11/16/2024

DAA LAB SUBMISSION

***Anshul Goel***

***BATCH: 53***

***SAP ID: 500123668***

***ROLL NO: R2142231828***

**Code:**

// 1. **Implement the insertion inside iterative and recursive Binary search tree and compare their performance.**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Structure for a BST node

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

// 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;

}

// Iterative BST insertion

struct Node\* iterativeInsert(struct Node\* root, int data) {

struct Node\* newNode = createNode(data);

if (root == NULL) return newNode;

struct Node\* parent = NULL;

struct Node\* current = root;

while (current != NULL) {

parent = current;

if (data < current->data)

current = current->left;

else if (data > current->data)

current = current->right;

else

return root; // No duplicates

}

if (data < parent->data)

parent->left = newNode;

else

parent->right = newNode;

return root;

}

// Recursive BST insertion

struct Node\* recursiveInsert(struct Node\* root, int data) {

if (root == NULL) return createNode(data);

if (data < root->data)

root->left = recursiveInsert(root->left, data);

else if (data > root->data)

root->right = recursiveInsert(root->right, data);

return root;

}

// Utility function to print BST in-order (for verification)

void inorderTraversal(struct Node\* root) {

if (root != NULL) {

inorderTraversal(root->left);

printf("%d ", root->data);

inorderTraversal(root->right);

}

}

// Time comparison function for both insertions

void compareInsertionTimes(int arrays[5][10], int sizes[5]) {

for (int i = 0; i < 5; i++) {

printf("\n--- Array %d ---\n", i + 1);

struct Node\* root1 = NULL; // For iterative insertions

struct Node\* root2 = NULL; // For recursive insertions

// Measure time for iterative insertion

clock\_t startIter = clock();

for (int j = 0; j < sizes[i]; j++) {

root1 = iterativeInsert(root1, arrays[i][j]);

}

clock\_t endIter = clock();

double timeIter = ((double)(endIter - startIter)) / CLOCKS\_PER\_SEC;

// Measure time for recursive insertion

clock\_t startRecur = clock();

for (int j = 0; j < sizes[i]; j++) {

root2 = recursiveInsert(root2, arrays[i][j]);

}

clock\_t endRecur = clock();

double timeRecur = ((double)(endRecur - startRecur)) / CLOCKS\_PER\_SEC;

printf("Iterative Insertion Time: %f seconds\n", timeIter);

printf("Recursive Insertion Time: %f seconds\n", timeRecur);

// Optional: Print BST (for verification)

printf("In-order traversal (Iterative): ");

inorderTraversal(root1);

printf("\nIn-order traversal (Recursive): ");

inorderTraversal(root2);

printf("\n");

}

}

// Driver function

int main() {

// Define five sample arrays

int arrays[5][10] = {

{50, 30, 20, 40, 70, 60, 80}, // 7 elements

{10, 20, 30, 40, 50, 60, 70, 80, 90}, // 9 elements

{25, 15, 50, 10, 22, 35, 70, 40, 80}, // 9 elements

{100, 90, 80, 70, 60}, // 5 elements

{5, 25, 15, 35, 20, 30, 10} // 7 elements

};

// Define the size of each array

int sizes[5] = {7, 9, 9, 5, 7};

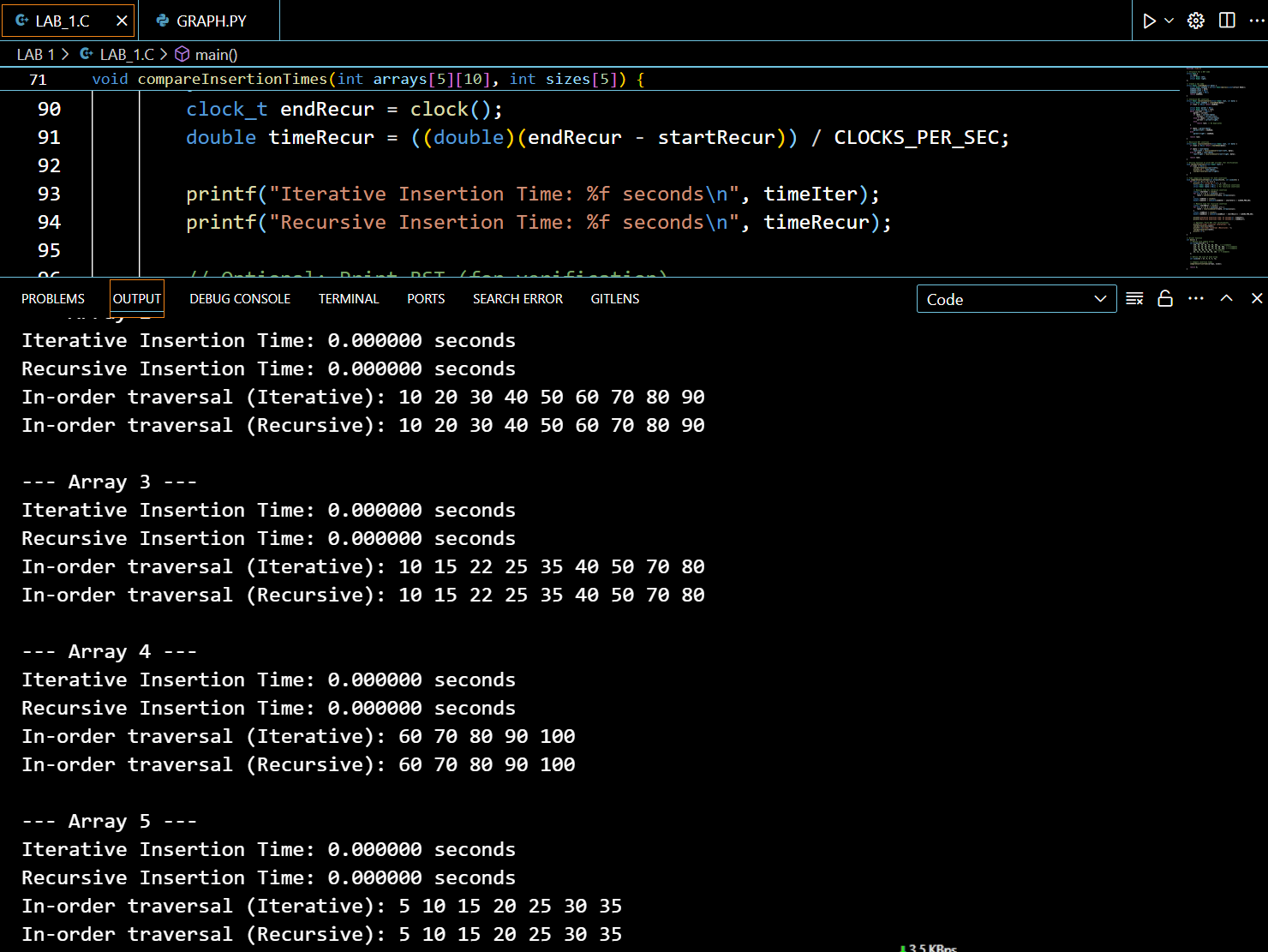
// Compare insertion times

compareInsertionTimes(arrays, sizes);

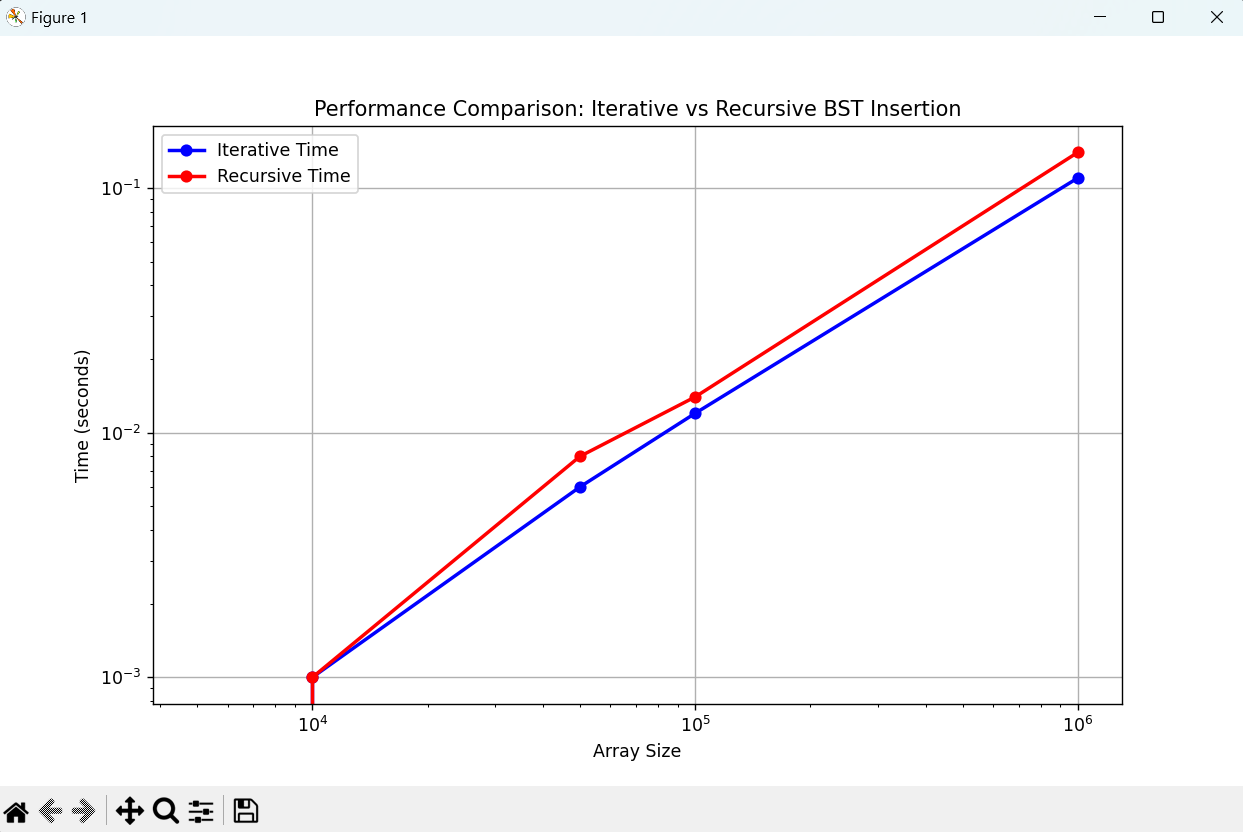
return 0;

}

**Output of the Code:**



**Graph:**



**// 2. Implement divide and conquer based merge sort and quick sort algorithms and compare their performance for the same set of elements.**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <string.h>

**// Merge function for merge sort**

void merge(int arr[], int left, int mid, int right) {

int i, j, k;

int n1 = mid - left + 1;

int n2 = right - mid;

int L[n1], R[n2];

for (i = 0; i < n1; i++)

L[i] = arr[left + i];

for (j = 0; j < n2; j++)

R[j] = arr[mid + 1 + j];

i = 0;

j = 0;

k = left;

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++;

}

}

**// Merge Sort function**

void mergeSort(int arr[], int left, int right) {

if (left < right) {

int mid = left + (right - left) / 2;

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

merge(arr, left, mid, right);

}

}

**// Function to swap two elements**

void swap(int\* a, int\* b) {

int t = \*a;

\*a = \*b;

\*b = t;

}

**// Partition function for quick sort**

int partition(int arr[], int low, int high) {

int pivot = arr[high];

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);

}

// 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);

}

}

**// Function to generate random array**

void generateRandomArray(int arr[], int n) {

for (int i = 0; i < n; i++) {

arr[i] = rand() % 10000; // Random numbers between 0 and 9999

}

}

// Function to measure sorting time

double measureSortingTime(void (\*sortFunction)(int[], int, int), int arr[], int n) {

clock\_t start, end;

double cpu\_time\_used;

int\* arrCopy = (int\*)malloc(n \* sizeof(int));

memcpy(arrCopy, arr, n \* sizeof(int));

start = clock();

sortFunction(arrCopy, 0, n - 1);

end = clock();

cpu\_time\_used = ((double) (end - start)) / CLOCKS\_PER\_SEC;

free(arrCopy);

return cpu\_time\_used;

}

int main() {

srand(time(NULL));

int sizes[] = {1000, 5000, 10000, 50000, 100000};

int num\_sets = sizeof(sizes) / sizeof(sizes[0]);

printf("Set\tSize\tMerge Sort Time\tQuick Sort Time\n");

for (int i = 0; i < num\_sets; i++) {

int n = sizes[i];

int\* arr = (int\*)malloc(n \* sizeof(int));

generateRandomArray(arr, n);

double mergeSortTime = measureSortingTime(mergeSort, arr, n);

double quickSortTime = measureSortingTime(quickSort, arr, n);

printf("%d\t%d\t%.6f\t\t%.6f\n", i+1, n, mergeSortTime, quickSortTime);

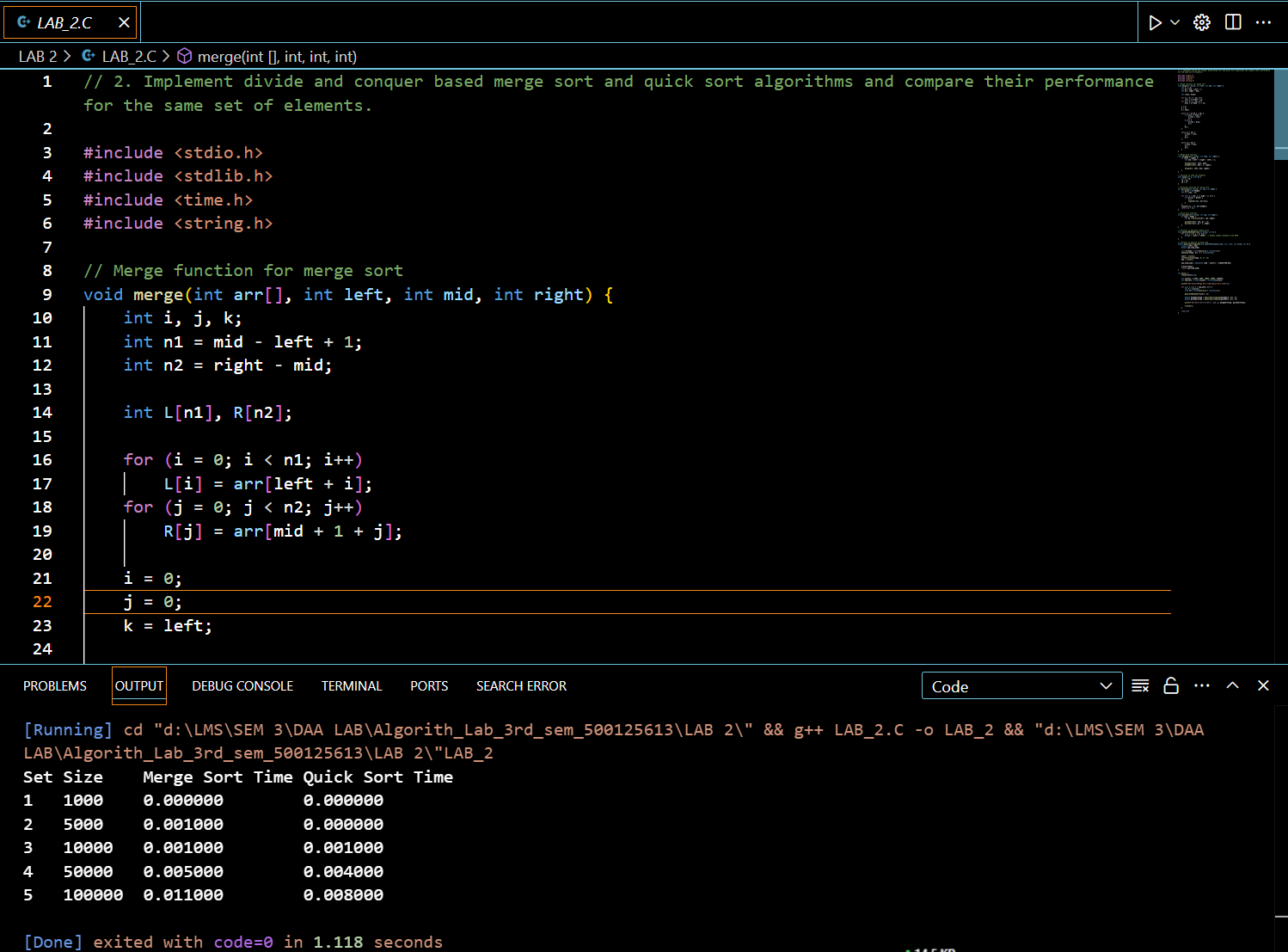
free(arr);

}

return 0;

}

**OUTPUT OF THE CODE**



**Graph :**



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

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Function to allocate memory for a matrix

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 memory of a matrix

void freeMatrix(int\*\* matrix, int n) {

for (int i = 0; i < n; i++) {

free(matrix[i]);

}

free(matrix);

}

// Function to add two matrices

void addMatrix(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] = A[i][j] + B[i][j];

}

}

}

// Function to subtract two matrices

void subtractMatrix(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] = A[i][j] - B[i][j];

}

}

}

// 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];

}

}

}

}

// Strassen's matrix multiplication

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

subtract Matrix(B12, B22, tempB, newSize);

Strassen Multiply(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);

Strassen Multiply(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 measure execution time

double measureExecutionTime(void (\*multiplyFunc)(int\*\*, int\*\*, int\*\*, int), int\*\* A, int\*\* B, int\*\* C, int n) {

clock\_t start, end;

double cpu\_time\_used;

start = clock();

multiplyFunc(A, B, C, n);

end = clock();

cpu\_time\_used = ((double) (end - start)) / CLOCKS\_PER\_SEC;

return cpu\_time\_used;

}

int main() {

srand(time(NULL));

int sizes[] = {64, 128, 256, 512, 1024, 2048};

int num\_sizes = sizeof(sizes) / sizeof(sizes[0]);

printf("Matrix Size\tTraditional Time\tStrassen Time\n");

for (int i = 0; i < num\_sizes; i++) {

int n = sizes[i];

int\*\* A = allocateMatrix(n);

int\*\* B = allocateMatrix(n);

int\*\* C = allocateMatrix(n);

// Initialize matrices A and B with random values

for (int j = 0; j < n; j++) {

for (int k = 0; k < n; k++) {

A[j][k] = rand() % 10;

B[j][k] = rand() % 10;

}

}

double traditionalTime = measureExecutionTime(traditionalMultiply, A, B, C, n);

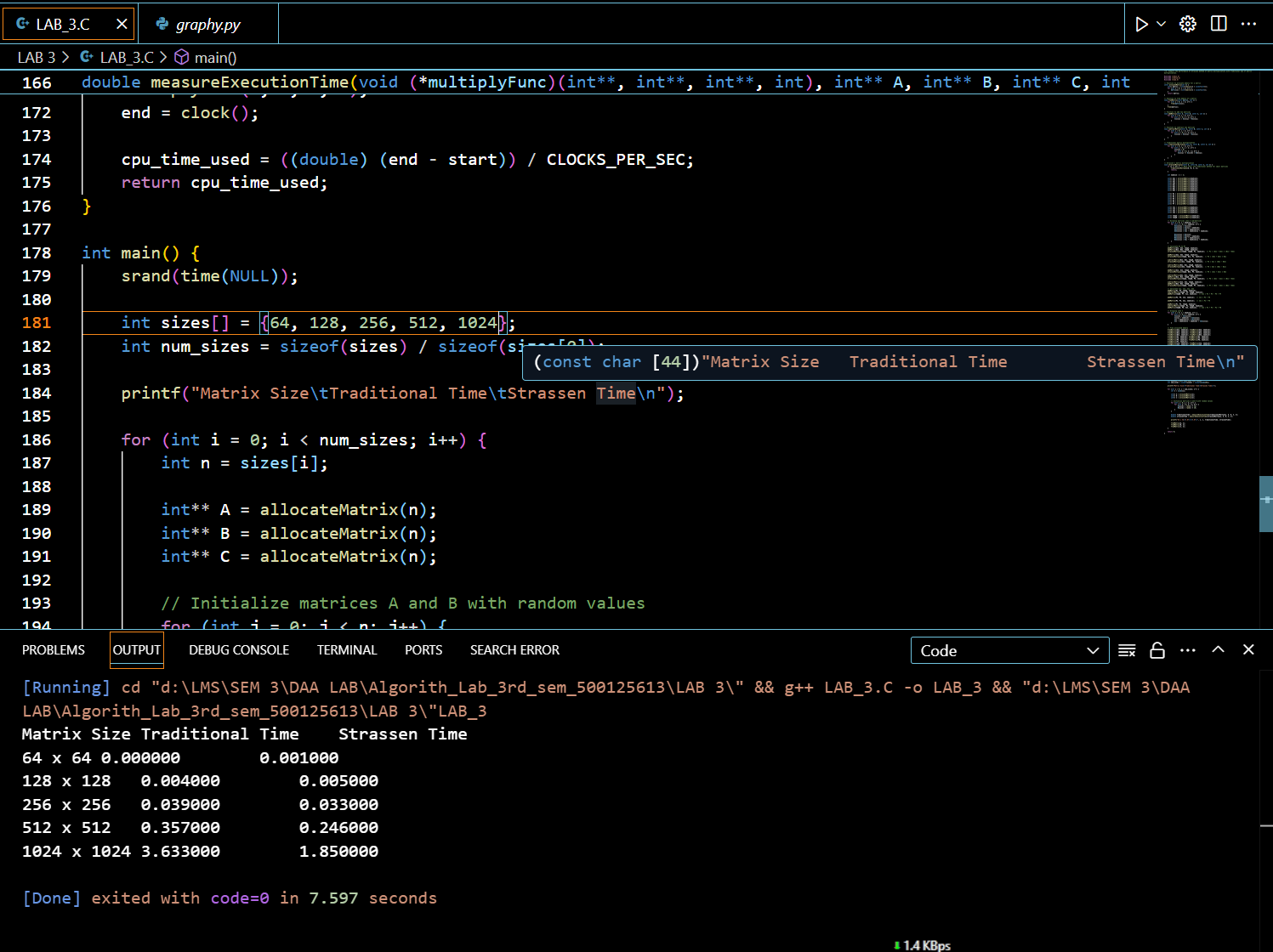
double strassenTime = measureExecutionTime(strassenMultiply, A, B, C, n);

printf("%d x %d\t%.6f\t\t%.6f\n", n, n, traditionalTime, strassenTime);

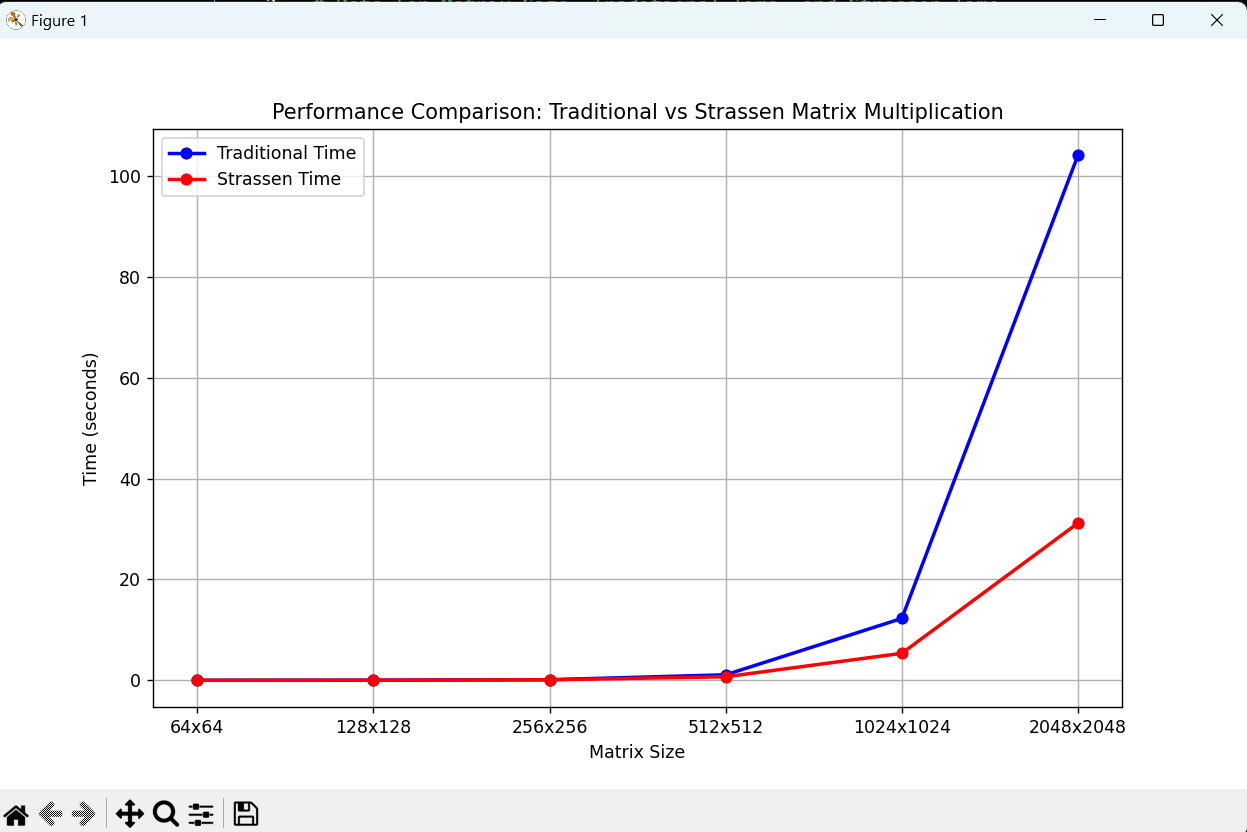
freeMatrix(A, n);

freeMatrix(B, n);

**Output of the code:**



**Graph:**



// 4**. Implement the activity selection problem to get a clear understanding of greedy approach.**

// 4. Implement the activity selection problem to get a clear understanding of greedy approach.

#include <stdio.h>

// Function to print the maximum number of activities that can be done

void activitySelection(int start[], int end[], int n) {

int i, j;

printf("Selected activities are:\n");

// The first activity is always selected

i = 0;

printf("Activity %d (Start: %d, End: %d)\n", i+1, start[i], end[i]);

// Consider rest of the activities

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

// If this activity has a start time greater than or equal to the

// end time of the previously selected activity, select it

if (start[j] >= end[i]) {

printf("Activity %d (Start: %d, End: %d)\n", j+1, start[j], end[j]);

i = j; // Update i to the current activity

}

}

}

int main() {

// Example set of activities with their start and end times

int start[] = {1, 3, 0, 5, 8, 5};

int end[] = {2, 4, 6, 7, 9, 9};

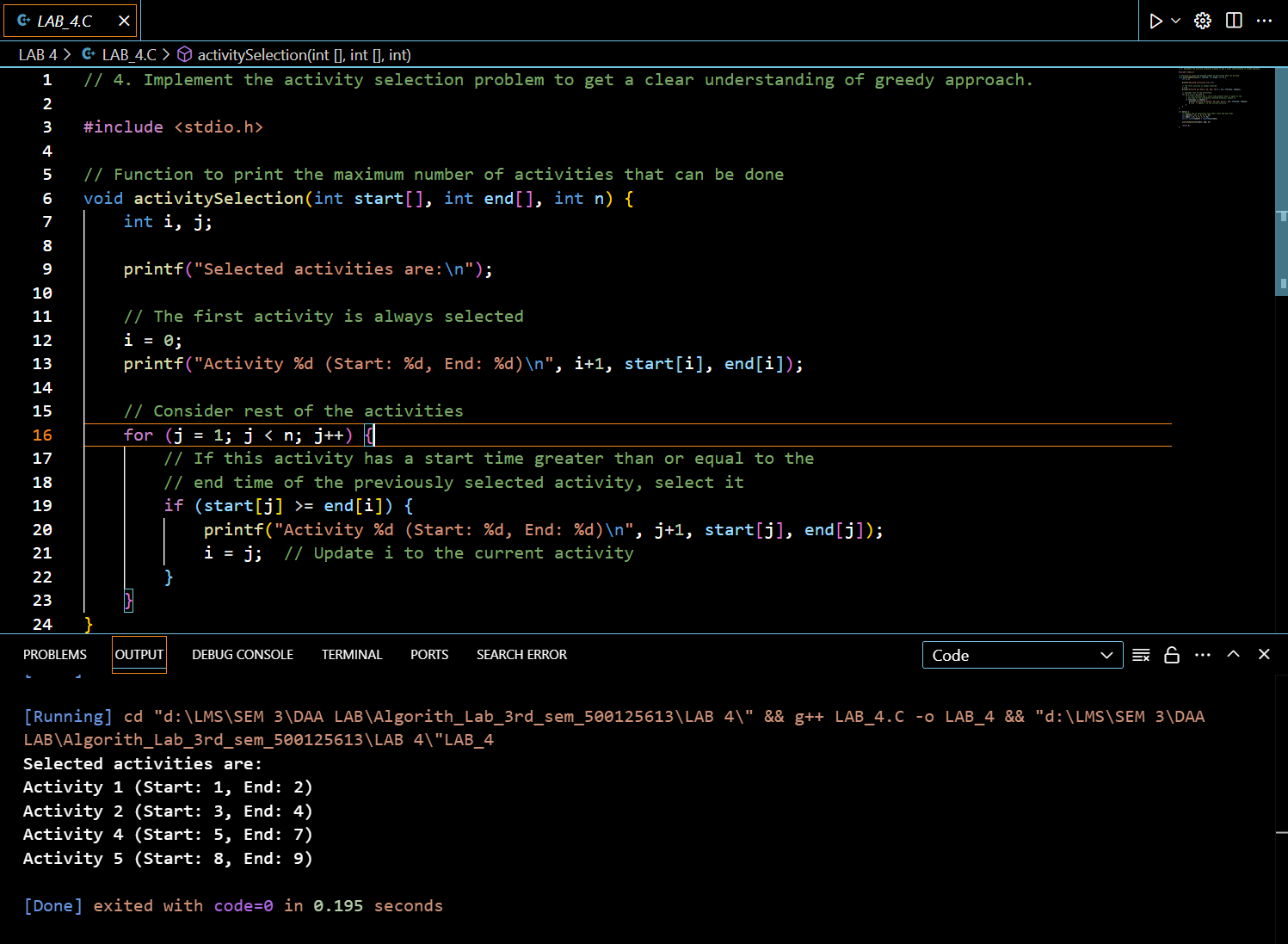
int n = sizeof(start) / sizeof(start[0]);

activitySelection(start, end, n);

return 0;

}}

**Output of the code:**



// 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.**

#include <stdio.h>

#include <limits.h>

#include <time.h>

**// Function to print the optimal parenthesis placement**

void printOptimalParenthesis(int i, int j, int n, int \*bracket, char \*name) {

if (i == j) {

printf("%c", (\*name)++);

return;

}

printf("(");

// Recursively print the left and right subsets

printOptimalParenthesis(i, \*((bracket + i \* n) + j), n, bracket, name);

printOptimalParenthesis(\*((bracket + i \* n) + j) + 1, j, n, bracket, name);

printf(")");

}

**// Matrix Chain Multiplication function using DP with detailed cost printing**

int MatrixChainOrder(int p[], int n) {

int m[n][n]; // Minimum number of multiplications table

int bracket[n][n]; // Stores optimal split point for printing order

// Initialize main diagonal to zero

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;

printf("Calculating cost for matrices from A%d to A%d\n", i, j);

for (int k = i; k <= j - 1; k++) {

int q = m[i][k] + m[k + 1][j] + p[i - 1] \* p[k] \* p[j];

printf("Split at A%d: Cost = %d\n", k, q);

if (q < m[i][j]) {

m[i][j] = q;

bracket[i][j] = k; // Store split point in bracket

}

}

printf("Minimum cost for multiplying A%d to A%d is %d\n\n", i, j, m[i][j]);

}

}

**// Display optimal order of multiplication**

char name = 'A';

printf("Optimal Parenthesization is: ");

printOptimalParenthesis(1, n - 1, n, (int \*)bracket, &name);

printf("\n");

return m[1][n - 1]; // Return minimum cost of multiplication

}

int main() {

int arr[] = {1, 2, 3, 4}; // Dimensions of matrices A1 (1x2), A2 (2x3), A3 (3x4)

int size = sizeof(arr) / sizeof(arr[0]);

clock\_t start = clock();

int min\_cost = MatrixChainOrder(arr, size);

clock\_t end = clock();

double time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC;

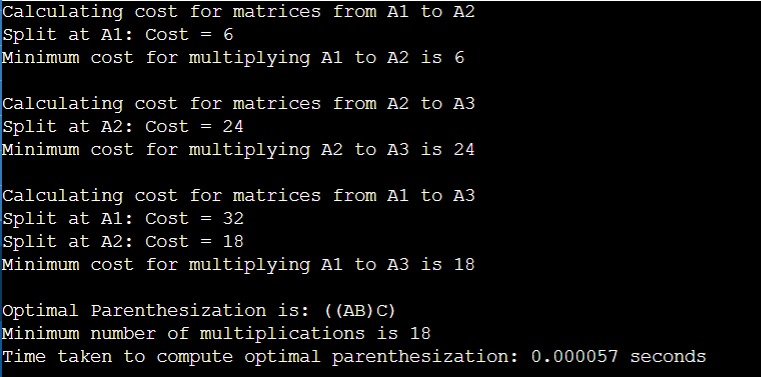
printf("Minimum number of multiplications is %d\n", min\_cost);

printf("Time taken to compute optimal parenthesization: %f seconds\n", time\_taken);

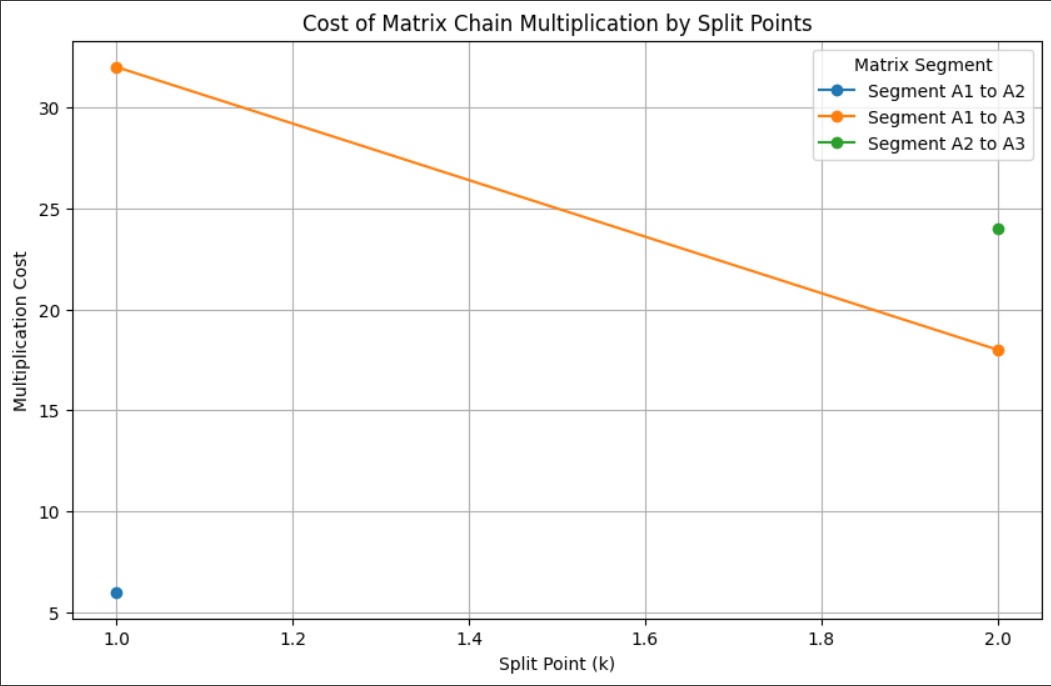
return 0;

}

**Output of the code:**



**Graph:**



// 6 **Compare the performance of Dijkstra and Bellman ford algorithm for the single**

**source shortest path problem**.

#include <stdio.h>

#include <limits.h>

#include <stdbool.h>

#include <time.h>

#define V 6 // Number of vertices in the graph

**// Dijkstra's Algorithm**

void dijkstra(int graph[V][V], int start) {

int distances[V];

bool visited[V];

**// Initialize distances and visited**

for (int i = 0; i < V; i++) {

distances[i] = INT\_MAX;

visited[i] = false;

}

distances[start] = 0;

for (int count = 0; count < V - 1; count++) {

int min\_distance = INT\_MAX, min\_index;

**// Find the vertex with the minimum distance**

for (int v = 0; v < V; v++) {

if (!visited[v] && distances[v] <= min\_distance) {

min\_distance = distances[v];

min\_index = v;

}

}

visited[min\_index] = true;

**// Update distances of the adjacent vertices**

for (int v = 0; v < V; v++) {

if (!visited[v] && graph[min\_index][v] && distances[min\_index] != INT\_MAX &&

distances[min\_index] + graph[min\_index][v] < distances[v]) {

distances[v] = distances[min\_index] + graph[min\_index][v];

}

}

}

**// Print the results**

printf("Dijkstra's Algorithm Result: ");

for (int i = 0; i < V; i++) {

printf("%d ", distances[i]);

}

printf("\n");

}

**// Bellman-Ford Algorithm**

void bellman\_ford(int graph[V][V], int start) {

int distances[V];

**// Initialize distances**

for (int i = 0; i < V; i++) {

distances[i] = INT\_MAX;

}

distances[start] = 0;

**// Relax edges**

for (int i = 0; i < V - 1; i++) {

for (int u = 0; u < V; u++) {

for (int v = 0; v < V; v++) {

if (graph[u][v] && distances[u] != INT\_MAX && distances[u] + graph[u][v] < distances[v]) {

distances[v] = distances[u] + graph[u][v];

}

}

}

}

**// Check for negative weight cycles**

for (int u = 0; u < V; u++) {

for (int v = 0; v < V; v++) {

if (graph[u][v] && distances[u] != INT\_MAX && distances[u] + graph[u][v] < distances[v]) {

printf("Graph contains a negative weight cycle\n");

return;

}

}

}

**// Print the results**

printf("Bellman-Ford Algorithm Result: ");

for (int i = 0; i < V; i++) {

printf("%d ", distances[i]);

}

printf("\n");

}

int main() {

// Example graph represented as an adjacency matrix

int graph[V][V] = {

{0, 7, 9, 0, 0, 14},

{7, 0, 10, 15, 0, 0},

{9, 10, 0, 11, 0, 2},

{0, 15, 11, 0, 6, 0},

{0, 0, 0, 6, 0, 9},

{14, 0, 2, 0, 9, 0}

};

clock\_t start, end;

// Measuring Dijkstra's execution time

start = clock();

dijkstra(graph, 0); // Start from vertex 0

end = clock();

double dijkstra\_time = ((double)(end - start)) / CLOCKS\_PER\_SEC;

**// Measuring Bellman-Ford execution time**

start = clock();

bellman\_ford(graph, 0); // Start from vertex 0

end = clock();

double bellman\_ford\_time = ((double)(end - start)) / CLOCKS\_PER\_SEC;

// Display execution times

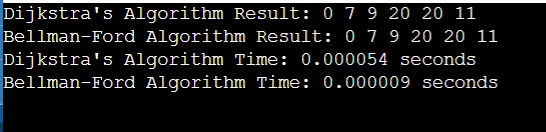
printf("Dijkstra's Algorithm Time: %f seconds\n", dijkstra\_time);

printf("Bellman-Ford Algorithm Time: %f seconds\n", bellman\_ford\_time);

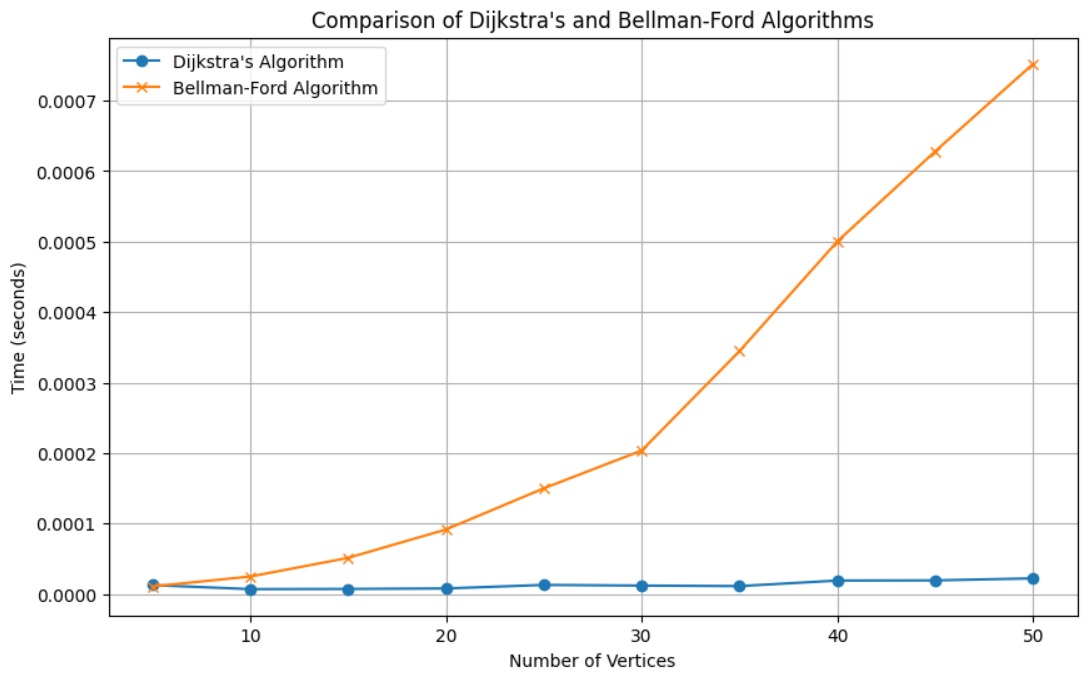
return 0;

}

**Output of the code:**



**Graph:**



**// 7 Through 0/1 Knapsack problem, analyze the greedy and dynamic programming**

**approach for the same dataset**.

#include <stdio.h>

// Structure to represent an item

typedef struct {

int weight;

int value;

} Item;

// Function to calculate the value-to-weight ratio

float ratio(Item item) {

return (float)item.value / item.weight;

}

// Function to sort items based on the ratio in descending order

void sortItems(Item items[], int n) {

for (int i = 0; i < n - 1; i++) {

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

if (ratio(items[i]) < ratio(items[j])) {

// Swap items

Item temp = items[i];

items[i] = items[j];

items[j] = temp;

}

}

}

}

// Function to solve the 0/1 Knapsack problem using the greedy approach

int greedyKnapsack(Item items[], int n, int capacity) {

int totalValue = 0;

int remainingCapacity = capacity;

sortItems(items, n);

for (int i = 0; i < n; i++) {

if (items[i].weight <= remainingCapacity) {

totalValue += items[i].value;

remainingCapacity -= items[i].weight;

}

}

return totalValue;

}

// Function to solve the 0/1 Knapsack problem using dynamic programming

int dynamicKnapsack(Item items[], int n, int capacity) {

int dp[n + 1][capacity + 1];

// Initialize the table

for (int i = 0; i <= n; i++) {

for (int j = 0; j <= capacity; j++) {

if (i == 0 || j == 0) {

dp[i][j] = 0;

} else if (items[i - 1].weight <= j) {

dp[i][j] = (dp[i - 1][j] > dp[i - 1][j - items[i - 1].weight] + items[i - 1].value) ? dp[i - 1][j] : dp[i - 1][j - items[i - 1].weight] + items[i - 1].value;

} else {

dp[i][j] = dp[i - 1][j];

}

}

}

return dp[n][capacity];

}

int main() {

// Define the items

Item items[] = {

{10, 60},

{20, 100},

{30, 120}

};

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

int capacity = 50;

int maxValueGreedy = greedyKnapsack(items, n, capacity);

int maxValueDynamic = dynamicKnapsack(items, n, capacity);

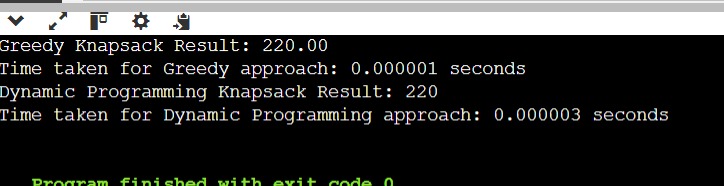
printf("Maximum value using greedy approach: %d\n", maxValueGreedy);

printf("Maximum value using dynamic programming approach: %d\n", maxValueDynamic);

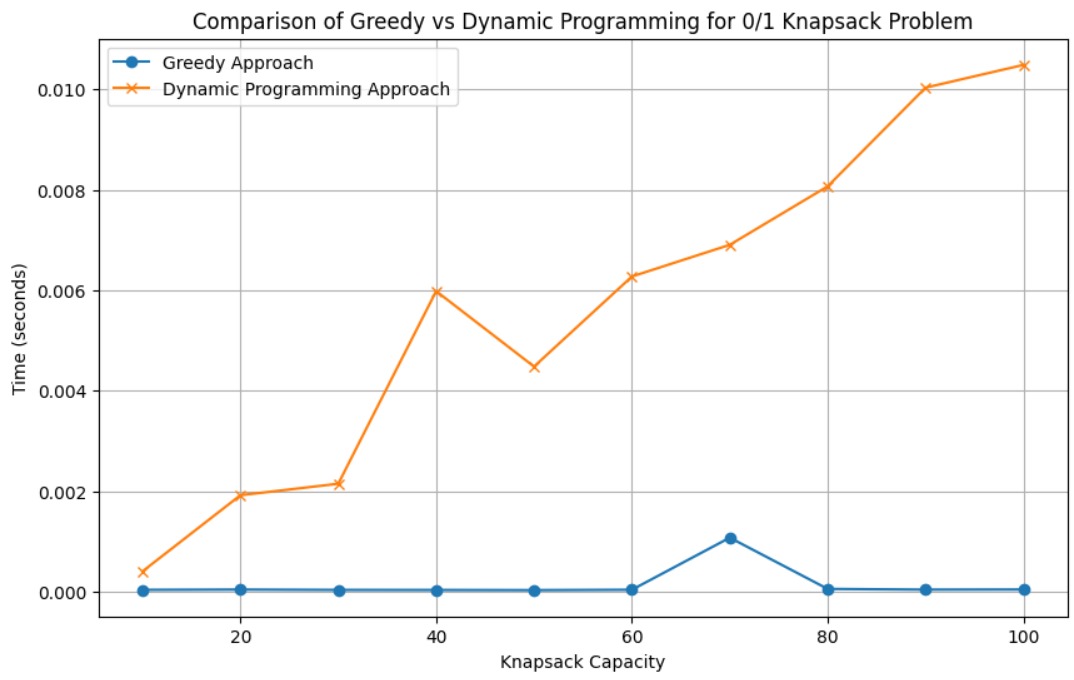
return 0;

}

**Output of the code:**



**Graph**



**// 8 Implement the sum of subset.**

#include <stdio.h>

#include <stdbool.h>

#include <time.h>

void print\_subset(int nums[], int n, int target, bool dp[n + 1][target + 1]) {

int i = n, j = target;

printf("Subset: ");

while (i > 0 && j > 0) {

// If this item is included in the subset

if (dp[i][j] && !dp[i - 1][j]) {

printf("%d ", nums[i - 1]);

j -= nums[i - 1];

}

i--;

}

printf("\n");

}

bool subset\_sum(int nums[], int n, int target) {

// Create a 2D array to store results of subproblems

bool dp[n + 1][target + 1];

// If the target is 0, then the answer is true (empty set)

for (int i = 0; i <= n; i++) {

dp[i][0] = true;

}

// Fill the subset sum table

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

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

if (nums[i - 1] <= j) {

dp[i][j] = dp[i - 1][j] || dp[i - 1][j - nums[i - 1]];

} else {

dp[i][j] = dp[i - 1][j];

}

}

}

**// Print the subset if the target sum is possible**

if (dp[n][target]) {

print\_subset(nums, n, target, dp);

}

return dp[n][target];

}

int main() {

int nums[] = {3, 34, 4, 12, 5, 2};

int target = 9;

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

**// Measure execution time**

clock\_t start\_time = clock();

bool result = subset\_sum(nums, n, target);

clock\_t end\_time = clock();

if (result) {

printf("Subset with given sum exists.\n");

} else {

printf("Subset with given sum does not exist.\n");

}

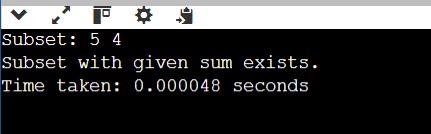
double time\_taken = (double)(end\_time - start\_time) / CLOCKS\_PER\_SEC;

printf("Time taken: %.6f seconds\n", time\_taken);

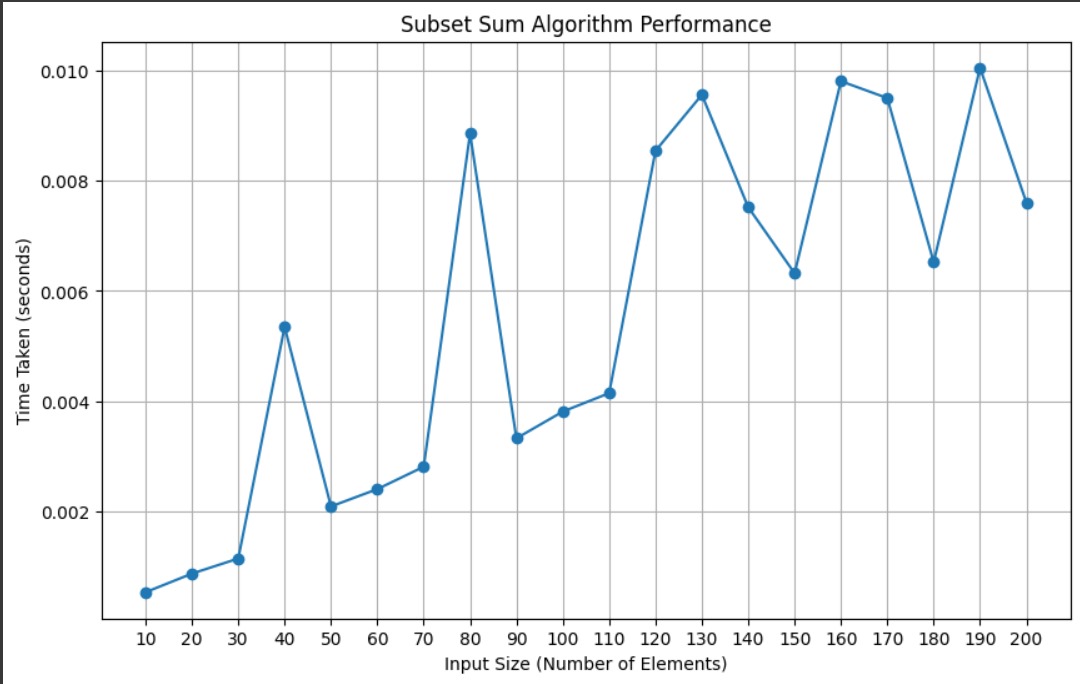
return 0;

}

**Output of the code:**



**Graph:**



**// 9 Compare the Backtracking and Branch & Bound Approach by the implementation**

**of 0/1 Knapsack problem. Also compare the performance with dynamic**

**programming approach.**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

typedef struct {

int weight;

int value;

} Item;

int max(int a, int b) {

return (a > b) ? a : b;

}

**// Dynamic Programming Approach**

int knapsack\_dp(Item items[], int n, int capacity) {

int dp[n + 1][capacity + 1];

for (int i = 0; i <= n; i++) {

for (int w = 0; w <= capacity; w++) {

if (i == 0 || w == 0) {

dp[i][w] = 0; // Base case

} else if (items[i - 1].weight <= w) {

dp[i][w] = max(items[i - 1].value + dp[i - 1][w - items[i - 1].weight], dp[i - 1][w]);

} else {

dp[i][w] = dp[i - 1][w]; // Item can't be included

}

}

}

return dp[n][capacity];

}

**// Backtracking Approach**

int knapsack\_backtracking(Item items[], int n, int capacity, int index, int currentValue) {

if (index >= n || capacity <= 0) {

return currentValue;

}

// Don't include the current item

int maxValue = knapsack\_backtracking(items, n, capacity, index + 1, currentValue);

// Include the current item, if it fits

if (items[index].weight <= capacity) {

maxValue = max(maxValue, knapsack\_backtracking(items, n, capacity - items[index].weight, index + 1, currentValue + items[index].value));

}

return maxValue;

}

**// Branch & Bound Approach**

typedef struct {

int level; // Current level in the decision tree

int profit; // Profit at this node

int bound; // Upper bound of profit

int weight; // Weight at this node

} Node;

int bound(Node u, int n, int capacity, Item items[]) {

if (u.weight >= capacity) return 0; // Can't include this node

int profitBound = u.profit;

int j = u.level + 1;

int totalWeight = u.weight;

**// Greedily add items to the profitBound**

while (j < n && totalWeight + items[j].weight <= capacity) {

totalWeight += items[j].weight;

profitBound += items[j].value;

j++;

}

**// If there's room for the last item, add a fraction of it**

if (j < n) {

profitBound += (capacity - totalWeight) \* items[j].value / items[j].weight;

}

return profitBound;

}

int knapsack\_branch\_bound(Item items[], int n, int capacity) {

Node u, v;

int maxProfit = 0;

// Sorting items by value/weight ratio

for (int i = 0; i < n; i++) {

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

if ((items[i].value / (double)items[i].weight) < (items[j].value / (double)items[j].weight)) {

Item temp = items[i];

items[i] = items[j];

items[j] = temp;

}

}

}

// Queue for BFS

u.level = -1;

u.profit = 0;

u.weight = 0;

u.bound = bound(u, n, capacity, items);

**// Use a simple array as a stack for backtracking**

Node stack[n]; // Stack for storing nodes

int top = -1;

stack[++top] = u;

while (top >= 0) {

u = stack[top--];

// Move to next level

v.level = u.level + 1;

// If we include the next item

v.weight = u.weight + items[v.level].weight;

v.profit = u.profit + items[v.level].value;

if (v.weight <= capacity && v.profit > maxProfit) {

maxProfit = v.profit; // Update maxProfit

}

v.bound = bound(v, n, capacity, items);

if (v.bound > maxProfit) {

stack[++top] = v; // Add node to stack if its bound is greater

}

**// Exclude the next item**

v.weight = u.weight;

v.profit = u.profit;

v.bound = bound(v, n, capacity, items);

if (v.bound > maxProfit) {

stack[++top] = v; // Add node to stack if its bound is greater

}

}

return maxProfit;

}

int main() {

Item items[] = {

{10, 60}, // weight, value

{20, 100},

{30, 120}

};

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

int capacity = 50;

**// Measure execution time for Dynamic Programming**

clock\_t start\_time = clock();

int dp\_result = knapsack\_dp(items, n, capacity);

clock\_t end\_time = clock();

printf("Dynamic Programming Result: %d\n", dp\_result);

printf("Time taken for Dynamic Programming: %.6f seconds\n", (double)(end\_time - start\_time) / CLOCKS\_PER\_SEC);

**// Measure execution time for Backtracking**

start\_time = clock();

int backtrack\_result = knapsack\_backtracking(items, n, capacity, 0, 0);

end\_time = clock();

printf("Backtracking Result: %d\n", backtrack\_result);

printf("Time taken for Backtracking: %.6f seconds\n", (double)(end\_time - start\_time) / CLOCKS\_PER\_SEC);

// Measure execution time for Branch & Bound

start\_time = clock();

int bb\_result = knapsack\_branch\_bound(items, n, capacity);

end\_time = clock();

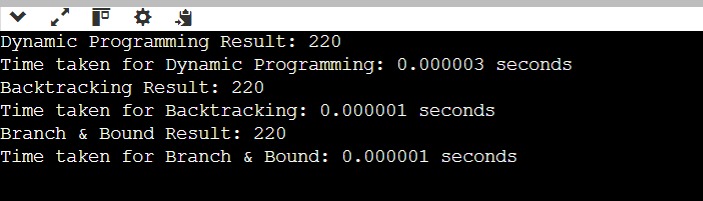
printf("Branch & Bound Result: %d\n", bb\_result);

printf("Time taken for Branch & Bound: %.6f seconds\n", (double)(end\_time - start\_time) / CLOCKS\_PER\_SEC);

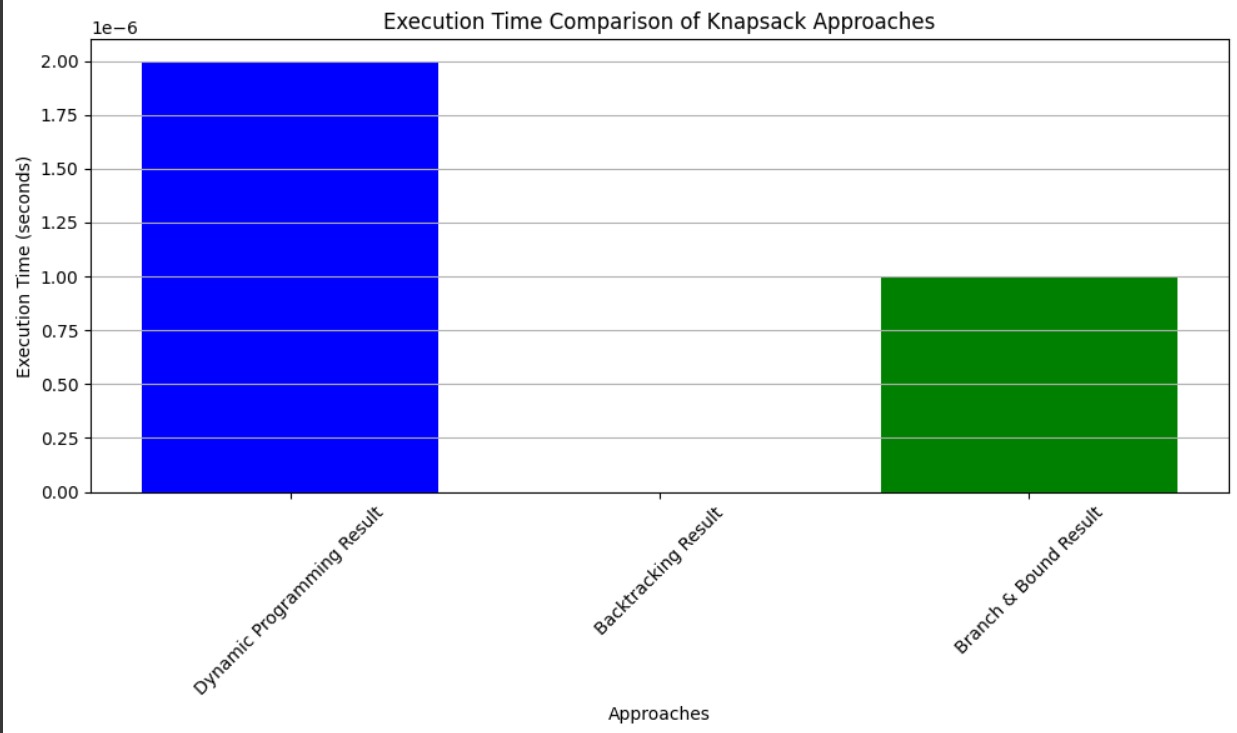
return 0;

}

**Output of the code:**



**Graph:**



// 10 **Compare the performance of Rabin-Karp, Knuth-Morris-Pratt and naive string-matching algorithms.**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <time.h>

#define d 256 // Number of characters in the input alphabet

**// Naive String Matching Algorithm**

void naive\_search(char\* txt, char\* pat) {

int M = strlen(pat);

int N = strlen(txt);

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

int j;

for (j = 0; j < M; j++) {

if (txt[i + j] != pat[j]) break;

}

if (j == M) {

printf("Naive: Pattern found at index %d\n", i);

}

}

}

**// Rabin-Karp Algorithm**

void rabin\_karp(char\* txt, char\* pat, int q) {

int M = strlen(pat);

int N = strlen(txt);

int p = 0; // hash value for pattern

int t = 0; // hash value for text

int h = 1;

for (int i = 0; i < M - 1; i++) {

h = (h \* d) % q; // The value of h would be "d^(M-1)%q"

}

**// Calculate the hash value of pattern and first window of text**

for (int i = 0; i < M; i++) {

p = (d \* p + pat[i]) % q;

t = (d \* t + txt[i]) % q;

}

**// Slide the pattern over text one by one**

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

if (p == t) { // Check for match

int j;

for (j = 0; j < M; j++) {

if (txt[i + j] != pat[j]) break;

}

if (j == M) {

printf("Rabin-Karp: Pattern found at index %d\n", i);

}

}

**// Calculate hash value for next window of text**

if (i < N - M) {

t = (d \* (t - txt[i] \* h) + txt[i + M]) % q;

if (t < 0) t += q; // We might get negative value of t

}

}

}

// KMP Algorithm

void KMPSearch(char\* pat, char\* txt) {

int M = strlen(pat);

int N = strlen(txt);

int lps[M]; // Longest Prefix Suffix

// Preprocess the pattern to create the LPS array

int len = 0; // Length of the previous longest prefix suffix

lps[0] = 0; // lps[0] is always 0

int i = 1;

while (i < M) {

if (pat[i] == pat[len]) {

len++;

lps[i] = len;

i++;

} else {

if (len != 0) {

len = lps[len - 1];

} else {

lps[i] = 0;

i++;

}

}

}

i = 0; // Index for txt[]

int j = 0; // Index for pat[]

while (i < N) {

if (pat[j] == txt[i]) {

i++;

j++;

}

if (j == M) {

printf("KMP: Pattern found at index %d\n", i - j);

j = lps[j - 1];

} else if (i < N && pat[j] != txt[i]) {

if (j != 0) {

j = lps[j - 1];

} else {

i++;

}

}

}

}

int main() {

char txt[] = "ABABDABACDABABCABAB";

char pat[] = "ABAB";

int q = 101; // A prime number for hashing

// Measure execution time for Naive Search

clock\_t start, end;

start = clock();

naive\_search(txt, pat);

end = clock();

double naive\_time = ((double)(end - start)) / CLOCKS\_PER\_SEC;

printf("Naive Search Execution Time: %.6f seconds\n", naive\_time);

// Measure execution time for Rabin-Karp

start = clock();

rabin\_karp(txt, pat, q);

end = clock();

double rabin\_time = ((double)(end - start)) / CLOCKS\_PER\_SEC;

printf("Rabin-Karp Execution Time: %.6f seconds\n", rabin\_time);

// Measure execution time for KMP

start = clock();

KMPSearch(pat, txt);

end = clock();

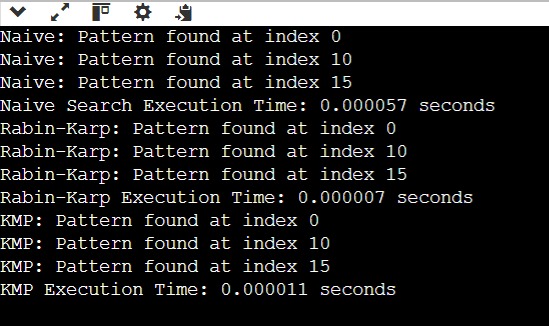
double kmp\_time = ((double)(end - start)) / CLOCKS\_PER\_SEC;

printf("KMP Execution Time: %.6f seconds\n", kmp\_time);

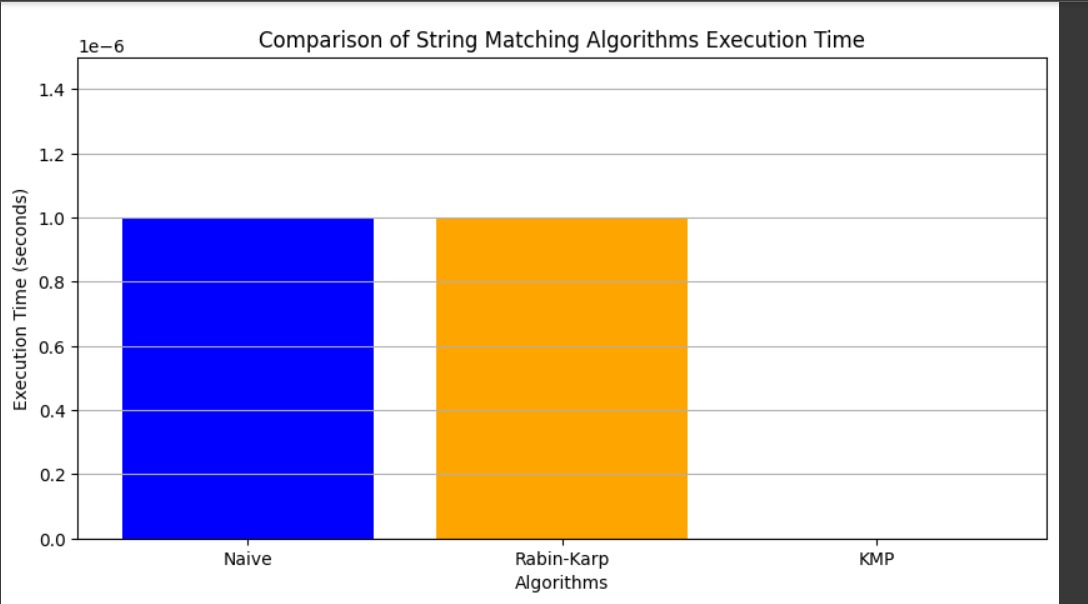
return 0;

}

**Output of the code:**



**Graph:**



**GITHUB LINK:**

**https://github.com/GoelAnshul24/Algorithm\_lab\_3rd\_sem\_-500123668-**