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

Lecture 5: Asymptotic Analysis II

CSE 373: Data Structures and Algorithms

Warm Up

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++) {
            +2 System.out.println(list.get(0));
        }
    }
}
```

n(2)

n(n(2))

Possible answer
$T(n) = 4n^2$

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

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

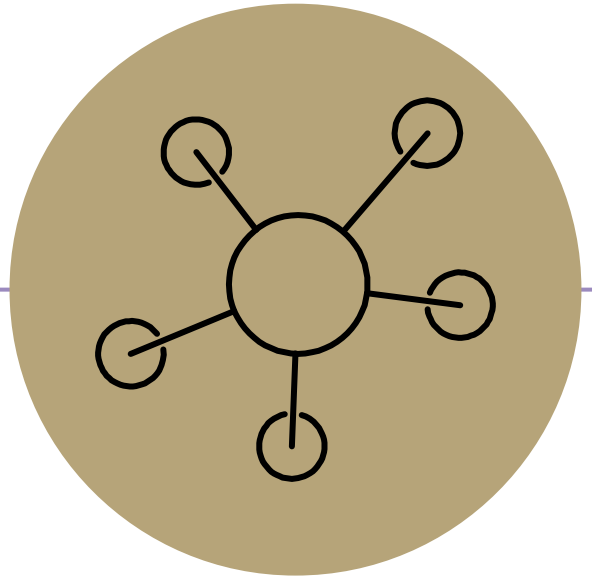
Proj 0 – 143 Review Project Due Tonight 11:59pm PST

Proj 1 Releases Tonight
- Partner Project!
- Due Wednesday April 14th

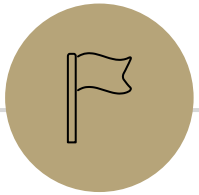
Partners
- Yes, 3 person groups are allowed
- Default is working alone
- Define your own partnerships and groups via Gradescope
- We can assign you a random partner

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

Lecture Questions Doc



Questions?



Iterators

Traversing Data

We could get through the data much more efficiently in the Linked List class itself.

```
Node curr = this.front;
while(curr!=null){
   System.out.println(curr.data);
   curr = curr.next;
}
```

What if the client wants to do something other than just print?

We should provide giving each element in order as a service to client classes.

```
for (T item : list) {
   System.out.println(item);
}
```

Iterator!

Review: Iterators

iterator: a Java interface that dictates how a collection of data should be traversed. Can only move in the forward direction and in a single pass.

Iterator Interface
behavior hasNext() – true if elements remain next() – returns next element

supported operations:

hasNext() – returns true if the iteration has more elements yet to be examined

next() – returns the next element in the iteration and moves the iterator forward to next item

```
ArrayList<Integer> list = new ArrayList<Integer>();
//fill up list

Iterator itr = list.iterator();
while (itr.hasNext()) {
   int item = itr.next();
}
```

```
ArrayList<Integer> list = new ArrayList<Integer>();
//fill up list

for (int i : list) {
   int item = i;
}
```

Implementing an Iterator

Usually: you'll have a private class for the iterator object.

That iterator class will have a class variable to remember where you are.

`hasNext()` – check if there's something left by examining the class variable.

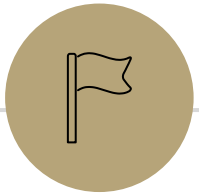
`next()` – return the current thing and update the class variable.

You have a choice:

- Variable might point to the thing you just processed
- Or the next thing that would be returned.

Both will work, one might be easier to think about/code up in some instances than others.

Punchline: Iterators make your client's code more efficient (which is what they care about)



Big O

Code to Big-Oh


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

143 general patterns: "O(1) constant is no loops, O(n) is one loop, $O(n^2)$ is nested loops"

- This is still useful!
- But in 373 we'll go much more in depth: we can explain more about *why*, and how to handle more complex cases when they arise (which they will!)

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 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

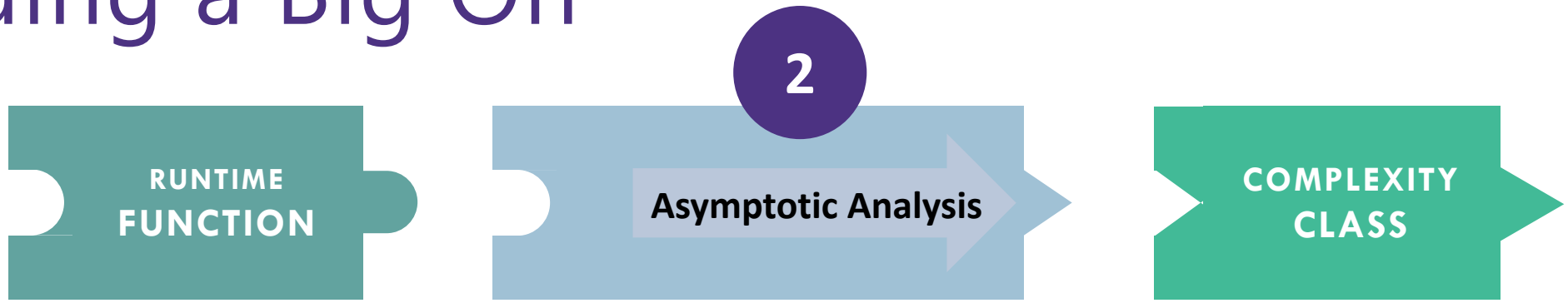
O(n)

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 Oh



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.

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

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

$\approx 9n^2$

$\approx n^2$

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

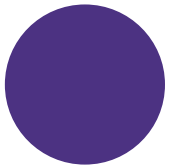
Can we really throw away all that info?

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

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

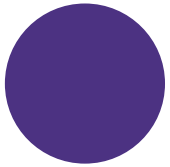
- We only care about what happens when n approaches infinity
- For small inputs, doesn't really matter: all code is "fast enough"
- Since we're dealing with infinity, constants and lower-order terms don't meaningfully add to the final result. The highest-order term is what drives growth!

Remember our goals:



Simple

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



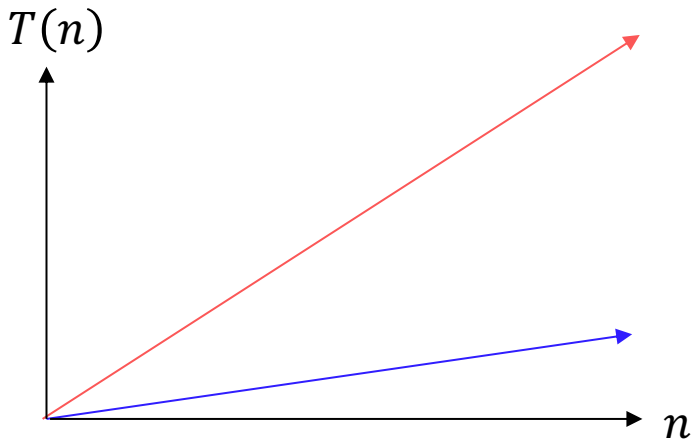
Decisive

Produce a clear comparison indicating which code takes "longer"

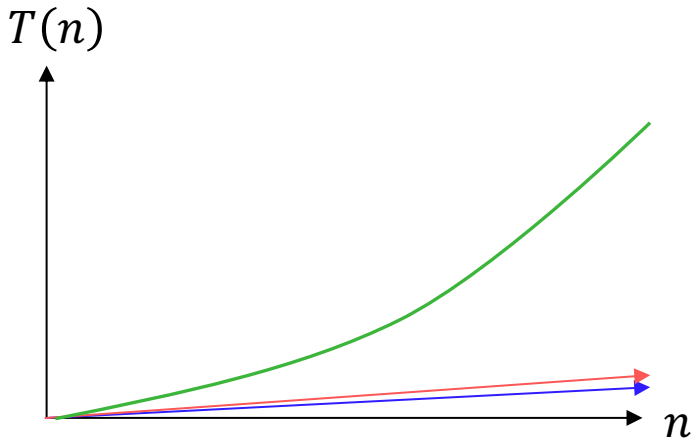
Function growth

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

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

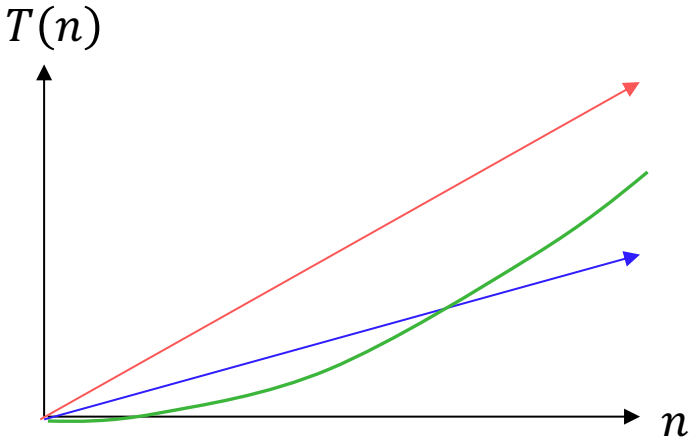


The growth rate for f(n) and g(n) looks very different for small numbers of input



...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

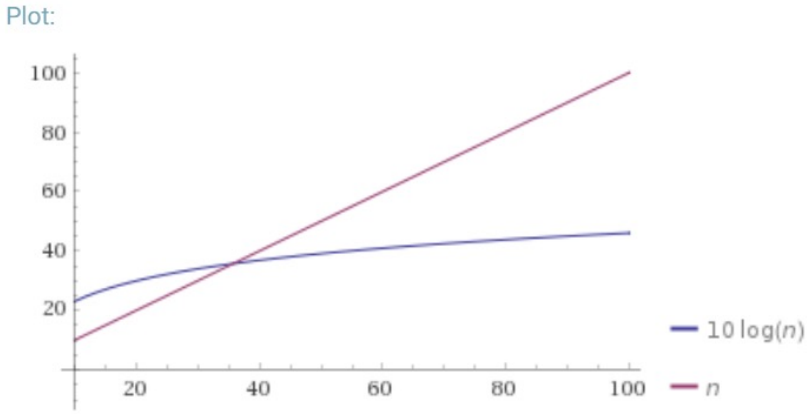
- We don't want to care about constant factors.
- We only care about what happens as n gets large.

Big-O

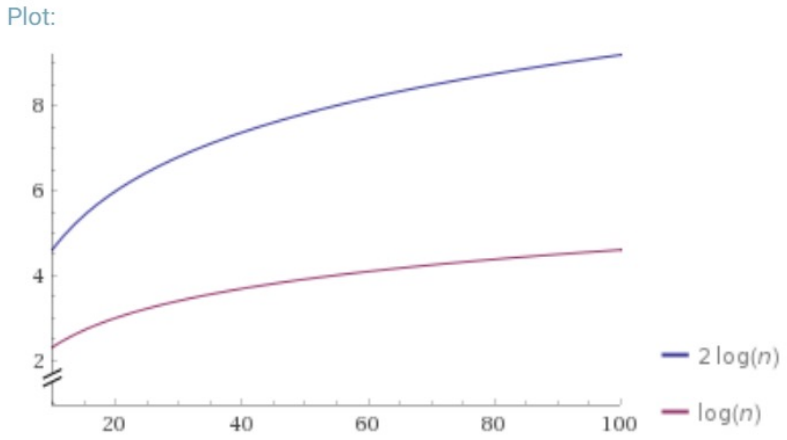
$f(n)$ is $O(g(n))$ if there exist positive constants c, n_0 such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$

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

Why n_0?



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

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

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

Add up all your truths

$10n + 15 \leq 10n + 15n = 25n$ 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

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 $c = ?$ $n >= ?$

$\boldsymbol{c = 14}$ & $\boldsymbol{n_0 = 1}$

Big-O

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

Writing Big-O proofs.

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.

Edge Cases

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. 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).

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

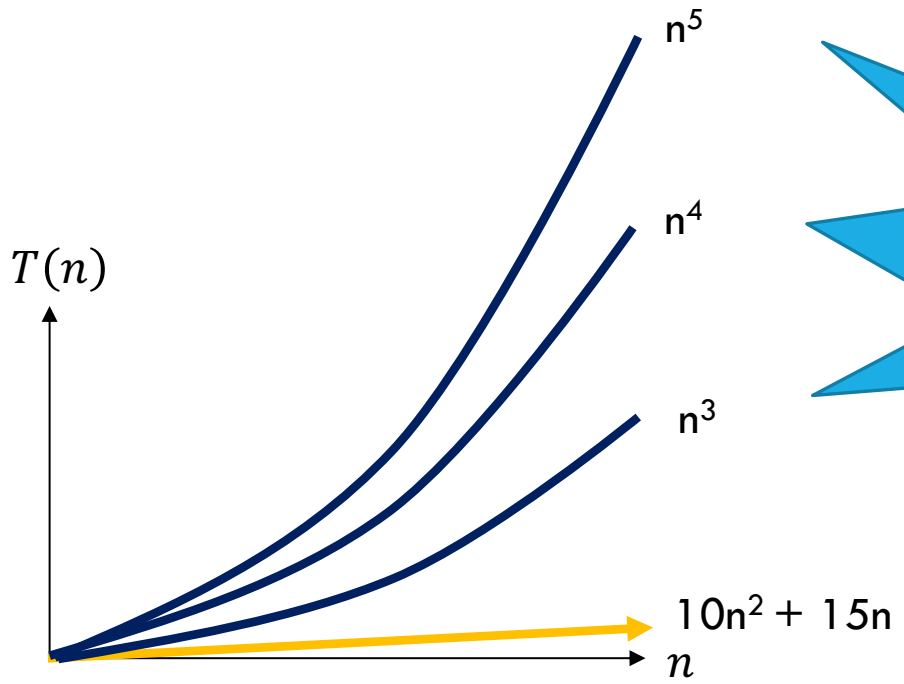
Note: Big-O definition is just an upper-bound, not always an exact bound (plots)

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.

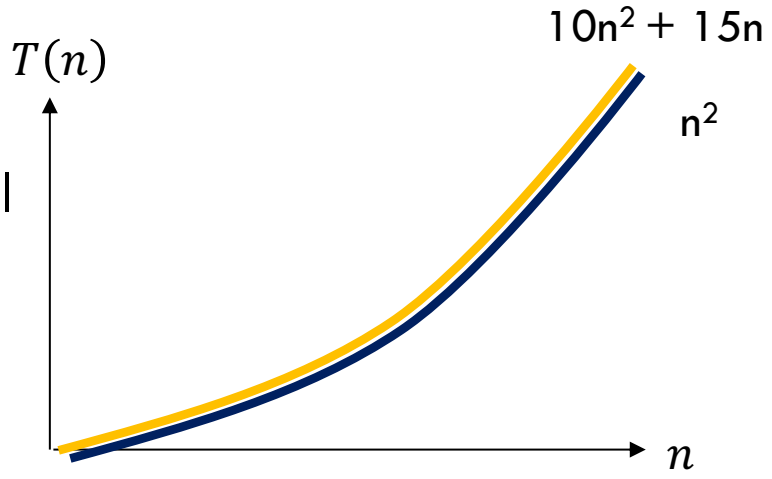
Computer scientists It is almost always technically correct to say your code runs in time $O(n!)$. (Warning: don't try this trick in an interview or exam)

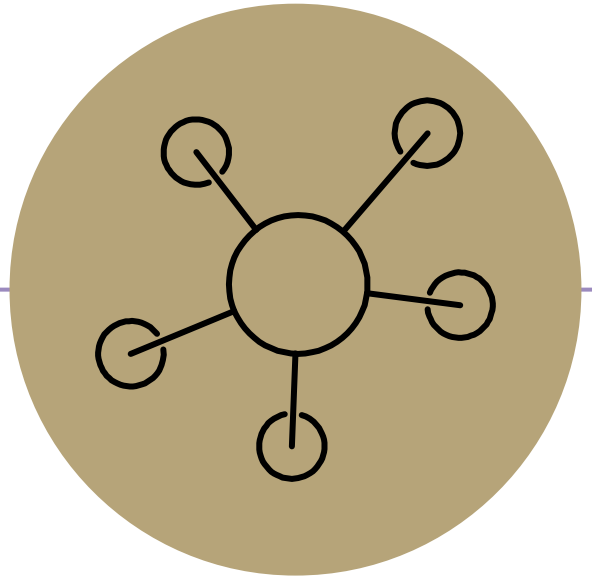
If you zoom out a bunch,

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. 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;
}
```

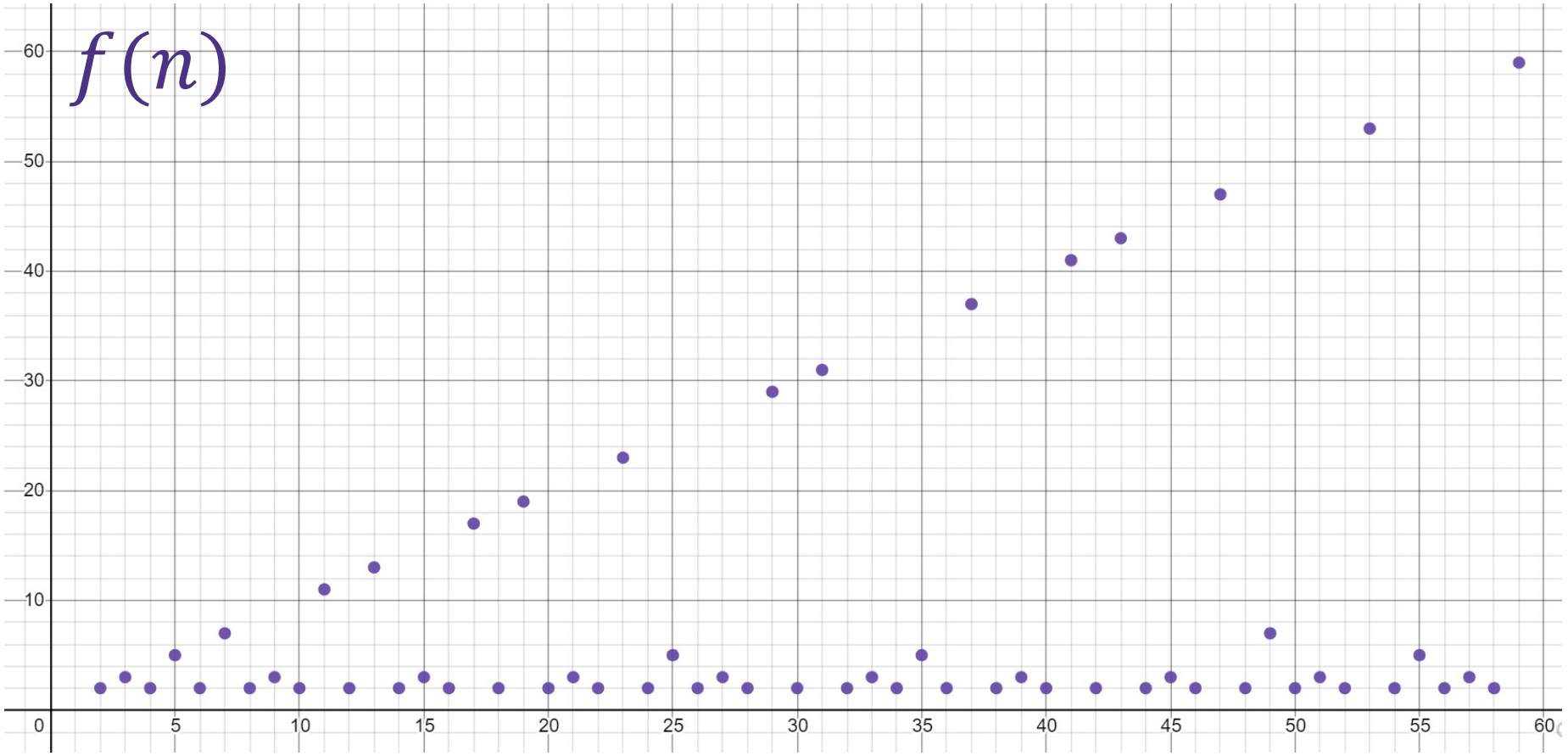
Remember, $f(n)$ = the number of basic operations performed on the input n.

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

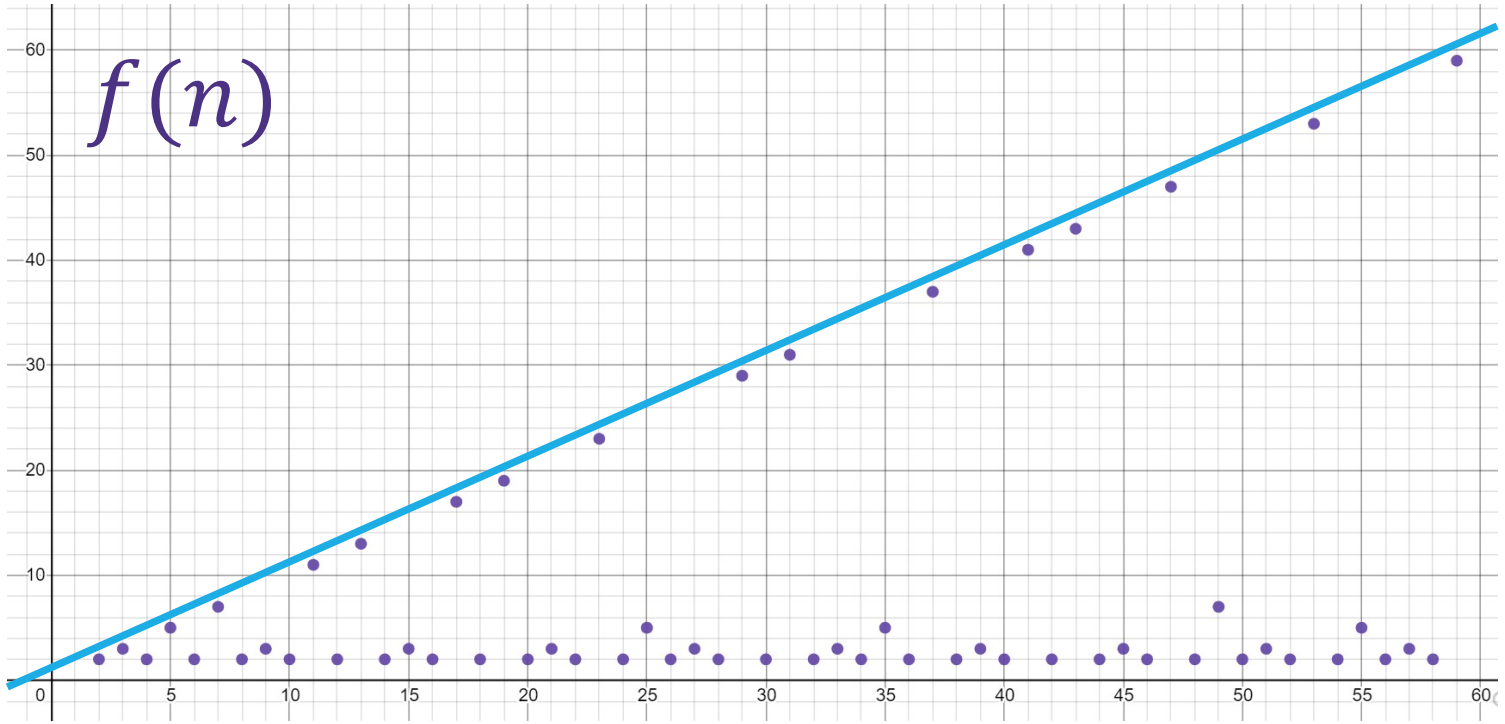


Is the running time of the code $O(1)$ or $O(n)$?

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

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

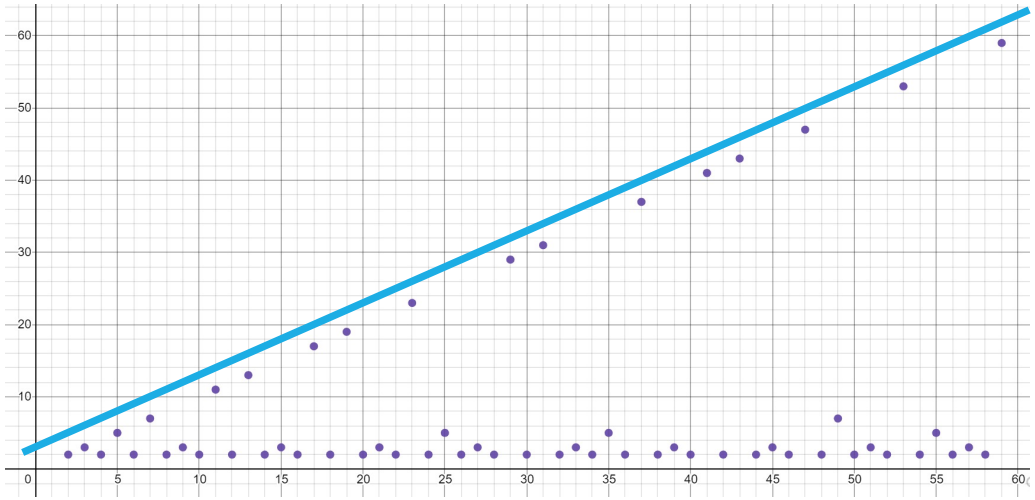
It's $O(n)$ but not $O(1)$

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

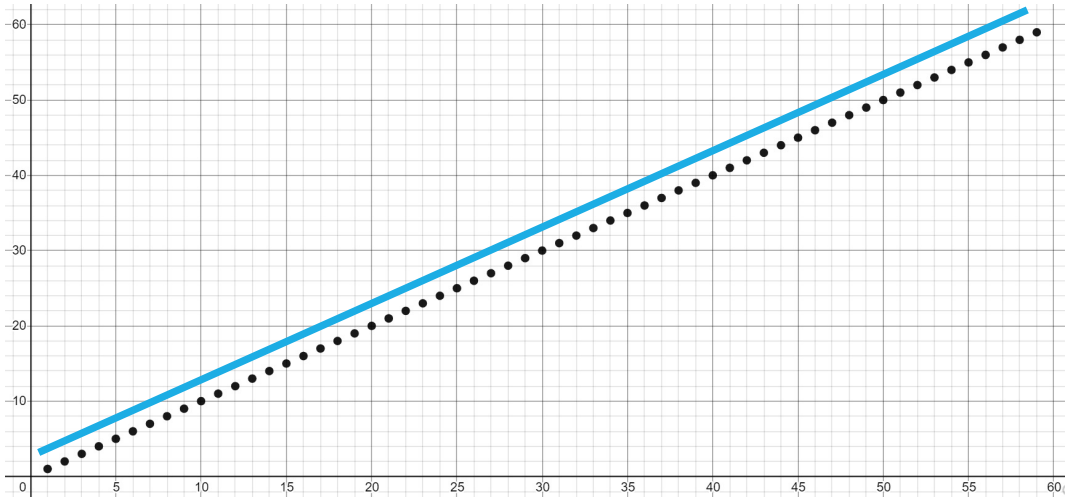
No! Choose your value of c. I can find a prime number k bigger than c.
And $f(k) = k > c \cdot 1$ so the definition isn't met!

Big-O isn't everything

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



$O(n)$



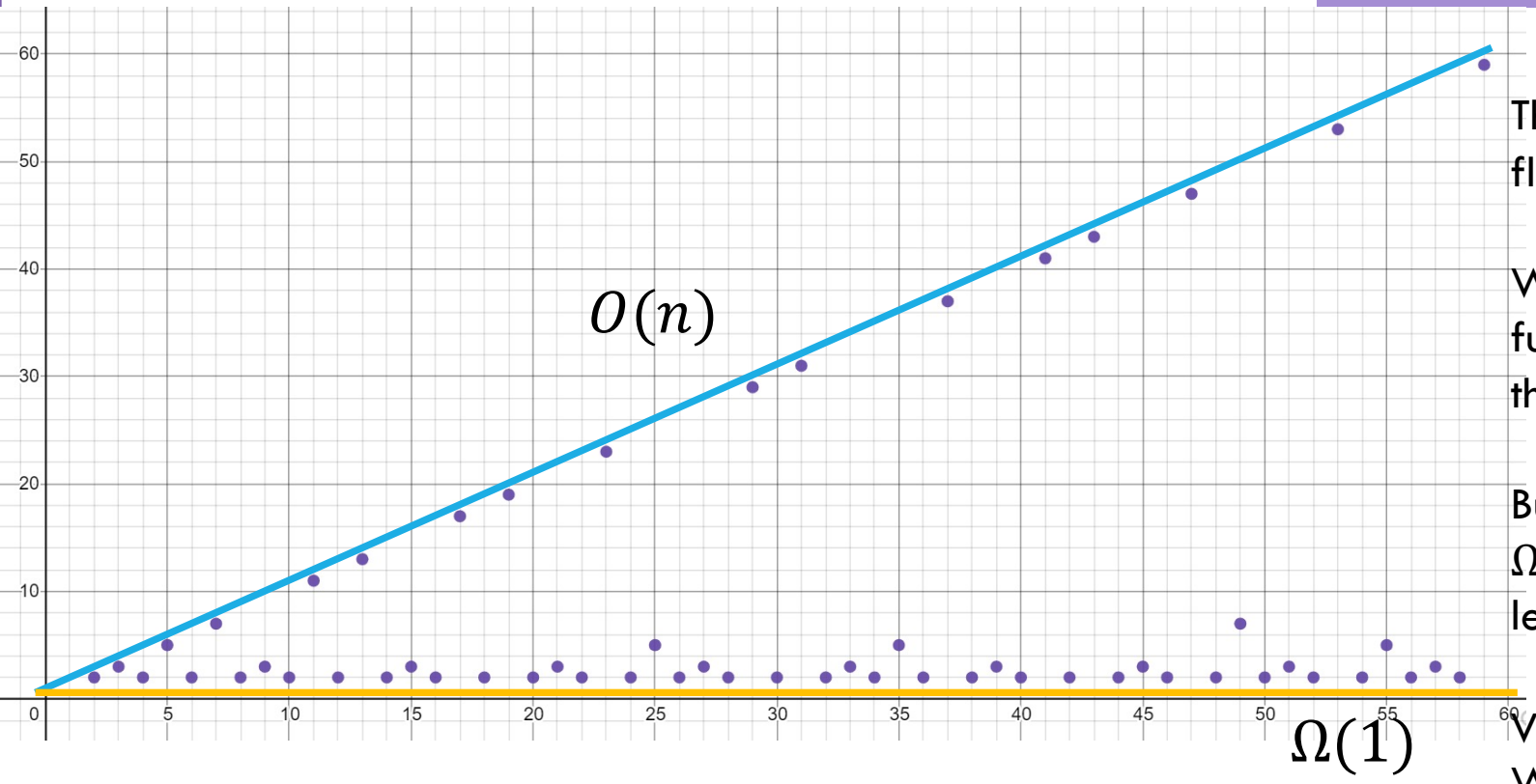
$O(n)$

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]

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?

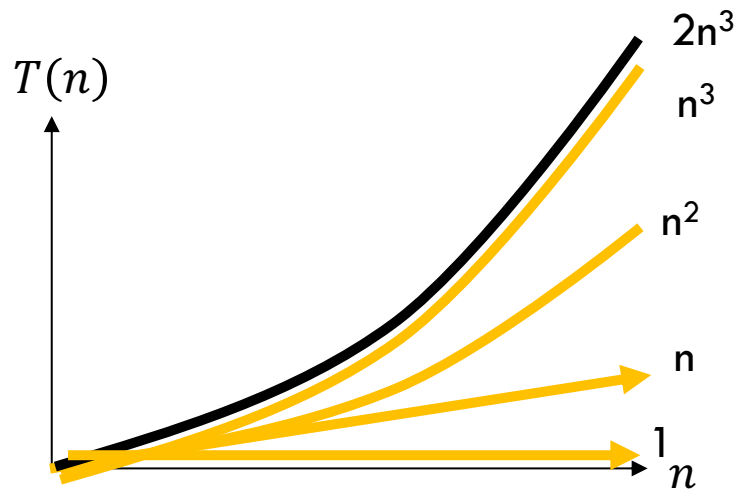
Big-Omega definition Plots

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

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

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

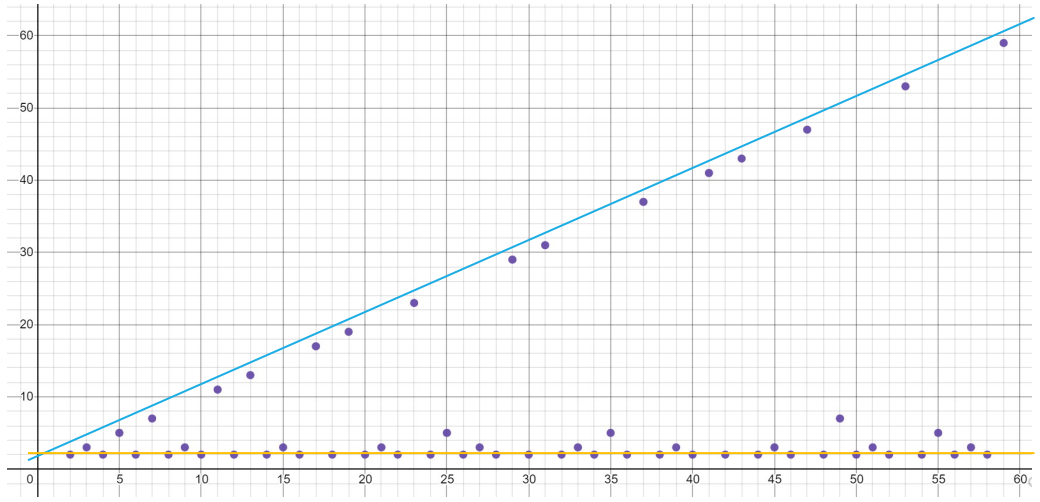
$2n^3$ is $\Omega(n^3)$



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

Big-O and Big-Ω shown together

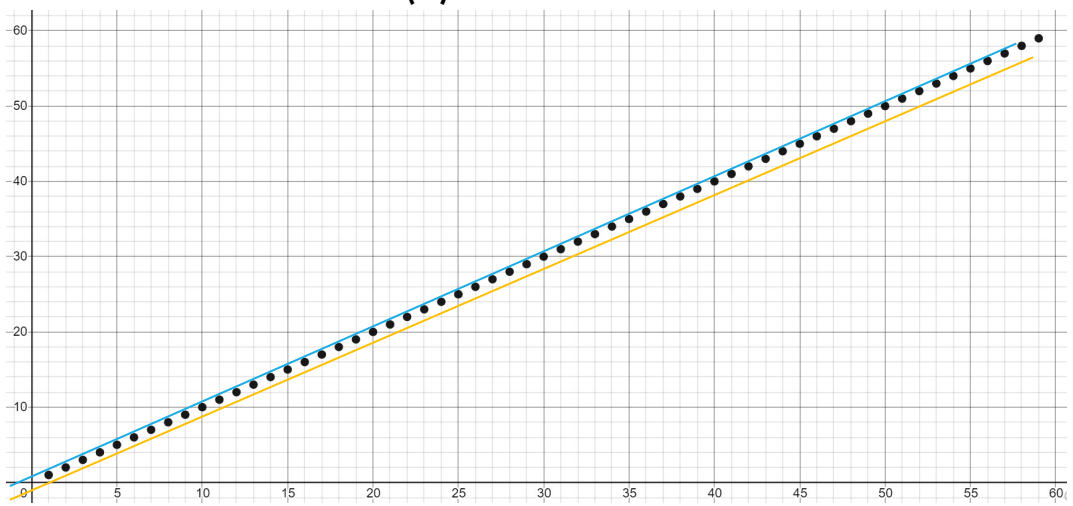
prime runtime function



$O(n)$

$\Omega(1)$

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.

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

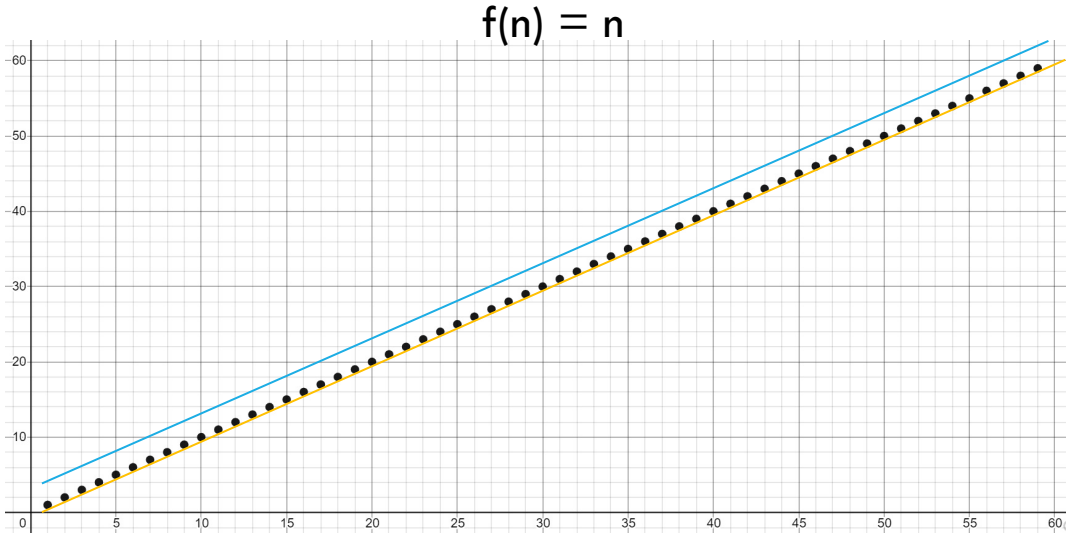
$f(n)$ is $\Theta(g(n))$ if
$f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.
(**in other words:** there exist positive constants $c1, c2, n_0$ such that for all $n \geq n_0$)

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

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 $\Omega(g(n))$.