# 并查集

## 定义

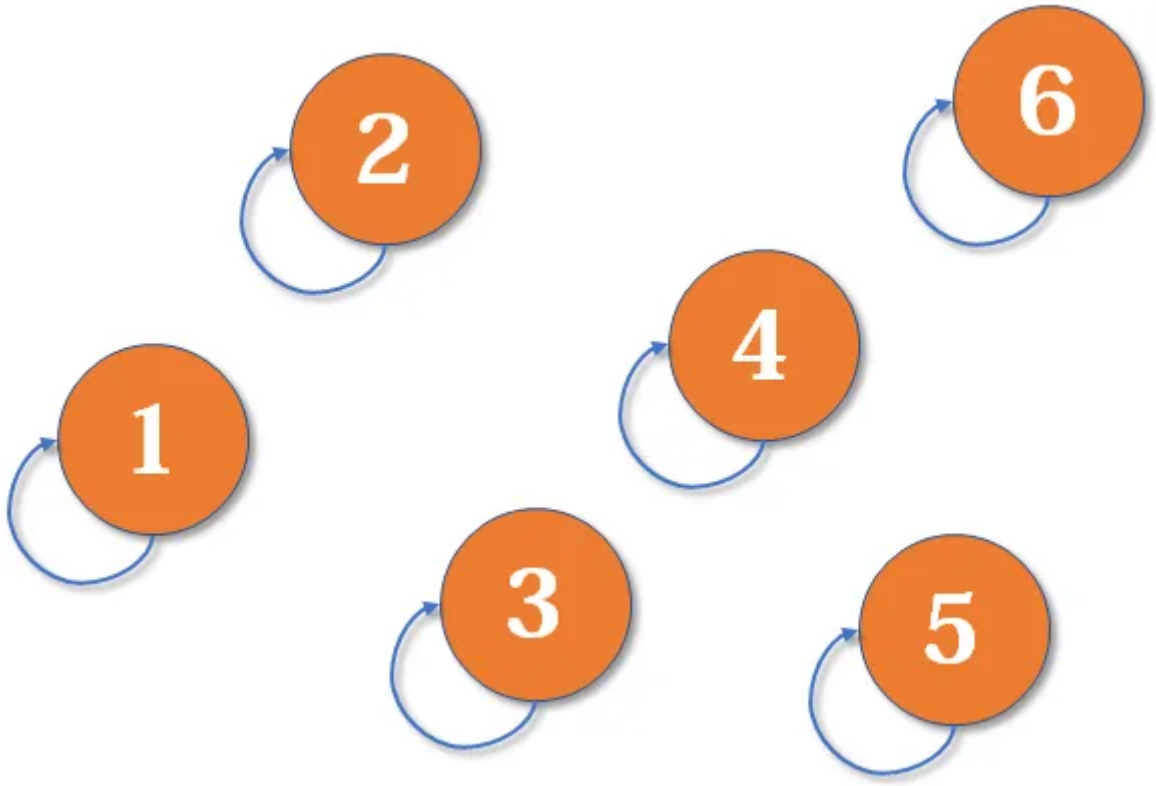
并查集是一种用于管理**元素所属集合**的数据结构，实现为一个**森林**，其中每棵树表示一个集合，树中的节点表示对应集合中的元素。

顾名思义，并查集支持两种操作：

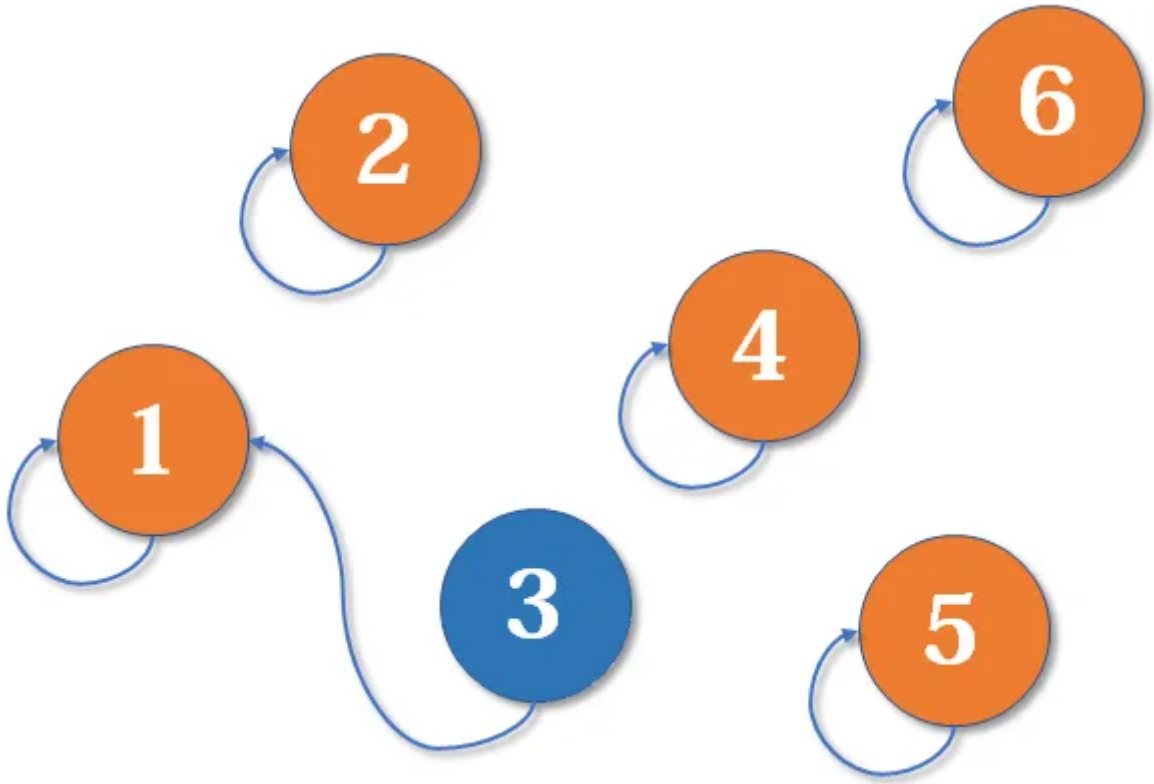
- 合并（Union）：合并两个元素所属集合（合并对应的树）
- 查询（Find）：查询某个元素所属集合（查询对应的树的根节点），这可以用于判断两个元素是否属于同一集合

并查集的重要思想在于，**用集合中的一个元素代表集合**。举个例子，把集合比喻成**帮派**，而代表元素则是**帮主**。接下来我们利用这个比喻，看看并查集是如何运作的。

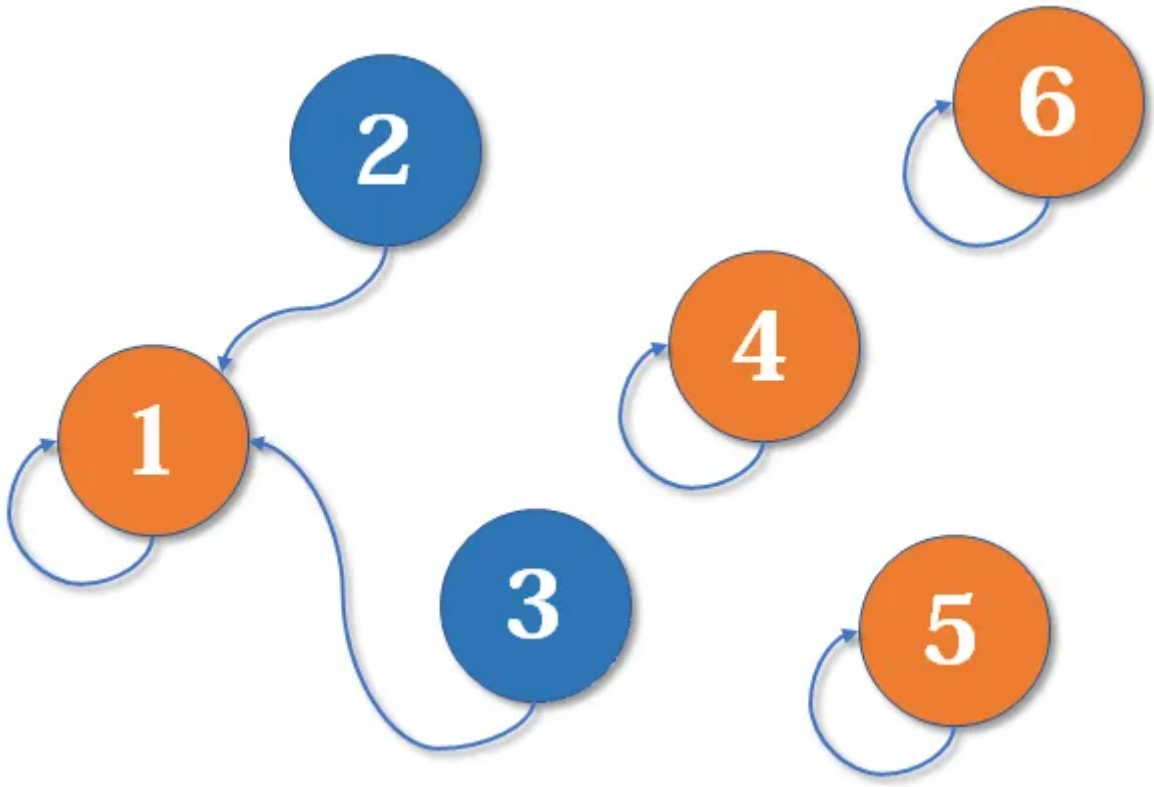
最开始，所有大侠各自为战。他们各自的帮主自然就是自己。（对于只有一个元素的集合，代表元素自然是唯一的那个元素）



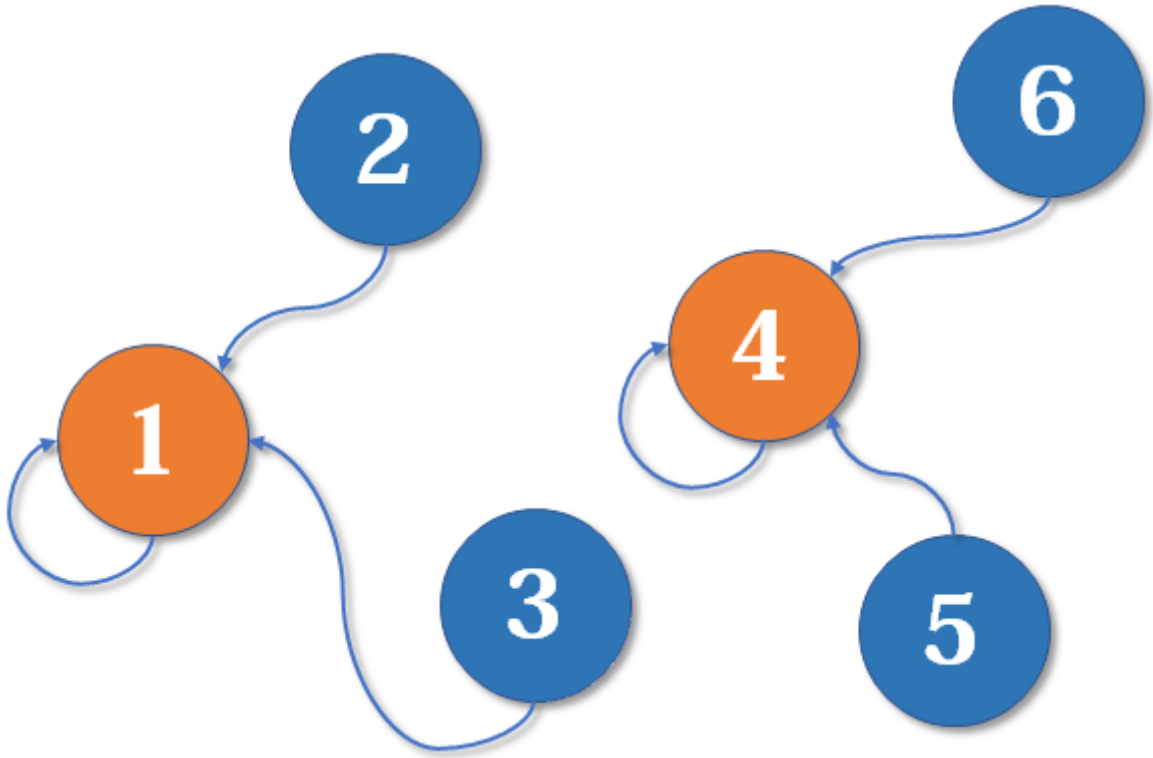

现在1号和3号比武，假设1号赢了，那么3号就认1号作帮主（合并1号和3号所在的集合，1号为代表元素）。



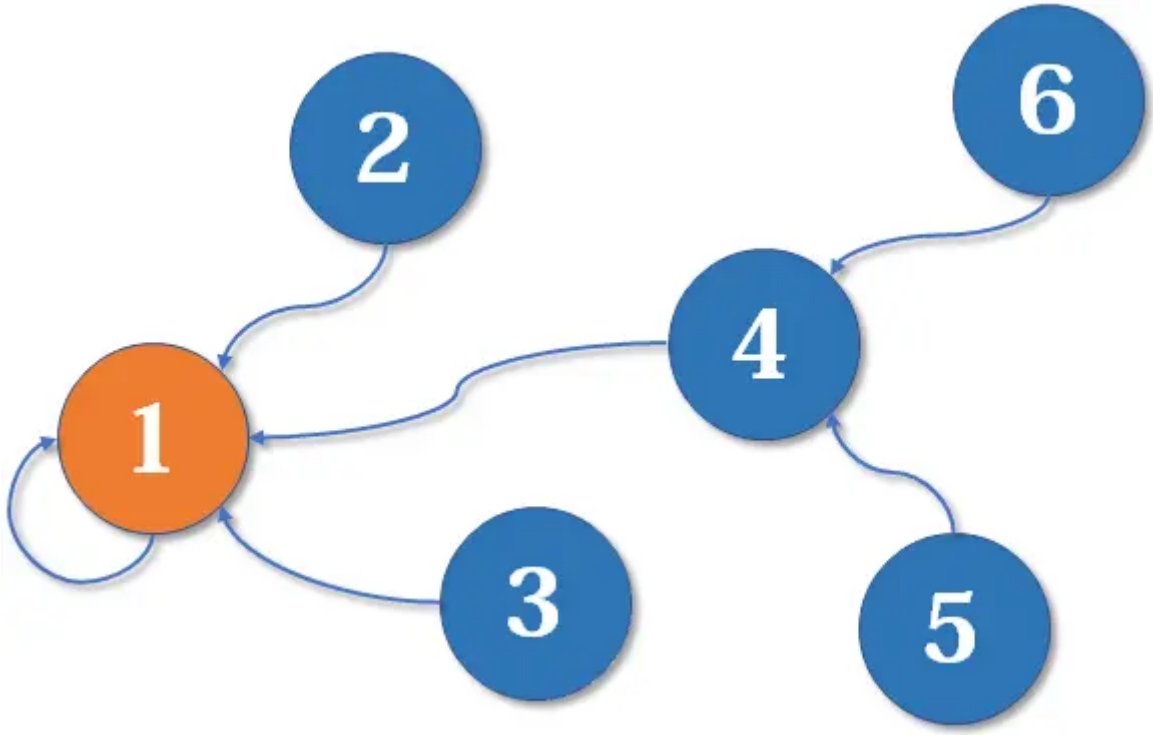

现在2号想和3号比武（合并3号和2号所在的集合），但3号表示，别跟我打，让我帮主来收拾你（合并代表元素）。不妨设这次又是1号赢了，那么2号也认1号做帮主。




现在我们假设4、5、6号也进行了一番帮派合并，江湖局势变成下面这样：




现在假设2号想与6号比，跟刚刚说的一样，喊帮主1号和4号出来打一架。1号胜利后，4号认1号为帮主，当然他的手下也都是跟着投降了。




从这个比喻中，我们可以看出来，每个大侠都**有且仅有一个直接**追随的帮主，并且不存在**循环帮主**，因此，我们并查集的结构是一个森林，对于每个节点，它们都具有**唯一的父亲**，特殊的，根节点的父亲是它本身

## 操作

### 初始化

一开始，我们并查集里面的元素都是**独立的**，因此我们很简单地能得到初始化的代码：

```
// 数组版本
int fa[maxn];
for (int i = 1; i <= n; i++) fa[i] = i;

// vector + STL版本
vector fa(n + 1, 0);
iota(fa.begin(), fa.end(), 0);
```

其中，`std::iota(begin, end, value)`函数，可以以**递增**的值初始化某一段连续区间$[begin, end)$，并且$begin = value,\ begin + 1 = val + 1, \cdots$

### 查询

对于每个集合，我们需要查询到该集合的**根**，根的特征是$fa[u] = u$，因此，我们能很简单地写出一个递归的查询函数

```
int find(int x) {
    return x == fa[x] ? x : find(fa[x]);
}
```

### 合并

从上文中我们也能看出，两个集合合并，相当于**将一棵树的树根连到另一棵树的树根上**，既然我们用`find`函数能很简单地查询到某个集合的树根，那么，我们也能很简单地写出合并函数

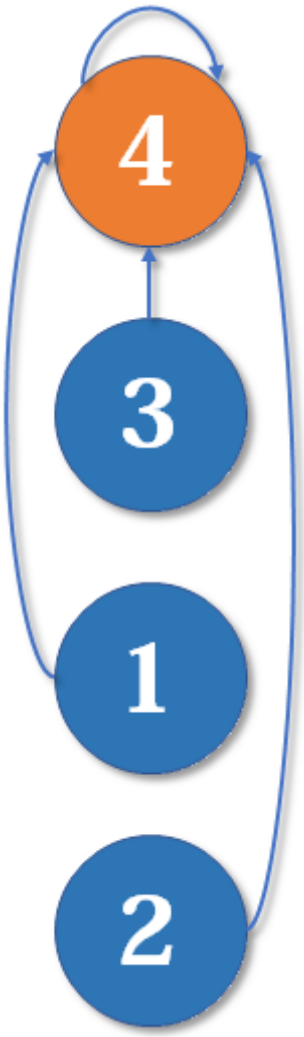
```
void merge(int u, int v) {
    fa[find(u)] = find(v);
}
```

## 优化

### 路径压缩

假设我们合并的结果中，有一条长度为$n$的**链**，我们每次查询都是查询**叶子**元素，这样，我们每次都会花费$O(n)$的时间去遍历这条链，这样我们查询花费代价就会比较大


怎么解决呢？我们可以使用**路径压缩**的方法。既然我们只关心一个元素对应的**根节点**，那我们希望每个元素到根节点的路径尽可能短，最好只需要一步，像这样：




我们在**递归查询**的过程中，在递归终点是找到了我们的根节点的，因此，只需要一步步地将我们的根节点，设为我们路径上所有点的**父亲节点**，就可以达到**路径压缩**的目的

```
int find(int x) {
    return (x == fa[x]) ? x : (fa[x] = find(fa[x]));
}
```

在只使用**路径压缩**的情况下，平均操作时间复杂度为$O(\log n)$

#### 启发式合并

合并两棵分别以$x,y$**为根**的树时，我们通常可以选用**启发式合并**的做法

记$siz[x]$为以$x$为根的树的大小，如果说$siz[x] > siz[y]$，我们就在合并时，将$fa[y] = x$，即将$y$作为$x$的子树

$siz$数组的维护比较简单，首先初始化成1，代表每棵树都只有一个节点

在合并是，例如上文中将$y$作为$x$的子树，我们就令$siz[x] += siz[y]$，代表$y$及其子树成为了$x$子树的一部分

```
void merge(int u, int v) {
    int fu = find(u), fv = find(v);
    // important
    if (u == v) return;
    if (siz[fu] < siz[fv]) swap(fu, fv);
    fa[fv] = fu; siz[fu] += siz[fv];
}
```

在只使用**启发式合并**的情况下，平均操作时间复杂度为$O(\log n)$

在**同时使用启发式合并和路径压缩**的情况下，平均操作时间复杂度为$O(\alpha(n))$

其中，$\alpha(n)$是阿克曼函数的反函数，其某些常用函数值如下：

$$\alpha(x)=\begin{cases}0 & 0\le x\le 2\\ 1 & x=3\\ 2 & 4\le x\le 7\\ 3 & 8\le x\le 2047\\ 4 & 2048\le x\le A_4(1)\ (2^{2058}<A_4(1))\end{cases}$$

因此，在我们讨论的范围内，$\alpha(n)$不会超过$4$，因此通常会被视为**常数**

## # 49805 [「CF EDU 并查集 Step1 A」Disjoint Sets Union]

**题目描述**

给定$n$个元素，它们初始分别在在$n$个集合中

我们有以下两种操作

- `union u v`，表示将$u,v$所在的集合**合并**
- `get u v`，表示检查$u,v$是否处于**同一集合**

**输入格式**

第一行输入两个整数$n,m(1\le n,m\le 10^5)$，代表元素个数和操作次数

接下来$m$行，对`union`操作，输入格式为union $u$ $v$ $(1\le u,v\le n)$，对`get`操作，输入格式为get $u$ $v$ $(1\le u,v\le n)$

**输出格式**

对于每个`get`操作，输出一行一个字符串，`YES`代表$u,v$处于同一集合，`NO`代表$u,v$处于不同集合

**样例输入**

```
4 4
union 1 2
union 1 3
get 1 4
get 2 3
```

**样例输出**

```
NO
YES
```

**Solution**

并查集板子题，根据并查集实现即可

**Code**

```
void solve() {
    int n, q; cin >> n >> q;
    vector fa(n + 1, 0);
    iota(all(fa), 0);

    auto find = [&](int x, auto find) -> int {
        return fa[x] == x ? x : (fa[x] = find(fa[x], find));
    };

    while (q--) {
        string t;
        int u, v;
        cin >> t >> u >> v;
        if (t == "union") {
            auto fu = find(u, find);
            auto fv = find(v, find);
            fa[fu] = fv;
        }
        else {
            auto fu = find(u, find);
            auto fv = find(v, find);
            cout << (fu == fv ? "YES\n" : "NO\n");
        }
    }
}
```

## # 49806 「CF EDU 并查集 Step1 B」Disjoint Sets Union 2

**题目描述**

给定$n$个元素，它们初始分别在在$n$个集合中

我们有以下两种操作

- `union u v`，表示将$u, v$所在的集合**合并**
- `get u`，表示检查$u$所在的集合，找到集合中的**最小元素，最大元素，以及集合中的总元素数**

**输入格式**

第一行输入两个整数$n, m(1 \leq n, m \leq 3 \times 10^5)$，代表元素个数和操作次数

接下来$m$行，对`union`操作，输入格式为union $u$ $v$ $(1 \leq u, v \leq n)$，对`get`操作，输入格式为 get $u$ $(1 \leq u \leq n)$

**输出格式**

对于每个`get`操作，输出三个数，$u$所在集合中的**最小元素，最大元素，以及集合中的总元素数**

**样例输入**

```
5 11
union 1 2
get 3
get 2
union 2 3
get 2
union 1 3
get 5
union 4 5
get 5
union 4 1
get 5
```

**样例输出**

```
3 3 1
1 2 2
1 3 3
5 5 1
4 5 2
1 5 5
```

**Solution**

我们用$minn[i]$记录集合$i$中，元素的最小值

很显然，在两个集合$u, v$合并时，我们设将$v$集合合并到$u$集合中，那么，我们$u$集合的最小值$minn[u]$，就应该按$minn[u] = \min(minn[u], minn[v])$更新

对于集合$i$中的元素最大值$maxx[u]$同理

而对于集合大小，我们用$siz[i]$表示集合$i$中，元素的个数

在两个集合$u, v$合并时，我们设将$v$集合合并到$u$集合中，那么，我们$u$集合的大小，需要加上$v$集合的大小，即$siz[u] += siz[v]$

**Code**

```
void solve() {
    int n, q; cin >> n >> q;
    vector fa(n + 1, 0); iota(all(fa), 0);
    vector minn = fa, maxx = fa, siz(n + 1, 1);
```

```
    auto find = [&](int x, auto find) -> int {
        return fa[x] == x ? x : (fa[x] = find(fa[x], find));
    };

    auto merge = [&](int u, int v) {
        auto fu = find(u, find), fv = find(v, find);
        if (fu == fv) return;
        if (siz[fu] < siz[fv]) swap(fu, fv);
        fa[fv] = fu; siz[fu] += siz[fv];
        cmax(maxx[fu], maxx[fv]); cmin(minn[fu], minn[fv]);
    };

    while (q--) {
        string t;
        int u, v;
        cin >> t >> u;
        if (t == "union") {
            cin >> v; merge(u, v);
        }
        else {
            auto fu = find(u, find);
            cout << minn[fu] << ' ' << maxx[fu] << ' ' << siz[fu] << '\n';
        }
    }
}
```

## # 49807 「CF EDU 并查集 Step1 C」Experience

**题目描述**

在一款新型在线游戏中，玩家像往常一样与怪物战斗并获得经验值。玩家们可以组成团队共同对抗怪物。在怪物被消灭后，团队中的所有玩家将获得**相同数量**的经验值。游戏的特殊之处在于团队不可拆分，也不能有人离开团队。唯一支持的操作是将两个团队合并在一起。

由于游戏中已经有很多玩家，请你来维护所有玩家的经验值。

最初，每个玩家都有$0$经验值，并且每个玩家都是**独立的队伍的唯一成员**。

**输入格式**

第一行两个整数$n, m(1 \leq n, m \leq 2 \times 10^5)$，表示玩家的数量和操作的数量

此后$m$行，根据操作不同，输入如下所示：

- $\text{join}\ x\ y$，代表把玩家$x$和玩家$y$所在的团队合并（如果玩家已经处于同一团队，则不用进行任何操作）
- $\text{add}\ x\ v$，代表添加$v(1 \leq v \leq 100)$点经验给玩家$x$所在团队的**每一位成员**
- $\text{get}\ x$，代表查询玩家$x$的经验值，**并输出**

**输出格式**

对于每个get $x$操作，输出一行一个整数，代表玩家$x$此时的经验值

**样例输入**

```
3 6
add 1 100
join 1 3
add 1 50
get 1
get 2
get 3
```

**样例输出**

```
150
0
50
```

**Solution**

首先考虑没有 join 操作的场景

对于一次**增加经验**操作，我们只需要加给利用并查集维护的根节点，查询某个人的经验时，只需要查询其对应的根节点的经验即可，记维护数组为$scr$

那么，对于 join 操作而言，假设我们将以$x$为根的集合，合并进入以$y$为根的集合，那么，对于$x$中的某个元素$x_i$而言，其实际的经验值应该是$scr[x]$，在合并进入以$y$为根的集合后，它的经验值变为了$scr[y]$，为了避免该种改变，我们需要使用一个新数组$ext$，记录每个元素的经验值和它所在集合的根元素的经验值的**差值**。

因此，在 join 操作时，我们需要对于每个**被合并**的集合中的元素$x_i$，将$ext[x_i]$加上$src[y]-src[u]$，从而避免更换根节点对总经验的影响

查询时，只需要输出$ext[x]+scr[fx]$即可（$fx$代表$x$所在集合的根节点）

注意，由于此题需要在**根元素**维护其集合内所有的元素（否则无法在合并的时候快速找到所有在集合中的元素从而维护$ext$数组），因此需要用到**启发式合并**优化

**Code**

```
void solve() {
    int n, q; cin >> n >> q;
    vector siz(n + 1, 1), fa(n + 1, 0), ext(n + 1, 0), scr(n + 1, 0);
    vector son(n + 1, vector(0, 0));

    iota(all(fa), 0);
    for (int i = 1; i <= n; i++) son[i].push_back(i);

    auto find = [&](int x, auto find) -> int {
        return fa[x] == x ? x : (fa[x] = find(fa[x], find));
    };

    auto merge = [&](int uu, int vv) {
```

```
        auto fu = find(uu, find), fv = find(vv, find);
        if (fu == fv) return;
        if (siz[fu] < siz[fv]) swap(fu, fv);
        fa[fv] = fu; siz[fu] += siz[fv];
        for (auto v: son[fv]) {
            ext[v] += scr[fv]; ext[v] -= scr[fu];
            son[fu].push_back(v);
        }
    };

    while (q--) {
        string t;
        int u, v;
        cin >> t >> u;
        if (t == "add") {
            cin >> v; u = find(u, find);
            scr[u] += v;
        }
        else if (t == "join") {
            cin >> v; merge(u, v);
        }
        else {
            auto fu = find(u, find);
            cout << ext[u] + scr[fu] << '\n';
        }
    }
}
```

## # 49808 「CF EDU 并查集 Step1 D」Cutting a graph

**题目描述**

存在一个无向图和两种类型的操作序列，格式如下：

- cut $u$ $v$ —— 从图中移除边$< u, v >$；
- ask $u$ $v$ —— 检查顶点 $u$ 和 $v$ 是否联通。

在所有操作应用之后，图中不包含任何边。请找出每一个类型为 ask 的操作的结果。

**输入格式**

输入第一行有三个整数，$n, m, k (1 \le n \le 5 \times 10^4, 0 \le m \le 10^5, m \le k \le 1.5 \times 10^5)$，代表图中的顶点数，边数和操作数

接下来$m$行，每行两个整数$u, v (1 \le u_i, v_i \le n)$，代表$u, v$之间有一条边，保证不存在自环和重边

接下来$k$行，描述了以下格式的操作

- cut $u$ $v (1 \le u_i, v_i \le n)$，从图中移除边$< u, v >$
- ask $u$ $v (1 \le u_i, v_i \le n)$，检查顶点 $u$ 和 $v$ 是否联通

**输出格式**

对于每个 ask 操作，输出一行一个字符串，`YES`表示$u, v$联通，`NO`表示$u, v$不连通

**样例输入**

```
3 3 7
1 2
2 3
3 1
ask 3 3
cut 1 2
ask 1 2
cut 1 3
ask 2 1
cut 2 3
ask 3 1
```

**样例输出**

```
YES
YES
NO
NO
```

### Solution

使用并查集**删边**比较困难，我们考虑反过来做，即**从后到前**处理所有的操作

由于最后所有边是被**删完**了，因此我们初始化并查集的时候，只需要将其初始化成单元素集合即可

将所有操作从后到前执行，就是一个简单的并查集维护联通块的操作了

### Code

```cpp
void solve() {
    int n, m, k; cin >> n >> m >> k;
    vector siz(n + 1, 1), fa(n + 1, 0);

    iota(all(fa), 0);

    auto find = [&](int x, auto find) -> int {
        return fa[x] == x ? x : (fa[x] = find(fa[x], find));
    };

    auto merge = [&](int uu, int vv) {
        auto fu = find(uu, find), fv = find(vv, find);
        if (fu == fv) return;
        if (siz[fu] < siz[fv]) swap(fu, fv);
        fa[fv] = fu; siz[fu] += siz[fv];
    };

    for (int i = 0; i < m; i++) {
        int u, v; cin >> u >> v;
```

```
    }

    struct Q {
        int type;
        int u, v;
    };
    vector qry(k, Q());
    for (auto& [tp, u, v]: qry) {
        string str; cin >> str;
        tp = (str == "ask");
        cin >> u >> v;
    }
    reverse(all(qry));

    vector ans(0, 0);

    for (auto& [tp, u, v]: qry) {
        if (tp) ans.push_back(find(u, find) == find(v, find));
        else merge(u, v);
    }
    reverse(all(ans));

    for (auto v: ans) cout << (v ? "YES\n" : "NO\n");
}
```

## # 10538 「CF EDU 并查集 Step1 E」Monkeys

**题目描述**

有$n$只猴子，在第0秒，每只猴子都挂在树上，其中，1号猴子的尾巴**直接**挂在树上

一开始时，每只猴子的左手和右手都可能拉着另一只猴子的尾巴，每一秒钟（从第0秒开始），就会有一只猴子松开它的左手（或右手），注意，每只猴子的尾巴可能挂了多只猴子

假设猴子的臂力和尾巴的承载能力都是无穷大，请问各个猴子落地的时刻是多少？

**输入格式**

第一行两个整数$n, m(1 \le n \le 2 \times 10^5, 0 \le m \le 4 \times 10^5)$，代表猴子的数量和时刻数（时刻从0到$m-1$）

此后$n$行，每行两个整数$l_i, r_i$，代表第$i$只猴子的左手和右手抓的尾巴所属猴子的编号，特殊的，如果$l_i$或$r_i$为$-1$，代表该猴子的这只手没有抓住任何尾巴

此后$m$行，每行两个数$p_i, h_i(1 \le p_i \le n, h_i \in \{1, 2\})$，代表第$m-1$时刻，编号为$p_i$的猴子松开了左手（$h_i = 1$）或右手($h_i = 2$)

**输出格式**

输出$n$行，每行一个整数$t_i$，表示第$i$只猴子的落地时间，如果第$i$只猴子能一直连在树上，则输出$-1$

**样例输入**

```
4 5
2 2
3 -1
4 -1
-1 2
1 1
2 1
3 1
1 2
4 2
```

**样例输出**

```
-1
3
2
3
```

### Solution

此题仍然属于并查集不好处理的**删边**类型，因此，我们仍然考虑**从后到前**处理所有的操作

首先，我们对并查集的初始化，需要找到所有在$m-1$时刻后**仍然没有放开手的猴子**，将这些猴子所在集合及其手连的尾巴对应的猴子所在的集合合并

随后，我们倒序遍历所有的时间节点，假设在某一时刻，我们发现集合$x$将要和**编号为**1的猴子所在的集合$y$合并，那么，如果按照时间顺序看待操作的话，在该时刻，所有处于集合$x$的猴子都会从树上**摔下**，因此，我们将集合$x$里面的所有元素$x_i$均设为当时的时刻即可

由于要维护某个集合里面所有的元素，因此我们需要开一个$son[x]$数组，存储根节点为$x$的集合里面的所有元素

因此，对于本题而言，我们仍然要用到**启发式合并**

### Code

```
void solve() {
    int n, k; cin >> n >> k;
    vector siz(n, 1), fa(n, 0), ans(n, -1);
    vector son(n, vector(0, 0));
    iota(all(fa), 0);
    for (int i = 0; i < n; i++) son[i].push_back(i);

    auto find = [&](int x, auto find) -> int {
        return fa[x] == x ? x : (fa[x] = find(fa[x], find));
    };

    auto merge = [&](int uu, int vv, int t) {
        if (uu == -2 || vv == -2) return;
        auto fu = find(uu, find), fv = find(vv, find);
        if (fu == fv) return;
        if (siz[fu] < siz[fv]) swap(fu, fv);
```

```
        auto f0 = find(0, find);

        if (fu == f0) for (auto v: son[fv]) ans[v] = t;
        else if (fv == f0) for (auto v: son[fu]) ans[v] = t;

        fa[fv] = fu; siz[fu] += siz[fv];
        son[fu].insert(son[fu].end(), all(son[fv]));
    };

    vector status(n, array<int, 2>()), unr(n, array<int, 2>({1, 1}));
    for (auto& [l, r]: status) {
        cin >> l >> r; l--; r--;
    }

    vector q(k, array<int, 2>());
    for (auto& [id, l]: q) {
        cin >> id >> l; l--; id--;
        unr[id][l] = 0;
    }

    for (int i = 0; i < n; i++) for (int j = 0; j < 2; j++)
        if (unr[i][j]) merge(i, status[i][j], -1);
    for (int i = k - 1; i >= 0; i--) merge(q[i][0], status[q[i][0]][q[i][1]],
i);

    for (auto& v: ans) cout << v << '\n';
}
```

## # 49809 「CF EDU 并查集 Step2 A」People are leaving

**题目描述**

$n$ 人站在 1 至 $n$ 位置。您必须执行两种类型的查询：

- "- $x$ "-- 位置 $x$ 的人离开；
- "? $x$ "-- 找到$x$及其右边最近的还站着的人。

**输入格式**

输入的第一行包含两个整数 $n$ 和 $m$ （ $1 \leq n, m \leq 10^6$ ），人数和查询次数。接下来的 $m$ 行每行一个查询。关于离开的查询，该行格式如 `- x` ( $1 \leq x \leq n$ )。对于最近的人的查询，该行格式如 `? x` ( $1 \leq x \leq n$ )。保证所有离开的人都是不同的。

**输出格式**

按相应顺序每行输出一个`? x`操作的结果。如果右边没有人，则输出 `-1`。

**样例输入**

```
5 10
? 1
- 3
? 3
- 2
? 1
? 2
- 4
? 3
- 5
? 3
```

**样例输出**

```
1
4
1
4
5
-1
```

**Solution**

在删除一个位置后，我们选择将其和其**左边的位置**合并，这样，对于某个连续的集合$[L,R]$，其中除了位置$L$以外的所有位置$L+1,L+2,\cdots R$均已被删除

我们在进行查询$u$时，找到集合中的最小值$L$，如果说$u=L$，代表$u$还没被删去，否则的话，我们找到集合中的**最大值**$R$，此时的$R+1$还没被删去。

但要特别注意的时，$R=n$时，$R+1$不存在，因此我们应该输出`-1`

**Code**

```
void solve() {
    int n, q; cin >> n >> q;
    vector fa(n + 1, 0); iota(all(fa), 0);
    auto maxx = fa;

    auto find = [&](int x, auto find) -> int {
        return fa[x] == x ? x : (fa[x] = find(fa[x], find));
    };

    auto merge = [&](int u, int v) {
        auto fu = find(u, find), fv = find(v, find);
        if (fu == fv) return;
        fa[fv] = fu; cmax(maxx[fu], maxx[fv]);
    };

    while (q--) {
        string t;
        int u;
        cin >> t >> u;
        if (t == "?") {
```

```
            auto fu = find(u, find);
            if (fu == u) cout << u << '\n';
            else if (maxx[fu] == n) cout << "-1\n";
            else cout << maxx[fu] + 1 << '\n';
        }
        else merge(u - 1, u);
    }
}
```

## # 49810 「CF EDU 并查集 Step2 B」Parking

**题目描述**

在一个编号从 1 到 $n$ 的**圆形**停车场上有 $n$ 个车位。有 $n$ 辆车想按顺序停放。第 $i$ 辆汽车想停在 $p_i$ 个车位上。如果汽车开到一个车位，而这个车位已被占用，它就会**向后开**，直至找到第一个空车位上停车。

**输入格式**

第一行包含整数 $n$（$1 \le n \le 300\,000$），表示停车场大小和汽车数量。

第二行包含 $n$ 个整数，$p_i$（$1 \le p_i \le n$）汽车 $i$ 希望占用的车位。

**输出格式**

输出 $n$ 个数字。第 $i$ 个数字是实际上被第 $i$ 辆汽车占用的停车位。

**样例输入**

```
3
2 2 2
```

**样例输出**

```
2 3 1
```

**Solution**

我们用一个 $nxt$ 数组维护第 $i$ 个车位的**下一个**位置

一开始，$nxt[1]=2, nxt[2]=3 \cdots, nxt[n]=1$。在某个车位 $u$ 从没被占用到被占用时，显然，它只需要和它的**前一个车位**所在的集合 $fu$ 进行合并，并且将 $nxt[fu]$（本来 $nxt[fu]=u$）设为 $nxt[u]$即可

在寻找车位 $u$ 时，如果我们发现 $u$ 没被占用，那么我们直接停在这个位置即可，否则，我们就需要停在 $nxt[fu]$ 这个位置。本段话中的 $fu$，是 $u$ 所在集合的**代表元素**

**Code**

```
void solve() {
    int n, q; cin >> n;
    vector fa(n + 1, 0), nxt = fa; iota(all(fa), 0);
    for(int i = 1; i <= n; i++) nxt[i] = (i % n) + 1;
```

```
    auto find = [&](int x, auto find) -> int {
        return fa[x] == x ? x : (fa[x] = find(fa[x], find));
    };

    auto merge = [&](int u, int v) {
        auto fu = find(u, find), fv = find(v, find);
        if (fu == fv) return;
        fa[fv] = fu; nxt[fu] = nxt[fv];
    };

    for (int i = 0; i < n; i++) {
        int u; cin >> u;
        auto fu = find(u, find);
        if (fu == u) {
            cout << u << ' ';
            merge((fu + n - 2) % n + 1, fu);
        }
        else {
            cout << nxt[fu] << ' ';
            merge(fu, nxt[fu]);
        }
    }
}
```

## # 49811 「CF EDU 并查集 Step2 C」Restructuring Company

**题目描述**

大型软件公司有 $n$ 名员工。每个人都隶属于某个部门。最初，每个人都在自己的部门负责自己的项目(因此，每个公司最初由 $n$ 个部门组成，每个部门有一个人)。

然而，严酷的时代已经来临，公司管理层不得不聘请一位危机管理者来重建工作流程，以提高效率。让我们用 $team(u)$ 来代表 $u$ 工作的团队。危机管理者可以做出两种决策：

1. 将部门 $team(x)$ 和 $team(y)$ 合并为一个大部门，其中包含 $team(x)$ 和 $team(y)$ 的所有员工，其中 $x$ 和 $y(1 \le x, y \le n)$ 是某些公司员工中两个人的编号。如果 $x$ 和 $y$ 处于一个部门，则不会发生任何事。
2. 合并部门 $team(x), team(x+1), \cdots, team(y)$ ，其中 $x$ 和 $y(1 \le x \le y \le n)$ 是某些公司员工中两个人的编号。

在这种情况下，危机管理者有时会怀疑员工 $x, y$ 是否在同一部门工作。

请帮助危机管理者整理公司结构并回答他的所有疑问

**输入格式**

输入的第一行包含两个整数 $n$ 和 $q(1 \le n \le 2 \times 10^5, 1 \le q \le 5 \times 10^5)$，分别代表公司员工人数和危机经理的查询次数。

接下来的 $q$ 行包含危机经理的查询。每个查询格式如 `type x y`，其中 $type \in \{1, 2, 3\}$。如果是 $type = 1, 2$ ，则查询表示危机管理者做出了决策一或决策二。如果是 $type = 3$ ，则您的任务是确定员工 $x$ 和 $y$ 是否在同一部门工作。

请注意，在任何类型的查询中， $x$ 都可以等于 $y$ 。

**输出格式**

对于每组 $type = 3$ 的查询，如果两个员工处于一个部门，则输出 `YES` ，否则输出 `NO`

**样例输入**

```
8 6
3 2 5
1 2 5
3 2 5
2 4 7
2 1 2
3 1 7
```

**样例输出**

```
NO
YES
YES
```

**Solution**

对于查询一和三我们很好地利用并查集维护，但对于查询二我们需要特殊处理

对于某个员工 $u$ 而言，我们维护一个 $nxt$ 数组，表示和该员工处于不同部门的，编号比该员工大的最小编号

那么，对于查询二而言，我们就可以从 $x + 1$ 这个员工开始，不断地向前一个员工合并，并且将 $nxt[i]$ 设为 $nxt[v]$，代表第 $i$ 个员工及其所在的集合合并进入员工 $i - 1$ 所在的集合

对于 $i$ 的遍历，我们从 $x + 1$ 开始，每次直接**跳到** $i = nxt[i]$ （修改前的），这样就能保证，我们每个点至多会被执行 $O(1)$ 次操作二，这样就能保证整体的时间复杂度

**Code**

```
void solve() {
    int n, q; cin >> n >> q;
    vector fa(n + 1, 0), nxt = fa; iota(all(fa), 0);
    for(int i = 1; i <= n; i++) nxt[i] = i + 1;

    auto find = [&](int x, auto find) -> int {
        return fa[x] == x ? x : (fa[x] = find(fa[x], find));
    };

    auto merge = [&](int u, int v) {
        auto fu = find(u, find), fv = find(v, find);
        fa[fv] = fu;
    };

    while (q--) {
        int tp, u, v;
        cin >> tp >> u >> v;
        if (tp == 1) merge(u, v);
        else if (tp == 2) {
```

```
            for (int i = u + 1, xx; i <= v; i = xx) {
                merge(i - 1, i);
                xx = nxt[i]; nxt[i] = nxt[v];
            }
        }
        else
            cout << (find(u, find) == find(v, find) ? "YES\n" : "NO\n");
    }
}
```

## # 49812 「CF EDU 并查集 Step2 D」Bosses

**题目描述**

一家公司有 $n$ 名员工，一开始，没有人是其他人的下属。也就是说，每个员工都是自己的老板。如果一个人不是任何人的下属，我们就称他为老板。

您需要处理两类查询：

- 老板 $a$ 成为老板 $b$ 的下属（老板不再是老板），
- 给定员工 $c$，我们应该通过他的多少个上级才能找到老板？

在第二种类型的查询中，如果 $c$ 是老板，则答案为 $0$，否则答案为雇员的**深度**。

**输入格式**

第一行包含两个整数 $n, m (1 \leq n, m \leq 3 \times 10^5)$，分别是员工人数和查询次数。

接下来的 $m$ 行每行一个查询。

第一种类型的查询描述为 `1 a b`。$(1 \leq a \neq b \leq n)$。

第二种类型的查询描述为 `2 c`。$(1 \leq c \leq n)$

**输出格式**

对每个第二种类型的查询，输出一行一个字符串，包含该查询的答案

**样例输入**

```
10 20
1 9 4
1 2 6
2 10
1 10 5
2 5
1 7 4
1 8 5
2 1
1 6 5
1 3 5
1 1 4
1 5 4
2 7
2 2
2 4
```

```
2 3
2 4
2 2
2 2
2 10
```

**样例输出**

```
0
0
0
1
3
0
2
0
3
3
2
```

**Solution**

本题和Step 1 C较为类似

定义 $fx$ 为$x$ 结点的老板

首先我们考虑较小的集合 $v$ 成为较大的集合 $u$ 的**下属**的情况：

- 维护两个数组$ext, scr$，其中某个点 $u$ 的**深度**由 $ext[u]+scr[fu]$得到
- 维护较小集合中所有元素的 $ext$ 值，确保 $ext\_new[v_i]+scr[fu]=ext\_old[v_i]+scr[fv]+1$，代表其深度在原来的基础上增加了 $1$

那较大的集合 $v$ 成为较小的集合 $u$ 的下属的情况呢？从**启发式合并**一节我们能知道，两个集合合并时，如果时间复杂度为 $O(|大集合|)$，则合并完全的时间复杂度不能保证为 $O(n\log(n))$，因此，我们仍然考虑修改小集合 $u$ 中的元素，从而保证复杂度

我们在维护小集合 $u$ 的值时，仍然分两步进行：

- 修改集合 $u$ 中所有点的 $ext[u_i]$，保证 $src[fv]+1+ext\_new[u_i]=src[fu]+ext\_old[u_i]$
- 将 $scr[fu]$ 修改为 $scr[fv]+1$，这样相当于保证集合 $v$ 中所有点深度 $+1$

注意维护某个节点的**所有子节点**时，我们仍然选择向**大集合**添加**小集合**，再用`std::swap`保证时间复杂度

**Code**

```cpp
void solve() {
    int n, q; cin >> n >> q;
    vector siz(n + 1, 1), fa(n + 1, 0), ext(n + 1, 0), scr(n + 1, 0);
    vector son(n + 1, vector(0, 0));

    iota(all(fa), 0);
    for (int i = 1; i <= n; i++) son[i].push_back(i);

    auto find = [&](int x, auto find) -> int {
```

```
        return fa[x] == x ? x : (fa[x] = find(fa[x], find));
    };

    auto merge = [&](int uu, int vv) {
        auto fu = find(uu, find), fv = find(vv, find);
        if (fu == fv) return;

        if (siz[fu] < siz[fv]) {
            for (auto v: son[fu]) {
                ext[v] += scr[fu]; ext[v] -= scr[fv] + 1;
            }

            scr[fu] = scr[fv] + 1;
            son[fv].insert(son[fv].end(), all(son[fu]));
            swap(son[fu], son[fv]);
        }
        else {
            for (auto v: son[fv]) {
                ext[v] += scr[fv] + 1; ext[v] -= scr[fu];
            }

            son[fu].insert(son[fu].end(), all(son[fv]));
        }

        fa[fv] = fu; siz[fu] += siz[fv];
    };

    while (q--) {
        int type; cin >> type;
        if (type == 1) {
            int u, v; cin >> u >> v;
            merge(v, u);
        }
        else {
            int c; cin >> c;
            cout << ext[c] + scr[find(c, find)] << '\n';
        }
    }
}
```

## # 49813 「CF EDU 并查集 Step2 E」Spanning Tree

**题目描述**

对于给定的连通带权无向图，找出权重之和最小的生成树。

**输入格式**

输入的第一行包含两个整数 $n$ 和 $m$（$2 \le n \le 200\,000, 1 \le m \le 200\,000$），分别是顶点和边的数量。

接下来的 $m$ 行每行描述一条边，格式如下：三个整数 $b_i$， $e_i$ 和 $w_i$ （$1 \le b_i, e_i \le n$、$0 \le w_i \le 100\,000$）--分别是边的两个端点和该边的权重。

**输出格式**

输出一行一个整数，表示最小生成树的边权之和

**样例输入**

```
4 4
1 2 1
2 3 2
3 4 5
4 1 4
```

**样例输出**

```
7
```

**Solution**

**最小生成树**是并查集的一个简单应用，对于最小生成树而言，只要它**每一条边**的边权就是最小的，则其边权之和一定是最小的。

由于并查集的结构类似**森林**，由树的性质我们可以知道，对于处于**同一棵树的节点**，如果我们将其连边，则它们所处的树就不再是一棵树。

因此，我们将边按照边权从小到大排序，并遍历所有的边，如果构成该边的两个节点 $u, v$ 处在同一集合中（即 $u, v$ 已经联通），那我们不需要进行任何操作，否则我们就将 $u, v$ 所在的集合合并，最后得到的边权之和就是我们最小生成树的边权之和

这就是求得一张图的最小生成树的 **Kruskal** 算法，时间复杂度为 $O(\alpha(n)m\log m)$

**Code**

```
void solve() {
    int n, m; cin >> n >> m;

    struct edge {
        int u, v, w;
    };

    vector e(m, edge());
    for (auto& [u, v, w]: e) cin >> u >> v >> w;
    sort(all(e), [&](auto& a, auto& b){
        return a.w < b.w;
    });

    vector fa(n + 1, 0); iota(all(fa), 0);
    vector siz(n + 1, 1);

    auto find = [&](int x, auto find) -> int {
        return fa[x] == x ? x : (fa[x] = find(fa[x], find));
    };

    auto merge = [&](int u, int v) {
        auto fu = find(u, find), fv = find(v, find);
```

```
        if (fu == fv) return;
        if (siz[fu] < siz[fv]) swap(fu, fv);
        fa[fv] = fu; siz[fu] += siz[fv];
    };


    ll ans = 0;

    int t = n - 1;
    for (auto& [u, v, w]: e) {
        auto fu = find(u, find), fv = find(v, find);
        if (fu == fv) continue;
        merge(fu, fv); ans += w;

        t--;
        if (!t) break;
    }

    cout << ans << endl;
}
```

## # 49814 「CF EDU 并查集 Step2 F」Dense spanning tree

**题目描述**

您需要在图中找到一棵生成树，使得其中最大边权与最小边权之间的差值最小

**输入格式**

第一行包含两个整数 $n, m(2 \leq n \leq 1000, 0 \leq m \leq 10^4)$，分别代表顶点数和边数

此后 $m$ 行每行三个整数 $u_i, v_i, w_i$，代表一条端点为 $u_i, v_i(1 \leq u_i \neq v_i \leq n)$，权值为 $w_i(-10^9 \leq w_i \leq 10^9)$的边

**输出格式**

首先输出一行一个字符串，`YES` 代表生成树存在，`NO` 代表生成树不存在；

如果生成树存在，在第二行输出最大边权与最小边权之间的最小差值

**样例输入**

```
4 5
1 2 1
1 3 2
1 4 1
3 2 2
3 4 2
```

**样例输出**

```
YES
0
```

**Solution**

我们可以对边按权值进行排序后，简单地枚举**权值最小**的边 $e_L$，然后从区间 $[L, m]$从小到大进行 **Kruskal** 算法，如果对于某条边 $e_R$，将其添加进去后整个图联通了（即已经添加了 $n-1$ 条边），此时我们计算他们的权值差 $|w_L - w_R|$，并和答案取 $\min$ 即可

**Code**

```
void solve() {
    int n, m; cin >> n >> m;

    struct edge {
        int u, v, w;
    };

    vector e(m, edge());
    for (auto& [u, v, w]: e) cin >> u >> v >> w;
    sort(all(e), [&](auto& a, auto& b){
        return a.w < b.w;
    });

    vector fa(n + 1, 0); iota(all(fa), 0);
    vector siz(n + 1, 1);

    auto find = [&](int x, auto find) -> int {
        return fa[x] == x ? x : (fa[x] = find(fa[x], find));
    };

    auto merge = [&](int u, int v) {
        auto fu = find(u, find), fv = find(v, find);
        if (fu == fv) return 0;
        if (siz[fu] < siz[fv]) swap(fu, fv);
        fa[fv] = fu; siz[fu] += siz[fv];
        return 1;
    };


    int ans = inf * 2;

    for (int i = 0; i < m; i++) {
        int t = n - 1;
        iota(all(fa), 0); fill(all(siz), 1);
        for (int j = i; j < m; j++) {
            auto& [u, v, w] = e[j];
            t -= merge(u, v);
            if (!t) cmin(ans, w - e[i].w);
        }
    }
    if (ans == inf * 2) cout << "NO\n";
    else cout << "YES\n" << ans << endl;
}
```

# # 49815 「CF EDU 并查集 Step2 G」No refuel

**题目描述**

城市$1,2,\cdots,n$之间有几条道路。每条路的长度都是已知的。道路网络是连通的，即任何一对城市之间都能够相互到达。加油站只设在城市里（且每座城市都有加油站）。你需要计算能够让汽车能够从任意城市出发，到达任意城市的最小油箱容量（即在不加油的情况下移动距离的最小值）

**输入格式**

第一行包含两个整数$n,k(1\le n\le 1500,1\le k\le 4\times 10^5)$，分别代表顶点数和边数

此后$k$行每行三个整数$u_i,v_i,w_i$，代表一条端点为$u_i,v_i(1\le u_i\neq v_i\le n)$，权值为$w_i(0\le w_i\le 10^4)$的路径

**输出格式**

输出一行一个整数，汽车在不加油的情况下应该能够通过的最远距离

**样例输入**

```
3 2
1 2 5
1 3 10
```

**样例输出**

```
10
```

**Solution**

显然，这道题是让我们找**最小生成树**（这样能够确保每个城市都联通）上的最长的边的长度

在进行 **Kruskal** 算法时，记录边的最大值即可

**Code**

```
void solve() {
    int n, m; cin >> n >> m;

    struct edge {
        int u, v, w;
    };

    vector e(m, edge());
    for (auto& [u, v, w]: e) cin >> u >> v >> w;
    sort(all(e), [&](auto& a, auto& b){
        return a.w < b.w;
    });

    vector fa(n + 1, 0); iota(all(fa), 0);
    vector siz(n + 1, 1);

    auto find = [&](int x, auto find) -> int {
        return fa[x] == x ? x : (fa[x] = find(fa[x], find));
```

```
    };

    auto merge = [&](int u, int v) {
        auto fu = find(u, find), fv = find(v, find);
        if (fu == fv) return;
        if (siz[fu] < siz[fv]) swap(fu, fv);
        fa[fv] = fu; siz[fu] += siz[fv];
    };


    int ans = 0;

    int t = n - 1;
    for (auto& [u, v, w]: e) {
        auto fu = find(u, find), fv = find(v, find);
        if (fu == fv) continue;
        merge(fu, fv);

        t--;
        if (!t) ans = w;
    }

    cout << ans << endl;
}
```

## # 49816 「CF EDU 并查集 Step2 H」Oil business

**题目描述**

给定一个包含个 $n$ 顶点和 $m$ 条边的无向连通图。对于每条边，你都知道删除的成本。你的目标是删除尽可能多的边，同时保持图的连通性，并且删除边的总成本不超过 $s$

**输入格式**

第一行包含三个整数 $n, m, s(2 \leq n \leq 5 \times 10^4, 1 \leq m \leq 10^5, 0 \leq s \leq 10^{18})$，分别代表顶点数，边数和删除的总成本

此后 $m$ 行每行三个整数 $u_i, v_i, w_i$，代表一条端点为 $u_i, v_i (1 \leq u_i \neq v_i \leq n)$，权值为 $w_i (0 \leq w_i \leq 10^9)$的路径

**输出格式**

第一行输出一个整数，删除边的最大数量 $N$

第二行从小到大输出 $N$ 条边的编号，以空格分开

**样例输入**

```
6 7 10
1 2 3
1 3 3
2 3 3
3 4 1
4 5 5
5 6 4
4 6 5
```

**样例输出**

```
2
1 6
```

### Solution

对于本题而言，我们需要尽可能多地删边，因此，我们可以先找到整张图的**最大生成树**，用这些边保证图的连通性，随后将剩下的边按权值**从小到大**依次删除即可

### Code

```cpp
void solve() {
    ll n, m, s, id = 1;
    cin >> n >> m >> s;

    struct edge {
        int u, v, w, idx;
    };

    vector e(m, edge());
    for (auto& [u, v, w, idx]: e) {
        cin >> u >> v >> w; idx = id++;
    }
    sort(all(e), [&](auto& a, auto& b){
        return a.w > b.w;
    });

    vector fa(n + 1, 0); iota(all(fa), 0);
    vector siz(n + 1, 1), kep(m + 1, 0);

    auto find = [&](int x, auto find) -> int {
        return fa[x] == x ? x : (fa[x] = find(fa[x], find));
    };

    auto merge = [&](int u, int v) {
        auto fu = find(u, find), fv = find(v, find);
        if (fu == fv) return;
        if (siz[fu] < siz[fv]) swap(fu, fv);
        fa[fv] = fu; siz[fu] += siz[fv];
    };

    int t = n - 1;
    for (auto& [u, v, w, id]: e) {
```

```
        auto fu = find(u, find), fv = find(v, find);
        if (fu == fv) continue;
        merge(fu, fv);

        t--; kep[id] = 1;
        if (!t) break;
    }

    vector ans(0, 0);
    reverse(all(e));
    for (auto& [u, v, w, id]: e) {
        if (kep[id]) continue;
        if (s < w) continue;

        ans.push_back(id); s -= w;
    }

    sort(all(ans));
    cout << ans.size() << endl;
    for (auto& v: ans) cout << v << ' ';
}
```

## # 49817 「CF EDU 并查集 Step2 I」Bipartite Graph 二分图

**题目描述**

给你一个有 $n$ 个顶点的空无向图。每个顶点在任何时刻都有 0 或 1 两种颜色，因此每条边都连接着不同颜色的顶点。

查询有两种类型：

- 给你两个来自两个不同连通分量的顶点 $x$ 和 $y$：在图中添加一条边 $(x, y)$，并改变某些节点的颜色以满足条件。
- 给你来自一个相同连通分量的两个顶点 $x$ 和 $y$：回答它们的颜色是否相同。

最初的图形是空的。

**输入格式**

第一行包含两个整数 $n$ 和 $m$ （ $1 \le n, m \le 2 \times 10^5$ ），分别是图中的顶点数和查询次数。顶点为 $0 - \text{index}$

接下来的 $m$ 行包含查询。

- 第一类查询的形式如下："0 $a$ $b$"（ $1 \le a, b \le n$ ），意思是必须将顶点为 $x$ 和 $y$ 的两个联通分量联合起来，$x, y$ 可以由以下公式得到：$x \bmod n = (a + shift) \bmod n$ 和 $y \bmod n = (b + shift) \bmod n$ 。
- 第二种类型的查询格式如下："1 $a$ $b$"（ $1 \le a, b \le n$ ），意思是要检查顶点 $x$ 和 $y$ 的颜色是否相同，$x, y$ 可以由以下公式得到：$x \bmod n = (a + shift) \bmod n$ 和 $y \bmod n = (b + shift) \bmod n$ 。

  如果查询答案为，则必须设置 $shift = (shift + 1) \bmod n$ 。

最初， $shift$ 等于零。

**输出格式**

对于第二种类型的每个查询，输出一行一个字符串，如果颜色相同，则输出 `YES`，否则，输出 `NO`。

**样例输入**

```
3 5
0 1 2
0 2 3
1 1 2
1 1 3
1 1 3
```

**样例输出**

```
NO
YES
NO
```

**Solution**

我们用一个 $col$ 数组来保存所有的点的颜色，初始颜色可以随意分配

接下来，我们考虑两种类型的操作一：

- 操作一连接的两个顶点 $u, v$ 的颜色**不同**，那么我们直接连接即可
- 操作一连接的两个顶点 $u, v$ 的颜色**相同**，那么我们需要对 $u$ **或者** $v$ 所在的集合进行**颜色反转**操作，此时，我们需要选择大小**较小**的集合，这样能保证我们启发式合并的时间复杂度

对于操作二的查询，只需要简单判断颜色是否相等即可

**Code**

```
void solve() {
    int n, q; cin >> n >> q;
    vector siz(n + 1, 1), fa(n + 1, 0), col(n + 1, 0);
    vector son(n + 1, vector(0, 0));

    iota(all(fa), 0);
    for (int i = 1; i <= n; i++) son[i].push_back(i);

    auto find = [&](int x, auto find) -> int {
        return fa[x] == x ? x : (fa[x] = find(fa[x], find));
    };

    auto merge = [&](int uu, int vv) {
        auto fu = find(uu, find), fv = find(vv, find);
        if (fu == fv) return;

        if (siz[fu] < siz[fv]) {
            if (col[uu] == col[vv]) for (auto v: son[fu]) col[v] ^= 1;
            son[fv].insert(son[fv].end(), all(son[fu]));
            swap(son[fu], son[fv]);
        }
```

```
        else {
            if (col[uu] == col[vv]) for (auto v: son[fv]) col[v] ^= 1;
            son[fu].insert(son[fu].end(), all(son[fv]));
        }

        fa[fv] = fu; siz[fu] += siz[fv];
    };

    int shift = 0;
    while (q--) {
        int type, u, v; cin >> type >> u >> v;
        u = ((u + shift) % n) + 1;
        v = ((v + shift) % n) + 1;
        if (type == 0) merge(v, u);
        else {
            shift += col[u] == col[v];
            cout << (col[u] == col[v] ? "YES\n" : "NO\n");
        }
    }
}
```

## # 49818 「CF EDU 并查集 Step2 J」First Non-Bipartite Edge

**题目描述**

二分图是这样一种无向图，可以将其顶点分成两部分，使得边只连接不同部分的顶点。

在这个问题中，$m$ 条边逐一添加到最初边集为空的 $n$ 个顶点的图中。你的任务是确定添加了哪一条边以后，图不再是二分图。

**输入格式**

第一行包含两个整数 $n, m(1 \leq n, m \leq 3 \times 10^5)$，分别代表顶点数，边数

此后 $m$ 行每行两个整数 $u_i, v_i$，代表一条端点为 $u_i, v_i(1 \leq u_i \neq v_i \leq n)$ 的边

**输出格式**

打印答案边的序号（边按在输入中出现的顺序从 $1$ 开始编号），如果没有这样的边，则打印 `-1` 。

**样例输入**

```
4 5
1 2
2 3
3 4
1 4
2 4
```

**样例输出**

```
5
```

**Solution**

我们用一个 $col$ 数组来保存所有的点的颜色，初始颜色可以随意分配

接下来，我们考虑四种类型的操作：

- 连接的两个顶点 $u, v$ 不处于同一个连通分量且颜色**不同**，那么我们直接连接即可
- 连接的两个顶点 $u, v$ 不处于同一个连通分量且的颜色**相同**，那么我们需要对 $u$ **或者** $v$ 所在的集合进行**颜色反转**操作，此时，我们需要选择大小**较小**的集合，这样能保证我们启发式合并的时间复杂度
- 连接的两个顶点 $u, v$ 处于同一个连通分量且颜色**相同**，就不进行任何操作
- 连接的两个顶点 $u, v$ 处于同一个连通分量且颜色**不同**，那么这条边就是冲突边

**Code**

```
void solve() {
    int n, m; cin >> n >> m;
    vector siz(n + 1, 1), fa(n + 1, 0), col(n + 1, 0);
    vector son(n + 1, vector(0, 0));

    iota(all(fa), 0);
    for (int i = 1; i <= n; i++) son[i].push_back(i);

    auto find = [&](int x, auto find) -> int {
        return fa[x] == x ? x : (fa[x] = find(fa[x], find));
    };

    auto merge = [&](int uu, int vv) {
        auto fu = find(uu, find), fv = find(vv, find);
        if (fu == fv) return col[uu] != col[vv];

        if (siz[fu] < siz[fv]) {
            if (col[uu] == col[vv]) for (auto v: son[fu]) col[v] ^= 1;
            son[fv].insert(son[fv].end(), all(son[fu]));
            swap(son[fu], son[fv]);
        }
        else {
            if (col[uu] == col[vv]) for (auto v: son[fv]) col[v] ^= 1;
            son[fu].insert(son[fu].end(), all(son[fv]));
        }

        fa[fv] = fu; siz[fu] += siz[fv];
        return true;
    };

    for (int i = 1; i <= m; i++) {
        int u, v; cin >> u >> v;
        if (!merge(u, v)) {cout << i << endl; return;}
    }
    cout << -1 << endl;
}
```

# # 3321 团伙(group)

**题目描述**

现在有 $n$ 个人，他们之间有两种关系：朋友和敌人。我们知道：

- 一个人的朋友的朋友是朋友
- 一个人的敌人的敌人是朋友

现在要对这些人进行组团。两个人在一个团体内当且仅当这两个人是朋友。请求出这些人中最多可能有的团体数。

**输入格式**

第一行输入一个整数 $n(1 \leq n \leq 10^3)$ 代表人数。

第二行输入一个整数 $m(1 \leq m \leq 10^5)$ 表示接下来要列出 $m$ 个关系。

接下来 $m$ 行，每行一个数字 $opt$ 和两个整数 $p, q(1 \leq p, q \leq n)$，分别代表关系（朋友或敌人），有关系的两个人之中的第一个人和第二个人。其中 $opt$ 有两种可能：

- 如果 $opt$ 为 `0`，则表明 $p$ 和 $q$ 是朋友。
- 如果 $opt$ 为 `1`，则表明 $p$ 和 $q$ 是敌人。

**输出格式**

一行一个整数代表最多的团体数。

**样例输入**

```
6 4
1 1 4
0 3 5
0 4 6
1 1 2
```

**样例输出**

```
3
```

**Solution**

对于普通的并查集，我们能很轻易地维护**朋友的朋友是朋友**这种关系

而对于**敌人的敌人是朋友**这种关系，我们就要用到**种类并查集**

我们将并查集元素翻倍，即将原来的 $1, 2, \cdots, n$ 更新为 $1, 2, \cdots, 2n$ ，其中， $1, 2, \cdots, n$ 仍然表示我们的**朋友关系**，而 $n+1, n+2, \cdots, n+n$ 则代表我们的敌人关系

对于两个人 $u, v$，如果说两人是朋友关系的话，我们可以很简单地合并 $u, v$ 所在的集合

而若两个人是敌人关系，我们合并 $u, v+n$ 和 $v, u+n$ 所在的集合，和 $x+n$ 处于**同一集合**中的元素（从合并关系重可以看出，这些元素都 **小于等于** $n$ ），都代表是 $x$ 的**敌人**，这样，我们就把 $x$ 的敌人集合合并了，同时也维护了**敌人的敌人的朋友（也是朋友的朋友）**这个集合

**Code**

```
namespace DSU {

vector fa(0, 0), siz(0, 0);

void init(int n) {
    fa.resize(n + 1), iota(all(fa), 0);
    siz.resize(n + 1); fill(all(siz), 1);
}

int find(int x) {
    return (fa[x] == x) ? x : (fa[x] = find(fa[x]));
}

void merge(int u, int v) {
    auto fu = find(u), fv = find(v);
    fa[fv] = fu;
}

}

void solve() {
    int n, q; cin >> n >> q;
    DSU::init(2 * n + 1);
    while (q--) {
        int ch; int u, v;
        cin >> ch >> u >> v;
        if (ch == 1) {
            DSU::merge(u, v + n);
            DSU::merge(v, u + n);
        }
        else DSU::merge(u, v);
    }

    int ans = 0;
    for (int i = 1; i <= n; i++) ans += DSU::fa[i] == i;

    cout << ans << endl;
}
```

## # 2860 [「NOIP2010」关押罪犯]

**题目描述**

S 城现有两座监狱，一共关押着 $N$ 名罪犯，编号分别为 $1 \sim N$。他们之间的关系自然也极不和谐。很多罪犯之间甚至积怨已久，如果客观条件具备则随时可能爆发冲突。我们用“怨气值”（一个正整数值）来表示某两名罪犯之间的仇恨程度，怨气值越大，则这两名罪犯之间的积怨越多。如果两名怨气值为 $c$ 的罪犯被关押在同一监狱，他们俩之间会发生摩擦，并造成影响力为 $c$ 的冲突事件。

每年年末，警察局会将本年内监狱中的所有冲突事件按影响力从大到小排成一个列表，然后上报到 S 城 Z 市长那里。公务繁忙的 Z 市长只会去看列表中的第一个事件的影响力，如果影响很坏，他就会考虑撤换警察局长。

在详细考察了 $N$ 名罪犯间的矛盾关系后，警察局长觉得压力巨大。他准备将罪犯们在两座监狱内重新分配，以求产生的冲突事件影响力都较小，从而保住自己的乌纱帽。假设只要处于同一监狱内的某两个罪犯间有仇恨，那么他们一定会在每年的某个时候发生摩擦。

那么，应如何分配罪犯，才能使 Z 市长看到的那个冲突事件的影响力最小？这个最小值是多少？

**输入格式**

每行中两个数之间用一个空格隔开。第一行为两个正整数 $N, M(1 \leq N \leq 2 \times 10^4, 1 \leq M \leq 10^5)$，分别表示罪犯的数目以及存在仇恨的罪犯对数。接下来的 $M$ 行每行为三个正整数 $a_j, b_j, c_j$，表示 $a_j$ 号和 $b_j$ 号罪犯之间存在仇恨，其怒气值为 $c_j$。数据保证 $1 < a_j \leq b_j \leq N, 0 < c_j \leq 10^9$，且每对罪犯组合只出现一次。

**输出格式**

共一行，为 Z 市长看到的那个冲突事件的影响力。如果本年内监狱中未发生任何冲突事件，请输出 `0`。

**样例输入**

```
4 6
1 4 2534
2 3 3512
1 2 28351
1 3 6618
2 4 1805
3 4 12884
```

**样例输出**

```
3512
```

### Solution

首先，有个很显而易见的**贪心**思路，就是我们尽量将怒气值**大**的罪犯分开关押，也就是让这两个罪犯**不在同一个监狱**

我们考虑**不在同一个监狱**这个事情的传递性，很显然，如果 $a, b$ 不在一个监狱，$b, c$ 不在一个监狱，说明 $a, c$ **在同一个监狱**，这代表我们可以利用种类并查集，维护**两个人同时和另一个人不在一个监狱，那这两个人就在同一个监狱**

那我们就能使用**种类并查集**来维护我们的**不在一个监狱**的关系，首先对所有的仇恨 $u_i, v_i, w_i$ 以 $w_i$ 为关键字进行排序，若 $u_i, v_i$ **不处于**同一并查集中，说明他们**暂时没有被关在一个监狱**，因此可以将他们分开关押，此时我们合并 $u_i, v_i + n$ ， $u_i + n, v_i$，确保他们的**不在同一监狱**的关系得到维护。若 $u_i, v_i$ **处于**同一并查集中，说明他们已经被关押在同一监狱中了，这样说明他们的冲突已经无法避免，由于我们已经按照 $w_i$ 排序了，那说明答案就是 $w_i$

### Code

```
namespace DSU {

vector fa(0, 0), siz(0, 0);

void init(int n) {
    fa.resize(n + 1), iota(all(fa), 0);
```

```
    siz.resize(n + 1); fill(all(siz), 1);
}

int find(int x) {
    return (fa[x] == x) ? x : (fa[x] = find(fa[x]));
}

void merge(int u, int v) {
    auto fu = find(u), fv = find(v);
    if (siz[fu] < siz[fv]) swap(fu, fv);
    fa[fv] = fu;
}

}

void solve() {
    int n, m; cin >> n >> m;
    DSU::init(2 * n + 1);
    vector<array<int, 3>> a(m);
    for (auto& [u, v, w]: a) cin >> u >> v >> w;

    sort(all(a), [](auto& x, auto& y){
        return x[2] > y[2];
    });

    for (auto& [u, v, w]: a) {
        auto fu = DSU::find(u), fv = DSU::find(v);
        if (fu == fv) {cout << w << endl; return;}
        DSU::merge(fu, fv + n);
        DSU::merge(fv, fu + n);
    }

    cout << 0 << endl;
}
```

## # 3326 「NOI2001」食物链

**题目描述**

动物王国中有三类动物 $A, B, C$，这三类动物的食物链构成了有趣的环形。$A$ 吃 $B$，$B$ 吃 $C$，$C$ 吃 $A$。

现有 $N$ 个动物，以 $1 \sim N$ 编号。每个动物都是 $A, B, C$ 中的一种，但是我们并不知道它到底是哪一种。

有人用两种说法对这 $N$ 个动物所构成的食物链关系进行描述：

- 第一种说法是 `1 X Y`，表示 $X$ 和 $Y$ 是同类。
- 第二种说法是 `2 X Y`，表示 $X$ 吃 $Y$。

此人对 $N$ 个动物，用上述两种说法，一句接一句地说出 $K$ 句话，这 $K$ 句话有的是真的，有的是假的。当一句话满足下列三条之一时，这句话就是假话，否则就是真话。

- 当前的话与前面的某些真的话冲突，就是假话；
- 当前的话中 $X$ 或 $Y$ 比 $N$ 大，就是假话；
- 当前的话表示 $X$ 吃 $X$，就是假话。

你的任务是根据给定的 $N$ 和 $K$ 句话，输出假话的总数。

**输入格式**

第一行两个整数，$N(1 \le N \le 5 \times 10^4), K(1 \le K \le 10^5)$，表示有 $N$ 个动物，$K$ 句话。

第二行开始每行一句话，代表一种食物链关系的描述

**输出格式**

一行，一个整数，表示假话的总数。

**样例输入**

```
100 7
1 101 1
2 1 2
2 2 3
2 3 3
1 1 3
2 3 1
1 5 5
```

**样例输出**

```
3
```

**Solution**

假话判定的三种形式中，$2, 3$ 条实际上只是对输入进行一个简单的合法性校验就可以得到，因此，我们的重点是第一条

考虑两种描述，第一种描述是一个简单的并查集合并关系，具有非常简单的传递性

第二种描述，我们去查看它的**传递关系**，可以发现，假设 $x$ 吃 $y$，$y$ 吃 $z$，我们就可以很轻易地推导出 $z$ 吃 $x$ 这种关系

那么，我们就需要开一个**三倍大小的并查集**，$i$ 代表 $i$ 是物种 $A$，$i+n$ 代表 $i$ 是物种 $B$，$i+2n$ 代表 $i$ 是物种 $C$

我们先考虑怎么维护关系，再考虑怎么处理矛盾

那么我们在维护 $x$ 吃 $y$ 这种关系时，我们可以分别假设 $x$ 的物种为 $A, B, C$，那么就需要连接 $x, y+n$ ， $x+n, y+2n$， $x+2n, y$

那维护 $x$ 和 $y$ 是同一类物种，同样的，我们可以分别假设 $x$ 的物种为 $A, B, C$，那么就需要连接 $x, y$ ， $x+n, y+n$， $x+2n, y+2n$

那如何处理冲突呢?

对于 $x$ 吃 $y$ 这种关系，我们只需要检查 $x$ 和 $y$ 是否是同一物种，或者 $y$ 是否吃 $x$ 即可，此时我们只需要简单地假定 $y$ 的物种为 $A$ 进行检验即可（因为我们在维护关系时同时维护了所有物种）

而对于 $x, y$ 是同一物种的关系，我们只需要简单地检查 $x$ 是否吃 $y$ 或者 $y$ 是否吃 $x$ 即可

**Code**

```
namespace DSU {

vector fa(0, 0), siz(0, 0);

void init(int n) {
    fa.resize(n + 1), iota(all(fa), 0);
    siz.resize(n + 1); fill(all(siz), 1);
}

int find(int x) {
    return (fa[x] == x) ? x : (fa[x] = find(fa[x]));
}

void merge(int u, int v) {
    auto fu = find(u), fv = find(v);
    if (siz[fu] < siz[fv]) swap(fu, fv);
    fa[fv] = fu;
}

}

void solve() {
    int n, m; cin >> n >> m;
    DSU::init(3 * n + 1);

    int ans = 0;
    while (m--) {
        int type, u, v;
        cin >> type >> u >> v;
        if (u > n || v > n) {ans++; continue;}
        if (type == 1) {
            if (DSU::find(u) == DSU::find(v + n)) ans++;
            else if (DSU::find(v) == DSU::find(u + n)) ans++;
            else {
                DSU::merge(u, v);
                DSU::merge(u + n, v + n);
                DSU::merge(u + 2 * n, v + 2 * n);
            }
        }
        else {
            if (DSU::find(u) == DSU::find(v)) ans++;
            else if (DSU::find(v) == DSU::find(u + n)) ans++;
            else {
                DSU::merge(u, v + n);
                DSU::merge(u + n, v + 2 * n);
                DSU::merge(u + 2 * n, v);
            }
        }
    }

    cout << ans << endl;
```

```
}
```