1. **Write a program to implement Quick sort algorithm for sorting a list of integers in ascending order**

#include <iostream>

using namespace std;

void swap(int& a, int& b) {

int temp = a;

a = b;

b = temp;

}

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

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

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

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

if (arr[j] < pivot) {

i++;

swap(arr[i], arr[j]); // Swap if element is smaller than pivot

}

}

swap(arr[i + 1], arr[high]); // Move pivot to correct position

return i + 1;

}

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

if (low < high) {

int pi = partition(arr, low, high); // Partitioning index

quickSort(arr, low, pi - 1); // Recursively sort left side

quickSort(arr, pi + 1, high); // Recursively sort right side

}

}

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

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

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

cout << endl;

}

int main() {

int arr[] = {34, 7, 23, 32, 5, 62};

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

cout << "Original array: ";

printArray(arr, n);

quickSort(arr, 0, n - 1);

cout << "Sorted array: ";

printArray(arr, n);

return 0;

}

Output:

Original array: 34 7 23 32 5 62

1. **Implement Merge sort algorithm for sorting a list of integers in ascending order.**

#include <iostream>

using namespace std;

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

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

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

int leftArr[n1], rightArr[n2];

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

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

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

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

int i = 0; // Initial index of left subarray

int j = 0; // Initial index of right subarray

int k = left; // Initial index of merged subarray

while (i < n1 && j < n2) {

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

arr[k] = leftArr[i];

i++;

} else {

arr[k] = rightArr[j];

j++;

}

k++;

}

while (i < n1) {

arr[k] = leftArr[i];

i++;

k++;

}

while (j < n2) {

arr[k] = rightArr[j];

j++;

k++;

}

}

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

if (left < right) {

int mid = left + (right - left) / 2; // Find the middle point

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

merge(arr, left, mid, right);

}

}

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

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

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

cout << endl;

}

int main() {

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

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

cout << "Original array: ";

printArray(arr, n);

mergeSort(arr, 0, n - 1);

cout << "Sorted array: ";

printArray(arr, n);

return 0;

}

Output:

Original array: 38 27 43 3 9 82 10

Sorted array: 3 9 10 27 38 43 82

1. **Write a program to find Maximum and Minimum element in an array using Divide and Conquer strategy**

#include <iostream>

using namespace std;

struct Pair {

int min;

int max;

};

Pair findMinMax(int arr[], int low, int high) {

Pair result, leftResult, rightResult;

if (low == high) {

result.max = arr[low];

result.min = arr[low];

return result;

}

if (high == low + 1) {

if (arr[low] > arr[high]) {

result.max = arr[low];

result.min = arr[high];

} else {

result.max = arr[high];

result.min = arr[low];

}

return result;

}

int mid = (low + high) / 2;

leftResult = findMinMax(arr, low, mid);

rightResult = findMinMax(arr, mid + 1, high);

result.min = (leftResult.min < rightResult.min) ? leftResult.min : rightResult.min;

result.max = (leftResult.max > rightResult.max) ? leftResult.max : rightResult.max;

return result;

}

int main() {

int arr[] = {12, 5, 18, 7, 3, 20, 10};

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

Pair minMax = findMinMax(arr, 0, n - 1);

cout << "Minimum element: " << minMax.min << endl;

cout << "Maximum element: " << minMax.max << endl;

return 0;

}

Output:

Minimum element: 3

Maximum element:20

1. **Write a program to implement Bellman-Ford Algorithm using Dynamic Programming**

#include <iostream>

#include <vector>

#include <climits>

using namespace std;

struct Edge {

int u, v, weight; // u -> source, v -> destination, weight -> edge weight

};

void bellmanFord(int V, int E, vector<Edge>& edges, int src) {

vector<int> dist(V, INT\_MAX);

dist[src] = 0;

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

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

int u = edges[j].u;

int v = edges[j].v;

int weight = edges[j].weight;

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

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

}

}

}

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

int u = edges[j].u;

int v = edges[j].v;

int weight = edges[j].weight;

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

cout << "Graph contains negative-weight cycle" << endl;

return;

}

}

cout << "Vertex\tDistance from Source" << endl;

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

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

}

}

int main() {

int V = 5; // Number of vertices in the graph

int E = 8; // Number of edges in the graph

vector<Edge> edges = {

{0, 1, -1},

{0, 2, 4},

{1, 2, 3},

{1, 3, 2},

{1, 4, 2},

{3, 2, 5},

{3, 1, 1},

{4, 3, -3}

};

int src = 0; // Source vertex

bellmanFord(V, E, edges, src);

return 0;

}

Output:

Vertex Distance from Source

0 0

1 -1

2 2

3 -2

4 1

1. **Write a program to solve the travelling salesman problem and to print the path and the cost using Dynamic Programming**

#include <iostream>

#include <vector>

#include <climits>

using namespace std;

const int N = 4;

int tsp(int mask, int pos, vector<vector<int>>& dist, vector<vector<int>>& dp) {

// Base case: all cities have been visited

if (mask == (1 << N) - 1) {

return dist[pos][0]; // Return to the starting city

} if (dp[mask][pos] != -1) {

return dp[mask][pos];

}

int ans = INT\_MAX;

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

if ((mask & (1 << city)) == 0) { // Check if the city is unvisited

int newAns = dist[pos][city] + tsp(mask | (1 << city), city, dist, dp);

ans = min(ans, newAns);

}

}

return dp[mask][pos] = ans; // Memoize and return the minimum cost

}

void printOptimalPath(vector<vector<int>>& dist, vector<vector<int>>& dp) {

int mask = 1; // Start from the first city (0th city)

int pos = 0; // Starting city is 0

cout << "Optimal Path: " << pos << " -> ";

while (mask != (1 << N) - 1) {

int nextCity = -1;

int minCost = INT\_MAX;

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

if ((mask & (1 << city)) == 0) {

int cost = dist[pos][city] + dp[mask | (1 << city)][city];

if (cost < minCost) {

minCost = cost;

nextCity = city;

}

}

}

cout << nextCity << " -> ";

pos = nextCity;

mask |= (1 << nextCity);

}

cout << "0" << endl; // End at the starting city

}

int main() {

// Distance matrix (graph)

vector<vector<int>> dist = {

{0, 10, 15, 20},

{10, 0, 35, 25},

{15, 35, 0, 30},

{20, 25, 30, 0}

};

vector<vector<int>> dp(1 << N, vector<int>(N, -1));

int minCost = tsp(1, 0, dist, dp);

cout << "Minimum cost: " << minCost << endl;

printOptimalPath(dist, dp);

return 0;

}

**Output:**

**Minimum cost: 80**

**Optimal Path: 0 -> 1 -> 3 -> 2 -> 0**

1. **Find Minimum Cost Spanning Tree of a given connected undirected graph using Kruskal's algorithm.**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

struct Edge {

int u, v, weight;

bool operator<(const Edge& other) const {

return weight < other.weight; // Sorting by weight

}

};

class DisjointSet {

private:

vector<int> parent, rank;

public:

DisjointSet(int n) {

parent.resize(n);

rank.resize(n, 0);

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

parent[i] = i; // Initially, each node is its own parent

}

}

int find(int u) {

if (u != parent[u]) {

parent[u] = find(parent[u]); // Path compression

}

return parent[u];

}

void unionSets(int u, int v) {

int rootU = find(u);

int rootV = find(v);

if (rootU != rootV) {

// Union by rank

if (rank[rootU] > rank[rootV]) {

parent[rootV] = rootU;

} else if (rank[rootU] < rank[rootV]) {

parent[rootU] = rootV;

} else {

parent[rootV] = rootU;

rank[rootU]++;

}

}

}

};

int kruskalMST(vector<Edge>& edges, int V) {

sort(edges.begin(), edges.end());

DisjointSet ds(V);

int mstCost = 0;

vector<Edge> mstEdges;

for (const auto& edge : edges) {

int u = edge.u;

int v = edge.v;

int weight = edge.weight;

if (ds.find(u) != ds.find(v)) {

// If not, include this edge in MST

ds.unionSets(u, v);

mstCost += weight;

mstEdges.push\_back(edge);

}

}

cout << "Edges in the Minimum Cost Spanning Tree:" << endl;

for (const auto& edge : mstEdges) {

cout << edge.u << " - " << edge.v << " : " << edge.weight << endl;

}

return mstCost;

}

int main() {

int V = 4; // Number of vertices

int E = 5; // Number of edges

vector<Edge> edges = {

{0, 1, 10},

{0, 2, 6},

{0, 3, 5},

{1, 3, 15},

{2, 3, 4}

};

1. **Find Minimum Cost Spanning Tree of undirected graph using Prim’s algorithm.**

#include <iostream>

#include <vector>

#include <queue>

#include <climits>

using namespace std;

typedef pair<int, int> pii;

int primMST(int V, vector<vector<pii>>& adj) {

priority\_queue<pii, vector<pii>, greater<pii>> pq;

vector<int> key(V, INT\_MAX); // Stores the minimum weight to add each vertex to MST

vector<bool> inMST(V, false); // To track vertices included in the MST

vector<int> parent(V, -1); // Stores the parent of each vertex in MST

int start = 0; // Starting vertex

key[start] = 0;

pq.push({0, start}); // Push the starting vertex into the priority queue

int mstCost = 0;

while (!pq.empty()) {

int u = pq.top().second; // Get the vertex with the smallest key value

pq.pop();

if (inMST[u]) {

continue; // Skip if already included in MST

}

inMST[u] = true; // Include this vertex in MST

mstCost += key[u]; // Add its key value to the total MST cost

for (auto& edge : adj[u]) {

int v = edge.first;

int weight = edge.second;

if (!inMST[v] && weight < key[v]) {

key[v] = weight;

pq.push({key[v], v});

parent[v] = u; // Track the parent vertex in MST

}

}

}

cout << "Edges in the Minimum Cost Spanning Tree:" << endl;

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

cout << parent[i] << " - " << i << " : " << key[i] << endl;

}

return mstCost;

}

int main() {

int V = 5; // Number of vertices

vector<vector<pii>> adj(V);

adj[0].push\_back({1, 2});

adj[0].push\_back({3, 6});

adj[1].push\_back({0, 2});

adj[1].push\_back({2, 3});

adj[1].push\_back({3, 8});

adj[1].push\_back({4, 5});

adj[2].push\_back({1, 3});

adj[2].push\_back({4, 7});

adj[3].push\_back({0, 6});

adj[3].push\_back({1, 8});

adj[4].push\_back({1, 5});

adj[4].push\_back({2, 7});

int minCost = primMST(V, adj);

cout << "Minimum cost of spanning tree: " << minCost << endl;

return 0;

}

**Output:**

**Edges in the Minimum Cost Spanning Tree:**

**0 - 1 : 2**

**1 - 2 : 3**

**0 - 3 : 6**

**1 - 4 : 5**

**Minimum cost of spanning tree: 16**

1. **Write a recursive program to find the solution of placing n queens on chessboard so that no two queens attack each other using Backtracking.**

#include <iostream>

#include <vector>

using namespace std;

void printBoard(vector<vector<int>>& board) {

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

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

if (board[i][j] == 1)

cout << "Q "; // Queen position

else

cout << ". "; // Empty position

}

cout << endl;

}

cout << endl;

}

bool isSafe(vector<vector<int>>& board, int row, int col) {

// Check the same column

for (int i = 0; i < row; i++)

if (board[i][col] == 1)

return false;

for (int i = row, j = col; i >= 0 && j >= 0; i--, j--)

if (board[i][j] == 1)

return false;

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

if (board[i][j] == 1)

return false;

return true;

}

bool solveNQueensUtil(vector<vector<int>>& board, int row) {

// If all queens are placed

if (row >= N)

return true;

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

if (isSafe(board, row, col)) {

// Place the queen

board[row][col] = 1;

if (solveNQueensUtil(board, row + 1))

return true; // If successful, return true

board[row][col] = 0; // Backtrack and remove the queen

}

}

return false; // No valid placement found for the current row

}

bool solveNQueens() {

// Initialize the chessboard with all 0s (no queens placed)

vector<vector<int>> board(N, vector<int>(N, 0));

if (solveNQueensUtil(board, 0)) {

// If a solution is found, print the board

printBoard(board);

return true;

} else {

cout << "Solution does not exist." << endl;

return false;

}

}

int main() {

solveNQueens();

return 0;

}

Output:

Q . . . . . . .

. . . . Q . . .

. . . . . . . Q

. . . Q . . . .

. . . . . Q . .

. Q . . . . . .

. . . . . . Q .

. . Q . . . . .