Big O Notation

1. Intro To Big O

A. Objectives

- Motivation for needing BigO Notation
- Describe Big O Notation
- Simplify Big O Notations
- Define "time" and "space" complexities
- Evaluate "time" and "space" complexities of algorithms
- Describe what an algorithm is

B. What's the motivation behind Big O?

> If we have multiple implementations of a function, then how do we determine which implementation is the "**best**"

C. What is Big O?

> Big O is a way of generalizing and comparing code and it's performance to other pieces of code

D. Why care about Big O?

- Gives us precise vocabulary to talk about how our code performs
- Useful for discussing trade-offs between different approaches
- Identify inefficiencies in our code
- Interviews

---

2. Timing Our Code

A. Example

Write a function that calculates the sum of all numbers from 1 up to (and including) some number n.

Slower Solution:

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

Faster Solution:

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

Which code is "better"? And what does "better" mean?

- faster ?
- less memory-intensive ?
- more readable ?

Comparing time for both codes:

Slower Solution:

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

> Time Elapsed: 0.9451825829744339 seconds.

Faster Solution:

```
const time1 = performance.now();
addUpTo(1000000000);
const time2 = performance.now();
console.log(`Time Elapsed: ${(time2 - time1) / 1000} seconds.`)
```

> Time Elapsed: 0.000010916948318481446 seconds.

B. The Problem with comparing **TIME**

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

> We want to be able to be able to talk about code in general terms, without having to measure time, and that's where Big O comes into play

---

3. Counting Operations

A. If not time, then what?

> Instead of counting seconds, count the **number of operations** the computer has to perform.

Faster Solution (Previous Example)

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

Counting operations for the faster solution:

1. Multiplication - `n * ...`
2. Addition - `n + ...`
3. Division - `...) / 2`

> There are 3 simple operations, regardless of the size of `n`

Slower Solution (Previous Example)

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

Counting operations for the slower solution:

- n additions
    1. `total = total + i`
    2. `i++` shorthand for `i = i + 1`
- n assignments
    1. `total = total + i`
    2. `i++` shorthand for `i = i + 1`

- 1 assignment
    1. `total = 0`
    2. `i = 1`
- n comparison
    1. `i <= n`

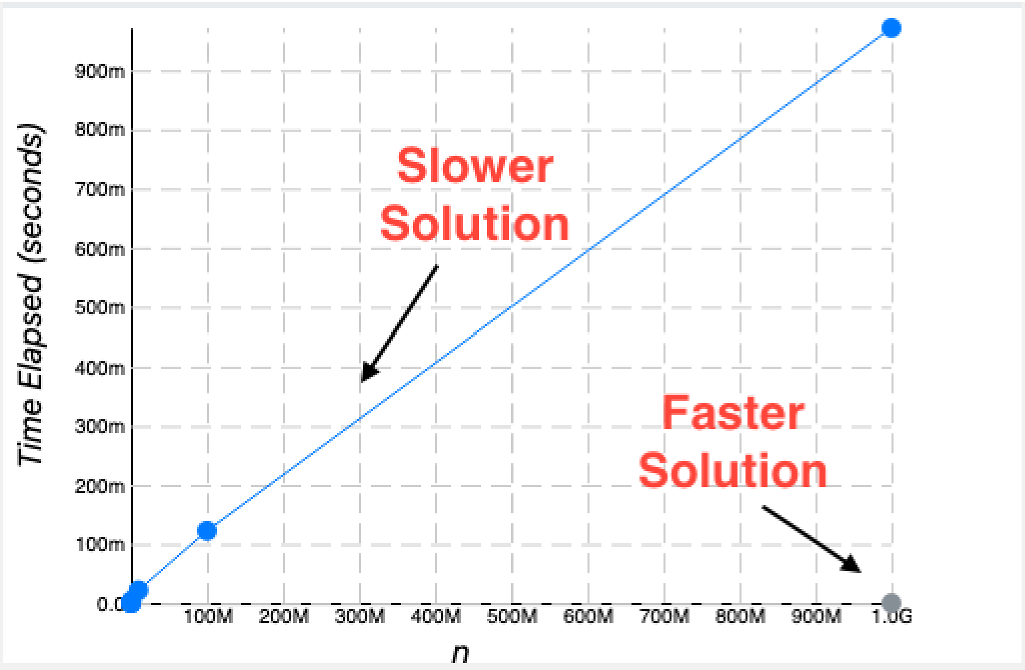
> Depending on what we count, the number of operations can be as low as 2n or as high as 5n+2. But regardless of the exact number, the number of operations grow roughly proportionally with n

---

4. Visualizing Time Complexities

A. Performance Tracker

> Link To: Performance Tracker



> In regards to the example above, as `n` increases, the execution time for the faster solution remains constant, whereas, for the slower solution, the time continues to increase linearly, as seen from the graph above.

---

5. Official Intro to Big O

A. Big O

- Big O Notation is a way to formalize fuzzy counting
- It allows us to talk formally about how the runtime of an algorithm grows as the input grows
- We don't care about the details, only the broad trends

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

> - f(n) could be linear (`f(n) = n`)
> - f(n) could be quadtratic (`f(n) = n`^{2})
> - f(n) could be constant (`f(n) = 1`)
> - f(n) could be entirely different

- Big O refers to the **worst case** scenario, i.e. the upperbound of runtime.

B. Example

1. Faster Solution:

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

- Always 3 operations: `O(1)`

> As `n` grows there's no change in the runtime

2. Slower Solution:

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

- Always 5n +2 operations: `O(n)`

> As `n` grows the number of operations is eventually bounded by a multiple of `n`, `O(n)`

3. Count Up and Down:

```
function countUpAndDown(n) {

  console.log("Going Up")

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

  console.log("At the top. \n Going down.")

  for (let j = n - 1; j >= 0; j--){
    console.log(j)
  }

  console.log("Back Down")
}
```

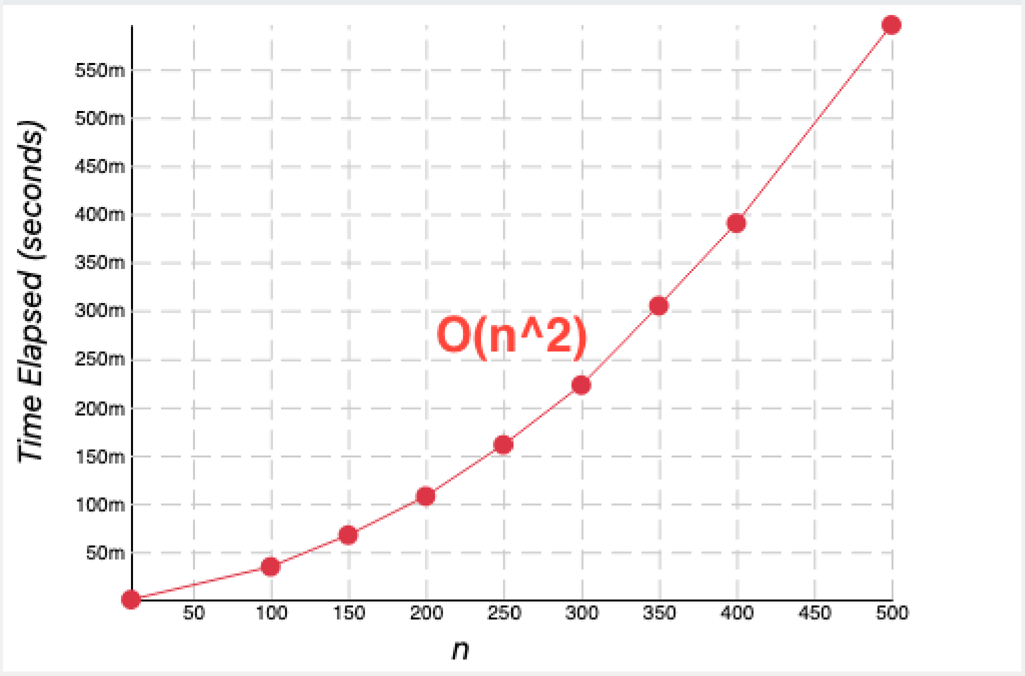
- First for loop: `O(n)`
- Second for loop: `O(n)`

> You might think that Big O is 2n, but the number of operations is (eventually) bounded by a multiple of n, so we simplify it to `O(n)`

4. Print All Pairs:

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

- First for loop: `O(n)`
- Nested loop: `O(n)`
  - if n = 2 -> [00, 01, 10, 11]: 4 pairs
  - if n = 3 -> [00, 01, 02,...]: 9 pairs

> 
>
> `O(n)` operation inside of an `O(n)` operation is `O(n*n)`, which is `O(n^2)`, so as n grows, the runtime growth is quadratic.

> An `O(n)` inside an `O(n)` is `O(n^2)`

6. Simplifying Big O Expressions

A. General Rules

Rule 1 - Constants Don't Matter

$O(\cancel{2}n)$	$O(n)$
$O(\cancel{500})$	$O(1)$
$O(\cancel{13}n^2)$	$O(n^2)$

Rule 2 - Smaller Terms Don't Matter

$O(n \cancel{+ 10})$	$O(n)$
$O(1\cancel{000}n + 50)$	$O(n)$
$O(n^2 \cancel{+ 5n} + 8)$	$O(n^2)$

Rule 3 - Arithmetic Operations are Constant

> Arithmetic operations roughly take the same amount of time regardless the size of `n`

Rule 4 - Variable Assignment is Constant

> Variable assignment roughly takes the same amount of time regardless of the size of `n`

Rule 5 - Accessing Element in an Array(by index) or Object(by key) is Constant

> Accessing an element in an array or an object is roughly the same time regardless of `n`

Rule 6 - Loop Complexity is based on length of loop times the inner complexity of the loop

> In a loop, the complexity is the length of the loop times the complexity of whatever happens inside of the loop

B. Examples

1. **Log at least Five**

```
function logAtLeat5(n) {
  for (let i = 0; i < Math.max(5, n); i++){
    console.log(i)
  }
}
```

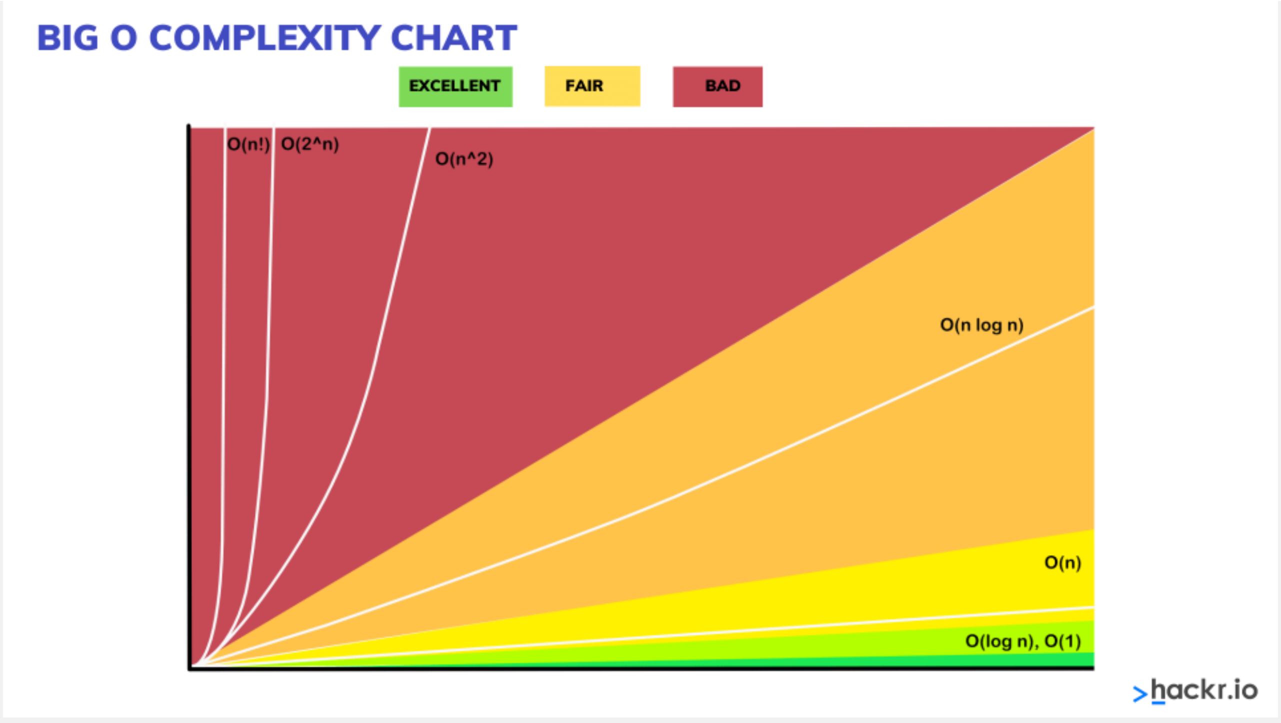
> Big O is `O(n)`

2. **Log at most Five**

```
function logAtMost5(n) {
  for (let i = 0; i < Math.min(5, n); i++) {
    console.log(i);
  }
}
```

> Big O is `O(1)`

C. Big O Complexity Chart



---

7. Space Complexity

A. Space vs Time Complexities

Time Complexity

> How can we analyze the **runtime** of an algorithm as the **size of the inputs** increases

Space Complexity

> How much additional **memory** do we need to allocate in order to **run the code** in our algorithm

B. What about the inputs?

> When we talk about space complexity, we are talking about **auxiallry space complexity**, which refers to the space required by the algorithm, not including the space taken up the inputs. So we don't care about the space taken by the input, since we know that as the input grows, the space needed to accomodate that input will also increase.

A. General Rules

Rule 1 - Most Primitives are Constant Space

> Booleans, numbers, undefined, null, bigint

Rule 2 - Strings are NOT Constant Space

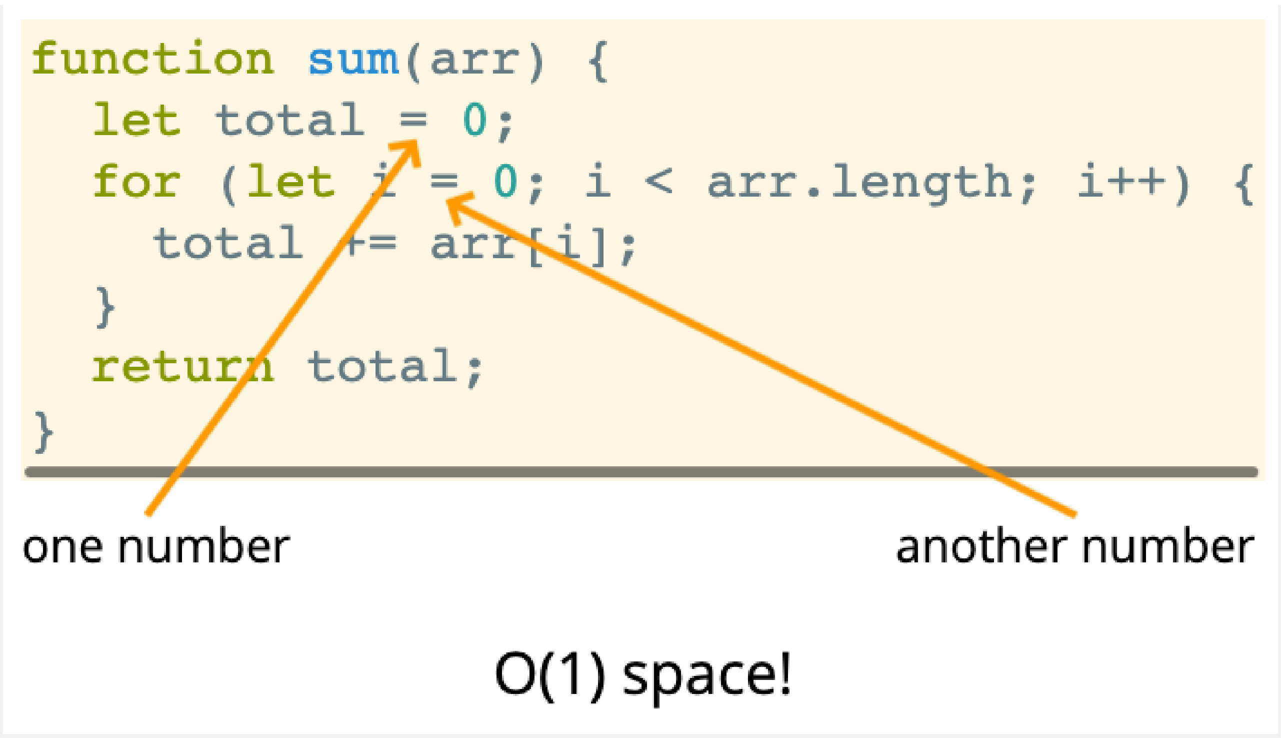
> Strings require `O(n)` space, where `n` is the string length

Rule 3 - Arrays and Objects are generally `O(n)`

> Reference types are generally `O(n)`, where `n` is the length (for arrays) or the number of keys (for objects)

B. Example

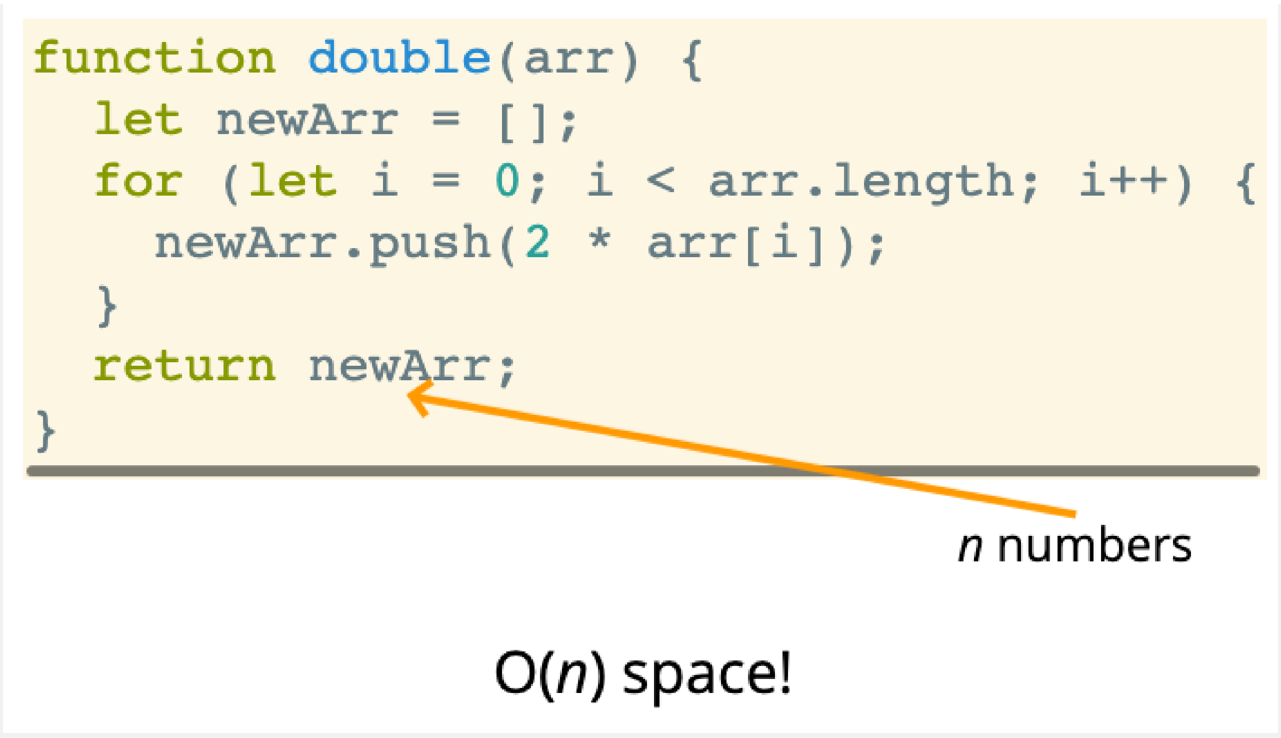
1. Sum



```
function sum(arr) {
  let total = 0;
  for (let i = 0; i < arr.length; i++) {
    total += arr[i];
  }
  return total;
}
```

> No matter the size of the input, the space taken up is constant, therefore, Big O is `O(1)`

2. Double



```
function double(arr) {
  let newArr = [];
  for (let i = 0; i < arr.length; i++) {
    newArr.push(2 * arr[i]);
  }
  return newArr;
}
```

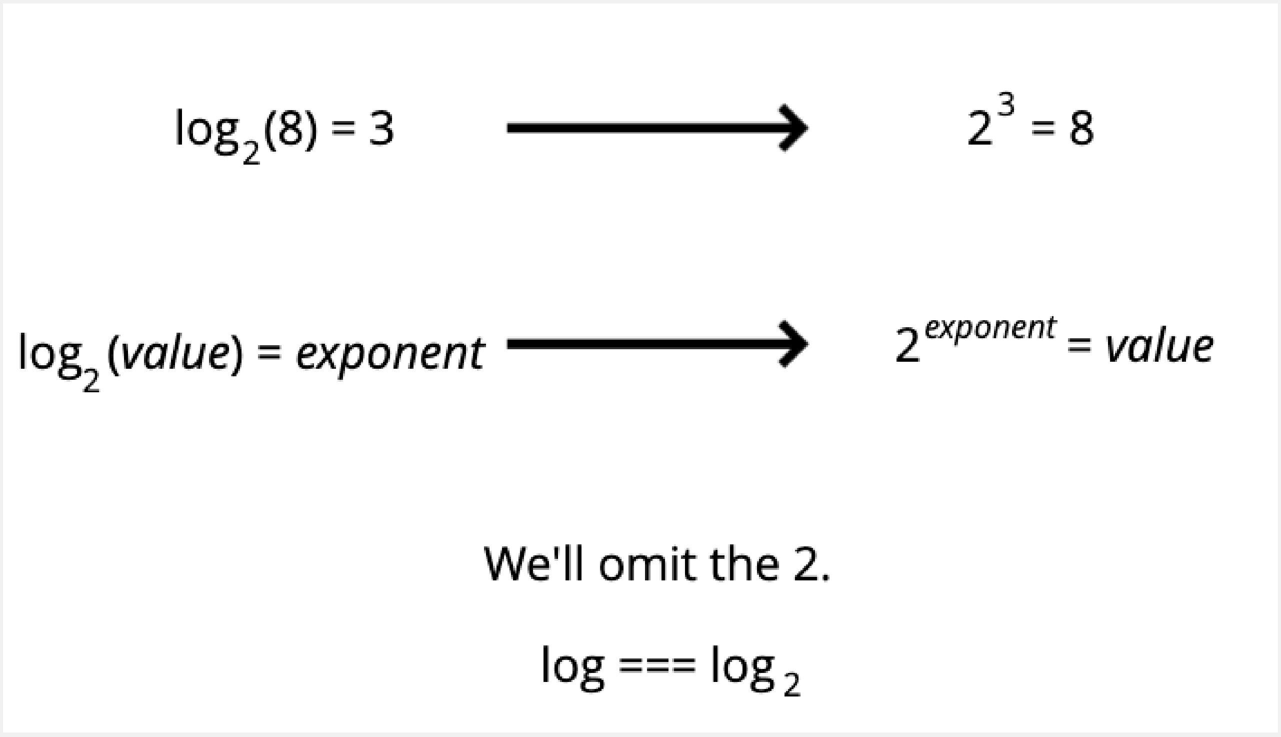
> As the input grows, the newArr grows directly in proportion to it, therefore, Big O is `O(n)`

---

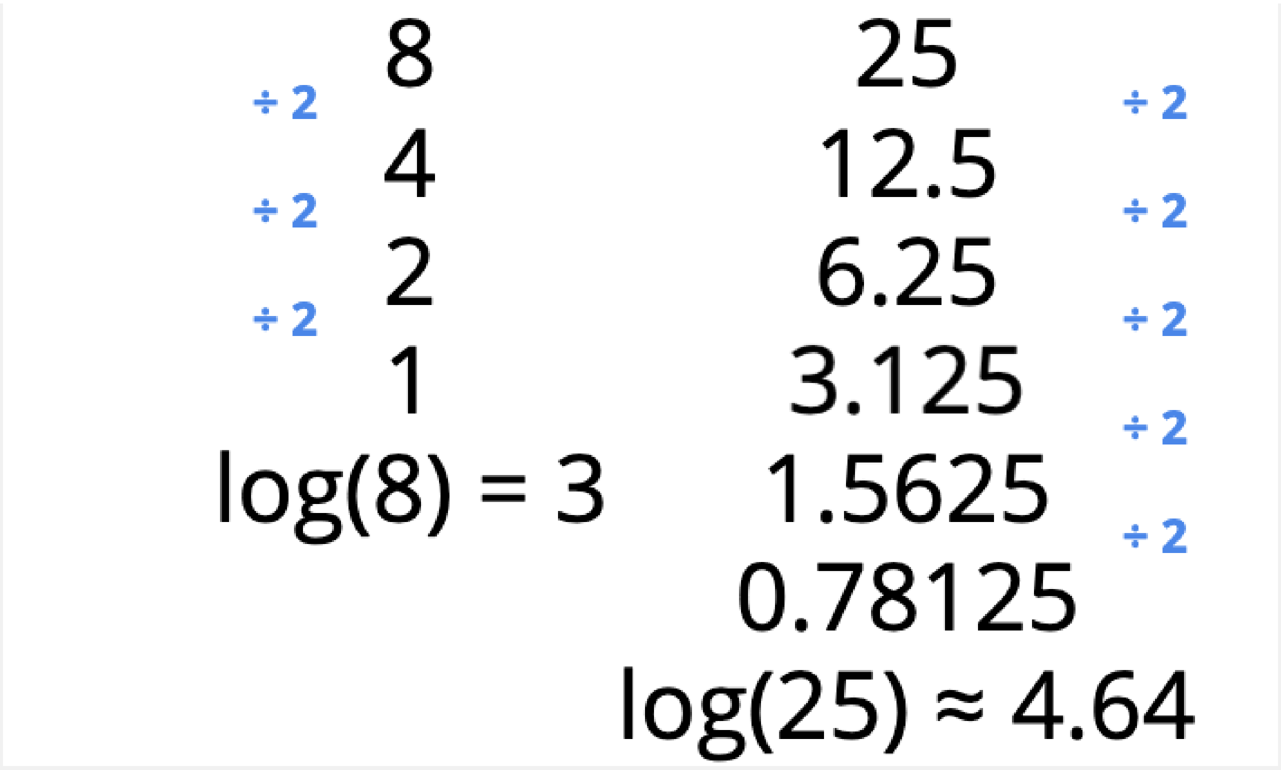
8. Logs and Section Recap

> Sometimes Big O expressions involve more complex mathematical expressions, such as logarithms

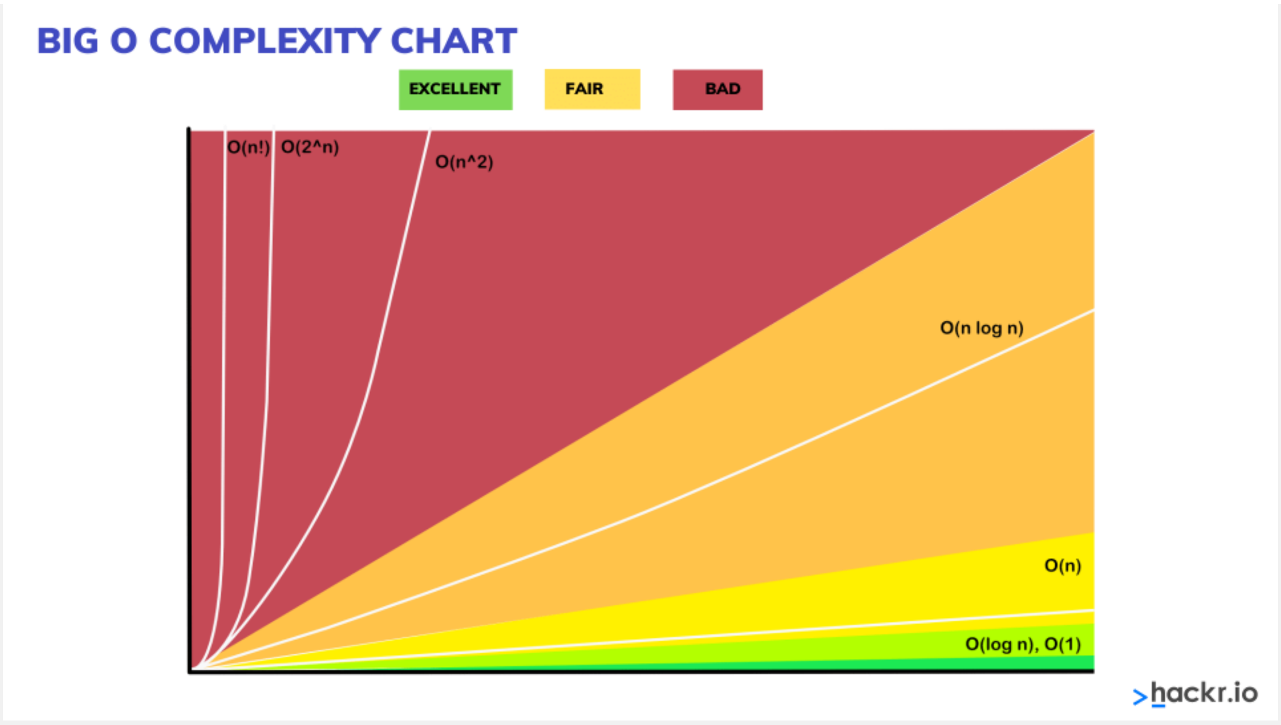
A. What's LOG?

> Logorithm is the inverse of exponentiation
>
> 
>
> $\log_2(8) = 3 \longrightarrow 2^3 = 8$
>
> $\log_2(\mathit{value}) = \mathit{exponent} \longrightarrow 2^{\mathit{exponent}} = \mathit{value}$
>
> We'll omit the 2.
>
> log === $\log_2$
>
> Logarthim isn't always base 2, but we only care about the big picture, the general trend.

B. What does logarithm of a number mean?

> 
>
> 8 ÷ 2 → 4 ÷ 2 → 2 ÷ 2 → 1
>
> $\log(8) = 3$
>
> 25 ÷ 2 → 12.5 ÷ 2 → 6.25 ÷ 2 → 3.125 ÷ 2 → 1.5625 ÷ 2 → 0.78125
>
> $\log(25) \approx 4.64$
>
> The logarithm of a number roughly measures the number of times you divide that number by 2 before you get a value that is **<u>less than or equal to 1.</u>**

C. Logarithmic Time Complexity is GREAT!



- Certain searching algorithms have logarithmic **time** complexity
- Efficient sorting algorithms involve logarithms
- Recursion sometimes involves logarithmic space complexity

Recap

- To analyze the performance of an algorithm, we use Big O notation
- Big O Notation can give us a high level understanding of the time or space (auxillary) complexity of an algorithm
- Big O Notation doesn't care about precision, only about trends
- The time and space complexity only depends on the algorithm, not the hardware used to run the algorithm