# BIG O

*Time and space complexity*

# BUT FIRST…LET’S GO BUY SOME DROIDS

1 day = 5s

1 day = 1s

1 day = 5s

1 day = 1s

1 day = 5s

1 day = 1s

2 days = 10s

2 days = 4s

1 day = 5s

2 days = 10s

1 day = 1s

2 days = 4s

1 day = 5s

2 days = 10s

1 day = 1s

2 days = 4s

3 days = 15s

3 days = 9s

1 day = 5s

2 days = 10s

3 days = 15s

1 day = 1s

2 days = 4s

3 days = 9s

1 day = 5s

2 days = 10s

3 days = 15s

1 day = 1s

2 days = 4s

3 days = 9s

4 days = 20s

4 days = 16s

1 day = 5s

2 days = 10s

3 days = 15s

4 days = 20s

1 day = 1s

2 days = 4s

3 days = 9s

4 days = 16s

1 day = 5s

2 days = 10s

3 days = 15s

4 days = 20s

1 day = 1s

2 days = 4s

3 days = 9s

4 days = 16s

5 days = 25s

5 days = 25s

| | |
|---|---|
| 1 day = 5s | 1 day = 1s |
| 2 days = 10s | 2 days = 4s |
| 3 days = 15s | 3 days = 9s |
| 4 days = 20s | 4 days = 16s |
| 5 days = 25s | 5 days = 25s |

1 day = 5s

2 days = 10s

3 days = 15s

4 days = 20s

5 days = 25s

6 days = 30s

1 day = 1s

2 days = 4s

3 days = 9s

4 days = 16s

5 days = 25s

6 days = 36s

| | |
|---|---|
| 1 day = 5s | 1 day = 1s |
| 2 days = 10s | 2 days = 4s |
| 3 days = 15s | 3 days = 9s |
| 4 days = 20s | 4 days = 16s |
| 5 days = 25s | 5 days = 25s |
| 6 days = 30s | 6 days = 36s |

1 day =  5s

2 days = 10s

3 days = 15s

4 days = 20s

5 days = 25s

6 days = 30s

**10 days = 50s**

1 day =  1s

2 days = 4s

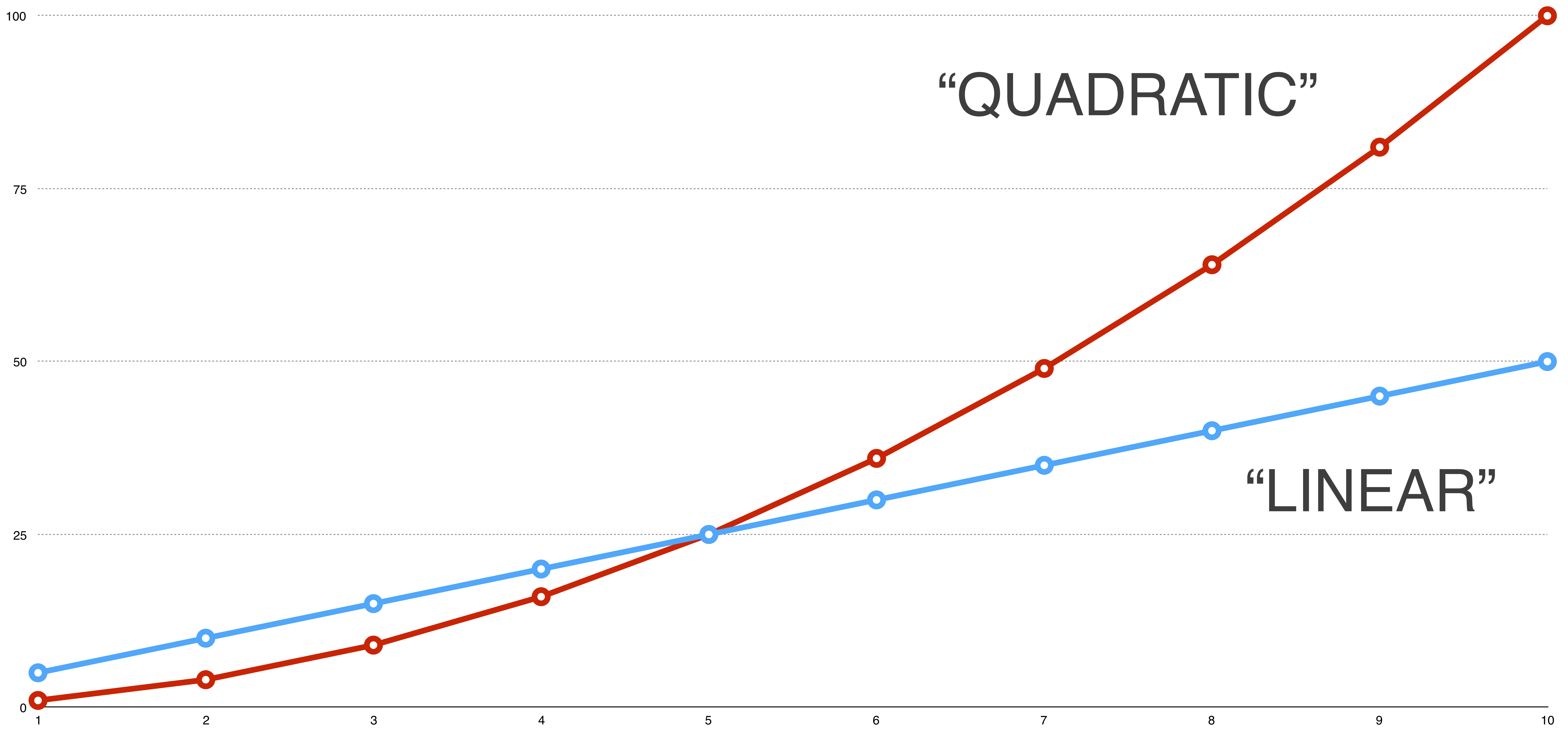
3 days = 9s

4 days = 16s

5 days = 25s

6 days = 36s

**10 days = 100s**



100
75
50
25
0
1
2
3
4
5
6
7
8
9
10
“QUADRATIC”
“LINEAR”

# BIG O

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

# BIG O

- Concerns *growth,* not *actual space or time*

- Focus on the “limiting behavior” (the behavior as input gets bigger and bigger)

# Example 1

```javascript
function example (array) {
  console.log(array.length)
  let someNumber = 4
  someNumber += array.length
  return someNumber
}
```

```javascript
function example (array) {
  console.log(array.length)  // 1
  let someNumber = 4
  someNumber += array.length
  return someNumber
}
```

```
function example (array) {
  console.log(array.length)  // 1
  let someNumber = 4         // 1
  someNumber += array.length
  return someNumber
}
```

```
function example (array) {
  console.log(array.length)  // 1
  let someNumber = 4          // 1
  someNumber += array.length // 1
  return someNumber
}
```

```
function example (array) {
  console.log(array.length)  // 1
  let someNumber = 4          // 1
  someNumber += array.length // 1
  return someNumber           // 1
}
```

```
function example (array) {
  console.log(array.length)  // 1
  let someNumber = 4         // 1
  someNumber += array.length // 1
  return someNumber          // 1
}
// O(1 + 1 + 1 + 1) = O(4) = O(1)
```

# Example 2

```javascript
// re-naming the array 'n'
function example (n) {
  const len = n.length
  let sum = 0

  for (let i = 0; i < len; i++) {
    sum += n[i]
  }

  return sum
}
```

```
// re-naming the array 'n'
function example (n) {
  const len = n.length                  // 1
  let sum = 0                           // 1

  for (let i = 0; i < len; i++) {
    sum += n[i]
  }

  return sum                           // 1
}
```

```
// re-naming the array 'n'
function example (n) {
  const len = n.length                    // 1
  let sum = 0                             // 1

  for (let i = 0; i < len; i++) { // n
    sum += n[i]                            // 1
  }

  return sum                             // 1
}
```

```javascript
// re-naming the array 'n'
function example (n) {
  const len = n.length                   // 1
  let sum = 0                            // 1

  for (let i = 0; i < len; i++) { // n
    sum += n[i]                          // 1
  }

  return sum                            // 1
}
// O(1 + 1 + (n * 1) + 1) = O(3 + n) = O(n)
```

# Example 3

```
function example (n) {
  const len = n.length

  for (let i = 0; i < len; i++) {
    console.log(n[i])
  }

  for (let j = 0; j < len; j++) {
    if (n[i] > 5) {
      console.log(n[i])
    }
  }

  return len
}
```

```
function example (n) {
  const len = n.length                  // 1

  for (let i = 0; i < len; i++) {
    console.log(n[i])
  }

  for (let j = 0; j < len; j++) {
    if (n[i] > 5) {
      console.log(n[i])
    }
  }

  return len                            // 1
}
```

```
function example (n) {
  const len = n.length                     // 1

  for (let i = 0; i < len; i++) { // n
    console.log(n[i])                      // 1
  }

  for (let j = 0; j < len; j++) { // n
    if (n[i] > 5) {                        // assume this always runs
      console.log(n[i])                    // 1
    }
  }

  return len                               // 1
}
```

```
function example (n) {
  const len = n.length                   // 1

  for (let i = 0; i < len; i++) { // n
    console.log(n[i])                   // 1
  }

  for (let j = 0; j < len; j++) { // n
    if (n[i] > 5) {                     // assume this always runs
      console.log(n[i])                 // 1
    }
  }

  return len                            // 1
}
// O(1 + (n * 1) + (n * 1) + 1) = O(2 + 2n) = O(2n) = O(n)
```

# Example 4

```
function example (n) {
  for (let i = 0; i < n.length; i++) {
    for (let j = 0; j < n.length; j++) {
      console.log(n[i] + n[j])
    }
  }
}
```

```
function example (n) {
  for (let i = 0; i < n.length; i++) {   // n
    for (let j = 0; j < n.length; j++) {
      console.log(n[i] + n[j])
    }
  }
}
```

```
function example (n) {
  for (let i = 0; i < n.length; i++) {   // n
    for (let j = 0; j < n.length; j++) {   // n
      console.log(n[i] + n[j])
    }
  }
}
```

```
function example (n) {
  for (let i = 0; i < n.length; i++) {   // n
    for (let j = 0; j < n.length; j++) {   // n
      console.log(n[i] + n[j])                 // 1
    }
  }
}
```

```
function example (n) {
  for (let i = 0; i < n.length; i++) {   // n
    for (let j = 0; j < n.length; j++) {   // n
      console.log(n[i] + n[j])                 // 1
    }
  }
}
// O((n * (n * 1)) = O(n^2)
```

# Example 5

```
// now, n is a number
function example (n) {
  let counter = 0

  while (n > 1) {
    n = n / 2
    counter++
  }

  return counter
}
```

```
// now, n is a number
function example (n) {
  let counter = 0  // 1

  while (n > 1) {
    n = n / 2
    counter++
  }

  return counter  // 1
}
```

```javascript
// now, n is a number
function example (n) {
  let counter = 0  // 1

  while (n > 1) {  // ?
    n = n / 2
    counter++
  }

  return counter  // 1
}
```

```
// now, n is a number
function example (n) {
  let counter = 0  // 1

  while (n > 1) {  // log(n)
    n = n / 2
    counter++
  }

  return counter  // 1
}
```

```javascript
// now, n is a number
function example (n) {
  let counter = 0  // 1

  while (n > 1) {  // log(n)
    n = n / 2
    counter++
  }

  return counter  // 1
}
// O(2 + log(n)) = O(log(n))
```

# Quick review of logarithms

**Logarithms are just the opposite of exponents**

$$\log_2(n)$$

**Read as:** ***what power do we need to raise 2 to in order to get n?***

$$\log_2(2) = 1$$

$$\log_2(4) = 2$$

$$\log_2(8) = 3$$

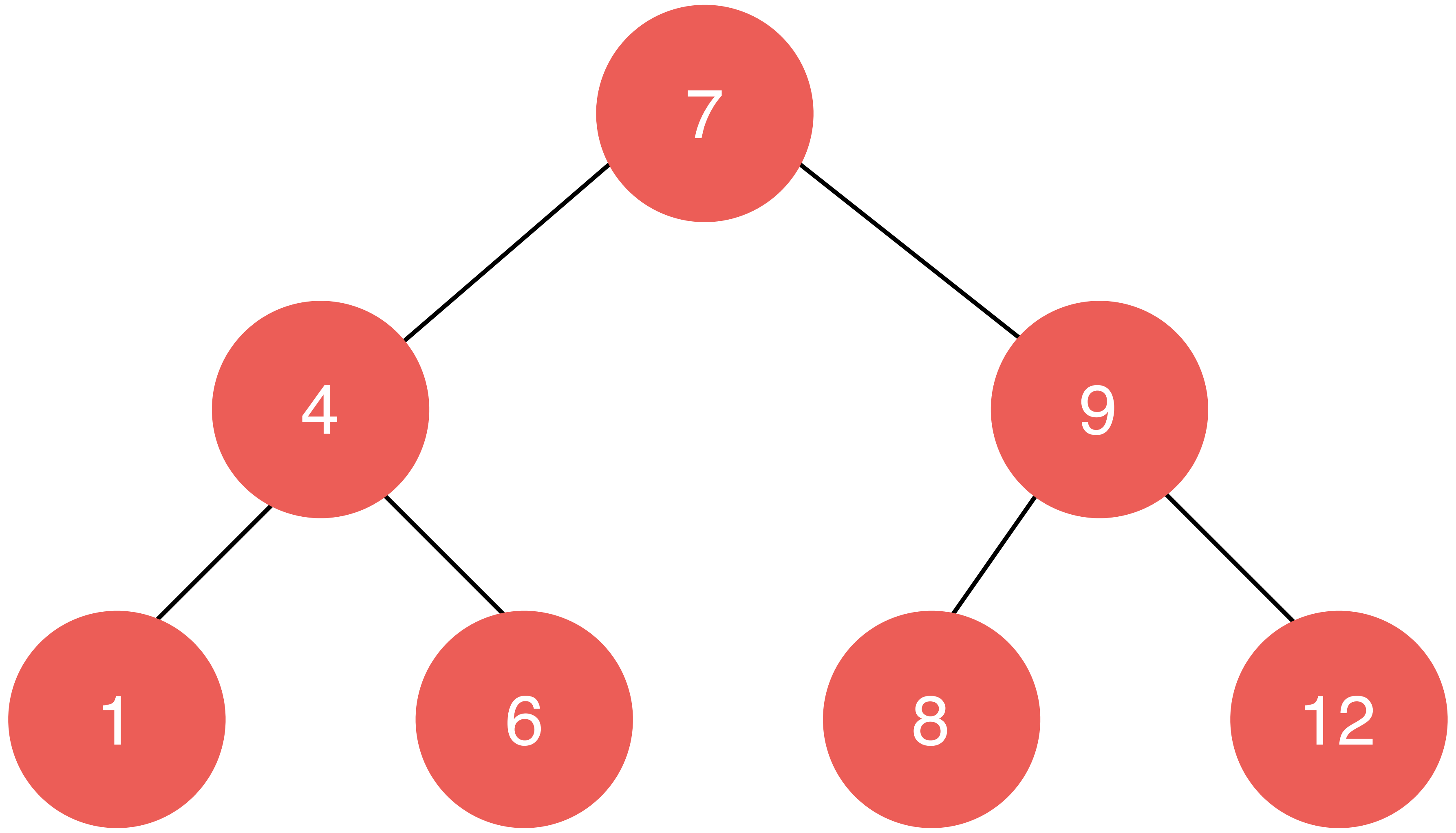
$$\log_2(5) = 2.32192809489$$

# Beyond the Basics

- Recursion
- Space Complexity
- Multivariate algorithms

# 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!




$n = 7$

$branches = 2$

$depth = 3$

$\log_2 7 \sim= 3$

$depth \sim= \log_2 n$

$branches \wedge depth = 2^{\log_2 n}$

$branches \wedge depth = n$

```javascript
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)
```

```
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)
```

```javascript
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)
```

```javascript
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)
```

```
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)
```

```
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 = {

  1: 1


}
```

```
                        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 = {
  0: 0,
  1: 1
  2: 1,

}
```

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

```
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(1)
     /     \
fib(1)     fib(0)
```

```
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?

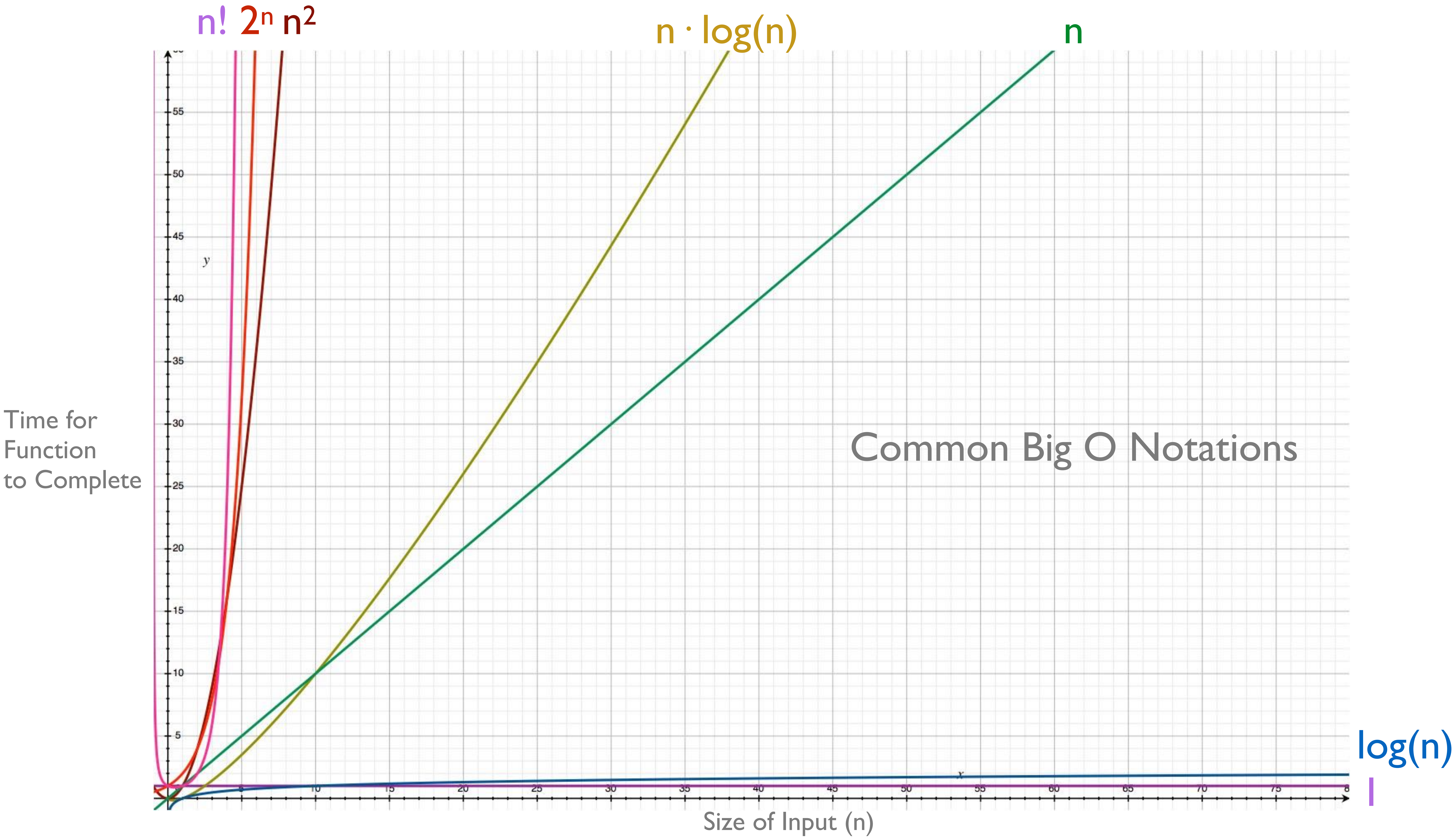
- Be careful to not to confuse the input and runtime of the "outer" algorithm with the input and runtime of the "inner" algorithm

```
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 $10 \cdot n$)


n! $2^n$ $n^2$
$n \cdot \log(n)$
n
Time for Function to Complete
Common Big O Notations
log(n)
1
Size of Input (n)