**Slip 1.**

**Q.1) Write a program to sort a list of n numbers in ascending order using selection sort and determine the time required to sort the elements**

Ans :

cpp

#include <iostream>

#include <vector>

#include <ctime>

#include <cstdlib>

using namespace std;

// Function to generate random numbers

vector<int> generateRandomArray(int n) {

    vector<int> arr(n);

    for (int i = 0; i < n; i++)

        arr[i] = rand() % 10000; // Random numbers up to 9999

    return arr;

}

// Selection Sort Function

void selectionSort(vector<int> &arr) {

    int n = arr.size();

    for (int i = 0; i < n - 1; i++) {

        int minIdx = i;

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

            if (arr[j] < arr[minIdx])

                minIdx = j;

        }

        swap(arr[i], arr[minIdx]);

    }

}

// Main function

int main() {

    int n;

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

    cin >> n;

    // Generate random array

    vector<int> data = generateRandomArray(n);

    cout << "\nOriginal Array:\n";

    for (int i = 0; i < n; i++) {

        cout << data[i] << " ";

    }

    cout << "\n";

    // Measure time

    clock\_t start = clock();

    selectionSort(data);

    clock\_t end = clock();

    double time\_taken = double(end - start) / CLOCKS\_PER\_SEC;

    // Output sorted array

    cout << "\nSorted Array (Ascending Order):\n";

    for (int i = 0; i < n; i++) {

        cout << data[i] << " ";

    }

    cout << "\n";

    // Output time taken

    cout << "\nTime taken to sort " << n << " elements using Selection Sort: " << time\_taken << " seconds\n";

    return 0;

}

OUTPUT:

g++ program.cpp -o my\_program

./my\_program # On Linux/macOS

g++ 1.cpp -o 1 && ./1

**Q.2) Write a program to sort a given set of elements using the Quick sort method and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted. The elements can be read from a file or can be generated using the random number generator.**

Ans:

#include <iostream>

#include <cstdlib>

#include <ctime>

#include <vector>

#include <fstream>

#include <chrono>

using namespace std;

using namespace std::chrono;

// Quick Sort Function

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

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

    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);

    }

}

// Generate Random Numbers

void generateRandomNumbers(vector<int>& arr, int n) {

    srand(time(0));

    for(int i = 0; i < n; ++i) {

        arr.push\_back(rand() % 100000); // values between 0 and 99999

    }

}

// Read Numbers from File

void readFromFile(vector<int>& arr, string filename) {

    ifstream file(filename);

    int value;

    while(file >> value) {

        arr.push\_back(value);

    }

    file.close();

}

int main() {

    vector<int> arr;

    int n, choice;

    cout << "Enter number of elements: ";

    cin >> n;

    cout << "Choose input method:\n1. Random Numbers\n2. Read from file\nChoice: ";

    cin >> choice;

    if(choice == 1) {

        generateRandomNumbers(arr, n);

    } else if(choice == 2) {

        string filename;

        cout << "Enter file name: ";

        cin >> filename;

        readFromFile(arr, filename);

        n = arr.size();

    } else {

        cout << "Invalid choice.\n";

        return 1;

    }

    // Time Measurement

    auto start = high\_resolution\_clock::now();

    quickSort(arr, 0, n - 1);

    auto end = high\_resolution\_clock::now();

    auto duration = duration\_cast<microseconds>(end - start);

    cout << "\nTime taken by Quick Sort for " << n << " elements: "

         << duration.count() << " microseconds\n";

    return 0;

}

OUTPUT:

g++ program.cpp -o my\_program

./my\_program # On Linux/macOS

g++ 1.cpp -o 1 && ./1

**Slip 2**

**Q.1) Write a program to sort n randomly generated elements using Heapsort method**

Ans:

#include <iostream>

#include <vector>

#include <cstdlib> // For rand() and srand()

#include <ctime> // For time()

#include <chrono> // For measuring time

using namespace std;

using namespace std::chrono;

// Function to heapify a subtree rooted at index i

void heapify(vector<int>& arr, int n, int i) {

int largest = i; // Initialize largest as root

int left = 2 \* i + 1; // left child

int right = 2 \* i + 2; // right child

// If left child is larger than root

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

largest = left;

// If right child is larger than largest so far

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

largest = right;

// If largest is not root

if (largest != i) {

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

// Recursively heapify the affected sub-tree

heapify(arr, n, largest);

}

}

// Heapsort function

void heapSort(vector<int>& arr, int n) {

// Build max heap

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

heapify(arr, n, i);

// One by one extract elements

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

// Move current root to end

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

// Call max heapify on the reduced heap

heapify(arr, i, 0);

}

}

int main() {

int n;

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

cin >> n;

vector<int> arr(n);

// Seed random number generator

srand(time(0));

// Generate random numbers

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

arr[i] = rand() % 10000; // Random number between 0 and 9999

}

cout << "\nUnsorted array:\n";

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

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

cout << "\n";

// Measure time taken by heapsort

auto start = high\_resolution\_clock::now();

heapSort(arr, n);

auto stop = high\_resolution\_clock::now();

auto duration = duration\_cast<milliseconds>(stop - start);

cout << "\nSorted array:\n";

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

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

cout << "\n";

cout << "\nTime taken by Heapsort: " << duration.count() << " milliseconds\n";

return 0;

}

OUTPUT:

g++ program.cpp -o my\_program

./my\_program # On Linux/macOS

g++ 1.cpp -o 1 && ./1

**Q.2) Write a program to implement Strassen’s Matrix multiplication**

Ans:

#include <iostream>

#include <vector>

using namespace std;

// Add two matrices

vector<vector<int>> add(vector<vector<int>> A, vector<vector<int>> B) {

    int n = A.size();

    vector<vector<int>> result(n, vector<int>(n));

    for(int i = 0; i < n; ++i)

        for(int j = 0; j < n; ++j)

            result[i][j] = A[i][j] + B[i][j];

    return result;

}

// Subtract two matrices

vector<vector<int>> subtract(vector<vector<int>> A, vector<vector<int>> B) {

    int n = A.size();

    vector<vector<int>> result(n, vector<int>(n));

    for(int i = 0; i < n; ++i)

        for(int j = 0; j < n; ++j)

            result[i][j] = A[i][j] - B[i][j];

    return result;

}

// Strassen's Matrix Multiplication

vector<vector<int>> strassen(vector<vector<int>> A, vector<vector<int>> B) {

    int n = A.size();

    if(n == 1) {

        return {{A[0][0] \* B[0][0]}};

    }

    int newSize = n / 2;

    vector<int> inner(newSize);

    vector<vector<int>>

        A11(newSize, inner), A12(newSize, inner), A21(newSize, inner), A22(newSize, inner),

        B11(newSize, inner), B12(newSize, inner), B21(newSize, inner), B22(newSize, inner);

    // Dividing matrices into 4 submatrices

    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];

        }

    }

    // Calculating M1 to M7:

    vector<vector<int>> M1 = strassen(add(A11, A22), add(B11, B22));

    vector<vector<int>> M2 = strassen(add(A21, A22), B11);

    vector<vector<int>> M3 = strassen(A11, subtract(B12, B22));

    vector<vector<int>> M4 = strassen(A22, subtract(B21, B11));

    vector<vector<int>> M5 = strassen(add(A11, A12), B22);

    vector<vector<int>> M6 = strassen(subtract(A21, A11), add(B11, B12));

    vector<vector<int>> M7 = strassen(subtract(A12, A22), add(B21, B22));

    // C11 to C22:

    vector<vector<int>> C11 = add(subtract(add(M1, M4), M5), M7);

    vector<vector<int>> C12 = add(M3, M5);

    vector<vector<int>> C21 = add(M2, M4);

    vector<vector<int>> C22 = add(subtract(add(M1, M3), M2), M6);

    // Combining results into a single matrix

    vector<vector<int>> C(n, vector<int>(n));

    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];

        }

    }

    return C;

}

// Helper function to print matrix

void printMatrix(vector<vector<int>> matrix) {

    for(auto row : matrix) {

        for(auto val : row)

            cout << val << " ";

        cout << endl;

    }

}

// Main Function

int main() {

    int n;

    cout << "Enter the size of matrix (must be power of 2): ";

    cin >> n;

    vector<vector<int>> A(n, vector<int>(n));

    vector<vector<int>> B(n, vector<int>(n));

    cout << "Enter elements of Matrix A:\n";

    for(int i = 0; i < n; ++i)

        for(int j = 0; j < n; ++j)

            cin >> A[i][j];

    cout << "Enter elements of Matrix B:\n";

    for(int i = 0; i < n; ++i)

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

cin >> B[i][j];

vector<vector<int>> C = strassen(A, B);

cout << "\nResultant Matrix (A x B):\n";

printMatrix(C);

return 0;

}

OUTPUT:

g++ program.cpp -o my\_program

./my\_program # On Linux/macOS

g++ 1.cpp -o 1 && ./1

**Slip 3)**

**Q.1) Write a program to sort a given set of elements using the Quick sort method and determine the time required to sort the elements.**

Ans:

#include <iostream>

#include <vector>

#include <cstdlib>      // for rand()

#include <ctime>        // for clock()

#include <algorithm>    // for random\_shuffle (optional)

using namespace std;

// Quick Sort Partition Function

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

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

    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(arr[i + 1], arr[high]);

    return i + 1;

}

// Quick Sort Recursive Function

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

    if (low < high) {

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

        quickSort(arr, low, p - 1);

        quickSort(arr, p + 1, high);

    }

}

int main() {

    int n;

    cout << "Enter number of elements to sort: ";

    cin >> n;

    vector<int> arr(n);

    // Generate random elements

    srand(time(0)); // Seed for randomness

    for (int i = 0; i < n; i++) {

        arr[i] = rand() % 1000; // Random numbers between 0 and 999

    }

    cout << "\nUnsorted array:\n";

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

    cout << "\n";

    // Start timing

    clock\_t start = clock();

    quickSort(arr, 0, n - 1);

    // End timing

    clock\_t end = clock();

    double time\_taken = double(end - start) / CLOCKS\_PER\_SEC;

    cout << "\nSorted array:\n";

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

    cout << "\n";

    cout << "\nTime taken by Quick Sort: " << time\_taken << " seconds\n";

    return 0;

}

OUTPUT:

g++ program.cpp -o my\_program

./my\_program # On Linux/macOS

g++ 1.cpp -o 1 && ./1

**Q.2) Write a program to find Minimum Cost Spanning Tree of a given undirected graph using Prims algorithm**

Ans:

#include <iostream>

#include <climits>

#include <vector>

using namespace std;

#define V 5 // You can change the number of vertices here

// Find the vertex with minimum key value

int minKey(vector<int>& key, vector<bool>& mstSet) {

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;

}

// Print MST

void printMST(vector<int>& parent, vector<vector<int>>& graph) {

    cout << "Edge \tWeight\n";

    int total = 0;

    for (int i = 1; i < V; i++) {

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

        total += graph[i][parent[i]];

    }

    cout << "Total cost of MST: " << total << endl;

}

// Prim's Algorithm

void primMST(vector<vector<int>>& graph) {

    vector<int> parent(V);     // To store constructed MST

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

    vector<bool> mstSet(V, false); // To represent vertices included in MST

    key[0] = 0;      // Start from first vertex

    parent[0] = -1;  // First node is root of MST

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

        int u = minKey(key, mstSet);

        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);

}

int main() {

    vector<vector<int>> graph = {

        {0, 2, 0, 6, 0},

        {2, 0, 3, 8, 5},

        {0, 3, 0, 0, 7},

        {6, 8, 0, 0, 9},

        {0, 5, 7, 9, 0}

    };

    primMST(graph);

    return 0;

}

OUTPUT:

g++ program.cpp -o my\_program

./my\_program # On Linux/macOS

g++ 1.cpp -o 1 && ./1

**Slip 4**

**Q.1) Write a program to implement a Merge Sort algorithm to sort a given set of elements and determine the time required to sort the elements**

#include <iostream>

#include <vector>

#include <cstdlib>   // For rand()

#include <ctime>     // For clock()

using namespace std;

// Merge function

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

    int n1 = mid - left + 1;

    int n2 = right - mid;

    // Temporary arrays

    vector<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[mid + 1 + j];

    // Merge the temp arrays back into arr

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {

        if (L[i] <= R[j]) {

            arr[k++] = L[i++];

        }

        else {

            arr[k++] = R[j++];

        }

    }

    // Copy remaining elements of L[]

    while (i < n1) {

        arr[k++] = L[i++];

    }

    // Copy remaining elements of R[]

    while (j < n2) {

        arr[k++] = R[j++];

    }

}

// Merge Sort function

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 main() {

    int n;

    cout << "Enter number of elements to sort: ";

    cin >> n;

    vector<int> arr(n);

    // Generate random elements

    srand(time(0));

    for (int i = 0; i < n; i++) {

        arr[i] = rand() % 1000; // Values from 0 to 999

    }

    cout << "\nUnsorted array:\n";

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

    cout << "\n";

    // Start time

    clock\_t start = clock();

    mergeSort(arr, 0, n - 1);

    // End time

    clock\_t end = clock();

    double time\_taken = double(end - start) / CLOCKS\_PER\_SEC;

    cout << "\nSorted array:\n";

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

    cout << "\n";

    cout << "\nTime taken by Merge Sort: " << time\_taken << " seconds\n";

    return 0;

}

OUTPUT:

g++ program.cpp -o my\_program

./my\_program # On Linux/macOS

g++ 1.cpp -o 1 && ./1

**Q.2) Write a program to implement Knapsack problems using Greedy method.**

Ans:

#include <iostream>

#include <vector>

#include <algorithm> // For sort function

using namespace std;

// Item structure to hold value, weight, and value-to-weight ratio

struct Item {

int value;

int weight;

double ratio;

};

// Function to compare items based on value/weight ratio

bool compare(Item a, Item b) {

return a.ratio > b.ratio;

}

// Function to solve the Knapsack problem using Greedy approach

double knapsackGreedy(vector<Item>& items, int W) {

sort(items.begin(), items.end(), compare); // Sort items based on value-to-weight ratio

int totalValue = 0; // Total value of items in the knapsack

int totalWeight = 0; // Total weight of items in the knapsack

for (auto& item : items) {

if (totalWeight + item.weight <= W) { // If the item can fit in the knapsack

totalValue += item.value;

totalWeight += item.weight;

} else {

// If the item cannot fit fully, take the fractional part

int remainingWeight = W - totalWeight;

totalValue += item.value \* (double)remainingWeight / item.weight;

break;

}

}

return totalValue;

}

int main() {

int n, W;

cout << "Enter number of items: ";

cin >> n;

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

cin >> W;

vector<Item> items(n);

cout << "Enter value and weight for each item (value weight): \n";

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

cin >> items[i].value >> items[i].weight;

items[i].ratio = (double)items[i].value / items[i].weight; // Calculate value-to-weight ratio

}

// Calculate maximum value that can be carried in the knapsack

double maxValue = knapsackGreedy(items, W);

cout << "Maximum value in Knapsack: " << maxValue << endl;

return 0;

}

**Slip 5**

**Q.1) Write a program for the Implementation of Kruskal’s algorithm to find minimum cost spanning tree.**

Ans:

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

// Structure to represent a graph edge

struct Edge {

    int u, v, weight;

};

// Structure to represent a subset for union-find

struct Subset {

    int parent, rank;

};

// Function to find the subset of an element using path compression

int find(Subset subsets[], int i) {

    if (subsets[i].parent != i)

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

    return subsets[i].parent;

}

// Function to do union of two subsets

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++;

    }

}

// Function to implement Kruskal's algorithm to find MST

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

    // Step 1: Sort all edges in increasing order of their weights

    sort(edges.begin(), edges.end(), [](Edge a, Edge b) {

        return a.weight < b.weight;

    });

    // Create V subsets for union-find

    Subset \*subsets = new Subset[V];

    for (int v = 0; v < V; ++v) {

        subsets[v].parent = v;

        subsets[v].rank = 0;

    }

    // Result to store the MST

    vector<Edge> result;

    // Step 2: Iterate through edges and apply union-find

    for (Edge e : edges) {

        int x = find(subsets, e.u);

        int y = find(subsets, e.v);

        // If including this edge does not cause a cycle

        if (x != y) {

            result.push\_back(e);

            Union(subsets, x, y);

        }

    }

    // Step 3: Output the MST

    cout << "Edges in the Minimum Spanning Tree: \n";

    int minCost = 0;

    for (Edge e : result) {

        cout << e.u << " -- " << e.v << " == " << e.weight << endl;

        minCost += e.weight;

    }

    cout << "Minimum Cost of the Spanning Tree: " << minCost << endl;

    delete[] subsets;

}

int main() {

    int V, E;

    cout << "Enter number of vertices: ";

    cin >> V;

    cout << "Enter number of edges: ";

    cin >> E;

    vector<Edge> edges(E);

    cout << "Enter edges (u v weight): \n";

    for (int i = 0; i < E; i++) {

        cin >> edges[i].u >> edges[i].v >> edges[i].weight;

    }

    Kruskal(V, edges);

    return 0;

}

Enter number of vertices: 4

Enter number of edges: 5

Enter edges (u v weight):

0 1 10

0 2 6

0 3 5

1 3 15

2 3 4

Edges in the Minimum Spanning Tree:

2 -- 3 == 4

0 -- 3 == 5

0 -- 1 == 10

Minimum Cost of the Spanning Tree: 19

**Q.2) Write a program to implement Huffman Code using greedy methods and also calculate the best case and worst-case complexity.**

Ans:

#include <iostream>

#include <queue>

#include <vector>

#include <unordered\_map>

using namespace std;

// Structure to represent a node in the Huffman Tree

struct Node {

char data;

int freq;

Node \*left, \*right;

Node(char data, int freq) {

this->data = data;

this->freq = freq;

left = right = nullptr;

}

};

// Comparison function to be used by priority queue (min heap)

struct compare {

bool operator()(Node\* l, Node\* r) {

return l->freq > r->freq;

}

};

// Function to build the Huffman Tree

Node\* buildHuffmanTree(const unordered\_map<char, int>& freq) {

priority\_queue<Node\*, vector<Node\*>, compare> minHeap;

// Create leaf nodes and add them to the min heap

for (auto pair : freq) {

minHeap.push(new Node(pair.first, pair.second));

}

// Build the tree

while (minHeap.size() != 1) {

Node \*left = minHeap.top();

minHeap.pop();

Node \*right = minHeap.top();

minHeap.pop();

// Create a new internal node with the sum of frequencies

Node \*top = new Node('$', left->freq + right->freq);

top->left = left;

top->right = right;

// Add the new node to the min heap

minHeap.push(top);

}

return minHeap.top();

}

// Function to generate Huffman codes from the Huffman tree

void generateHuffmanCodes(Node\* root, string str, unordered\_map<char, string>& huffmanCodes) {

if (!root) return;

// If it's a leaf node, store the code

if (!root->left && !root->right) {

huffmanCodes[root->data] = str;

}

// Recur for left and right subtrees

generateHuffmanCodes(root->left, str + "0", huffmanCodes);

generateHuffmanCodes(root->right, str + "1", huffmanCodes);

}

// Function to display the Huffman codes

void displayHuffmanCodes(const unordered\_map<char, string>& huffmanCodes) {

cout << "Character Huffman Codes:\n";

for (auto pair : huffmanCodes) {

cout << pair.first << ": " << pair.second << endl;

}

}

int main() {

string text;

cout << "Enter the text: ";

cin >> text;

unordered\_map<char, int> freq;

// Calculate frequency of each character

for (char c : text) {

freq[c]++;

}

// Build the Huffman Tree

Node\* root = buildHuffmanTree(freq);

// Generate Huffman codes

unordered\_map<char, string> huffmanCodes;

generateHuffmanCodes(root, "", huffmanCodes);

// Display Huffman Codes

displayHuffmanCodes(huffmanCodes);

return 0;

}

**Slip 6**

**Q-1) Write a program for the Implementation of Prim’s algorithm to find minimum cost spanning tree.**

Ans:

#include <iostream>

#include <vector>

#include <queue>

#include <climits>

using namespace std;

// Edge structure to represent an edge between two vertices with a weight

struct Edge {

    int u, v, weight;

    Edge(int u, int v, int weight) : u(u), v(v), weight(weight) {}

};

// Comparator for priority queue to prioritize edges with the smallest weight

struct Compare {

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

        return e1.weight > e2.weight; // min-heap

    }

};

// Function to implement Prim's Algorithm

void prim(int vertices, vector<vector<int>>& graph) {

    vector<bool> inMST(vertices, false); // To track whether a vertex is in the MST

    priority\_queue<Edge, vector<Edge>, Compare> pq; // Min-heap to store edges

    // Start with vertex 0

    inMST[0] = true;

    int mstCost = 0;

    vector<Edge> mstEdges;

    // Add all edges from vertex 0 to the priority queue

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

        if (graph[0][v] != 0) {

            pq.push(Edge(0, v, graph[0][v]));

        }

    }

    // Iterate to find the MST

    while (!pq.empty()) {

        Edge edge = pq.top();

        pq.pop();

        int u = edge.u;

        int v = edge.v;

        int weight = edge.weight;

        if (inMST[v]) continue; // Skip if v is already in MST

        // Add edge to MST

        inMST[v] = true;

        mstCost += weight;

        mstEdges.push\_back(edge);

        // Add all edges from vertex v to the priority queue

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

            if (!inMST[i] && graph[v][i] != 0) {

                pq.push(Edge(v, i, graph[v][i]));

            }

        }

    }

    // Print the edges of the MST

    cout << "Edges in the Minimum Spanning Tree (MST):" << endl;

    for (auto& edge : mstEdges) {

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

    }

    cout << "Minimum Cost of the Spanning Tree: " << mstCost << endl;

}

int main() {

    int vertices, edges;

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

    cin >> vertices;

    cout << "Enter the adjacency matrix for the graph (0 represents no edge):" << endl;

    vector<vector<int>> graph(vertices, vector<int>(vertices, 0));

    // Input the adjacency matrix

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

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

            cin >> graph[i][j];

        }

    }

    // Call the Prim’s Algorithm function

    prim(vertices, graph);

    return 0;

}

**Q.2) Write a Program to find only length of Longest Common Subsequence.**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

// Function to find the length of Longest Common Subsequence

int lcsLength(const string& X, const string& Y) {

int m = X.length();

int n = Y.length();

// Create a 2D DP array to store lengths of longest common subsequence

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

// Fill the DP table

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

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

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

dp[i][j] = dp[i - 1][j - 1] + 1; // Characters match

} else {

dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]); // No match

}

}

}

// The length of the LCS will be in dp[m][n]

return dp[m][n];

}

int main() {

string X, Y;

// Input two strings

cout << "Enter the first string: ";

cin >> X;

cout << "Enter the second string: ";

cin >> Y;

// Call function to find the length of LCS

int result = lcsLength(X, Y);

// Output the length of LCS

cout << "Length of Longest Common Subsequence: " << result << endl;

return 0;

}

**Slip 7**

**Q-1) Write a program for the Implementation of Dijkstra’s algorithm to find shortest path to other vertices**

Ans;

#include <iostream>

#include <vector>

#include <climits>

#include <queue>

using namespace std;

#define V 9 // Number of vertices in the graph

// A utility function to find the vertex with the minimum distance value

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

int min = INT\_MAX, min\_index;

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

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

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

}

}

return min\_index;

}

// Function to implement Dijkstra's algorithm for finding the shortest path

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

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

vector<bool> sptSet(V, false); // Shortest Path Tree set

dist[src] = 0;

// Priority queue to select the minimum distance vertex

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

int u = minDistance(dist, sptSet); // Get vertex with minimum distance

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];

}

}

}

// Print the shortest distance from the source

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

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

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

}

}

int main() {

// Adjacency matrix representation of the graph

int graph[V][V] = {

{0, 4, 0, 0, 0, 0, 0, 8, 0},

{4, 0, 8, 0, 0, 0, 0, 11, 0},

{0, 8, 0, 7, 0, 4, 0, 0, 2},

{0, 0, 7, 0, 9, 14, 0, 0, 0},

{0, 0, 0, 9, 0, 10, 0, 0, 0},

{0, 0, 4, 14, 10, 0, 2, 0, 0},

{0, 0, 0, 0, 0, 2, 0, 1, 6},

{8, 11, 0, 0, 0, 0, 1, 0, 7},

{0, 0, 2, 0, 0, 0, 6, 7, 0}

};

int source = 0; // Define source vertex (0 in this case)

// Run Dijkstra's algorithm

dijkstra(graph, source);

return 0;

}

**Q.2) Write a program for finding Topological sorting for Directed Acyclic Graph (DAG)**

Ans:

#include <iostream>

#include <vector>

#include <queue>

#include <algorithm>

using namespace std;

// Function to perform Topological Sort

void topologicalSort(int V, vector<int> adj[]) {

vector<int> in\_degree(V, 0);

// Calculate in-degree (number of incoming edges for each vertex)

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

for (int v : adj[u]) {

in\_degree[v]++;

}

}

// Queue to store vertices with no incoming edges (in-degree = 0)

queue<int> q;

// Add all vertices with in-degree 0 to the queue

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

if (in\_degree[i] == 0) {

q.push(i);

}

}

vector<int> topoOrder;

// Process vertices one by one

while (!q.empty()) {

int u = q.front();

q.pop();

topoOrder.push\_back(u);

// Decrease the in-degree of adjacent vertices

for (int v : adj[u]) {

in\_degree[v]--;

// If in-degree becomes 0, add it to the queue

if (in\_degree[v] == 0) {

q.push(v);

}

}

}

// If all vertices are processed, print the topological order

if (topoOrder.size() == V) {

cout << "Topological Sort: ";

for (int i : topoOrder) {

cout << i << " ";

}

cout << endl;

} else {

cout << "The graph contains a cycle, topological sort is not possible." << endl;

}

}

int main() {

// Number of vertices

int V = 6;

// Adjacency list for the graph

vector<int> adj[V];

// Add edges (directed edges)

adj[5].push\_back(2);

adj[5].push\_back(0);

adj[4].push\_back(0);

adj[4].push\_back(1);

adj[2].push\_back(3);

adj[3].push\_back(1);

// Perform topological sort

topologicalSort(V, adj);

return 0;

}

**Slip 8:**

**Q.1) Write a program to implement Fractional Knapsack problems using Greedy Method.**

Ans:

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

// Structure to represent an item

struct Item {

    int value;

    int weight;

    // Constructor

    Item(int v, int w) : value(v), weight(w) {}

    // Function to calculate value per unit weight

    double valuePerWeight() const {

        return (double)value / weight;

    }

};

// Comparator to sort items based on value/weight ratio in descending order

bool compare(Item a, Item b) {

    return a.valuePerWeight() > b.valuePerWeight();

}

double fractionalKnapsack(int capacity, vector<Item>& items) {

    // Sort items by value per unit weight

    sort(items.begin(), items.end(), compare);

    double totalValue = 0.0;

    for (Item& item : items) {

        if (capacity == 0) break;  // No more capacity in the knapsack

        // Take the whole item if it fits

        if (item.weight <= capacity) {

            totalValue += item.value;

            capacity -= item.weight;

        }

        // Take as much as possible from the remaining item

        else {

            totalValue += item.value \* ((double)capacity / item.weight);

            break;

        }

    }

    return totalValue;

}

int main() {

    int n, capacity;

    // Input number of items and knapsack capacity

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

    cin >> n;

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

    cin >> capacity;

    vector<Item> items;

    // Input the items' values and weights

    cout << "Enter the value and weight for each item:" << endl;

    for (int i = 0; i < n; ++i) {

        int value, weight;

        cout << "Item " << i + 1 << " - Value: ";

        cin >> value;

        cout << "Item " << i + 1 << " - Weight: ";

        cin >> weight;

        items.push\_back(Item(value, weight));

    }

    // Get the maximum value that can be carried in the knapsack

    double maxValue = fractionalKnapsack(capacity, items);

    cout << "Maximum value in the knapsack = " << maxValue << endl;

    return 0;

}

**Q.2) Write Program to implement Traveling Salesman Problem using nearest neighbor algorithm**

Ans:

#include <iostream>

#include <vector>

#include <climits>

#include <cmath>

using namespace std;

// Function to calculate the distance between two cities

double distance(int x1, int y1, int x2, int y2) {

    return sqrt(pow(x2 - x1, 2) + pow(y2 - y1, 2));

}

// Function to implement Nearest Neighbor Algorithm for TSP

double nearestNeighborTSP(vector<pair<int, int>>& cities) {

    int n = cities.size();

    vector<bool> visited(n, false);

    // Start from the first city

    visited[0] = true;

    int currentCity = 0;

    double totalDistance = 0.0;

    for (int i = 1; i < n; i++) {

        double minDist = INT\_MAX;

        int nearestCity = -1;

        // Find the nearest unvisited city

        for (int j = 0; j < n; j++) {

            if (!visited[j]) {

                double dist = distance(cities[currentCity].first, cities[currentCity].second, cities[j].first, cities[j].second);

                if (dist < minDist) {

                    minDist = dist;

                    nearestCity = j;

                }

            }

        }

        // Add the distance to the total and mark the city as visited

        totalDistance += minDist;

        visited[nearestCity] = true;

        currentCity = nearestCity;

    }

    // Return to the starting city

    totalDistance += distance(cities[currentCity].first, cities[currentCity].second, cities[0].first, cities[0].second);

    return totalDistance;

}

int main() {

    int n;

    // Input the number of cities

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

    cin >> n;

    vector<pair<int, int>> cities(n);

    // Input the coordinates (x, y) of each city

    cout << "Enter the coordinates of each city:" << endl;

    for (int i = 0; i < n; ++i) {

        cout << "City " << i + 1 << " - x: ";

        cin >> cities[i].first;

        cout << "City " << i + 1 << " - y: ";

        cin >> cities[i].second;

    }

    // Get the shortest path using Nearest Neighbor Algorithm

    double shortestDistance = nearestNeighborTSP(cities);

    cout << "Total distance of the shortest path: " << shortestDistance << endl;

    return 0;

}

**Slip 9**

**Q.1) Write a program to implement optimal binary search tree and also calculate the best-case complexity.**

Ans:

#include <iostream>

#include <vector>

#include <climits>

#include <numeric>

using namespace std;

// Function to calculate the optimal cost of the Binary Search Tree

int optimalBST(const vector<int>& keys, const vector<int>& freq, int n) {

    // Create a table to store results of subproblems

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

    // Fill the diagonal of the cost matrix (one node)

    for (int i = 0; i < n; i++) {

        cost[i][i] = freq[i];

    }

    // Calculate the cost for chains of length 2 to n

    for (int chainLen = 2; chainLen <= n; chainLen++) {

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

            int j = i + chainLen - 1;

            cost[i][j] = INT\_MAX;

            // Try making every node in the current chain the root

            for (int r = i; r <= j; r++) {

                // Calculate cost of left and right subtrees

                int c = (r > i ? cost[i][r - 1] : 0) + (r < j ? cost[r + 1][j] : 0);

                // Add the total frequency in the current chain to the cost

                c += accumulate(freq.begin() + i, freq.begin() + j + 1, 0);

                // Update the minimum cost

                cost[i][j] = min(cost[i][j], c);

            }

        }

    }

    return cost[0][n - 1];

}

int main() {

    int n;

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

    cin >> n;

    vector<int> keys(n);

    vector<int> freq(n);

    cout << "Enter the keys:" << endl;

    for (int i = 0; i < n; i++) {

        cin >> keys[i];

    }

    cout << "Enter the frequencies of the keys:" << endl;

    for (int i = 0; i < n; i++) {

        cin >> freq[i];

    }

    // Calculate and print the minimum cost of the optimal BST

    int minCost = optimalBST(keys, freq, n);

    cout << "The minimum cost of the optimal binary search tree is: " << minCost << endl;

    return 0;

}

**Q.2) Write a program to implement Sum of Subset by Backtracking**

Ans:

#include <iostream>

#include <vector>

using namespace std;

// Function to print the subset

void printSubset(const vector<int>& subset) {

cout << "{ ";

for (int num : subset) {

cout << num << " ";

}

cout << "}" << endl;

}

// Backtracking function to find subsets that sum to a given sum

void findSubsetSum(const vector<int>& set, vector<int>& subset, int index, int sum, int target) {

if (sum == target) {

printSubset(subset);

return;

}

if (index == set.size() || sum > target) {

return;

}

// Include the current element

subset.push\_back(set[index]);

findSubsetSum(set, subset, index + 1, sum + set[index], target);

// Exclude the current element

subset.pop\_back();

findSubsetSum(set, subset, index + 1, sum, target);

}

int main() {

int n, target;

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

cin >> n;

vector<int> set(n);

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

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

cin >> set[i];

}

cout << "Enter the target sum: ";

cin >> target;

vector<int> subset;

cout << "Subsets that sum to " << target << " are:" << endl;

findSubsetSum(set, subset, 0, 0, target);

return 0;

}

**Slip 10**

**Q.1) Write a program to implement Huffman Code using greedy methods**

Ans:

#include <iostream>

#include <queue>

#include <vector>

#include <unordered\_map>

#include <string>

using namespace std;

// Define a structure to represent a node in the Huffman Tree

struct Node {

char data;

int freq;

Node\* left;

Node\* right;

Node(char data, int freq) {

this->data = data;

this->freq = freq;

left = right = nullptr;

}

};

// Compare function to help with priority queue sorting

struct compare {

bool operator()(Node\* l, Node\* r) {

return l->freq > r->freq;

}

};

// Recursive function to generate the Huffman codes

void generateCodes(Node\* root, string str, unordered\_map<char, string>& huffmanCode) {

if (root == nullptr) return;

if (!root->left && !root->right) {

huffmanCode[root->data] = str;

}

generateCodes(root->left, str + "0", huffmanCode);

generateCodes(root->right, str + "1", huffmanCode);

}

// Function to implement Huffman Coding

void huffmanCoding(const string& input) {

unordered\_map<char, int> freq;

// Step 1: Calculate frequency of each character in the input string

for (char ch : input) {

freq[ch]++;

}

// Step 2: Create a priority queue to build the Huffman tree

priority\_queue<Node\*, vector<Node\*>, compare> minHeap;

for (auto& pair : freq) {

minHeap.push(new Node(pair.first, pair.second));

}

// Step 3: Build the Huffman Tree

while (minHeap.size() > 1) {

Node\* left = minHeap.top();

minHeap.pop();

Node\* right = minHeap.top();

minHeap.pop();

Node\* internalNode = new Node('$', left->freq + right->freq);

internalNode->left = left;

internalNode->right = right;

minHeap.push(internalNode);

}

// Step 4: Generate Huffman codes from the tree

Node\* root = minHeap.top();

unordered\_map<char, string> huffmanCode;

generateCodes(root, "", huffmanCode);

// Step 5: Print the Huffman codes

cout << "Huffman Codes for the given input:" << endl;

for (auto& pair : huffmanCode) {

cout << pair.first << ": " << pair.second << endl;

}

}

int main() {

string input;

cout << "Enter the input string: ";

getline(cin, input);

huffmanCoding(input);

return 0;

}

**Q-2) Write a program to solve 4 Queens Problem using Backtracking**

Ans:

#include <iostream>

#include <vector>

using namespace std;

const int N = 4; // Size of the board (4x4)

// Function to check if it's safe to place a queen at board[row][col]

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

// Check the column

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

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

return false;

}

}

// Check the upper left diagonal

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

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

return false;

}

}

// Check the upper right diagonal

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

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

return false;

}

}

return true;

}

// Backtracking function to solve the 4 Queens problem

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

if (row == N) {

// All queens are placed successfully

return true;

}

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

// Check if it's safe to place the queen at (row, col)

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

board[row][col] = 1; // Place the queen

// Recur to place the next queen

if (solveNQueens(board, row + 1)) {

return true;

}

// If placing queen in (row, col) doesn't lead to a solution, backtrack

board[row][col] = 0;

}

}

return false; // If no position is found

}

// Function to print the board

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

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

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

cout << (board[i][j] == 1 ? "Q" : ".") << " ";

}

cout << endl;

}

}

int main() {

vector<vector<int>> board(N, vector<int>(N, 0)); // Create a 4x4 board initialized with 0

if (solveNQueens(board, 0)) {

cout << "Solution to the 4 Queens Problem:" << endl;

printBoard(board);

} else {

cout << "No solution exists" << endl;

}

return 0;

}

**Slip 11**

**Q.1) Write a programs to implement DFS (Depth First Search) and determine the time complexity for the same.**

Ans:

#include <iostream>

#include <vector>

#include <stack>

using namespace std;

// Function to perform DFS traversal

void DFS(int vertex, vector<vector<int>>& adjList, vector<bool>& visited) {

stack<int> s;

s.push(vertex);

visited[vertex] = true;

while (!s.empty()) {

int current = s.top();

s.pop();

cout << current << " "; // Print the vertex

// Traverse all the adjacent vertices of the current vertex

for (int neighbor : adjList[current]) {

if (!visited[neighbor]) {

visited[neighbor] = true;

s.push(neighbor);

}

}

}

}

int main() {

int vertices, edges;

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

cin >> vertices >> edges;

vector<vector<int>> adjList(vertices);

vector<bool> visited(vertices, false);

cout << "Enter edges (start vertex, end vertex):" << endl;

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

int u, v;

cin >> u >> v;

adjList[u].push\_back(v);

adjList[v].push\_back(u); // For undirected graph

}

cout << "DFS Traversal starting from vertex 0: ";

DFS(0, adjList, visited);

return 0;

}

**Q.2 Write a program to find shortest paths from a given vertex in a weighted connected graph, to other vertices using Dijkstra’s algorithm.**

Ans:

#include <iostream>

#include <vector>

#include <queue>

#include <climits>

using namespace std;

typedef pair<int, int> pii; // Pair to store (distance, vertex)

// Function to implement Dijkstra's algorithm

void dijkstra(int start, int vertices, vector<vector<pii>>& adjList) {

vector<int> dist(vertices, INT\_MAX); // Distance array, initialized to infinity

dist[start] = 0; // Distance to the source is 0

priority\_queue<pii, vector<pii>, greater<pii>> pq; // Min-heap priority queue

pq.push({0, start}); // Push the source with distance 0

while (!pq.empty()) {

int u = pq.top().second;

int d = pq.top().first;

pq.pop();

if (d > dist[u]) continue; // Skip if the distance is not optimal

// Explore all the adjacent vertices of u

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

int v = edge.first;

int weight = edge.second;

if (dist[u] + weight < dist[v]) {

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

pq.push({dist[v], v}); // Push the updated distance to the queue

}

}

}

// Output the shortest distances

cout << "Shortest distances from vertex " << start << " are:" << endl;

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

if (dist[i] == INT\_MAX) {

cout << "Vertex " << i << " is unreachable." << endl;

} else {

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

}

}

}

int main() {

int vertices, edges;

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

cin >> vertices >> edges;

vector<vector<pii>> adjList(vertices);

cout << "Enter edges (start vertex, end vertex, weight):" << endl;

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

int u, v, w;

cin >> u >> v >> w;

adjList[u].push\_back({v, w});

adjList[v].push\_back({u, w}); // For undirected graph

}

int start;

cout << "Enter the source vertex: ";

cin >> start;

dijkstra(start, vertices, adjList);

return 0;

}

**Slip 12**

**Q.1) Write a program to implement BFS (Breadth First Search) and determine the time complexity for the same.**

Ans:

#include <iostream>

#include <vector>

#include <queue>

using namespace std;

// Function to perform BFS traversal

void BFS(int start, vector<vector<int>>& adjList, vector<bool>& visited) {

queue<int> q;

visited[start] = true;

q.push(start);

while (!q.empty()) {

int current = q.front();

q.pop();

cout << current << " "; // Print the vertex

// Traverse all the adjacent vertices of the current vertex

for (int neighbor : adjList[current]) {

if (!visited[neighbor]) {

visited[neighbor] = true;

q.push(neighbor);

}

}

}

}

int main() {

int vertices, edges;

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

cin >> vertices >> edges;

vector<vector<int>> adjList(vertices);

vector<bool> visited(vertices, false);

cout << "Enter edges (start vertex, end vertex):" << endl;

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

int u, v;

cin >> u >> v;

adjList[u].push\_back(v);

adjList[v].push\_back(u); // For undirected graph

}

cout << "BFS Traversal starting from vertex 0: ";

BFS(0, adjList, visited);

return 0;

}

**Q.2) Write a program to sort a given set of elements using the Selection sort method and determine the time required to sort the elements**.

Ans:

#include <iostream>

#include <vector>

#include <ctime>

using namespace std;

// Function to perform Selection Sort

void selectionSort(vector<int>& arr) {

int n = arr.size();

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

int minIndex = i;

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

if (arr[j] < arr[minIndex]) {

minIndex = j;

}

}

// Swap the found minimum element with the first element

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

}

}

int main() {

int n;

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

cin >> n;

vector<int> arr(n);

// Generating random numbers

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

arr[i] = rand() % 1000; // Random numbers between 0 and 999

}

// Print the array before sorting

cout << "Array before sorting: ";

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

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

}

cout << endl;

// Start timing

clock\_t start = clock();

selectionSort(arr);

// End timing

clock\_t end = clock();

// Print the sorted array

cout << "Array after sorting: ";

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

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

}

cout << endl;

// Calculate time taken for sorting

double time\_taken = double(end - start) / CLOCKS\_PER\_SEC;

cout << "Time taken to sort the array: " << time\_taken << " seconds" << endl;

return 0;

}

**Slip 13**

**Q.1) Write a program to find minimum number of multiplications in Matrix Chain Multiplication.**

Ans:

#include <iostream>

#include <vector>

#include <climits>

using namespace std;

// Function to find the minimum number of multiplications needed

int matrixChainOrder(const vector<int>& dims) {

int n = dims.size();

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

// dp[i][j] will hold the minimum number of multiplications needed to multiply matrices i through j

for (int length = 2; length < n; length++) {

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

int j = i + length - 1;

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

for (int k = i; k <= j - 1; k++) {

int q = dp[i][k] + dp[k + 1][j] + dims[i - 1] \* dims[k] \* dims[j];

dp[i][j] = min(dp[i][j], q);

}

}

}

return dp[1][n - 1]; // The result will be in dp[1][n-1]

}

int main() {

int n;

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

cin >> n;

vector<int> dims(n + 1);

cout << "Enter the dimensions of matrices:" << endl;

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

cin >> dims[i];

}

cout << "Minimum number of multiplications: " << matrixChainOrder(dims) << endl;

return 0;

}

**Q.2) Write a program to implement an optimal binary search tree and also calculate the best case and worst-case complexity.**

Ans:

#include <iostream>

#include <vector>

#include <climits>

using namespace std;

// Function to find the cost of optimal BST

int optimalBST(const vector<int>& keys, const vector<int>& freq, int n) {

vector<vector<int>> dp(n, vector<int>(n, 0)); // dp[i][j] will store the minimum cost for keys[i..j]

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

// Filling the dp table

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

dp[i][i] = freq[i]; // Cost of a single key is just its frequency

}

// Build the dp table for chains of increasing length

for (int length = 2; length <= n; length++) {

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

int j = i + length - 1;

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

int totalFreq = 0;

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

totalFreq += freq[k];

}

// Try every key as the root and calculate the minimum cost

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

int costLeft = (k > i) ? dp[i][k - 1] : 0;

int costRight = (k < j) ? dp[k + 1][j] : 0;

dp[i][j] = min(dp[i][j], costLeft + costRight + totalFreq);

}

}

}

return dp[0][n - 1]; // The result is the minimum cost for the entire tree

}

int main() {

int n;

cout << "Enter number of keys: ";

cin >> n;

vector<int> keys(n);

vector<int> freq(n);

cout << "Enter the keys: ";

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

cin >> keys[i];

}

cout << "Enter the frequencies of the keys: ";

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

cin >> freq[i];

}

cout << "Minimum cost of optimal BST: " << optimalBST(keys, freq, n) << endl;

return 0;

}

**Slip 14:**

**Q.1) Write a program to sort a list of n numbers in ascending order using Insertion sort and determine the time required to sort the elements.**

Ans:

#include <iostream>

#include <vector>

#include <chrono> // For measuring time

using namespace std;

using namespace std::chrono;

// Function to perform Insertion Sort

void insertionSort(vector<int>& arr) {

int n = arr.size();

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

int key = arr[i];

int j = i - 1;

// Move elements of arr[0..i-1] that are greater than key to one position ahead of their current position

while (j >= 0 && arr[j] > key) {

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

j--;

}

arr[j + 1] = key;

}

}

int main() {

int n;

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

cin >> n;

vector<int> arr(n);

cout << "Enter the elements: ";

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

cin >> arr[i];

}

// Start time

auto start = high\_resolution\_clock::now();

// Perform Insertion Sort

insertionSort(arr);

// End time

auto stop = high\_resolution\_clock::now();

// Calculate the duration

auto duration = duration\_cast<microseconds>(stop - start);

// Output the sorted array

cout << "Sorted elements: ";

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

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

}

cout << endl;

// Output the time taken to sort

cout << "Time taken to sort the elements: " << duration.count() << " microseconds" << endl;

return 0;

}

**Q.2) Write a program to implement DFS and BFS. Compare the time complexity.**

Ans :

#include <iostream>

#include <vector>

#include <queue>

#include <stack>

using namespace std;

// DFS Recursive function

void DFS(int node, vector<vector<int>>& adj, vector<bool>& visited) {

visited[node] = true;

cout << node << " ";

// Visit all the adjacent nodes of the current node

for (int i : adj[node]) {

if (!visited[i]) {

DFS(i, adj, visited);

}

}

}

// BFS function

void BFS(int start, vector<vector<int>>& adj) {

vector<bool> visited(adj.size(), false);

queue<int> q;

visited[start] = true;

q.push(start);

while (!q.empty()) {

int node = q.front();

q.pop();

cout << node << " ";

// Visit all the adjacent nodes of the current node

for (int i : adj[node]) {

if (!visited[i]) {

visited[i] = true;

q.push(i);

}

}

}

}

int main() {

int n, m;

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

cin >> n >> m;

vector<vector<int>> adj(n);

cout << "Enter the edges (u v): " << endl;

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

int u, v;

cin >> u >> v;

adj[u].push\_back(v);

adj[v].push\_back(u); // For undirected graph

}

// Perform DFS

vector<bool> visited(n, false);

cout << "DFS starting from node 0: ";

DFS(0, adj, visited);

cout << endl;

// Perform BFS

cout << "BFS starting from node 0: ";

BFS(0, adj);

cout << endl;

return 0;

}

**Slip 15**

**Q.1) Write a program to implement to find out solution for 0/1 knapsack problem using LCBB (Least Cost Branch and Bound).**

Ans:

#include <iostream>

#include <vector>

#include <queue>

#include <algorithm>

using namespace std;

// Structure to represent an item with its value and weight

struct Item {

    int value, weight;

};

// Structure to represent a node in the search tree

struct Node {

    int level, profit, weight;

    float bound;

};

// Comparison function for priority queue (to get node with highest bound first)

bool operator<(Node a, Node b) {

    return a.bound < b.bound;

}

// Function to calculate the bound for a node

float calculateBound(Node u, int n, int W, vector<Item>& items) {

    if (u.weight >= W) {

        return 0;

    }

    float bound = u.profit;

    int j = u.level + 1;

    int totalWeight = u.weight;

    // Add items to the knapsack until the weight limit is reached

    while (j < n && totalWeight + items[j].weight <= W) {

        totalWeight += items[j].weight;

        bound += items[j].value;

        j++;

    }

    // Take the fraction of the next item if the knapsack is not full

    if (j < n) {

        bound += (W - totalWeight) \* (float(items[j].value) / float(items[j].weight));

    }

    return bound;

}

// Function to find the maximum profit using LCBB

int knapsackLCBB(int W, vector<Item>& items, int n) {

    // Sort items by value/weight ratio in descending order

    sort(items.begin(), items.end(), [](Item a, Item b) {

        return (float(a.value) / float(a.weight)) > (float(b.value) / float(b.weight));

    });

    // Priority queue for the nodes, sorted by bound

    priority\_queue<Node> pq;

    // Initializing the first node

    Node u, v;

    u.level = -1;

    u.profit = 0;

    u.weight = 0;

    u.bound = 0.0;

    // Calculate the bound for the first node

    u.bound = calculateBound(u, n, W, items);

    pq.push(u);

    int maxProfit = 0;

    // Loop to explore the tree

    while (!pq.empty()) {

        u = pq.top();

        pq.pop();

        // If this node's profit is greater than the maximum profit, update maxProfit

        if (u.profit > maxProfit) {

            maxProfit = u.profit;

        }

        // If this node cannot yield a better solution, skip it

        if (u.bound <= maxProfit) {

            continue;

        }

        // Generate the left child (including the current item)

        v.level = u.level + 1;

        v.weight = u.weight + items[v.level].weight;

        v.profit = u.profit + items[v.level].value;

        if (v.weight <= W && v.profit > maxProfit) {

            v.bound = calculateBound(v, n, W, items);

            if (v.bound > maxProfit) {

                pq.push(v);

            }

        }

        // Generate the right child (excluding the current item)

        v.weight = u.weight;

        v.profit = u.profit;

        v.bound = calculateBound(v, n, W, items);

        if (v.bound > maxProfit) {

            pq.push(v);

        }

    }

    return maxProfit;

}

int main() {

    int W, n;

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

    cin >> n;

    vector<Item> items(n);

    cout << "Enter the weight and value of each item:" << endl;

    for (int i = 0; i < n; i++) {

        cin >> items[i].weight >> items[i].value;

    }

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

    cin >> W;

    int maxProfit = knapsackLCBB(W, items, n);

    cout << "Maximum profit is: " << maxProfit << endl;

    return 0;

}

**Q.2) Write a program to implement Graph Coloring Algorithm**

Ans:

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

// Function to perform graph coloring

bool isSafe(int v, vector<vector<int>>& graph, vector<int>& color, int c) {

    for (int i = 0; i < graph.size(); i++) {

        if (graph[v][i] == 1 && color[i] == c) {

            return false;

        }

    }

    return true;

}

// Function to solve graph coloring problem

bool graphColoringUtil(vector<vector<int>>& graph, int m, vector<int>& color, int v) {

    if (v == graph.size()) {

        return true;

    }

    for (int c = 1; c <= m; c++) {

        if (isSafe(v, graph, color, c)) {

            color[v] = c;

            if (graphColoringUtil(graph, m, color, v + 1)) {

                return true;

            }

            color[v] = 0;

        }

    }

    return false;

}

void graphColoring(vector<vector<int>>& graph, int m) {

    vector<int> color(graph.size(), 0);

    if (graphColoringUtil(graph, m, color, 0)) {

        cout << "Graph coloring solution: " << endl;

        for (int i = 0; i < graph.size(); i++) {

            cout << "Vertex " << i << " ---> Color " << color[i] << endl;

        }

    } else {

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

    }

}

int main() {

    int V, E, m;

    cout << "Enter number of vertices: ";

    cin >> V;

    cout << "Enter number of edges: ";

    cin >> E;

    vector<vector<int>> graph(V, vector<int>(V, 0));

    cout << "Enter the edges (u v): " << endl;

    for (int i = 0; i < E; i++) {

        int u, v;

        cin >> u >> v;

        graph[u][v] = graph[v][u] = 1;

    }

    cout << "Enter number of colors: ";

    cin >> m;

    graphColoring(graph, m);

    return 0;

}

**Slip 16**

**Q.1) Write a program to implement to find out solution for 0/1 knapsack problem using dynamic programming.**

Ans:

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

// Function to solve the 0/1 Knapsack problem using dynamic programming

int knapsack(int W, vector<int>& weights, vector<int>& values, int n) {

// Create a 2D table to store the maximum value at each subproblem

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

// Fill the DP table

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

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

// If the weight of the current item is less than or equal to the capacity

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

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

} else {

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

}

}

}

// Return the maximum value that can be achieved with the given weight capacity

return dp[n][W];

}

int main() {

int n, W;

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

cin >> n;

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

cout << "Enter the weight and value for each item:" << endl;

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

cin >> weights[i] >> values[i];

}

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

cin >> W;

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

cout << "Maximum value in the knapsack: " << maxValue << endl;

return 0;

}

**Q.2) Write a program to determine if a given graph is a Hamiltonian cycle or not.**

Ans:

#include <iostream>

#include <vector>

using namespace std;

// Utility function to check if the current vertex can be added to the Hamiltonian Cycle

bool isSafe(int v, vector<vector<int>>& graph, vector<int>& path, int pos) {

    // Check if this vertex is an adjacent vertex of the previously added vertex.

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

        return false;

    }

    // Check if the vertex has already been included in the path.

    for (int i = 0; i < pos; i++) {

        if (path[i] == v) {

            return false;

        }

    }

    return true;

}

// Utility function to solve the Hamiltonian Cycle problem using backtracking

bool hamiltonianCycleUtil(vector<vector<int>>& graph, vector<int>& path, int pos, int V) {

    // If all vertices are included in the cycle

    if (pos == V) {

        // And if there is an edge from the last vertex to the first vertex

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

            return true;

        } else {

            return false;

        }

    }

    // Try different vertices as the next candidate in the Hamiltonian Cycle.

    for (int v = 1; v < V; v++) {

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

            path[pos] = v;

            // Recur to construct the rest of the path

            if (hamiltonianCycleUtil(graph, path, pos + 1, V)) {

                return true;

            }

            // If adding vertex v doesn't lead to a solution, remove it

            path[pos] = -1;

        }

    }

    return false;

}

// Function to check if there is a Hamiltonian Cycle

bool hamiltonianCycle(vector<vector<int>>& graph, int V) {

    vector<int> path(V, -1);

    // Let the first vertex in the path be 0

    path[0] = 0;

    // Try to find a Hamiltonian Cycle using backtracking

    if (hamiltonianCycleUtil(graph, path, 1, V)) {

        cout << "Hamiltonian Cycle found: ";

        for (int i = 0; i < V; i++) {

            cout << path[i] << " ";

        }

        cout << endl;

        return true;

    }

    cout << "No Hamiltonian Cycle found" << endl;

    return false;

}

int main() {

    int V, E;

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

    cin >> V;

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

    cin >> E;

    // Create the adjacency matrix for the graph

    vector<vector<int>> graph(V, vector<int>(V, 0));

    cout << "Enter the edges (u v) in the format u v:" << endl;

    for (int i = 0; i < E; i++) {

        int u, v;

        cin >> u >> v;

        graph[u][v] = graph[v][u] = 1;

    }

    // Check for the Hamiltonian cycle

    hamiltonianCycle(graph, V);

    return 0;

}

**Slip 17:**

**Q.1) Write a program to implement solve ‘N’ Queens Problem using Backtracking.**

Ans:

#include <iostream>

#include <vector>

using namespace std;

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

// Check column

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

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

return false;

// Check upper left diagonal

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

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

return false;

// Check upper right diagonal

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

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

return false;

return true;

}

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

if (row == n)

return true;

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

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

board[row][col] = 1;

if (solveNQueens(board, row + 1, n))

return true;

board[row][col] = 0; // Backtrack

}

}

return false;

}

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

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

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

cout << (board[i][j] ? "Q " : ". ");

}

cout << endl;

}

}

int main() {

int n;

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

cin >> n;

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

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

cout << "Solution found:\n";

printBoard(board, n);

} else {

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

}

return 0;

}

**Q.2) Write a program to find out solution for 0/1 knapsack problem.**

Ans:

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int knapsack(int W, vector<int>& weights, vector<int>& values, int n) {

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

    for (int i = 1; i <= n; i++) {

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

            if (weights[i - 1] <= w)

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

            else

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

        }

    }

    return dp[n][W];

}

int main() {

    int n, W;

    cout << "Enter number of items: ";

    cin >> n;

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

    cout << "Enter weights and values of items:\n";

    for (int i = 0; i < n; i++) {

        cin >> weights[i] >> values[i];

    }

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

    cin >> W;

    int result = knapsack(W, weights, values, n);

    cout << "Maximum value in knapsack: " << result << endl;

    return 0;

}

**Slip 18:**

**Q.1) Write a program to implement Graph Coloring Algorithm.**

Ans :

#include <iostream>

#include <vector>

using namespace std;

bool isSafe(int v, vector<vector<int>>& graph, vector<int>& color, int c, int V) {

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

if (graph[v][i] && color[i] == c)

return false;

return true;

}

bool graphColoringUtil(vector<vector<int>>& graph, int m, vector<int>& color, int v, int V) {

if (v == V)

return true;

for (int c = 1; c <= m; c++) {

if (isSafe(v, graph, color, c, V)) {

color[v] = c;

if (graphColoringUtil(graph, m, color, v + 1, V))

return true;

color[v] = 0; // Backtrack

}

}

return false;

}

bool graphColoring(vector<vector<int>>& graph, int m, int V) {

vector<int> color(V, 0);

if (!graphColoringUtil(graph, m, color, 0, V)) {

cout << "Solution does not exist.\n";

return false;

}

cout << "Solution Exists: Following are the assigned colors:\n";

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

cout << "Vertex " << i << " ---> Color " << color[i] << endl;

return true;

}

int main() {

int V = 4; // Number of vertices

vector<vector<int>> graph = {

{0, 1, 1, 1},

{1, 0, 1, 0},

{1, 1, 0, 1},

{1, 0, 1, 0}

};

int m = 3; // Number of colors

graphColoring(graph, m, V);

return 0;

}

**Q.2) Write a program to find out live node, E node and dead node from a given graph.**

Ans:

#include <iostream>

#include <queue>

#include <vector>

using namespace std;

void classifyNodes(vector<vector<int>>& graph, int start) {

int n = graph.size();

vector<bool> visited(n, false);

queue<int> q;

vector<int> deadNodes;

q.push(start);

visited[start] = true;

while (!q.empty()) {

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

cout << "E-Node: " << eNode << endl;

deadNodes.push\_back(eNode);

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

if (graph[eNode][i] == 1 && !visited[i]) {

cout << "Live Node: " << i << endl;

q.push(i);

visited[i] = true;

}

}

}

cout << "\nDead Nodes: ";

for (int d : deadNodes)

cout << d << " ";

cout << endl;

}

int main() {

vector<vector<int>> graph = {

{0, 1, 1, 0},

{1, 0, 1, 1},

{1, 1, 0, 0},

{0, 1, 0, 0}

};

int start = 0;

classifyNodes(graph, start);

return 0;

}

**Slip 19:**

**Q.1) Write a program to determine if a given graph is a Hamiltonian cycle or Not.**

Ans:

#include <iostream>

#include <vector>

using namespace std;

#define V 5

bool isSafe(int v, bool graph[V][V], vector<int>& path, int pos) {

if (!graph[path[pos - 1]][v])

return false;

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

if (path[i] == v)

return false;

return true;

}

bool hamCycleUtil(bool graph[V][V], vector<int>& path, int pos) {

if (pos == V) {

return graph[path[pos - 1]][path[0]] == 1;

}

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

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

path[pos] = v;

if (hamCycleUtil(graph, path, pos + 1))

return true;

path[pos] = -1; // Backtrack

}

}

return false;

}

bool hamCycle(bool graph[V][V]) {

vector<int> path(V, -1);

path[0] = 0;

if (!hamCycleUtil(graph, path, 1)) {

cout << "No Hamiltonian Cycle exists\n";

return false;

}

cout << "Hamiltonian Cycle Exists: ";

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

cout << path[i] << " ";

cout << path[0] << endl;

return true;

}

int main() {

bool graph1[V][V] = {

{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}

};

hamCycle(graph1);

return 0;

}

**Q.2) Write a program to show board configuration of 4 queens’ problem.**

#include <iostream>

#include <vector>

using namespace std;

#define N 4

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

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

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

cout << (board[i][j] ? "Q " : ". ");

cout << endl;

}

cout << endl;

}

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

if (col >= N)

return true;

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

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

board[i][col] = 1;

if (solveNQUtil(board, col + 1))

return true;

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

}

}

return false;

}

bool solveNQ() {

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

if (!solveNQUtil(board, 0)) {

cout << "Solution does not exist\n";

return false;

}

cout << "4 Queens Board Configuration:\n";

printSolution(board);

return true;

}

int main() {

solveNQ();

return 0;

}

**Slip 20:**

**Q.1) Write a program to implement for finding Topological sorting and determine the time complexity for the same.**

Ans:

#include <iostream>

#include <vector>

#include <stack>

using namespace std;

void topologicalSortUtil(int v, vector<bool>& visited, stack<int>& Stack, const vector<vector<int>>& adj) {

visited[v] = true;

for (int u : adj[v]) {

if (!visited[u])

topologicalSortUtil(u, visited, Stack, adj);

}

Stack.push(v);

}

void topologicalSort(int V, const vector<vector<int>>& adj) {

vector<bool> visited(V, false);

stack<int> Stack;

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

if (!visited[i])

topologicalSortUtil(i, visited, Stack, adj);

cout << "Topological Sort: ";

while (!Stack.empty()) {

cout << Stack.top() << " ";

Stack.pop();

}

cout << endl;

}

int main() {

int V = 6; // Number of vertices

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

// Directed edges (example)

adj[5].push\_back(2);

adj[5].push\_back(0);

adj[4].push\_back(0);

adj[4].push\_back(1);

adj[2].push\_back(3);

adj[3].push\_back(1);

topologicalSort(V, adj);

return 0;

}

**Q.2) Write a program to solve N Queens Problem using Backtracking.**

Ans:

#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 solveNQUtil(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 (solveNQUtil(board, col + 1, N))

return true;

board[i][col] = 0;

}

}

return false;

}

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

cout << N << " Queens Board Configuration:\n";

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

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

cout << (board[i][j] ? "Q " : ". ");

cout << endl;

}

}

void solveNQueens(int N) {

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

if (!solveNQUtil(board, 0, N))

cout << "Solution does not exist for " << N << " queens\n";

else

printBoard(board, N);

}

int main() {

int N = 8;

solveNQueens(N);

return 0;

}