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: Shrutika Pradeep Bagdi

Roll No: CS22130

Semester/Year: 5th / 3rd

Academic Session: 2024-2025

Date of Performance:

Date of Submission:

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:

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(n^2): 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 best case 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.

1. **Sum of all elements of a list/array**

ALGORITHM:

1. Initialization:

- Define a constant ‘n’ with a value of ‘1000’.
- Declare an integer array ‘list’ of size ‘n’.

2. Create List Function (‘create_list’):

- Seed the random number generator using ‘srand(time(0))’ to ensure different random sequences each time the program is run.
- Loop through each index of the array ‘list’ from ‘0’ to ‘n-1’:
- Assign a random integer between ‘0’ and ‘99’ to ‘list[i]’ using ‘rand() % 100’.

3. Print List Function (‘print_list’):

- Print a header "List elements:".
- Loop through each index of the array ‘list’ from ‘0’ to ‘n-1’:
- Print the value of ‘list[i]’ followed by a tab character.
- Print a newline character after printing all elements.

4. Process List Function (‘proc_list’):

- Record the current time using ‘clock()’ at the start of processing.
- Initialize a variable ‘sum’ to ‘0’.
- Loop through each index of the array ‘list’ from ‘0’ to ‘n-1’:
- Add the value of ‘list[i]’ to ‘sum’.
- Record the current time using ‘clock()’ at the end of processing.
- Calculate the time taken for processing as the difference between ‘end_time’ and ‘start_time’ divided by ‘CLOCKS_PER_SEC’.
- Print the total sum of the list elements.
- Print the time taken to process the list.
- Print the memory required to store the list.

5. Main Function (main):

- Call ‘create_list()’ to populate the array with random values.
- Call ‘print_list()’ to display the elements of the array.
- Call ‘proc_list()’ to compute and display the sum of the elements and the time taken to process the list.

CODE:

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

#define n 1000

int list[n]; // Size of the list

void create_list(){
   // Seed the random number generator
   srand(time(0)); // sets the starting point for generating random numbers

   // Initialize the array with random integer values
   for (int i = 0; i < n; i++){
      list[i] = rand() % 100; // Random values between 0 and 99
   }
}

void print_list(){  // Print the list
   printf("List elements:\n");
   for (int i = 0; i < n; i++)
      printf("%d\t", list[i]);
   printf("\n");
}
```

```
void proc_list(){
    // Get the start time
    clock_t start_time = clock();

    long sum = 0;
    for (int i = 0; i < n; i++){
        sum += list[i];
    }

    // Get the end time
    clock_t end_time = clock();

    // Calculate the time taken
    double time_taken = ((double) (end_time - start_time)) / CLOCKS_PER_SEC;

    // Print the result and execution time
    printf("Sum of %d elements: %ld\n", n, sum);
    printf("Time taken to process the list: %f seconds\n", time_taken);
    printf("Memory required to store %d elements in the list: %ld bytes\n", n, sizeof(int) * n);
}

int main(){
    create_list();
    print_list();
    proc_list();

    return 0;
}
```

INPUT & OUTPUT WITH DIFFERENT TEST CASES:

```
List elements:
44      72      37      86      0       30      96      60      36      32
Sum of 10 elements: 493
Time taken to process the list: 0.000002 seconds
Memory required to store 10 elements in the list: 40 bytes


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

```
List elements:
53      35      81      12      73      44      33      59      4       76      97      71      74      93      48      43      22      67
      48      35      90      44      98      77      6       13      96      13      13      77      6       18      13      87      30
    38      83      15      97      87      91      46      10      17      91      58      13      14      78      61      1       68
  57      0       98      63      13      94      77      78      24      35      96      37      22      26      27      57      93
76      45      85      74      7       2       66      18      15      32      96      28      33      16      37      85      14      53
      51      61      82      29      85      17      26      74      39      4       53      48      98
Sum of 100 elements: 4963
Time taken to process the list: 0.000002 seconds
Memory required to store 100 elements in the list: 400 bytes


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

```
3       79      7       80      21      14      16      77      23      33      46      18      68      16      87      60      81      9
1       0       45      79      54      32      25      77      85      51      20      27      61      94      62      41      1       4
3       14      15      59      91      91      92      38      9       12      6       49      24      87      92      76      85      2
3       31      69      48      60      6       99      80      86      61      74      48      54      28      91      68      95      2
59      38      46      97      0       10      3       1       86      43      93      63      80      16      46      1       16      6
7       16      38      93      29      64      42      83      92      85      51      40      40      10      78      38      8       3
0       1       63      31      39      6       24      2       38      92      0       39      9       58      47      77      96      9
2       58      61      86      41      5       24      44      97      16      54      28      54      14      58      7       30      9
0       47      88      66      1       27      11      2       66      72      12      65      1       61      58      59      74      9
6       52      79      20      96      29      36      2       57      43      69      15      50      99      57      49      87      7
6       51      14      87      5       33      11      17      98      12      78      8       23      52      5       75      84      2
5       23      13      14      25      22      57      46      37      59      45      47      9       85      23      12      51      6
2       17      84      25      86      35      37      17      43      12      21      0       39      57      26      62      22      4
0       39      44      49      86      34      60      83      81      69      68      56      33      72      70      2       8       9
5       41      43      84      58      39      96      31      39      35      89      17      97      63      9       88      8       1
0       74      42      71      58      75      92      78      31      26      50      1       80      11      48      21      6       3
2       79      45      28      63      37      15      4       6       12      67      16      0       27      78      27      69      4
9       37      96      94      15      27      20      18      80      0       81      28      74      87      12      5       85      4
0       68      22      55      24      80      19      92      48      72      19      27      51      41      76      88      37      7
0       55      17      42      25      49      95      6
Sum of 1000 elements: 48730
Time taken to process the list: 0.000004 seconds
Memory required to store 1000 elements in the list: 4000 bytes
```

2. **Addition of two matrices.**

ALGORITHM:

1. **Initialization:**

- Define constants ‘ROWS’ and ‘COLS’ as ‘3’.
- Declare three 3x3 integer matrices: ‘matrix1’, ‘matrix2’, and ‘result’.

2. **Create Matrix (‘create_matrix’):**

- Seed the random number generator with ‘srand(time(0))’.
- Fill the given matrix with random integers between ‘0’ and ‘99’.

3. **Print Matrix (‘print_matrix’):**

- Print each element of the given matrix in a tabular format.

4. **Add Matrices (‘add_matrices’):**

- Record start time with ‘clock()’.
- Compute the element-wise sum of ‘matrix1’ and ‘matrix2’, storing the result in ‘result’.
- Record end time with ‘clock()’.
- Calculate and print the time taken for matrix addition.

5. **Main (main):**

- Create and print ‘matrix1’.
- Create and print ‘matrix2’.
- Add ‘matrix1’ and ‘matrix2’ to get ‘result’ and print it.
- Print the memory required to store each matrix.

CODE:

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

#define ROWS 3
```

```
#define COLS 3

int matrix1[ROWS][COLS];
int matrix2[ROWS][COLS];
int result[ROWS][COLS];

void create_matrix(int matrix[ROWS][COLS]) {
    srand(time(0)); // Seed the random number generator

    for (int i = 0; i < ROWS; i++) {
        for (int j = 0; j < COLS; j++) {
            matrix[i][j] = rand() % 100; // Random values between 0 and 99
        }
    }
}

void print_matrix(int matrix[ROWS][COLS]) {
    for (int i = 0; i < ROWS; i++) {
        for (int j = 0; j < COLS; j++) {
            printf("%d\t", matrix[i][j]);
        }
        printf("\n");
    }
}

void add_matrices() {
    // Get the start time
    clock_t start_time = clock();

    for (int i = 0; i < ROWS; i++) {
        for (int j = 0; j < COLS; j++) {
```

```
            result[i][j] = matrix1[i][j] + matrix2[i][j];
        }
    }

    // Get the end time
    clock_t end_time = clock();

    // Calculate the time taken
    double time_taken = ((double) (end_time - start_time)) / CLOCKS_PER_SEC;

    // Print the execution time
    printf("Time taken to add matrices: %f seconds\n", time_taken);
}

int main() {
    // Create and print the first matrix
    printf("Matrix 1:\n");
    create_matrix(matrix1);
    print_matrix(matrix1);

    // Create and print the second matrix
    printf("Matrix 2:\n");
    create_matrix(matrix2);
    print_matrix(matrix2);

    // Add the matrices and print the result
    add_matrices();
    printf("Resultant Matrix:\n");
    print_matrix(result);

    // Print the memory required to store the matrices
```

```
    printf("Memory required to store each matrix: %ld bytes\n", sizeof(matrix1));

    return 0;
}
```

INPUT & OUTPUT WITH DIFFERENT TEST CASES:

```
Matrix 1:
66      54      26
45      4       0
0       42      24
Matrix 2:
66      54      26
45      4       0
0       42      24
Time taken to add matrices: 0.000001 seconds
Resultant Matrix:
132     108     52
90      8       0
0       84      48
Memory required to store each matrix: 36 bytes


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

3. **Multiplication of two matrices.**

ALGORITHM:

1. **Initialization:**

- Define constants ROWS and COLS as 3.
- Declare three 3x3 integer matrices: matrix1, matrix2, and result.
- Declare loop variables i, j, and k.

2. **Create Matrix (create_matrix):**

- Seed the random number generator with srand(time(0)).
- Fill the matrix with random integers between 0 and 9.

3. **Print Matrix (print_matrix):**

- Print each element of the matrix in a formatted manner.

4. **Multiply Matrices (multiply_matrices):**

- Record the start time with clock().
- Compute the matrix product:
- Initialize each element of result to 0.
- For each element in result, compute the dot product of the corresponding row from matrix1 and column from matrix2.
- Record the end time with clock().
- Calculate and print the time taken for multiplication.

5. **Main (main):**

- Create and print matrix1.
- Create and print matrix2.
- Multiply matrix1 and matrix2 to get result and print it.
- Print the memory required to store all matrices.

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define ROWS 3
#define COLS 3

int matrix1[ROWS][COLS];
int matrix2[ROWS][COLS];
int result[ROWS][COLS];
int i, j, k;
```

```
void create_matrix(int matrix[ROWS][COLS]) {
    srand(time(0)); // Seed the random number generator

    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLS; j++) {
            matrix[i][j] = rand() % 10; // Random values between 0 and 9
        }
    }
}

void print_matrix(int matrix[ROWS][COLS]) {
    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLS; j++) {
            printf("%d\t", matrix[i][j]);
        }
        printf("\n");
    }
}

void multiply_matrices() {
    // Get the start time
    clock_t start_time = clock();

    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLS; j++) {
            result[i][j] = 0;
            for (k = 0; k < COLS; k++) {
                result[i][j] += matrix1[i][k] * matrix2[k][j];
            }
        }
    }
```

```
    // Get the end time
    clock_t end_time = clock();

    // Calculate the time taken
    double time_taken = ((double) (end_time - start_time)) / CLOCKS_PER_SEC;

    // Print the execution time
    printf("Time taken to multiply matrices: %f seconds\n", time_taken);
}

int main() {
    // Create and print the first matrix
    printf("Matrix 1:\n");
    create_matrix(matrix1);
    print_matrix(matrix1);

    // Create and print the second matrix
    printf("Matrix 2:\n");
    create_matrix(matrix2);
    print_matrix(matrix2);

    // Multiply the matrices and print the result
    multiply_matrices();
    printf("Resultant Matrix:\n");
    print_matrix(result);
    // Print the memory required to store the matrices
    printf("Memory required to store each matrix: %ld bytes\n", sizeof(matrix1) + sizeof(matrix2) +
sizeof(result));
    return 0;
}
```

INPUT & OUTPUT WITH DIFFERENT TEST CASES:

```
Matrix 1:
1       3       1
0       2       0
6       3       6
Matrix 2:
1       3       1
0       2       0
6       3       6
Time taken to multiply matrices: 0.000002 seconds
Resultant Matrix:
7       12      7
0       4       0
42      42      42
Memory required to store each matrix: 108 bytes


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

CONCLUSION:

Thus understand the concept of complexity and analyze the time complexities and space complexities of the programs Sum of all elements of a list/array , Addition of two matrices, 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 worst case 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