# Time Complexity and Space Complexity

Generally, there is always more than one way to solve a problem in computer science with different algorithms. Therefore, it is highly required to use a method to compare the solutions in order to judge which one is more optimal. The method must be:

* Independent of the machine and its configuration, on which the algorithm is running on.
* Shows a direct correlation with the number of inputs.
* Can distinguish two algorithms clearly without ambiguity.

There are two such methods used, [**time complexity**](https://www.geeksforgeeks.org/understanding-time-complexity-simple-examples/) and [**space complexity**](https://www.geeksforgeeks.org/g-fact-86/) which are discussed below:

**Time Complexity:**The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input. Note that the time to run is a function of the length of the input and not the actual execution time of the machine on which the algorithm is running on.

In order to calculate time complexity on an algorithm, it is assumed that a **constant time c** is taken to execute one operation, and then the total operations for an input length on **N** are calculated. Consider an example to understand the process of calculation: Suppose a problem is to[find whether a pair **(X, Y)** exists in an array, A of **N** elements whose sum is **Z**](https://www.geeksforgeeks.org/given-an-array-a-and-a-number-x-check-for-pair-in-a-with-sum-as-x/). The simplest idea is to consider every pair and check if it satisfies the given condition or not.

The pseudo-code is as follows:

int a[n];

for(int i = 0;i < n;i++)

cin >> a[i]

for(int i = 0;i < n;i++)

for(int j = 0;j < n;j++)

if(i!=j && a[i]+a[j] == z)

return true

return false

Below is the implementation of the above approach:

# Python3 program for the above approach

# Function to find a pair in the given

# array whose sum is equal to z

def findPair(a, n, z) :

# Iterate through all the pairs

for i in range(n) :

for j in range(n) :

# Check if the sum of the pair

# (a[i], a[j]) is equal to z

if (i != j and a[i] + a[j] == z) :

return True

return False

# Driver Code

# Given Input

a = [ 1, -2, 1, 0, 5 ]

z = 0

n = len(a)

# Function Call

if (findPair(a, n, z)) :

print("True")

else :

print("False")

**Output**

False

Assuming that each of the operations in the computer takes approximately constant time, let it be **c**. The number of lines of code executed actually depends on the value of **Z**. During analyses of the algorithm, mostly the worst-case scenario is considered, i.e., when there is no pair of elements with sum equals **Z**. In the worst case,

* **N\*c** operations are required for input.
* The outer loop **i** loop runs **N** times.
* For each **i**, the inner loop **j** loop runs **N** times.

So total execution time is **N\*c + N\*N\*c + c**. Now ignore the lower order terms since the lower order terms are relatively insignificant for large input, therefore only the highest order term is taken (without constant) which is **N\*N** in this case. Different notations are used to describe the limiting behavior of a function, but since the worst case is taken so [big-O notation](https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/) will be used to represent the time complexity.

Hence, the time complexity is **O(N2)** for the above algorithm. Note that the time complexity is solely based on the number of elements in array **A** i.e the input length, so if the length of the array will increase the time of execution will also increase.

**Order of growth** is how the time of execution depends on the length of the input. In the above example, it is clearly evident that the time of execution quadratically depends on the length of the array. Order of growth will help to compute the running time with ease.