Please fill out the Poll at- pollev.com/21sp373

Lecture 6: Case Analysis

CSE 373: Data Structures and Algorithms

Warm Up!

Please fill out the Poll at- pollev.com/21sp373

Big-O

$f(n) \in O(g(n))$ 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

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

$f(n) \in \Theta(g(n))$ if
$f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

Which of the following is in $O(n^2)$? $\Omega(n^2)$? $\Theta(n^2)$?

a. $f(n) = 42$

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

b. $f(n) = 5n + 100$

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

c. $f(n) = nlog_2(3n)$

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

d. $f(n) = 4n^2 - 2n + 10$

$f(n) \in O(n^2)$ $f(n) \in \Omega(n^2)$ $f(n) \in \Theta(n^2)$

e. $f(n) = 2^n$

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

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.

Announcements

Proj 1 Due Wednesday April 14th

- Partner Project!
- Due Wednesday April 14th

Partners

- Yes, 3 person groups are allowed
- Default is working alone
- Define your own partnerships and groups via Gradescope
- We can assign you a random partner – respond by today

Kasey OH posted

- Wednesdays 11-1
- Thursdays 4-5:30
- Calendly for 1:1s
  - Wednesdays 4-5:30
  - Fridays 2-4

Lecture Questions Doc

P1 Deques

P1: Deques

Deque ADT: a double-ended queue

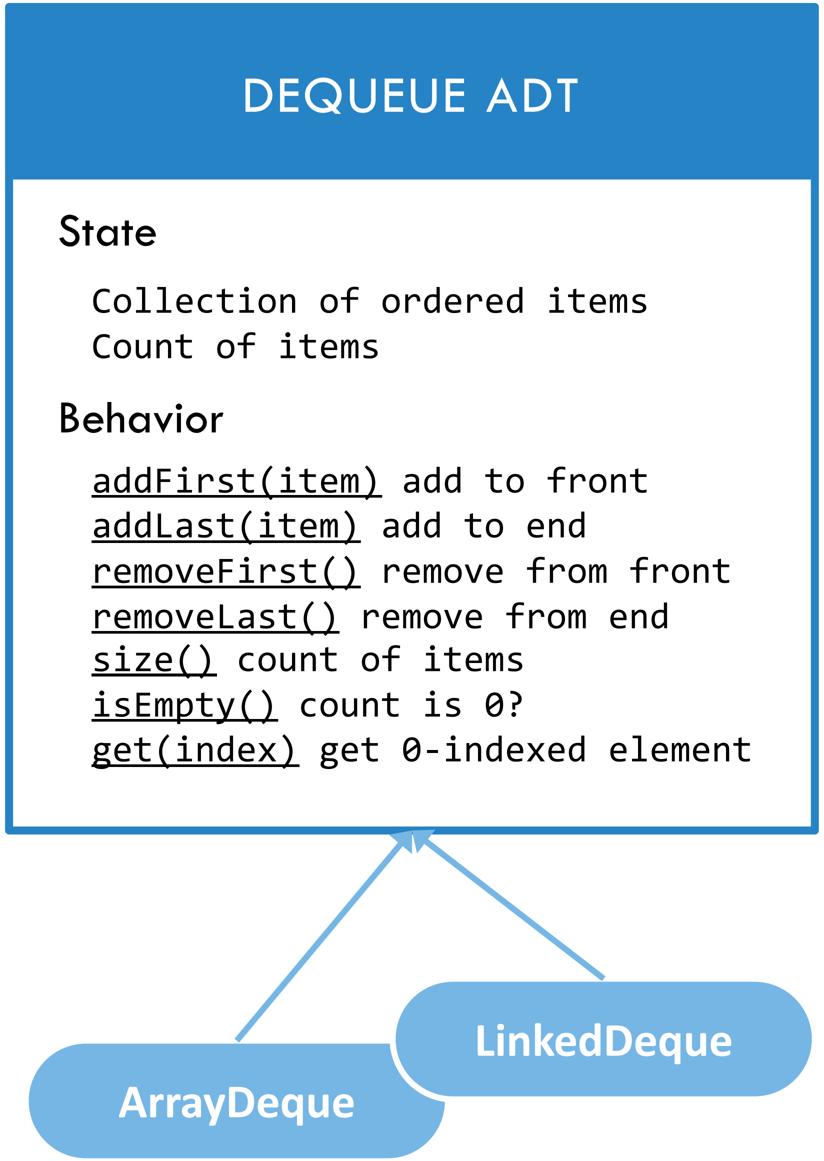
- Add/remove from both ends, get in middle

This project builds on ADTs vs. Data Structure Implementations, Queues, and a little bit of Asymptotic Analysis

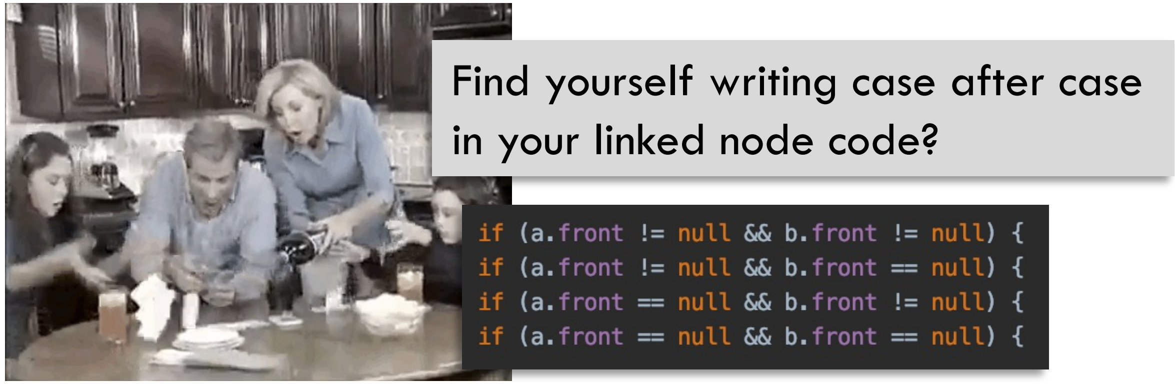
- Practice the techniques and analysis covered in LEC 02 & LEC 03!

3 components:

- Debug **`ArrayDeque`** implementation
- Implement **`LinkedDeque`**
- Run experiments



P1: Sentinel Nodes



Introducing

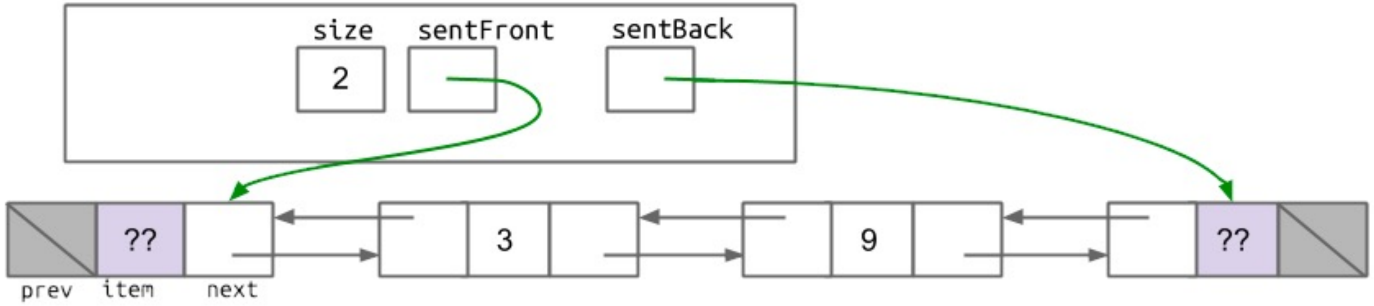
Sentinel Nodes

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

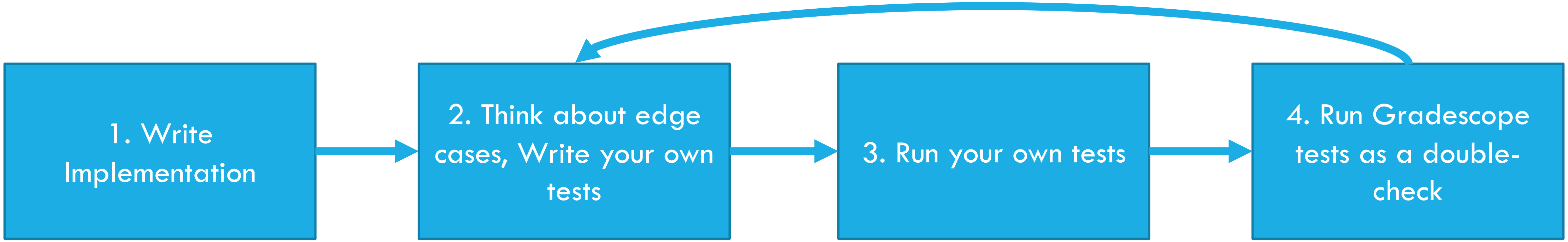
- Run & give feedback when you submit, but only give a general name!

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?"

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 large.

Big-O

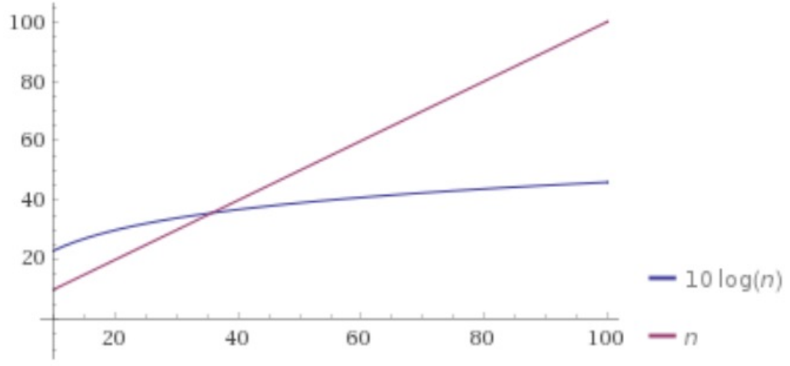
$f(n)$ is $O(g(n))$ if there exist positive constants c, n_0 such that for all $n \geq n_0$,

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

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

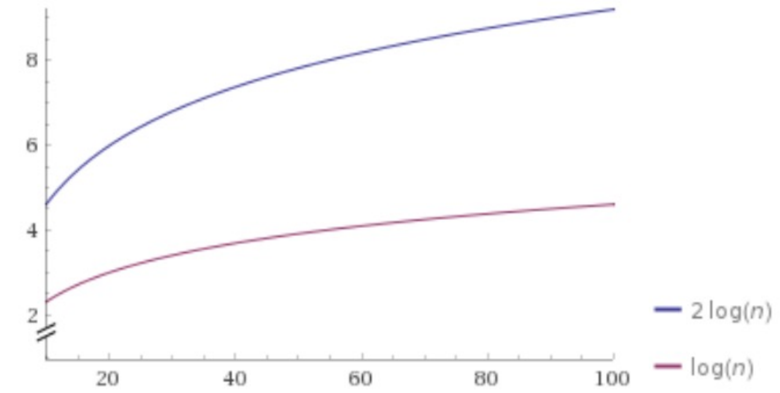
Why n_0?

Plot:



Why c?

Plot:



Note: Big-O definition is just an upper-bound, not always an exact bound

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

This is a big idea!

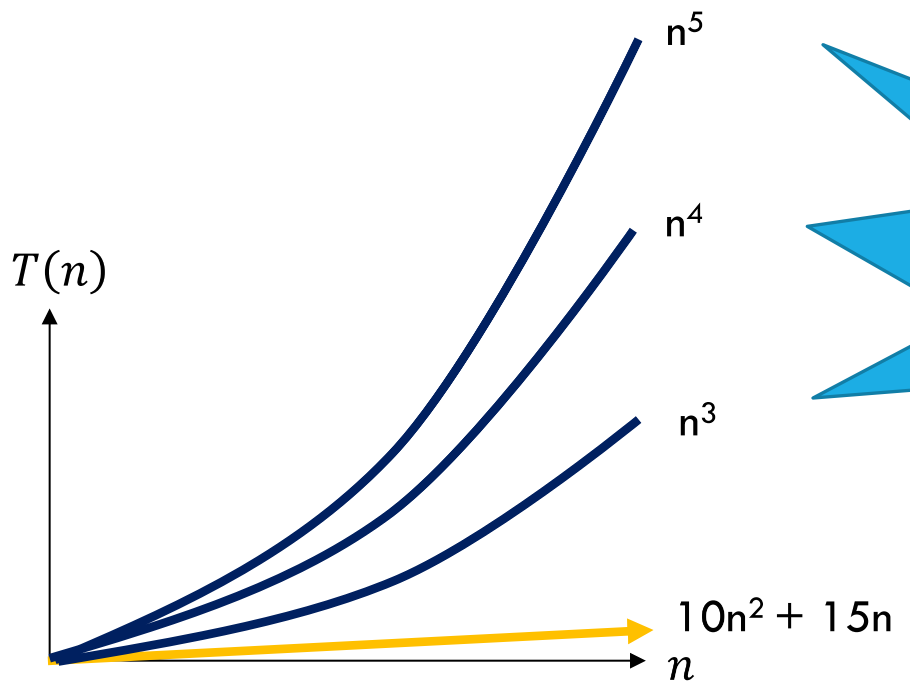
Note: Big-O definition is just an upper-bound, not always an exact bound (plots)

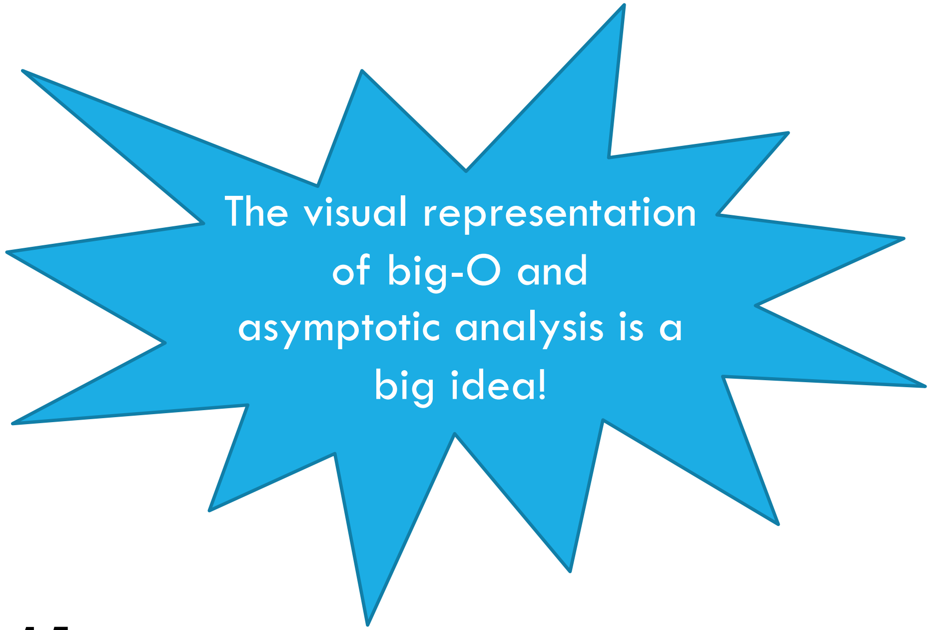
What do we want to look for on a plot to determine if one function is in the big-O of the other?

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 …





Tight Big-O Definition Plots

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

$O(n^2)$ is the tight bound for the function f(n) = $10n^2+15n$. See the graph below – the tight big-O bound is the smallest upperbound within the definition of big-O.

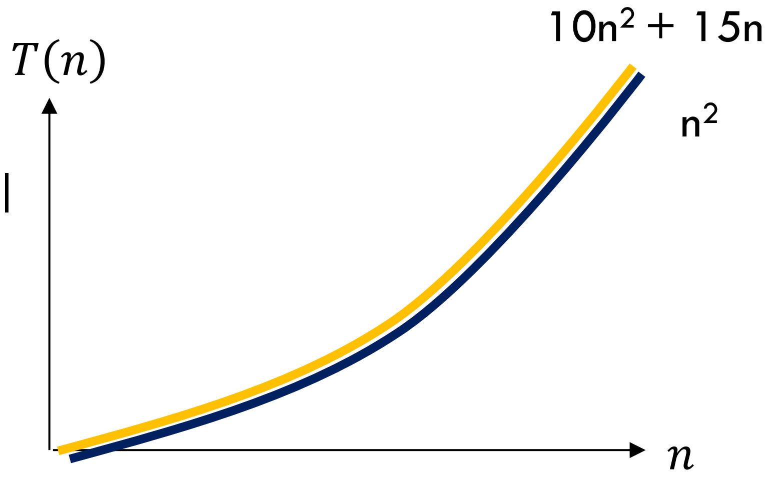
Computer scientists It is almost always technically correct to say your code runs in time $O(n!)$. (Warning: don't try this trick in an interview or exam)

If you zoom out a bunch,

the your tight bound and your function will

be overlapping compared to other

complexity classes.



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;
    while(toTest < n){
        if(toTest % n == 0) {
            return true;
        } else {
            toTest++;
        }
    }
    return false;
}
```

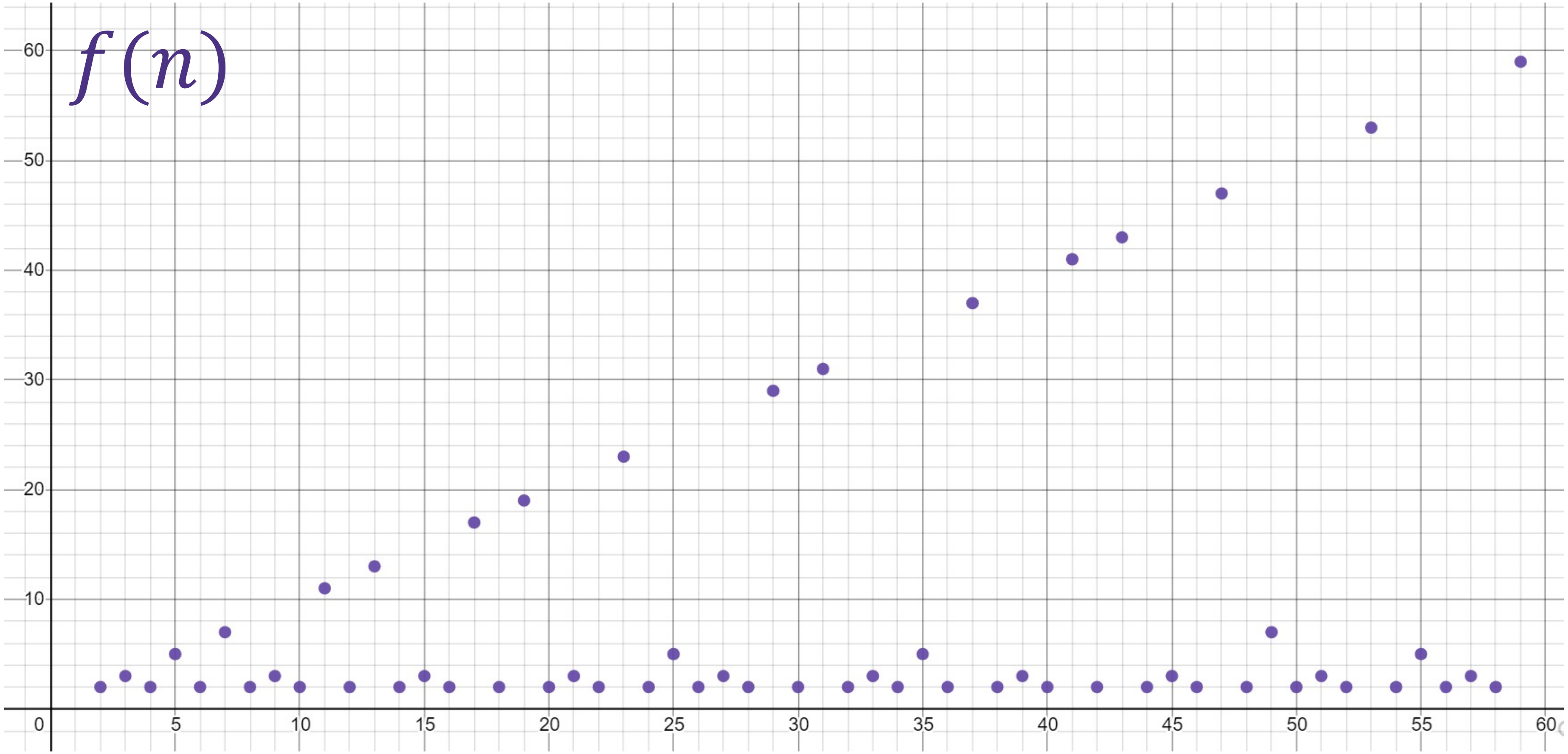
Remember, $f(n)$ = the number of basic operations performed on the input n.

Operations per iteration: let's just call it 1 to keep all the future slides simpler.

Number of iterations?

- Smallest divisor of n

Prime Checking Runtime



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

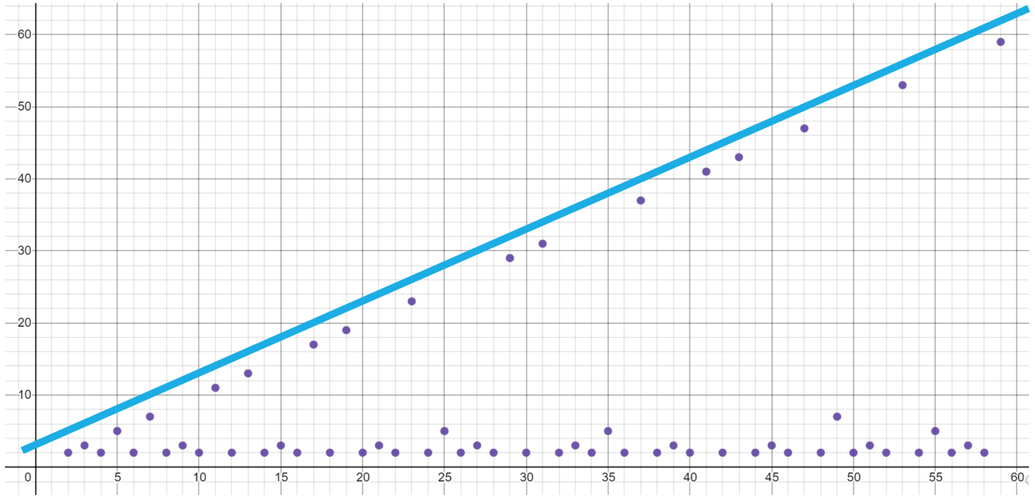
More than half the time we need 3 or fewer iterations. Is it $O(1)$?

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

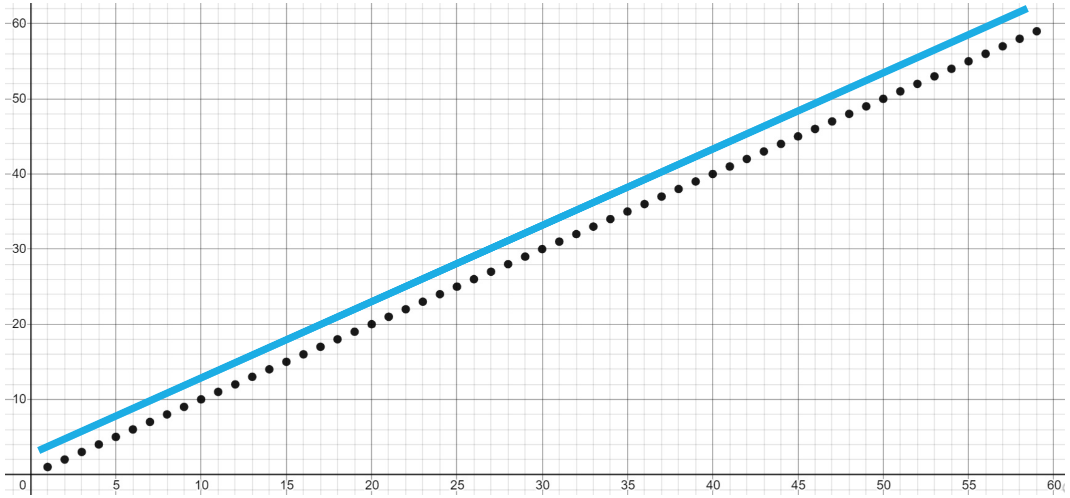
This is why we have definitions!

Big-O isn't everything

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



$O(n)$



$O(n)$

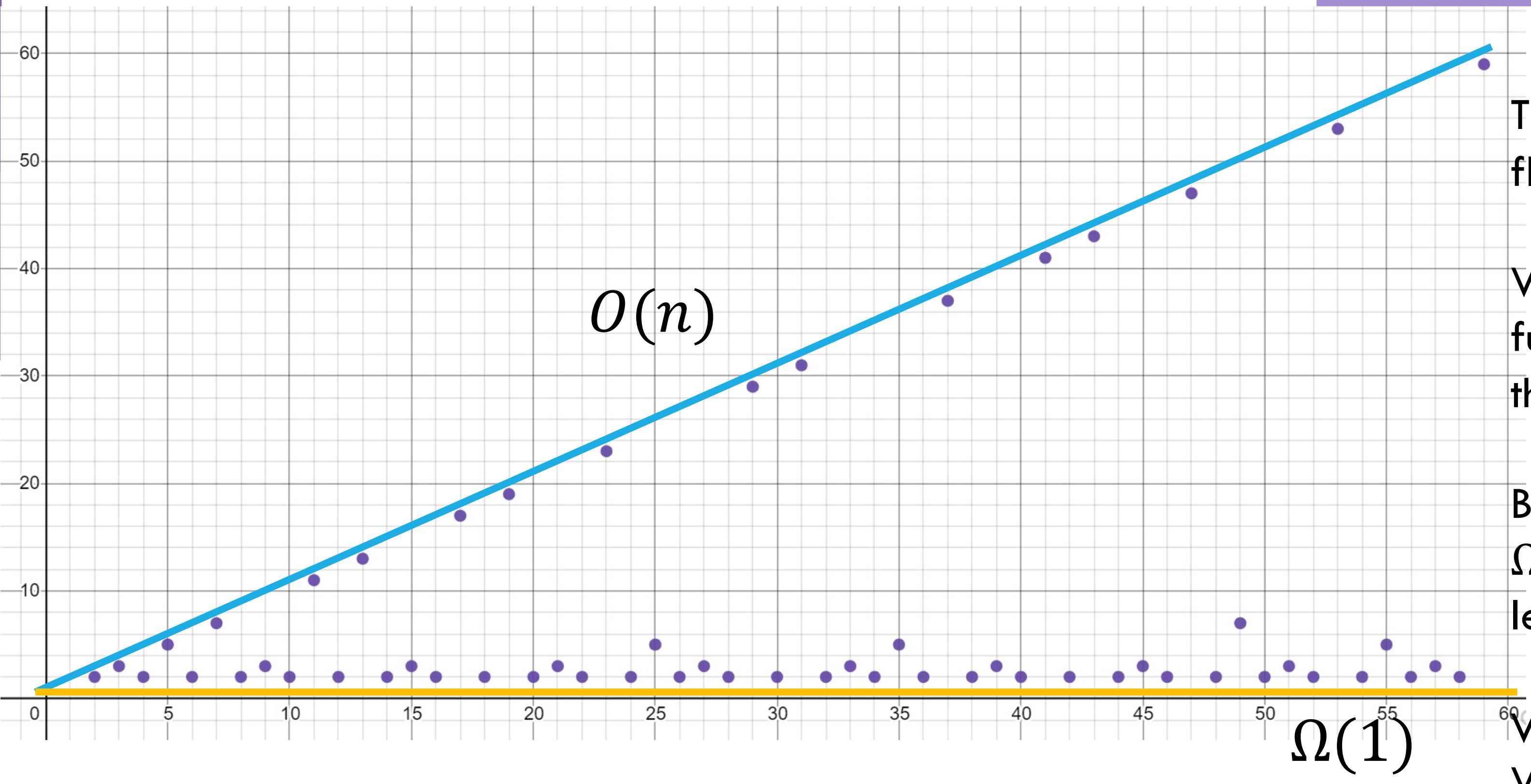
Your experience running these two pieces of code is going to be very different.
It's disappointing that the $O()$ are the same – that's not very precise.
Could we have some way of pointing out the list code always takes AT LEAST n operations?

Big-Ω [Omega]

Big-Omega

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

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



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).

Visually: what is the lower limit of this function? What is bounded on the bottom by?

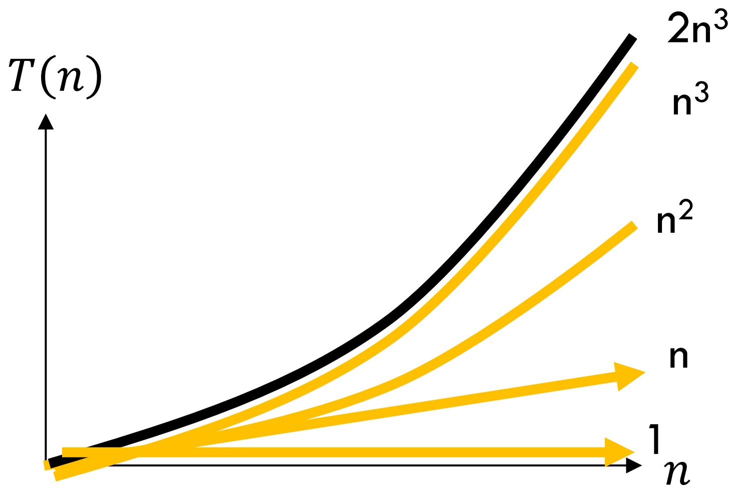
Big-Omega definition Plots

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

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

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

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



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

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 $\Omega(g(n))$.

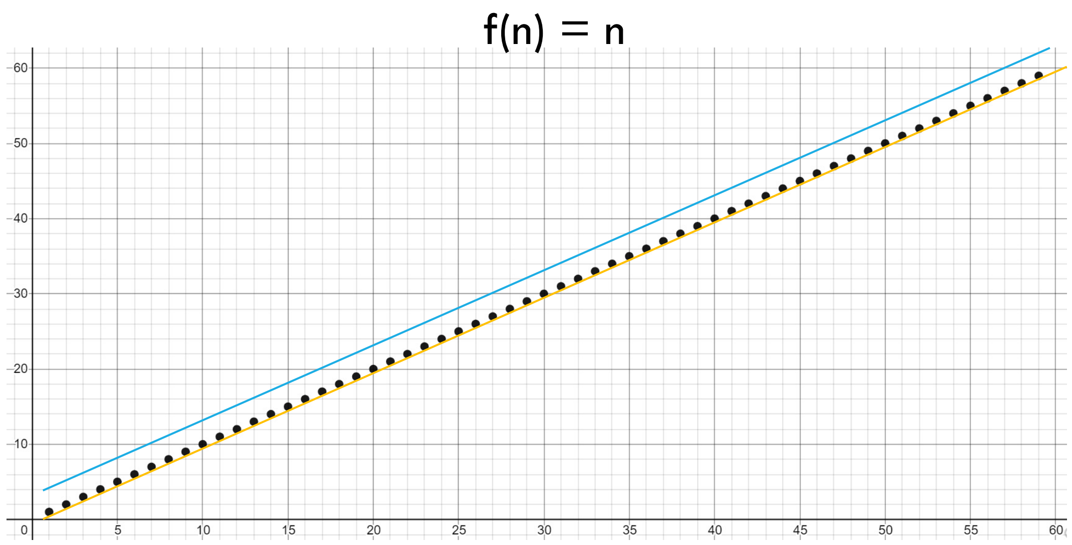
(in other words: there exist positive constants $c1, c2, n_0$ such that for all $n \geq n_0$)

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

O, 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 term



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

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 $c1, c2, n_0$ 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)

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

$f(n) \in O(g(n))$ 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

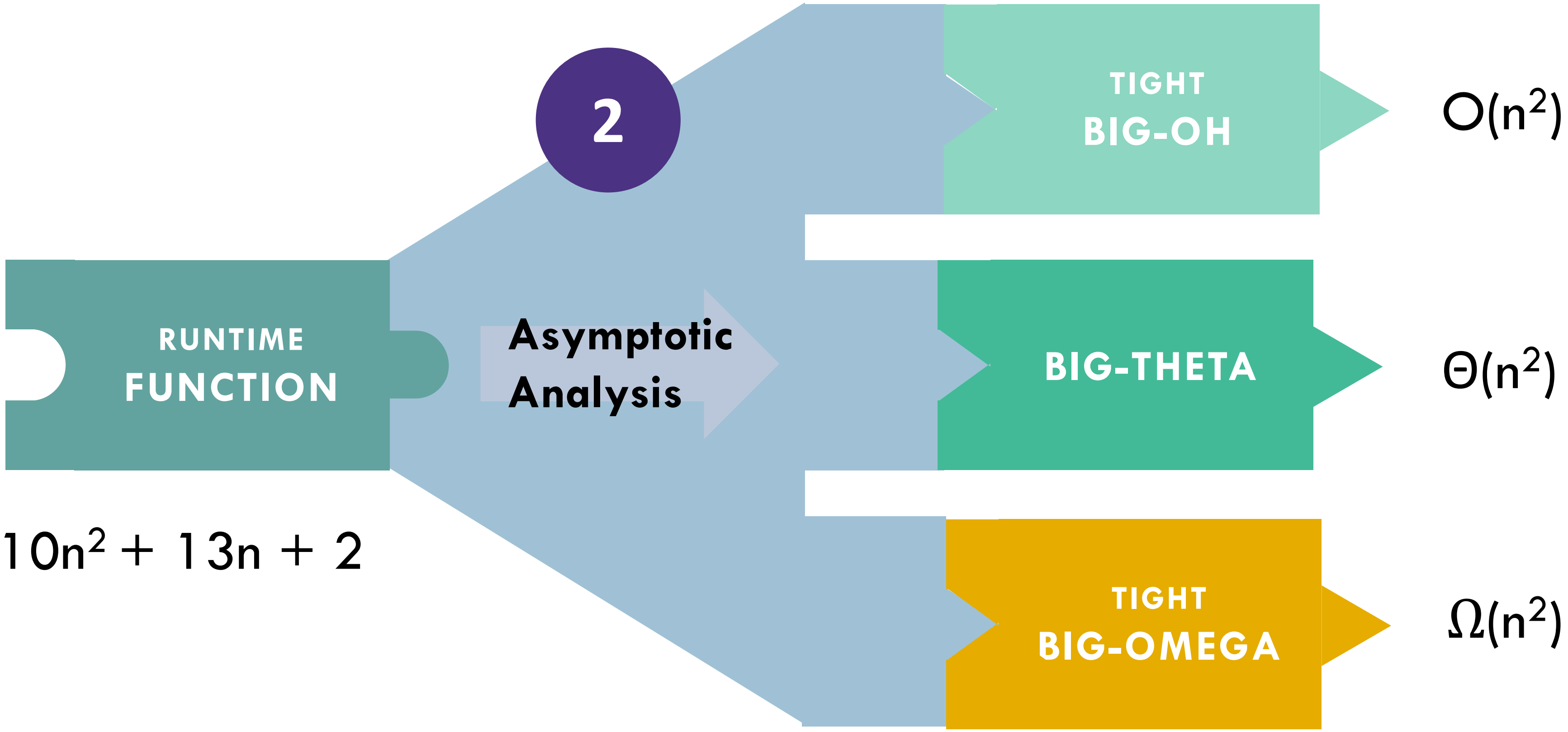
$f(n) \in \Omega(g(n))$ if there exist positive constants c, n_0 such that for all $n \geq n_0$,

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

Big-Theta

$f(n) \in \Theta(g(n))$ if
$f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

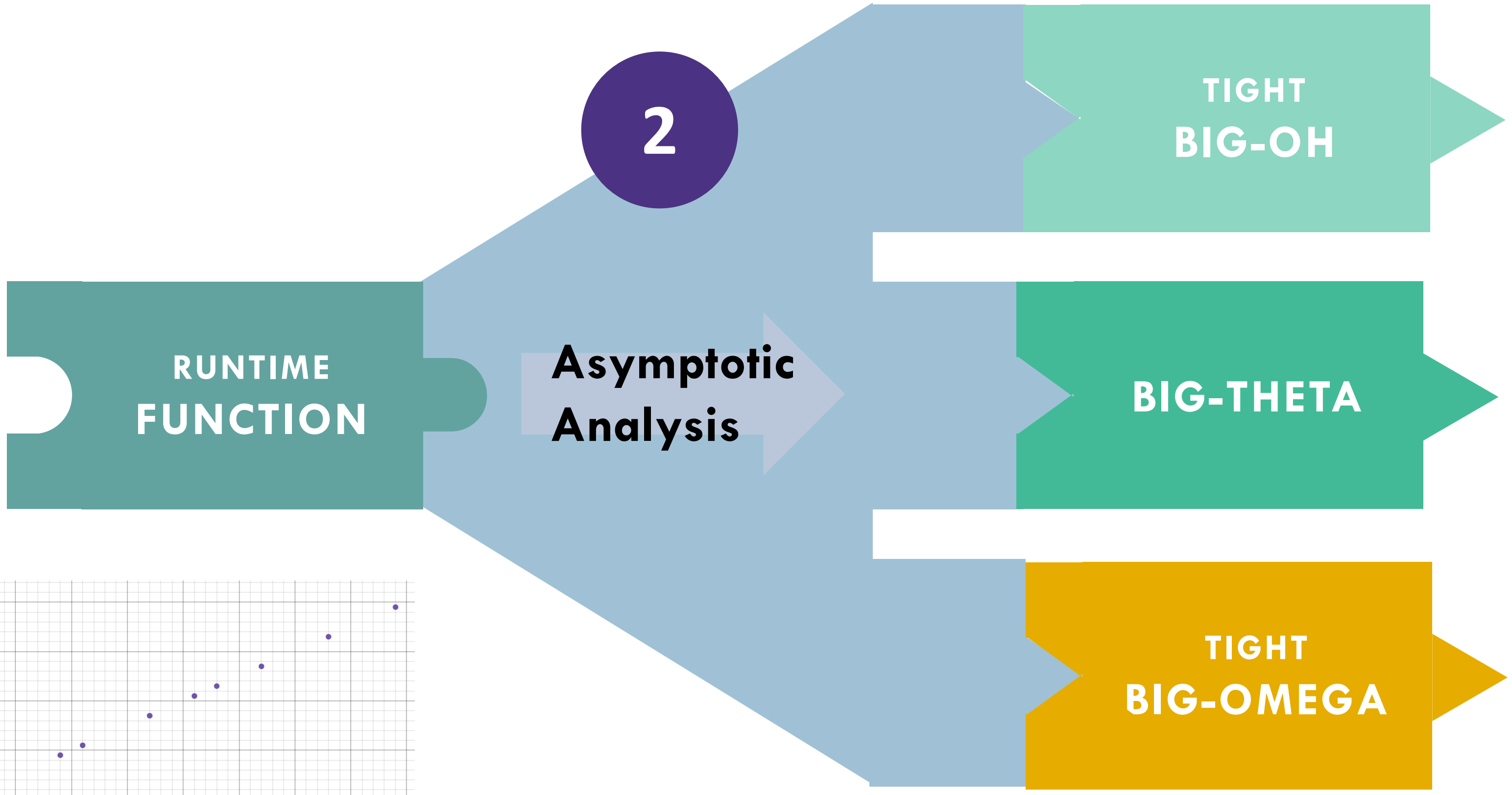
Our Upgraded Tool: Asymptotic Analysis



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

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.

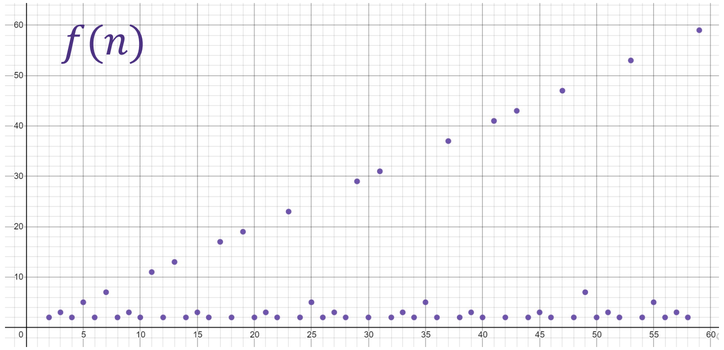
Our Upgraded Tool: Asymptotic Analysis



O(n)

Does not exist for this function

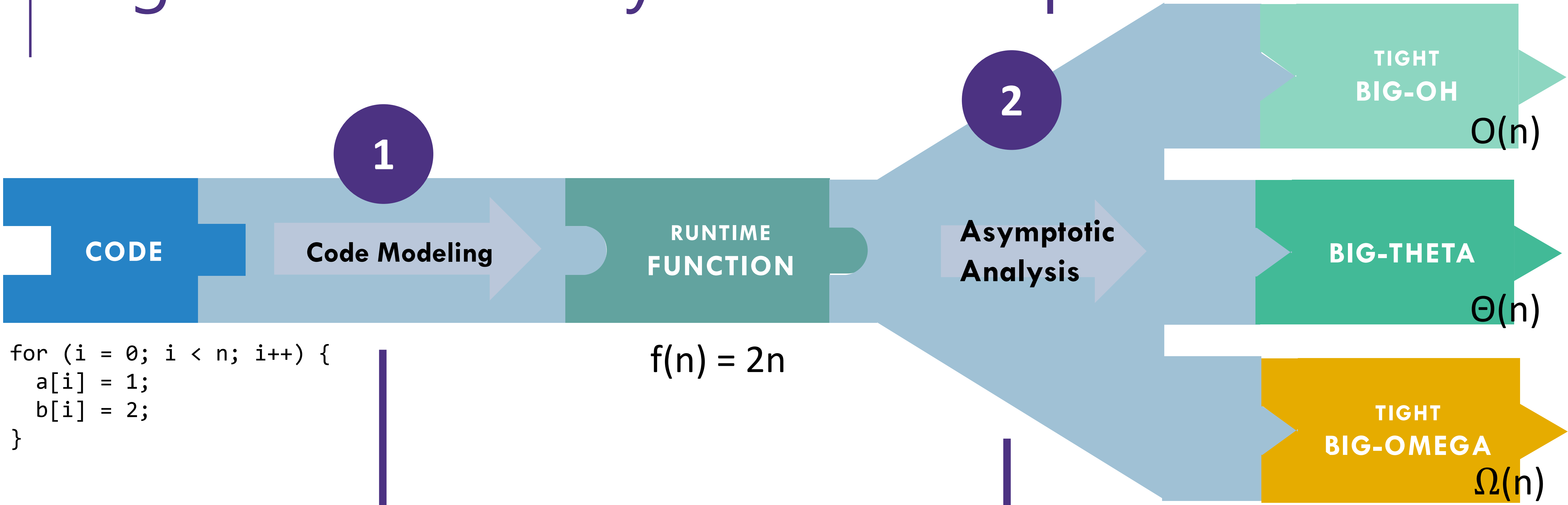
Ω(1)



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!

Case Analysis

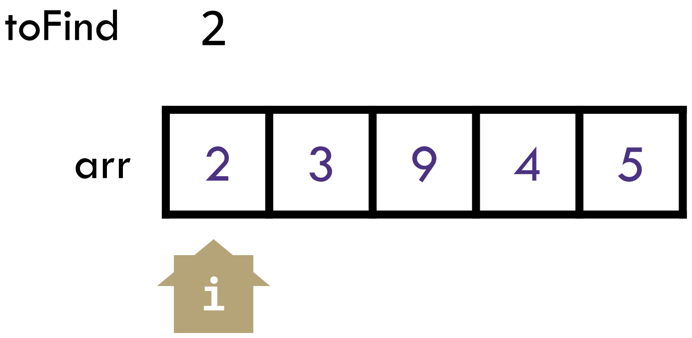
Case Study: Linear Search

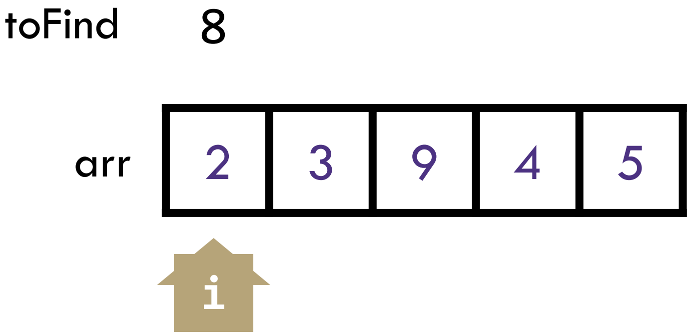
```
int linearSearch(int[] arr, int toFind) {
   for (int i = 0; i < arr.length; i++) {
    if (arr[i] == toFind) {
       return i;
      }
   }
   return -1;
}
```

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.

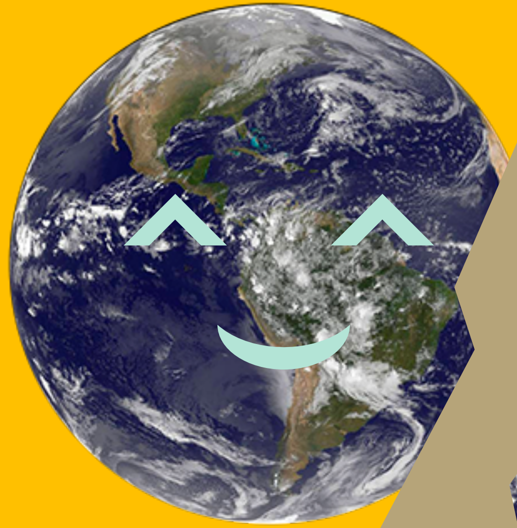
- If `toFind` is in `arr[0]`, we'll only need one iteration, $f(n) = 4$.
- If `toFind` is not in `arr`, we'll need n iterations. $f(n) = 3n + 1$.
- And there are a bunch of cases in-between.





Best Case

On Lucky Earth

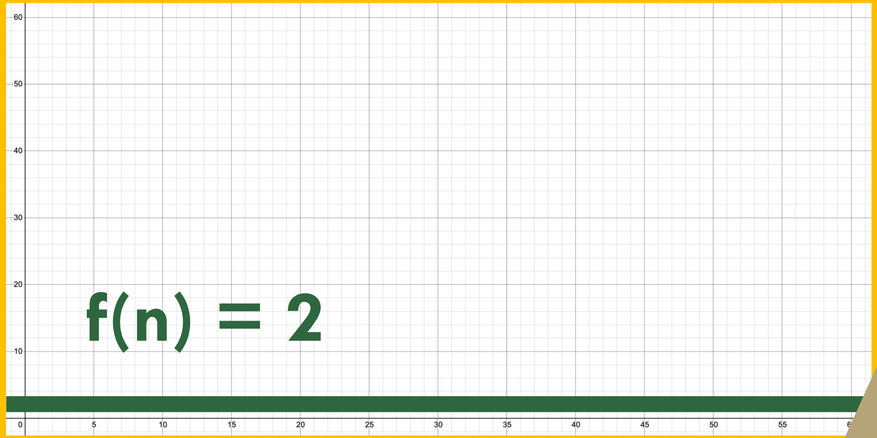


toFind 2

arr

2	3	9	4	5

i

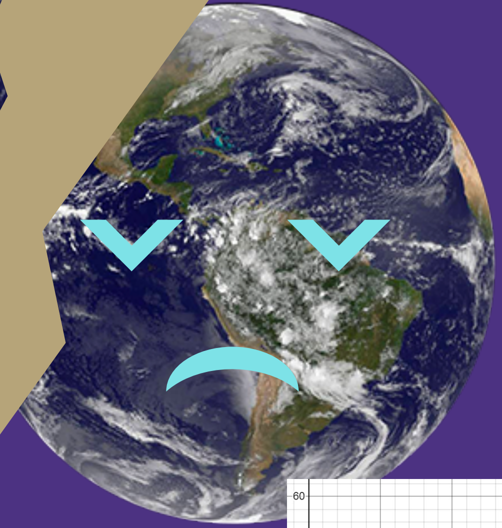


After asymptotic analysis:

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

Worst Case

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

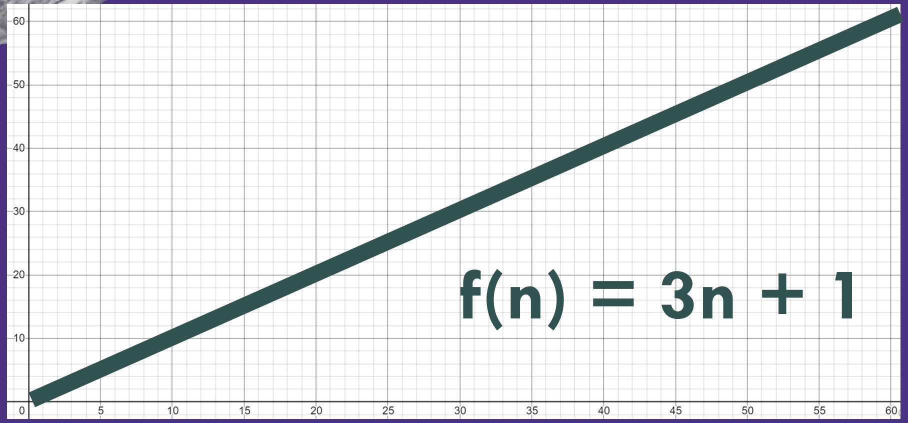


toFind 8

arr

2	3	9	4	5

i



After asymptotic analysis:

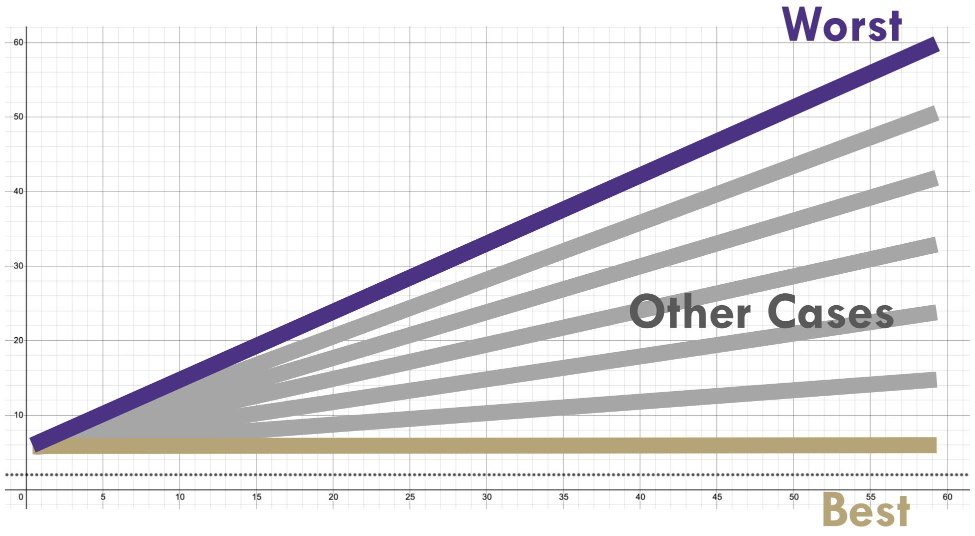
$O(n)$ $\Theta(n)$ $\Omega(n)$

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

Keep separate the ideas of best/worse case and O, Ω, Θ.

Big-O is an upper bound, regardless of whether we're doing worst or best-case analysis.

Worst case vs. best case is a question **once we've fixed $\boldsymbol{n}$** to choose the state of our data that decides how the code will evolve.

What is the exact state of our data structure, which value did we choose to insert?

O, Ω, Θ are choices of how to summarize the information in the model.

	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.

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!

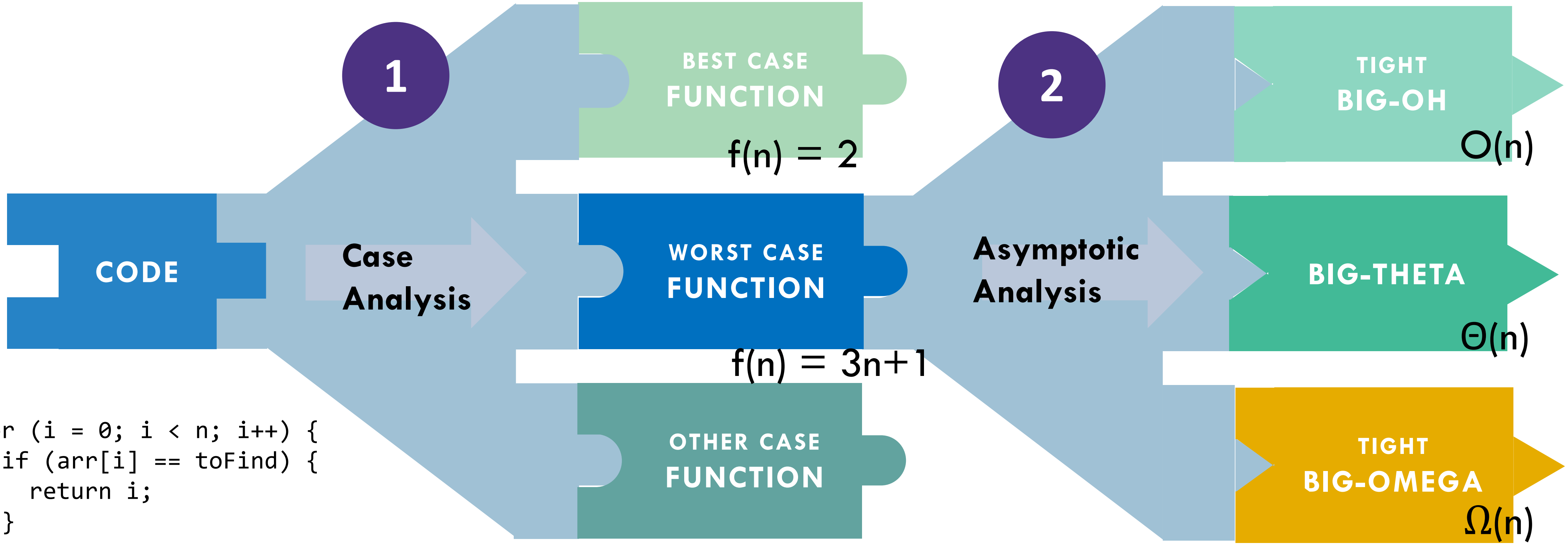
How to do case analysis

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 inputs can cause you to hit the (best/worst) parts of the code.

3. Now do the analysis like normal!

Algorithmic Analysis Roadmap




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