

Design and Analysis of Algorithm

**LAB REPORT**

SUBMITTED IN PARTIAL FULFILLMENT REQUIREMENT FOR THE AWARD OF DEGREE OF BACHELOR OF TECHNOLOGY

(COMPUTER SCIENCE & ENGINEERING)

SUBMITTED BY

**Yash Arora(220449)**

UNDER THE SUPERVISION OF

**DR. Meenakshi Malik**

SCHOOL OF ENGINEERING AND TECHNOLOGY

**BML MUNJAL UNIVERSITY Gurugram, Haryana - 122413**

# INDEX

|  |  |  |
| --- | --- | --- |
| Serial no | Experment | Page no. |
| 1) | Implementing graph/tree data-structures and traversal | 3-6 |
| 2) | Problem solving with sorting and searching - implementing sorting and searching algorithms | 6-9 |
| 3) | Implementing Divide-and-Conquer algorithms and solving practical problems. | 9-13 |
| 4) | Implementing Greedy algorithms and solving practical problems. | 13-19 |
| 5) | Implementing solution solving Dynamic Programming | 19-24 |
| 6) | Implement LCS problem using dynamic programming | 24-26 |
| 7) | Implementing and applying string matching algorithms for solving text/string related problem | 26-28 |
| 8) | Solving computationally hard problems with Backtracking – implementation | 28-29 |
| 9) | Practical real world problem |  |

**Experiment 1**

Aim: Implementing graph/tree data-structures and traversal.

Code:

BFS

#include <iostream> #include <queue> #include <unordered\_set>

using namespace std;

const int MAX\_VERTICES = 6;

class Graph { public:

int vertices;

int adjList[MAX\_VERTICES][MAX\_VERTICES];

Graph(int V) : vertices(V) {

for (int i = 0; i < MAX\_VERTICES; ++i) { for (int j = 0; j < MAX\_VERTICES; ++j) {

adjList[i][j] = 0;

}

}

}

void addEdge(int u, int v) { adjList[u][v] = 1;

adjList[v][u] = 1;

}

};

void BFS(const Graph& graph, int startVertex) { queue<int> q;

unordered\_set<int> visited;

q.push(startVertex); visited.insert(startVertex);

while (!q.empty()) {

int currentVertex = q.front(); q.pop();

cout << currentVertex << " ";

for (int neighbor = 0; neighbor < graph.vertices; ++neighbor) { if (graph.adjList[currentVertex][neighbor] == 1 &&

visited.find(neighbor) == visited.end()) {

q.push(neighbor); visited.insert(neighbor);

}

}

}

}

int main() {

Graph graph(6);

graph.addEdge(0, 1);

graph.addEdge(0, 2);

graph.addEdge(1, 3);

graph.addEdge(2, 4);

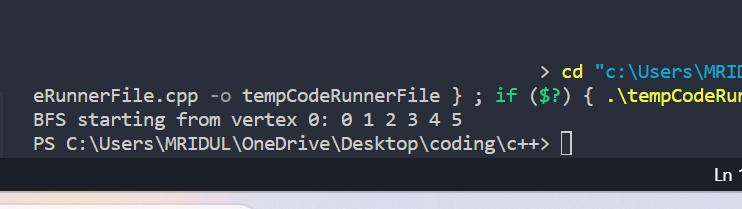
graph.addEdge(2, 5);

cout << "BFS starting from vertex 0: "; BFS(graph, 0);

return 0;

}

Output:



DFS

#include <iostream> #include <stack> #include <unordered\_set>

using namespace std;

const int MAX\_VERTICES = 6;

class Graph { public:

int vertices;

int adjList[MAX\_VERTICES][MAX\_VERTICES];

Graph(int V) : vertices(V) {

for (int i = 0; i < MAX\_VERTICES; ++i) { for (int j = 0; j < MAX\_VERTICES; ++j) {

adjList[i][j] = 0;

}

}

}

void addEdge(int u, int v) { adjList[u][v] = 1;

adjList[v][u] = 1;

}

};

void DFS(const Graph& graph, int startVertex) { stack<int> s;

unordered\_set<int> visited;

s.push(startVertex); visited.insert(startVertex);

while (!s.empty()) {

int currentVertex = s.top(); s.pop();

cout << currentVertex << " ";

for (int neighbor = 0; neighbor < graph.vertices; ++neighbor) { if (graph.adjList[currentVertex][neighbor] == 1 &&

visited.find(neighbor) == visited.end()) {

s.push(neighbor); visited.insert(neighbor);

}

}

}

}

int main() {

int numVertices, numEdges;

cout << "Enter the number of vertices: "; cin >> numVertices;

Graph graph(numVertices);

cout << "Enter the number of edges: "; cin >> numEdges;

cout << "Enter the edges (vertex pairs):\n"; for (int i = 0; i < numEdges; ++i) {

int u, v;

cout << "Edge " << i + 1 << ": "; cin >> u >> v;

graph.addEdge(u, v);

}

int startVertex;

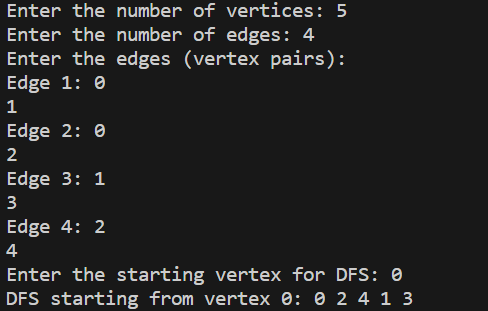
cout << "Enter the starting vertex for DFS: "; cin >> startVertex;

cout << "DFS starting from vertex " << startVertex << ": "; DFS(graph, startVertex);

return 0;

}

Output:



**Experiment 2**

Aim: Problem solving with sorting and searching implementing sorting and searching algorithms.

Code:

Heap Sort

#include <iostream> using namespace std;

void heapify(int arr[], int n, int i) { int largest = i;

int left = 2 \* i + 1; int right = 2 \* i + 2;

if (left < n && arr[left] > arr[largest]) largest = left;

if (right < n && arr[right] > arr[largest]) largest = right;

if (largest != i) { swap(arr[i], arr[largest]); heapify(arr, n, largest);

}

}

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

for (int i = n / 2 - 1; i >= 0; i--) heapify(arr, n, i);

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

swap(arr[0], arr[i]); heapify(arr, i, 0);

}

}

int binarySearch(const int arr[], int low, int high, int target) { while (low <= high) {

int mid = low + (high - low) / 2;

if (arr[mid] == target) return mid;

if (arr[mid] < target) low = mid + 1;

else

high = mid - 1;

}

return -1;

}

int main() {

int arr[] = {12, 11, 13, 5, 6, 7};

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

cout << "Original array: ";

for (int i = 0; i < arrSize; ++i) { cout << arr[i] << " ";

}

cout << endl;

heapSort(arr, arrSize);

cout << "Array after Heap Sort: "; for (int i = 0; i < arrSize; ++i) {

cout << arr[i] << " ";

}

cout << endl;

int target = 13;

int result = binarySearch(arr, 0, arrSize - 1, target);

if (result != -1)

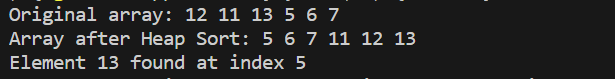
cout << "Element " << target << " found at index " << result << endl; else

cout << "Element " << target << " not found in the array" << endl;

return 0;

}

Output:



Linear Search

#include <iostream> using namespace std;

int linearSearch(const int arr[], int size, int target) { for (int i = 0; i < size; ++i) {

if (arr[i] == target) { return i;

}

}

return -1;

}

int main() {

const int maxSize = 100; int arr[maxSize];

int size;

cout << "Enter the size of the array: "; cin >> size;

if (size <= 0 || size > maxSize) {

cerr << "Invalid array size. Please enter a positive size up to " << maxSize << "." << endl;

return 1;

}

cout << "Enter the elements of the array:" << endl; for (int i = 0; i < size; ++i) {

cout << "Element " << i + 1 << ": "; cin >> arr[i];

}

int target;

cout << "Enter the element to search for: "; cin >> target;

int result = linearSearch(arr, size, target); if (result != -1) {

cout << "Element " << target << " found at index " << result << "." <<

endl;

} else {

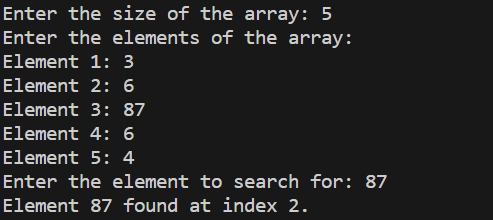
cout << "Element " << target << " not found in the array." << endl;

}

return 0;

}

Output:



**Experiment 3**

Aim: Implementing Divide-and-Conquer algorithms and solving practical problems.

Code:

Merge Sort:

#include <iostream> using namespace std;

void merge(int arr[], int left, int middle, int right) { int n1 = middle - left + 1;

int n2 = right - middle;

int L[n1], R[n2];

for (int i = 0; i < n1; i++) L[i] = arr[left + i];

for (int j = 0; j < n2; j++) R[j] = arr[middle + 1 + j];

int 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++;

}

}

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

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

mergeSort(arr, left, middle); mergeSort(arr, middle + 1, right); merge(arr, left, middle, right);

}

}

int main() {

const int maxSize = 100; int arr[maxSize];

int size;

cout << "Enter the size of the array: "; cin >> size;

if (size <= 0 || size > maxSize) {

cerr << "Invalid array size. Please enter a positive size up to " << maxSize << "." << endl;

return 1;

}

cout << "Enter the elements of the array:" << endl; for (int i = 0; i < size; ++i) {

cout << "Element " << i + 1 << ": "; cin >> arr[i];

}

mergeSort(arr, 0, size - 1);

cout << "Array after Merge Sort: "; for (int i = 0; i < size; ++i) {

cout << arr[i] << " ";

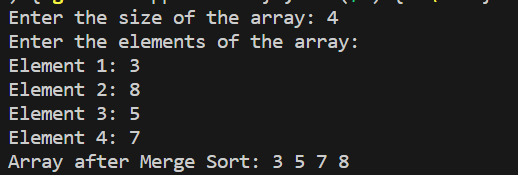
}

cout << endl;

return 0;

}

Output:



Quick Sort

#include <iostream> using namespace std;

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;

}

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

int pivotIndex = partition(arr, low, high);

quickSort(arr, low, pivotIndex - 1); quickSort(arr, pivotIndex + 1, high);

}

}

int main() {

const int maxSize = 100; int arr[maxSize];

int size;

cout << "Enter the size of the array: "; cin >> size;

if (size <= 0 || size > maxSize) {

cerr << "Invalid array size. Please enter a positive size up to " << maxSize << "." << endl;

return 1;

}

cout << "Enter the elements of the array:" << endl; for (int i = 0; i < size; ++i) {

cout << "Element " << i + 1 << ": "; cin >> arr[i];

}

quickSort(arr, 0, size - 1);

cout << "Array after Quick Sort: "; for (int i = 0; i < size; ++i) {

cout << arr[i] << " ";

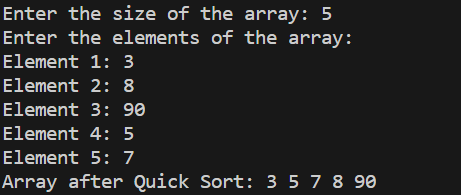
}

cout << endl;

return 0;

}

Output:



**Experiment 4**

Aim: Implementing Greedy algorithms and solving practical problems.

Prism Algorithm

#include <iostream> #include <climits>

using namespace std; const int MAX\_SIZE = 100;

int findMinKey(int key[], bool mstSet[], int vertices) { int minKey = INT\_MAX, minIndex;

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

if (!mstSet[v] && key[v] < minKey) { minKey = key[v];

minIndex = v;

}

}

return minIndex;

}

void printMST(int parent[], int graph[MAX\_SIZE][MAX\_SIZE], int vertices) { cout << "Edge \tWeight\n";

for (int i = 1; i < vertices; i++)

cout << parent[i] << " - " << i << "\t" << graph[i][parent[i]] <<

endl;

}

void primMST(int graph[MAX\_SIZE][MAX\_SIZE], int vertices) { int parent[MAX\_SIZE];

int key[MAX\_SIZE]; bool mstSet[MAX\_SIZE];

for (int i = 0; i < vertices; i++) { key[i] = INT\_MAX;

mstSet[i] = false;

}

key[0] = 0;

parent[0] = -1;

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

int u = findMinKey(key, mstSet, vertices);

mstSet[u] = true;

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

if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v]) { parent[v] = u;

key[v] = graph[u][v];

}

}

}

printMST(parent, graph, vertices);

}

int main() {

int vertices;

int graph[MAX\_SIZE][MAX\_SIZE];

cout << "Enter the number of vertices: "; cin >> vertices;

cout << "Enter the adjacency matrix (0 for no edge, positive values for weights):\n";

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

for (int j = 0; j < vertices; ++j) { cin >> graph[i][j];

}

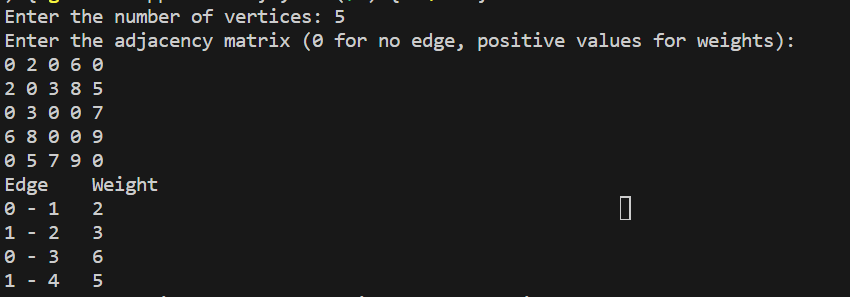
}

primMST(graph, vertices);

return 0;

}

Output:



Kruskal's Algorithm

#include <iostream> #include <algorithm>

using namespace std; const int MAX\_SIZE = 100;

struct Edge {

int src, dest, weight;

};

struct Subset {

int parent, rank;

};

int find(Subset subsets[], int i) { if (subsets[i].parent != i)

subsets[i].parent = find(subsets, subsets[i].parent); return subsets[i].parent;

}

void Union(Subset subsets[], int x, int y) { int xroot = find(subsets, x);

int yroot = find(subsets, y);

if (subsets[xroot].rank < subsets[yroot].rank) subsets[xroot].parent = yroot;

else if (subsets[xroot].rank > subsets[yroot].rank) subsets[yroot].parent = xroot;

else {

subsets[yroot].parent = xroot; subsets[xroot].rank++;

}

}

bool compareEdges(Edge a, Edge b) {

return a.weight < b.weight;

}

void kruskalMST(Edge edges[], int vertices, int edgesCount) {

Subset subsets[MAX\_SIZE];

for (int v = 0; v < vertices; v++) { subsets[v].parent = v; subsets[v].rank = 0;

}

sort(edges, edges + edgesCount, compareEdges);

Edge result[vertices]; int resultIndex = 0;

int edgeIndex = 0;

while (resultIndex < vertices - 1 && edgeIndex < edgesCount) { Edge nextEdge = edges[edgeIndex++];

int x = find(subsets, nextEdge.src); int y = find(subsets, nextEdge.dest);

if (x != y) {

result[resultIndex++] = nextEdge; Union(subsets, x, y);

}

}

cout << "Edges in the Minimum Spanning Tree:\n"; for (int i = 0; i < resultIndex; ++i) {

cout << result[i].src << " - " << result[i].dest << " \tWeight: " << result[i].weight << endl;

}

}

int main() {

int vertices, edgesCount;

cout << "Enter the number of vertices: "; cin >> vertices;

cout << "Enter the number of edges: ";

cin >> edgesCount; Edge edges[MAX\_SIZE];

cout << "Enter the edges (source destination weight):\n"; for (int i = 0; i < edgesCount; ++i) {

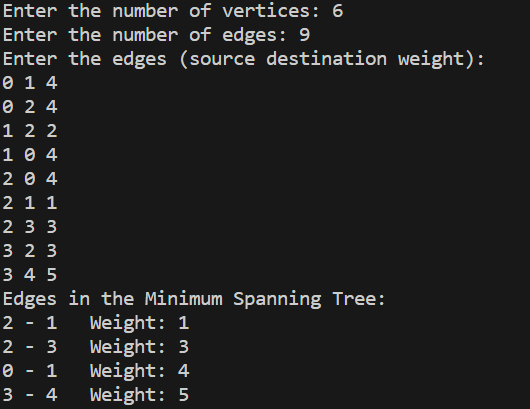
cin >> edges[i].src >> edges[i].dest >> edges[i].weight;

}

kruskalMST(edges, vertices, edgesCount); return 0;

}

Output:



Dijkstra's algo

#include <iostream> #include <climits>

using namespace std; const int MAX\_SIZE = 100;

int minDistance(int dist[], bool sptSet[], int vertices) { int minDist = INT\_MAX, minIndex;

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

if (!sptSet[v] && dist[v] <= minDist) { minDist = dist[v];

minIndex = v;

}

}

return minIndex;

}

void printSolution(int dist[], int vertices) { cout << "Vertex \t Distance from Source\n"; for (int i = 0; i < vertices; i++)

cout << i << " \t " << dist[i] << endl;

}

void dijkstra(int graph[MAX\_SIZE][MAX\_SIZE], int src, int vertices) { int dist[MAX\_SIZE];

bool sptSet[MAX\_SIZE];

for (int i = 0; i < vertices; i++) { dist[i] = INT\_MAX;

sptSet[i] = false;

}

dist[src] = 0;

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

sptSet[u] = true;

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

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

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

}

}

}

printSolution(dist, vertices);

}

int main() {

int vertices;

cout << "Enter the number of vertices: "; cin >> vertices;

int graph[MAX\_SIZE][MAX\_SIZE];

cout << "Enter the adjacency matrix (0 for no edge, positive values for weights):\n";

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

for (int j = 0; j < vertices; ++j) { cin >> graph[i][j];

}

}

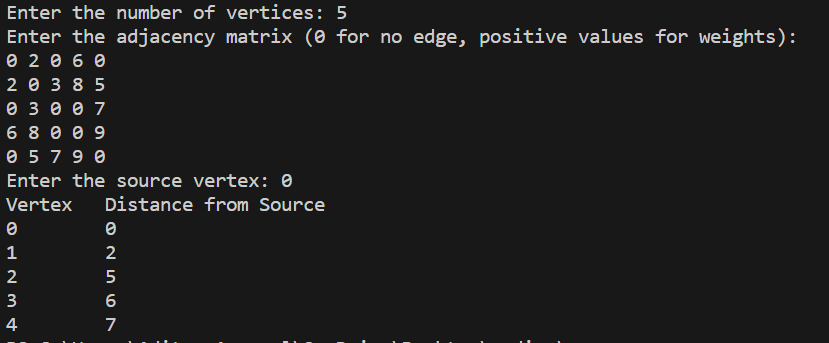
int source;

cout << "Enter the source vertex: "; cin >> source;

dijkstra(graph, source, vertices); return 0;

}

Output:



**Experiment 5**

Aim: Implementing solutions for solving Dynamic Programming problems.

0/1 kanpsack problem

#include <iostream> #include <cstring>

int knapsack(int capacity, int weights[], int values[], int n) { 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;

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

dp[i][w] = std::max(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][capacity];

}

int main() {

const int maxItems = 100; int weights[maxItems]; int values[maxItems];

int n;

int capacity;

std::cout << "Enter the number of items: "; std::cin >> n;

std::cout << "Enter the knapsack capacity: "; std::cin >> capacity;

std::cout << "Enter the weights of items:" << std::endl; for (int i = 0; i < n; i++) {

std::cin >> weights[i];

}

std::cout << "Enter the values of items:" << std::endl; for (int i = 0; i < n; i++) {

std::cin >> values[i];

}

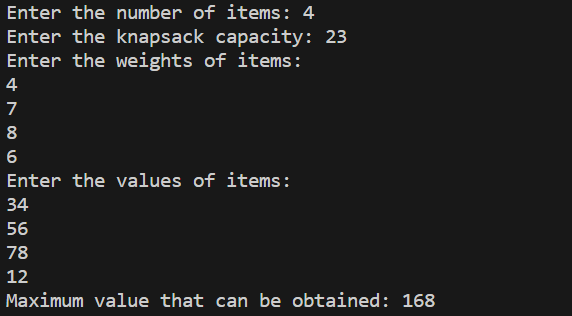
int maxValue = knapsack(capacity, weights, values, n);

std::cout << "Maximum value that can be obtained: " << maxValue << std::endl;

return 0;

}

Output:



Bellman Ford

#include <iostream> #include <climits>

const int maxVertices = 100; struct Edge {

int src, dest, weight;

};

void bellmanFord(int graph[maxVertices][maxVertices], int V, int E, int src) { int dist[V];

for (int i = 0; i < V; i++) dist[i] = INT\_MAX;

dist[src] = 0;

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

int u = graph[j][0]; int v = graph[j][1];

int weight = graph[j][2];

if (dist[u] != INT\_MAX && dist[u] + weight < dist[v]) dist[v] = dist[u] + weight;

}

}

for (int i = 0; i < E; i++) { int u = graph[i][0];

int v = graph[i][1];

int weight = graph[i][2];

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

std::cout << "Graph contains negative-weight cycle!" << std::endl; return;

}

}

std::cout << "Shortest distances from the source vertex " << src << " are:" << std::endl;

for (int i = 0; i < V; i++) { if (dist[i] == INT\_MAX)

std::cout << "Vertex " << i << ": INF" << std::endl; else

std::cout << "Vertex " << i << ": " << dist[i] << std::endl;

}

}

int main() {

int graph[maxVertices][maxVertices];

int V, E;

std::cout << "Enter the number of vertices: "; std::cin >> V;

std::cout << "Enter the number of edges: "; std::cin >> E;

std::cout << "Enter the edges (source, destination, weight):" << std::endl;

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

std::cin >> graph[i][0] >> graph[i][1] >> graph[i][2];

}

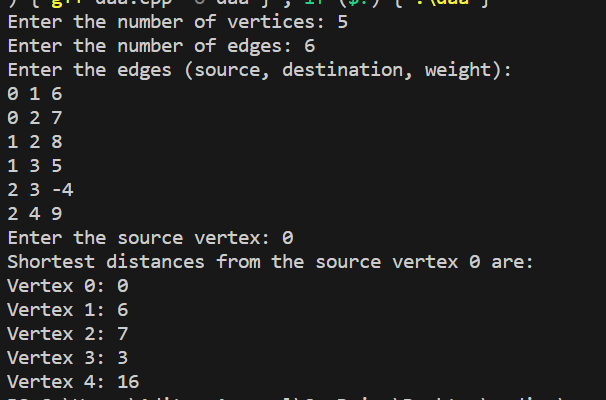
int src;

std::cout << "Enter the source vertex: "; std::cin >> src;

bellmanFord(graph, V, E, src); return 0;

}

Output:



Floyd Warshall algorithm

#include <iostream> #include <climits>

const int maxVertices = 100;

void floydWarshall(int graph[maxVertices][maxVertices], int V) { int dist[V][V];

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

for (int j = 0; j < V; j++) { dist[i][j] = graph[i][j];

}

}

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

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

if (dist[i][k] != INT\_MAX && dist[k][j] != INT\_MAX && dist[i][k] + dist[k][j] < dist[i][j]) {

dist[i][j] = dist[i][k] + dist[k][j];

}

}

}

}

std::cout << "Shortest distances between all pairs of vertices are:" << std::endl;

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

for (int j = 0; j < V; j++) { if (dist[i][j] == INT\_MAX)

std::cout << "INF\t"; else

std::cout << dist[i][j] << "\t";

}

std::cout << std::endl;

}

}

int main() {

int graph[maxVertices][maxVertices];

int V;

std::cout << "Enter the number of vertices: "; std::cin >> V;

std::cout << "Enter the adjacency matrix (use -1 for infinity):" << std::endl;

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

for (int j = 0; j < V; j++) { std::cin >> graph[i][j];

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

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

}

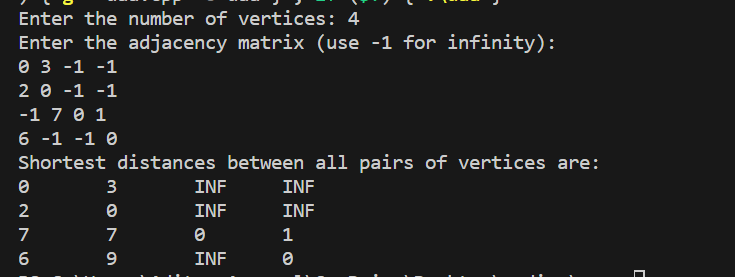
}

}

floydWarshall(graph, V); return 0;

}

Output:



**Experiment 6**

Aim: LCS problem using dynamic programming

LCS problem

#include <iostream> #include <cstring>

int lcsLength(char\* X, char\* Y, int m, int n) { int dp[m + 1][n + 1];

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

for (int j = 0; j <= n; j++) { if (i == 0 || j == 0) dp[i][j] = 0;

else if (X[i - 1] == Y[j - 1])

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

dp[i][j] = std::max(dp[i - 1][j], dp[i][j - 1]);

}

}

return dp[m][n];

}

void printLCS(char\* X, char\* Y, int m, int n) { int dp[m + 1][n + 1];

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

for (int j = 0; j <= n; j++) { if (i == 0 || j == 0) dp[i][j] = 0;

else if (X[i - 1] == Y[j - 1])

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

dp[i][j] = std::max(dp[i - 1][j], dp[i][j - 1]);

}

}

char lcs[dp[m][n] + 1];

int i = m, j = n, index = dp[m][n]; lcs[index] = '\0';

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

if (X[i - 1] == Y[j - 1]) {

lcs[--index] = X[i - 1]; i--;

j--;

} else if (dp[i - 1][j] > dp[i][j - 1]) { i--;

} else {

j--;

}

}

std::cout << "LCS: " << lcs << std::endl;

}

int main() {

const int maxStringLength = 1000; char X[maxStringLength];

char Y[maxStringLength];

std::cout << "Enter the first string: "; std::cin.getline(X, maxStringLength);

std::cout << "Enter the second string: "; std::cin.getline(Y, maxStringLength);

int m = strlen(X); int n = strlen(Y);

int length = lcsLength(X, Y, m, n);

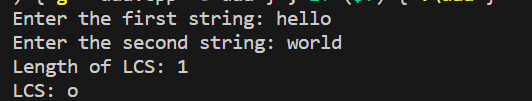
std::cout << "Length of LCS: " << length << std::endl; if (length > 0)

printLCS(X, Y, m, n);

return 0;

}

Output:



**Experiment 7**

Aim: Implementing and applying string matching algorithms for solving text/string related problems

KMP algorithm

#include <iostream> #include <cstring>

void computeLPSArray(char\* pattern, int m, int\* lps) { int len = 0;

lps[0] = 0; int i = 1;

while (i < m) {

if (pattern[i] == pattern[len]) { len++;

lps[i] = len; i++;

} else {

if (len != 0) {

len = lps[len - 1];

} else {

lps[i] = 0; i++;

}

}

}

}

void KMPSearch(char\* pattern, char\* text) { int m = strlen(pattern);

int n = strlen(text);

int lps[m]; computeLPSArray(pattern, m, lps);

int i = 0; int j = 0;

while (i < n) {

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

i++;

}

if (j == m) {

std::cout << "Pattern found at index " << i - j << std::endl; j = lps[j - 1];

} else if (i < n && pattern[j] != text[i]) { if (j != 0)

j = lps[j - 1]; else

i = i + 1;

}

}

}

int main() {

const int maxTextSize = 1000; char text[maxTextSize];

char pattern[maxTextSize];

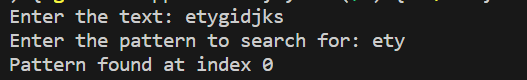
std::cout << "Enter the text: "; std::cin.getline(text, maxTextSize);

std::cout << "Enter the pattern to search for: "; std::cin.getline(pattern, maxTextSize);

KMPSearch(pattern, text); return 0;

}

Output:



**Experiment 8**

Aim: Solving computationally hard problem with backtracking implemention

N-Queen problem

#include <iostream> #define N 4

void printSolution(int board[N][N]) { for (int i = 0; i < N; i++) {

for (int j = 0; j < N; j++) std::cout << board[i][j] << " ";

std::cout << std::endl;

}

}

bool isSafe(int board[N][N], int row, int col) { for (int i = 0; i < col; i++)

if (board[row][i]) return false;

for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) if (board[i][j])

return false;

for (int i = row, j = col; i < N && j >= 0; i++, j--) if (board[i][j])

return false;

return true;

}

bool solveNQueens(int board[N][N], int col) { if (col == N) {

printSolution(board); return true;

}

bool res = false;

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

if (isSafe(board, i, col)) { board[i][col] = 1;

res = solveNQueens(board, col + 1) || res; board[i][col] = 0;

}

}

return res;

}

int board[N][N] = {0};

if (!solveNQueens(board, 0))

std::cout << "Solution does not exist" << std::endl;

}

int main() {

solveNQueens(); return 0;

}

Output:

