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.

Big O defines the runtime required to execute an algorithm by identifying how the performance of your algorithm will change as the input size grows. But it does not tell you how 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. Similarly, an algorithm's space complexity specifies the total amount of space or memory required to execute an algorithm.

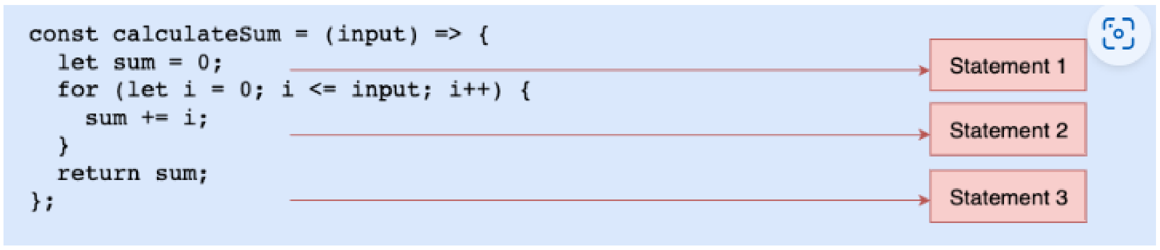
We will be focusing on time complexity in this guide. This will be an in-depth cheat sheet to help you understand how to calculate the time complexity for any algorithm.

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 function of the input size.**

In Big O, there are six major types of complexities (time and space):

- Constant: O(1)
- Linear time: O(n)
- Logarithmic time: O(n log n)
- Quadratic time: O(n^2)
- Exponential time: O(2^n)
- Factorial time: O(n!)

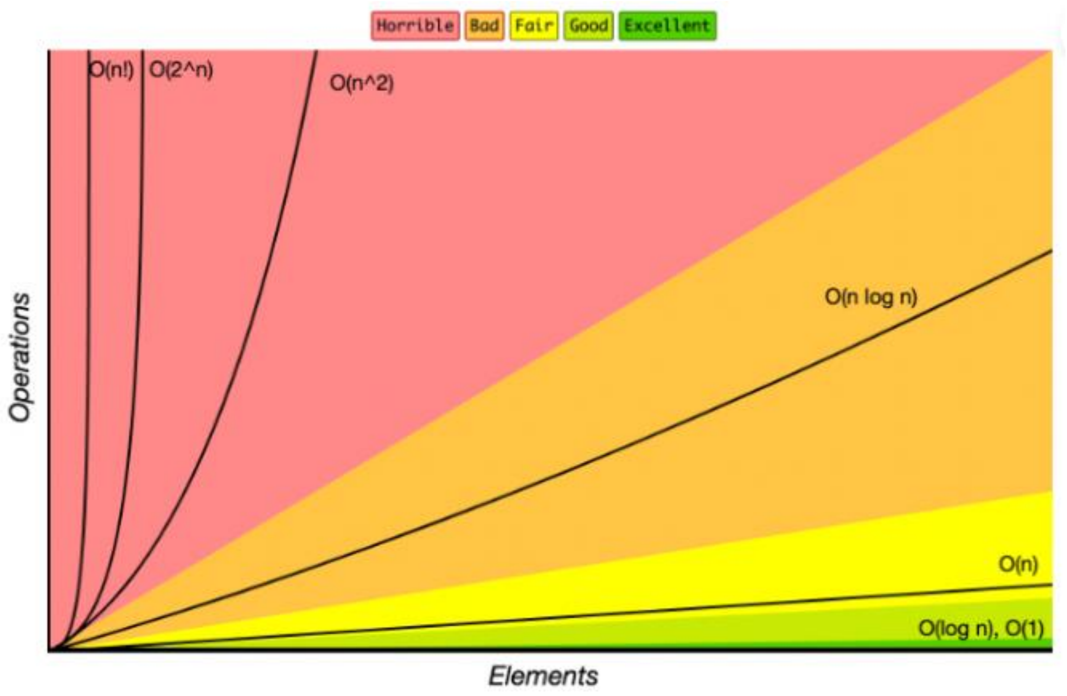
Before we look at examples for each time complexity, let's understand the Big O time complexity chart.

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 help 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
- **O(n log n) -** Bad
- **O(n^2), O(2^n)** and **O(n!) -** Horrible/Worst

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.

- 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 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 (O(2^n)).

Big O Time Complexity Examples

Constant Time: O(1)

When your algorithm is not dependent on the input size n, it is sard 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).

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 a 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 an inputted number. This means if you input 5 then you are to loop through and multiple 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).

Logarithmic 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];
console.log(binarySearch(score, 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 the middle value to adjust the first and last index, reducing the input size of 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).

Exponential Time: O(2^n)

Exponential time complexity is a way to describe how the running time of an algorithm grows very quickly as the size of the input data increases. In simple terms, it means that as you give the algorithm more data to process, the time it takes to complete the task increases dramatically.

Let's use an example to illustrate this concept:

Consider a simple algorithm that tries to find all possible combinations of set of items. Suppose you have a set of 10 items, you want to generate all possible combinations of those items. The algorithm might work like this:

1. Start with an empty combination.
2. For each item in the set, you have two choices: either include it in the combination or exclude it.
3. Recursively repeat step 2 for the remaining items.
4. Keep track of all the valid combinations.

Now, as you increase the number of items, the number of combinations grows exponentially. For example:

- With 1 item, you have 2 combinations (include or exclude)
- With 2 items, you have 4 combinations (2 choices for the first item, 2 for the second)
- With 3 items, you have 8 combinations.
- With 4 items, you have 16 combinations.

You can see that the number of combinations doubles each time you add an item. This is an example of an exponential algorithm. The running time grows very quickly as you increase the input size. If you had 20 items instead of 10, the number of combinations would be 2^20, which is over a million! The algorithm would take an extremely long time to complete.

In Big O notation, exponential time complexity is often represented as O(2^n), where 'n' is the size of the input. This indicates that the running time increases exponentially with the size of the problem, making it very inefficient for large inputs.