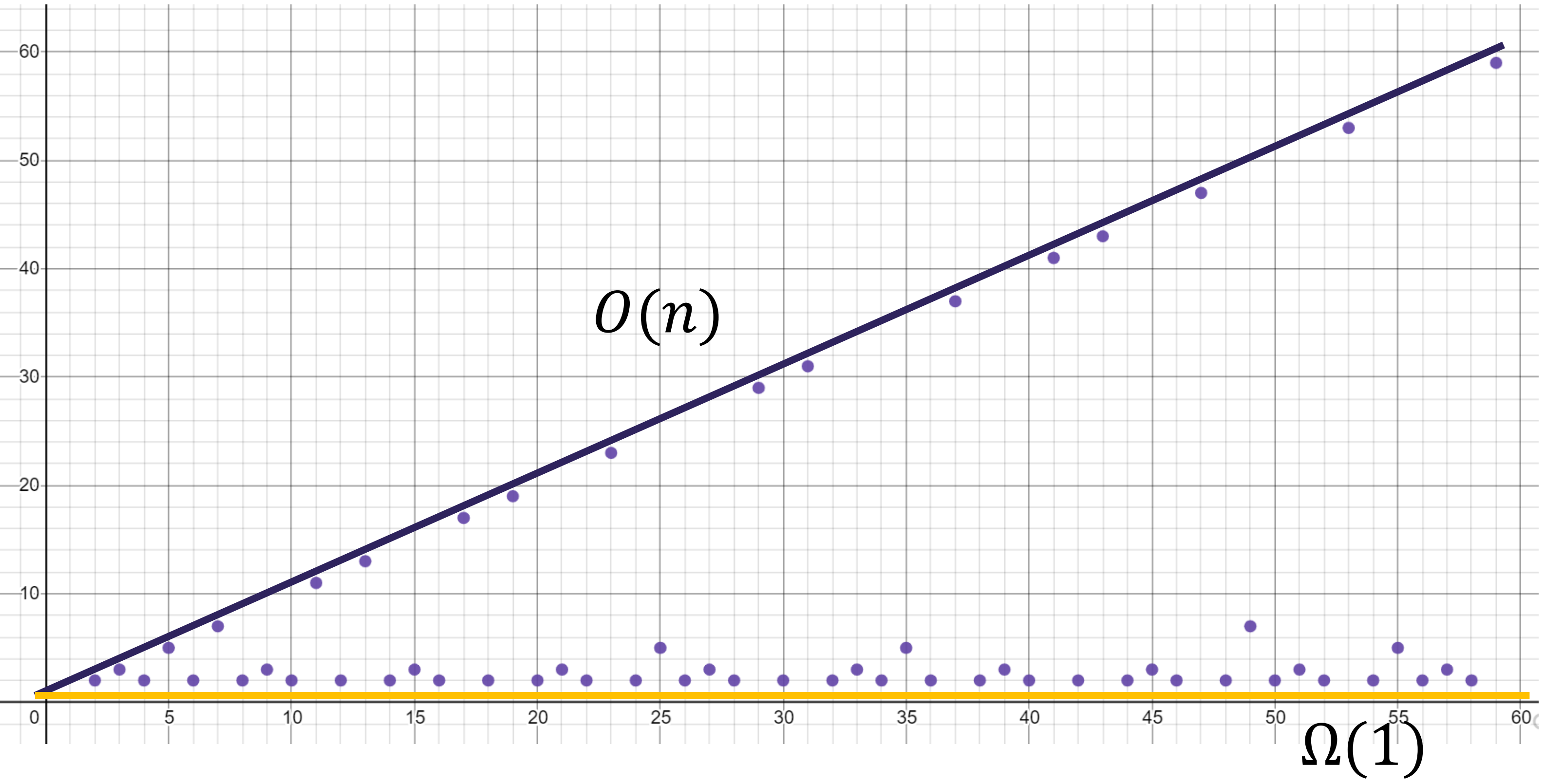
$f(n)$ is $\Omega(g(n))$ if there exist positive constants c, n_0 such that for all $n \geq n_0$,

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

Big-O

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

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



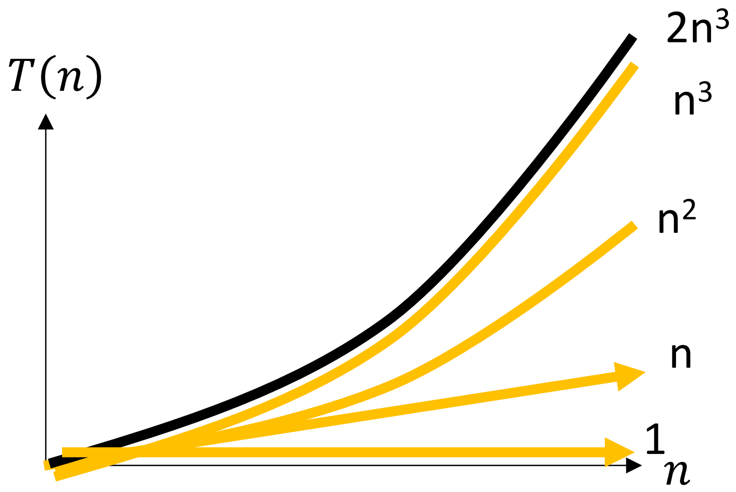
The formal definition of Big-Omega is the flipped version of Big-Oh!

"f(n) is O(g(n))" : f(n) grows at most as fast as g(n)

"f(n) is Ω(g(n))" : f(n) grows at least as fast as g(n)

Big-Omega Also Doesn't Have to be Tight

- $2n^3$ is $\Omega(1)$
- $2n^3$ is $\Omega(\mathrm{n})$
- $2n^3$ is $\Omega(n^2)$
- $2n^3$ is $\Omega(n^3)$



- $2n^3$ is lowerbounded by all the complexity classes listed above $(1, \mathrm{n}, n^2, n^3)$

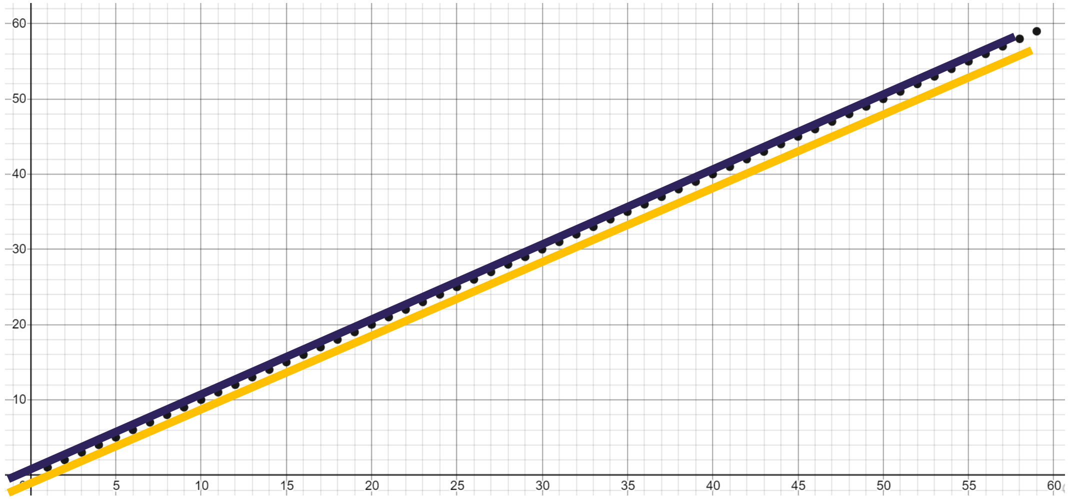
Tight Big-O and Big-Ω Bounds Together

Prime runtime function

$O(n)$

$\Omega(1)$

f(n) = n



$O(n)$

$\Omega(\mathrm{n})$

Note: *most* functions look like the one on the right, with the same tight Big-Oh and Big-Omega bound. But we'll see important examples of the one on the left.

Oh, and Omega, and Theta, oh my

- Big-Oh is an **upper bound**
  - My code takes at most this long to run
- Big-Omega is a **lower bound**
  - My code takes at least this long to run
- Big Theta is **"equal to"**
  - My code takes "exactly"* this long to run
  - *Except for constant factors and lower order terms
  - Only exists when Big-Oh == Big-Omega!

Big-Oh

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

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

Big-Omega

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

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

Big-Theta

$f(n)$ is $\Theta(g(n))$ if
$f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.
(in other words: there exist positive constants $c1, c2, n_0$ such that for all $n \geq n_0$)

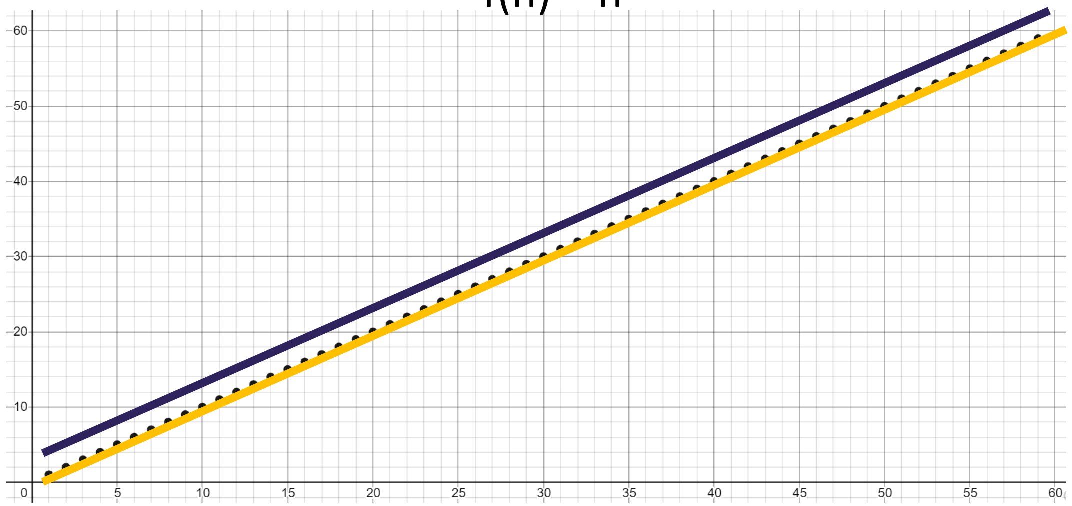
$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

Oh, and Omega, and Theta, oh my

Big Theta is **"equal to"**

- My code takes "exactly"* this long to run
- *Except for constant factors and lower order terms

f(n) = n



$O(n)$ $\Omega(\mathrm{n})$ → $\Theta(n)$

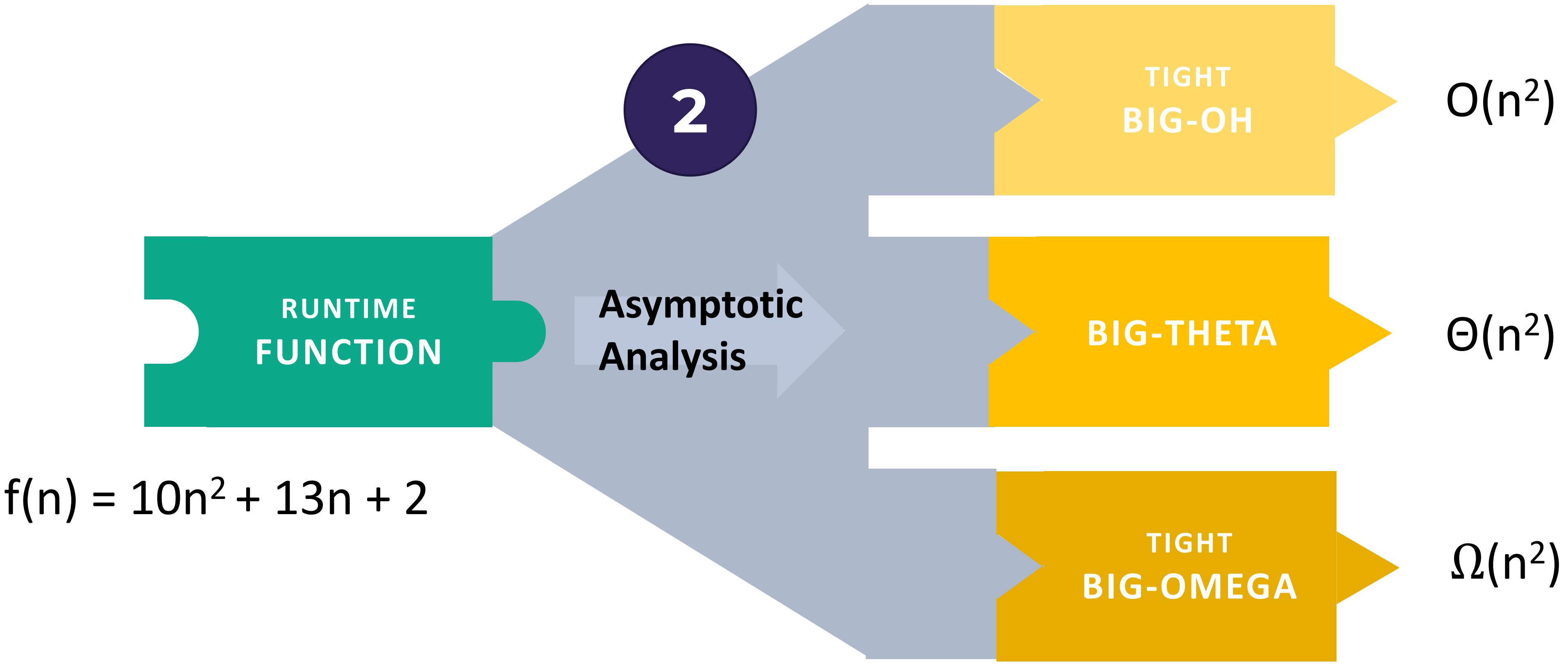
Big-Theta

$f(n)$ is $\Theta(g(n))$ if
$f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.
(in other words: there exist positive constants $c1, \mathrm{c}2, n_0$ such that for all $n \geq n_0$)

$$\mathrm{c}_1 \cdot g(n) \leq f(n) \leq \mathrm{c}_2 \cdot g(n)$$

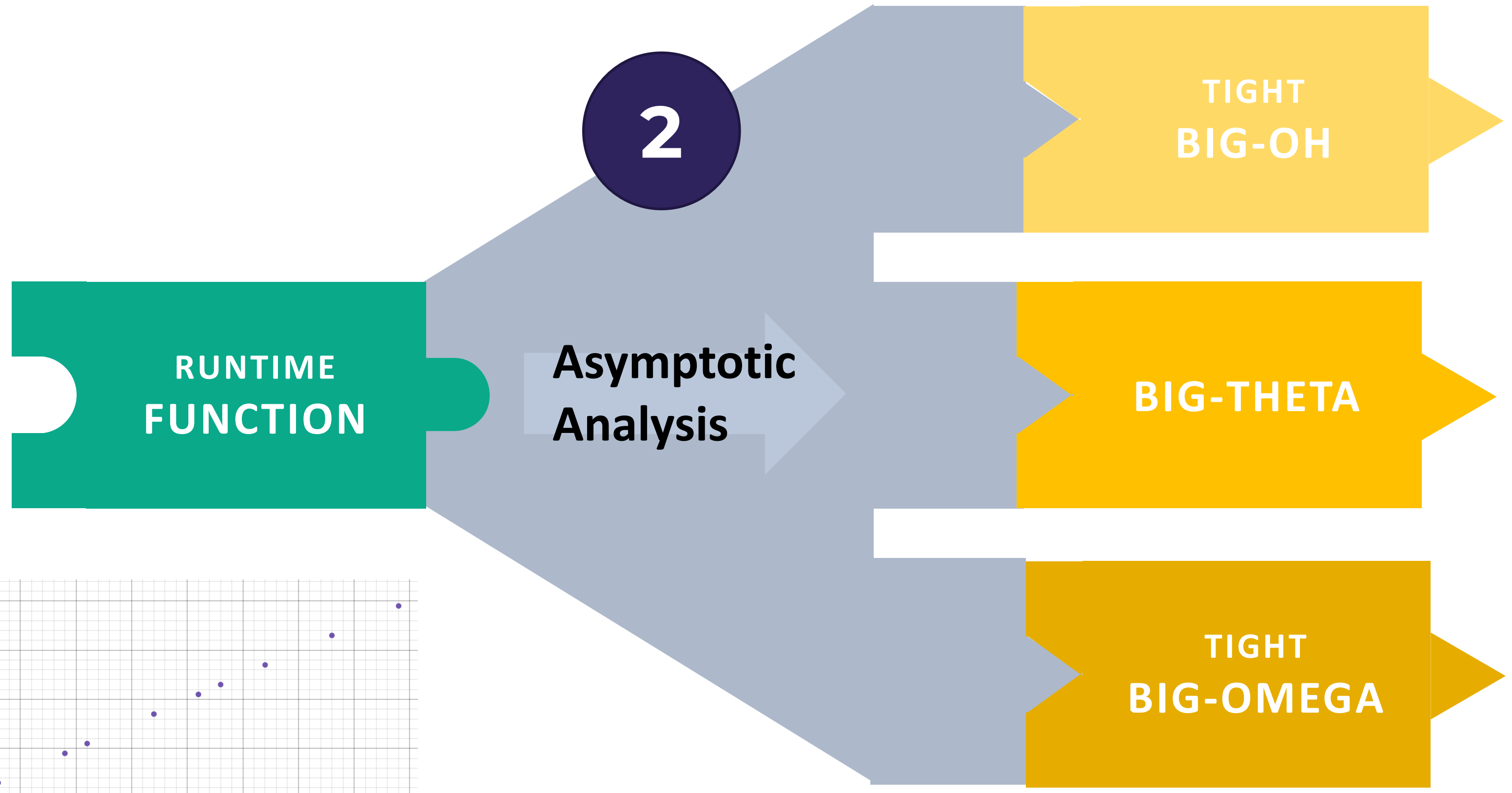
To define a big-Theta, you expect the tight big-Oh and tight big-Omega bounds to be touching on the graph (the same complexity class)

Our Upgraded Tool: Asymptotic Analysis



We've upgraded our Asymptotic Analysis tool to convey more useful information! Having 3 different types of bounds means we can still characterize the function in simple terms, but describe it more thoroughly than just Big-Oh.

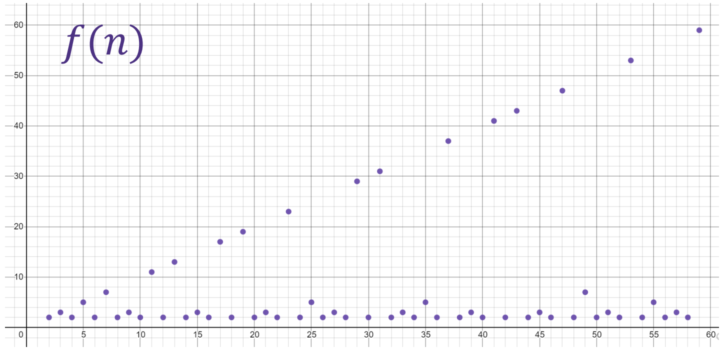
Our Upgraded Tool: Asymptotic Analysis



O(n)

Does not exist for this function

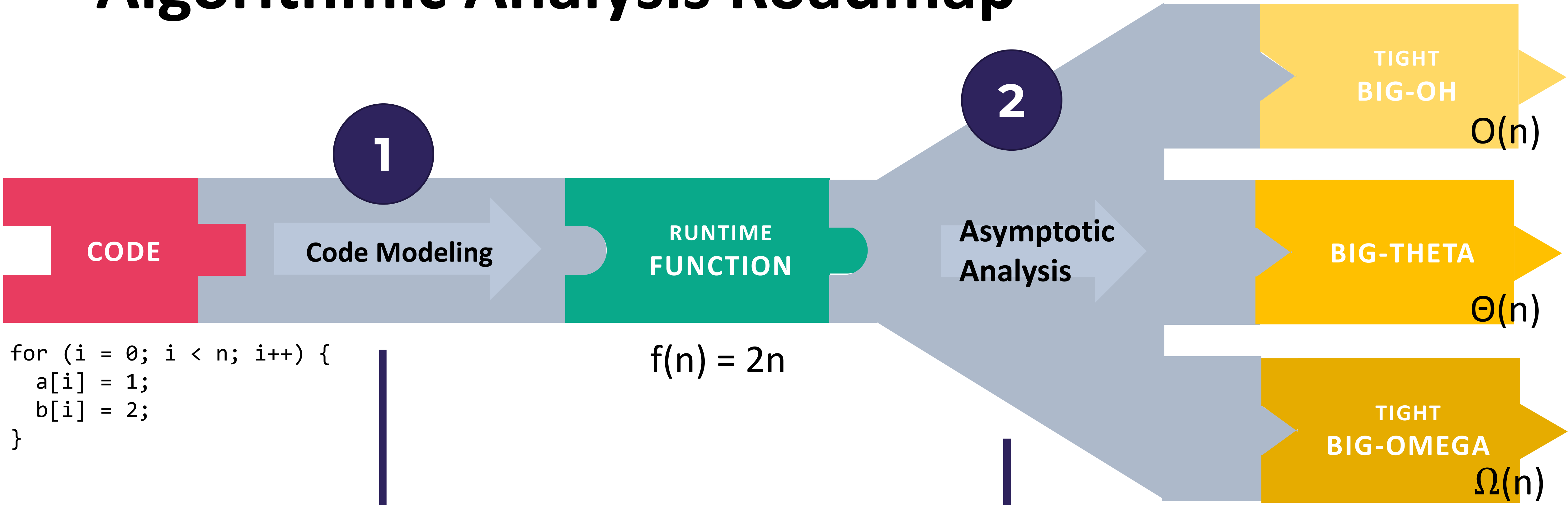
Ω(1)



isPrime()

Big-Theta doesn't always exist for every function! But the information that Big-Theta doesn't exist can *itself* be a useful characterization of the function.

Algorithmic Analysis Roadmap




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

$f(n) = 2n$

Now, let's look at this tool in more depth. How exactly are we coming up with that function?

We just finished building this tool to characterize a function in terms of some useful bounds!

Lecture Outline

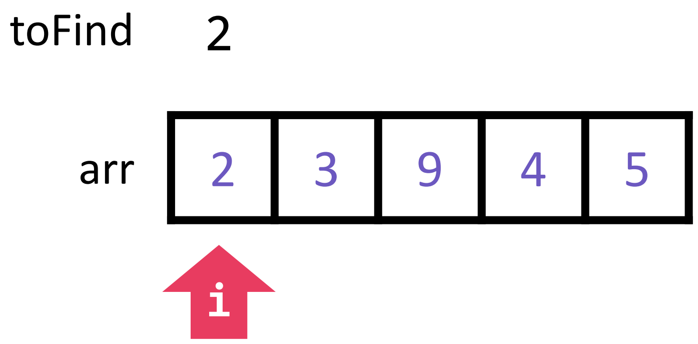
- Big-O, Big-Omega, Big-Theta
- **Case Study: Linear Search**
- A New Tool: Case Analysis

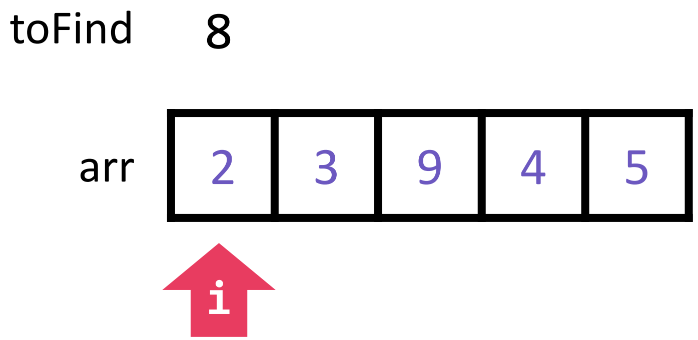
Case Study: Linear Search

- Let’s analyze this realistic piece of code!

```
int linearSearch(int[] arr, int toFind) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == toFind) {
            return i;
        }
    }
    return -1;
}
```

- What’s the first step?
  - We have code, so we need to convert to a function describing its runtime
  - Then we know we can use asymptotic analysis to get bounds





Let's Model This Code!

*Remember, these constants don't really matter (we'll start phasing them out soon)

```
int linearSearch(int[] arr, int toFind) {
    for (int i = 0; i < arr.length; i++) {   +2
        if (arr[i] == toFind) {   +1
            return i;   +1
        }
    }
    return -1;   +1
}
```

+3 *?

Same problem as before:
How many times does loop run?

When would that happen?

- Suppose the loop runs n times?
  - $f(n) = 3n + 1$
- Suppose the loop only runs once?
  - $f(n) = 2$

toFind not present

toFind at beginning

These are key!

Best Case

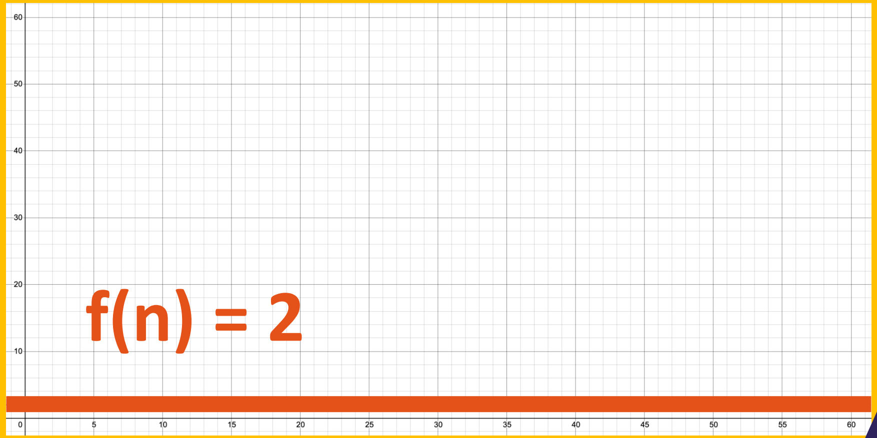
On Lucky Earth

toFind 2

arr

2	3	9	4	5

i

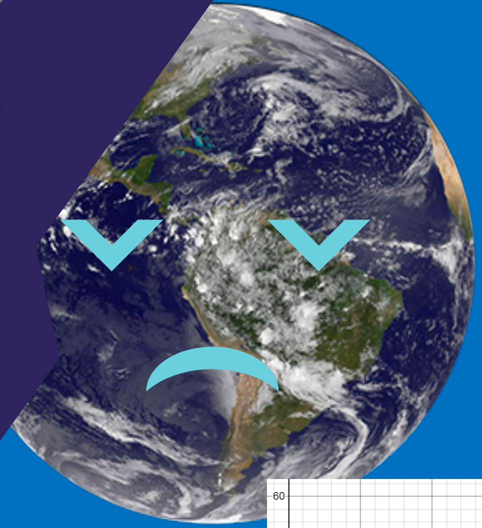


After asymptotic analysis:

$O(1)$ $\Theta(1)$ $\Omega(1)$

Worst Case

On Unlucky Earth (where today is 6/31)

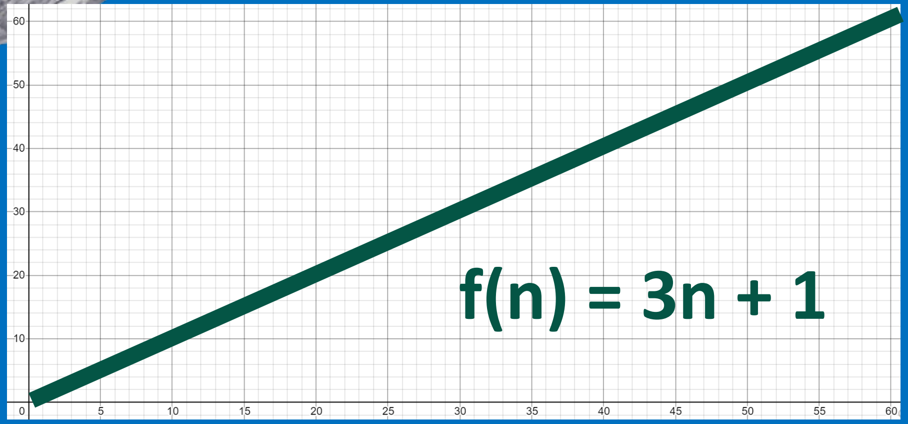


toFind 8

arr

2	3	9	4	5

i



After asymptotic analysis:

$O(n)$ $\Theta(n)$ $\Omega(n)$

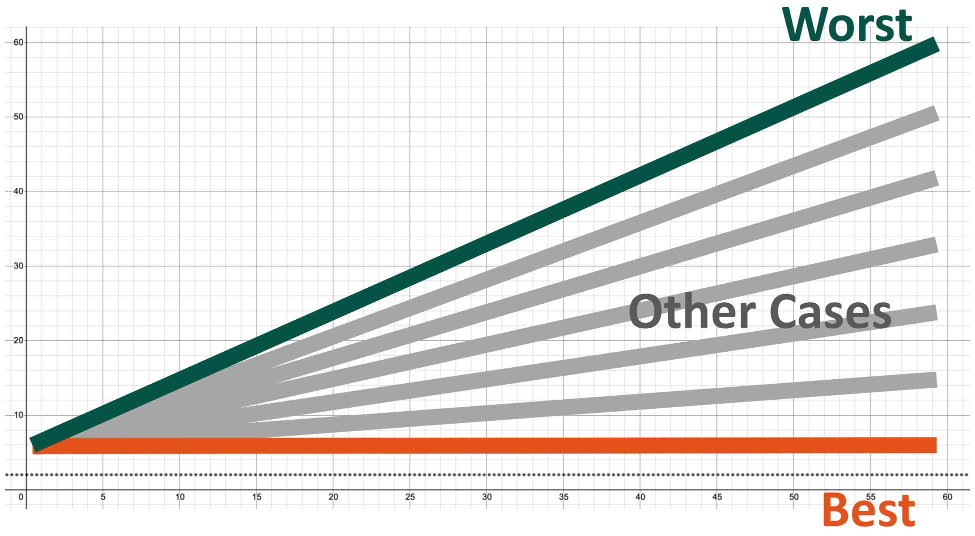
Lecture Outline

- Big-O, Big-Omega, Big-Theta
- Case Study: Linear Search
- **A New Tool: Case Analysis**

Case Analysis

- **Case**: a description of inputs/state for an algorithm that is specific enough to build a code model (runtime function) whose only parameter is the input size
  - Case Analysis is our tool for reasoning about **all variation other than n**!
  - Occurs during the code → function step instead of function → O/Ω/Θ step!

- (Best Case: fastest/Worst Case: slowest) that our code could finish on input of size n.
- Importantly, *any* position of toFind in arr could be its own case!
  - For this simple example, probably don't care (they all still have bound Θ(n))
  - But intermediate cases will be important later



When to do Case Analysis?

- Why are the different functions in `isPrime` not Case Analysis, but the different functions in `linearSearch` are?
  - In `isPrime`, they're different bounds on a single function over n.
  - in `linearSearch`, they're entirely different functions over n, each with its own set of bounds!

- The difference? `linearSearch` uses **another input** as well, the *contents* of the array – **that variation creates different functions over n!**

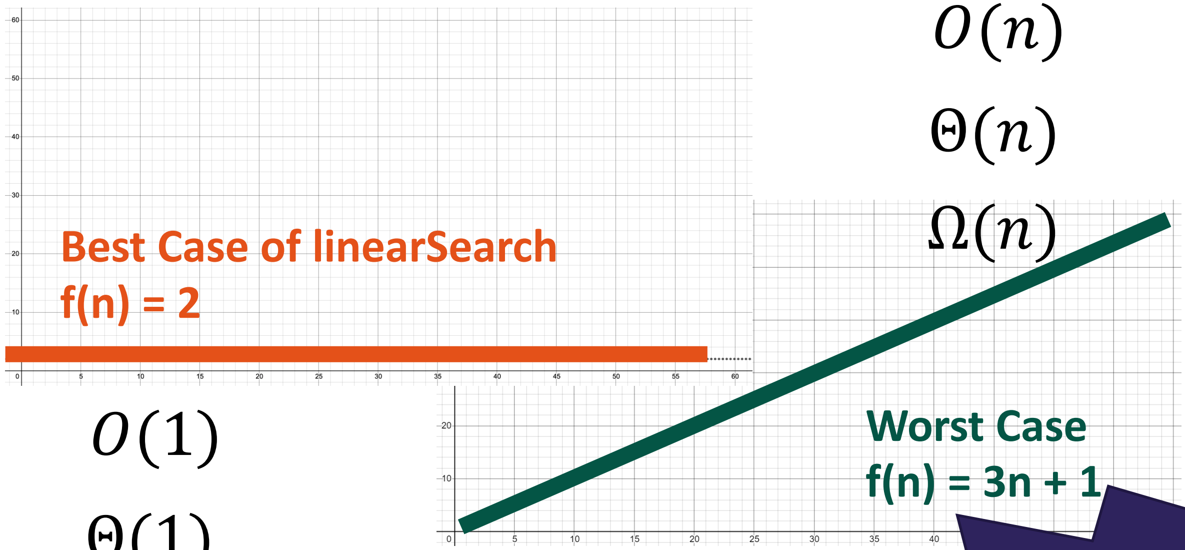
```
boolean isPrime(int n) {

int linearSearch(int[] arr, int toFind) {
```

When to do Case Analysis?

Case Analysis, then Asymptotic Analysis

`linearSearch`:
- multiple different functions over n, because runtime can be affected by something other than n!
- for each function, we'll do asymptotic analysis



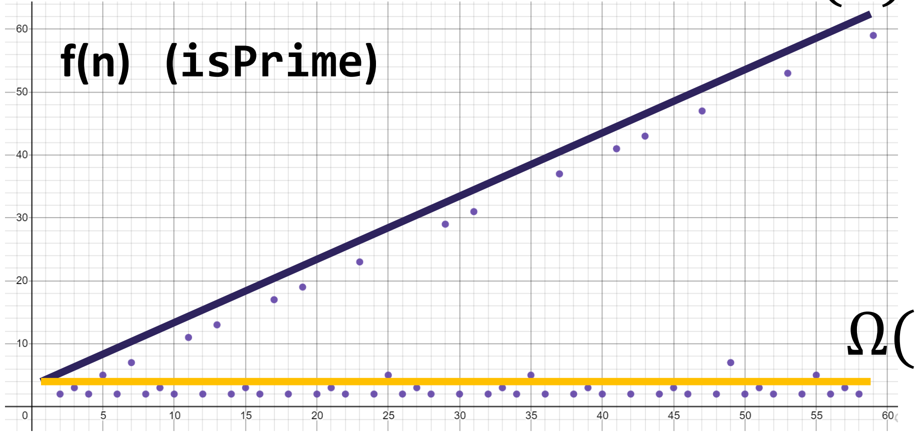
$O(n)$
$\Theta(n)$
$\Omega(n)$

$O(1)$
$\Theta(1)$
$\Omega(1)$

Straight to Asymptotic Analysis

`isPrime`:
- only has one function to consider, because only input is n!

$O(n)$

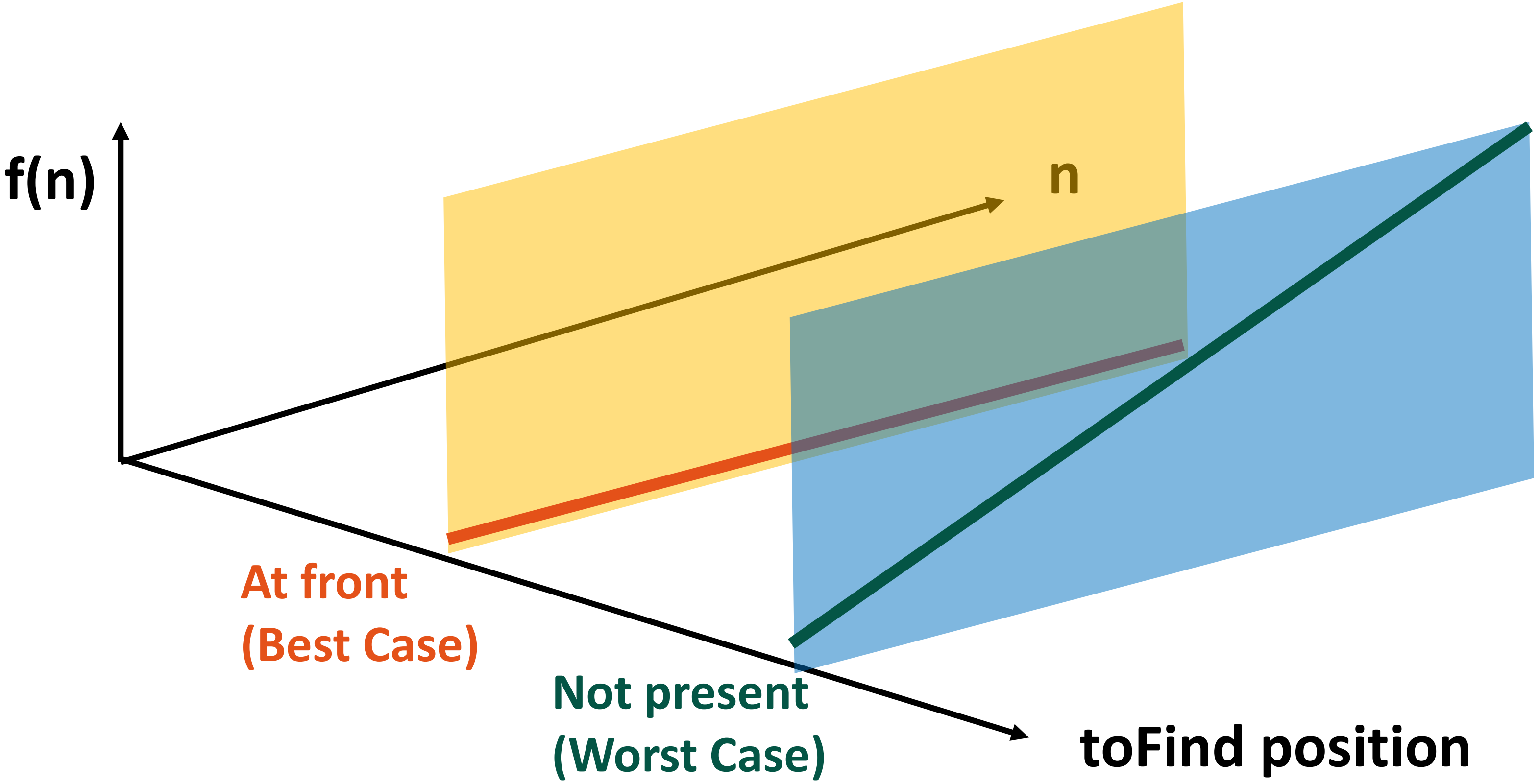


$\Omega(1)$

Do Case Analysis when varying other input properties *besides* n can change runtime!

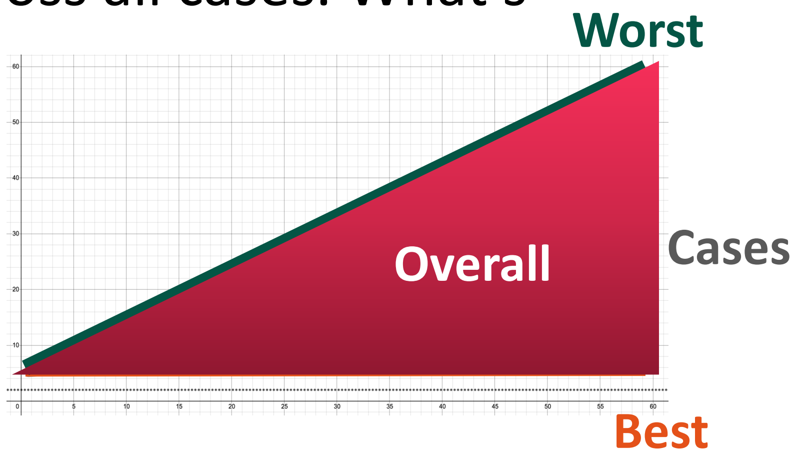
When to do Case Analysis?

- Imagine a 3-dimensional plot
  - Which case we're considering is one dimension
  - Choosing a case lets us take a "slice" of the other dimensions: n and f(n)
  - We do asymptotic analysis on each slice in step 2

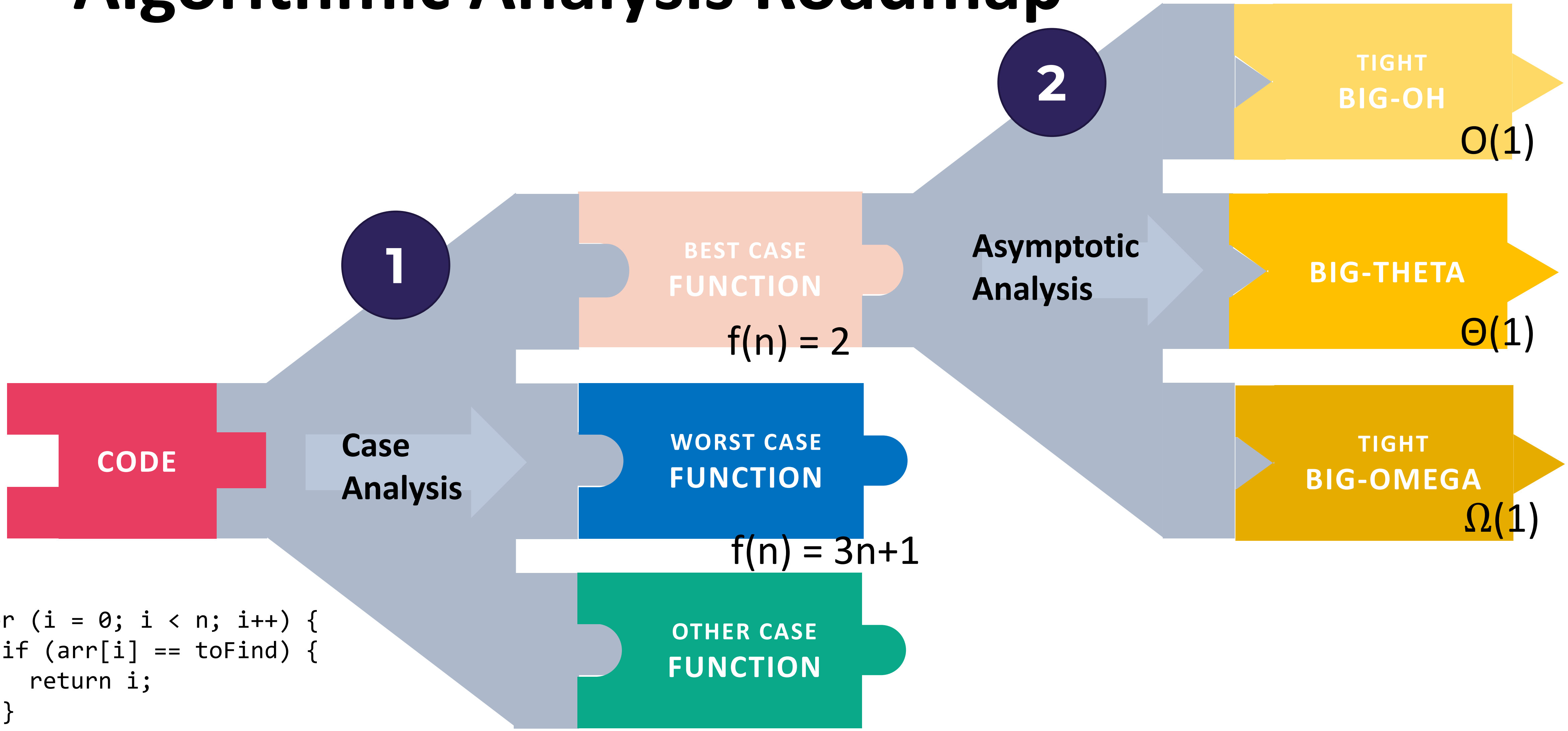


Other Useful Cases You Might See

- Overall Case:
  - Model code as a "cloud" that covers all *possibilities* across all cases. What's the O/Ω/Θ of that cloud?
- "Assume X Won't Happen Case":
  - E.g. Assume array won't need to resize
- "Average Case":
  - Assume random input
  - Lots of complications – what distribution of random?
- "In-Practice Case":
  - Not a real term, but a useful idea
  - Make reasonable assumptions about how the world will work, then do worst-case analysis under those assumptions.



Algorithmic Analysis Roadmap




```
for (i = 0; i < n; i++) {
  if (arr[i] == toFind) {
    return i;
  }
}
return -1;
```

How Can You Tell if Best/Worst Cases Exist?

- Are there other possible models for this code?
- **If n is given, are there still other factors that determine the runtime?**

- Note: sometimes there aren't significantly different cases! Sometimes we just want to model the code with a single function and go straight to asymptotic analysis!

Can We Choose n=0 as the Best Case?

- Remember that each case needs to be a “slice”: a function over n
  - The input to asymptotic analysis is a function over all of n, because we’re concerned with growth rate
  - Fixing n doesn’t work with our tools because it wouldn’t let us examine the bound asymptotically

- Think of it as “Best Case as n grows infinitely large”, not “Best Case of all inputs, including n”

How to do Case Analysis

1. Are there significantly different cases?
   - Do other variables/parameters/fields affect the runtime, other than input size? For many algorithms, the answer is no.
2. Figure out how things could change depending on the input (excluding n, the input size)
   - Can you exit loops early?
   - Can you return early?
   - Are some branches much slower than others?
3. Determine what inputs could cause you to hit the best/worst parts of the code.

Cases vs. Asymptotic

	Big-O	Big-Omega	Big-Theta
Worst Case	No matter what, as n gets bigger, the code takes at most this much time	Under certain circumstances, as n gets bigger, the code takes at least this much time	On the worst input, as n gets bigger, the code takes precisely this much time (up to constants).
Best Case	Under certain circumstances, even as n gets bigger, the code takes at most this much time.	No matter what, even as n gets bigger, the code takes at least this much time.	On the best input, even as n gets bigger, the code takes precisely this much time (up to constants)

"worst input": input that causes the code to run slowest.

Cases vs. Asymptotic

	Big-O	Big-Omega	Big-Theta
Worst Case	No matter what, as n gets bigger, the code takes at most this much time	Under certain circumstances, as n gets bigger, the code takes at least this much time	On the worst input, as n gets bigger, the code takes precisely this much time (up to constants).
Best Case	Under certain circumstances, even as n gets bigger, the code takes at most this much time.	No matter what, even as n gets bigger, the code takes at least this much time.	On the best input, even as n gets bigger, the code takes precisely this much time (up to constants)

"worst input": input that causes the code to run slowest.

Cases vs. Asymptotic

	Big-O	Big-Omega	Big-Theta
Worst Case	No matter what, as n gets bigger, the code takes at most this much time	Under certain circumstances, as n gets bigger, the code takes at least this much time	On the worst input, as n gets bigger, the code takes precisely this much time (up to constants).
Best Case	Under certain circumstances, even as n gets bigger, the code takes at most this much time.	No matter what, even as n gets bigger, the code takes at least this much time.	On the best input, even as n gets bigger, the code takes precisely this much time (up to constants)

“worst input”: input that causes the code to run slowest.