Lecture 4: Formal Big-O, Omega and Theta

CSE 373: Data Structures and Algorithms

Adminstrivia

Project 0 due Wednesday

Fill out the Project 1 partner form **today!**

Everyone needs to fill out the form

- If you have a partner, **both of you** need to fill out the form.
- If you don't, you need to tell us if you want us to assign you one, or if you want to work alone.

Please fill out class survey

4 Minutes

Warm Up

Construct a mathematical function modeling the runtime for the following functions

```
public void mystery1(ArrayList<String> list) {
   for (int i = 0; i < 3000; i++) {
      for (int j = 0; j < 1000; j++) {
         int index = (i + j) % list.size();
         System.out.println(list.get(index));
      }
      for (int j = 0; j < list.size(); j++) {
         System.out.println(":)");
      }
   }
}
```

+4
+2
1000(8)
3000(1000(8) + n(1) + 2)
+1
n(1)

Possible answer
T(n) = 3000 (8002 + n)

Approach

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

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

```
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));
      }
   }
}
```

+4
n(4)
n(4n(4))

Possible answer
$T(n) = 16n^2$

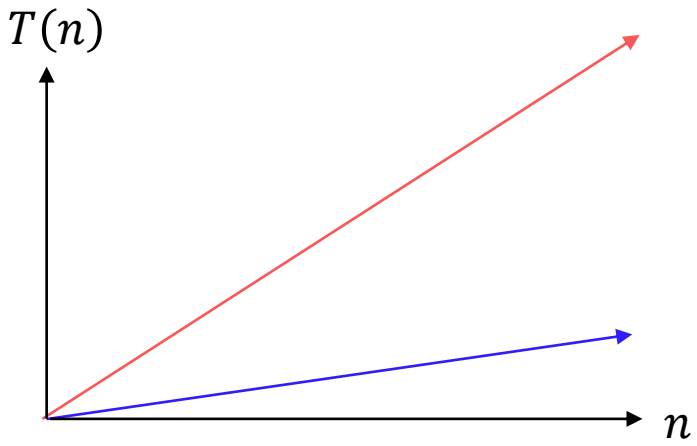
pollEV.com/cse373su19

What is the big-O for the code model for mystery1?

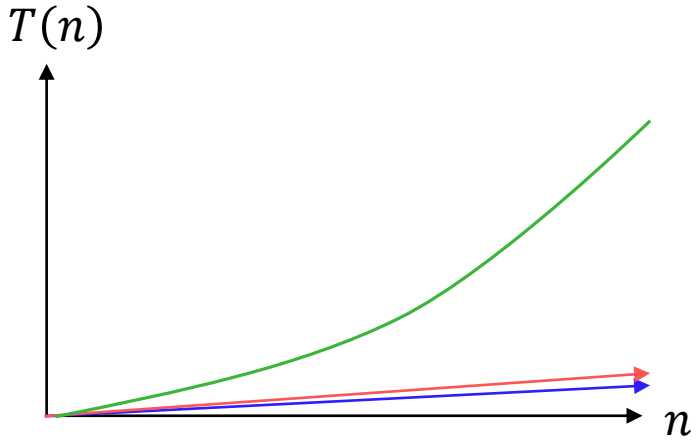
Function growth

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

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

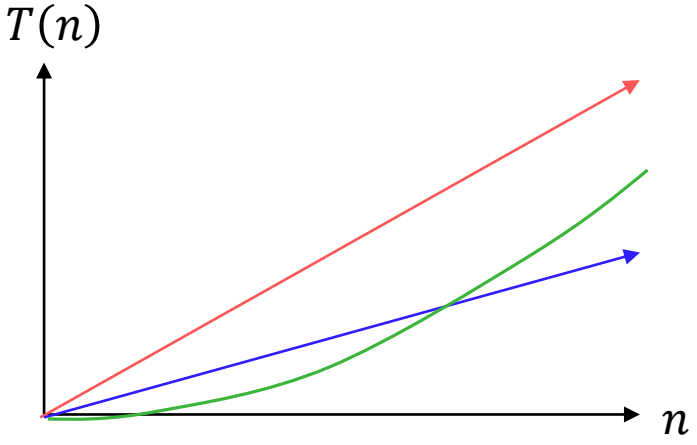


The growth rate for f(n) and g(n) looks very different for small numbers of input



...but since both are linear eventually look similar at large input sizes

whereas h(n) has a distinctly different growth rate

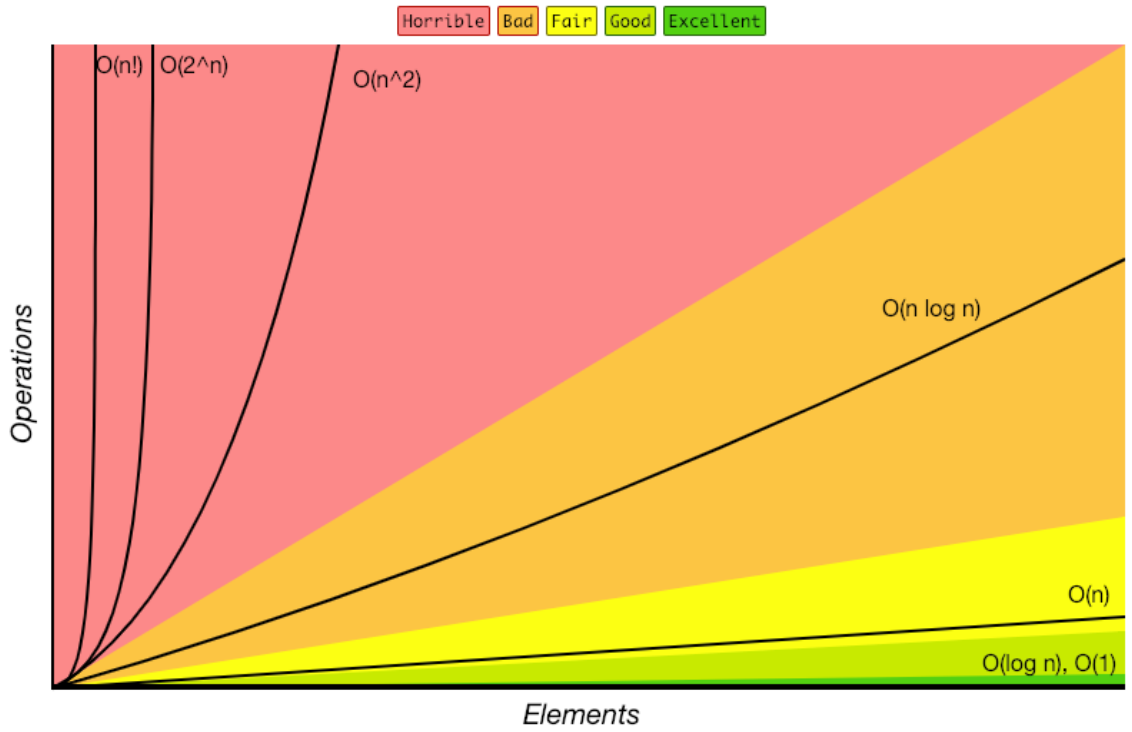


But for very small input values h(n) actually has a slower growth rate than either f(n) or g(n)

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(nlog 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/

Formal Definitions: Why?

You might already intuitively understand what big-O means.

At the very least, you know how to go from a code model to the big-O

Who needs a formal definition anyway?

We do!

Think of your intuitive understanding as your internal sense of direction and map of the world.

And the formal definition as the google maps map of the world.

Formal Definitions: Why?

If you're walking around an area you're familiar with, you just need an internal sense of direction – you don't waste your phone battery for the "official" map.

If you're analyzing simple code – similar to the kind you've analyzed before, you don't bother with the formal definition, and just use your intuitive definition.

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 google maps, so if you get lost (come across a weird edge case) you know how to get your bearings.

Definition: Big-O

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

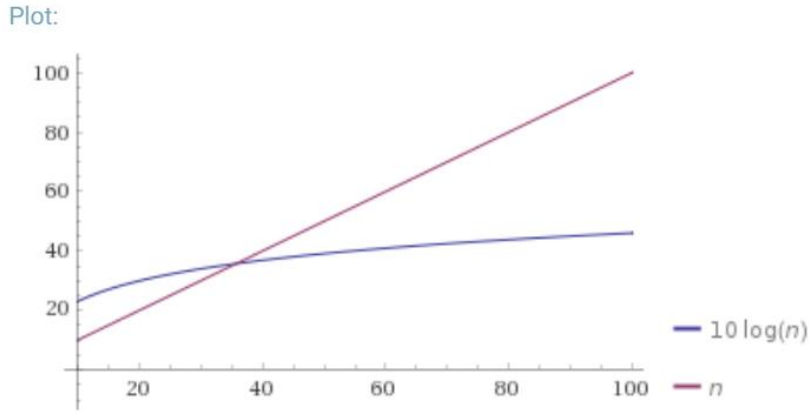
- We don't want to care about constant factors.
- We only care about what happens as n gets large.

Big-O

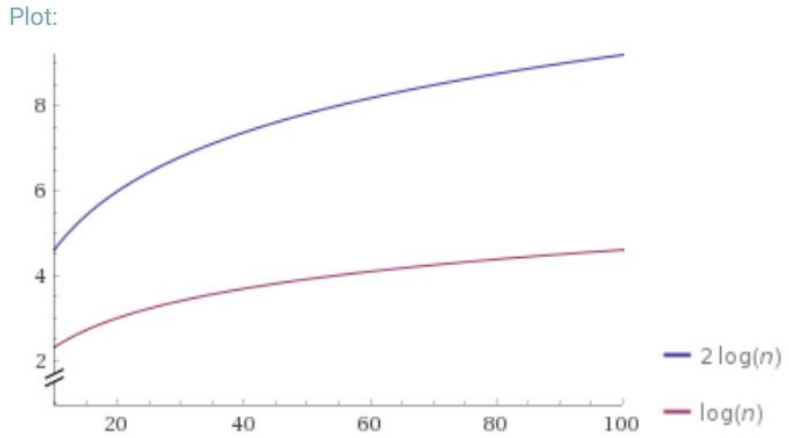
$f(n)$ is $O(g(n))$ if there exist positive constants c, n_0 such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$

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 $O(n)$

3 Minutes

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.

1 Minute

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).

Uncharted Waters

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;
}
```

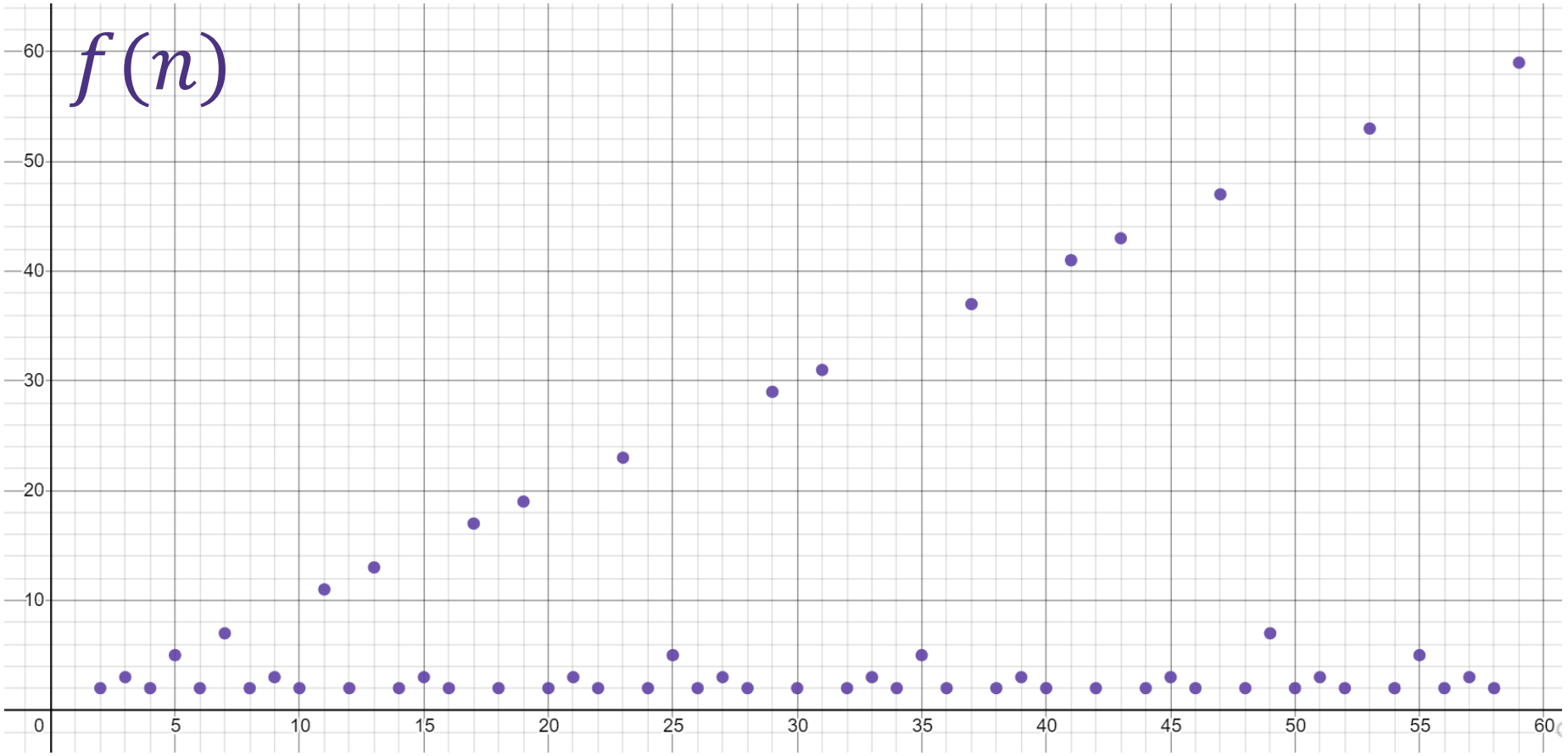
Remember, $f(n)$ = the number of basic operations performed on the input n.

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

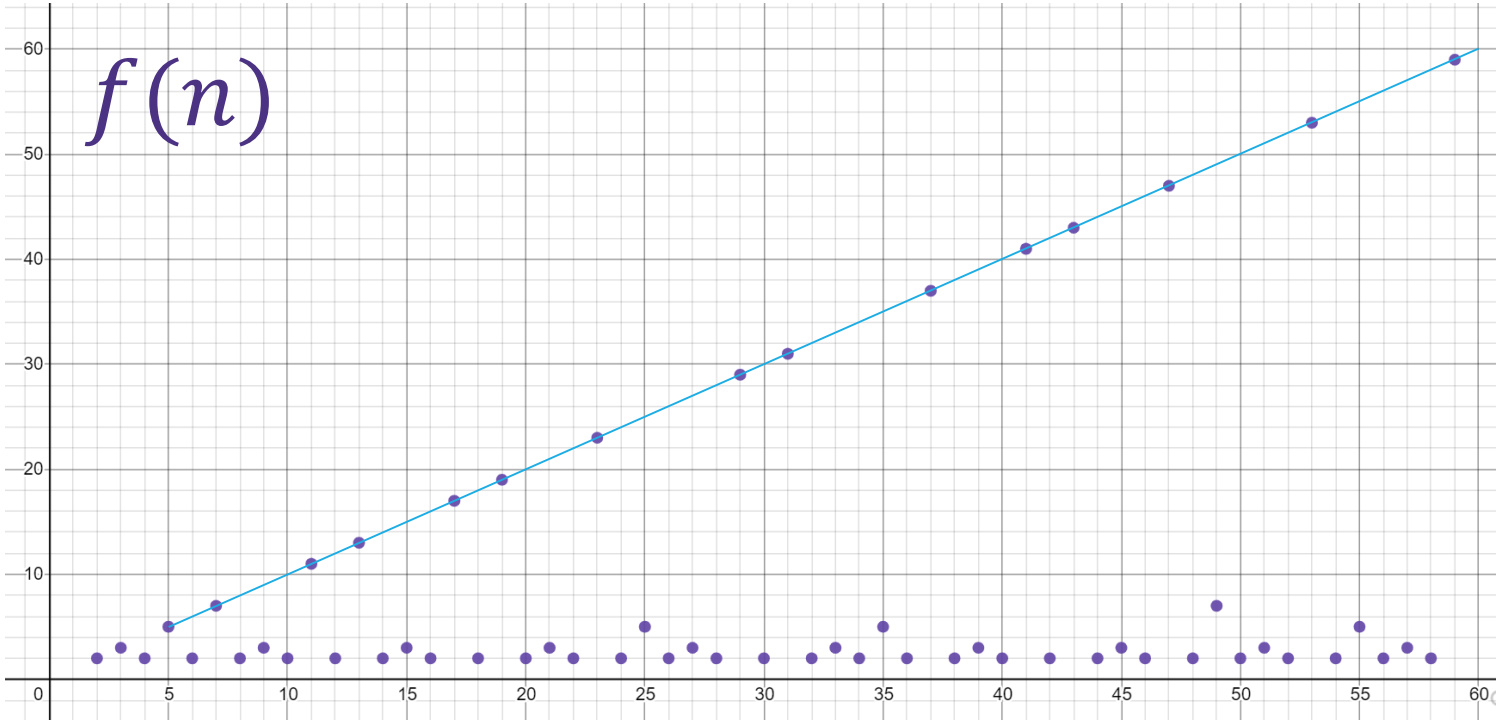


Is the running time of the code $O(1)$ or $O(n)$?

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 $n \geq 5$

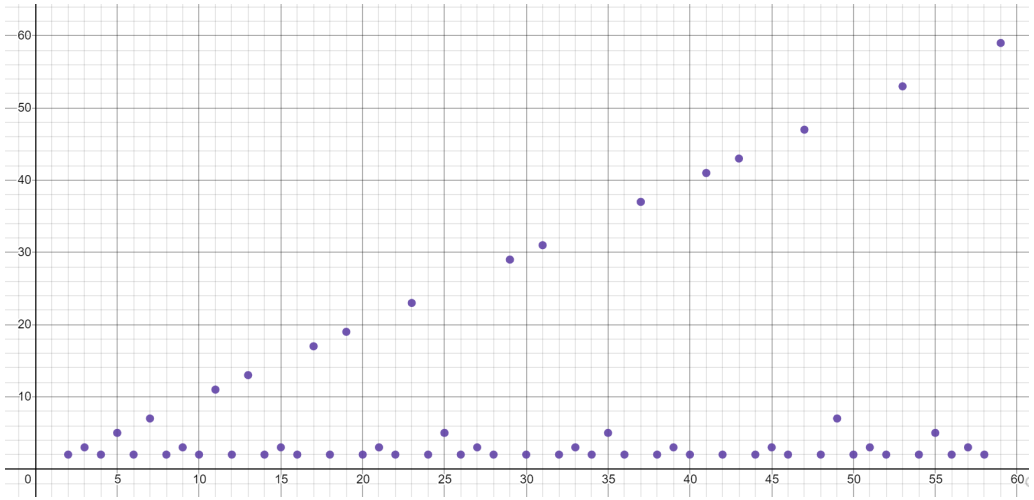
It's $O(n)$ but not $O(1)$

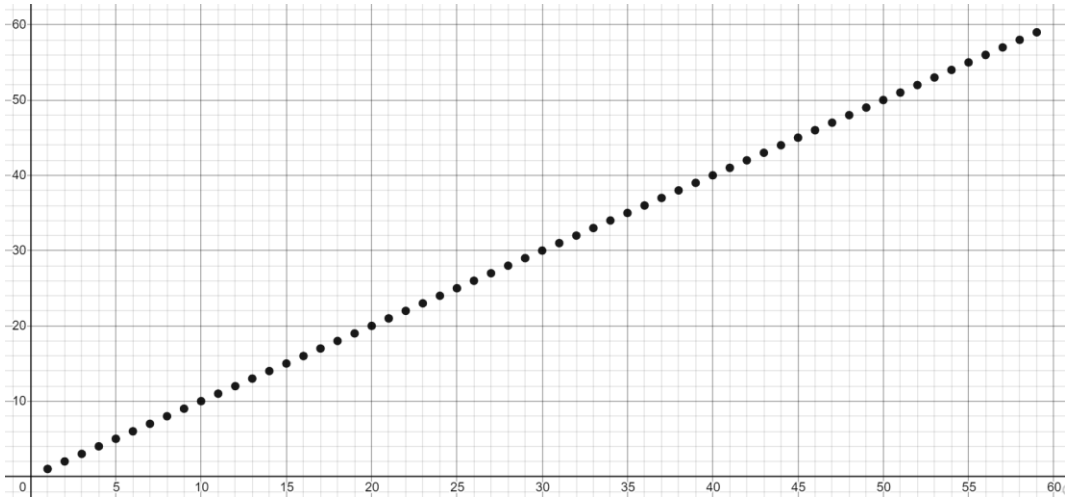
Is the running time $O(1)$?
Can you find constants c and n_0?

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.



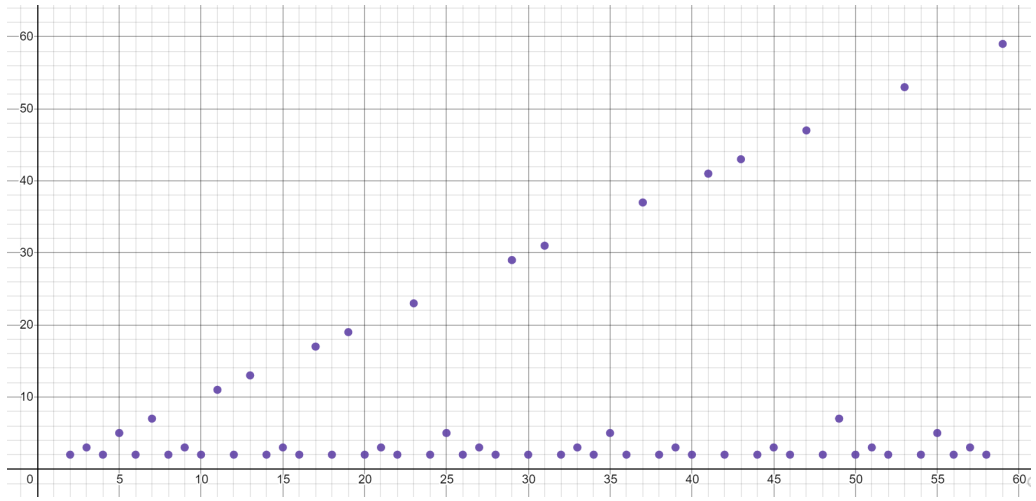


Your experience running these two pieces of code is going to be very different.
It's disappointing that the $O()$ are the same – that's not very precise.
Could we have some way of pointing out the list code always takes AT LEAST n operations?

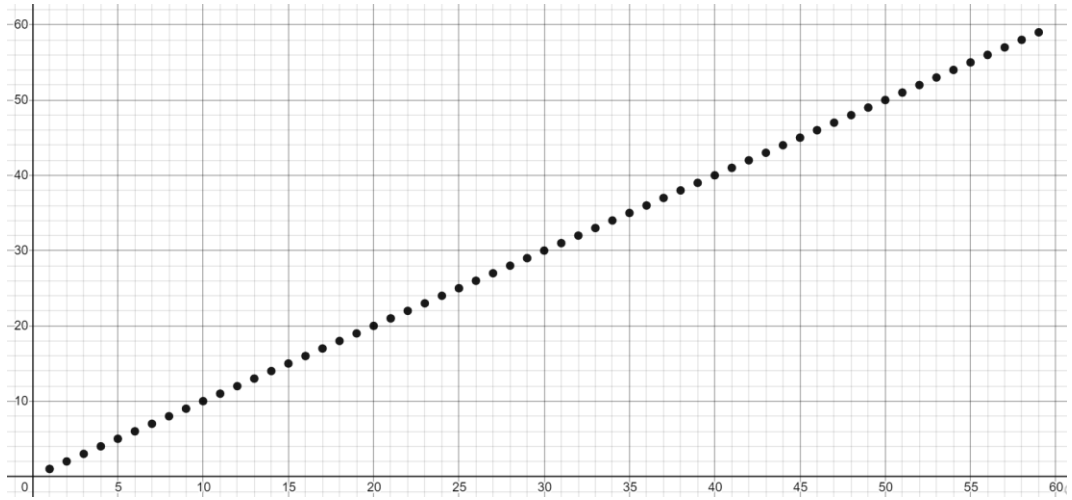
Big-Ω [Omega]

Big-Omega

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



$f_1(n)$ is not $\Omega(n)$



$f_2(n)$ is $\Omega(n)$

For any c, n_0 you suggest, I'll take k to be an even number such that $k > 1/c$. Then

$$f(k) = 1 = \frac{c}{c} < ck = c \cdot g(k)$$

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 $\Omega(g(n))$.

Viewing O as a class

Sometimes you'll see big-O defined as a family or set of functions.

Big-O (alternative definition)

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

For that reason, we sometimes write $f(n) \in O(g(n))$ instead of "$f(n)$ is $O(g(n))$".
Other people write "$f(n) = O(g(n))$" to mean the same thing.
The set of all functions that run in linear time (i.e. $O(n)$) is a "complexity class."
We never write $O(5n)$ or $O(n+1)$ instead of $O(n)$ – they're the same thing!

It's like writing $\frac{6}{2}$ instead of 3. It just looks weird.

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 $\Omega(g(n))$.

3 Minutes

Practice

5n + 3 is $O(n)$ True

n is $O(5n + 3)$ True

$5n + 3 = O(n)$ True

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

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

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

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

$n^2 \in O(n^{100})$ 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 $\Omega(g(n))$.

Simplified, tight big-O

Why not always just say $f(n)$ is $O(f(n))$.

It's always true! (Take $c = 1, n_0 = 1$).

The goal of big-O/Ω/Θ is to group similar functions together.

We want a simple description of f, if we wanted the full description of f we wouldn't use O

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. $\log(\log(n))$, $n \log n$, $(\log(n))^2$)

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.