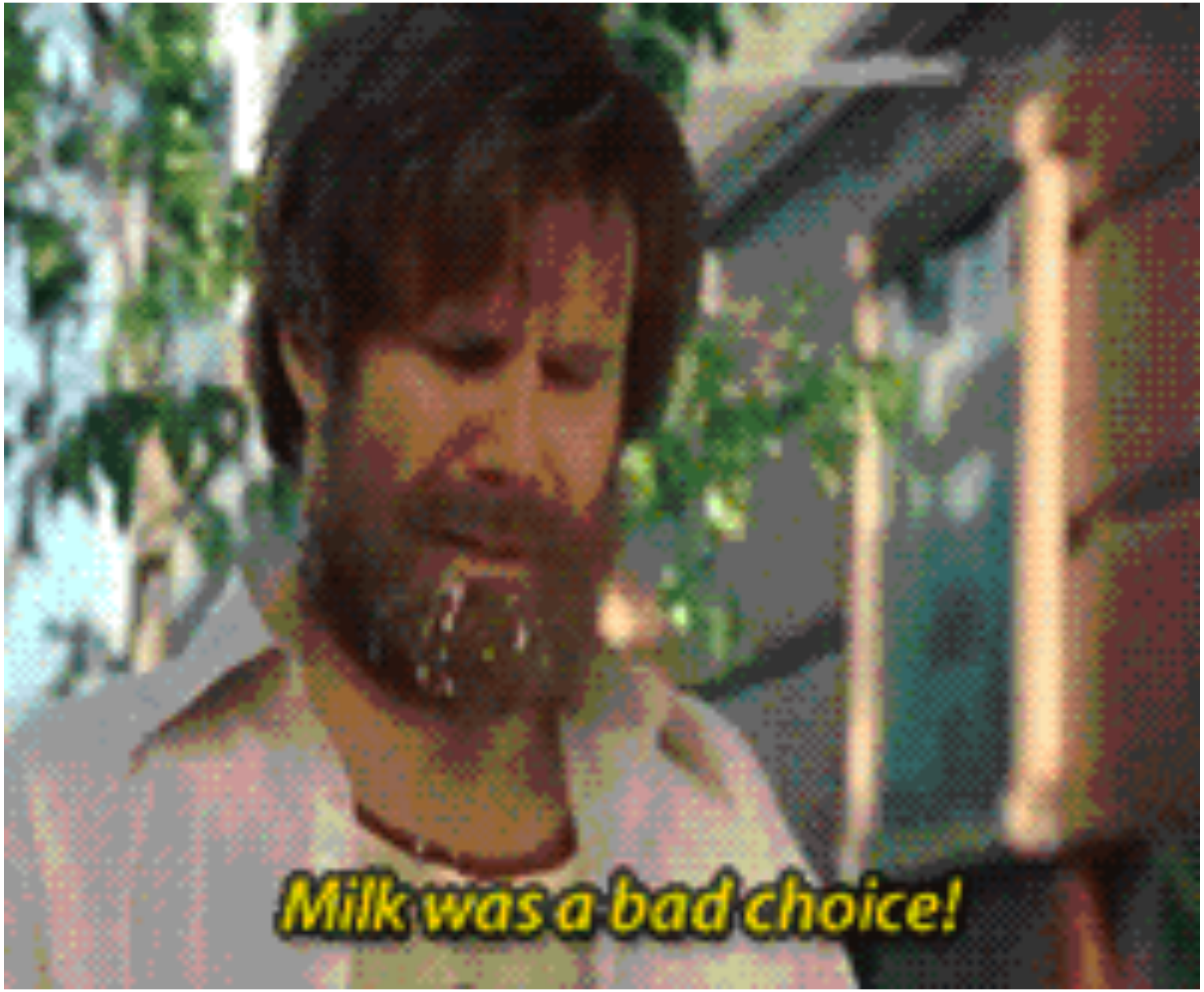
BIG O

Give it to me straight, doc; How bad is it?: Or measuring the efficiency of the code


Milk was a bad choice!

LEARNING OBJECTIVES

- By the end of this we will be able to:
  - Define what Big O is and why it's important
  - Calculate Big O for some simple algorithms
  - Be able to compare different time complexities
  - Calculate Big O for recursive algorithms
  - Calculate Big O for multi-level algorithms
  - Understand Big O in the context of time and space

MEASURING PERFORMANCE: SUM UP THE NUMBERS FROM 1 TO N INCLUSIVE

REPLIT

DISCUSSION: WHY CAN'T WE JUST BENCHMARK?

SO WHAT IS BIG O?

THE ANALYSIS OF HOW MUCH TIME (OR SPACE) AN ALGORITHM TAKES UP, RELATIVE TO ITS INPUT, AS THAT INPUT GETS BIGGER AND BIGGER

SO WHAT IS BIG O?

THE ANALYSIS OF HOW MUCH TIME (OR SPACE) AN ALGORITHM TAKES UP, **RELATIVE TO ITS INPUT, AS THAT INPUT GETS BIGGER AND BIGGER**

SO WHAT IS BIG O?

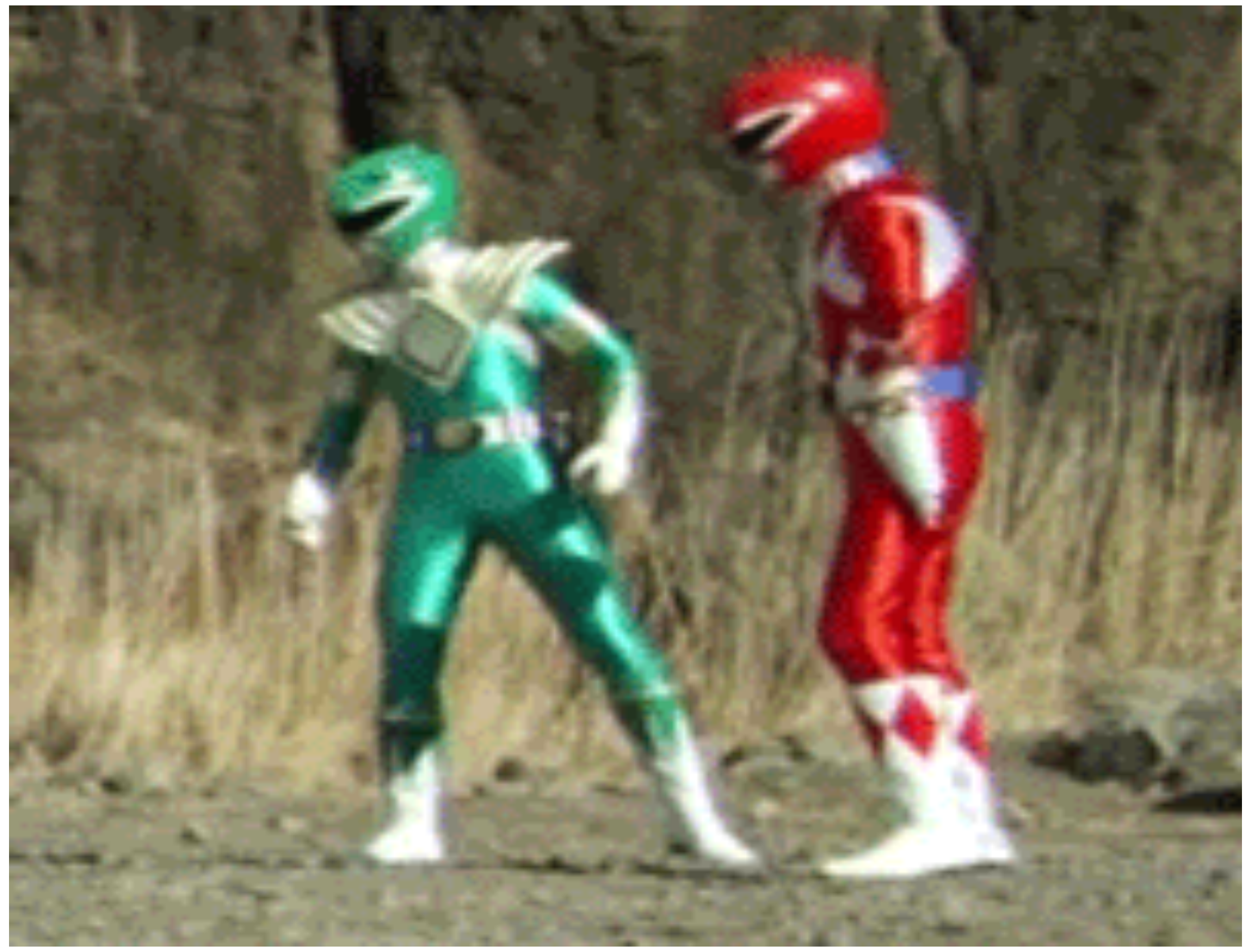
- How the runtime (or space) requirements grow…
  - A whole bunch of external factors are important when measuring the speed of an algorithm; so instead of using an absolute measure of speed, we use Big O Notation to express how quick it grows
- … Relative to the input
  - Since we are not looking for an absolute number, we need to measure speed relative to something; so we measure speed relative to the change in size of input

SO WHAT IS BIG O?

- The letter ‘O’ is used because the growth rate of a function is also called the ‘order of the function’
- Refers to worst case scenario

WHY SHOULD WE CARE?

- Being able to evaluate the runtime/space complexity of an algorithm is critical to writing performant code
- When asked to optimize an algorithm, one of the first things you might do is determine its Big O complexity
- Most importantly, it comes up in interviews



SOME BASIC STRATEGIES

- Measure the complexity at every step of the algorithm - We should ask ourselves, “Would this change if the input got larger?”
- Add the complexity for each line at the same level of indentation, multiple inner levels by their outer levels
  - Nested loops -> multiply
  - Scoped/Sibling loops -> add
- Simplify your terms, then drop everything by the largest term

SOME EXAMPLES

- Example 1
- Example 2
- Example 3
- Example 4

Beyond the Basics

- Recursion
- Space Complexity
- Multivariate algorithms

RECURSION WARMUP

Recursion

- It's helpful to think of recursion as a tree
- When there is only one "branch" of recursion, this is usually like a standard "for" loop, where the number of times we recursive corresponds to the input size
- When there are multiple recursive branches, the runtime will often be similar to O(branches ^ depth)
  - Each level of "depth" has "branch" number more calls than the level before - an exponential relationship!

WE ARE GIVEN A SET OF N STAIRS. THERE IS A PERSON STANDING AT THE BOTTOM. THIS PERSON CAN ONLY CLIMB EITHER 1 OR 2 STAIRS AT A TIME. COUNT THE NUMBER OF WAYS THIS PERSON CAN REACH THE TOP OF THE STAIRS

REPLIT

```
function fib (n) {
  if (n === 1 || n === 0) return n;
  else return fib(n – 1) + fib(n – 2);
}
```

```
function fib (n) {
  if (n === 1 || n === 0) return n;
  else return fib(n - 1) + fib(n - 2);
}
```

```
function fib (n) {
  if (n === 1 || n === 0) return n;
  else return fib(n - 1) + fib(n - 2);
}
```

fib(4)

```javascript
function fib (n) {
  if (n === 1 || n === 0) return n;
  else return fib(n - 1) + fib(n - 2);
}
```

```
              fib(4)
             /
      fib(3)
```

```
function fib (n) {
  if (n === 1 || n === 0) return n;
  else return fib(n - 1) + fib(n - 2);
}
```

```
                fib(4)
               /
          fib(3)
         /
     fib(2)
```

```
function fib (n) {
  if (n === 1 || n === 0) return n;
  else return fib(n - 1) + fib(n - 2);
}
```

```
                    fib(4)
                   /
              fib(3)
             /
         fib(2)
        /
    fib(1)
```

```
function fib (n) {
  if (n === 1 || n === 0) return n;
  else return fib(n - 1) + fib(n - 2);
}
```

```
                    fib(4)
                   /
             fib(3)
            /
       fib(2)
      /      \
  fib(1)      fib(0)
```

```
function fib (n) {
  if (n === 1 || n === 0) return n;
  else return fib(n - 1) + fib(n - 2);
}
```

```
                    fib(4)
                   /
            fib(3)
           /      \
      fib(2)       fib(1)
     /      \
fib(1)       fib(0)
```

```
function fib (n) {
  if (n === 1 || n === 0) return n;
  else return fib(n - 1) + fib(n - 2);
}
```

```
                 fib(4)
                /      \
          fib(3)          fib(2)
         /      \
     fib(2)    fib(1)
    /      \
fib(1)    fib(0)
```

```
function fib (n) {
  if (n === 1 || n === 0) return n;
  else return fib(n - 1) + fib(n - 2);
}
```

```
                 fib(4)
                /      \
          fib(3)        fib(2)
         /     \        /
    fib(2)    fib(1)  fib(1)
    /    \
fib(1)    fib(0)
```

```
function fib (n) {
  if (n === 1 || n === 0) return n;
  else return fib(n - 1) + fib(n - 2);
}
```

```
                        fib(4)
                       /      \
                 fib(3)          fib(2)
                /      \        /      \
           fib(2)    fib(1)  fib(1)    fib(0)
          /      \
     fib(1)      fib(0)
```

```
function fib (n) {
  if (n === 1 || n === 0) return n;
  else return fib(n - 1) + fib(n - 2);
}
```

our input is equal to 4: n = 4

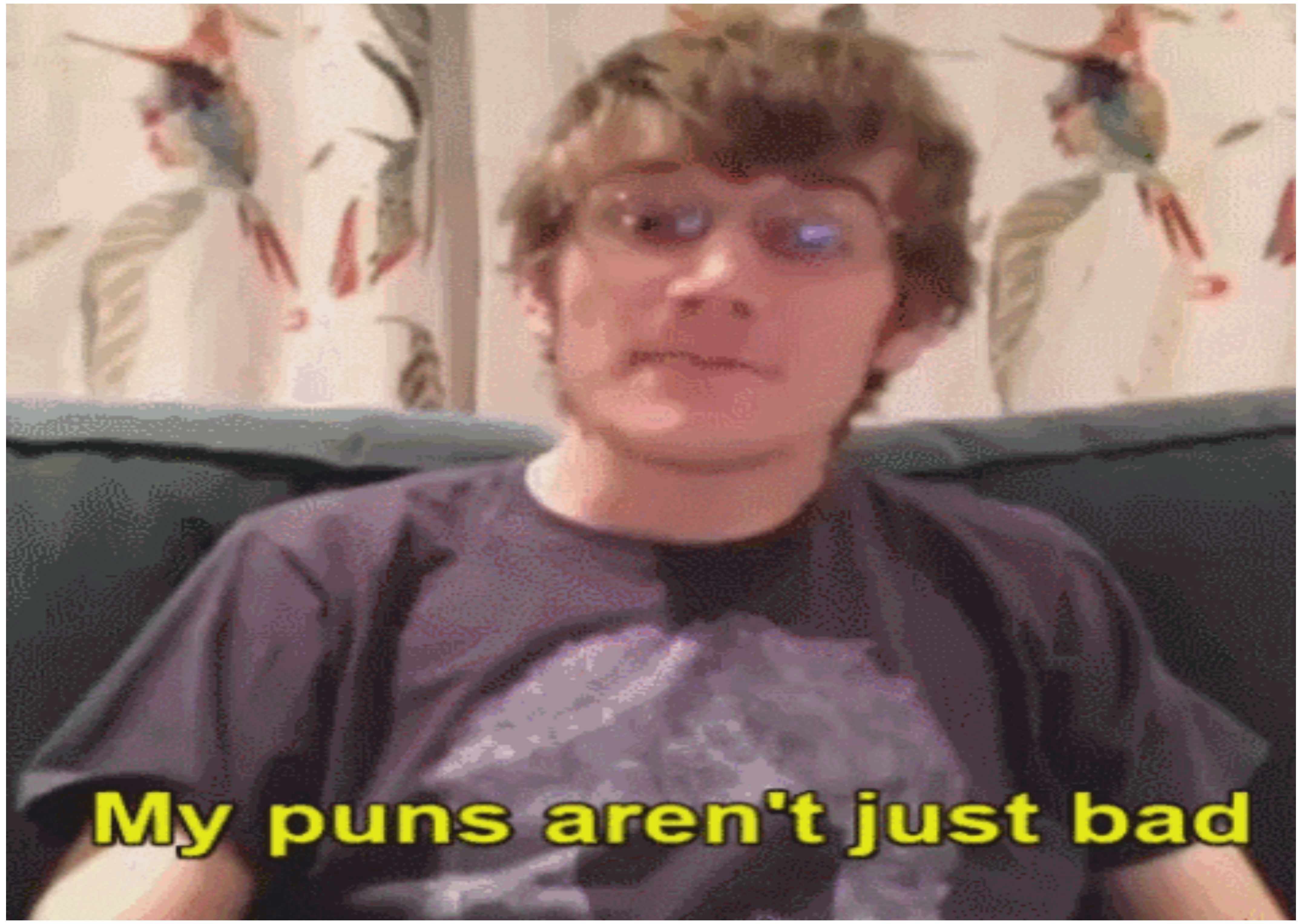
we go four levels deep, so depth = n

we branch twice with each recursive call

therefore, runtime is O(2^n)!

```
                          fib(4)
                         /      \
              fib(3)                  fib(2)
             /      \                /      \
      fib(2)          fib(1)    fib(1)       fib(0)
     /      \
fib(1)        fib(0)
```

WOW, THAT'S TEARABLE



BUT SERIOUSLY, LET'S MAKE IT BETTER

REPLIT

```
function fib (n, memo = {}) {
  if (n === 1 || n === 0) return n;
  else if (memo[n]) return memo[n];
  else memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
  return memo[n];
}
```

```javascript
function fib (n, memo = {}) {
  if (n === 1 || n === 0) return n;
  else if (memo[n]) return memo[n];
  else memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
  return memo[n];
}
```

```
function fib (n, memo = {}) {
  if (n === 1 || n === 0) return n;
  else if (memo[n]) return memo[n];
  else memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
  return memo[n];
}
```

```
memo = {

}
```

```
fib(4)
```

```javascript
function fib (n, memo = {}) {
  if (n === 1 || n === 0) return n;
  else if (memo[n]) return memo[n];
  else memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
  return memo[n];
}
```

```
memo = {

}
```

```
            fib(4)
           /
    fib(3)
```

```
function fib (n, memo = {}) {
  if (n === 1 || n === 0) return n;
  else if (memo[n]) return memo[n];
  else memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
  return memo[n];
}
```

```
memo = {

}
```

```
              fib(4)
             /
        fib(3)
       /
  fib(2)
```

```
function fib (n, memo = {}) {
  if (n === 1 || n === 0) return n;
  else if (memo[n]) return memo[n];
  else memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
  return memo[n];
}
```

```
memo = {

}
```

```
                fib(4)
               /
          fib(3)
         /
     fib(2)
    /
fib(1)
```

```
function fib (n, memo = {}) {
  if (n === 1 || n === 0) return n;
  else if (memo[n]) return memo[n];
  else memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
  return memo[n];
}
```

```
memo = {
  2: 1,

}
```

```
                    fib(4)
                   /
              fib(3)
             /
        fib(2)
       /     \
  fib(1)      fib(0)
```

```javascript
function fib (n, memo = {}) {
  if (n === 1 || n === 0) return n;
  else if (memo[n]) return memo[n];
  else memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
  return memo[n];
}
```

```
memo = {
  2: 1,
  3: 2
}
```

```
                    fib(4)
                   /
             fib(3)
            /      \
       fib(2)      fib(1)
      /      \
 fib(1)      fib(0)
```

```javascript
function fib (n, memo = {}) {
  if (n === 1 || n === 0) return n;
  else if (memo[n]) return memo[n];
  else memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
  return memo[n];
}
```

```
memo = {
  0: 0,
  1: 1
  2: 1,
  3: 2
}
```

```
                        fib(4)
                       /      \
               fib(3)              fib(2)
              /      \
       fib(2)        fib(1)
      /      \
fib(1)        fib(0)
```

Space Complexity

- Big O can also express space complexity
- Measures how much space (i.e. memory) we use relative to the input (ex. by storing values in arrays and hash tables, and simultaneous calls on the call stack
  - Remember: what matters is the growth curve. not the actual number of bytes we store!
- Space can be taken a freed up again - the same can't be said of time!
- Usually, we have enough space…but not enough time!

```javascript
// assume `callback` performs an O(1) operation
function map (arr, callback) {
  const newArr = []
  for (let i = 0; i < arr.length; i++) {
    newArr.push(callback(arr[i]))
  }
  return newArr
}
```

```
// assume `callback` performs an O(1) operation
function map (arr, callback) {
  const newArr = []
  for (let i = 0; i < arr.length; i++) {
    newArr.push(callback(arr[i]))
  }
  return newArr
}
```

Multivariate Algorithms

- What if you have an algorithm that uses another algorithm? For example, what if you loop over an array of strings and sort each string?

- Be careful to not to confuse the input and runtime of the "outer" algorithm with the input and runtime of the "inner" algorithm

```
function sortedStrings (arr) {
  for (let i = 0; i < arr.length; i++) {
      arr[i].sort();                              // Let's say that .sort is O(n log n)
  }
}
```

```
function sortedStrings (arr) {
  for (let i = 0; i < arr.length; i++) {   // O(n)
      arr[i].sort();                          // O(s log s)
  }
}
```

Algorithm Analysis: Big O Notation

- A *comparative* way to classify different algorithms
- Based on *shape* of *growth curve* (*time* vs *input size(s)*)
- For *big enough* inputs
  - Might not be true when n is small, but who cares when n is small?
- Establishing an *upper bound* on the time
  - Not worse than this. Might be better, but it ain't worse!
- Including just the *highest order* term
  - In $f(n) = n^3 + 5n + 3$, only n^3 matters as n gets large
- *Ignores constants* (mostly irrelevant; $0.1 \cdot n^2$ will overtake $10 \cdot n$)

A NICE MNEMONIC

- Different terms for different inputs
- Remove constants
- Axe the non-dominant terms
- Worst Case

n! 2n n2
n · log(n)
n
Time for Function to Complete
Common Big O Notations
log(n)
1
Size of Input (n)

FUN EXAMPLE: REVISITING SUM UP

- We can actually optimize it to O(1)
- Replit

LEARNING OBJECTIVES

- It's the end and we are able to:
  - Define what Big O is and why it's important
  - Calculate Big O for some simple algorithms
  - Be able to compare different time complexities
  - Calculate Big O for recursive algorithms
  - Calculate Big O for multi-level algorithms
  - Understand Big O in the context of time and space