Leetcode python 并查集 中等

130被围绕的区域

题目：给定一个二维的矩阵，包含 'X' 和 'O'（**字母 O**）。

找到所有被 'X' 围绕的区域，并将这些区域里所有的 'O' 用 'X' 填充。

**示例:**

X X X X

X O O X

X X O X

X O X X

运行你的函数后，矩阵变为：

X X X X

X X X X

X X X X

X O X X

**解释:**

被围绕的区间不会存在于边界上，换句话说，任何边界上的 'O' 都不会被填充为 'X'。 任何不在边界上，或不与边界上的 'O' 相连的 'O' 最终都会被填充为 'X'。如果两个元素在水平或垂直方向相邻，则称它们是“相连”的。  
思路：对边界的O进行dfs，把边界O的’O’变为“ \* ” ，然后对边界上的" O "进行DFS，和它连通的‘ O’，都标记为‘ \* ’，最后在遍历数组，把‘ O’变成‘X’，把‘ \* ’，变成‘’O‘

代码：

1. **if** board==None:
2. **return** []
3. **for** i in range(len(board)):
4. **for** j in range(len(board[0])):
5. **if** board[i][j]=='O':
6. **if** i==0 or i==len(board)-1 or j==0 or j==len(board[0])-1:
7. self.robot(board,i,j,len(board),len(board[0]))
8. **for** i in range(len(board)):
9. **for** j in range(len(board[0])):
10. **if** board[i][j]=='O':
11. board[i][j]='X'
12. elif board[i][j]=='\*':
13. board[i][j]='O'
14. def robot(self,board,i,j,m,n):
15. dx=[0,0,1,-1]
16. dy=[1,-1,0,0]
17. board[i][j]='\*'
18. **for** idx in range(4):
19. **if** i+dx[idx]>=0 and i+dx[idx]<m and j+dy[idx]>=0 and j+dy[idx]<n and board[i+dx[idx]][j+dy[idx]]=="O":
20. self.robot(board,i+dx[idx],j+dy[idx],m.n)

200岛屿的个数

题目：

给定一个由 '1'（陆地）和 '0'（水）组成的的二维网格，计算岛屿的数量。一个岛被水包围，并且它是通过水平方向或垂直方向上相邻的陆地连接而成的。你可以假设网格的四个边均被水包围。

**示例 1:**

**输入:**

11110

11010

11000

00000

**输出:** 1

**示例 2:**

**输入:**

11000

11000

00100

00011

**输出:** 3  
思路：我们对每个有“1"的位置进行dfs，把和它四联通的位置全部变成“0”，这样就能把一个点推广到一个岛。

所以，我们总的进行了dfs的次数，就是总过有多少个岛的数目。

注意理解dfs函数的意义：已知当前是1，把它周围相邻的所有1全部转成

代码：

1. **class** Solution(object):
2. def numIslands(self, grid):
3. """
4. :type grid: List[List[str]]
5. :rtype: **int**
6. """
7. res=0
8. **for** r in range(len(grid)):
9. **for** c in range(len(grid[0])):
10. **if** grid[r][c]=="1":
11. self.dfs(grid,r,c)
12. res+=1
13. **return** res
14. def dfs(self,grid,i,j):
15. dirs=[[-1,0],[0,1],[0,-1],[1,0]]
16. grid[i][j]="0"
17. **for** dir in dirs:
18. nr,nc=i+dir[0],j+dir[1]
19. **if** nr>=0 and nc>=0 and nr<len(grid) and nc<len(grid[0]):
20. **if** grid[nr][nc]=="1":
21. self.dfs(grid,nr,nc)

547朋友圈

题目：

班上有 **N**名学生。其中有些人是朋友，有些则不是。他们的友谊具有是传递性。如果已知 A 是 B 的朋友，B 是 C 的朋友，那么我们可以认为 A 也是 C 的朋友。所谓的朋友圈，是指所有朋友的集合。

给定一个 **N \* N**的矩阵 **M**，表示班级中学生之间的朋友关系。如果M[i][j] = 1，表示已知第 i 个和 j 个学生**互为**朋友关系，否则为不知道。你必须输出所有学生中的已知的朋友圈总数。

**示例 1:**

**输入:**

[[1,1,0],

[1,1,0],

[0,0,1]]

**输出:** 2

**说明：**已知学生0和学生1互为朋友，他们在一个朋友圈。

第2个学生自己在一个朋友圈。所以返回2。

**示例 2:**

**输入:**

[[1,1,0],

[1,1,1],

[0,1,1]]

**输出:** 1

**说明：**已知学生0和学生1互为朋友，学生1和学生2互为朋友，所以学生0和学生2也是朋友，所以他们三个在一个朋友圈，返回1。  
思路：典型并查集

代码：

1. **class** DisjointSet:
2. **def** \_\_init\_\_(self,n):
3. self.set\_id=[i **for** i **in** range(n)]
4. self.set\_size=[1 **for** \_ **in** range(n)]
5. self.count=n
6. **def** find(self,p):
7. **while** p!=self.set\_id[p]:
8. self.set\_id[p]=self.set\_id[self.set\_id[p]]
9. p=self.set\_id[p]
10. **return** p
11. **def** union(self,p,q):
12. i=self.find(p)
13. j=self.find(q)
14. **if** i==j:
15. **return**
16. **if** self.set\_size[i] < self.set\_size[j]:
17. self.set\_id[i]=j
18. self.set\_size[j]+=self.set\_size[i]
19. **else**:
20. self.set\_id[j]=i
21. self.set\_size[i]+=self.set\_size[j]
22. self.count -=1
23. **class** Solution(object):
24. **def** findCircleNum(self, M):
25. """
26. :type M: List[List[int]]
27. :rtype: int
28. """
29. disjoint\_set=DisjointSet(len(M))
30. **for** i **in** range(len(M)):
31. **for** j **in** range(len(M)):
32. **if** M[i][j]==1:
33. disjoint\_set.union(i,j)
34. **return** disjoint\_set.count

721账户合并

题目：

给定一个列表 accounts，每个元素 accounts[i] 是一个字符串列表，其中第一个元素 accounts[i][0] 是 *名称 (name)*，其余元素是 *emails* 表示该帐户的邮箱地址。

现在，我们想合并这些帐户。如果两个帐户都有一些共同的邮件地址，则两个帐户必定属于同一个人。请注意，即使两个帐户具有相同的名称，它们也可能属于不同的人，因为人们可能具有相同的名称。一个人最初可以拥有任意数量的帐户，但其所有帐户都具有相同的名称。

合并帐户后，按以下格式返回帐户：每个帐户的第一个元素是名称，其余元素是按顺序排列的邮箱地址。accounts 本身可以以任意顺序返回。

**例子 1:**

**Input:**

accounts = [["John", "johnsmith@mail.com", "john00@mail.com"], ["John", "johnnybravo@mail.com"], ["John", "johnsmith@mail.com", "john\_newyork@mail.com"], ["Mary", "mary@mail.com"]]

**Output:** [["John", 'john00@mail.com', 'john\_newyork@mail.com', 'johnsmith@mail.com'], ["John", "johnnybravo@mail.com"], ["Mary", "mary@mail.com"]]

**Explanation:**

第一个和第三个 John 是同一个人，因为他们有共同的电子邮件 "johnsmith@mail.com"。

第二个 John 和 Mary 是不同的人，因为他们的电子邮件地址没有被其他帐户使用。

我们可以以任何顺序返回这些列表，例如答案[['Mary'，'mary@mail.com']，['John'，'johnnybravo@mail.com']，

['John'，'john00@mail.com'，'john\_newyork@mail.com'，'johnsmith@mail.com']]仍然会被接受。  
思路：

代码：

1. **class** Solution(object):
2. **def** accountsMerge(self, accounts):
3. """
4. :type accounts: List[List[str]]
5. :rtype: List[List[str]]
6. """
7. dsu = DSU()
8. em\_to\_name = dict()
9. em\_to\_id = dict()
10. i = 0
11. **for** acc **in** accounts:
12. name = acc[0]
13. **for** email **in** acc[1:]:
14. em\_to\_name[email] = name
15. **if** email **not** **in** em\_to\_id:
16. em\_to\_id[email] = i
17. i += 1
18. dsu.union(em\_to\_id[acc[1]], em\_to\_id[email])
20. ans = collections.defaultdict(list)
21. **for** email **in** em\_to\_name:
22. ans[dsu.find(em\_to\_id[email])].append(email)
24. **return** [[em\_to\_name[v[0]]] + sorted(v) **for** v **in** ans.values()]
26. **class** DSU:
27. **def** \_\_init\_\_(self):
28. self.par = range(10001)
30. **def** find(self, x):
31. **if** x != self.par[x]:
32. self.par[x] = self.find(self.par[x])
33. **return** self.par[x]
35. **def** union(self, x, y):
36. self.par[self.find(x)] = self.find(y)
38. **def** same(self, x, y):
39. **return** self.find(x) == self.find(y)

947移除最多的同行或同列石头

题目：在二维平面上，我们将石头放置在一些整数坐标点上。每个坐标点上最多只能有一块石头。  
现在，*move* 操作将会移除与网格上的某一块石头共享一列或一行的一块石头。  
我们最多能执行多少次 *move* 操作？

**示例 1：**

**输入：**stones = [[0,0],[0,1],[1,0],[1,2],[2,1],[2,2]]

**输出：**5

**示例 2：**

**输入：**stones = [[0,0],[0,2],[1,1],[2,0],[2,2]]

**输出：**3

**示例 3：**

**输入：**stones = [[0,0]]

**输出：**0

**提示：**

1. 1 <= stones.length <= 1000
2. 0 <= stones[i][j] < 10000  
   思路：横或者纵坐标相等的坐标点会互相链接构成一个区域，问总的有多少个独立的区域。结果是总的石头数减去独立区域数。

所以，我们根本不用考虑太多，只需要统计有多少区域即可。这个方法最简单的就是并查集。

思路是，两重循环，分别判断石头两两之间是否有链接，如果有链接，那么把他们组成同一个区域。这样的优点是我们只需要和石头等长的数组放每个的parent即可。最后统计最后的区域中-1的数量就是独立区域的个数。石头个数减去独立区域数即可

我们不用对石头进行两两判断，而是对他们的横纵坐标同等看待。怎么区分横纵坐标呢？使用的方法是把纵坐标+10000，这样行的索引没变，纵坐标的范围跑到了后面去了。

这个做法的思路是，一个坐标其实就是把横纵坐标对应的两个区域进行了链接。所以，只需要对stones进行一次遍历把对应的区域链接到一起即可。在完成链接之后，我们最后统计一下有多少个独立的区域，需要使用set+find。

代码：

1. **class** Solution(object):
2. **def** removeStones(self, stones):
3. """
4. :type stones: List[List[int]]
5. :rtype: int
6. """
7. N = len(stones)
8. self.map = [-1] \* 20000
9. **for** x, y **in** stones:
10. self.union(x, y + 10000)
11. count = set()
12. **return** N - len({self.find(x) **for** x, y **in** stones})
14. **def** find(self, x):
15. **return** x **if** self.map[x] == -1 **else** self.find(self.map[x])
17. **def** union(self, x, y):
18. fx = self.find(x)
19. fy = self.find(y)
20. **if** fx != fy:
21. self.map[fx] = fy