Big O Notation

(https://github.com/learn-co-curriculum/phase-1-algorithms-big-o) (https://github.com/learn-co-curriculum/phase-1-algorithms-big-o/issues/new)

Learning Goals

- Define Big O notation
- Calculate time and space complexity using Big O notation

Introduction

Big O notation is one of the most important concepts to understand when it comes to writing and analyzing the performance of algorithms. In any algorithm interview problem you're given, there's an extremely high chance that your interviewer will ask you to identify the time complexity (and possibly the space complexity) of your solution using Big O notation. In this lesson, we'll discuss what Big O notation is and how to determine the Big O of any given algorithm with a few simple steps.

Defining Big O

Big O can sound like an intimidating term, but it's incredibly useful — and it's very likely that you already have an intuitive sense of how it works! Let's start with a definition:

> In computer science, **Big O notation** is used to classify algorithms according to how their run **time** or **space** requirements grow as the input size grows. — **Wikipedia** (https://en.wikipedia.org/wiki/Big_O_notation)

What does that mean? We'll use a real-world example. Imagine you're looking for a sock in a pile of unfolded laundry. An algorithm for finding the sock might look like this:

```
take one item from the pile
check if it's the sock we're looking for
```

```
if it is
  put on the sock
otherwise, keep looking
```

How much time will it take us to find the sock? Well, that depends on a couple factors.

One factor to consider is how quickly we're able to pull out socks from the pile and check what they look like. That variable changes from person to person, so it's not a factor of our *algorithm's* efficiency, and not something we can account for in Big O notation.

The other factor to consider is the size of our input, or in this case, how big is the pile of laundry? If it's just one item, we'll find the sock right away! If it's 1000 items, the **worst case** scenario is that the sock we're looking for is the last one we'll pull out of the pile. So, for our sock finding algorithm, **as the size of the input increases, the time requirement increases linearly** (the bigger the pile, the longer it takes).

Let's translate this analogy to code:

```
function findSock(laundry) {
  for (const item of laundry) {
    if (item === "sock") return item;
  }
}

findSock(["shirt", "shorts", "sock", "pants"]);
```

Let's think about the time complexity of this algorithm (how long it will take to run) step by step:

- The `for...of` block will iterate over every element in the `laundry` array.
- On the first iteration, the first element, `"shirt"` is assigned to the variable `item`.
- The `if` statement checks: is the `item` a `"sock"`? Nope! We continue iterating.
- On the second iteration, the second element, `"shorts"` is assigned to the variable `item`.
- The `if` statement checks: is the `item` a `"sock"`? Nope! We continue iterating.
- On the third iteration, the third element, `"sock"` is assigned to the variable `item`.
- The `if` statement checks: is the `item` a `"sock"`? Yep! We stop iterating and return from the function.

What happens if our sock is the last element of the array? Or what if the sock isn't in the array at all?

If `laundry` is an unsorted array of strings, the **worst case** scenario for our algorithm is that it will have to iterate over every item in the array. In regards to Big O, whether it might find the element on the first try, or on the third try, or sometimes not at all is not important; developers simply care about describing the algorithm from the highest, most abstract level.

Besides the size of the input, the other factor that will determine how long our algorithm will take to find a sock is how powerful the computer is that is running our code. Once again, that variable changes from computer to computer, so it's not a factor of our algorithm's efficiency.

Therefore, we can summarize the time complexity of our function like this:

Given an input array of `n` elements, the worst case scenario is that the algorithm needs to make `n` iterations.

In Big O notation, we can express this as `O(n)`, which is also known as **linear time**.

Identifying Time Complexity With Big O

When calculating the time complexity of an algorithm using Big O notation, we're looking for a high-level summary of the algorithm's performance. Let's look at another example of an algorithm. This one calculates the average of an array of numbers:

```
function average(array) {
  let total = 0;
  for (const num of array) {
    total += num;
  }
  return total / array.length;
}
```

To determine the time complexity of any algorithm, all we need to do is:

- Count the number of steps the computer will take to run our code
- Remove any constants (more on this shortly!)

Let's start by counting the steps:

```
function average(array) {
  // 1 step
  let total = 0;

  for (const num of array) {
    // n steps (dependent on the size of the array)
    total += num;
  }

  // 1 step
  return total / array.length;
}
```

This gives us a runtime of `O(n + 2)`. We can remove the constants (`+ 2`) because we're just looking for a high-level summary, so we end up with `O(n)`.

What if we use the `reduce` method instead of a `for...of` loop?

```
function average(array) {
  return array.reduce((total, num) => total + num, 0) / array.length;
}
```

Under the hood, the `reduce` method still needs to iterate over every element in the array, so we still end up with the same runtime: `O(n)`. It's important to keep this in mind for built-in methods: just because we might end up with fewer lines of code doesn't mean our runtime improves.

Let's try another example. Here's another algorithm for determining if an array contains the average of all its values:

```
function containsAverage(array) {
  const averageValue = average(array);
  for (const num of array) {
    if (num === averageValue) return true;
```

```
  }
}
```

Let's count the steps once more:

```
function containsAverage(array) {
  // n steps (depending on the size of the array)
  const averageValue = average(array);

  for (const num of array) {
    // n steps (depending on the size of the array)
    if (num === averageValue) return true;
  }
}
```

Counting the steps here, we end up with `O(n * 2)`. Even though it takes twice as long as our first algorithm, we can still remove the constants and end up with `O(n)`. Since we're looking for a high-level summary, this approximation still gives us a good sense as to how well our algorithm will scale to larger and larger inputs when compared to some other common Big O runtimes.

Space Complexity With Big O

We've spent some time discussing how to calculate time complexity (how long our function will take to run) using Big O notation, but there's another consideration: space complexity (how much memory our function will use).

Just like with time complexity, we measure space complexity *relative to the size of the input*: for an input that takes up `n` memory, how much memory will our algorithm use? In addition to the amount of memory needed for our inputs, we need to consider what auxiliary (additional) space is taken up for new data structures when our algorithm runs.

Let's break down a couple algorithms to see some examples:

```
function reverseString(word) {
  const wordArray = word.split("");
  const reversedWordArray = wordArray.reverse();
```

```
  const reversedWord = reversedWordArray.join("");
  return reversedWord;
}
```

In this case, we need to allocate memory for the input `word`, as well as these auxiliary pieces of data:

- an array `wordArray` that will grow in size linearly with the input
- an array `reversedWordArray` that will grow in size linearly with the input
- a string `reversedWord` that will grow in size linearly with the input

Putting that together, we end up with `O(4n)` space complexity: for an input string of length `n`, we also need to declare two arrays of length `n` and another string of length `n`. We can simplify as `O(n)` after dropping the constant.

In general, space complexity can be a bit more difficult to calculate than time complexity because it can vary between different programming languages and environments, depending on how they allocate memory under the hood; so again, keeping a high-level summary is more important than trying to be too precise.

For a function that simply adds two numbers together:

```
function sum(num1, num2) {
  return num1 + num2;
}
```

We only allocate memory for the inputs and the return value:

- A number `num1`
- A number `num2`
- A number (the return value)

In this case, we can summarize the space complexity as constant: `O(1)`.

In many cases, you can improve an algorithm's space complexity by sacrificing its time complexity, and vice-versa; navigating these tradeoffs is one of the keys to being a good developer, and it comes with practice.

Conclusion

To recap:

- **Big O notation** is used to classify algorithms according to how their run time or space requirements grow as the input size grows.
- To calculate the time complexity of an algorithm using Big O notation, count the number of steps the computer will take to run our code and then remove any constants (so `O(2n + 1)` becomes just `O(n)`).
- To calculate the space complexity of an algorithm using Big O notation, check what memory is needed for the inputs as well as any auxiliary data structures (and also remove any constants).

In the next lesson, we'll look at some other examples of algorithms that use other common Big O formulas.

Resources

- **Big O Cheat Sheet (https://www.bigocheatsheet.com/)**