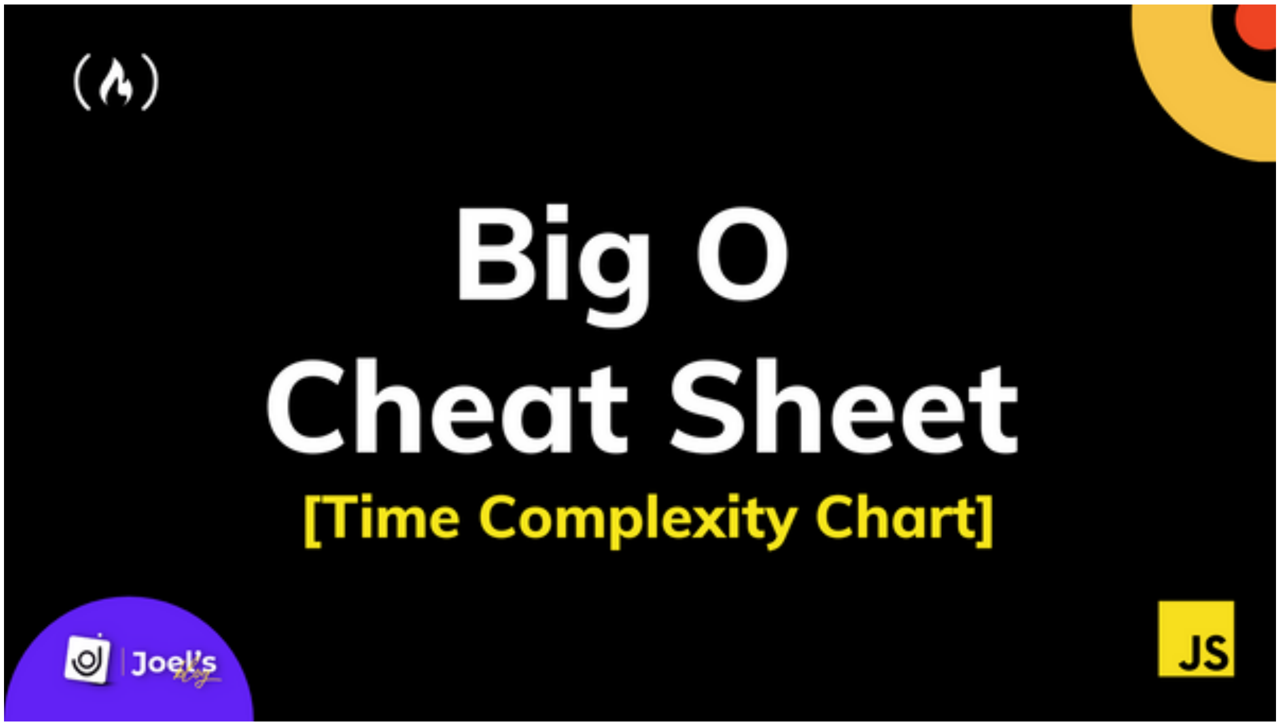
OCTOBER 5, 2022 / #BIG O NOTATION

Big O Cheat Sheet – Time Complexity Chart

Joel Olawanle



An algorithm is a set of well-defined instructions for solving a specific problem. You can solve these problems in various ways.

This means that the method you use to arrive at the same solution may differ from mine, but we should both get the same result.

Because there are various ways to solve a problem, there must be a way to evaluate these solutions or algorithms in terms of performance and efficiency (the time it will take for your algorithm to run/execute and the total amount of memory it will consume).

This is critical for programmers to ensure that their applications run properly and to help them write clean code.

This is where Big O Notation enters the picture. Big O Notation is a metric for determining the efficiency of an algorithm. It allows you to estimate how long your code will run on different sets of inputs and measure how effectively your code scales as the size of your input increases.

What is Big O?

Big O, also known as Big O notation, represents an algorithm's worst-case complexity. It uses algebraic terms to describe the complexity of an algorithm.

fast your algorithm's runtime is.

Big O notation measures the efficiency and performance of your algorithm using time and space complexity.

What is Time and Space Complexity?

One major underlying factor affecting your program's performance and efficiency is the hardware, OS, and CPU you use.

But you don't consider this when you analyze an algorithm's performance. Instead, the time and space complexity as a function of the input's size are what matters.

An algorithm's time complexity specifies how long it will take to execute an algorithm **as a function of its input size**. Similarly, an algorithm's space complexity specifies the total amount of space or memory required to execute an algorithm **as a function of the size of the input**.

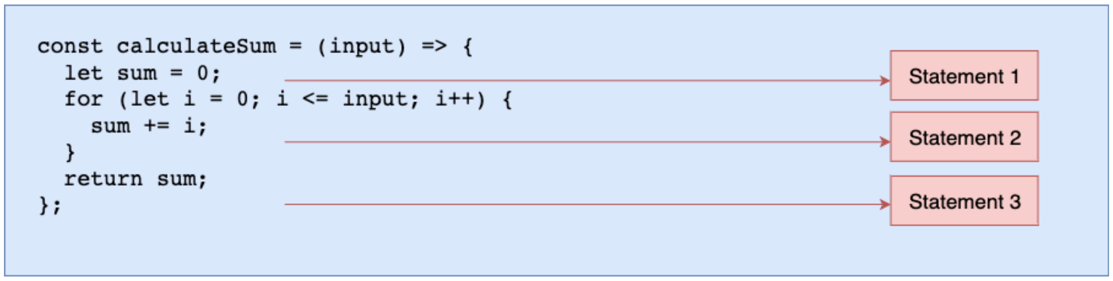
We will be focusing on time complexity in this guide. This will be an in-depth cheatsheet to help you understand how to calculate the time complexity for any algorithm.

Why is time complexity a function of its input size?

To perfectly grasp the concept of "as a function of input size," imagine you have an algorithm that computes the sum of numbers based on your input. If your input is 4, it will add 1+2+3+4 to output 10; if your input is 5, it will output 15 (meaning 1+2+3+4+5).

```
const calculateSum = (input) => {
  let sum = 0;
  for (let i = 0; i <= input; i++) {
    sum += i;
  }
  return sum;
};
```

In the code above, we have three statements:



```
const calculateSum = (input) => {
  let sum = 0;
  for (let i = 0; i <= input; i++) {
    sum += i;
  }
  return sum;
};
```

Looking at the image above, we only have three statements. Still, because there is a loop, the second statement will be executed based on the input size, so if the input is four, the second statement (statement 2) will be executed four times, meaning the entire algorithm will run six (4 + 2) times.

In plain terms, the algorithm will run **input + 2** times, where input can be any number. This shows that **it's expressed in terms of the input. In other words, it is a function of the input size**.

- Linear time: O(n)
- Logarithmic time: O(n log n)
- Quadratic time: O(n^2)
- Exponential time: O(2^n)
- Factorial time: O(n!)

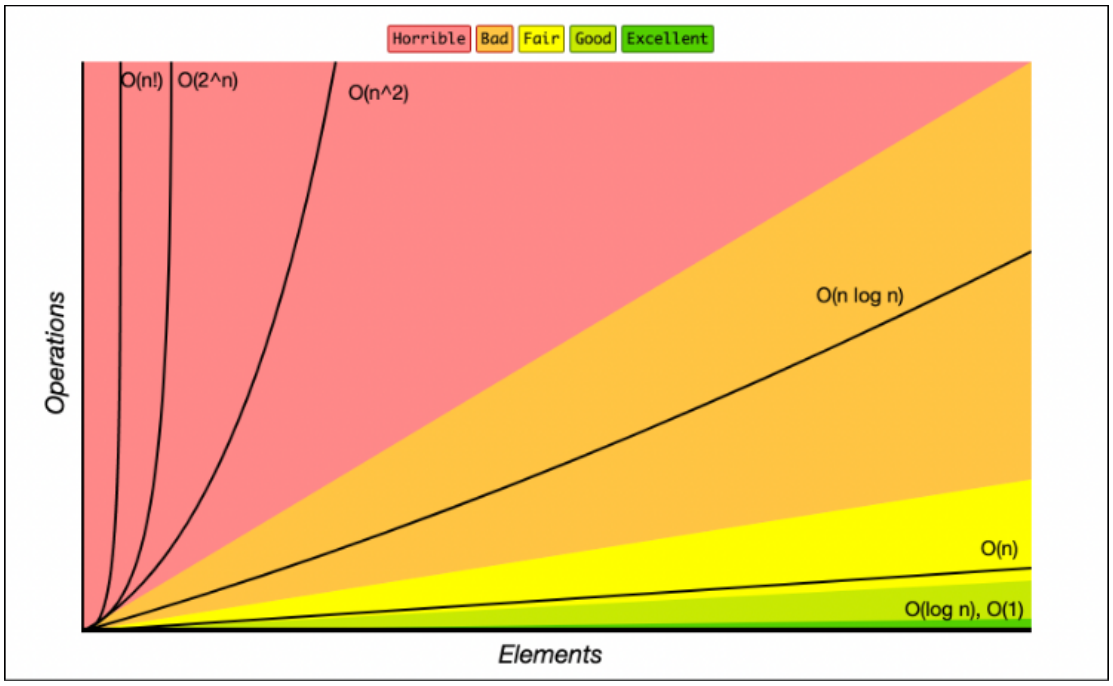
Before we look at examples for each time complexity, let's understand the Big O time complexity chart.

Big O Complexity Chart

The Big O chart, also known as the Big O graph, is an asymptotic notation used to express the complexity of an algorithm or its performance as a function of input size.

This helps programmers identify and fully understand the worst-case scenario and the execution time or memory required by an algorithm.

The following graph illustrates Big O complexity:



The Big O chart above shows that O(1), which stands for constant time complexity, is the best. This implies that your algorithm processes only one statement without any iteration. Then there's O(log n), which is good, and others like it, as shown below:

- **O(1)** - Excellent/Best
- **O(log n)** - Good
- **O(n)** - Fair

You now understand the various time complexities, and you can recognize the best, good, and fair ones, as well as the bad and worst ones (always avoid the bad and worst time complexity).

The next question that comes to mind is how you know which algorithm has which time complexity, given that this is meant to be a cheatsheet 😂.

- When your calculation is not dependent on the input size, it is a constant time complexity (O(1)).
- When the input size is reduced by half, maybe when iterating, handling recursion, or whatsoever, it is a logarithmic time complexity (O(log n)).
- When you have a single loop within your algorithm, it is linear time complexity (O(n)).
- When you have nested loops within your algorithm, meaning a loop in a loop, it is quadratic time complexity (O(n^2)).
- When the growth rate doubles with each addition to the input, it is exponential time complexity (O2^n).

Let's begin by describing each time's complexity with examples. It's important to note that I'll use JavaScript in the examples in this guide, but the programming language isn't important as long as you understand the concept and each time complexity.

Big O Time Complexity Examples

Constant Time: O(1)

When your algorithm is not dependent on the input size n, it is said to have a constant time complexity with order O(1). This means that the run time will always be the same regardless of the input size.

For example, if an algorithm is to return the first element of an array. Even if the array has 1 million elements, the time complexity will be constant if you use this approach:

```
const firstElement = (array) => {
  return array[0];
};

let score = [12, 55, 67, 94, 22];
console.log(firstElement(score)); // 12
```

The function above will require only one execution step, meaning the function is in constant time with time complexity O(1).

But as I said earlier, there are various ways to achieve a solution in programming. Another programmer might decide to first loop through the array before returning the first element:

```
const firstElement = (array) => {
  for (let i = 0; i < array.length; i++) {
    return array[0];
  }
};

let score = [12, 55, 67, 94, 22];
console.log(firstElement(score)); // 12
```

this is no longer constant time but now linear time with the time complexity O(n).

Linear Time: O(n)

You get linear time complexity when the running time of an algorithm increases linearly with the size of the input. This means that when a function has an iteration that iterates over an input size of n, it is said to have a time complexity of order O(n).

For example, if an algorithm is to return the factorial of any inputted number. This means if you input 5 then you are to loop through and multiply 1 by 2 by 3 by 4 and by 5 and then output 120:

```
const calcFactorial = (n) => {
  let factorial = 1;
  for (let i = 2; i <= n; i++) {
    factorial = factorial * i;
  }
  return factorial;
};

console.log(calcFactorial(5)); // 120
```

The fact that the runtime depends on the input size means that the time complexity is linear with the order O(n).

Logarithm Time: O(log n)

This is similar to linear time complexity, except that the runtime does not depend on the input size but rather on half the input size. When the input size decreases on each iteration or step, an algorithm is said to have logarithmic time complexity.

This method is the second best because your program runs for half the input size rather than the full size. After all, the input size decreases with each iteration.

A great example is binary search functions, which divide your sorted array based on the target value.

For example, suppose you use a binary search algorithm to find the index of a given element in an array:

```
const binarySearch = (array, target) => {
  let firstIndex = 0;
  let lastIndex = array.length - 1;
  while (firstIndex <= lastIndex) {
    let middleIndex = Math.floor((firstIndex + lastIndex) / 2);

    if (array[middleIndex] === target) {
      return middleIndex;
    }

    if (array[middleIndex] > target) {
      lastIndex = middleIndex - 1;
    } else {
      firstIndex = middleIndex + 1;
    }
  }
  return -1;
};

let score = [12, 22, 45, 67, 96];
```

In the code above, since it is a binary search, you first get the middle index of your array, compare it to the target value, and return the middle index if it is equal. Otherwise, you must check if the target value is greater or less than the middle value to adjust the first and last index, reducing the input size by half.

Because for every iteration the input size reduces by half, the time complexity is logarithmic with the order O(log n).

Quadratic Time: O(n^2)

When you perform nested iteration, meaning having a loop in a loop, the time complexity is quadratic, which is horrible.

A perfect way to explain this would be if you have an array with n items. The outer loop will run n times, and the inner loop will run n times for each iteration of the outer loop, which will give total n^2 prints. If the array has ten items, ten will print 100 times (10^2).

Here is an example by Jared Nielsen, where you compare each element in an array to output the index when two elements are similar:

```
const matchElements = (array) => {
  for (let i = 0; i < array.length; i++) {
    for (let j = 0; j < array.length; j++) {
      if (i !== j && array[i] === array[j]) {
        return `Match found at ${i} and ${j}`;
      }
    }
  }
  return "No matches found 😞";
};

const fruit = ["🍓", "🍐", "🍊", "🍌", "🍍", "🍑", "🍎", "🍈", "🍊", "🍇"];
console.log(matchElements(fruit)); // "Match found at 2 and 8"
```

In the example above, there is a nested loop, meaning that the time complexity is quadratic with the order O(n^2).

Exponential Time: O(2^n)

You get exponential time complexity when the growth rate doubles with each addition to the input (n), often iterating through all subsets of the input elements. Any time an input unit increases by 1, the number of operations executed is doubled.

The recursive Fibonacci sequence is a good example. Assume you're given a number and want to find the nth element of the Fibonacci sequence.

The Fibonacci sequence is a mathematical sequence in which each number is the sum of the two preceding numbers, where 0 and 1 are the first two numbers. The third number in the sequence is 1, the fourth is 2, the fifth is 3, and so on... (0, 1, 1, 2, 3, 5, 8, 13, ...).

This means that if you pass in 6, then the 6th element in the Fibonacci sequence would be 8:

```
const recursiveFibonacci = (n) => {
};

console.log(recursiveFibonacci(6)); // 8
```

In the code above, the algorithm specifies a growth rate that doubles every time the input data set is added. This means the time complexity is exponential with an order O(2^n).

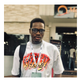
Wrapping Up

In this guide, you have learned what time complexity is all about, how performance is determined using the Big O notation, and the various time complexities that exists with examples.

You can learn more via freeCodeCamp's JavaScript Algorithms and Data Structures curriculum.

Happy learning!

You can access over 200 of my articles by visiting my website. You can also use the search field to see if I've written a specific article.



Joel Olawanle

Frontend Developer & Technical Writer

Forum Donate

Learn to code — free 3,000-hour curriculum

JS isEmpty Equivalent	Coalesce SQL
Submit a Form with JS	Python join()
Add to List in Python	JS POST Request
Grep Command in Linux	JS Type Checking
String to Int in Java	Read Python File
Add to Dict in Python	SOLID Principles
Java For Loop Example	Sort a List in Java
Matplotlib Figure Size	For Loops in Python
Database Normalization	JavaScript 2D Array
Nested Lists in Python	SQL CONVERT Function
Rename Column in Pandas	Create a File in Terminal
Delete a File in Python	Clear Formatting in Excel
K-Nearest Neighbors Algo	Accounting Num Format Excel
iferror Function in Excel	Check if File Exists Python
Remove From String Python	Iterate Over Dict in Python