并查集用来解决元素分组的问题，主要是对一系列不相交集合的管理。支持如下两种操作：1.查找find：查询两个元素是否在同一个集合中；2.合并union：把两个不相交的集合合并为一个集合。

举个容易理解的例子：亲戚是一个神奇的种群，亲戚的亲戚就是我的亲戚。在找亲戚的过程中，如果发现一个人，怎么判断是不是亲戚呢？需要有一个代表，这个代表作为整个亲戚的领头人，在判断是否和人属于亲戚关系时，向上查找代表亲戚是谁，如果和自己的代表是同一个人，那么就是亲戚关系。并且，只要是有亲戚关系的（直接或间接，哪怕间接了几千个人），就一定是亲戚关系。

这样在建模方面就是如下的结构：将所有人划分到若干个不相交的集合中，每个集合中的人彼此之间有亲戚关系，而不在一个集合的则不是。在判断两个人是否是亲戚时，只要判断是否属于同一个集合即可。而集合查找最关键的地方则时需要一个领头人，这样在判断是否属于一个集合时不需要遍历所有成员，只需要向上查找比较领头人是否一样即可。可以想象成一个集合对应一个帮派，这个领头人就是掌门，只要掌门是一样的，自然属于同一个帮派。

在查找阶段，如果自己不是掌门，说明自己上级还有人，继续向上查找，如果自己是掌门则直接返回即可。初始化时每个人都是掌门。

int fa[MAXN];

void init(n): {

for (int i = 1; i <= n; i++)

fa[i] = i;

}

查询操作:

int find(int x) {

if (fa[x] == x) return x;

else return find(fa[x]);

}

递归写法查询元素，逐层访问直到自己做自己的掌门。

合并操作：

void merge(int i, int j) {

fa[find(i)] = find(j);

}

首先找到两个集合的掌门，然后将前者的掌门设为后者，这样就表示这两者及其附属手下均属于后者的门派。

路径压缩

上述并查集效率较低，在合并操作过程可能会造成原本的树结构退化为单链表结构，导致从叶子节点或非根节点查询到根节点的路径过长。既然要查询到根节点，直接减少树的高度就是最初始的想法。可以将整棵树构造成所有非根节点的父节点均为根节点，这样在find时可以最多两步找到掌门。在实现方面也很简单，在查询过程中，把沿途的每个节点的父节点都设为根节点即可。

合并（路径压缩）

int find(int x) {

if (fa[x] == x)

return x

else {

fa[x] = find(fa[x]);

return fa[x];

}

}

通常可以简写为一行

int find(int x) {

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

}

路径压缩后时间复杂度已经较低，可以通过大部分并查集问题，但对于某些卡时间但对空间要求不高的题目可以使用空间换时间的方式进一步优化。

按秩合并

路径压缩优化后，并查集可能并不是一个菊花图（只有两层树）。例如fa[1]=2之后，对1再没有访问（包括find和unoin），那么在后续对2及2的父节点访问时将2合并到根节点，但1到根节点的深度或距离为2。当这种情况出现较多时会有更复杂的树结构出现。而一棵复杂树和一棵简单树的合并，可想而知将深度小的树合并到更大的树上对深度无影响，总体的查询效率趋于优化。

这就需要维护一个深度数组rank来记录每个节点的深度，在合并操作时，将深度小的树合并到深度大的树上，而深度相同时合并关系随意，但合并到的节点深度需要+1，因为有一棵同样深度的树作为子数合并到根节点。

合并（按秩合并）

void merge(int i, int j) {

int x = find(i), y = find(j);

if (rank[x] <= rank[y]) fa[x] = y;

else rank[y] = x;

if (rank[x] == rank[y] && x != y) rank[y]++;//上述if在相等的情况下将x合并到y上

}