**DAA lab**

Name:- Devansh jain

Sap Id:- 500119595

Batch:- 54

**Experiment 1:**

IMPLEMENT THE INSERTION INSIDE ITERATIVE AND RECURSIVE BINARY SEARCH TREE AND COMPARE THEIR PERFORMANCE.

// 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 BST NODE

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

// CREATING 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;

}

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");

}

}

int main() {

// DEFINE FIVE SAMPLE ARRAYS

int arrays[5][10] = {

{5, 35, 67, 60, 80, 10, 20},

{7, 20, 80, 40, 50, 60, 70, 80, 90},

{25, 35, 58, 10, 22, 35, 70, 40, 80},

{10, 90, 80, 70, 60},

{9, 75, 15, 35, 20, 30, 10}

};

// DEFINE THE SIZE OF EACH ARRAY

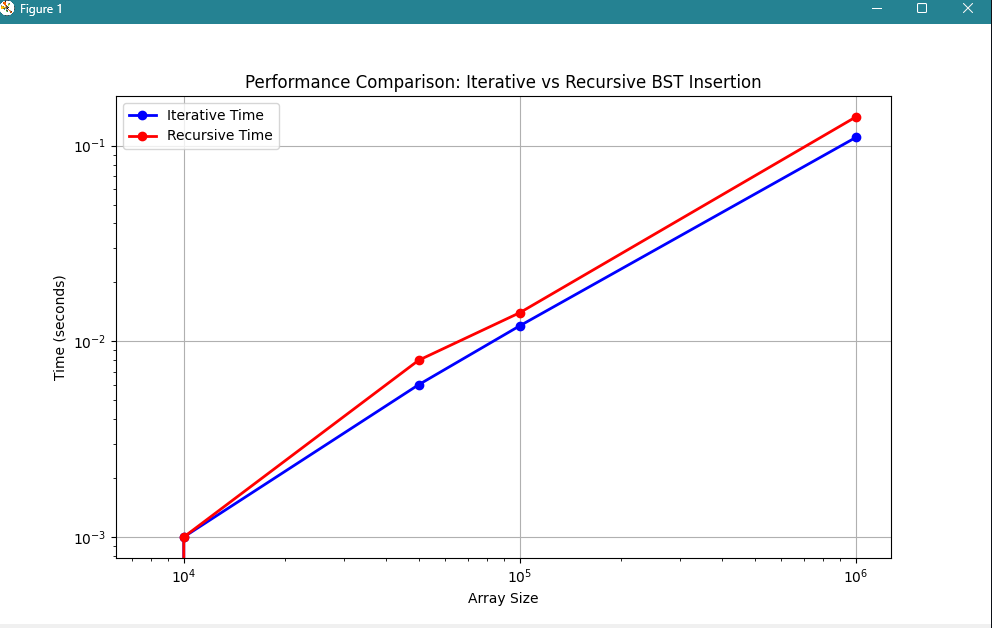
int sizes[5] = {2, 10, 19, 8, 7};

// COMPARE INSERTION TIMES

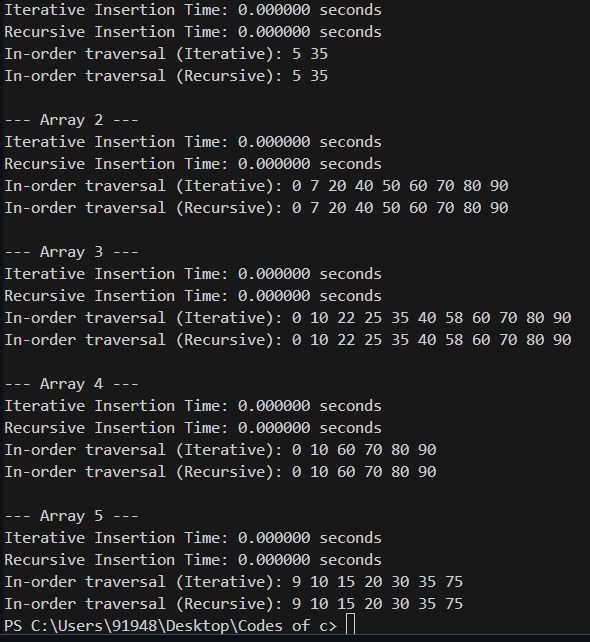
compareInsertionTimes(arrays, sizes);

return 0;

}

Graph:

Output:



**Experiment 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);

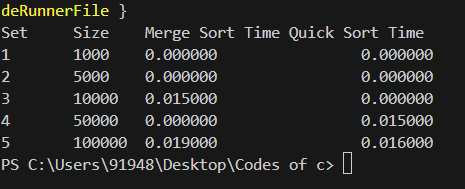
free(arr);

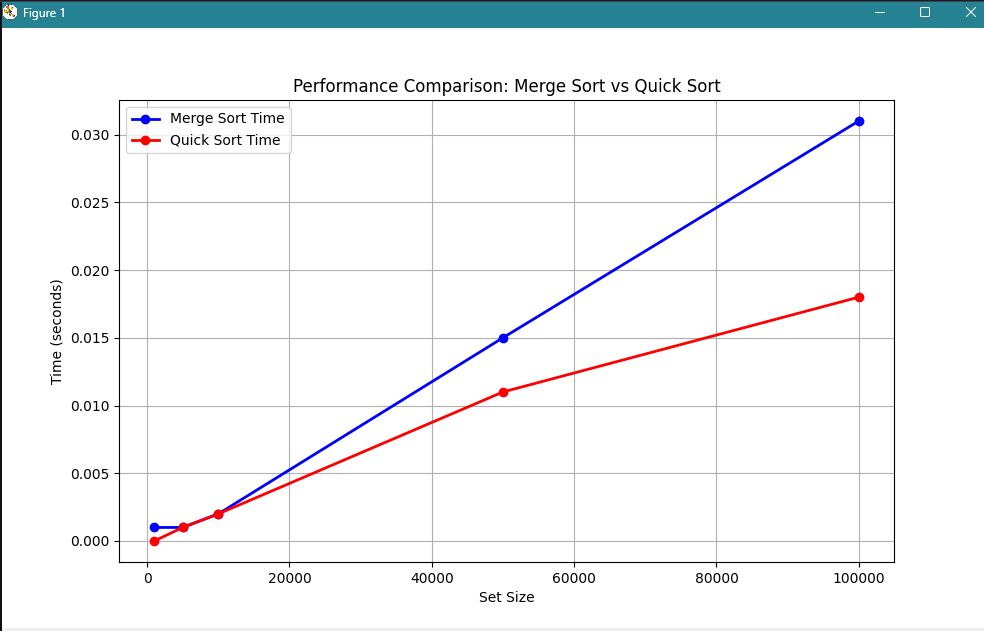
}

return 0;

}

Output:



Graph:

**Experiment 3:**

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

//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

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 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);

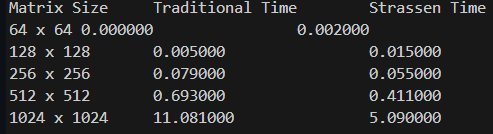
freeMatrix(C, n);

}

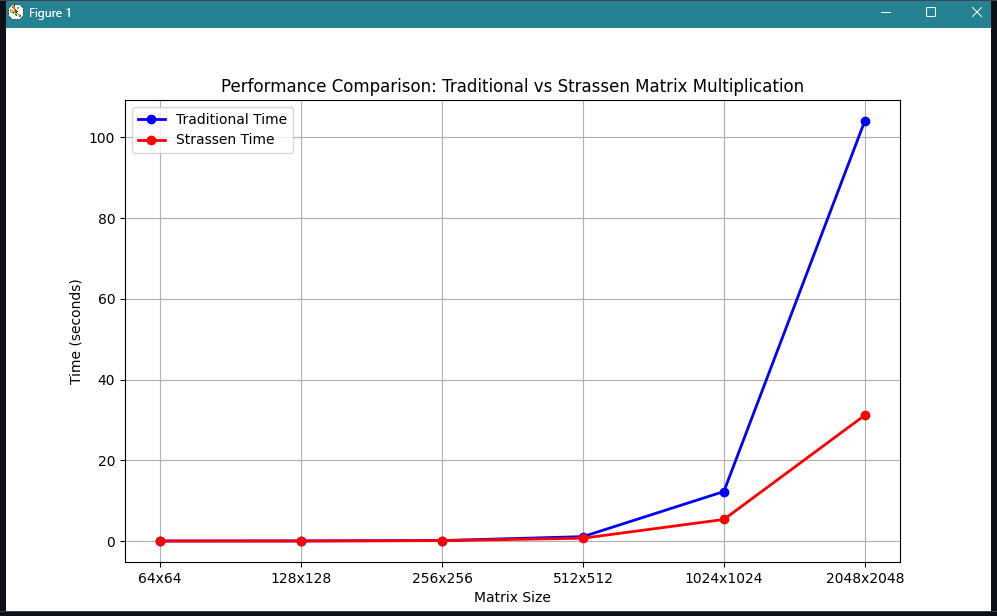
return 0;

}

Output:



Graph:



**Experiment 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};

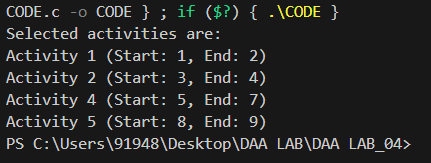
int n = sizeof(start) / sizeof(start[0]);

activitySelection(start, end, n);

return 0;

}

Output:



**Experiment 5:**

Implement the Matrix Chain Multiplication problem using Dynamic Programming.

#include <stdio.h>

#include <limits.h>

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

int m[n][n];

int i, j, k, L;

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

m[i][i] = 0;

}

for (L = 2; L < n; L++) {

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

j = i + L - 1;

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

for (k = i; k < j; k++) {

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

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

m[i][j] = q;

}

}

}

}

return m[1][n - 1];

}

int main() {

int p[] = {30, 35, 15, 5, 10};

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

int result = matrixChainOrder(p, n);

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

return 0;

}

Output:



**Experiment 6:**

Implement and compare two algorithms for finding the shortest path from a single source to all other vertices in a directed graph. The two algorithms are:

1. Dijkstra's Algorithm
2. Bellman-Ford Algorithm

#include <stdio.h>

#include <limits.h>

#define INF INT\_MAX

void dijkstra(int graph[][5], int source) {

int distance[5];

int visited[5];

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

distance[i] = INF;

visited[i] = 0;

}

distance[source] = 0;

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

int min\_distance = INF;

int min\_index = -1;

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

if (!visited[j] && distance[j] < min\_distance) {

min\_distance = distance[j];

min\_index = j;

}

}

visited[min\_index] = 1;

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

if (!visited[j] && graph[min\_index][j] != 0 && distance[min\_index] + graph[min\_index][j] < distance[j]) {

distance[j] = distance[min\_index] + graph[min\_index][j];

}

}

}

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

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

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

}

}

void bellman\_ford(int graph[][5], int source) {

int distance[5];

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

distance[i] = INF;

}

distance[source] = 0;

for (int i = 0; i < 5 - 1; i++) {

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

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

if (graph[j][k] != 0 && distance[j] + graph[j][k] < distance[k]) {

distance[k] = distance[j] + graph[j][k];

}

}

}

}

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

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

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

}

}

int main() {

int graph[][5] = {

{0, 4, 0, 0, 0},

{0, 0, 8, 0, 0},

{0, 0, 0, 7, 0},

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

{0, 0, 0, 0, 0}

};

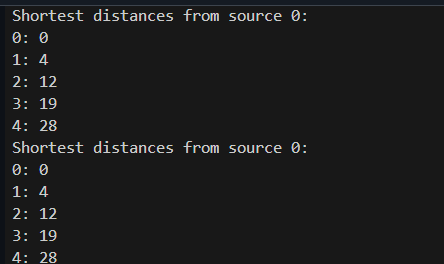
dijkstra(graph, 0);

bellman\_ford(graph, 0);

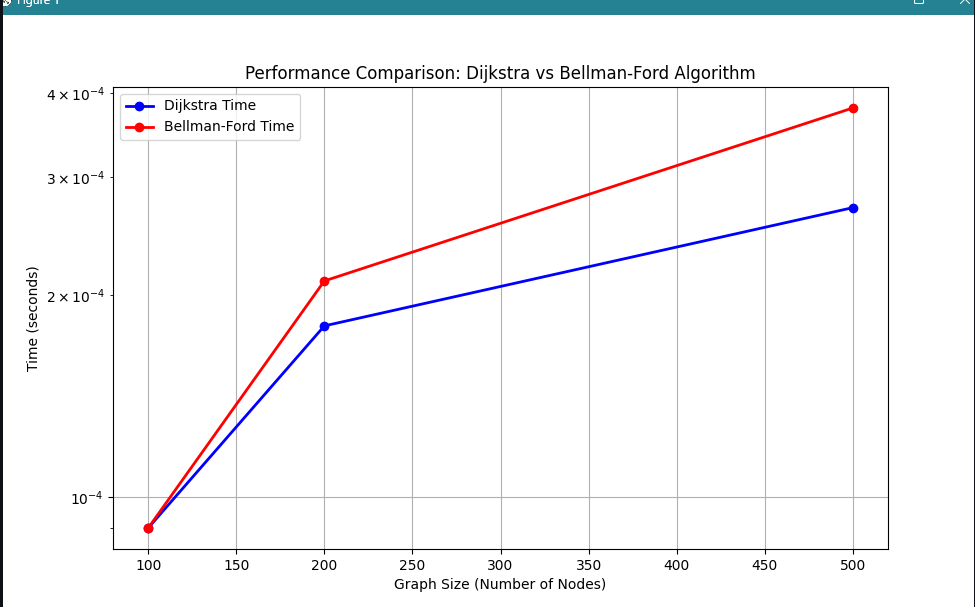
return 0;

}

Output:



Graph:



**Experiment 7:**

Solve the 0/1 Knapsack Problem using two different approaches: Greedy Approach and Dynamic Programming.

#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);

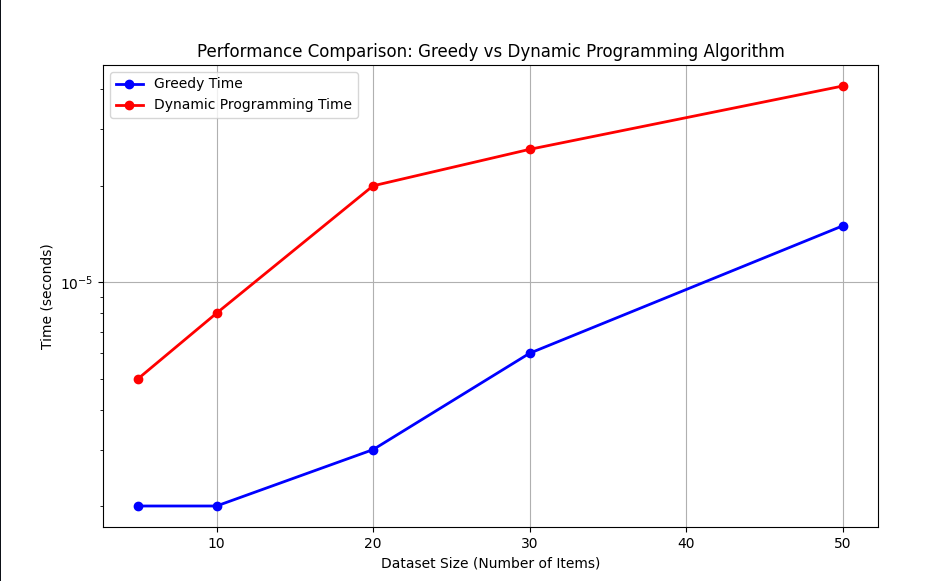
return 0;

}

Output:



Graph:



**Experiment 8:**

Solve the Subset Sum Problem, where the goal is to determine subsets of a given array that sum up to a specific target value.

#include <stdio.h>

// Function to calculate the sum of a subset

void sumOfSubsets(int arr[], int n, int sum, int index, int currentSum) {

if (index == n) {

if (currentSum == sum) {

printf("Subset with sum %d: ", sum);

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

if (arr[i] <= sum) {

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

sum -= arr[i];

}

}

printf("\n");

}

return;

}

// Include the current element in the subset

sumOfSubsets(arr, n, sum, index + 1, currentSum + arr[index]);

// Exclude the current element from the subset

sumOfSubsets(arr, n, sum, index + 1, currentSum);

}

int main() {

int arr[] = {2, 3, 5, 7};

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

int sum = 10;

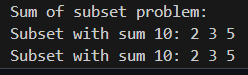
printf("Sum of subset problem:\n");

sumOfSubsets(arr, n, sum, 0, 0);

return 0;

}

Output:



**Experiment 9:**

Solve the 0/1 Knapsack Problem using three different methods: Backtracking, Branch and Bound, and Dynamic Programming.

#include <stdio.h>

// Structure to represent an item

typedef struct {

int weight;

int value;

} Item;

// Function to implement backtracking approach

void backtrackKnapsack(Item items[], int n, int capacity, int i, int totalValue, int totalWeight) {

if (i == n) {

if (totalWeight <= capacity) {

printf("Backtracking Approach: Total value = %d\n", totalValue);

}

return;

}

// Include the current item in the knapsack

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

backtrackKnapsack(items, n, capacity, i + 1, totalValue + items[i].value, totalWeight + items[i].weight);

}

// Exclude the current item from the knapsack

backtrackKnapsack(items, n, capacity, i + 1, totalValue, totalWeight);

}

// Function to implement branch and bound approach

void branchAndBoundKnapsack(Item items[], int n, int capacity, int i, int totalValue, int totalWeight, int upperBound) {

if (i == n) {

if (totalWeight <= capacity) {

printf("Branch and Bound Approach: Total value = %d\n", totalValue);

}

return;

}

// Calculate the upper bound

int newUpperBound = upperBound - items[i].value;

// Include the current item in the knapsack

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

branchAndBoundKnapsack(items, n, capacity, i + 1, totalValue + items[i].value, totalWeight + items[i].weight, newUpperBound);

}

// Exclude the current item from the knapsack

branchAndBoundKnapsack(items, n, capacity, i + 1, totalValue, totalWeight, upperBound);

}

// Function to implement dynamic programming approach

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;

printf("Backtracking Approach:\n");

backtrackKnapsack(items, n, capacity, 0, 0, 0);

printf("\nBranch and Bound Approach:\n");

branchAndBoundKnapsack(items, n, capacity, 0, 0, 0, 1000);

printf("\nDynamic Programming Approach:\n");

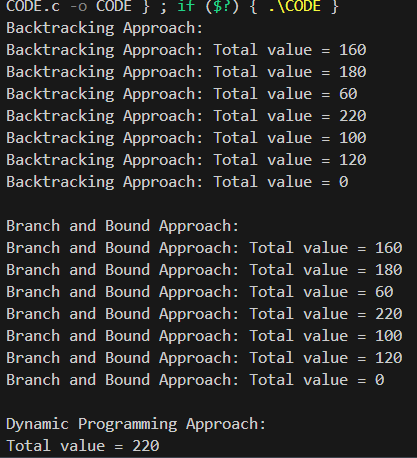
int maxValue = dynamicKnapsack(items, n, capacity);

printf("Total value = %d\n", maxValue);

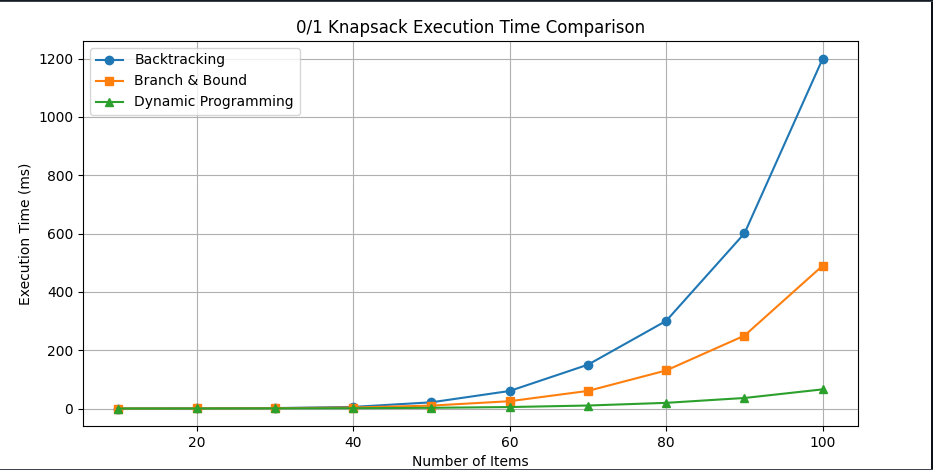
return 0;

}

Output:



Graph:



**Experiment 10:**

Demonstrate and compare three classic string matching algorithms—Naive String Matching, Rabin-Karp, and Knuth-Morris-Pratt (KMP)—using a given text and pattern.

#include <stdio.h>

#include <string.h>

#include <time.h>

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

#define q 101 // A prime number

// Naive String Matching Algorithm

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

int n = strlen(text);

int m = strlen(pattern);

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) {

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

}

}

}

// Rabin-Karp Algorithm

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

int n = strlen(text);

int m = strlen(pattern);

int p = 0; // hash value for pattern

int t = 0; // hash value for text

int h = 1;

// Calculate the value of h

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

h = (h \* d) % q;

// Calculate hash value for pattern and first window of text

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

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

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

}

// Slide the pattern over text

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

if (p == t) {

int j;

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

if (text[i + j] != pattern[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 - text[i] \* h) + text[i + m]) % q;

if (t < 0) t += q;

}

}

}

// KMP Algorithm

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++;

}

}

}

}

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

int n = strlen(text);

int m = strlen(pattern);

int lps[m];

computeLPSArray(pattern, m, lps);

int i = 0; // index for text

int j = 0; // index for pattern

while (i < n) {

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

i++;

j++;

}

if (j == m) {

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

j = lps[j - 1];

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

if (j != 0)

j = lps[j - 1];

else

i++;

}

}

}

int main() {

char text[] = "ABABDABACDABABCABAB";

char pattern[] = "ABABCABAB";

printf("Text: %s\nPattern: %s\n", text, pattern);

// Naive String Match

printf("\nRunning Naive String Matching...\n");

clock\_t start = clock();

naiveStringMatch(text, pattern);

clock\_t end = clock();

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

// Rabin-Karp

printf("\nRunning Rabin-Karp...\n");

start = clock();

rabinKarp(text, pattern);

end = clock();

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

// KMP

printf("\nRunning KMP...\n");

start = clock();

KMP(text, pattern);

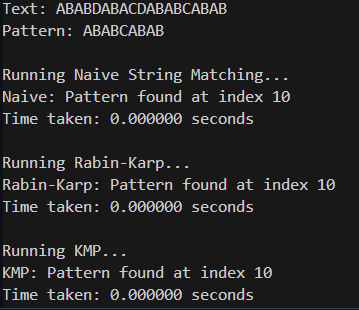
end = clock();

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

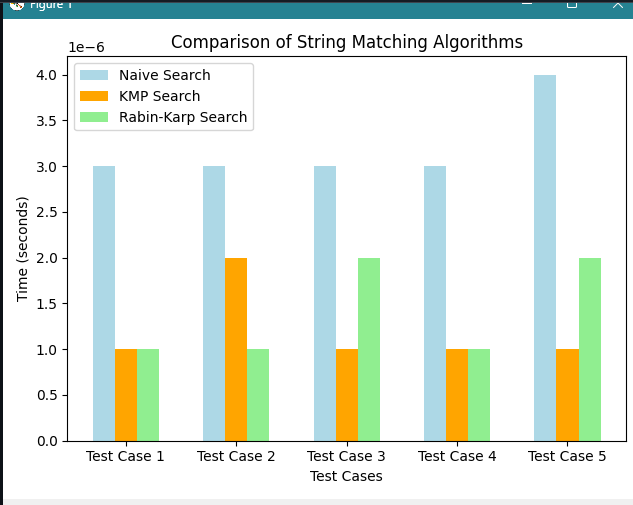
return 0;

}

Output:



Graph:



**Github Link:**