ANALYSIS OF EXECUTION TIME FOR DIFFERENT TIME COMPLEXITIES

CSE373

Submitted By

Mahbub Ahmed Turza

ID:2211063042

Section:12

Submitted To:

Prof. Nadeem Ahmed

SUBMISSION DATE

04 April, 2024

Introduction:

In this study, we analyze the execution time of algorithms with different time complexities concerning varying input sizes. We explore algorithms with time complexities of O(n), O(log n), O(n log n), and O(n^2). By measuring the execution time for each complexity class with increasing input sizes, we aim to understand their scalability and efficiency.

Methodology:

I implemented four algorithms, each representing one of the mentioned time complexities:

- **Linear algorithm (O(n))** - A simple loop iterating over elements.
- **Binary search (O(log n))** - Utilizing a binary search algorithm.
- **Merge sort (O(n log n))** - Employing a divide-and-conquer sorting algorithm.
- **Nested loop (O(n^2))** - Nested iteration over elements.

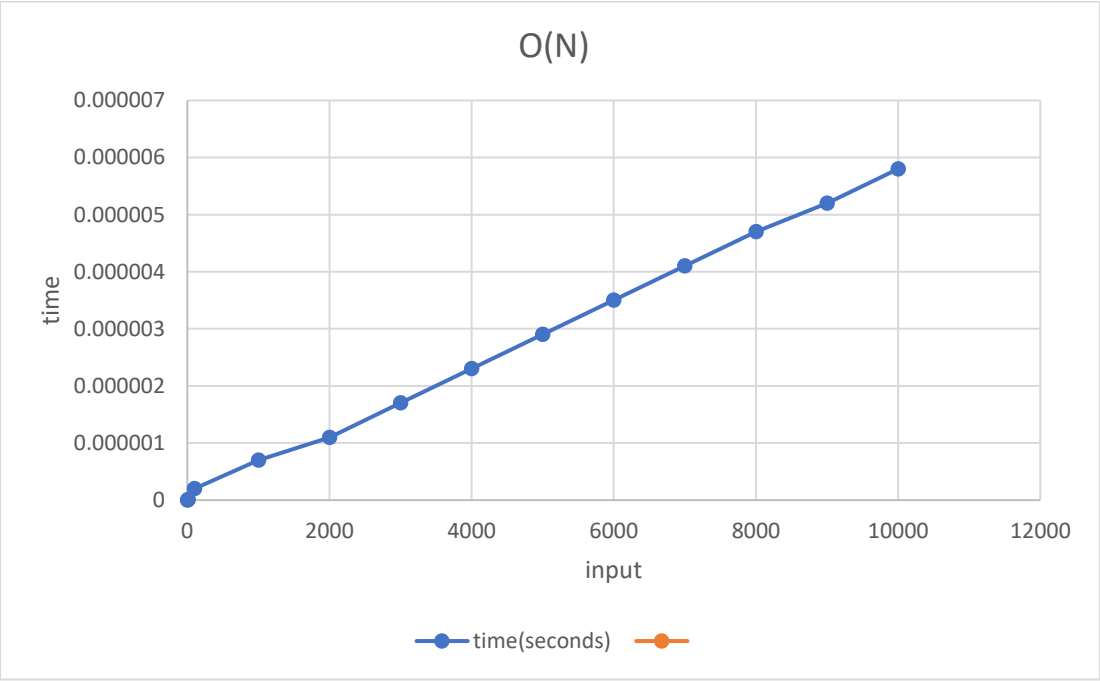
Input Types:

10,100,1000,2000,3000,40000,5000,

6000,7000,8000,9000,10000

Time Complexity O(N):

```cpp
#include <bits/stdc++.h>
using namespace std;
using namespace std::chrono;
int main()
{
    int n;
    vector<int> b{10, 100, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000};

    for (auto val : b)
    {
        n = val;
        auto start = high_resolution_clock::now();
        int result = 0;
        for (int i = 0; i < n; i++)
        {
            result += i;
        }
        auto stop = high_resolution_clock::now();
        auto duration = duration_cast<nanoseconds>(stop - start);
        cout << "Execution time: " << duration.count() << "seconds"
             << "\n";
    }

    return 0;
}
```

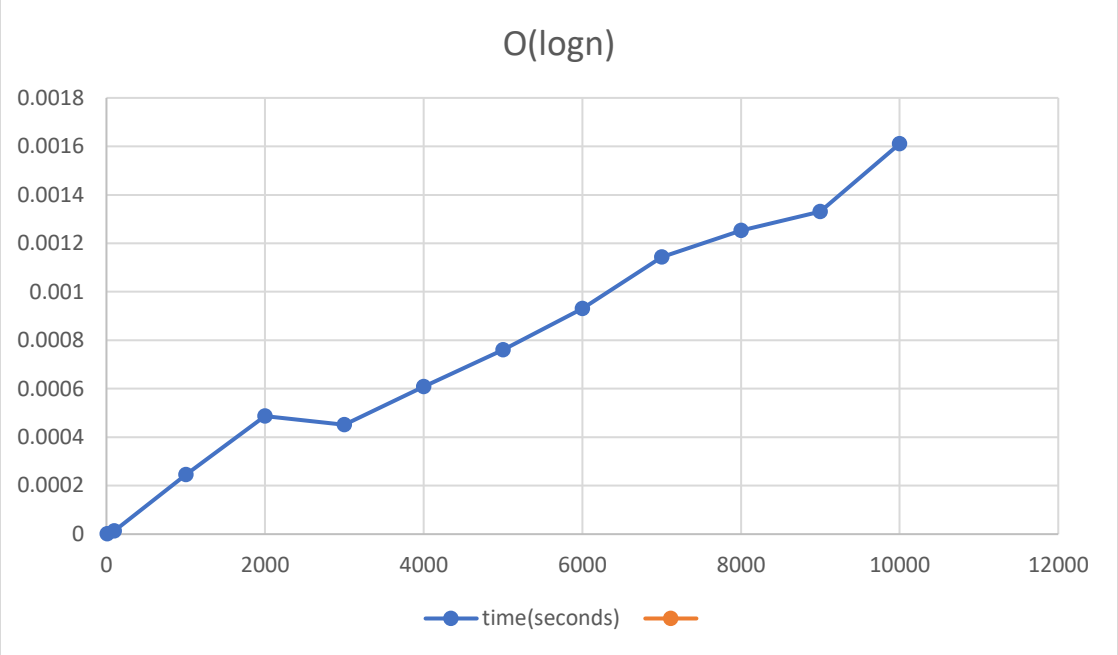


We get linear time complexity when the running time of an algorithm increases linearly with the size of the input. This means that when a function has an iteration that iterates over an input size of n, it is said to have a time complexity of order O(n).

In the example, the provided code snippet calculates the execution time of a loop that iterates n times, where n varies from 10 to 10,000. The execution time increases linearly with the increase in the value of n, consistent with O(n) time complexity.

Time Complexity O(logN):

```cpp
#include <bits/stdc++.h>
using namespace std;
using namespace std::chrono;
int binarySearch(const vector<int> &a, int target)
{
    int left = 0;
    int right = a.size() - 1;

    while (left <= right)
    {
        int mid = left + (right - left) / 2;
        if (a[mid] == target)
            return mid;
        if (a[mid] < target)
            left = mid + 1;
        else
            right = mid - 1;
    }
    return -1;
}
int main()
{
    vector<int> b{10, 100, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000};
    for (auto val : b)
    {
        vector<int> a;
        for (int i = 0; i < val; i++)
        {
            a.push_back(rand() % 100);
        }
        int target = rand() % 100;
        auto start = high_resolution_clock::now();
        sort(a.begin(), a.end());
        int index = binarySearch(a, target);
        auto stop = high_resolution_clock::now();
        auto duration = duration_cast<nanoseconds>(stop - start);
        if (index != -1)
            cout << "Element found at index " << index << endl;
        else
            cout << "Element not found in the array" << endl;

        cout << "Execution time: " << duration.count() << " nanoseconds"
             << "\n";
    }

    return 0;
}
```

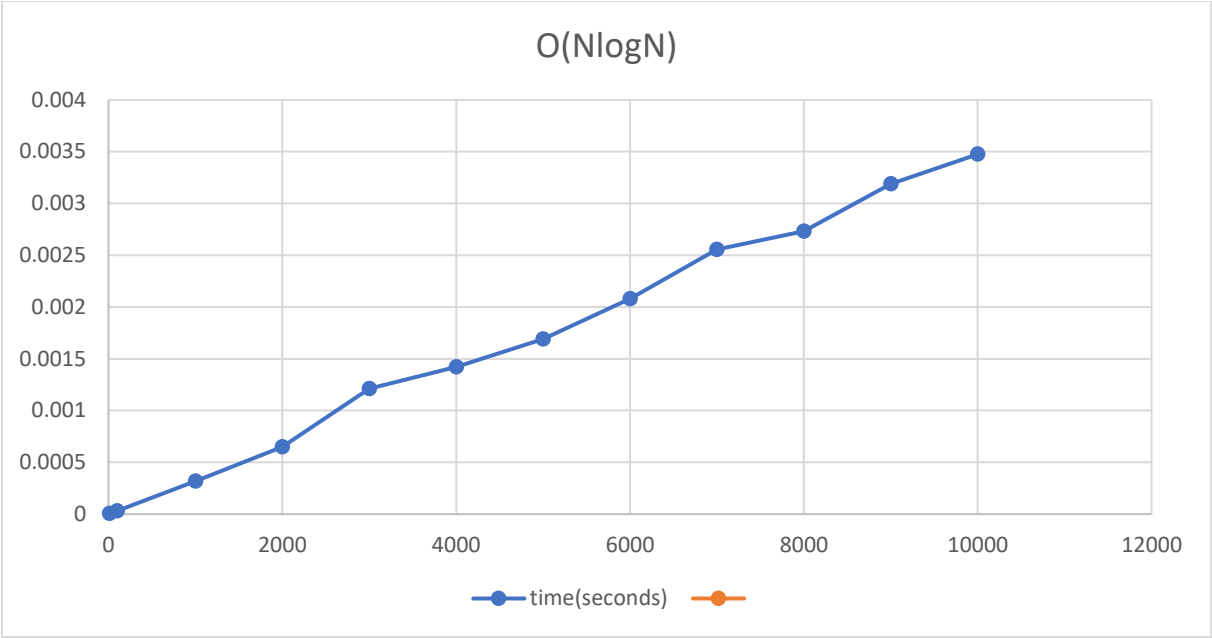


This is similar to linear time complexity, except that the runtime does not depend on the input size but rather on half the input size. When the input size decreases on each iteration or step, an algorithm is said to have logarithmic time complexity.

In the example, the provided code snippet calculates the execution time of a loop that iterates n times, where n varies from 10 to 10,000. The execution time increases steadily with the increase in the value of n.

Time Complexity O(nlogn)

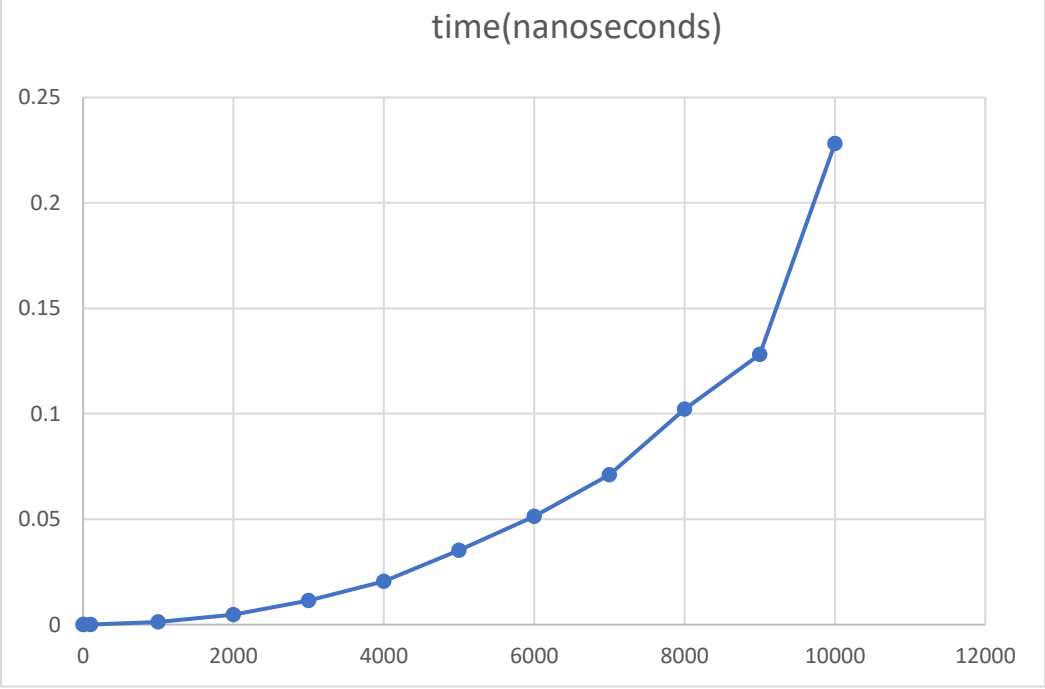
```cpp
#include <bits/stdc++.h>
using namespace std;
using namespace std::chrono;
void merge(vector<int> &arr, int low, int mid, int high)
{
    int n1 = mid - low + 1;
    int n2 = high - mid;
    vector<int> L(n1);
    vector<int> R(n2);

    for (int i = 0; i < n1; i++)
        L[i] = arr[low + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    int i = 0;
    int j = 0;
    int k = low;
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(vector<int> &arr, int low, int high)
{
    if (low < high)
    {
        int mid = low + (high - low) / 2;
        mergeSort(arr, low, mid);
        mergeSort(arr, mid + 1, high);
        merge(arr, low, mid, high);
    }
}

void printArray(const vector<int> &arr)
{
    for (int i = 0; i < arr.size(); i++)
    {
        cout << arr[i] << " ";
    }
    cout << endl;
}
int main()
{
    int n;
    vector<int> b{10, 100, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000};
    for (auto val : b)
    {
        vector<int> arr;
        for (int i = 0; i < val; i++)
        {
            arr.push_back(rand() % 100);
        }
        auto start = high_resolution_clock::now();

        mergeSort(arr, 0, arr.size() - 1);

        auto stop = high_resolution_clock::now();
        auto duration = duration_cast<nanoseconds>(stop - start);

        // cout << "Sorted array: ";
        // printArray(arr);
        cout << "Execution time: " << duration.count() << " nanoseconds"
             << "\n";
    }
    return 0;
}
```



O(nlogn) is known as loglinear complexity. O(nlogn) implies that logn operations will occur n times. O(nlogn) time is common in recursive sorting algorithms, binary tree sorting algorithms and most other types of sorts

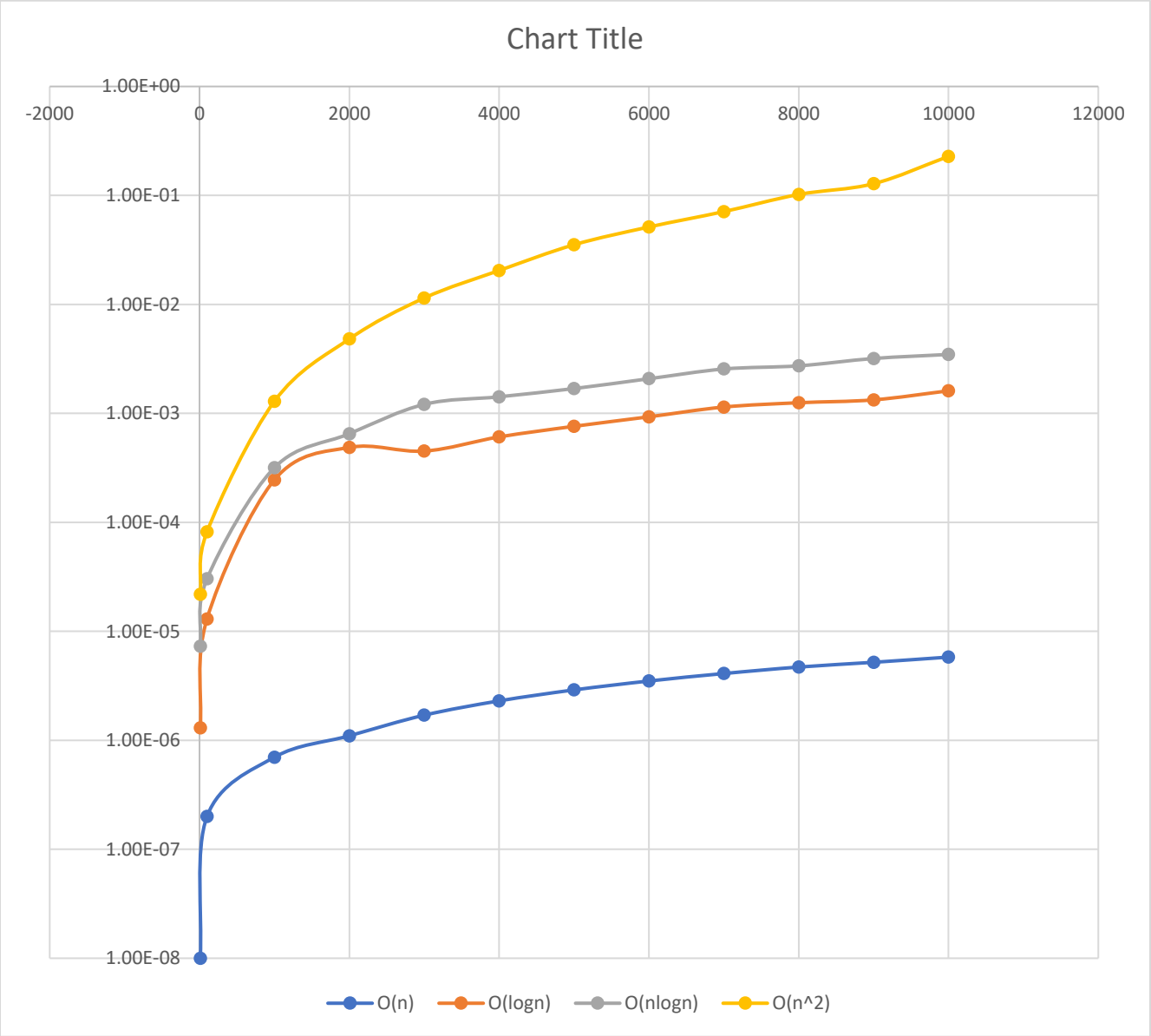
Time COmplexity O(N^2):

```cpp
#include <bits/stdc++.h>
using namespace std;
using namespace std::chrono;

int main()
{
    int N;
    cout << "Enter the value of N: ";
    cin >> N;
    auto start = high_resolution_clock::now();

    for (int i = 0; i < N; ++i)
    {
        for (int j = 0; j < N; ++j)
        {

            cout << "(" << i << ", " << j << ") ";
        }
        cout << endl;
    }
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<nanoseconds>(stop - start);
    cout << "Execution time: " << duration.count() << " nanoseconds"
         << "\n";

    return 0;
}
```



When we perform nested iteration, meaning having a loop in a loop, the time complexity is quadratic, which is horrible.

Comparison of these complexities:



Results:

The execution times for each algorithm and input size are summarized as follows:

Linear Algorithm (O(n)):

Execution time ranges from 1.00E-08 to 0.0000058 seconds.

Binary Search (O(log n)):

Execution time ranges from 0.0000013 to 0.0016107 seconds.

Merge Sort (O(n log n)):

Execution time ranges from 0.0000073 to 0.0034751 seconds.

Nested Loop (O(n^2)):

Execution time ranges from 0.0000218 to 0.228132 seconds.

Discussion:

- **Linear Algorithm:** The execution time increases linearly with the input size, indicating its linear time complexity.
- **Binary Search:** Despite logarithmic complexity, execution time increases steadily but at a slower rate compared to linear algorithms.
- **Merge Sort:** Demonstrates superior performance, growing moderately with input size due to its efficient divide-and-conquer approach.
- **Nested Loop:** Exhibits significant increase in execution time with larger inputs, confirming its quadratic time complexity.

Conclusion:

This analysis illustrates the relationship between algorithmic time complexity and execution time. Understanding how algorithms perform with different input sizes is crucial for designing efficient solutions, especially when dealing with large datasets.