Big O Notation

Big-O notation is the language we use for talking about how long an algorithm takes to run (time complexity) or how much memory is used by an algorithm (space complexity). Big-O notation can express the best, worst, and average-case running time of an algorithm. For our purposes here, we are going to focus primarily on Big-O as it relates to time complexity.

As a software engineer, you'll find that most discussions of Big-O focus on the "upper-bound" running time of an algorithm, which is often termed the worst-case. An important piece to note is the "running time" when using Big-O notation does not directly equate to time as we know it (e.g., seconds, milliseconds, microseconds, etc.). Analysis of running times does not take certain things into account, such as the processor, the language, or the run time environment. Instead, we can think of time as the number of operations or steps it takes to complete a problem of size n. In other words, Big-O notation is a way to track how quickly the runtime grows relative to the size of the input.

When we are thinking about the worst-case, the question becomes — what are the most operations/steps that could happen for an input of size n?

O(1) → Constant Time

O(1) means that it takes a constant time to run an algorithm, regardless of the size of the input.

Bookmarks are a great example of how constant time would play out in the “real world.” Bookmarks allow a reader to find the last page that you read in a quick, efficient manner. It doesn’t matter if you are reading a book that has 30 pages or a book that has 1000 pages. As long as you’re using a bookmark, you will find that last page in a single step.

In programming, a lot of operations are constant. Here are some examples:

- math operations
- accessing an array via the index
- accessing a hash via the key
- pushing and popping on a stack
- insertion and removal from a queue
- returning a value from a function

Take a look at **findFirstIndex,** listed below. Passing **smallCollection** or **giganticCollection** will produce the same runtime of O(1) when accessing the 0 index. The return of **firstIndex** is also a constant time operation. Regardless of the size of n, both of these operations will take a constant amount of time.

```
const smallCollection = [1, 2, 3, 4];
const giganticCollection = [1, 2, 3, ..., 1000000000];

function findFirstIndex(n) {
  const firstIndex = n[0];

  return firstIndex;
}
```

O(n) → Linear Time

O(n) means that the run time increases at the same pace as the input.

The act of reading a book is an example of how linear time would play out in the "real world." Let's assume that it takes me exactly 1 minute to read a single page of a large print book. Given that, a book that has 30 pages will take me 30 minutes to read. Likewise, a book that has 1000 pages = 1000 minutes of reading time. Now, I don't force myself to finish books that aren't great — so there is always a chance that I won't get through that 1,000-page book. However, before I start reading it, I can know that my worst-case reading time is 1000 minutes for a 1000 page book.

In programming, one of the most common linear-time operations is traversing an array. In JavaScript, methods like **forEach**, **map**, and **reduce** run through the entire collection of data, from start to finish.

Take a look at our **printAllValues** function below. The number of operations that it will take to loop through n is directly related to the size of n. Generally speaking (but not always), seeing a loop is a good indicator that the particular chunk of code you're examining has a run time of O(n).

```
const smallCollection = [1, 2, 3, 4];
const giganticCollection = [1, 2, 3, ..., 1000000000];

function printAllValues(n) {
  for (let i = 0; i < n.length; i++) {
    console.log(n[i]);
  }
}
```

But what about the method **find**? Since it doesn't always run through the entire collection, is it actually linear? In the example below, the first value that is less than 3 is at index 0 of the collection. Why wouldn't this be constant time?

```
const numbers = [2, 3, 4, 6, 3, 6, 7];

const lessThanThree = numbers.find(number => number < 3);
```

Remember — since we are looking for the worst-case scenario, we must assume that the input will not be ideal and that the element or

value that we seek could be the last value in the array. In the second scenario (below), you'll see just that. Finding the number that is less than three, in the least ideal situation, would require iterating through the entire array.

```
const numbers = [3, 4, 6, 3, 6, 7, 2];

const lessThanThree = numbers.find(number => number < 3);
```

O(n²) → Quadratic Time

O(n²) means that the calculation runs in quadratic time, which is the squared size of the input data.

In programming, many of the more basic sorting algorithms have a worst-case run time of O(n²):

- Bubble Sort
- Insertion Sort
- Selection Sort

Let's look at **countOperations** below. Here we have two nested loops that are incrementing the **operations** variable after each iteration. If n is our **smallCollection**, we will end up with a count of 16 operations. Not horrible. But what about if n is our **gigantic collection**? A billion times a billion is a quintillion — or 1,000,000,000,000,000,000. Yikes.

😬 That's a LOT of operations. Even an array with as little as 1,000 elements would end up creating a million operations.

```
const smallCollection = [1, 2, 3, 4];
const giganticCollection = [1, 2, 3, ..., 1000000000];

function countOperations(n){
  let operations = 0;

  for (let i = 0; i < n; i++) {
    for (let j = 0; j < n; j++) {
      operations++;
    }
  }

  return operations;
}
```

Generally speaking (but not always), seeing two nested loops is typically a good indicator that the piece of code you're looking at has a run time of $O(n^2)$. Likewise — three nested loops would indicate a run time of $O(n^3)$.

O(log n) → Logarithmic Time

O(log n) means that the running time grows in proportion to the logarithm of the input size, meaning that the run time barely increases as you exponentially increase the input.

Finding a word in a physical dictionary by halving my sample size is an excellent example of how logarithmic time works in the “real world.” For instance, when looking for the word “senior,” I could open the dictionary precisely in the middle — at which point I could determine whether the words that begin with “s” come before or after the words that I’m currently viewing. Once I determine that “s” is in the second half of the book, I can dismiss all of the pages in the first half. I then repeat the same process. By following this algorithm to the end, I would cut the number of pages I must search through in 1/2 every time until I find the word.

In programming, this act of searching through a physical dictionary is an example of a binary search operation — which is the most typical example used when discussing logarithmic run times.

Let’s take a look at a modified version of our **countOperations** function. Note that n is now a number: n could be an input (number) or the size of an input (the length of an array).

```
const smallNumber = 1000;
const biggerNumber = 10000;

function countOperations(n) {
  let operations = 0;
  let i = 1;

  while (i < n) {
    i = i * 2;
    operations++;
  }

  return operations;
}
```

In our example above, we end up with 11 operations if n = 2000. If n = 4,000, we end up with 12 operations. Every time that we **double** the amount of n, the amount of operations only increases by 1. Algorithms that run in logarithmic time have big implications when it comes to larger inputs. Using our example below, O(log(7)) would return three operations. An O(log(1000000)) would return only 20 operations. 😍

Note: O(n log n), which is often confused with O(log n), means that the running time of an algorithm is linearithmic, which is a combination of linear and logarithmic complexity. Sorting algorithms that utilize a divide and conquer strategy are linearithmic, such as the following:

* merge sort

* timsort

* heapsort

When looking at time complexity, O(n log n) lands between O(n2) and O(n).