

Union-Find講習会



東京工業大学
デジタル創作同好会

注意

今回の講習会で扱う言語はC++とPythonのみです。
スライドで書かれているコードは全てC++のコードのみ
です。

資料

 <https://github.com/yuyu5510/Union-Find/tree/main>

にコードとスライドを置いています。

自己紹介

AtCoder ID: @yuyu5510

splatoonとAtCoderをしています。Codeforcesは一回参加してそれっきりです。(やれ)

splatoonのアップデを生きがいとして生きてています。

前提知識



東京工業大学
デジタル創作同好会

グラフの用語(1/4)

- ▶ 頂点: 点のこと
- ▶ 辺: 頂点と頂点を結ぶもの
- ▶ グラフ: 頂点と辺からなるもの
- ▶ パス: 経路のこと
- ▶ 閉路: ある頂点から辺をいくつか辿って同じ頂点につく時の経路、ループのこと

グラフの用語(2/4)

- ▶ 有向グラフ: 辺に向きが存在するグラフ
- ▶ 無向グラフ: 辺に向きが存在しないグラフ
- ▶ 連結: 無向グラフ上では任意の2頂点間にパスが存在する=全ての頂点が行き来可能のこと
- ▶ 木: 無向グラフで連結かつ閉路が存在しないグラフ N頂点の時、辺がN-1個になる

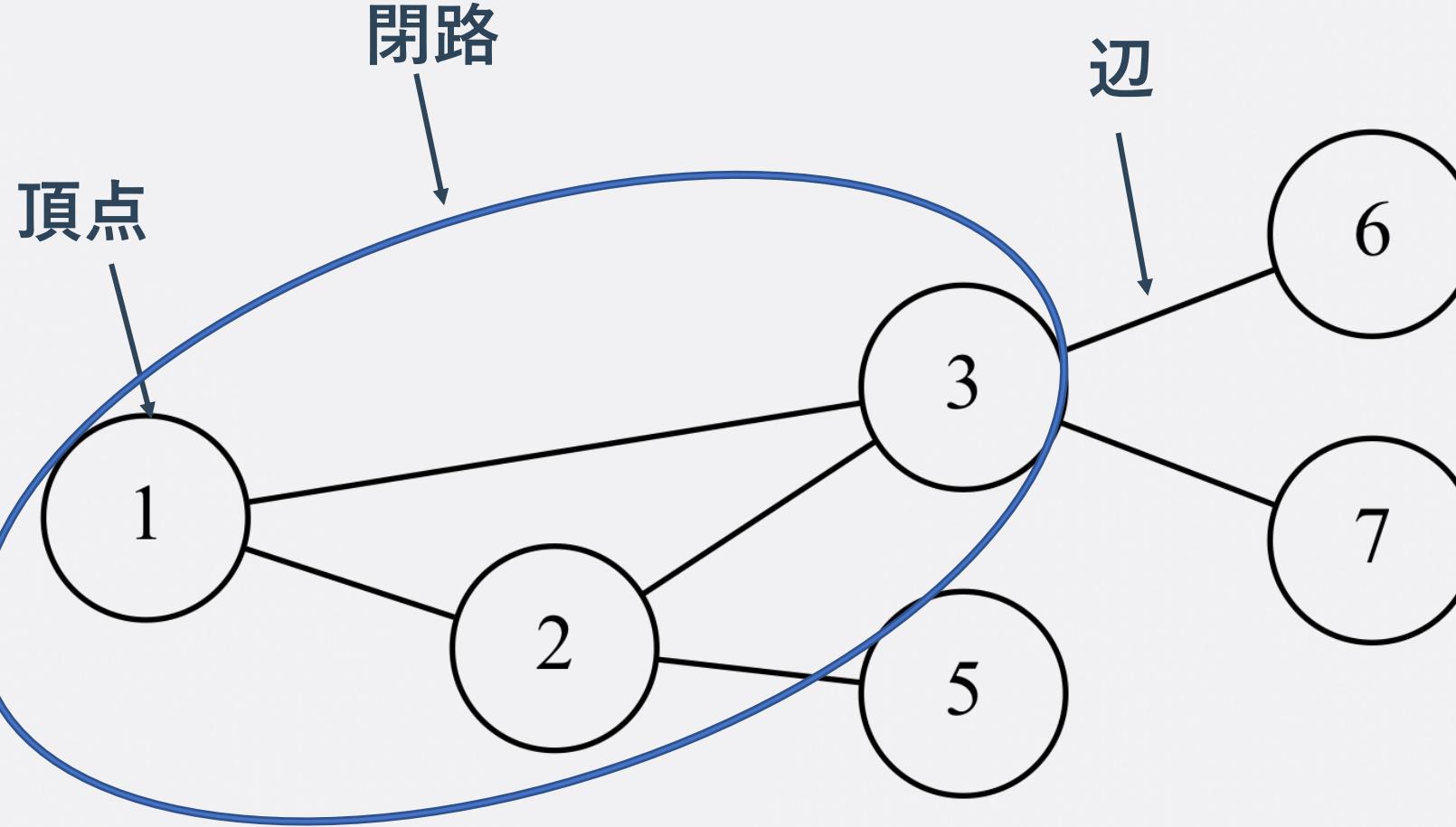
グラフの用語(3/4)

- ▶ 森: 木が複数ある状態
- ▶ 根: 木の一つの頂点を特別扱いしたもの
- ▶ 親: 木において、ある頂点まで木の根から辿った時一個前の頂点
- ▶ 子: 木において、ある頂点から木の根から遠い方の辺を一個たどった時の頂点

グラフの用語(4/4)

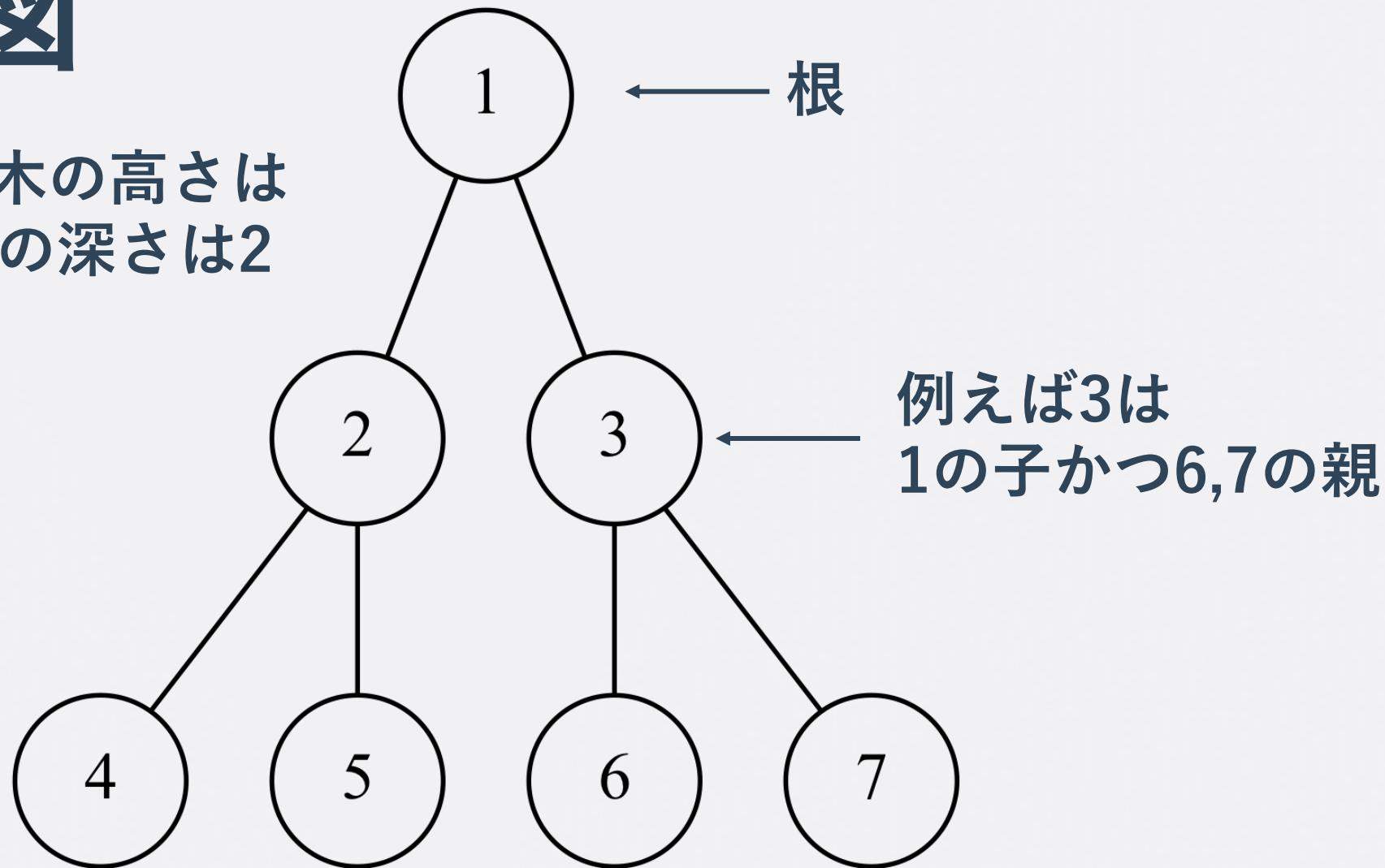
- ▶ 葉: 子が存在しない頂点
- ▶ 深さ: 木において、根からその頂点にたどり着くまでに通った辺の数のこと
- ▶ 木の高さ: 木において、頂点の深さの最大値のこと

無向グラフの図



木の図

この図では木の高さは3で、頂点3の深さは2



Union-Findを学ぼう



Union-Findって何？

Union-Findはあるグループと別のグループを併合する操作と、ある要素と別の要素が同じグループにあるかどうかを高速に判定することができます。

実装を工夫することで要素の数をNとしてならし計算量 $O(\alpha(N))$ でこれらの操作を行うことができます。

Union-Findって何？

Union-Findは森(木がいくつある状態)の構造をしたデータ構造になります。辺を追加することでグループの併合を行い、要素が属する木の根を比較することで同じグループかどうかを判定していきます。

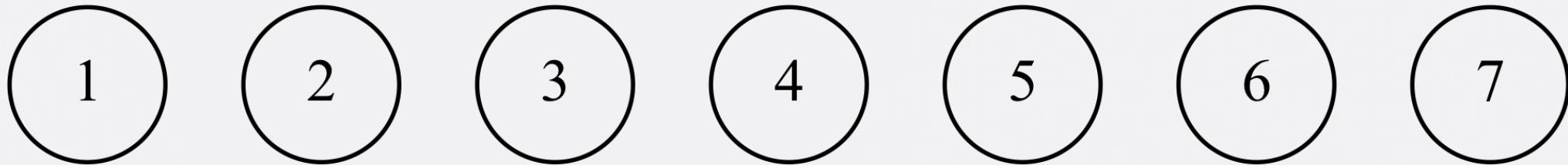
今回扱うUnion-Findは切り離す操作は高速にできません。切り離す操作が高速にできるUnion-Findも存在するらしいです。

$O(\alpha(N))$ の速さ

$\alpha(N)$ はアッカーマン関数の逆関数で、 $N \leq 10^{80}$ に対して $\alpha(N) \leq 4$ が成り立つので、ほぼ定数とみなしています。

アッカーマン関数の詳細はここでは触れませんが再帰的な関数で、数が爆発するような定義になっています。

Union-Findの併合操作(1/7)

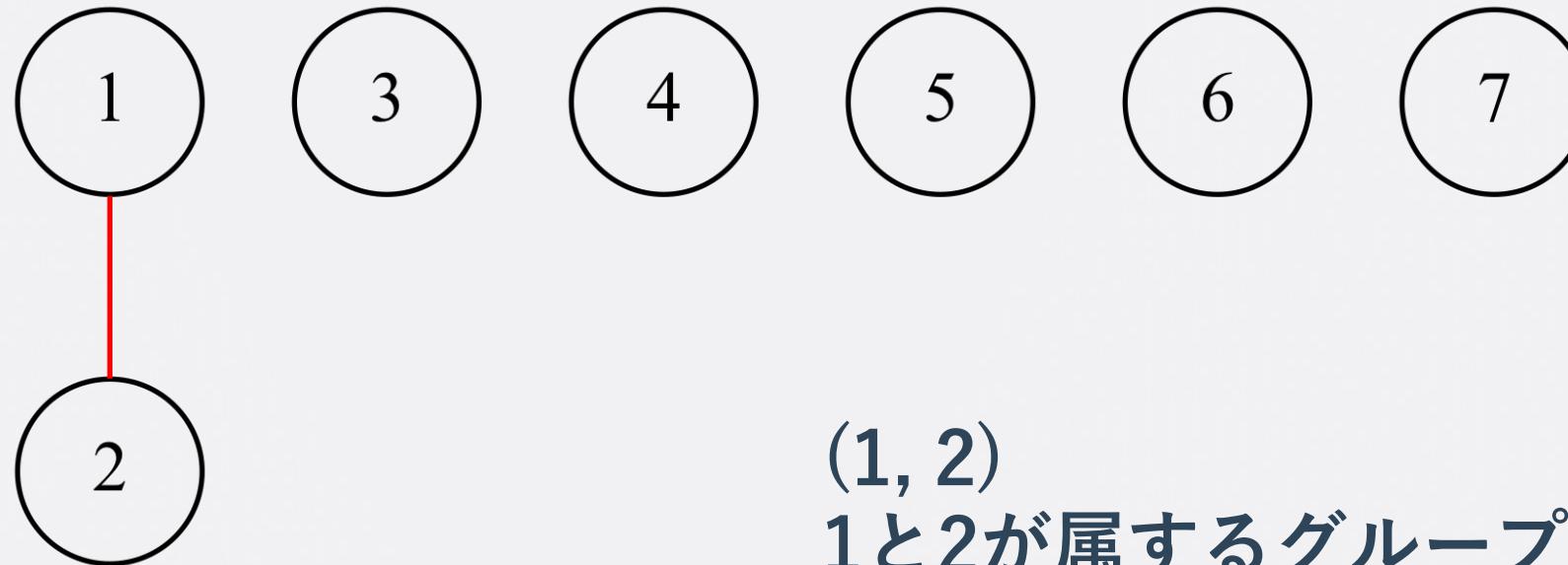


最初は全ての頂点が根の状態です。
併合操作はグループが違う場合辺をはる操作になります。

例として以下の2組が属するグループを併合する操作を見ていきます

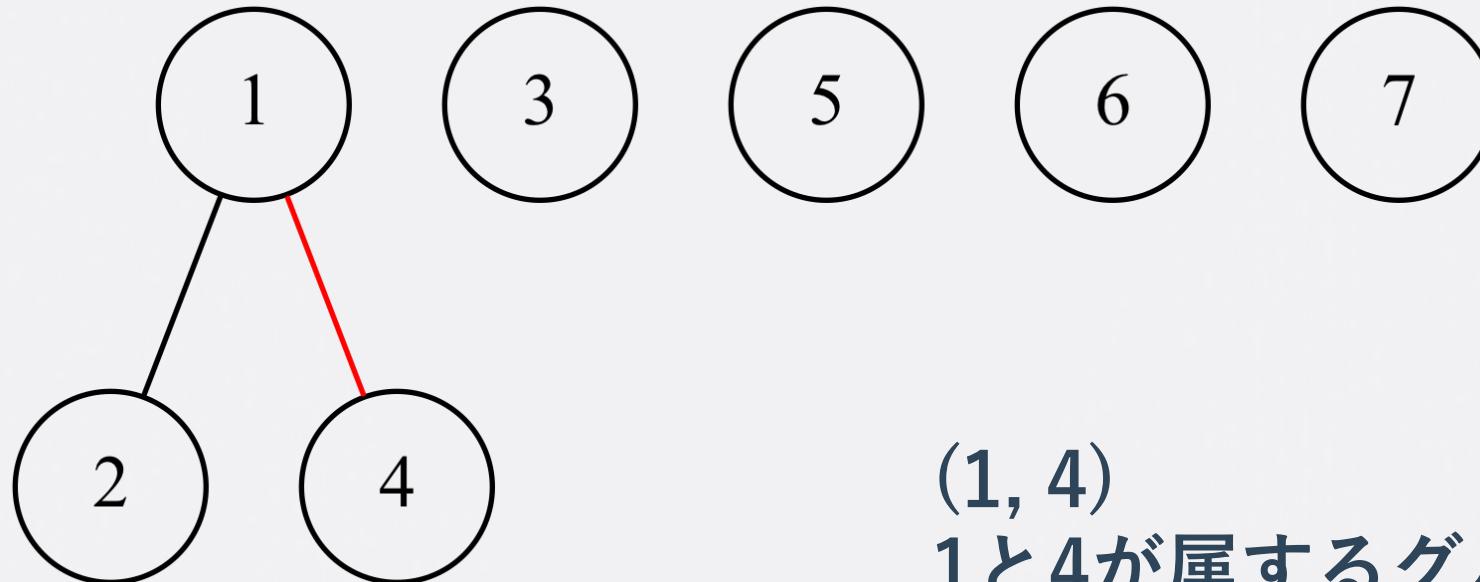
(1, 2) (1, 4) (3, 5) (5, 6) (1, 5) (2, 7)

Union-Findの併合操作(2/7)



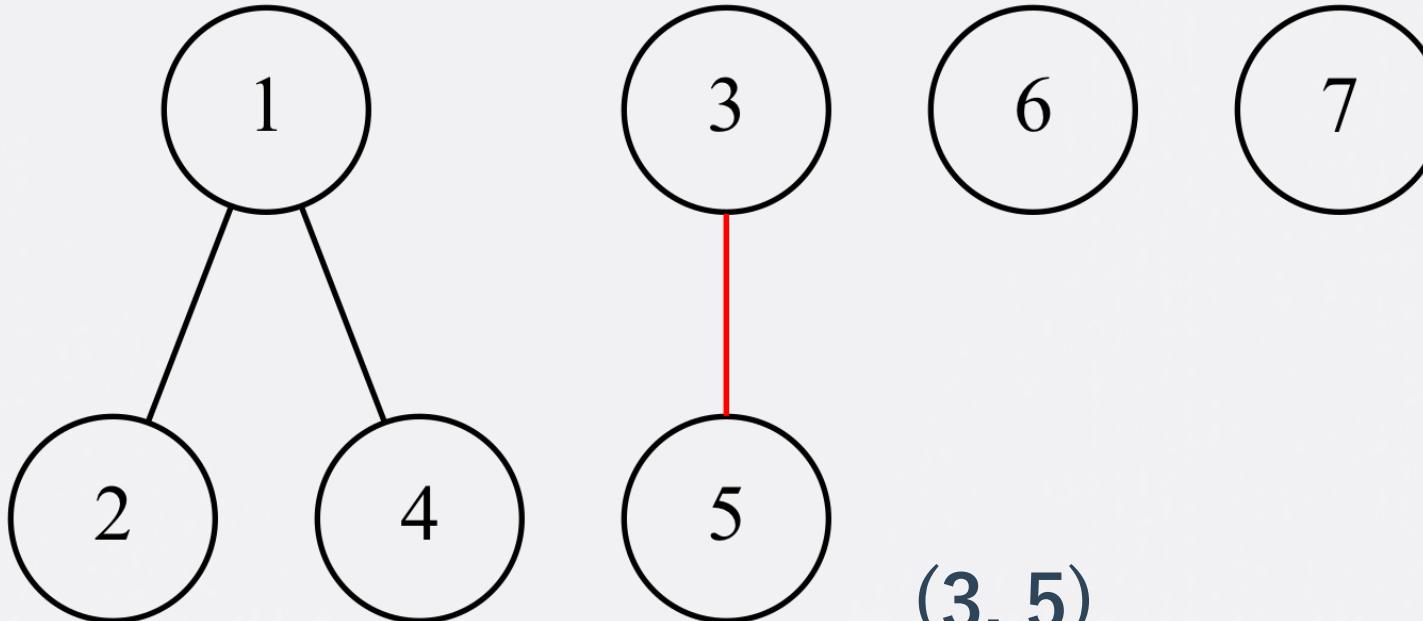
(1, 2)
1と2が属するグループを
併合する操作です。

Union-Findの併合操作(3/7)



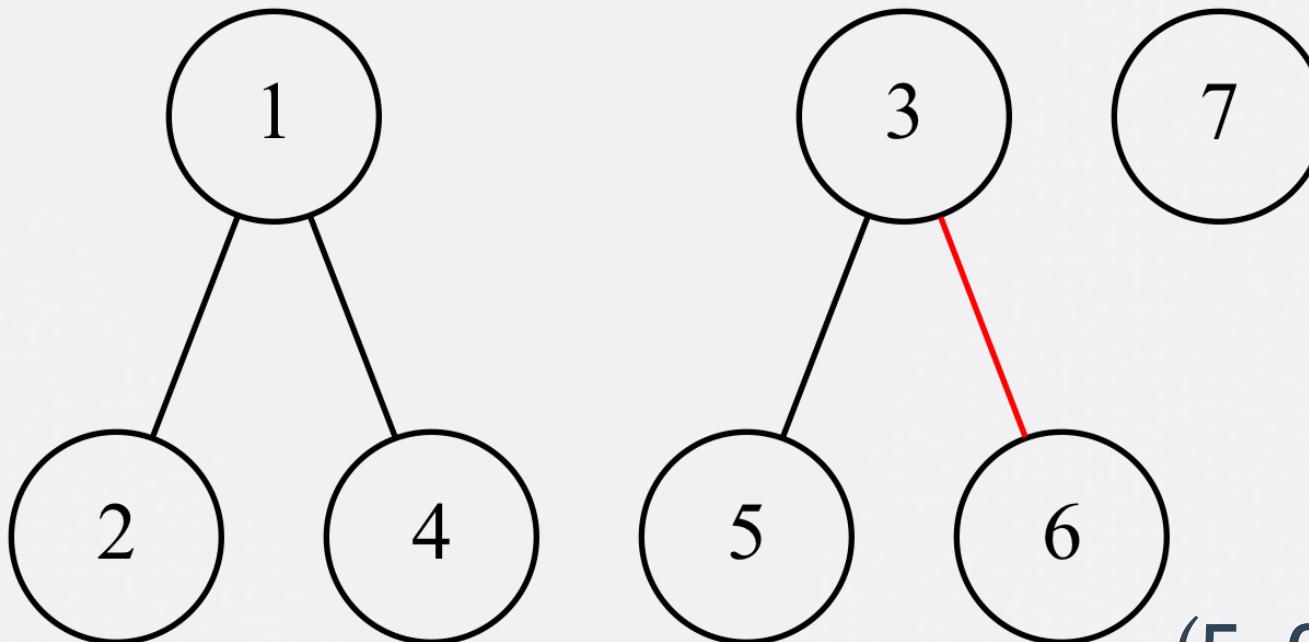
(1, 4)
1と4が属するグループを
併合する操作です。

Union-Findの併合操作(4/7)



(3, 5)
3と5が属するグループを
併合する操作です。

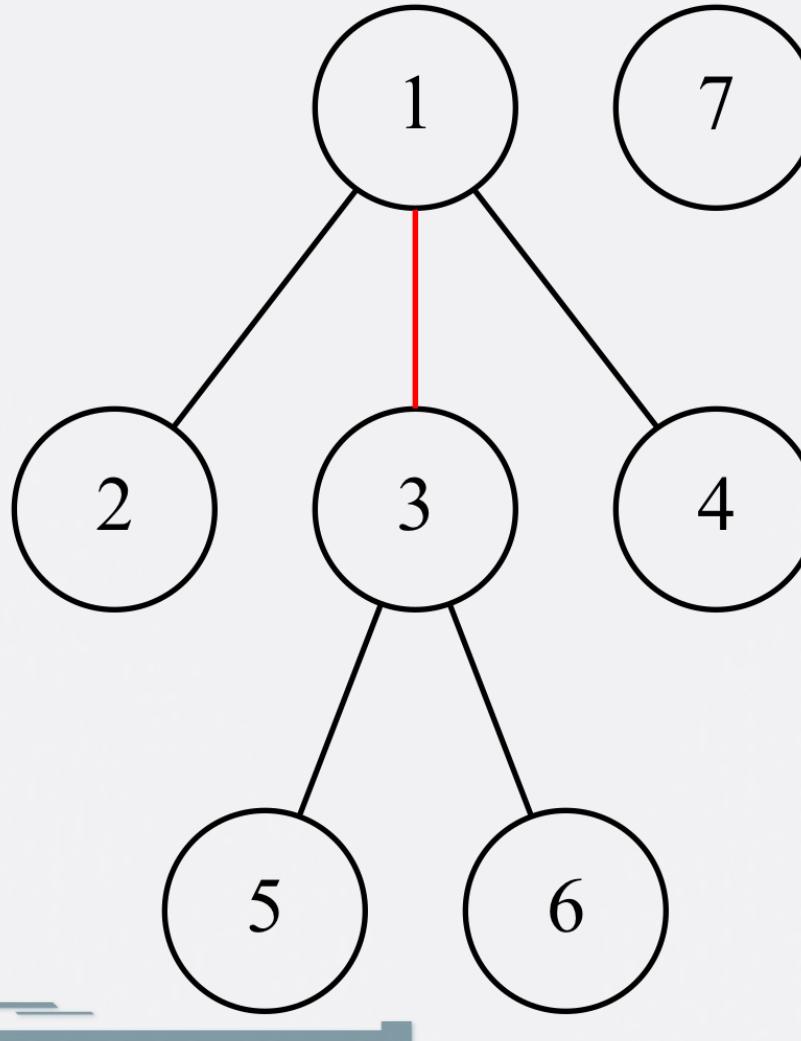
Union-Findの併合操作(5/7)



(5, 6)

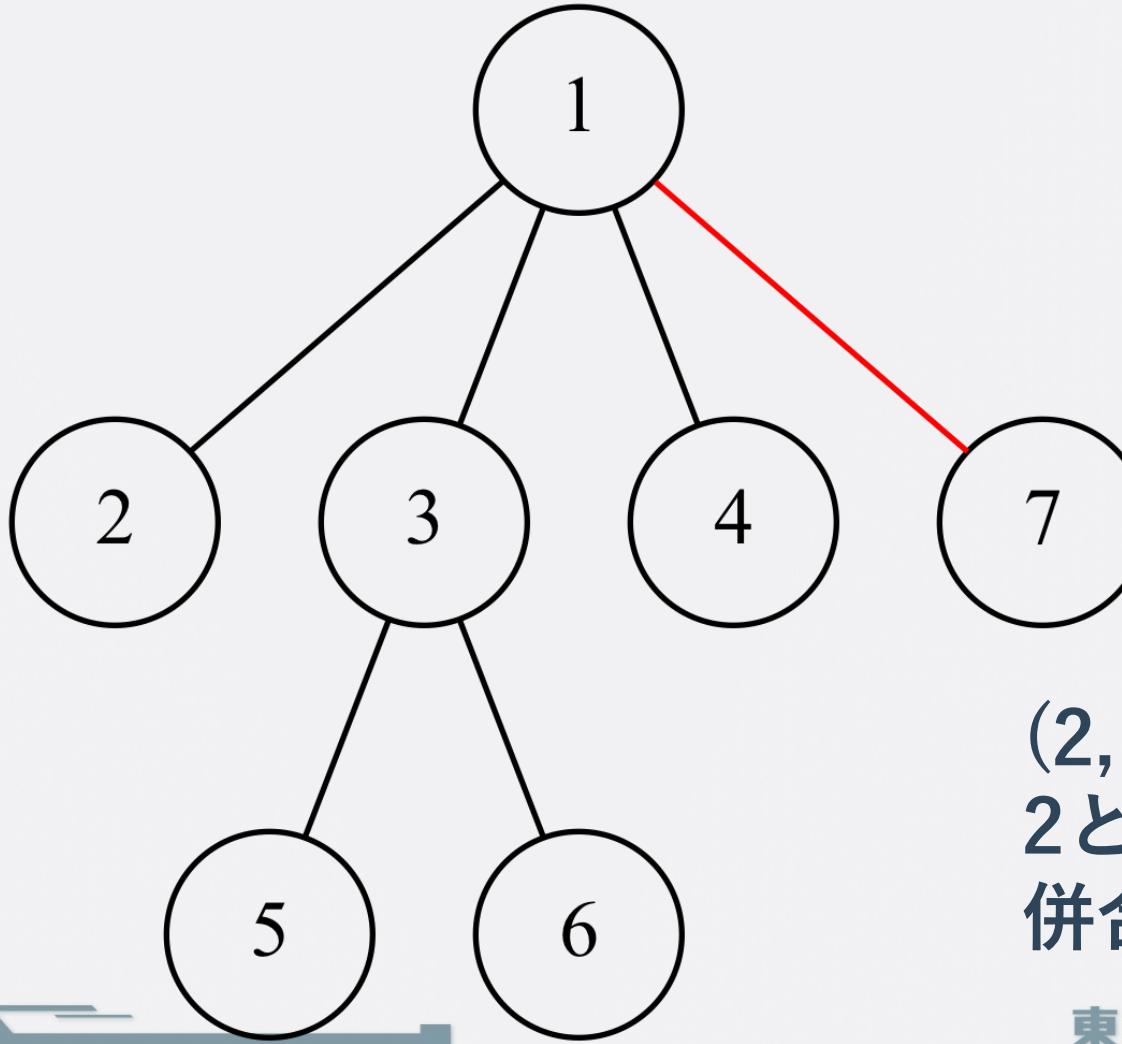
5と6が属するグループを
併合する操作です。

Union-Findの併合操作(6/7)



(1, 5)
1と5が属するグループを
併合する操作です。

Union-Findの併合操作(7/7)

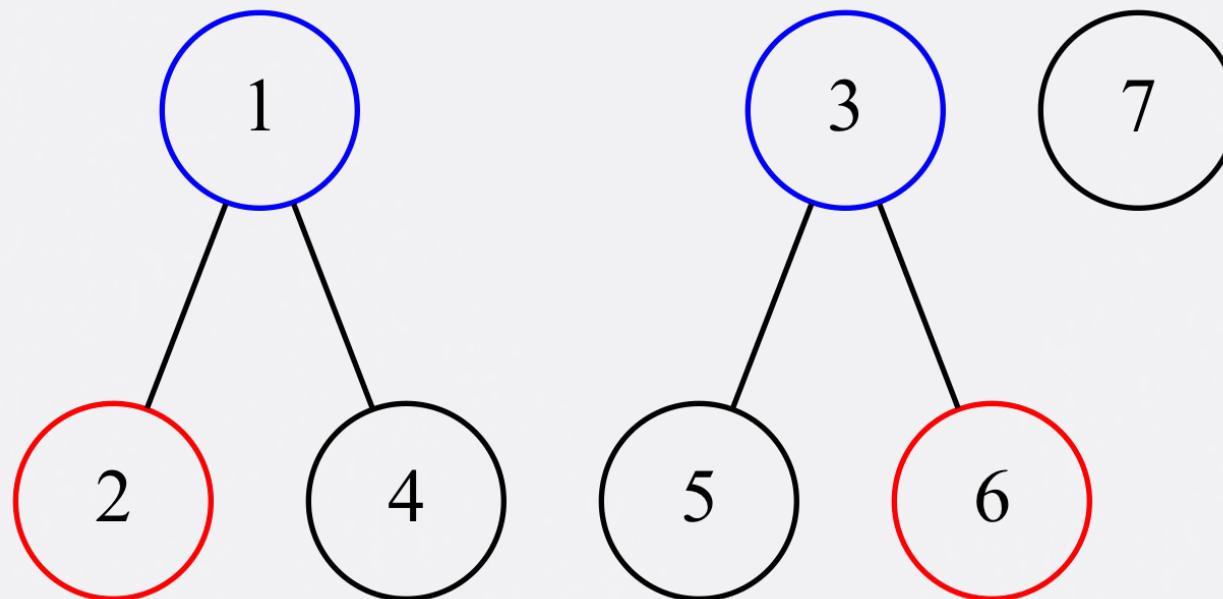


(2, 7)
2と7が属するグループを
併合する操作です。



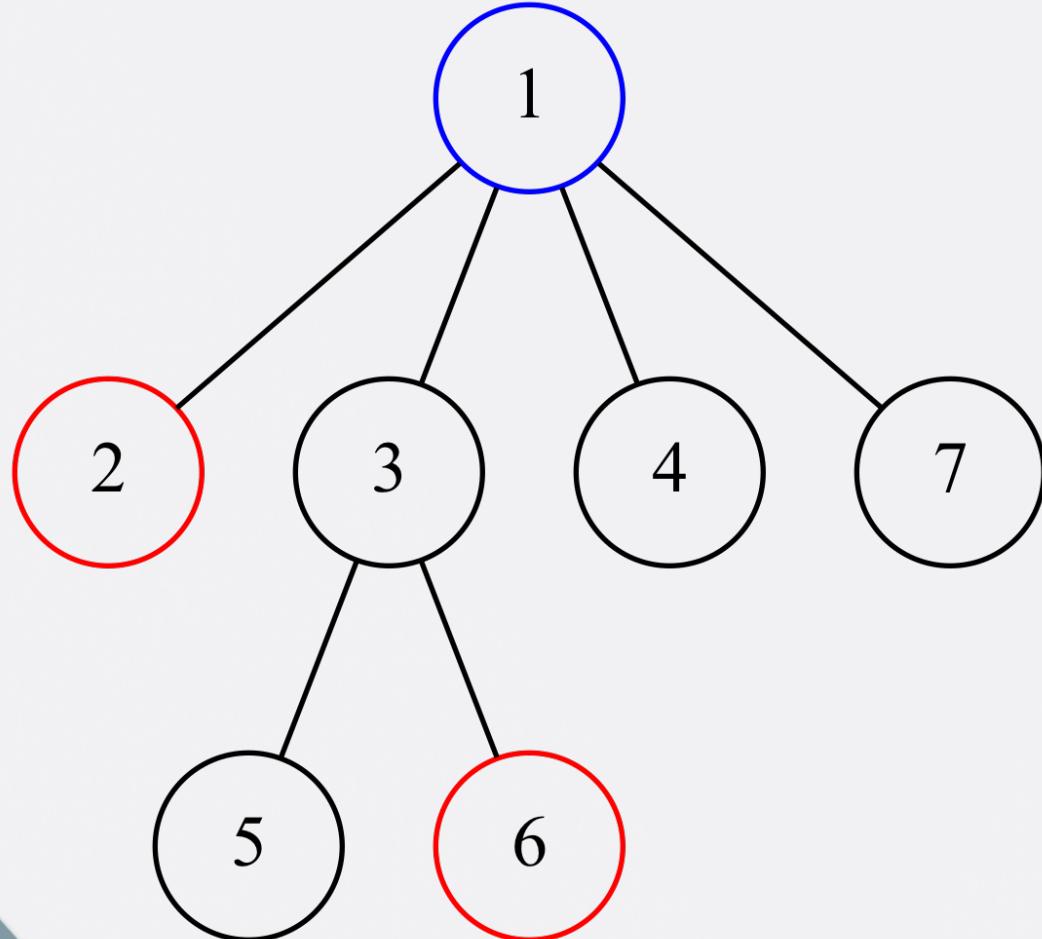
Union-Findの判定操作(1/2)

Union-Findのグループが同じか判定するには
2点の根を見れば良いです。



例として(2, 6)が同じグループ
が判定するのを見ると(1, 3)で
異なるので違うグループだと
いうことがわかります。

Union-Findの判定操作(2/2)



同様にこちらの状態で(2, 6)が同じグループかどうか判定するために根を見ると(1, 1)で等しく、同じグループになっていることがわかります

Union-Findの要素が持つ情報

今回の講習会で扱うUnion-Findは各要素は次の情報を持ります。別的情報が欲しい時は追加してください。

- ▶ par: その要素の親頂点の番号 自分が根の時は-1
- ▶ siz: その要素が根である時、その要素が属するグループの頂点数

Union-Findの初期化

今回Union-Findはclassで実装していきます。

`UnionFind(n)` でn要素の配列(リスト)が用意できるようコンストラクタを書きます。最初はどの要素も根があるのでparは全て-1,sizは全て1の値が入ります。

```
class UnionFind{
    std::vector<int> par, siz;
    UnionFind(int n): par(n, -1), siz(n, 1){}
};
```

Union-Findの関数(1/2)

今回実装する予定なのは以下の4つです。

- ▶ `root(x)`: 要素xが属するグループの木の根の頂点を返す関数
- ▶ `unite(x, y)`: 要素xが属するグループの木と要素yが属するグループの木を併合する関数

Union-Findの関数(2/2)

- ▶ `same(x, y)`: 要素xと要素yが属するグループが同じか判定する関数
- ▶ `size(x)`: 要素xが属するグループの木の要素数を返す関数

root関数

`par[x]`に`x`の親もしくは`x`が根であれば-1が入っているので次のようにかけます。

```
int root(int x){  
    if(par[x] == -1) return x;  
    return root(par[x]);  
}
```

高速化のための工夫(経路圧縮)

先ほどの実装では、根を取り出す時に木の高さに比例した時間がかかるてしまいます。

併合する際、何も工夫しないと木の高さは最悪 $O(N)$ になり、後述する工夫をしても $O(\log N)$ 程度のため、同じ x に対して毎回この計算量がかかるのは嬉しくありません。

高速化のための工夫(経路圧縮)

そこでxから根にたどり着くまでに通った頂点のparに
根の頂点番号を入れてしまします。

これにより根を求める操作が高速化できます。

しかし、一個上の親の情報は失われるので、その情報
が必要な時はしない方が良いです。後述する併合の際の
工夫のみで大体 $O(\log N)$ になるのでこれでも十分高速
です。

+ 工夫したroot関数

```
int root(int x){  
    if(par[x] == -1) return x;  
    return par[x] = root(par[x]);  
}
```

same関数

先ほど定義したroot関数を使うことで、簡単に実装できます。

```
bool same(int x, int y){  
    return root(x) == root(y);  
}
```

size関数

先ほど定義したroot関数を使うことで、簡単に実装できます。

```
int size(x){  
    return siz[root(x)];  
}
```

unite関数

まずは何も工夫しない実装を見ます。

片方の根の親をもう一方の根にすれば良いです。

```
bool unite(int x, int y){  
    int rx = root(x), ry = root(y);  
    if(rx == ry) return false;  
    par[ry] = rx;  
    siz[rx] += siz[ry];  
    return true;  
}
```

高速化のための工夫

x が属するグループと y が属するグループの大きさが小さい方の根の親を大きい方の根とすることによって木の高さが $O(\log N)$ 以下になります。

こちらの高速化の工夫で木の高さが $O(\log N)$ 程度になります。

なぜ $O(\log N)$ になるのか

上の工夫を用いたunite(x, y)で併合する前と併合した後の頂点xの深さに注目します。

$\text{size}(x) < \text{size}(y)$ の時と、 $\text{size}(x) \geq \text{size}(y)$ の時の場合分けをするとわかります。

等号はどちらにつけてもいいです。(実装をちょっと変えるだけで計算量に影響はないです。)

size(x) < size(y)の時:

xが属するグループの根はyが属するグループの根につくため、深さは+1されます。

この時併合後のグループは $2 * \text{size}(x)$ よりも大きくなっています。

$\text{size}(x) \geq \text{size}(y)$ の時

x が属するグループの根に y が属するグループの根がつく
ため、 x の深さは変わりません。

なぜ $O(\log N)$ になるのか

併合した時に、頂点数が2倍以上になる時のみ、深さが+1されることを踏まえると、木の高さが $\log N$ 以下におさまることがわかります。

経路圧縮は状況により、するかどうか決めた方が良いですが、こちらはして困る状況はおそらくないので、しましよう。

工夫したunite関数

```
bool unite(int x, int y){  
    int rx = root(x), ry = root(y);  
    if(rx == ry) return false;  
    if(size(rx) < size(ry)) std::swap(rx, ry);  
  
    par[ry] = rx;  
    siz[rx] += siz[ry];  
    return true;  
}
```

問題を解いてみよう

 ABC177Dを解いてみよう

他にもUnion-Findで解ける問題

 ABC264E

 ABC097D

 ARC032B

Union-Findを使いこなそう

Union-Findでできること

Union-Findの用途は色々ありますが上で述べた操作をそのまま行う以外の用途で、ここでは二つ代表的なもの(だと自分が思っていること)をあげます。

- ▶ 無向辺からなるグラフの閉路検出
- ▶ 最小全域木の構成(クラスカル法)

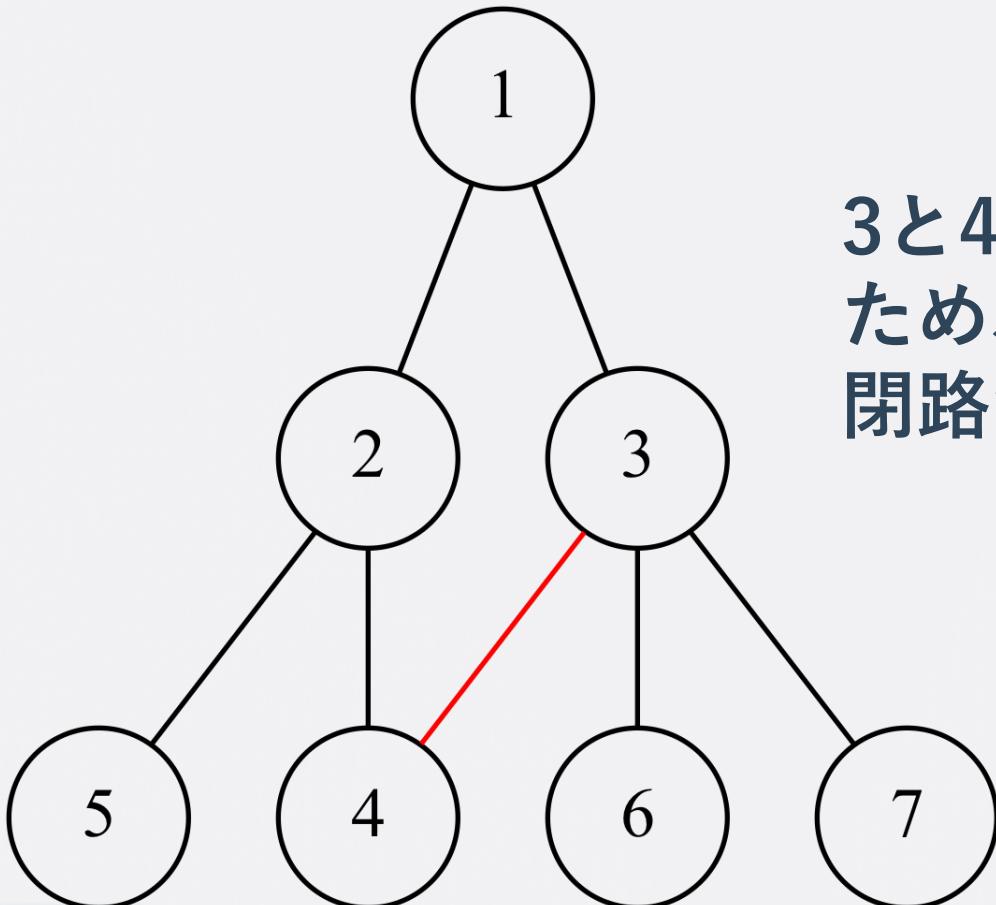
無向辺からなるグラフの閉路検出

頂点 x と y が連結でない場合、 $\text{unite}(x,y)$ で x と y を結んで連結にします。

頂点 x と y が既に連結な時に、新たに頂点 x と y を結ぶ辺が存在したときに閉路が存在することがわかります。

x と y を含む木が頂点数が変わらずに新たに辺が追加されることで必ず閉路ができます。

閉路検出のイメージ



3と4は既に同じグループに存在しているため、新しく3と4に辺をはると(赤の線)閉路が新しくできてしまします。

閉路検出の実装

```
std::vector<std::pair<int, int>> edge(m);
for(int i = 0; i < m; ++i){
    std::cin >> edge[i].first >> edge[i].second;
}
//trueの時に閉路が存在
bool cycle = false;
for(int i = 0; i < m; ++i){
    if(unite(x, y) == false){
        cycle = true;
    }
}
```

全域木

全域木とは無向辺かつ全ての頂点が互いに行き来可能(連結)なN頂点M辺のグラフをグラフが連結である条件を保ったまま辺を削除したグラフのことを指します。

この時、全域木は木であり辺の数はN-1個となります。
全域木は1通りとは限りません。

最小全域木

頂点 x, y を結ぶ辺を張るコストが c の時、全ての頂点を連結にする(全域木を構成する)最小のコストで構成される全域木が最小全域木になります。

最小全域木は必ずしも1通りとは限りません。 (x, y) を結ぶ操作のコストが全て異なる場合は1通り)

クラスカル法

クラスカル法は最小全域木と構成に必要な最小コストを求めることができます。

1. コストが昇順になるようにソート
2. 頂点 x と y が連結でないとき、辺を張る そうでない時は辺を張らない

クラスカル法

`unite(x, y)`は既に連結な時に`false`を返すだけなので、順番に`unite(x, y)`を呼び出しの返り値が`true`の時のみコストを加算するだけでクラスカル法ができます。

クラスカル法の実装(1/2)

```
std::vector<std::tuple<int,int,int>> edge(m);
for(int i = 0;i < m;++i){
    int x, y, c;
    std::cin >> x >> y >> c;
    --x;--y;
    edge[i] = std::make_tuple(c, x, y);
}
UnionFind uf(n);
std::sort(edge.begin(), edge.end());
```

クラスカル法の実装(2/2)

```
int total_cost = 0;
for(int i = 0;i < m; ++i){
    int x = std::get<1>(tuple[i]), y = std::get<2>(tuple[i]);
    int c = std::get<0>(tuple[i]);

    if(uf.unite(x, y)){
        total_cost += c;
    }
}
```

問題を解いてみよう

 ABC231D

 ABC235E

の二つを解いてみよう

他にもクラスカル法を使う問題

 ABC210E

 ARC029C

以上で終わりです。