反集：

我们在做并查集的问题时，一般都是分三步去解决问题的：

- 给定n个点，初始化并查集。
- 给你m个关系，每个关系对应地连接两个x,y对应的集合。
- 求一些问题：例如是否联通，连通块个数。

这些问题都可以在$O(1)$的时间完成。

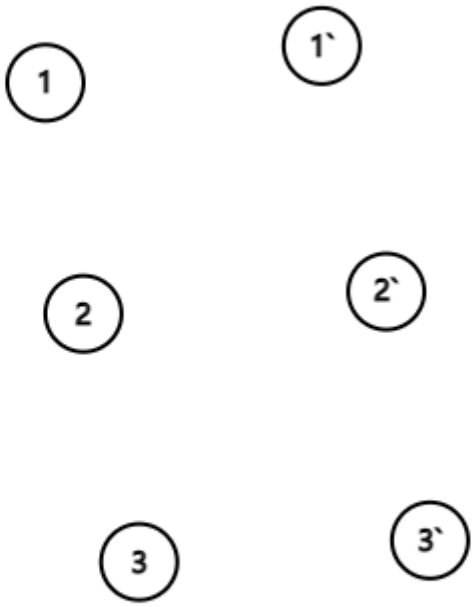
但是，如果在第二步上做一些变更呢？比如给定两种关系。第一种：给定一对朋友关系；第二种，**给定一对敌人关系**，满足**敌人的敌人就是朋友**，。那么传统并查集就捉襟见肘了。

此时，反集就派上用场了。

思路：

反集的思路是再构造一个集合（称之为反集），然后将"敌人"关系通过原集和反集表示出来。

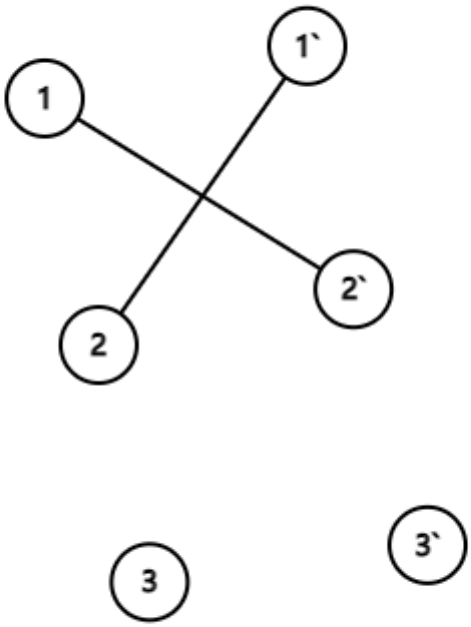
比如假设有 3个元素，1，2，3。我们称他们的反集元素分别为 $1'$，$2'$，$3'$



那么，如何体现 **敌人关系**呢？

假如有一对敌人关系 $(1,2)$，我们只需要连接$(1,2')$和$(1',2)$就行了。

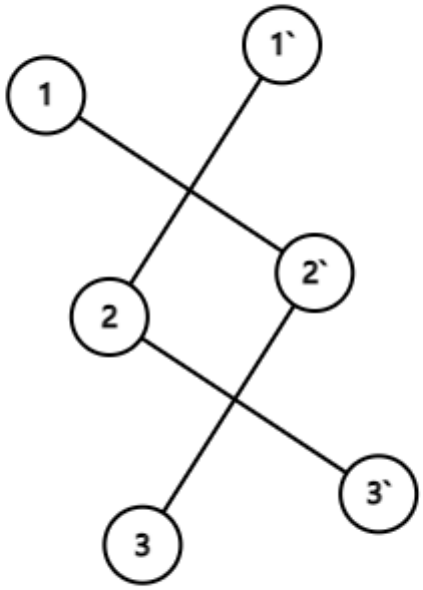
由于$1'$和$2'$不会联通， 不会相连，所以这在并查集中也就意味着， **这两个结点不会连到一起了**，这样，在1，2之间构成了一个敌对关系，如图。



等等！还有一个 **敌人的敌人就是朋友**的关系，在这里是正确的吗？

假如还有一对敌人关系 $(2,3)$，那么，根据规则，1与3应该是朋友关系，即在一个集合中。

我们按照之前的规则，连接$(3,2')$，$(3'，2)$，则图变成了下图这样。图中有两个连通块：$1,2',3$和$1',2,3'$。自然的$1,3$就在一个连通块里了。



因此我们就实现了**敌人的敌人是朋友。**

实现：

我们可以开$2 \times n$的数组，利用$i + n$表示i的反集。不要忘记初始化这些节点。

初始化：

```
for(int i = 1; i <= 2 * n; i++)
{
    fa[i]=i;
}
```

反集操作：

```
cin >> isfriend >> u >> v;
if(isfriend)
{
    union(u, v);
}
else
{
    union(u, v + n);
    union(v, u + n);
}
```

最后，反集虽然不是一个特别常见的数据结构，但是在并查集的优化中很有用。

注意：反集的合并只能让`v + n`合并到`u`去，即`fa[fy] = fx`。