

**“AZƏRBAYCAN HAVA YOLLARI” CJSC NATIONAL AVIATION ACADEMY**

**Individual Work № 1:**

**Topic: Computer Management Software in Windows**

**Subject: Obyektyönümlü proqramlaşdırma**

**Teacher: Mehemmed Shahmaliyev**

**Group: 1459i Student: Hesenova Firuze**

**Date: Signature:H.F**

**Baku 2022**

A lot of students get confused while understanding the concept of time-complexity, but in this article, we will explain it with a very simple example:  
Imagine a classroom of 100 students in which you gave your pen to one person. Now, you want that pen. Here are some ways to find the pen and what the O order is.  
O(): You go and ask the first person of the class, if he has the pen. Also, you ask this person about other 99 people in the classroom if they have that pen and so on,   
This is what we call O().   
O(n): Going and asking each student individually is O(N).   
O(log n): Now I divide the class into two groups, then ask: “Is it on the left side, or the right side of the classroom?” Then I take that group and divide it into two and ask again, and so on. Repeat the process till you are left with one student who has your pen. This is what you mean by O(log n).   
I might need to do the O() search if only one student knows on which student the pen is hidden. I’d use the O(n) if one student had the pen and only they knew it. I’d use the O(log n) search if all the students knew, but would only tell me if I guessed the right side.

*NOTE* : We are interested in rate of growth of time with respect to the inputs taken during the program execution .

Another Example:  
Time Complexity of algorithm/code is not equal to the actual time required to execute a particular code but the number of times a statement executes. We can prove this by using time command. For example, Write code in C/C++ or any other language to find maximum between N numbers, where N varies from 10, 100, 1000, 10000. And compile that code on Linux based operating system (Fedora or Ubuntu) with below command:

*gcc program.c – o program*

*run it with time ./program*

You will get surprising results i.e. for N = 10 you may get 0.5ms time and for N = 10, 000 you may get 0.2 ms time. Also, you will get different timings on the different machine. So, we can say that actual time requires to execute code is machine dependent (whether you are using pentium1 or pentium5) and also it considers network load if your machine is in LAN/WAN. Even you will not get the same timings on the same machine for the same code, the reason behind that the current network load.   
Now, the question arises if time complexity is not the actual time require executing the code then what is it?

The answer is : Instead of measuring actual time required in executing each statement in the code, we consider how many times each statement execute.   
Now-a-days, for one problem we can write the solution in n number of ways, but, how can we decide which type is better. We can use different types of algorithms to solve one problem. We need to compare these algorithms and have to choose the best one to solve the problem.

# **What is an algorithm?**

An algorithm is a set of instructions, which are created to get the required output. Many different algorithms can give same output.

To perform these instructions a computer should have memory, and it also requires time to perform those actions.

# **What is Time Complexity?**

The amount of time it takes to run the program and perform the functions in it is known as ****Time Complexity****. By using Time Complexity we can determine whether the program is efficient or we have to use another algorithm which take less time compared to the other one.

Reducing Time Complexity of an algorithm is often difficult in Data Science, rather than difficult we can say its a bigger challenge.

We will tell the time complexity of a program by calculating the time taken to run the algorithm in the ****worst-case**** scenario.

To quantify the Time Complexity, we will use ****Big-O**** notation.

# **Big-O Notation**

Big-O notation is used to classify algorithms according to how their run time or space requirements grow as the input size grows. The letter O is used because the growth rate of a function is also referred to as the **order of the function**or **order of the program**. We will always refer order of the function in it’s worst-case.

Let’s say we have a list which consists of integers and we have to check whether the number given by the user is present in that list or not.

l = [1,2,3,6,4,9,10,12]

k = 12

The simple code for this is

*l = [1,2,3,6,4,9,10,12]*

*k = 12*

*for i in range(0, len(l)):*

*if l[i] == k:*

*print("Yes")*

*break*

Here, the worst-case for this algorithm is to check the number which is present in the last element of the given list. So, if we go by the above program, first it’ll start with index 0 and check whether that element in the list is equal to k or not, i.e, one operation and we have to check for every element in the list for worst-case scenario.

In general terms, if a list has ’n’ elements in it. In the worst case scenario, we have to perform ’n’ check operations. That is denoted by ****O(n)**** [order of n]. So, the time complexity of above program is O(n).

*For better understanding of order of n, let’s see one more example:*

Given two lists of length ’n’ and check whether the sum of the elements in those two lists are even or odd.

l = [1,2,3,6,4,9,10,12]

k = [12,13,14,15,16,17]

*for i in l:*

*for j in k:*

*if (i+j)%2 == 0:*

*print("Even")*

*else:*

*print("Odd")*

Now, let see how we can determine the order for the above program in terms of n. Let’s assume, there are n elements in the given two lists. First for each value of i, j will have ’n’ elements to check for even or odd. Also, ‘i’ will have to run the loop for ’n’ elements.

So, there will be n\*n operations needed to perform the above program. So the time complexity of above program is ****O(n²)****.

Whenever you are calculating the order of a function if you come across as the following:

O(4n³ + 2n² + 3n + 1)

You can ignore the n², n and the constant and represent ****O(n³).****Because, when compared to O(n³), the values n² and n are pretty negligible, so there will be no impact on the output time even if we neglect them.

This is how, we will calculate or determine the Time Complexity of an algorithm.

In ****Data Science,****we often come across a large amount data and we need to search for particular entity or particular entry from that data. Reducing the time complexity in these type of programs will have huge impact in the real world.

Let’s see one of the searching technique which helps in reducing the time complexity.

# **Linear Search**

Let’s see the example I stated above once again to understand the Linear Search.

We have a list which consists of integers and we have to check whether the number given by the user is present in that list or not.

l = [1,2,3,6,4,9,10,12]

k = 12

*for i in range(0, len(l)):*

*if l[i] == k:*

*print("Yes")*

*break*

A ****linear search**** is the most basic kind of search that is performed. A linear or sequential search, is done when you inspect each item in a list one by one from one end to the other to find a match for what you are searching for.

The Time Complexity of the above program is ****O(n)****.

There are some more searches called ****Binary Search****, ****Jump Search****, ****Interpolation Search****, ****Exponential Search**** etc.

****Binary Search****is the most commonly used searching algorithm. You can find about Binary

We have seen how one can reduce Time Complexity for an algorithm by using different searches for sorted data.

Now, let’s see how we can sort the data. Sorting data is one of the most basic step you take in data exploration. There are so many sorting algorithms present but we will learn the three important sorting algorithms.

# **Merge Sort**

**Merge sort** is one of the most prominent *divide-and-conquer* sorting algorithms in the modern era. It can be used to sort the values in any data structure such as a list.

Merge sort works by splitting the input list into two halves, repeating the process on those halves, and finally merging the two sorted halves together.

The algorithm first moves from top to bottom, dividing the list into smaller and smaller parts until only the separate elements remain.

From there, it moves back up, ensuring that the merging lists are sorted.

Let’s see an example for Merge Sort.

L = [12, 13, 3, 7, 4, 2,1]

Output = [1, 2, 3, 4, 7, 12, 13]

*def mergeSort(arr):*

*if len(arr) >1:*

*mid = len(arr)//2*

*L = arr[:mid]*

*R = arr[mid:]*

*mergeSort(L)*

*mergeSort(R)*

*i = j = k = 0*

*while i < len(L) and j < len(R):*

*if L[i] < R[j]:*

*arr[k] = L[i]*

*i+= 1*

*else:*

*arr[k] = R[j]*

*j+= 1*

*k+= 1*

*while i < len(L):*

*arr[k] = L[i]*

*i+= 1*

*k+= 1*

*while j < len(R):*

*arr[k] = R[j]*

*j+= 1*

*k+= 1*

*l = []*

*def printList(arr):*

*for i in range(len(arr)):*

*print (arr[i],end=" ")*

*print()*

*if \_\_name\_\_ == '\_\_main\_\_':*

*L = [12, 13, 3, 7, 4, 2, 1]*

*print ("Given array is", end ="\n")*

*printList(L)*

*mergeSort(L)*

*print("Sorted array is: ", end ="\n")*

*printList(L)*

The Time Complexity for the above program is ****O(n\*logn)****. Which is less time when compared to Insertion and Selection Sort.

There are some more sorting algorithms ****Insertion Sort**** and ****Selection Sort.****There is also another type of called Quick Sort which follows the same divide and conquer rule like Merge Sort.

Most of the time, you will be using the inbuilt sort function in Python.