1. Linear Search

#include<iostream>

#include<vector>

using namespace std;

int main() {

int n, s;

cout << "Enter size of vector" << endl;

cin >> n;

vector<int> arr(n);

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

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

cin >> arr[i];

}

cout << "Enter Element For Search" << endl;

cin >> s;

// Searching Operation

bool found = false;

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

if (arr[i] == s) {

cout << "Element Found at Index " << i << endl;

found = true;

break;

}

}

if (!found) {

cout << "Element Not Found" << endl;

}

return 0;}



Time Complexity O(n)

Space Complexity O(1)

Stable yes

In-place YES

1. Binary Search

#include <iostream>

#include <vector>

using namespace std;

int main() {

vector<int> arr = {2,5,8,12,16,23,38,56,72,91};

int target = 23;

int left = 0, right = arr.size() - 1;

// Binary Search

while (left <= right) {

int mid = left + (right - left) / 2;

if (arr[mid] == target) {

cout << "Element found at index: " << mid << endl;

return 0;

}

else if (arr[mid] > target) {

right = mid - 1;

}

else {

left = mid + 1;

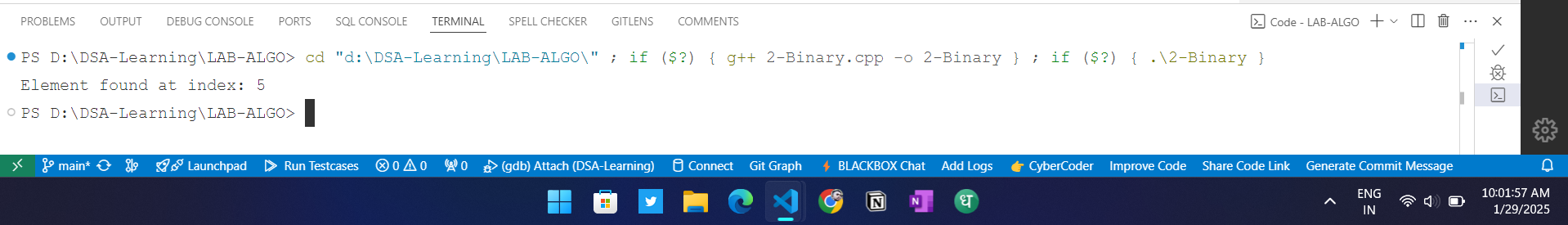
}

}

cout << "Element not found" << endl;

return 0;

}



Time Complexity O(log n)

Space Complexity O(1)

Stable NO

In-place YES

1. BUBBLE SORT

#include <iostream>

#include <vector>

using namespace std;

int main() {

int n;

cout << "Enter Size of Vector: ";

cin >> n;

vector<int> arr(n);

// Element Entry

cout << "Enter Elements: ";

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

cin >> arr[i];

}

// Bubble Sort

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

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

if (arr[j] > arr[j + 1]) {

swap(arr[j], arr[j + 1]);

}

}

}

// Print Sorted Values

cout << "Sorted Elements: ";

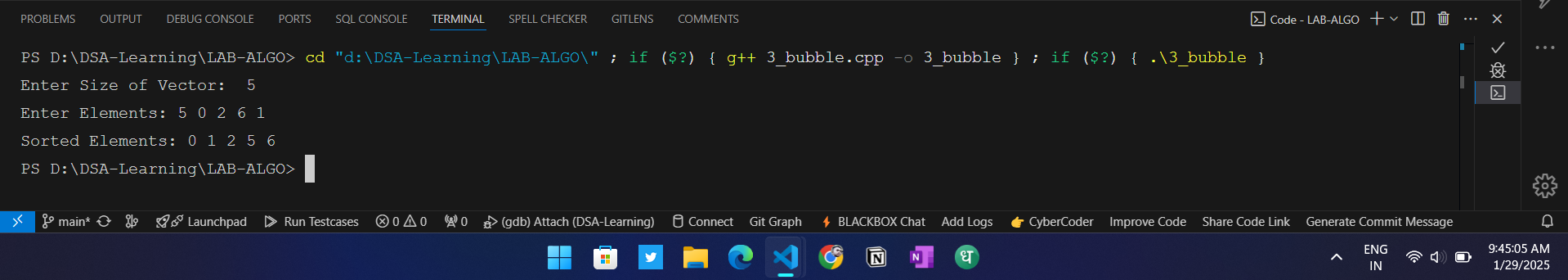
for (int i = 0; i < n; i++) {

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

}

cout << endl;

return 0;

}

Time Complexity O(n^2)

Space Complexity O(1)

Stable YES

In-place YES

1. Insertation Sort

#include <iostream>

#include <vector>

using namespace std;

int main() {

int n;

cout << "Enter Size of Vector: ";

cin >> n;

vector<int> arr(n);

// Element Entry

cout << "Enter Elements: ";

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

cin >> arr[i];

}

// Insertion Sort

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

int key = arr[i];

int j = i - 1;

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

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

j--;

}

arr[j + 1] = key;

}

// Print Sorted Values

cout << "Sorted Elements: ";

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

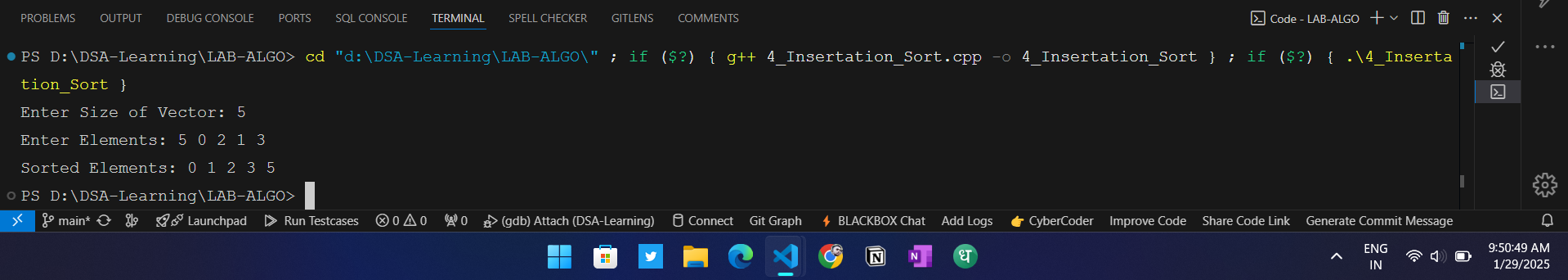
cout << arr[i] << " ";

}

cout << endl;

return 0;

}



Time Complexity O(n^2)

Space Complexity O(1)

Stable YES

In-place YES

1. Selection Sort

#include <iostream>

#include <vector>

using namespace std;

int main() {

int n;

cout << "Enter Size of Vector: ";

cin >> n;

vector<int> arr(n);

// Element Entry

cout << "Enter Elements: ";

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

cin >> arr[i];

}

// Selection Sort

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

}

// Print Sorted Values

cout << "Sorted Elements: ";

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

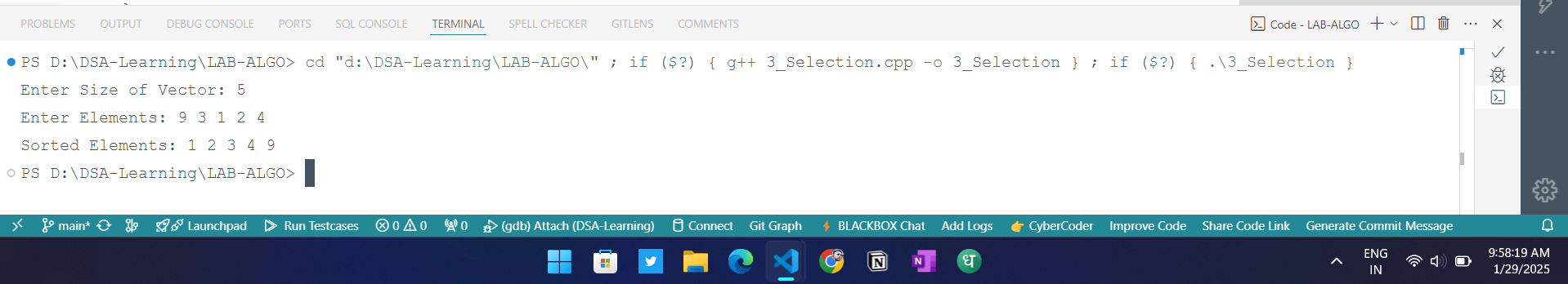
cout << arr[i] << " ";

}

cout << endl;

return 0;

}



Time Complexity O(n^2)

Space Complexity O(1)

Stable NO

In-place YES

6 Quick Sort

#include<iostream>

using namespace std;

// Partition

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

int pos = start;

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

if(arr[i] <= arr[end]){

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

pos++;

}

}

return pos - 1;

}

// Quick Sort

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

if(start >= end)

return;

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

// Left Side

quickSort(arr, start, pivot - 1);

// Right Side

quickSort(arr, pivot, end);

}

int main(){

int arr[] = {10,3,4,1,5,6,3,2,11,9};

quickSort(arr, 0, 9);

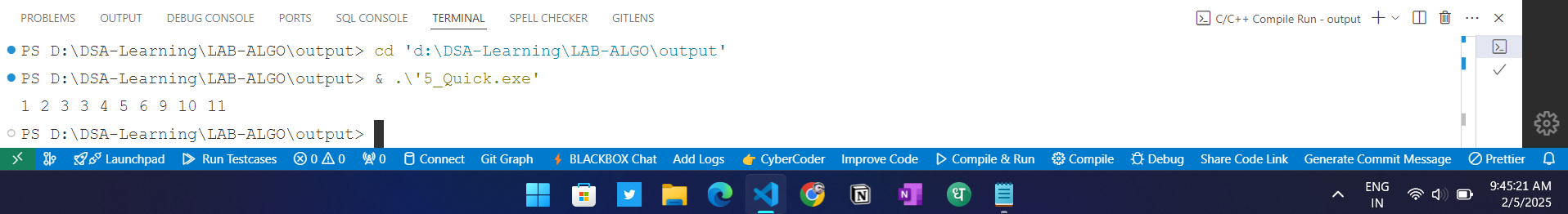
for(int i=0;i<10;i++){

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

}

return 0;

}



**Time Complexity**

* **Best Case**: *O*(*n*log*n*)
* **Worst Case**: O(n2)
* **Average Case**:  *O*(*n*log*n*)

**Space Complexity**

* **Space Complexity**: *O*(log*n*) (due to recursive stack space)

Stability

* **Stable**: No (does not preserve the relative order of equal elements)
* **In-Place**: Yes (requires a constant amount of additional space)

Why is Quick Sort Not Stable?

Quick Sort is not stable because it can swap elements that are equal during the partitioning process. For example, if two equal elements are in different partitions, their relative order may change after the sorting process.

1. Merge Sort

#include <iostream>

#include <vector>

using namespace std;

void Merge(int arr[], int start, int mid, int end) {

vector<int> temp(end - start + 1);

int left = start, right = mid + 1, index = 0;

// Merging the two halves

while (left <= mid && right <= end) {

if (arr[left] <= arr[right]) {

temp[index++] = arr[left++];

} else {

temp[index++] = arr[right++];

}

}

while (left <= mid) {

temp[index++] = arr[left++];

}

while (right <= end) {

temp[index++] = arr[right++];

}

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

arr[start + i] = temp[i];

}

}

// Merge Sort function

void MergeSort(int arr[], int start, int end) {

if (start < end) { // Change condition to allow sorting

int mid = start + (end - start) / 2;

MergeSort(arr, start, mid); // Sort left half

MergeSort(arr, mid + 1, end); // Sort right half

Merge(arr, start, mid, end); // Merge both halves

}

}

int main() {

int arr[] = {6, 3, 1, 2, 8, 9, 10, 7, 3, 10};

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

MergeSort(arr, 0, n - 1);

cout << "Sorted array: ";

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

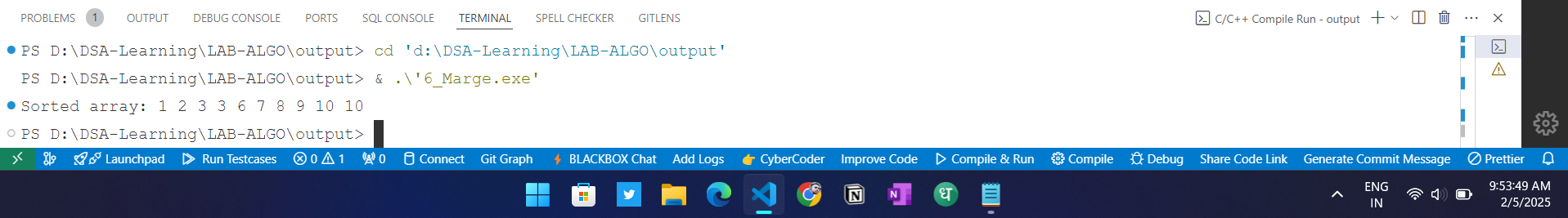
cout << arr[i] << " ";

}

cout << endl;

return 0;

}



**Time Complexity**

**Best Case**: *O*(*n*log*n*)

**Worst Case**: *O*(*n*log*n*)

**Average Case**: *O*(*n*log*n*)

**Space Complexity**

**Space Complexity**: *O*(1) (in-place sorting, no additional array needed)

**Stability**

**Stable**: No (does not preserve the relative order of equal elements)

**In-Place**: Yes (does not require additional storage for sorting)

**Why is Heap Sort Not Stable?**

Heap Sort is not stable because it may change the relative order of equal elements when they are moved around in the heap structure. During the Heapification process, equal elements can be swapped, thus disrupting their original order. These sections summarize the characteristics of Quick Sort and Heap Sort, highlighting their time and space complexities, stability, and reasons for stability or instability.

1. Heap Sort

#include <iostream>

using namespace std;

void heapify(int arr[], int n, int i) {

int largest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

// 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]);

heapify(arr, n, largest);

}

}

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

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

heapify(arr, n, i);

}

// One by one extract elements from heap

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

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

// Call max heapify on the reduced heap

heapify(arr, i, 0);

}

}

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

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

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

}

cout << endl;

}

int main() {

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

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

cout << "Original array: ";

printArray(arr, n);

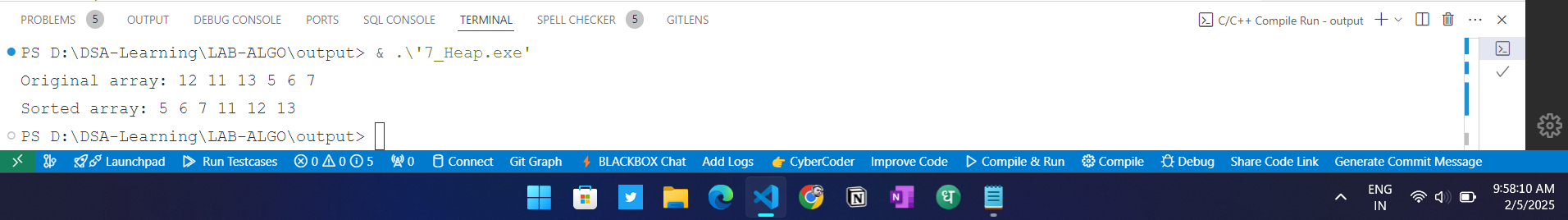
heapSort(arr, n);

cout << "Sorted array: ";

printArray(arr, n);

return 0;

}



Hfbvljfbl

BFS

#include <iostream>

#include <vector>

#include <queue>

using namespace std;

class Graph {

public:

int V; // Number of vertices

vector<vector<int>> adj; // Adjacency list

Graph(int V) {

this->V = V;

adj.resize(V);

}

void addEdge(int u, int v) {

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

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

}

void BFS(int start) {

vector<bool> visited(V, false);

queue<int> q;

visited[start] = true;

q.push(start);

while (!q.empty()) {

int node = q.front();

q.pop();

cout << node << " ";

for (int neighbor : adj[node]) {

if (!visited[neighbor]) {

visited[neighbor] = true;

q.push(neighbor);

}

}

}

}

};

int main() {

Graph g(6); // Creating a graph with 6 vertices (0 to 5)

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 3);

g.addEdge(1, 4);

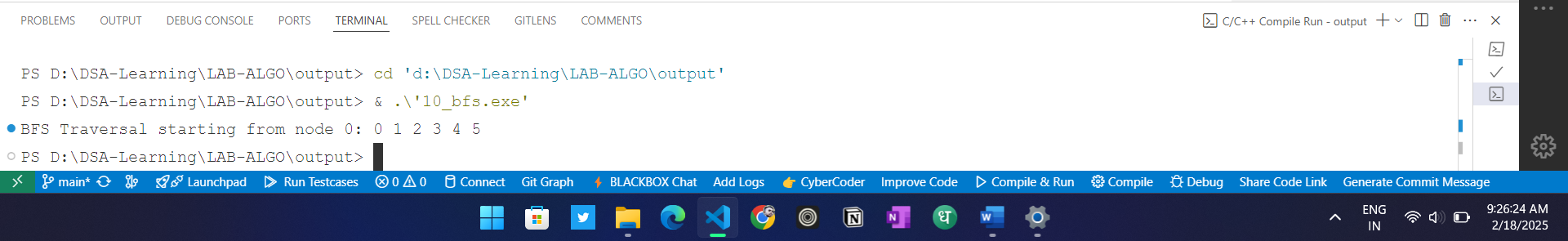
g.addEdge(2, 5);

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

g.BFS(0);

return 0;

}



**Time And Space Complexity**

|  |  |
| --- | --- |
| **Best Case Time** | **O(1)** |
| **Worst Case Time** | **O(V + E) or O(V²) for dense graphs** |
| **Average Case Time** | **O(V + E)** |
| **Space Complexity** | **O(V + E)** |

**Why BFS is Used:**

* BFS is used for level-order traversal, finding the shortest path in unweighted graphs, checking connectivity, and solving problems like puzzles or web crawling. It is simple, efficient, and guarantees optimal results for unweighted graphs.

**Kruskal's algorithm**

**#include<bits/stdc++.h>**

**using namespace std;**

**// Creating shortcut for an integer pair**

**typedef pair<int, int> iPair;**

**// Structure to represent a graph**

**struct Graph**

**{**

**int V, E;**

**vector< pair<int, iPair> > edges;**

**// Constructor**

**Graph(int V, int E)**

**{**

**this->V = V;**

**this->E = E;**

**}**

**// Utility function to add an edge**

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

**{**

**edges.push\_back({w, {u, v}});**

**}**

**// Function to find MST using Kruskal's**

**// MST algorithm**

**int kruskalMST();**

**};**

**// To represent Disjoint Sets**

**struct DisjointSets**

**{**

**int \*parent, \*rnk;**

**int n;**

**// Constructor.**

**DisjointSets(int n)**

**{**

**// Allocate memory**

**this->n = n;**

**parent = new int[n+1];**

**rnk = new int[n+1];**

**// Initially, all vertices are in**

**// different sets and have rank 0.**

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

**{**

**rnk[i] = 0;**

**//every element is parent of itself**

**parent[i] = i;**

**}**

**}**

**// Find the parent of a node 'u'**

**// Path Compression**

**int find(int u)**

**{**

**/\* Make the parent of the nodes in the path**

**from u--> parent[u] point to parent[u] \*/**

**if (u != parent[u])**

**parent[u] = find(parent[u]);**

**return parent[u];**

**}**

**// Union by rank**

**void merge(int x, int y)**

**{**

**x = find(x), y = find(y);**

**/\* Make tree with smaller height**

**a subtree of the other tree \*/**

**if (rnk[x] > rnk[y])**

**parent[y] = x;**

**else // If rnk[x] <= rnk[y]**

**parent[x] = y;**

**if (rnk[x] == rnk[y])**

**rnk[y]++;**

**}**

**};**

**/\* Functions returns weight of the MST\*/**

**int Graph::kruskalMST()**

**{**

**int mst\_wt = 0; // Initialize result**

**// Sort edges in increasing order on basis of cost**

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

**// Create disjoint sets**

**DisjointSets ds(V);**

**// Iterate through all sorted edges**

**vector< pair<int, iPair> >::iterator it;**

**for (it=edges.begin(); it!=edges.end(); it++)**

**{**

**int u = it->second.first;**

**int v = it->second.second;**

**int set\_u = ds.find(u);**

**int set\_v = ds.find(v);**

**// Check if the selected edge is creating**

**// a cycle or not (Cycle is created if u**

**// and v belong to same set)**

**if (set\_u != set\_v)**

**{**

**// Current edge will be in the MST**

**// so print it**

**cout << u << " - " << v << endl;**

**// Update MST weight**

**mst\_wt += it->first;**

**// Merge two sets**

**ds.merge(set\_u, set\_v);**

**}**

**}**

**return mst\_wt;**

**}**

**// Driver program to test above functions**

**int main()**

**{**

**/\* Let us create above shown weighted**

**and undirected graph \*/**

**int V = 9, E = 14;**

**Graph g(V, E);**

**// making above shown graph**

**g.addEdge(0, 1, 4);**

**g.addEdge(0, 7, 8);**

**g.addEdge(1, 2, 8);**

**g.addEdge(1, 7, 11);**

**g.addEdge(2, 3, 7);**

**g.addEdge(2, 8, 2);**

**g.addEdge(2, 5, 4);**

**g.addEdge(3, 4, 9);**

**g.addEdge(3, 5, 14);**

**g.addEdge(4, 5, 10);**

**g.addEdge(5, 6, 2);**

**g.addEdge(6, 7, 1);**

**g.addEdge(6, 8, 6);**

**g.addEdge(7, 8, 7);**

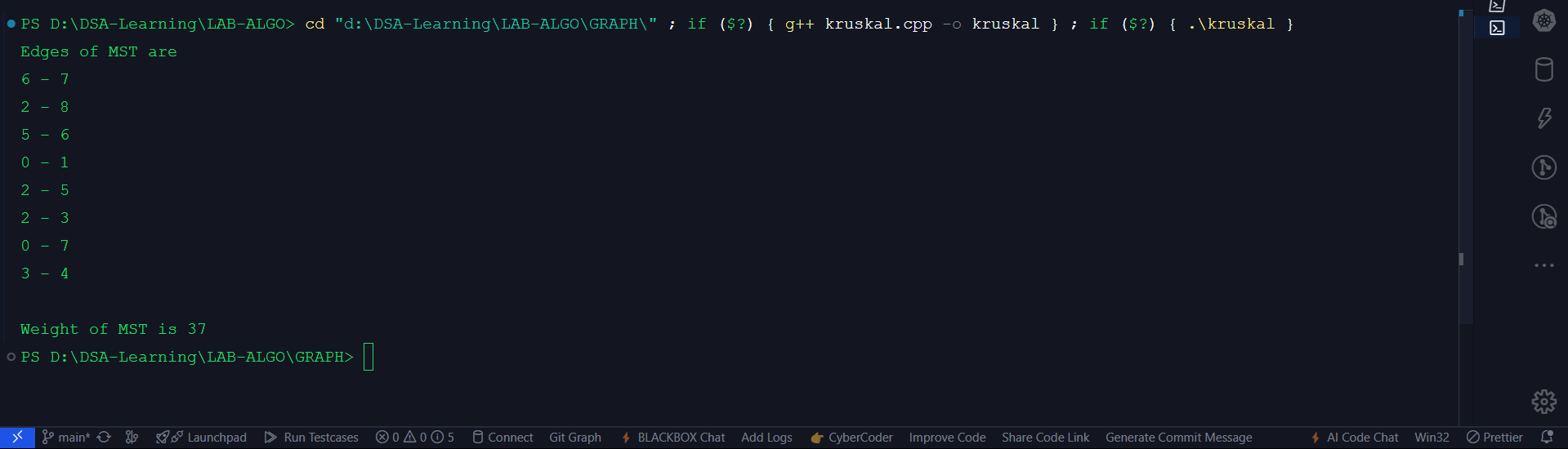
**cout << "Edges of MST are \n";**

**int mst\_wt = g.kruskalMST();**

**cout << "\nWeight of MST is " << mst\_wt;**

**return 0;**

**}**

****

**Prim's Algorithm**

**#include <bits/stdc++.h>**

**using namespace std;**

**// Structure to represent a weighted edge**

**typedef pair<int, int> pii;**

**// Function to find the Minimum Spanning Tree (MST) using Prim's algorithm**

**void primMST(int n, vector<vector<pii>> &adj) {**

**priority\_queue<pii, vector<pii>, greater<pii>> pq; // Min-Heap (weight, vertex)**

**vector<bool> inMST(n, false); // Track nodes included in MST**

**vector<int> key(n, INT\_MAX); // Minimum weight edge to a node**

**vector<int> parent(n, -1); // Store MST structure**

**int start = 0; // Start from node 0**

**key[start] = 0;**

**pq.push({0, start}); // (weight, node)**

**while (!pq.empty()) {**

**int u = pq.top().second; // Pick the minimum weight node**

**pq.pop();**

**if (inMST[u]) continue;**

**inMST[u] = true;**

**// Traverse all adjacent nodes of u**

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

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

**key[v] = weight;**

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

**parent[v] = u;**

**}**

**}**

**}**

**// Print MST edges and total cost**

**int totalCost = 0;**

**cout << "Edges in the MST:\n";**

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

**cout << parent[i] << " - " << i << " (Weight: " << key[i] << ")\n";**

**totalCost += key[i];**

**}**

**cout << "Total Minimum Cost: " << totalCost << endl;**

**}**

**// Driver code**

**int main() {**

**int n, m;**

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

**cin >> n >> m;**

**vector<vector<pii>> adj(n); // Adjacency list**

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

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

**int u, v, w;**

**cin >> u >> v >> w;**

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

**adj[v].push\_back({w, u}); // Undirected graph**

**}**

**primMST(n, adj);**

**return 0;**

**}**