**LAB Assignment Submission**

for

**Data Structures and Algorithms**

**Course Code: CSE2711**

**B.Tech CSE-VII/ECOM**

**Batch 2024**

**Name:** Vedansh Mathur

**Enrollment Number:** 240686

**Problem Statement-**

*To understand and experimentally verify different time complexity classes — O(1), O(log n),*

*O(n), O(n log n), O(n2), O(n3) and O(2n) — by implementing simple functions in C and*

*measuring the number of basic operations (comparisons) for various input sizes.*

**Solution -**

#include <iostream>

using namespace std;

int counter1 = 0; //global counter for O(1)

int counter2 = 0; //global counter for O(n)

int counter3 = 0; //global counter for O(log n)

int counter4 = 0; //global counter for O(n log n)

int counter5 = 0; //global counter for O(n^2)

int counter6 = 0; //global counter for O(2^n)

void constantfunction (int n){

    counter1++;

}

void linearfunction(int n){

    for (int i =1; i<=n; i++){

        counter2++;

    }

}

void logarithmicFunction(int n) {

    for (int i = 1; i < n; i \*= 2) {

        counter3++;

    }

}

void linearArithmicFunction(int n) {

    for (int i = 1; i <= n; i++) {

        for (int j = 1; j < n; j \*= 2) {

            counter4++;

        }

    }

}

void quadraticFunction(int n) {

    for (int i = 1; i <= n; i++) {

        for (int j = 1; j <= n; j++) {

            counter5++;

        }

    }

}

void exponentialFunction(int n) {

    for (int i = 1; i <= (1 << n); i++) {

        counter6++;

    }

}

int main () {

    int sizes[] = {1, 2, 4, 8, 16, 32, 64, 128};

    cout << "n\tO(1)\tO(n)\tO(log n)\tO(n log n)\tO(n^2)\tO(2^n)\n";

    for (int i = 0; i < 8; i++) {

        int n = sizes[i];

        // Reset all counters

        counter1 = counter2 = counter3 = counter4 = counter5 = counter6 = 0;

        constantfunction(n);

        linearfunction(n);

        logarithmicFunction(n);

        linearArithmicFunction(n);

        quadraticFunction(n);

        if (n <= 16) {

            exponentialFunction(n);

            cout << n << "\t" << counter1 << "\t" << counter2 << "\t" << counter3

                 << "\t\t" << counter4 << "\t\t" << counter5 << "\t" << counter6 << endl;

        } else {

            cout << n << "\t" << counter1 << "\t" << counter2 << "\t" << counter3

                 << "\t\t" << counter4 << "\t\t" << counter5 << "\t" << "Skipped" << endl;

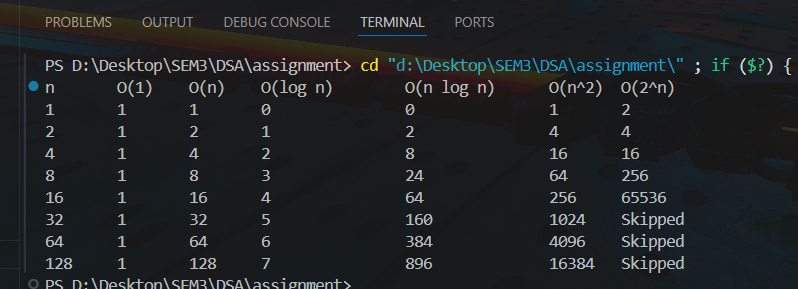
        }

    }

    return 0;

}

**Output -**

**

**Comments –**

O(1) - Constant Time:

Always takes the same amount of steps, no matter what n is.

It's most efficient and predictable.

For straightforward operations such as accessing an element in an array or returning a value.

O(n) - Linear Time:

The number of steps grows proportionally to n.

For n = 2, steps = 2; for n = 128, steps = 128.

This occurs with repeated loops that are executed once for each element (such as iterating over an array).

O(log n) - Logarithmic Time:

Grows slowly, even if n grows rapidly.

e.g., n = 128 has only 7 steps.

Applied to fast algorithms such as binary search.

O(n log n) - Linearithmic Time:

Grows more than linear and less than quadratic.

Steps are n times log n, so it grows steeply.

Used in fast sorting algorithms such as Merge Sort, Heap Sort.

O(n²) - Quadratic Time:

Steps grow extremely rapidly — gets very inefficient soon.

For n = 64, steps = 4096.

Used in simple brute force methods or nested loops.

Scalable only for small n.

O(2ⁿ) - Exponential Time:

Increases very rapidly, impractical for large n.

Ignored for n > 16 as it is very time-consuming to compute.

Used in instances such as recursion-intensive algorithms, subset generation, brute-force search (e.g., NP problems).