

MST

History

The airline operates m flights between n cities, and the cost of the i -th flight is X_i euros.

The airline guarantees that using its flights, it is possible to travel from any city to any other.

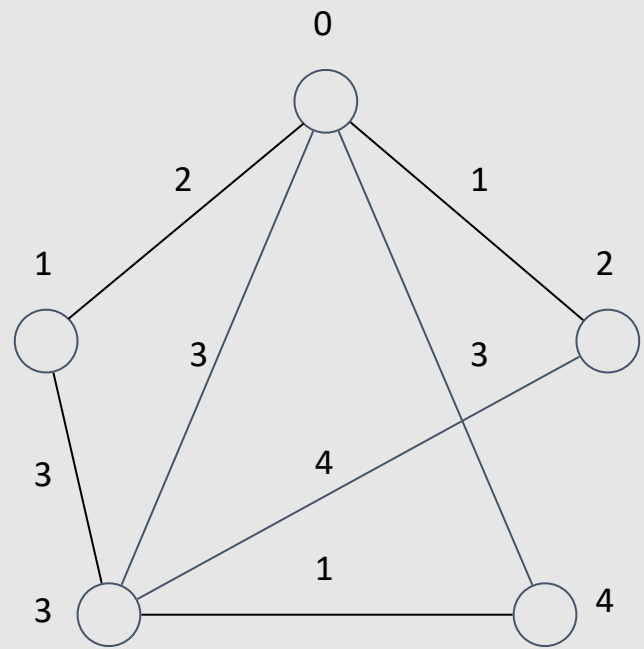
Due to the crisis in the country, the airline faces a challenging task - to remove certain flights while making the cost of the remaining flights as minimal as possible. However, this must be done while ensuring that the guarantee of traveling from any city to any other city remains intact.

MST

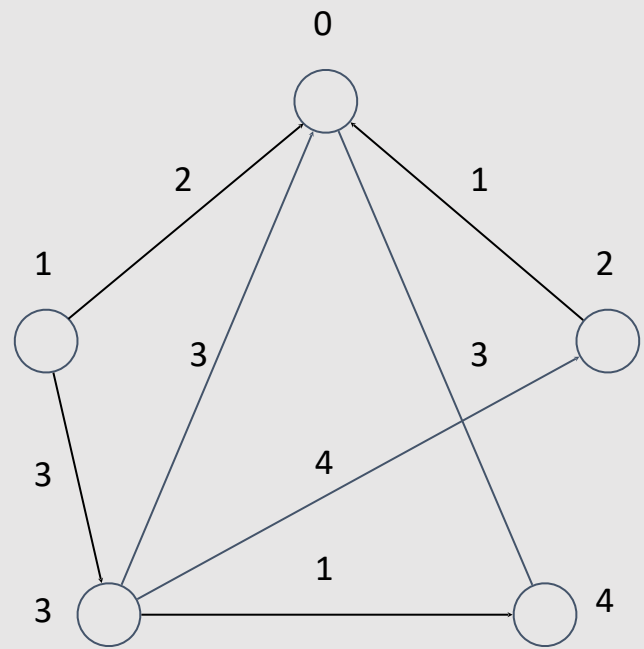
The task of MST (Minimal Spanning Tree) is to find a tree of minimal total weight in a connected weighted graph. It is typically assumed that the graph is undirected.

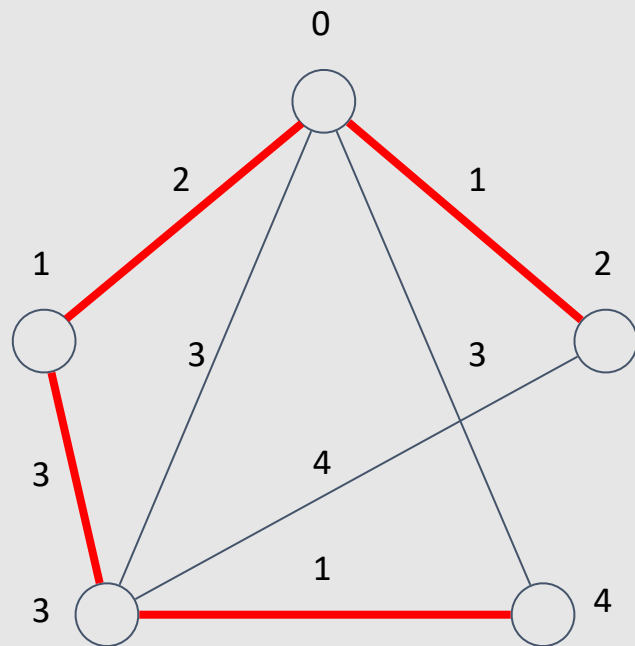
In a directed graph, this task is significantly more complex.

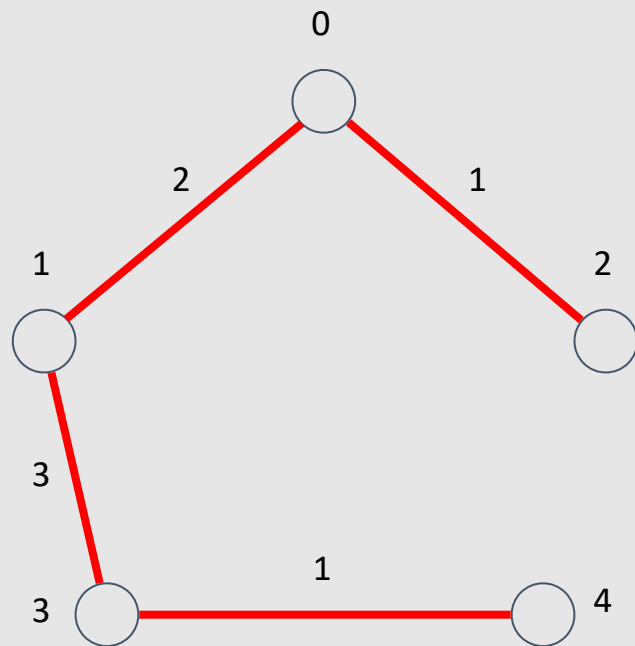
not directed graph



directed graph





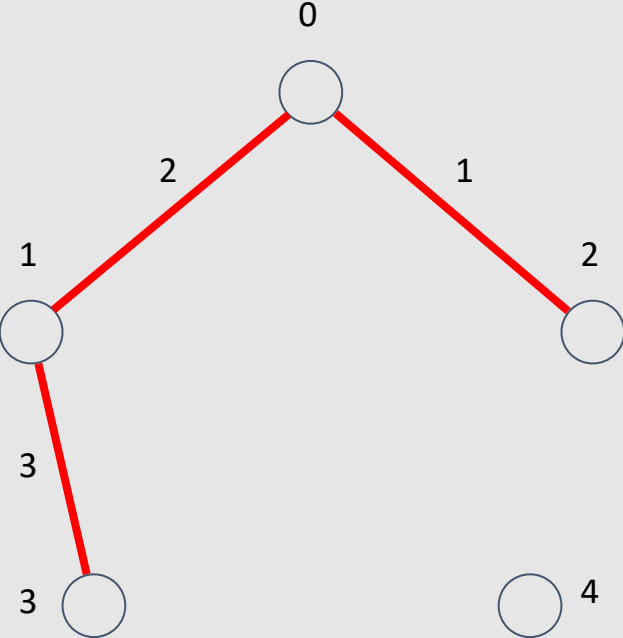


Definitions

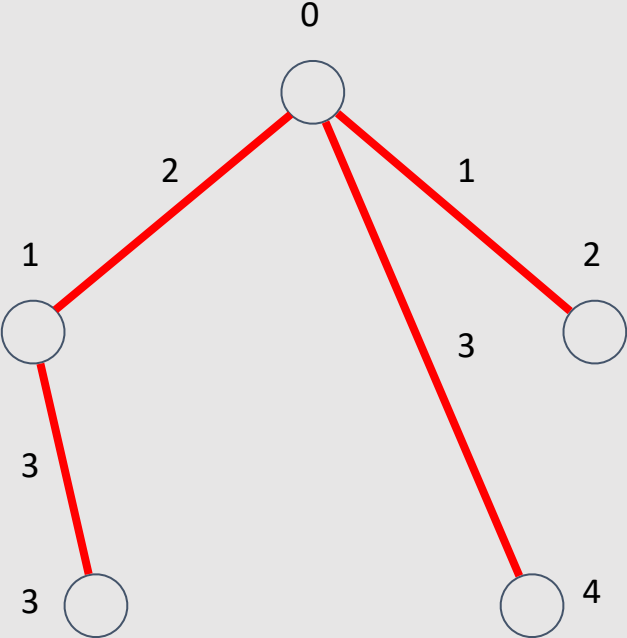
A subgraph G' of a graph G is called “safe” if it is a subgraph of some minimal spanning tree.

An edge (u, v) is called “safe” if, upon its addition to the subgraph G , the resulting subgraph G' remains safe, meaning it is also a subgraph of some minimal spanning tree.

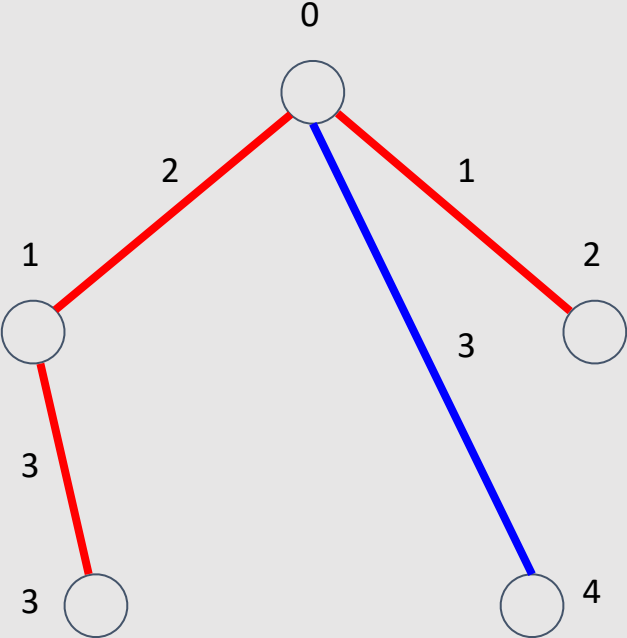
Safe graph G



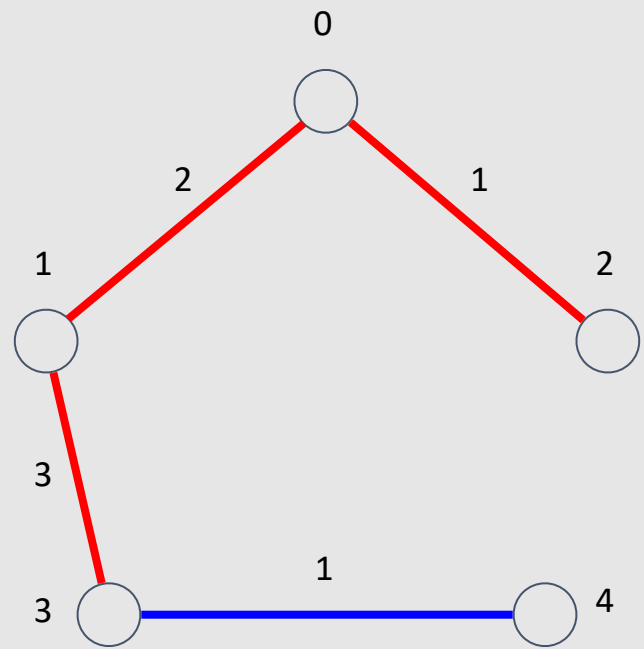
Not safe graph G



not safe edge (0, 4)

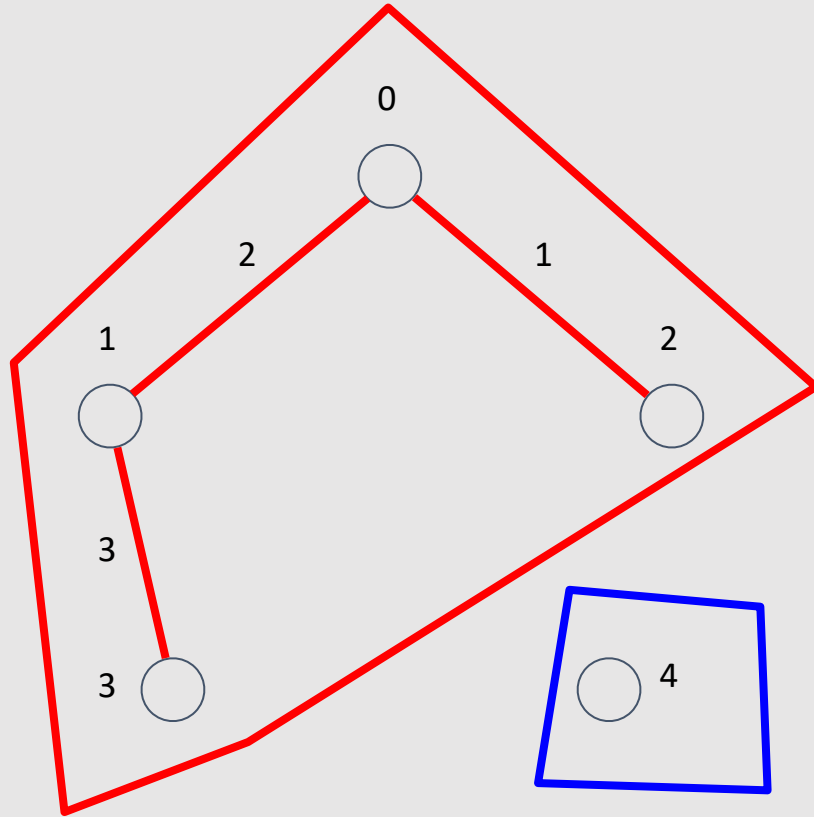


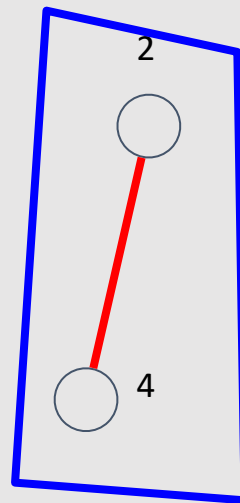
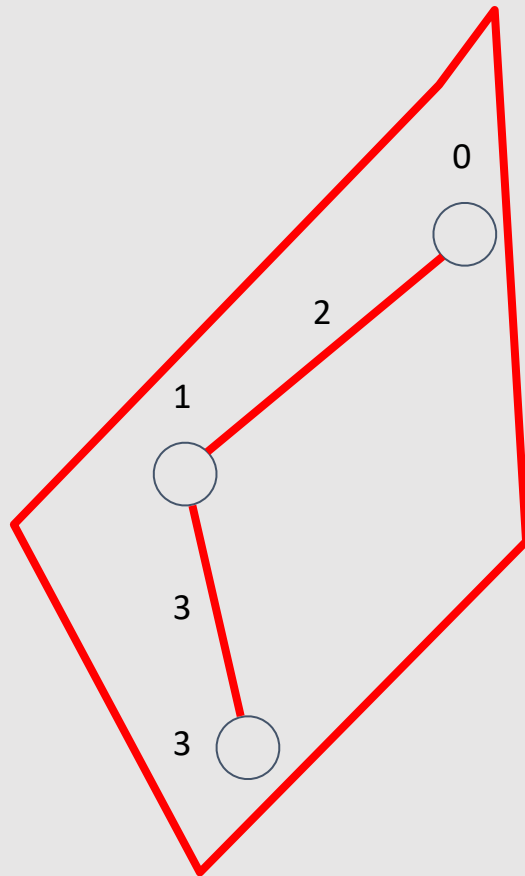
safe edge (3, 4)



Cut

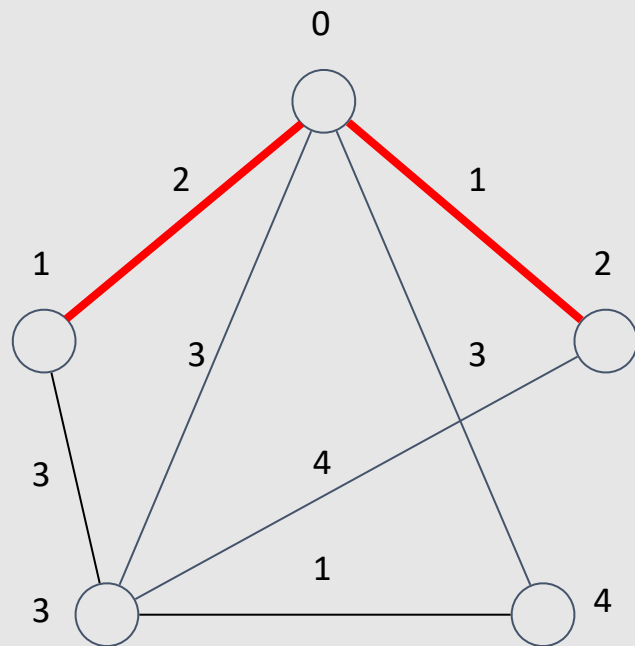
A cut is a partition of the vertices of a graph into two sets, between which there are no edges.

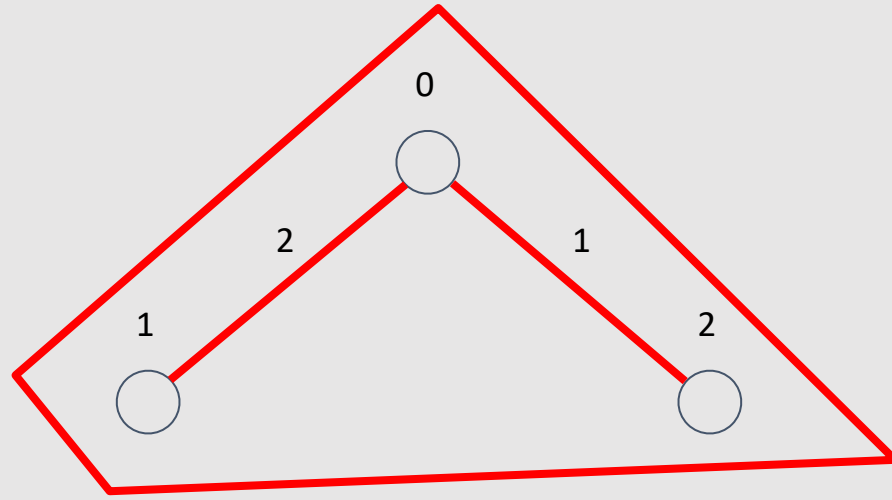


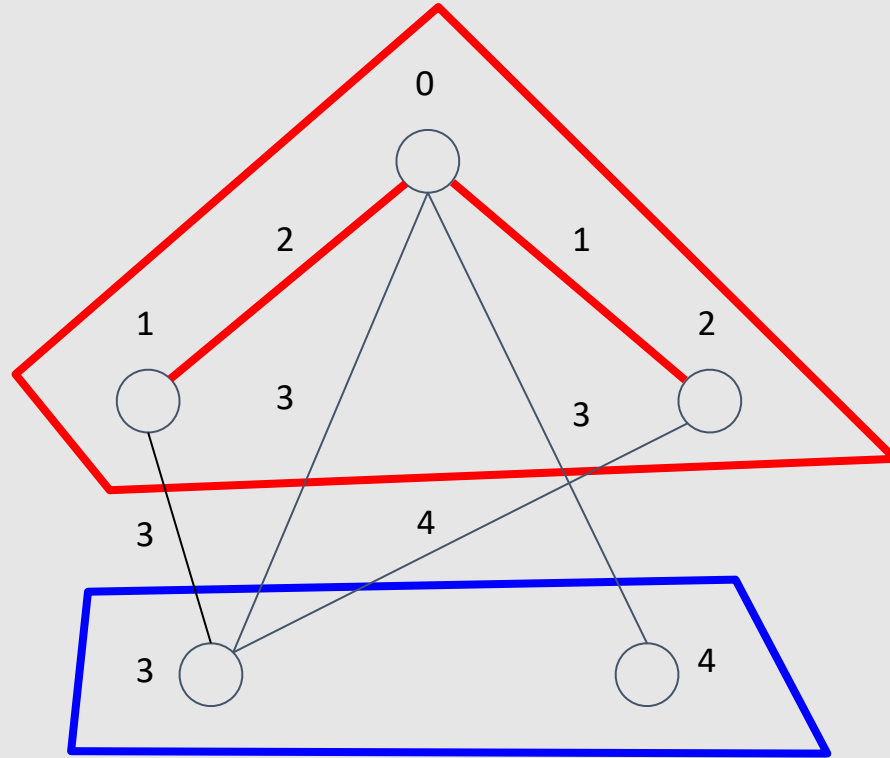


Safe edge lemma.

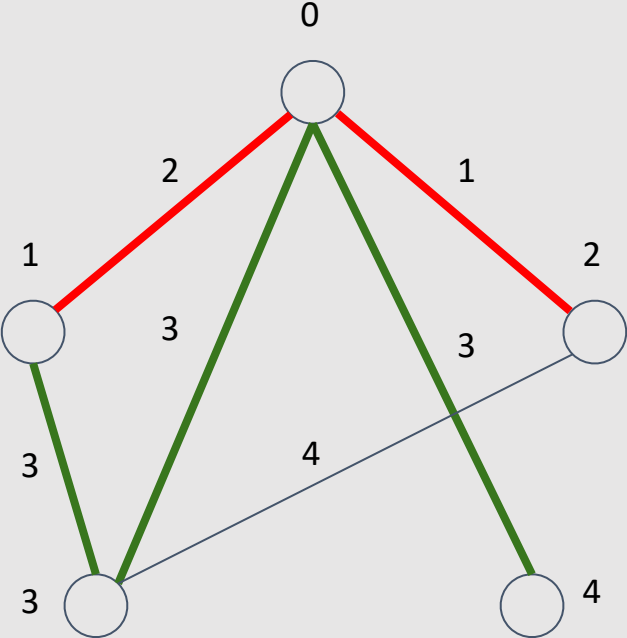
Consider any arbitrary cut within a subgraph of a minimal spanning tree. Then, the edge with the minimum weight that crosses this cut (i.e., connects the partitions when added) is “safe”.







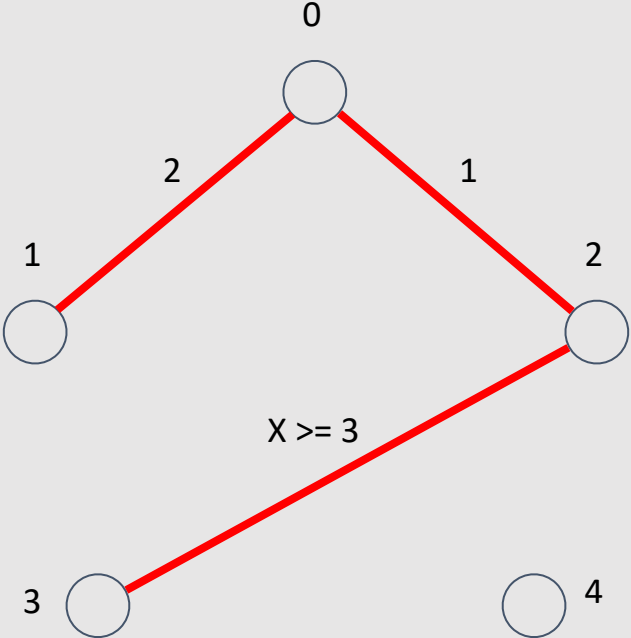
green - safe



Proof

Let's consider a particular minimal spanning tree in which this edge is not included.

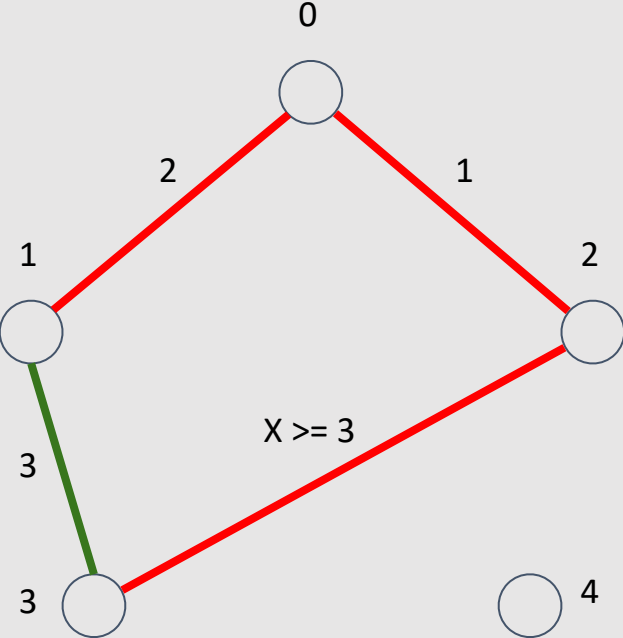
$W_{\text{cycle}} = C + X + 3$



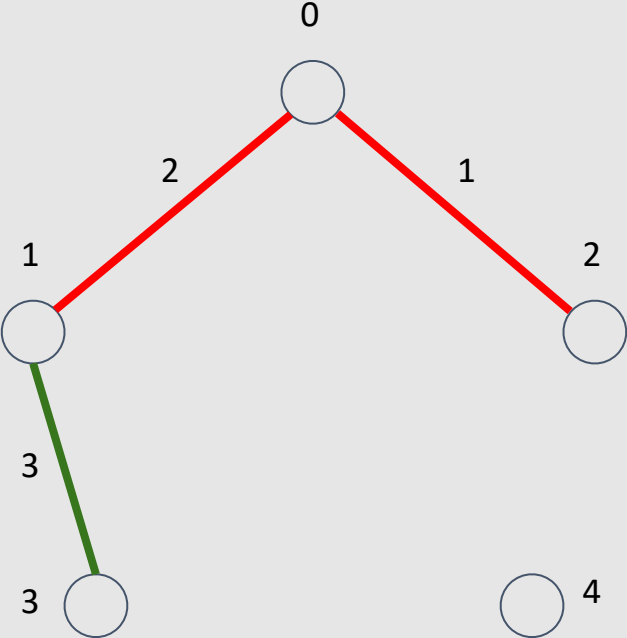
Proof

If it is added, a cycle is formed, from which an edge of no lesser weight can be removed, resulting in a solution that is at least not worse.

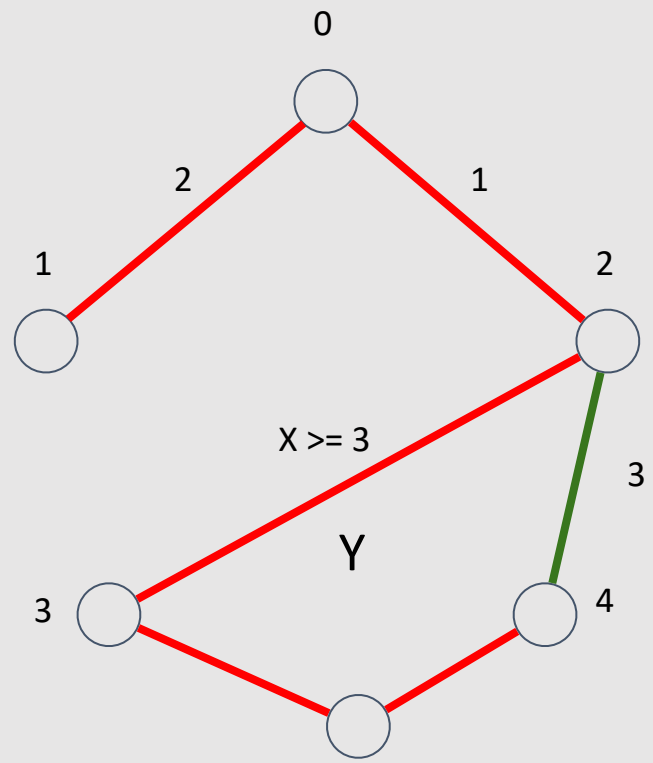
$W_{\text{cycle}} = C + X + 3$



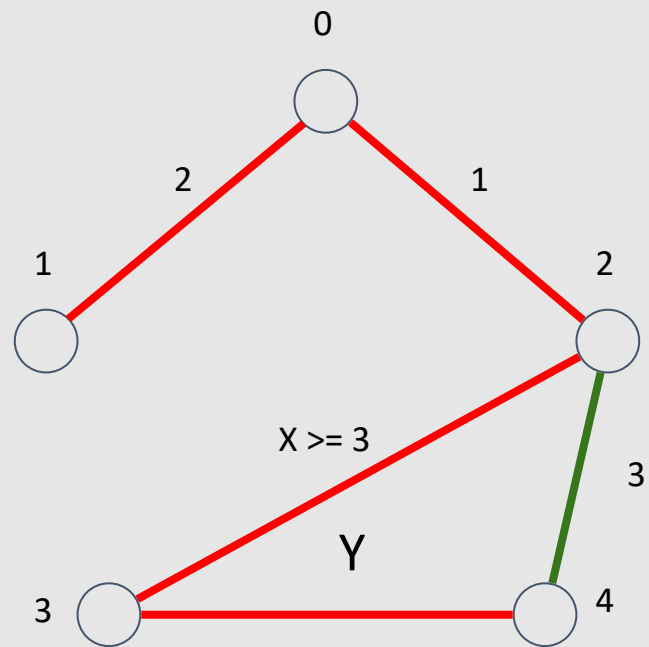
$W' = C + 3 \leq C + X$



$W_{\text{cycle}} = C + X + 3 + Y$



$W_{\text{cycle}} = C + 3 + Y \leq C + X + y$



Proof

In fact, we have reached a contradiction.

Algorithm

Actually, all minimum spanning tree algorithms work exactly like this - they swiftly identify a certain cut and locate the best edge within it.

Prim's algorithm

The Prim's algorithm operates step by step.

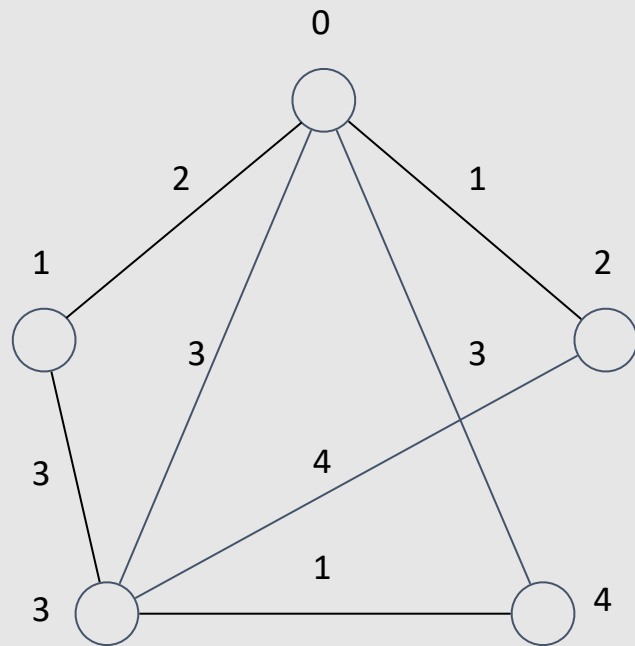
At each step, we have two sets of vertices:

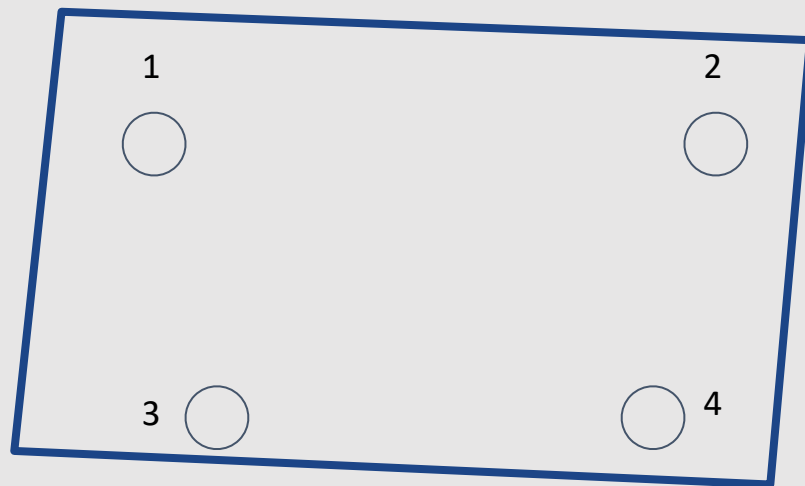
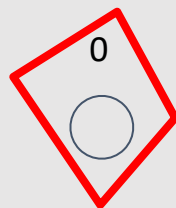
"Used" - set of vertices containing the vertices already in the spanning tree.

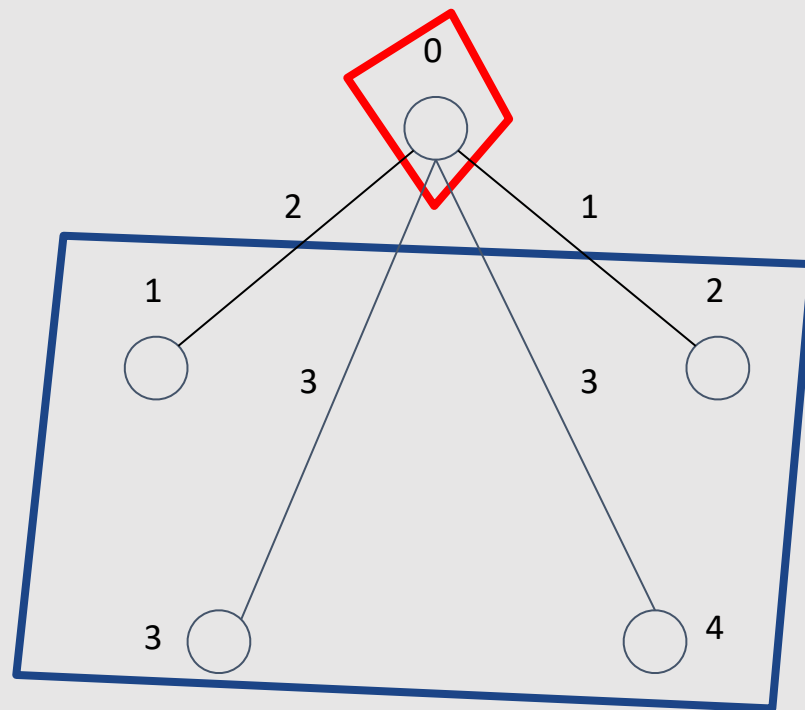
"Not used" - set of vertices not yet included in the spanning tree.

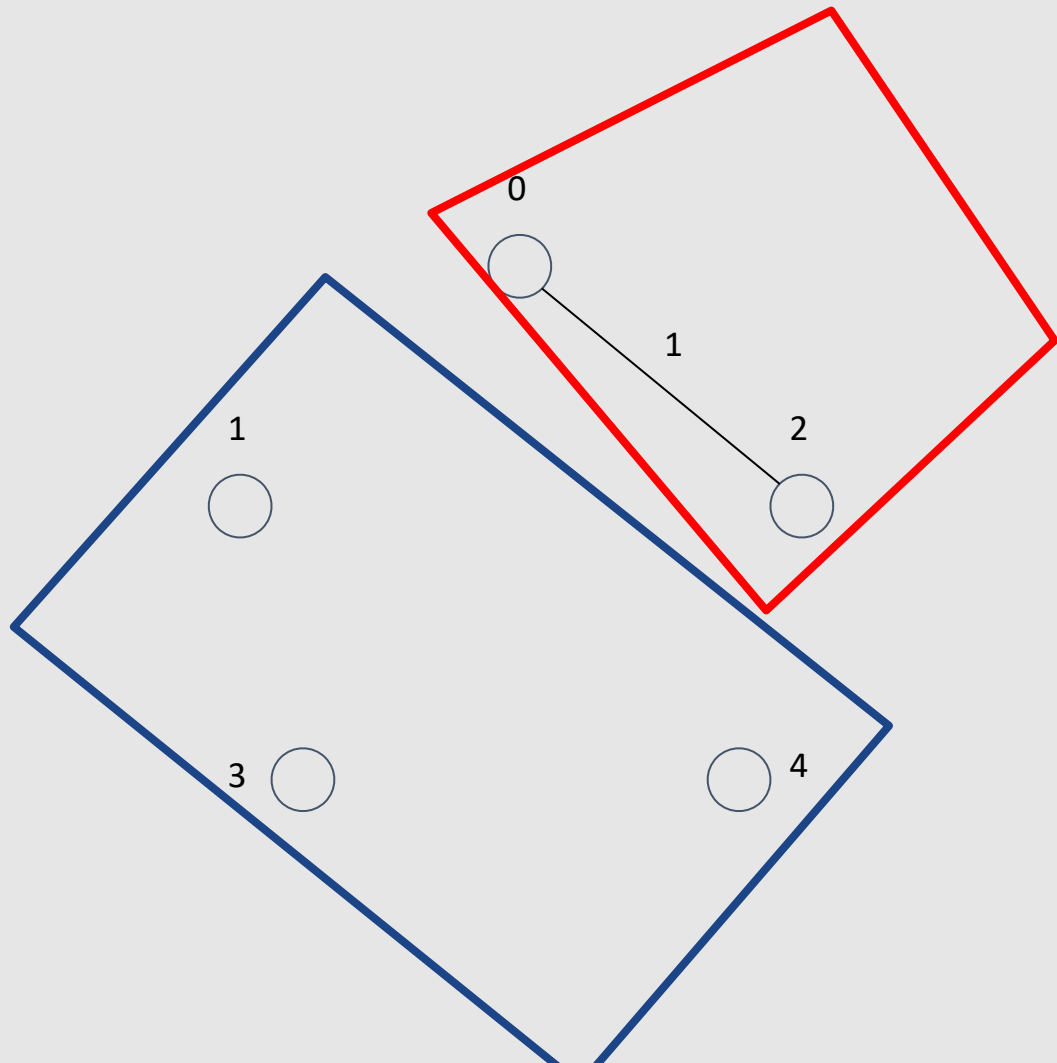
Step

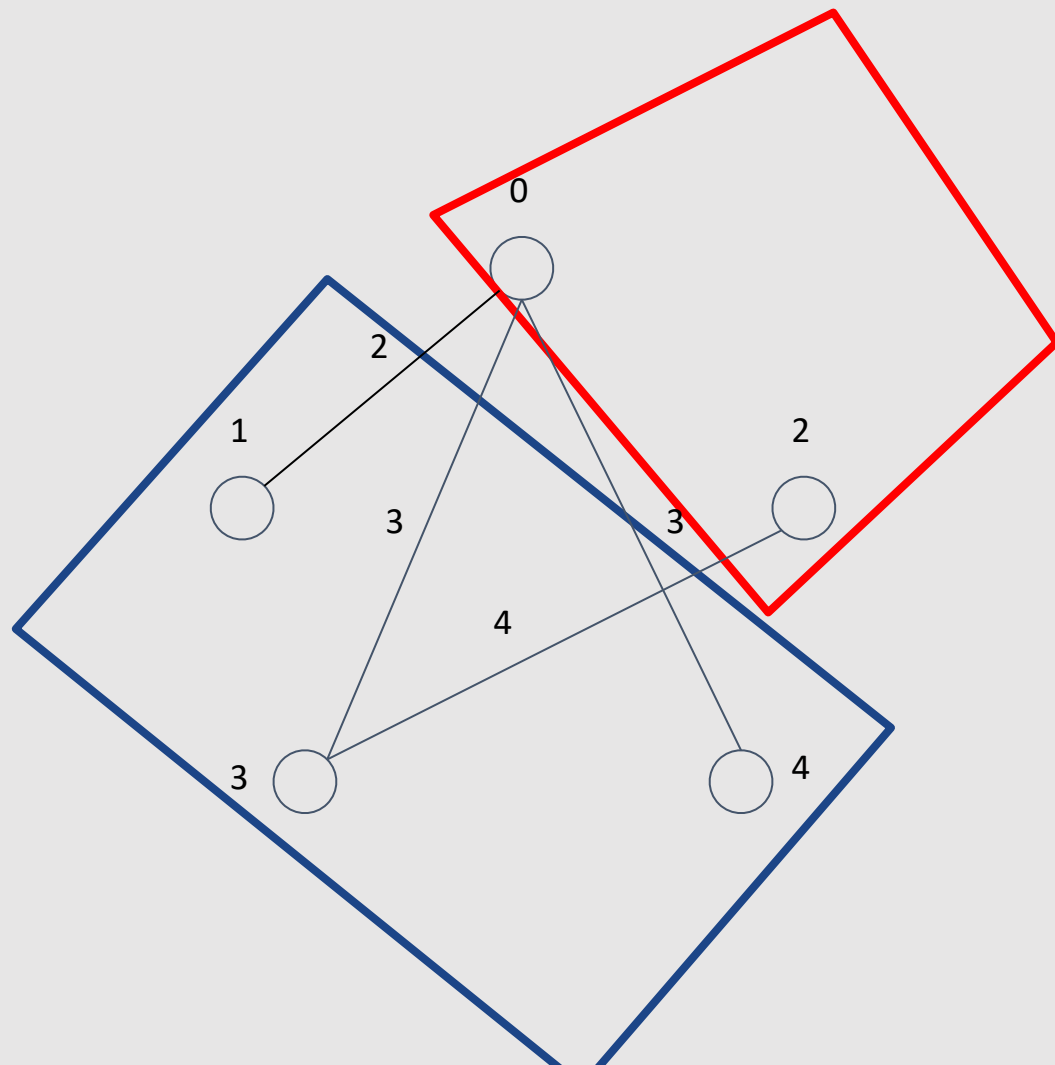
At each step, we will take the minimum edge from "used" to "not used" and add this edge and its corresponding vertex to the spanning tree.

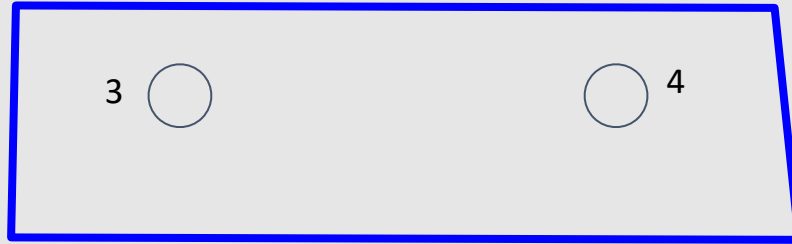
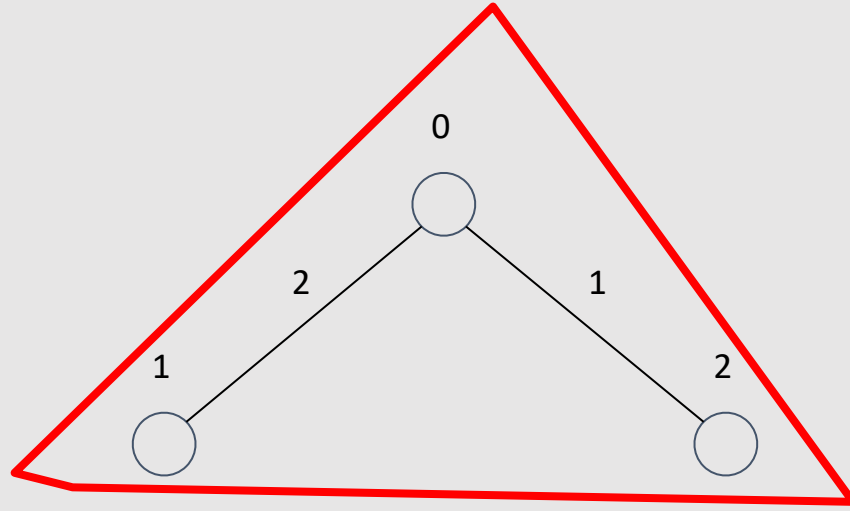


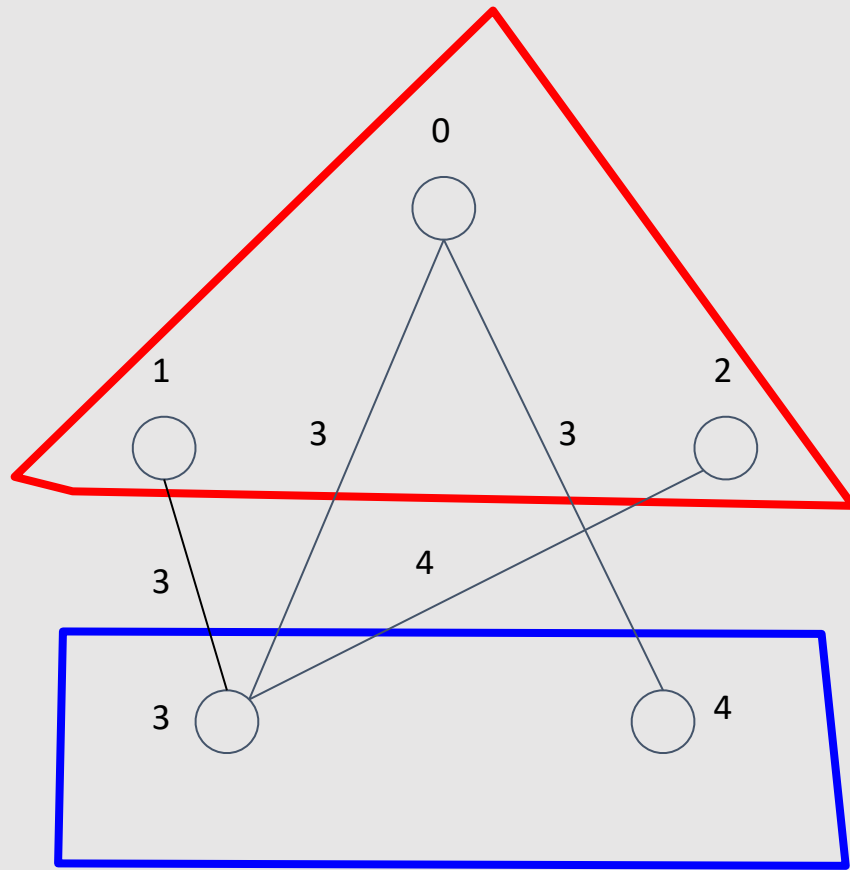


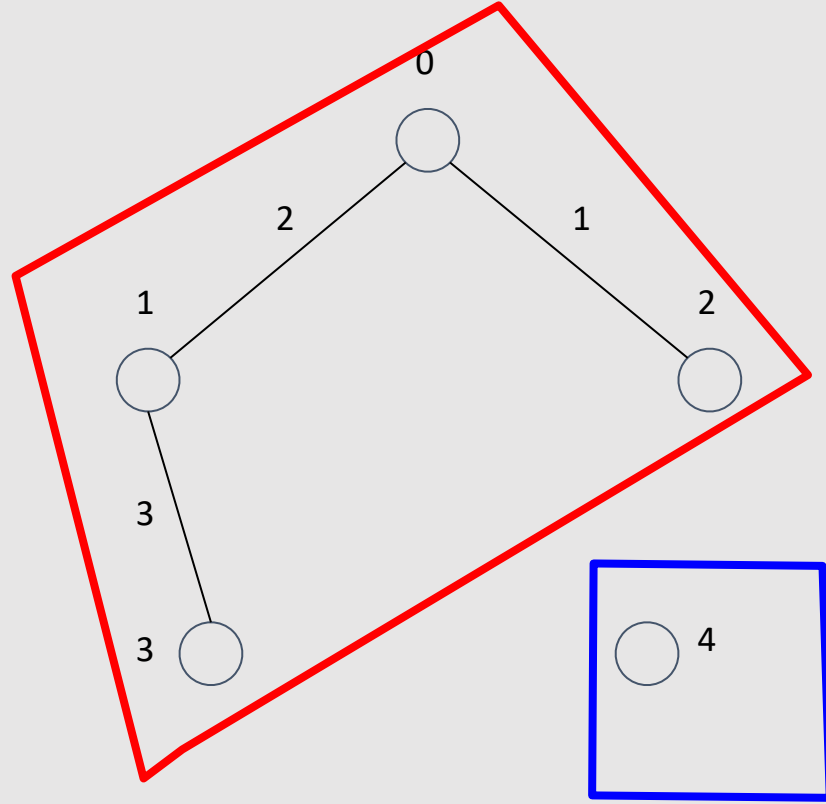


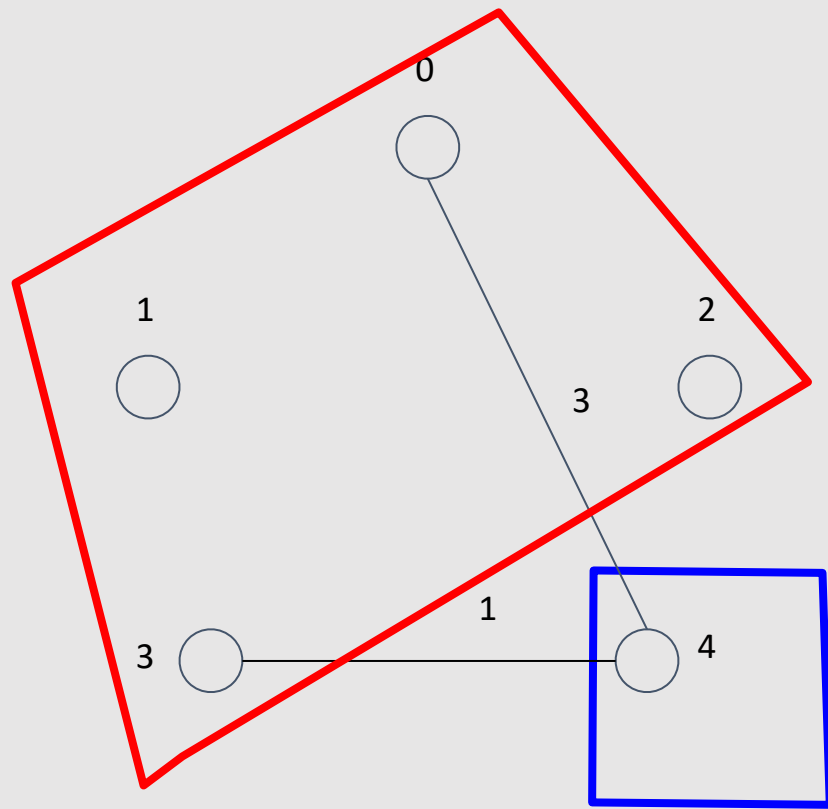


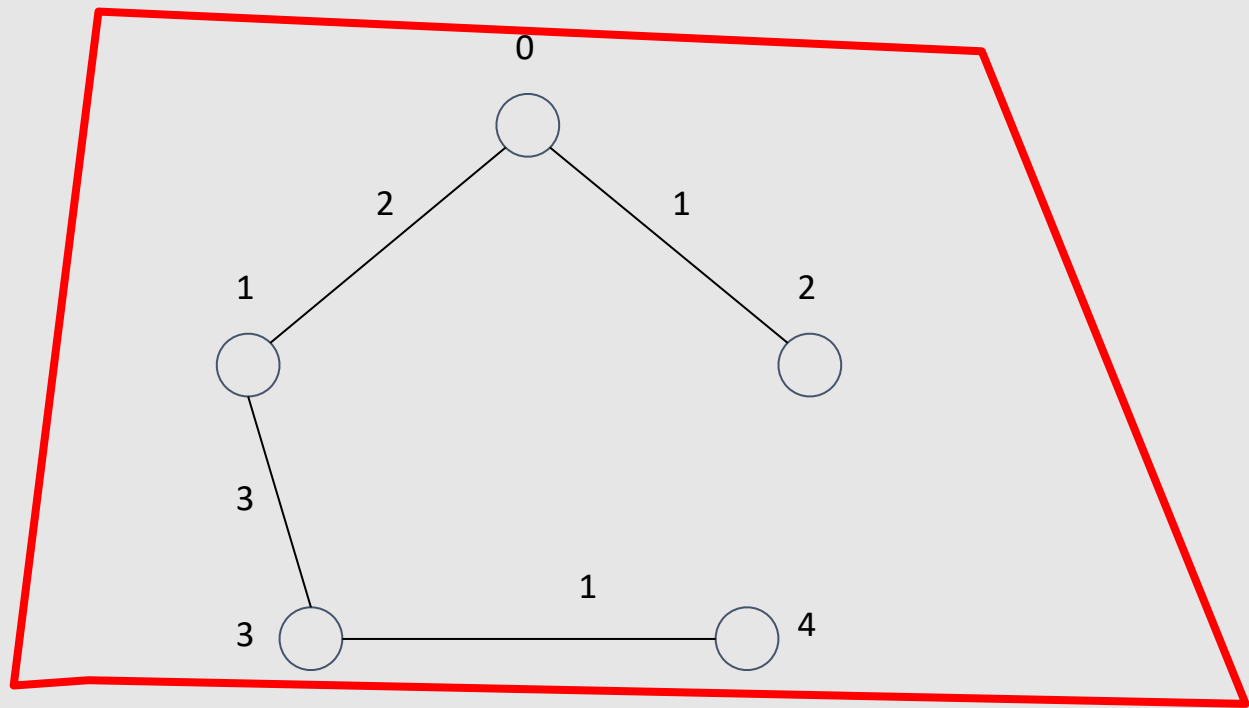












Algo

But how do we find this minimum edge?

The logic will be very similar to Dijkstra's algorithm.

Algo

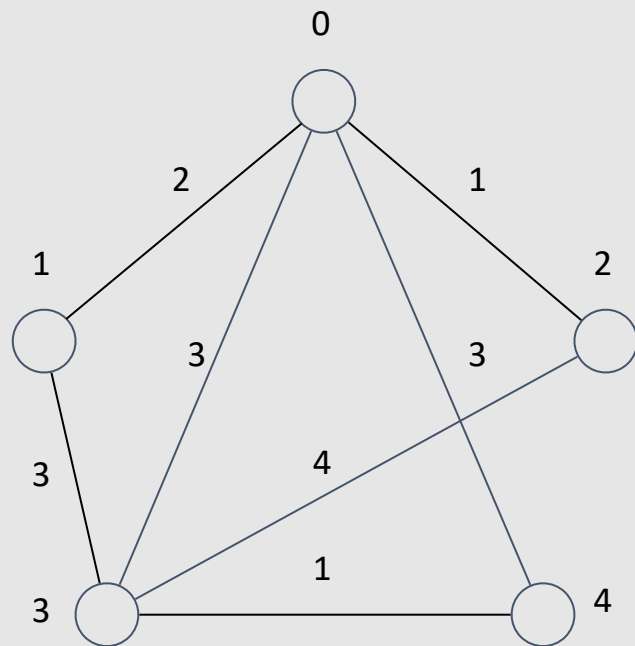
Let $\text{dist}[i]$ be the minimum edge from "used" to vertex i .

Then at each step, we need to select the vertex with the minimum $\text{dist}[i]$, such that i belongs to "not used".

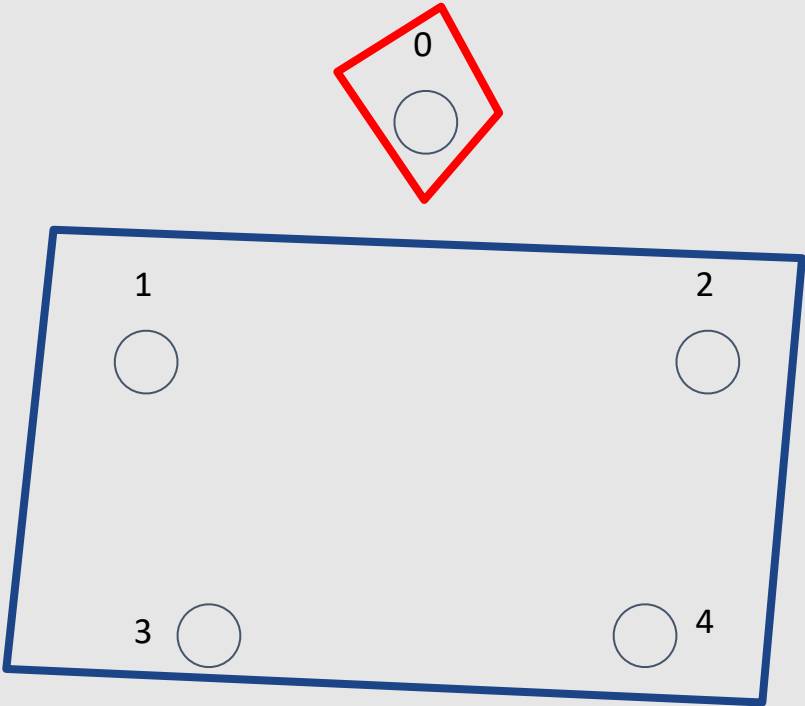
Algo

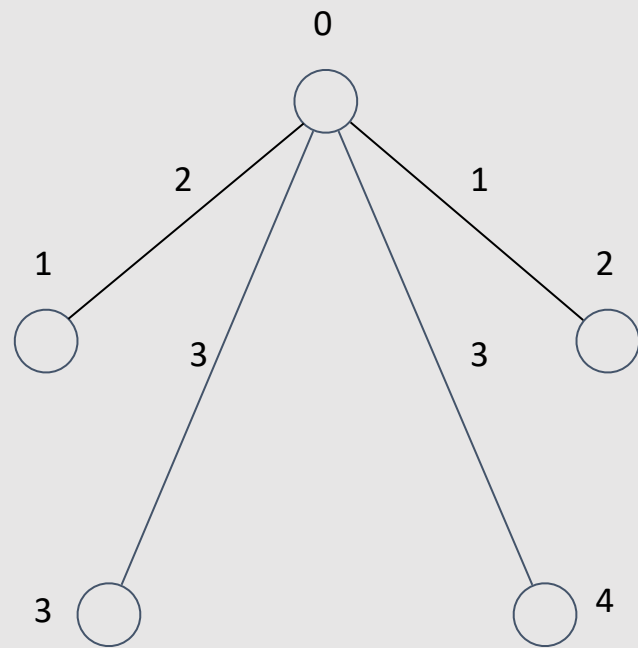
Let's implement this with a quadratic time approach, as we discussed earlier.

We will maintain an array 'dist,' and in each step, we will efficiently find the minimum element in linear time.

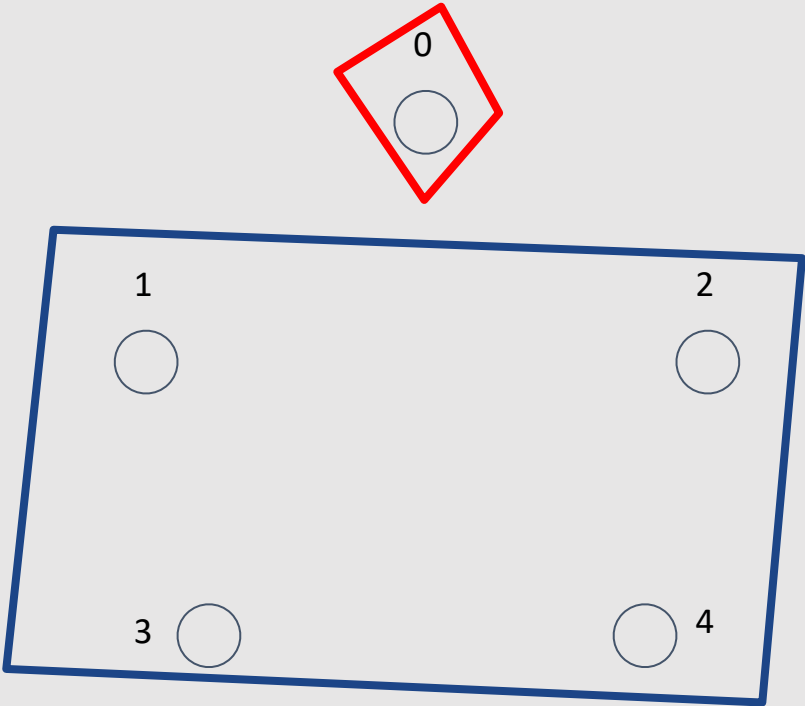


dist = [0, inf, inf, inf, inf]

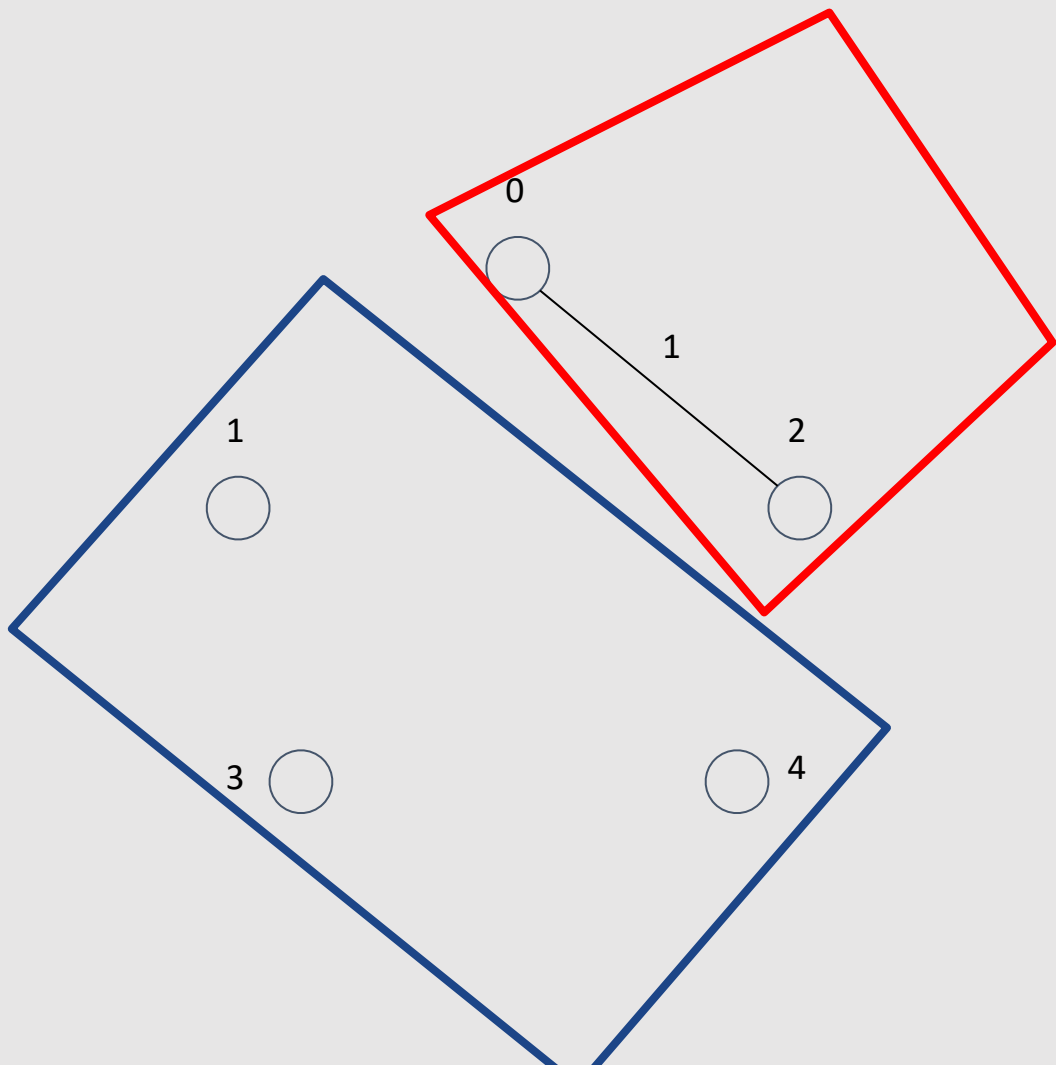


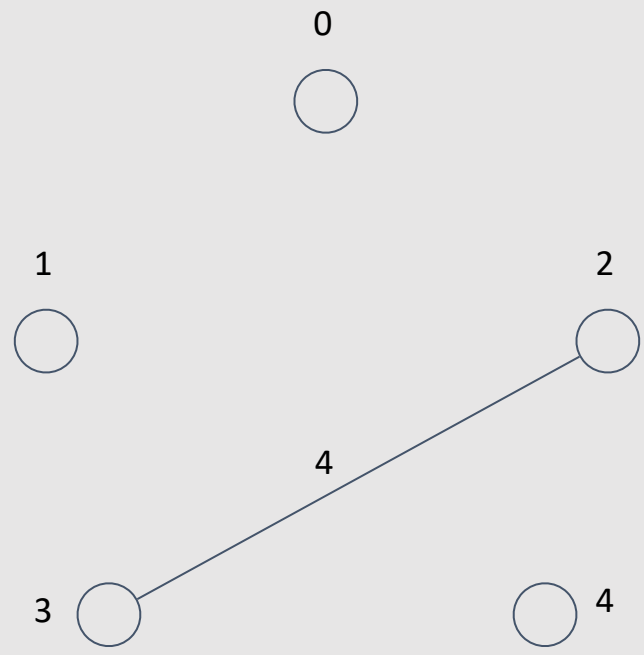


dist = [0, 2, 1, 3, 3]

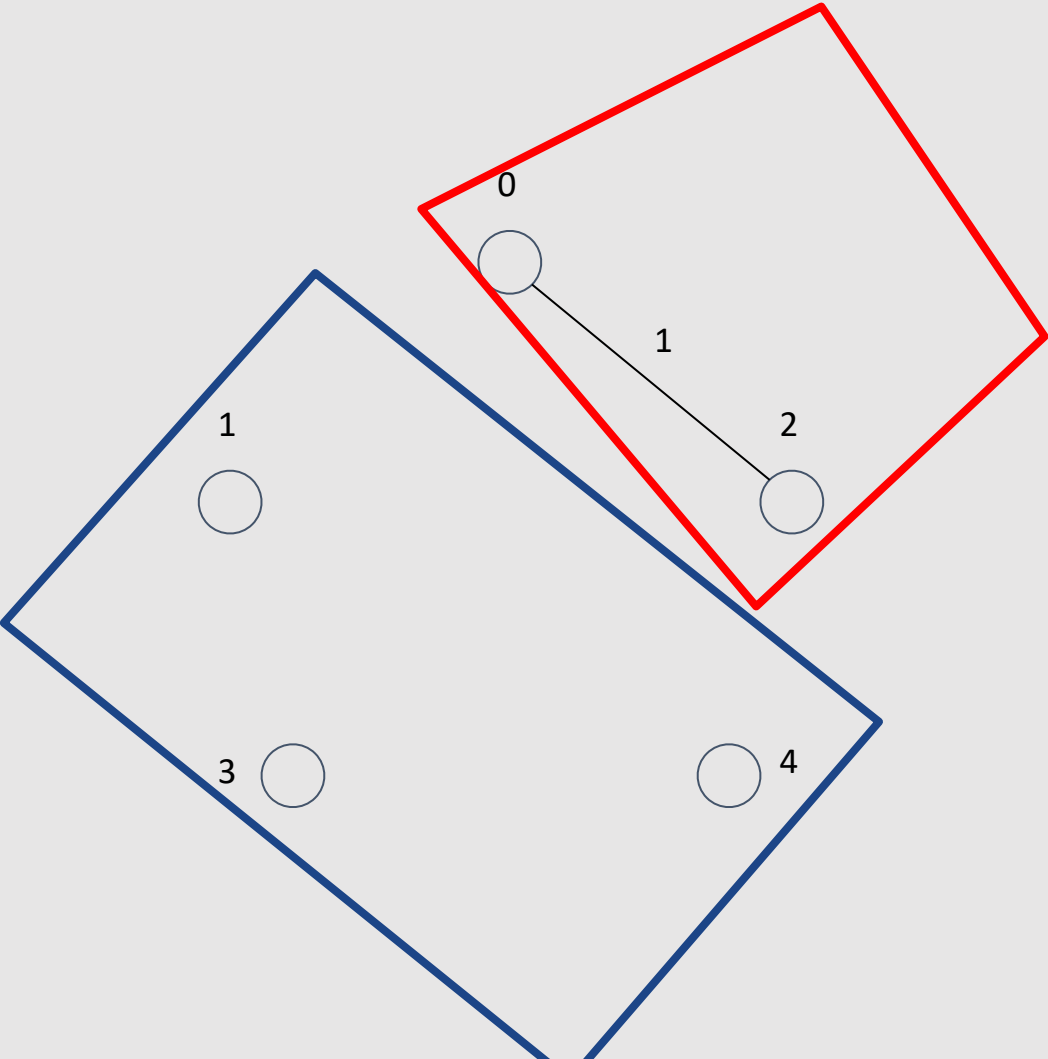


dist = [0, 2, 1, 3, 3]

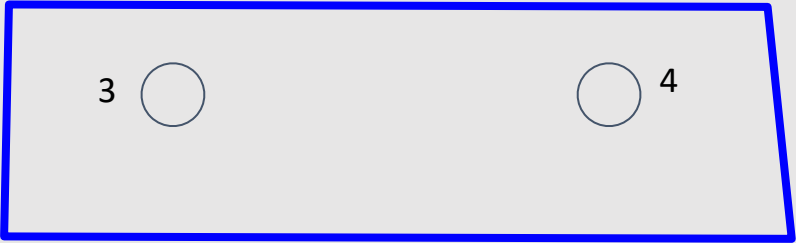
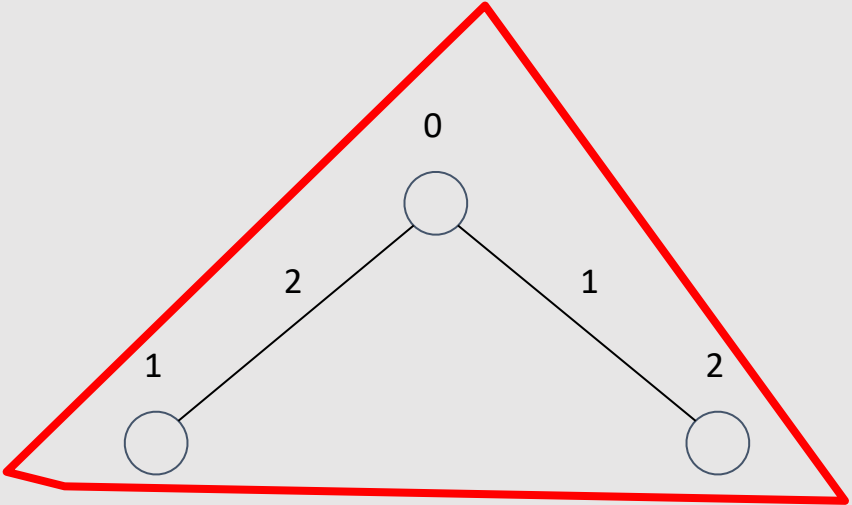




dist = [0, 2, 1, 3, 3]



dist = [0, 2, 1, 3, 3]



0

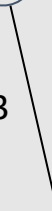


1



3

3



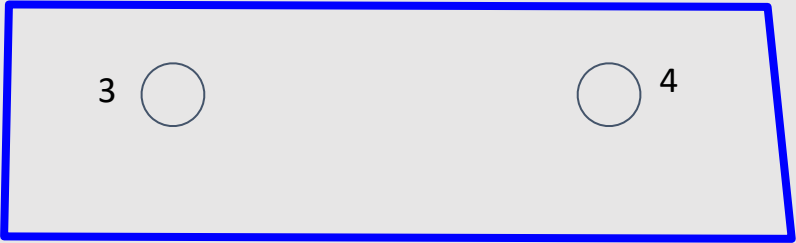
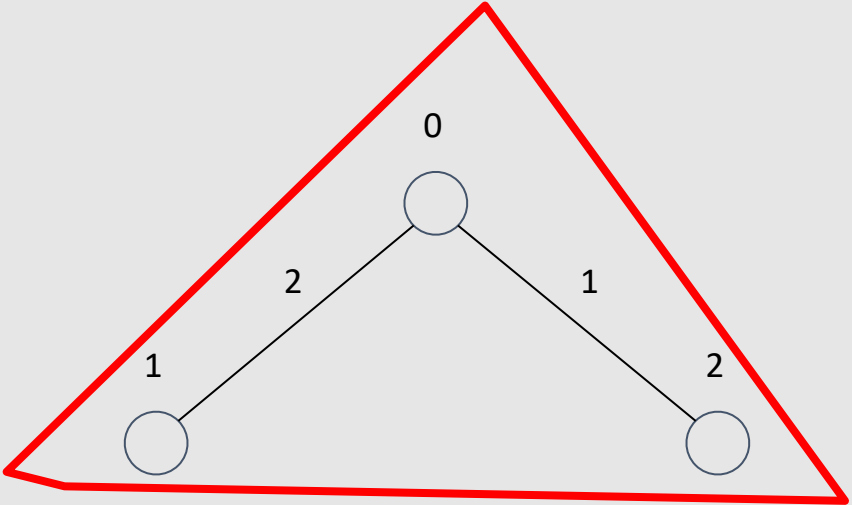
2



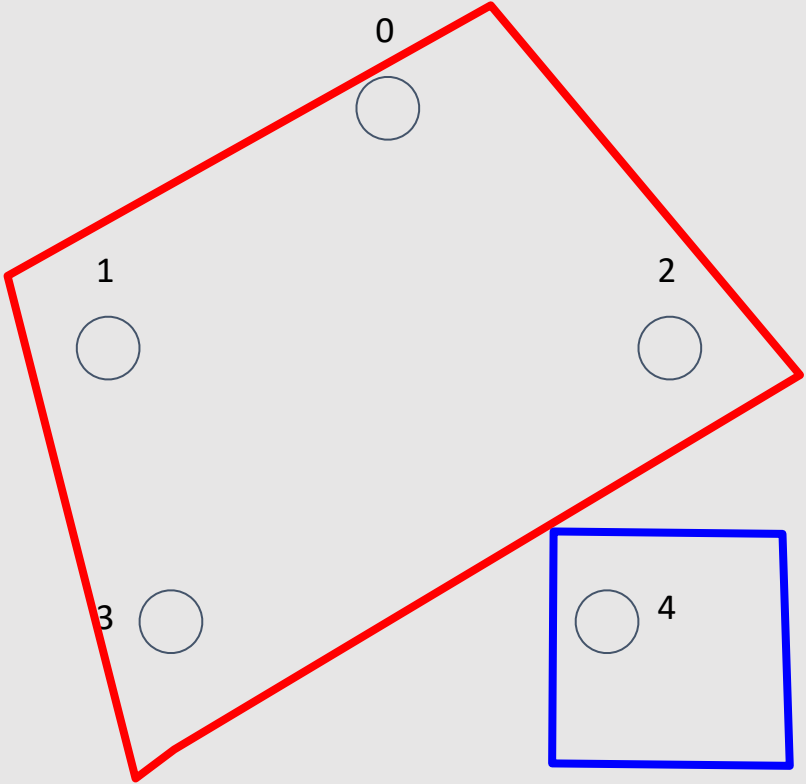
4

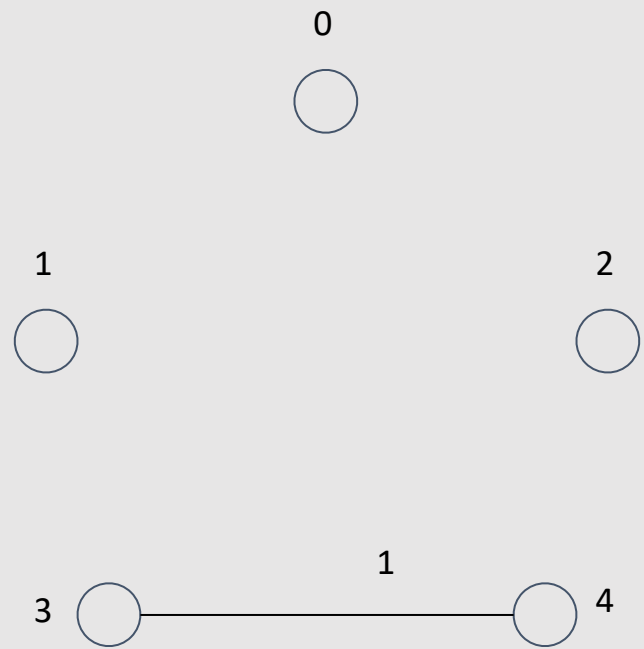


dist = [0, 2, 1, 3, 3]

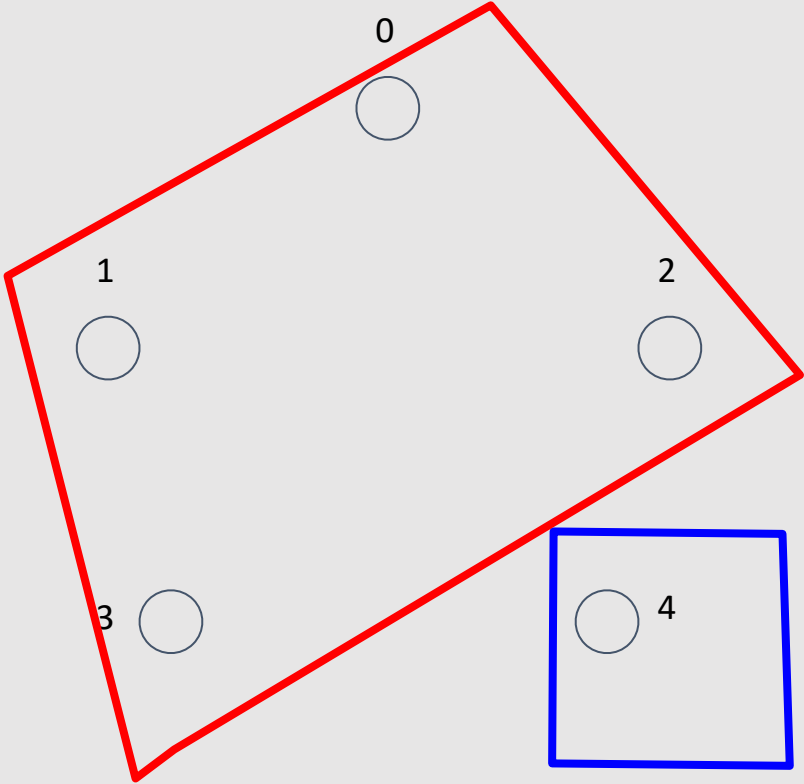


dist = [0, 2, 1, 3, 3]





dist = [0, 2, 1, 3, 1]



```
6.  const int INF = 1e9;
7.
8.  int findMinVertex(int n, vector<int>& key, vector<bool>& mstSet) {
9.      int minKey = INF;
10.     int minIndex = -1;
11.
12.     for (int v = 0; v < n; ++v) {
13.         if (!mstSet[v] && key[v] < minKey) {
14.             minKey = key[v];
15.             minIndex = v;
16.         }
17.     }
18.     return minIndex;
19. }
```

```

21. void primMST(vector<vector<int>>& graph) {
22.     int n = graph.size();
23.     vector<int> parent(n);
24.     vector<int> key(n);
25.     vector<bool> mstSet(n, false);
26.
27.     for (int i = 0; i < n; ++i) {
28.         key[i] = INF;
29.     }
30.
31.     key[0] = 0;
32.     parent[0] = -1;
33.
34.     for (int count = 0; count < n - 1; ++count) {
35.         int u = findMinVertex(n, key, mstSet);
36.         mstSet[u] = true;
37.
38.         for (int v = 0; v < n; ++v) {
39.             if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v]) {
40.                 parent[v] = u;
41.                 key[v] = graph[u][v];
42.             }
43.         }
44.     }
45.
46.     cout << "Edges:\n";
47.     for (int i = 1; i < n; ++i) {
48.         cout << parent[i] << " - " << i << " : " << graph[i][parent[i]] << endl;
49.     }
50. }

```

```
52. int main() {
53.     vector<vector<int>> graph = {
54.         {0, 2, 1, 3, 3},
55.         {2, 0, 0, 3, 0},
56.         {1, 0, 0, 3, 0},
57.         {3, 3, 3, 0, 1},
58.         {3, 0, 0, 1, 0}
59.     };
60.
61.     primMST(graph);
62.
63.     return 0;
64. }
65.
```

Success #stdin #stdout 0s 5400KB

 stdin

Standard input is empty

 stdout

Edges:

0 - 1 : 2

0 - 2 : 1

0 - 3 : 3

3 - 4 : 1

Algo

Just like in Dijkstra's algorithm, instead of searching for the minimum in the array, we can search for the minimum using a priority queue.

```

7. void primMST(vector<vector<pair<int, int>>>& graph) {
8.     int n = graph.size();
9.     vector<int> parent(n, -1);
10.    vector<int> key(n, INF);
11.    vector<bool> inMST(n, false);
12.
13.    // priority queue (weight, vertex)
14.    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
15.
16.    pq.push({0, 0});
17.    key[0] = 0;
18.
19.    while (!pq.empty()) {
20.        int u = pq.top().second;
21.        pq.pop();
22.        inMST[u] = true;
23.
24.        for (auto& neighbor : graph[u]) {
25.            int v = neighbor.first;
26.            int weight = neighbor.second;
27.
28.            if (!inMST[v] && weight < key[v]) {
29.                parent[v] = u;
30.                key[v] = weight;
31.                pq.push({key[v], v});
32.            }
33.        }
34.    }
35.
36.    cout << "Edges:\n";
37.    for (int i = 1; i < n; ++i) {
38.        cout << parent[i] << " - " << i << " : " << key[i] << endl;
39.    }
40. }

```

```
42. int main() {
43.     vector<vector<pair<int, int>>> graph = {
44.         {{1, 2}, {2, 1}, {3, 3}, {4, 3}},
45.         {{0, 2}, {3, 3}},
46.         {{0, 1}, {3, 3}},
47.         {{0, 3}, {1, 3}, {2, 3}, {4, 1}},
48.         {{0, 3}, {3, 1}}
49.     };
50.
51.     primMST(graph);
52.
53.     return 0;
54. }
55.
```

Success #stdin #stdout 0.01s 5356KB

 stdin

Standard input is empty

 stdout

Edges:

0 - 1 : 2

0 - 2 : 1

0 - 3 : 3

3 - 4 : 1

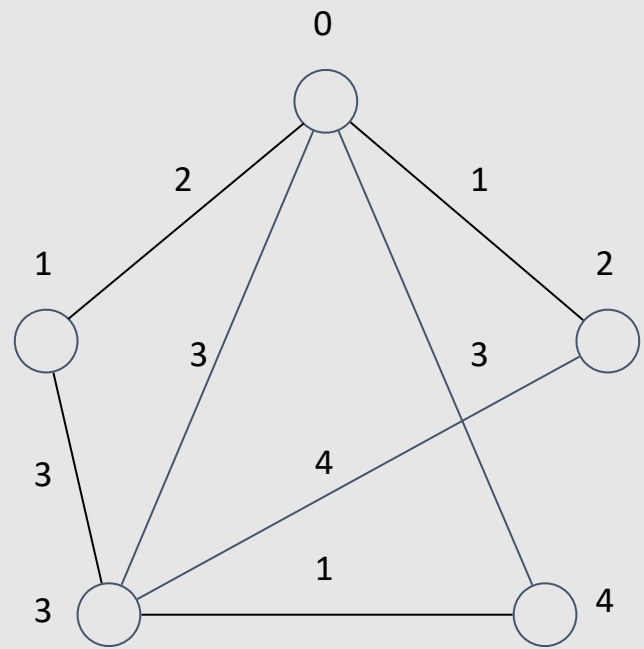
Kruskal's algorithm

Overall, the algorithm looks as follows:

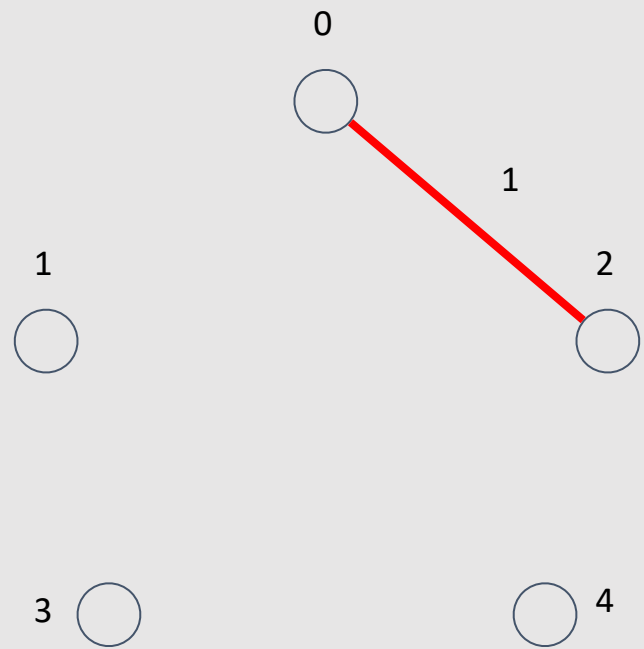
We traverse the list of edges, sorted in ascending order by weight.

Each time, we check whether the edge passes through any cut (if it connects components that are not yet merged).

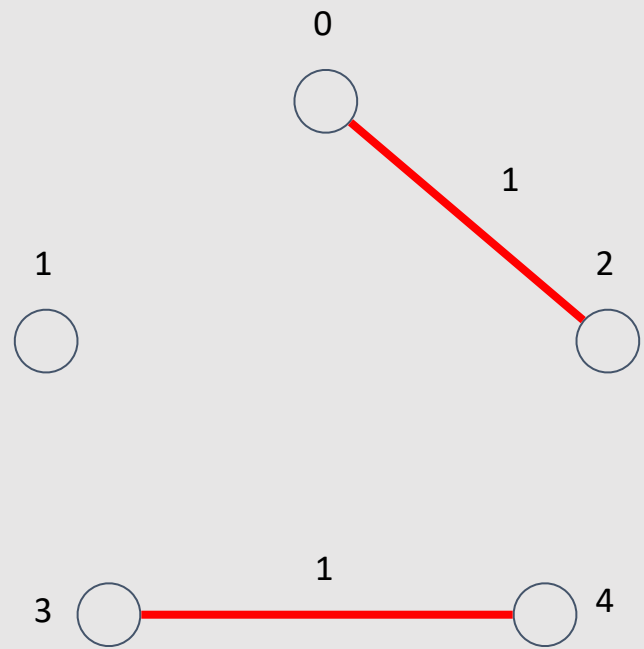
{from, to, weight} = {{0, 2, 1}, {3, 4, 1}, {0, 1, 2}, {1, 3, 3}, {0, 3, 3}, {0, 4, 3}, {2, 3, 4}}



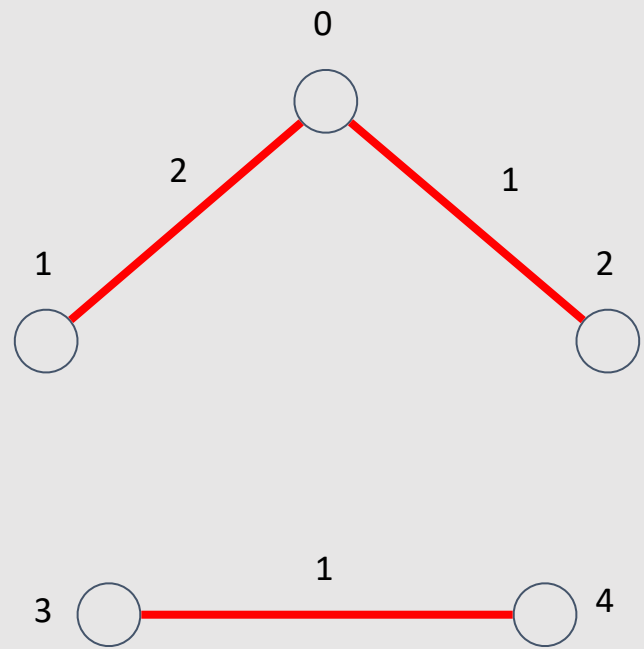
{from, to, weight} = {{0, 2, 1}, {3, 4, 1}, {0, 1, 2}, {1, 3, 3}, {0, 3, 3}, {0, 4, 3}, {2, 3, 4}}



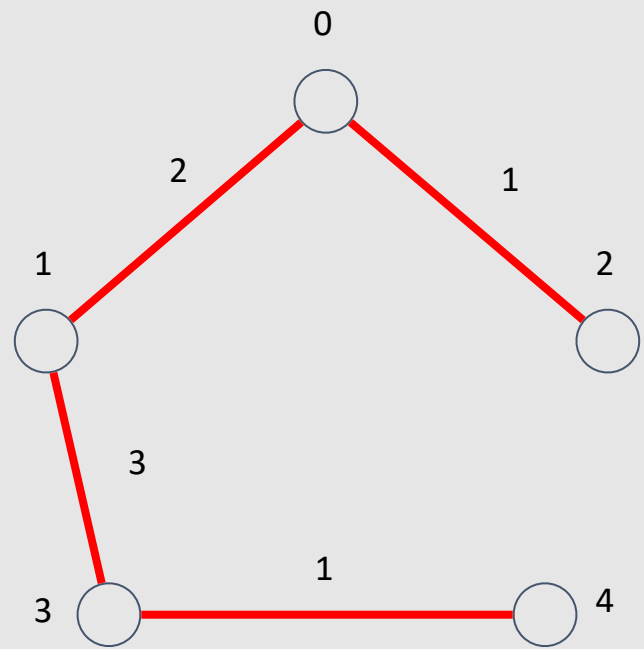
{from, to, weight} = {{0, 2, 1}, {3, 4, 1}, {0, 1, 2}, {1, 3, 3}, {0, 3, 3}, {0, 4, 3}, {2, 3, 4}}



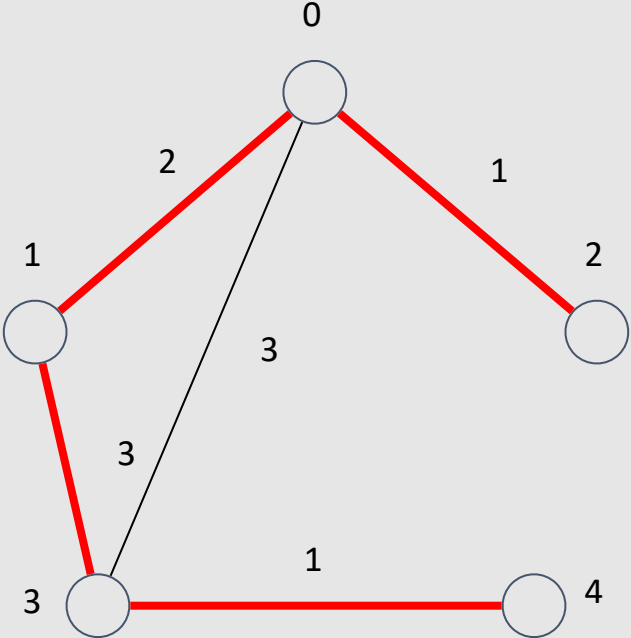
{from, to, weight} = {{0, 2, 1}, {3, 4, 1}, {0, 1, 2}, {1, 3, 3}, {0, 3, 3}, {0, 4, 3}, {2, 3, 4}}



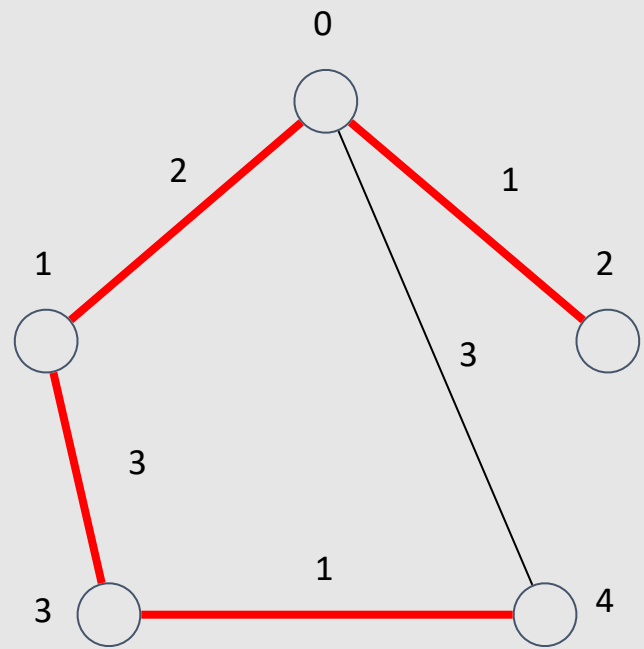
{from, to, weight} = {{0, 2, 1}, {3, 4, 1}, {0, 1, 2}, {1, 3, 3}, {0, 3, 3}, {0, 4, 3}, {2, 3, 4}}



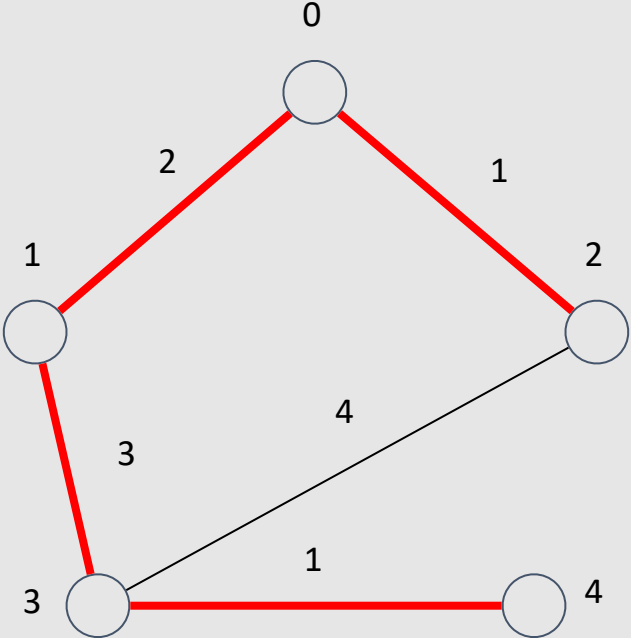
{from, to, weight} = {{0, 2, 1}, {3, 4, 1}, {0, 1, 2}, {1, 3, 3}, {0, 3, 3}, {0, 4, 3}, {2, 3, 4}}



{from, to, weight} = {{0, 2, 1}, {3, 4, 1}, {0, 1, 2}, {1, 3, 3}, {0, 3, 3}, {0, 4, 3}, {2, 3, 4}}



{from, to, weight} = {{0, 2, 1}, {3, 4, 1}, {0, 1, 2}, {1, 3, 3}, {0, 3, 3}, {0, 4, 3}, {2, 3, 4}}



Kruskal's algorithm

Essentially, it is clear how to implement this quite easily. We can simply check connectivity using Depth-First Search (DFS), but such an algorithm would operate in $O(NM)$ time. How can we make it faster? This is where a data structure called the Disjoint Set Union (DSU) comes to our help.

DSU

DSU - Disjoint Set Union, a structure required to merge sets and check whether two elements belong to the same set.

DSU

We can think of DSU as an array where each element holds the index of some element in the same set as itself.

Additionally, we assume that one element always holds its own index.

sets

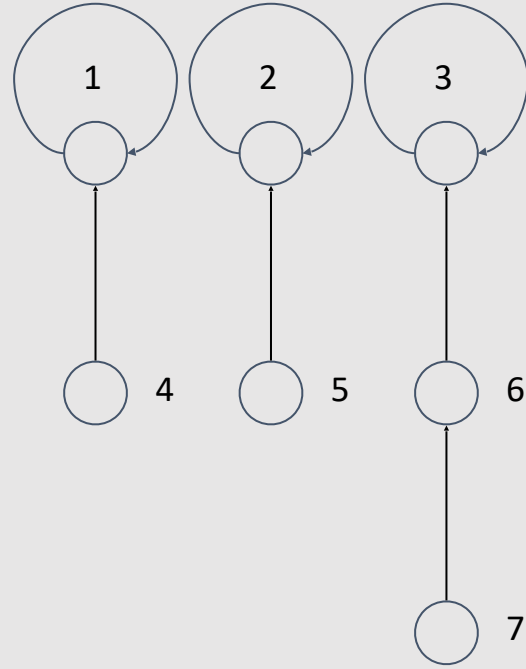
1	2	3	4	5	6	7
---	---	---	---	---	---	---

DSU

DSU can be represented as trees, where the set number is the number of the tree's root.

dsu

1	2	3	1	2	3	6
---	---	---	---	---	---	---

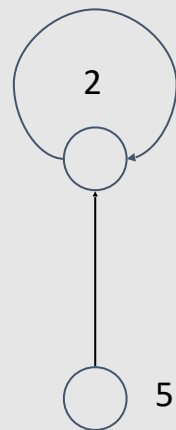
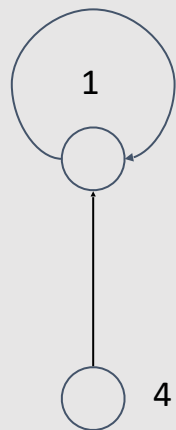


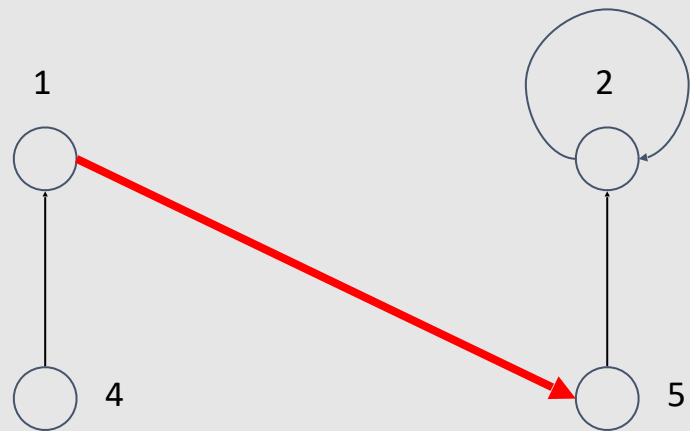
DSU

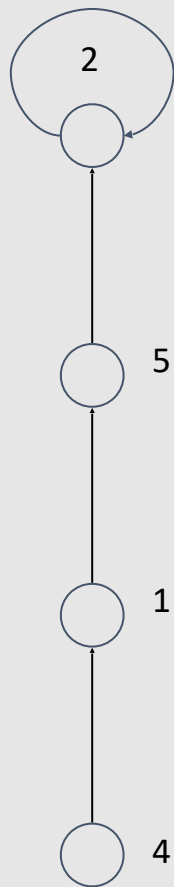
A naive implementation can be done as follows:

When two sets are merged into one, simply assign the index of one of them to be connected to the other.

Let's call this operation "SetUnion".



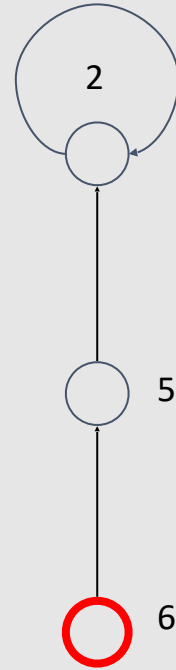
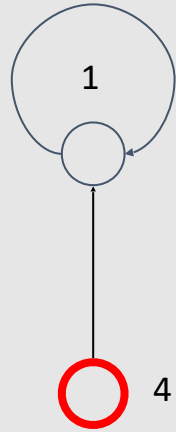


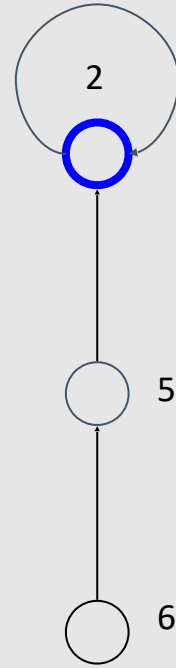
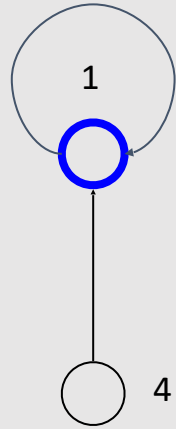


DSU

Now, let's learn how to check whether two elements belong to the same set.

First, we find the root for each of them. Let's call this operation "SetFind". Then, we compare the roots of the sets these two elements belong to.





```
1.  
5.  int SetFind(const vector<int> &dsu, int u) {  
6.      if (u == dsu[u]) {  
7.          return u;  
8.      }  
9.      else {  
10.         return SetFind(dsu, dsu[u]);  
11.     }  
12. }  
13.  
14. void SetUnite(vector<int> &dsu, int a, int b) {  
15.     dsu[a] = b;  
16. }
```

```

18. int main() {
19.     int n, m;
20.     cin >> n >> m;
21.     vector<int> dsu(n);
22.
23.     for (int i = 0; i < n; i++) {
24.         dsu[i] = i; // create new set from vertex i
25.     }
26.
27.     for (int i = 0; i < m; i++) {
28.         int a, b;
29.         cin >> a >> b;
30.         a--;
31.         b--;
32.         SetUnite(dsu, a, b);
33.     }
34.
35.     for (int i = 0; i < n; i++) {
36.         cout << SetFind(dsu, i) << " ";
37.     }
38.     return 0;
39. }

```

Success #stdin #stdout 0.01s 5284KB

 stdin

6 4
1 2
2 3
4 5
3 6

 stdout

5 5 5 4 4 5

DSU

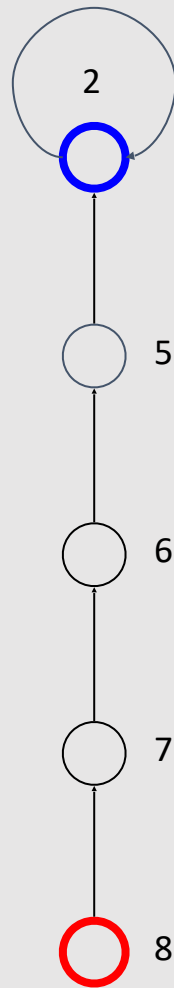
This implementation will still work in linear time for a chain-like structure.

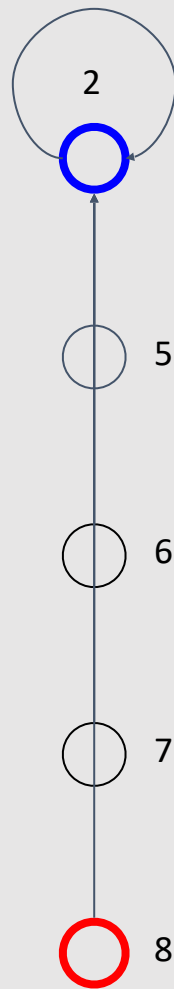
Can we do better?

Path Compression

Firstly, let's make SetFind better.

Instead of traversing upwards each time, we can store the previously calculated results while moving upwards.





Faster

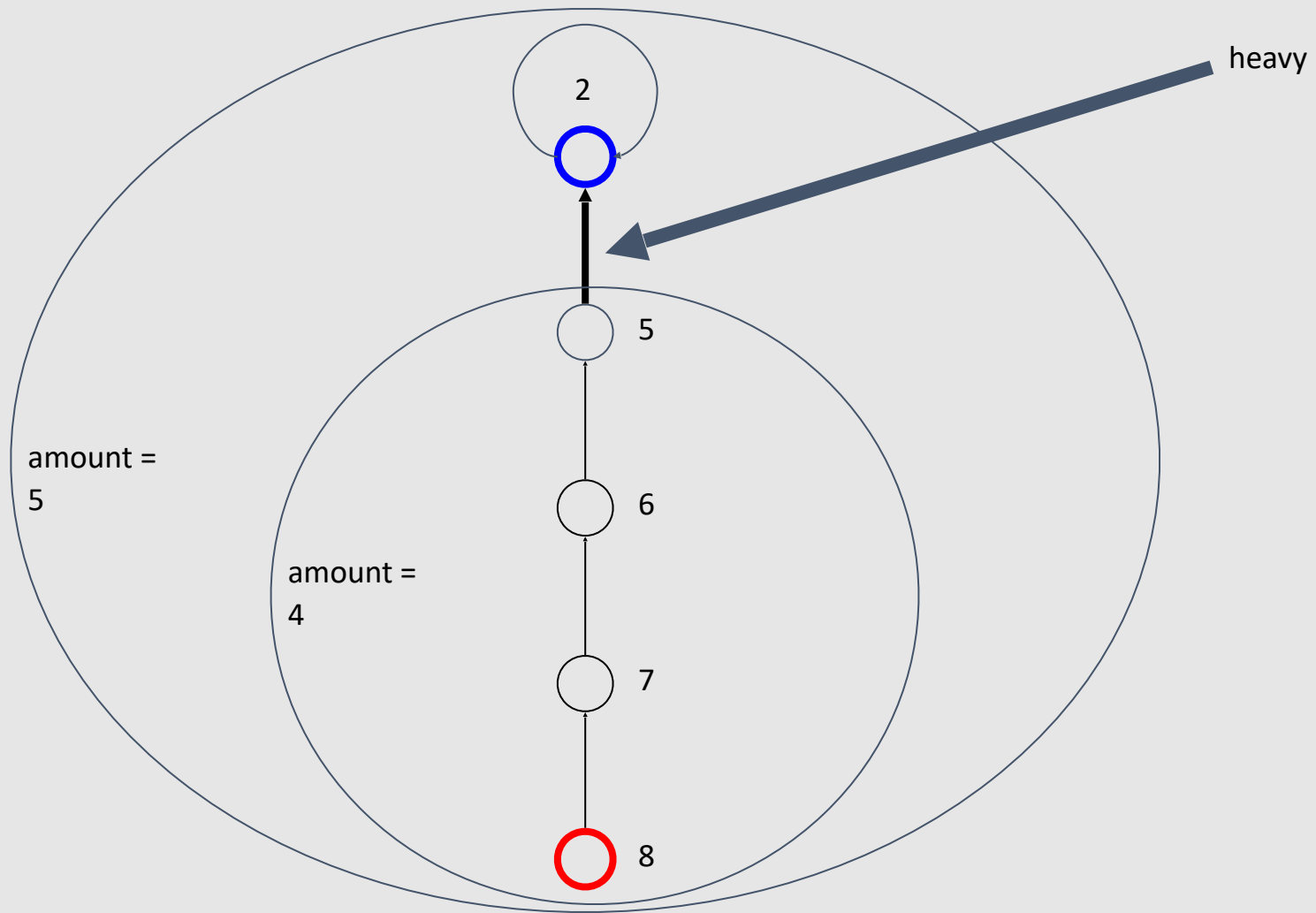
Why would this speed up the process?

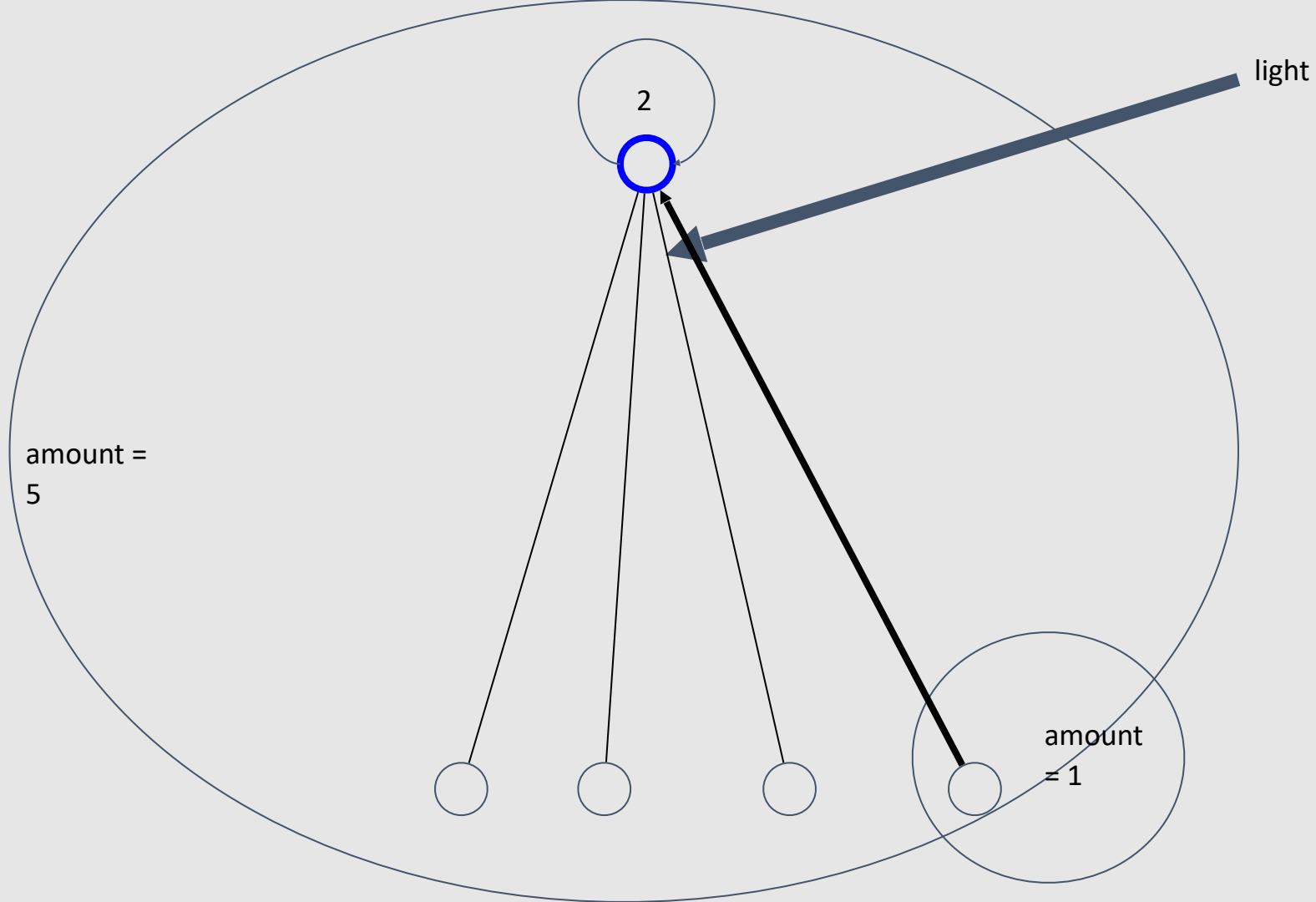
Path Compression

Let's divide the path into light and heavy edges.

A light edge is one where the child is at least twice lighter (i.e., has at least half the number of children) compared to its ancestor.

A heavy edge is the opposite.





Path Compression

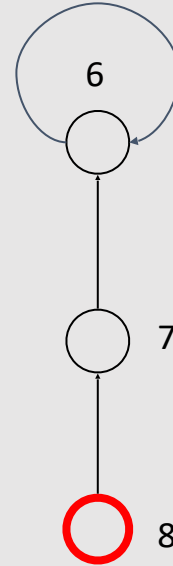
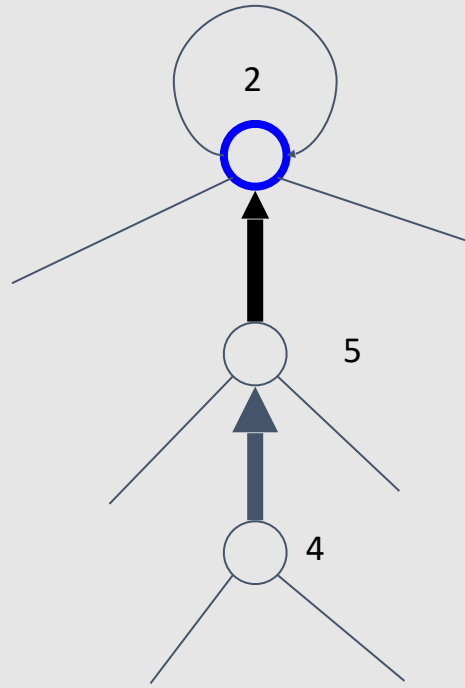
Statement 1: In any path, there cannot be more than $\log(n)$ light edges.

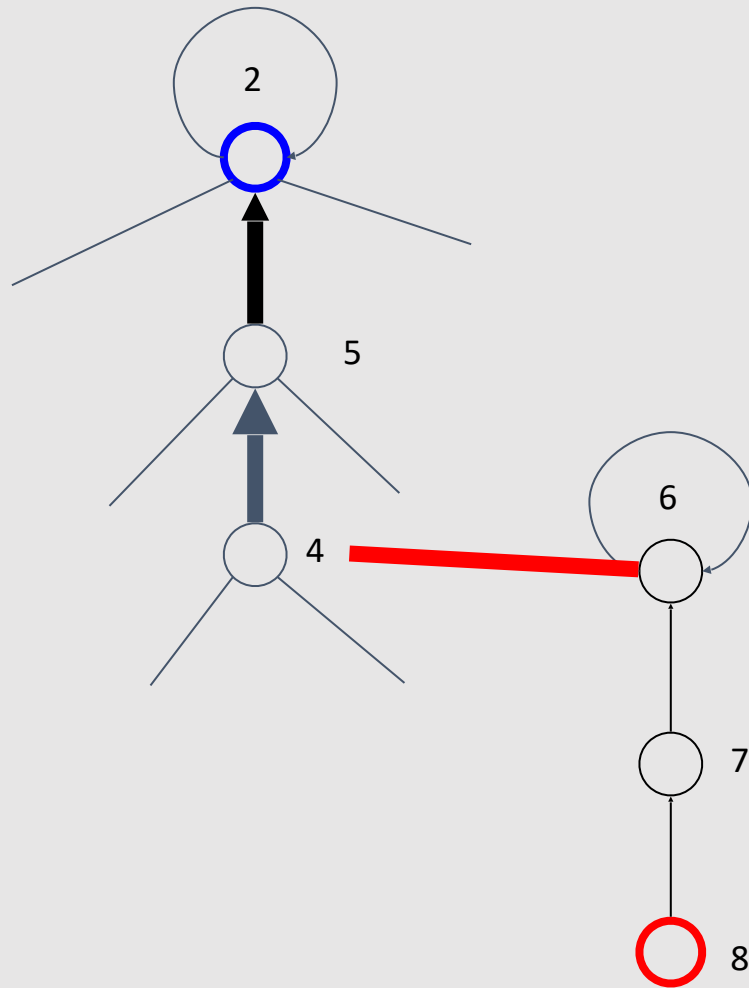
This is because each light edge reduces the number of vertices in the subtree by at least half, so from n vertices to 1, it will take a maximum of $\log(n)$ operations.

Path Compression

Statement 2: There can't be more than $O(n \log(n))$ heavy edges in total.

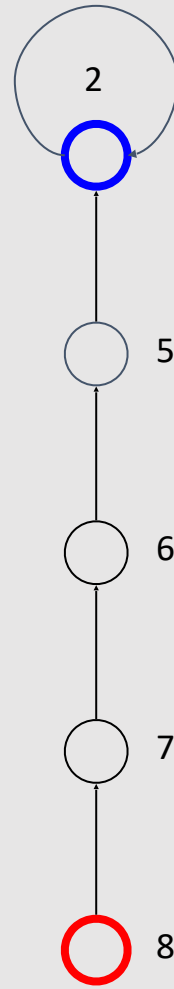
Each merge of two components can create no more than $\log(n)$ heavy edges because on a path, there can be no more than $\log(n)$ light edges, plus, after merging components, we might obtain 1 new edge.

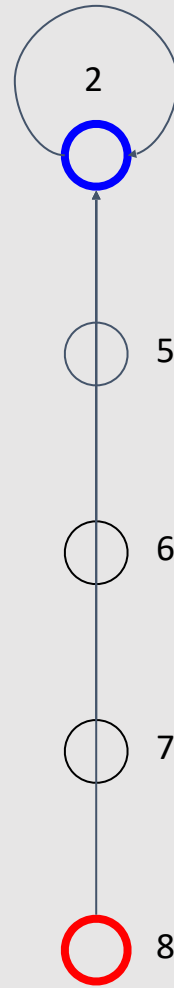




Path Compression

If we compress any heavy edge in path compression, it will inevitably become light, hence overall, we process no more than $n \log(n)$ heavy edges. This is how we achieve an amortized estimate - $\log(n)$ per query.





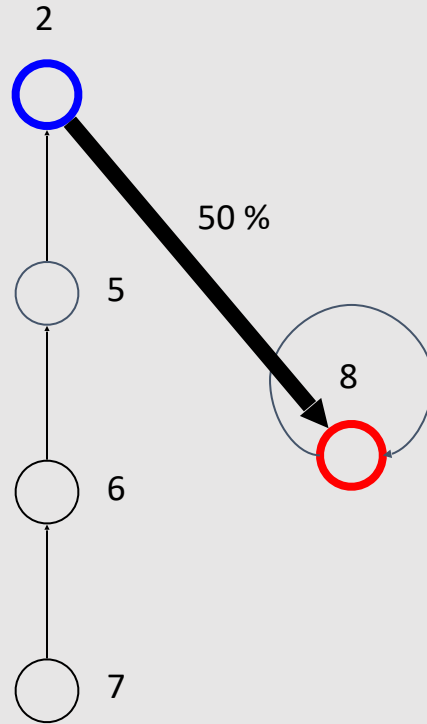
Random

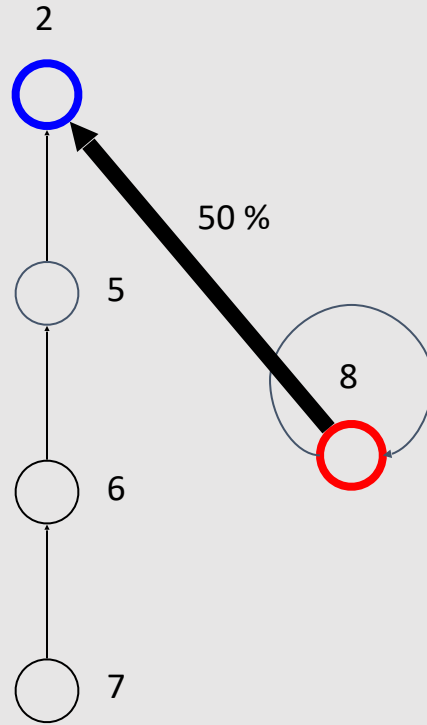
Now, let's work with SetUnion.

The first obvious idea is not to attach in some order, but randomly.

Random

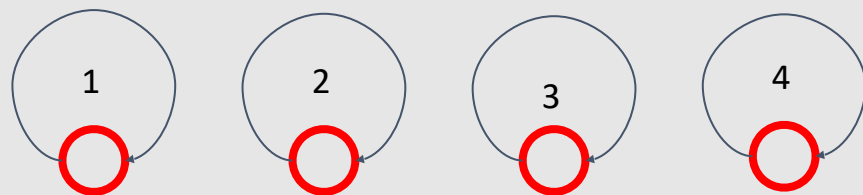
Consider a chain-like structure. The probability of attaching one vertex to the long chain is $\frac{1}{2}$.





Random

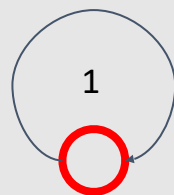
In this case, the expected value of the tree's size, if you attach a new vertex to the existing DSU each time, is $n / 2$. So such an optimization does not improve the asymptotic complexity.



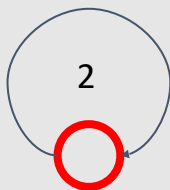
Rank heuristic

We'll maintain the rank for each vertex, which represents the height of its subtree.

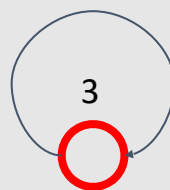
When merging trees, we'll choose the vertex with the higher rank as the root of the new tree and update the ranks (the leader's rank should increase by one if it matched the rank of the other vertex).



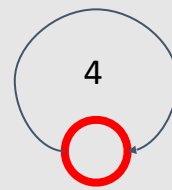
rank = 0



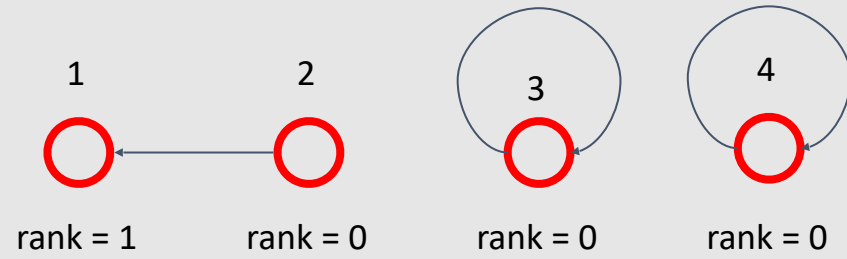
rank = 0

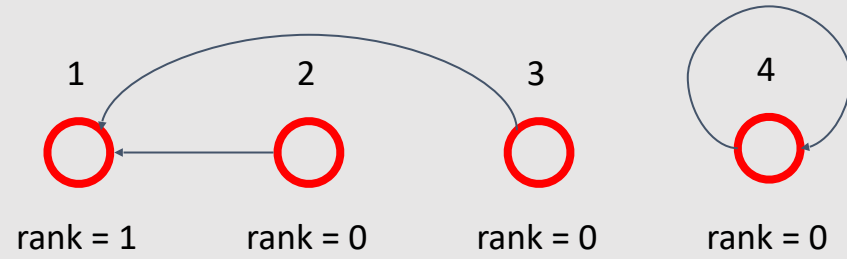


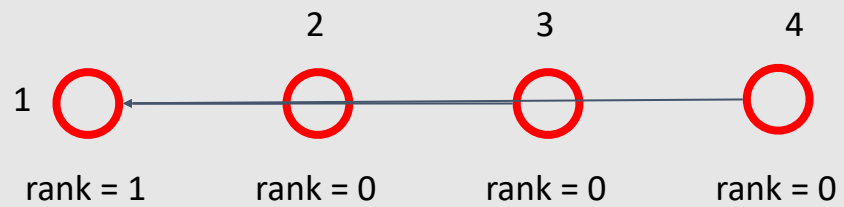
rank = 0

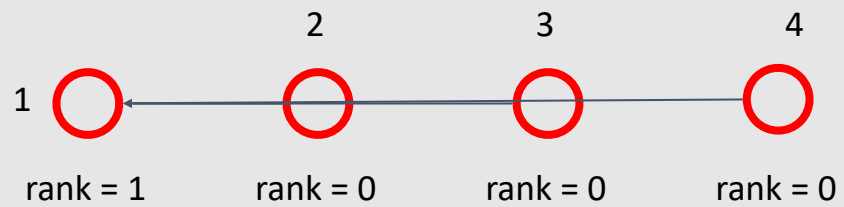


rank = 0









Asymptotic

One such heuristic also operates in $O(\log(n))$ time.

Proof

If set has rank k , it has at least 2^k elements.

Induction:

for $k = 0$: obvious.

$$1 \bigcirc$$

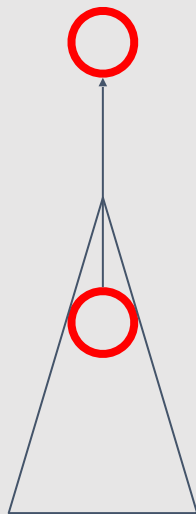
$k = 0$

Proof

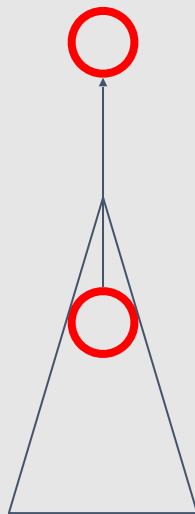
for k :

If we have rank k , then we merged at least 2 trees of rank $k-1$, they have at least $2^{(k-1)}$ elements, $\text{sum} = 2^{(k-1)} + 2^{(k-1)} = 2^k$

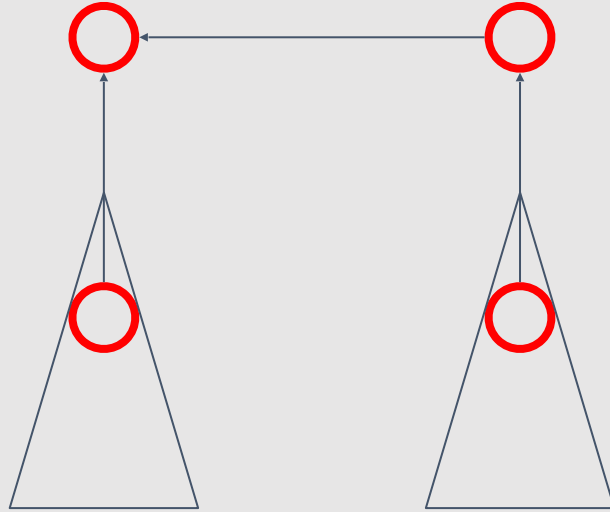
$$\geq 2^k$$



$$\geq 2^k$$

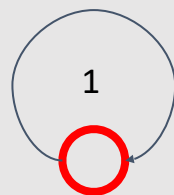


$$\geq 2^{k+1}$$

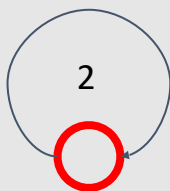


Size

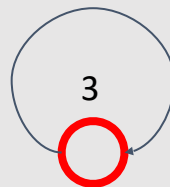
You can also compare by size instead of rank. You can attach the smaller-sized set to the larger-sized set. When merging sets, their sizes also merge. We won't use this heuristic in the code because it's more complex in practice than the previous one, but in theory, it works in the same way.



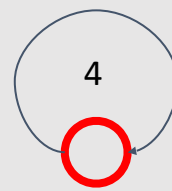
size = 1



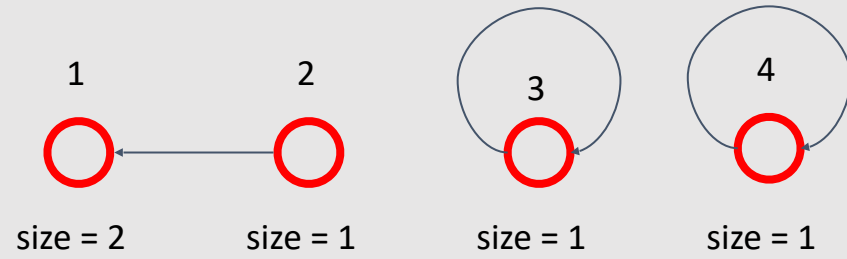
size = 1

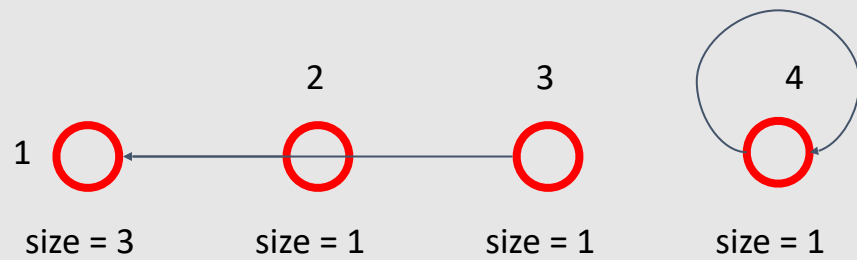


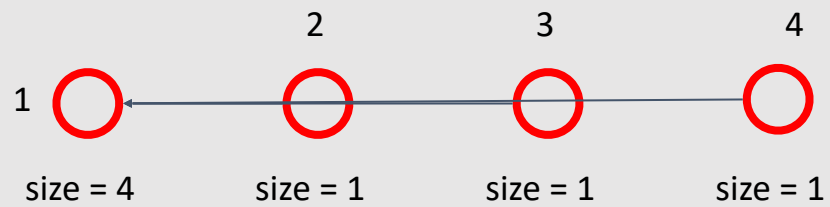
size = 1



size = 1







Size

The combined use of two heuristics (rank + path compression or size + path compression) gives us a complexity of $O(a)$ or $O(\log^*)$.

Size

$O(\log^*)$ = amount of times you need to take $\log(x)$ to get 1

Example $\log_2^*(4096) = 4$, $\log_2(\log_2(\log_2(\log_2(4096))))$

≤ 1

$O(a)$ is even less.


```
5.  int SetFind(vector<int> &dsu, int u) {  
6.      if (u == dsu[u]) {  
7.          return u;  
8.      }  
9.      else {  
10.         return dsu[u] = SetFind(dsu, dsu[u]);  
11.     }  
12. }
```

```
14. void SetUnite(vector<int> &dsu, vector<int> &rank, int a, int b) {
15.     a = SetFind(dsu, a);
16.     b = SetFind(dsu, b);
17.     if (a == b) {
18.         return;
19.     }
20.     if (rank[a] < rank[b]) {
21.         dsu[a] = b;
22.     }
23.     else if (rank[a] > rank[b]) {
24.         dsu[b] = a;
25.     }
26.     else {
27.         dsu[a] = b;
28.         rank[b]++;
29.     }
30. }
```

```

35.     vector<int> dsu(n), rank(n);
36.
37.     for (int i = 0; i < n; i++) {
38.         dsu[i] = i; // create new set from vertex i
39.         rank[i] = i;
40.     }
41.
42.     for (int i = 0; i < m; i++) {
43.         int a, b;
44.         cin >> a >> b;
45.         a--;
46.         b--;
47.         SetUnite(dsu, rank, a, b);
48.     }
49.
50.     for (int i = 0; i < n; i++) {
51.         cout << SetFind(dsu, i) << " ";
52.     }
53.     return 0;
54. }

```

Success #stdin #stdout 0.01s 5284KB

 stdin

6 4
1 2
2 3
4 5
3 6

 stdout

5 5 5 4 4 5

Back to MST

Let's go back to Kruskal's algorithm.

Now that we can quickly check if two vertices belong to the same component, the final algorithm will work in $O(m \log(n))$ for sorting and $O(ma)$ for the algorithm itself.

```
5. struct Edge {  
6.     int a, b, c;  
7. };
```

```
40. int main() {  
41.     int n, m;  
42.     cin >> n >> m;  
43.     vector<int> dsu(n), rank(n);  
44.  
45.     for (int i = 0; i < n; i++) {  
46.         dsu[i] = i; // create new set from vertex i  
47.         rank[i] = i;  
48.     }  
49.  
50.     vector<Edge> edges;  
51.  
52.     for (int i = 0; i < m; i++) {  
53.         int a, b, c;  
54.         cin >> a >> b >> c;  
55.         a--;  
56.         b--;  
57.         edges.push_back({a, b, c});  
58.     }  
59.  
60.     sort(edges.begin(), edges.end(), comp);  
61.  
62.     for (auto [a, b, c] : edges) {  
63.         if (SetFind(dsu, a) != SetFind(dsu, b)) {  
64.             cout << a << " " << b << " " << c << "\n";  
65.             SetUnite(dsu, rank, a, b);  
66.         }  
67.     }  
68.     return 0;  
69. }
```

 stdin

5 7
1 2 2
1 3 1
1 4 3
1 5 3
2 4 3
3 4 3
4 5 1

 stdout

0 2 1
3 4 1
0 1 2
0 3 3

Task 1

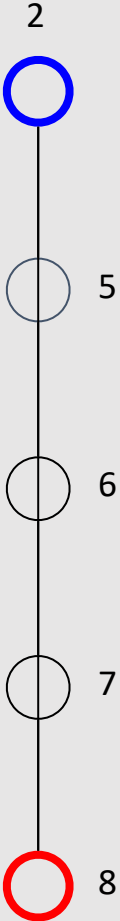
Given an undirected graph, operations of two types are performed in a specified order:

cut - cut the graph, i.e., remove an edge from it.

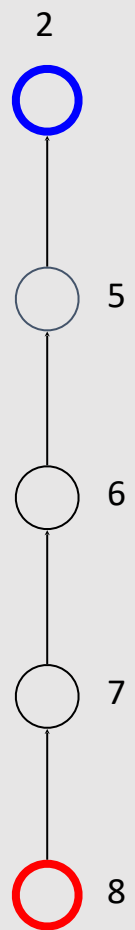
ask - check if two vertices of the graph belong to the same connected component.

It is known that after executing all cut operations, no edges remain in the graph. Determine the result of each ask operation.

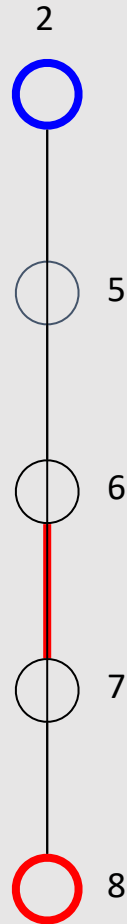
graph

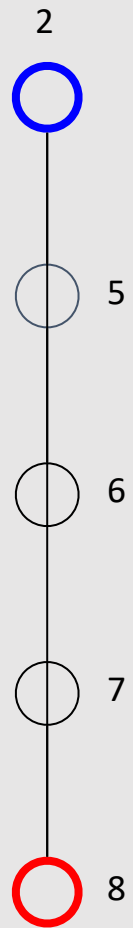


dsu

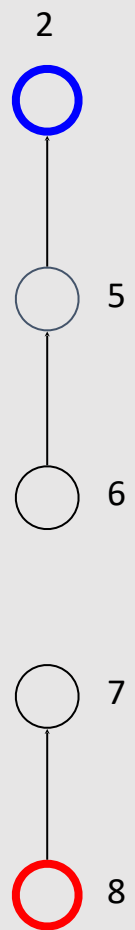


graph





dsu



Task 1

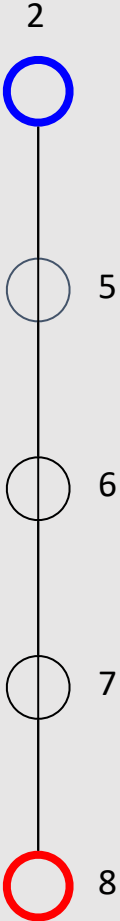
Given an undirected graph, operations of two types are performed in a specified order:

cut - cut the graph, i.e., remove an edge from it.

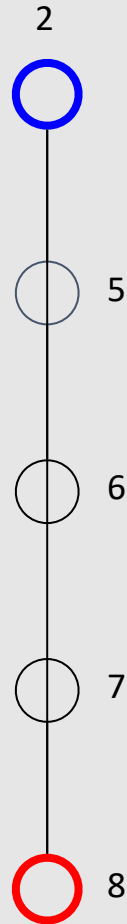
ask - check if two vertices of the graph belong to the same connected component.

It is known that after executing all cut operations, no edges remain in the graph. Determine the result of each ask operation.

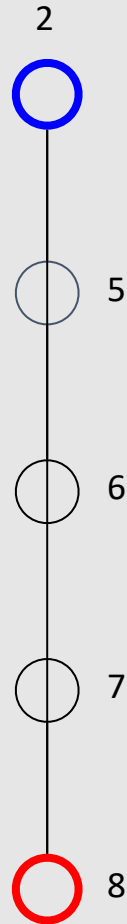
graph



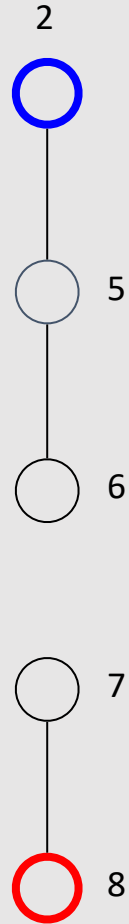
cut (6,7)



ask (6,7) - True



cut (2,8)



cut (5, 6)



ask (2, 6) - False



cut (2, 5)



cut (7, 8)

2



5



6



7



8



Reversed Task 1

Determine the result of each ask operation.

At the start we have empty graph.

create - create an edge

ask - check if two vertices of the graph belong to the same connected component.

Et the end we have some undirected graph.

unite (7, 8)



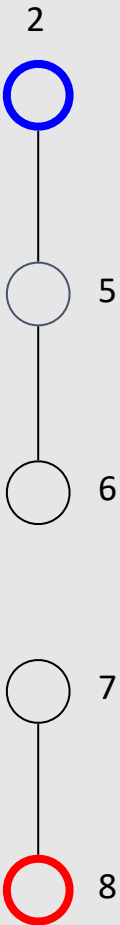
unite (2, 5)



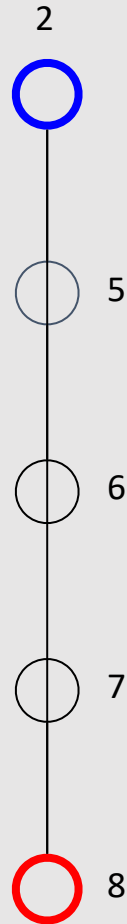
ask (2, 6) - False



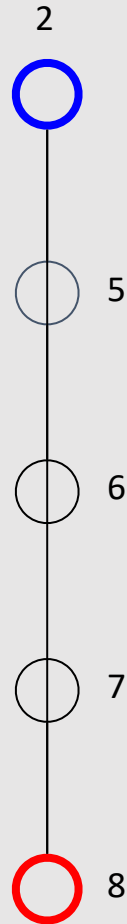
unite (5, 6)



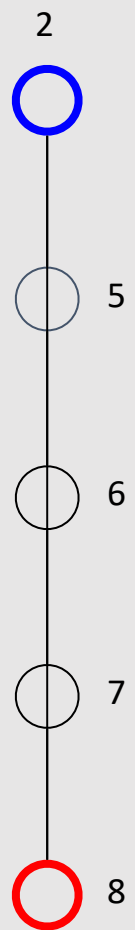
cut (2,8)



ask (6,7) - True



unite (6,7)



Task 2

The problem of painting subsegments: there is a segment of length L , each cell of which (i.e., each individual unit of length 1) has a color of zero. Requests of the form (l, r, c) are received - repaint the segment $[l; r]$ in color c . It is required to find the final color of each cell. The queries are considered known in advance, i.e., the problem is offline.

0	0	0	0	0
---	---	---	---	---

`a[2..3] = 1`

0	0	1	1	0
---	---	---	---	---

$a[1..2] = 2$

0	2	2	1	0
---	---	---	---	---

dsu

ar

0	0	0	0	0
---	---	---	---	---

dsu

0	1	2	3	4
---	---	---	---	---

dsu

ar	0	2	0	3	0
dsu	0	2	2	4	4

dsu

ar	0	2	2	3	0
dsu	0	3	3	4	4

Task 2

We will start from end.

Imagine we have some query $[l..r] = c$;

$$a[l] = r + 1$$

$$a[l + 1] = r + 1$$

$$a[l + 2] = r + 1$$

dsu

dsu	0	1	2	3	4
-----	---	---	---	---	---

dsu

dsu	0	2	3	4	4
-----	---	---	---	---	---

dsu (path optimization)

dsu

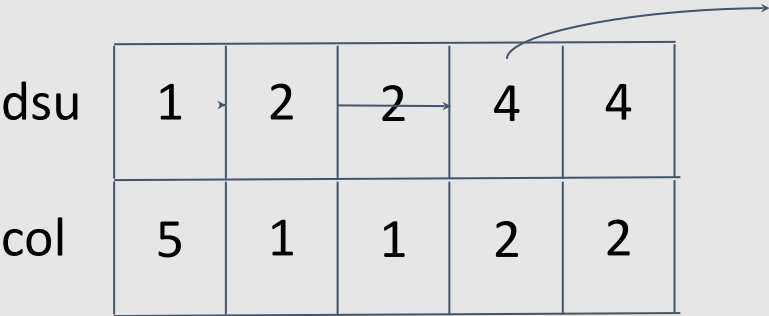
0	4	4	4	4
---	---	---	---	---

some complex segment (1..5) = 5

dsu	0	2	2	4	4
col	0	1	1	2	2

some complex segment (1..5) = 5

dsu	1	2	2	4	4
col	5	1	1	2	2



```
void init() {
    for (int i=0; i<L; ++i)
        make_set (i);
}

void process_query (int l, int r, int c) {
    for (int v=l; ; ) {
        v = find_set (v);
        if (v >= r) break;
        answer[v] = c;
        parent[v] = v+1;
    }
}
```

Task 2

segments: ($[1..3] = 5$, $[2..5] = 3$, $[4..6] = 4$)

How to solve:

we create dsu: $[0, 1, 2, 3, 4, 5, 6]$

r_segments: ($[4..6] = 4$, $[2..5] = 3$, $[1..3] = 5$)

dsu	0	1	2	3	4	5	6
col	0	0	0	0	0	0	0

$[4..6] = 4$

dsu	0	1	2	3	5	5	6
col	0	0	0	0	4	0	0

[4..6] = 4

dsu	0	1	2	3	5 →	6 →	6
col	0	0	0	0	4	4	0


$[4..6] = 4$

dsu	0	1	2	3	5	6	7
col	0	0	0	0	4	4	4



the path optimization

dsu	0	1	2	3	7	7	7
col	0	0	0	0	4	4	4

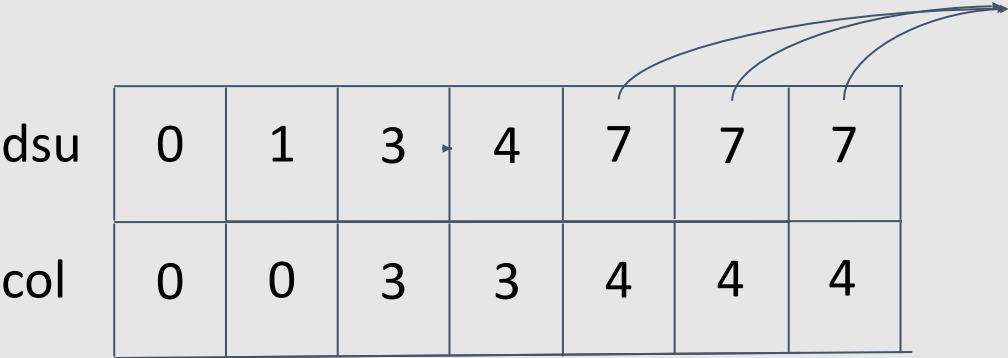


$[2..5] = 3$

dsu	0	1	3	3	7	7	7
col	0	0	3	0	4	4	4

$[2..5] = 3$

dsu	0	1	3	4	7	7	7
col	0	0	3	3	4	4	4

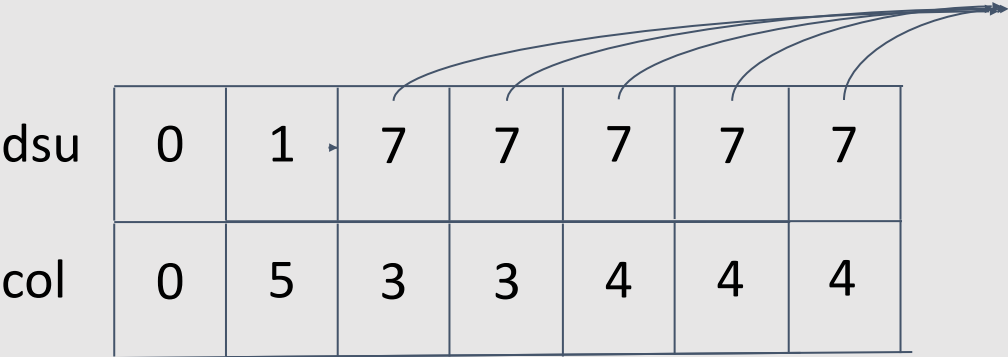


the path optimization

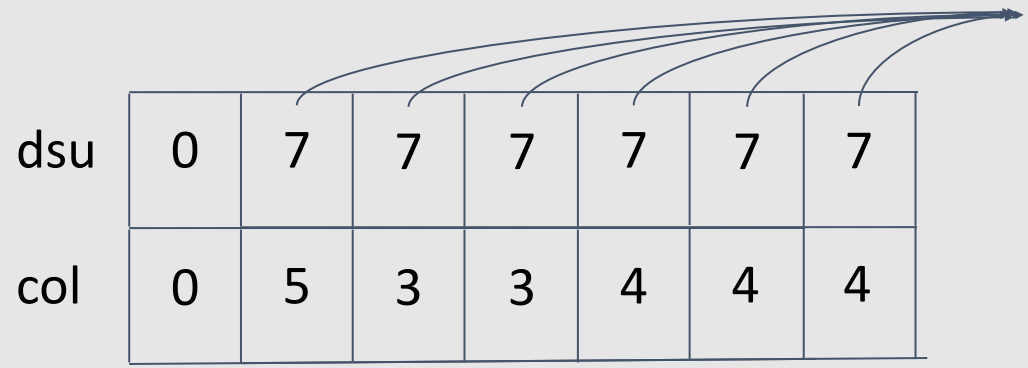


[1..3] = 5

dsu	0	1	7	7	7	7	7
col	0	5	3	3	4	4	4



the path optimization

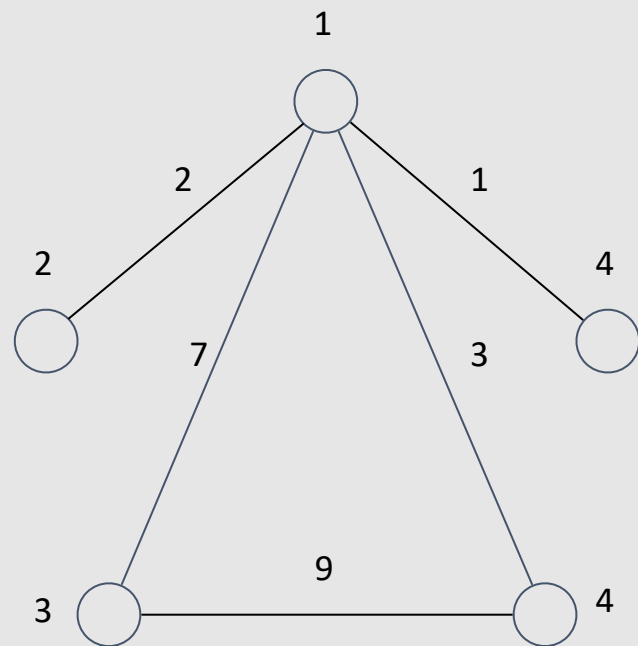


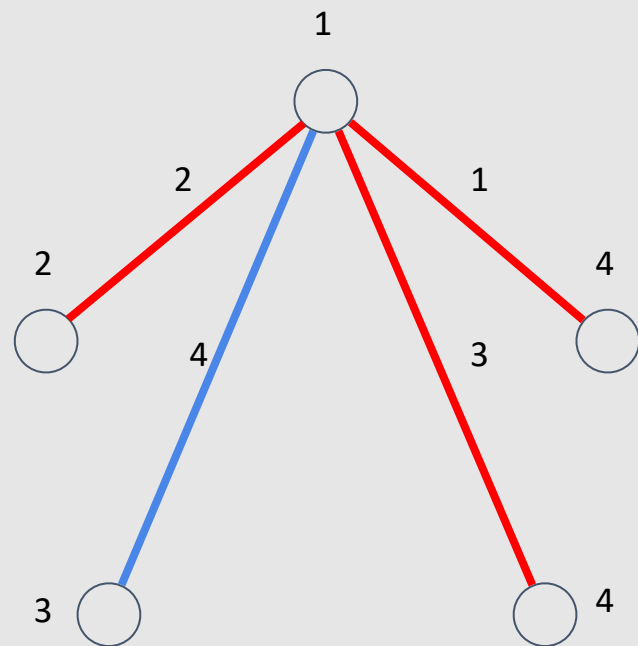
Task 3

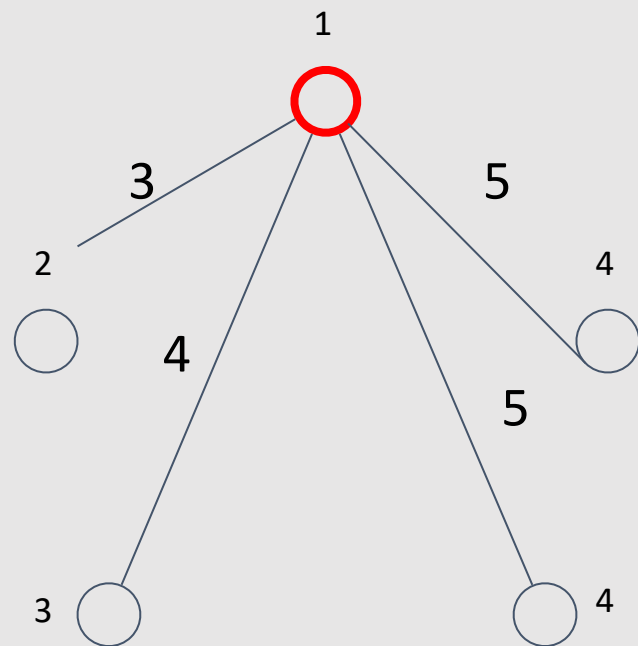
Given a graph with n vertices, where each vertex i has a number a_i assigned. Initially, the graph has no edges.

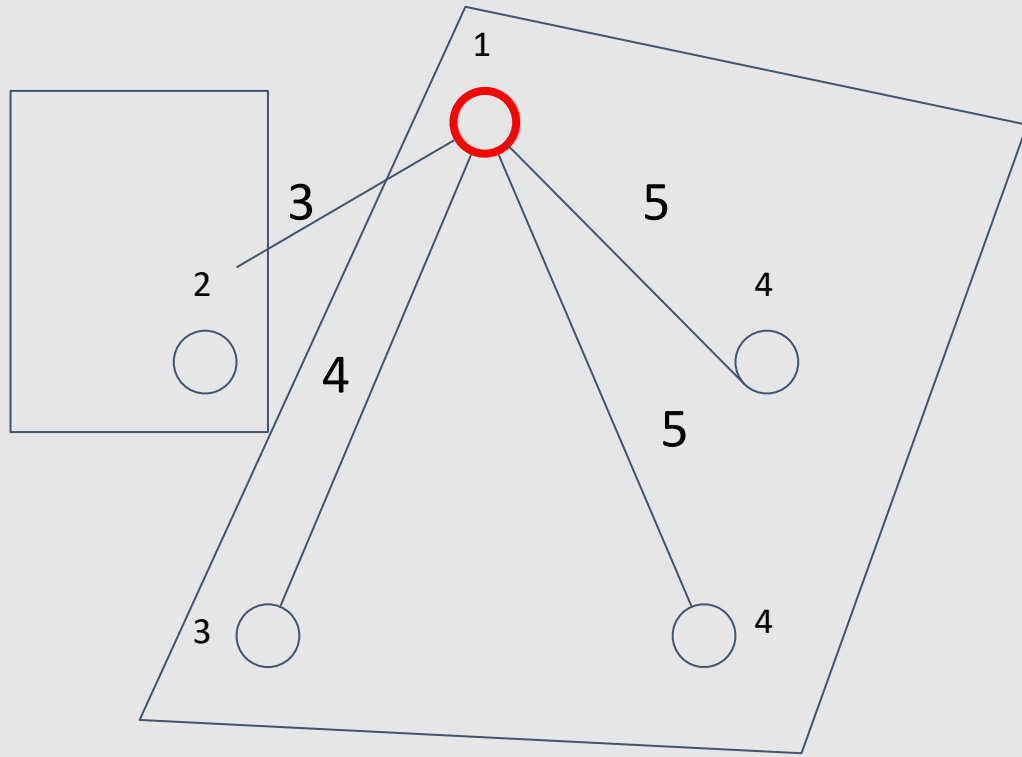
Adding an edge between vertices x and y requires a payment of $a_x + a_y$ coins. Additionally, there are m special offers, each characterized by three numbers x , y , and w . These offers allow adding an edge between vertices x and y for w coins.

How many coins, at a minimum, will you need to make the graph connected?









Task 4 (practical)

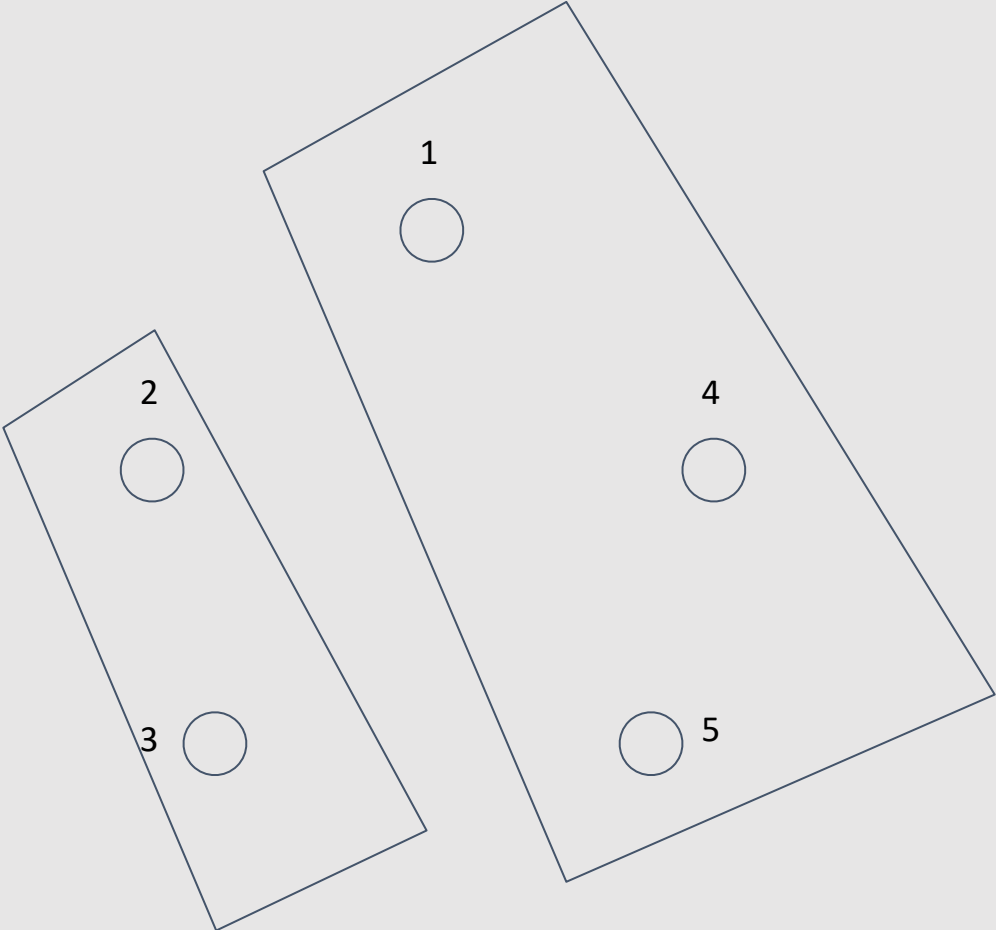
Come up with a way to generate a maze with a single path and a single entrance and exit.

0	1				5
0					

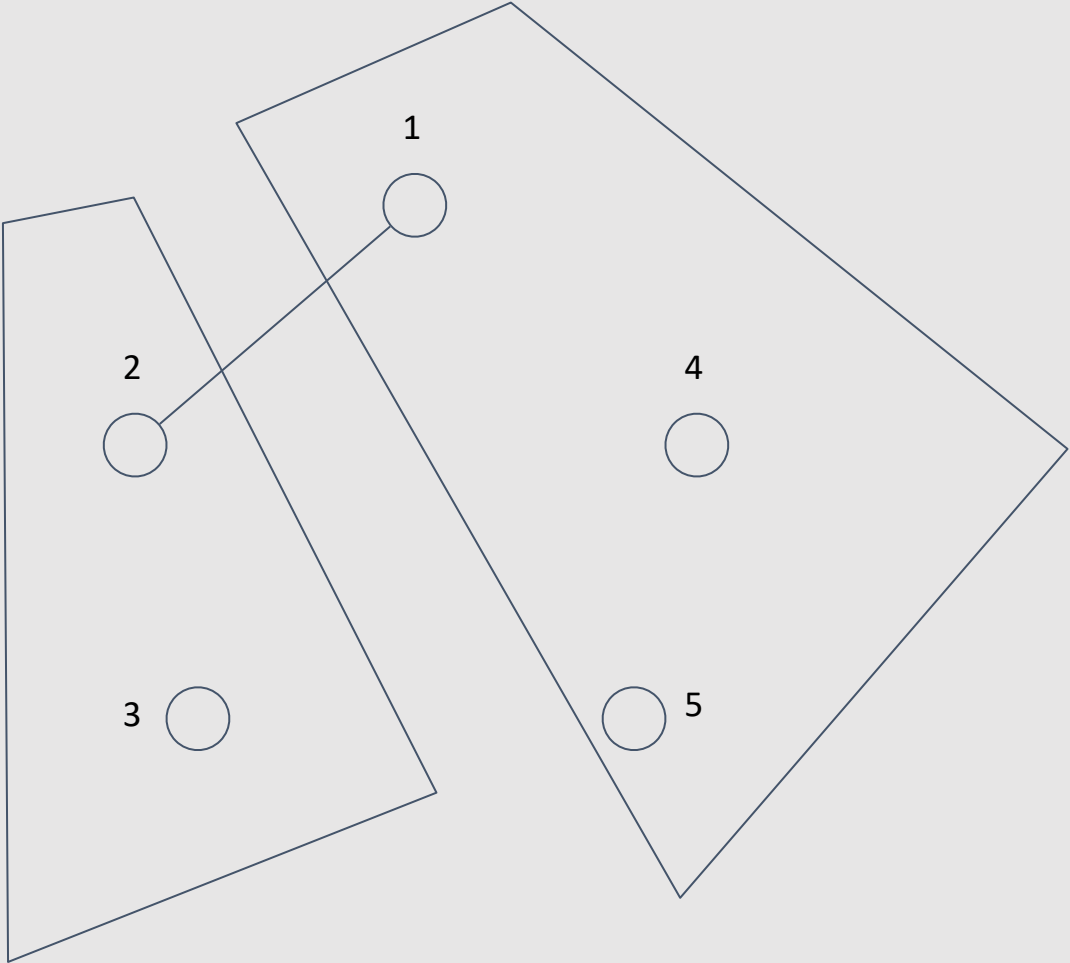
Task 5

At first, an empty graph is provided, and edges can be added to it. Requests of the following type are received: 'Is the connected component containing a given vertex bipartite?'

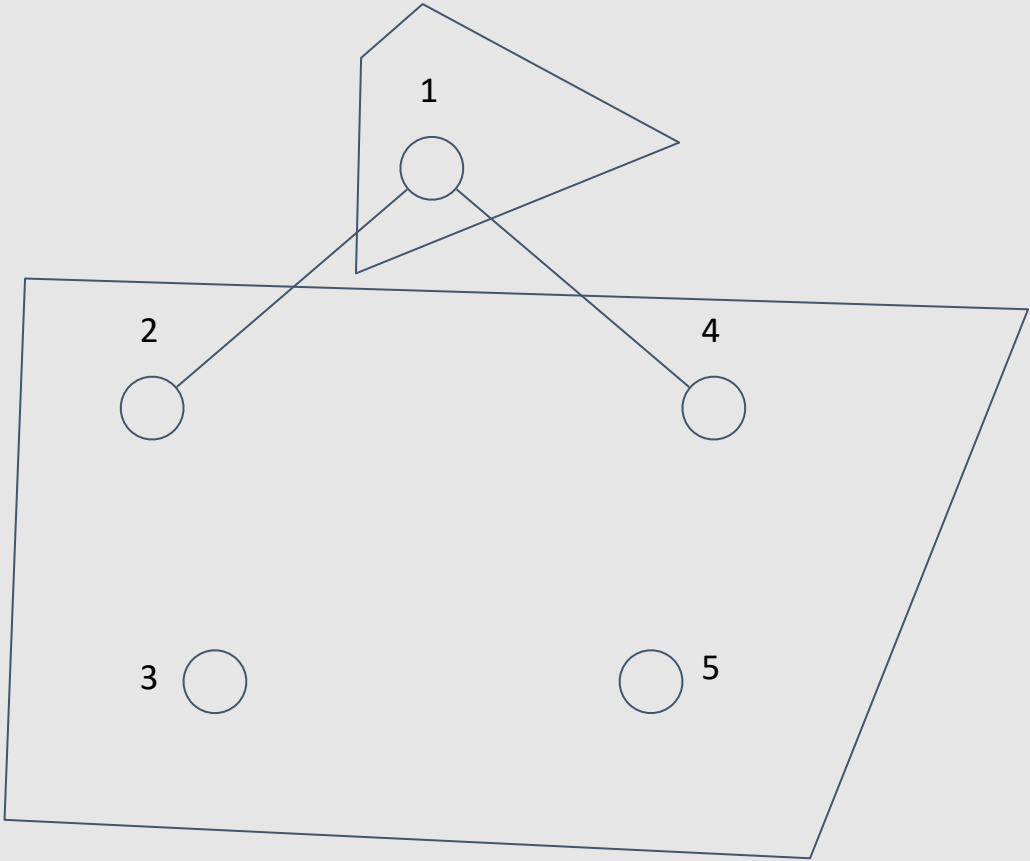
YES



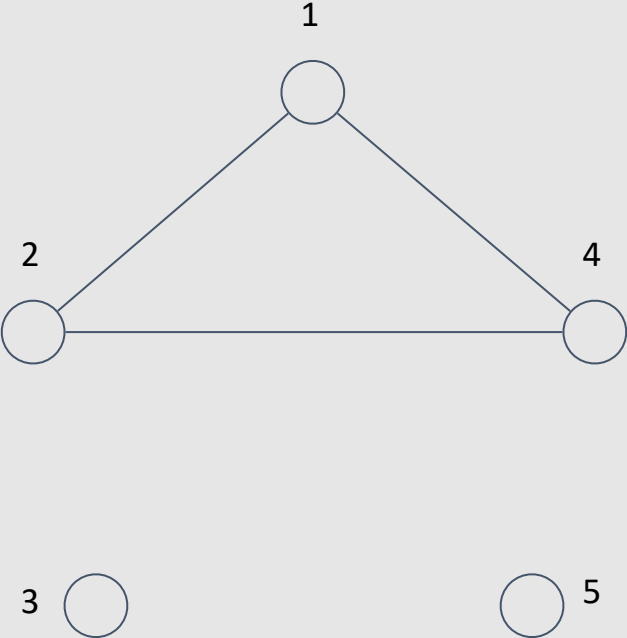
YES



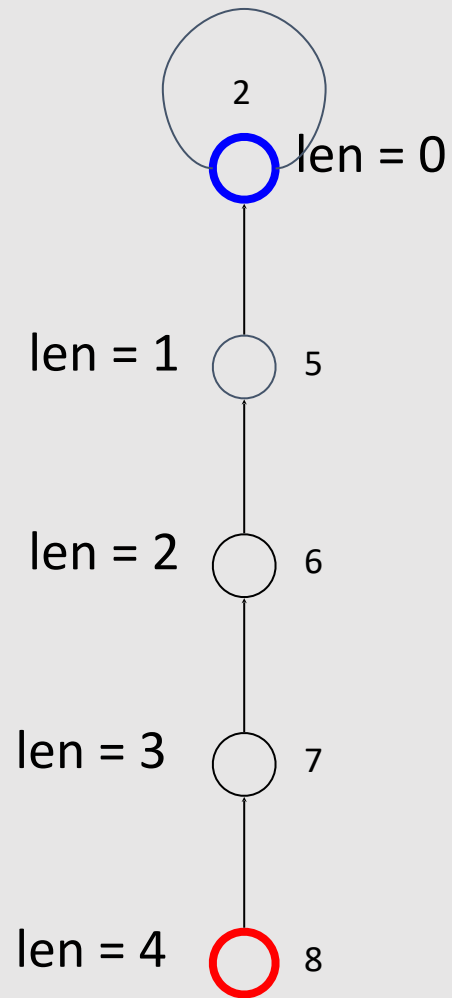
YES



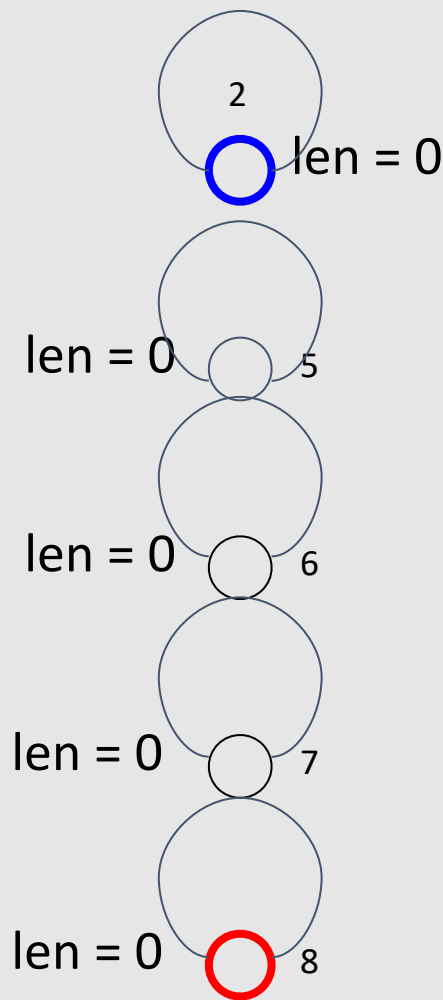
NO



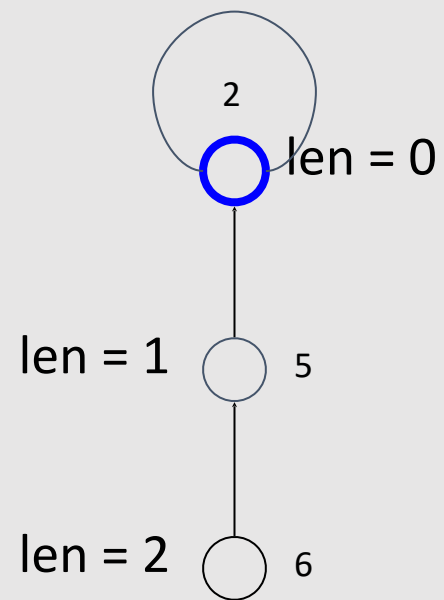
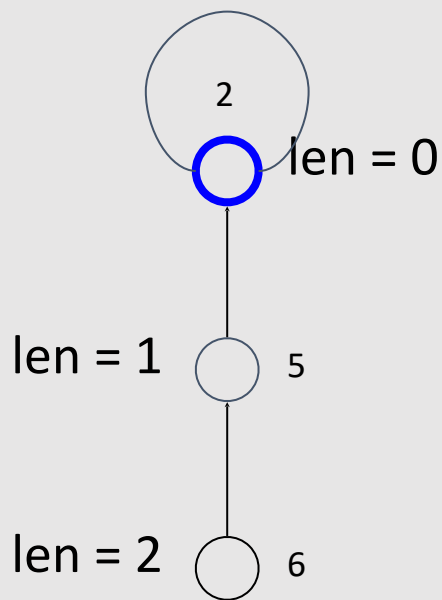
dsu



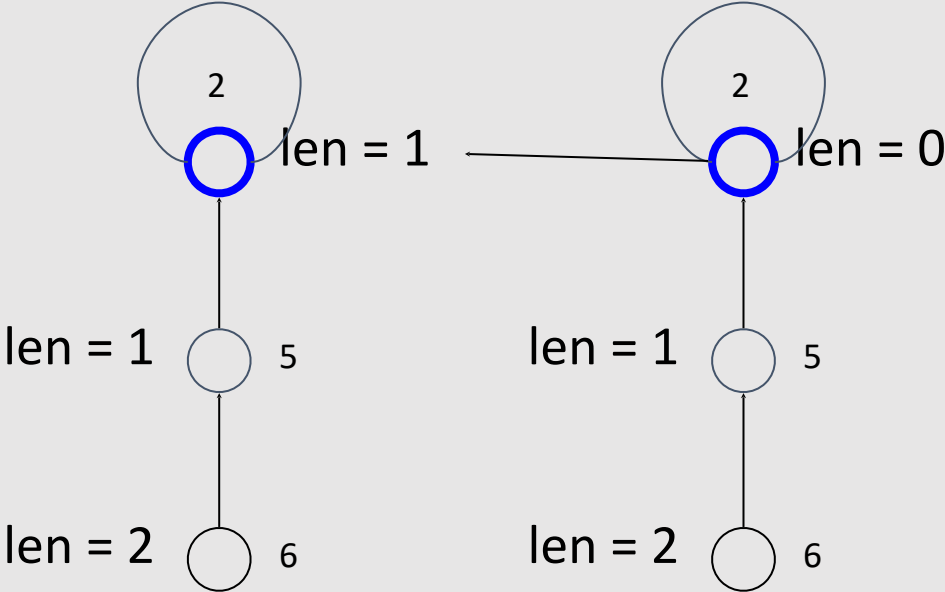
dsu



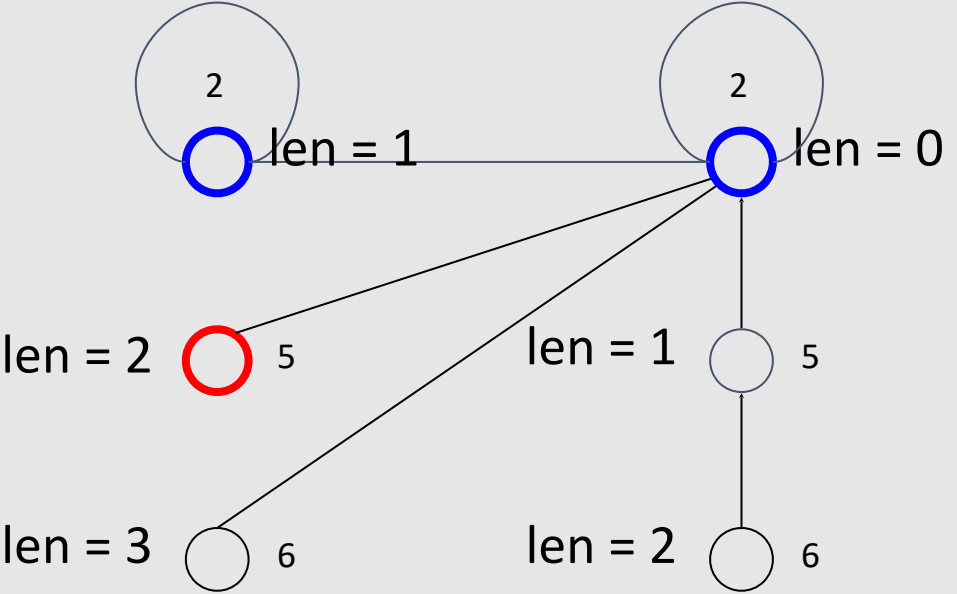
dsu



dsu



dsu



```
pair<int,int> find_set (int v) {  
    if (v != parent[v].first) {  
        int len = parent[v].second;  
        parent[v] = find_set (parent[v].first);  
        parent[v].second += len;  
    }  
    return parent[v];  
}
```


Task *

You need to solve RMQ, using dsu

RMQ offline

all numbers from 0 to $1e6$

insert number (from 0 to $1e6$)

find minimum

Task *

insert 1

find - 0

insert 0

find - 2

insert 3

find - 2

All codes

Prim(c++) - <https://ideone.com/s1tPQP>

dsu-naive(c++) - <https://ideone.com/papSUh>

dsu(c++) - <https://ideone.com/yCGWjh>

kruskal(c++) - <https://ideone.com/VwtEn0>