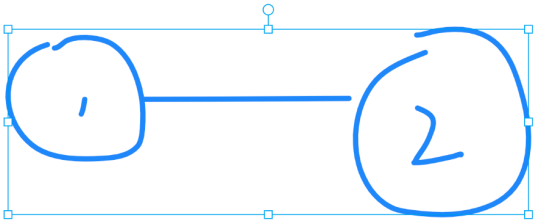
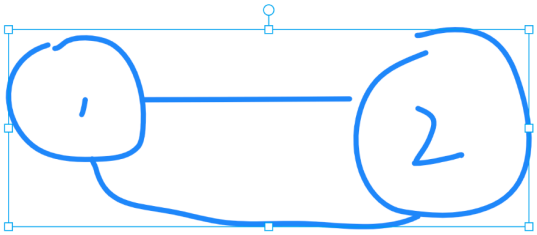
How to Make a Adjacency List:-

```
vector<vector<int>>& src_dst = [[0,1],[1,2],[2,3]]
Int n= src_dst.size();
vector<vector<int>> adj_lsit(total_unique_nodes);
for(int i=0;i<n;i++)
{
  adj_lsit[prerequisites[i][1]].push_back(prerequisites[i][0]); //For Directed use it twice for un-direct

 }
```

CYCLE DETECTION In7 Undirected Cycle

DFS:-





Adjacency List :-

1-2
2-1

1-2,2
2-1,1

We will keep a visited note and the parent , so that to know if it's already visited and if it's not a parent then there is a cycle, this will work only in non-directed graph

```
int dfs(vector<vector<int> >v,vector<int> &vis,int node,int parent)
{
    vis[node]=1;    //Mark the node as visited
    for(int x=0;x<v[node].size();x++) // Traverse through all the child node
    {
        int y=v[node][x];
        if(vis[y]==0)// if it's  not visited
        {
            if(dfs(v,vis,y,node)==1) //then goto that node
            return 1;
        }
        else if(y!=parent) //if its visited and it's not parent then return cycle
        return 1;
    }
```

```
    return 0;}
```

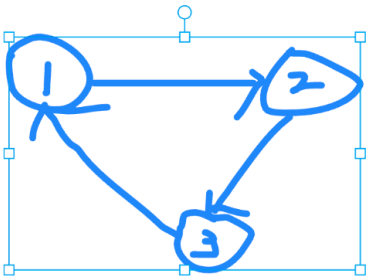
BFS:-

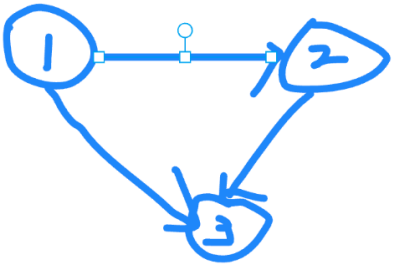
Use the queue and pair so that the data structure we can store the parent node as well.

```
   queue<pair<int,int> > qp;
   vint vis(n,0);
   int flag=0;
   for(int i=0;i<n;i++)
   {
       if(!vis[i])
       {
           qp.push(make_pair(i,-1));
           while(!qp.empty())
           {
               pair<int,int> node=qp.front(); qp.pop();
               vis[node.first]=1;
               cout<<vis[node.first]<<" "<<adj_lis[node.first].size()<<"
"<<node.first<<endl;
               for(int j=0;j<adj_lis[node.first].size();j++)
               {
                   int child=adj_lis[node.first][j];
                   cout<<"Child = "<<child<<endl;
                   if(!vis[child])
                   {
                           qp.push(make_pair(child,node.first));
                   }
                   else{
                       if(child!=node.second)
                       {
                       cout<<"Contain Cycle"<<endl;
                       flag=1;
                       break;

                       }
                   }

               }
           }

       }
   }
   if(flag!=1)
```

```
{
    cout<<"Doesn;t contain CYcle"<<endl;
}
```

Cycle in the directed Graph:-





Here the first approach will be to create a visited array and retraces back as there cannot be a concept of the parent here.
We also need to keep in mind that we need to retrace the visited array as well so that we can have the proper cycle detection technique:-
For case 1:-
1-2
2-3
3-1
Vis[1]=1
Vis[2]=1
Vis[3]=1
And when we check again for vis[1] which is already true so there is a cycle.
For case 2:-
1-2,3
2-3
vis[1]=1
vis[2]=1
vis[3=1
But as there no edge her we will retrace back to node one and again set the visited array to zero :-
vis[1]=1
vis[2]=0
vis[3]=0
Now when we will visit the node 3 we will set it as 1 and then when we will check for other nodes in the node-3 there will be no edge and there will no cycle.

DFS implementation as below:-

```
int dfs(vector<vector<int> >v,vector<int> &vis,int node)
{
    if(vis[node]==1) return 1;
    vis[node]=1;
    for(int x=0;x<v[node].size();x++)
    {
        int y=v[node][x];
        cout<<y<<endl;
        if(vis[y]==0)
        {
            if(dfs(v,vis,y,node)==1)
            return 1;
        }

        else
        return 1;

    }
    vis[node]=0;
    return 0;

}
```

But the problem her is the implementation will take the time of O(V*E)
So to minimize the complexity we can use another local visited array so that we can keep track of the cycle locally.
We create a dfsvis vector to keep the track of local changes.

```
int dfs(vector<vector<int> >v,vector<int> &vis,vector<int> &dfsvis,int node)
{
    if(vis[node]==1) return 1;
    vis[node]=1;
    dfsvis[node]=1;
    for(int x=0;x<v[node].size();x++)
    {
        int y=v[node][x];
        cout<<y<<endl;
```

```
        if(vis[y]==0)
        {
            if(dfs(v,vis,dfsvis,y)==1)
            return 1;
        }

        else if(dfsvis[x]==1)
        return 1;

    }
    dfsvis[node]=0;
    return 0;

}
```

Now we can implement the same for the BFS as well.

But the problem is we are using an extra Space of dfsvis.
How can we reduce it ????????
By using three coloring technique

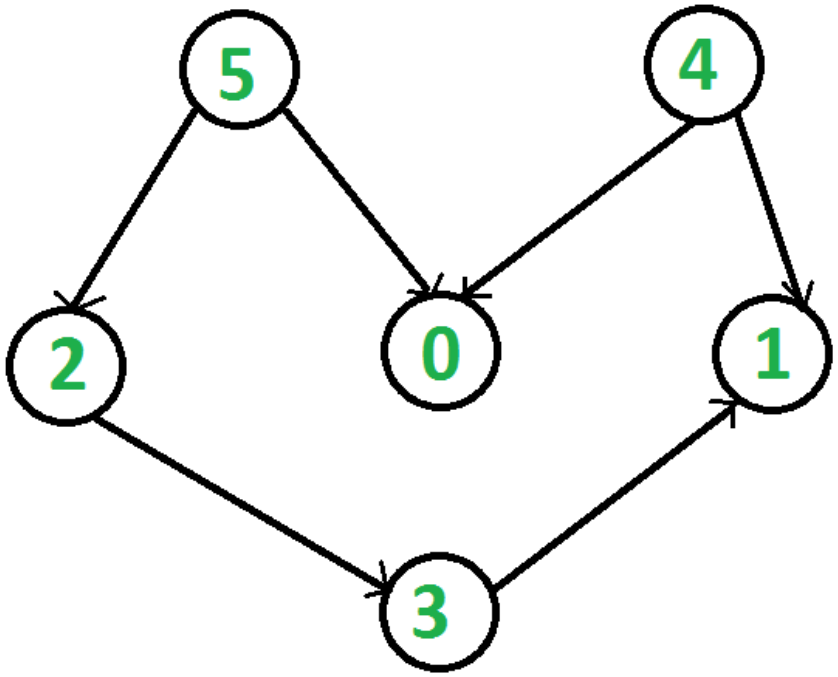
When to use the Gray - Red - White MEthod to find the Cycle:-
We need this to find a cycle in the directed graph .

```
int dfs(vector<vector<int>>& graph,int node,vector<int>& vis)
{
    if(vis[node]!=0) return vis[node] ;
    vis[node]=2;
    for(auto& child:graph[node])
    {
            if(dfs(graph,child,vis)==2)
                return 2;
    }
    return vis[node]=1;
}
vector<int> eventualSafeNodes(vector<vector<int>>& graph) {
    vector<int> ans;
    int n=graph.size();
    vector<int> vis(n,0);
    for(int i=0;i<n;i++)
    {

        if(dfs(graph,i,vis)==1)
```

```
            ans.push_back(i);
        }    }
```

Now we will understand topological sort:-

What does topological sort say ?
Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge u v, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.



Eg:-

Here the topological sort for this will be :-5,4,2,3,1,0.
We can implement this using the bfs and dfs.

Let's start with DFS:-

We just uses the the normal DFS and if the every child nodes are being visited we will put that in the stack.

```
int dfs(vector<vector<int> >v,vector<int> &vis,stack<int> &st,int node)
{
    if(vis[node]==1) return 1;
    vis[node]=1;
    for(int x=0;x<v[node].size();x++)
```

```
    {
        int y=v[node][x];
        if(vis[y]==0)
        {
            dfs(v,vis,st,y);
        }


    }
    st.push(node);
    return 0;

}
```

Now we will get the position but the problem here is that even if the graph contains cycle the topological sort then also the topological sort will be generated.
So we will map the output stack and will check the priority using the adjacency list , if we find the priority higher then it's child element then there is a cycle for sure.

```
map<int,int> mp;
int index=0;
while(!st.empty())
{
   int temp=st.top();
   mp[temp]=index;
   index++;
   st.pop();
}
for(int i=0;i<n;i++)
{
   for(auto &ed: adj_lis[i])
   {
      if(mp[i]>mp[ed])
         return false;
   }
}

return true;
```

Now we will check with the BFS

We cannot go to the deepest node to check whether there is any further child node or not so we will be using the indegree vector to track the the node which has no child as if there is no indegree that mean that those node will be at the top.

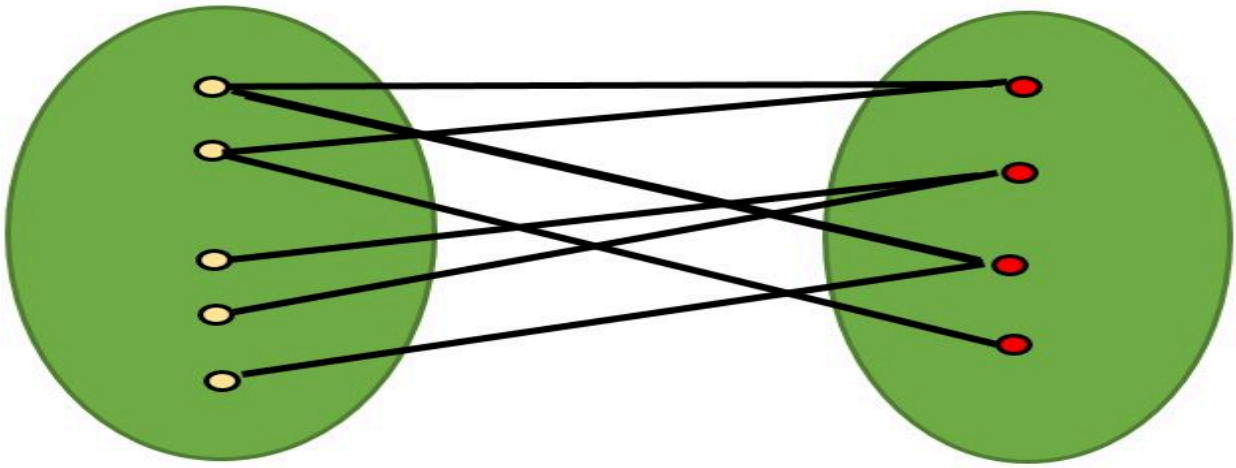
Now if there is no indegree value as zero then it means there must be a cyle .
Now to be acyclic the graph must have total indegree of value n in the stack/queue .

```
vector<int> indegree(numCourses);
      for(int i=0;i<n;i++)
      {
          adj_lsit[prerequisites[i][1]].push_back(prerequisites[i][0]);
          indegree[prerequisites[i][0]]++;
      }

      queue<int> q;
      for(int i=0;i<numCourses;i++)
      {
          if(indegree[i]==0)
              q.push(i);
      }
      int count=0;
      while(!q.empty())
      {
          int top=q.front();
          q.pop();
          for(auto &it:adj_lsit[top])
          {
              indegree[it]--;
              if(indegree[it]==0)
                  q.push(it);

          }
          count++;
      }

      return count==numCourses;
```

So with the topological sort also we can verify the whether it contains a cycle or not.

Bipartite Graph

A Bipartite Graph is a graph whose vertices can be divided into two independent sets, U and V such that every edge (u, v) either connects a vertex from U to V or a vertex from V to U. In other words, for every edge (u, v), either u belongs to U and v to V, or u belongs to V and v to U. We can also say that there is no edge that connects vertices of same set.



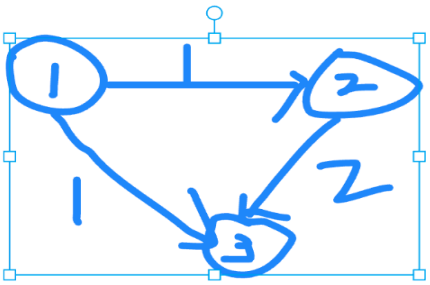
We can find if the graph is a bipartite graph or not using the Two-Color technique.

```
//Graph coloring: 0->not visited...1->visited...2->visited & processed
   bool detectCycle_util(vector<vector<int>>& adj,vector<int>& visited,int v)
   {
       if(visited[v]==1)
           return true;
       if(visited[v]==2)
           return false;

       visited[v]=1;    //Mark current as visited
       for(int i=0;i<adj[v].size();++i)
           if(detectCycle_util(adj,visited,adj[v][i]))
               return true;

       visited[v]=2;    //Mark current node as processed
       return false;
   }
}  }
```

Paths in the Graph:-

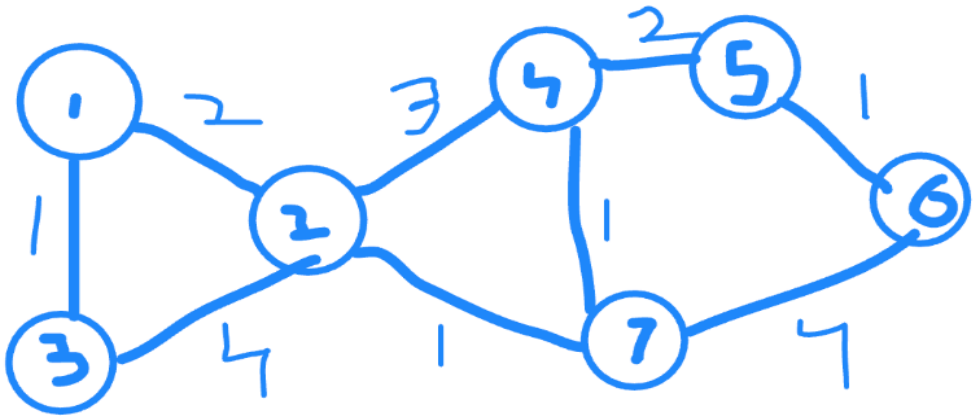
We need to focus on the graph that has shortest path from one edge or so:-

Let’s consider a graph that has unit weight.




```
int solve(int n, int N,vvint adj_lis)
{

vint dist(n,INT_MAX);
q.push(0);
dist[0]=0;
while(!q.empty())
{
   int parent=q.front();
   q.pop();
   for(int i=0;i<adj_lis[parent].size();i++)
   {
       int child=adj_lis[parent][i];
       if(dist[child]>dist[parent]+1)
       {
           dist[child]=dist[parent]+1;
           q.push(child);
       }
   }
}



}
```

It will work for unit vector graphs, but not for every graph around as the distance between some nodes may vary.
And we many need to change the initial parent distance as well hence all it's child node will also affected accordingly.
For eg:- node 4, when it's coming from node 2, it will have distance of 5, hence its child node 5 will have distance of 7.
But when the node 4 will coming from edge 7 instead of 2, then it's distance from source will be 4 and it's child node's distance will be decreased to 6(node 5).So whenever a parent node is changed all it's child node distance need to be updated as well.

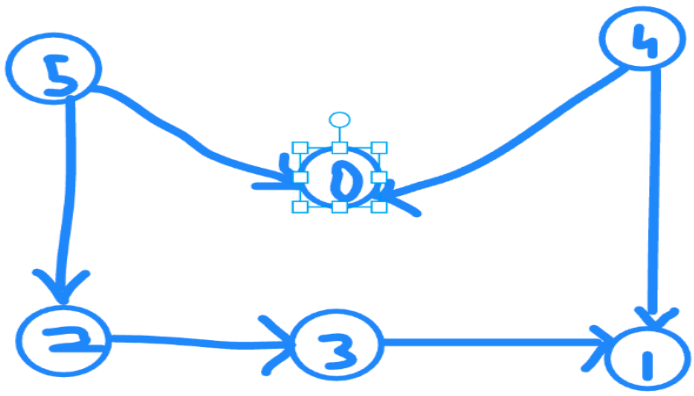



```
vector<vector<pair<int,int>>> adj_list(n+1);
      for(auto &i:times)
      {
          adj_list[i[0]].push_back(make_pair(i[1],i[2]));
      }
      vector<int> dist(n+1,INT_MAX);
      dist[k]=0;
      vector<int> vis(n+1,0);
      queue<int> q;
      q.push(k);
      while(!q.empty())
      {
          int node=q.front();
          q.pop();
          for(auto &ed:adj_list[node])
          {
              if(dist[ed.first]>dist[node]+ed.second)
              {
                  vis[ed.first]=0;
              dist[ed.first]=dist[node]+ed.second;
                   q.push(ed.first);
              }
           }
      }
```

But here the time complexity will be increased as every time we may need to change the distance and its complexity can reach to O(VXE) . So it might not be that much effective.

Next thing we will try to find the shortest path in a Directed Acylic Garph.

We can find the Topological Sort of the graph and then we can find the distance according to the topological sort




```
vector<int> dist(n+1,INT_MAX-1000);
        dist[5]=0;
        vector<int> vis(n+1,0);
        stack<int> st;
        for(int i=0;i<n;i++)
        {
            if(!vis[i])
            {
                dfs(adj_lis,vis,st,i);
            }
        }
        vint temp_ans;
        while (!st.empty())
        {
            temp_ans.push_back(st.top());
            st.pop();
        }


        queue<int> q;
        for(int i=0;i<temp_ans.size();i++)
        {
            q.push(temp_ans[i]);
        }
        //q.push(k);
        while(!q.empty())
        {
            int node=q.front();
```

```
            cout<<node<<endl;
            q.pop();
            for(int i=0;i<adj_lis[node].size();i++)
            {
                //cout<<ed.first<<" ";
                pair<int,int> ed;
                ed.first=adj_lis[node][i];

                ed.second=1;
                //cout<<" dist_node = "<<dist[node]<<" ";
                cout<<ed.first<<" "<<ed.second<<endl;
                if(dist[ed.first]>dist[node]+ed.second)
                {
                dist[ed.first]=dist[node]+ed.second;
                cout<<"ed.first= "<<ed.first<<" "<<dist[ed.first]<<endl;
                }
                //cout<<"ed.first= "<<ed.first<<" "<<dist[ed.first]<<endl;

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


            cout<<dist[i]<<" ";
        }
```

The output will be

Dijkstra's Algorithm

In this Algorithm we will use an Priority Queue to store the distance and edge pair. The priority queue will store the Vertices and Distance pair in the ascending order of distance.**shortest paths** from the source to all vertices in the given graph.

We will first create an Vector of pair to store the Vertices and Distance.

```
vector<vector<pair<int,int> > >adj_lis(n+1);
   for(int i=0;i<N;i++)
   {
       adj_lis[v[i][0]].push_back(make_pair(v[i][1],v[i][2]));
       //adj_lis[v[i][1]].push_back(v[i][0]);
   }
```

Now we get the adj_list with the distance we need a Priority Queue to keep the Vertices-Distance pair in the sorted order so that we can revisit those pair accordingly.

```
priority_queue< pair<int,int>,vector< pair<int,int> >,compare> pq;
```

We will need a distance vector to track the minimum distance traversed till now.

```
struct compare
{
 bool operator()(const pair<int,int> p1,const pair<int,int> p2)
   {return p2.second>p1.second;}
};
int solve(int n, int N,vector<vector<pair<int,int> > >adj_lis)
{
   int k=5;
  vector<int> dist(n+1,INT_MAX);
   dist[k]=0;
   priority_queue< pair<int,int>,vector< pair<int,int> >,compare> pq;
   pq.push(make_pair(k,0));
   while(!pq.empty())
   {
       pair<int,int> temp=pq.top();
       int node=temp.first;
       pq.pop();
       for(auto &ed:adj_lis[node])
       {       int child_node=ed.first;
               int weight=ed.second;
               cout<<child_node<<" "<<weight<<endl;
               if(dist[child_node]>dist[node]+weight)
               {
                    dist[child_node]=dist[node]+weight;
                   pq.push(make_pair(child_node,dist[child_node]));
               }
```

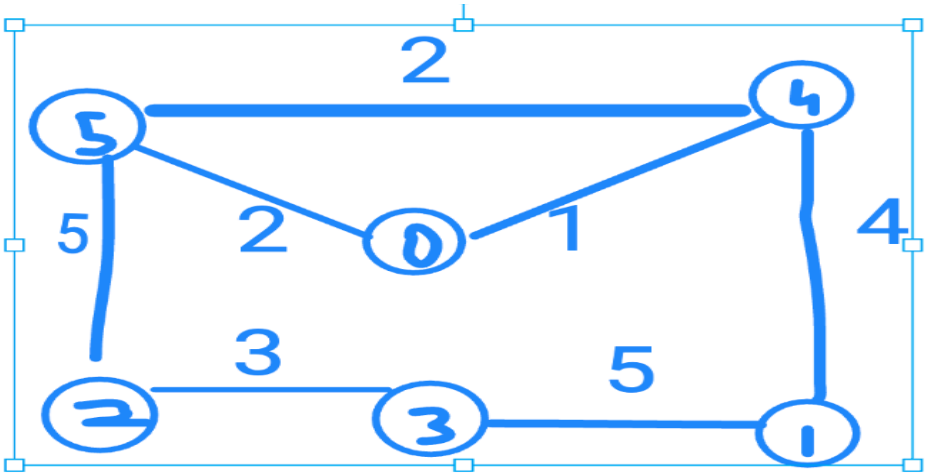
```
        }
    }

    int ans=INT_MIN;
    cout<<"dist f 5 = "<<dist[k]<<endl;
           for(int i=0;i<n;i++)
        {
            if(i==k) continue;
            cout<<"i = "<<i<<endl;
            cout<<dist[i]<<" ";
            ans=max(ans,dist[i]);
            cout<<"ans= "<<ans<<endl;

        }
        cout<<endl;
        cout<<ans;
return 0;
}
```

Minimum Spanning Tree

It has N node and N-1 Edges. We can reach any Node to Any Node in a Tree. For a graph there can be multiple Spanning tree.The Tree which we can make with Minimum Distance then that tree is MST.

There are two algorithms for it one is Prim's and other is Krushkal.

In the **Prim's Algorithm** we do the following : -



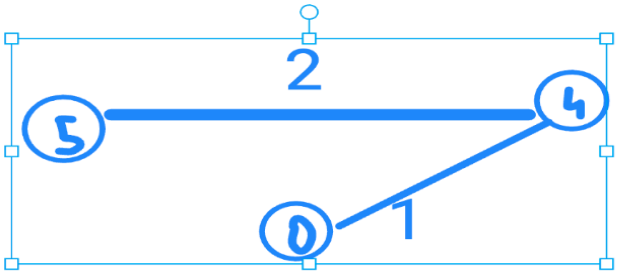
Assuming the 5 as the Source.

There are 3 nodes connected to it that is 0,2 and 4 that their respective distance is 2,5,2.

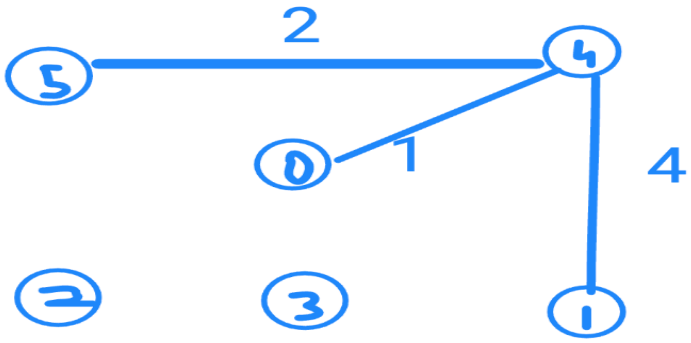
We will select the least that can ne node 2 or 4. Let's say we are going with node 4

Then we have some thing like

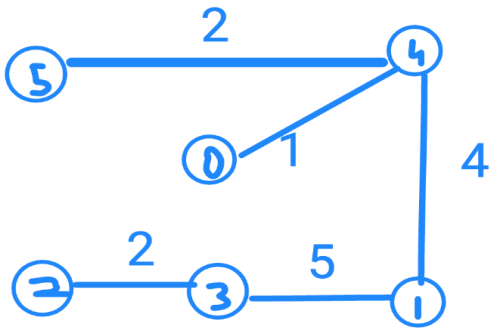
5 - – - - - - - 4.
Now we have Node 5 and 4 as one component now we have to figure out least distance from all the combined component.For node 5 we have node 0 and 2 and for node 4 we have node 0 and node 1. The least of them is node 1 with weight 1.

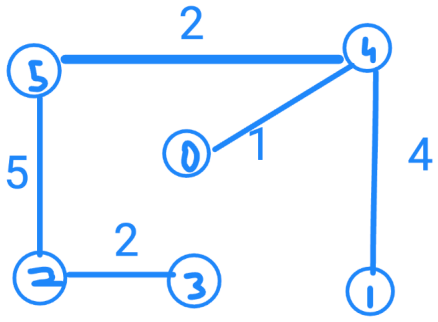


No we have this component and we have to find all the adjacent nodes attached to it and find the node with the least distance.
For node 5 we have node 0 and 2 and for node 4 we have node 1 and for the node 0 we have node 5.
ANd the least of them is node 0 to node 5 but if we take that then it will form a cycle so we cannot take that node into account and will go for next smaller .That is node 4 to node 1.



Now we have node 5 to node 2 and node 1 to node 3 as rest are covered and we see that both have similar distance so we can go for either than we have to go for 3.
So the two possible result's will be .





So we will write the code for it
Here we are creating vector or for checking visited,parent,key.
We are creating Vis vector to node to check if the node is visited or not to avoid the cycle.
Parent is to store the parent of the node.
Key is to check the least distance coming to that node.
First we put visited[source]=0
Then we will check the least value of key vector and will take that as the source node and will update the child's node parent value .

```
int solve(int n, int N,vector<vector<pair<int,int> > >adj_lis)
{
   int k=5;

  vector<int> key(n+1,INT_MAX);
  vector<int> vis(n+1,0);
  vector<int> parent(n+1,-1);
   key[k]=0;
   priority_queue<pair<int,int>,vector<pair<int,int> >,compare >pq;
   pq.push(make_pair(k,0));
   for(int i =0;i<n;i++)
   {
       int mini=INT_MAX,u;
       for(int v=0;v<n;v++)
       {   if(vis[v]==false && key[v]<mini) // we want to find the minimum so that we
can find the u perfectly
           {
           mini=min(mini,key[v]);
           u=v;
           }
       }
       // u=pq.top().first; //will use the priority queue for the better optimisation
       // pq.pop();
       cout<<"u = "<<u<<endl;
       vis[u]=1;
       for(auto &edj:adj_lis[u])
       {
           int child=edj.first;
           int dist=edj.second;
           if(vis[child]==0 && dist<key[child])
           {
               parent[child]=u;
               // pq.push(make_pair(child,key[child]));
               key[child]=dist;
```

```
            }

        }
    }

    for(int i=0;i<n;i++)
    {
        if(i==k) continue;
        cout<<parent[i]<<" ----- "<<i<<" "<<key[i]<<endl;
    }
    cout<<endl;

return 0;
}
```

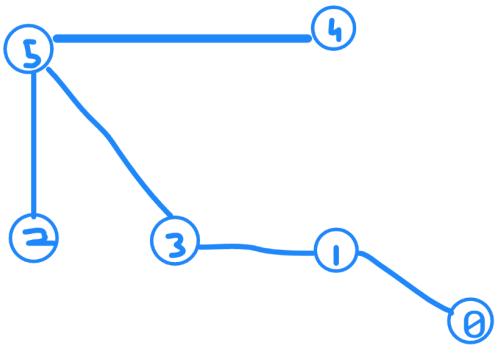
Union Find

Union Find —- find_parent() +union()
Used to detect the cycle.
Parent- 1 2 3 4 5 6 7
Union (1,2) - 1 —----->2 (Parent of 1 is 1 and Parent of 2 is 1)
Union (2,3) - 3←—-1—---->2 (Parent of 3 is 2 and 2's parent is 1 hence parent of 3 is 1)
Union (4,5) - 4—----->5
Union (6,7) - 6—----->7
Union (5,6) - 4—-5—-6—7 (All will have parent as 4)

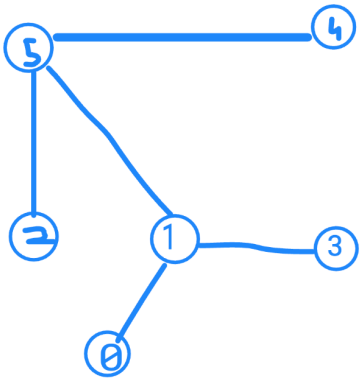
When we don't use the path compression we will get a tree with more depth so using the path compression we will get less rooted tree.

```
int find_parent(int x,vint& parent)
{
    if(parent[x]==x)
    return x;
    return parent[x]=find_parent(parent[x],parent);
}
int solve(int x, int N,vector<vector<int> >adj_lis)
{
int n=adj_lis.size();
vector<int> parent(x,0);
vector<int> rank(x,0);
for(int i=0;i<x;i++) parent[i]=i;
for(int i=0;i<n;i++)
```

```
{
   int u=find_parent(adj_lis[i][0],parent);
   int v=find_parent(adj_lis[i][1],parent);
   if(u!=v)
   {
       if(rank[u]>rank[v]) parent[v]=u;
       else if(rank[v]>rank[u]) parent[u]=v;
       else
       {
           parent[v]=u;
           rank[u]++;
       }
   }
   else
   {
       cout<<"There is a cycle"<<endl;
       break;
   }
}

for(int i=0;i<x;i++) cout<<i<<" ----- "<<parent[i]<<endl;



return 0;
}
```



Without Graph Compression
Depth -3



With the Graph Compression
Depth-2

Krushkal's Algorithm

It's a greedy algorithm that helps to find the minimum spanning tree.
We sort the adj_list on the basis of the edge_weight.
Then we pick the first set and put in a union and add it's weight
If we find the cyle that means the vertice is in the component so we ignore that vertices.
Continue till all the edge to get the MST.

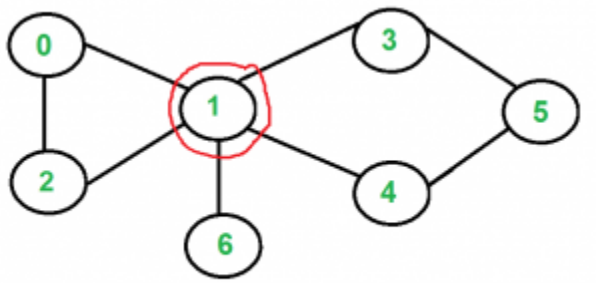
```
int find_parent(int x,vint& parent)
{
   if(parent[x]==x)
   return x;
   return parent[x]=find_parent(parent[x],parent);

}
int solve(int n, int N,vector<vector<int> > adj_lis)
{
//cout<<n;

vector<int> parent(n,0);
//vector<int> parent(n,0);
for(int i=0;i<n;i++) parent[i]=i;
for(auto& ed:adj_lis)
{

   int x=find_parent(ed[0],parent);
   int y=find_parent(ed[1],parent);
   if(x==y)
   continue;
   else
   parent[y]=x;
    cout<<ed[0]<<"  ---- "<<ed[1]<<" "<<ed[2]<<endl;



int graph_input()
{
   vvint v;
   int n,N;
   cin>>n>>N;
   int a,b,c;
   for(int i=0;i<N;i++)
   {   vint temp;
       cin>>a>>b; cin >>c;
       temp.push_back(a);
```

```
        temp.push_back(b);
        temp.push_back(c);
        v.push_back(temp);

    }


    sort(v.begin(),v.end(),compare2);

    cout<<"I am here"<<endl;

     solve(n,N,v);
    return 0;


}
```

This all the Greedy Algorithm.

Articulation Points (or Cut Vertices) in a Graph

It's simply says that remove an vertices and if the graph is divided into two components then that point is know as Articulation point.

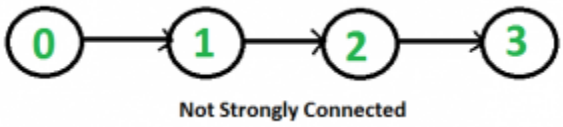


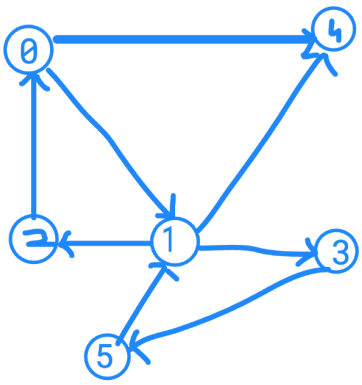
Articulation Point is 1

The first approach is to remove the point(Vertices) and run a dfs/bfs to check if there is any disconnected component.
It 's complexity will be O(V*(V+E)).
We need to optimize the BFS.

Kosaraju's Connected Components:-

A **connected component** is a subgraph of a graph in which there exists a path between any two vertices, and no vertex of the subgraph shares an edge with a vertex outside of the subgraph.





Here the 0 1 2 3 5 are strongly connected and 4 is not.

The intuition behind it was to get the edge that is being we have to find those vertices which are fully visited and then push them in a stack, in this way we will get ordering of the vertices in sorted manner of their visit.
Use Stack in a dfs to store the visited the node.
The stack will look like - 2,5,3,4,1,0.
Now we will reverse the adjacency list so that the graph will be fully transposed in this way we can get the Truly connected graph as the definition says we should be able to other edge from on edge in a Strongly Connected Components.

After taking the transpose we will perform the dfs again on the transposed adjacency list and will print the vertices along the way till it comes to main function . then we will print endl and will continue.

```
int dfs(vector<vector<int> > &adj_lis,int node,vector<int> &vis,stack<int> &st)
{
    vis[node]=1;
    for(auto &child:adj_lis[node])
    {
        if(!vis[child])
            dfs(adj_lis,child, vis,st);
    }
    st.push(node);
}
int reversedfs(vector<int> transpose[],int node,vector<int> &vis)
{
    vis[node]=1;
    cout<<node<<" ";
    for(auto &child:transpose[node])
    {
        if(!vis[child])
               reversedfs(transpose,child,vis);
    }
}

int solve(int n, int N,vector<vector<int> > adj_lis)
{

   stack<int> st;
   vector<int> vis(n, 0);
   int timer = 0;
   for(int i = 0;i<n;i++) {
       if(!vis[i]) {
           dfs(adj_lis,i, vis,st);
       }
   }

   vector<int> transpose[n];
   for(int i=0;i<n;i++)
   {
       for(auto &ed:adj_lis[i])
       {
           transpose[ed].push_back(i);
```

```
        }
    }

    vis.assign(n,0);
    while(!st.empty())
    {
        int node=st.top(); st.pop();
        if(!vis[node])
        {
            cout<<"SCC :- ";
            reversedfs(transpose,node,vis);
            cout<<endl;
        }
    }

return 0;
}
```

Bellman Ford Algorithm:-

This algorithm is introduced as the Dijkstra Algorithm cannot help us to find the shortest path when the edges are negative.As whenever we will push a weight it will always move on the top of the priority queue. So we can't use in case of negative edges.

To use bellman ford algorithm we have keep in mind that there shouldn't be a cycle with negative edges as it occurs then there will be a infinite loop of diminish weight's. So when there is a negative cycle the algorithm will not work properly.

To implement the undirected graph we have to converted to directed graph.
In the cycle there can be negative edge but the total cyle sum weight should not be zero.

Let's take an example
0—---->1—-------> 2—------>3—------>4
   -1   2   4   -1
If their adjacency list is formed differently rather than in sorted order and considering the source as 0.
[[3,4,-1],[2,3,4],[0,1,-1],[1,2,2]]----- In this case till we haven't processed the distance for 0—->1 we cannot get other's distance. To find all the distance we fill iterate the loop for n-1 time n being the number of nodes.

```
nt solve(int n, int N,vvint v)
{
    vint dist(n,100000);
    dist[0]=0;
```

```
   for(int i=0;i<n-1;i++)
   {
       for(int j=0;j<N;j++)
       {
           if(dist[v[j][0]]+v[j][2]<dist[v[j][1]])
           {
               dist[v[j][1]]=dist[v[j][0]]+v[j][2];
           }
       }

   }


       int flag=0;
       for(int j=0;j<N;j++)
       {
           if(dist[v[j][0]]+v[j][2]<dist[v[j][1]])
           {
               flag=1;
           }
       }

       if(flag==1)
       cout<<"There is a cycle with a negative weights"<<endl;
       else
       {
       for(int j=0;j<n;j++)
       {
          cout<<dist[j]<<endl;
       }
       }




return 0;
}
vvint v;
   int n,N;
   cin>>n>>N;
   int a,b,c;
   for(int i=0;i<N;i++)
   {   vint temp;
       cin>>a>>b;cin >>c;
```

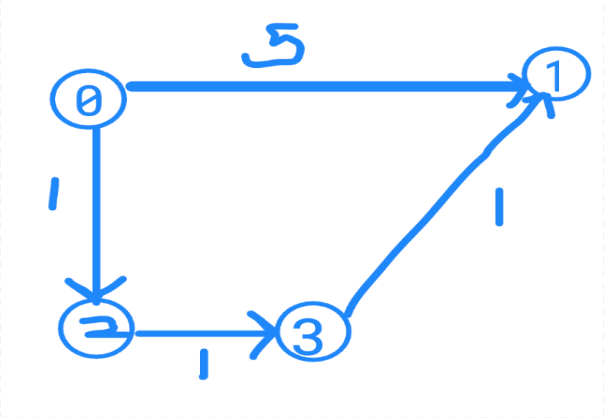
```
        temp.push_back(a);
        temp.push_back(b);
        temp.push_back(c);
        v.push_back(temp);

    }
solve(n,N,v);
}
```

Floyd warshall algorithm:-

THis is not single source shortest path unlike algorithm, here we will find all pair shortest path that means we will find the shortest path from all vertices to all other vertices if it's reachable.

The one thing that we have to always remember is that we have to avoid the negative weighted cycle.

Step1:- We will compute all the edges and their distances. To find the distances.
For example to find the distance from 0 5 :-



We will take various approaches :-
0-0-1 will give us 5
0-2-1 will give us infinite as there is no direct patch for 2-1 as of now , need to be calculated.
0-1-1 will give us 5
0-3-1 no path for 0-5 directly as of now.

The main concept here is
dist[i][j]=dist[i][intermediate_node]+dist[intermediate_node][j];
So how we can do that:-
Let's assume we are going through the way :-

[0][0]= [0][0]+[0][0]
[0][1]+[1][0]
[0][2]+[2][0]
[0][3]+[3][0]

[0][1]= [0][0]+[0][1]
[0][1]+[1][1]
[0][2]+[2][1]
[0][3]+[3][1]

[0][2]= [0][0]+[0][2]
[0][1]+[1][2]
[0][2]+[2][2]
[0][3]+[3][2]

[0][3]= [0][0]+[0][3]
[0][1]+[1][3]
[0][2]+[2][3]
[0][3]+[3][3]

Similarly we can do for the rest of the edges but there is a problem here:-

Let's take the case of 0—1 we will get a value of 5 but that will not be a correct answer because the shortest path would be from 0-2-3-1 but we can't calculate that earlier as we haven't calculated the distance for 3-1 or 2-3 or 2-1 . **Hence if we follow above approach we won't get the correct answer.**

So we will find it another way:-

[0][0] = [0][0] + [0][0]
[0][1] = [0][0] + [0][1]
[0][2] = [0][0] + [0][2]
[0][3] = [0][0] + [0][3]

[1][0] = [1][0] + [0][0]
[1][1] = [1][0] + [0][1]
[1][2] = [1][0] + [0][2]
[1][3] = [1][0] + [0][3]

[2][0] = [2][0] + [0][0]
[2][1] = [2][0] + [0][1]
[2][2] = [2][0] + [0][2]
[2][3] = [2][0] + [0][3]

[3][0] = [3][0] + [0][0]
[3][1] = [3][0] + [0][1]
[3][2] = [3][0] + [0][2]
[3][3] = [3][0] + [0][3]

[0][0] = [0][1] + [1][0]
[0][1] = [0][1] + [1][1]
[0][2] = [0][1] + [1][2]
[0][3] = [0][1] + [1][3]

[1][0] = [1][1] + [1][0]
[1][1] = [1][1] + [1][1]
[1][2] = [1][1] + [1][2]
[1][3] = [1][1] + [1][3]

[2][0] = [2][1] + [1][0]
[2][1] = [2][1] + [1][1]
[2][2] = [2][1] + [1][2]
[2][3] = [2][1] + [1][3]

[3][0] = [3][1] + [1][0]
[3][1] = [3][1] + [1][1]
[3][2] = [3][1] + [1][2]
[3][3] = [3][1] + [1][3]
We are trying to find the distance of one edge to other using different times but using different edges.If we use this method we will get the proper results and get the answer for this.

The below is code for that:-

```
int solve(int n, int N,vector<vector<int> > adj)
{

cout<<" I am here"<<endl;
   for(int i=0;i<n;i++)
   {
      adj[i][i]=0;



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

               // if(j==k) continue;

               if(adj[i][k]==1000 && adj[j][i]==1000) continue;

                cout<<"["<<j<<"]"<<"["<< k<<"]";
```

```
                cout<<" = ";
                cout<<"["<<j<<"]"<<"["<<i<<"]";
                cout<<" + ";
                cout<<"["<<i<<"]"<<"["<<k<<"]";



            }
            cout<<endl;
        }
    }
      for(int i=0;i<n;i++)
    {
        for(int j=0;j<n;j++)
        {
            if(adj[i][j]==INT_MAX)
            {
                cout<<"INF"<<" ";continue;
            }
            cout<<adj[i][j]<<" ";
        }
        cout<<endl;
    }


    return 0;

}
```