



**S. B. JAIN INSTITUTE OF TECHNOLOGY,
MANAGEMENT & RESEARCH, NAGPUR.**

Pre-Lab

Aim: Understand the concept of complexity and analyze the time complexities and space complexities of following program:

1. Sum of all elements of a list/array
2. Addition of two matrices.
3. Multiplication of two matrices.

Name of Student:

Roll No:

Semester/Year:

Academic Session:

Date of Performance:

Date of Submission:

Design and Analysis of Algorithms Lab(PCCCS503P)

AIM: Understand the concept of complexity and analyze the time complexities and space complexities of following program:

1. Sum of all elements of a list/array
2. Addition of two matrices.
3. Multiplication of two matrices.

OBJECTIVE/EXPECTED LEARNING OUTCOME:

The objectives and expected learning outcome of this practical are:

- To successfully understand the concept of performance analysis of algorithm.
- To find the space and time complexities of any given algorithm.

THEORY:

The term algorithm complexity measures how many steps are required by the algorithm to solve the given problem. It evaluates the order of count of operations executed by an algorithm as a function of input data size.

Performance Analysis of Algorithm:

The field of computer science, which studies the efficiency of algorithms, is known as analysis of algorithms.

1. Analysis of algorithms is the determination of the amounts of resources such as time and space resources required to implement the algorithm.
2. It is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size of memory for storage while implementation.
3. The complexity of an algorithm describes the efficiency of the algorithm in terms of the amount of the memory required to process the data and the processing time.
4. Complexity of an algorithm is analyzed in two perspectives: **Time** and **Space**.

Time Complexity:

Design and Analysis of Algorithms Lab(PCCCS503P)

It's a function describing the amount of time required to run an algorithm in terms of the size of the input. "Time" can mean the number of memory accesses performed, the number of comparisons between integers, the number of times some inner loop is executed, or some other natural unit related to the amount of real time the algorithm will take.

1. Time complexity will depend on the implementation of algorithm, programming language, optimizing capability of the compiler used, CPU speed, and hardware specifications of machines.
2. To measure the time complexity accurately, we have to count all sort of operations performed by the algorithm.
3. Time complexity depends on the amount of data inputted to the algorithm.

Space Complexity:

It's a function describing the amount of memory an algorithm takes in terms of the size of input to the algorithm. Space complexity typically includes,

1. **Instruction Space:** It is the amount of memory used to store compiled version of program instructions.
2. **Environmental Stack:** It is the amount of memory used to store information of partially executed functions at the time of function call.
3. **Data Space:** It is the amount of memory used to store all the variables and constants.
4. **Recursion Stack Space:** The amount of space needed by recursive function is called the recursion stack space. For each recursive function this space depends on the space needed by local variables and formal parameters. It also depends on the maximum depth of the recursion.

O(1): This denotes the constant time. $O(1)$ usually means that an algorithm will have constant time regardless of the input size. Hash Maps are perfect examples of constant time.

O(log n): This denotes logarithmic time. $O(\log n)$ means to decrease with each instance for the operations. Binary search trees are the best examples of logarithmic time.

O(n): This denotes linear time. $O(n)$ means that the performance is directly proportional to the input size. In simple terms, the number of inputs and the time taken to execute those inputs will be proportional or the same. Linear search in arrays is the best example of linear time complexity.

O(n²): This denotes quadratic time. $O(n^2)$ means that the performance is directly proportional to the square

of the input taken. In simple, the time taken for execution will take square times the input size. Nested loops are perfect examples of quadratic time complexity.

Average, Best and Worst Case Analysis:

There are three cases which are usually used to compare various data structure's execution time in a relative manner.

Worst Case

1. This is the scenario where a particular data structure operation takes maximum time it can take.
2. It calculates upper bound on running time of an algorithm.
3. The worst-case complexity of the algorithm is the function defined by the maximum number of steps taken on any instance of size n .
4. Example: The worst case for linear search algorithm on an array occurs when the desired element is the last element of the array or element not found in the array at all.

Average Case

1. This is the scenario depicting the average execution time of an operation of a data structure.
2. The average case complexity of the algorithm is the function defined by the average number of steps taken on any instance of size n .
3. It takes all possible inputs and calculates the computing time for all of the inputs.
4. Example: The average case for linear search algorithm on an array occurs when the desired element is found in the middle of the array.

Best Case

1. This is the scenario depicting the least possible execution time of an operation of a data structure. In the bestcase analysis, we calculate lower bound on running time of an algorithm.
2. We must know the case that causes minimum number of operations to be executed.
3. The best case complexity of the algorithm is the function defined by the minimum number of steps taken on any instance of size n .
4. Best case performance is less widely found.
5. Example: The best case for linear search algorithm on an array occurs when the desired element is the first element of the array.

ALGORITHM:

1) Sum of all elements of a list/array

```
Add (A,n)
{
    S=0;
    For (i=0; i<n; i++)
    {
        S=S+A[i];
    }
    return 0;
}
```

2) Addition of two matrices.

```
Add(A,B,n)
{
    For (i=0; i<n; i++)
    {
        For (j=0; j<n; j++)
        {
            C[i][j]=A[i][j]+B[i][j];
        }
    }
    return 0 ;
}
```

3) Multiplication of two matrices.

```
Multiply(A,B,n)
{
    For (i=0; i<n; i++)
    {
        For (j=0; j<n; j++)
        {
            C[i] [j]=0
        }
    }
}
```

```
For (k=0; k<n; k++)
{
C[i] [j]= C[i] [j] +A[i] [k] * B[k] [j];
}
}
}
}
```

CODE:

1)Sum of all elements of a list/array

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE 10 // Size of the list
int sumList(int arr[], int size) {
int sum = 0;
for (int i = 0; i < size; i++) {
sum += arr[i];
}
return sum;
}
int main() {
int *arr = malloc(SIZE * sizeof(int));
// Initialize the array with some values
for (int i = 0; i < SIZE; i++) {
arr[i] = i + 1;
}
clock_t start_time, end_time;
double time_used;
start_time = clock();
int sum = sumList(arr, SIZE);
end_time = clock();
time_used = ((double) (end_time - start_time)) / CLOCKS_PER_SEC;
```

```
printf("No of elements in array : %d\n",SIZE);
printf("Sum of all elements in the list: %d\n", sum);
printf("Time taken to compute the sum: %f seconds\n", time_used);
free(arr);
return 0;
}
```

2) Addition of two matrices.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define n 10
int A[n][n];
int B[n][n];
int C[n][n];
int rows, cols;
void initializeMatrices() {
    srand(time(NULL));
    printf("Enter number of rows and columns for matrices (max %d): ", n);
    scanf("%d %d", &rows, &cols);
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            A[i][j] = rand() % 10 + 1;
            B[i][j] = rand() % 10 + 1;
        }
    }
}
void printMatrix(int matrix[][n]) {
    printf("Matrix Elements:\n");
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            printf("%d ", matrix[i][j]);
        }
    }
}
```

```
    }
    printf("\n");
}
}

double addMatrices() {
    clock_t st = clock();
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            C[i][j] = A[i][j] + B[i][j];
        }
    }
    clock_t ed = clock();
    return (double)(ed - st) / CLOCKS_PER_SEC;
}

int main() {
    initializeMatrices();
    printf("\nMatrix A:\n");
    printMatrix(A);
    printf("\nMatrix B:\n");
    printMatrix(B);
    double res = addMatrices();
    printf("\nMatrix C (A + B):\n");
    printMatrix(C);
    printf("Time taken: %f seconds\n", res);
    printf("Space complexity:%ld byte",sizeof(A)+sizeof(B)+sizeof(C));
    return 0;
}
```

3) Multiplication of two matrices.

```
#include <stdio.h>
#include <time.h>
```



```
#include <stdlib.h>

int main()
{
    int i, j, k;
    int a[3][3], b[3][3], c[3][3];
    clock_t start, end;
    double cpu_time_used;
    srand(time(NULL));
    start = clock();
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            a[i][j] = rand() % 10;
            b[i][j] = rand() % 10;
            c[i][j] = 0;
        }
    }
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            for (k = 0; k < 3; k++) {
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }
    end = clock();
    printf("Matrix A:\n");
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            printf("%d ", a[i][j]);
        }
        printf("\n");
    }
}
```

```
printf("\nMatrix B:\n");
for (i = 0; i < 3; i++) {
    for (j = 0; j < 3; j++) {
        printf("%d ", b[i][j]);
    }
    printf("\n");
}
printf("\nThe matrix multiplication (C = A * B):\n");
for (i = 0; i < 3; i++) {
    for (j = 0; j < 3; j++) {
        printf("%d ", c[i][j]);
    }
    printf("\n");
}
cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
printf("\nTime complexity: %f seconds\n", cpu_time_used);
printf("Estimated space complexity: %ld bytes\n",
    sizeof(a) + sizeof(b) + sizeof(c) + sizeof(i) + sizeof(j) + sizeof(k));
return 0;
}
```

INPUT & OUTPUT WITH DIFFERENT TEST CASES:

1)Sum of all elements of a list/array

The screenshot shows the OnlineGDB interface with a C program that calculates the sum of elements in an array of size 10. The program includes `<stdio.h>`, `<stdlib.h>`, and `<time.h>`. It defines a constant `SIZE` as 10. The `sumList` function takes an array and its size, iterates through the elements, and returns the sum. The `main` function allocates memory for the array, checks for successful allocation, and calls `sumList`. The output shows the sum of the first 10 elements of the array is 55, and the time taken to compute the sum is 0.000001 seconds.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 #define SIZE 10 // Size of the list
6
7 int sumList(int arr[], int size) {
8     int sum = 0;
9     for (int i = 0; i < size; i++) {
10         sum += arr[i];
11     }
12     return sum;
13 }
14
15 int main() {
16     int *arr = malloc(SIZE * sizeof(int));
17
18     // Check if memory allocation was successful
19     if (arr == NULL) {
20         printf("Memory allocation failed\n");
21         return 1;
22     }
```

input

No of elements in array: 10
Sum of all elements in the list: 55
Time taken to compute the sum: 0.000001 seconds

...Program finished with exit code 0
Press ENTER to exit console.

The screenshot shows the OnlineGDB interface with a C program that calculates the sum of elements in an array of size 100. The program is similar to the first one, but the `SIZE` constant is defined as 100. The output shows the sum of the first 100 elements of the array is 5050, and the time taken to compute the sum is 0.000002 seconds.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 #define SIZE 100 // Size of the list
6
7 int sumList(int arr[], int size) {
8     int sum = 0;
9     for (int i = 0; i < size; i++) {
10         sum += arr[i];
11     }
12     return sum;
13 }
14
15 int main() {
16     int *arr = malloc(SIZE * sizeof(int));
17
18     // Check if memory allocation was successful
19     if (arr == NULL) {
20         printf("Memory allocation failed\n");
21         return 1;
22     }
```

input

No of elements in array: 100
Sum of all elements in the list: 5050
Time taken to compute the sum: 0.000002 seconds

...Program finished with exit code 0
Press ENTER to exit console.

Design and Analysis of Algorithms Lab(PCCCS503P)

The screenshot shows the OnlineGDB interface. The code in the editor is as follows:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 #define SIZE 1000 // Size of the list
6
7 int sumList(int arr[], int size) {
8     int sum = 0;
9     for (int i = 0; i < size; i++) {
10         sum += arr[i];
11     }
12     return sum;
13 }
14
15 int main() {
16     int *arr = malloc(SIZE * sizeof(int));
17     // Check if memory allocation was successful
18 }
```

The console output shows the following:

```
input
No of elements in array: 1000
Sum of all elements in the list: 500500
Time taken to compute the sum: 0.000002 seconds

...Program finished with exit code 0
Press ENTER to exit console.
```

2) Addition of two matrices.

The screenshot shows the OnlineGDB interface. The code in the editor is as follows:

```
55
56 printf("\nMatrix A:\n");
57 printMatrix(A);
```

The console output shows the following:

```
input
Enter number of rows and columns for matrices (max 10): 3
3

Matrix A:
Matrix Elements:
10 9 7
5 10 4
8 1 4

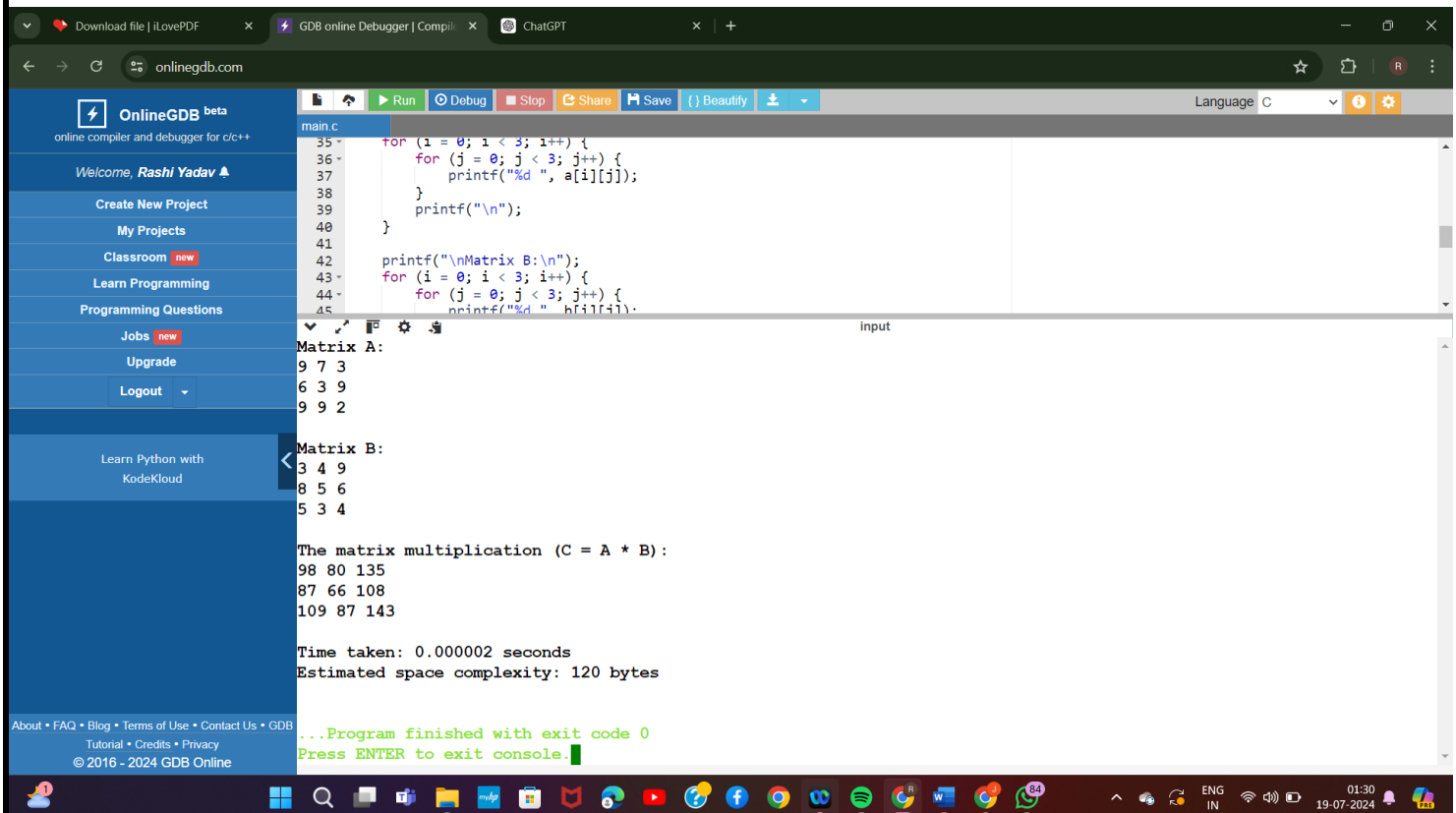
Matrix B:
Matrix Elements:
9 8 4
10 9 10
4 4 9

Matrix C (A + B):
Matrix Elements:
19 17 11
15 19 14
12 5 13

Time taken: 0.000001 seconds
Space complexity: 1200 bytes

...Program finished with exit code 0
Press ENTER to exit console.
```

3) Multiplication of two matrices.



```
main.c
35-   for (i = 0; i < 3; i++) {
36-       for (j = 0; j < 3; j++) {
37-           printf("%d ", a[i][j]);
38-       }
39-       printf("\n");
40-   }
41-
42-   printf("\nMatrix B:\n");
43-   for (i = 0; i < 3; i++) {
44-       for (j = 0; j < 3; j++) {
45-           printf("%d " b[i][j]);
46-       }
47-       printf("\n");
48-   }
49-
50-   printf("\nThe matrix multiplication (C = A * B) :");
51-   for (i = 0; i < 3; i++) {
52-       for (j = 0; j < 3; j++) {
53-           printf("%d ", c[i][j]);
54-       }
55-       printf("\n");
56-   }
57-
58-   printf("\nTime taken: 0.000002 seconds\n");
59-   printf("Estimated space complexity: 120 bytes\n");
60-
61-   return 0;
62- }
```

Matrix A:
9 7 3
6 3 9
9 9 2

Matrix B:
3 4 9
8 5 6
5 3 4

The matrix multiplication (C = A * B) :
98 80 135
87 66 108
109 87 143

Time taken: 0.000002 seconds
Estimated space complexity: 120 bytes

...Program finished with exit code 0
Press ENTER to exit console.

CONCLUSION: Hence we have successfully understand the concept of complexity and analyze the time complexities and space complexity of Sum of all elements of a list/array, addition of two matrices and multiplication of two matrices.

DISCUSSION AND VIVA VOCE:

- Elaborate the concept of time complexities.
- What is auxiliary space?
- What is the best case analysis?
- What is the worstcase analysis?
- What is the time complexity and space complexity of addition of two matrices and multiplication of two matrices?
- Explain asymptotic notations

REFERENCES:

<https://www.javatpoint.com/introduction-to-computational-complexity-theory>
<https://www.javatpoint.com/daa-complexity-of-algorithm>

Department of Computer Science and Engineering, S.B.J.I.T.M.R, Nagpur.

