**Graph Adjacent Node**

#include <iostream>

using namespace std;

struct adjNode {

    int val, cost;

    adjNode\* next;

};

struct graphEdge {

    int start\_ver, end\_ver, weight;

};

class DiaGraph{

    adjNode\* getAdjListNode(int value, int weight, adjNode\* head)   {

        adjNode\* newNode = new adjNode;

        newNode->val = value;

        newNode->cost = weight;

        newNode->next = head;

        return newNode;

    }

    int N;

public:

    adjNode \*\*head;

    DiaGraph(graphEdge edges[], int n, int N)  {

        head = new adjNode\*[N]();

        this->N = N;

        for (int i = 0; i < N; ++i)

            head[i] = nullptr;

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

            int start\_ver = edges[i].start\_ver;

            int end\_ver = edges[i].end\_ver;

            int weight = edges[i].weight;

            adjNode\* newNode = getAdjListNode(end\_ver, weight, head[start\_ver]);

            head[start\_ver] = newNode;

             }

    }

     ~DiaGraph() {

    for (int i = 0; i < N; i++)

        delete[] head[i];

        delete[] head;

     }

};

void display\_AdjList(adjNode\* ptr, int i)

{

    while (ptr != nullptr) {

        cout << "(" << i << ", " << ptr->val

            << ", " << ptr->cost << ") ";

        ptr = ptr->next;

    }

    cout << endl;

}

int main()

{

    graphEdge edges[] = {

        {0,1,2},{0,2,4},{1,4,3},{2,3,2},{3,1,4},{4,3,3}

    };

    int N = 6;

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

    DiaGraph diagraph(edges, n, N);

    cout<<"Graph adjacency list "<<endl<<"(start\_vertex, end\_vertex, weight):"<<endl;

    for (int i = 0; i < N; i++)

    {

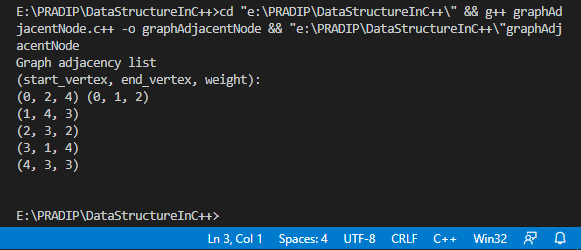
        display\_AdjList(diagraph.head[i], i);

    }

    return 0;

}

**Output:**



**Unweighted Graph Shortest path**

#include <bits/stdc++.h>

using namespace std;

void add\_edge(vector<int> adj[], int src, int dest)

{

    adj[src].push\_back(dest);

    adj[dest].push\_back(src);

}

bool BFS(vector<int> adj[], int src, int dest, int v,

         int pred[], int dist[])

{

    list<int> queue;

    bool visited[v];

    for (int i = 0; i < v; i++) {

        visited[i] = false;

        dist[i] = INT\_MAX;

        pred[i] = -1;

    }

    visited[src] = true;

    dist[src] = 0;

    queue.push\_back(src);

    while (!queue.empty()) {

        int u = queue.front();

        queue.pop\_front();

        for (int i = 0; i < adj[u].size(); i++) {

            if (visited[adj[u][i]] == false) {

                visited[adj[u][i]] = true;

                dist[adj[u][i]] = dist[u] + 1;

                pred[adj[u][i]] = u;

                queue.push\_back(adj[u][i]);

                if (adj[u][i] == dest)

                    return true;

            }

        }

    }

    return false;

}

void printShortestDistance(vector<int> adj[], int s,

                           int dest, int v)

{

    int pred[v], dist[v];

    if (BFS(adj, s, dest, v, pred, dist) == false) {

        cout << "Given source and destination"

             << " are not connected";

        return;

    }

    vector<int> path;

    int crawl = dest;

    path.push\_back(crawl);

    while (pred[crawl] != -1) {

        path.push\_back(pred[crawl]);

        crawl = pred[crawl];

    }

    cout << "Shortest path length is : "

         << dist[dest];

    cout << "\nPath is::\n";

    for (int i = path.size() - 1; i >= 0; i--)

        cout << path[i] << " ";

}

int main()

{

    int v = 8;

    vector<int> adj[v];

    add\_edge(adj, 0, 1);

    add\_edge(adj, 0, 3);

    add\_edge(adj, 1, 2);

    add\_edge(adj, 3, 4);

    add\_edge(adj, 3, 7);

    add\_edge(adj, 4, 5);

    add\_edge(adj, 4, 6);

    add\_edge(adj, 4, 7);

    add\_edge(adj, 5, 6);

    add\_edge(adj, 6, 7);

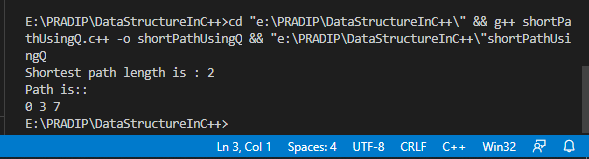
    int source = 0, dest = 7;

    printShortestDistance(adj, source, dest, v);

    return 0;

}

**Output:**



**Unweighted Graph Shortest path using Queue**

#include <iostream>

using namespace std;

void add\_edge(vector<int> adj[], int src, int dest)

{

    adj[src].push\_back(dest);

    adj[dest].push\_back(src);

}

bool BFS(vector<int> adj[], int src, int dest, int v,

         int pred[], int dist[])

{

    list<int> queue;

    bool visited[v];

    for (int i = 0; i < v; i++) {

        visited[i] = false;

        dist[i] = INT\_MAX;

        pred[i] = -1;

    }

    visited[src] = true;

    dist[src] = 0;

    queue.push\_back(src);

    while (!queue.empty()) {

        int u = queue.front();

        queue.pop\_front();

        for (int i = 0; i < adj[u].size(); i++) {

            if (visited[adj[u][i]] == false) {

                visited[adj[u][i]] = true;

                dist[adj[u][i]] = dist[u] + 1;

                pred[adj[u][i]] = u;

                queue.push\_back(adj[u][i]);

                if (adj[u][i] == dest)

                    return true;

            }

        }

    }

    return false;

}

void printShortestDistance(vector<int> adj[], int s,

                           int dest, int v)

{

    int pred[v], dist[v];

    if (BFS(adj, s, dest, v, pred, dist) == false) {

        cout << "Given source and destination"

             << " are not connected";

        return;

    }

    vector<int> path;

    int crawl = dest;

    path.push\_back(crawl);

    while (pred[crawl] != -1) {

        path.push\_back(pred[crawl]);

        crawl = pred[crawl];

    }

    cout << "Shortest path length is : "

         << dist[dest];

    cout << "\nPath is::\n";

    for (int i = path.size() - 1; i >= 0; i--)

        cout << path[i] << " ";

}

int main()

{

    int v = 8;

    vector<int> adj[v];

    add\_edge(adj, 0, 1);

    add\_edge(adj, 0, 3);

    add\_edge(adj, 1, 2);

    add\_edge(adj, 3, 4);

    add\_edge(adj, 3, 7);

    add\_edge(adj, 4, 5);

    add\_edge(adj, 4, 6);

    add\_edge(adj, 4, 7);

    add\_edge(adj, 5, 6);

    add\_edge(adj, 6, 7);

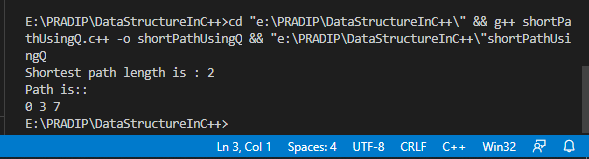
    int source = 0, dest = 7;

    printShortestDistance(adj, source, dest, v);

    return 0;

}

**Output:**



**Dijkstra's Weighted Graph Shortest Path**

#include <limits.h>

#include <stdio.h>

#define V 9

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

{

    int min = INT\_MAX, min\_index;

    for (int v = 0; v < V; v++)

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

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

    return min\_index;

}

void printSolution(int dist[])

{

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

    for (int i = 0; i < V; i++)

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

}

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

{

    int dist[V];

    bool sptSet[V];

    for (int i = 0; i < V; i++)

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

    dist[src] = 0;

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

        int u = minDistance(dist, sptSet);

        sptSet[u] = true;

        for (int v = 0; v < V; v++)

            if (!sptSet[v] && graph[u][v] && dist[u] != INT\_MAX

                && dist[u] + graph[u][v] < dist[v])

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

    }

    printSolution(dist);

}

int main()

{

    int graph[V][V] = { { 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 },

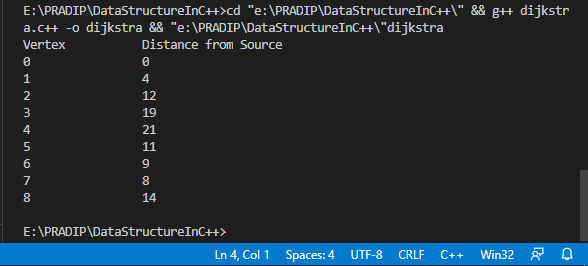
                        { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };

    dijkstra(graph, 0);

    return 0;

}

**Output:**



**Priority Queue using Min Heap**

#include <iostream>

using namespace std;

int main ()

{

    priority\_queue <int> pq;

    pq.push(5);

    pq.push(1);

    pq.push(10);

    pq.push(30);

    pq.push(20);

    while (pq.empty() == false)

    {

        cout << pq.top() << " ";

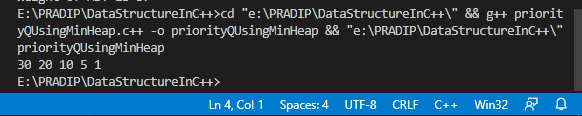
        pq.pop();

    }

    return 0;

}

Output:



**Max Heap**

#include <iostream>

using namespace std;

int main ()

{

    priority\_queue <int> pq;

    pq.push(5);

    pq.push(1);

    pq.push(10);

    pq.push(30);

    pq.push(20);

    while (pq.empty() == false)

    {

        cout << pq.top() << " ";

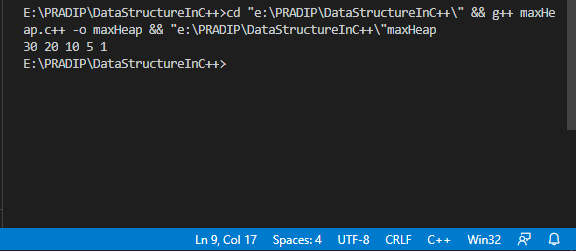
        pq.pop();

    }

    return 0;

}

**Output:**



**Heap sort**

#include <iostream>

using namespace std;

void heapify(int arr[], int n, int i)

{

    int largest = i;

    int l = 2 \* i + 1;

    int r = 2 \* i + 2;

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

        largest = l;

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

        largest = r;

    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);

    for (int i = n - 1; i > 0; i--) {

        swap(arr[0], arr[i]);

        heapify(arr, i, 0);

    }

}

void printArray(int arr[], int n)

{

    for (int i = 0; i < n; ++i)

        cout << arr[i] << " ";

    cout << "\n";

}

int main()

{

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

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

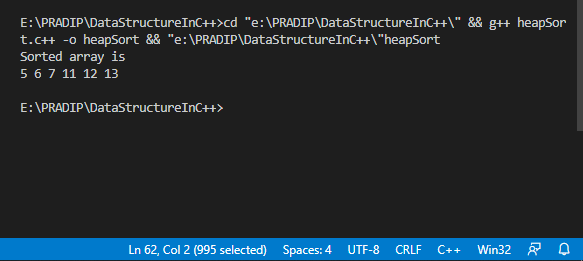
    heapSort(arr, n);

    cout << "Sorted array is \n";

    printArray(arr, n);

}

**Output:**



**Quick Sort**

#include <iostream>

using namespace std;

void swap(int\* a, int\* b)

{

    int t = \*a;

    \*a = \*b;

    \*b = t;

}

int partition (int arr[], int low, int high)

{

    int pivot = arr[high];

    int i = (low - 1);

    for (int j = low; j <= high - 1; j++)

    {

        if (arr[j] < pivot)

        {

            i++;

            swap(&arr[i], &arr[j]);

        }

    }

    swap(&arr[i + 1], &arr[high]);

    return (i + 1);

}

void quickSort(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);

    }

}

void printArray(int arr[], int size)

{

    int i;

    for (i = 0; i < size; i++)

        cout << arr[i] << " ";

    cout << endl;

}

int main()

{

    int arr[] = {10, 7, 8, 9, 1, 5};

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

    quickSort(arr, 0, n - 1);

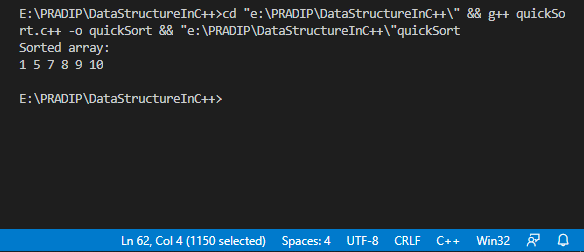
    cout << "Sorted array: \n";

    printArray(arr, n);

    return 0;

}

**Output:**



**Radix Sort**

#include <iostream>

using namespace std;

int getMax(int arr[], int n)

{

    int mx = arr[0];

    for (int i = 1; i < n; i++)

        if (arr[i] > mx)

            mx = arr[i];

    return mx;

}

void countSort(int arr[], int n, int exp)

{

    int output[n];

    int i, count[10] = { 0 };

    for (i = 0; i < n; i++)

        count[(arr[i] / exp) % 10]++;

    for (i = 1; i < 10; i++)

        count[i] += count[i - 1];

    for (i = n - 1; i >= 0; i--) {

        output[count[(arr[i] / exp) % 10] - 1] = arr[i];

        count[(arr[i] / exp) % 10]--;

    }

    for (i = 0; i < n; i++)

        arr[i] = output[i];

}

void radixsort(int arr[], int n)

{

    int m = getMax(arr, n);

    for (int exp = 1; m / exp > 0; exp \*= 10)

        countSort(arr, n, exp);

}

void print(int arr[], int n)

{

    for (int i = 0; i < n; i++)

        cout << arr[i] << " ";

}

int main()

{

    int arr[] = { 170, 45, 75, 90, 802, 24, 2, 66 };

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

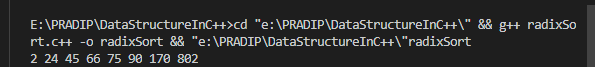
      radixsort(arr, n);

    print(arr, n);

    return 0;

}

**Output:**



**Shell sort**

#include  <iostream>

using namespace std;

int shellSort(int arr[], int n)

{

    for (int gap = n/2; gap > 0; gap /= 2)

    {

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

        {

            int temp = arr[i];

            int j;

            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)

                arr[j] = arr[j - gap];

            arr[j] = temp;

        }

    }

    return 0;

}

void printArray(int arr[], int n)

{

    for (int i=0; i<n; i++)

        cout << arr[i] << " ";

}

int main()

{

    int arr[] = {12, 34, 54, 2, 3}, i;

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

    cout << "Array before sorting: \n";

    printArray(arr, n);

    shellSort(arr, n);

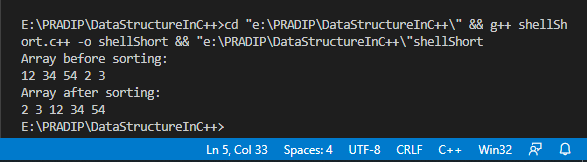
    cout << "\nArray after sorting: \n";

    printArray(arr, n);

    return 0;

}

**Output:**



**Merge sort**

#include <iostream>

using namespace std;

void merge(int arr[], int l, int m, int r)

{

    int n1 = m - l + 1;

    int n2 = r - m;

    int L[n1], R[n2];

    for (int i = 0; i < n1; i++)

        L[i] = arr[l + i];

    for (int j = 0; j < n2; j++)

        R[j] = arr[m + 1 + j];

    int i = 0;

    int j = 0;

    int k = l;

    while (i < n1 && j < n2) {

        if (L[i] <= R[j]) {

            arr[k] = L[i];

            i++;

        }

        else {

            arr[k] = R[j];

            j++;

        }

        k++;

    }

    while (i < n1) {

        arr[k] = L[i];

        i++;

        k++;

    }

    while (j < n2) {

        arr[k] = R[j];

        j++;

        k++;

    }

}

void mergeSort(int arr[],int l,int r){

    if(l>=r){

        return;

    }

    int m = (l+r-1)/2;

    mergeSort(arr,l,m);

    mergeSort(arr,m+1,r);

    merge(arr,l,m,r);

}

void printArray(int A[], int size)

{

    for (int i = 0; i < size; i++)

        cout << A[i] << " ";

}

int main()

{

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

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

    cout << "Given array is \n";

    printArray(arr, arr\_size);

    mergeSort(arr, 0, arr\_size - 1);

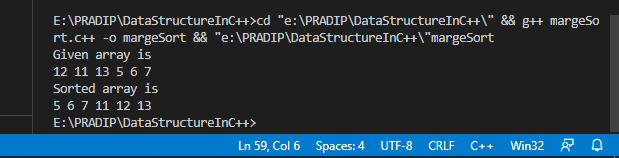
    cout << "\nSorted array is \n";

    printArray(arr, arr\_size);

    return 0;

}

**Output:**



**BFS**

#include<iostream>

#include <list>

using namespace std;

class Graph

{

    int V;

    list<int> \*adj;

public:

    Graph(int V);

    void addEdge(int v, int w);

    void BFS(int s);

};

Graph::Graph(int V)

{

    this->V = V;

    adj = new list<int>[V];

}

void Graph::addEdge(int v, int w)

{

    adj[v].push\_back(w);

}

void Graph::BFS(int s)

{

    bool \*visited = new bool[V];

    for(int i = 0; i < V; i++)

        visited[i] = false;

    list<int> queue;

    visited[s] = true;

    queue.push\_back(s);

    list<int>::iterator i;

    while(!queue.empty())

    {

        s = queue.front();

        cout << s << " ";

        queue.pop\_front();

        for (i = adj[s].begin(); i != adj[s].end(); ++i)

        {

            if (!visited[\*i])

            {

                visited[\*i] = true;

                queue.push\_back(\*i);

            }

        }

    }

}

int main()

{

    Graph g(4);

    g.addEdge(0, 1);

    g.addEdge(0, 2);

    g.addEdge(1, 2);

    g.addEdge(2, 0);

    g.addEdge(2, 3);

    g.addEdge(3, 3);

    cout << "Following is Breadth First Traversal "

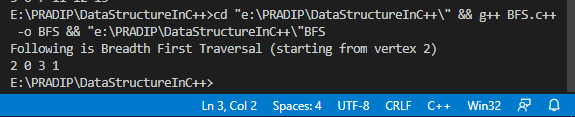
         << "(starting from vertex 2) \n";

    g.BFS(2);

    return 0;

}

**Output:**



**DFS**

#include <iostream>

using namespace std;

class Graph {

    int V;

    list<int>\* adj;

    void DFSUtil(int v, bool visited[]);

public:

    Graph(int V);

    void addEdge(int v, int w);

    void DFS(int v);

};

Graph::Graph(int V)

{

    this->V = V;

    adj = new list<int>[V];

}

void Graph::addEdge(int v, int w)

{

    adj[v].push\_back(w);

}

void Graph::DFSUtil(int v, bool visited[])

{

    visited[v] = true;

    cout << v << " ";

    list<int>::iterator i;

    for (i = adj[v].begin(); i != adj[v].end(); ++i)

        if (!visited[\*i])

            DFSUtil(\*i, visited);

}

void Graph::DFS(int v)

{

    bool\* visited = new bool[V];

    for (int i = 0; i < V; i++)

        visited[i] = false;

    DFSUtil(v, visited);

}

int main()

{

    Graph g(4);

    g.addEdge(0, 1);

    g.addEdge(0, 2);

    g.addEdge(1, 2);

    g.addEdge(2, 0);

    g.addEdge(2, 3);

    g.addEdge(3, 3);

    cout << "Following is Depth First Traversal"

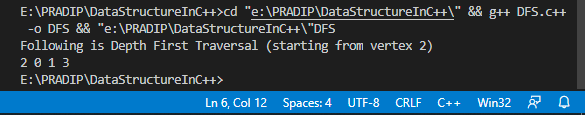
            " (starting from vertex 2) \n";

    g.DFS(2);

    return 0;

}

**Output:**



**Kruskals**

#include <iostream>

using namespace std;

typedef  pair<int, int> iPair;

struct Graph

{

    int V, E;

    vector< pair<int, iPair> > edges;

    Graph(int V, int E)

    {

        this->V = V;

        this->E = E;

    }

    void addEdge(int u, int v, int w)

    {

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

    }

    int kruskalMST();

};

struct DisjointSets

{

    int \*parent, \*rnk;

    int n;

    DisjointSets(int n)

    {

        this->n = n;

        parent = new int[n+1];

        rnk = new int[n+1];

        for (int i = 0; i <= n; i++)

        {

            rnk[i] = 0;

            parent[i] = i;

        }

    }

    int find(int u)

    {

        if (u != parent[u])

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

        return parent[u];

    }

    void merge(int x, int y)

    {

        x = find(x), y = find(y);

        if (rnk[x] > rnk[y])

            parent[y] = x;

        else

            parent[x] = y;

        if (rnk[x] == rnk[y])

            rnk[y]++;

    }

};

int Graph::kruskalMST()

{

    int mst\_wt = 0;

    sort(edges.begin(), edges.end());

    DisjointSets ds(V);

    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);

        if (set\_u != set\_v)

        {

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

            mst\_wt += it->first;

            ds.merge(set\_u, set\_v);

        }

    }

    return mst\_wt;

}

int main()

{

    int V = 9, E = 14;

    Graph g(V, E);

    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;

}

**Output:**

