Lecture 4: Code Modeling and Asymptotic Analysis

CSE 373: Data Structures and Algorithms

Warm Up

ArrayQueue<E>

state

```
data[]
Size
front index
back index
```

behavior

```
add – data[size] = value, if out of room grow data
remove – return data[size - 1], size-1
peek – return data[size - 1]
size – return size
isEmpty – return size == 0
```

LinkedQueue<E>

state

```
Node front
Node back
size
```

behavior

```
add – add node to back
remove – return and remove node at front
peek – return node at front
size – return size
isEmpty – return size == 0
```

Respond to the poll everywhere with what complexity class (constant or linear) would best the runtime of the following situations.

Situation #1 – adding a new element to an ArrayQueue when there is still unused capacity in the underlying array "data[]"

Situation #2 - adding a new element to an ArrayQueue when there is no unused capcity in the underlying array "data[]"

Situation #3 – adding a new element to a LinkedQueue

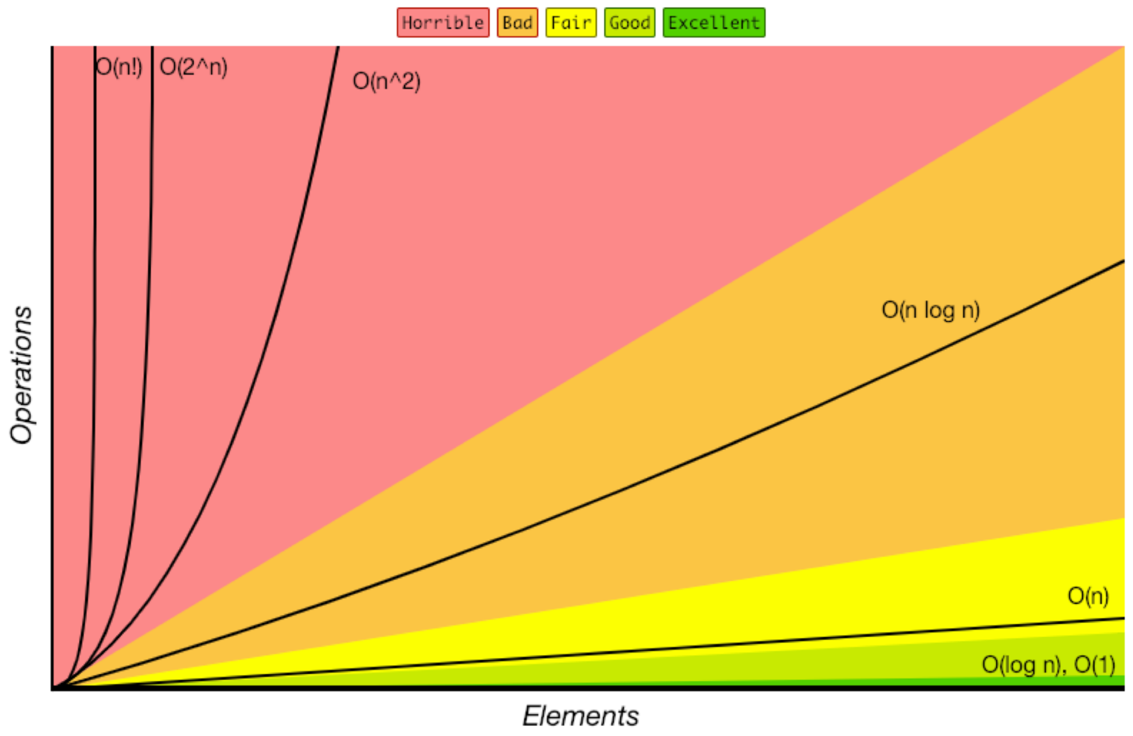
Take 2 Minutes

1. www.pollev.com/cse373activity for participating in our active learning questions. For this question label your answer with
   - what situation #
   - Constant or Linear
   - why.
2. https://www.pollev.com/cse373studentqs to ask your own questions

Review: Complexity Classes

complexity class – a category of algorithm efficiency based on the algorithm's relationship to the input size N

Class	Big O	If you double N…	Example algorithm
constant	$O(1)$	unchanged	Add to front of linked list
logarithmic	$O(\log n)$	Increases slightly	Binary search
linear	$O(n)$	doubles	Sequential search
"n log n"*	$O(n\log n)$	Slightly more than doubles	Merge sort
quadratic	$O(n^2)$	quadruples	Nested loops traversing a 2D array
cubic	$O(n^3)$	Multiplies by 8	Triple nested loop
polynomial	$O(n^c)$		
exponential	$O(c^n)$	Increases drastically	



*There's no generally agreed on term. "near[ly]-linear" is sometimes used.

http://bigocheatsheet.com/

Administrivia

- Project 0 is due Wednesday at 11:59pm PST
- Office hours start this week, check out the calendar
- Project 1 will go live by Wednesday 11:59pm (you might want to find a partner)
- Check out the slack!
  - Find a partner
- Individual written exercise going out on Friday
- Feedback: too many breaks for questions

General process

a piece of code → code modeling → a mathematical function modeling the runtime of a piece of code → asymptotic analysis → big O runtime

An example of the process

`method1()` → code modeling → f(n) = $5n^2$+3n+2 → asymptotic analysis → O(n^2)

Disclaimer

This topic has lots of details/subtle relationships between concepts.

We're going to try to introduce things one at a time (all at once can be overwhelming).

"We'll see that later" might be the answer to a lot of questions.

"Could you go over ____ again? or " ____ part of this topic is confusing" are totally valid questions / opinions to voice. If you're able to say those on pollev / chat we can probably all learn and benefit from it.

Code Modeling

code modeling – the process of mathematically representing how many operations a piece of code will run in relation to the number of inputs n. (We're going to turn code into a function representing it's runtime)

What counts as an "operation"?

Assume all basic operations run in equivalent time

Basic operations
- Adding ints or doubles
- Variable update
- Return statement
- Accessing array index or object field

Consecutive statements
- Sum time of each statement

Function calls
- Count runtime of function body
- Remember that `new` calls a function!

Conditionals
- Time of test + appropriate branch
  - We'll talk about which branch to analyze when we get to cases.

Loops
- (Number of loop iterations) * (runtime of loop body)

Code	Runtime function

```
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
}
```

Approach

-> start with basic operations

- Each basic operation = +1
- Loop = #iterations * (operations in loop body)

- the loop runs n times
- 6 steps per iteration

= 6n

f(n) = 6n + 3

```
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) { +1
                    // do nothing, just for fun!
               }
               sum = sum + (i * 3) + j;  +4
               j = j + 1;  +2
          }
          i = i + 1;  +2
     } return sum;
```

- inner loop: runs n times
- 8 steps per iteration

= 8n

- outer loop: runs n times
- 8n + 3 steps per iteration

= n(8n+3)

f(n) = n(8n+3) + 3

General process

a piece of code → code modeling → a function modeling the runtime of a piece of code → asymptotic analysis → big O runtime

An example of the process

`method1()` → code modeling → f(n) = $5n^2$+3n+2 or visual plot representation → asymptotic analysis → O(n^2)

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.
   - 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.

3. Write the final big-O – (basically just putting the O symbol around your remaining n-term)

$f(n) = \mathrm{n(8n+3)} + 3$

$f(n) = \mathrm{n(8n+3)} + 3 = 8n^2 + 3n + 3$

$f(n) = 8n^2 + 3n + 3 \approx 8n^2$

$f(n) \approx 8n^2 \approx n^2$

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

Wait, what? **Asymptotic Analysis - big ideas**

Why did we just throw out all of that information? Big-O is like the “significant digits” of computer science.

Asymptotic Analysis is how a function behaves as n→ ∞ so when we do asymptotic analysis (putting functions inside a big-O), <u>we only care about what happens when n gets bigger and approaches infinity</u>.

We don’t care about smaller values because all code is “fast enough” for small n in practice. If you’re only focusing on small inputs /small n, you’re not doing asymptotic analysis.

<u>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 matters and drives growth, and is why we hone in on it and drop everything else.</u>

Caring about infinity and the highest order term are big ideas!

Using the constants isn't more accurate either

```
public static void method1(int[] input)
{
    int n = input.length;
    input[n-1] = input[3] + input[4];
    input[0]+= input[1];
}
```

```
public static void method1(int[]); Code:
0: aload_0
1: arraylength
2: istore_1
3: aload_0
4: iload_1
5: iconst_1
6: isub
7: aload_0
8: iconst_3
9: iaload
10: aload_0
11: iconst_4
12: iaload
13: iadd
14: iastore
15: aload_0
16:
20: iconst_1
21: iaload
22: iadd
23: iastore
24: return
```

```
public static void method2(int[] input)
{
    int five = 5;
    input[five] = input[five] + 1;
    input[five]--;
}
```

```
public static void method2(int[]); Code:
0: iconst_5
1: istore_1
2: aload_0
3: iload_1
4: aload_0
5: iload_1
10: aload_0
11: iload_1
12: dup2
13: iaload
14: iconst_1
15: isub
16: iastore
17: return
```

Code Modeling anticipating asymptotic analysis

We can't accurately model the constant factors just by staring at the code.

And the lower-order terms matter even less than the constant factors.

So we just ignore them for the big-O.

This does not mean you shouldn't care about constant factors ever – they are important in real code!

- Our theoretical tools aren't precise enough to analyze them well.

Code modeling: more practice

Write the specific mathematical code model for the following code and indicate the big-O runtime in terms of k.

```
public void method3 (int k) {
    int j = 0; +1
    while (j < k) { +k/5 (body)
        for (int i = 0; i < k; i++) { +k(body)
            System.out.println("Hello world"); +1
        }
        j = j + 5; +2
    }
}
```

$$f(k) = \frac{k(k+2)}{5}$$

quadratic -> $O(k^2)$

Approach

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

- Each basic operation = +1
- Loop = #iterations * (operations in loop body)

Code modeling takeaways

- We talked about counting +1s to give you an intuition and point out what's not important (variable assignments, math operators, etc.) and what is important (loops, method calls, etc.)

- Once you've gotten some practice with a couple of these, you'll find that you won't need to count up the individual +1s. Those +1s won't really matter at a high-level (for the most part we're going to drop constants when we turn the code model function into a big-O), we instead look at what the more expensive operations are (loops, method calls, recursion) and see how the +n's or the *n's add up;

Questions

Formal Definition of Big-O

Formal Definitions: Why?

If you're analyzing simple functions that are similar to those you've analyzed before, you don't bother with the formal definition. You can just be comfortable using your intuitive definition.

If you're analyzing more complex code or functions, however, this formal definition is a good fallback.

We're going to be making more subtle big-O statements in this class.

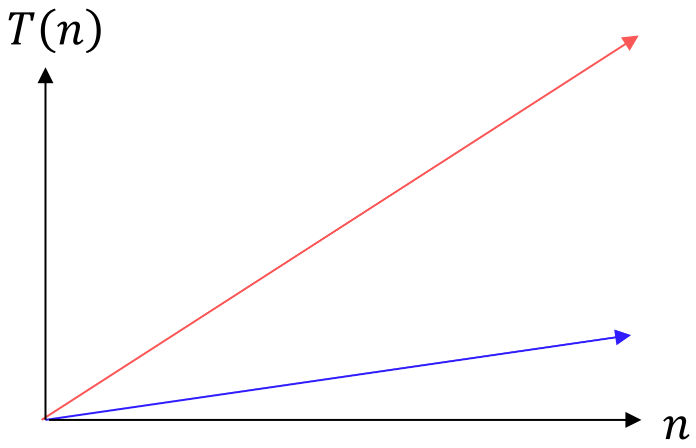
- We need a mathematical definition to be sure we know exactly where we are.

We're going to teach you how to use the formal definition, so if you get lost (come across a weird edge case) you know how to get your bearings.

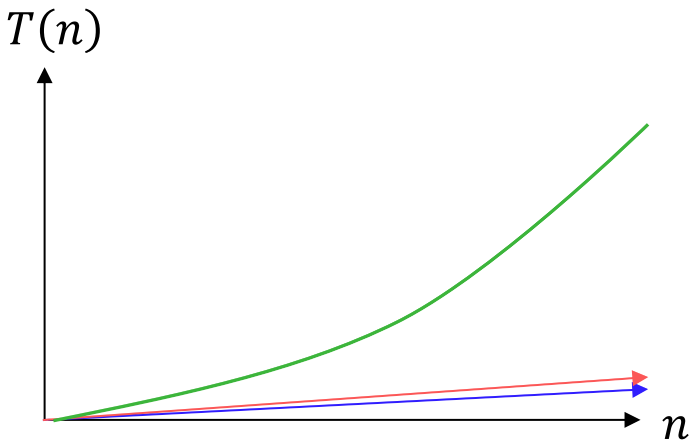
Function growth and what we want out of our formal definition

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

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

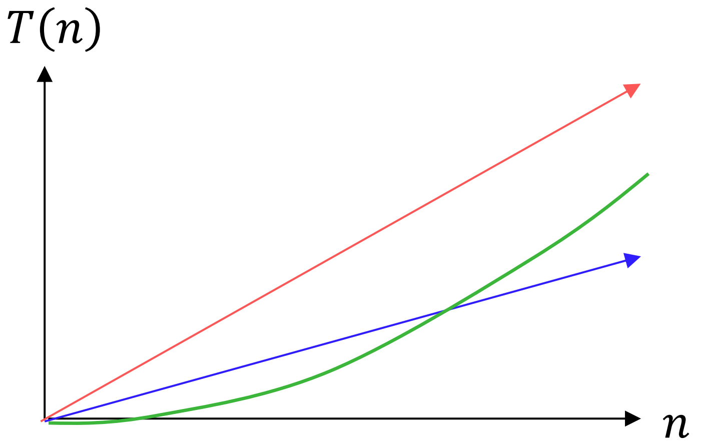


The growth rate for f(n) and g(n) looks very different for small numbers of input



...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

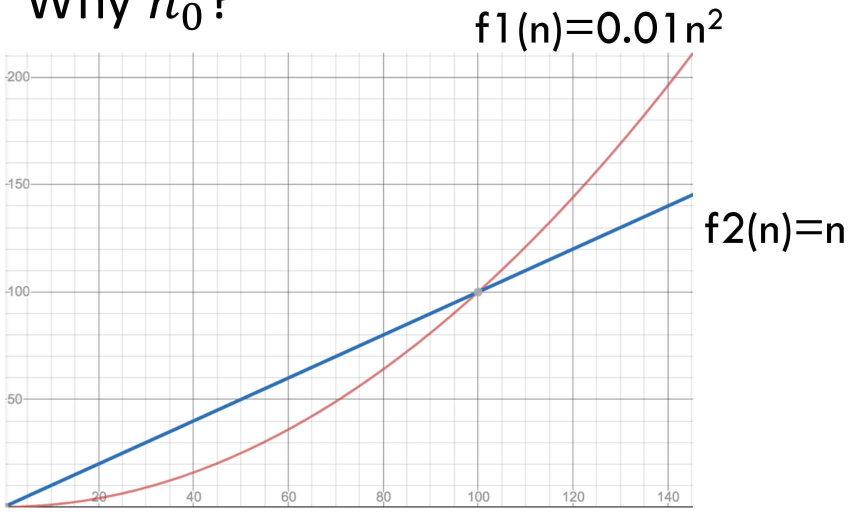
We wanted to find an upper bound on our algorithm's running time, but

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

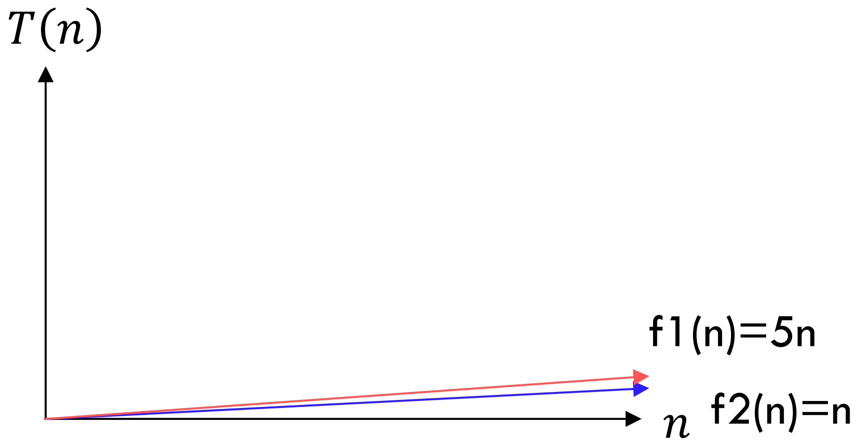
Big-O
$f(n)$ is $O(g(n))$ if there exist positive constants c, n_0 such that for all $n \geq n_0$, $$f(n) \leq c \cdot g(n)$$

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

Why n_0?



Why c?



Big O Definition Proofs

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

$10n \leq c \cdot n$ when $c = 10$ for all values of n. $So\ 10n \leq 10n$

$15 \leq c \cdot n$ when $c = 15\ for\ n \geq 1.\ So\ 15 \leq 15n$

Add up all your truths

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

10n + 15 <= 25n for n >= 1.

which is in the form of the definition

f(n) <= c * g(n)

where c = 25 and n0 = 1.

Big O Definition Proofs: more practice

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 $n \geq 1$

$5n^2 + 3n^2 + 6n^2 = 14n^2$

$5n^2 + 3n + 6 \leq 14n^2$ for $n \geq 1$

$14n^2 \leq c*n^2$ for c = ? n >= ?

c = 14 & n_0 = 1

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-O Definition Proofs: outline

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

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.

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!

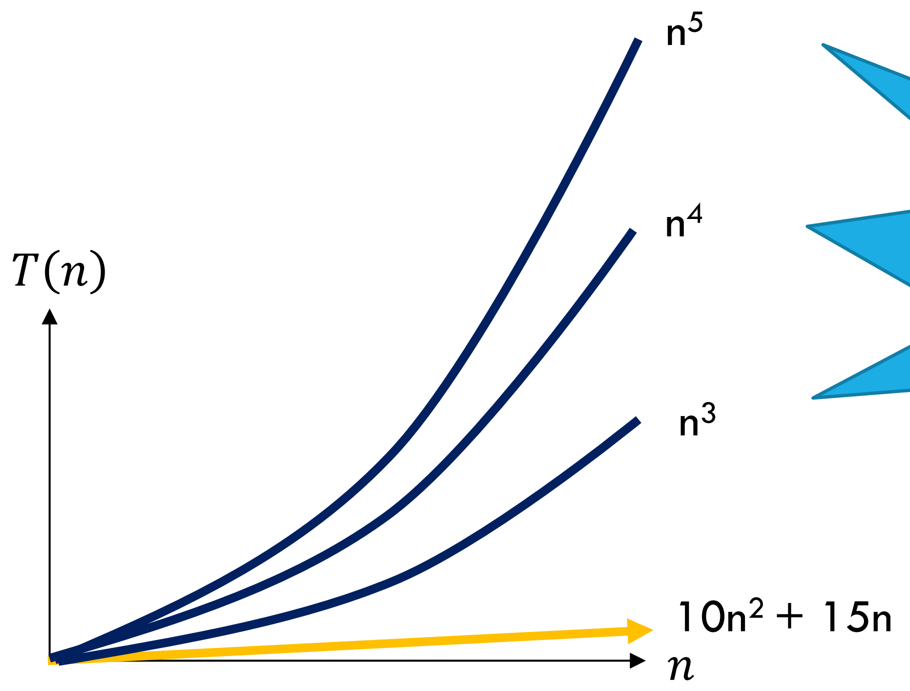
Note: Big-O definition is just an upper-bound, not always an exact bound (plots)

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 …



The visual representation of big-O and asymptotic analysis is a big idea!

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.

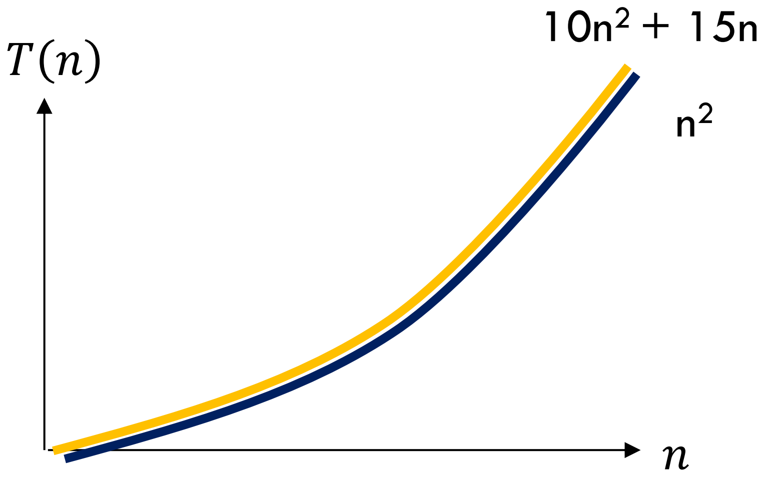
Computer scientists It is almost always technically correct to say your code runs in time $O(n!)$. (Warning: don't try this trick in an interview or exam)

If you zoom out a bunch,

the your tight bound and your function will

be overlapping compared to other

complexity classes.



Questions

Uncharted Waters: a different type of code model

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

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

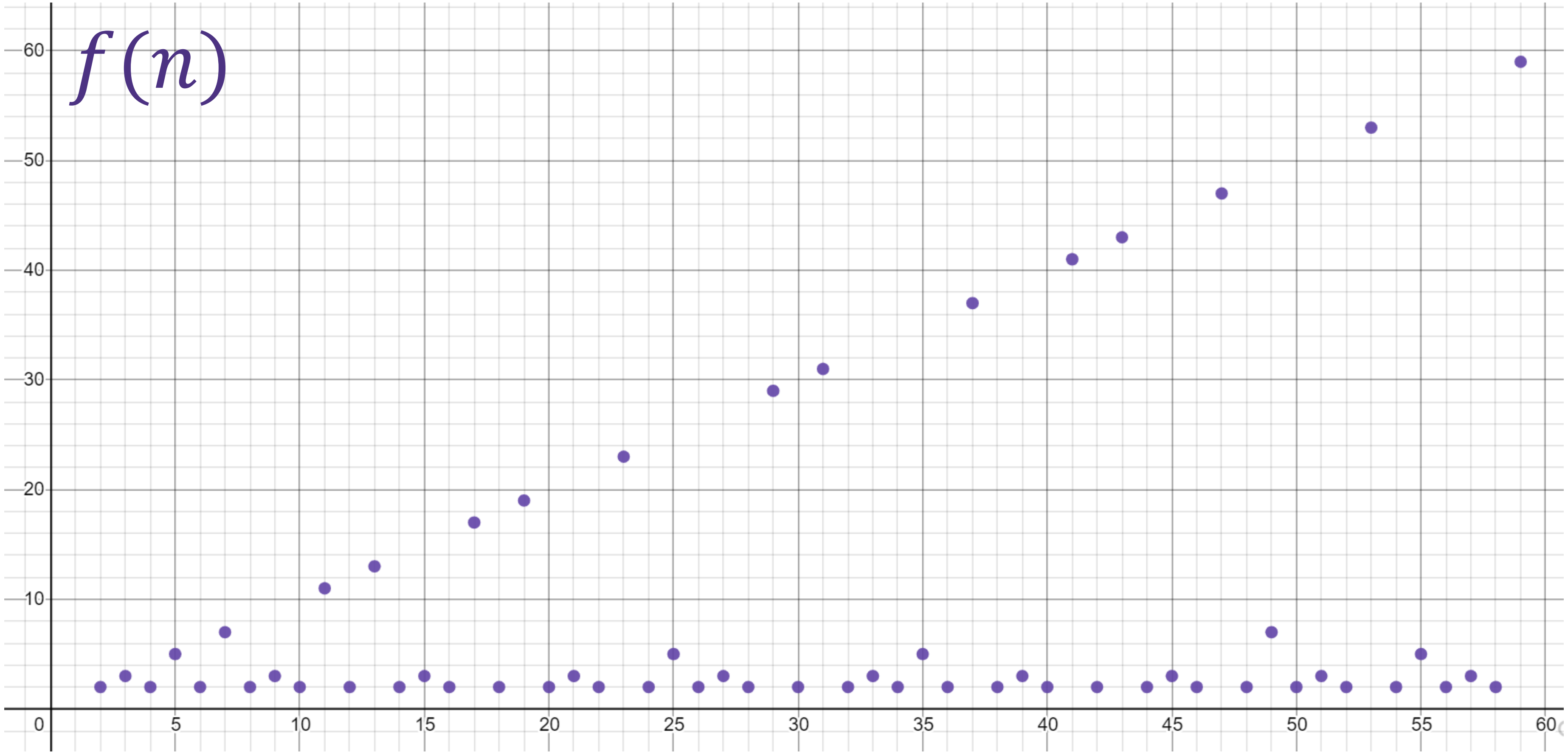
Remember, $f(n)$ = the number of basic operations performed on the input n.

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

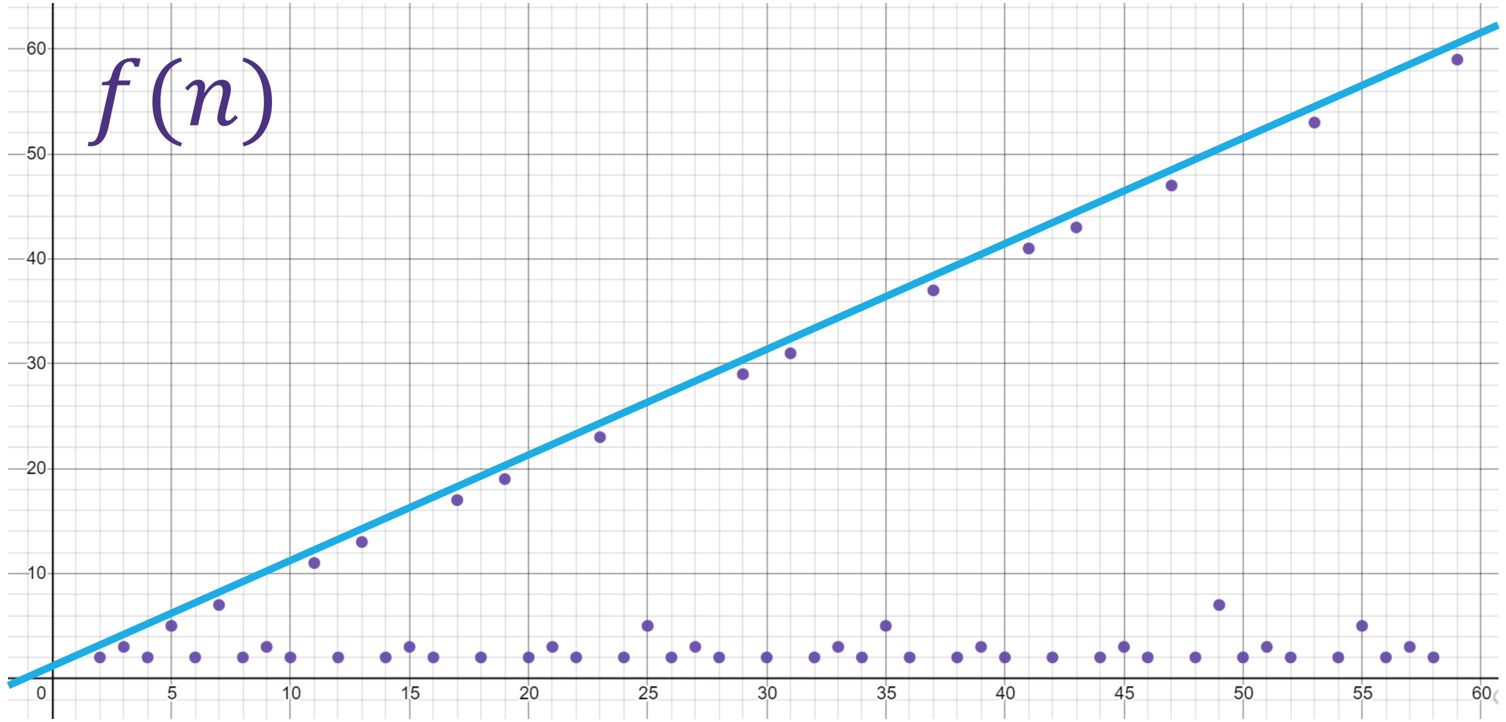


Is the running time of the code $O(1)$ or $O(n)$?

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

$f(n)$ 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 $n \geq 5$

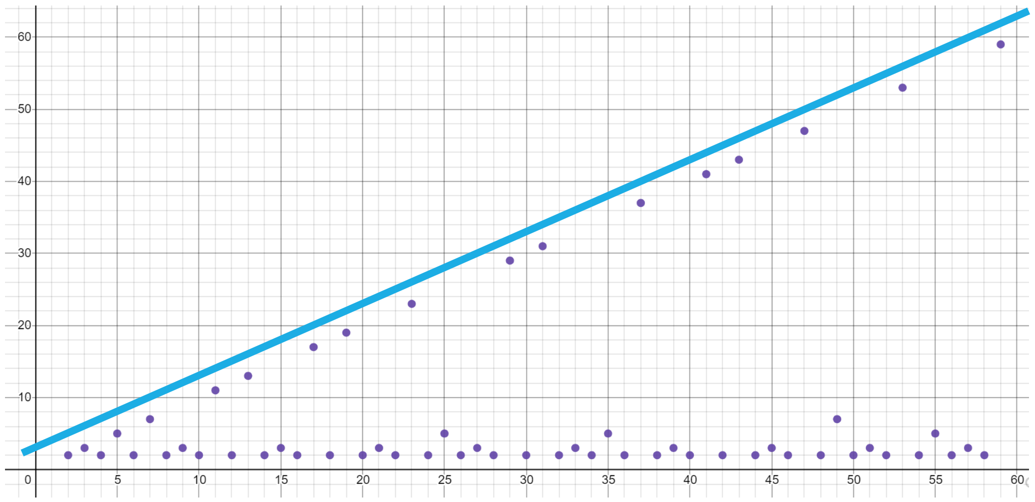
It's $O(n)$ but not $O(1)$

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

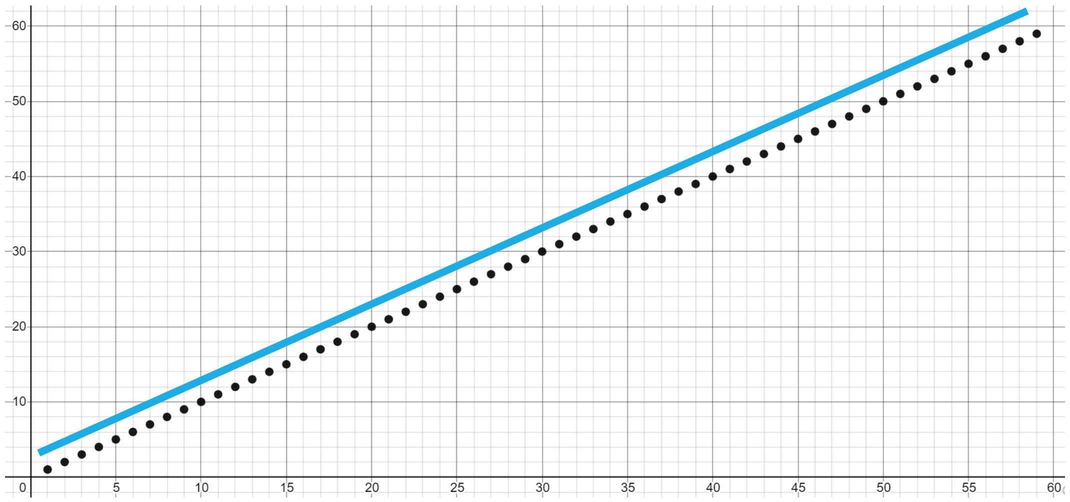
No! Choose your value of c. I can find a prime number k bigger than c.
And $f(k) = k > c \cdot 1$ so the definition isn't met!

Big-O isn’t everything

Our prime finding code is $O(n)$ as tight bound. But so is 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-O

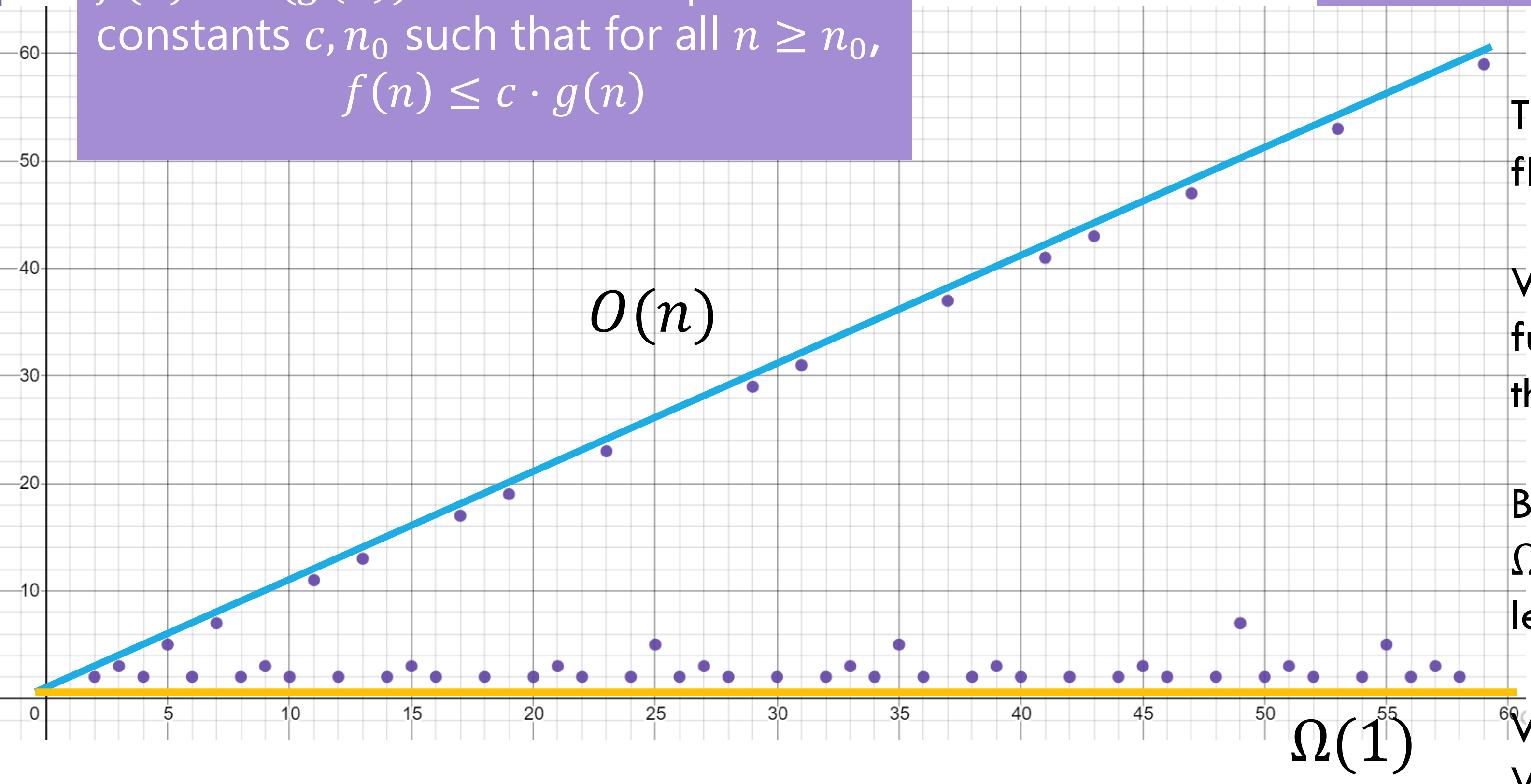
$f(n)$ is $O(g(n))$ if there exist positive constants c, n_0 such that for all $n \geq n_0$,

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

Big-Omega

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

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



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?

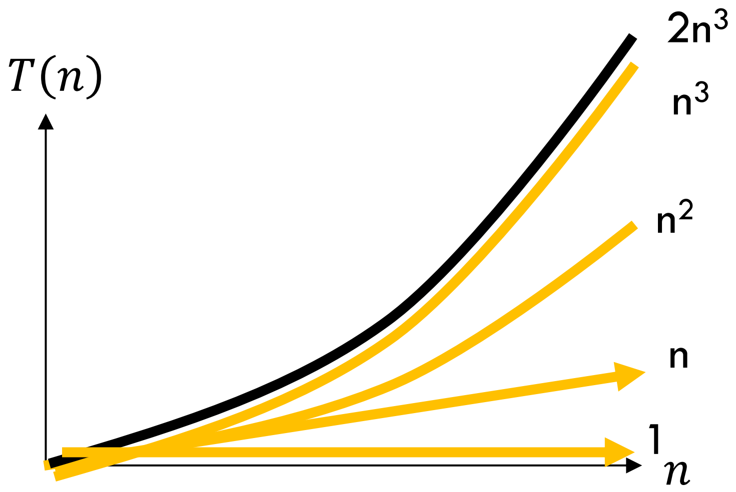
Big-Omega definition Plots

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

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

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

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



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

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 $n \geq n_0$,

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

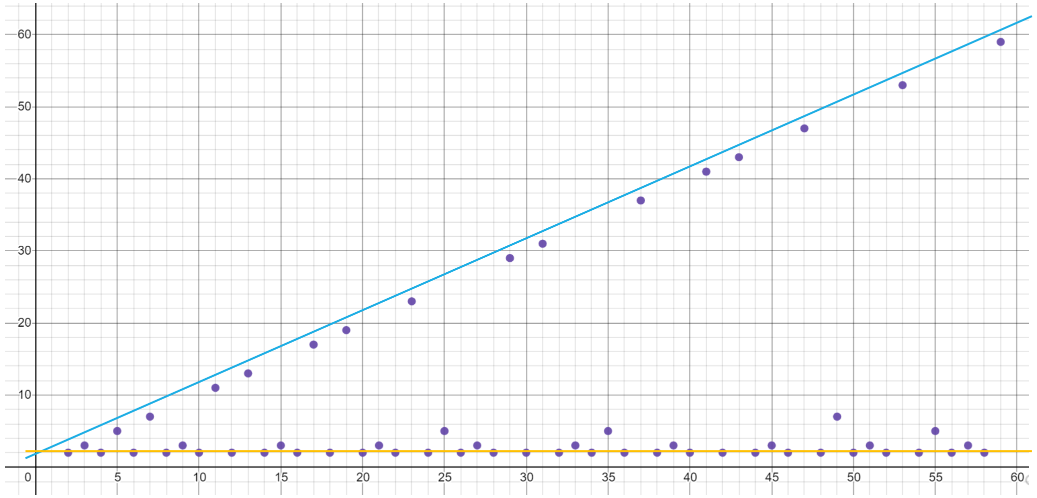
Big-Omega

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

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

Tight Big-O and Big-Ω bounds shown together

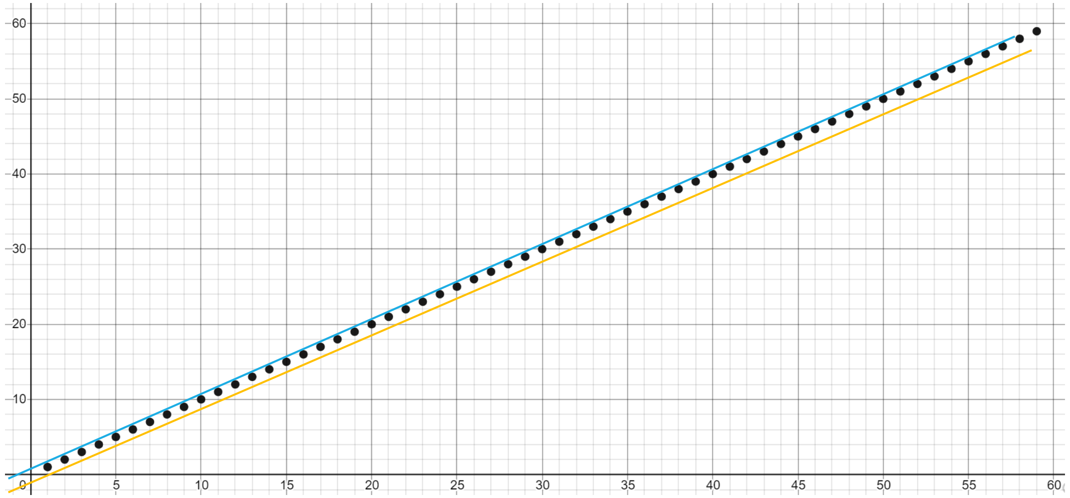
prime runtime function



$O(n)$

$\Omega(1)$

f(n) = n



$O(n)$

$\Omega(\mathrm{n})$

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.

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 $\Omega(g(n))$.

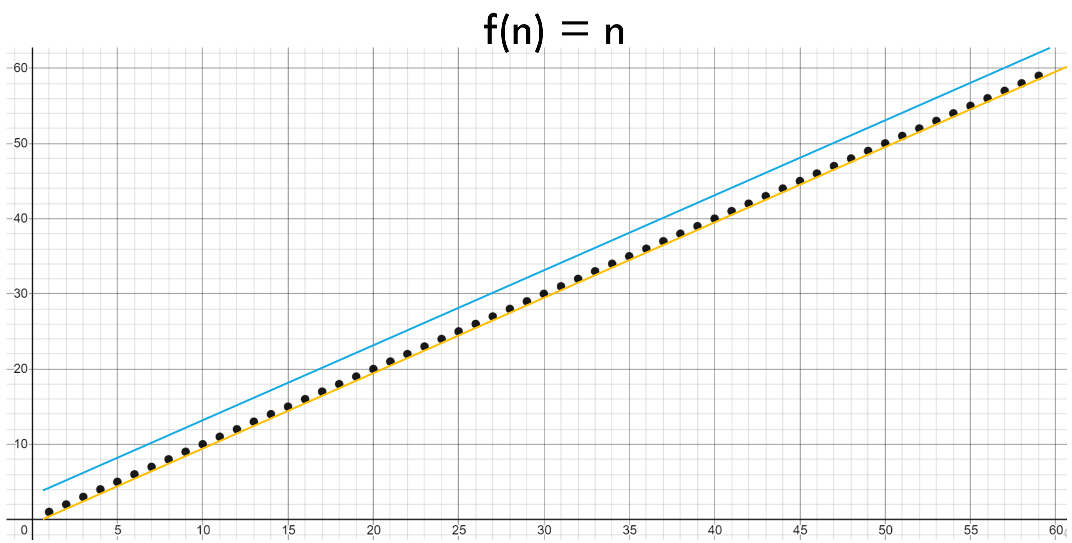
(in other words: there exist positive constants $c1, c2, n_0$ such that for all $n \geq n_0$)

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

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 terms



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

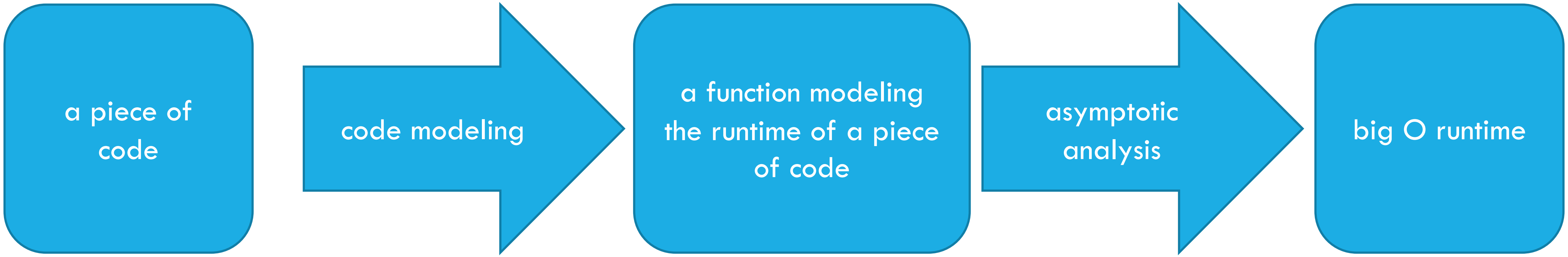
Big-Theta

$f(n)$ 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 $c1, c2, n_0$ such that for all $n \geq n_0$)

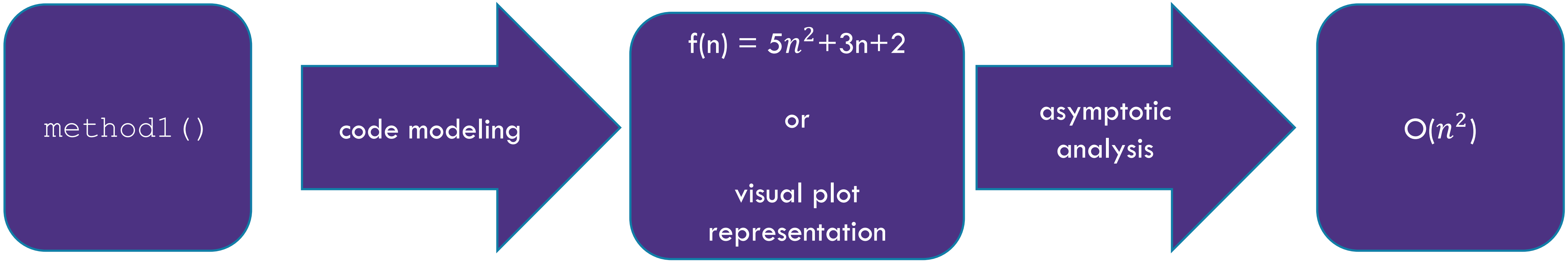
$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

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)

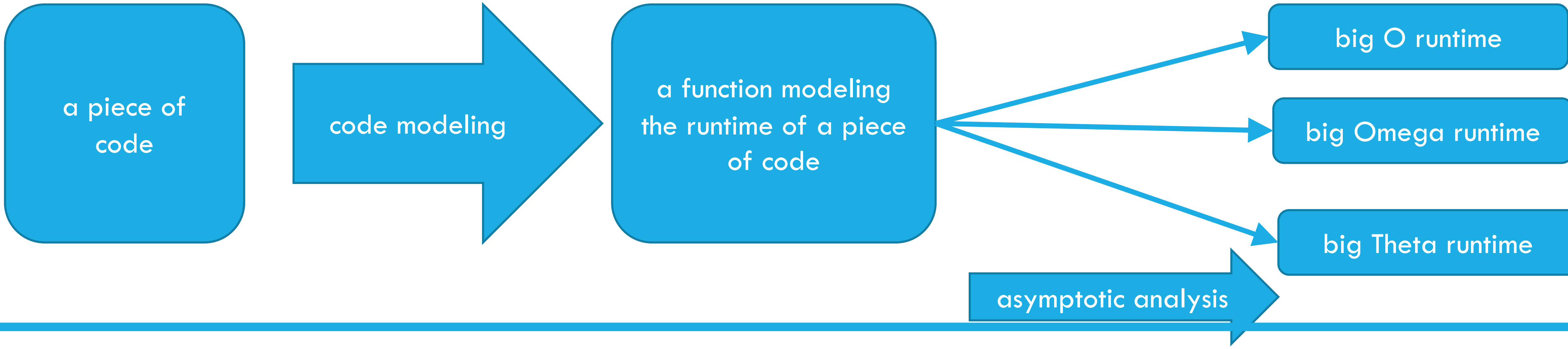
Initial general process



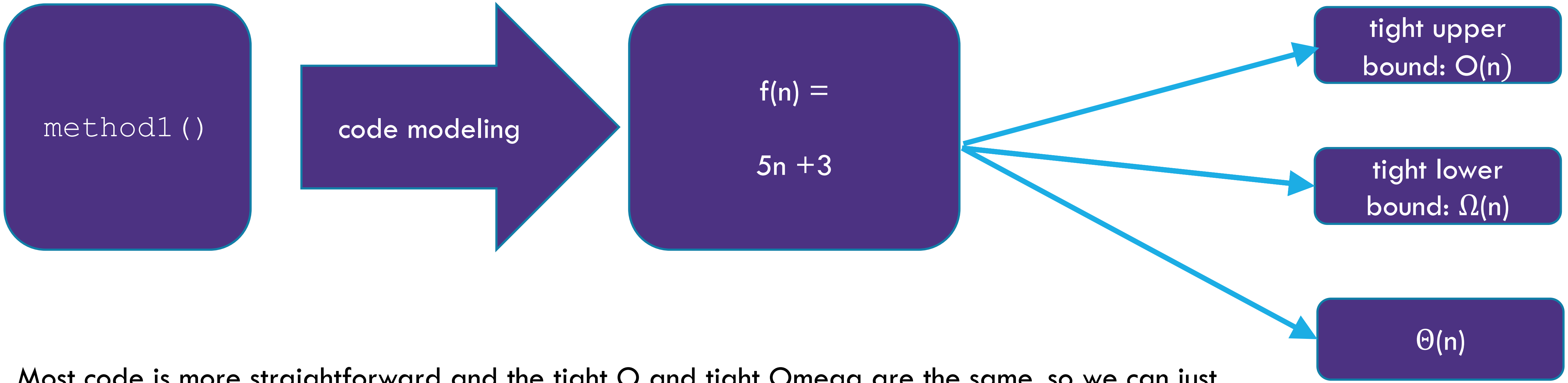
An example of the process



General process after knowing about Omega and Theta

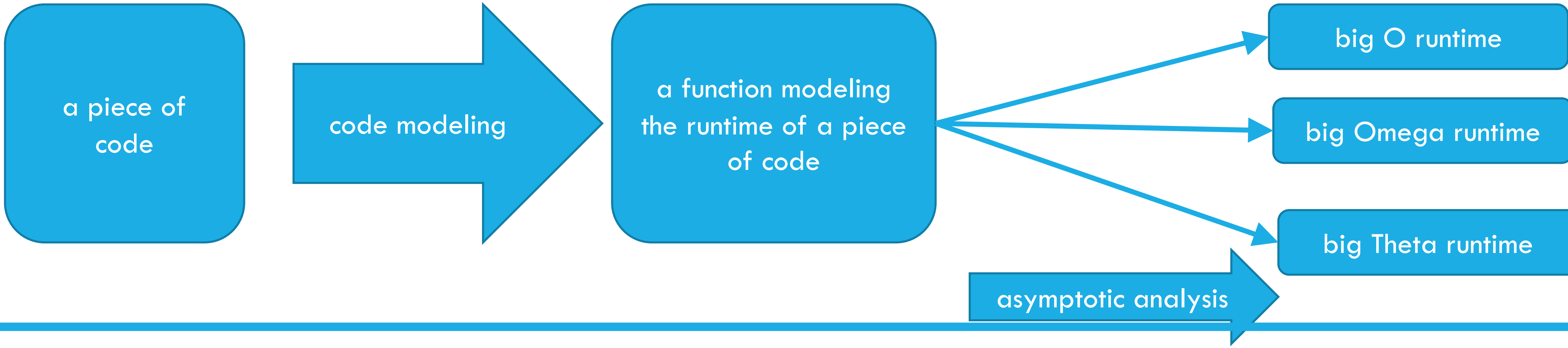


An example of the process

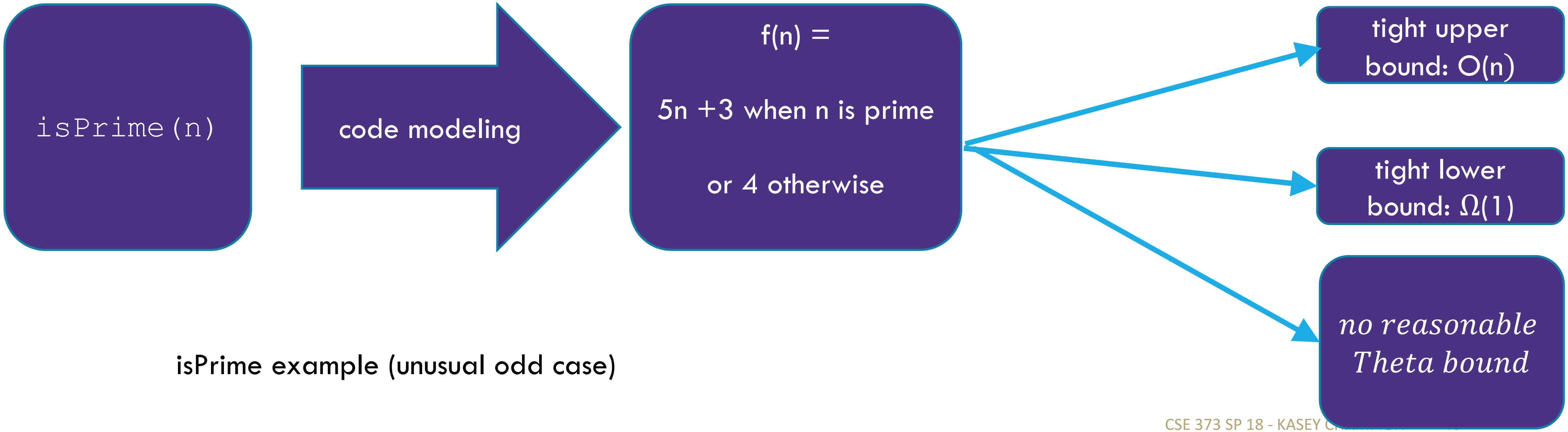


Most code is more straightforward and the tight O and tight Omega are the same, so we can just refer to the Theta runtime.

General process



An example of the process



isPrime example (unusual odd case)

Questions

Takeaways

- rough idea of how to turn a piece of code into a function that we can categorize with big-O, Omega, and Theta

- definition of big O, Omega Theta and how functions fit into them:
  - visually with plots
  - through formal math definitions
  - tight/loose bounds