

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

**Individual Work № 1:**

**Topic: Time Complexity**

**Subject: Obyektyönümlü proqramlaşdırma**

**Teacher: Mamed Shahmaliyev**

**Group: 1459i Student: Mehrali Babayev**

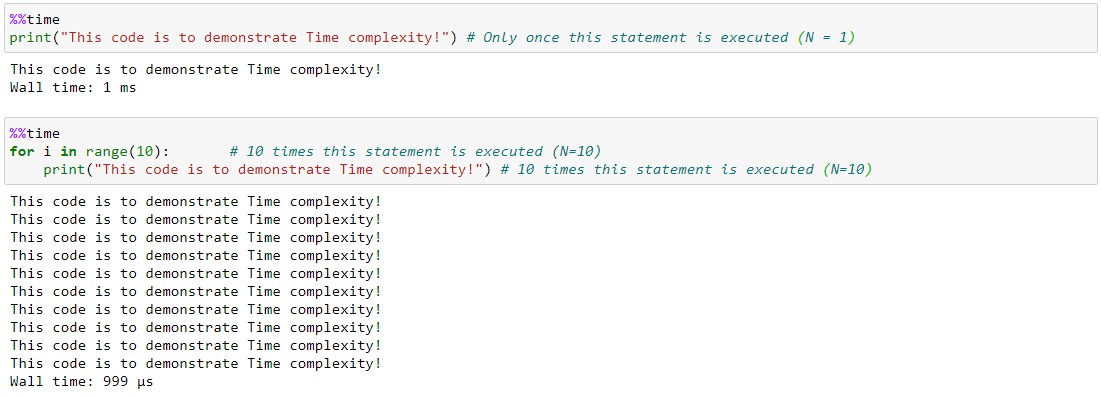
**Date: 11.10.2021 Signature: \_\_\_\_\_\_\_\_\_\_\_\_\_**

**­ Baku 2021**

## ****What is Time complexity?****

By definition, the time complexity is the amount of time taken by an algorithm to run, as a function of the length of the input. Here, the length of input indicates the number of operations to be performed by the algorithm. This gives a clear indication of what exactly Time complexity tells us. It is not going to examine the total execution time of an algorithm. Rather, it is going to give information about the variation (increase or decrease) in execution time when the number of operations (increase or decrease) in an algorithm. Yes, as the definition says, the amount of time taken is a function of the length of input only.

To elaborate, Time complexity measures the time taken to execute each statement of code in an algorithm. If a statement is set to execute repeatedly then the number of times that statement gets executed is equal to N multiplied by the time required to run that function each time.

For example, look at the code below: ****

The first algorithm is defined to print the statement only once. The time taken to execute is shown as **0 nanoseconds**. While the second algorithm is defined to print the same statement but this time it is set to run the same statement in FOR loop 10 times. In the second algorithm, the time taken to execute both the line of code – FOR loop and print statement, is **2 milliseconds**. And, the time taken increases, as the N value increases, since the statement is going to get executed N times.

**Note:** This code is run in Python-Jupyter Notebook with Windows 64-bit OS + processor Intel Core i7 ~ 2.4GHz. The above time value can vary with different hardware, with different OS and in different programming languages, if used.

By now, you could have concluded that when an algorithm uses statements that get executed only once, will always require the same amount of time, and when the statement is in loop condition, the time required increases depending on the number of times the loop is set to run. And, when an algorithm has a combination of both single executed statements and LOOP statements or with nested LOOP statements, the time increases proportionately, based on the number of times each statement gets executed.

This leads us to ask the next question, about how to determine the relationship between the input and time, given a statement in an algorithm. To define this, we are going to see how each statement gets an order of notation to describe time complexity, which is called Big**O Notation**.

**Time Complexity Introduction**

Space and Time define any physical object in the Universe. Similarly, Space and Time complexity can define the effectiveness of an algorithm. While we know there is more than one way to solve the problem in programming, knowing how the algorithm works efficiently can add value to the way we do programming. To find the effectiveness of the program/algorithm, knowing how to evaluate them using Space and Time complexity can make the program behave in required optimal conditions, and by doing so, it makes us efficient programmers.

While we reserve the space to understand Space complexity for the future, let us focus on Time complexity in this post. Time is Money! In this post, you will discover a gentle introduction to the Time complexity of an algorithm, and how to evaluate a program based on Time complexity.

## ****Why is Time complexity so significant?****

An Algorithm, in computer programming, is a finite sequence of well-defined instructions, typically executed in a computer, to solve a class of problems or to perform a common task. Based on the definition, there needs to be a sequence of defined instructions that have to be given to the computer to execute an algorithm/ perform a specific task. In this context, the variation can occur the way how the instructions are defined. There can be any number of ways, a specific set of instructions can be defined to perform the same task. Also, with options available to choose any one of the available programming languages, the instructions can take any form of syntax along with the performance boundaries of the chosen programming language. We also indicated the algorithm to be performed in a computer, which leads to the next variation, in terms of the operating system, processor, hardware, etc. that are used, which can also influence the way an algorithm can be performed.

Now that we know different factors can influence the outcome of an algorithm being executed, it is wise to understand how efficiently such programs are used to perform a task. To gauge this, we require to evaluate both the Space and Time complexity of an algorithm.

By definition, the Space complexity of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input. While 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. Now that we know why Time complexity is so significant, it is time to understand what is time complexity and how to evaluate it.

Python is a great tool to implement algorithms if you wish to become a programmer. Take up the free Python for Machine Learning Certificate Course and enhance your skills to power ahead your career.

## ****What are the different types of Time complexity notation used?****

As we have seen, Time complexity is given by time as a function of the length of the input. And, there exists a relation between the input data size (n) and a number of operations performed (N) with respect to time. This relation is denoted as Order of growth in Time complexity and given notation O[n] where O is the order of growth and n is the length of the input. It is also called as **‘Big O Notation’**

Big O Notation expresses the run time of an algorithm in terms of how quickly it grows relative to the input ‘n’ by defining the N number of operations that are done on it. Thus, the time complexity of an algorithm is denoted by the combination of all O[n] assigned for each line of function.

There are different types of time complexities used, let’s see one by one:

**1. Constant time – O (1)**

**2. Linear time – O (n)**

**3. Logarithmic time – O (log n)**

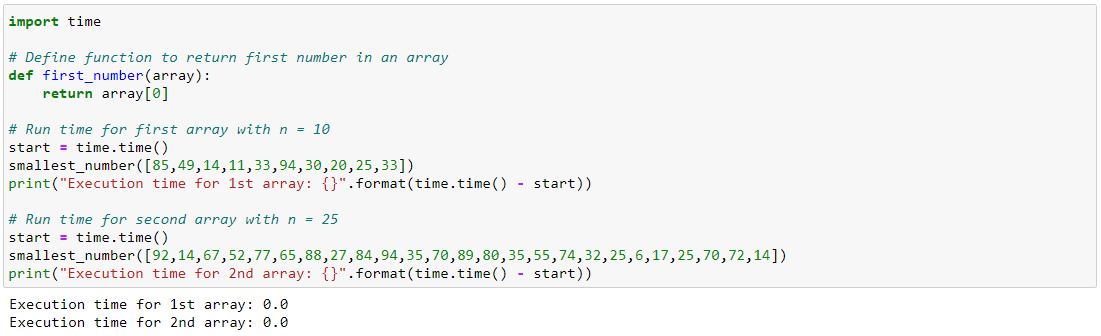
**4. Quadratic time – O (n^2)**

**5. Cubic time – O (n^3)**

and many more complex notations like **Exponential time, Quasilinear time, factorial time, etc.** are used based on the type of functions defined.

## ****Constant time – O (1)****

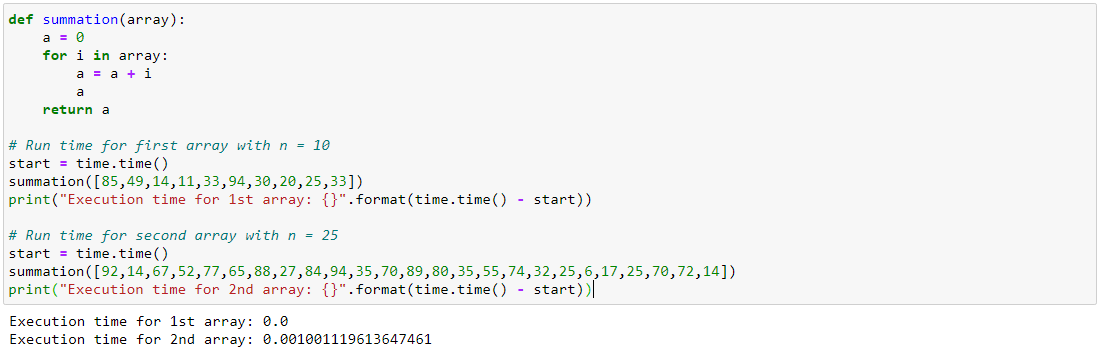
An algorithm is said to have constant time with order O (1) when it is not dependent on the input size n. Irrespective of the input size n, the runtime will always be the same. Example as shown:

****

The above code shows that irrespective of the length of the array (n), the runtime to get the first element in an array of any length is the same. If the run time is considered as 1 unit of time, then it takes only 1 unit of time to run both the arrays, irrespective of length. Thus, the function comes under constant time with order O (1).

## ****Linear time – O(n)****

An algorithm is said to have a linear time complexity when the running time increases linearly with the length of the input. When the function involves checking all the values in input data, such function has Time complexity with this order O(n). For example:

****

The above code shows that based on the length of the array (n), the run time will get linearly increased. If the run time is considered as 1 unit of time, then it takes only n times 1 unit of time to run the array. Thus, the function runs linearly with input size and this comes with order O(n).

**Logarithmic time – O (log n)**

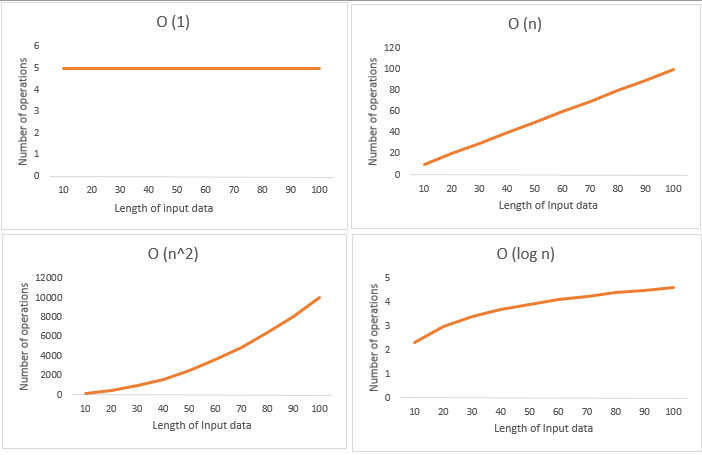
An algorithm is said to have a logarithmic time complexity when it reduces the size of the input data in each step. This indicates that the number of operations is not the same as the input size. The number of operations gets reduced as the input size increases. Algorithms with Logarithmic time complexity are found in binary trees or binary search functions. This involves the search of a given value in an array by splitting the array into two and starting searching in one split. This ensures the operation is not done on every element of the data.

**Quadratic time – O (n^2)**

An algorithm is said to have a non – linear time complexity where the running time increases non-linearly (n^2) with the length of the input. Generally, nested loops come under this time complexity order where for one loop takes O(n) and if the function involves a loop within a loop, then it goes for O(n)\*O(n) = O(n^2) order.

Similarly, if there are ‘m’ loops defined in the function, then the order is given by O (n ^ m), which are called **polynomial time complexity** functions.

The order of growth for all time complexities are indicated in the graph below:

****

**How to evaluate an algorithm for Time complexity?**

We have seen how the order notation is given to each function and the relation between runtime vs no of operations, input size. Now, it is time to know how to evaluate the Time complexity of an algorithm based on the order notation it gets for each operation & input size and compute the total run time required to run an algorithm for a given n.

Let us illustrate how to evaluate the time complexity of an algorithm with an example:

The algorithm is defined as:

1. Given 2 input matrix, which is a square matrix with order n

2. The values of each element in both the matrices are selected randomly using np.random function

3. Initially assigned a result matrix with 0 values of order equal to the order of input matrix

4. Each element of X is multiplied with every element of Y and the resultant value is stored in the result matrix

5. The resulting matrix is then converted to list type

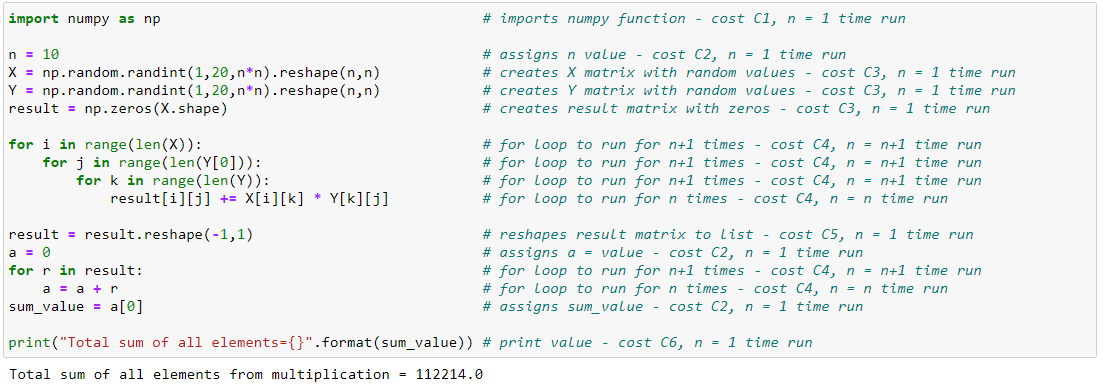
6. For every element in the result list, is added together to give the final answer

Let us assume cost function C as per unit time taken to run a function while ‘n’ represents the number of times the statement is defined to run in an algorithm.

For example, if time taken to run print function is say 1 microseconds (C) and if the algorithm is defined to run PRINT function for 1000 times (n),

then total run time = (C \* *n) = 1 microsec \** 1000 = 1 millisec

With this let us evaluate the Time complexity for our algorithm:

****

Run time for each line is given by:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | Line 1 = C1 \* 1  Line 2 = C2 \* 1  Line 3,4,5 = (C3 \* 1) + (C3 \* 1) + (C3 \* 1)  Line 6,7,8 = (C4\*[n+1]) \* (C4\*[n+1]) \* (C4\*[n+1])  Line 9 = C4\*[n]  Line 10 = C5 \* 1  Line 11 = C2 \* 1  Line 12 = C4\*[n+1]  Line 13 = C4\*[n]  Line 14 = C2 \* 1  Line 15 = C6 \* 1 |

Total run time = (C1\*1) + 3(C2\*1) + 3(C3\*1) + (C4\*[n+1]) \* (C4\*[n+1]) \* (C4\*[n+1]) + (C4\*[n]) + (C5\*1) + (C4\*[n+1]) + (C4\*[n]) + (C6\*1)

Replacing all cost with C to estimate the Order of notation,

Total Run Time

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | = C + 3C + 3C + ([n+1]C \* [n+1]C \* [n+1]C) + nC + C + [n+1]C + nC + C                                 = 7C + ((n^3) C + 3(n^2) C + 3nC + C + 3nC + 3C              = 12C + (n^3) C + 3(n^2) C + 6nC                = C(n^3) + C(n^2) + C(n) + C              = O(n^3) + O(n^2) + O(n) + O (1) |

By replacing all cost functions as C, we can get the degree of input size as 3, which tells the order of time complexity of this algorithm. Here, from the final equation, it is evident that the run time varies with the polynomial function of input size ‘n’ as it relates to the cubic, quadratic and linear forms of input size.

This is how the order of time complexity is evaluated for any given algorithm and to estimate how it spans out in terms of runtime if the input size is increased or decreased. Also note, for simplicity, all cost values like C1, C2, C3, etc. are replaced with C, to know the order of notation. In real-time, we need to know the value for every C, which can give the exact run time of an algorithm given the input value ‘n’.

## ****Time Complexity of Sorting algorithms****

Understanding the time complexities of sorting algorithms helps us in picking out the best sorting technique in a situation. Here are the time complexities of some sorting techniques:

### **Time Complexity of Insertion Sort:**

The time complexity of [Insertion Sort](https://www.mygreatlearning.com/blog/insertion-sort-with-a-real-world-example/) in the best case is O(n). In the worst case, the time complexity is O(n^2).

### **Time Complexity of Merge Sort:**

This sorting technique has a stable time complexity for all kinds of cases. The time complexity of Merge Sort in the best case is O(nlogn). In the worst case, the time complexity is O(nlogn). This is because Merge Sort implements a same number of sorting steps for all kinds of cases.

### **Time Complexity of Bubble Sort:**

The time complexity of [Bubble Sort](https://www.mygreatlearning.com/blog/bubble-sort/) in the best case is O(n). In the worst case, the time complexity is O(n^2).

### **Time Complexity of Quick Sort:**

The time complexity of Quick Sort in the best case is O(nlogn). In the worst case, the time complexity is O(n^2). Quicksort is considered to be the fastest of the sorting algorithms due to its performance of O(nlogn) in best and average cases.