

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

**Individual Work № 5:**

**Topic:** **Logarithmic time complexity. Binary seach.**

**Subject: Obyektyönümlü proqramlaşdırma**

**Teacher: Mamed Shahmaliyev**

**Group: 1459i Student: Rustam Azadaliyev**

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

**­ Baku 2021**

What is Time Complexity?

*In computer science, the time complexity is the computational complexity that describes the amount of time it takes to run an algorithm. Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, supposing that each elementary operation takes a fixed amount of time to perform.*

When analyzing the time complexity of an algorithm we may find three cases: **best-case**, **average-case** and **worst-case**. Let’s understand what it means.

Suppose we have the following unsorted list **[1, 5, 3, 9, 2, 4, 6, 7, 8]** and we need to find the index of a value in this list using [linear search](https://en.wikipedia.org/wiki/Linear_search).

* **best-case**: this is the complexity of solving the problem for the best input. In our example, the best case would be to search for the value 1. Since this is the first value of the list, it would be found in the first iteration.
* **average-case**: this is the average complexity of solving the problem. This complexity is defined with respect to the distribution of the values in the input data. Maybe this is not the best example but, based on our sample, we could say that the average-case would be when we’re searching for some value in the “middle” of the list, for example, the value 2.
* **worst-case**: this is the complexity of solving the problem for the worst input of size n. In our example, the worst-case would be to search for the value 8, which is the last element from the list.

Usually, when describing the time complexity of an algorithm, we are talking about the worst-case.

Ok, but how we describe the time complexity of an algorithm?

We use a mathematical notation called Big-O.

# Big-O Notation

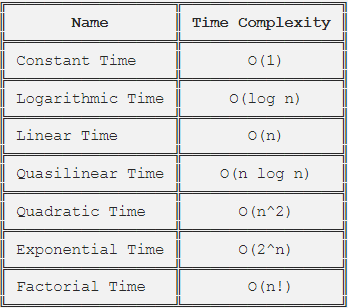
**Big-O notation**, sometimes called “asymptotic notation”, **is a mathematical notation that describes the limiting behavior of a function** when the argument tends towards a particular value or infinity.

In computer science, Big-O notation is used to classify algorithms according to how their running time or space requirements grow as the input size (n) grows. This notation characterizes functions according to their growth rates: different functions with the same growth rate may be represented using the same O notation.

Let’s see some common time complexities described in the Big-O notation.

## **Table of common time complexities**

These are the most common time complexities expressed using the Big-O notation:



Note that we will focus our study in these common time complexities but there are some other time complexities out there which you can study later.

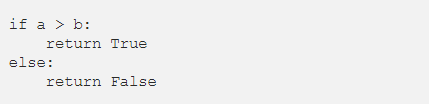
As already said, we generally use the Big-O notation to describe the time complexity of algorithms. There’s a lot of math involved in the formal definition of the notation, but informally we can assume that the Big-O notation gives us the algorithm’s approximate run time in the worst case. When using the Big-O notation, we describe the algorithm’s efficiency based on the increasing size of the input data (n). For example, if the input is a string, the n will be the length of the string. If it is a list, the n will be the length of the list and so on.

Now, let’s go through each one of these common time complexities and see some examples of algorithms. Note that I tried to follow the following approach: present a little description, show a simple and understandable example and show a more complex example (usually from a real-world problem).

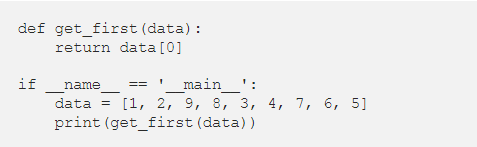
# Time Complexities

## **Constant Time — O(1)**

An algorithm is said to have a constant time when it is not dependent on the input data (n). No matter the size of the input data, the running time will always be the same. For example:



Now, let’s take a look at the function **get\_first** which returns the first element of a list:



Independently of the input data size, it will always have the same running time since it only gets the first value from the list.

An algorithm with constant time complexity is excellent since we don’t need to worry about the input size.

## **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 (it don’t need to look at all values of the input data), for example:

for index in range(0, len(data), 3):  
print(data[index])

Algorithms with logarithmic time complexity are commonly found in operations on binary trees or when using binary search. Let’s take a look at the example of a binary search, where we need to find the position of an element in a sorted list:

def binary\_search(data, value):  
 n = len(data)  
 left = 0  
 right = n - 1  
 while left <= right:  
 middle = (left + right) // 2  
 if value < data[middle]:  
 right = middle - 1  
 elif value > data[middle]:  
 left = middle + 1  
 else:  
 return middle  
 raise ValueError('Value is not in the list')  
   
if \_\_name\_\_ == '\_\_main\_\_':  
 data = [1, 2, 3, 4, 5, 6, 7, 8, 9]  
 print(binary\_search(data, 8))

Steps of the binary search:

* Calculate the middle of the list.
* If the searched value is lower than the value in the middle of the list, set a new right bounder.
* If the searched value is higher than the value in the middle of the list, set a new left bounder.
* If the search value is equal to the value in the middle of the list, return the middle (the index).
* Repeat the steps above until the value is found or the left bounder is equal or higher the right bounder.

It is important to understand that an algorithm that must access all elements of its input data cannot take logarithmic time, as the time taken for reading input of size n is of the order of n.

## **Linear Time — O(n)**

An algorithm is said to have a linear time complexity when the running time increases at most linearly with the size of the input data. This is the best possible time complexity when the algorithm must examine all values in the input data. For example:

for value in data:  
 print(value)

Let’s take a look at the example of a [linear search](https://en.wikipedia.org/wiki/Linear_search), where we need to find the position of an element in an unsorted list:

def linear\_search(data, value):  
 for index in range(len(data)):  
 if value == data[index]:  
 return index  
 raise ValueError('Value not found in the list')  
   
if \_\_name\_\_ == '\_\_main\_\_':  
 data = [1, 2, 9, 8, 3, 4, 7, 6, 5]  
 print(linear\_search(data, 7))

Note that in this example, we need to look at all values in the list to find the value we are looking for.

## **Quasilinear Time — O(n log n)**

An algorithm is said to have a quasilinear time complexity when each operation in the input data have a logarithm time complexity. It is commonly seen in sorting algorithms (e.g. mergesort, timsort, heapsort).

For example: for each value in the data1 (**O(n)**) use the binary search (**O(log n)**) to search the same value in data2.

for value in data1:  
 result.append(binary\_search(data2, value))

Another, more complex example, can be found in the Mergesort algorithm. Mergesort is an efficient, general-purpose, comparison-based sorting algorithm which has quasilinear time complexity, let’s see an example:

def merge\_sort(data):  
 if len(data) <= 1:  
 return  
   
 mid = len(data) // 2  
 left\_data = data[:mid]  
 right\_data = data[mid:]  
   
 merge\_sort(left\_data)  
 merge\_sort(right\_data)  
   
 left\_index = 0  
 right\_index = 0  
 data\_index = 0  
   
 while left\_index < len(left\_data) and right\_index < len(right\_data):  
 if left\_data[left\_index] < right\_data[right\_index]:  
 data[data\_index] = left\_data[left\_index]  
 left\_index += 1  
 else:  
 data[data\_index] = right\_data[right\_index]  
 right\_index += 1  
 data\_index += 1  
   
 if left\_index < len(left\_data):  
 del data[data\_index:]  
 data += left\_data[left\_index:]  
 elif right\_index < len(right\_data):  
 del data[data\_index:]  
 data += right\_data[right\_index:]  
   
if \_\_name\_\_ == '\_\_main\_\_':  
 data = [9, 1, 7, 6, 2, 8, 5, 3, 4, 0]  
 merge\_sort(data)  
 print(data)