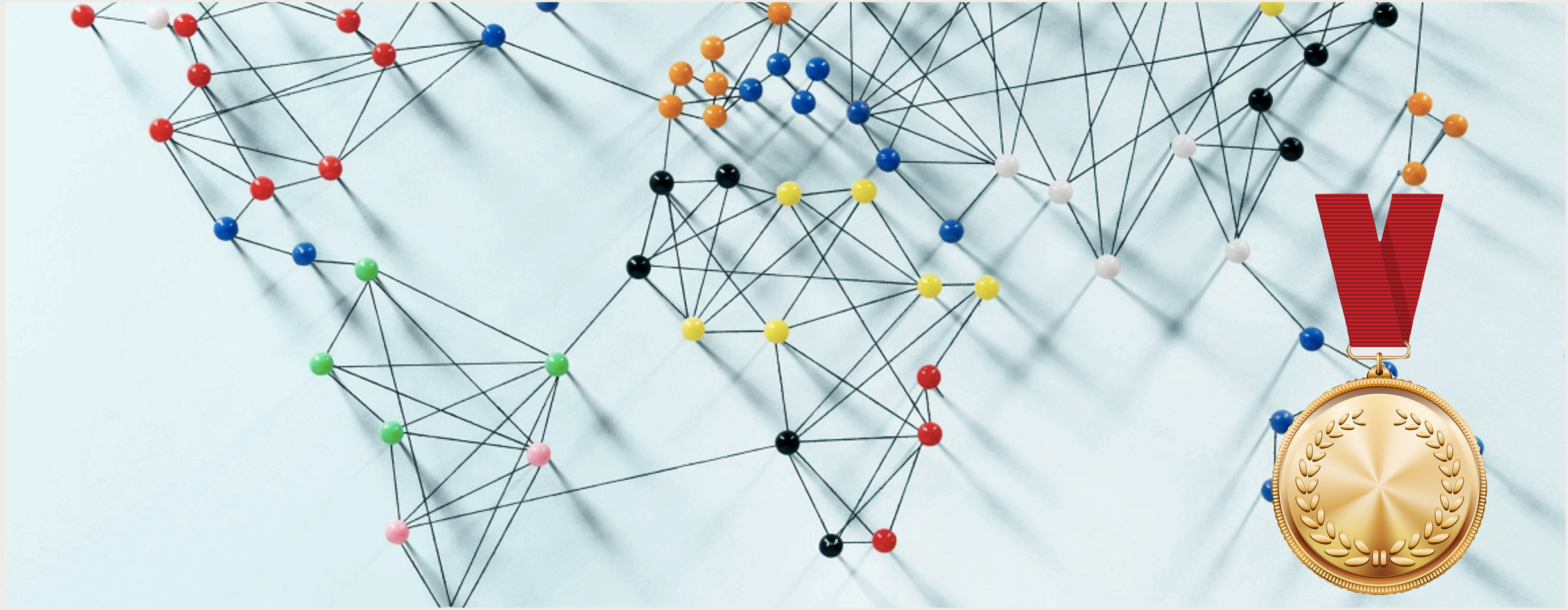


GRAPH PROBLEMS

AJAY KUMAR

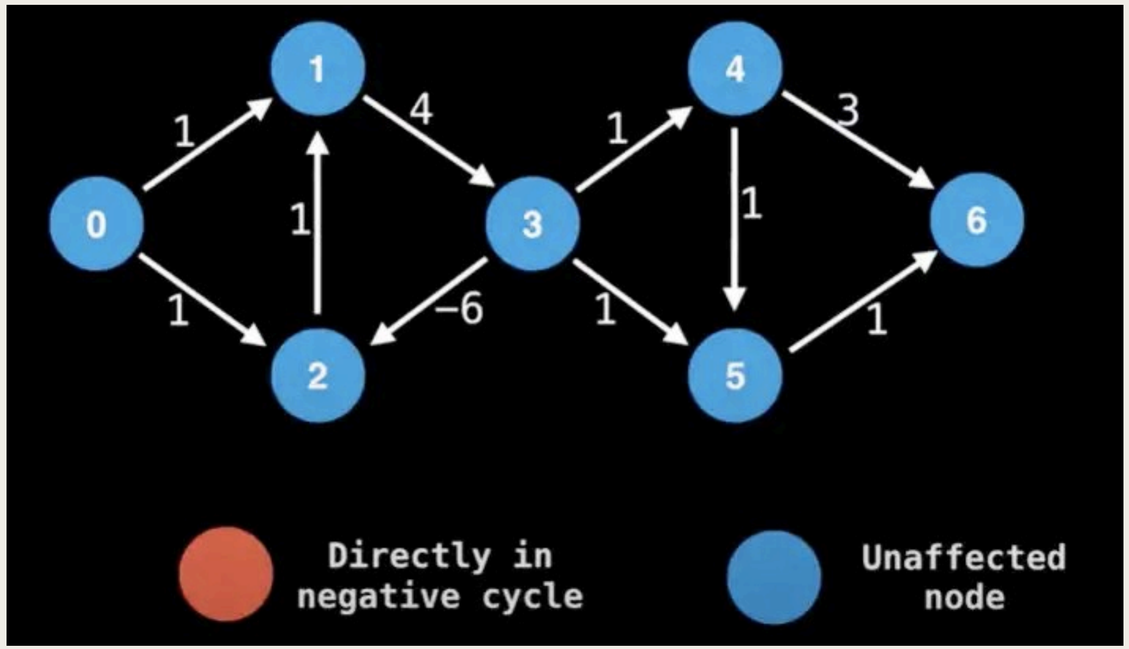
Codeforces

Negative Cycles –

Does my weighted digraph have any negative cycles? If so, where?

→ **Algorithms** – Bellman-Ford and Floyd-Warshall

Usecase example – currency exchange



Minimum Spanning Tree

- A minimum spanning tree (MST) is a subset of the edges of a connected, edge-weighted graph that connects all the vertices together, without any cycles, and with the minimum possible total edge weight/cost.
- → It’s a tree (meaning has no cycles) and spans the graph at minimum cost
- → All MSTs of the graph has the same cost and are not necessarily identical.
- → **Algorithms –** Kruskal’s, Prim’s & Boruvka’s algorithm
- → **Application –** Designing the least cost network, circuit design, transportation networks, and several approximation algorithms also rely on minimum spanning trees

Prim's Algorithm

- Prim's algorithm is also a **Greedy algorithm.**
- Prim's algorithm always starts with a **single node** and it moves through several adjacent nodes, in order to explore all of the connected edges along the way.
- The idea is to maintain two sets of vertices.
  1. The first set contains the vertices already included in the MST,
  2. the other set contains the vertices not yet included.
- At every step, it considers all the edges that connect the two sets and picks the minimum weight edge from these edges.
- After picking the edge, it moves the other endpoint of the edge to the set containing MST.

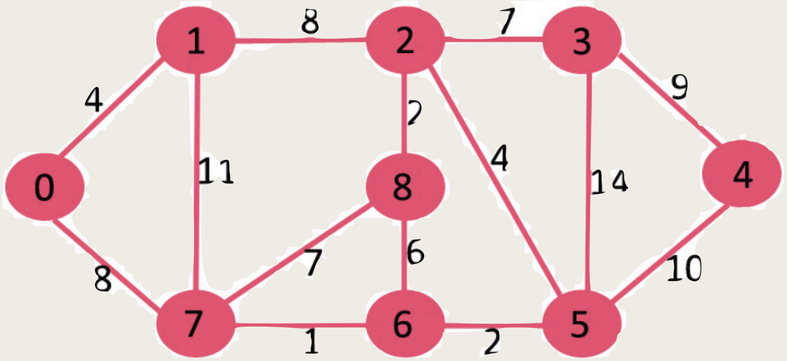
Steps:

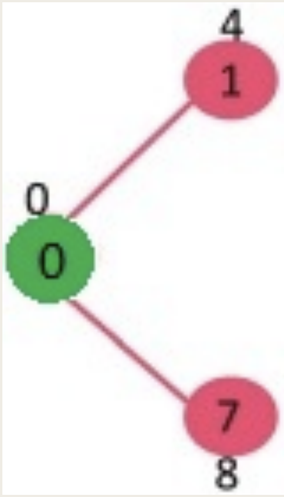
- Make a priority queue which store the neighbours in increasing order of their weight which are not yet visited
- So Make a visited array
- Initiallise the priority queue as {0,0} as {weight,node}
- Priority queue representation :

priority_queue<……>

```
Breakdown: first          pair<int,int>,
           Second         vector<pair<int,int>>,
           Third.         greater<pair<int,int>>
```

```
//Observation: This is kind of bfs like implementation.
Only weight is included and we have to visit the node on their priority
```





```
    priority_queue<pair<int,int>,vector<pair<int,int>>, greater<pair<int,int>>> pq;
    vector<int> vis(n);
    pq.push({0, 0});
    long long ans = 0;
    while(pq.size()){
        pair<int,int> p = pq.top();
        pq.pop();
        if(vis[p.second]){
            continue;
        }
        vis[p.second] = true;
        ans += p.first;
        int node = p.second;
        for(auto e : adj[node]){                    //Here adj is adjacency list of the graph
            int v = e.first;
            int w= e.second;
            if(!vis[v]){                //if the node is not visited and
                pq.push({w, v});
            }
        }
    }
    cout<<ans<<"\n";
    return 0;
}
//Practice it
```

Problem Statement

A box contains N balls, each with an integer between 1 and $M-1$ written on it. For $i=1,2,\ldots,N$, the integer written on the i-th ball is A_i.

While the box has two or more balls remaining, Takahashi will repeat the following.

- First, choose two balls arbitrarily.
- Then, get a score equal to the remainder when x^y+y^x is divided by M, where x and y are the integers written on the two balls.
- Finally, choose one of the two balls arbitrarily, eat it, and return the other to the box.

Print the maximum possible total score Takahashi will get.

Constraints

- $2 \leq N \leq 500$
- $2 \leq M \leq 10^9$
- $1 \leq A_i \leq M-1$
- All values in the input are integers.

Input

The input is given from Standard Input in the following format:

```
N M
A_1 A_2 ... A_N
```

Output

Print the answer.

Editorial

- Consider a **complete Graph G** with N vertices corresponding to the N balls in which the edge between vertex is and vertex j has a weight $(A_i^{A_j} + A_j^{A_i})\ mod\ M$ (that is score obtained when ball I and ball j chosen from the box) for each integer pair (I,j) such that 1<=j<=N.

- The Answer of the problem is the **weight of maximum spaning tree**

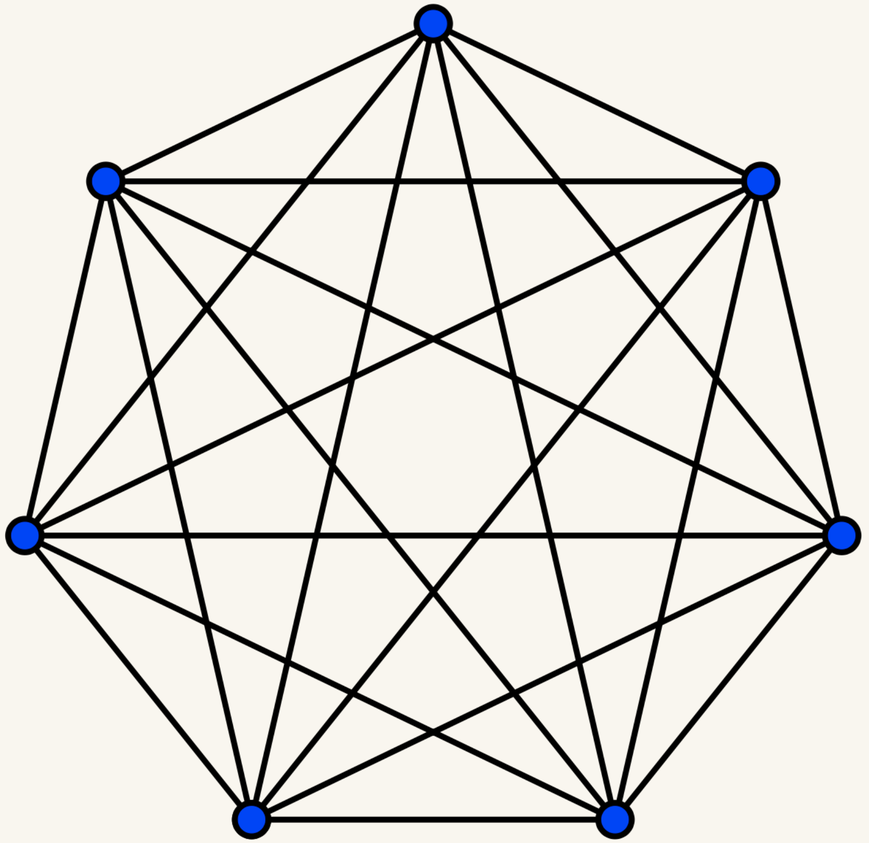
- $(A_i^{A_j} + A_j^{A_i})\ modM$ can be computer **fast enough with the fast exponentiation**

- **Maximum Spanning tree graph can be found with Prim's Algorithm or Kruskal's Algorithm**

Complete Graph

- A complete graph is **a simple undirected graph in which every pair of distinct vertices is connected by a unique edge**.

- A complete digraph is a directed graph in which every pair of distinct vertices is connected by a pair of unique edges (one in each direction).



Steps to solve the maximum spanning tree

- First for any spanning tree T of the graph G , there is procedure to obtain to score equal to the wieght of T, This can be achieved by ‘Consider T as rooted tree with arbitary root and repeating the following operation until ball contains just 1 balls’
- Choose one arbitarty leaf v of T, and let p its parent
- Choose ball p and v correponding to vertices p and v from the box gaining a score equals to the weight of edge {p,v}.
- Eat ball v and return ball p to the box.
- Also remove vertex { and edge {p,v}) from T

Krushkal’s algorithms

Instead Kruskal’s algorithm, the initial spanning tree only contains the nodes of the graph and does not contain any edges.

Then the algorithm goes through the edges ordered by their weights, and always adds an edge to the tree if it does not create a cycle.

The algorithm maintains the components of the tree.

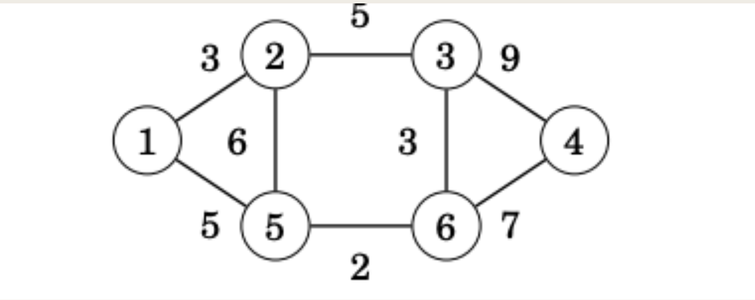
Initially, each node of the graph belongs to a separate component.

Always when an edge is added to the tree, two components are joined.

Finally, all nodes belong to the same component, and a minimum spanning tree has been found.

Example

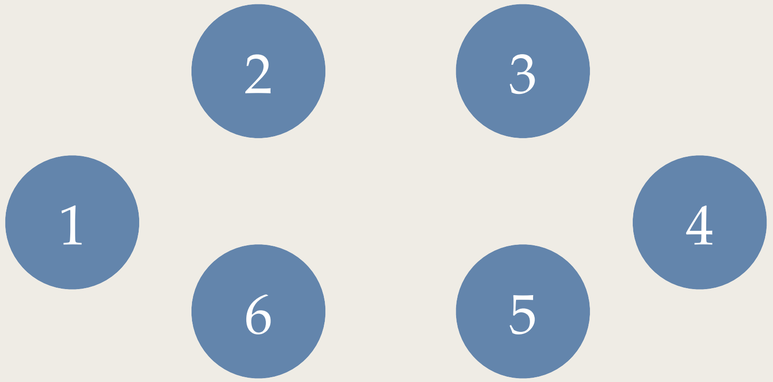
Let us consider how Kruskal's algorithm processes the following graph:



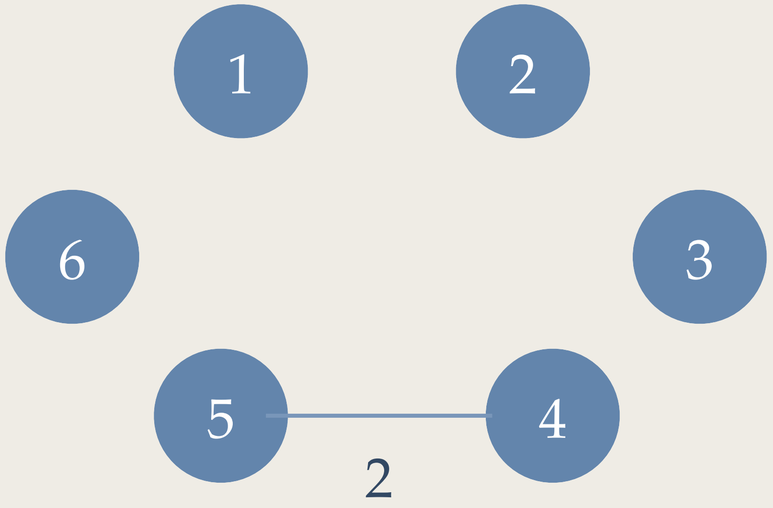
The first step of the algorithm is to sort the edges in increasing order of their weights. The result is the following list:

edge weight
5–6 2
1–2 3
3–6 3
1–5 5
2–3 5
2–5 6
4–6 7
3–4 9

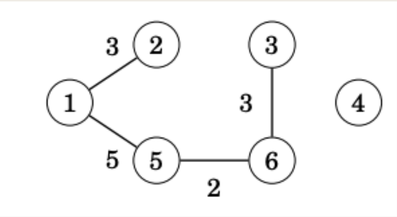
- After this, the algorithm goes through the list and adds each edge to the tree if it joins two separate components. Initially, each node is in its own component:
- After this, the algorithm goes through the list and adds each edge to the tree if it joins two separate components. Initially, each node is in its own component:



The first edge to be added to the tree is the edge 5–6 that creates a component {5,6} by joining the components {5} and {6}:

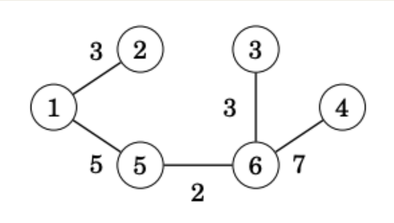


After this, the edges 1–2, 3–6 and 1–5 are added in a similar way:



After those steps, most components have been joined and there are two components in the tree: {1,2,3,5,6} and {4}. The next edge in the list is the edge 2–3, but it will not be included in the tree, because nodes 2 and 3 are already in the same component. For the same reason, the edge 2–5 will not be included in the tree.

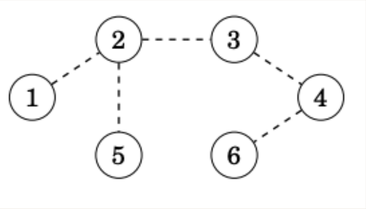
Finally, the edge 4–6 will be included in the tree:



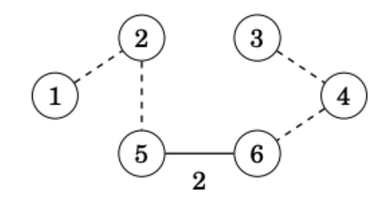
After this, the algorithm will not add any new edges, because the graph is connected and there is a path between any two nodes. The resulting graph is a minimum spanning tree with weight 2+3+3+5+7 = 20.

Why does this work?

It is a good question why Kruskal's algorithm works. Why does the greedy strategy guarantee that we will find a minimum spanning tree? Let us see what happens if the minimum weight edge of the graph is not included in the spanning tree. For example, suppose that a spanning tree for the previous graph would not contain the minimum weight edge 5–6. We do not know the exact structure of such a spanning tree, but in any case it has to contain some edges. Assume that the tree would be as follows



However, it is not possible that the above tree would be a minimum spanning tree for the graph. The reason for this is that we can remove an edge from the tree and replace it with the minimum weight edge 5–6. This produces a spanning tree whose weight is smaller:



For this reason,

- it is always optimal to include the minimum weight edge in the tree to produce a minimum spanning tree.
- Using a similar argument, we can show that it is also optimal to add the next edge in weight order to the tree, and so on.
- Hence, Kruskal's algorithm works correctly and always produces a minimum spanning tree.
- Implementation When implementing Kruskal's algorithm,
- it is convenient to use the edge list representation of the graph.
- The first phase of the algorithm sorts the edges in the list in O(mlogm) time.
- After this, the second phase of the algorithm builds the minimum spanning tree as follows
- for (...) { if (!same(a,b)) unite(a,b); }

The loop goes through the edges in the list and always processes an edge a–b where a and b are two nodes.

Two functions are needed:

1. The function same determines if a and b are in the same component, and
2. the function unite joins the components that contain a and b.

Need for Union find Structure

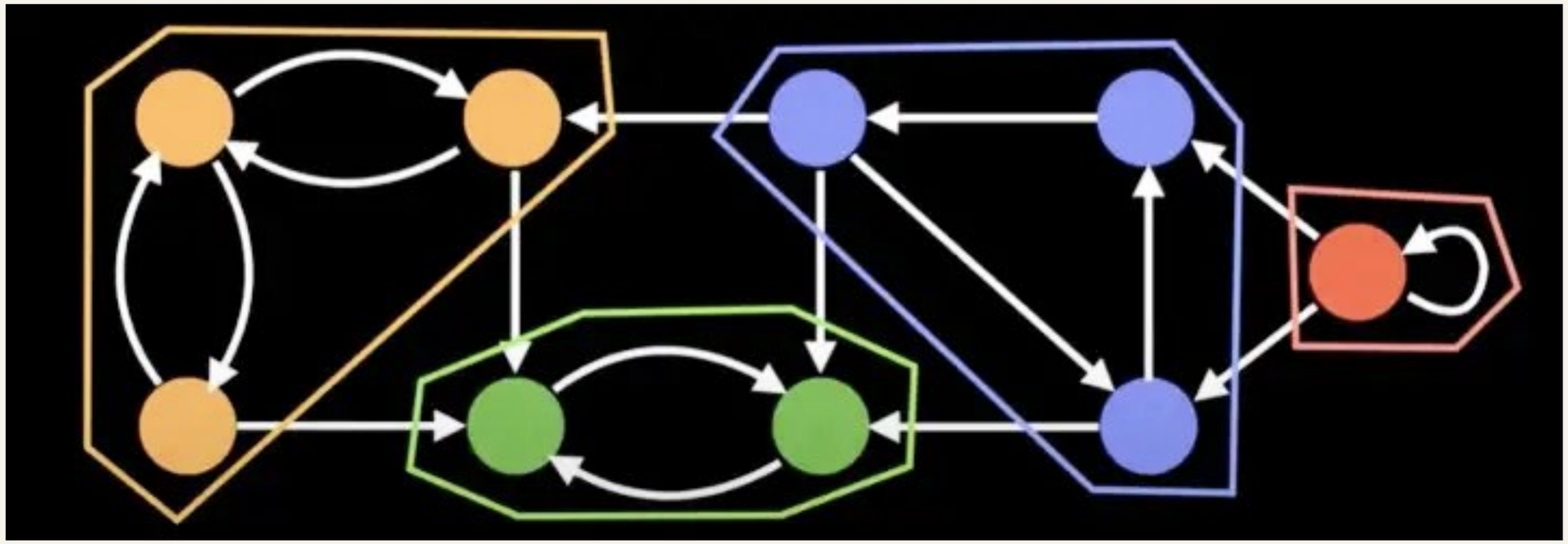
- The problem is how to efficiently implement the functions same and unite.
- One possibility is to implement the function same as a graph traversal and check if we can get from node a to node b.
- However, the time complexity of such a function would be O(n+ m) and the resulting algorithm would be slow, because the function same will be called for each edge in the graph.
- We will solve the problem using a union-find structure that implements both functions in O(logn) time.
- Thus, the time complexity of Kruskal's algorithm will be O(mlogn) after sorting the edge list.

```
struct Edge {
    int u, v, weight;
    bool operator<(Edge const& other) {
        return weight < other.weight;
    }
};

int n;
vector<Edge> edges;

int cost = 0;
vector<int> tree_id(n);
vector<Edge> result;
for (int i = 0; i < n; i++)
    tree_id[i] = i;

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

for (Edge e : edges) {
    if (tree_id[e.u] != tree_id[e.v]) {
        cost += e.weight;
        result.push_back(e);

        int old_id = tree_id[e.u], new_id = tree_id[e.v];
        for (int i = 0; i < n; i++) {
            if (tree_id[i] == old_id)
                tree_id[i] = new_id;
        }
    }
}
```

Strongly connected components

Can be thought of as self-contained cycles within a directed graph where every vertex in a given cycle can reach every other vertex in the same cycle.

→ **Algorithms –** Tarjan's and Kosaraju's algorithm



Strongly connected component is a maximal subset of vertices C such that any two vertices of this subset are reachable from each other, i.e. for any $u, v \in C$

$$u \mapsto v, v \mapsto u$$

- It is obvious, that strongly connected components do not intersect each other, i.e. this is a partition of all graph vertices.
- Thus we can give a definition of condensation graph G^{SCC}as a graph containing every strongly connected component as one vertex.
- Each vertex of the condensation graph corresponds to the strongly connected component of graph G
- There is an oriented edge between two vertices $C_i \; and \; C_j$ of the condensation graph if and only if there are two vertices $u \in C_i, v \in C_j$ such that there is an edge in initial graph, i.e. $(u, v) \in E$

The most important property of the condensation graph is that it is **acyclic**.

Indeed, suppose that there is an edge between C and C'

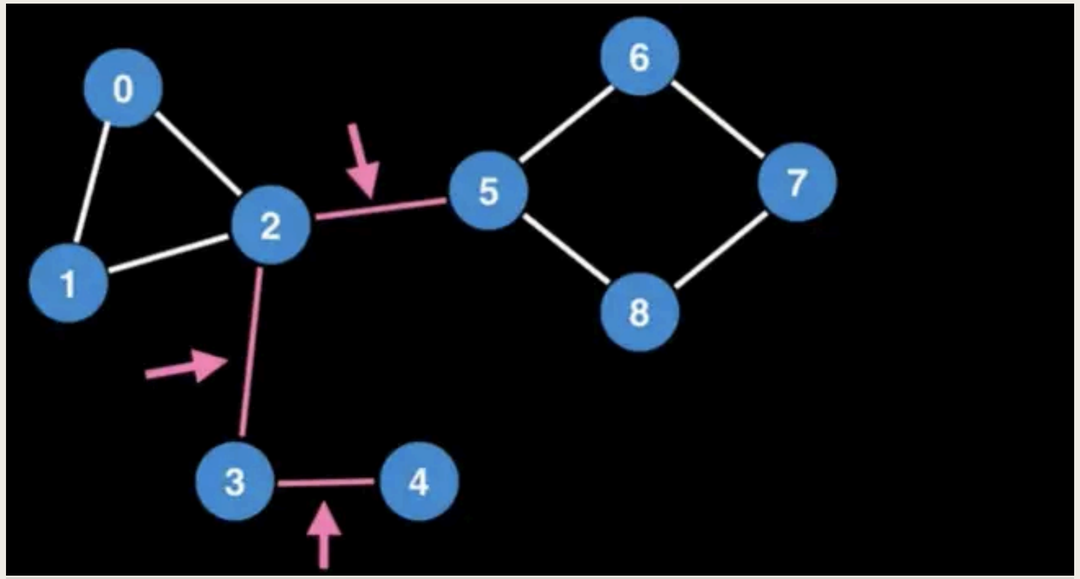
let's prove that there is no edge from C' to C.

Proof:

- Suppose that $C' \mapsto C$.
- Then there are two vertices $u' \in C$ and $v' \in C'$ such that $v' \mapsto u'$
- But since u and u’ are in the same connected component then there is a path between them;
- the same for v and v’.
- As a result if we joint these paths we have that $v \mapsto u$ and at the same time $u \mapsto v$.
- There u and v should be at the same strongly connected componect.
- So this is contradiction. This completes the proof.

Finding Bridges

- A bridge/cut edge is any **edge** in a graph whose removal increases the number of connected components.
- → Bridges are important in graph theory because they often hint at weak points, bottlenecks, or vulnerabilities in a graph.
- → Think of the graph as a telephone network, bridges between islands



27.2 Finding bridges in a graph in O(N + M)

We are given an undirected graph. A bridge is defined as an edge which, when removed, makes the graph disconnected (or more precisely, increases the number of connected components in the graph). The task is to find all bridges in the given graph.

Informally, the problem is formulated as follows: given a map of cities connected with roads, find all “important” roads, i.e. roads which, when removed, cause disappearance of a path between some pair of cities.

The algorithm described here is based on depth first search and has O(N + M) complexity, where N is the number of vertices and M is the number of edges in the graph.

Note that there is also the article Finding Bridges Online - unlike the offline algorithm described here, the online algorithm is able to maintain the list of all bridges in a changing graph (assuming that the only type of change is addition of new edges).

27.2.1 Algorithm

Pick an arbitrary vertex of the graph root and run depth first search from it. Note the following fact (which is easy to prove):

- Let's say we are in the DFS, looking through the edges starting from vertex v. The current edge (v, to) is a bridge if and only if none of the vertices to and its descendants in the DFS traversal tree has a back-edge to vertex v or any of its ancestors. Indeed, this condition means that there is no other way from v to to except for edge (v, to).

Now we have to learn to check this fact for each vertex efficiently. We'll use "time of entry into node" computed by the depth first search.

So, let tin[v] denote entry time for node v. We introduce an array low which will let us check the fact for each vertex v. low[v] is the minimum of tin[v], the entry times tin[p] for each node p that is connected to node v via a back-edge (v, p) and the values of low[to] for each vertex to which is a direct descendant of v in the DFS tree:

$$low[v] = \min \begin{cases} tin[v] & \\ tin[p] & \text{for all } p \text{ for which } (v, p) \text{ is a back edge} \\ low[to] & \text{for all } to \text{ for which } (v, to) \text{ is a tree edge} \end{cases}$$

Now, there is a back edge from vertex v or one of its descendants to one of its ancestors if and only if vertex v has a child to for which low[to] ≤ tin[v]. If low[to] = tin[v], the back edge comes directly to v, otherwise it comes to one of the ancestors of v.

Thus, the current edge (v,to) in the DFS tree is a bridge if and only if low[to] > tin[v].

27.2.2 Implementation

The implementation needs to distinguish three cases: when we go down the edge in DFS tree, when we find a back edge to an ancestor of the vertex and when we return to a parent of the vertex. These are the cases:

- visited[to] = false - the edge is part of DFS tree;
- visited[to] = true && to/= parent - the edge is back edge to one of the ancestors;
- to = parent - the edge leads back to parent in DFS tree.

To implement this, we need a depth first search function which accepts the parent vertex of the current node.

C++ implementation

```
int n;                                  // number of nodes
vector<vector<int>> adj;                // adjacency list of graph
vector<bool> visited;
vector<int> tin, low;
int timer;
void dfs(int v, int p = -1)
{
    visited[v] = true;
    tin[v] = low[v] = timer++;
    for (int to : adj[v])
    {
        if (to == p)
            continue;
        if (visited[to])
        {
            low[v] = min(low[v], tin[to]);
        }
        else
        {
            dfs(to, v);
            low[v] = min(low[v], low[to]);
            if (low[to] > tin[v])
            IS_BRIDGE(v, to);
        }
    }
}
```

```
void find_bridges()
{
    timer = 0;
    visited.assign(n, false);
    tin.assign(n, -1);
    low.assign(n, -1);
    for (int i = 0; i < n; ++i)
    {
        if (!visited[i])
            dfs(i);
    }
}
```

Main function is find_bridges; it performs necessary initialization and starts depth first search in each connected component of the graph.

Function IS_BRIDGE(a, b) is some function that will process the fact that edge (a, b) is a bridge, for example, print it.

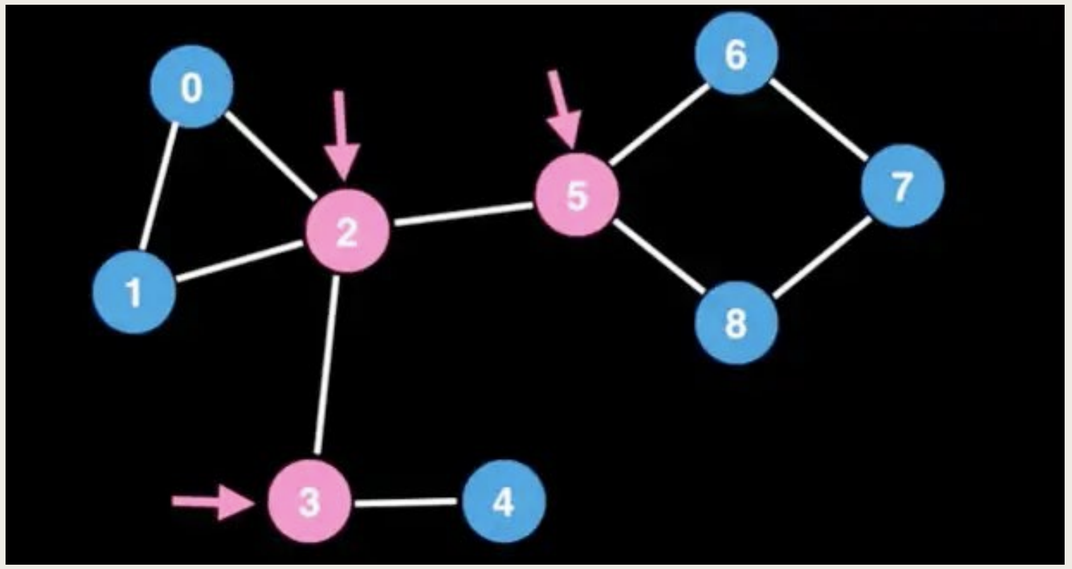
Note that this implementation malfunctions if the graph has multiple edges, since it ignores them. Of course, multiple edges will never be a part of the answer, so IS_BRIDGE can check additionally that the reported bridge is not a multiple edge. Alternatively it’s possible to pass to dfs the index of the edge used to enter the vertex instead of the parent vertex (and store the indices of all vertices).

Articulation Point

Articulation Points —

- An articulation point/cut vertex is any **node** in a graph whose removal increases the number of connected components.

- → Articulation points are important in graph theory because they often hint at weak points, bottlenecks or vulnerabilities in a graph.



27.4 Finding articulation points in a graph in O(N + M)

We are given an undirected graph. An articulation point (or cut vertex) is defined as a vertex which, when removed along with associated edges, makes the graph disconnected (or more precisely, increases the number of connected components in the graph). The task is to find all articulation points in the given graph.

The algorithm described here is based on depth first search and has O(N + M) complexity, where N is the number of vertices and M is the number of edges in the graph.

27.4.1 Algorithm

Pick an arbitrary vertex of the graph root and run depth first search from it. Note the following fact (which is easy to prove):

- Let's say we are in the DFS, looking through the edges starting from vertex v ≠ root.

If the current edge (v, to) is such that none of the vertices to or its descendants in the DFS traversal tree has a back-edge to any of ancestors of v, then v is an articulation point.
Otherwise, v is not an articulation point.

- Let's consider the remaining case of v = root. This vertex will be the point of articulation if and only if this vertex has more than one child in the DFS tree.

Now we have to learn to check this fact for each vertex efficiently.

- We'll use "time of entry into node" computed by the depth first search.
- So, let tin[v] denote entry time for node v.
- We introduce an array low[v] which will let us check the fact for each vertex v.
- low[v] is the minimum of tin[v], the entry times tin[p] for each node p that is connected to node v via a back-edge (v, p) and the values of low[to] for each vertex to which is a direct descendant of v in the DFS tree:

$$low[v] = \min \begin{cases} tin[v] & \\ tin[p] & \text{for all } p \text{ for which } (v,p) \text{ is a back edge} \\ low[to] & \text{for all } to \text{ for which } (v,to) \text{ is a tree edge} \end{cases}$$

- Now, there is a back edge from vertex v or one of its descendants to one of its ancestors if and only if vertex v has a child to for which low[to] < tin[v]. If low[to] = tin[v], the back edge comes directly to v, otherwise it comes to one of the ancestors of v.
- Thus, the vertex v in the DFS tree is an articulation point if and only if low[to] ≥ tin[v].

27.4.2 Implementation

The implementation needs to distinguish three cases: when we go down the edge in DFS tree, when we find a back edge to an ancestor of the vertex and when we return to a parent of the vertex. These are the cases:

- visited[to] = false - the edge is part of DFS tree;
- visited[to] = true && to/= parent - the edge is back edge to one of the ancestors;
- to = parent - the edge leads back to parent in DFS tree.

To implement this, we need a depth first search function which accepts the parent vertex of the current node.

C++ implementation

```
int n; // number of nodes
vector<vector<int>> adj; // adjacency list of graph

vector<bool> visited;
vector<int> tin, low;
int timer;

void dfs(int v, int p = -1) {
    visited[v] = true;
    tin[v] = low[v] = timer++;
    int children=0;
    for (int to : adj[v]) {
        if (to == p) continue;
        if (visited[to]) {
            low[v] = min(low[v], tin[to]);
        } else {
            dfs(to, v);
            low[v] = min(low[v], low[to]);
            if (low[to] >= tin[v] && p!=-1)
                IS_CUTPOINT(v);
            ++children;
        }
    }
    if(p == -1 && children > 1)
        IS_CUTPOINT(v);
}
```

```
void find_cutpoints() {
    timer = 0;
    visited.assign(n, false);
    tin.assign(n, -1);
    low.assign(n, -1);
    for (int i = 0; i < n; ++i) {
        if (!visited[i])
            dfs (i);
    }
}
```