并查集

概念

并查集（Union-Find Set），也称为不相交集数据结构（Disjointed Set Data Structure）。是指一系不相交的集合（Sets），提供合并（Union）和查找(Find)两种操作。

作用

用来解决连通性问题

重点

1.判集合find：确定元素属于哪一个子集。这个确定方法是不断向上查找找到它的根节点

2.加入union

实现

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct DisjointSet{
    //声明一个数组    存储他的父亲节点
    int *father;
} DisjointSet;

void init(DisjointSet *s, int size) {
    //申请空间
    s->father = (int *)malloc(sizeof(int) * size);
    for (int i = 0; i < size; ++i) {
        s->father[i] = i;//将其全部初始化为自己
    }
}

void swap(int *a, int *b) {  //交换
    int temp = *a;
    *a = *b;
    *b = temp;
}

int max(int a, int b) { //找最大值
    return a > b ? a : b;
}

int find_set(DisjointSet *s, int node) {//返回他的根节点
    if (s->father[node] != node) {
        return find_set(s, s->father[node]);
    }
    return node;
}

int merge(DisjointSet *s, int node1, int node2) {//合并操作
    int ancestor1 = find_set(s, node1);  //node1的根节点
    int ancestor2 = find_set(s, node2); //node2的根节点
    if (ancestor1 != ancestor2) {  //如果两个人老大不同就合并操作
        s->father[ancestor1] = ancestor2;
        return 1;//成功返回 1
    }
    return 0;//两个人本身就在同一个集合
}

void clear(DisjointSet *s) {//清除操作
    free(s->father);
    free(s);
}

int main() {
    DisjointSet *dsu = (DisjointSet *)malloc(sizeof(DisjointSet));
    //申请空间
    init(dsu, 100);

    int m, x, y;
```

```
    scanf("%d", &m);
    for (int i = 0; i < m; i++) {
        scanf("%d%d", &x, &y);
        int ans = merge(dsu, x, y);
        if (ans) {
            printf("success\n");
        } else {
            printf("failed\n");
        }
    }

    clear(dsu);
    return 0;
}
```

比赛用find

```
int find(int x){
   if(x != f[x]) {
       f[x]=find(f[x]);
   }
   return f[x];
}

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

比赛用union

Union操作就是将两个不相交的子集合合并成一个大集合。简单的Union操作是非常容易实现的，因为只需要把一棵子树的根结点指向另一棵子树即可完成合并。

```
void union(int x, int y){
    int fx = find(x);
    int fy = find(y);
    if(fx != fy){
        f[fx] = fy;
    }
    return ;
}
```

优化

1.按秩合并

术语“秩”替代了“深度”

在一种极端情况下如N个元素退化为一条链，而查找时就会遍历整条连时间复杂度为O(n)

总是将更小的树连接至更大的树上。因为影响运行时间的是树的深度，更小的树添加到更深的树的根上将不会增加秩除非它们的秩相同。

实现

为了避免这种情况，我们可以再合并的时候尽可能的让树的深度不要过深

我们就需要申请一个新的数组 rank 存储深度

将 rank 数组全部初始化为 1

```
int merge(DisjointSet *s, int node1, int node2) {
    int ancestor1 = find_set(s, node1);//查找node1的根节点
    int ancestor2 = find_set(s, node2);//查找node2的根节点
    if (ancestor1 != ancestor2) { //两个不属于一个集合合并
        if (s->rank[ancestor1] > s->rank[ancestor2]) {
            swap(&ancestor1, &ancestor2);
        }
        //将深度大的定义为ａｎｃｅｓｔｏｒ２
        s->father[ancestor1] = ancestor2;//将ａｎｃｅｓｔｏｒ２设置为ａｎｃｅｓｔｏｒ１的父亲节点
        s->rank[ancestor2] = max(s->rank[ancestor2], s->rank[ancestor1] + 1);
        //合并后深度变为原来的深度加上一个根和原来ｒａｎｋ２深度教大的
        return 1;
    }
    return 0;
}
```

比赛用

单元素的树的秩定义为0，当两棵秩同为r的树联合时，它们的秩r+1。

给每个点一个秩，其实就是树高 每次合并的时候都用秩小的指向秩大的，可以保证树高最高为log2(n) 操作的时候，一开始所有点的秩都为1

fa[x]为x的父亲，就是x指向的点，rank[x]为点x的秩

在一次合并后，假设是点x和点y，x的秩小 当然x和y都是原来x和y所在区间的顶点 设点x秩为rank[x] 将fa[x]指向y，然后将rank[y]的与rank[x+1]取max

因为rank[x]为区间x的高，将它连向y之后，y的树高就会是x的树高+1，当然也可能y在另一边树高更高，所以取max

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

void unite(int x,int y) {
    x = find(x);
    y = find(y);
    if(x != y) {
        if(rank[x] <= rank[y]) {
            f[x] = y;
            rank[y] = max(rank[y],rank[x]+1);
        } else {
            f[y] = x;
            rank[x] = max(rank[x],rank[y]+1);
        }
    }
    return;
}
```

2.路径压缩

是一种在执行“查找”时扁平化树结构的方法。关键在于在路径上的每个节点都可以直接连接到根上；他们都有同样的表示方法。为了达到这样的效果，Find递归地经过树，改变每一个节点的引用到根节点。得到的树将更加扁平，为以后直接或者间接引用节点的操作加速。

原来的只是将根节点返回，我们可以将自身节点直接连接到其根节点当中

实现

```
int find_set(DisjointSet *s, int node) {
    if (s->father[node] != node) {
        s->father[node] = find_set(s, s->father[node]);//将找到的根节点返回
    }
    return s->father[node];
}
```

比赛用

```
int find_set(int x) {//递归写法
    if(f[x] == x) return x;
    f[x] = find_set(f[x]);
    return find_set(f[x]);
}

int find_set(int x) {//非递归写法 更好，因为不会RE
    int root = x;
    while(root != f[root]) {//先要找到根节点root
        root = f[root];
    }
    int y = x;
    while(y != root) {
        int father = f[y];
        f[y] = root;
        y = father;
    }
    return root;
}
```

练习题

P3367【模板】并查集

代码

```
#include<iostream>
#include<stdio.h>

using namespace std;

int i, j, k, n, m, s, ans, f[10010], p1, p2, p3;
//f[i]表示i的集合名

int find(int k) {
    //路径压缩
    if(f[k] == k) return k;
    return f[k] = find(f[k]);
}

int main() {
    cin >> n >> m;
    for(i = 1; i <= n; i++) {
        f[i]=i;
    }
    for(i = 1; i <= m; i++) {
        cin >> p1 >> p2 >> p3;
        if(p1 == 1) {
            f[find(p2)] = find(p3);
        } else {
            if(find(p2) == find(p3)) {
                printf("Y\n");
            } else {
                printf("N\n");
            }
        }
    }
    return 0;
}
```

P1551 亲戚

代码

```
#include<cstdio>

using namespace std;

int fa[500010];

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

int main() {
    int n,m,p;
    scanf("%d %d %d", &n, &m, &p);
    for(int i = 1; i <= n; i++) {
        fa[i] = i;
    }
    int x, y;
    for(int i = 1; i <= m; i++) {
        scanf("%d %d", &x, &y);
        fa[find(x)]=find(y);//X祖先为y祖先
    }
    for(int i = 1; i <= p; i++) {
        scanf("%d %d", &x, &y);
        if(find(x) == find(y)) printf("Yes\n");
        else printf("No\n");
    }
    return 0;
}
```

P1111 修复公路

解题思路

修公路，给出了每个公路修的时间，此题求最少需要的时间，我们知道他要求任意的两个村庄都能够通车,即所有的村庄都在一个集合中 题目还要求我们时间最短我们直接按时间来排序一下即可 所以我们直接用并查集按照我们排好的序列来进行合并并计算时间 按照返回值正确的个数和村庄的个数－ 1 相同时就是最小的时间 如果结束后所得值小于村庄个数－ 1 返回－ 1

代码

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int x, y, z;
} Node;
//模版
int find(int *father, int node) {
    if (father[node] != node) {
        return find(father, father[node]);
    }
    return father[node];

}
//快排
void sort(Node *num, int l, int r) {
    if (r <= l) return ;
    int x = l, y = r;
    Node mi = num[l];
    while (x < y) {
        while (x < y && num[y].z >= mi.z) --y;
        if (x < y) num[x++] = num[y];
        while (x < y && num[x].z <= mi.z) ++x;
        if (x < y) num[y--] = num[x];
    }
    num[x] = mi;
    sort(num, l, x - 1);
    sort(num, x + 1, r);
    return ;
}

int main() {
    int m, n;
    scanf("%d %d", &m, &n);
    //为申请node数据类型空间
    Node *p = (Node *) malloc (sizeof(Node) * 100001);
    for (int i = 0; i < n; i++) {
        scanf("%d %d %d", &p[i].x, &p[i].y, &p[i].z);
    }
    int num = 0, hh = 0;
    //按照时间来排序
    sort(p, 0, n);
    int father[100001];
    for (int i = 0; i <= m+5; i++) {
        //将数组初始化
        father[i] = i;
    }
    for (int i = 0; i < n; i++) {
        int a1 = find(father, p[i].x);
        int a2 = find(father, p[i].y);
        if (a1 != a2) {
            father[a1] = a2;
```

```
            num++;
            hh = p[i].z;
        }
    }
    //最后求得总和小于城镇的数量减一个则返回－1
    if (num < m - 1) {
        printf("-1\n");
    } else {
        printf("%d\n", hh);
    }
    free(p);
    return 0;
}
```