**Quick sort and optimal merge pattern**

#include <iostream>

#include <queue>

#include <vector>

#include <algorithm>

using namespace std;

// Quick Sort function

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

if (low < 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]);

int pi = i + 1;

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

// Optimal Merge Pattern function

int optimalMergePattern(int files[], int n) {

priority\_queue<int, vector<int>, greater<int>> minHeap;

int totalCost = 0;

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

minHeap.push(files[i]);

}

while (minHeap.size() > 1) {

int first = minHeap.top(); minHeap.pop();

int second = minHeap.top(); minHeap.pop();

int mergedFile = first + second;

totalCost += mergedFile;

minHeap.push(mergedFile);

}

return totalCost;

}

// Print array function

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

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

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

}

cout << endl;

}

int main() {

int numFiles;

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

cin >> numFiles;

if (numFiles > 100) {

cout << "Number of files exceeds the limit (100)." << endl;

return 1;

}

int files[100];

cout << "Enter the weight of each file:\n";

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

cout << "File " << i << ": ";

cin >> files[i];

}

// Step 1: Sort files using Quick Sort

quickSort(files, 0, numFiles - 1);

cout << "\nFiles after Quick Sort: ";

printArray(files, numFiles);

// Step 2: Use Optimal Merge Pattern on the sorted files

int result = optimalMergePattern(files, numFiles);

cout << "\nMinimum cost to merge files: " << result << endl;

return 0;

}

**Prims and kruskal**

#include <iostream>

#include <vector>

#include <algorithm>

#include <climits>

#include <utility>

#include <queue>

#include <tuple>

using namespace std;

class Graph {

public:

int V; // Number of vertices

vector<vector<pair<int, int>>> adj; // Adjacency list for Prim's (node, weight)

vector<tuple<int, int, int>> edges; // Edge list for Kruskal's (weight, u, v)

Graph(int V) : V(V) {

adj.resize(V);

}

// Add edge function for both Prim's and Kruskal's

void addEdge(int u, int v, int w) {

adj[u].emplace\_back(v, w);

adj[v].emplace\_back(u, w);

edges.emplace\_back(w, u, v);

}

// Prim's Algorithm

void primMST() {

priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

vector<int> key(V, INT\_MAX); // Key values used to pick minimum weight edge

vector<bool> inMST(V, false); // To represent vertices not yet included in MST

vector<int> parent(V, -1); // Array to store constructed MST

int start = 0; // Starting node

pq.push({0, start});

key[start] = 0;

while (!pq.empty()) {

int u = pq.top().second;

pq.pop();

inMST[u] = true;

// Process all adjacent vertices of u

for (auto &[v, weight] : adj[u]) {

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

key[v] = weight;

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

parent[v] = u;

}

}

}

cout << "Prim's MST edges (u - v):\n";

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

cout << parent[i] << " - " << i << " with weight " << key[i] << endl;

}

}

// Find function for union-find (used in Kruskal's)

int findParent(int u, vector<int> &parent) {

if (parent[u] != u) {

parent[u] = findParent(parent[u], parent); // Path compression

}

return parent[u];

}

// Union function for union-find (used in Kruskal's)

void unionSets(int u, int v, vector<int> &parent, vector<int> &rank) {

int rootU = findParent(u, parent);

int rootV = findParent(v, parent);

if (rootU != rootV) {

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

parent[rootV] = rootU;

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

parent[rootU] = rootV;

} else {

parent[rootV] = rootU;

rank[rootU]++;

}

}

}

// Kruskal's Algorithm

void kruskalMST() {

sort(edges.begin(), edges.end()); // Sort edges by weight

vector<int> parent(V);

vector<int> rank(V, 0);

int totalCost = 0;

// Initialize each vertex to be its own parent (for union-find)

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

parent[i] = i;

}

cout << "Kruskal's MST edges (u - v):\n";

for (auto &[weight, u, v] : edges) {

int rootU = findParent(u, parent);

int rootV = findParent(v, parent);

// If u and v are not in the same set, include this edge

if (rootU != rootV) {

cout << u << " - " << v << " with weight " << weight << endl;

totalCost += weight;

unionSets(u, v, parent, rank);

}

}

cout << "Total weight of Kruskal's MST: " << totalCost << endl;

}

};

int main() {

int V, E;

cout << "Enter number of vertices and edges: ";

cin >> V >> E;

Graph graph(V);

cout << "Enter the edges (u v w) where u and v are vertices and w is weight:\n";

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

int u, v, w;

cin >> u >> v >> w;

graph.addEdge(u, v, w);

}

cout << "\nRunning Prim's Algorithm:\n";

graph.primMST();

cout << "\nRunning Kruskal's Algorithm:\n";

graph.kruskalMST();

return 0;

}

**Floydd and dijkstras**

#include <iostream>

#include <vector>

#include <limits>

#include <algorithm>

#include <string>

using namespace std;

class FloydWarshall {

public:

const int INF = numeric\_limits<int>::max();

void floydWarshall(vector<vector<int>>& weight) {

int n = weight.size();

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

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

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

if (weight[i][k] != INF && weight[k][j] != INF && weight[i][k] + weight[k][j] < weight[i][j]) {

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

}

}

}

}

printSolution(weight);

}

void printSolution(const vector<vector<int>>& weight) {

int n = weight.size();

cout << "\nAll-pairs shortest paths (Floyd-Warshall):" << endl;

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

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

if (weight[i][j] == INF) {

cout << "INF ";

} else {

cout << weight[i][j] << " ";

}

}

cout << endl;

}

}

};

int minDistance(const vector<int>& dist, const vector<bool>& sptSet, int V) {

int min = numeric\_limits<int>::max(), min\_index = -1;

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

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

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

return min\_index;

}

void printDijkstraSolution(const vector<int>& dist, const vector<string>& vertices) {

cout << "\nVertex \t Distance from Source (Dijkstra):" << endl;

for (size\_t i = 0; i < dist.size(); i++) {

cout << vertices[i] << " \t\t " << (dist[i] == numeric\_limits<int>::max() ? "INF" : to\_string(dist[i])) << endl;

}

}

void dijkstra(const vector<vector<int>>& graph, int src, const vector<string>& vertices) {

int V = graph.size();

vector<int> dist(V, numeric\_limits<int>::max());

vector<bool> sptSet(V, false);

dist[src] = 0;

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

int u = minDistance(dist, sptSet, V);

sptSet[u] = true;

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

if (!sptSet[v] && graph[u][v] && dist[u] != numeric\_limits<int>::max() && dist[u] + graph[u][v] < dist[v])

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

}

printDijkstraSolution(dist, vertices);

}

int main() {

int V;

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

cin >> V;

vector<string> vertices(V);

cout << "Enter the vertex names:" << endl;

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

cout << "Vertex " << i + 1 << ": ";

cin >> vertices[i];

}

vector<vector<int>> graph(V, vector<int>(V));

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

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

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

int input;

cin >> input;

graph[i][j] = (input == -1) ? numeric\_limits<int>::max() : input;

}

}

cout << "\nChoose an algorithm:\n1. Dijkstra's Algorithm (Single-source shortest path)\n2. Floyd-Warshall Algorithm (All-pairs shortest path)\n";

int choice;

cin >> choice;

if (choice == 1) {

string srcVertex;

cout << "Enter the source vertex: ";

cin >> srcVertex;

auto it = find(vertices.begin(), vertices.end(), srcVertex);

if (it == vertices.end()) {

cout << "Invalid source vertex!" << endl;

return -1;

}

int src = distance(vertices.begin(), it);

dijkstra(graph, src, vertices);

} else if (choice == 2) {

FloydWarshall fw;

fw.floydWarshall(graph);

} else {

cout << "Invalid choice!" << endl;

}

return 0;

}

**Fractional knapsack and 0/1 knapsack**

#include <iostream>

#include <vector>

#include <iomanip>

#include <algorithm>

using namespace std;

// Function to calculate profit-to-weight ratio

double profitRatio(int profit, int weight) {

return (double)profit / weight;

}

// Fractional Knapsack-like sorting function to sort items by profit-to-weight ratio

bool compare(int i, int j, const vector<int>& profit, const vector<int>& weights) {

return profitRatio(profit[i], weights[i]) > profitRatio(profit[j], weights[j]);

}

// 0/1 Knapsack Dynamic Programming function

int knapsack(int capacity, const vector<int>& weights, const vector<int>& profits) {

int n = weights.size();

vector<vector<int>> dp(n + 1, vector<int>(capacity + 1, 0));

// DP table filling for 0/1 Knapsack

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

for (int w = 1; w <= capacity; ++w) {

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

dp[i][w] = max(dp[i - 1][w], profits[i - 1] + dp[i - 1][w - weights[i - 1]]);

} else {

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

}

}

}

// Display DP Table

cout << "\nDP Table:\n";

for (const auto& row : dp) {

for (int val : row) {

cout << setw(5) << val << " ";

}

cout << endl;

}

// Backtracking to find selected items

int maxProfit = dp[n][capacity];

cout << "\nSelected items in the knapsack:\n";

int w = capacity;

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

if (maxProfit != dp[i - 1][w]) {

cout << "Item " << i << ": (Profit: " << profits[i - 1] << ", Weight: " << weights[i - 1] << ")\n";

maxProfit -= profits[i - 1];

w -= weights[i - 1];

}

}

return dp[n][capacity];

}

// Combined Knapsack function

int combinedKnapsack(int capacity, vector<int>& weights, vector<int>& profits) {

int n = weights.size();

// Step 1: Sort items based on profit-to-weight ratio (Fractional Knapsack approach)

vector<int> indices(n);

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

indices[i] = i;

}

sort(indices.begin(), indices.end(), [&profits, &weights](int a, int b) {

return profitRatio(profits[a], weights[a]) > profitRatio(profits[b], weights[b]);

});

// Step 2: Create sorted weight and profit arrays

vector<int> sortedWeights(n), sortedProfits(n);

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

sortedWeights[i] = weights[indices[i]];

sortedProfits[i] = profits[indices[i]];

}

// Step 3: Apply 0/1 Knapsack DP on sorted items

return knapsack(capacity, sortedWeights, sortedProfits);

}

int main() {

int capacity, n;

cout << "Enter the maximum capacity of the knapsack: ";

cin >> capacity;

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

cin >> n;

vector<int> weights(n), profits(n);

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

cout << "Enter the weight of item " << (i + 1) << ": ";

cin >> weights[i];

cout << "Enter the profit of item " << (i + 1) << ": ";

cin >> profits[i];

}

int maxProfit = combinedKnapsack(capacity, weights, profits);

cout << "\nMaximum profit for the knapsack = " << maxProfit << endl;

return 0;

}

**N queens backtracking**

#include <iostream>

#include <vector>

using namespace std;

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

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 solveNQueensUtil(vector<vector<int>>& board, int col, int n) {

if (col >= n) return true;

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

if (isSafe(board, i, col, n)) {

board[i][col] = 1;

if (solveNQueensUtil(board, col + 1, n)) return true;

board[i][col] = 0; // backtrack

}

}

return false;

}

void solveNQueens(int n) {

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

if (solveNQueensUtil(board, 0, n)) {

cout << "Solution:\n";

for (const auto& row : board) {

for (int cell : row) cout << (cell ? "Q " : ". ");

cout << endl;

}

} else {

cout << "No solution exists.\n";

}

}

int main() {

int n;

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

cin >> n;

solveNQueens(n);

return 0;

}

**Merge sort quick sort**

#include <iostream>

#include <vector>

#include <cstdlib>

#include <ctime>

using namespace std;

void merge(vector<int>& arr, int left, int mid, int right) {

vector<int> temp;

int i = left, j = mid + 1;

while (i <= mid && j <= right) {

if (arr[i] <= arr[j]) temp.push\_back(arr[i++]);

else temp.push\_back(arr[j++]);

}

while (i <= mid) temp.push\_back(arr[i++]);

while (j <= right) temp.push\_back(arr[j++]);

for (int k = 0; k < temp.size(); ++k) arr[left + k] = temp[k];

}

void mergeSort(vector<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);

}

}

int partition(vector<int>& arr, int low, int high) {

int pivot = arr[high];

int i = low - 1;

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

if (arr[j] <= pivot) {

i++;

swap(arr[i], arr[j]);

}

}

swap(arr[i + 1], arr[high]);

return i + 1;

}

void quickSort(vector<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);

}

}

int main() {

int n;

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

cin >> n;

vector<int> arr1(n), arr2(n);

cout << "Enter elements: ";

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

cin >> arr1[i];

arr2[i] = arr1[i];

}

mergeSort(arr1, 0, n - 1);

quickSort(arr2, 0, n - 1);

cout << "\nSorted array using Merge Sort: ";

for (int num : arr1) cout << num << " ";

cout << "\nSorted array using Quick Sort: ";

for (int num : arr2) cout << num << " ";

return 0;

}