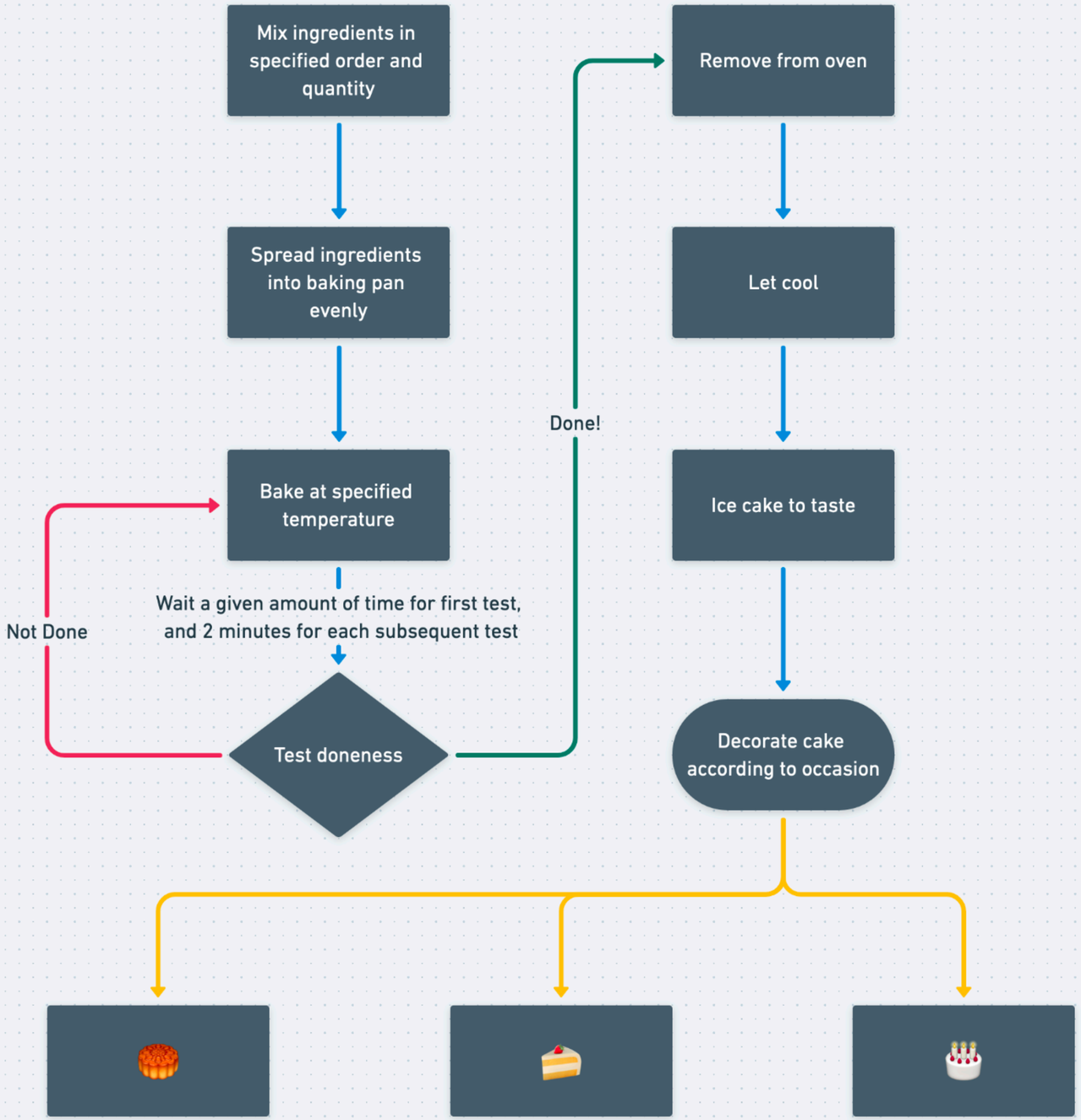
Algorithms & Big O Notation

CS in the Morning! 🌤️

Algorithm

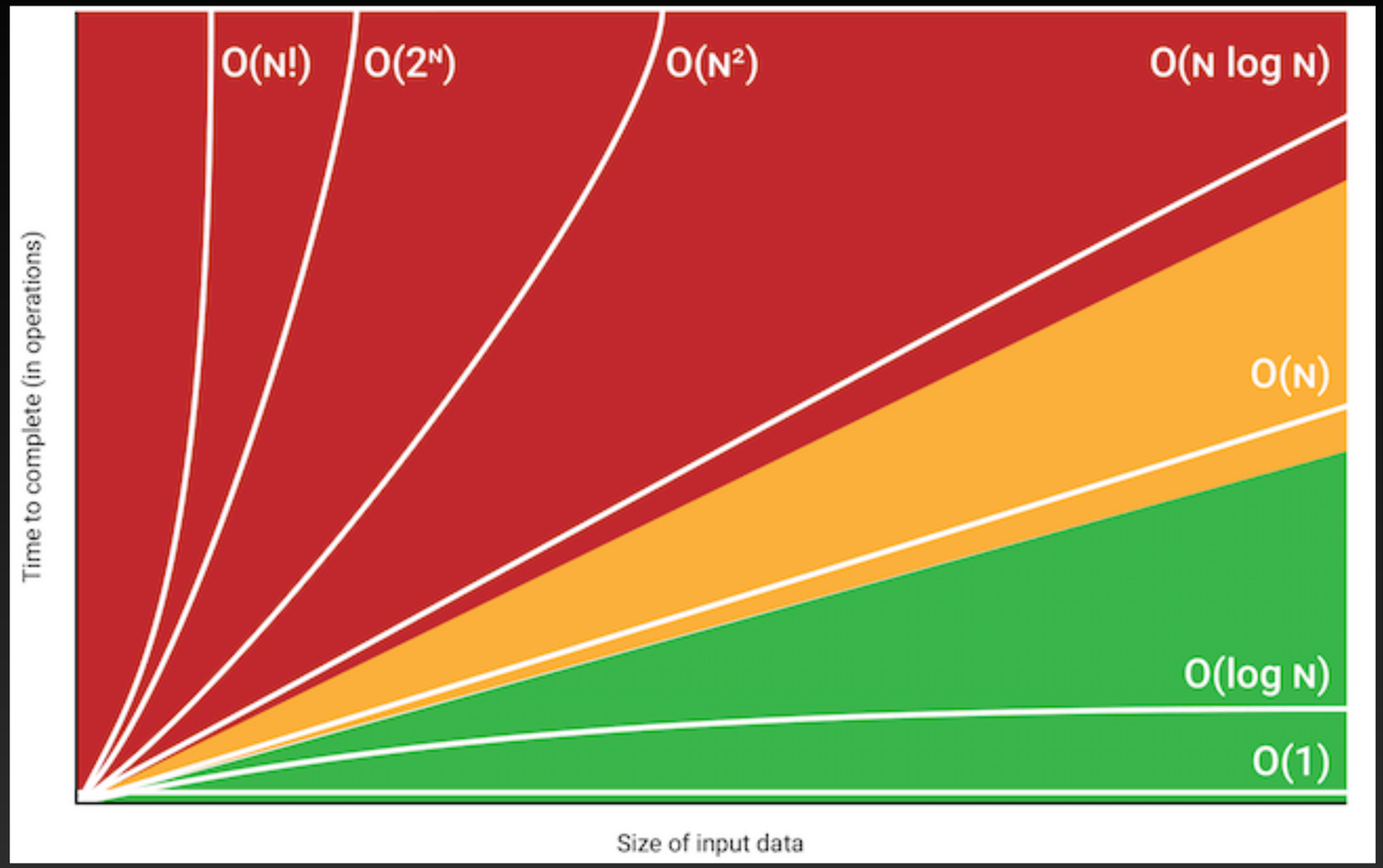
A series of steps taken to solve a problem or achieve a goal.



Big O

Big O time is the language and metric used to describe the efficiency of algorithms.

-Cracking the Coding Interview



src: danielmiessler.com/study/big-o-notation/

Why Does Big O Matter?

- Provides a way to talk about the efficiency of algorithms (broadly described as their *runtime*)
- Allows us to judge what exactly makes an algorithm faster, slower, and more or less space efficient than a similar algorithm
- Which means we can make better programming decisions!

How Though?

- Big O Measures *time complexity (runtimes)* and *space complexity* in a given algorithm
- Big O Notation is written according to its performance in the *worst case scenario*

Scenarios

Best Case

- The best possible result we could expect from an algorithm at runtime. Expressed as Ω.
- Not a useful concept. Many algorithms can get lucky and produce a result of Ω(N)

Expected Case

- The average result we could expect from an algorithm at runtime. Expressed as Θ
- Useful for differentiating searching and sorting algorithms
- The “real world” performance of an algorithm. Outliers should be expected and taken into account, but are not the norm

Worst Case

- How we express the performance of algorithms in Big O Notation
- For some algorithms, worst case scenarios and expected case scenarios are the same

Space Complexity

Space Complexity of Data

- The vast majority of data structures have a space complexity of O(N) - as the size of the data structure increases the space it takes up increases along with it. Many sorts have a space complexity of either O(1) or O(N).

Space Complexity vs. Time Complexity

- While space and time complexity are measured with the same notation, an algorithm will frequently have different space and time complexities

O(1)

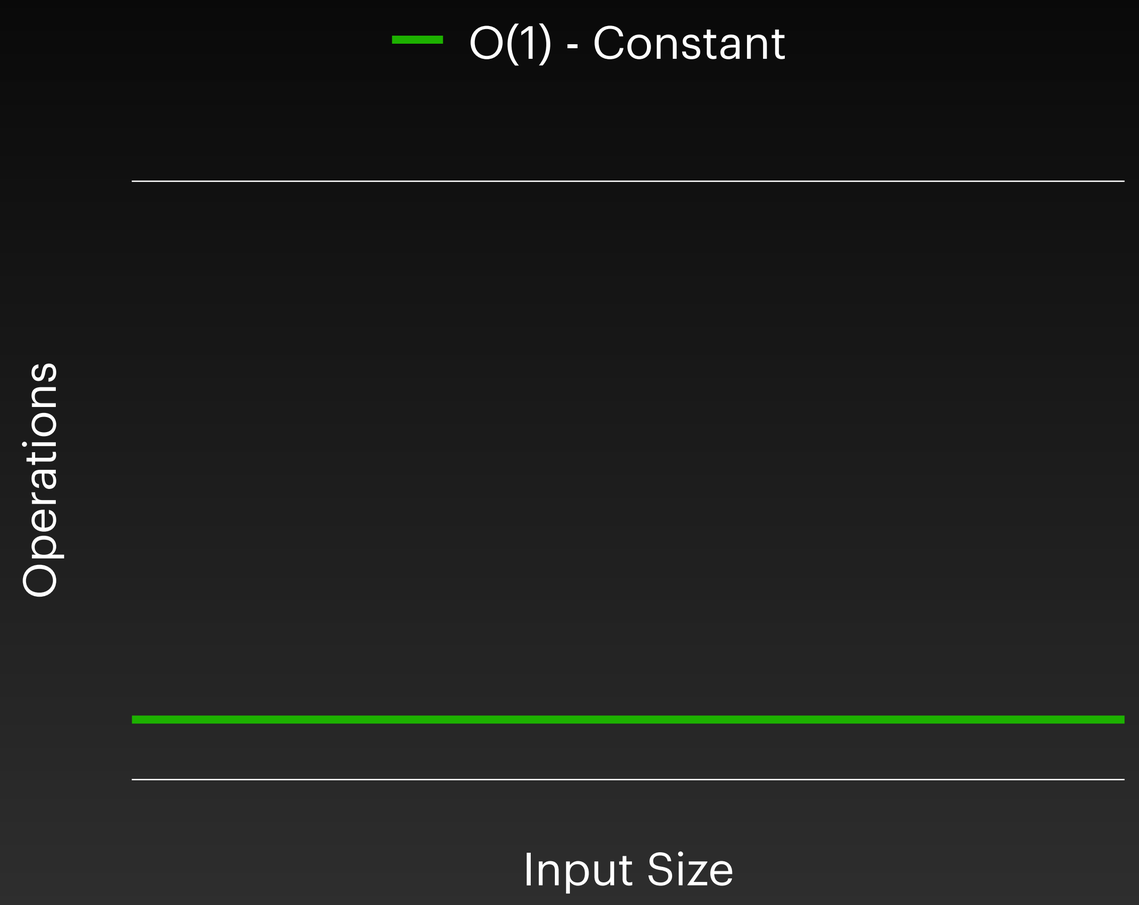
Constant Complexity

O(1)

The big idea:

The input to the algorithm does not matter - the algorithm will still take the same amount of time to run.

O(1) is extremely efficient.



O(1) - Constant Complexity

Examples:

- Determine if a binary number is odd or even
- Access a given index in an array
- Adding or removing an element to a stack
- Adding or removing an element from a queue
- Physically delivering data on a storage device

If 1 item takes 1 second to process, 100 items also take 1 second to process

O(1)
Code Example

In this example, findFifthElementOfArray(tinyArray) and findFifthElementOfArray(largeArray) will take the same amount of time.

```
1  tinyArray = Array(10).fill(0)
2  largeArray = Array(100000000).fill(0)
3
4  findFifthElementOfArray(tinyArray)
5  findFifthElementOfArray(largeArray)
6
7  function findFifthElementOfArray(arr) {
8    return arr[4]
9  }
10
```

O(N)

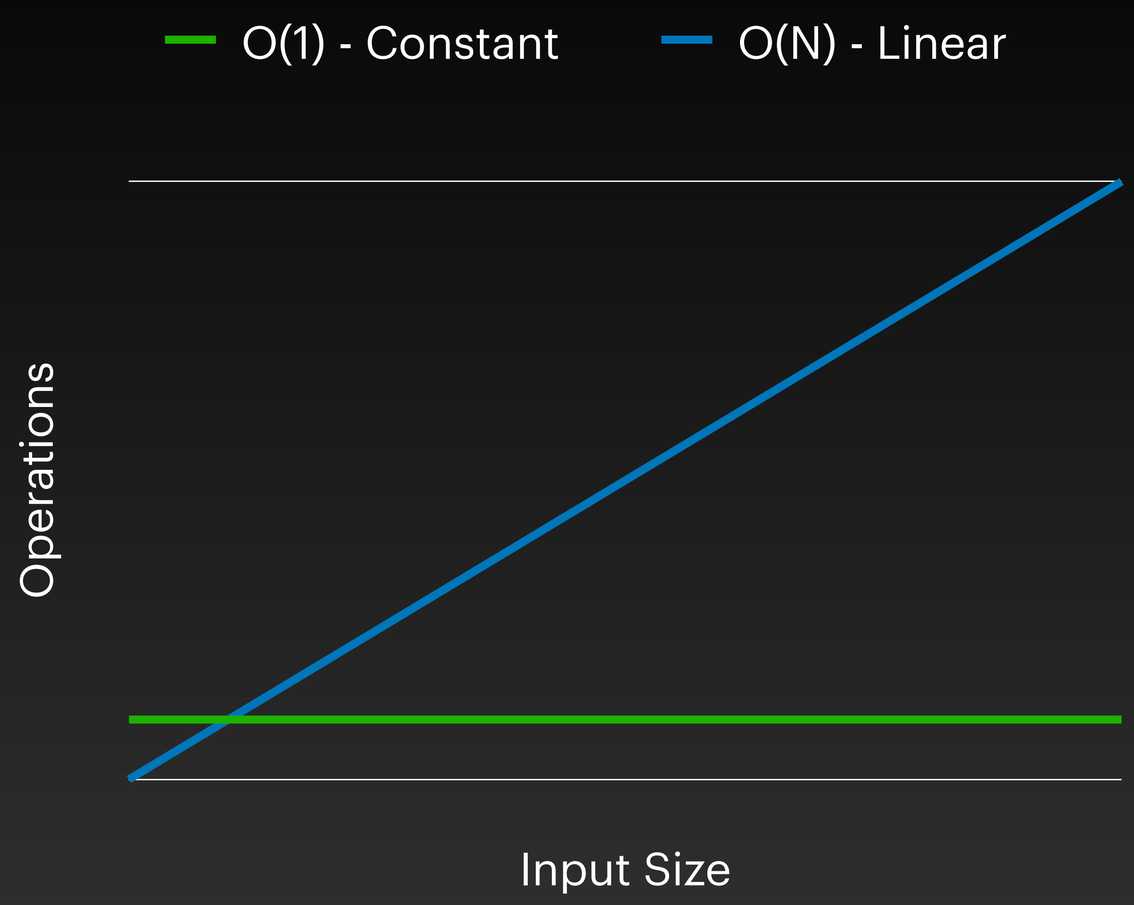
Linear Complexity

O(N)

The big idea:

As the size of the input grows the processing time required by the algorithm will grow at the same pace. N represents the size of the input.

O(N) is somewhat efficient. No matter how big the constant is or how slow the linear increase is, at some point the linear algorithm will have a longer runtime



O(N) - Linear Complexity

Examples:

- Searching an array
- Performing an action on every element in an array.
- Inserting an element alphabetically into an array
- Downloading something (this example overly simplifies this process)
- Searching page by page for a name in a phone book.

If 1 item takes 1 second to process, 100 items take 100 seconds to process

O(N)

Code Example

In this example, findRandomNumberInArray(tinyArray) will complete in no more than 10 iterations.

findRandomNumberInArray(largeArray) will complete in no more than 100,000,000 iterations.

Remember that although a match is likely to be found before the end of an array is reached, Big O Notation is always written assuming the *worst case scenario*.

```
// Creates this array: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
tinyArray = Array.from(Array(10).keys())
// Creates a very large array starting with: [0, 1, 2, 3 ...]
largeArray = Array.from(Array(100000000).keys())

findRandomNumberInArray(tinyArray)
findRandomNumberInArray(largeArray)

function findRandomNumberInArray(arr) {
  randomNum = Math.floor(Math.random() * arr.length)
  for (let i = 0; i < arr.length; i++) {
    if (randomNum === arr[i]) return arr[i]
  }
}
```

Dropping Constants

Dropping Constants

The big idea:

When iterating over the same set of data twice in a single algorithm it may be tempting to label the algorithm as O(2N), but this would be incorrect.

Take these two examples, which one of them is slower?

```
let min = Number.POSITIVE_INFINITY
let max = Number.NEGATIVE_INFINITY
let arr = [10, 4, 2, 7, 9]

arr.forEach(num => {
  if (num < min) min = num
  if (num > max) max = num
})
```

```
let min = Number.POSITIVE_INFINITY
let max = Number.NEGATIVE_INFINITY
let arr = [10, 4, 2, 7, 9]

arr.forEach(num => {
  if (num < min) min = num
})

arr.forEach(num => {
  if (num > max) max = num
})
```

Figuring this out for every algorithm we would write would ultimately be unproductive.

Remember that the ultimate goal of Big O is to determine the major impacts on the runtime of an algorithm as the input scales.

In reality, O(N) algorithms aren't the same as one another, but **they scale in the same way as their inputs grow or shrink**.

```
let min = Number.POSITIVE_INFINITY
let max = Number.NEGATIVE_INFINITY
let arr = [10, 4, 2, 7, 9]

arr.forEach(num => {
  if (num < min) min = num
  if (num > max) max = num
})
```

```
let min = Number.POSITIVE_INFINITY
let max = Number.NEGATIVE_INFINITY
let arr = [10, 4, 2, 7, 9]

arr.forEach(num => {
  if (num < min) min = num
})

arr.forEach(num => {
  if (num > max) max = num
})
```

O(log(N))

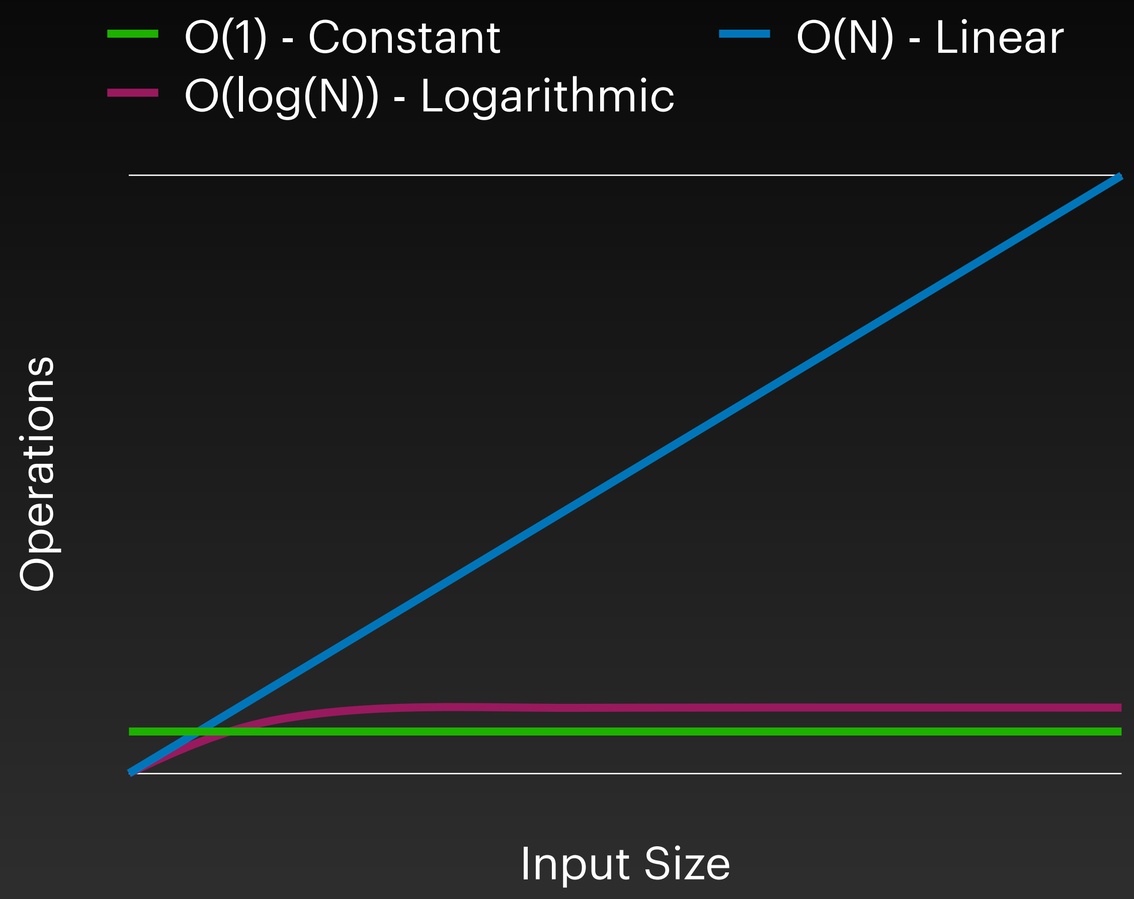
Logarithmic Complexity

O(log(N))

The big idea:

For each time the input grows the processing time required by the algorithm will increase by half what it previously increased by. N represents the size of the input.

O(log(N)) is very efficient. By their nature these algorithms can solve complex problems very quickly.



O(log(N)) - Logarithmic Complexity

Examples:

- Many kinds of tree data structures, such as a Binary Search Tree
- Looking up a number in a phone book by searching one half of it at a time.

If 1 item takes 1 second to process, 100 items take 3 seconds to process

O(log(N))

Code Example

To avoid covering binary searches, this is not a practical code example, but purely demonstrates the concept of O(log(N)).

In 24 iterations, the algorithm reduces the input from over 16 million to 1

```
let n = 16777216

log(n)

function log(n) {
  let j = 0
  for (let i = n; i > .999; i /= 2) {
    let result = i;
    console.log(`The result of iteration ${j} is ${result}`)
    j++
  }
}
```

```
The result of iteration 0 is 16777216
The result of iteration 1 is 8388608
The result of iteration 2 is 4194304
The result of iteration 3 is 2097152
The result of iteration 4 is 1048576
The result of iteration 5 is 524288
The result of iteration 6 is 262144
The result of iteration 7 is 131072
The result of iteration 8 is 65536
The result of iteration 9 is 32768
The result of iteration 10 is 16384
The result of iteration 11 is 8192
The result of iteration 12 is 4096
The result of iteration 13 is 2048
The result of iteration 14 is 1024
The result of iteration 15 is 512
The result of iteration 16 is 256
The result of iteration 17 is 128
The result of iteration 18 is 64
The result of iteration 19 is 32
The result of iteration 20 is 16
The result of iteration 21 is 8
The result of iteration 22 is 4
The result of iteration 23 is 2
The result of iteration 24 is 1
```

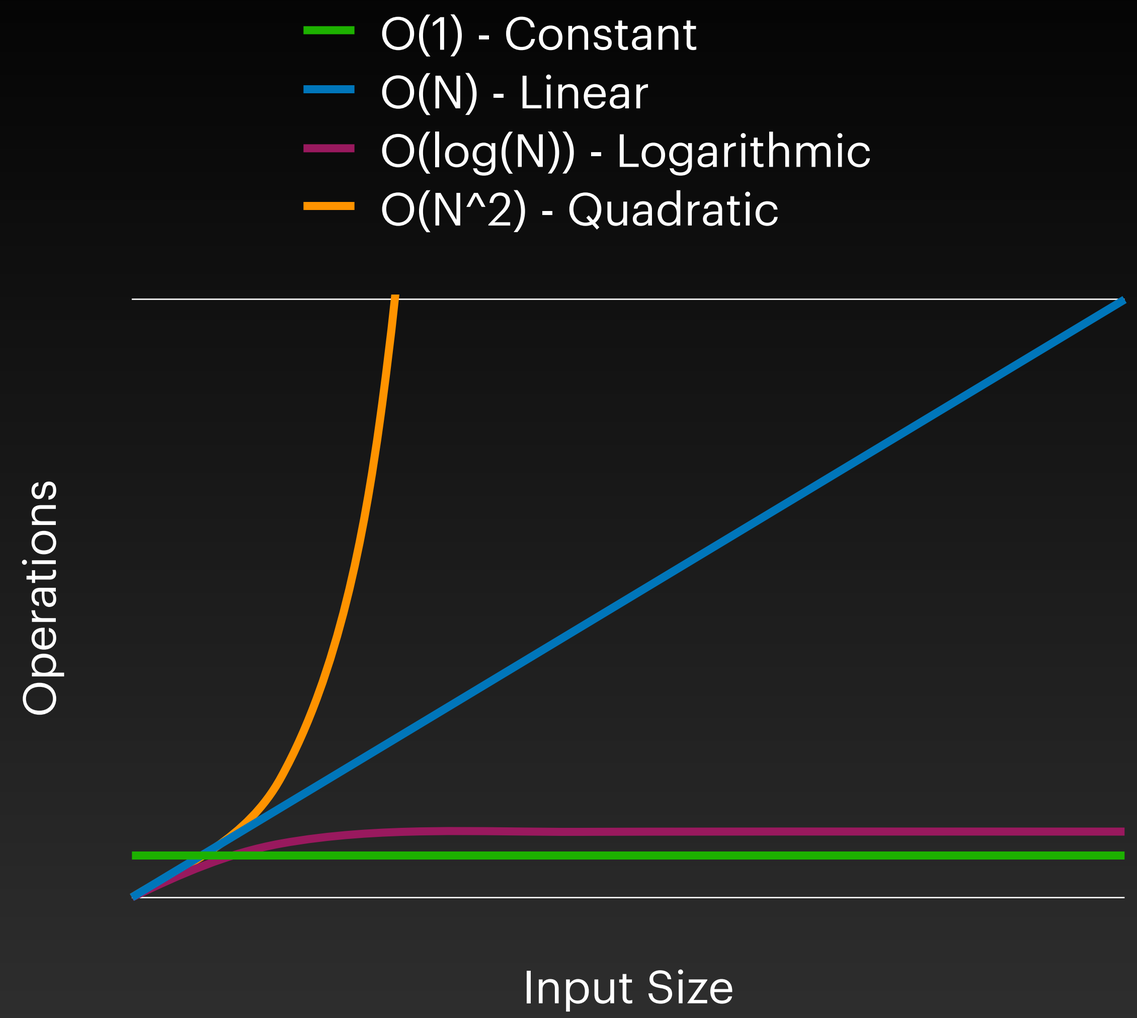
Quadratic Complexity (Also written as O(N^2)

$O(N^2)$

The big idea:

For each time the input grows the processing time required by the algorithm will grow exponentially. N represents the size of the input.

$O(N^2)$ is inefficient and should be avoided if possible.



$O(N^2)$ - Quadratic Complexity

Examples:

- Many sorting algorithms have quadratic complexity (quicksort, bubble sort, insertion sort, etc.)
- Performing an action on every item in a 2D array
- Searching in a 2D array
- Nested loops.

If 1 item takes 1 second to process, 100 items take 10,000 seconds to process

O(N²)

Code Example

This example demonstrates looping through a 2D array (using a normal array). There is an initial step for each input - and then a follow up step that runs from beginning to end of the array.

```
logArray([1,2,3,4,5])

function logArray(arr){
  for(let i=0; i<arr.length; i++){
    console.log(arr[i])
    for(let j=0; j<arr.length; j++){
      console.log('i: ', arr[i], 'j: ', arr[j])
    }
  }
}
```

```
1
i:  1 j:  1
i:  1 j:  2
i:  1 j:  3
i:  1 j:  4
i:  1 j:  5
2
i:  2 j:  1
i:  2 j:  2
i:  2 j:  3
i:  2 j:  4
i:  2 j:  5
3
i:  3 j:  1
i:  3 j:  2
i:  3 j:  3
i:  3 j:  4
i:  3 j:  5
4
i:  4 j:  1
i:  4 j:  2
i:  4 j:  3
i:  4 j:  4
i:  4 j:  5
5
i:  5 j:  1
i:  5 j:  2
i:  5 j:  3
i:  5 j:  4
i:  5 j:  5
```

Dropping Non-Dominant Terms

Dropping Non-Dominant Terms

The big idea:

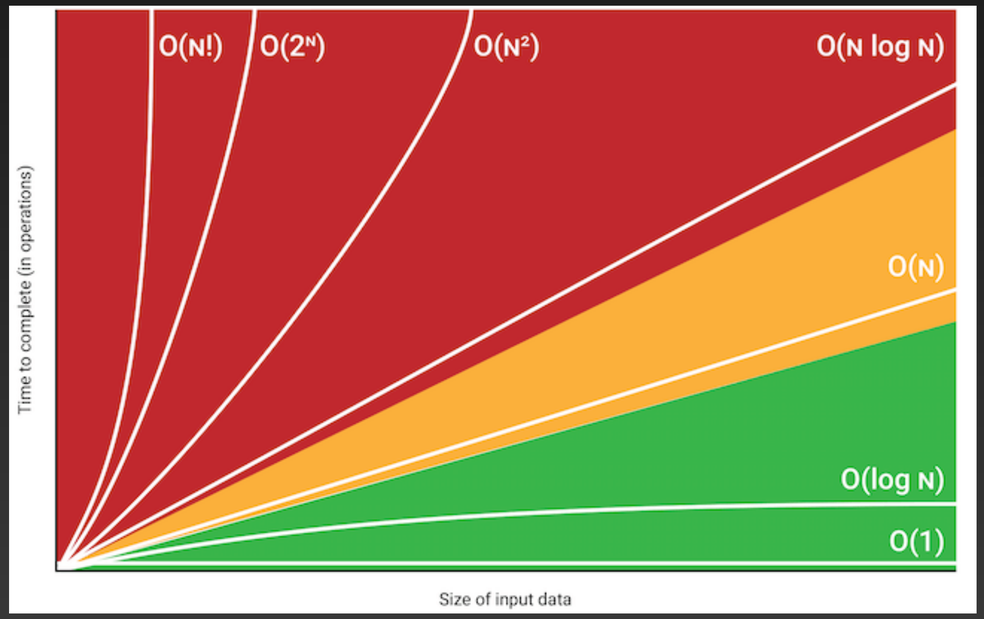
When calculating Big O we dispose of all but the most impactful terms. This previously used example of an $O(N^2)$ algorithm has an O(N) instruction appended to the end of it.

```
logArray([1,2,3,4,5])

function logArray(arr){
  for(let i=0; i<arr.length; i++){
    console.log(arr[i])
    for(let j=0; j<arr.length; j++){
      console.log('i: ', arr[i], 'j: ', arr[j])
    }
  }
  for(let i=0; i<arr.length; i++){
    console.log(arr[i])
  }
}
```

This instruction will continue to become less relevant as the input array grows.

Keep this chart in mind:




```
logArray([1,2,3,4,5])

function logArray(arr){
  for(let i=0; i<arr.length; i++){
    console.log(arr[i])
    for(let j=0; j<arr.length; j++){
      console.log('i: ', arr[i], 'j: ', arr[j])
    }
  }
  for(let i=0; i<arr.length; i++){
    console.log(arr[i])
  }
}
```

Adding and Multiplying Runtimes

Curveball incoming!

Multi-part algorithms

An algorithm may deal with multiple sets of data or the same set of data multiple times.

When this occurs we should add or multiply the runtimes of each set of data.

So when do we add the runtimes together and when do we multiply them?

Adding Runtimes

The big idea:

If your algorithm is in the form “do this, then, when you’re all done, do that” then you add the runtimes.

— *Cracking the Coding Interview*

```
logTwoArrays(["A", "B", "C", "D"], [1, 2, 3, 4])

function logTwoArrays(arrA, arrB){
  for(let i=0; i<arrA.length; i++){
    console.log(arrA[i])
  }
  for(let j=0; j<arrB.length; j++){
    console.log(arrB[j])
  }
}
```

In this example, the runtime is influenced by the size of arrA and the size of arrB, so the Big O notation is O(arrA + arrB). This may be abstracted to O(N + M).

Multiplying Runtimes

The big idea:

If your algorithm is in the form “do this for each time you do that” then you multiply the runtimes.

— *Cracking the Coding Interview*

```
1   logTwoArrays(["A", "B", "C", "D"], [1, 2, 3, 4])
2
3   function logTwoArrays(arrA, arrB){
4     for(let i=0; i<arrA.length; i++){
5       console.log(arrA[i])
6       for(let j=0; j<arrB.length; j++){
7         console.log(arrB[j])
8       }
9     }
10  }
11
```

In this example, we iterate through arrB for every item in arrA, so the runtimes are multiplied together making the Big O Notation O(arrA * arrB), which may be abstracted as O(N * M)

Heads up!

We do not drop terms from different data sets when we are not aware of the data they contain, because we are unable to ascertain if they will be dominant.

In this example, we iterate through arrA for every item in arrA. The runtimes are multiplied together making the Big O Notation O(arrA * arrA) or more simply O($arrA^2$) (abstracted as O(N^2), *then* we add the runtime of arrB making the final Big O notation of this function O($arrA^2$ + arrB) which may be abstracted as O (N^2 + M)

```
3   function logTwoArrays(arrA, arrB){
4     for(let i=0; i<arrA.length; i++){
5       console.log(arrA[i])
6       for(let j=0; j<arrA.length; j++){
7         console.log(arrA[j])
8       }
9     }
10    for(let j=0; j<arrB.length; j++){
11      console.log(arrB[j])
12    }
13  }
```

Heads up!

This runs counter to how we calculate Big O when engaging with a single data input, because we always know that the single for loop will be a non-dominate term compared to the nested for loop.

In this example, the Big O notation is $O(N^2)$ - we have dropped the non-dominant loop through arrA.

```
3   function logTwoArrays(arrA, arrB){
4     for(let i=0; i<arrA.length; i++){
5       console.log(arrA[i])
6       for(let j=0; j<arrA.length; j++){
7         console.log(arrA[j])
8       }
9     }
10    for(let j=0; j<arrA.length; j++){
11      console.log(arrA[j])
12    }
13  }
14
```

big o memes

All More Settings Tools

Your search - **big o memes** - did not match any documents.

Suggestions:

- Make sure all words are spelled correctly.
- Try different keywords.
- Try more general keywords.
- Try fewer keywords.

O(N log(N))

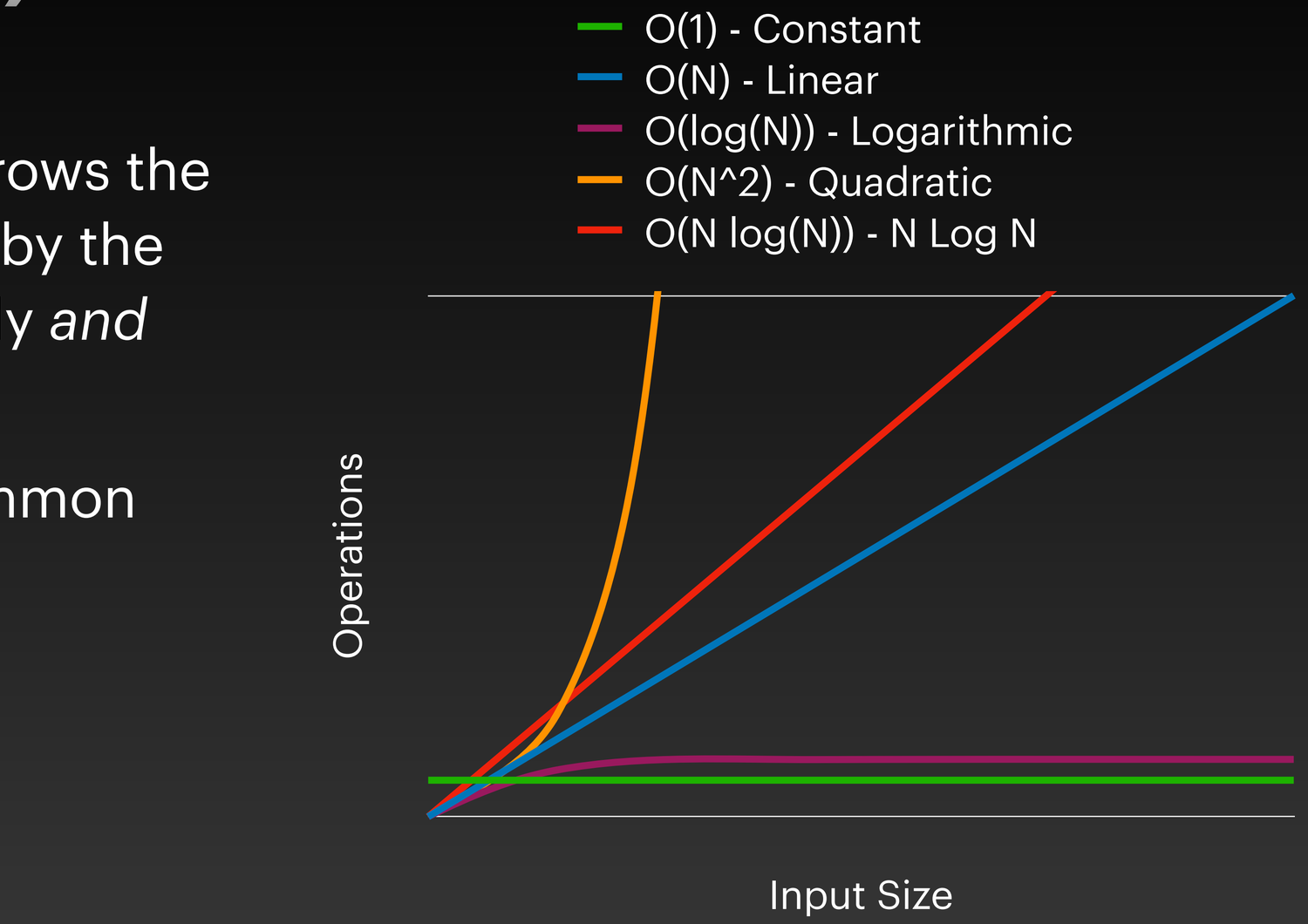
N log N Complexity

O(N log (N))

The big idea:

For each time the input grows the processing time required by the algorithm will grow linearly *and* logarithmically.

This Big O variant is a common runtime of many sorting algorithms.



O(N log(N)) - N Log N Complexity

Examples:

- Many sorting algorithms such as:
- Merge Sort
- Tim Sort
- Heap Sort

If 1 item takes 1 second to process, 100 items take 300 seconds to process

O(N log(N))

Code Example

To avoid covering sorting, this is not a practical code example, but purely demonstrates the concept of O(N log(N)).

This code block simulates performing a logarithmic function on every item in an array.

```
log(8);
function log(n) {
  let j = 1;
  for (let k = 0; k < n; k++) {
    for (let i = n; i > 0.999; i /= 2) {
      let result = i;
      console.log(`The result of iteration ${j} is ${result}`);
      j++;
    }
  }
}
```

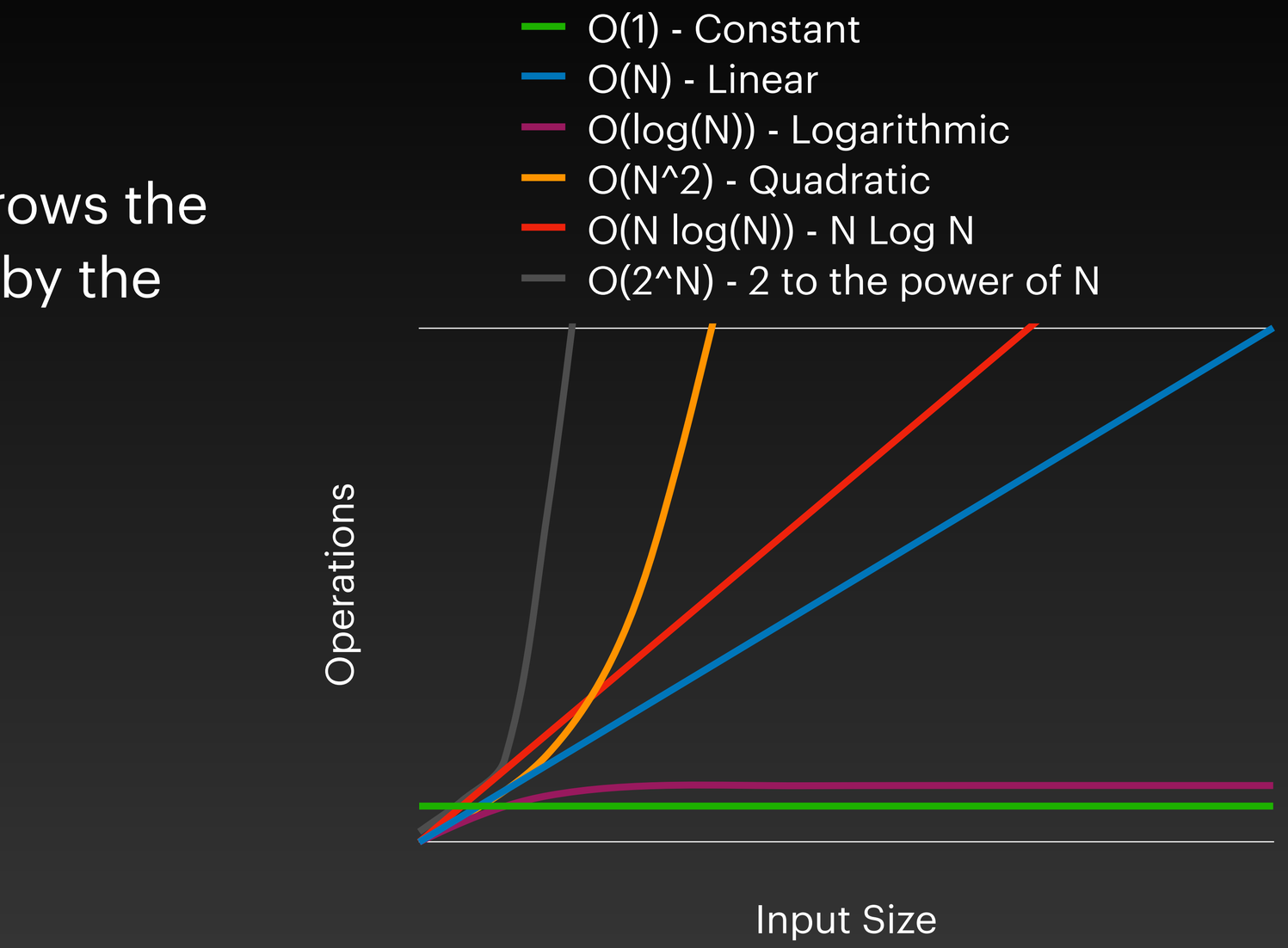
```
The result of iteration 1 is 8
The result of iteration 2 is 4
The result of iteration 3 is 2
The result of iteration 4 is 1
The result of iteration 5 is 8
The result of iteration 6 is 4
The result of iteration 7 is 2
The result of iteration 8 is 1
The result of iteration 9 is 8
The result of iteration 10 is 4
The result of iteration 11 is 2
The result of iteration 12 is 1
The result of iteration 13 is 8
The result of iteration 14 is 4
The result of iteration 15 is 2
The result of iteration 16 is 1
The result of iteration 17 is 8
The result of iteration 18 is 4
The result of iteration 19 is 2
The result of iteration 20 is 1
The result of iteration 21 is 8
The result of iteration 22 is 4
The result of iteration 23 is 2
The result of iteration 24 is 1
The result of iteration 25 is 8
The result of iteration 26 is 4
The result of iteration 27 is 2
The result of iteration 28 is 1
The result of iteration 29 is 8
The result of iteration 30 is 4
The result of iteration 31 is 2
The result of iteration 32 is 1
```

Power of 2 Complexity - also written as 2^N

$O(2^N)$

The big idea:

For each time the input grows the processing time required by the algorithm will double.



$O(2^N)$ - Power of 2 Complexity

Examples:

- A fibonacci sequence
- Calling a function recursively twice within a recursive function.

If 1 item takes 1 second to process, 100 items take 1,267,650,600,228,229,401,496,703,205,376 (over 1 nonillion) seconds to process

$O(2^N)$

Code Example

This code creates a fibonacci sequence, and returns the value at the nth value of the sequence. The code that makes this a Power of 2 Complexity is line 4 of the fibonacci function - note the fibonacci is called twice.

```
1  function fibonacci(n) {
2    return n <= 1 ? 0
3         : n <= 2 ? 1
4         : fibonacci(n - 1) + fibonacci(n - 2);
5  }
6
```

O(N!)

Factorial Complexity

O(N!)

The big idea:

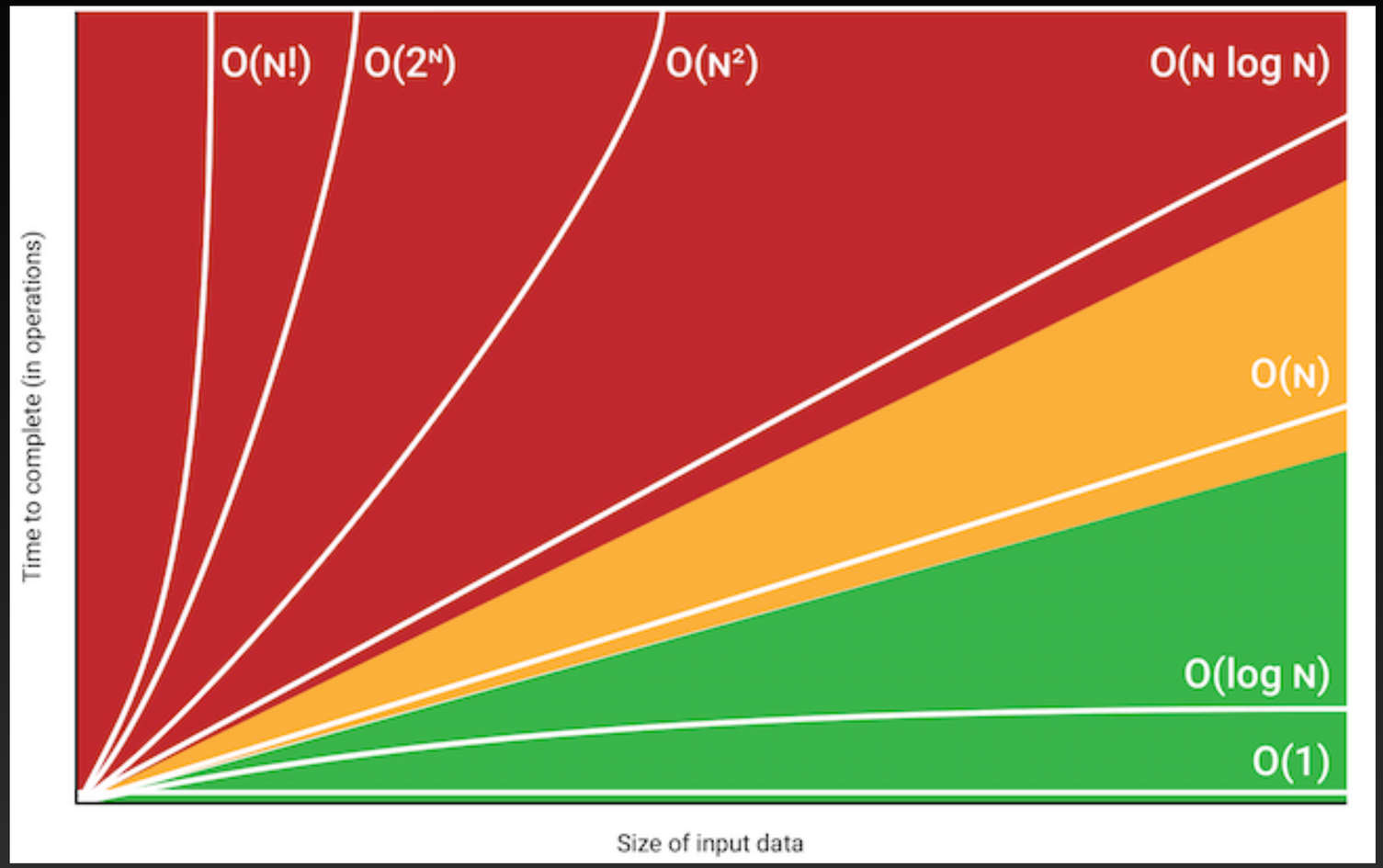
Factorial Complexity occurs when every possible solution to a problem may have to be calculated for the correct output to be determined. An example of this would be a brute force solution to the traveling salesperson problem - where the shortest possible route between a series of locations has to be determined.

If 1 item takes 1 second to process, 100 items will take an extremely long time to be processed

Big O

Big O time is the language and metric used to describe the efficiency of algorithms.

-Cracking the Coding Interview



src: danielmiessler.com/study/big-o-notation/

Examples

```
function sumAndProductOfArray(arr) {
  let sum = 0
  let product = 1
  for (let i = 0; i < arr.length; i++) {
    sum += arr[i]
  }
  arr.forEach(element => {
    product *= element
  })
  console.log(`Sum: ${sum}`)
  console.log(`Product: ${product}`)
}

```

O(N)

Linear Complexity. The function has a complexity of 2N, but remember that we drop constants, making it just O(N)

```
function logPairs(arr) {
  for (let i = 0; i < arr.length; i++) {
    for (let j = 0; j < arr.length; j++) {
      console.log("i: ", arr[i], "j: ", arr[j]);
    }
  }
}
```

$O(N^2)$

Quadratic Complexity. The inner loop has O(N) iterations and is called N times.
This makes the runtime O(N*N) or more simply $O(N^2)$

```
function logPairs(arrA, arrB) {
  for (let i = 0; i < arrA.length; i++) {
    for (let j = 0; j < arrB.length; j++) {
      console.log("i: ", arrA[i], "j: ", arrB[j]);
    }
  }
}
```

O(arrA*arrB)

Quadratic Complexity. The inner loop has O(arrB) iterations and is called arrA times. This makes the runtime O(arrA*arrB) or abstracted as O(N*M)

```
function logPairs(arrA, arrB) {
  for (let i = 0; i < arrA.length; i++) {
    for (let j = 0; j < arrB.length; j++) {
      for (let k = 0; k < 100000; k++) {
        console.log("i: ", arrA[i], "j: ", arrB[j])
      }
    }
  }
}
```

O(arrA*arrB)

Quadratic Complexity. The inner loop has O(arrB) iterations and is called arrA times. The inner loop is called a constant number of times (100,000) making the unsimplified Big O Notation O(100,000arrA*arrB). Remember we drop constants, so the final notation is simplified to O(arrA*arrB)

Which of the following are equivalent to O(N)?

1) O(N + M), where M < N/2

2) O(2N)

3) O(N + log N)

4) O(N + M)

Which of the following are equivalent to O(N)?

1) O(N + P), where P < N/2 - When P is less than half of N it is not significant enough to include, so it can be simplified to O(N)

2) O(2N) - We can drop the constant 2, leaving O(N)

3) O(N + log N) - log N is a non-dominant term, so it can be dropped, leaving us with O(N)

4) O(N + M) - Without knowing anything about N or M we are unable to simplify this further

Algorithms & Big O Notation

CS in the Morning! 🌤️