**1.**

/\* Merge sort\*/

#include <iostream>

using namespace std;

void Merge(int arr[], int low, int mid, int high) {

    int n1 = mid - low + 1;

    int n2 = high - mid;

    int left[n1], right[n2];

    for (int i = 0; i < n1; i++)

        left[i] = arr[low + i];

    for (int j = 0; j < n2; j++)

        right[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = low;

    while (i < n1 && j < n2) {

        if (left[i] <= right[j]) {

            arr[k] = left[i];

            i++;

        } else {

            arr[k] = right[j];

            j++;

        }

        k++;

    }

    while (i < n1) {

        arr[k] = left[i];

        i++;

        k++;

    }

    while (j < n2) {

        arr[k] = right[j];

        j++;

        k++;

    }

}

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

    if (low < high) {

        int mid = (low + high) / 2;

        MergeSort(arr, low, mid);

        MergeSort(arr, mid + 1, high);

        Merge(arr, low, mid, high);

    }

}

int main() {

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

    int arr\_size = sizeof(arr) / sizeof(arr[0]);

    cout << "Original array: ";

    for (int i = 0; i < arr\_size; i++) {

        cout << arr[i] << " ";

    }

    MergeSort(arr, 0, arr\_size - 1);

    cout << "\nSorted array: ";

    for (int i = 0; i < arr\_size; i++) {

        cout << arr[i] << " ";

    }

    return 0;

}

**2.**

/\*QUICK SORT\*/

#include <iostream>

using namespace std;

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

    int pivot = arr[low];

    int i = low + 1;

    int j = high;

    while (true) {

        while (i <= j && arr[i] <= pivot) {

            i++;

        }

        while (j >= i && arr[j] > pivot) {

            j--;

        }

        if (i <= j) {

            swap(arr[i], arr[j]);

        } else {

            swap(arr[low], arr[j]);

            return j;

        }

    }

}

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

    if (low < high) {

    int partitionIndex = Partition(arr, low, high);

        QuickSort(arr, low, partitionIndex - 1);

        QuickSort(arr, partitionIndex + 1, high);

    }

}

int main() {

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

    int arr\_size = sizeof(arr) / sizeof(arr[0]);

    cout << "Original array: ";

    for (int i = 0; i < arr\_size; i++) {

        cout << arr[i] << " ";

    }

    QuickSort(arr, 0, arr\_size - 1);

    cout << "\nSorted array: ";

    for (int i = 0; i < arr\_size; i++) {

        cout << arr[i] << " ";

    }

    return 0;

}

**3.knapsack**

#include<iostream>

#include <bits/stdc++.h>

using namespace std;

int max(int a, int b) { return (a > b) ? a : b; }

int knapSack(int W, int wt[], int val[], int n)

{

    if (n == 0 || W == 0)

        return 0;

    if (wt[n - 1] > W)

        return knapSack(W, wt, val, n - 1);

    else

        return max(val[n - 1] + knapSack(W - wt[n - 1], wt, val, n - 1),  knapSack(W, wt, val, n - 1));

}

int main()

{

    int x;

    cout << "Enter number of items: ";

    cin >> x;

    int profit[x];

    int weight[x];

    cout << "Enter profit of items: ";

    for(int i = 0; i < x; i++) {

        cin >> profit[i];

    }

    cout << "Enter weights of items: ";

    for(int i = 0; i < x; i++) {

        cin >> weight[i];

    }

    int W;

    cout << "Enter capacity of knapsack: ";

    cin >> W;

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

    cout << knapSack(W, weight, profit, n) << endl;

    return 0;

}

**4.assignment**

#include <iostream>

#include <climits>

using namespace std;

int minCost(int cost[][4], int n) {

  if (n == 1) {

    return cost[0][0];

  }

  int min\_cost = INT\_MAX;

  for (int i = 0; i < n; i++) {

  int rec\_cost = cost[i][0] + minCost(cost, n - 1);

    if (rec\_cost < min\_cost) {

      min\_cost = rec\_cost;

    }

  }

  return min\_cost;

}

int main() {

  int cost[4][4] = //write a matrix;

  int n = 4;

  cout << "Minimum cost using Brute Force: " << minCost(cost, n) << endl;

  return 0;

}

**5.//** **single source shortest paths**

#include <iostream>

using namespace std;

#include <limits.h>

#define V 9

int minDistance(int dist[], bool sptSet[])

{

    int min = INT\_MAX, min\_index;

    for (int v = 0; v < V; v++)

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

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

    return min\_index;

}

void printSolution(int dist[])

{

    cout << "Vertex \t Distance from Source" << endl;

    for (int i = 0; i < V; i++)

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

}

void dijkstra(int graph[V][V], int src)

{

    int dist[V];

   bool sptSet[V];

    for (int i = 0; i < V; i++)

        dist[i] = INT\_MAX, sptSet[i] = false;

    dist[src] = 0;

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

        int u = minDistance(dist, sptSet);

        sptSet[u] = true;

        for (int v = 0; v < V; 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);

}

int main()

{

    int graph[V][V] = { { } };

    dijkstra(graph, 0);

    return 0;

}

**6.//prim’s algorthim**

#include <iostream>

#include <climits>

const int MAX\_VERTICES = 100; // Maximum number of vertices

int minKey(int key[], bool mstSet[], int V) {

    int min = INT\_MAX, min\_index;

    for (int v = 0; v < V; v++)

        if (!mstSet[v] && key[v] < min)

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

    return min\_index;

}

void printMST(int parent[], int graph[MAX\_VERTICES][MAX\_VERTICES], int V) {

    std::cout << "Edge \tWeight\n";

    for (int i = 1; i < V; i++)

        std::cout << parent[i] << " - " << i << "\t" << graph[i][parent[i]] << "\n";

}

void primMST(int graph[MAX\_VERTICES][MAX\_VERTICES], int V) {

    int parent[MAX\_VERTICES]; // Array to store the MST

    int key[MAX\_VERTICES];    // Key values used to pick minimum weight edge

    bool mstSet[MAX\_VERTICES]; // To represent set of vertices included in MST

    for (int i = 0; i < V; i++)

        key[i] = INT\_MAX, mstSet[i] = false;

    key[0] = 0;

    parent[0] = -1; // First node is always the root

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

        int u = minKey(key, mstSet, V);

        mstSet[u] = true;

        for (int v = 0; v < V; v++)

            if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v])

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

    }

    printMST(parent, graph, V);

}

int main() {

    int V; // Number of vertices

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

    std::cin >> V;

    int graph[MAX\_VERTICES][MAX\_VERTICES];

    std::cout << "Enter the adjacency matrix:\n";

    for (int i = 0; i < V; i++)

        for (int j = 0; j < V; j++)

            std::cin >> graph[i][j];

    primMST(graph, V);

    return 0;

}

**6. Kruskal algorthim**

#include <iostream>

#include <algorithm>

const int MAX\_VERTICES = 100; // Maximum number of vertices

const int MAX\_EDGES = 1000; // Maximum number of edges

struct Edge {

    int src, dest, weight;

};

bool compareEdges(const Edge& e1, const Edge& e2) {

    return e1.weight < e2.weight;

}

int findParent(int parent[], int node) {

    if (parent[node] == -1)

        return node;

    return parent[node] = findParent(parent, parent[node]);

}

void unionSets(int parent[], int rank[], int x, int y) {

    int xRoot = findParent(parent, x);

    int yRoot = findParent(parent, y);

  if (rank[xRoot] < rank[yRoot])

        parent[xRoot] = yRoot;

    else if (rank[xRoot] > rank[yRoot])

        parent[yRoot] = xRoot;

    else {

        parent[yRoot] = xRoot;

        rank[xRoot]++;

    }

}

void kruskalMST(Edge edges[], int numEdges, int numVertices) {

    Edge mst[MAX\_VERTICES - 1]; // MST edges

    int mstSize = 0; // Number of edges in the MST

    int totalWeight = 0; // Total weight of the MST

    std::sort(edges, edges + numEdges, compareEdges);

    int parent[MAX\_VERTICES];

    int rank[MAX\_VERTICES];

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

        parent[i] = -1;

        rank[i] = 0;

    }

    for (int i = 0; i < numEdges && mstSize < numVertices - 1; i++) {

        int srcRoot = findParent(parent, edges[i].src);

        int destRoot = findParent(parent, edges[i].dest);

        if (srcRoot != destRoot) {

            mst[mstSize++] = edges[i];

            totalWeight += edges[i].weight;

            unionSets(parent, rank, srcRoot, destRoot);

        }

    }

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

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

        std::cout << mst[i].src << " -- " << mst[i].dest << " (weight: " << mst[i].weight << ")" << std::endl;

    }

    std::cout << "Total weight of the Minimum Spanning Tree: " << totalWeight << std::endl;

}

int main() {

    int numVertices, numEdges;

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

    std::cin >> numVertices;

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

    std::cin >> numEdges;

    Edge edges[MAX\_EDGES];

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

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

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

    }

    kruskalMST(edges, numEdges, numVertices);

    return 0;

}

**7. // job sequencing with deadlines**

#include <algorithm>

#include <iostream>

using namespace std;

struct Job

{

    char id;    // Job Id

    int dead;   // Deadline of job

    int profit; // Profit if job is over before or on

                // deadline

};

bool comparison(Job a, Job b)

{

    return (a.profit > b.profit);

}

void printJobScheduling(Job arr[], int n)

{

    sort(arr, arr + n, comparison);

    int result[n]; // To store result (Sequence of jobs)

    bool slot[n];  // To keep track of free time slots

    for (int i = 0; i < n; i++)

        slot[i] = false;

    for (int i = 0; i < n; i++)

    {

        for (int j = min(n, arr[i].dead) - 1; j >= 0; j--)

        {

            // Free slot found

            if (slot[j] == false)

            {

                result[j] = i;  // Add this job to result

                slot[j] = true; // Make this slot occupied

                break;

            }

        }

    }

    for (int i = 0; i < n; i++)

        if (slot[i])

            cout << arr[result[i]].id << " ";

}

int main()

{

    Job arr[] = {{'a', 2, 100},

                 {'b', 1, 19},

                 {'c', 2, 27},

                 {'d', 1, 25},

                 {'e', 3, 15}};

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

    cout << "Following is maximum profit sequence of jobs "

            "\n";

    printJobScheduling(arr, n);

    return 0;

}

**8.// Floyd warshall**

#include <iostream>

using namespace std;

#define nV 4

#define INF 999

void printMatrix(int matrix[][nV]);

void floydWarshall(int graph[][nV]) {

  int matrix[nV][nV], i, j, k;

  for (i = 0; i < nV; i++)

    for (j = 0; j < nV; j++)

      matrix[i][j] = graph[i][j];

  for (k = 0; k < nV; k++) {

    for (i = 0; i < nV; i++) {

      for (j = 0; j < nV; j++) {

        if (matrix[i][k] + matrix[k][j] < matrix[i][j])

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

      }

    }

  }

  printMatrix(matrix);

}

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

  for (int i = 0; i < nV; i++) {

    for (int j = 0; j < nV; j++) {

      if (matrix[i][j] == INF)

        printf("%4s", "INF");

      else

        printf("%4d", matrix[i][j]);

    }

    printf("\n");

  }

}

int main() {

  int graph[nV][nV] = {{0, 3, INF, 5},

             {2, 0, INF, 4},

             {INF, 1, 0, INF},

             {INF, INF, 2, 0}};

  floydWarshall(graph);

}

**9. Travelling sales man problem**

#include <iostream>

#include <limits>

#define N 4

int calculateTotalDistance(int route[], int distanceMatrix[N][N]) {

    int totalDistance = 0;

    for (int i = 0; i < N - 1; i++) {

        totalDistance += distanceMatrix[route[i]][route[i + 1]];

    }

    totalDistance += distanceMatrix[route[N - 1]][route[0]];

    return totalDistance;

}

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

    int temp = a;

    a = b;

    b = temp;

}

void permute(int route[], int distanceMatrix[N][N], int start, int end, int &minDistance, int optimalRoute[]) {

    if (start == end) {

        int currentDistance = calculateTotalDistance(route, distanceMatrix);

        if (currentDistance < minDistance) {

            minDistance = currentDistance;

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

                optimalRoute[i] = route[i];

            }

        }

    } else {

        for (int i = start; i <= end; i++) {

            swap(route[start], route[i]);

            permute(route, distanceMatrix, start + 1, end, minDistance, optimalRoute);

            swap(route[start], route[i]);

        }

    }

}

int main() {

    int distanceMatrix[N][N] = {

//enter input    };

    int route[N];

    for (int i = 0; i < N; i++) {

        route[i] = i;

    }

    int minDistance = std::numeric\_limits<int>::max();

    int optimalRoute[N];

    permute(route, distanceMatrix, 0, N - 1, minDistance, optimalRoute);

    std::cout << "Optimal Route: ";

    for (int i = 0; i < N; i++) {

        std::cout << optimalRoute[i] << " ";

    }

    std::cout << "\nMinimum Distance: " << minDistance << std::endl;

    return 0;

}

**10.Nqueens**

#include <iostream>

#include <cmath>

const int MAX\_SIZE = 100; // Maximum board size

class NQueenSolver {

private:

    int n;

    int positions[MAX\_SIZE];

public:

    NQueenSolver(int n) : n(n) {

        for (int i = 0; i < n; i++) {

            positions[i] = -1;

        }

    }

    bool IsSafe(int k, int i) {

        for (int j = 0; j < k; j++) {

            if (positions[j] == i || // Check same column

                std::abs(positions[j] - i) == std::abs(j - k)) // Check diagonals

                return false;

        }

        return true;

    }

    void PlaceQueens(int k) {

        for (int i = 0; i < n; i++) {

            if (IsSafe(k, i)) {

                positions[k] = i; // Place queen

                if (k == n - 1) {

                    PrintSolution(); // All queens placed

                } else {

                    PlaceQueens(k + 1); // Solve for next position

                }

                positions[k] = -1; // Backtrack

            }

        }

    }

    void PrintSolution() {

        for (int i = 0; i < n; i++) {

            std::cout << "Queen at: Row " << i << ", Col " << positions[i] << std::endl;

        }

        std::cout << "---" << std::endl; // Just for a clearer output

    }

};

int main() {

    int n ;

    std :: cout<<"Enter the value of N\n";

    std::cin >>n; // You can set this to any number

    NQueenSolver solver(n);

    std::cout << "Placing " << n << " Queens on a " << n << "x" << n << " chess board." << std::endl;

    solver.PlaceQueens(0); // Start placing queens from the first row

    return 0;

}

**11. Hamilton graph**

#include<iostream>

using namespace std;

const int N = 5; // Change this variable based on the number of vertices in the graph

const int MAX\_VERTICES = 10; // Maximum number of vertices

void printSolution(const int path[], int pathSize) {

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

        cout << path[i] << " ";

    cout << path[0] << endl; // End by showing the first vertex to form a cycle

}

bool isSafe(int v, const bool graph[MAX\_VERTICES][MAX\_VERTICES], int path[], int pos) {

    if (graph[path[pos - 1]][v] == 0)

        return false;

    for (int i = 0; i < pos; i++)

        if (path[i] == v)

            return false;

    return true;

}

bool HamiltonianCycleUtil(bool graph[MAX\_VERTICES][MAX\_VERTICES], int path[], int pos) {

    if (pos == N) {

        if (graph[path[pos - 1]][path[0]] == 1) {

            return true;  // A cycle is formed, return true

        } else {

            return false;

        }

    }

for (int v = 1; v < N; v++) {

        if (isSafe(v, graph, path, pos)) {

            path[pos] = v;

            if (HamiltonianCycleUtil(graph, path, pos + 1) == true)

                return true;

            path[pos] = -1; // Remove vertex if it doesn't lead to a solution

        }

    }

    return false;

}

bool findHamiltonianCycle(bool graph[MAX\_VERTICES][MAX\_VERTICES]) {

    int path[MAX\_VERTICES];

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

        path[i] = -1;

  path[0] = 0; // Start from the first vertex

    if (HamiltonianCycleUtil(graph, path, 1) == false) {

        cout << "Solution does not exist";

        return false;

    }

  printSolution(path, N);

    return true;

}

int main() {

    // Create a graph in the form of an adjacency matrix

    bool graph[MAX\_VERTICES][MAX\_VERTICES] = {

        {0, 1, 0, 1, 0},

        {1, 0, 1, 1, 1},

        {0, 1, 0, 0, 1},

        {1, 1, 0, 0, 1},

        {0, 1, 1, 1, 0},

    };

    findHamiltonianCycle(graph);

    return 0;

}

**12. Graph colouring**

#include <iostream>

const int MAX\_VERTICES = 100; // Maximum number of vertices

template <typename T>

class Vector {

private:

    T arr[MAX\_VERTICES];

    int size;

public:

    Vector() : size(0) {}

    void push\_back(const T& value) {

        arr[size++] = value;

    }

  T& operator[](int index) {

        return arr[index];

    }

    int getSize() const {

        return size;

    }

};

class GraphColoring {

private:

    int n; // Number of vertices

    int m; // Number of colors

    bool G[MAX\_VERTICES][MAX\_VERTICES]; // Adjacency matrix

    int colors[MAX\_VERTICES]; // Colors assigned to vertices

public:

    GraphColoring(int n, int m, const bool graph[MAX\_VERTICES][MAX\_VERTICES])

        : n(n), m(m) {

        for (int i = 0; i < n; ++i) {

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

                G[i][j] = graph[i][j];

            }

        }

        for (int i = 0; i < n; ++i) {

            colors[i] = 0; // Initialize colors to 0

        }

    }

    // Try to assign a valid color to vertex k

    void NextValue(int k) {

        while (true) {

            colors[k] = (colors[k] + 1) % (m + 1); // Next highest color

            if (colors[k] == 0) return; // All colors attempted, backtrack

            int j;

            for (j = 0; j < n; ++j) {

                if (G[k][j] && colors[k] == colors[j]) {

                    // If k and j are adjacent and have the same color

                    break;

                }

            }

            if (j == n) return; // No conflicts, color is valid

        }

    }

    void mColoring(int k) {

        do {

            NextValue(k);

            if (colors[k] == 0) return; // Backtrack if no color possible

            if (k == n - 1) {

                PrintSolution(); // Print solution if all vertices are colored

            } else {

                mColoring(k + 1); // Color next vertex

            }

        } while (true);

    }

    void PrintSolution() {

        std::cout << "One of the solutions: ";

        for (int i = 0; i < n; ++i) {

            std::cout << "Vertex " << i + 1 << " -> Color " << colors[i] << ", ";

        }

        std::cout << '\n';

    }

};

int main() {

    int n = 4; // Number of vertices

    int m = 3; // Number of colors

    bool graph[MAX\_VERTICES][MAX\_VERTICES] = {

        {0, 1, 1, 0},

        {1, 0, 1, 1},

        {1, 1, 0, 1},

        {0, 1, 1, 0}

    };

    GraphColoring solver(n, m, graph);

    std::cout << "Graph Coloring Solutions:" << std::endl;

    solver.mColoring(0); // Start the process from the first vertex

    return 0;}