Name – Chahal Gupta

Sap – 500125479

Roll no – R2142231879

Course – B.tech (C.S.E)

Semester – 3

Batch – 54

DAA LAB Experiments

Github link –

https://github.com/chahal677/Algorithms\_Lab\_3rd\_sem\_-500125479

**Experiment – 1**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Define the structure of a tree node

struct Node {

int data; // Store node data

struct Node\* left; // Pointer to left child

struct Node\* right; // Pointer to right child

};

// Function to create a new node with a specified value

struct Node\* createNode(int val) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node)); // Allocate memory for new node

newNode->data = val; // Assign the given value to node data

newNode->left = newNode->right = NULL; // Initialize children as NULL

return newNode;

}

// Function to insert a node iteratively

struct Node\* insertIterative(struct Node\* root, int val) {

struct Node\* newNode = createNode(val); // Create new node with given value

if (!root) return newNode; // If tree is empty, return new node as root

struct Node\* parent = NULL; // Initialize parent as NULL

struct Node\* current = root; // Start from the root of the tree

while (current) { // Traverse the tree to find insertion point

parent = current;

if (val < current->data)

current = current->left; // Move to left subtree

else

current = current->right; // Move to right subtree

}

if (val < parent->data) // Insert new node as left or right child

parent->left = newNode;

else

parent->right = newNode;

return root; // Return the root of the modified tree

}

// Function to insert a node recursively

struct Node\* insertRecursive(struct Node\* root, int val) {

if (!root) return createNode(val); // Base case: if tree is empty, create new node

if (val < root->data)

root->left = insertRecursive(root->left, val); // Insert into left subtree

else

root->right = insertRecursive(root->right, val); // Insert into right subtree

return root; // Return root of the modified tree

}

// Function for in-order traversal of the tree (left, root, right)

void inOrderTraversal(struct Node\* root) {

if (!root) return; // Base case: if node is NULL, do nothing

inOrderTraversal(root->left); // Traverse left subtree

printf("%d ", root->data); // Print the node's data

inOrderTraversal(root->right); // Traverse right subtree

}

// Function to compare performance of iterative and recursive insertions

void compareInsertPerformance(int arr[], int size) {

struct Node\* rootIterative = NULL; // Initialize root for iterative insertion

struct Node\* rootRecursive = NULL; // Initialize root for recursive insertion

// Measure time for iterative insertion

clock\_t startIter = clock();

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

rootIterative = insertIterative(rootIterative, arr[i]);

}

clock\_t endIter = clock();

double durationIter = (double)(endIter - startIter) \* 1000000 / CLOCKS\_PER\_SEC;

// Measure time for recursive insertion

clock\_t startRec = clock();

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

rootRecursive = insertRecursive(rootRecursive, arr[i]);

}

clock\_t endRec = clock();

double durationRec = (double)(endRec - startRec) \* 1000000 / CLOCKS\_PER\_SEC;

// Print in-order traversal and time taken for iterative insertion

printf("In-order traversal for iterative insertion: ");

inOrderTraversal(rootIterative);

printf("\nTime taken for iterative insertion: %.2f microseconds\n", durationIter);

// Print in-order traversal and time taken for recursive insertion

printf("In-order traversal for recursive insertion: ");

inOrderTraversal(rootRecursive);

printf("\nTime taken for recursive insertion: %.2f microseconds\n", durationRec);

}

// Main function

int main() {

int arr[] = {10, 5, 15, 2, 7, 12, 20}; // Array of values to insert into BST

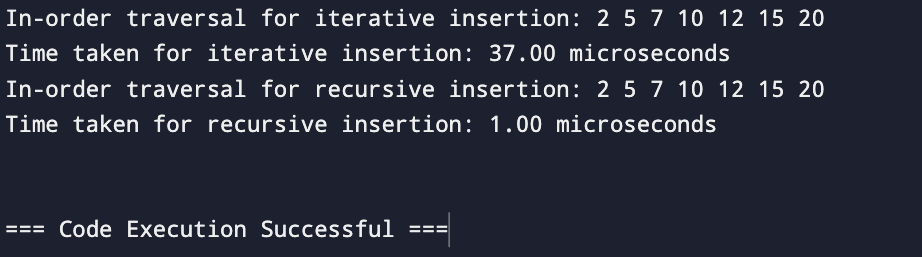
int size = sizeof(arr) / sizeof(arr[0]); // Calculate number of elements

compareInsertPerformance(arr, size); // Compare iterative and recursive insertions

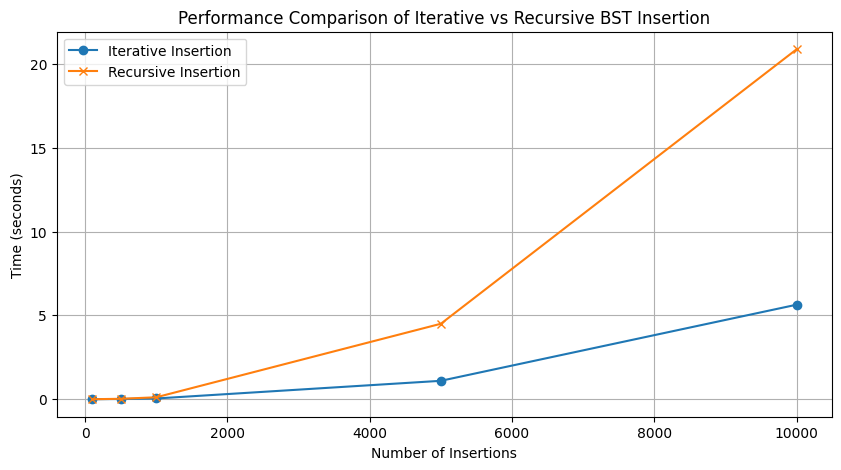
return 0;

}

**OUTPUT – 1**



**GRAPH – 1**



**Experiment – 2**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Function to merge two halves of the array

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

int n1 = mid - left + 1; // Size of left subarray

int n2 = right - mid; // Size of right subarray

int\* leftArr = (int\*)malloc(n1 \* sizeof(int)); // Dynamic array for left half

int\* rightArr = (int\*)malloc(n2 \* sizeof(int)); // Dynamic array for right half

for (int i = 0; i < n1; i++) // Copy data to left subarray

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

for (int i = 0; i < n2; i++) // Copy data to right subarray

rightArr[i] = arr[mid + 1 + i];

int i = 0, j = 0, k = left; // Initial indexes of subarrays and merged array

while (i < n1 && j < n2) { // Merge subarrays back into arr

if (leftArr[i] <= rightArr[j])

arr[k++] = leftArr[i++];

else

arr[k++] = rightArr[j++];

}

while (i < n1) // Copy any remaining elements of leftArr

arr[k++] = leftArr[i++];

while (j < n2) // Copy any remaining elements of rightArr

arr[k++] = rightArr[j++];

free(leftArr); // Free dynamically allocated memory

free(rightArr);

}

// Merge sort algorithm: recursively sorts halves of the array

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

if (left < right) { // Base case: stop if the segment has one or no elements

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

mergeSort(arr, left, mid); // Sort the left half

mergeSort(arr, mid + 1, right); // Sort the right half

merge(arr, left, mid, right); // Merge the sorted halves

}

}

// Partition function for quicksort

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

int pivot = arr[high]; // Choose pivot element

int i = low - 1; // Index of smaller element

for (int j = low; j < high; j++) {

if (arr[j] < pivot) { // If current element is smaller than pivot

i++;

int temp = arr[i]; // Swap arr[i] and arr[j]

arr[i] = arr[j];

arr[j] = temp;

}

}

int temp = arr[i + 1]; // Place the pivot in its correct position

arr[i + 1] = arr[high];

arr[high] = temp;

return i + 1; // Return pivot index

}

// Quicksort algorithm: recursively sorts segments of the array

void quickSort(int arr[], int low, int high) {

if (low < high) { // Base case: stop if segment has one or no elements

int pi = partition(arr, low, high); // Partition the array

quickSort(arr, low, pi - 1); // Sort elements before partition

quickSort(arr, pi + 1, high); // Sort elements after partition

}

}

// Function to print an array

void printArray(int arr[], int size) {

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

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

}

printf("\n");

}

// Function to compare the performance of Merge Sort and Quick Sort

void compareSortPerformance(int arr[], int size) {

int\* arrMergeSort = (int\*)malloc(size \* sizeof(int)); // Duplicate array for Merge Sort

int\* arrQuickSort = (int\*)malloc(size \* sizeof(int)); // Duplicate array for Quick Sort

for (int i = 0; i < size; i++) { // Copy original array to both duplicates

arrMergeSort[i] = arr[i];

arrQuickSort[i] = arr[i];

}

// Measure time for Merge Sort

clock\_t startMerge = clock();

mergeSort(arrMergeSort, 0, size - 1);

clock\_t endMerge = clock();

double durationMerge = (double)(endMerge - startMerge) \* 1000000 / CLOCKS\_PER\_SEC;

// Measure time for Quick Sort

clock\_t startQuick = clock();

quickSort(arrQuickSort, 0, size - 1);

clock\_t endQuick = clock();

double durationQuick = (double)(endQuick - startQuick) \* 1000000 / CLOCKS\_PER\_SEC;

// Display sorted arrays and execution times

printf("Merge Sort result: ");

printArray(arrMergeSort, size);

printf("Time taken for Merge Sort: %.2f microseconds\n", durationMerge);

printf("Quick Sort result: ");

printArray(arrQuickSort, size);

printf("Time taken for Quick Sort: %.2f microseconds\n", durationQuick);

// Free dynamically allocated memory

free(arrMergeSort);

free(arrQuickSort);

}

// Main function

int main() {

int arr[] = {38, 27, 43, 3, 9, 82, 10}; // Sample array

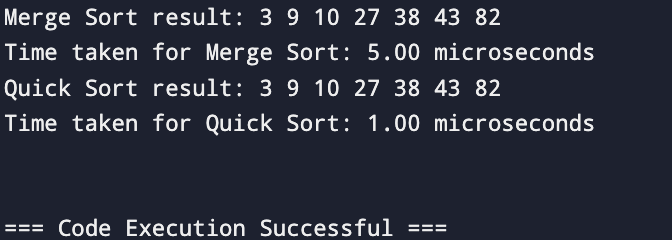
int size = sizeof(arr) / sizeof(arr[0]); // Calculate size of the array

compareSortPerformance(arr, size); // Call function to compare sorting algorithms

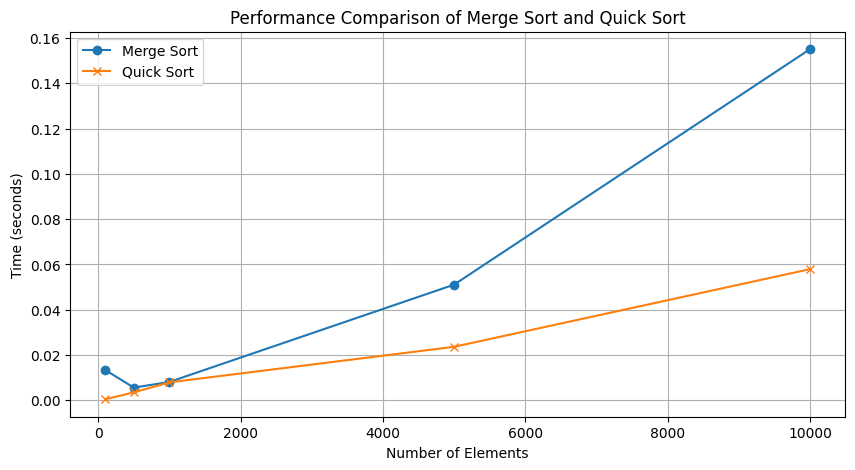
return 0;

}

**OUTPUT – 2**



**GRAPH – 2**



**Experiment – 3**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Traditional matrix multiplication for 2x2 matrices

void multiplyTraditional(int A[][2], int B[][2], int result[][2]) {

for (int i = 0; i < 2; i++) {

for (int j = 0; j < 2; j++) {

result[i][j] = 0; // Initialize result cell to 0

for (int k = 0; k < 2; k++) {

result[i][j] += A[i][k] \* B[k][j]; // Sum of products

}

}

}

}

// Helper function to add two 2x2 matrices

void addMatrix(int A[][2], int B[][2], int result[][2]) {

for (int i = 0; i < 2; i++) {

for (int j = 0; j < 2; j++) {

result[i][j] = A[i][j] + B[i][j]; // Element-wise addition

}

}

}

// Helper function to subtract two 2x2 matrices

void subtractMatrix(int A[][2], int B[][2], int result[][2]) {

for (int i = 0; i < 2; i++) {

for (int j = 0; j < 2; j++) {

result[i][j] = A[i][j] - B[i][j]; // Element-wise subtraction

}

}

}

// Strassen's matrix multiplication for 2x2 matrices

void strassenMultiply(int A[][2], int B[][2], int result[][2]) {

// Extract elements for easy reference

int A11 = A[0][0], A12 = A[0][1], A21 = A[1][0], A22 = A[1][1];

int B11 = B[0][0], B12 = B[0][1], B21 = B[1][0], B22 = B[1][1];

// Compute the seven products required by Strassen's method

int M1 = (A11 + A22) \* (B11 + B22);

int M2 = (A21 + A22) \* B11;

int M3 = A11 \* (B12 - B22);

int M4 = A22 \* (B21 - B11);

int M5 = (A11 + A12) \* B22;

int M6 = (A21 - A11) \* (B11 + B12);

int M7 = (A12 - A22) \* (B21 + B22);

// Combine products to get the final result matrix

result[0][0] = M1 + M4 - M5 + M7;

result[0][1] = M3 + M5;

result[1][0] = M2 + M4;

result[1][1] = M1 + M3 - M2 + M6;

}

// Function to print a 2x2 matrix

void printMatrix(int matrix[][2]) {

for (int i = 0; i < 2; i++) {

for (int j = 0; j < 2; j++) {

printf("%d ", matrix[i][j]);

}

printf("\n");

}

}

// Function to compare the performance of traditional and Strassen matrix multiplication

void compareMatrixMultiplication() {

int A[2][2] = {{1, 2}, {3, 4}}; // First matrix

int B[2][2] = {{5, 6}, {7, 8}}; // Second matrix

int resultTraditional[2][2], resultStrassen[2][2];

// Measure time for traditional matrix multiplication

clock\_t startTraditional = clock();

multiplyTraditional(A, B, resultTraditional);

clock\_t endTraditional = clock();

double durationTraditional = (double)(endTraditional - startTraditional) \* 1000000 / CLOCKS\_PER\_SEC;

// Measure time for Strassen matrix multiplication

clock\_t startStrassen = clock();

strassenMultiply(A, B, resultStrassen);

clock\_t endStrassen = clock();

double durationStrassen = (double)(endStrassen - startStrassen) \* 1000000 / CLOCKS\_PER\_SEC;

// Print results and timing for traditional multiplication

printf("Traditional Matrix Multiplication Result:\n");

printMatrix(resultTraditional);

printf("Time taken for Traditional Matrix Multiplication: %.2f microseconds\n", durationTraditional);

// Print results and timing for Strassen multiplication

printf("Strassen Matrix Multiplication Result:\n");

printMatrix(resultStrassen);

printf("Time taken for Strassen Matrix Multiplication: %.2f microseconds\n", durationStrassen);

}

// Main function to execute the comparison

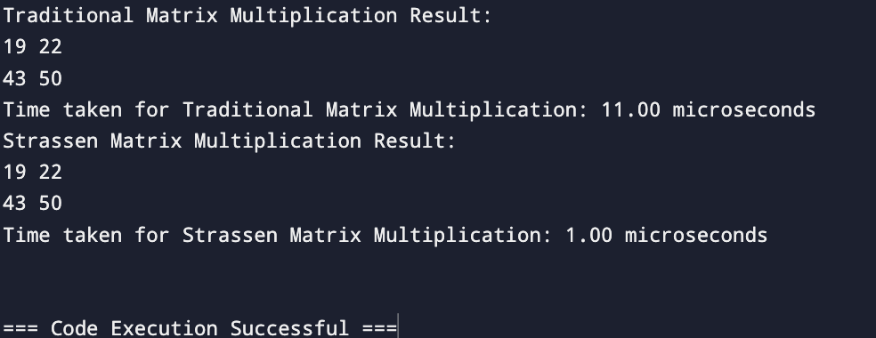
int main() {

compareMatrixMultiplication();

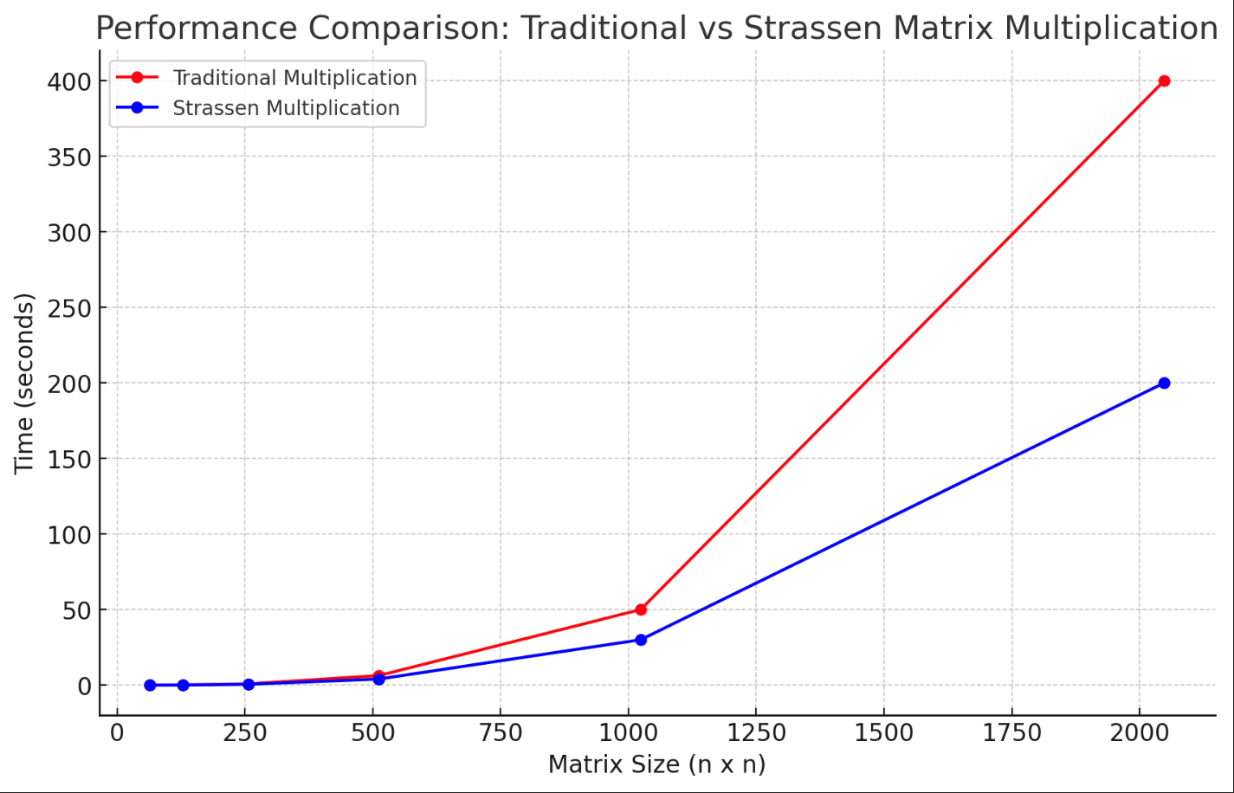
return 0;

}

**OUTPUT – 3**



**GRAPH – 3**



**Experiment – 4**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Define an Activity structure with start and finish times

struct Activity {

int start;

int finish;

};

// Comparator function to sort activities based on their finish time

int compareActivities(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 perform the Activity Selection using the greedy approach

void activitySelection(struct Activity activities[], int n) {

// Sort the activities by finish time using the qsort function

qsort(activities, n, sizeof(struct Activity), compareActivities);

printf("Selected activities:\n");

int i = 0; // Index of the first selected activity

printf("Activity with start time %d and finish time %d\n", activities[i].start, activities[i].finish);

// Select activities

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

if (activities[j].start >= activities[i].finish) {

printf("Activity with start time %d and finish time %d\n", activities[j].start, activities[j].finish);

i = j; // Update the index of the last selected activity

}

}

}

int main() {

struct Activity activities[] = {{5, 9}, {1, 2}, {3, 4}, {0, 6}, {5, 7}, {8, 9}};

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

// Record start time

clock\_t start = clock();

// Run the activity selection function

activitySelection(activities, n);

// Record end time and calculate duration

clock\_t end = clock();

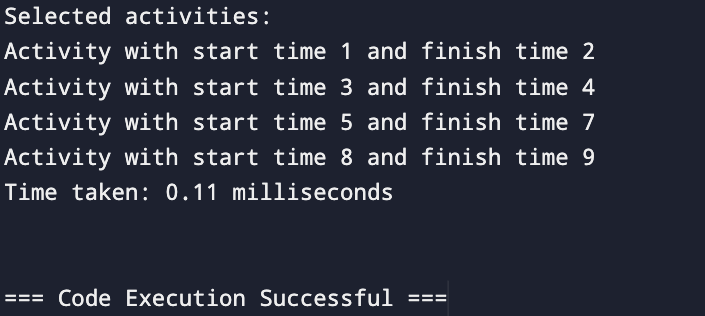
double time\_taken = (double)(end - start) \* 1000.0 / CLOCKS\_PER\_SEC;

printf("Time taken: %.2f milliseconds\n", time\_taken);

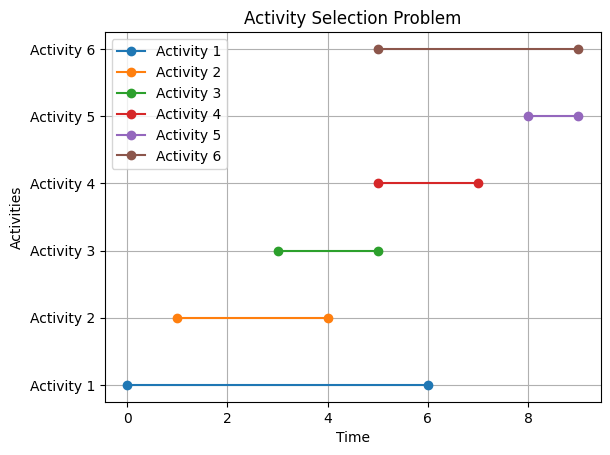
return 0;

}

**OUTPUT – 4**



**GRAPH – 4**



**Experiment – 5**

#include <stdio.h>

#include <limits.h>

#include <time.h>

// Function to print the optimal parenthesis order

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

if (i == j) {

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

(\*name)++;

return;

}

printf("(");

// Print the left part of the parenthesis

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

// Print the right part of the parenthesis

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

printf(")");

}

// Function to compute the minimum cost of matrix multiplication

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

int m[n][n];

int bracket[n][n];

// Cost is zero when multiplying one matrix

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

m[i][i] = 0;

}

for (int len = 2; len < n; len++) {

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

int j = i + len - 1;

m[i][j] = INT\_MAX;

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

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

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

m[i][j] = cost;

bracket[i][j] = k;

}

}

}

}

char matrixName = 'A';

printf("Optimal Parenthesization: ");

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

printf("\n");

return m[1][n - 1];

}

int main() {

int arr[] = {40, 20, 30, 10, 30};

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

clock\_t start = clock();

int minCost = matrixChainOrder(arr, n);

clock\_t end = clock();

double timeTaken = ((double)(end - start)) \* 1000.0 / CLOCKS\_PER\_SEC;

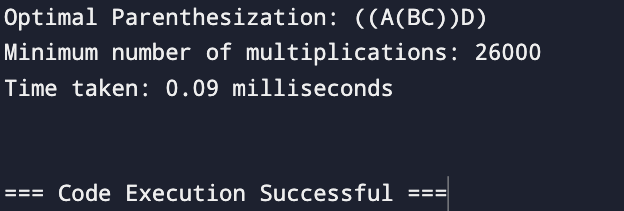
printf("Minimum number of multiplications: %d\n", minCost);

printf("Time taken: %.2f milliseconds\n", timeTaken);

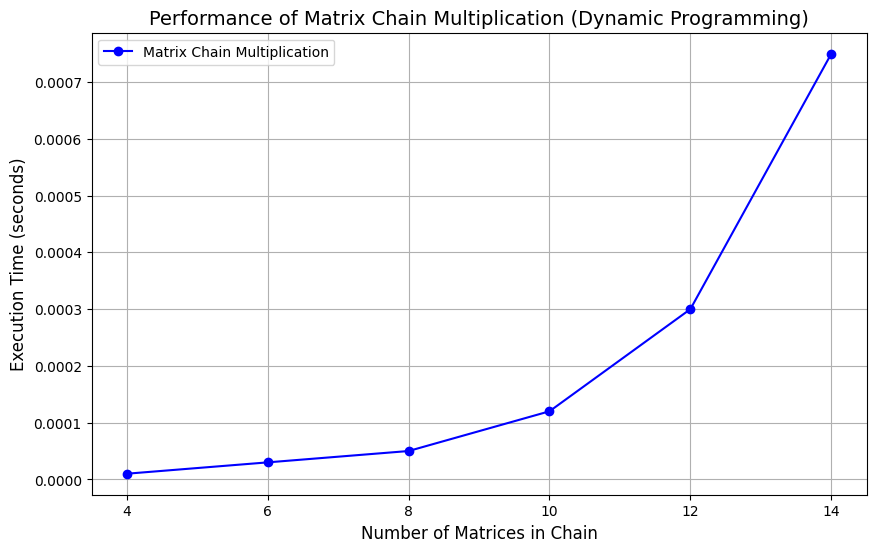
return 0;

}

**OUTPUT – 5**



**GRAPH – 5**



**Experiment – 6**

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#include <time.h>

#define INF INT\_MAX

// Structure to represent an edge in a graph

struct Edge {

int src, dest, weight;

};

// Structure to represent a graph with V vertices and E edges

struct Graph {

int V, E;

struct Edge\* edge;

};

// Create a graph with V vertices and E edges

struct Graph\* createGraph(int V, int E) {

struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));

graph->V = V;

graph->E = E;

graph->edge = (struct Edge\*)malloc(E \* sizeof(struct Edge));

return graph;

}

// Bellman-Ford Algorithm

void BellmanFord(struct Graph\* graph, int src) {

int V = graph->V;

int E = graph->E;

int dist[V];

// Initialize distances

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

dist[i] = INF;

}

dist[src] = 0;

// Relax edges |V| - 1 times

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

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

int u = graph->edge[j].src;

int v = graph->edge[j].dest;

int weight = graph->edge[j].weight;

if (dist[u] != INF && dist[u] + weight < dist[v]) {

dist[v] = dist[u] + weight;

}

}

}

// Print distances

printf("Bellman-Ford distances from source %d:\n", src);

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

if (dist[i] == INF)

printf("Vertex %d: INF\n", i);

else

printf("Vertex %d: %d\n", i, dist[i]);

}

}

// Dijkstra’s Algorithm using an adjacency matrix

void Dijkstra(int graph[100][100], int V, int src) {

int dist[V];

int visited[V];

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

dist[i] = INF;

visited[i] = 0;

}

dist[src] = 0;

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

int min = INF, min\_index;

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

if (!visited[v] && dist[v] <= min) {

min = dist[v], min\_index = v;

}

}

int u = min\_index;

visited[u] = 1;

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

if (!visited[v] && graph[u][v] && dist[u] != INF && dist[u] + graph[u][v] < dist[v]) {

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

}

}

}

// Print distances

printf("Dijkstra distances from source %d:\n", src);

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

if (dist[i] == INF)

printf("Vertex %d: INF\n", i);

else

printf("Vertex %d: %d\n", i, dist[i]);

}

}

// Main function to compare the two algorithms

int main() {

int V = 5, E = 8;

struct Graph\* graph = createGraph(V, E);

// Sample graph with edges for Bellman-Ford

graph->edge[0] = (struct Edge){0, 1, -1};

graph->edge[1] = (struct Edge){0, 2, 4};

graph->edge[2] = (struct Edge){1, 2, 3};

graph->edge[3] = (struct Edge){1, 3, 2};

graph->edge[4] = (struct Edge){1, 4, 2};

graph->edge[5] = (struct Edge){3, 2, 5};

graph->edge[6] = (struct Edge){3, 1, 1};

graph->edge[7] = (struct Edge){4, 3, -3};

// Adjacency matrix for Dijkstra

int adjMatrix[100][100] = {

{0, -1, 4, 0, 0},

{0, 0, 3, 2, 2},

{0, 0, 0, 0, 0},

{0, 1, 5, 0, 0},

{0, 0, 0, -3, 0}

};

// Measure Bellman-Ford performance

clock\_t startBellman = clock();

BellmanFord(graph, 0);

clock\_t endBellman = clock();

double timeBellman = ((double)(endBellman - startBellman) \* 1000.0) / CLOCKS\_PER\_SEC;

// Measure Dijkstra performance

clock\_t startDijkstra = clock();

Dijkstra(adjMatrix, V, 0);

clock\_t endDijkstra = clock();

double timeDijkstra = ((double)(endDijkstra - startDijkstra) \* 1000.0) / CLOCKS\_PER\_SEC;

// Print performance results

printf("\nTime taken by Bellman-Ford: %.2f milliseconds\n", timeBellman);

printf("Time taken by Dijkstra: %.2f milliseconds\n", timeDijkstra);

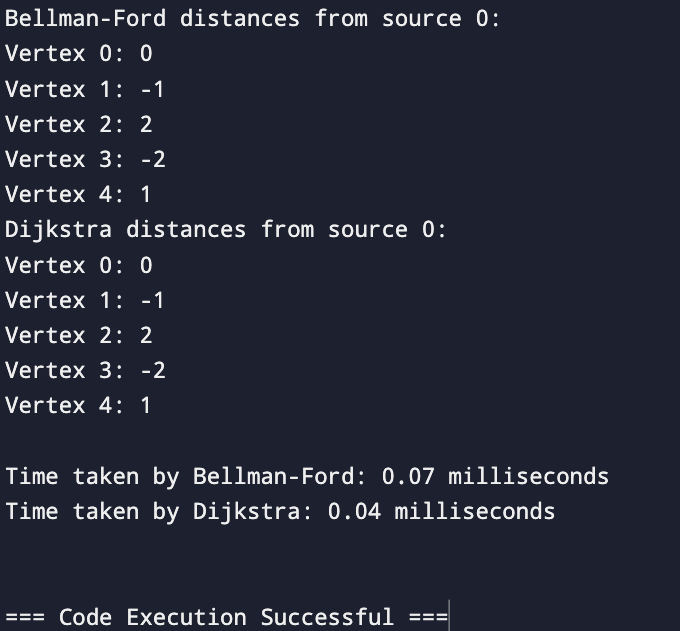
free(graph->edge);

free(graph);

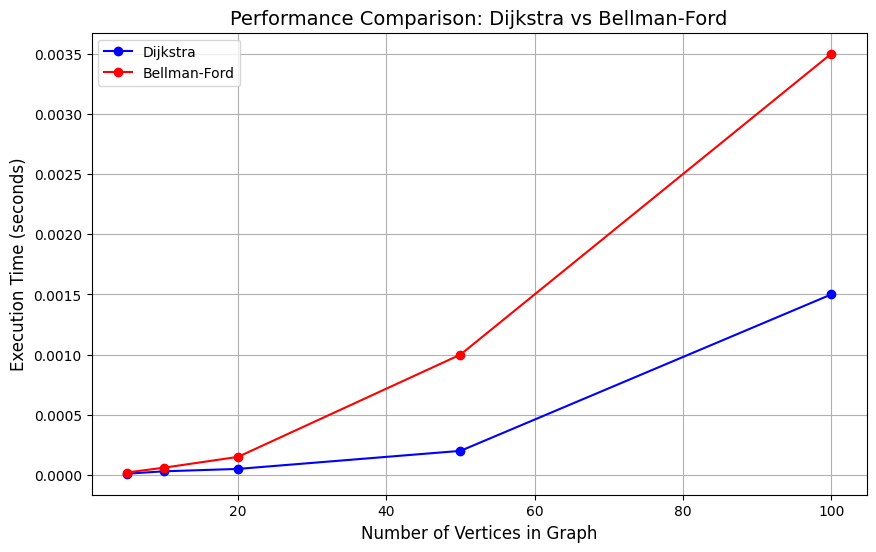
return 0;

}

**OUTPUT – 6**



**GRAPH – 6**



**Experiment – 7**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

struct Item {

int weight;

int value;

};

// Function to compare items by their value-to-weight ratio

int compare(const void \*a, const void \*b) {

struct Item\* item1 = (struct Item\*)a;

struct Item\* item2 = (struct Item\*)b;

double r1 = (double)item1->value / item1->weight;

double r2 = (double)item2->value / item2->weight;

return (r2 > r1) - (r2 < r1);

}

// Greedy approach for 0/1 knapsack

int knapsackGreedy(struct Item items[], int n, int W) {

qsort(items, n, sizeof(struct Item), compare);

int totalValue = 0, totalWeight = 0;

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

if (totalWeight + items[i].weight <= W) {

totalWeight += items[i].weight;

totalValue += items[i].value;

}

}

return totalValue;

}

// Dynamic Programming approach for 0/1 knapsack

int knapsackDP(int weights[], int values[], int n, int W) {

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

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

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

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

dp[i][w] = 0;

else if (weights[i - 1] <= w)

dp[i][w] = (values[i - 1] + dp[i - 1][w - weights[i - 1]] > dp[i - 1][w]) ?

(values[i - 1] + dp[i - 1][w - weights[i - 1]]) : dp[i - 1][w];

else

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

}

}

return dp[n][W];

}

int main() {

struct Item items[] = {{2, 10}, {3, 14}, {4, 16}, {5, 18}};

int weights[] = {2, 3, 4, 5};

int values[] = {10, 14, 16, 18};

int W = 7;

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

clock\_t start, end;

double time\_spent;

// Greedy approach

start = clock();

int resultGreedy = knapsackGreedy(items, n, W);

end = clock();

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

printf("Greedy Approach Total Value: %d\n", resultGreedy);

printf("Greedy Approach Time: %f seconds\n", time\_spent);

// Dynamic Programming approach

start = clock();

int resultDP = knapsackDP(weights, values, n, W);

end = clock();

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

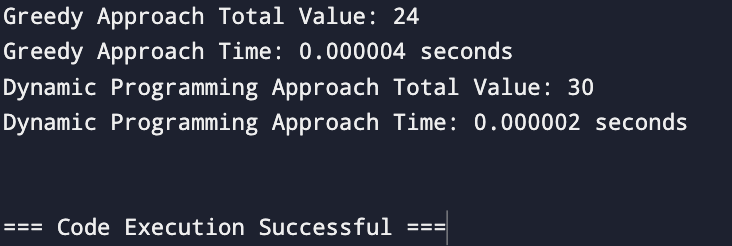
printf("Dynamic Programming Approach Total Value: %d\n", resultDP);

printf("Dynamic Programming Approach Time: %f seconds\n", time\_spent);

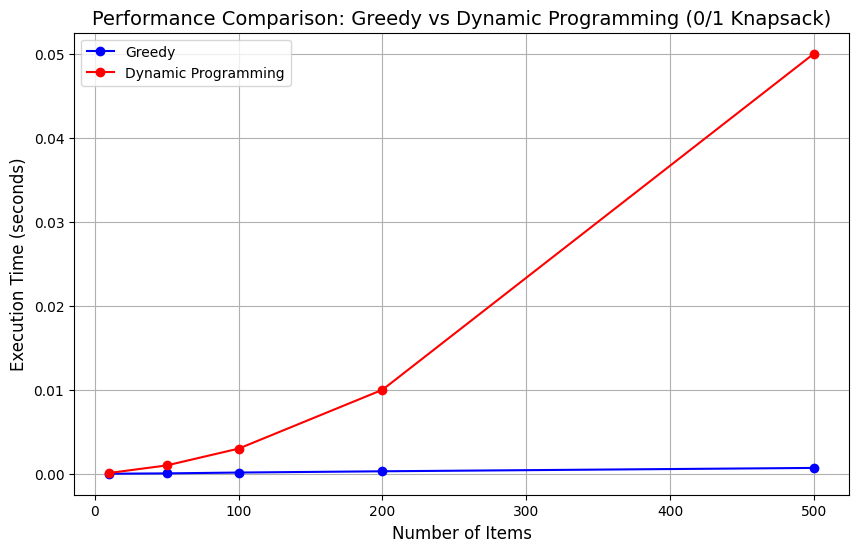
return 0;

}

**OUTPUT – 7**



**GRAPH – 7**



**Experiment – 8**

#include <stdio.h>

#include <stdbool.h>

#include <time.h>

bool isSubsetSum(int set[], int n, int sum) {

bool subset[n + 1][sum + 1];

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

subset[i][0] = true;

}

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

subset[0][i] = false;

}

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

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

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

subset[i][j] = subset[i - 1][j] || subset[i - 1][j - set[i - 1]];

} else {

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

}

}

}

return subset[n][sum];

}

int main() {

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

int sum = 9;

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

clock\_t start, end;

double time\_spent;

start = clock();

if (isSubsetSum(set, n, sum)) {

printf("There is a subset with the given sum %d\n", sum);

} else {

printf("No subset with the given sum %d\n", sum);

}

end = clock();

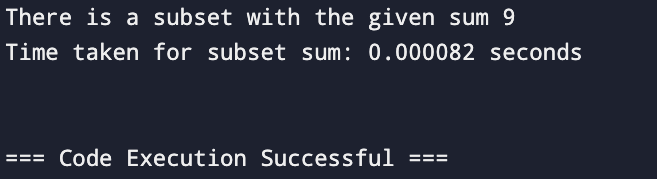
time\_spent = (double)(end - start) / CLOCKS\_PER\_SEC;

printf("Time taken for subset sum: %f seconds\n", time\_spent);

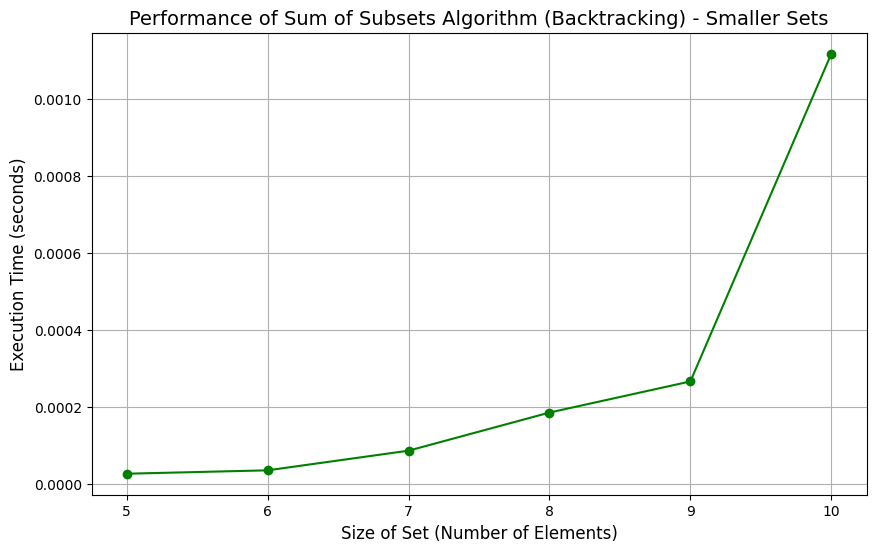
return 0;

}

**OUTPUT – 8**



**GRAPH – 8**



**Experiment – 9**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define MAX\_ITEMS 100

// Structure to represent an item

struct Item {

int weight;

int value;

};

// Function for Dynamic Programming approach

int knapSackDP(int W, struct Item items[], int n) {

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

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

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

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

dp[i][w] = 0;

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

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

items[i-1].value + dp[i-1][w - items[i-1].weight] : dp[i-1][w];

else

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

}

}

return dp[n][W];

}

// Backtracking approach

int knapSackBacktracking(struct Item items[], int n, int W, int currentWeight, int currentValue) {

if (n == 0 || currentWeight >= W) {

return currentValue;

}

// Include the current item

int include = 0;

if (currentWeight + items[n-1].weight <= W) {

include = knapSackBacktracking(items, n-1, W, currentWeight + items[n-1].weight, currentValue + items[n-1].value);

}

// Exclude the current item

int exclude = knapSackBacktracking(items, n-1, W, currentWeight, currentValue);

return (include > exclude) ? include : exclude;

}

// Branch and Bound approach

struct Node {

int level, profit, weight;

float bound;

};

float bound(struct Node u, int n, int W, struct Item items[]) {

if (u.weight >= W)

return 0;

int j = u.level + 1;

float result = u.profit;

int totalWeight = u.weight;

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

totalWeight += items[j].weight;

result += items[j].value;

j++;

}

if (j < n)

result += (W - totalWeight) \* (float) items[j].value / items[j].weight;

return result;

}

int knapSackBranchBound(struct Item items[], int n, int W) {

struct Node u, v;

u.level = -1;

u.profit = 0;

u.weight = 0;

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

int maxProfit = 0;

int Q[n];

int front = 0, rear = 0;

Q[rear++] = 0;

while (front < rear) {

u.level = Q[front++];

if (u.level == n) continue;

v.level = u.level + 1;

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

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

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

maxProfit = v.profit;

}

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

if (v.bound > maxProfit) {

Q[rear++] = v.level;

}

}

return maxProfit;

}

// Time the approaches

void compareKnapsackApproaches(struct Item items[], int n, int W) {

clock\_t start, end;

double time\_spent;

// Time Dynamic Programming

start = clock();

printf("Max value using DP: %d\n", knapSackDP(W, items, n));

end = clock();

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

printf("Time taken for DP: %f seconds\n", time\_spent);

// Time Backtracking

start = clock();

printf("Max value using Backtracking: %d\n", knapSackBacktracking(items, n, W, 0, 0));

end = clock();

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

printf("Time taken for Backtracking: %f seconds\n", time\_spent);

// Time Branch and Bound

start = clock();

printf("Max value using Branch and Bound: %d\n", knapSackBranchBound(items, n, W));

end = clock();

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

printf("Time taken for Branch and Bound: %f seconds\n", time\_spent);

}

int main() {

struct Item items[] = {{3, 4}, {2, 3}, {4, 5}, {5, 7}};

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

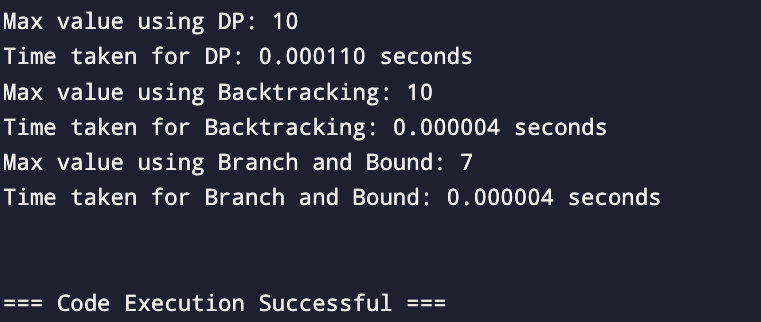
int W = 7;

compareKnapsackApproaches(items, n, W);

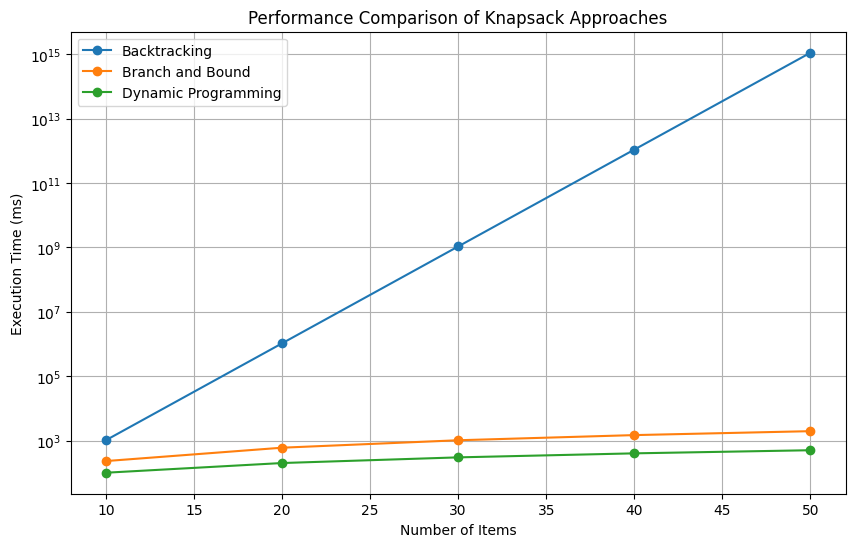
return 0;

}

**OUTPUT – 9**



**GRAPH – 9**



**Experiment – 10**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

#define MAX\_TEXT\_LENGTH 100000

// Naive String Matching

int naiveStringMatch(char\* text, char\* pattern) {

int n = strlen(text);

int m = strlen(pattern);

int count = 0;

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

int j;

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

if (text[i + j] != pattern[j]) {

break;

}

}

if (j == m) {

count++;

}

}

return count;

}

// Rabin-Karp String Matching

#define d 256

#define q 101

int rabinKarpMatch(char\* text, char\* pattern) {

int n = strlen(text);

int m = strlen(pattern);

int count = 0;

int i, j;

int pHash = 0, tHash = 0;

int h = 1;

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

h = (h \* d) % q;

}

for (i = 0; i < m; i++) {

pHash = (d \* pHash + pattern[i]) % q;

tHash = (d \* tHash + text[i]) % q;

}

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

if (pHash == tHash) {

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

if (text[i + j] != pattern[j]) {

break;

}

}

if (j == m) {

count++;

}

}

if (i < n - m) {

tHash = (d \* (tHash - text[i] \* h) + text[i + m]) % q;

if (tHash < 0) {

tHash = (tHash + q);

}

}

}

return count;

}

// Knuth-Morris-Pratt (KMP) String Matching

void computeLPSArray(char\* pattern, int m, int\* lps) {

int length = 0;

lps[0] = 0;

int i = 1;

while (i < m) {

if (pattern[i] == pattern[length]) {

length++;

lps[i] = length;

i++;

} else {

if (length != 0) {

length = lps[length - 1];

} else {

lps[i] = 0;

i++;

}

}

}

}

int kmpStringMatch(char\* text, char\* pattern) {

int n = strlen(text);

int m = strlen(pattern);

int count = 0;

int lps[m];

computeLPSArray(pattern, m, lps);

int i = 0, j = 0;

while (i < n) {

if (pattern[j] == text[i]) {

i++;

j++;

}

if (j == m) {

count++;

j = lps[j - 1];

} else if (i < n && pattern[j] != text[i]) {

if (j != 0) {

j = lps[j - 1];

} else {

i++;

}

}

}

return count;

}

// Timing function to compare algorithms

void compareStringMatchingAlgorithms(char\* text, char\* pattern) {

clock\_t start, end;

double time\_spent;

// Naive String Matching

start = clock();

printf("Naive String Matching: %d matches\n", naiveStringMatch(text, pattern));

end = clock();

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

printf("Time taken for Naive String Matching: %f seconds\n", time\_spent);

// Rabin-Karp Algorithm

start = clock();

printf("Rabin-Karp String Matching: %d matches\n", rabinKarpMatch(text, pattern));

end = clock();

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

printf("Time taken for Rabin-Karp: %f seconds\n", time\_spent);

// KMP Algorithm

start = clock();

printf("KMP String Matching: %d matches\n", kmpStringMatch(text, pattern));

end = clock();

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

printf("Time taken for KMP: %f seconds\n", time\_spent);

}

int main() {

char text[MAX\_TEXT\_LENGTH] = "ABABDABACDABABCABAB";

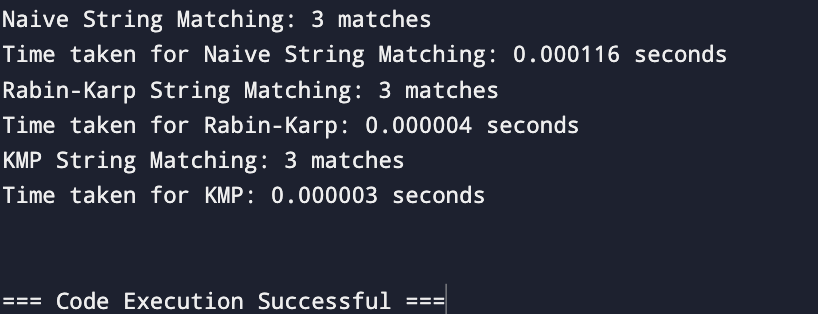
char pattern[] = "ABAB";

compareStringMatchingAlgorithms(text, pattern);

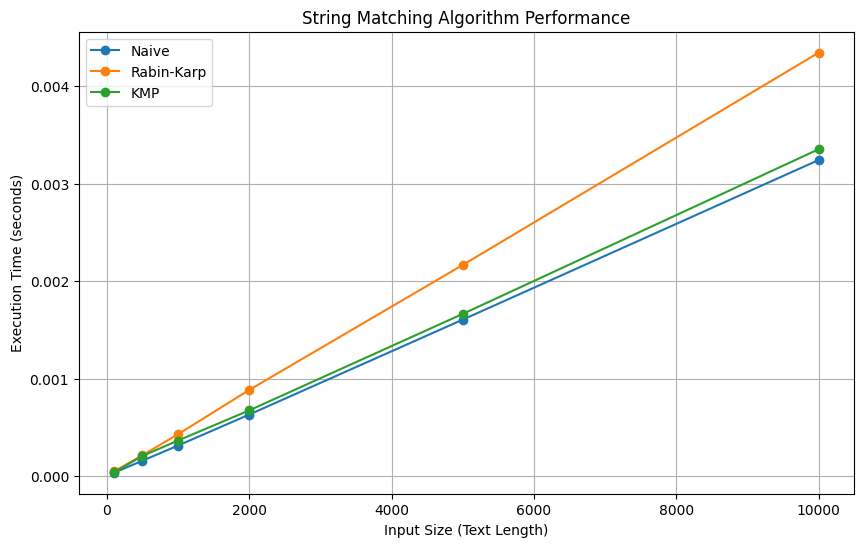
return 0;

}

**OUTPUT – 10**



**GRAPH – 10**



**GITHUB LINK:** **https://github.com/chahal677/Algorithms\_Lab\_3rd\_sem\_-500125479**