# How To Analyze Algorithms?

Analyzing an algorithm depends upon various factors, but there are two factors that are particularly important.

* **Time Complexity** - The amount of time taken by an algorithm to run with respect to the size of the input.
* **Space Complexity** - The amount of space that the algorithm takes to run with respect to the size of the input.

# Introduction

Time complexity is the amount of time an algorithm takes to run (with respect to the size of the input). It is the most important factor to consider while analyzing an algorithm.

**But how do we calculate the time complexity of an algorithm?**

We could calculate the time complexity of an algorithm based on the execution time. For example, if a code takes **16** minutes to run, its time complexity is **16** minutes.

However, this approach would be wrong. It's because the execution time will be different on different

* devices
* programming languages
* networks
* and so on

That's why we calculate the time complexity based on the number of steps required for a given input. For example,

If the time complexity is O(n)

* it takes **5** steps for an input of **5**
* it takes **10** steps for an input of **10**

We will cover what O(n) means in detail later in this course.

For now, just remember that the time complexity is calculated based on the number of steps required (regardless of the device or programming language being used).

# Analysis of Algorithms

There are three ways to analyze an algorithm:

* Best Case - The scenario where the algorithm performs at its best.
* Average Case - The scenario where the algorithm performs on average.
* Worst Case - The scenario where the algorithm performs at its worst.

Consider the following list/array:

numbers = [9, 11, 12, 18, 15]

**Best case**

Suppose we have to search **9** from the numbers list.

Then, we would find **9** in the first search. If the size of the list increases, we would still be able to search **9** in our first search. This is the **best-case** analysis.

**Worst case**

Suppose we have to search **19** from the numbers list.

Then, it would take us **5** steps to search **19**. It's because the list doesn't contain **19**, and we have to search until the end of the list to check if **19** is present in the list or not.

This is the **worst-case** analysis because we have to go through every element present in the list.

**Average case**

We might need to search for an element that is somewhere in between the first and last element.

If the element is present in the first position, it would take **1** search. If the element is present in the third position, it would take **3** searches.

Similarly, if the element is present in the **nth** index, it would take **n** searches. This is the **average case** analysis.

Average Case=nsum of all possible cases / n​

Therefore, the average case analysis to search for an element in the numbers list is:

Avarage Case=1+2+3+4+5 / 5​

# Frequency Count Method

We use the frequency count method to calculate the time complexity of an algorithm. The method is based on the number of steps our program takes to execute (like we did in our earlier example).

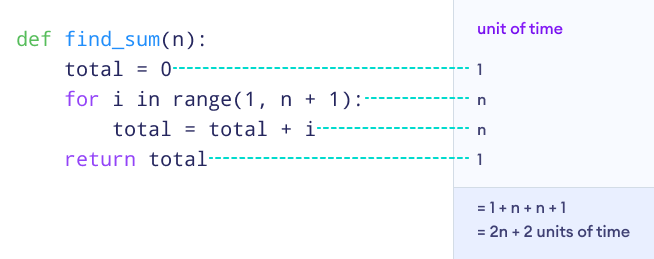
Here's the same code to find the sum of first n natural numbers.

|  |  |
| --- | --- |
|  | *def* *find\_sum*(n): |
|  | total = 0 |
|  |  |
|  | *for* i *in* *range*(1, n + 1): |
|  | total = total + i |
|  |  |
|  | *return* total |
|  |  |
|  | result = find\_sum(10) |
|  | *print*(result) |

Run Code  >>

Now let's calculate the time complexity of the find\_sum() function using the frequency count method.

In this method, we assign **1** unit of time to each statement. For example,

Figure: Frequency count method

If we write the number of steps in terms of a polynomial function, it would be:

f(n) = 2n + 2

This means the above program takes 2n + 2 steps to execute.

Now that we know how to calculate the number of steps required to run a program, we will learn about the degree of a polynomial function next.

Understanding the degree of a polynomial will help us to find the time complexity of a given function, f(n).

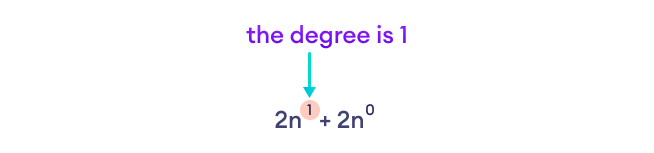
# Degree of Polynomial

The degree of a polynomial is the largest exponent of an equation. For example,

f(n) = 2n + 2

Here, this polynomial has two terms: **2n** and **2**.

And its degree is:

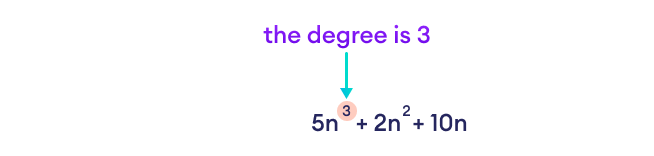
Figure: Degree of a Polynomial

**Example**

Let's take another example.

�(�)=5�3+2�2+10�f(n)=5n3+2n2+10n

Its degree is:

Figure: Degree of a Polynomial

Next, we will learn to calculate the time complexity of an equation.

# Calculating Time Complexity

The time complexity of a function is the term with the highest degree without the coefficient (a constant that multiplies the variable). For example,

The function of the steps required to complete the sum of the first n natural numbers was:

f(n) = 2n + 2

Here, the term with the highest degree is **2n**. Since we don't include the coefficient in time complexity, its time complexity is n.

This means our program takes **n** steps to complete.

**Example**

Let's take another example.

f(n)=5n^3 + 2n^2 + 10n

Here, the degree of the polynomial is **3**, and the term with the highest degree is 5n^3. Hence, its time complexity is 3n^3.

This means a program with 3n^3 time complexity takes 3n^3 steps to complete.