Leetcode(python) 图 中等题

133克隆图

题目：

克隆一张无向图，图中的每个节点包含一个 label （标签）和一个 neighbors （邻接点）列表 。

**OJ的无向图序列化：**

节点被唯一标记。

我们用 # 作为每个节点的分隔符，用 , 作为节点标签和邻接点的分隔符。

例如，序列化无向图 {0,1,2#1,2#2,2}。

该图总共有三个节点, 被两个分隔符  # 分为三部分。

1. 第一个节点的标签为 0，存在从节点 0 到节点 1 和节点 2 的两条边。
2. 第二个节点的标签为 1，存在从节点 1 到节点 2 的一条边。
3. 第三个节点的标签为 2，存在从节点 2 到节点 2 (本身) 的一条边，从而形成自环。

我们将图形可视化如下：

1

/ \

/ \

0 --- 2

/ \

\\_/

思路：直接深度拷贝

代码：

1. # Definition for a undirected graph node
2. # class UndirectedGraphNode:
3. #     def \_\_init\_\_(self, x):
4. #         self.label = x
5. #         self.neighbors = []
7. **class** Solution:
8. # @param node, a undirected graph node
9. # @return a undirected graph node
10. **def** cloneGraph(self, node):
11. **if** node==None:
12. **return** node
13. d={}
14. **def** dfs(n):
15. **if** n **in** d:
16. **return** d[n]
17. ans=UndirectedGraphNode(n.label)
18. d[n]=ans
19. **for** i **in** n.neighbors:
20. ans.neighbors.append(dfs(i))
21. **return** ans
22. **return** dfs(node)

207课程表

题目：

现在你总共有 *n* 门课需要选，记为 0 到 n-1。

在选修某些课程之前需要一些先修课程。 例如，想要学习课程 0 ，你需要先完成课程 1 ，我们用一个匹配来表示他们: [0,1]

给定课程总量以及它们的先决条件，判断是否可能完成所有课程的学习？

**示例 1:**

**输入:** 2, [[1,0]]

**输出:** true

**解释:** 总共有 2 门课程。学习课程 1 之前，你需要完成课程 0。所以这是可能的。

**示例 2:**

**输入:** 2, [[1,0],[0,1]]

**输出:** false

**解释:** 总共有 2 门课程。学习课程 1 之前，你需要先完成​课程 0；并且学习课程 0 之前，你还应先完成课程 1。这是不可能的。

说明:

1. 输入的先决条件是由边缘列表表示的图形，而不是邻接矩阵。详情请参见[图的表示法](http://blog.csdn.net/woaidapaopao/article/details/51732947" \t "_blank)。
2. 你可以假定输入的先决条件中没有重复的边。

提示:

1. 这个问题相当于查找一个循环是否存在于有向图中。如果存在循环，则不存在拓扑排序，因此不可能选取所有课程进行学习。
2. [通过 DFS 进行拓扑排序](https://www.coursera.org/specializations/algorithms) - 一个关于Coursera的精彩视频教程（21分钟），介绍拓扑排序的基本概念。
3. 拓扑排序也可以通过 [BFS](https://baike.baidu.com/item/%E5%AE%BD%E5%BA%A6%E4%BC%98%E5%85%88%E6%90%9C%E7%B4%A2/5224802?fr=aladdin&fromid=2148012&fromtitle=%E5%B9%BF%E5%BA%A6%E4%BC%98%E5%85%88%E6%90%9C%E7%B4%A2) 完成。

思路一：判断这些课程能否构成有向无环图（DAG）。而任何时候判断DAG的方法要立刻想到拓扑排序。

拓扑排序是对有向无环图（DAG）而言的，对图进行拓扑排序即求其中节点的一个拓扑序列，对于所有的有向边（U, V）（由U指向V），在该序列中节点U都排在节点V之前。

方法是每次选择入度为0的节点，作为序列的下一个节点，然后移除该节点和以从节点出发的所有边。

那这个方法比较简单粗暴了：要循环N次，这个循环次数并不是遍历节点的意思，而是我们如果正常取点的话，N次就能把所有的节点都取完了，如果N次操作结束还没判断出来，那么就不是DAG.在这N次中，每次都找一个入度为0的点，并把它的入度变为-1，作为已经取过的点不再使用，同时把从这个点指向的点的入度都-1.

这个过程中，如果找不到入度为0的点，那么说明存在环。如果N次操作，每次都操作成功的去除了一个入度为0的点，那么说明这个图是DAG.

时间复杂度是O(N ^ 2)，空间复杂度是O(N)。

代码一：

1. **class** Solution(object):
2. **def** canFinish(self, numCourses, prerequisites):
3. """
4. :type numCourses: int
5. :type prerequisites: List[List[int]]
6. :rtype: bool
7. """
8. graph=collections.defaultdict(list)
9. indegrees=collections.defaultdict(int)
10. **for** u,v **in** prerequisites:
11. graph[v].append(u)
12. indegrees[u]+=1
13. **for** i **in** range(numCourses):
14. zeroDegree=False
15. **for** j **in** range(numCourses):
16. **if** indegrees[j]==0:
17. zeroDegree=True
18. **break**
19. **if** **not** zeroDegree:
20. **return** False
21. indegrees[j]=-1
22. **for** node **in** graph[j]:
23. indegrees[node]-=1
24. **return** True

思路二：这个方法是，我们每次找到一个新的点，判断从这个点出发是否有环。

具体做法是使用一个visited数组，当visited[i]值为0，说明还没判断这个点；当visited[i]值为1，说明当前的循环正在判断这个点；当visited[i]值为2，说明已经判断过这个点，含义是从这个点往后的所有路径都没有环，认为这个点是安全的。

那么，我们对每个点出发都做这个判断，检查这个点出发的所有路径上是否有环，如果判断过程中找到了当前的正在判断的路径，说明有环；找到了已经判断正常的点，说明往后都不可能存在环，所以认为当前的节点也是安全的。如果当前点是未知状态，那么先把当前点标记成正在访问状态，然后找后续的节点，直到找到安全的节点为止。最后如果到达了无路可走的状态，说明当前节点是安全的。

findOrder函数中的for循环是怎么回事呢？这个和BFS循环次数不是同一个概念，这里的循环就是看从第i个节点开始能否到达合理结果。这个节点可能没有出度了，那就把它直接放到path里；也可能有出度，那么就把它后面的节点都进行一次遍历，如果满足条件的节点都放到path里，同时把这次遍历的所有节点都标记成了已经遍历；如果一个节点已经被安全的访问过，那么就放过它，继续遍历下个节点。

时间复杂度是O(N)，空间复杂度是O(N)。

代码二：

1. **class** Solution(object):
2. **def** canFinish(self, numCourses, prerequisites):
3. """
4. :type numCourses: int
5. :type prerequisites: List[List[int]]
6. :rtype: bool
7. """
8. graph = collections.defaultdict(list)
9. **for** u, v **in** prerequisites:
10. graph[u].append(v)
11. # 0 = Unknown, 1 = visiting, 2 = visited
12. visited = [0] \* N
13. **for** i **in** range(N):
14. **if** **not** self.dfs(graph, visited, i):
15. **return** False
16. **return** True
17. **def** dfs(self, graph, visited, i):
18. **if** visited[i] == 1: **return** False
19. **if** visited[i] == 2: **return** True
20. visited[i] = 1
21. **for** j **in** graph[i]:
22. **if** **not** self.dfs(graph, visited, j):
23. **return** False
24. visited[i] = 2
25. **return** True

210课程表II

题目：

现在你总共有 *n* 门课需要选，记为 0 到 n-1。

在选修某些课程之前需要一些先修课程。 例如，想要学习课程 0 ，你需要先完成课程 1 ，我们用一个匹配来表示他们: [0,1]

给定课程总量以及它们的先决条件，返回你为了学完所有课程所安排的学习顺序。

可能会有多个正确的顺序，你只要返回一种就可以了。如果不可能完成所有课程，返回一个空数组。

**示例 1:**

**输入:** 2, [[1,0]]

**输出:** [0,1]

**解释:** 总共有 2 门课程。要学习课程 1，你需要先完成课程 0。因此，正确的课程顺序为 [0,1] 。

**示例 2:**

**输入:** 4, [[1,0],[2,0],[3,1],[3,2]]

**输出:** [0,1,2,3] or [0,2,1,3]

**解释:** 总共有 4 门课程。要学习课程 3，你应该先完成课程 1 和课程 2。并且课程 1 和课程 2 都应该排在课程 0 之后。

  因此，一个正确的课程顺序是 [0,1,2,3] 。另一个正确的排序是 [0,2,1,3] 。

思路一：第一种做法是使用BFS，也是拓扑排序最朴素的思想：每次找到入度为0的节点，把他放到结果里，然后再找第二个入度为0的点等等。

那这个方法比较简单粗暴了：要循环N次，这个循环次数并不是遍历节点的意思，而是我们如果正常取点的话，N次就能把所有的节点都取完了，如果N次操作结束还没判断出来，那么就不是DAG.在这N次中，每次都找一个入度为0的点，并把它的入度变为-1，作为已经取过的点不再使用，同时把从这个点指向的点的入度都-1.

这个过程中，如果找不到入度为0的点，那么说明存在环。如果N次操作，每次都操作成功的去除了一个入度为0的点，那么说明这个图是DAG.

这个做法确实很简洁，只要使用列表Path保存每次入度为0的点，就保存了每次访问的路径

代码：

1. **class** Solution(object):
2. **def** findOrder(self, numCourses, prerequisites):
3. """
4. :type numCourses: int
5. :type prerequisites: List[List[int]]
6. :rtype: List[int]
7. """
8. graph = collections.defaultdict(list)
9. indegrees = collections.defaultdict(int)
10. **for** u, v **in** prerequisites:
11. graph[v].append(u)
12. indegrees[u] += 1
13. path = []
14. **for** i **in** range(numCourses):
15. zeroDegree = False
16. **for** j **in** range(numCourses):
17. **if** indegrees[j] == 0:
18. zeroDegree = True
19. **break**
20. **if** **not** zeroDegree:
21. **return** []
22. indegrees[j] -= 1
23. path.append(j)
24. **for** node **in** graph[j]:
25. indegrees[node] -= 1
26. **return** path

思路二：

具体做法是使用一个visited数组，当visited[i]值为0，说明还没判断这个点；当visited[i]值为1，说明当前的循环正在判断这个点；当visited[i]值为2，说明已经判断过这个点，含义是从这个点往后的所有路径都没有环，认为这个点是安全的。

那么，我们对每个点出发都做这个判断，检查这个点出发的所有路径上是否有环，如果判断过程中找到了当前的正在判断的路径，说明有环；找到了已经判断正常的点，说明往后都不可能存在环，所以认为当前的节点也是安全的。如果当前点是未知状态，那么先把当前点标记成正在访问状态，然后找后续的节点，直到找到安全的节点为止。最后如果到达了无路可走的状态，说明当前节点是安全的。

findOrder函数中的for循环是怎么回事呢？这个和BFS循环次数不是同一个概念，这里的循环就是看从第i个节点开始能否到达合理结果。这个节点可能没有出度了，那就把它直接放到path里；也可能有出度，那么就把它后面的节点都进行一次遍历，如果满足条件的节点都放到path里，同时把这次遍历的所有节点都标记成了已经遍历；如果一个节点已经被安全的访问过，那么就放过它，继续遍历下个节点。

代码：

1. **class** Solution(object):
2. **def** findOrder(self, numCourses, prerequisites):
3. """
4. :type numCourses: int
5. :type prerequisites: List[List[int]]
6. :rtype: List[int]
7. """
8. graph = collections.defaultdict(list)
9. **for** u, v **in** prerequisites:
10. graph[u].append(v)
11. # 0 = Unknown, 1 = visiting, 2 = visited
12. visited = [0] \* numCourses
13. path = []
14. **for** i **in** range(numCourses):
15. **if** **not** self.dfs(graph, visited, i, path):
16. **return** []
17. **return** path
19. **def** dfs(self, graph, visited, i, path):
20. **if** visited[i] == 1: **return** False
21. **if** visited[i] == 2: **return** True
22. visited[i] = 1
23. **for** j **in** graph[i]:
24. **if** **not** self.dfs(graph, visited, j, path):
25. **return** False
26. visited[i] = 2
27. path.append(i)
28. **return** True

310最小高度树

题目：对于一个具有树特征的无向图，我们可选择任何一个节点作为根。图因此可以成为树，在所有可能的树中，具有最小高度的树被称为最小高度树。给出这样的一个图，写出一个函数找到所有的最小高度树并返回他们的根节点。

**格式**

该图包含 n 个节点，标记为 0 到 n - 1。给定数字 n 和一个无向边 edges 列表（每一个边都是一对标签）。

你可以假设没有重复的边会出现在 edges 中。由于所有的边都是无向边， [0, 1]和 [1, 0] 是相同的，因此不会同时出现在 edges 里。

**示例 1:**

**输入:** n = 4, edges = [[1, 0], [1, 2], [1, 3]]

0

|

1

/ \

2 3

**输出:** [1]

**示例 2:**

**输入:** n = 6, edges = [[0, 3], [1, 3], [2, 3], [4, 3], [5, 4]]

0 1 2

\ | /

3

|

4

|

5

**输出:** [3, 4]

思路：这个题给定的是个图，但是让我们构建成树，也就是说构建出来的并不是二叉树。题目其实想考我们的是，整个图最靠近中间的节点是什么。我们使用类似与拓扑排序的BFS进行解决。

拓扑排序我们都知道，每次选择入度为0的节点进行删除。在这个题中，因为我们要找到无向图最靠近中间的节点，所以，我们先使用一个字典保存每个节点的所有相邻节点set。每次把所有只有一个邻接的节点（叶子节点，类似于入度为0，但是这是个无向图，入度等于出度）都放入队列，然后遍历队列中的节点u，把和每个节点u相邻的节点v的set删去u，所以这一步操作得到的是去除了叶子节点的新一轮的图。所以我们需要再次进行选择只有一个邻接节点的叶子节点，然后放入队列中，再次操作。最后结束的标准是，整个图只留下了1个或者两个元素。为什么不能是3个呢？因为题目第一句话说了给出的图是具有树的特性的，所以一定没有环存在。

这个题整体的思路就是把所有的叶子节点放入队列中，然后同时向中间遍历，这样最后剩下来的就是整棵树中间的元素

代码：

1. **class** Solution(object):
2. **def** findMinHeightTrees(self, n, edges):
3. """
4. :type n: int
5. :type edges: List[List[int]]
6. :rtype: List[int]
7. """
8. **if** n==1:
9. **return** [0]
10. leaves=collections.defaultdict(set)
11. **for** u,v **in** edges:
12. leaves[u].add(v)
13. leaves[v].add(u)
14. que=collections.deque()
15. **for** u,vs **in** leaves.items():
16. **if** len(vs)==1:
17. que.append(u)
18. **while** n>2:
19. \_len=len(que)
20. n-=\_len
21. **for** \_ **in** range(\_len):
22. u=que.popleft()
23. **for** v **in** leaves[u]:
24. leaves[v].remove(u)
25. **if** len(leaves[v])==1:
26. que.append(v)
27. **return** list(que)

332重新安排行程

题目：给定一个机票的字符串二维数组 [from, to]，子数组中的两个成员分别表示飞机出发和降落的机场地点，对该行程进行重新规划排序。所有这些机票都属于一个从JFK（肯尼迪国际机场）出发的先生，所以该行程必须从 JFK 出发。

**说明:**

1. 如果存在多种有效的行程，你可以按字符自然排序返回最小的行程组合。例如，行程 ["JFK", "LGA"] 与 ["JFK", "LGB"] 相比就更小，排序更靠前
2. 所有的机场都用三个大写字母表示（机场代码）。
3. 假定所有机票至少存在一种合理的行程。

**示例 1:**

**输入:** [["MUC", "LHR"], ["JFK", "MUC"], ["SFO", "SJC"], ["LHR", "SFO"]]

**输出:** ["JFK", "MUC", "LHR", "SFO", "SJC"]

**示例 2:**

**输入:** [["JFK","SFO"],["JFK","ATL"],["SFO","ATL"],["ATL","JFK"],["ATL","SFO"]]

**输出:** ["JFK","ATL","JFK","SFO","ATL","SFO"]

**解释:** 另一种有效的行程是 ["JFK","SFO","ATL","JFK","ATL","SFO"]。但是它自然排序更大更靠后。

思路：这道题的本质是计算一个最"小"的欧拉路径(Eulerian path)。对于一个节点（当然先从JFK开始)，贪心地访问最小的邻居，访问过的边全部删除。当碰到死路的时候就回溯到最近一个还有出路的节点，然后把回溯的路径放到最后去访问，这个过程和后序遍历的一样。1. 如果子节点没有死路（每个节点都只左子树），前序遍历便是欧拉路径。2. 如果子节点1是死路，子节点2完成了遍历，那么子节点2先要被访问。1，2都和后序遍历的顺序正好相反。

其中，如果碰到死路，而没有把所有的边都走过一遍的话，就说明这种走法不满足itinerary，需要沿着树根向上找到最近的一个有其他路可以走的节点N，把新的路走一遍。因为题目保证一定存在一条满足要求的itinerary路径，那么一条这样的死路，一定会相对的在这个节点N上存在另一条路，这条路存在一个回到该节点N的环。先把这个环走过之后再去走这条死路，就可以保证把以N为树根的这个路径上的所有点都走到。

首先肯定是要把路径保存成链表法表示的图的。然后对每个顶点的所有邻接顶点进行排序，这样我们每次都优先选择字典序最小的那个顶点作为下次遍历的节点。我们做了后序遍历即可。最后还要把后序遍历的结果再翻转，才是从根节点出发到每个位置的路径。

代码：

1. **class** Solution(object):
2. **def** findItinerary(self, tickets):
3. """
4. :type tickets: List[List[str]]
5. :rtype: List[str]
6. """
7. graph=collections.defaultdict(list)
8. **for** frm,to **in** tickets:
9. graph[frm].append(to)
10. **for** frm,tos **in** graph.items():
11. tos.sort(reverse=True)
12. res=[]
13. self.dfs(graph,"JFK",res)
14. **return** res[::-1]
15. **def** dfs(self,graph,source,res):
16. **while** graph[source]:
17. v=graph[source].pop()
18. self.dfs(graph,v,res)
19. res.append(source)

399除法求值

题目：

给出方程式 A / B = k, 其中 A 和 B 均为代表字符串的变量， k 是一个浮点型数字。根据已知方程式求解问题，并返回计算结果。如果结果不存在，则返回 -1.0。

**示例 :**  
给定 a / b = 2.0, b / c = 3.0  
问题: a / c = ?, b / a = ?, a / e = ?, a / a = ?, x / x = ?   
返回 [6.0, 0.5, -1.0, 1.0, -1.0 ]

输入为: vector<pair<string, string>> equations, vector<double>& values, vector<pair<string, string>> queries(方程式，方程式结果，问题方程式)， 其中 equations.size() == values.size()，即方程式的长度与方程式结果长度相等（程式与结果一一对应），并且结果值均为正数。以上为方程式的描述。 返回vector<double>类型。

基于上述例子，输入如下：

equations(方程式) = [ ["a", "b"], ["b", "c"] ],

values(方程式结果) = [2.0, 3.0],

queries(问题方程式) = [ ["a", "c"], ["b", "a"], ["a", "e"], ["a", "a"], ["x", "x"] ].

输入总是有效的。你可以假设除法运算中不会出现除数为0的情况，且不存在任何矛盾的结果。

思路：题目中给了顶点和顶点之间的关系，其实就是制定了这个图的样子。然后要求的新的比值其实就是从一个顶点到达另外一个顶点的路径，并且把这条路径上所有的权重相乘。

注意，如果a/b=3，那么从a到b是3，那么从b到a是1/3.

既然是从一个顶点出发到达另外一个顶点，所以应该是dfs解决的问题。

代码：

1. **class** Solution(object):
2. **def** calcEquation(self, equations, values, queries):
3. """
4. :type equations: List[List[str]]
5. :type values: List[float]
6. :type queries: List[List[str]]
7. :rtype: List[float]
8. """
9. table = collections.defaultdict(dict)
10. **for** (x, y), value **in** zip(equations, values):
11. table[x][y] = value
12. table[y][x] = 1.0 / value
13. ans = [self.dfs(x, y, table, set()) **if** x **in** table **and** y **in** table **else** -1.0 **for** (x, y) **in** queries]
14. **return** ans
16. **def** dfs(self, x, y, table, visited):
17. **if** x == y:
18. **return** 1.0
19. visited.add(x)
20. **for** n **in** table[x]:
21. **if** n **in** visited: **continue**
22. visited.add(n)
23. d = self.dfs(n, y, table, visited)
24. **if** d > 0:
25. **return** d \* table[x][n]
26. **return** -1.0

684冗余连接（做过）

785判断二分图

题目：

给定一个无向图graph，当这个图为二分图时返回true。

如果我们能将一个图的节点集合分割成两个独立的子集A和B，并使图中的每一条边的两个节点一个来自A集合，一个来自B集合，我们就将这个图称为二分图。

graph将会以邻接表方式给出，graph[i]表示图中与节点i相连的所有节点。每个节点都是一个在0到graph.length-1之间的整数。这图中没有自环和平行边： graph[i] 中不存在i，并且graph[i]中没有重复的值。

**示例 1:**

输入**:** [[1,3], [0,2], [1,3], [0,2]]

**输出:** true

**解释:**

无向图如下:

0----1

| |

| |

3----2

我们可以将节点分成两组: {0, 2} 和 {1, 3}。

**示例 2:**

**输入:** [[1,2,3], [0,2], [0,1,3], [0,2]]

**输出:** false

**解释:**

无向图如下:

0----1

| \ |

| \ |

3----2

我们不能将节点分割成两个独立的子集。

**注意:**

* graph 的长度范围为 [1, 100]。
* graph[i] 中的元素的范围为 [0, graph.length - 1]。
* graph[i] 不会包含 i 或者有重复的值。
* 图是无向的: 如果j 在 graph[i]里边, 那么 i 也会在 graph[j]里

思路：使用染色法。可以通过BFS或者DFS来解决。我使用的是BFS.

使用一个visited数组来保存每个节点被染的颜色。0代表没染色，1代表染成蓝色，2代表染成红色。对图的每个顶点进行一个遍历，把和这个顶点相邻的顶点全部染成相反的颜色。如果相邻顶点已经染色，而且染色和当前顶点染色相同，则返回False。全部成功染色后返回True。

这个题没有说明是连通图，这个就很坑爹了，不能通过一次的BFS就把所有的顶点染色成功。所以需要的是一个外层的对顶点进行遍历，一个内层的对每个顶点相邻的顶点遍历，这样两重遍历才能保证每个顶点、这个顶点相邻的顶点都被强行的染色。

时间复杂度是O(E+V)，空间复杂度是O(E).

代码：

1. **class** Solution(object):
2. **def** isBipartite(self, graph):
3. """
4. :type graph: List[List[int]]
5. :rtype: bool
6. """
7. visited=[0]\*len(graph)
8. **for** i **in** range(len(graph)):
9. **if** graph[i] **and** visited[i]==0:
10. visited[i]=1
11. q=collections.deque()
12. q.append(i)
13. **while** q:
14. v=q.popleft()
15. **for** e **in** graph[v]:
16. **if** visited[e]!=0:
17. **if** visited[e]==visited[v]:
18. **return** False
19. **else**:
20. visited[e]=3-visited[v]
21. q.append(e)
22. **return** True

802找到最终的安全状态

题目：在有向图中, 我们从某个节点和每个转向处开始, 沿着图的有向边走。 如果我们到达的节点是终点 (即它没有连出的有向边), 我们停止。

现在, 如果我们最后能走到终点，那么我们的起始节点是*最终安全*的。 更具体地说, 存在一个自然数 K,  无论选择从哪里开始行走, 我们走了不到 K 步后必能停止在一个终点。

哪些节点最终是安全的？ 结果返回一个有序的数组。

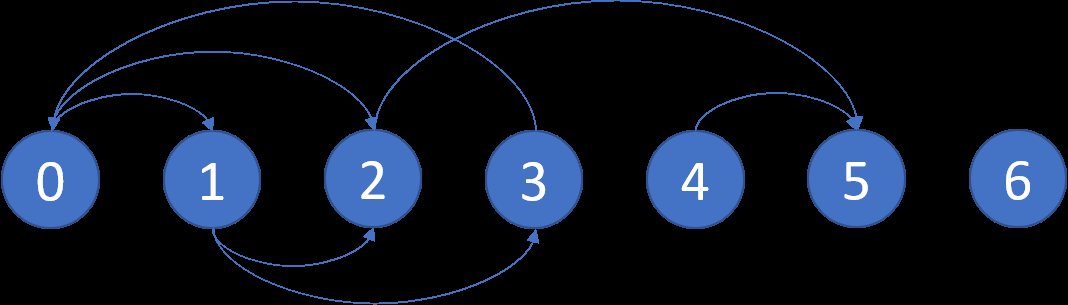
该有向图有 N 个节点，标签为 0, 1, ..., N-1, 其中 N 是 graph 的节点数.  图以以下的形式给出: graph[i] 是节点 j 的一个列表，满足 (i, j) 是图的一条有向边。

**示例：**

**输入：**graph = [[1,2],[2,3],[5],[0],[5],[],[]]

**输出：**[2,4,5,6]

这里是上图的示意图。



**提示：**

* graph 节点数不超过 10000.
* 图的边数不会超过 32000.
* 每个 graph[i] 被排序为不同的整数列表， 在区间 [0, graph.length - 1] 中选取。

思路：题目很容易抽象成一个查找一个节点是否在环中，或者经过一段路径之后在一个环中。所以使用的方法是DFS。

用0代表没有访问过，用1代表安全，用2代表不安全。其实就是把visited数组给拓展成了染色数组。

dfs函数的含义就是返回start节点是否是安全，如果是，返回True。

值得注意的是，默认是不安全还是安全。我刚开始考虑的是默认不安全，如果找到一个安全的路径就是安全的。这个是不对的，因为虽然这个节点通过一段路径之后能到达一个终点，但是经过另一个路径它就会进入环中。题目问的就是无论如何走都必须到达终点，即无论如何走都不会到达环中，这样的才是安全的。所以默认应该是不安全的

代码：

1. **class** Solution(object):
2. **def** eventualSafeNodes(self, graph):
3. """
4. :type graph: List[List[int]]
5. :rtype: List[int]
6. """
8. color = [0] \* len(graph)
9. res = []
10. **for** start **in** range(len(graph)):
11. **if** self.dfs(graph, start, color):
12. res.append(start)
13. res.sort()
14. **return** res
16. **def** dfs(self, graph, start, color):
17. # 返回start节点是否是安全，如果是，返回True
18. **if** color[start] != 0:
19. **return** color[start] == 1
20. color[start] = 2
21. **for** e **in** graph[start]:
22. **if** **not** self.dfs(graph, e, color):
23. **return** False
24. color[start] = 1
25. **return** True

841钥匙和房间

题目：有 N 个房间，开始时你位于 0 号房间。每个房间有不同的号码：0，1，2，...，N-1，并且房间里可能有一些钥匙能使你进入下一个房间。

在形式上，对于每个房间 i 都有一个钥匙列表 rooms[i]，每个钥匙 rooms[i][j] 由 [0,1，...，N-1] 中的一个整数表示，其中 N = rooms.length。 钥匙 rooms[i][j] = v 可以打开编号为 v 的房间。

最初，除 0 号房间外的其余所有房间都被锁住。

你可以自由地在房间之间来回走动。

如果能进入每个房间返回 true，否则返回 false。

**示例 1：**

**输入:** [[1],[2],[3],[]]

**输出:** true

**解释:**

我们从 0 号房间开始，拿到钥匙 1。

之后我们去 1 号房间，拿到钥匙 2。

然后我们去 2 号房间，拿到钥匙 3。

最后我们去了 3 号房间。

由于我们能够进入每个房间，我们返回 true。

**示例 2：**

**输入：**[[1,3],[3,0,1],[2],[0]]

**输出：**false

**解释：**我们不能进入 2 号房间。

思路：我们发现最后要求出是否存在一个解。这个解是通过一段深度遍历求得。

所以很快的写出来一段dfs，dfs里把当前的门打开，并看这个房间的钥匙，找到还没去过的房间，把门打开，依次类推。

这样，我们就遍历了所有的能去到的房间，最后看一下是否所有的房间都经历过即可。

代码：

1. **class** Solution(object):
2. **def** canVisitAllRooms(self, rooms):
3. """
4. :type rooms: List[List[int]]
5. :rtype: bool
6. """
7. visited=[0]\*len(rooms)
8. self.dfs(rooms,0,visited)
9. **return** sum(visited)==len(rooms)
10. **def** dfs(self,rooms,index,visited):
11. visited[index]=1
12. **for** key **in** rooms[index]:
13. **if** **not** visited[key]:
14. self.dfs(rooms,key,visited)

959由斜杆划分区域

题目：在由 1 x 1 方格组成的 N x N 网格 grid 中，每个 1 x 1 方块由 /、\ 或空格构成。这些字符会将方块划分为一些共边的区域。

（请注意，反斜杠字符是转义的，因此 \ 用 "\\" 表示。）。

返回区域的数目。

**示例 1：**

**输入：**

[

  " /",

  "/ "

]

**输出：**2

**解释：**2x2 网格如下：



**示例 2：**

**输入：**

[

  " /",

  " "

]

**输出：**1

**解释：**2x2 网格如下：



**示例 3：**

**输入：**

[

  "\\/",

  "/\\"

]

**输出：**4

**解释：**（回想一下，因为 \ 字符是转义的，所以 "\\/" 表示 \/，而 "/\\" 表示 /\。）

2x2 网格如下：



**示例 4：**

**输入：**

[

  "/\\",

  "\\/"

]

**输出：**5

**解释：**（回想一下，因为 \ 字符是转义的，所以 "/\\" 表示 /\，而 "\\/" 表示 \/。）

2x2 网格如下：



**示例 5：**

**输入：**

[

  "//",

  "/ "

]

**输出：**3

**解释：**2x2 网格如下：



**提示：**

1. 1 <= grid.length == grid[0].length <= 30
2. grid[i][j] 是 '/'、'\'、或 ' '。

思路：

这个题给的例子虽然都是22的，但是事实上是NN的，需要注意下。这个题其实没那么难，它主要是给定了切割的方式，问我们切割之后还有多少个联通的区域。

既然是问联通的方案，那么我们就根据划定的范围来知道哪些是联通的。具体做法当然就是并查集。

--------

| \ 0 / |

| 3\ / |

| /\ 1 |

|/ 2 \ |

|-------|

像上面这样划分把每一个格子进行4块划分，从上面开始顺时针编号为0,1,2,3. 并查集的使用方式是，如果两个格子相邻，那么就把他们格子的相邻编号合并到一起。然后根据格子自身的画线方式，再把自身的不同区域合并到一起。

显而易见，每行的左边各自的1和右边的那个3是相邻的，每列的上边的2和下边的2是相邻的。自身的格子划分方式是：如果是'/'，那么0和3,1和2分别相邻；如果格子划分方式是'\\'，那么0和1,3和2分别相邻。

最后统计不同的区域有多少，想一下，如果都不相邻，那么有N\*N个区域，但是每次合并都会造成少了一个区域。所以直接在合并的时候减去区域即可。

---------------------

作者：负雪明烛

来源：CSDN

原文：https://blog.csdn.net/fuxuemingzhu/article/details/85039057

版权声明：本文为博主原创文章，转载请附上博文链接！

代码：

1. **class** Solution(object):
2. **def** regionsBySlashes(self, grid):
3. """
4. :type grid: List[str]
5. :rtype: int
6. """
7. self.N = len(grid)
8. m\_ = range(self.N \* self.N \* 4)
9. self.count = self.N \* self.N \* 4
10. **for** r **in** range(self.N):
11. line = grid[r]
12. **for** c **in** range(self.N):
13. w = line[c]
14. **if** r > 0:
15. self.u(m\_, self.g(r - 1, c, 2), self.g(r, c, 0))
16. **if** c > 0:
17. self.u(m\_, self.g(r, c - 1, 1), self.g(r, c, 3))
18. **if** w != '/':
19. self.u(m\_, self.g(r, c, 0), self.g(r, c, 1))
20. self.u(m\_, self.g(r, c, 3), self.g(r, c, 2))
21. **if** w != '\\':
22. self.u(m\_, self.g(r, c, 0), self.g(r, c, 3))
23. self.u(m\_, self.g(r, c, 1), self.g(r, c, 2))
24. **return** self.count
26. **def** f(self, m\_, a):
27. **if** m\_[a] == a:
28. **return** a
29. **return** self.f(m\_, m\_[a])
31. **def** u(self, m\_, a, b):
32. pa = self.f(m\_, a)
33. pb = self.f(m\_, b)
34. **if** (pa == pb):
35. **return**
36. m\_[pa] = pb
37. self.count -= 1
39. **def** g(self, r, c, i):
40. **return** (r \* self.N + c) \* 4 + i