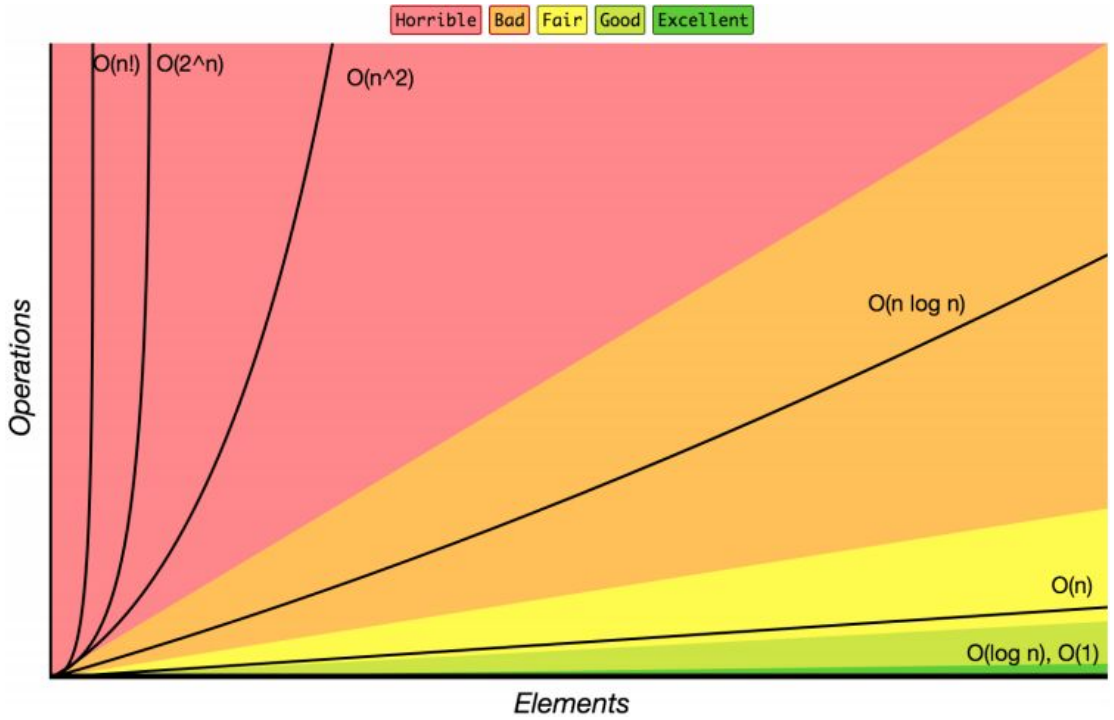
Lecture 4: Intro to Runtime Analysis

CSE 373: Data Structures and Algorithms

Review: Complexity Class

complexity class: A category of algorithm efficiency based on the algorithm's relationship to the input size N.

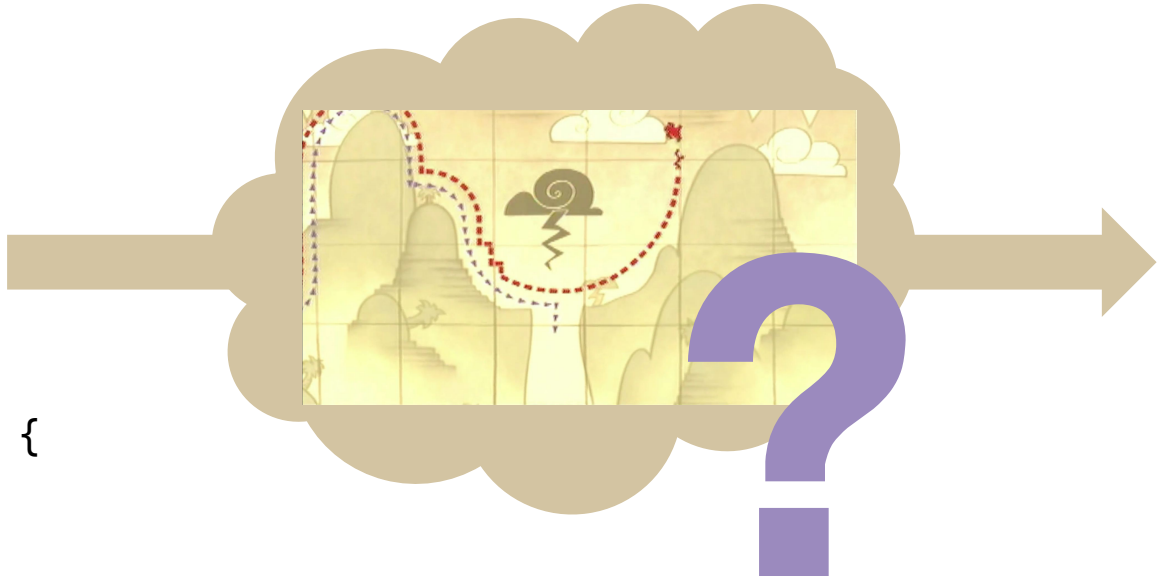
Complexity Class	Big-O	Runtime if you double N	Example Algorithm
constant	O(1)	unchanged	Accessing an index of an array
logarithmic	$O(\log_2 N)$	increases slightly	Binary search
linear	O(N)	doubles	Looping over an array
log-linear	$O(N \log_2 N)$	slightly more than doubles	Merge sort algorithm
quadratic	$O(N^2)$	quadruples	Nested loops!
…	…	…	…
exponential	$O(2^N)$	multiplies drastically	Fibonacci with recursion



bigocheatsheet.com

Code to Big-O

CODE



BIG-O

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

O(n)

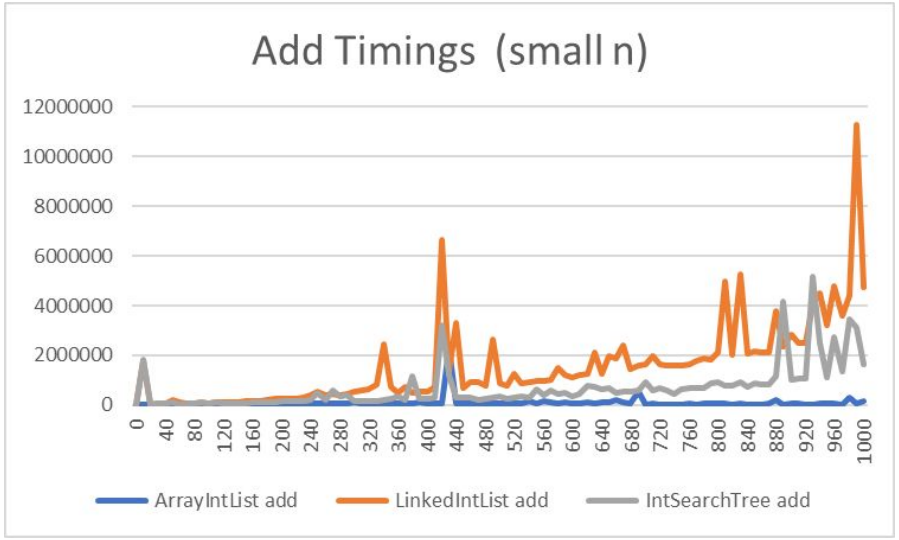
123/143 general patterns:

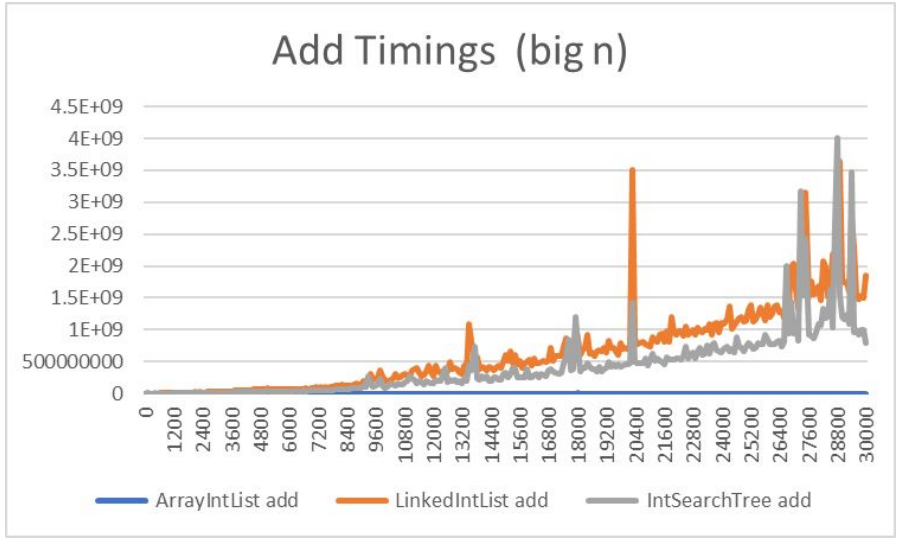
- O(1) constant is no loops
- O(n) is one loop
- $O(n^2)$ is nested loops

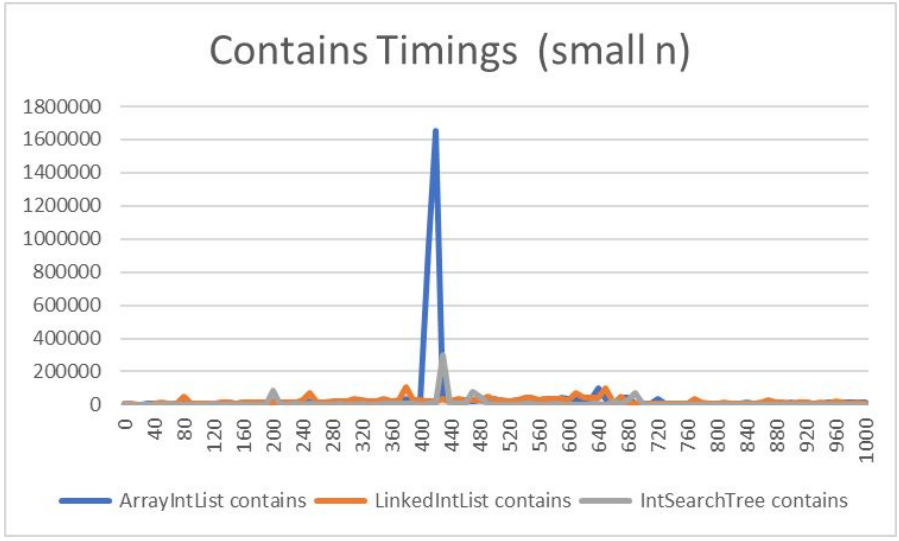
373:

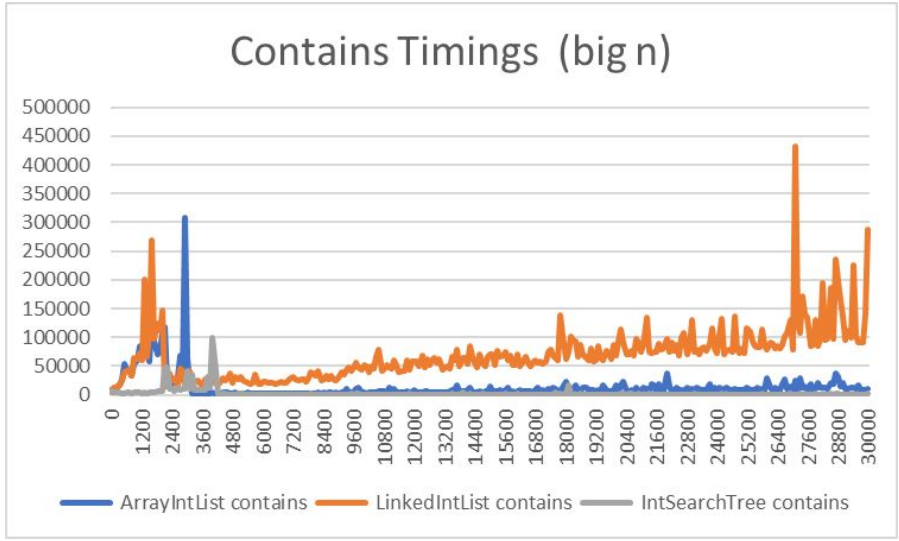
We need a way to definitively determine Big O for all code

Why not time code?









Actual time to completion can vary depending on hardware, state of computer and many other factors.

These graphs are of times to run add and contains on structures of various sizes of N and you can see inconsistencies in individual runs which can make determining the overall relationship between the code and runtime less clear.

You can find the code to run these tests on your own machine on the course website!

Meet Algorithmic Analysis


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

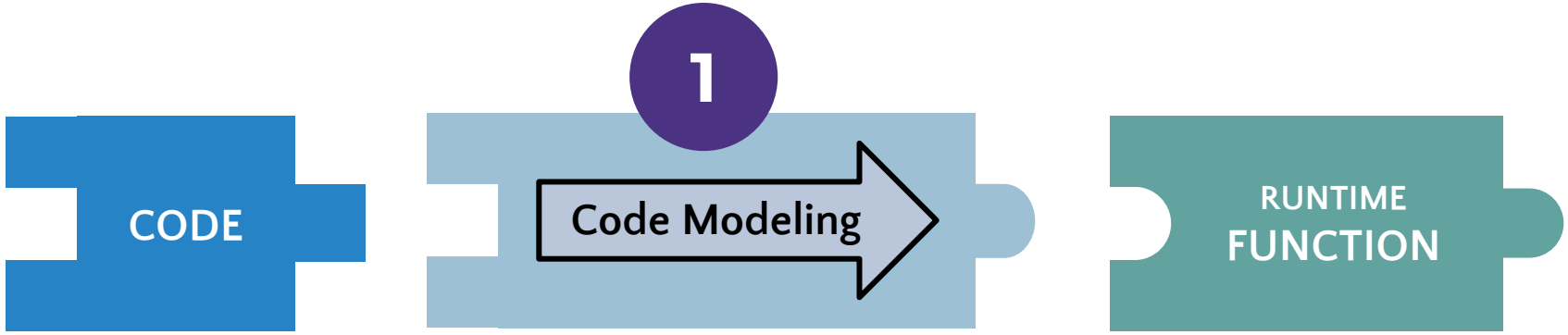
f(n) = 2n

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

Code Modeling



Code Modeling – the process of mathematically representing how many operations a piece of code will run in relation to the input size n.

- Convert from code to a function representing its runtime

Example:

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

- One array element update = "1" runtime count
- Loop that runs "n" times = "n" runtime count
- Loop N times(2 runtime counts inside loop)

f(n) = 2N

What Code Count as an "operation"?

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

- Basic commands count as "1":
  - +, -, /, *, %, ==
  - Assignment
  - Returning
  - Variable/array access
- Function Calls
  - Total runtime in body
  - Remember: new calls a function (constructor)
- Conditionals
  - Test + time for the followed branch
    - Learn how to reason about branch later
- Loops
  - Number of iterations * total runtime in condition and body
  - For loop header operations don't count, but while loop headers do

Code Modeling Example 1

```
public void method1(int n) {
    int sum = 0;                    +1
    int i = 0;                      +1
    while (i < n) {                 +1
        sum = sum + (i * 3);        +3
        i = i + 1;                  +2
    }
    return sum;                     +1
}
```

Loop runs n times

+6 *n

$f(n) = 6n + 3$

Practice time!

Code Modeling Example 2

```
public void method2(int n) {
    int sum = 0;                          +1
    int i = 0;                            +1
    while (i < n) {                       +1
        int j = 0;                        +1
        while (j < n) {                   +1
            if (j % 2 == 0) {             +2
                // do nothing
            }
            sum = sum + (i * 3) + j;      +4
            j = j + 1;                    +2
        }
        i = i + 1;                        +2
    } return sum;                         +1
}
```

This inner loop runs n times

+9 *n

This outer loop runs n times

9n + 4 *n

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

Where are we?

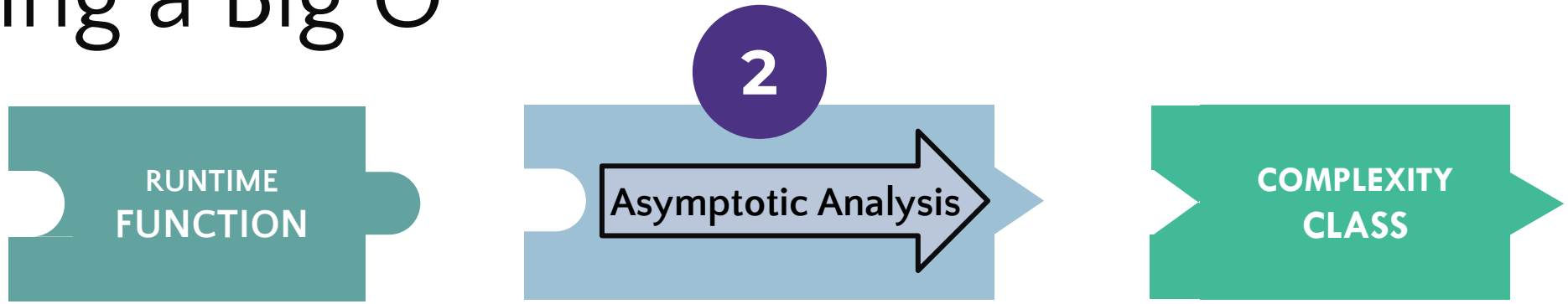

```
for (i = 0; i < n; i++) {
  a[i] = 1;
  b[i] = 2;
}
```

f(n) = 2n

O(n)

- We just turned a piece of code into a function!
  - We'll look at better alternatives for code modeling later
- Now to focus on step 2, asymptotic analysis

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
   a. The "dominating term" is the one that is the largest as n gets bigger. in this class, often the largest power of n.
2. Remove all coefficients
3. Remove all constant factors

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

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

$\approx 9n^2$

$\approx n^2$

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

Practice time!

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

$f(n) = 6n + 3$

$= 6n + 3$

$\approx 6n$

$\approx n$

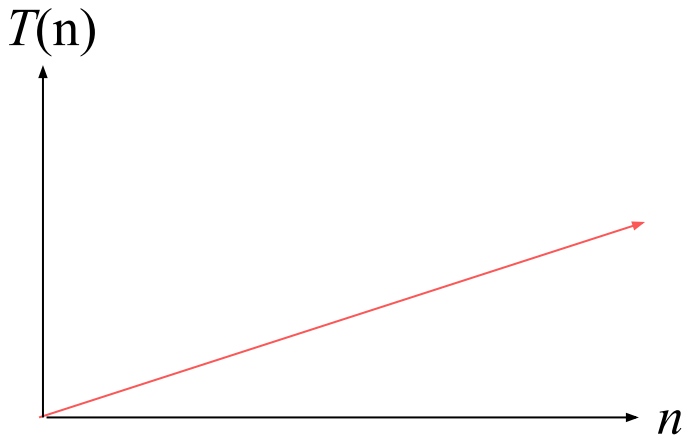
f(n) is O(n)

What is a "dominating term"?

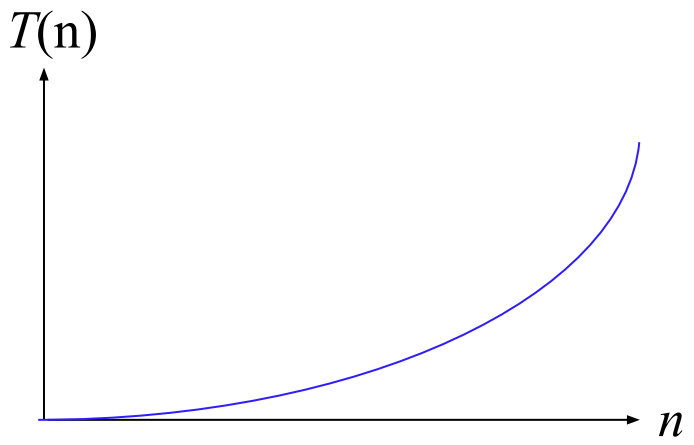
Asymptotic Analysis: Analysis of function behavior as its input approaches **infinity**

Dominating terms have the largest **influence** on the behavior of f(n) as they are the largest, and "dominate" the smaller terms

$f(n) = \mathbf{n} + 10$



$g(n) = \mathbf{n\text{^}2} + n + 10$



Practice time!

What is a "dominating term"?

What is the dominating term?

1. $n^2 + n$ — n^2
2. $n + 1000$ — n
3. $n^{100} + n^{50} + n^2 + 5$ — n^{100}
4. $n^2 + 2^n$ — 2^n
5. $3^n + 4^n$ — 4^n

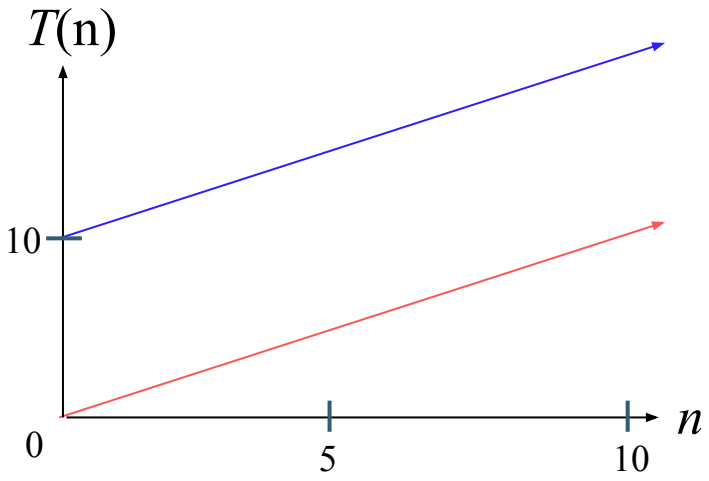
hint: ask yourself "which term is going to be the largest as the value of n increases?"

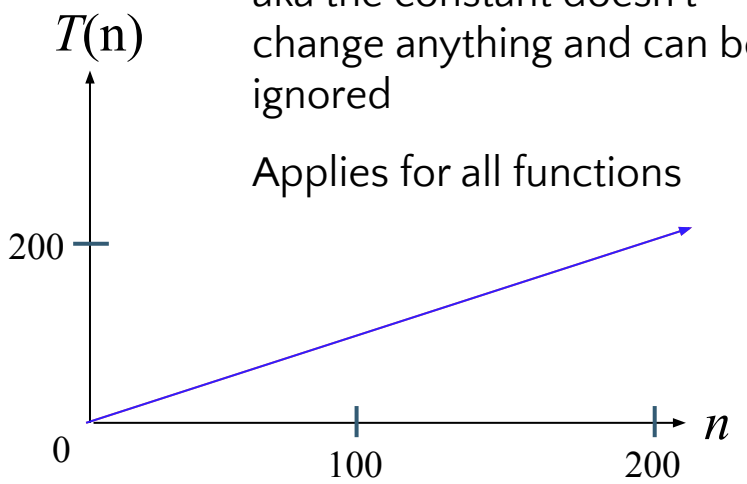
Can we really throw away all that info?

Asymptotic Analysis: Analysis of function behavior as its input approaches **infinity**

Let's look at linear functions and think about the effect of constants

$f(n) = n$ $g(n) = n + 10$





At the scale of infinity, f(n) and g(n) have identical behavior, aka the constant doesn't change anything and can be ignored

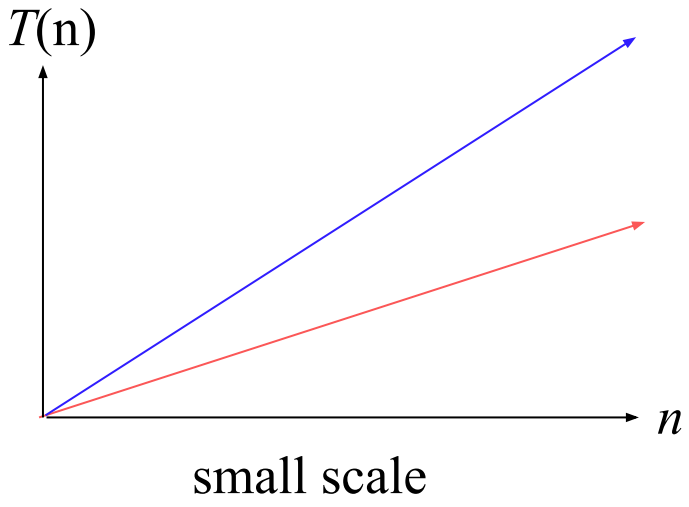
Applies for all functions

Can we really throw away all that info?

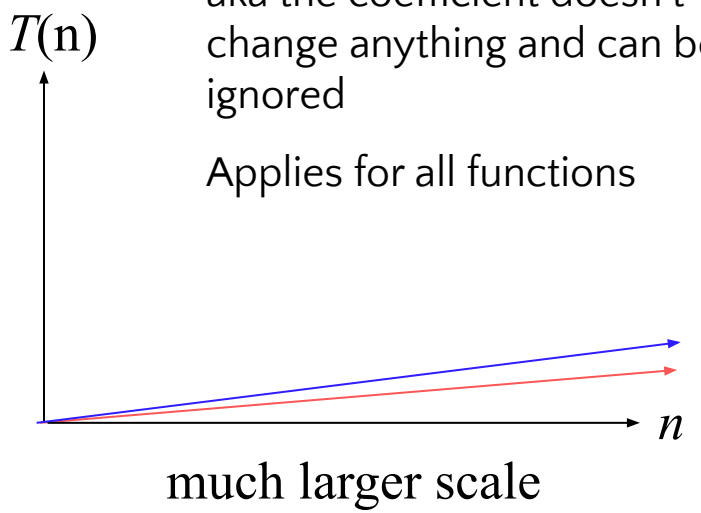
Asymptotic Analysis: Analysis of function behavior as its input approaches **infinity**

Let's look at linear functions and think about the effect of coefficients

$f(n) = n$ $g(n) = 2n$



small scale



much larger scale

At the scale of infinity, f(n) and g(n) have identical behavior, aka the coefficient doesn't change anything and can be ignored

Applies for all functions

Can we really throw away all that info?

Big-Oh is like the "significant digits" of computer science

Asymptotic Analysis: Analysis of function behavior as its input approaches infinity

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

Remember our goals:

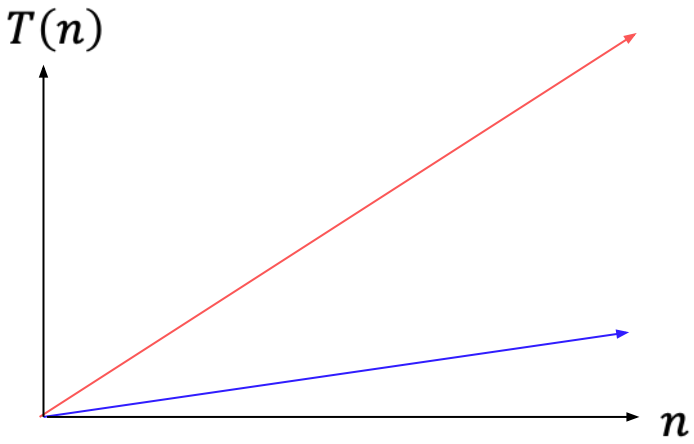
1. **Simple**
   We don't care about tiny differences in implementation, want the big picture result

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

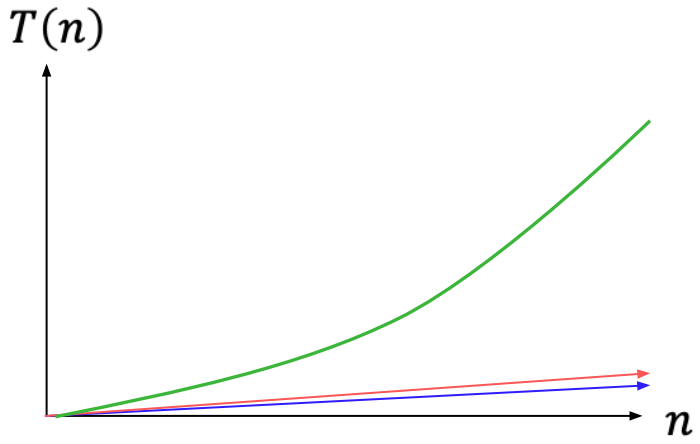
Function growth

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

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

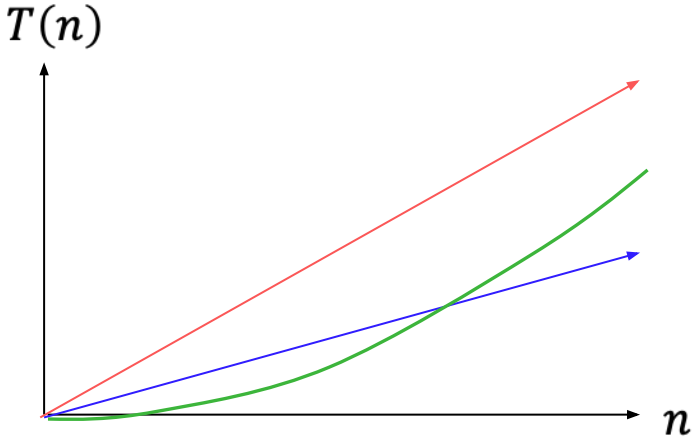


The growth rate for f(n) and g(n) looks very different for small numbers of input



...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:

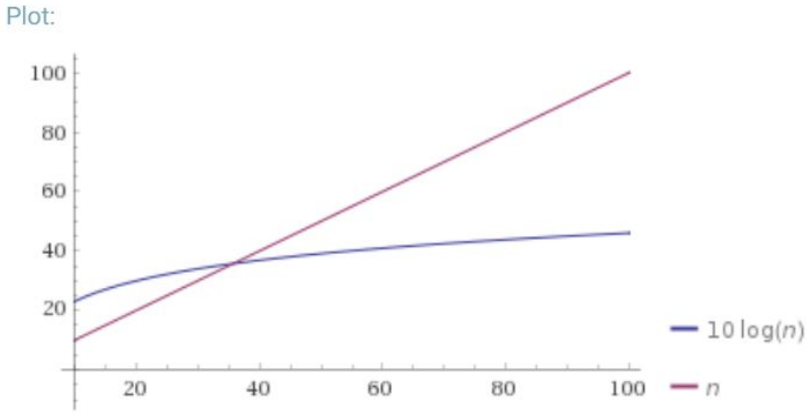
- We don't want to care about constant factors
- We only care about what happens as *n* gets larger

Big-O

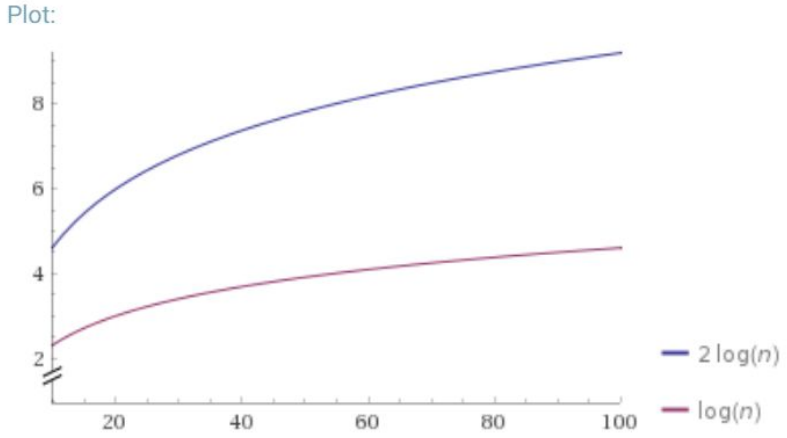
$f(n)$ is $O(g(n))$ if there exist positive constants c, n_0, such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$

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

Why n_0?

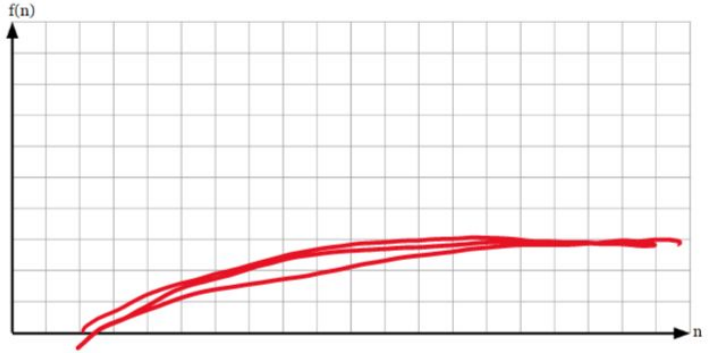


Why c?

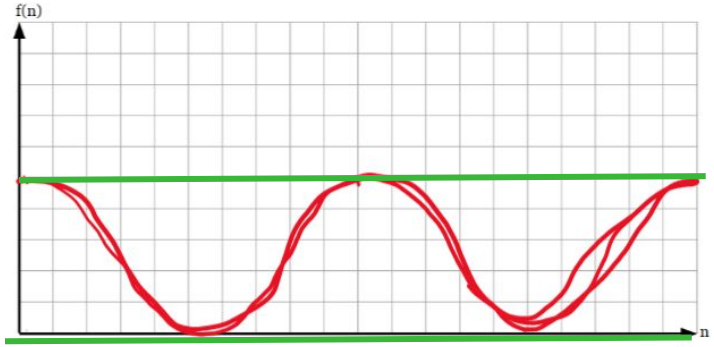


EX0: What's the Big O?

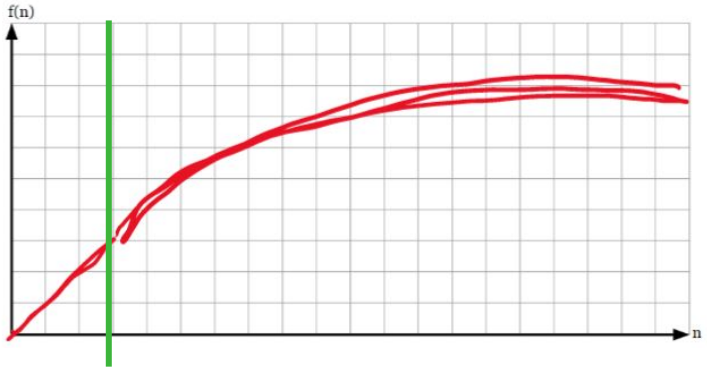
O(logn)
This graph appears to follow the pattern of logarithmic growth



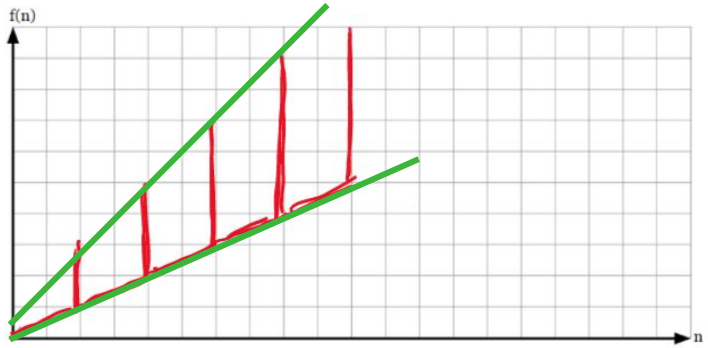
O(1)
Though this graph oscillates, the upper and lower bounds are constant



O(logn)
Though this graph has two different growth rates, we only count the one that tends to infinity



O(n²)
This graph appears to follow the pattern of quadratic growth



O(n)
Though this graph has different upper and lower bounds, they are both linear

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 {                       +4
            toTest++;                  +2
        }
    }
    return true;                       +1
}
```

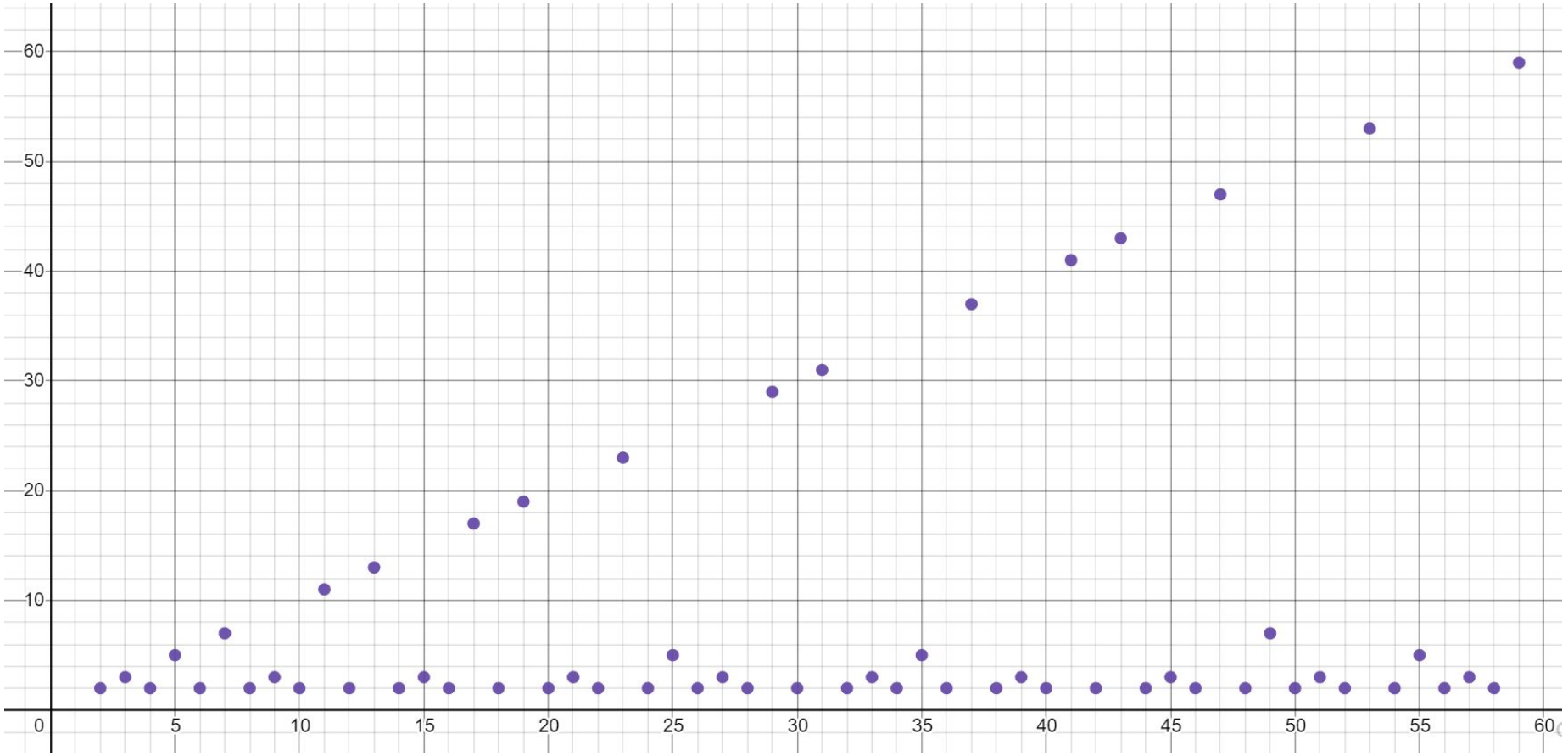
*n ... sometimes?

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

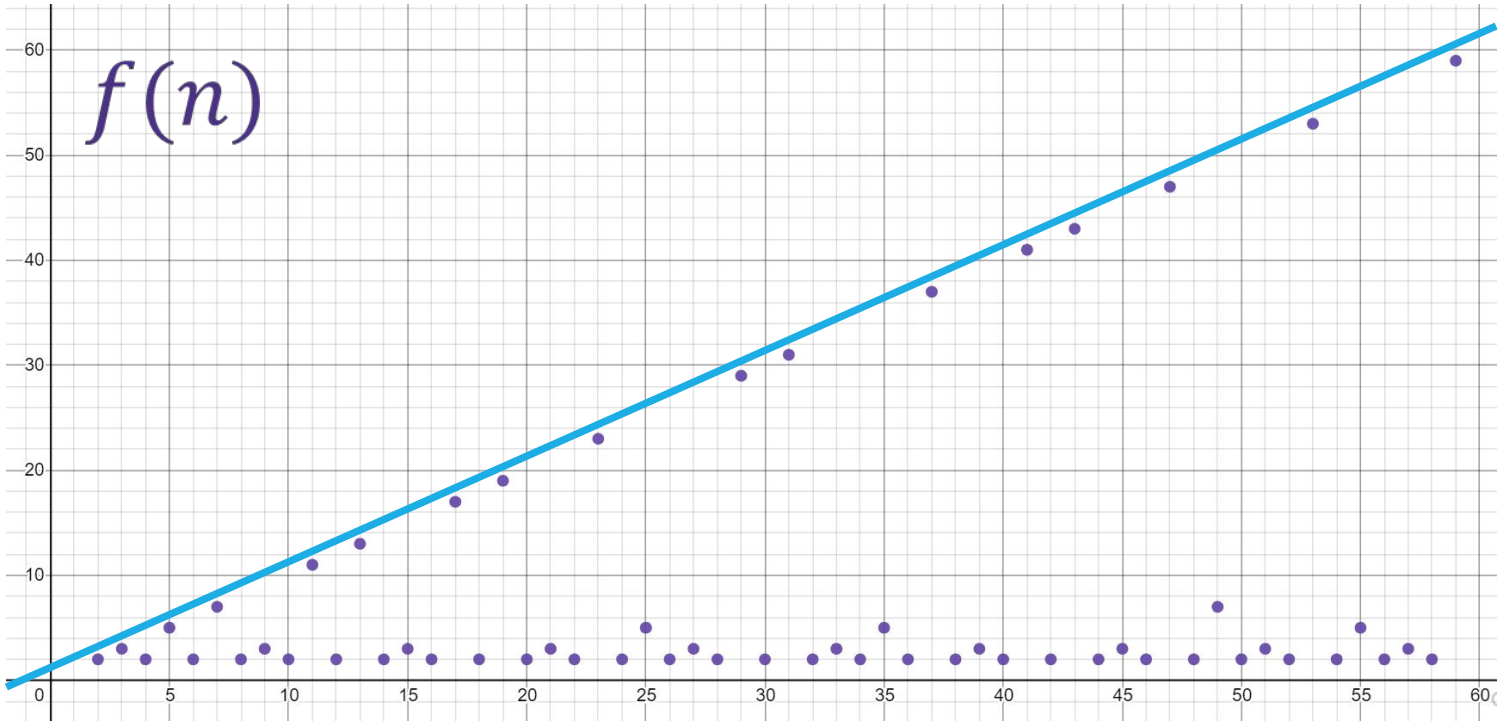


Is the running time of the code $O(1)$ or $O(n)$?

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 $n \geq 5$

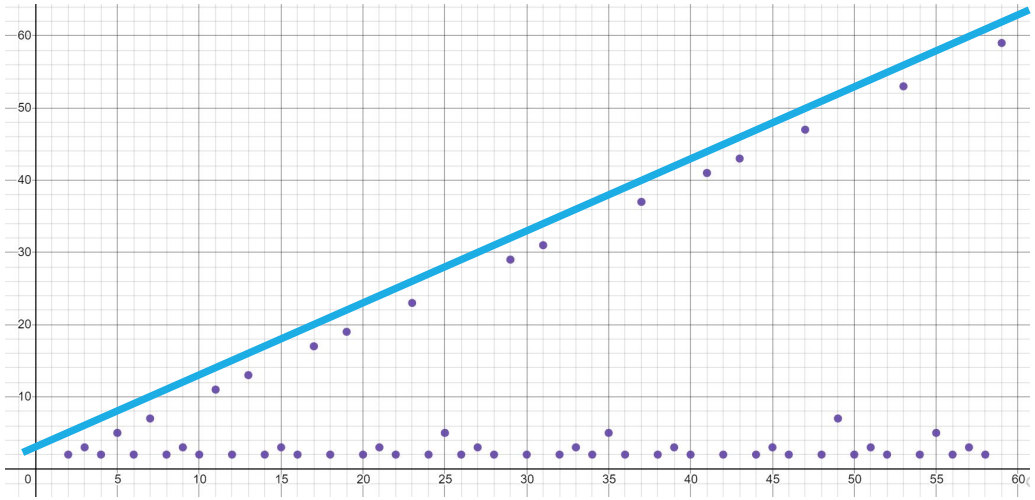
It's $O(n)$ but not $O(1)$

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

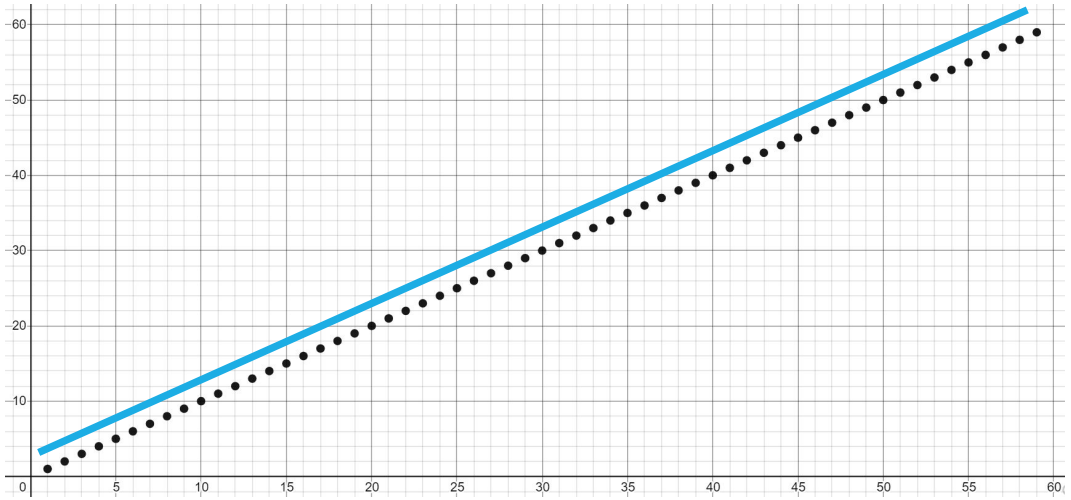
No! Choose your value of c. I can find a prime number k bigger than c.
And $f(k) = k > c \cdot 1$ so the definition isn't met

Big-O isn't everything

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



$O(n)$



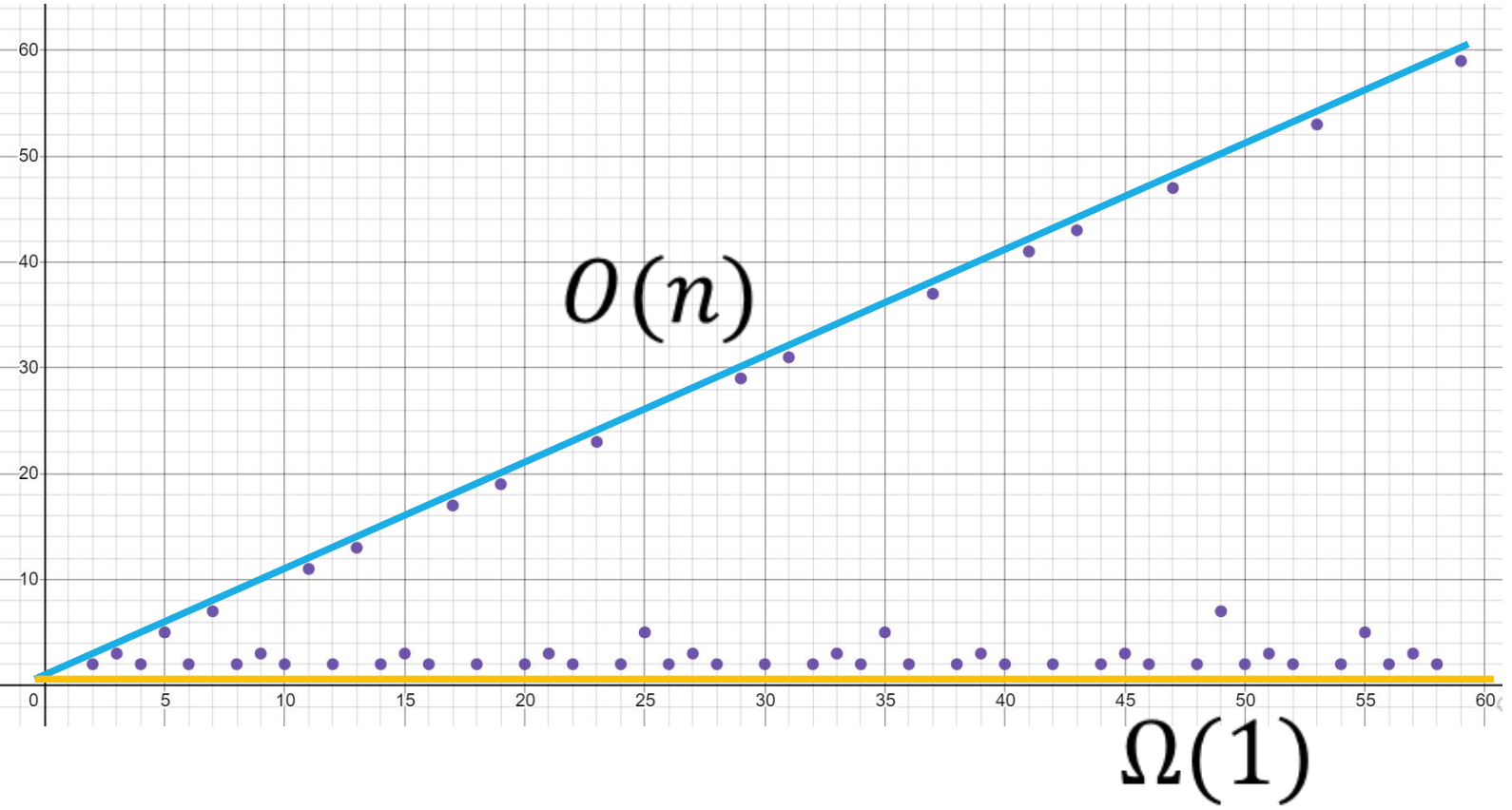
$O(n)$

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]



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

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).

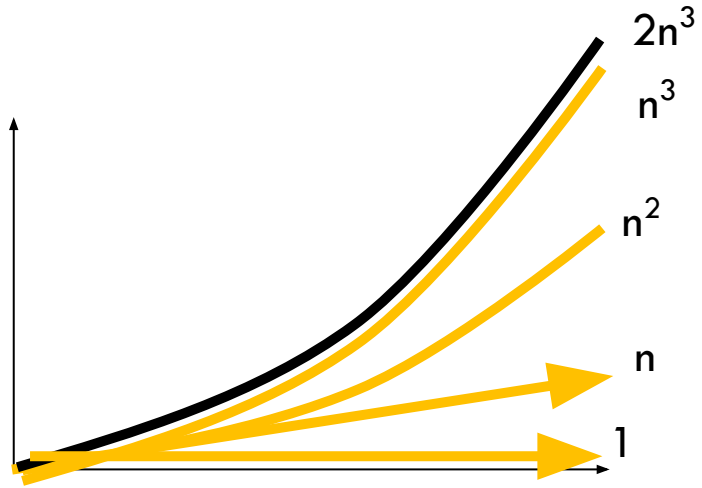
Big-Omega definition Plots

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

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

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

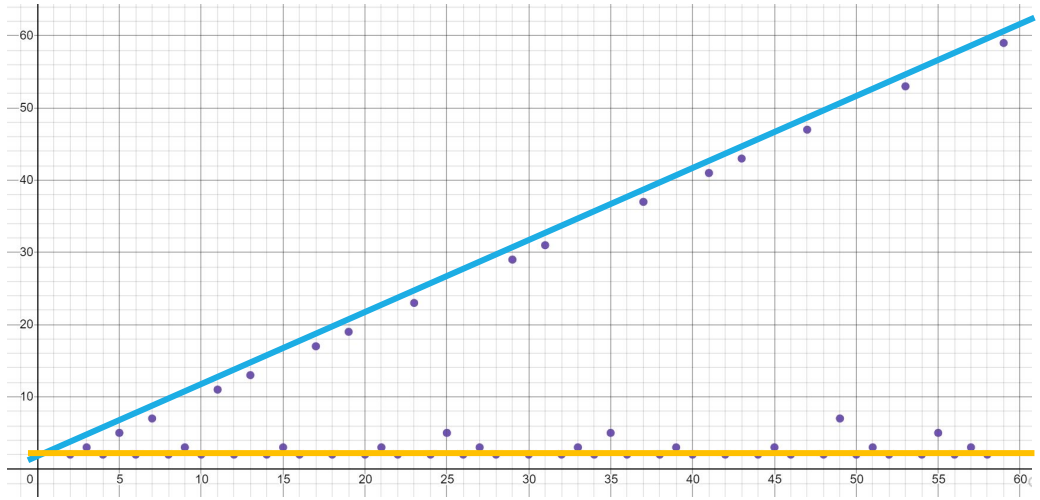
$2n^3$ is $\Omega(n^3)$



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

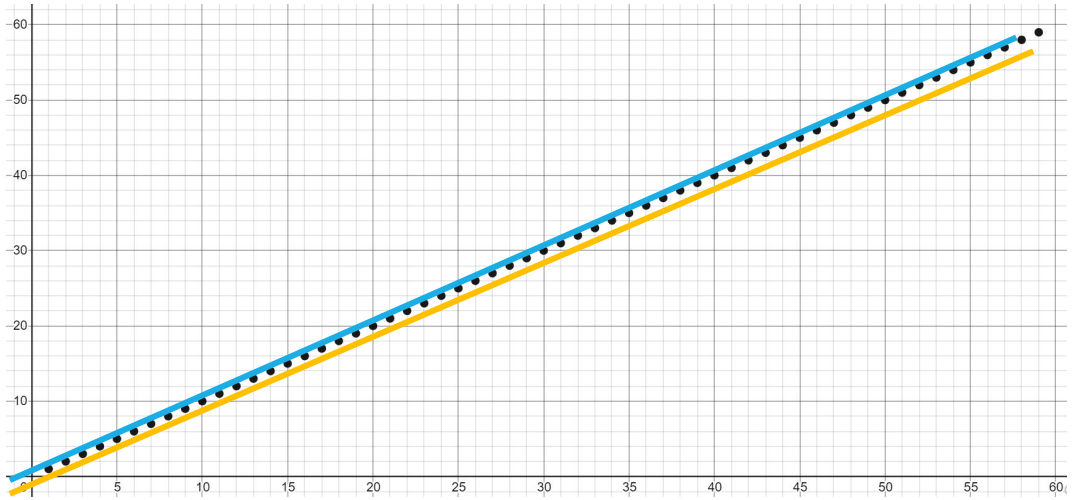
Big-O and Big-Ω shown together

prime runtime function



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

$f(n) = n$



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

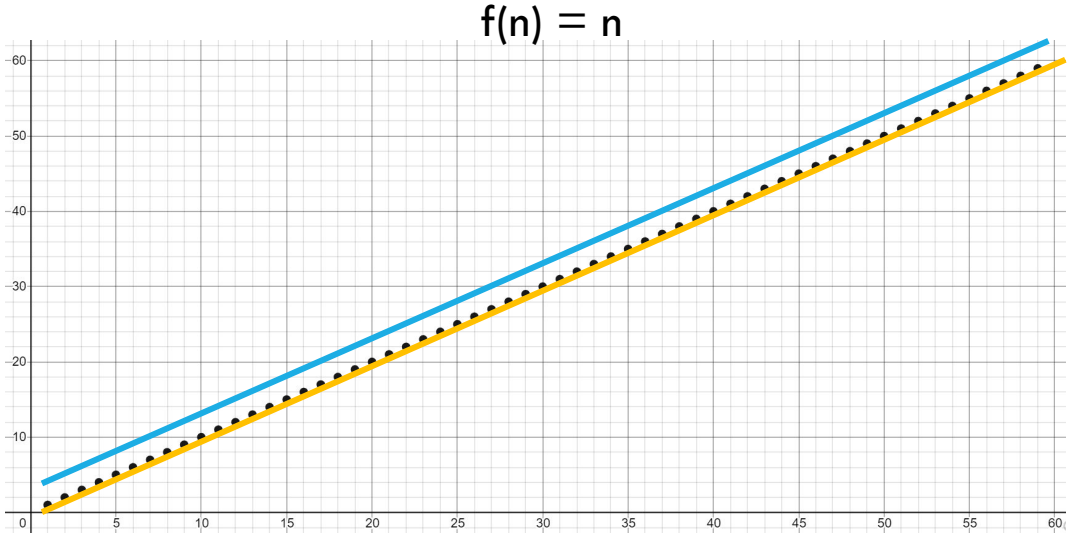
Note: this right graph's tight O bound is O(n) and its tight Omega bound is Omega(n). This is what most of the functions we'll deal with will look like, but there exists some code that would produce runtime functions like on the left.

Big-Theta

Big Theta is **"equal to"**

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

  *Except for constant factors and lower order terms



Big-Theta

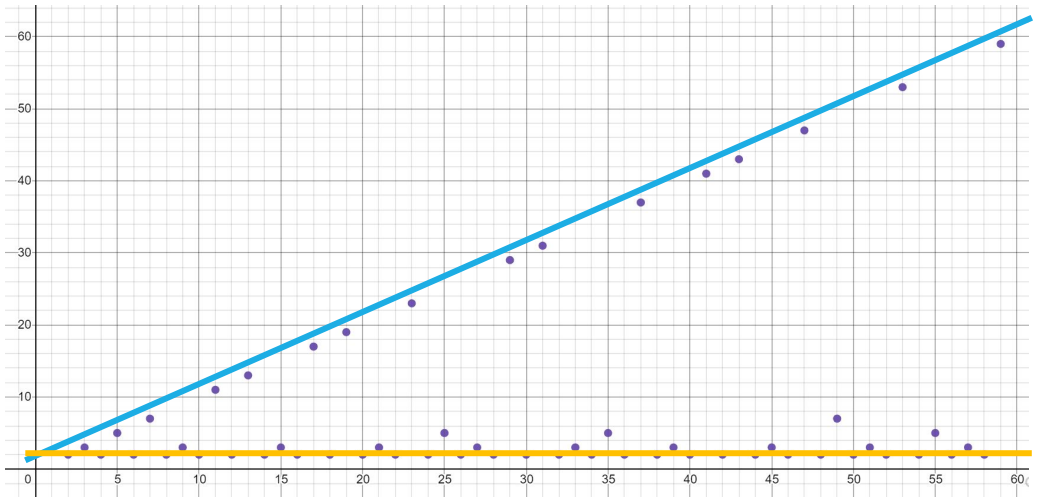
$f(n)$ is $\Theta(g(n))$ if
$f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.
In other words, there exist positive constants $c1$, $c2$, n_0 such that for all $n \geq n_0$
$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

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

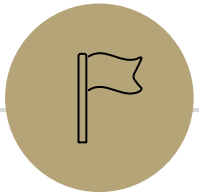
$f(n)$ is $O(g(n))$ if there exist positive constants c, n_0, such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$

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



That’s all!