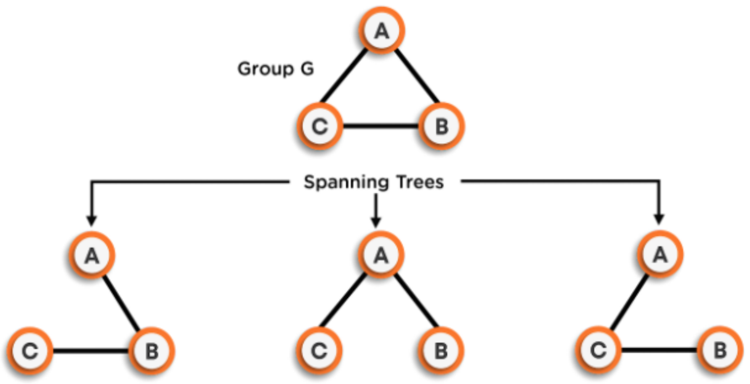
Graphs II

Minimum Spanning Tree

Introduction

To understand the concept of a minimum spanning tree, we must understand what a **Spanning Tree is**.



Spanning Tree

A **graph** is a pair *G* = (*V*, *E*), where *V* is a set whose elements are called ***vertices***, and *E* is a set of two sets of vertices, whose elements are called ***edges***. The spanning tree of the above graph can be represented as **G' = (V', E').** In this case, **V' = V**, states that the number of vertices in the spanning tree would be the same as the number of vertices present in the original graph. But the number of edges will be different and should be the subset of the number of edges present in the original graph.

The edges of minimum spanning tree can be written as –

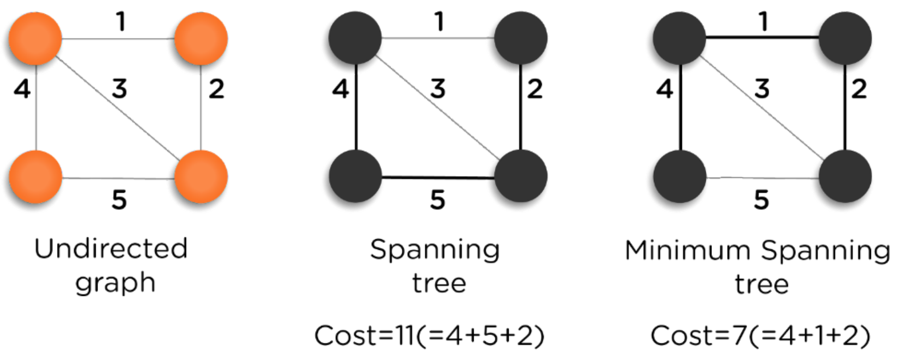
E' ϵ E.

Conditions for Spanning Tree:

- The number of vertices in the spanning tree would be the same as that of the number of vertices in the original graph **(V' = V).**
- The number of edges in the spanning tree should be one less than number of edges of original graph **(E' = E -1)**
- The spanning tree should not contain any cycle.
- The spanning tree should not be disconnected.

Minimum Spanning Tree

The minimum spanning tree is a spanning tree whose sum of the edges is minimum. It is a subset of the edges of a connected, edge-weighted directed or undirected graph that connects the vertices together, without any cycle and with minimum possible weight



Minimum Spanning Tree

Properties:

- **Possible Multiplicity:** If there are **n vertices** in the graph, then each spanning tree has **n-1 edges**

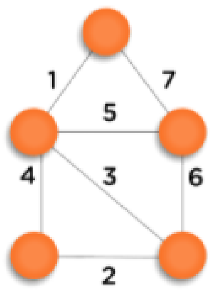
- **Uniqueness:** If each edge has a distinct weight then there will be only one, unique minimum spanning tree.
- **Minimum cost Subgraph:** If the weights are positive, then the minimum spanning tree is in fact a minimum cost subgraph connecting all vertices.

Applications:

- Minimum spanning tree is used to design a network
- Circuit Designing
- Topological observability in power systems

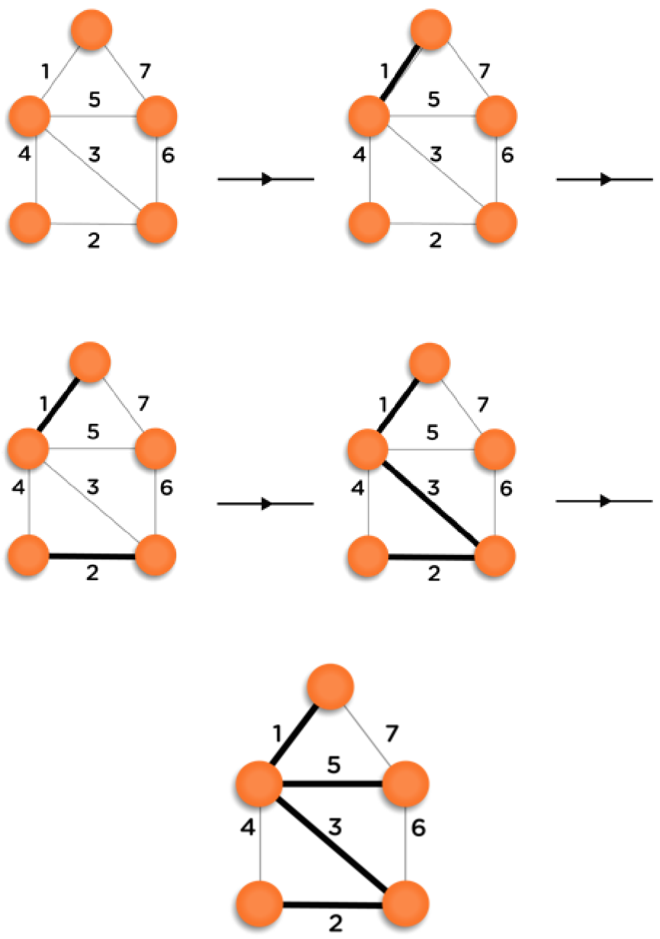
Kruskal's Algorithm

Implementation of Kruskal's Algorithm:



In Kruskal's algorithm, we start from edges with the lowest weight and keep adding the edges until the goal is reached. The steps to implement Kruskal's algorithm are listed as follows -

- First, sort all the edges from **low weight to high.**
- Now, take the edge with the lowest weight and add it to the spanning tree. If the edge to be added creates a cycle, then reject the edge.
- Continue to add the edges until we reach all vertices, and **a minimum spanning tree is created.**



Code:

Till now, we have studied the logic behind Kruskal's algorithm for **finding MST.** Now, let's discuss how to implement it in code.

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

using namespace std;

struct Edge{
      int a;
      int b;
      int w;
};

Edge edgeList[1000001];
```

```
int parent[1000001];

bool comp(Edge a , Edge b){
      return a.w < b.w;
}

int find(int a){
      if(parent[a] == a) return a;

      parent[a] = find(parent[a]);
}

void merge(int a , int b){
      a = find(a);
      b = find(b);

      parent[b] = a;
}

int main(){
      int n , m;
      cin>>n>>m;

      for(int i=1;i<=n;i++) parent[i] = i;

      for(int i=1;i<=m;i++){
            cin>>edgeList[i].a>>edgeList[i].b>>edgeList[i].w;
      }

      sort(edgeList + 1 , edgeList + m + 1 , comp);

      int mstCost = 0;

      for(int i=1;i<=m;i++){
            Edge e = edgeList[i];

            if(find(e.a) != find(e.b)){
                  mstCost += e.w;
                  merge(e.a , e.b);
            }
      }
      cout<<"MST Cost = "<<mstCost;
}
```

Cycle Detection:

While inserting a new edge in the **MST,** we have to check if introducing that edge makes the **MST cyclic or not.** If not, then we can include that edge, otherwise not.

Now, let’s figure out a way to detect the cycle in a graph. The following are the possible cases:

- By including an edge between the **nodes A and B**, if both the nodes A and B are not present in the graph, then it is safe to include that edge as including it, will not bring a cycle to the graph.
- Out of **two vertices,** if any of them has not been visited (or not present in the MST), then that vertex can also be included in the **MST**.
- If both the vertices are already present in the graph, they can introduce a cycle in the MST. It means we can’t use this method to detect the presence of the cycle.

Let’s think of a better approach. We have already solved the **hasPath** question in the previous module, which returns true if there is a path present between two **vertices v1 and v2,** otherwise false.

Now, before adding an edge to the MST, we will check if a path between two vertices of that edge already exists in the MST or not. If not, then it is safe to add that edge to the MST.

As discussed in previous lectures, the time complexity of the **hasPath** function is **O(E+V),** where E is the number of edges in the graph and, **V** is the number of vertices. So, for **(n-1) edges**, this function will run **(n-1)** times, leading to bad time complexity, as in the worst case, **$E = V^2$.**

Time Complexity of Kruskal's Algorithm:

In our code, we have the following three steps: (Here, the total number of vertices is n, and the total number of edges is E)

- Take input in the array of **size E.**
- Sort the input array on the basis of edge-weight. This step has the time complexity of **O (E log(E)).**
- Pick **(n-1) edges** and put them in MST one-by-one. Also, before adding the edge to the MST, we checked for cycle detection for each edge. For cycle detection, in the worst-case time complexity of E edges will be **O(E.n)**, as discussed earlier.

Hence, the total time complexity of Kruskal's algorithm becomes **O(E log(E) + n.E).**

Applications of Kruskal's Algorithm:

- Kruskal's algorithm can be used to layout electrical wiring among cities.
- It can be used to lay down LAN connections.

Prim's Algorithm

Prim's Algorithm is used to find the minimum spanning tree and uses a greedy approach. It is similar with respect to the shortest path first algorithms.

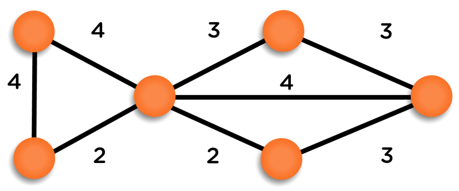
In Prim's algorithms, the nodes are being treated as a single tree and keep on adding new nodes to the spanning tree from the given graph.

This algorithm is used to find MST for a given undirected-weighted graph (which can also be achieved using Kruskal's Algorithm).

It falls under a class of algorithms called greedy algorithms that find the local optimum in the hopes of finding a global optimum. We start from one vertex and keep adding edges with the lowest weight until we reach our goal.

Implementation of Prim's Algorithm:

- Initialise the minimum spanning tree with a **vertex chosen at random.**
- Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree

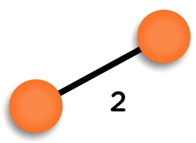


Step: 1

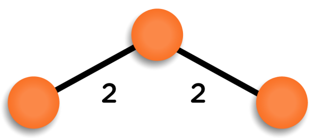
Start with a weighted graph

- **Keep repeating step 2** until we get a minimum spanning tree. Consider the following example for a better understanding.

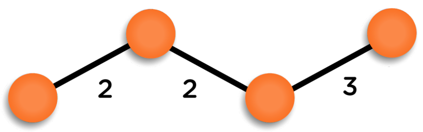
Step: 2
Choose a vertex



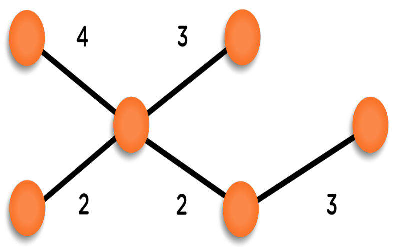
Step: 3
Choose the shortest edge from this vertex add it



Step: 4
Choose the nearest vertex not yet in the solution



Step: 5
Choose the nearest edge not yet in the solution,
if there are multiple choices, choose one at random



Step: 6

Repeat until you have a spanning tree

- We are considering the starting vertex to be **0 with a parent equal to -1**, and weight is equal to 0 (The weight of the edge from vertex 0 to vertex 0 itself).

- The parent of all other vertices is assumed to be **NIL**, and the weight will be equal to infinity, which means that the vertex has not been visited yet.
- We will mark the **vertex 0** as visited and rest as unvisited. If we add any vertex to the **MST,** then that vertex will be shifted from the unvisited section to the visited section.
- Now, we will update the weights of direct neighbours of **vertex 0** with the edge weights as these are smaller than infinity. We will also update the parent of these vertices and assign them 0 as we reached these vertices from **vertex 0.**
- This way, we will keep updating the weights and parents, according to the edge, which has the minimum weight connected to the respective vertex.

Code:

```
#include<bits/stdc++.h>
using namespace std;

const int N = 1e5;
const int INF = 1e9;

int  min_w[N+1];
bool selected[N+1];

vector<pair<int,int> > adj[N+1];

void primsMST(int n , int src){

      int mstCost = 0;
      set<pair<int,int> > edgeSet;

      edgeSet.insert({0 , src});

      for(int i=1;i<=n;i++){
            int node = edgeSet.begin()->second;
            int w = edgeSet.begin()->first;
            edgeSet.erase(edgeSet.begin());

            mstCost += w;
            selected[node] = true;

            for(pair<int,int> e : adj[node]){
                  if(selected[e.first] == false && e.second <
min_w[e.first]){
                        edgeSet.erase({min_w[e.first] , e.first});
                        min_w[e.first] = e.second;
                        edgeSet.insert({min_w[e.first] , e.first});
                  }
            }

      }

      cout<<"MST Cost = "<<mstCost;
}
int main(){
      int n , m;
      int a , b , w;

      cin>>n>>m;

      for(int i=1;i<=n;i++) min_w[i] = INF , selected[i] = false;

      for(int i=0;i<m;i++){
            cin>>a>>b>>w;
            adj[a].push_back({b , w});
            adj[b].push_back({a , w});
      }

      primsMST(n , 1);
}
```

Time Complexity of Prim's Algorithm:

Here, **n** is the number of **vertices**, and **E** is the number of **edges.**

- The time complexity for finding the minimum weighted vertex is **O (n)** for each iteration. So for **(n-1) edges,** it becomes **O (n^2).**
- Similarly, for exploring the neighbour vertices, the time taken is **O (n^2).**

It means the time complexity of Prim's algorithm is **O (n^2).** We can improve this in the following ways:

- For exploring neighbours, we are required to visit each and every vertex because of the adjacency matrix. We can improve this by using an adjacency list instead of a matrix.

- Now, the second important thing is the time taken to find the minimum weight vertex, which is also taking a time of **O (n^2).** Here, out of the available list, we are trying to figure out the one with minimum weight. This can be optimally achieved using a **priority queue** where the priority will be taken as weights of the vertices. This will take **O (log (n))** time complexity to remove a vertex from the priority queue.

These optimizations can lead us to the time complexity of **O ((n+E)log(n))**, which is much better than the earlier one.

Applications of Prim's Algorithm

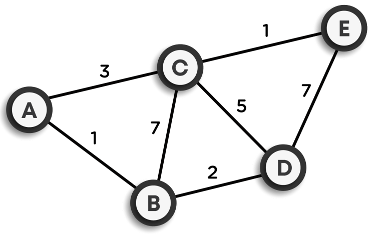
- Network for roads and Rail tracks connecting all the cities.
- Irrigation Channels and placing microwave towers

Dijkstra's Algorithm

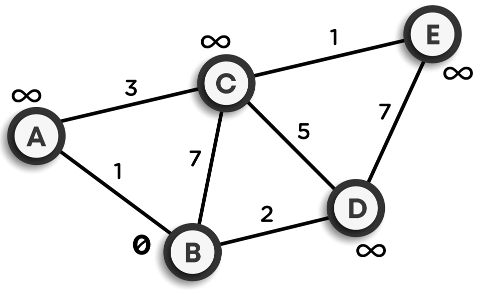
Dijkstra's algorithm is used to find the shortest distance between any two vertices in a **weighted non-cyclic graph**. Here, we will be using a slight modification of the algorithm according to which we will be figuring out the minimum distance of all the vertices from the particular **source vertex.**

Implementation of Dijkstra's Algorithm:

1. We want to calculate the shortest path between the **source vertex C** and all other vertices in the following graph.

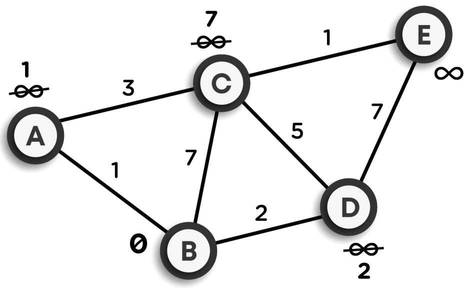


2. While executing the algorithm, we will mark every node with its **minimum distance** to the selected node, which is C in our case. Obviously, for node C itself, this distance will be **0,** and for the rest of the nodes, we will assume that the distance is infinity, which also denotes that these vertices have not been visited till now.

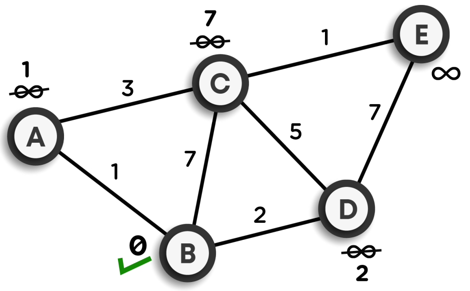


3. Now, we will check for the neighbours of the current node, which in our case is **A, B, and D.** Now, we will add the minimum cost of the current node to the weight of the edge connecting the current node and the particular neighbour node. For example, for node B, it's weight will

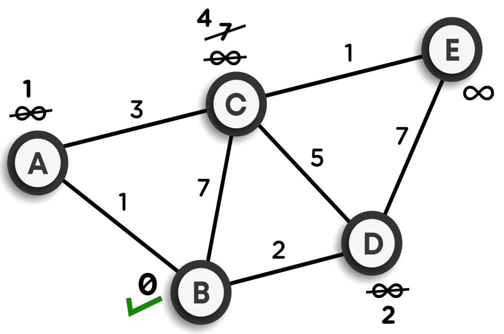
become minimum **(infinity, 0+7) = 7.** This same process is repeated for other neighbour nodes.

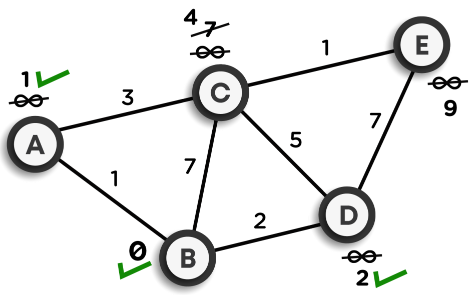


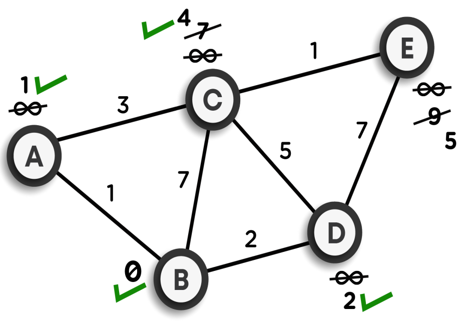
4. Now, as we have updated the distance of all the neighbour nodes of the **current node,** we will mark the current node as visited.



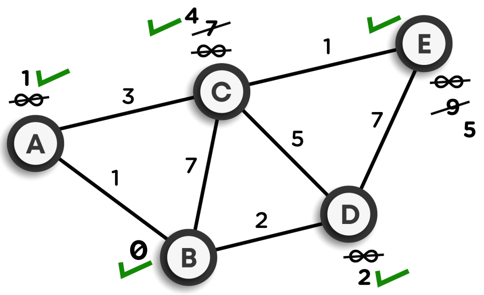
5. After this, we will be selecting the **minimum weighted node** among the remaining vertices. In this case, it is node A. Take this node as the current node.
6. Now, we will repeat the above steps for the rest of the vertices. The pictorial representation of the same is shown below:







7. Finally, we will get the graph as follows:



The distances finally marked at each node are minimum from **node C.**

Code:

```
#include<bits/stdc++.h>
using namespace std;

const int N = 200000;
const int INF = 1e9;
vector<pair<int,int> > adj[N+1];
int dist[N+1];

void dijkstra(int n , int src){

      priority_queue<pair<int,int> ,vector<pair<int,int> > ,
greater<pair<int,int>>> pq;
      dist[src] = 0;
      pq.push({0 , src});

      while(!pq.empty()){
            int node = pq.top().second;
            int d = pq.top().first;
            pq.pop();

            if(dist[node] < d) continue;

            for(pair<int,int> e : adj[node])
            if(dist[e.first] > dist[node] + e.second){
                  dist[e.first] = dist[node] + e.second;
                  pq.push({dist[e.first] , e.first});
            }
      }

}

int main(){

      freopen("input.txt" , "r" , stdin);
      freopen("output.txt" , "w" , stdout);

      int n , m , src;
      int a , b , w;
```

```
	cin>>n>>m>>src;

	for(int i=1;i<=n;i++) dist[i] = INF;

	while(m--){
		cin>>a>>b>>w;
		adj[a].push_back({b , w});
		adj[b].push_back({a , w});
	}

	dijkstra(n , src);

	for(int i=1;i<=n;i++) cout<<"Node "<<i<<" has dist = 
"<<dist[i]<<endl;

	return 0;
}
```

Time Complexity of Dijkstra's algorithm:

The **time complexity** of Dijkstra's Algorithm is **O (n^2)**. But it is reduced to **O (V + E log V)** when solved with a min-priority queue.

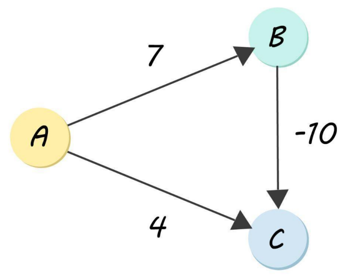
Applications of Dijkstra's algorithm:

- Mapping Services in Google Maps
- IP routing to find the shortest path first.

Bellman Ford Algorithm

Bellman Ford Algorithm is used to find the shortest distance between any two vertices in a **weighted non-cyclic graph.** It overcomes the shortcomings of Dijkstra's algorithm, which fails for a graph with **negative weight.**

Implementation of Bellman Ford Algorithm:



If we apply Dijkstra's algorithm in the above example it gives us the path **A → C.**

However in reality the path **A → B → C** is the correct answer with the distance of **7 - 10 = -3.**

So **Bellman Ford** suggests that in order to take into account the negative edges, we must calculate the shortest path between any two vertices through different numbers of edges so as to find out the correct answer amongst them.

In the above example, if we find the shortest path with at most one edge then our distance is **4**. But if we find the shortest path once more and this time we could take at most **2 edges** to travel from our source vertex to the destination vertex then our total distance is **-3.**

So amongst **4** and **-3, -3** is shorter and hence our final path will be **A → B → C.**

Algorithm:

1) This step initialises distances from the source to all vertices as infinite and distance to the source itself as 0. Create an array **dist[] of size |V|** with all values as infinite except **dist[src]** where src is source vertex.

2) This step calculates shortest distances. Do following **|V|-1** times where **|V|** is the number of vertices in a given graph.

a) Do following for each **edge u-v**

If **dist[v] > dist[u] + weight of edge uv, then update dist[v]**

dist[v] = dist[u] + weight of edge uv

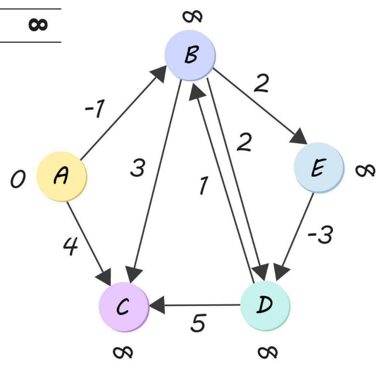
3) This step reports if there is a negative weight cycle in the graph. Do following for each **edge u-v**

If **dist[v] > dist[u] + weight of edge uv**, then "Graph contains negative weight cycle"

The idea of step 3 is, step 2 guarantees the shortest distances if the graph doesn't contain a negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle.

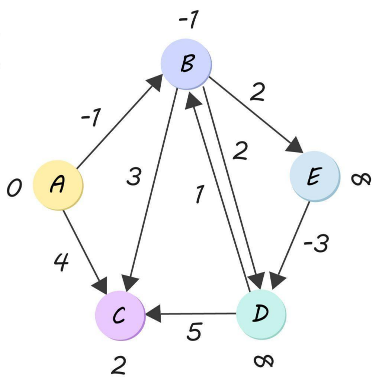
Example:

A	B	C	D	E
0	∞	∞	∞	∞



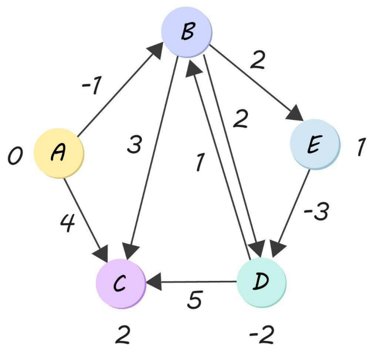
Let all edges are processed in the following order: **(B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D).**

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞



The first iteration guarantees to give all shortest paths which are at most 1 edge long.

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1



The second iteration guarantees to give all shortest paths which are at **most 2 edges** long. The algorithm processes all edges 2 more times. The distances are minimised after the second iteration, so **third and fourth iterations** don't update the distances.

Code:

```
#include<bits/stdc++.h>
using namespace std;

const int N = 1e5;
const int INF = 1e9;

struct Edge{
      int a;
      int b;
      int w;
};
int dist[N+1];
Edge edgeList[N+1];

int main(){
      int n , m , src;

      cin>>n>>m>>src;

      for(int i=1;i<=n;i++) dist[i] = INF;
      dist[src] = 0;

      for(int i=1;i<=m;i++)
cin>>edgeList[i].a>>edgeList[i].b>>edgeList[i].w;

      //go for n-1 phases
      for(int i=1;i<n;i++){
            bool flag = false;

            for(int j=1;j<=m;j++){
                  Edge e = edgeList[j];

                  if(dist[e.b] > dist[e.a] + e.w){
                        dist[e.b] = dist[e.a] + e.w;
                        flag = true;
                  }
            }

            if(!flag) break;
      }

      for(int i=1;i<=n;i++) cout<<"Node "<<i<<" has distance :
"<<dist[i]<<endl;
}
```

Time Complexity of Bellman Ford algorithm:

In a complete graph with edges between every pair of vertices, and assuming you found the shortest path in the first few iterations or repetitions but still go on with edge relaxation, you would have to relax **|E| * (|E| - 1) / 2 edges, (|V| - 1)** number of times.

The worst-case scenario in the case of a complete graph, the time complexity is as follows:

$$O(|V|^2) = O(E\ V).\ O(|V|) = O\ (V^3)$$

$$\begin{tikzcd} \text{Time Complexity} \arrow[d] & = O(E\cdot V) \\ \text{for complete Graph} & \\ & E = \frac{V(V-1)}{2} \\ & E = O(V^2) \arrow[uuu, bend right] \end{tikzcd}$$

$$T\ (h) = O(V^2 \cdot V) = O(V^3)$$

Applications for Bellman Ford Algorithm:

- Used for distance-routing protocol helping in routing the data packets on the network
- Used in internet gateway routing protocol
- Used in routing information protocol

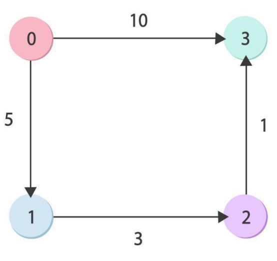
Floyd Warshall's Algorithm

Floyd Warshall's algorithm is used to find shortest distances between every pair of vertices in a given edge **weighted directed Graph.**

We store the distances in a matrix of **N X N** in which if the value at any **cell [i][j]** is infinity, then that means there exists no path between **vertex i and vertex j.**

Example:

The idea is to one by one pick all vertices and update all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices **{0, 1, 2, .. k-1}** as intermediate vertices. For every pair **(i, j)** of the source and destination vertices respectively, there are two possible cases.



Graph

	0	1	2	3
0	0	5	∞	10
1	∞	0	3	∞
2	∞	∞	0	1
3	∞	∞	∞	0

Shortest Distance matrix

1. **k** is not an intermediate vertex in the shortest path from **i to j.** We keep the value of **dist[i][j]** as it is.
2. **k** is an intermediate vertex in shortest path from **i to j**. We update the value of **dist[i][j] as dist[i][k] + dist[k][j] if dist[i][j] > dist[i][k] + dist[k][j]**

Code:

```
#include<bits/stdc++.h>
using namespace std;

const int N = 1e2;
const int INF = 1e9;
int dp[N+1][N+1];
int main(){
      int n , m;
      int a , b , w;

      cin>>n>>m;
      for(int i=1;i<=n;i++){
            for(int j=1;j<=n;j++){
                  if(i == j) dp[i][i] = 0;
                  else     dp[i][j] = INF;
            }
      }

      for(int i=1;i<=m;i++){
            cin>>a>>b>>w;
            if(a != b)
            dp[a][b] = dp[b][a] = w;
      }

      //floyd-warshall
      for(int i=1;i<=n;i++){
            for(int a=1;a<=n;a++){
                  for(int b=1;b<=n;b++)
                        dp[a][b] = min(dp[a][b] , dp[a][i] +
dp[i][b]);
            }
      }

      for(int a=1;a<=n;a++){
            for(int b=1;b<=n;b++){
                  cout<<"distance between "<<a<<" and "<<b<<" =
"<<dp[a][b]<<endl;
            }
      }
}
```

Time Complexity of Floyd Warshall algorithm:

The **time complexity** of Floyd Warshall Algorithm is **O(n^3)**

The Floyd Warshall Algorithm consists of three loops over all the nodes. The innermost loop consists of only constant complexity operations. Hence, the asymptotic complexity of the Floyd Warshall algorithm is **$O(n^3)$.** Here, **n is the number of nodes in the given graph.**

Applications for Floyd Warshall Algorithm:

- To find the transitive closure of directed graphs
- To find the inversion of real matrices