**Code:-**

#include <iostream>

using namespace std;

int binarySearch(int arr[], int low, int high, int x) {

if (low > high) {

return -1;

}

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

if (arr[mid] == x) {

return mid;

} else if (arr[mid] < x) {

return binarySearch(arr, mid + 1, high, x);

} else {

return binarySearch(arr, low, mid - 1, x);

}

}

int main() {

int arr[] = {1, 3, 5, 7, 9};

cout << "Enter the element to search: ";

int x;

cin >> x;

int index = binarySearch(arr, 0, sizeof(arr) / sizeof(arr[0]) - 1, x);

if (index != -1) {

cout << "The element is found at index " << index+1 << endl;

} else {

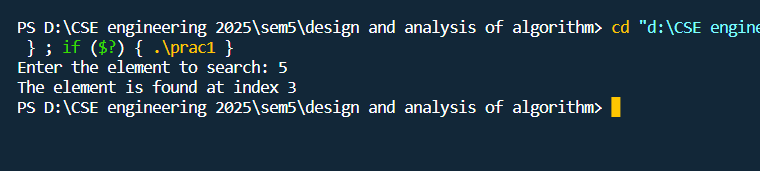
cout << "The element is not found" << endl;

}

return 0;

} 1

**Output:-**

****

1

**Code:- quick sort**

// Quick Sort

#include <iostream>

#include <chrono>

using namespace std;

using namespace chrono;

int partition(int arr[], int start, int end)

{

int pivot = arr[start];

int count = 0;

for (int i = start + 1; i <= end; i++)

{

if (arr[i] <= pivot)

count++;

}

int pivotIndex = start + count;

swap(arr[pivotIndex], arr[start]);

int i = start, j = end;

while (i < pivotIndex && j > pivotIndex)

{

while (arr[i] <= pivot)

{

i++; 2.1

}

while (arr[j] > pivot)

{

j--;

}

if (i < pivotIndex && j > pivotIndex)

{

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

}

}

return pivotIndex;

}

void quickSort(int arr[], int start, int end)

{

if (start >= end)

return;

int p = partition(arr, start, end);

quickSort(arr, start, p - 1);

quickSort(arr, p + 1, end);

}

void display(int \*array, int size)

{ 2.2

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

cout << array[i] << " ";

cout << endl;

}

int main()

{

int n;

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

cin >> n;

int arr[n];

cout << "Enter elements:" << endl;

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

{

cin >> arr[i];

}

cout << "Array before Sorting: ";

display(arr, n);

// Start the timer

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

// Sort the array using merge sort

quickSort(arr, 0, n - 1);

// Stop the timer

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

// Get the duration

auto duration = duration\_cast<microseconds>(end - start); 2.3

cout << "Array after Sorting: ";

display(arr, n);

// Print the time taken to sort the array

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

return 0;

}

**Code:- merge sort**

// merge sort

#include <iostream>

#include <chrono>

using namespace std;

using namespace chrono;

void swapping(int &a, int &b)

{

int temp;

temp = a;

a = b;

b = temp;

}

void display(int \*array, int size)

{

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

cout << array[i] << " ";

cout << endl;

} 2.4

void merge(int \*array, int l, int m, int r)

{

int i, j, k, nl, nr;

nl = m - l + 1;

nr = r - m;

int larr[nl], rarr[nr];

for (i = 0; i < nl; i++)

larr[i] = array[l + i];

for (j = 0; j < nr; j++)

rarr[j] = array[m + 1 + j];

i = 0;

j = 0;

k = l;

while (i < nl && j < nr)

{

if (larr[i] <= rarr[j])

{

array[k] = larr[i];

i++;

}

else

{

array[k] = rarr[j];

j++;

}

k++;

}

while (i < nl) 2.5

{

array[k] = larr[i];

i++;

k++;

}

while (j < nr)

{

array[k] = rarr[j];

j++;

k++;

}

}

void mergeSort(int \*array, int l, int r)

{

int m;

if (l < r)

{

int m = l + (r - l) / 2;

mergeSort(array, l, m);

mergeSort(array, m + 1, r);

merge(array, l, m, r);

}

}

int main()

{

int n;

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

cin >> n;

int arr[n];

cout << "Enter elements:" << endl; 2.6

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

{

cin >> arr[i];

}

cout << "Array before Sorting: ";

display(arr, n);

// Start the timer

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

// Sort the array using merge sort

mergeSort(arr, 0, n - 1);

// Stop the timer

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

// Get the duration

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

mergeSort(arr, 0, n - 1); //(n-1) for last index

cout << "Array after Sorting: ";

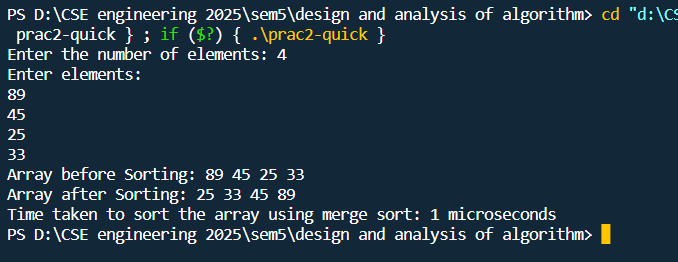
display(arr, n);

// Print the time taken to sort the array

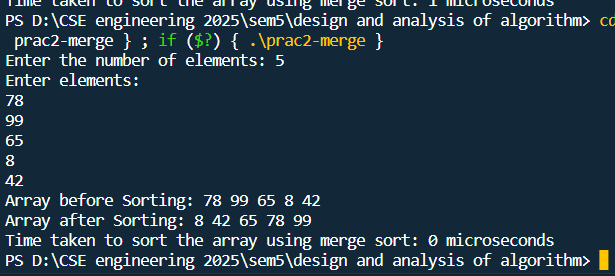
cout << "Time taken to sort the array using merge sort: " << duration.count() << " microseconds" << endl;

}

2.7

**Output:- quick sort**

**Output:- merge sort**



2.8

**Code:-**

#include <iostream>

#include <bits/stdc++.h>

using namespace std;

typedef struct

{

int v;

int w;

float d;

} Item;

void input(Item items[], int sizeOfItems)

{

cout << "Enter total " << sizeOfItems << " item's values and weight" << endl;

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

{

cout << "Enter " << i + 1 << " Value: ";

cin >> items[i].v;

cout << "Enter " << i + 1 << " Weight: ";

cin >> items[i].w;

}

}

void display(Item items[], int sizeOfItems)

{

int i;

cout << "values: ";

for (i = 0; i < sizeOfItems; i++)

{

cout << items[i].v << "\t";

}

cout << endl 3.1

<< "weight: ";

for (i = 0; i < sizeOfItems; i++)

{

cout << items[i].w << "\t";

}

cout << endl;

}

bool compare(Item i1, Item i2)

{

return (i1.d > i2.d);

}

float knapsack(Item items[], int sizeOfItems, int W)

{

int i, j;

float totalValue = 0, totalWeight = 0;

for (i = 0; i < sizeOfItems; i++)

{

items[i].d = (float)items[i].v / items[i].w;

}

sort(items, items + sizeOfItems, compare);

cout << "values : ";

for (i = 0; i < sizeOfItems; i++)

{

cout << items[i].v << "\t";

}

cout << endl

<< "weights: ";

for (i = 0; i < sizeOfItems; i++)

{

cout << items[i].w << "\t"; 3.2

}

cout << endl

<< "ratio : ";

for (i = 0; i < sizeOfItems; i++)

{

cout << items[i].d << "\t";

}

cout << endl;

for (i = 0; i < sizeOfItems; i++)

{

if (totalWeight + items[i].w <= W)

{

totalValue += items[i].v;

totalWeight += items[i].w;

}

else

{

int wt = W - totalWeight;

totalValue += (wt \* items[i].d);

totalWeight += wt;

break;

}

}

cout << "Total weight in bag " << totalWeight << endl;

return totalValue;

}

int main()

{

int W;

Item items[4]; 3.3

input(items, 4);

cout << "Entered data \n";

display(items, 4);

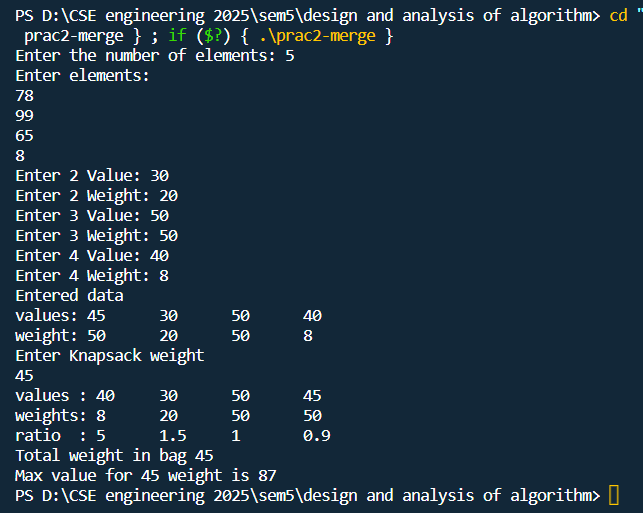
cout << "Enter Knapsack weight \n";

cin >> W;

float mxVal = knapsack(items, 4, W);

cout << "Max value for " << W << " weight is " << mxVal;

}

**Output:-**

3.4

**Code:-**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

const int MAX = 1e6-1;

int root[MAX];

const int nodes = 4, edges = 5;

pair <long long, pair<int, int> > p[MAX];

int parent(int a)

{

while(root[a] != a)

{

root[a] = root[root[a]];

a = root[a];

}

return a;

}

void union\_find(int a, int b)

{

int d = parent(a);

int e = parent(b);

root[d] = root[e];

}

long long kruskal(){

int a, b;

long long cost, minCost = 0;

for(int i = 0 ; i < edges ; ++i) 4.1

{

a = p[i].second.first;

b = p[i].second.second;

cost = p[i].first;

if(parent(a) != parent(b))

{

minCost += cost;

union\_find(a, b);

}

}

return minCost;

}

int main()

{

int x, y;

long long weight, cost, minCost;

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

{

root[i] = i;

}

p[0] = make\_pair(10, make\_pair(0, 1));

p[1] = make\_pair(18, make\_pair(1, 2));

p[2] = make\_pair(13, make\_pair(2, 3));

p[3] = make\_pair(21, make\_pair(0, 2));

p[4] = make\_pair(22, make\_pair(1, 3));

sort(p, p + edges);

minCost = kruskal();

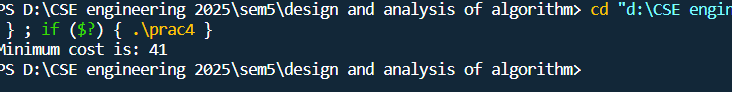
cout << "Minimum cost is: "<< minCost << endl;

return 0;

}

4.2

**Output:-**



4.3

**Code:-**

#include <bits/stdc++.h>

using namespace std;

#define V 5

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

{

int min = INT\_MAX, min\_index;

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

if (mstSet[v] == false && key[v] < min)

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

return min\_index;

}

void printMST(int parent[], int graph[V][V])

{

cout << "Edge \tWeight\n";

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

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

<< graph[i][parent[i]] << " \n";

}

void primMST(int graph[V][V])

{

int parent[V]; 5.1

int key[V];

bool mstSet[V];

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

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

key[0] = 0;

parent[0] = -1;

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] == false && graph[u][v] < key[v])

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

}

printMST(parent, graph);

}

int main()

{

int graph[V][V] = {{0, 2, 0, 6, 0}, 5.2

{2, 0, 3, 8, 5},

{0, 3, 0, 0, 7},

{6, 8, 0, 0, 9},

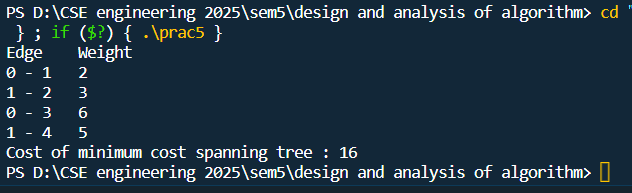
{0, 5, 7, 9, 0}};

primMST(graph);

return 0;

}

**Output:-**



5.3

**Code:-**

#include <iostream>

#include <queue>

#include <unordered\_map>

#include <vector>

using namespace std;

struct HuffmanNode {

char data;

int frequency;

HuffmanNode \*left, \*right;

HuffmanNode(char data, int frequency) : data(data), frequency(frequency), left(nullptr), right(nullptr) {}

};

struct CompareNodes {

bool operator()(HuffmanNode\* a, HuffmanNode\* b) {

return a->frequency > b->frequency;

}

};

HuffmanNode\* buildHuffmanTree(const unordered\_map<char, int>& frequencies) {

priority\_queue<HuffmanNode\*, vector<HuffmanNode\*>, CompareNodes> pq;

for (const auto& pair : frequencies) {

pq.push(new HuffmanNode(pair.first, pair.second)); 6.1

}

while (pq.size() > 1) {

HuffmanNode\* left = pq.top();

pq.pop();

HuffmanNode\* right = pq.top();

pq.pop();

HuffmanNode\* newNode = new HuffmanNode('$', left->frequency + right->frequency);

newNode->left = left;

newNode->right = right;

pq.push(newNode);

}

return pq.top();

}

void printHuffmanCodes(HuffmanNode\* root, string code, unordered\_map<char, string>& huffmanCodes) {

if (!root)

return;

if (root->data != '$') {

cout << root->data << ": " << code << endl;

huffmanCodes[root->data] = code;

}

printHuffmanCodes(root->left, code + "0", huffmanCodes); 6.2

printHuffmanCodes(root->right, code + "1", huffmanCodes);

}

int main() {

string input;

cout << "Enter the input string: ";

cin >> input;

unordered\_map<char, int> frequencies;

for (char c : input) {

frequencies[c]++;

}

HuffmanNode\* root = buildHuffmanTree(frequencies);

unordered\_map<char, string> huffmanCodes;

printHuffmanCodes(root, "", huffmanCodes);

cout << "Huffman Codes:" << endl;

for (const auto& pair : huffmanCodes) {

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

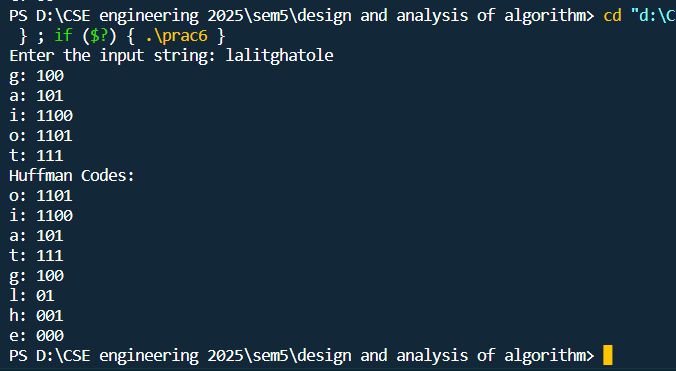
}

return 0;

}

6.3

**Output:-**



6.4

**Code:-**

#include <stdio.h>

#include <stdbool.h>

#include <limits.h>

#define V 6

int minDistance(int dist[], 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;

}

void printSolution(int dist[])

{

printf("Vertex Distance from Source\n");

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

{

printf("%d \t %d\n", i, dist[i]);

}

}

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

{

int dist[V];

bool sptSet[V]; 7.1

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] = {{0, 2, 4, 0, 0, 0}, {2, 0, 3, 7, 0, 0}, {4, 3, 0, 1, 5, 0}, {0, 7, 1, 0, 0, 2}, {0, 0, 5, 0, 0, 6}, {0, 0, 0, 2, 6, 0}};

int src = 0;

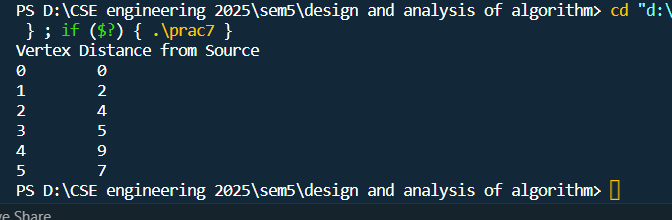
dijkstra(graph, src);

return 0;

}

7.2

**Output:-**



7.3

**Code:-**

#include <iostream>

#include <vector>

#include <string>

using namespace std;

// Function to find the length of the Longest Common Subsequence (LCS)

int findLCSLength(const string& str1, const string& str2) {

int m = str1.length();

int n = str2.length();

// Create a 2D DP table to store the LCS lengths

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

// Fill the DP table using bottom-up dynamic programming

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

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

if (str1[i - 1] == str2[j - 1]) {

dp[i][j] = dp[i - 1][j - 1] + 1; // Common character, extend LCS

} else {

dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]); // Not a common character, take the maximum

}

}

}

return dp[m][n]; // Length of LCS

} 8.1

// Function to find the Longest Common Subsequence (LCS)

string findLCS(const string& str1, const string& str2) {

int m = str1.length();

int n = str2.length();

// Create a 2D DP table to store the LCS lengths

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

// Fill the DP table using bottom-up dynamic programming

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

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

if (str1[i - 1] == str2[j - 1]) {

dp[i][j] = dp[i - 1][j - 1] + 1; // Common character, extend LCS

} else {

dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]); // Not a common character, take the maximum

}

}

}

// Reconstruct the LCS from the DP table

int length = dp[m][n];

string lcs(length, ' ');

int i = m, j = n;

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

if (str1[i - 1] == str2[j - 1]) {

lcs[length - 1] = str1[i - 1];

i--;

j--; 8.2

length--;

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

i--;

} else {

j--;

}

}

return lcs; // Longest Common Subsequence

}

int main() {

string str1, str2;

cout << "Enter the first string: ";

cin >> str1;

cout << "Enter the second string: ";

cin >> str2;

int length = findLCSLength(str1, str2);

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

string lcs = findLCS(str1, str2);

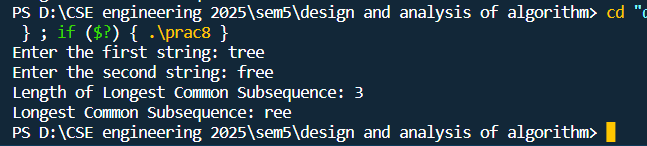
cout << "Longest Common Subsequence: " << lcs << endl;

return 0;

}

8.3

**Output:-**



8.4

**Code:-**

#include <iostream>

#include <vector>

struct Edge {

int source, destination, weight;

};

void bellmanFord(std::vector<Edge>& edges, int numVertices, int source) {

std::vector<int> distance(numVertices, INT\_MAX);

distance[source] = 0;

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

for (const Edge& edge : edges) {

if (distance[edge.source] != INT\_MAX && distance[edge.source] + edge.weight < distance[edge.destination]) {

distance[edge.destination] = distance[edge.source] + edge.weight;

}

}

}

// Check for negative weight cycles

for (const Edge& edge : edges) {

if (distance[edge.source] != INT\_MAX && distance[edge.source] + edge.weight < distance[edge.destination]) {

std::cout << "Graph contains negative weight cycle." << std::endl;

return;

}

}

9.1

// Print shortest distances from the source

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

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

std::cout << "Vertex " << i << ": Distance = " << distance[i] << std::endl;

}

}

int main() {

int numVertices = 5;

std::vector<Edge> edges = {

{0, 1, -1},

{0, 2, 4},

{1, 2, 3},

{1, 3, 2},

{1, 4, 2},

{3, 2, 5},

{3, 1, 1},

{4, 3, -3}

};

int sourceVertex = 0;

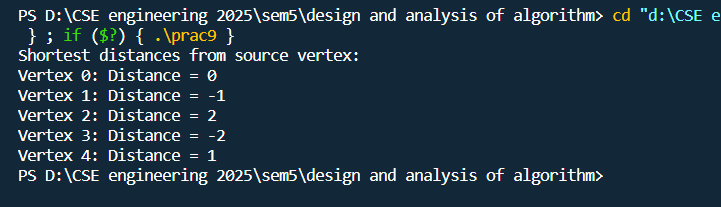
bellmanFord(edges, numVertices, sourceVertex);

return 0;

}

9.2

**Output:-**



9.3

**Code:-**

#include <iostream>

#include <vector>

const int INF = INT\_MAX;

void floydWarshall(std::vector<std::vector<int>>& graph, int numVertices) {

// Initialize the distance matrix with the input graph

std::vector<std::vector<int>> distance(numVertices, std::vector<int>(numVertices, 0));

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

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

distance[i][j] = graph[i][j];

}

}

// Calculate shortest paths using dynamic programming

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

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

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

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

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

}

}

}

}

// Print the shortest paths

std::cout << "All-Pair Shortest Paths:" << std::endl;

for (int i = 0; i < numVertices; ++i) { 10.1

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

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

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

} else {

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

}

}

std::cout << std::endl;

}

}

int main() {

int numVertices = 4;

std::vector<std::vector<int>> graph = {

{0, 3, INF, 0},

{INF, 0, 1, 8},

{INF, INF, 0, INF},

{INF, 2, 4, 0}

};

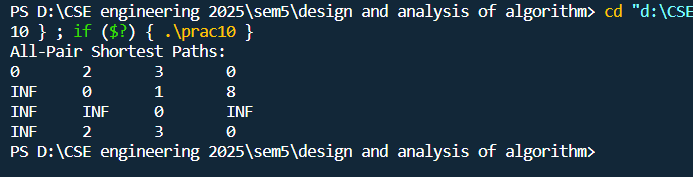
floydWarshall(graph, numVertices);

return 0;

}

10.2

**Output:-**



10.3

**Code:-**

#include <iostream>

#include <list>

#include <queue>

class Graph {

private:

int numVertices;

std::list<int>\* adjList;

public:

Graph(int vertices) {

numVertices = vertices;

adjList = new std::list<int>[vertices];

}

void addEdge(int v, int w) {

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

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

}

void printGraph() {

std::cout << "Graph (Adjacency List):" << std::endl;

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

std::cout << "Vertex " << i << " -> ";

for (const auto& neighbor : adjList[i]) {

std::cout << neighbor << " ";

}

std::cout << std::endl;

} 11.1

}

void bfs(int startVertex) {

bool\* visited = new bool[numVertices];

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

visited[i] = false;

}

std::queue<int> queue;

visited[startVertex] = true;

queue.push(startVertex);

std::cout << "Breadth-First Traversal (starting from vertex " << startVertex << "):" << std::endl;

while (!queue.empty()) {

int currentVertex = queue.front();

std::cout << currentVertex << " ";

queue.pop();

for (const auto& neighbor : adjList[currentVertex]) {

if (!visited[neighbor]) {

visited[neighbor] = true;

queue.push(neighbor);

}

}

}

delete[] visited;

}

};

11.2

int main() {

Graph graph(7); // Create a graph with 7 vertices

// Add edges to the graph

graph.addEdge(0, 1);

graph.addEdge(0, 2);

graph.addEdge(1, 3);

graph.addEdge(1, 4);

graph.addEdge(2, 5);

graph.addEdge(3, 6);

// Print the input graph (adjacency list)

graph.printGraph();

// Perform BFS traversal starting from vertex 0

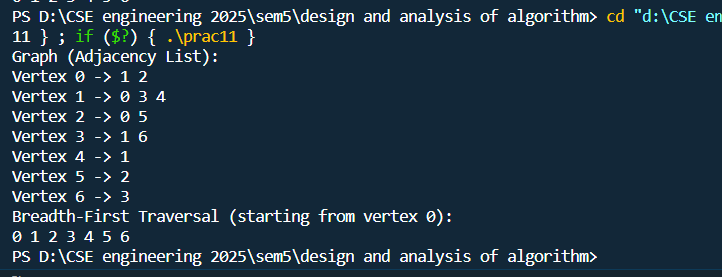
graph.bfs(0);

return 0;

}

11.3

**Output:-**



11.4

**Code:-**

#include <iostream>

#include <list>

class Graph {

private:

int numVertices;

std::list<int>\* adjList;

public:

Graph(int vertices) {

numVertices = vertices;

adjList = new std::list<int>[vertices];

}

void addEdge(int v, int w) {

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

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

}

const std::list<int>& getAdjList(int vertex) const {

return adjList[vertex];

}

void dfsUtil(int vertex, bool\* visited) {

visited[vertex] = true;

std::cout << vertex << " ";

for (const auto& neighbor : adjList[vertex]) {

if (!visited[neighbor]) { 12.1

dfsUtil(neighbor, visited);

}

}

}

void dfs(int startVertex) {

bool\* visited = new bool[numVertices];

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

visited[i] = false;

}

std::cout << "Depth-First Traversal (starting from vertex " << startVertex << "):" << std::endl;

dfsUtil(startVertex, visited);

delete[] visited;

}

};

int main() {

Graph graph(7); // Create a graph with 7 vertices

// Add edges to the graph

graph.addEdge(0, 1);

graph.addEdge(0, 2);

graph.addEdge(1, 3);

graph.addEdge(1, 4);

graph.addEdge(2, 5);

graph.addEdge(3, 6);

std::cout << "Graph (Adjacency List):" << std::endl; 12.2

for (int i = 0; i < 7; i++) {

std::cout << "Vertex " << i << " -> ";

for (const auto& neighbor : graph.getAdjList(i)) {

std::cout << neighbor << " ";

}

std::cout << std::endl;

}

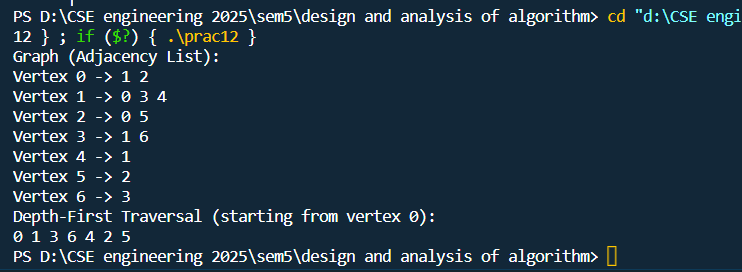
// Perform DFS traversal starting from vertex 0

graph.dfs(0);

return 0;

}

**Output:-**



12.3