Introduction to big O

> If you already understand big O, you can skip this article.

Before we talk about big O, it's important that we first understand what exactly an "algorithm" is, especially in the context of LeetCode.

An algorithm can be seen as a recipe for a computer to follow. It's a set of instructions that a computer will follow step-by-step to solve a problem.

Algorithms take an input and produce an output. The output will be the answer to a question regarding the input. For example, let's say you had a non-empty array of positive integers called `nums`, and you wanted to answer the question: "What is the largest number in `nums`?".

To answer this question, you would write an algorithm that takes an array called `nums` as **input**, and **outputs** the largest number in `nums`. Here is an example of such an algorithm:

1. Create a variable `maxNum` and initialize it to `0`.
2. Iterate over each element `num` in `nums`.
3. If `num` is greater than `maxNum`, update `maxNum = num`.
4. Output `maxNum`.

Here, we have written down a set of instructions that when followed, will solve the problem. We can now implement these instructions in code so that a computer can quickly solve the problem. There are some important requirements for algorithms in the context of LeetCode:

- Algorithms should be **deterministic**. Given the same **input**, the algorithm should **always** produce the same **output**. Basically, there shouldn't be any randomness.
- The algorithm should be correct for any arbitrary valid **input**. In our example, we said that `nums` is a non-empty array of positive integers. There are infinitely many of such arrays, and our algorithm works for **all** of them. Note that if `nums` had negative numbers, the input would be invalid since we stated the integers are positive. In fact, our algorithm would actually break because we initialized `maxNum` to `0`, so if all of `nums` was negative, we would incorrectly output `0`.

Big O

Big O is a notation used to describe the computational complexity of an algorithm. The computational complexity of an algorithm is split into two parts: time complexity and space complexity. The time complexity of an algorithm is the amount of time the algorithm needs to run relative to the input size. The space complexity of an algorithm is the amount of memory allocated by the algorithm when run relative to the input size.

> Typically, people care about the time complexity more than the space complexity, but both are important to know.

Time complexity: as the input size grows, how much longer does the algorithm take to complete?

Space complexity: as the input size grows, how much more memory does the algorithm use?

How complexity works

Complexity is described by a function (math formula). What should the arguments to this function be?

The arguments are variables defined by the programmer, but they should represent values that change between different inputs, and these values should affect the algorithm. For example, the most common variable you'll see is n, which usually denotes the length of an input array or string. In the example above, we could say that n is equal to the length of `nums`.

Here, "the length of `nums`" is a value that changes between inputs, and it directly affects the algorithm. The longer `nums` is, the more elements we need to iterate through, and thus the longer our algorithm will take to complete.

> Note that choosing the letter n to denote this value is arbitrary. There is no requirement that we use n, it's just that n is the commonly accepted standard that everyone uses. If you wanted, you could use a banana (🍌) to represent the length of `nums`. The programmer is the one who decides which variables represent what values.

In the context of LeetCode, there are some common assumptions that we make. When dealing with integers, the larger the integer, the more time operations like addition, multiplication, or printing will take. While this **is** relevant in theory, we typically ignore this fact because the difference is practically very small, and treat all integers the same. If you are given an array of integers as an input, the only variable you would use is n to denote the length of the array. Technically, you *could* introduce another variable, let's say k which denotes the average value of the integers in the array. However, nobody does this.

When written, the function is wrapped by a capital O. Here are some example complexities:

- $O(n)$
- $O(n^2)$
- $O(2^n)$
- $O(\log n)$
- $O(n \cdot m)$

> You might be thinking, what is m? Remember: we define the variables. As these are simply examples with no associated problem, m could denote any arbitrary variable. For example, we could have a problem where the input is two arrays. n could denote the length of one while m denotes the length of the other.

These functions represent the complexity. For example, you would say "The time complexity of my algorithm is O(n)" or "The space complexity of my algorithm is O(n^2)".

Calculating complexity

Roughly, your function calculates the number of operations or amount of memory (depending on if you're analyzing time or space complexity, respectively) your algorithm consumes relative to the input size. Using the example from above (find the largest number in `nums`), we have a time complexity of $O(n)$. The algorithm involves iterating over each element in `nums`, so if we define n as the length of `nums`, our algorithm uses approximately n steps. If we pass an array with a length of `10`, it will perform approximately `10` steps. If we pass an array with a length of `10,000,000,000`, it will perform approximately `10,000,000,000` steps.

> Time complexity is not meant to be an **exact** representation of the number of operations. For example, we needed to initialize `maxNum = 0` and we also needed to output `maxNum` at the end. Thus, you could argue that for an array of length `10`, we need `12` operations. This **is not the point** of time complexity. The point of time complexity is to describe how the number of operations changes as the input changes. The number of iterations we do depends on `nums`, but initializing `maxNum = 0` doesn't.

Being able to analyze an algorithm and calculate its time and space complexity is a crucial skill. Interviewers will **almost always** ask you for your algorithm's complexity to check that you actually understand your algorithm and didn't just memorize/copy the code. Being able to analyze an algorithm also enables you to determine what parts of it can be improved.

Rules

There are a few rules when it comes to calculating complexity. First, **we ignore constants**. That means $O(9999999n) = O(8n) = O(n) = O(\frac{n}{500})$. Why do we do this? Imagine you had two algorithms. Algorithm A uses approximately n operations and algorithm B uses approximately $5n$ operations.

When $n = 100$, algorithm A uses 100 operations and algorithm B uses 500 operations. What happens if we double n? Then algorithm A uses 200 operations and algorithm B uses 1000 operations. As you can see, when we double the value of n, both algorithms require double the amount of operations. If we were to 10x the value of n, then both algorithms would require 10x more operations.

Remember: the point of complexity is to analyze the algorithm **as the input changes**. We don't care that algorithm B is 5x slower than algorithm A. For both algorithms, as the input size increases, the number of operations required increases **linearly**. That's what we care about. Thus, both algorithms are $O(n)$.

The second rule is that we consider the complexity as the variables **tend to infinity**. When we have addition/subtraction between terms of the **same** variable, we ignore all terms except the most powerful one. For example, $O(2^n + n^2 - 500n) = O(2^n)$. Why? Because as n tends to infinity, 2^n becomes so large that the other two terms are effectively zero in comparison.

Let's say that we had an algorithm that required $n + 500$ operations. It has a time complexity of $O(n)$. When n is small, let's say $n = 5$, the $+500$ term is very significant - but we don't care about that. We need to perform the analysis as if n is tending toward infinity, and in that scenario, the 500 is nothing.

> The best complexity possible is $O(1)$, called "constant time" or "constant space". It means that the algorithm ALWAYS uses the same amount of resources, regardless of the input.
>
> Note that a constant time complexity doesn't necessarily mean that an algorithm is fast ($O(5000000) = O(1)$), it just means that its runtime is independent of the input size.

When talking about complexity, there are normally three cases:

- Best case scenario
- Average case
- Worst case scenario

In most algorithms, all three of these will be equal, but some algorithms will have them differ. If you have to choose only one to represent the algorithm's time or space complexity, never choose the best case scenario. It is most correct to use the worst case scenario, but you should be able to talk about the difference between the cases.

Analyzing time complexity

Let's look at some example algorithms in pseudo-code and talk about their time complexities.

```
// Given an integer array "arr" with length n,

for (int num: arr) {
    print(num)
}
```

This algorithm has a time complexity of $O(n)$. In each for loop iteration, we are performing a print, which costs $O(1)$. The for loop iterates n times, which gives a time complexity of $O(1 \cdot n) = O(n)$.

```
// Given an integer array "arr" with length n,

for (int num: arr) {
    for (int i = 0; i < 500,000; i++) {
        print(num)
    }
}
```

This algorithm has a time complexity of $O(n)$. In each inner for loop iteration, we are performing a print, which costs $O(1)$. This for loop iterates 500,000 times, which means each outer for loop iteration costs $O(500000) = O(1)$. The outer for loop iterates n times, which gives a time complexity of $O(n)$.

Even though the first two algorithms *technically* have the same time complexity, in reality the second algorithm is **much** slower than the first one. It's correct to say that the time complexity is $O(n)$, but it's important to be able to discuss the differences between practicality and theory.

```
// Given an integer array "arr" with length n,

for (int num: arr) {
    for (int num2: arr) {
        print(num * num2)
    }
}
```

This algorithm has a time complexity of $O(n^2)$. In each inner for loop iteration, we are performing a multiplication and print, which both cost $O(1)$. The inner for loop runs n times, which means each outer for loop iteration costs $O(n)$. The outer for loop runs $O(n)$ times, which gives a time complexity of $O(n \cdot n) = O(n^2)$.

```
// Given integer arrays "arr" with length n and "arr2" with length m,

for (int num: arr) {
    print(num)
}

for (int num: arr) {
    print(num)
}

for (int num: arr2) {
    print(num)
}
```

This algorithm has a time complexity of $O(n+m)$. The first two for loops both cost $O(n)$, whereas the final for loop costs $O(m)$. This gives a time complexity of $O(2n+m) = O(n+m)$.

```
// Given an integer array "arr" with length n,

for (int i = 0; i < arr.length; i++) {
    for (int j = i; j < arr.length; j++) {
        print(arr[i] + arr[j])
    }
}
```

This algorithm has a time complexity of $O(n^2)$. The inner for loop is dependent on what iteration the outer for loop is currently on. The first time the inner for loop is run, it runs n times. The second time, it runs $n - 1$ times, then $n-2$, $n-3$, and so on.

That means the total iterations is 1 + 2 + 3 + 4 + ... + n, which is the partial sum of this series, which is equal to $\frac{n \cdot (n+1)}{2}$ = $\frac{n^2+n}{2}$. In big O, this is $O(n^2)$ because the addition term in the numerator and the constant term in the denominator are both ignored.

Logarithmic time

A logarithm is the inverse operation to exponents. The time complexity $O(\log n)$ is called logarithmic time and is **extremely** fast. A common time complexity is $O(n \cdot \log n)$, which is reasonably fast for most problems and also the time complexity of efficient sorting algorithms.

Typically, the base of the logarithm will be `2`. This means that if your input is size `n`, then the algorithm will perform `x` operations, where $2^x = n$. However, the base of the logarithm doesn't actually matter for big O, since all logarithms are related by a constant factor.

$O(\log n)$ means that somewhere in your algorithm, the input is being reduced by a percentage at every step. A good example of this is binary search, which is a searching algorithm that runs in $O(\log n)$ time (there is a chapter dedicated to binary search later on). With binary search, we initially consider the entire input (`n` elements). After the first step, we only consider `n / 2` elements. After the second step, we only consider `n / 4` elements, and so on. At each step, we are reducing our search space by 50%, which gives us a logarithmic time complexity.

Analyzing space complexity

When you initialize variables like arrays or strings, your algorithm is allocating memory. We never count the space used by the input (it is bad practice to modify the input), and usually don't count the space used by the output (the answer) unless an interviewer asks us to.

> In the below examples, the code is only allocating memory so that we can analyze the space complexity, so we will consider everything we allocate as part of the space complexity (there is no "answer").

```
// Given an integer array "arr" with length n

for (int num: arr) {
    print(num)
}
```

This algorithm has a space complexity of $O(1)$. The only space allocated is an integer variable `num`, which is constant relative to n.

```
// Given an integer array "arr" with length n

Array doubledNums = int[]

for (int num: arr) {
    doubledNums.add(num * 2)
}
```

This algorithm has a space complexity of $O(n)$. The array `doubledNums` stores n integers at the end of the algorithm.

```
// Given an integer array "arr" with length n

Array nums = int[]
int oneHundredth = n / 100

for (int i = 0; i < oneHundredth; i++) {
    nums.add(arr[i])
}
```

This algorithm has a space complexity of $O(n)$. The array `nums` stores the first 1% of numbers in `arr`. This gives a space complexity of $O(\frac{n}{100}) = O(n)$.

```
// Given integer arrays "arr" with length n and "arr2" with length m,

Array grid = int[n][m]

for (int i = 0; i < arr.length; i++) {
    for (int j = 0; j < arr2.length; j++) {
        grid[i][j] = arr[i] * arr2[j]
    }
}
```

This algorithm has a space complexity of $O(n \cdot m)$. We are creating a `grid` that has dimensions $n \cdot m$.

> In this course, we will talk extensively about time and space complexity. If it's a new concept to you, don't worry - with practice, you will become more and more comfortable with analyzing algorithms on your own.