【第八课】并查集

注意：五一假后讲课安排！！！

今晚4月30号并查集讲一次，下周一还在放假，就不讲课了；放假来了之后是周四，**5月6日周四晚上看船长讲快排**；**5月7日周五晚上**我们把剩下的并查集的课讲完了；下周一也就是**5月10号**，我们把快排一次性讲完

547. 省份数量

1、初始时，每个城市都属于不同的连通分量，遍历矩阵isConnected,如果两个城市之间有相连关系，那么他们属于同一个连通分量，对他们进行合并。
2、把全部的元素都遍历完成之后，计算连通分量的总数就是省份的总数。

```
/**
 * @param {number[][]} isConnected
 * @return {number}
 */

var findCircleNum = function (isConnected) {
    let circleNum = isConnected.length;
    let uf = new UnionFind(circleNum);
    for (let i = 0; i < circleNum; i++) {
        for (let j = i + 1; j < circleNum; j++) {
            if (isConnected[i][j] == 1) {
                uf.unite(i, j);
            }
        }
    }
    return uf.getCount();
};

class UnionFind {
    constructor(n) {
        this.parent = new Array(n).fill(0).map((value, index) => index);
        this.rank = new Array(n).fill(1);
        this.setCount = n;
    }

    findSet(index) {
        if (this.parent[index] != index) {
            this.parent[index] = this.findSet(this.parent[index]);
        }
        return this.parent[index];
    }

    unite(index1, index2) {
        let root1 = this.findSet(index1), root2 = this.findSet(index2);
        if (root1 != root2) {
            if (this.rank[root1] < this.rank[root2]) {
                [root1, root2] = [root2, root1];
            }
            this.parent[root2] = root1;
```

```
            this.rank[root1] += this.rank[root2];
            this.setCount--;
        }
    }

    getCount() {
        return this.setCount;
    }

    connected(index1, index2) {
        let root1 = this.findSet(index1), root2 = this.findSet(index2);
        return root1 == root2;
    }
}
```

200. 岛屿数量

1、使用并查集代替搜索，扫描整个二维数组。如果一个位置为1，则将与其相邻的四个方向上的1在并查集中进行合并。
2、接着不断重复这个操作，连通分量的数目就是岛屿的数量。主要考虑当前节点的上下左右四种情况

```
/**
 * @param {character[][]} grid
 * @return {number}
 */
let count;
var numIslands = function (grid) {
    let m = grid.length;
    let n = grid[0].length;
    let uf = new UnionFind(m * n);
    count = 0;
    for (let i = 0; i < m; i++) {
        for (let j = 0; j < n; j++) {
            if (grid[i][j] == '1') {
                count++;
            }
        }
    }
// 找当前节点的上下左右是否为1
    for (let i = 0; i < m; i++) {
        for (let j = 0; j < n; j++) {
            if (grid[i][j] == '1') {//当前岛屿数量加1
                if (i - 1 >= 0 && grid[(i - 1)][j] == '1') {// 上
                    uf.unite(i * n + j, (i - 1) * n + j);
                }
                if (i + 1 < m && grid[i + 1] [j] == '1') {// 下
                    uf.unite(i * n + j, (i + 1) * n + j);
                }
                if (j - 1 >= 0 && grid[i][j - 1] == '1') {// 左
                    uf.unite(i * n + j, i * n + j - 1);
                }//
                if (j + 1 < n && grid[i] [j + 1] == '1') {// 右
                    uf.unite(i * n + j, i * n + j + 1);
                }
            }
        }
```

```
    }
    return count;
};

class UnionFind {
    constructor(n) {
        this.parent = new Array(n).fill(0).map((value, index) => index);
        this.size = new Array(n).fill(1);
        this.setCount = n;
    }

    findSet(n) {
        if (this.parent[n] != n)
        this.parent[n] = this.findSet(this.parent[n]);
        return this.parent[n];
    }

    unite(index1, index2) {
        let root1 = this.findSet(index1), root2 = this.findSet(index2);
        if (root1 != root2) {
            if (this.size[root1] < this.size[root2]) {
                [root1, root2] = [root2, root1];
            }
            this.parent[root2] = root1;
            count--;
            this.size[root1] += this.size[root2];
            this.setCount--;
        }
    }

        getCount() {
        return this.setCount;
    }

    connected(index1, index2) {
        return this.findSet(index1) == this.findSet(index2);
    }
}
```

990. 等式方程的可满足性

1、第一次循环扫描，遇到 “=” 将两边的字符合并。

2、第二次扫描，遇到 “!”,如果两边的字符拥有共同的祖先，返回false。

```
/**
* @param {string[]} equations
* @return {boolean}
*/
var equationsPossible = function (equations) {
    let size = equations.length;
    let uf = new UnionFind(26);
    for (str of equations) {//第一次循环扫描，遇到 “=” 将两边的字符合并。
        if (str.charAt(1) == '=') {//前端字符不能相减，转下码；97是小a的ASCII码
```

```
            uf.unite(str.charCodeAt(0) - 97, str.charCodeAt(3) -
97);//charCodeAt() 方法可返回指定位置的字符的 Unicode 编码。用于考虑26个小写字母；
        }
    }
    for (str of equations) {
        if (str.charAt(1) == '!') {
            if (uf.findSet(str.charCodeAt(0) - 97) ==
uf.findSet(str.charCodeAt(3) - 97)) {//第二次扫描，遇到 “!”,如果两边的字符拥有共同的祖
先，返回false。
                return false;
            }
        }
    }
    return true;
};
class UnionFind {
    constructor(n) {
        this.parent = new Array(n).fill(0).map((value, index) => index);
        this.size = new Array(n).fill(1);
        this.setCount = n;
    }

    findSet(n) {
        if (this.parent[n] != n) this.parent[n] = this.findSet(this.parent[n]);
        return this.parent[n];
    }

    unite(index1, index2) {
        let root1 = this.findSet(index1), root2 = this.findSet(index2);
        if (root1 != root2) {
            if (this.size[root1] < this.size[root2]) {
                [root1, root2] = [root2, root1];
            }
            this.parent[root2] = root1;
            this.size[root1] += this.size[root2];
            this.setCount--;
        }
    }

        getCount() {
        return this.setCount;
    }

    connected(index1, index2) {
        return this.findSet(index1) == this.findSet(index2);
    }
}
```

684. 冗余连接

1、如果两个顶点属于不同的连通分量，则说明在遍历到当前的边之前，这两个顶点之间不连通，因此当前的边不会导致环出现，合并这两个顶点的连通分量。

2、如果两个顶点属于相同的连通分量，则说明在遍历到当前的边之前，这两个顶点之间已经连通，因此当前的边导致环出现，为附加的边，将当前的边作为答案返回。

```
 /**
 * @param {number[][]} edges
 * @return {number[]}
 */
var findRedundantConnection = function (edges) {
    let nodesCount = edges.length;
    let uf = new UnionFind(nodesCount);
    for (let i = 0; i < nodesCount; i++) {
        let edge = edges[i];
        let node1 = edge[0], node2 = edge[1];
        if (uf.findSet(node1) != uf.findSet(node2)) {
            uf.unite(node1, node2);
        } else {
            return edge;
        }
    }
    return [0];
};
class UnionFind {
    constructor(n) {
        this.parent = new Array(n).fill(0).map((value, index) => index);
        this.size = new Array(n).fill(1);
        this.setCount = n;
    }

    findSet(n) {
        if (this.parent[n] != n) this.parent[n] = this.findSet(this.parent[n]);
        return this.parent[n];
    }

    unite(index1, index2) {
        let root1 = this.findSet(index1), root2 = this.findSet(index2);
        if (root1 != root2) {
            if (this.size[root1] < this.size[root2]) {
                [root1, root2] = [root2, root1];
            }
            this.parent[root2] = root1;
            this.size[root1] += this.size[root2];
            this.setCount--;
        }
    }

        getCount() {
        return this.setCount;
    }

    connected(index1, index2) {
        return this.findSet(index1) == this.findSet(index2);
    }
}
```

1319. 连通网络的操作次数

1、当计算机的数量为n 时，我们至少需要n-1根线才能把他们进行连接。如果线的数量少于n-1，那么我们无论如何都无法将这n台计算机进行连接。

2、如果数组connections的长度小于n-1,我们可以直接返回-1作为答案，否则我们一定可以找到一种操作方式。

3、如果其包含n个节点，那么初始时连通分量数为n，每成功进行一个合并操作，连通分量数就会减少1；

```
 /**
 * @param {number} n
 * @param {number[][]} connections
 * @return {number}
 */
var makeConnected = function (n, connections) {
    if (connections.length < n - 1) {//如果边的数量小于 n-1，则无论怎么样都无法连接所有节点
        return -1;
    }
    let uf = new UnionFind(n);
    for (conn of connections) {
        uf.unite(conn[0], conn[1]);
    }

    return uf.getCount()-1;//遍历结束时，算出操作次数 setCount，setCount-1即为最少操作数

    // 只需改动setCount - 1条边，就能连接所有集合
};

class UnionFind {
    constructor(n) {
        this.parent = new Array(n).fill(0).map((value, index) => index);
        this.size = new Array(n).fill(1);
        this.setCount = n;
    }

    findSet(n) {
        if (this.parent[n] != n) this.parent[n] = this.findSet(this.parent[n]);
        return this.parent[n];
    }

    unite(index1, index2) {
        let root1 = this.findSet(index1), root2 = this.findSet(index2);
        if (root1 != root2) {
            if (this.size[root1] < this.size[root2]) {
                [root1, root2] = [root2, root1];
            }
            this.parent[root2] = root1;
            this.size[root1] += this.size[root2];
            this.setCount--;
        }
    }

        getCount() {
        return this.setCount;
    }

    connected(index1, index2) {
        return this.findSet(index1) == this.findSet(index2);
```

```
    }
}
```