Lecture 5: Big-O and Cases

CSE 373 – Data Structures and Algorithms

Administrivia

No section tomorrow (Independence Day)

Friday's lecture will be primarily lead by TAs as a replacement section.

- Will still be recorded.

Practicing Big-O stuff!

Project 0 due tonight at 11:59 PM.

If you want to use a late day, fill out the form (linked on projects section of webpage).

Project 1 out very soon!

- Check you email (and spam folder) for a link to your gitlab repo.
- Make sure you know who your partner is (go to repo, settings [in left sidebar], members. Will see course staff as "owners" and you/your partner as "maintainers"
- Forget to fill out partner form/something went wrong? Make a private post on piazza.

Project notes

How to effectively work on partner projects:

Pair program! See the document on the webpage.

- Two brains is better than one when debugging
- We expect you to understand the full projects, not just half of the projects.

Meet in-person with your partner.

Please don't:

- Come to office hours and say "my partner wrote this code, I don't understand it. Please help me debug it."
- Just split the project in-half and each do half (or alternate projects)
- Be mean to your partner.

Double check runners! (sometimes the checkmark appears when it shouldn't)

Passing the runners does not guarantee full credit.

Project Notes

Start early!

Where Are We?

We're using big-O to analyze code.

So far:

- Going from a set of code, to a code model $f(n)$.
- From $f(n)$, find the big-O,
- Formally prove big-O

Today

- More big-O-like tools: what are Ω, Θ?
- How do we model code with complicated if/else branches?

Next week

- Recursive code

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(n % toTest == 0)
                  return false;
            else
                  toTest++;
      }
      return true;
}
```

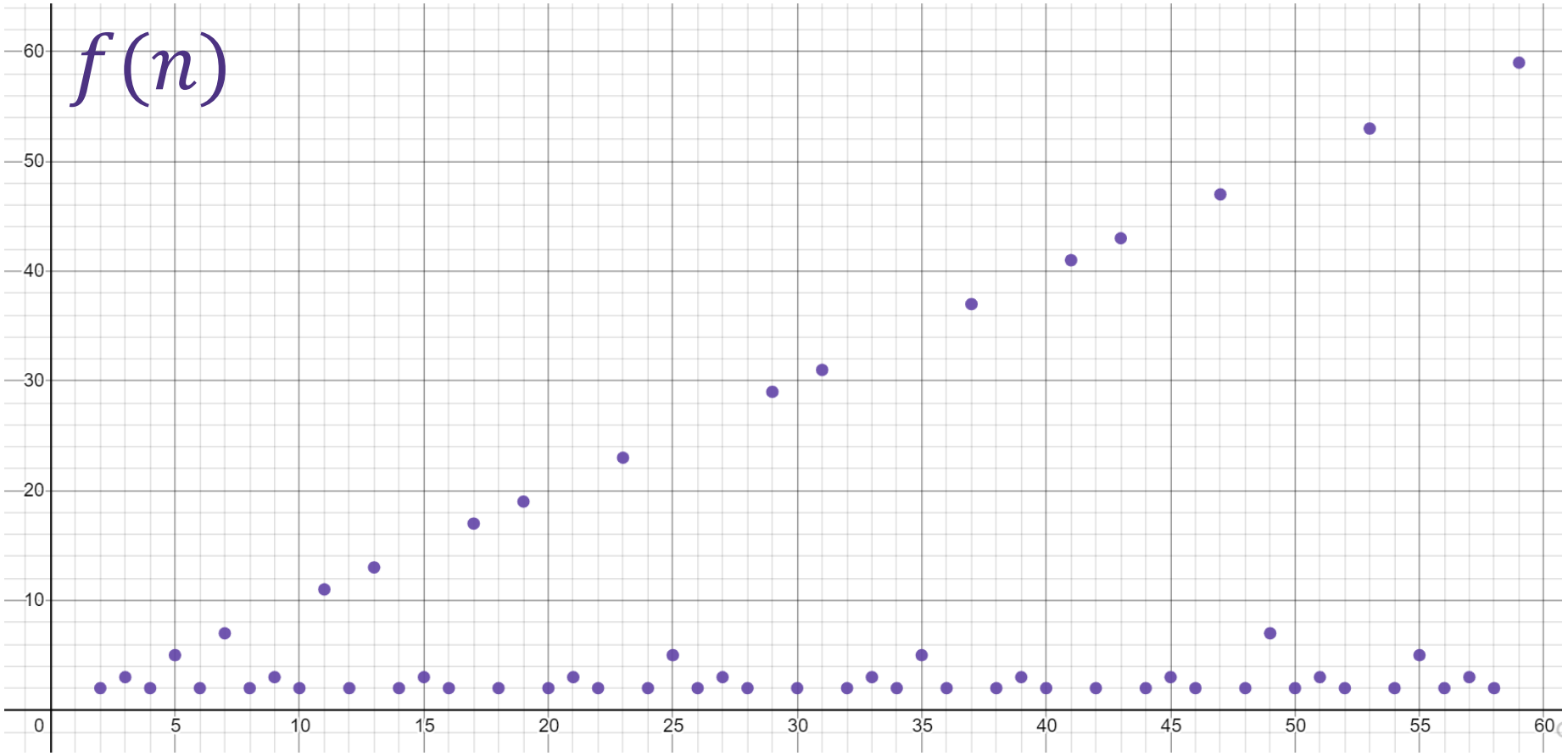
Remember, $f(n)$ = the number of basic operations performed on the input n.

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

Number of iterations?

- One less than the smallest divisor of n

Prime Checking Runtime

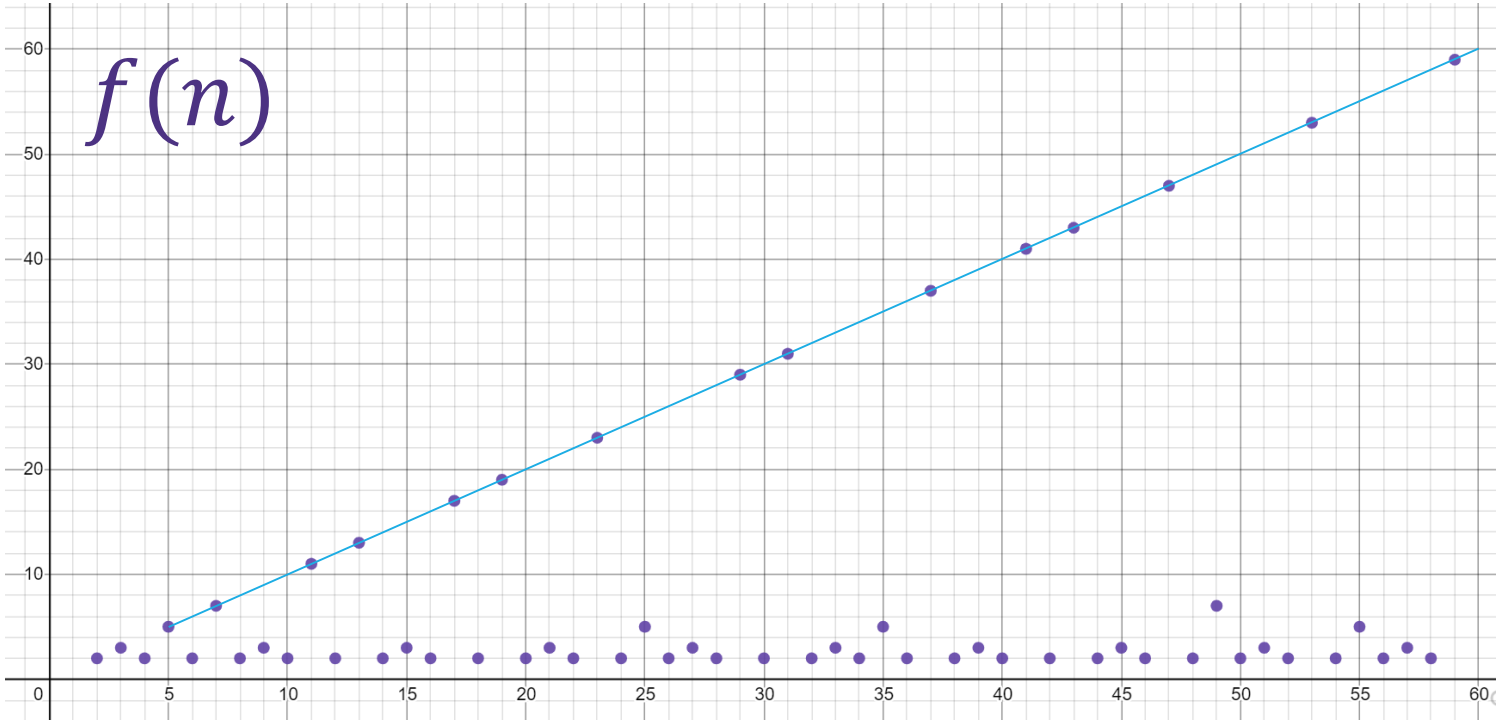


Is the running time of the code $O(1)$ or $O(n)$?

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 $n \geq 5$

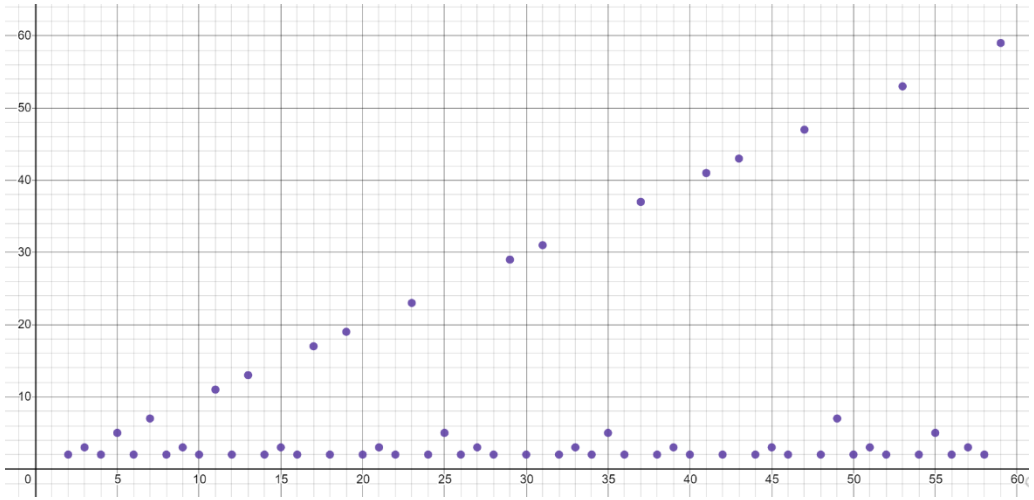
It's $O(n)$ but not $O(1)$

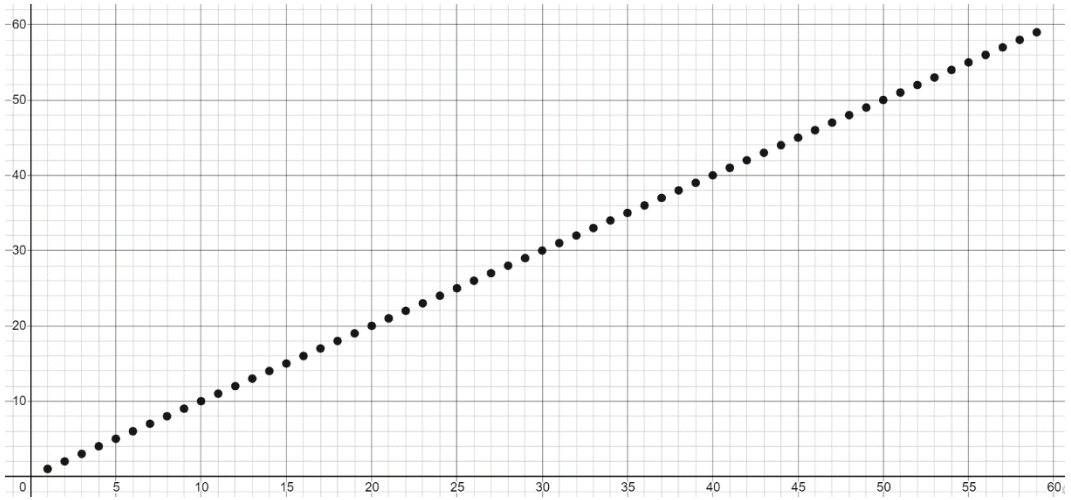
Is the running time $O(1)$?
Can you find constants c and n_0?

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.



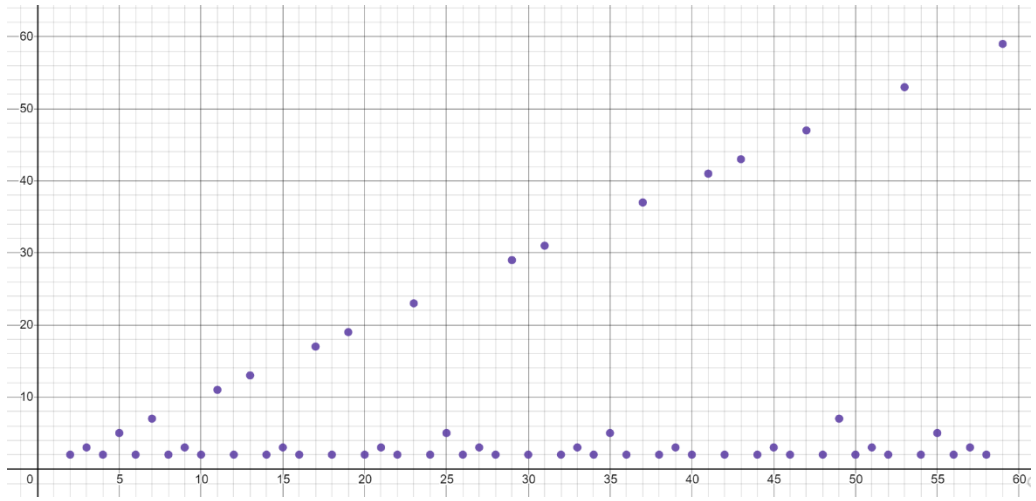


Your experience running these two pieces of code is going to be very different.
It's disappointing that the $O()$ are the same – that's not very precise.
Could we have some way of pointing out the list code always takes AT LEAST n operations?

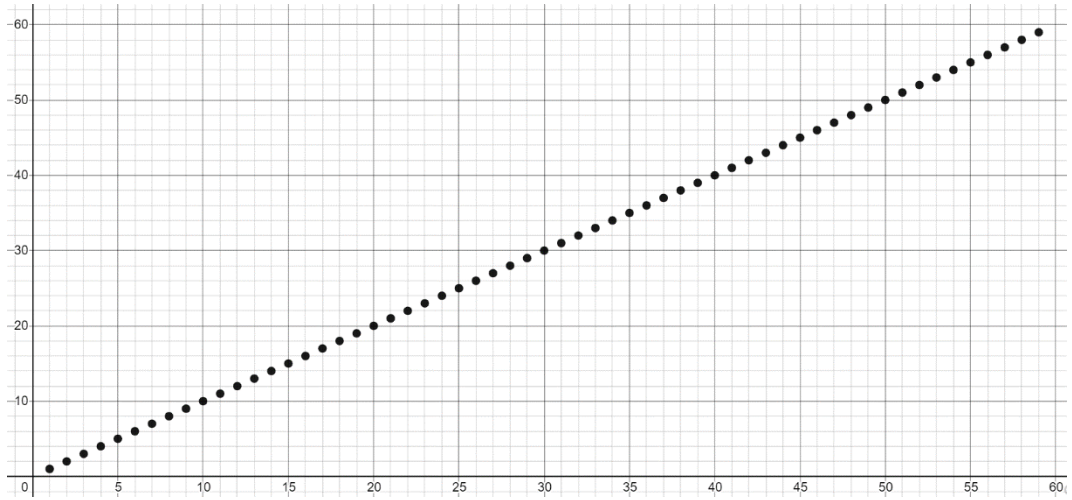
Big-Ω [Omega]

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

$4n^2 \in \Omega(1)$	$4n^2 \in O(1)$
true	false
$4n^2 \in \Omega(n)$	$4n^2 \in O(n)$
true	false
$4n^2 \in \Omega(n^2)$	$4n^2 \in O(n^2)$
true	true
$4n^2 \in \Omega(n^3)$	$4n^2 \in O(n^3)$
false	true
$4n^2 \in \Omega(n^4)$	$4n^2 \in O(n^4)$
false	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

pollEV.com/cse373su19

What are the answers to the first two questions?

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.

Cases

We defined $f(n)$ to be (our model for) the number of operations the code does on an input of size n.

$f(n)$ doesn't always have a nice formula, but so far if I tell you n, you can tell me $f(n)$.

Knowing n isn't always enough to know how long our code will take to run.

Linear Search

/* given an array and int toFind, return index where toFind is located, or -1 if not in array.*/

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

Cases

The number of operations doesn't depend just on n.

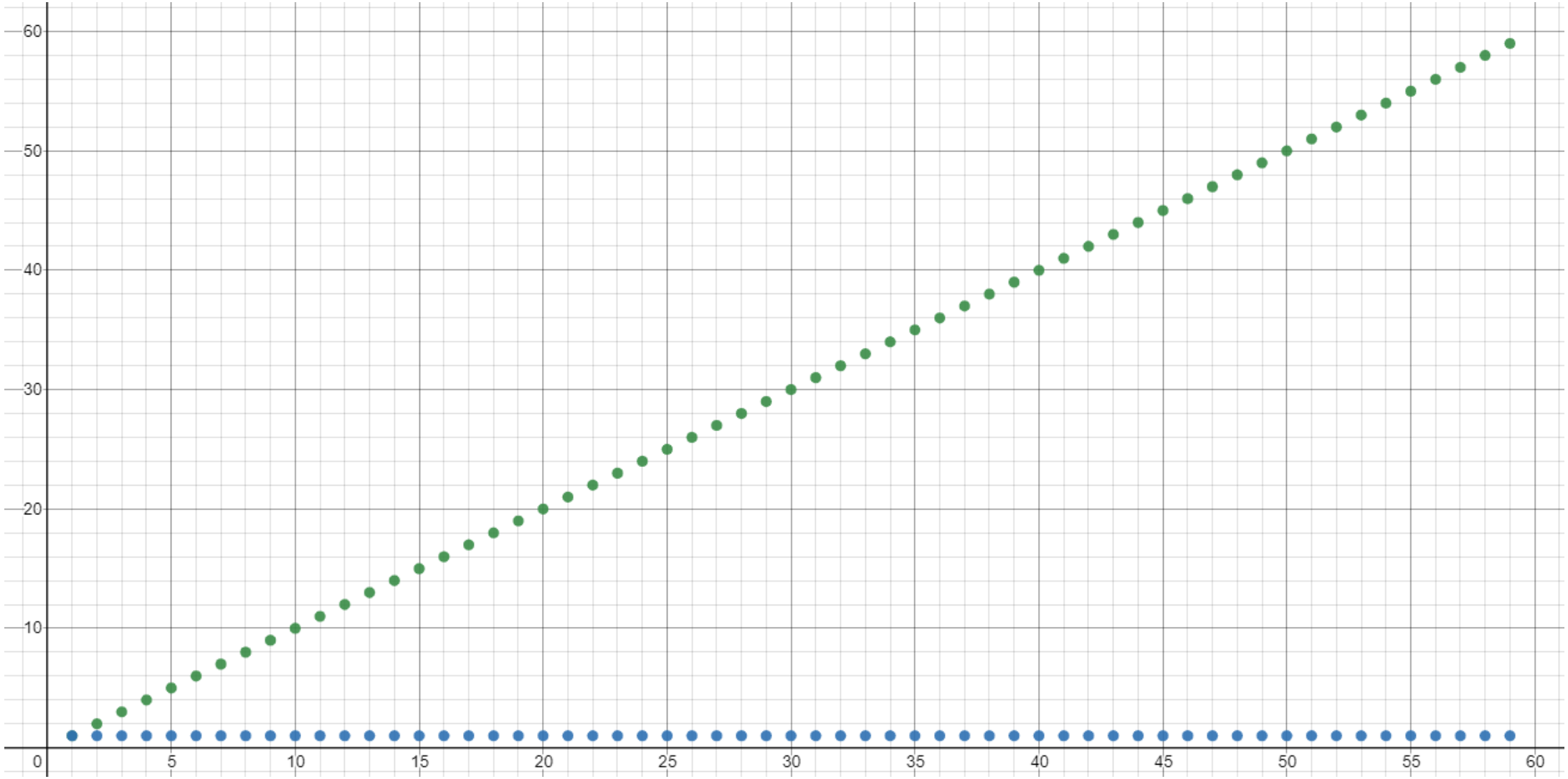
Even once you fix n (the size of the array) there are still a number of cases to consider.

If `toFind` is in `arr[0]`, we'll only need one iteration, $f(n) = 4$.

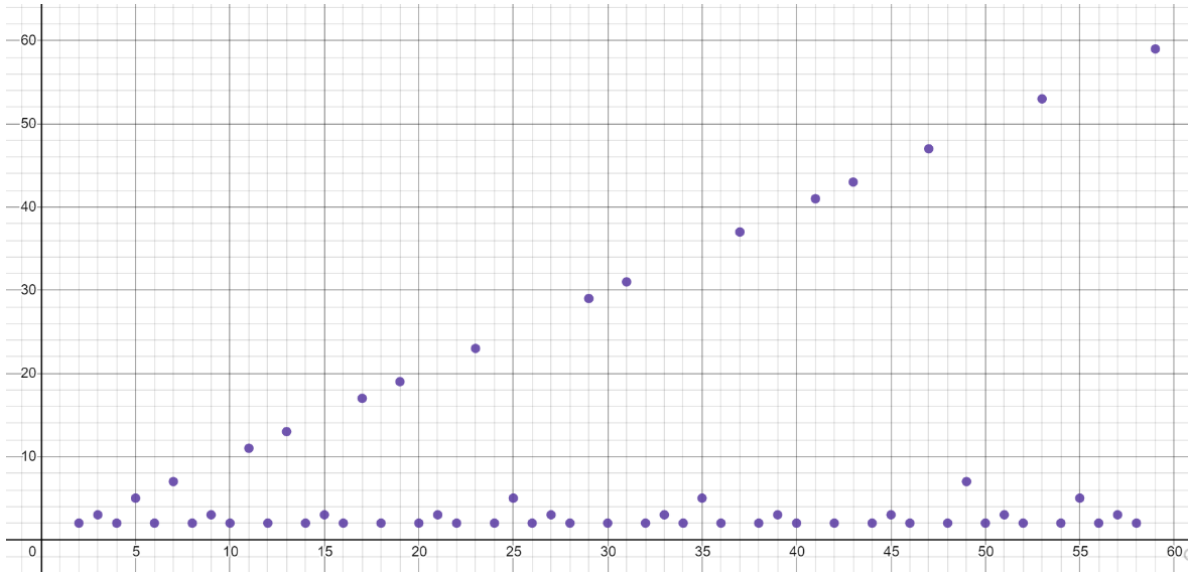
If `toFind` is not in `arr`, we'll need n iterations. $f(n) = 3n + 1$.

And there are a bunch of cases in-between.

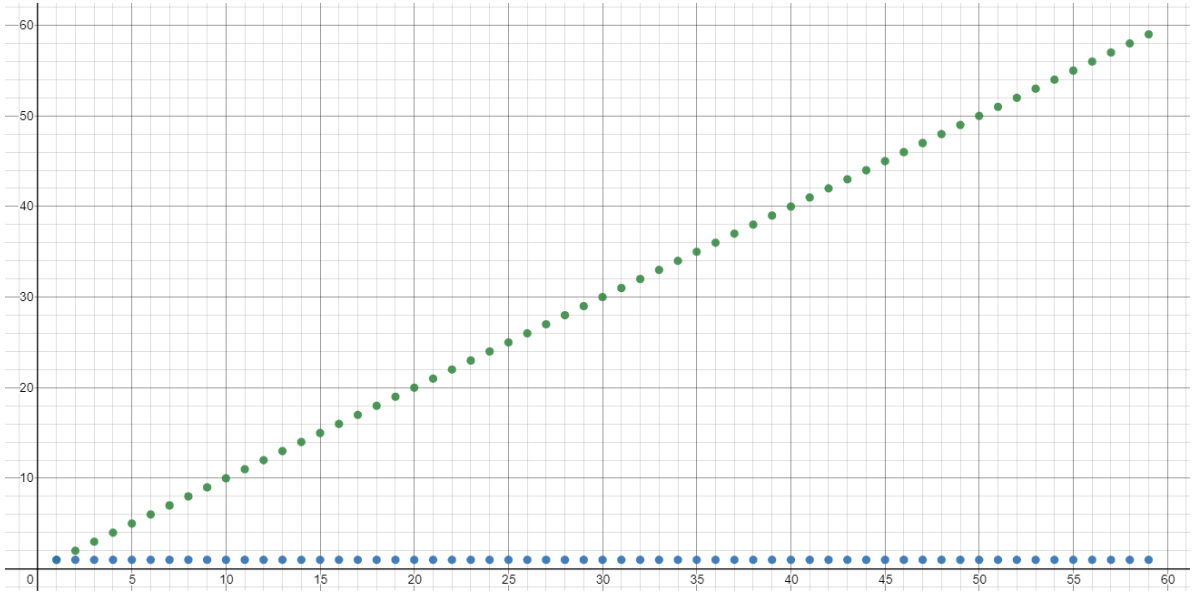
Linear Search Models



Prime Checker



Linear Search



For a given n, prime checker had only one model.
For a given n, Lienar Search has multiple possible models.

Cases

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.

So, which is the right model?

It depends on what you care about.

Cases

Usually we care about the longest our code could run on an input of size n.

This is **worst-case** analysis.

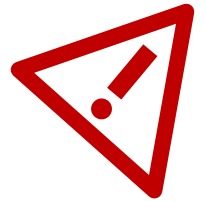
But sometimes we care about the fastest our code could finish on an input of size n.

This is **best-case** analysis.

For `linearSearch`, the model for the worst case is $f(n) = 3n + 1$

The model for the best case is $f(n) = 4$.

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 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.

	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.

	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.

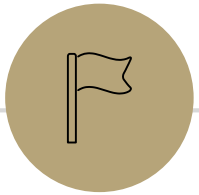
All of these can be combined with any of O, Ω, and Θ!

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!

```
/* given an array of integers, count the number of positive
integers that appear only once */
//There are more efficient versions of this.

int countUniquePositives(int[] arr){
      int count = 0;
      for(int i=0; i<arr.length; i++){
            if(arr[i] <= 0)
                  break;
            else{
                  boolean repeat = false;
                  for(int j=0; j < arr.length; j++){
                        if(i!=j && arr[i] == arr[j])
                              repeat = true;
                  }
                  if(!repeat)
                        count++;
            }
      }
      return count;
}
```



Dictionaries and best/worst case analysis

Dictionaries (aka Maps)

Every Programmer's Best Friend

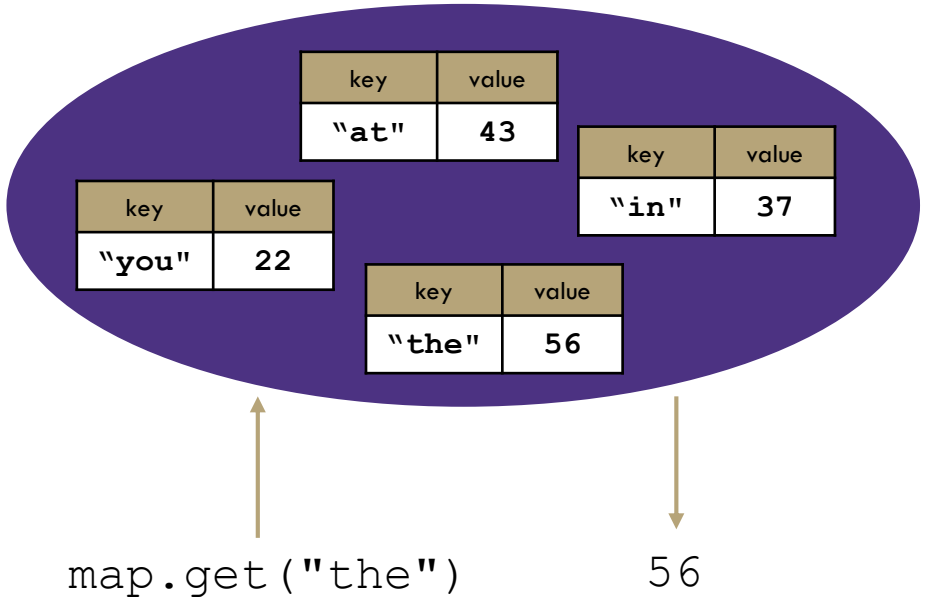
You'll probably use one in almost every programming project.

- Because it's hard to make a big project without needing one sooner or later.

Review: Maps

map: Holds a set of distinct *keys* and a collection of *values*, where each key is associated with one value.

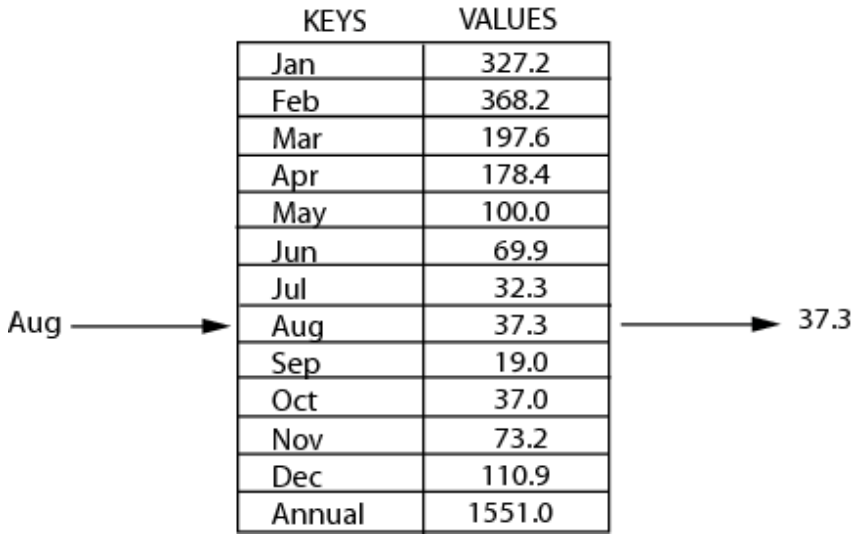
- a.k.a. "dictionary"



Dictionary ADT
state Set of items & keys Count of items
behavior put(key, item) add item to collection indexed with key get(key) return item associated with key containsKey(key) return if key already in use remove(key) remove item and associated key size() return count of items

supported operations:

- **put**(*key*, *value*): Adds a given item into collection with associated key,
  - **if the map previously had a mapping for the given key, old value is replaced**
- **get**(*key*): Retrieves the value mapped to the key
- **containsKey**(key): returns true if key is already associated with value in map, false otherwise
- **remove**(*key*): Removes the given key and its mapped value



2 Minutes

Implementing a Dictionary with an Array

Dictionary ADT

state

Set of items & keys
Count of items

behavior

put(key, item) add item to collection indexed with key
get(key) return item associated with key
containsKey(key) return if key already in use
remove(key) remove item and associated key
size() return count of items

ArrayDictionary<K, V>

state

```
Pair<K, V>[] data
```

behavior

put find key, overwrite value if there. Otherwise create new pair, add to next available spot, grow array if necessary
get scan all pairs looking for given key, return associated item if found
containsKey scan all pairs, return if key is found
remove scan all pairs, replace pair to be removed with last pair in collection
size return count of items in dictionary

```
put('a', 1)
put('b', 2)
put('c', 3)
put('d', 4)
remove('b')
put('a', 97)
```

0	1	2	3
('a', 97)	('b', 2)	('c', 3)	('d', 4)

Big O Analysis – Worst case

`put()`	O(N) linear
`get()`	O(N) linear
`containsKey()`	O(N) linear
`remove()`	O(N) linear
`size()`	O(1) constant

Big O Analysis – Best case

`put()`	O(1) constant
`get()`	O(1) constant
`containsKey()`	O(1) constant
`remove()`	O(1) constant
`size()`	O(1) constant

2 Minutes

Implementing a Dictionary with Nodes

Dictionary ADT

state

Set of items & keys
Count of items

behavior

put(key, item) add item to collection indexed with key
get(key) return item associated with key
containsKey(key) return if key already in use
remove(key) remove item and associated key
size() return count of items

LinkedDictionary<K, V>

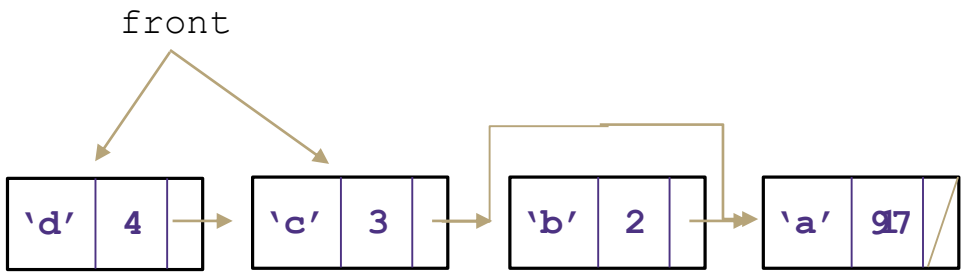
state

front
size

behavior

put if key is unused, create new with pair, add to front of list, else replace with new value
get scan all pairs looking for given key, return associated item if found
containsKey scan all pairs, return if key is found
remove scan all pairs, skip pair to be removed
size return count of items in dictionary

```
put('a', 1)
put('b', 2)
put('c', 3)
put('d', 4)
remove('b')
put('a', 97)
```



Big O Analysis – Worst Case

put()	O(N) linear
get()	O(N) linear
containsKey()	O(N) linear
remove()	O(N) linear
size()	O(1) constant

Big O Analysis – Best Case

put()	O(1) constant
get()	O(1) constant
containsKey()	O(1) constant
remove()	O(1) constant
size()	O(1) constant

Dictionaries

Running times summary:

Worst case is slow for EVERY interesting operation.

For lists, we had usually one slow operation.

Dictionaries are really useful.

We'll spend a week-and-a-half designing faster versions.

- But first, more big-O.