УДК 811.111:004.05

TIME COMPLEXITY OF ALGORITHMS

Henry A.A., student
Lopatina A.N., student
Scientific supervisor – Vanik I.Y., senior lecturer
English language department №1
Belarusian National University of Technology
Minsk, Republic of Belarus

Understanding the complexity and efficiency of an algorithm execution is a crucial skill for a software engineer. This context can greatly help evaluate potential algorithm performance with various workloads, as it provides a good understanding of how fast or how efficient the code will handle different input sizes.

Unfortunately, calculating an exact run time for each potential in-put size can take a lot of time. Time complexity of an algorithm denotes the total time needed to complete execution. The most practical approach would be to calculate the worst scenario for the given data. And the complexity of this script is represented using such notation as Big-O.

In simple terms, Big-O describes how fast a function grows or declines [1]. This helps software engineers to compare different algorithms to determine the best solution to a given problem.

The simplest complexity of the algorithm is **O(1)**. It means that execution time doesn't change, and it doesn't matter how the size of the input data changes [2]. An example of an algorithm of such complexity may be the output of the second element of the array (Fig.1):

```cpp
#include <iostream>
using namespace std;

int main() {
    int arr[] = { 10, 20, 30, 40, 50 };

    cout << arr[1] << endl; // Always takes constant time

    return 0;
}
```

Figure 1 – Example of an algorithm with constant time complexity in C++

Next time complexity is **O(n)** – linear complexity. For algorithms with such difficulty, the execution time will increase by n times if you increase the size of the data by the same number of times [3]. An example of such an algorithm is passing through the entire array using a loop (Fig.2):

```cpp
#include <iostream>
using namespace std;

int main() {
    int arr[] = { 1, 2, 3, 4, 5 };
    int sum = 0;

    for (int i = 0; i < 5; i++) {
        sum += arr[i];
    }

    // Linear time complexity O(n), as each element is processed once.
    cout << "The sum of the array elements is: " << sum << endl;

    return 0;
}
```

Figure 2 – Example of an algorithm with linear time complexity in C++

With the embedding of one cycle into another, the complexity becomes quadratic – **O(n^2)**. That means, if you double the size of the input data, the execution time will increase 4 times. A bubble sort algorithm is the most common example of such difficulty (Fig.3) [2]:

```cpp
#include <iostream>
using namespace std;

void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap arr[j] and arr[j + 1]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

Figure 3 – Example of an algorithm with quadratic time complexity in C++

Another complexity is a logarithmic one – **O(log n)**. This difficulty would scale worse than constant, but better than quadratic (or even linear). A typical example of an algorithm with such complexity is a binary search in a sorted array (Fig. 4) [1]:

```cpp
#include <vector>
using namespace std;

// Function to perform binary search on a sorted vector
int binarySearch(const vector<int>& arr, int target) {
    int left = 0; // Initial index of the left boundary
    int right = arr.size() - 1; // Initial index of the right boundary

    // While the search range is valid
    while (left <= right) {
        int mid = left + (right - left) / 2; // Calculate the middle index

        if (arr[mid] == target) {
            return mid; // Element is found, return its index
        }
        else if (arr[mid] < target) {
            left = mid + 1; // Narrow search to the right half
        }
        else {
            right = mid - 1; // Narrow search to the left half
        }
    }

    return -1; // If the element is not found, return -1
}
```

Figure 4 – Example of an algorithm with logarithmic complexity in C++

Determining the time complexity of algorithms plays a particularly important role when working with large amounts of data, as well as when creating algorithms for sorting, searching, and filtering information.

References

1. Сложность алгоритмов. Разбор Big O. – URL: https://habr.com/ru/articles/782608/ (date of access: 22.03.2025).
2. Diwan, D. Calculate Space and Time Complexity of an Algorithm: A Beginner's Guide. – URL: https://devdiwan.medium.com/calculate-space-and-time-complexity-of-an-algorithm-a-beginners-guide-de0ffeb09e7e (date of access: 24.03.2025).
3. Time Complexity of Algorithms. – URL: https://www.studytonight.com/data-structures/time-complexity-of-algorithms (date of access: 25.03.2025).