**Introduction to 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 having a better time complexity more than having a better space complexity.

Complexity is described by a function of variables that can change with the input. The most common variable is n*n*, which usually describes the length of an input array or string. This function is wrapped by a capital O. Here are some example complexities:

* O(n)*O*(*n*)
* O(n^2)*O*(*n*2)
* O(2^n)*O*(2*n*)
* O(\log{}n)*O*(log*n*)
* O(n \cdot m)*O*(*n*⋅*m*)

These functions describe how the amount of operations/memory needed by the algorithm grows as the arguments **tend to infinity**. Because the variables are tending to infinity, **constants are always ignored**. That means that O(9999999n) = O(8n) = O(n) = O(\frac{n}{500})*O*(9999999*n*)=*O*(8*n*)=*O*(*n*)=*O*(500*n*​).

The point of computational complexity is to analyze how an algorithm scales with the input. Let's say you have two algorithms. One runs in O(100n)*O*(100*n*) and the other runs in O(5n)*O*(5*n*). Even though the second algorithm runs 20x faster, both algorithm's operations grow linearly with n*n*. If you double n*n*, then both algorithms will take 2x longer regardless of what the coefficients are.

This also means that when you have addition/subtraction between terms of the **same** variable, some of the terms can be ignored. For example, O(2^n + n^2 - 500n) = O(2^n)*O*(2*n*+*n*2−500*n*)=*O*(2*n*), because as n*n* tends to infinity, 2^n2*n* completely dominates the other two terms.

Being able to analyze an algorithm and derive 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.

The best complexity possible is O(1)*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)*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 cases, 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 array "arr" with length n,

for (int num: arr) {

print(num)

}

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

Given an 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)*O*(*n*). In each inner for loop iteration, we are performing a print, which costs O(1)*O*(1). This for loop iterates 500,000 times, which means each outer for loop iteration costs O(500000) = O(1)*O*(500000)=*O*(1). The outer for loop iterates n*n* times, which gives a time complexity of O(n)*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)*O*(*n*), but it's important to be able to discuss the differences between practicality and theory.

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

Sometimes, work is done by an algorithm that isn't included in the time complexity. For example, here we said that multiplication costs O(1)*O*(1). Multiplication takes longer on larger numbers, but n*n* has nothing to do with the size of the numbers in the array. We could introduce another variable k*k* that is the largest number in the array, but this is unnecessary for integers in algorithm problems.

Given an array "arr" with length n,

Given an array "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)*O*(*n*+*m*). The first two for loops both cost O(n)*O*(*n*), whereas the final for loop costs O(m)*O*(*m*). This gives a time complexity of O(2n + m)*O*(2*n*+*m*) = O(n + m)*O*(*n*+*m*).

Given an 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)*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*n* times. The second time, it runs n - 1*n*−1 times, then n - 2*n*−2, n - 3*n*−3, and so on.

That means the total iterations is 1 + 2 + 3 + 4 + ... + n, which is the partial sum of [this series](https://en.wikipedia.org/wiki/1_%2B_2_%2B_3_%2B_4_%2B_%E2%8B%AF#Partial_sums), which is equal to \frac{n \cdot (n + 1)}{2}2*n*⋅(*n*+1)​ = \frac{n^2 + n}{2}2*n*2+*n*​. In big O, this is O(n^2)*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)*O*(log*n*) is called logarithmic time and is **extremely** fast. A common time complexity is O(n \cdot \log{}n)*O*(*n*⋅log*n*), which is 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 = n2*x*=*n*. However, the base of the logarithm [doesn't actually matter](https://stackoverflow.com/questions/1569702/is-big-ologn-log-base-e/1569710#1569710) for big O, since all logarithms are related by a constant factor.

O(\log{}n)*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})*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 array "arr" with length n,

for (int num: arr) {

print(num)

}

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

Given an 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)*O*(*n*). The array doubledNums stores n*n* integers at the end of the algorithm.

Given an 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)*O*(*n*). The array nums stores the first 1% of numbers in arr. This gives a space complexity of O(\frac{n}{100})*O*(100*n*​) = O(n)*O*(*n*).

Given an array "arr" with length n,

Given an array "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)*O*(*n*⋅*m*). We are creating a grid that has dimensions n \cdot m*n*⋅*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 time, you will become more and more comfortable with analyzing algorithms on your own.