DESIGN AND ANALYSIS OF ALGORITHMS

LAB FILE

Submitted by –

Name-Prabhat Mishra

Sap_id-500120575

Enrolment No-R2142231785

Batch-52

3rd Semester,

BTech CSE

UPES Dehradun, Uttarakhand

CONTENTS

EXPERIMENT 1

Implement the insertion inside iterative and recursive Binary search tree and compare their performance.

CODE:

Recursive BST:

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

#define SIZE 60000

struct Node* insertRecursive(struct Node* , int);  // Prototype functions for creating and and inserting iteratively in a binary tree.
struct Node* createNode(int);

// Structure for a node in the binary search tree.
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

int main(){

    clock_t start_time;  // clock_t is a data type for measuring processor time in clock ticks.
    clock_t end_time;
    double total_time;

    start_time=clock();

// Initialize the array to store random numbers
    int arr[SIZE];

    // Seed the random number generator
    srand(time(0));

    // Generate random numbers and store them in the array
    for (int i = 0; i < SIZE; i++) {
        arr[i] = rand();
    }
    printf("\n");
```

```c
    // Create an empty binary search tree (root node is NULL)
    struct Node* root = NULL;

    // Insert numbers from the array into the binary search tree iteratively
    for (int i = 0; i < SIZE; i++) {
        root = insertRecursive(root, arr[i]);
    }

    end_time=clock();
    total_time=((double)(end_time - start_time)) / CLOCKS_PER_SEC;

    printf("Total elapsed time is %f seconds", total_time);


    return 0;
}


// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Function to insert a node recursively in a binary search tree
struct Node* insertRecursive(struct Node* root, int data) {
    // Base case: If the tree is empty, return the new node as the root
    if (root == NULL) {
        return createNode(data);
    }

    // Recursive case: Traverse the tree to find the correct position
    if (data < root->data) {
        root->left = insertRecursive(root->left, data);  // Insert in the left subtree
    } else if (data > root->data) {
        root->right = insertRecursive(root->right, data); // Insert in the right subtree
    }

    // Return the unchanged root pointer
    return root;
```

ITERATIVE BST:

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

#define SIZE 60000


struct Node* insertIterative(struct Node* , int );  // Prototype functions for creating and and inserting iteratively in a binary tree.
struct Node* createNode(int );

// Structure for a node in the binary search tree
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

int main(){
     clock_t start_time;  // clock_t is a data type for measuring processor time in clock ticks.
     clock_t end_time;
     double total_time;

    // Initialize the array to store random numbers
    int arr[SIZE];

    // Seed the random number generator
    srand(time(0));

    // Generate random numbers and store them in the array
    for (int i = 0; i < SIZE; i++) {
        arr[i] = rand();
    }
    printf("\n");

    // Create an empty binary search tree (root node is NULL)
    struct Node* root = NULL;

    start_time=clock();     // Starting the clock to measure time.

    // Insert numbers from the array into the binary search tree iteratively.
    for (int i = 0; i < SIZE; i++) {
        root = insertIterative(root, arr[i]);
    }
```

```c
    for (int i = 0; i < SIZE; i++) {

    end_time=clock();
    total_time=((double)(end_time - start_time)) / CLOCKS_PER_SEC;

    printf("Total elapsed time is %f seconds", total_time);


    return 0;
}



// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}


// Function to insert a node iteratively in a binary search tree
struct Node* insertIterative(struct Node* root, int data) {
    struct Node* newNode = createNode(data);

    if (root == NULL) {
        return newNode; // If the tree is empty, return the new node as root
    }

    struct Node* current = root;
    struct Node* parent = NULL;

    while (current != NULL) {
        parent = current;
        if (data < current->data) {
            current = current->left;
        } else if (data > current->data) {
            current = current->right;
        } else {
            // Duplicates not allowed, return the root unchanged
```

```c
        } else {
            // Duplicates not allowed, return the root unchanged
            return root;
        }
    }

    // Insert the new node at the correct position
    if (data < parent->data) {
        parent->left = newNode;
    } else {
        parent->right = newNode;
    }

    return root; // Return the unchanged root pointer
}
```

OUTPUTS:

RECURSIVE BST:

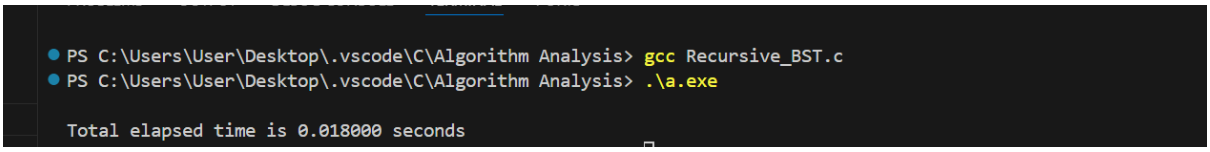
1. N=10000

```
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> gcc Recursive_BST.c
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe

Total elapsed time is 0.007000 seconds
```

2. N=20000

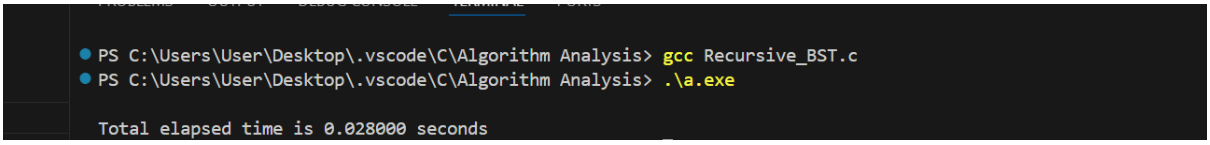
```
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> gcc Recursive_BST.c
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe

Total elapsed time is 0.011000 seconds
```

3. N=30000



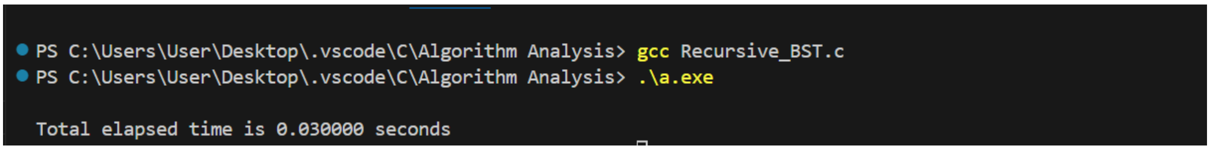
```
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> gcc Recursive_BST.c
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe

Total elapsed time is 0.018000 seconds
```

4. N=40000



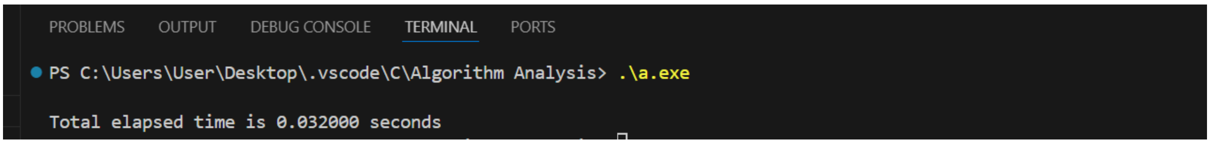
```
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> gcc Recursive_BST.c
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe

Total elapsed time is 0.028000 seconds
```

5. N=50000



```
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> gcc Recursive_BST.c
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe

Total elapsed time is 0.030000 seconds
```

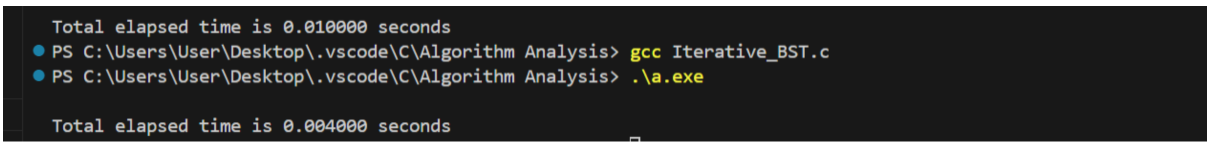
6. N=60000



```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe

Total elapsed time is 0.032000 seconds
```

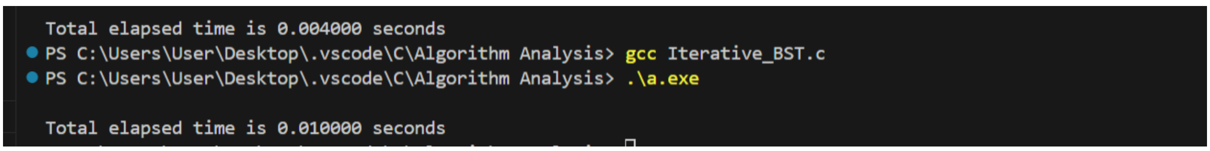
ITERATIVE BST:

1. N=10000



```
Total elapsed time is 0.010000 seconds
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> gcc Iterative_BST.c
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe

Total elapsed time is 0.004000 seconds
```

2. N=20000



```
Total elapsed time is 0.004000 seconds
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> gcc Iterative_BST.c
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe

Total elapsed time is 0.010000 seconds
```

3. N=30000

```
 Total elapsed time is 0.010000 seconds
● PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> gcc Iterative_BST.c
● PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe

 Total elapsed time is 0.011000 seconds
```

4. N=40000

```
 Total elapsed time is 0.011000 seconds
● PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> gcc Iterative_BST.c
● PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe

 Total elapsed time is 0.016000 seconds
```

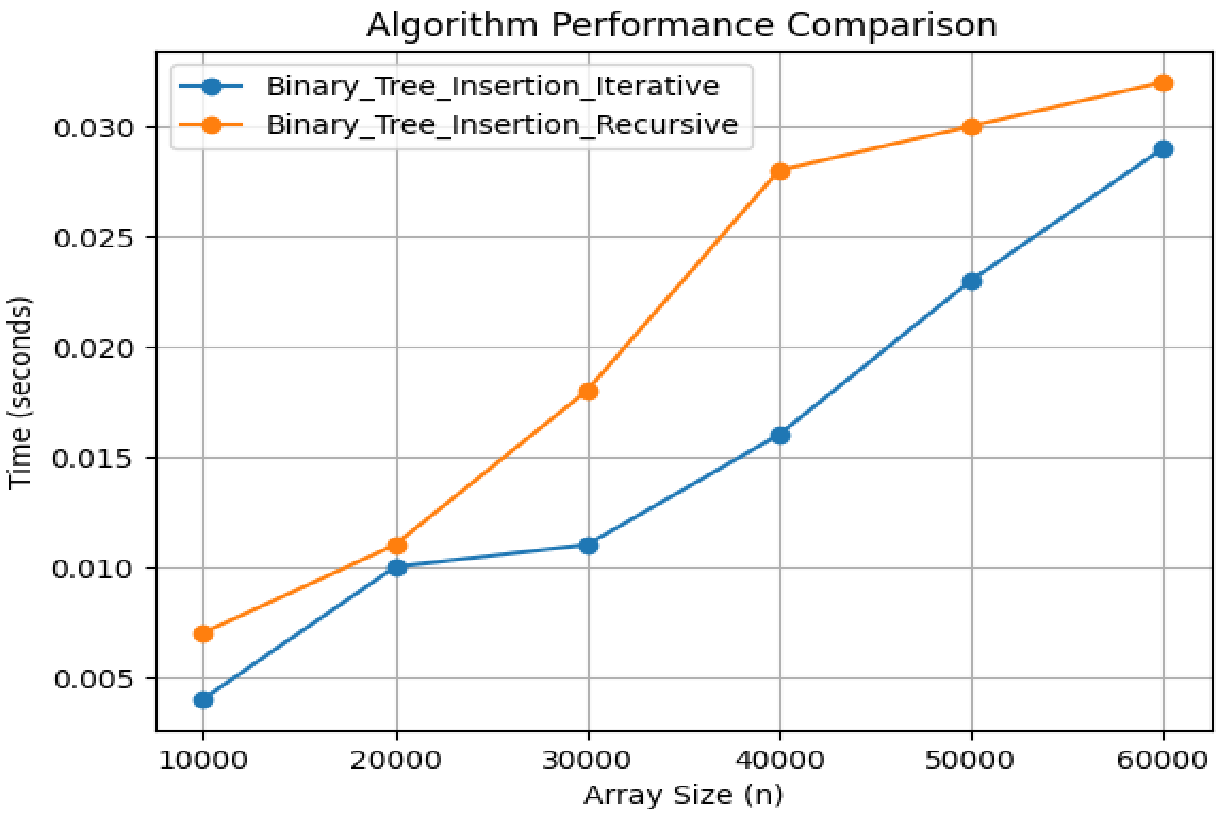
5. N=50000

```
 Total elapsed time is 0.016000 seconds
● PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> gcc Iterative_BST.c
● PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe

 Total elapsed time is 0.023000 seconds
```

6. N=60000

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

● PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> gcc Iterative_BST.c
● PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe

 Total elapsed time is 0.029000 seconds
```

<u>GRAPH ANALYSIS:</u>



Algorithm Performance Comparison
Binary_Tree_Insertion_Iterative
Binary_Tree_Insertion_Recursive
Time (seconds)
0.030
0.025
0.020
0.015
0.010
0.005
10000
20000
30000
40000
50000
60000
Array Size (n)

EXPERIMENT 2

Implement divide and conquer based merge sort and quick sort algorithms and compare their performance for the same set of elements.

CODE:

MERGE SORT:

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

#define SIZE 60000

void mergeSort(int arr[],int,int);     // Prototype functions for mergeSort and merge functions.
void merge(int arr[],int,int,int);



int main(){

    clock_t start_time;  // clock_t is a data type for measuring processor time in clock ticks.
    clock_t end_time;
    double total_time;



    // Initialize the array to store random numbers
    int arr[SIZE];

    // Seed the random number generator
    srand(time(0));

    // Generate random numbers and store them in the array
    for (int i = 0; i < SIZE; i++) {
        arr[i] = rand();
    }
    printf("\n");

    start_time=clock();

    mergeSort(arr,0,SIZE-1); // Implementing Merge Sort.

    end_time=clock();   // Ending the clock here as merge sort function is implemented.
```

```
    total_time=((double)(end_time - start_time)) / CLOCKS_PER_SEC;  //Measuring the time taken.
    printf("Total elapsed time is %f seconds", total_time);

    return 0;
}


void merge(int arr[], int beg, int mid, int end) // Function using the conquer approach to merge and sort the divided sub-arrays.
{
    int i=beg, j=mid+1, index=beg, k;
    int temp[SIZE];
    while ((i<=mid) &&(j<=end))   // Copying the elements into a temporary array after sorting them.
    {
        if (arr[i]<arr[j])
        {
          temp[index]=arr[i];
          i++;
        }
        else
        {
          temp[index]=arr[j];
          j++;
        }
        index++;
    }
    if (i>mid)                // Copying the remaining elements of the right subarray if there exists any.
    {
        while (j<=end)
        {
          temp[index]=arr[j];
          j++;
          index++;
        }
    }
    else
```

```
    {
        while (i<=mid)      // Copying the remaining elements of the left subarray if there exists any.
        {
            temp[index]=arr[i];
            i++;
            index++;
        }
    }

    for (k=beg;k<index;k++)        // Finally copying the sorted elements of the temporary back into the original array.
    {
        arr[k]=temp[k];
    }
}

void mergeSort(int arr[], int beg, int end) // Function using the divide approach to divide the array into subarrays recursively until all subarrays have 1 element.
{
    if (beg < end) {

        int mid = beg + (end - beg) / 2;

        // Sort first and second halves.
        mergeSort(arr, beg, mid);
        mergeSort(arr, mid + 1, end);

        // Merge the sorted halves.
        merge(arr, beg, mid, end);
    }
}
```

QUICK SORT:

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

#define SIZE 60000


void swap(int* , int* );               // Prototype functions for swap, parition and quick sort functions.
int partition(int arr[], int , int );
void quickSort(int arr[], int , int );


int main(){


    clock_t start_time;  // clock_t is a data type for measuring processor time in clock ticks.
    clock_t end_time;
    double total_time;

    // Starting the clock to measure time.

    // Initialize the array to store random numbers
    int arr[SIZE];

    // Seed the random number generator
    srand(time(0));

    // Generate random numbers and store them in the array
    for (int i = 0; i < SIZE; i++) {
        arr[i] = rand();
    }
    printf("\n");

    start_time=clock();  // Starting the clock to measure time.

    quickSort(arr,0,SIZE-1); // Implementing Quick Sort.
```

```c
    end_time=clock();    // Ending the clock here as Quick sort function is implemented.

    total_time=((double)(end_time - start_time)) / CLOCKS_PER_SEC;  //Measuring the time taken.
    printf("Total elapsed time is %f seconds", total_time);

    return 0;
}


// Function to swap two elements
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Quick Sort partition function
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Taking the last element as pivot
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
```

```c
// Quick Sort function
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

OUTPUTS:

MERGE SORT:

1. N=10000

```
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> gcc Merge_DAC.c
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe

Total elapsed time is 0.003000 seconds
```

2. N=20000

```
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> gcc Merge_DAC.c
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe

Total elapsed time is 0.007000 seconds
```

3. N=30000

```
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> gcc Merge_DAC.c
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe

Total elapsed time is 0.010000 seconds
```

4. N=40000

```
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> gcc Merge_DAC.c
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe

Total elapsed time is 0.022000 seconds
```

5. N=50000

```
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> gcc Merge_DAC.c
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe

Total elapsed time is 0.030000 seconds
```

6. N=60000

```
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> gcc Merge_DAC.c
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe

Total elapsed time is 0.039000 seconds
```

QUICK SORT

1. N=10000

```
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> gcc Quick_DAC.c
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe

Total elapsed time is 0.002000 seconds
```

2. N=20000

```
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> gcc Quick_DAC.c
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe

Total elapsed time is 0.007000 seconds
```

3. N=30000

```
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> gcc Quick_DAC.c
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe

Total elapsed time is 0.008000 seconds
```

4. N=40000

```
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> gcc Quick_DAC.c
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe

Total elapsed time is 0.013000 seconds
```

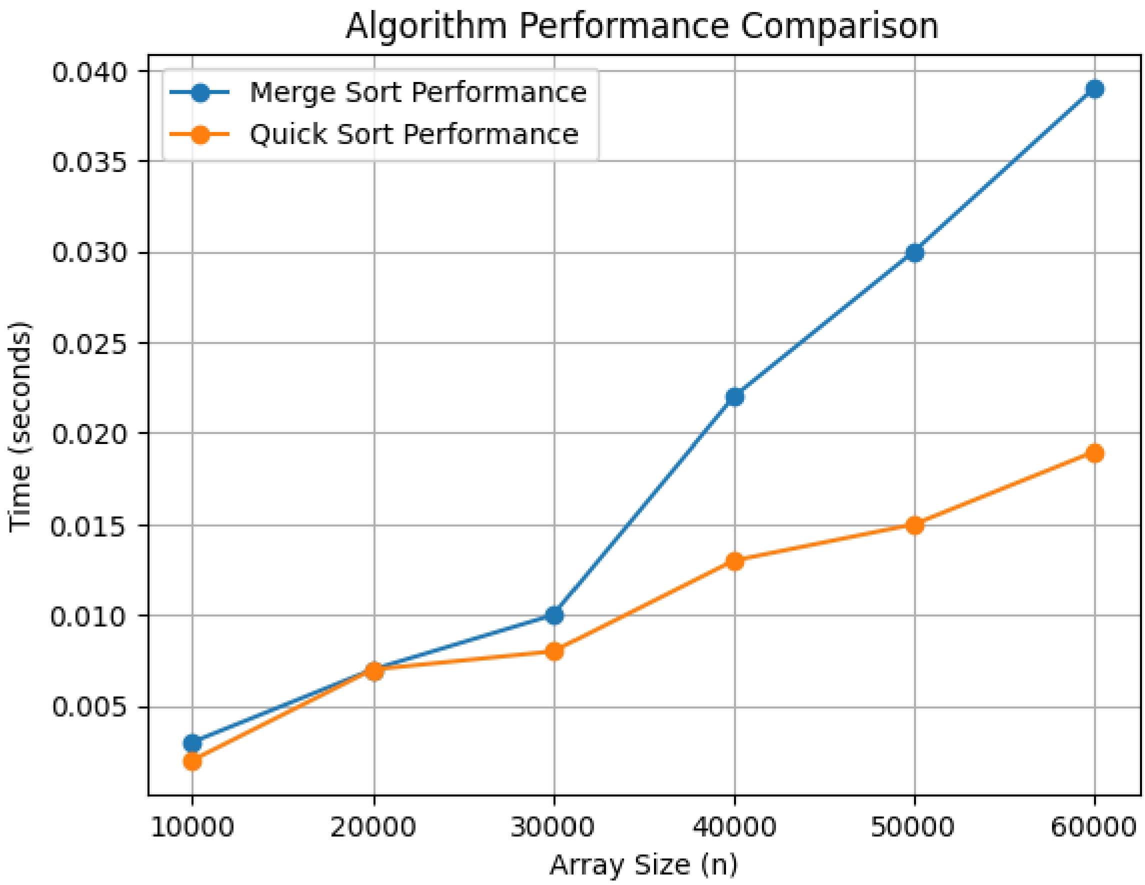
5. N=50000

```
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> gcc Quick_DAC.c
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe

Total elapsed time is 0.015000 seconds
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis>
```

6. N=60000

```
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> gcc Quick_DAC.c
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe

Total elapsed time is 0.019000 seconds
```

GRAPH ANALYSIS:



EXPERIMENT 3

Compare the performance of Strassen method of matrix multiplication with traditional way of matrix multiplication.

CODE:

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

void printMatrix(int** , int );
void fillMatrix(int** , int );
void strassenMultiply(int** , int** , int** , int );
int** allocateMatrix(int );
void freeMatrix(int** , int );
void traditionalMultiply(int** , int** , int** , int );
void addMatrix(int** , int** , int** , int );
void subtractMatrix(int** , int** , int** , int );

int main(){
    int n;
    clock_t end_time,start_time;
    double Traditional_time,Strassen_time;

    printf("Enter the size of the matrix (must be a power of 2): ");
    scanf("%d", &n);

    // Allocate matrices A, B, and C
    int** A = allocateMatrix(n);
    int** B = allocateMatrix(n);
    int** C = allocateMatrix(n);

    // Fill matrices A and B with random numbers
    fillMatrix(A, n);
    fillMatrix(B, n);

    start_time = clock();
    traditionalMultiply(A, B, C, n);
    end_time = clock();
    Traditional_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;

    // Print time taken by  traditional multiplication
    printf("Time taken by Traditional Multiplication: %f seconds\n", Traditional_time);
```

```c
    // Strassen's matrix multiplication
    start_time = clock();
    strassenMultiply(A, B, C, n);
    end_time = clock();
    Strassen_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;

    // Print time taken by Strassen multiplication
    printf("Time taken by Strassen's Multiplication: %f seconds\n", Strassen_time);

    // Free allocated matrices
    freeMatrix(A, n);
    freeMatrix(B, n);
    freeMatrix(C, n);

    return 0;
}

// Function to allocate a matrix of size n x n
int** allocateMatrix(int n) {
    int** matrix = (int**)malloc(n * sizeof(int*));
    for (int i = 0; i < n; i++) {
        matrix[i] = (int*)malloc(n * sizeof(int));
    }
    return matrix;
}

// Function to free the allocated matrix
void freeMatrix(int** matrix, int n) {
    for (int i = 0; i < n; i++) {
        free(matrix[i]);
    }
    free(matrix);
}
```

```c
// Traditional matrix multiplication
void traditionalMultiply(int** A, int** B, int** C, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            C[i][j] = 0;
            for (int k = 0; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

// Add two matrices
void addMatrix(int** A, int** B, int** result, int size) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            result[i][j] = A[i][j] + B[i][j];
        }
    }
}

// Subtract two matrices
void subtractMatrix(int** A, int** B, int** result, int size) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            result[i][j] = A[i][j] - B[i][j];
        }
    }
}

void strassenMultiply(int** A, int** B, int** C, int n) {
    if (n <= 64) {  // Base case: use traditional method for small matrices
        traditionalMultiply(A, B, C, n);
        return;
    }
```

```
    int newSize = n / 2;

    int** A11 = allocateMatrix(newSize);
    int** A12 = allocateMatrix(newSize);
    int** A21 = allocateMatrix(newSize);
    int** A22 = allocateMatrix(newSize);
    int** B11 = allocateMatrix(newSize);
    int** B12 = allocateMatrix(newSize);
    int** B21 = allocateMatrix(newSize);
    int** B22 = allocateMatrix(newSize);

    int** P1 = allocateMatrix(newSize);
    int** P2 = allocateMatrix(newSize);
    int** P3 = allocateMatrix(newSize);
    int** P4 = allocateMatrix(newSize);
    int** P5 = allocateMatrix(newSize);
    int** P6 = allocateMatrix(newSize);
    int** P7 = allocateMatrix(newSize);

    int** C11 = allocateMatrix(newSize);
    int** C12 = allocateMatrix(newSize);
    int** C21 = allocateMatrix(newSize);
    int** C22 = allocateMatrix(newSize);

    int** tempA = allocateMatrix(newSize);
    int** tempB = allocateMatrix(newSize);

    // Dividing matrices into 4 sub-matrices
    for (int i = 0; i < newSize; i++) {
        for (int j = 0; j < newSize; j++) {
            A11[i][j] = A[i][j];
            A12[i][j] = A[i][j + newSize];
            A21[i][j] = A[i + newSize][j];
            A22[i][j] = A[i + newSize][j + newSize];
```

```
            B11[i][j] = B[i][j];
            B12[i][j] = B[i][j + newSize];
            B21[i][j] = B[i + newSize][j];
            B22[i][j] = B[i + newSize][j + newSize];
        }
    }

    // Calculate P1 to P7
    addMatrix(A11, A22, tempA, newSize);
    addMatrix(B11, B22, tempB, newSize);
    strassenMultiply(tempA, tempB, P1, newSize);  // P1 = (A11 + A22) * (B11 + B22)

    addMatrix(A21, A22, tempA, newSize);
    strassenMultiply(tempA, B11, P2, newSize);  // P2 = (A21 + A22) * B11

    subtractMatrix(B12, B22, tempB, newSize);
    strassenMultiply(A11, tempB, P3, newSize);  // P3 = A11 * (B12 - B22)

    subtractMatrix(B21, B11, tempB, newSize);
    strassenMultiply(A22, tempB, P4, newSize);  // P4 = A22 * (B21 - B11)

    addMatrix(A11, A12, tempA, newSize);
    strassenMultiply(tempA, B22, P5, newSize);  // P5 = (A11 + A12) * B22

    subtractMatrix(A21, A11, tempA, newSize);
    addMatrix(B11, B12, tempB, newSize);
    strassenMultiply(tempA, tempB, P6, newSize);  // P6 = (A21 - A11) * (B11 + B12)

    subtractMatrix(A12, A22, tempA, newSize);
    addMatrix(B21, B22, tempB, newSize);
    strassenMultiply(tempA, tempB, P7, newSize);  // P7 = (A12 - A22) * (B21 + B22)
```

```
    // Calculate C11, C12, C21, C22
    addMatrix(P1, P4, tempA, newSize);
    subtractMatrix(tempA, P5, tempB, newSize);
    addMatrix(tempB, P7, C11, newSize);  // C11 = P1 + P4 - P5 + P7

    addMatrix(P3, P5, C12, newSize);  // C12 = P3 + P5

    addMatrix(P2, P4, C21, newSize);  // C21 = P2 + P4

    addMatrix(P1, P3, tempA, newSize);
    subtractMatrix(tempA, P2, tempB, newSize);
    addMatrix(tempB, P6, C22, newSize);  // C22 = P1 + P3 - P2 + P6

    // Grouping into C
    for (int i = 0; i < newSize; i++) {
        for (int j = 0; j < newSize; j++) {
            C[i][j] = C11[i][j];
            C[i][j + newSize] = C12[i][j];
            C[i + newSize][j] = C21[i][j];
            C[i + newSize][j + newSize] = C22[i][j];
        }
    }

    // Free allocated memory
    freeMatrix(A11, newSize); freeMatrix(A12, newSize);
    freeMatrix(A21, newSize); freeMatrix(A22, newSize);
    freeMatrix(B11, newSize); freeMatrix(B12, newSize);
    freeMatrix(B21, newSize); freeMatrix(B22, newSize);
    freeMatrix(P1, newSize); freeMatrix(P2, newSize);
    freeMatrix(P3, newSize); freeMatrix(P4, newSize);
    freeMatrix(P5, newSize); freeMatrix(P6, newSize);
    freeMatrix(P7, newSize);
    freeMatrix(C11, newSize); freeMatrix(C12, newSize);
    freeMatrix(C21, newSize); freeMatrix(C22, newSize);
    freeMatrix(tempA, newSize); freeMatrix(tempB, newSize);
```

```
}


// Function to fill a matrix with random integers
void fillMatrix(int** matrix, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            matrix[i][j] = rand() % 10;
        }
    }
}

```

OUTPUTS:

1.

```
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> gcc Matrix_MUL.c
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe
Enter the size of the matrix (must be a power of 2): 2
Time taken by Traditional Multiplication: 0.000000 seconds
Time taken by Strassen's Multiplication: 0.000000 seconds
```

2.

```
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe
Enter the size of the matrix (must be a power of 2): 4
Time taken by Traditional Multiplication: 0.000000 seconds
Time taken by Strassen's Multiplication: 0.000000 seconds
```

3.

```
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe
Enter the size of the matrix (must be a power of 2): 8
Time taken by Traditional Multiplication: 0.000000 seconds
Time taken by Strassen's Multiplication: 0.000000 seconds
```

4.

```
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe
Enter the size of the matrix (must be a power of 2): 16
Time taken by Traditional Multiplication: 0.000000 seconds
Time taken by Strassen's Multiplication: 0.000000 seconds
```

5.

```
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe
Enter the size of the matrix (must be a power of 2): 32
Time taken by Traditional Multiplication: 0.000000 seconds
Time taken by Strassen's Multiplication: 0.000000 seconds
```

6.

```
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe
Enter the size of the matrix (must be a power of 2): 64
Time taken by Traditional Multiplication: 0.000000 seconds
Time taken by Strassen's Multiplication: 0.000000 seconds
```

7.

```
Enter the size of the matrix (must be a power of 2): 128
Time taken by Traditional Multiplication: 0.008000 seconds
Time taken by Strassen's Multiplication: 0.011000 seconds
```

8.

```
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe
Enter the size of the matrix (must be a power of 2): 256
Time taken by Traditional Multiplication: 0.074000 seconds
Time taken by Strassen's Multiplication: 0.048000 seconds
```

9.

```
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe
Enter the size of the matrix (must be a power of 2): 512
Time taken by Traditional Multiplication: 0.477000 seconds
Time taken by Strassen's Multiplication: 0.331000 seconds
```

10.

```
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe
Enter the size of the matrix (must be a power of 2): 1024
Time taken by Traditional Multiplication: 5.734000 seconds
Time taken by Strassen's Multiplication: 2.265000 seconds
```

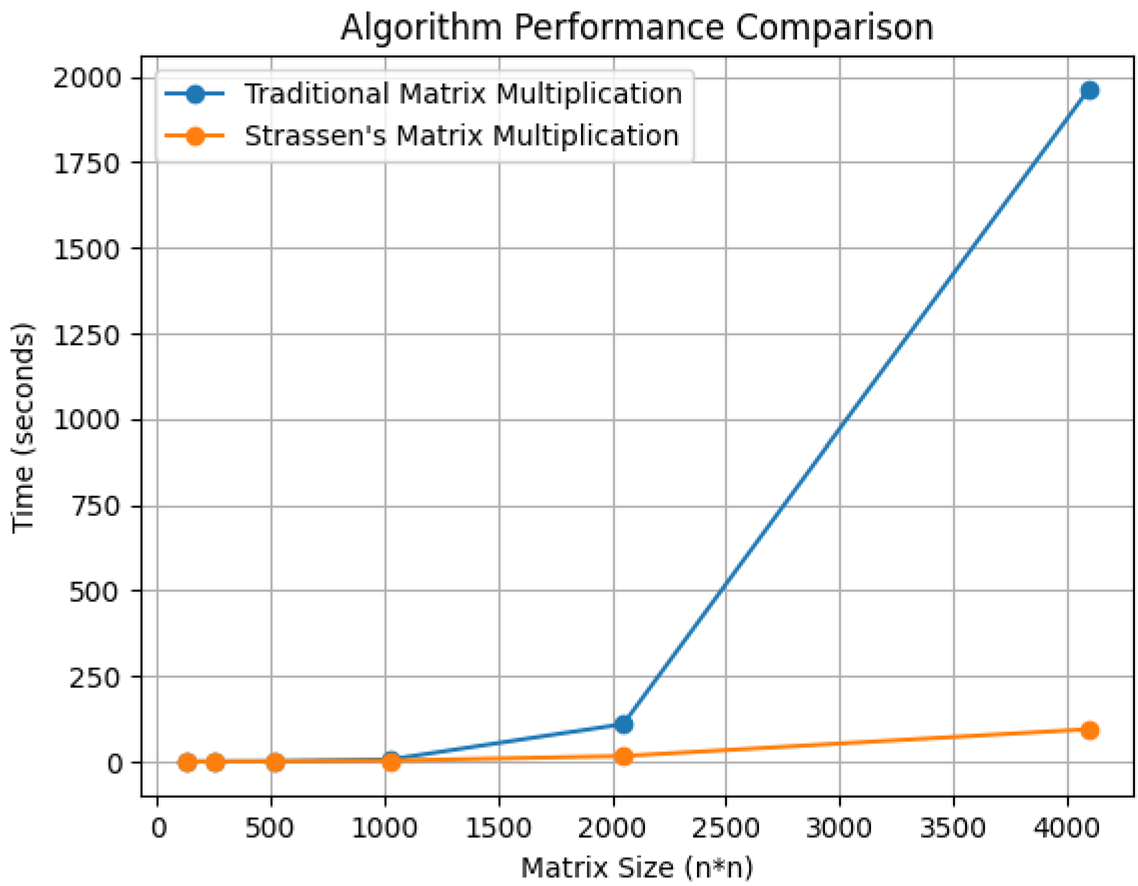
11.

```
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe
Enter the size of the matrix (must be a power of 2): 2048
Time taken by Traditional Multiplication: 109.054000 seconds
Time taken by Strassen's Multiplication: 15.850000 seconds
```

12.

```
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe
Enter the size of the matrix (must be a power of 2): 4096
Time taken by Traditional Multiplication: 1962.552000 seconds
Time taken by Strassen's Multiplication: 94.159000 seconds
```

GRAPH ANALYSIS:



EXPERIMENT 4

Implement the activity selection problem to get a clear understanding of greedy approach.

CODE:

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

// Structure to represent an activity
struct Activity {
    int start;
    int finish;
};

int activityCompare(const void* , const void* );
void activitySelection(struct Activity activities[] , int );

int main(){

    // Example activities (start and finish times)
    struct Activity activities[] = {
    {0, 3},
    {2, 4},
    {1, 5},
    {6, 7},
    {5, 9},
    {8, 11}
    };

    int n = sizeof(activities) / sizeof(activities[0]);

    // Call the activity selection function
    printf("\nSelected activities:\n");
    activitySelection(activities, n);


    return 0;
}
```

```c
// Function to compare two activities based on their finish times
int activityCompare(const void* a, const void* b) {
    struct Activity* activityA = (struct Activity*)a;
    struct Activity* activityB = (struct Activity*)b;
    return activityA->finish - activityB->finish;
}

// Function to select the maximum number of activities
void activitySelection(struct Activity activities[], int n) {
    // Sort activities based on their finish time
    qsort(activities, n, sizeof(activities[0]), activityCompare);

    printf("Selected activities based on indices: \n");

    // The first activity always gets selected
    int i = 0;
    printf("%d (start: %d, finish: %d)\n", i, activities[i].start, activities[i].finish);

    // Consider the rest of the activities
    for (int j = 1; j < n; j++) {
        // If this activity's start time is greater than or equal to
        // the finish time of the last selected activity, select it
        if (activities[j].start >= activities[i].finish) {
            printf("%d (start: %d, finish: %d)\n", j, activities[j].start, activities[j].finish);
            i = j; // Update the last selected activity
        }
    }
}
```

OUTPUTS:

1.

```c
int main(){

    // Example activities (start and finish times)
    struct Activity activities[] = {
    {1, 4},
    {3, 5},
    {0, 6},
    {4, 7},
    {3, 8},
    {5, 9},
    {6, 10},
    {8, 11},
    {8, 12},
    {2, 13},
    {12, 14}
    };
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
10 (start: 12, finish: 14)
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> gcc Activity_Selection_Problem.c
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe

Selected activities:
Selected activities based on indices:
0 (start: 1, finish: 4)
3 (start: 4, finish: 7)
7 (start: 8, finish: 11)
10 (start: 12, finish: 14)
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis>
```

2.

```c
void activitySelection(struct Activity activities[], int);

int main(){

    // Example activities (start and finish times)
    struct Activity activities[] = {
        {1, 4},
        {3, 5},
        {0, 6},
        {5, 7},
        {8, 9}
    };

    int n = sizeof(activities) / sizeof(activities[0]);

    // Call the activity selection function
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> gcc Activity_Selection_Problem.c
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe


Selected activities:
Selected activities based on indices:
0 (start: 1, finish: 4)
3 (start: 5, finish: 7)
3 (start: 5, finish: 7)
4 (start: 8, finish: 9)
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis>
```

3.

```c
int main(){

    // Example activities (start and finish times)
    struct Activity activities[] = {
    {2, 4},
    {1, 3},
    {5, 9},
    {6, 10},
    {8, 11},
    {12, 16}
    };

    int n = sizeof(activities) / sizeof(activities[0]);

    // Call the activity selection function
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> gcc Activity_Selection_Problem.c
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe

Selected activities:
Selected activities based on indices:
0 (start: 1, finish: 3)
2 (start: 5, finish: 9)
5 (start: 12, finish: 16)
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis>
```

4.

```c
int main(){

    // Example activities (start and finish times)
    struct Activity activities[] = {
    {0, 3},
    {2, 4},
    {1, 5},
    {6, 7},
    {5, 9},
    {8, 11}
    };

    int n = sizeof(activities) / sizeof(activities[0]);

    // Call the activity selection function
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
^C
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> gcc Activity_Selection_Problem.c
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe

Selected activities:
Selected activities based on indices:
0 (start: 0, finish: 3)
3 (start: 6, finish: 7)
5 (start: 8, finish: 11)
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis>
```

EXPERIMENT 5

Get a detailed insight of dynamic programming approach by the implementation of Matrix Chain Multiplication problem and see the impact of parenthesis positioning on time requirements for matrix multiplication.

CODE:

```c
#include <stdio.h>
#include <limits.h>

void print_optimal_parens(int , int , int n, int s[n][n], char *);
void matrix_chain_order(int p[], int );


// Function to print the optimal parenthesization
void print_optimal_parens(int i, int j, int n, int s[n][n], char *name) {
    if (i == j) {
        printf("A%c", *name);  // Print matrix name (e.g., A1, A2, etc.)
        (*name)++;
        return;
    }
    printf("(");
    print_optimal_parens(i, s[i][j], n, s, name);
    print_optimal_parens(s[i][j] + 1, j, n, s, name);
    printf(")");
}

// Function to find the minimum cost of matrix chain multiplication
void matrix_chain_order(int p[], int n) {
    int m[n][n];  // Table to store minimum multiplications
    int s[n][n];  // Table to store split points

    // Initialize number of multiplications for a single matrix as 0
    for (int i = 1; i < n; i++)
        m[i][i] = 0;

    // L is the chain length
    for (int L = 2; L < n; L++) {
        for (int i = 1; i < n - L + 1; i++) {
            int j = i + L - 1;
            m[i][j] = INT_MAX;  // Initialize with a large value

            // Test all positions to split the product
            for (int k = i; k <= j - 1; k++) {
```

```c
                // Calculate cost of scalar multiplications
                int q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];

                // Update minimum cost and store split point
                if (q < m[i][j]) {
                    m[i][j] = q;
                    s[i][j] = k;
                }
            }
        }
    }

    // Output the minimum number of scalar multiplications
    printf("Minimum number of multiplications is: %d\n", m[1][n - 1]);

    // Output the optimal parenthesization
    printf("Optimal parenthesization: ");
    char name = '1';  // Start naming matrices as A1, A2, ...
    print_optimal_parens(1, n - 1, n, s, &name);
    printf("\n");
}

// Main function
int main() {
    // Matrix dimensions: A1(30x35), A2(35x15), A3(15x5), A4(5x10), A5(10x20), A6(20x25)
    int p[] = {30, 35, 15, 5, 10, 20, 25};  // Array of matrix dimensions
    int n = sizeof(p) / sizeof(p[0]);  // Number of matrices is n-1

    // Call the function to calculate the minimum multiplications and print the result
    matrix_chain_order(p, n);

    return 0;
}
```

OUTPUT:

C > Algorithm Analysis > C Matrix_Chain_MUL.c > print_optimal_parens(int, int, int, int [n][n], char *)

```c
#include <stdio.h>
#include <limits.h>

void print_optimal_parens(int , int , int n, int s[n][n], char *);
void matrix_chain_order(int p[], int );


// Function to print the optimal parenthesization
void print_optimal_parens(int i, int j, int n, int s[n][n], char *name) {
    if (i == j) {
        printf("A%c", *name);  // Print matrix name (e.g., A1, A2, etc.)
        (*name)++;
        return;
    }
    printf("(");
    print_optimal_parens(i, s[i][j], n, s, name);
    print_optimal_parens(s[i][j] + 1, j, n, s, name);
    printf(")");
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS JUPYTER

```
except the first
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> gcc Matrix_Chain_MUL.c
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe
Minimum number of multiplications is: 15125
Optimal parenthesization: ((A1(A2A3))((A4A5)A6))
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> gcc Matrix_Chain_MUL.c
PS C:\Users\User\Desktop\.vscode\C\Algorithm Analysis> .\a.exe
Minimum number of multiplications is: 15125
Optimal parenthesization: ((A1(A2A3))((A4A5)A6))
```

EXPERIMENT 6

Compare the performance of Dijkstra and Bellman ford algorithm for the single source shortest path problem.

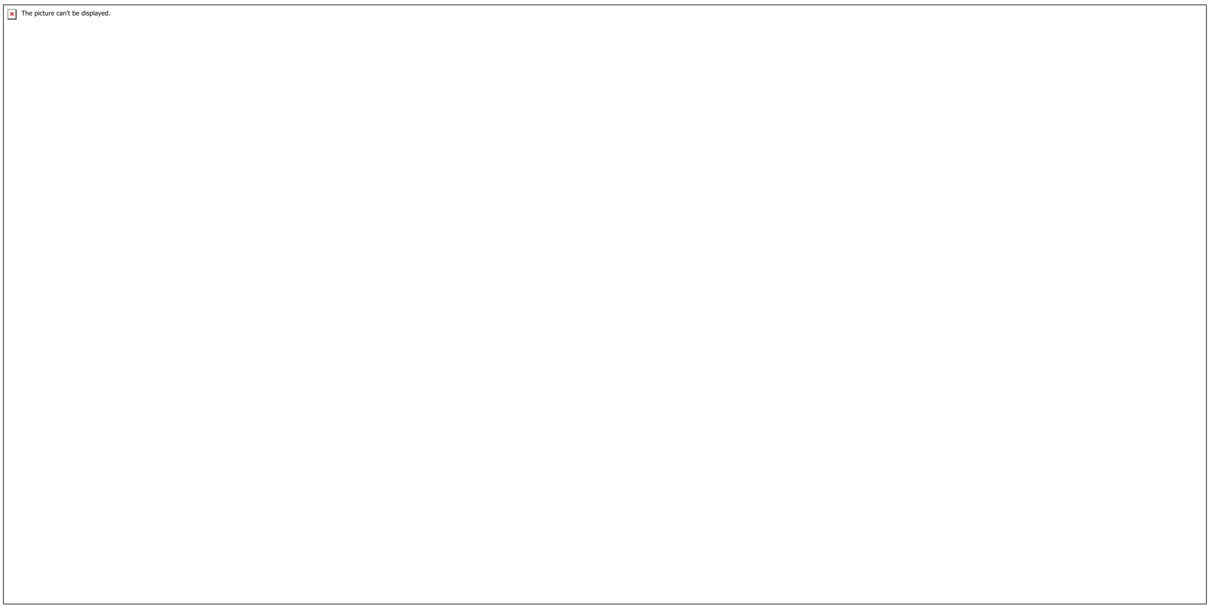
CODE:

```c
#include <stdio.h>
#include <limits.h>
#include <stdbool.h>
#include <time.h>

#define INF INT_MAX
#define REPEAT 10000  // Increase number of repetitions for better timing
int minDistance(int dist[], bool sptSet[], int V);
void dijkstra(int graph[20][20], int src, int V);
void bellmanFord(int graph[20][3], int V, int E, int src);
double calculateExecutionTimeDijkstra(void (*func)(int[][20], int, int), int graph[20][20], int src, int V);
double calculateExecutionTimeBellmanFord(void (*func)(int[][3], int, int, int), int graph[20][3], int V, int E, int src);

// Function to find the vertex with the minimum distance
int minDistance(int dist[], bool sptSet[], int V) {
    int min = INF, min_index;
    for (int v = 0; v < V; v++)
        if (!sptSet[v] && dist[v] <= min) {
            min = dist[v];
            min_index = v;
        }
    return min_index;
}

// Dijkstra's algorithm
void dijkstra(int graph[20][20], int src, int V) {
    int dist[V];
    bool sptSet[V];

    for (int i = 0; i < V; i++) {
        dist[i] = INF;
        sptSet[i] = false;
    }
    dist[src] = 0;

    for (int count = 0; count < V - 1; count++) {
        int u = minDistance(dist, sptSet, V);
```

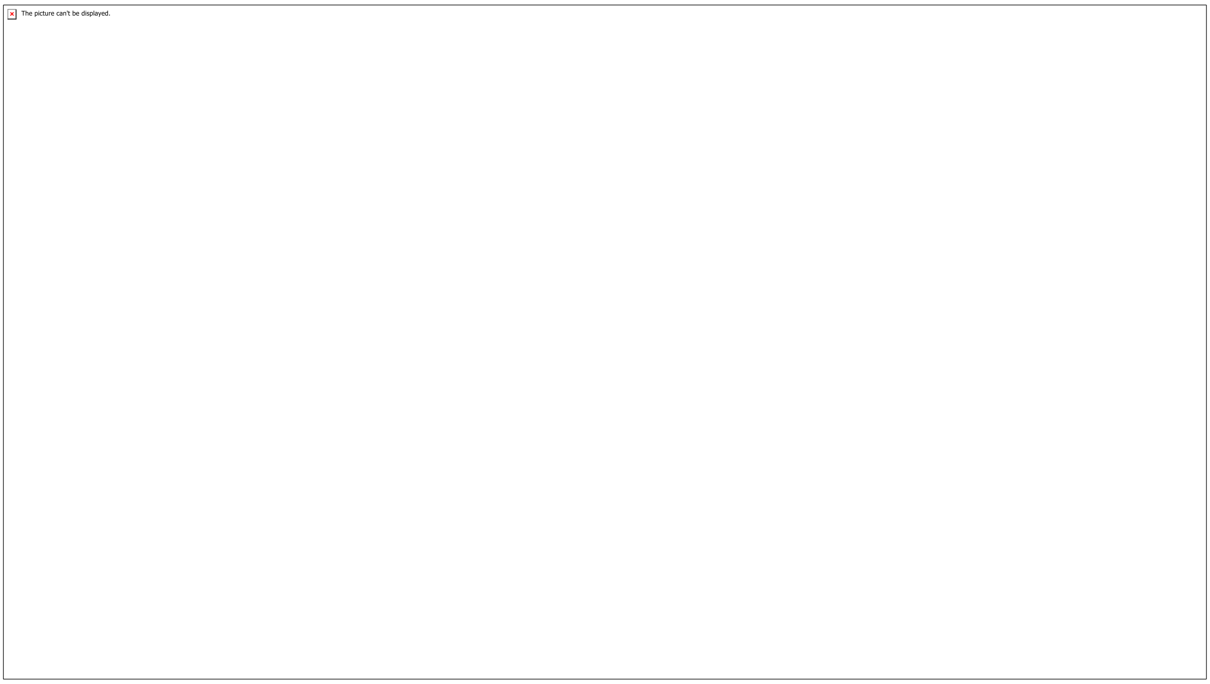
```c
        sptSet[u] = true;

        for (int v = 0; v < V; v++)
            if (!sptSet[v] && graph[u][v] && dist[u] != INF && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }
}

// Bellman-Ford algorithm
void bellmanFord(int graph[20][3], int V, int E, int src) {
    int dist[V];
    for (int i = 0; i < V; i++)
        dist[i] = INF;
    dist[src] = 0;

    for (int i = 1; i <= V - 1; i++) {
        for (int j = 0; j < E; j++) {
            int u = graph[j][0];
            int v = graph[j][1];
            int weight = graph[j][2];
            if (dist[u] != INF && dist[u] + weight < dist[v])
                dist[v] = dist[u] + weight;
        }
    }
}

// Utility function to calculate execution time for Dijkstra
double calculateExecutionTimeDijkstra(void (*func)(int[][20], int, int), int graph[20][20], int src, int V) {
    clock_t start, end;
    start = clock();
    for (int i = 0; i < REPEAT; i++) {  // Repeat the algorithm
        func(graph, src, V);
    }
    end = clock();
```

```c
    return ((double)(end - start)) / CLOCKS_PER_SEC * 1000 / REPEAT; // Average Time in ms
}

// Utility function to calculate execution time for Bellman-Ford
double calculateExecutionTimeBellmanFord(void (*func)(int[][3], int, int, int), int graph[20][3], int V, int E, int src) {
    clock_t start, end;
    start = clock();
    for (int i = 0; i < REPEAT; i++) {  // Repeat the algorithm
        func(graph, V, E, src);
    }
    end = clock();
    return ((double)(end - start)) / CLOCKS_PER_SEC * 1000 / REPEAT; // Average Time in ms
}

// Test function
void compareAlgorithms() {
    // Graphs input data for Dijkstra
    int graph1[20][20] = {
        {0, 4, 1, 0},
        {0, 0, 2, 5},
        {0, 0, 0, 3},
        {0, 0, 0, 0}
    };

    int graph2[20][20] = {
        {0, 3, 2, 0, 0, 0},
        {0, 0, 0, 7, 4, 0},
        {0, 0, 0, 1, 0, 5},
        {0, 0, 0, 0, 0, 2},
        {0, 0, 0, 0, 0, 1},
        {0, 0, 0, 0, 0, 0}
    };

```

```c
    int graph3[20][20] = {
        {0, 6, 2, 5, 0, 0, 0, 0},
        {0, 0, 0, 1, 4, 0, 0, 0},
        {0, 0, 0, 0, 0, 5, 0, 0},
        {0, 0, 0, 0, 0, 0, 3, 0},
        {0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 1, 0},
        {0, 0, 0, 0, 0, 0, 0, 2},
        {0, 0, 0, 0, 0, 0, 0, 0}
    };

    // Graphs input data for Bellman-Ford (converted to edge list)
    int graph1_edges[20][3] = {
        {0, 1, 4}, {0, 2, 1}, {1, 2, 2}, {1, 3, 5}, {2, 3, 3}
    };
    int graph2_edges[20][3] = {
        {0, 1, 3}, {0, 2, 2}, {1, 3, 7}, {1, 4, 4}, {2, 3, 1}, {2, 5, 5}, {3, 5, 2}, {4, 5, 1}
    };
    int graph3_edges[20][3] = {
        {0, 1, 6}, {0, 2, 2}, {0, 3, 5}, {1, 3, 1}, {1, 4, 4}, {2, 5, 5}, {2, 4, 5}, {3, 6, 3}, {4, 5, 3}, {5, 6, 1}, {5, 7, 4}, {6, 7, 2}
    };

    // Time to run Dijkstra
    double dijkstra_time_1 = calculateExecutionTimeDijkstra(dijkstra, graph1, 0, 4);
    double dijkstra_time_2 = calculateExecutionTimeDijkstra(dijkstra, graph2, 0, 6);
    double dijkstra_time_3 = calculateExecutionTimeDijkstra(dijkstra, graph3, 0, 8);

    // Time to run Bellman-Ford
    double bellman_time_1 = calculateExecutionTimeBellmanFord(bellmanFord, graph1_edges, 4, 5, 0);
    double bellman_time_2 = calculateExecutionTimeBellmanFord(bellmanFord, graph2_edges, 6, 8, 0);
    double bellman_time_3 = calculateExecutionTimeBellmanFord(bellmanFord, graph3_edges, 8, 12, 0);

```


The picture can't be displayed.

OUTPUT:


The picture can't be displayed.

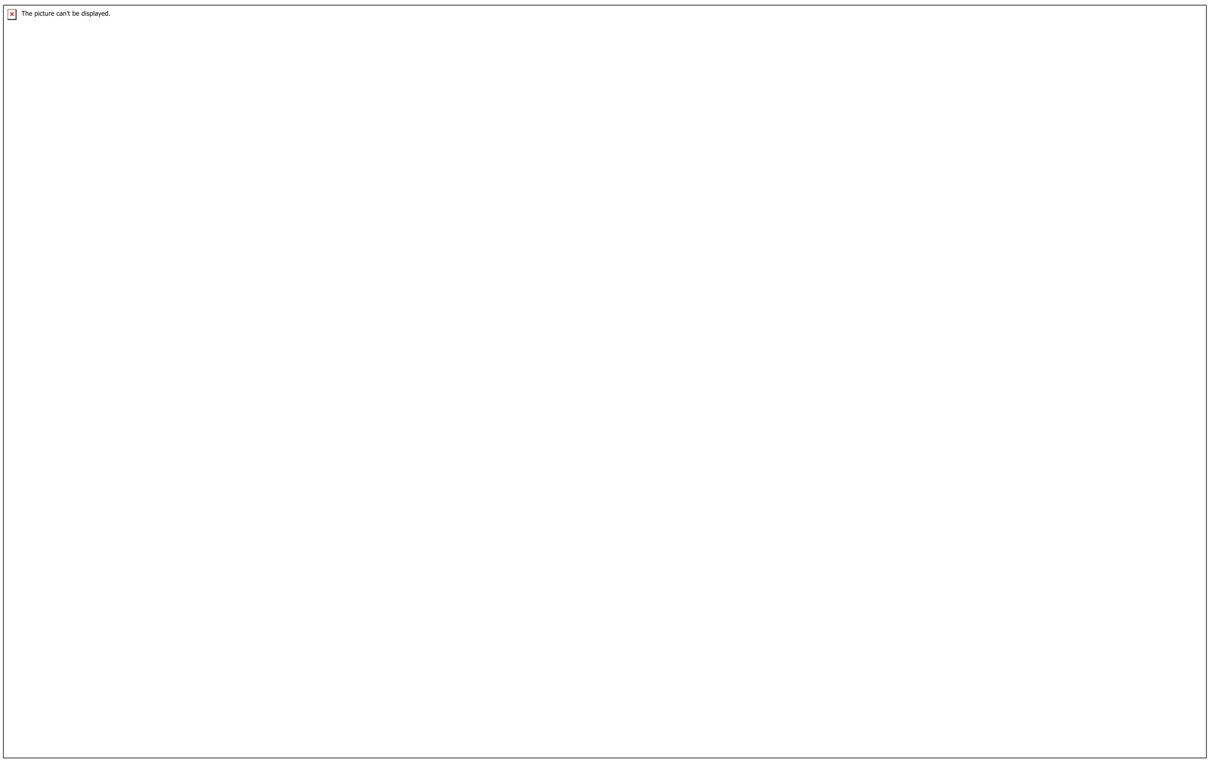
GRAPH ANALYSIS:

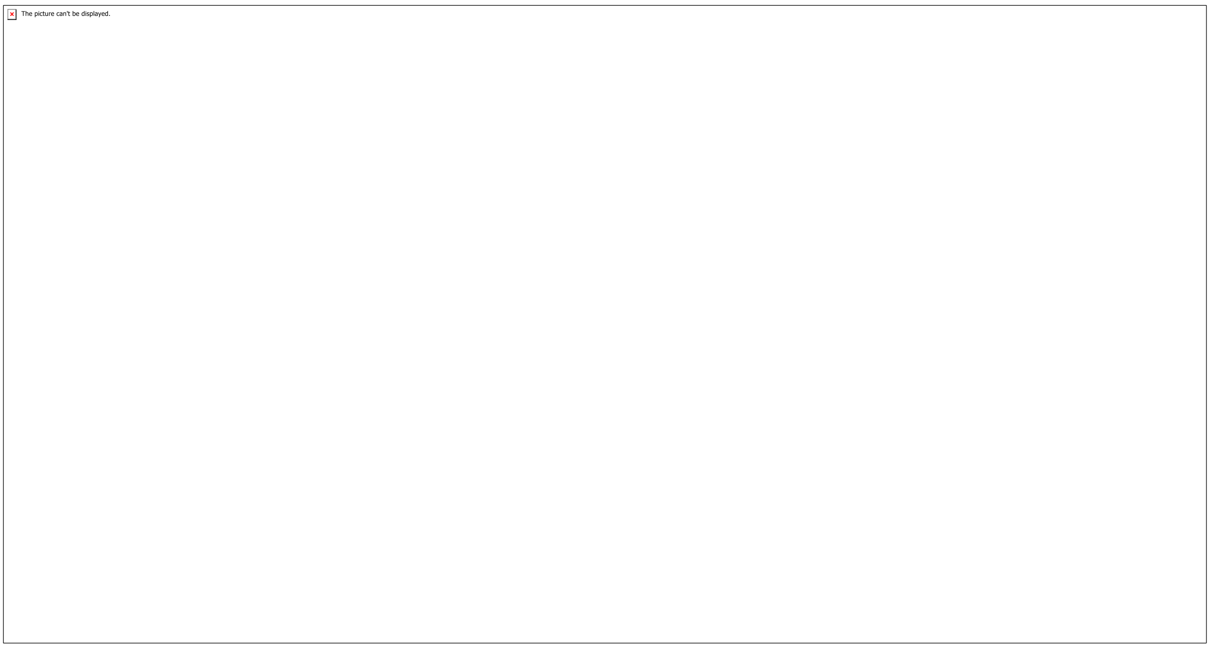
The picture can't be displayed.

EXPERIMENT 7

Through 0/1 Knapsack problem, analyse the greedy and dynamic programming approach for the same dataset.

CODE:





The picture can't be displayed.

The picture can't be displayed.

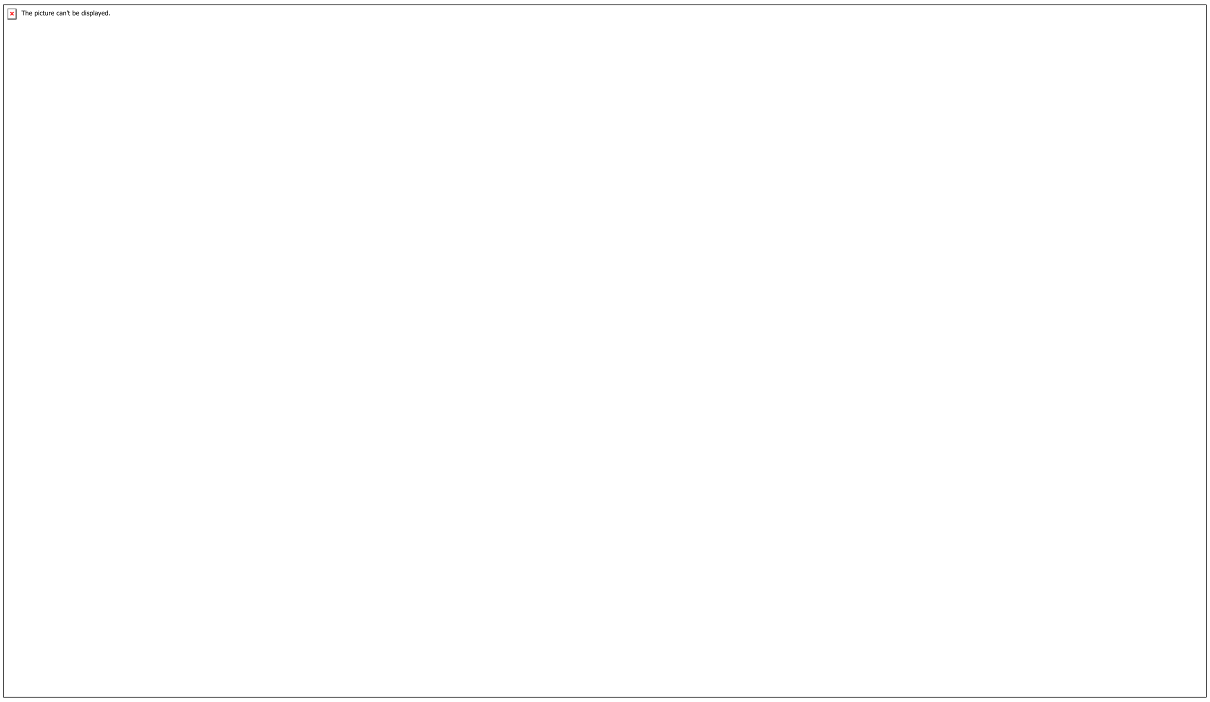
The picture can't be displayed.

The picture can't be displayed.

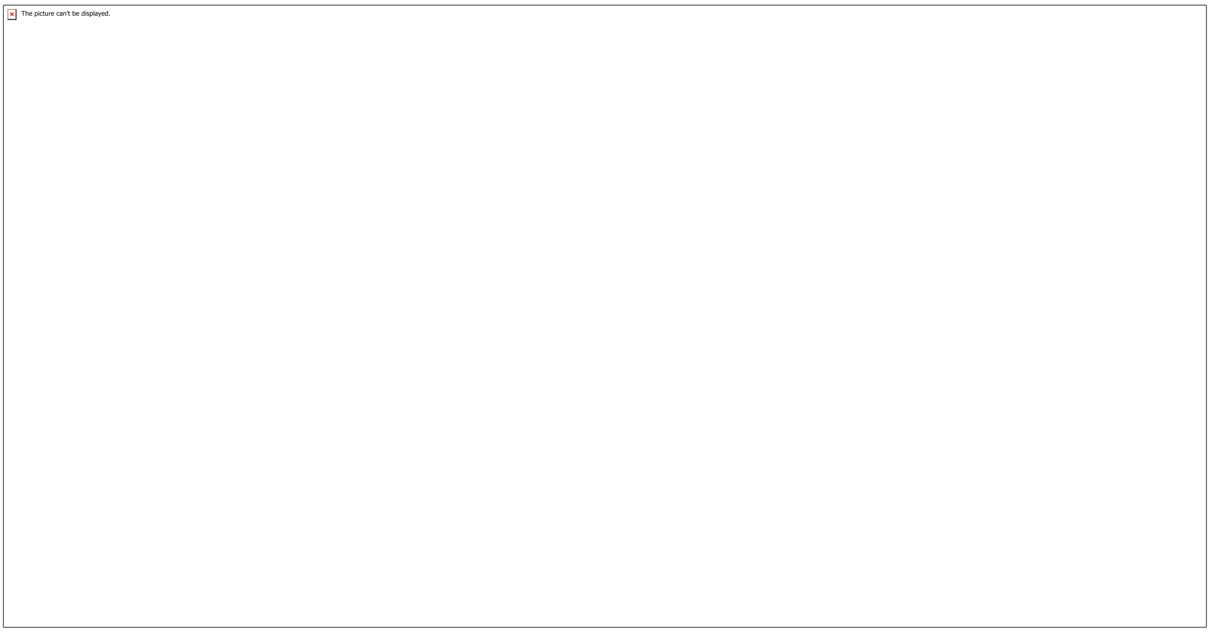
The picture can't be displayed.

<u>OUTPUTS:</u>

1.



2.



3.

The picture can't be displayed.

4.

The picture can't be displayed.

5.

The picture can't be displayed.

EXPERIMENT 8

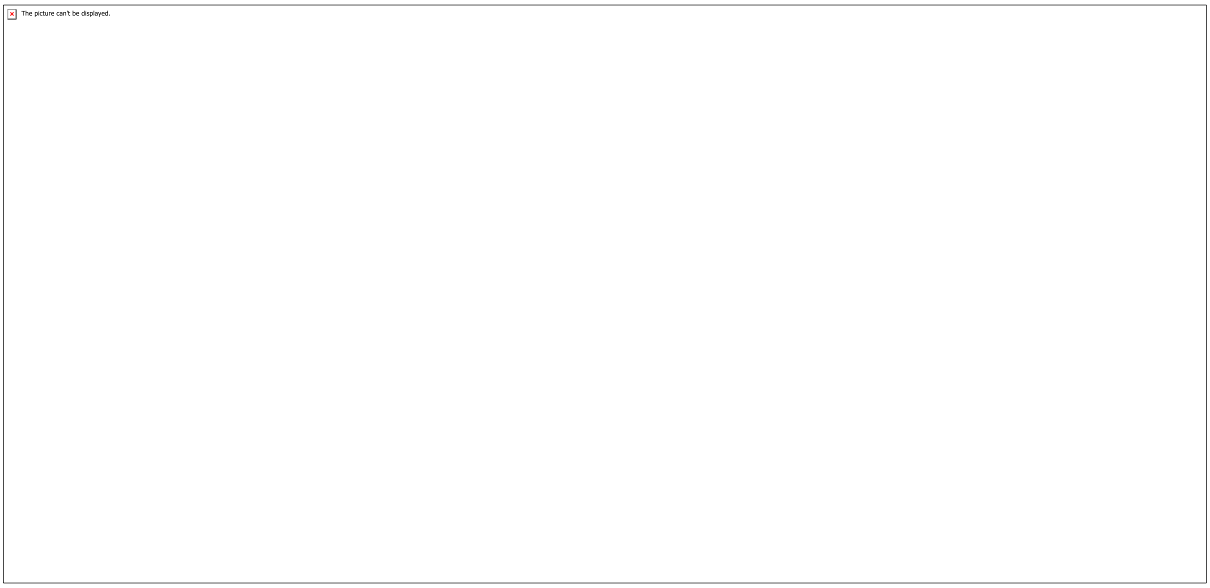
Implement the sum of subset.

CODE:

The picture can't be displayed.

The picture can't be displayed.

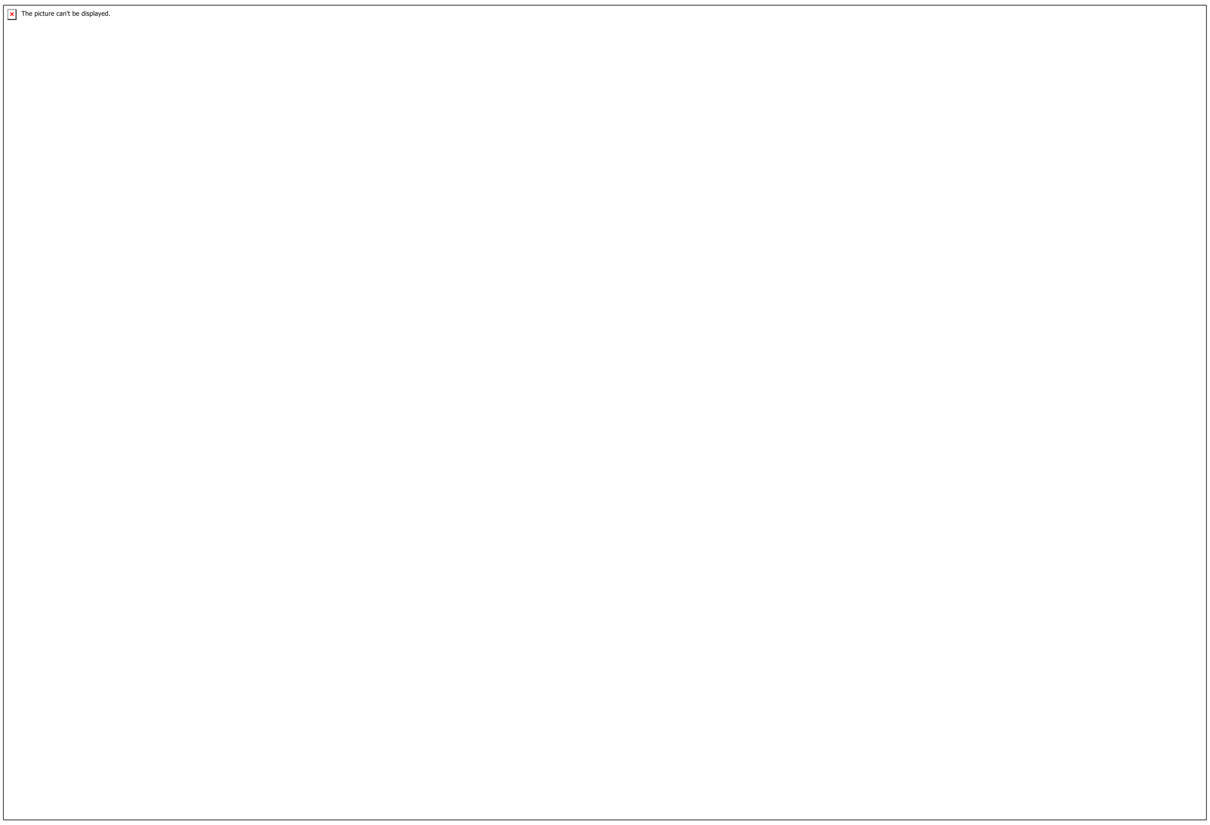
OUTPUT:



EXPERIMENT 9

Compare the Backtracking and Branch & Bound Approach by the implementation of 0/1 Knapsack problem. Also compare the performance with dynamic programming approach.

CODE:



The picture can't be displayed.

The picture can't be displayed.

The picture can't be displayed.

```c
    printf("Dynamic Programming Result: %d, Time: %lf ms\n", result_dp,
           (double)(end - start) * 1000 / (CLOCKS_PER_SEC * repetitions));

    return 0;
}
```

OUTPUTS:

1.

The picture can't be displayed.

2.

The picture can't be displayed.

3.

The picture can't be displayed.

GRAPH ANALYSIS:

The picture can't be displayed.

EXPERIMENT 10

Compare the performance of Rabin-Karp, Knuth-Morris-Pratt and naive string-matching algorithms.

CODE:

The picture can't be displayed.

The picture can't be displayed.

The picture can't be displayed.

The picture can't be displayed.

The picture can't be displayed.

The picture can't be displayed.

OUTPUTS:

1.

The picture can't be displayed.

2.

The picture can't be displayed.

3.

The picture can't be displayed.

4.

The picture can't be displayed.

5.

The picture can't be displayed.

GRAPH ANALYSIS:

The picture can't be displayed.

MY GITHUB LINK:

https://github.com/COBR-A/Algorithm_lab_3_sem_500120575