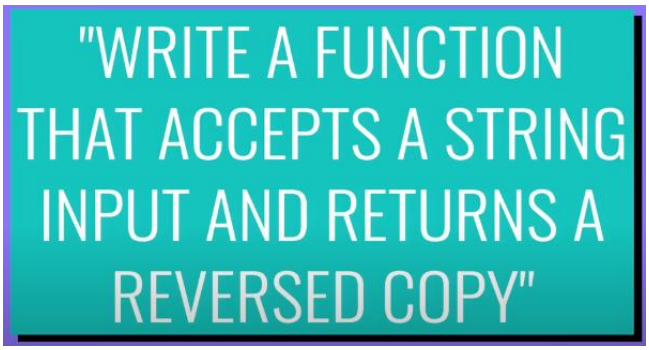
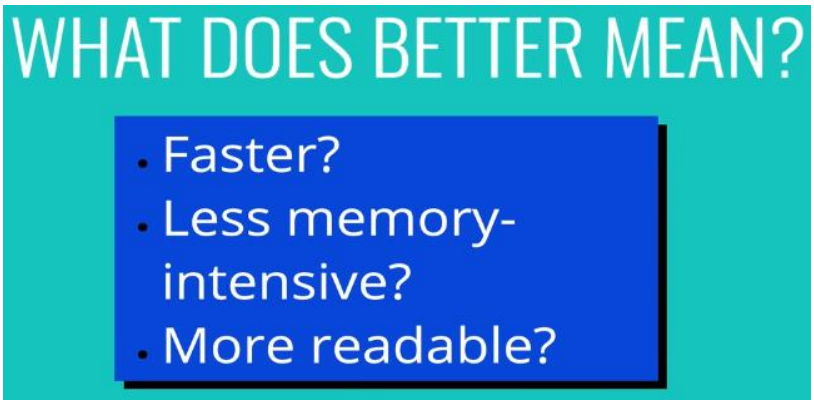
Imagine we have multiple implementations of the same function.

How can we determine which one is the "best?"

Let's say the desired function must do the following:



We can easily find 10 different ways to implement the desired function... but how do we know which one is the best?... Also, what does “best” mean?



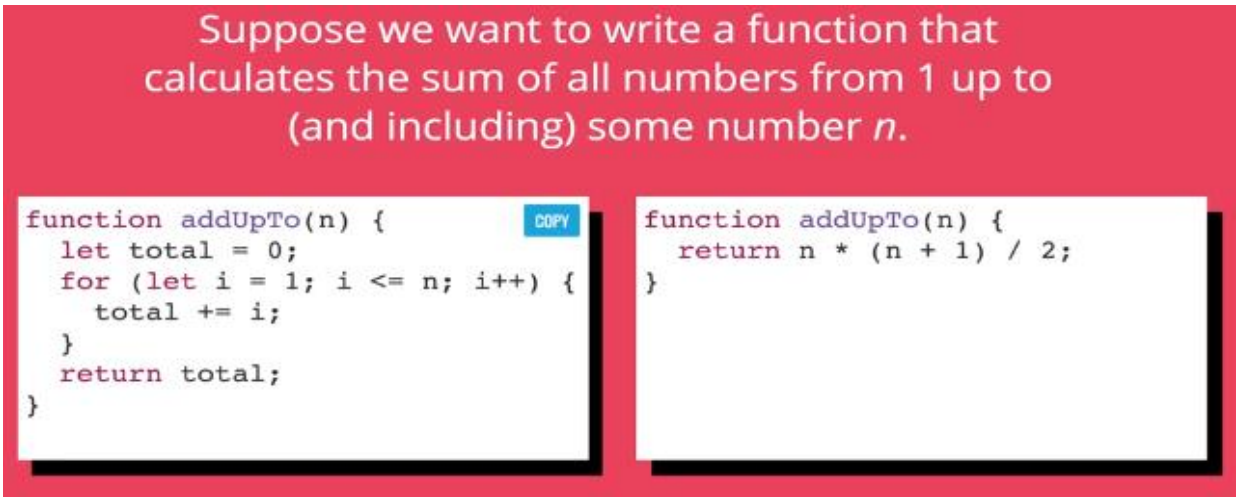
Ideally, it would be nice to have a chart that rates each implementation:

… it would describe, in details, the mechanics of each code.

… as well as the trade-offs between them.

… also, in case your code slows down or crashes, it’s important to identify pain points and inefficiencies.

Let’s look at 2 simple codes:



The one on the left:

-Uses a loop for ***n*** times

-Starts at zero, then added by increments of 1 until we reach ***n***.

The one on the right:

-A lot shorter, but works faster.

-The ***n*** is multiplied by itself plus 1 and is divided by half.

A good way to start judging speed is with a timer.

Let’s gauge the first code.

1) The t1 represents the current time.

2) We call the function addUpTo.

3) The t2 represents the current time after it finishes.

4) We print in the console the difference between t2 and t1.

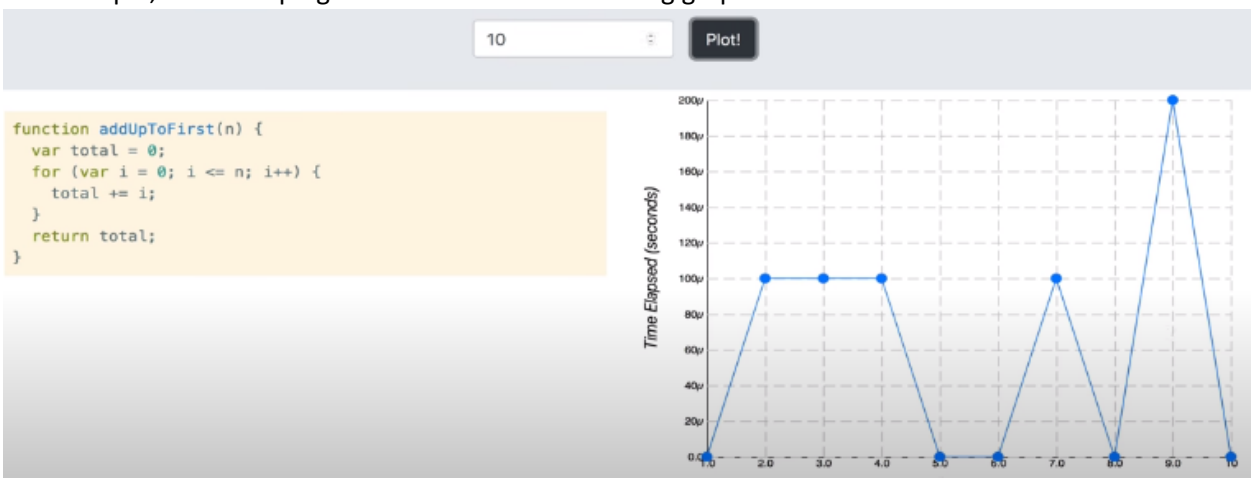
```
function addUpTo(n) {
  let total = 0;
  for (let i = 1; i <= n; i++) {
    total += i;
  }
  return total;
}

let t1 = performance.now();
addUpTo(1000000000);
let t2 = performance.now();
console.log(`Time Elapsed: ${(t2 - t1) / 1000} seconds.`)
```

… however, there is one problem with time:

THE PROBLEM WITH TIME

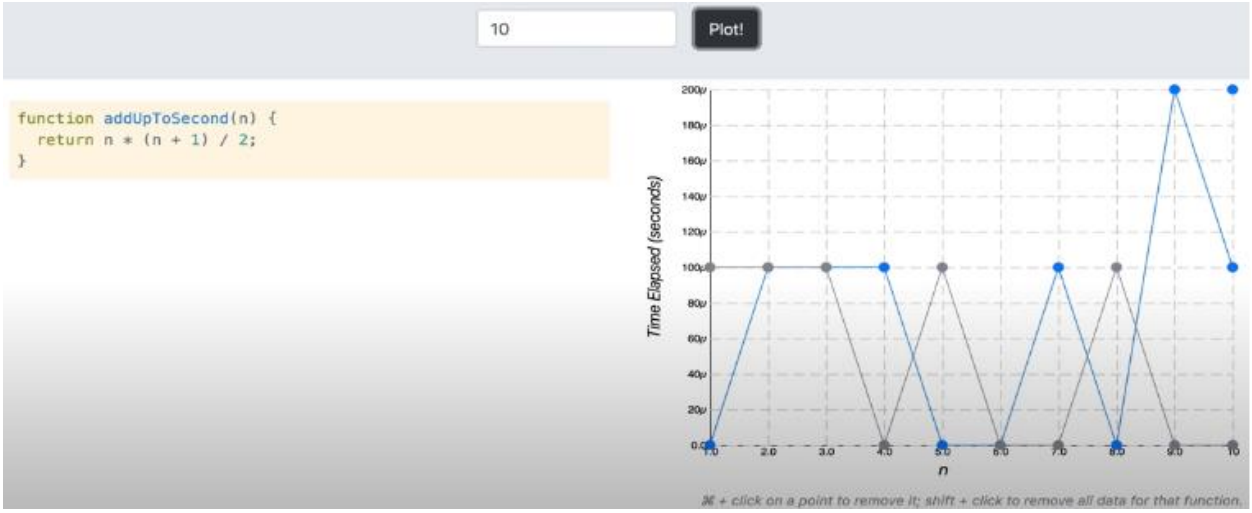
- Different machines will record different times
- The *same* machine will record different times!
- For fast algorithms, speed measurements may not be precise enough

For example, when we plug in 1-10 for n in the following graph...



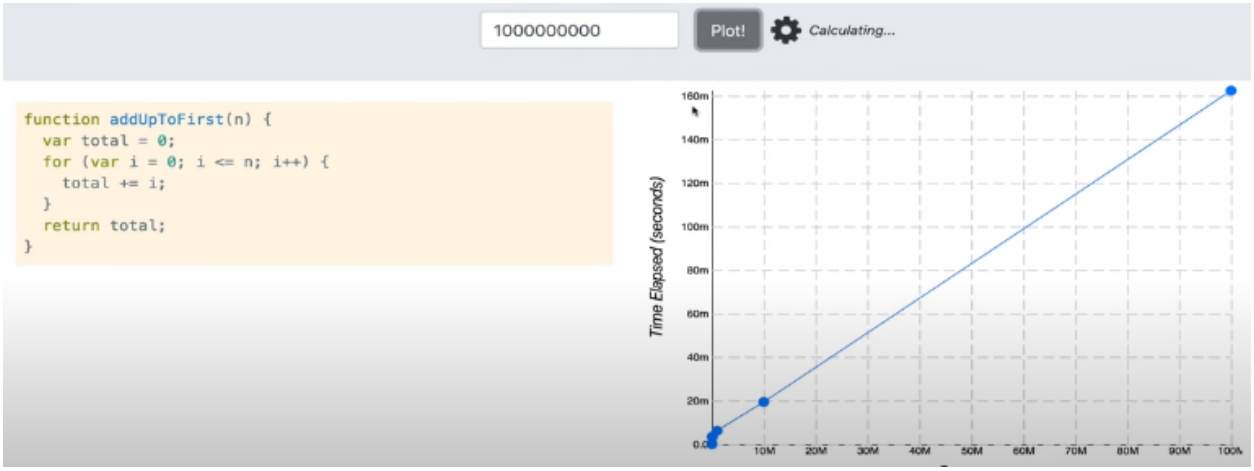
...the 1st code is executing so quickly, the javascript timer is not precise enough to capture it accurately.

Now if we try the 2nd code with values 1-10...

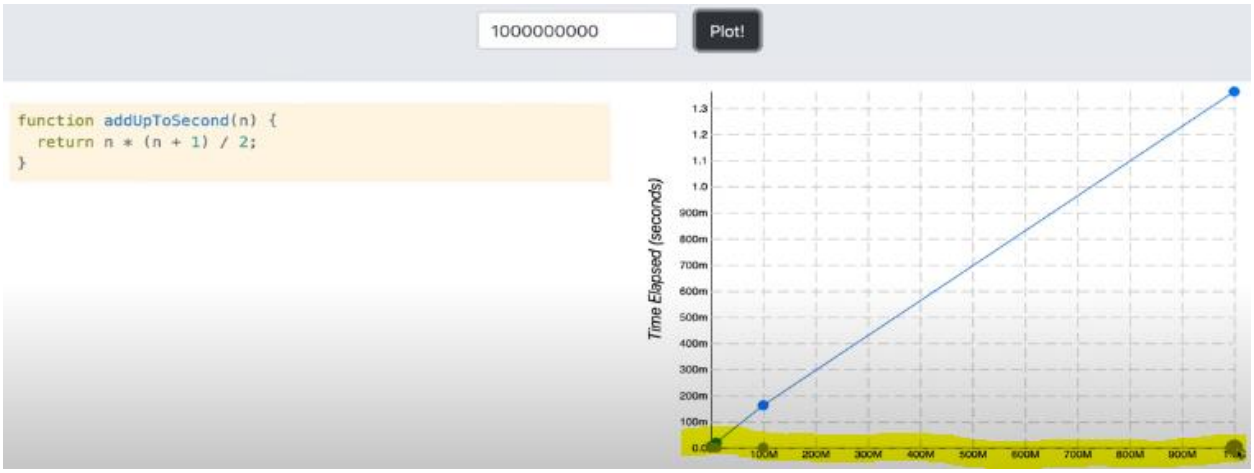


...we see it’s just as unpredictable as the first code.

If we plug in values 100 through 1,000,000,000 for both codes:



… we see that the 2nd code is way at the bottom along the x-axis, thus it's way faster than the 1st code:

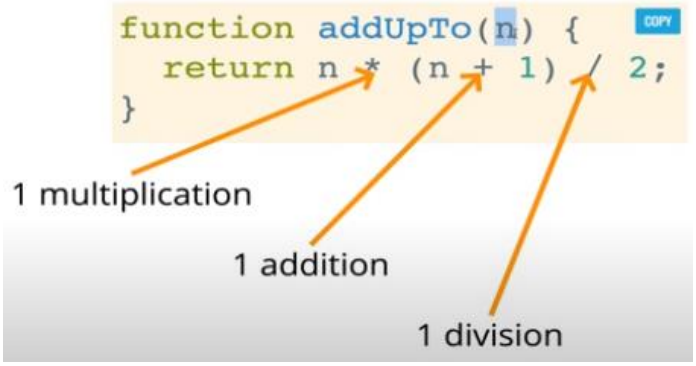


… to sum it all up, we get a better understanding as ***n*** increases and we approach infinity.

This is how we test our code: to see how it performs as we approach infinity.

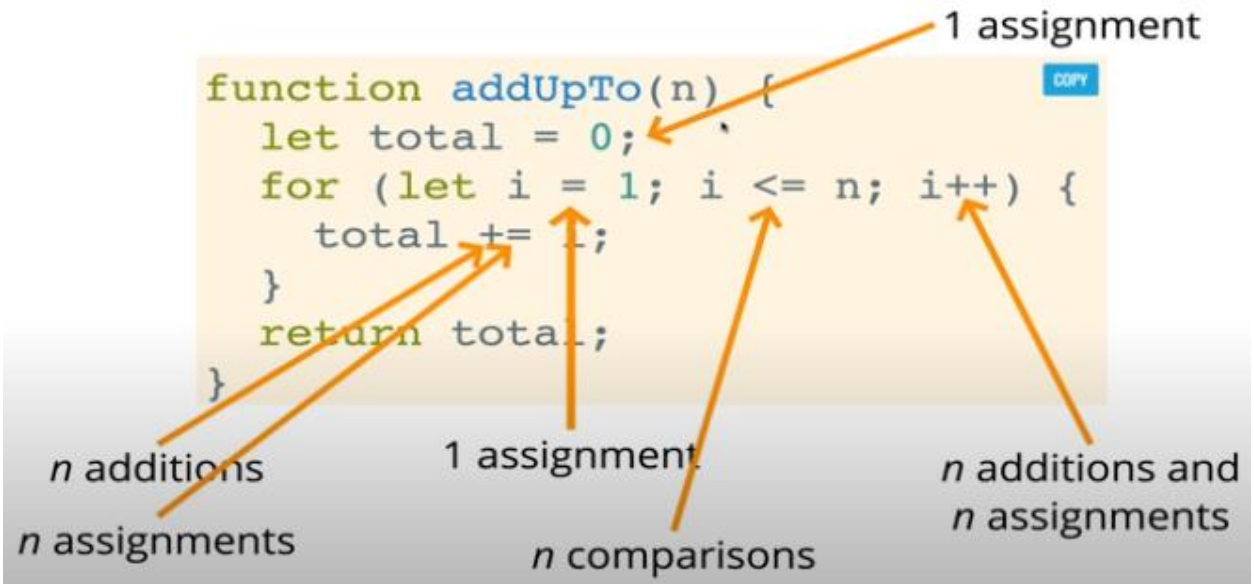
What Actually Matters... is Counting **Operations**!

Recall from our 2 codes, the faster one was very simple.



It only had 3 operations: multiplication, addition, division. There is no regard for the size of ***n***, even as it grows and approaches infinity.

The slower code, on the other hand, has a for loop, and as ***n*** grows, the number of operations will also increase proportionately.



… so the point is: the number of operations grows roughly proportionate with ***n*** as it increases.

And now, without further ado...

INTRODUCING....BIG O

what is it?

simplified analysis of an algorithm's efficiency

It allows us to talk formally about how the runtime of an algorithm grows as the inputs grow

We say that an algorithm is **O(f(n))** if the number of simple operations is eventually less than a constant times **f(n)**, as **n** increases

- f(n) could be linear (f(n) = n)
- f(n) could be quadratic ($f(n) = n^2$)
- f(n) could be constant (f(n) = 1)

EXAMPLES:

Linear: As n increases, the number of operations always stays the same:

```
function addUpTo(n) {
  return n * (n + 1) / 2;
}
```

Always 3 operations

O(1)

Quadratic: Nested loops are inefficient and should be avoided.

```
function printAllPairs(n) {
  for (var i = 0; i < n; i++) {
    for (var j = 0; j < n; j++) {
      console.log(i, j);
    }
  }
}
```

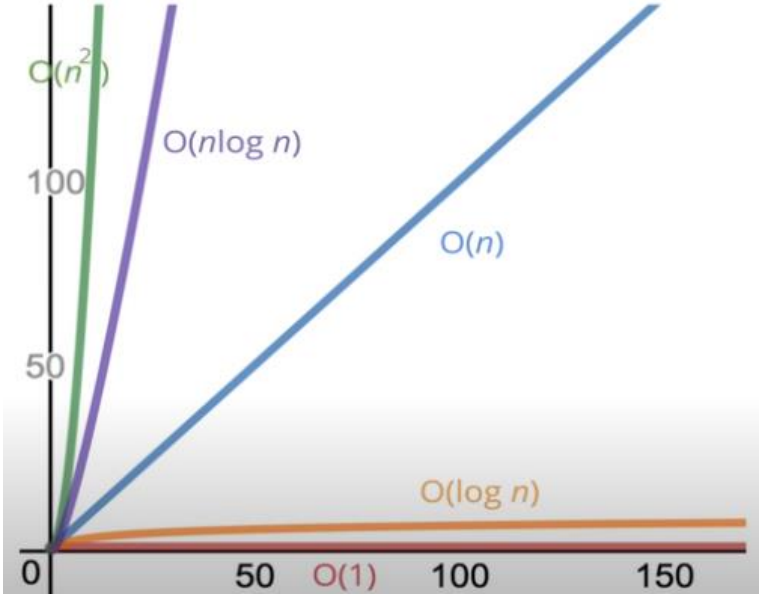
O(n) O(n)

O(n) operation inside of an O(n) operation.

$O(n * n)$ ⇨ **$O(n^2)$**

Here is the chart of the various Big O functions:

$O(1)$	constant
$O(\log n)$	logarithmic
$O(n)$	linear
$O(n \log n) = O(\log n!)$	linearithmic, loglinear, or quasilinear
$O(n^2)$	quadratic



… so for O(1), no matter how big n gets, the number of operations stays constant; this is optimal.

… so the more complex the algorithm, the more time-intensive it is.

… and remember:

CONSTANTS DON'T MATTER

O(2n) doesn't impact anything; it's still **O(n)**

O(500) doesn't impact anything; it's still **O(1)**

O($13n^2$) doesn't impact anything; it's still **O(n^2)**

… and similarly:

SMALLER TERMS DON'T MATTER

O(n + 10) doesn't impact anything; it's still **O(n)**

O(1000n + 50) doesn't impact anything; it's still **O(n)**

O(n^2 + 5n + 8) doesn't impact anything; it's still **O(n^2)**

BIG O SHORTHANDS

1. Arithmetic operations are constant
2. Variable assignment is constant
3. Accessing elements in an array (by index) or object (by key) is constant
4. In a loop, the complexity is the length of the loop times the complexity of whatever happens inside of the loop