Lecture 5: Big O and Case Analysis

CSE 373: Data Structures and Algorithms

Warm Up

Slido Event #3113134
https://app.sli.do/event/eZSybDnHLRnuNXw3WN2zGj

Construct a mathematical function modeling the runtime for the following functions

```
public void mystery2(ArrayList<String> list) {
    for (int i = 0; i < list.size(); i++) {
        for (int j = 0; j < list.size(); j++) {
            System.out.println(list.get(0)); +2
        }
    }
}
```

*n

*n

$f(n) = 2n^2$

Approach

-> start with basic operations, work inside out for control structures

- Each basic operation = +1
- Conditionals = test operations + appropriate branch
- Loop = iterations * loop body

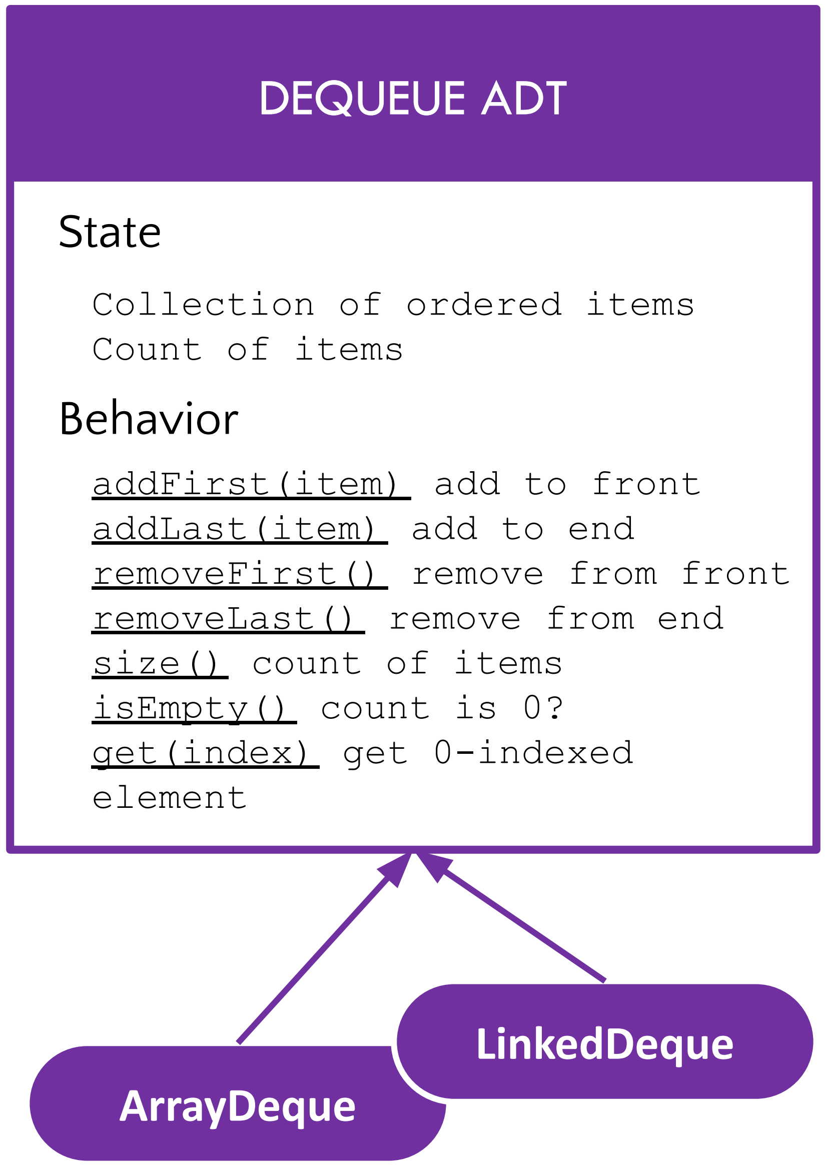
Announcements

- Project 0 – 143 Review Project Due Tonight 11:59pm PST
- Project 1 - Deques releases tonight
  - Due Wednesday April 12th
- Exercise 0 out - Due Monday 4/10
  - Individual submissions

P1 Deques

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



DEQUEUE ADT

State

```
Collection of ordered items
Count of items
```

Behavior

```
addFirst(item) add to front
addLast(item) add to end
removeFirst() remove from front
removeLast() remove from end
size() count of items
isEmpty() count is 0?
get(index) get 0-indexed
element
```

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) {
```

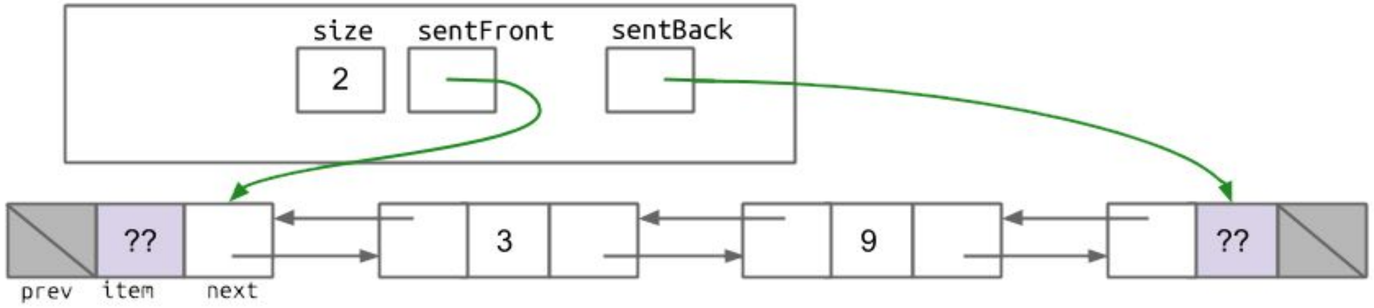
Introducing
Sentinel Nodes

Reduce code complexity & bugs

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

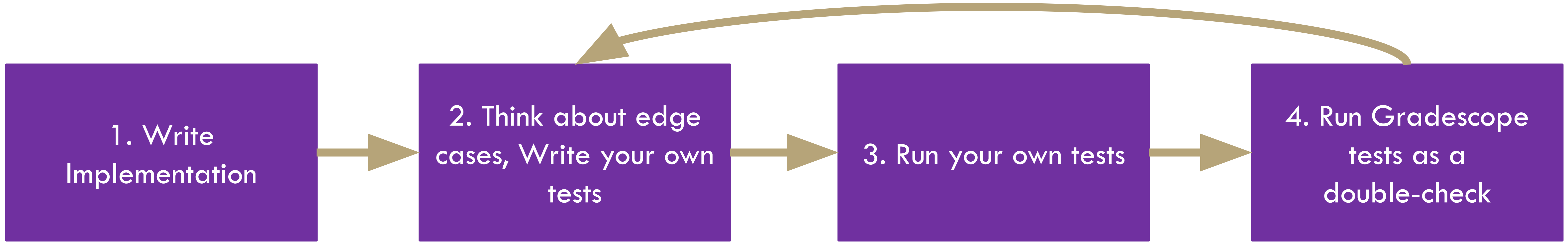
Client View: `[3, 9]`

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


1. Write Implementation
2. Think about edge cases, Write your own tests
3. Run your own tests
4. Run Gradescope tests as a double-check

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?"

Questions?

Big O

Definition: Big-O

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

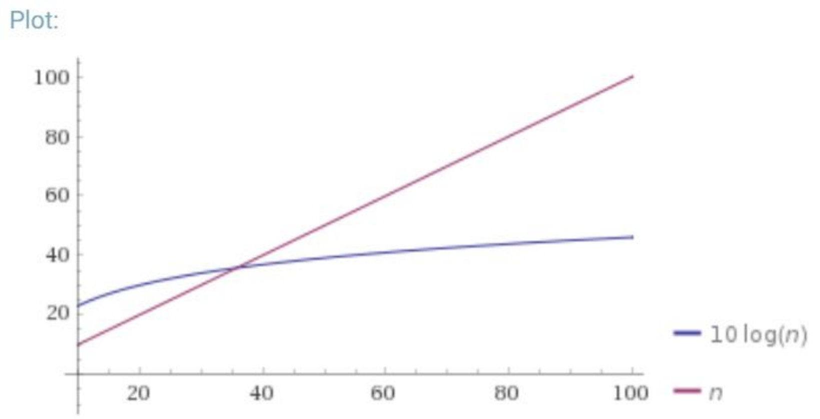
- We don't want to care about constant factors
- We only care about what happens as *n* gets larger

Big-O

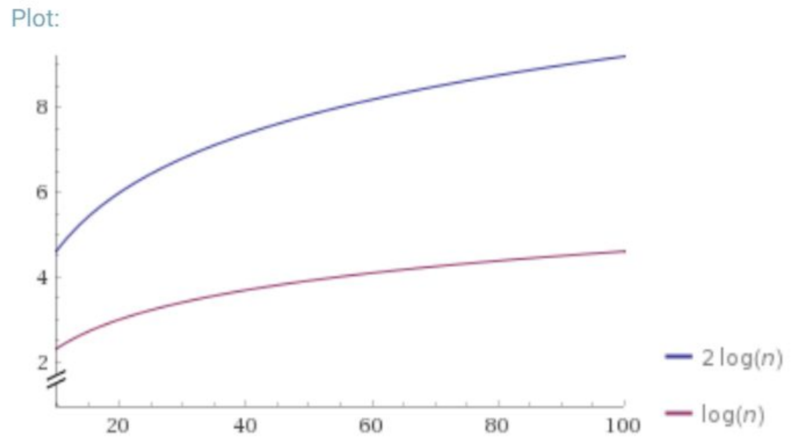
f(n) is O(g(n)) if there exist positive constants c, n_0, such that for all $n \geq n_0$,
$f(n) \leq c \cdot g(n)$

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

Why n_0?



Why c?



Applying Big O Definition

Big-O

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

Show that $f(n) = 10n + 15$ is $O(n)$

Apply definition term by term

$$10n \leq c \cdot n \text{ when } c = 10 \text{ for all values of } n$$

$$15 \leq c \cdot n \text{ when } c = 15 \; for \; n \geq 1$$

Add up all your truths

$$10n + 15 \leq 10n + 15n = 25n \text{ for } n \geq 1$$

Select values for c and n_0 and prove they fit the definition

Take $c = 25$ and $n_0 = 1$

$10n \leq 10n \; for \; all \; values \; of \; n$

$15 \leq 15n \; for \; n \geq 1$

So $10n + 15 \leq 25n$ for all $n \geq 1$, as required.

because a c and n_0 exist, $f(n)$ is $O(n)$

Exercise: Proving Big O

Big-O

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

Demonstrate that $5n^2 + 3n + 6$ is dominated by n^2 (i.e. that $5n^2 + 3n + 6$ is $O(n^2)$, by finding a c and n_0 that satisfy the definition of domination

$5n^2 + 3n + 6 \leq 5n^2 + 3n^2 + 6n^2$ when $n \geq 1$

Writing Big-O Proofs

Steps to a big-O proof, to show $f(n)$ is $O\left(g(n)\right)$.

1. Find a c, n_0 that fit the definition for each of the terms of f.
   - Each of these is a mini, easier big-O proof.
2. Add up all your c, take the max of your n_0.
3. Add up all your inequalities to get the final inequality you want.
4. Clearly tell us what your c and n_0 are!

For any big-O proof, there are many c and n_0 that work.

You might be tempted to find the smallest possible c and n_0 that work.

You might be tempted to just choose $c = 1{,}000{,}000{,}000$ and $n_0 = 73{,}000{,}000$ for all the proofs.

Don't do either of those things.

A proof is designed to convince your reader that something is true. They should be able to easily verify every statement you make. – We don't care about the best c, just an easy-to-understand one.

We have to be able to see your logic at every step.

Big-O as an upper bound

True or False: $10n^2 + 15n$ is $O(n^3)$

$10n^2 \leq c \cdot n^3$ $when\ c = 10\ for\ n \geq 1$

$15n \leq c \cdot n^3$ $when\ c = 15\ for\ n \geq 1$

$10n^2 + 15n \leq 10n^3 + 15n^3 \leq 25n^3\ for\ n \geq 1$

$10n^2 + 15n$ is $O(n^3)$ because $10n^2 + 15n \leq 25n^3\ for\ n \geq 1$

Big-O is just an upper bound. It doesn't have to be a good upper bound

If we want the best upper bound, we'll ask you for a simplified, **tight** big-O bound.
O(n^2) is the tight bound for this example.

It is (almost always) technically correct to say your code runs in time O(n!).
DO NOT TRY TO PULL THIS TRICK IN AN INTERVIEW (or exam)!

Big-O is an upper-bound, not a fit

True or False: $10n^2 + 15n$ is $O(n^3)$

It's true – it fits the definition

$10n^2 \leq c \cdot n^3$ $when\ c = 10\ for\ n \geq 1$

$15n \leq c \cdot n^3$ $when\ c = 15\ for\ n \geq 1$

$10n^2 + 15n \leq 10n^3 + 15n^3 \leq 25n^3\ for\ n \geq 1$

$10n^2 + 15n$ is $O(n^3)$ because $10n^2 + 15n \leq 25n^3\ for\ n \geq 1$

Big-O is just an upper bound that may be loose and not describe the function fully.
For example, all of the following are true:

$10n^2 + 15n$ is $O(n^3)$
$10n^2 + 15n$ is $O(n^4)$
$10n^2 + 15n$ is $O(n^5)$
$10n^2 + 15n$ is $O(n^n)$
$10n^2 + 15n$ is $O(n!)$ ... and so on

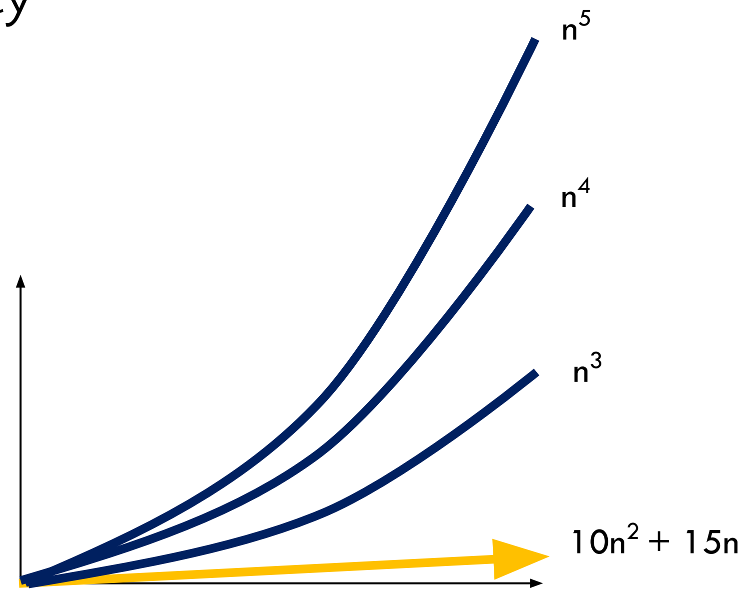
Big-O is an upper-bound, not a fit

What do we want to look for on a plot to determine if one function is in the big-O of the other?

You can sanity check that your g(n) function (the dominating one) overtakes or is equal to your f(n) function after some point and continues that greater-than-or-equal-to trend towards infinity

$10n^2 + 15n$ is $O(n^3)$
$10n^2 + 15n$ is $O(n^4)$
$10n^2 + 15n$ is $O(n^5)$

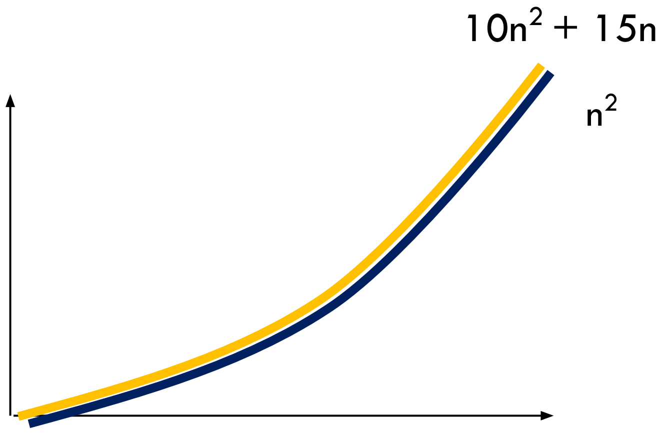
… and so on …



If we want the most-informative upper bound, we'll ask you for a simplified, tight big-O bound.

O(nˆ2) is the tight bound for the function f(n) = 10n2+15n. See the graph below – the tight big-O bound is the smallest upper bound within the definition of big-O.

If you zoom out a bunch, the your tight bound and your function will be overlapping compared to other complexity classes.



Questions?

Uncharted Waters: a different type of code model

Find a model f(n) for the running time of this code on input n. What's the Big-O?

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

+4

*n

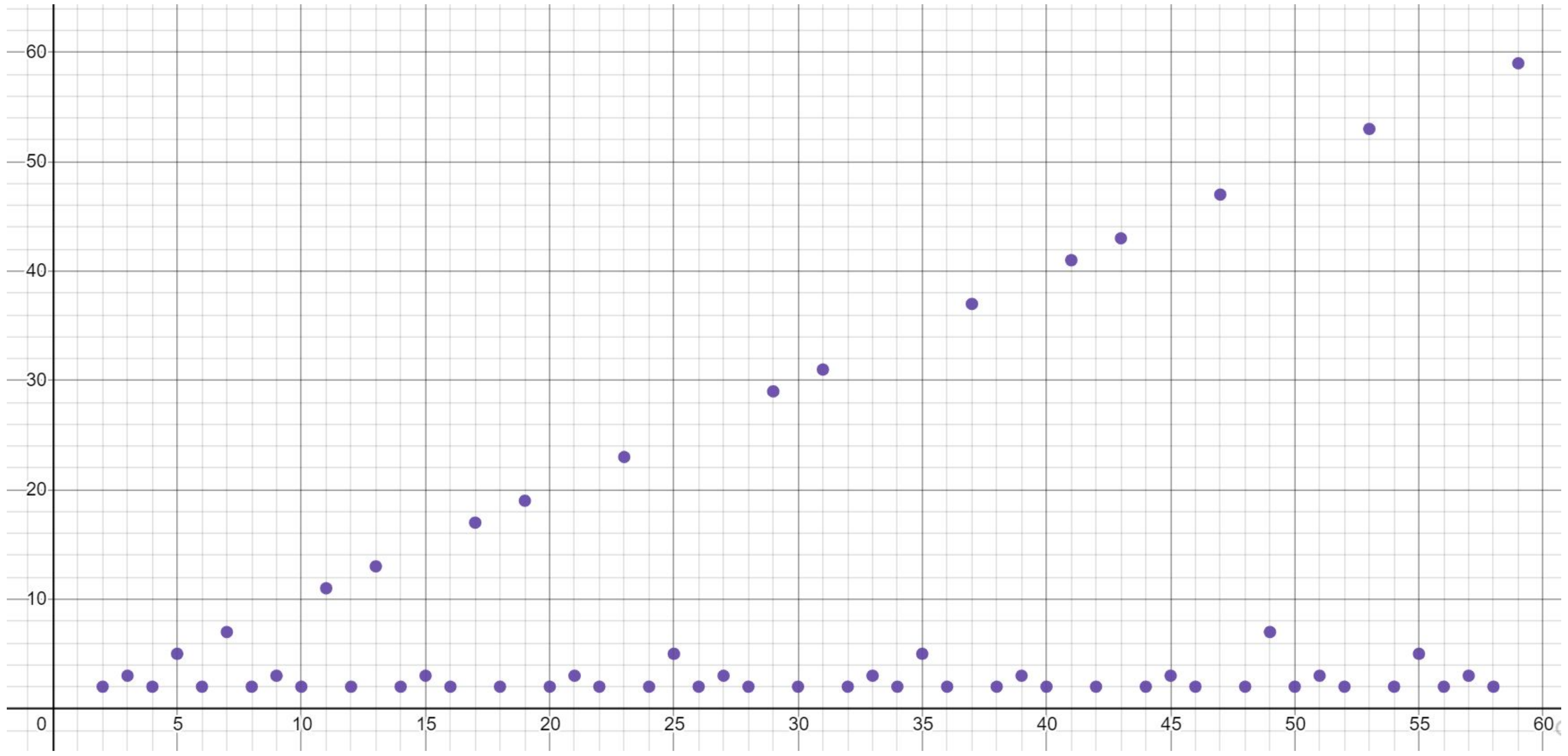
… sometimes?

ˆ we always pick the larger number to represent the slowest possible interpretation for Big O analysis

so even with the "sometimes n" loop we pick n to get a code model of:

$$f(n) = 6n+2$$

Prime Checking Runtime

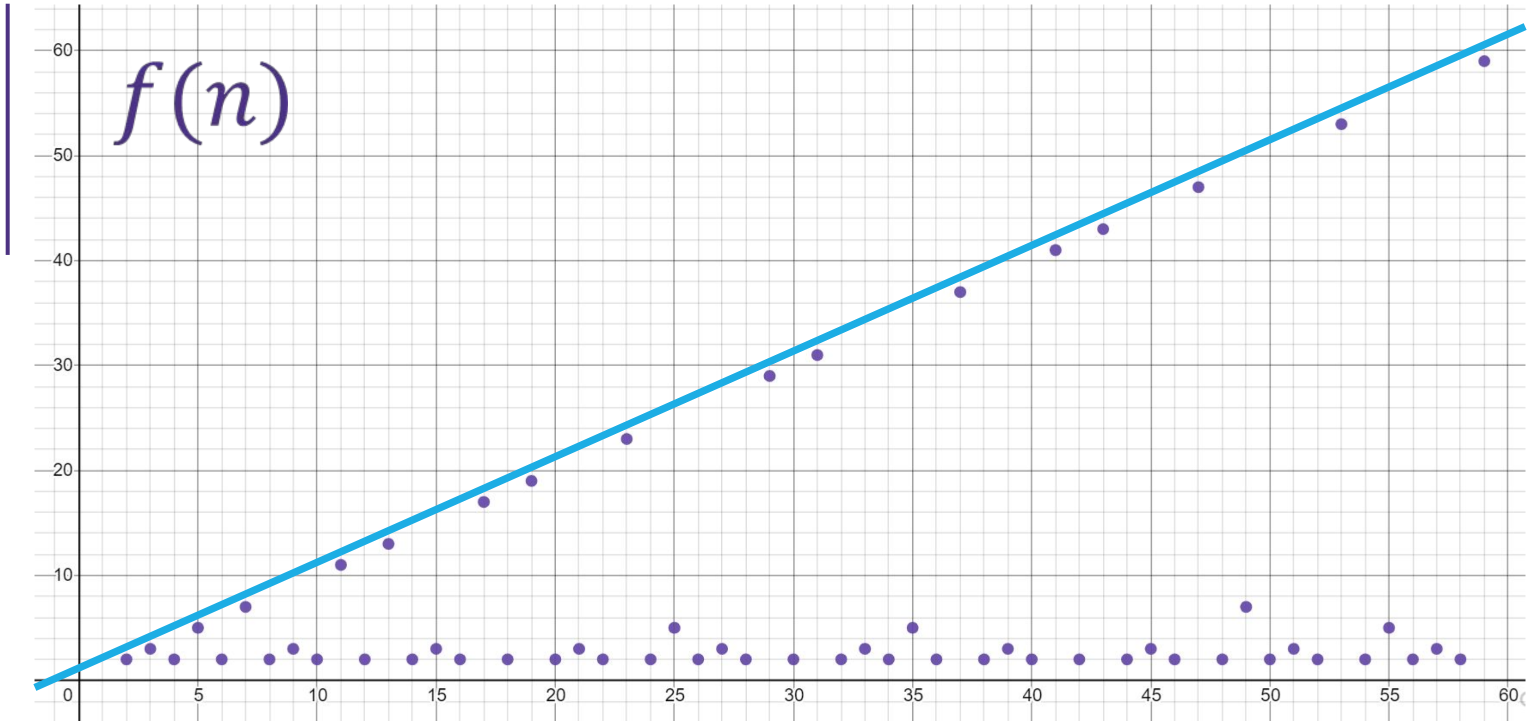


Is the running time of the code O(1) or O(n)?

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

But there's still always another number where the conde takes *n* iterations. So O(n)?

This is why we we define Big-O as the upper bound!



Big-O

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

Is the running time $O(n)$?
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 $n \geq 5$

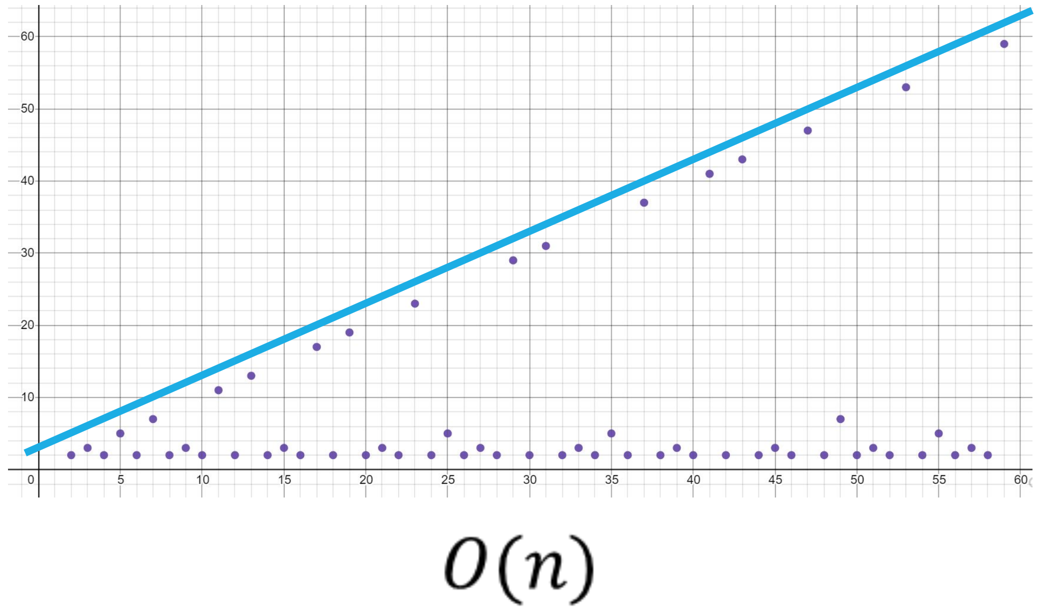
It's $O(n)$ but not $O(1)$

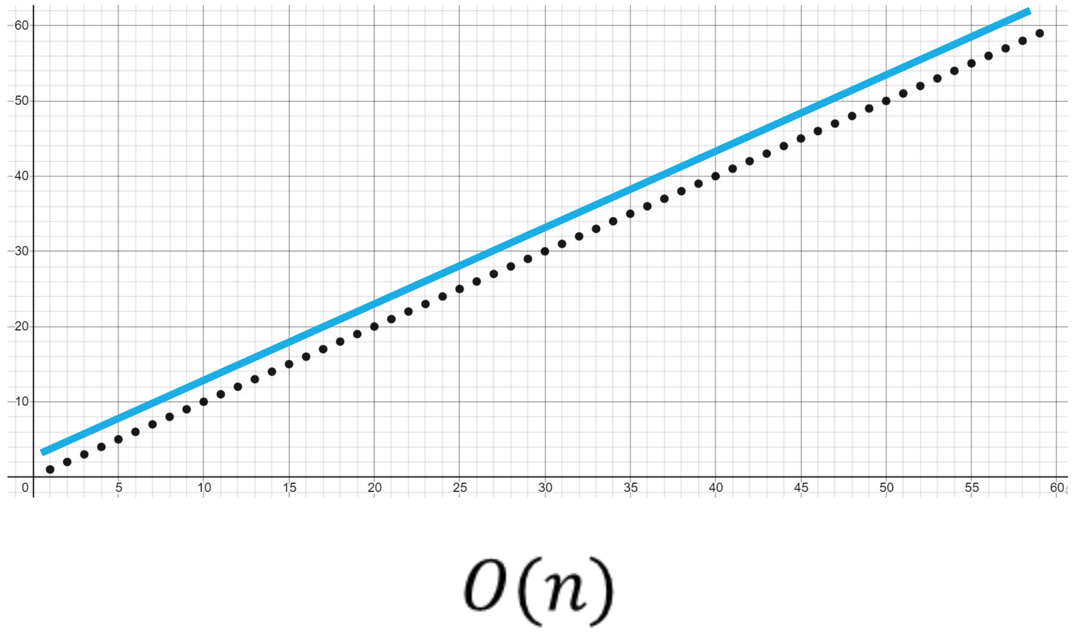
Is the running time $O(1)$?
Can you find constants c and n_0?

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-O isn't everything

Our prime finding code is O(n). But so is, for example, printing all the elements of a list.





Your experience running these two pieces of code is going to be very different.

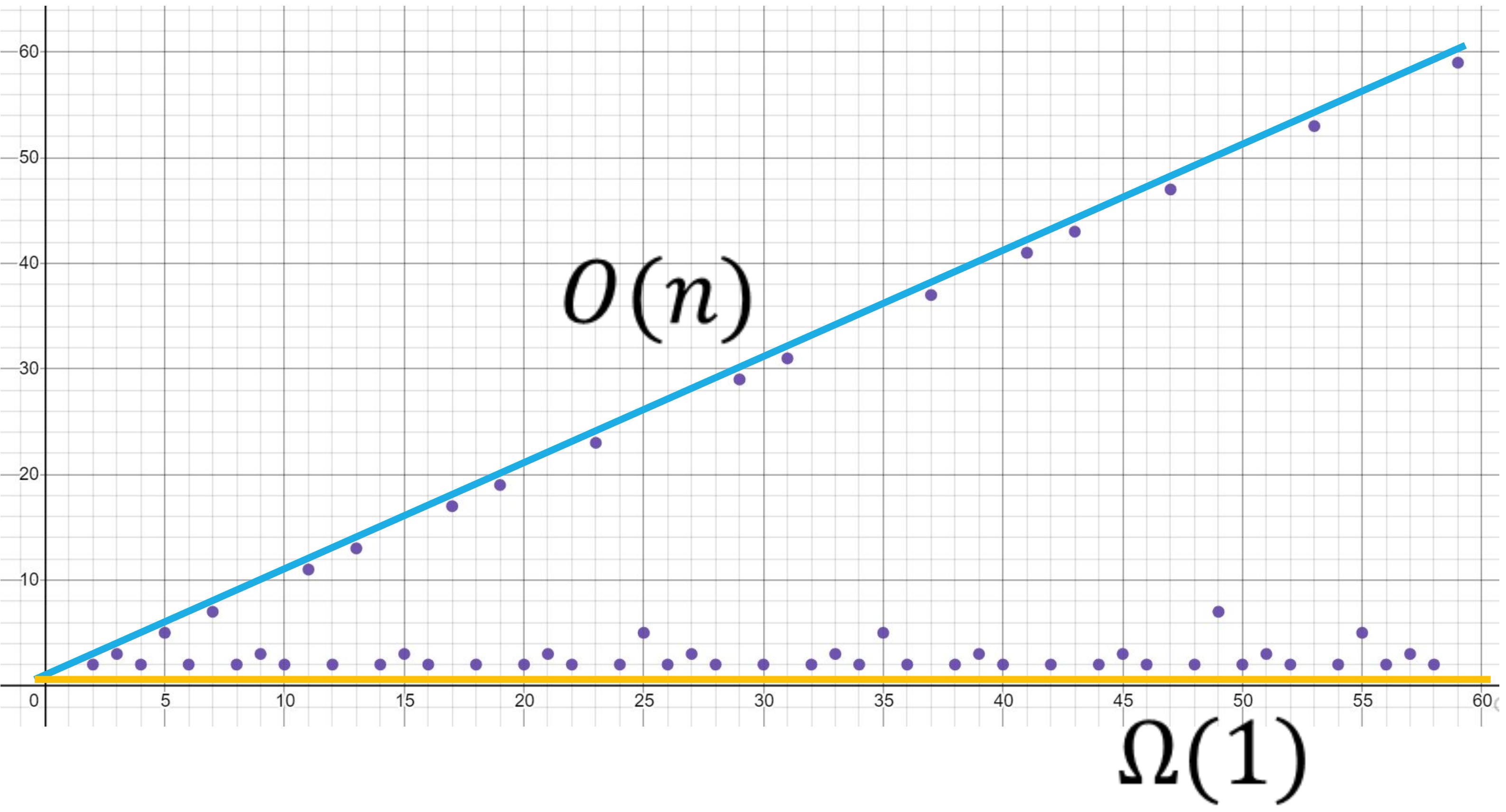
It's disappointing that the O() are the same – that's not very precise.

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 $n \geq n_0$, $f(n) \geq c \cdot g(n)$



Visually: what is the lower limit of this function?

What is bounded on the bottom by?

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

When we make Big-Oh statements about a function and say f(n) is O(g(n)) we're saying that f(n) grows at most as fast as g(n).

But with Big-Omega statements like f(n) is Ω(g(n)), we're saying that f(n) will grows at least as fast as g(n).

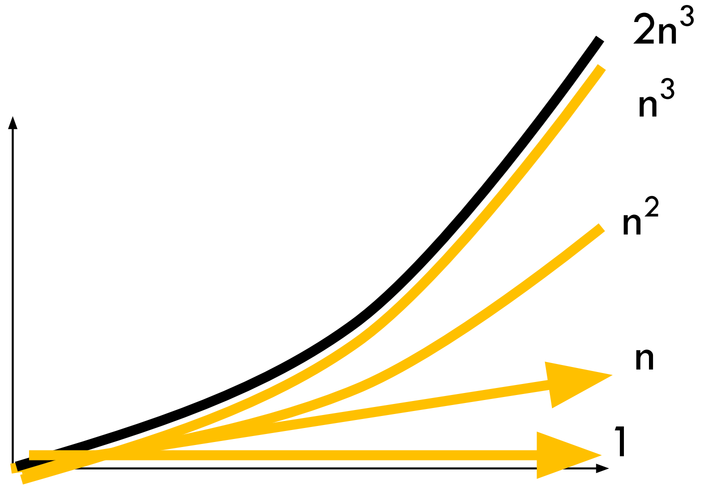
Big-Omega definition Plots

$2n^3$ is $\Omega(1)$

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

$2n^3$ is $\Omega(n^2)$

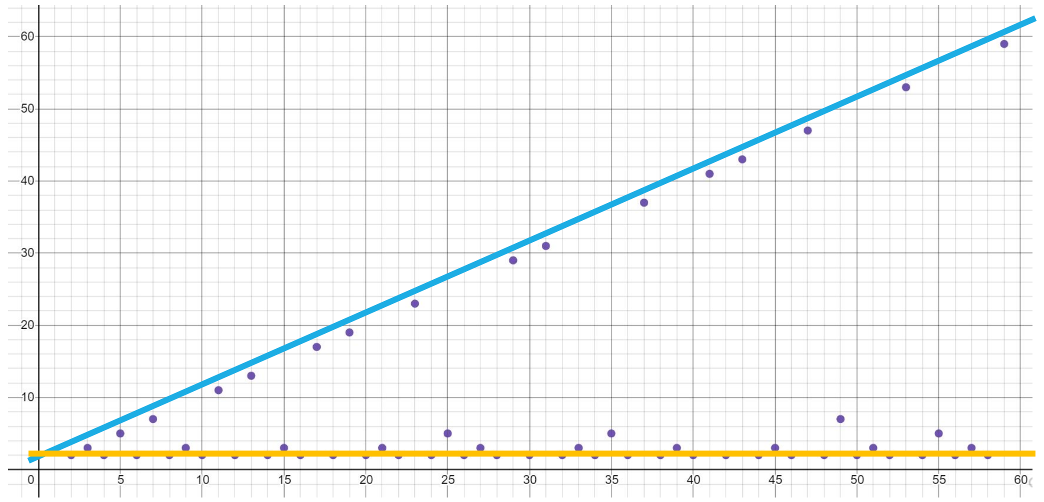
$2n^3$ is $\Omega(n^3)$



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

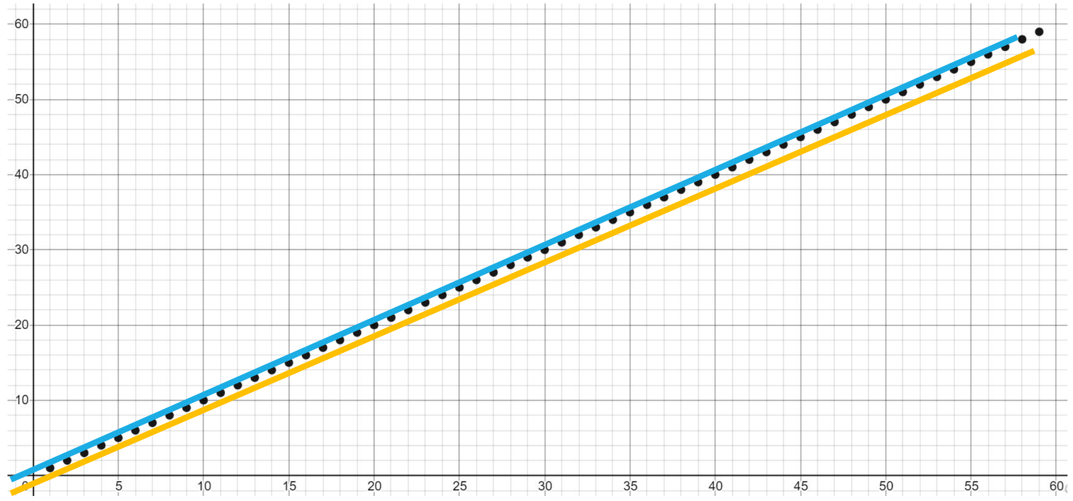
Big-O and Big-Ω shown together

prime runtime function



$O(n)$ $\Omega(1)$

f(n) = n



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

Note: this right graph's tight O bound is O(n) and its tight Omega bound is Omega(n). This is what most of the functions we'll deal with will look like, but there exists some code that would produce runtime functions like on the left.

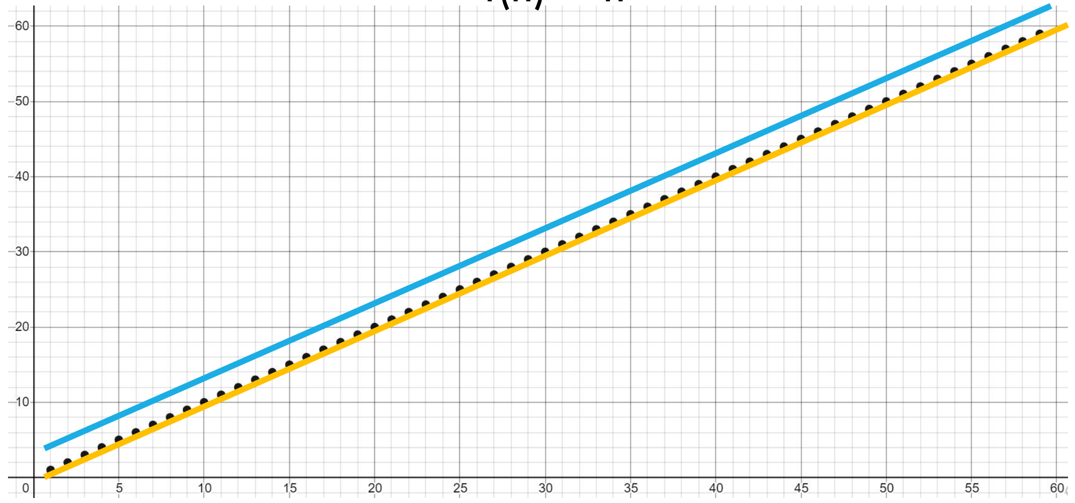
Big-Theta

Big Theta is **"equal to"**

- My code takes "exactly"* this long to run

  *Except for constant factors and lower order terms

f(n) = n



Big-Theta

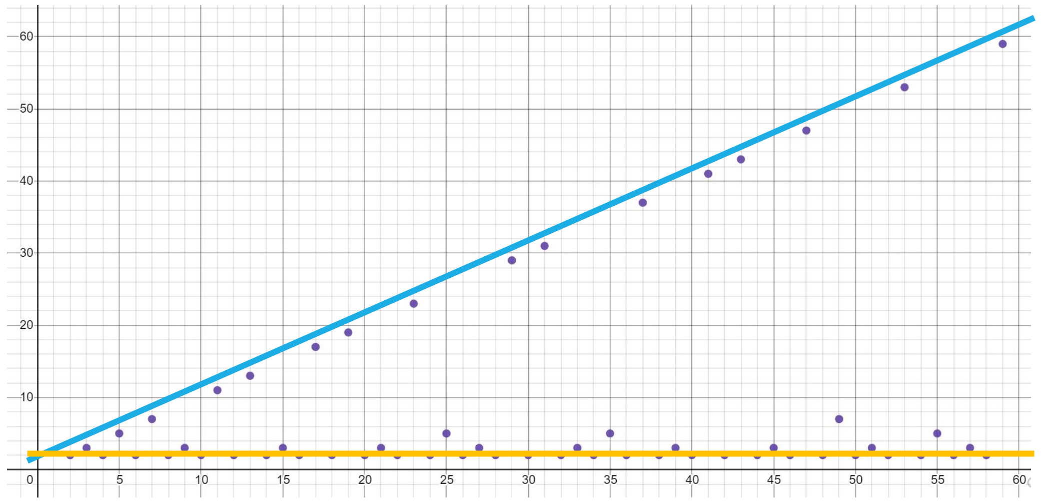
$f(n)$ is $\Theta(g(n))$ if
$f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.
In other words, there exist positive constants $c1$, $c2$, n_0 such that for all $n \geq n_0$
$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

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

Big-Theta

If the upper bound (BigO) and lower bound (Big Omega) are in different complexity classes, there is no fit so…

prime runtime function



$O(n)$ $\Omega(1)$

O, and Omega, and Theta [oh my?]

Big-O 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

Big-O

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

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 $n \geq n_0$
$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

Examples

$4n^2 \in \Omega(1)$

true

$4n^2 \in \Omega(n)$

true

$4n^2 \in \Omega(n^2)$

true

$4n^2 \in \Omega(n^3)$

false

$4n^2 \in \Omega(n^4)$

false

$4n^2 \in O(1)$

false

$4n^2 \in O(n)$

false

$4n^2 \in O(n^2)$

true

$4n^2 \in O(n^3)$

true

$4n^2 \in O(n^4)$

true

Big-O

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

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 $n \geq n_0$
$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

Simplified, tight big-O

In this course, we'll essentially use:

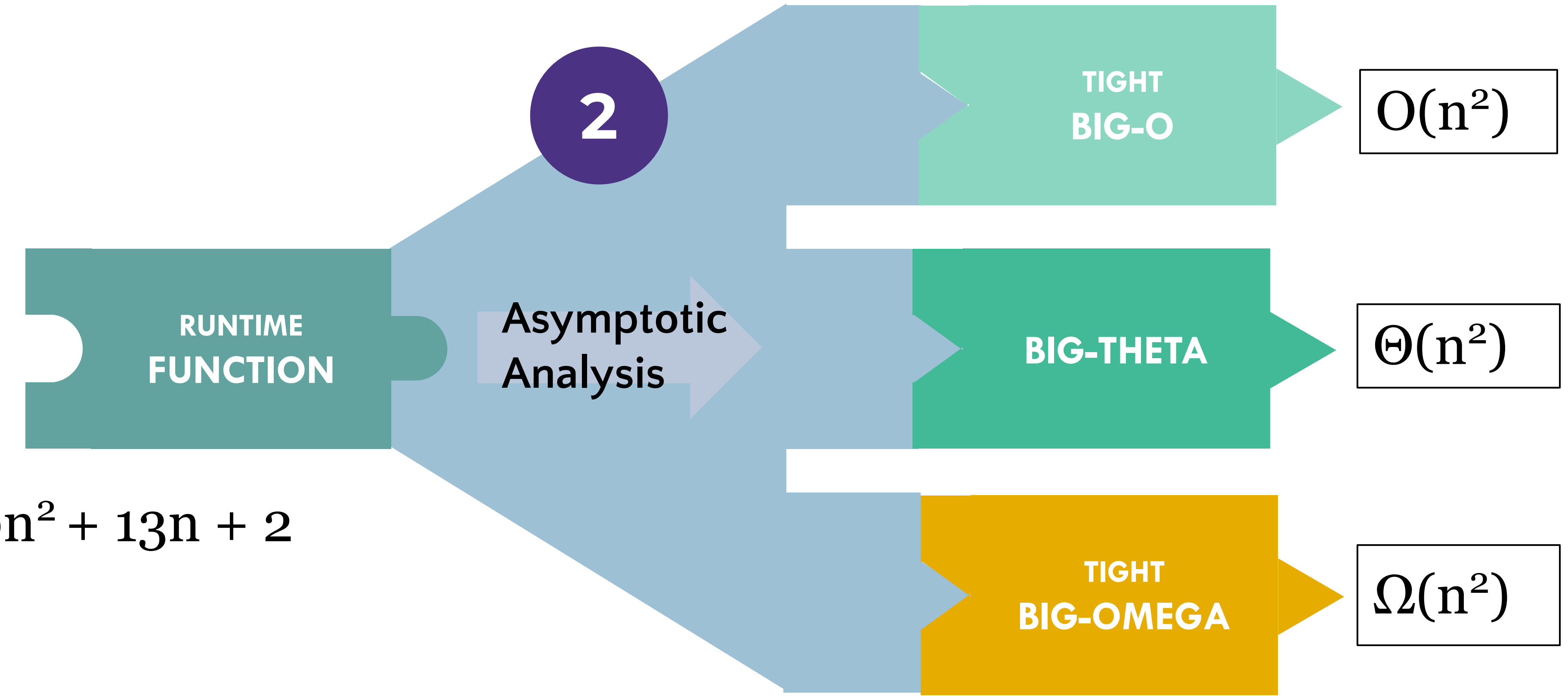
- Polynomials (n^c where c is a constant: e.g. $n, n^3, \sqrt{n}, 1$)
- Logarithms $\log n$
- Exponents (c^n where c is a constant: e.g. $2^n, 3^n$)
- Combinations of these (e.g. $\log(\log(n))$, $n \log n$, $\left(\log(\mathrm{n})\right)^2$)

For **this course**:

- A "tight big-O" is the slowest growing function among those listed.
- A "tight big-Ω" is the fastest growing function among those listed.
- (A Θ is always tight, because it's an "equal to" statement)
- A "simplified" big-O (or Omega or Theta)
  - Does not have any dominated terms.
  - Does not have any constant factors – just the combinations of those functions.

Questions?

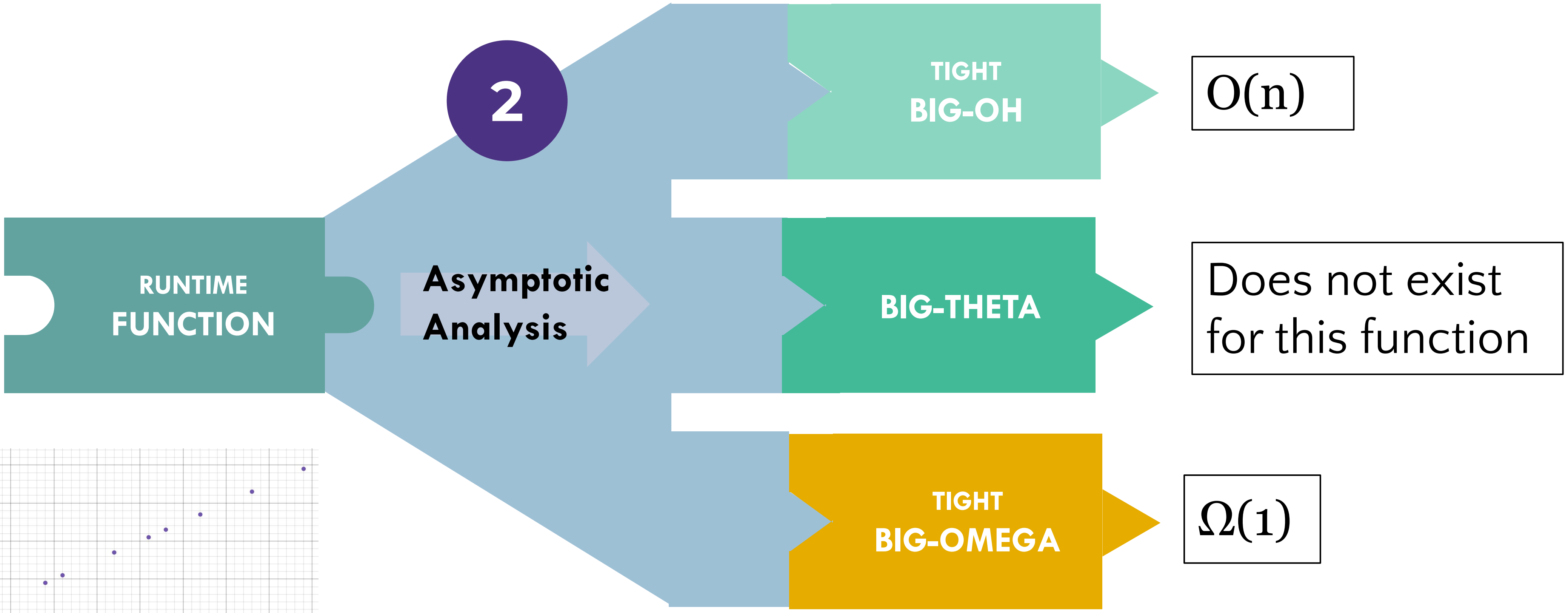
Our Upgraded Tool: Asymptotic Analysis

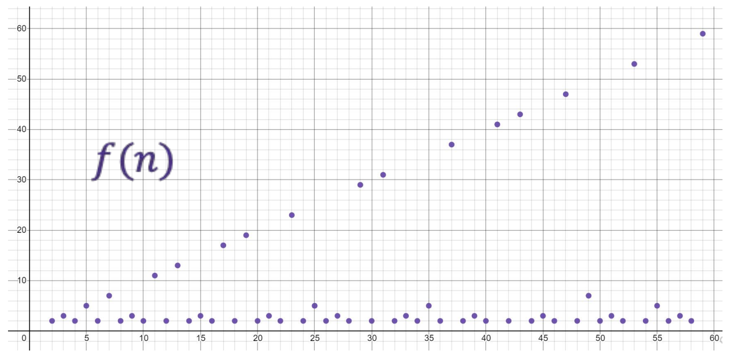


$f(n) = 10n^2 + 13n + 2$

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

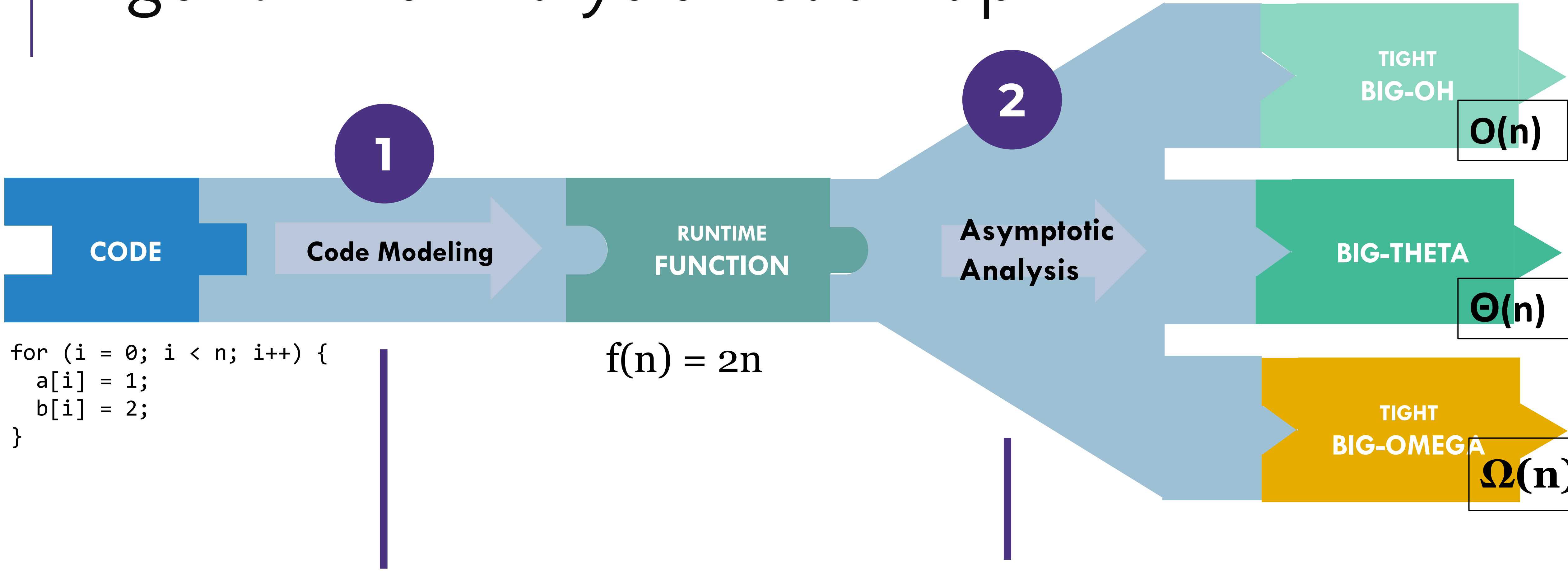




isPrime()

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;
}
```

$f(n) = 2n$

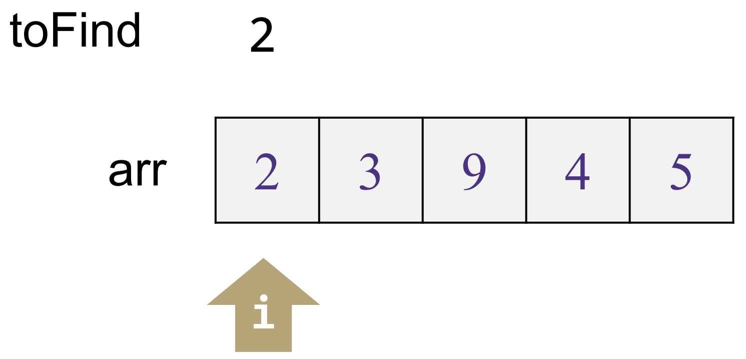
Now, let's look at this tool in more depth. How exactly are we coming up with that function?

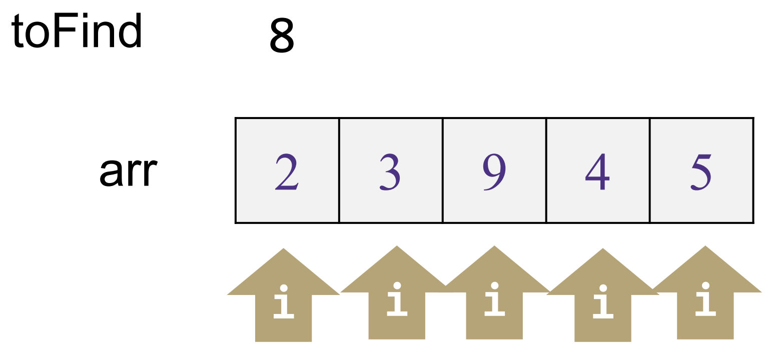
We just finished building this tool to characterize a function in terms of some useful bounds!

Case Analysis

Case Study: Linear Search

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





The number of operations doesn't depend just on n.
Even once you fix n (the size of the array) there are still a number of cases to consider.

- If `toFind` is in `arr[0]`, we'll only need one iteration, $f(n) = 4$.
- If `toFind` is not in `arr`, we'll need n iterations. $f(n) = 3n + 1$.
- And there are a bunch of cases in-between.

Best Case

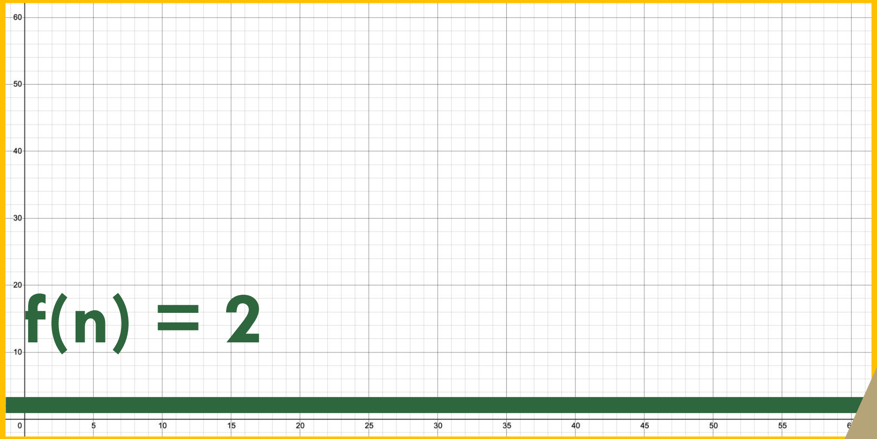
On Lucky Earth

toFind 2

arr

2	3	9	4	5

i

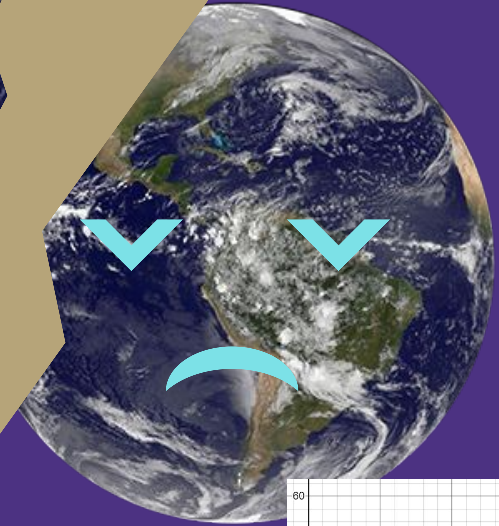


After asymptotic analysis:

O(1) Θ(1) Ω(1)

Worst Case

On Unlucky Earth (where it's 2020 every year)

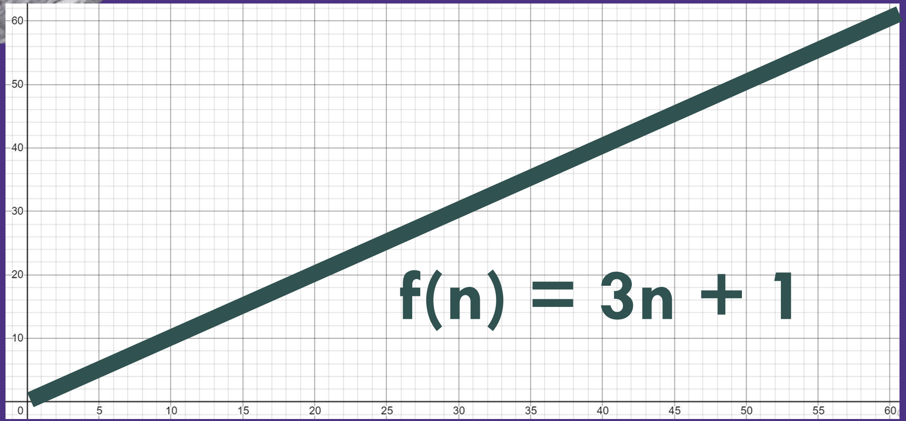


toFind 8

arr

2	3	9	4	5

i



After asymptotic analysis:

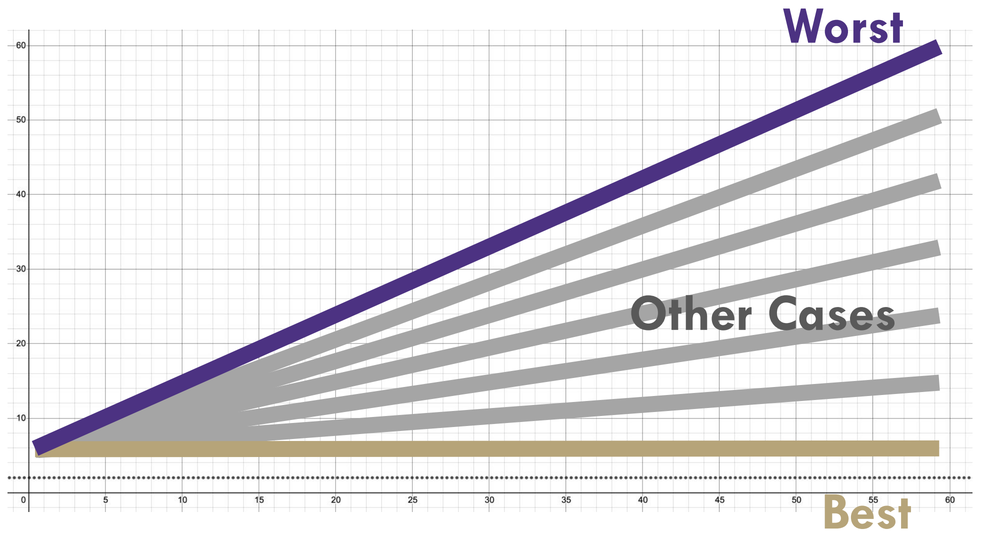
O(n) Θ(n) Ω(n)

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 O(n))
  - But intermediate cases will be important later



Caution: Common Misunderstanding

Best/Worst case is based on all variation **<u>other</u>** than value of n

Incorrect

"The best case is when n=1, worst is when n=infinity"

"The best case is when front is null"

"The best case is when overallRoot is null"

Correct

"The best case is when the node I'm looking for is at front, the worst is when it's not in the list"

"The best case is when the BST is perfectly balanced, the worst is when it's a single line of nodes"

Other cases

"Assume X won't happen case"

- Assume our array won't need to resize is the most common.

"Average case"

- Assume your input is random
- Need to specify what the possible inputs are and how likely they are.
- $f(n)$ is now the **average** number of steps on a **random** input of size n.

"In-practice case"

- This isn't a real term. (I just made it up)
- Make some reasonable assumptions about how the real-world is probably going to work
  - We'll tell you the assumptions, and won't ask you to come up with these assumptions on your own.
- Then do worst-case analysis under those assumptions.

All of these can be combined with any of O, Ω, and Θ!

How to do case analysis

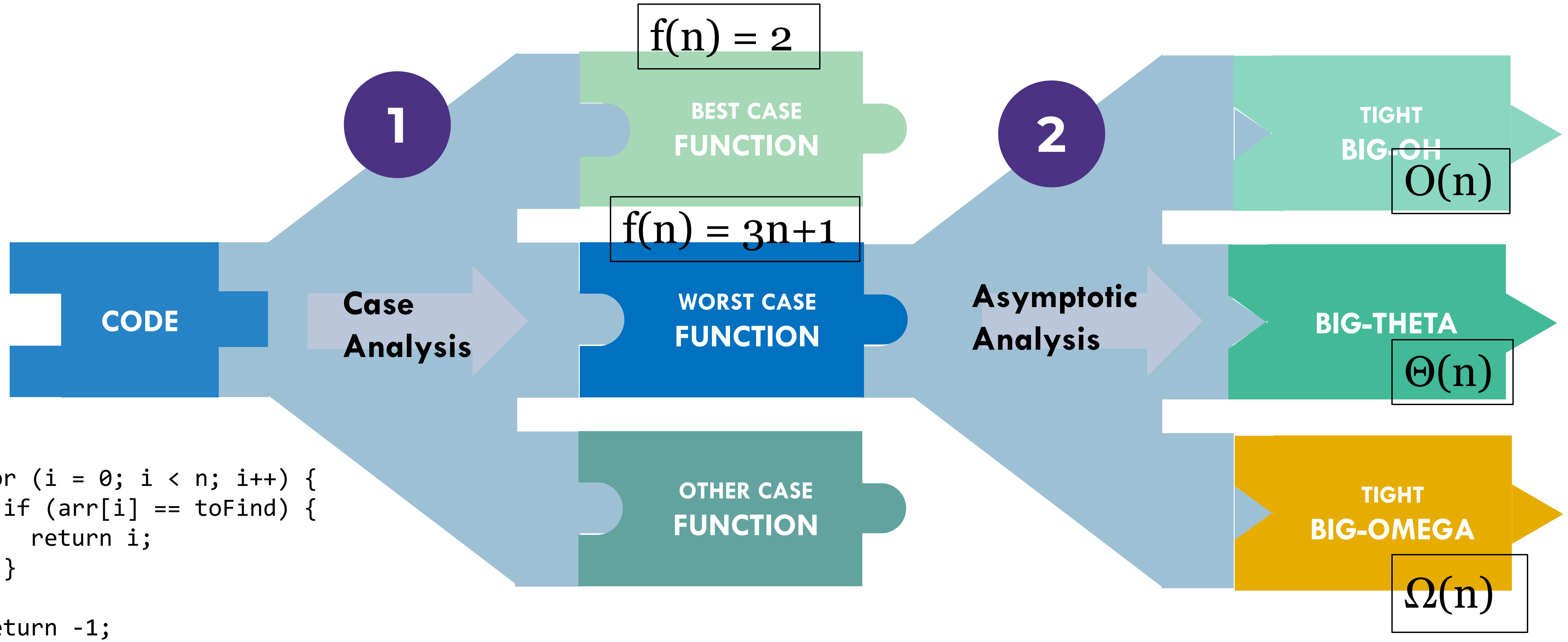
1. Look at the code, understand how thing could change depending on the input.
    - How can you exit loops early?
    - Can you return (exit the method) early?
    - Are some if/else branches much slower than others?

2. Figure out what input **values** can cause you to hit the (best/worst) parts of the code.

- not to be confused with <u>number</u> of inputs

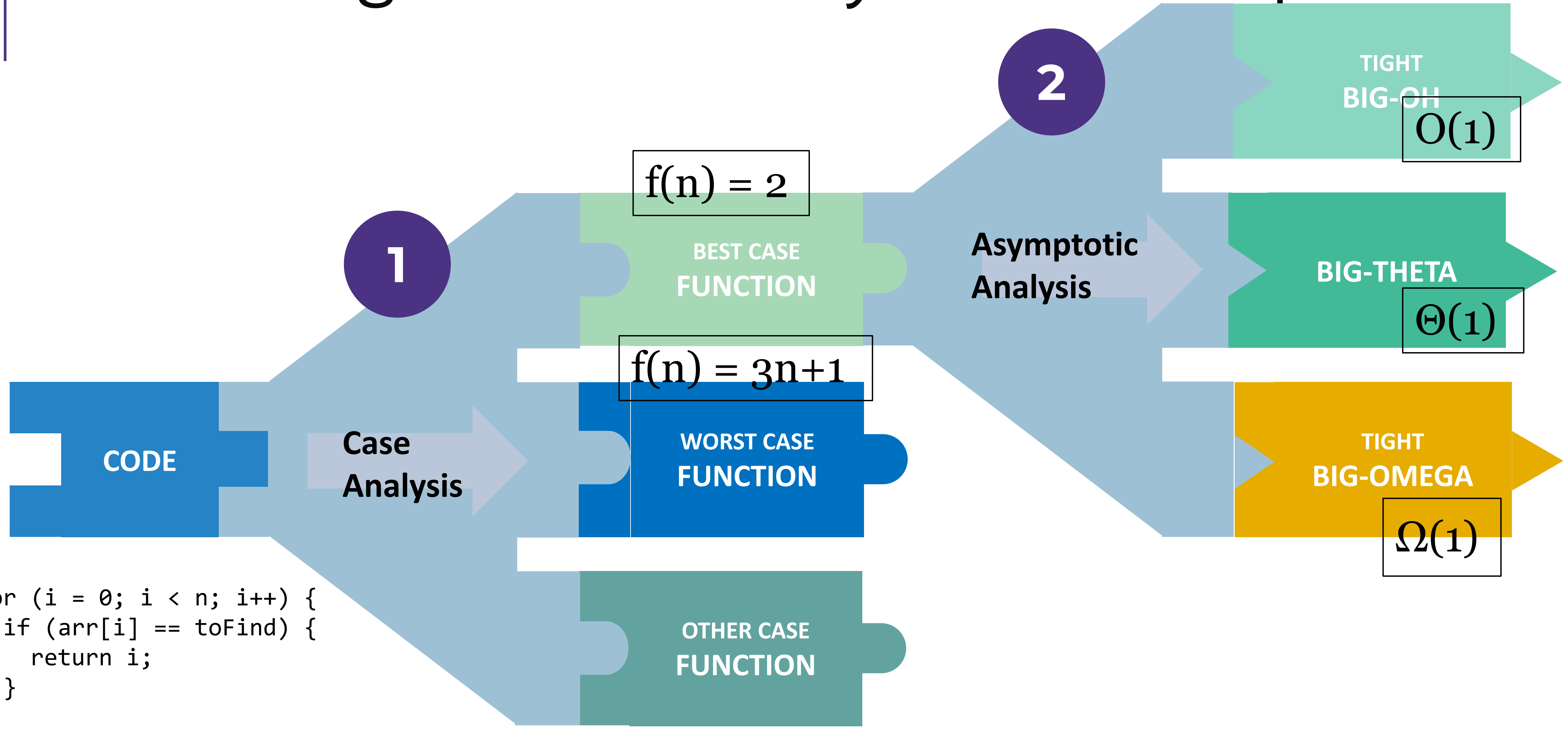
3. Now do the analysis like normal!

Algorithmic Analysis Roadmap



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

Review Algorithmic Analysis Roadmap

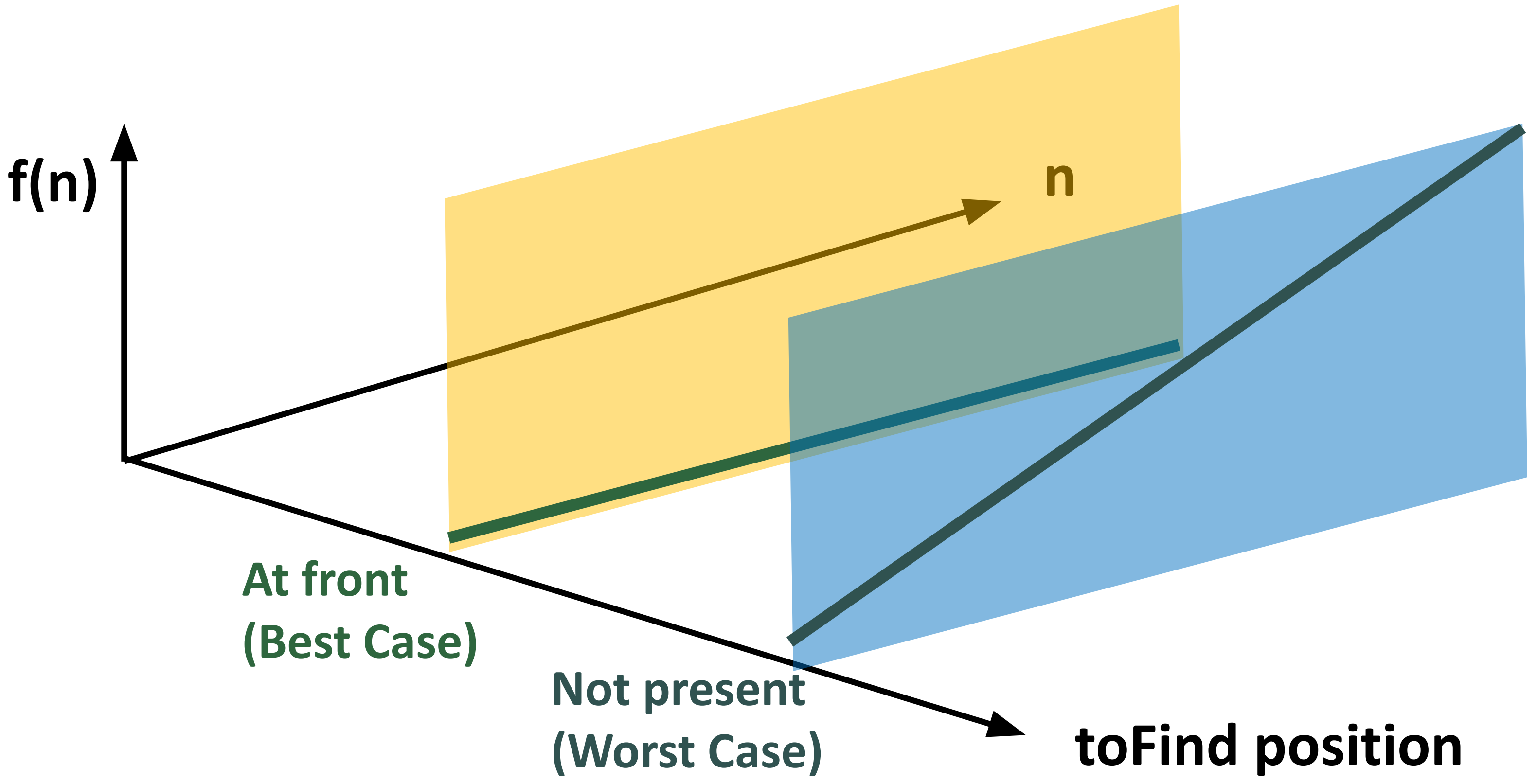


```
for (i = 0; i < n; i++) {
  if (arr[i] == toFind) {
    return i;
  }
}
return -1;
```

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


f(n)
n
At front
(Best Case)
Not present
(Worst Case)
toFind position