**684. 冗余连接**

在本问题中, 树指的是一个连通且无环的无向图。

输入一个图，该图由一个有着N个节点 (节点值不重复1, 2, ..., N) 的树及一条附加的边构成。附加的边的两个顶点包含在1到N中间，这条附加的边不属于树中已存在的边。

结果图是一个以边组成的二维数组。每一个边的元素是一对[u, v] ，满足 u < v，表示连接顶点u 和v的无向图的边。

返回一条可以删去的边，使得结果图是一个有着N个节点的树。如果有多个答案，则返回二维数组中最后出现的边。答案边 [u, v] 应满足相同的格式 u < v。

输入: [[1,2], [1,3], [2,3]]

输出: [2,3]

解释: 给定的无向图为:

1

/ \

2 - 3

输入: [[1,2], [2,3], [3,4], [1,4], [1,5]]

输出: [1,4]

解释: 给定的无向图为:

5 - 1 - 2

| |

4 - 3

注意:

输入的二维数组大小在 3 到 1000。

二维数组中的整数在1到N之间，其中N是输入数组的大小。

|  |
| --- |
| class UnionFind {  public:  UnionFind(int n) {  parent\_.resize(n);  iota(parent\_.begin(), parent\_.end(), 0);  }  int Find(int target) {  if (parent\_[target] != target) {  parent\_[target] = Find(parent\_[target]);  }  return parent\_[target];  }  bool IsUnion(int p, int q) {  return Find(p) == Find(q);  }  void Union(int p, int q) {  int pRoot = Find(p);  int qRoot = Find(q);  parent\_[pRoot] = qRoot;  }  vector<int> parent\_;  };  class Solution {  public:  vector<int> findRedundantConnection(vector<vector<int>>& edges) {  int maxNum = 0;  for (auto &edge : edges) {  maxNum = max(maxNum, max(edge[0], edge[1]));  }  UnionFind uf(maxNum + 1);  for (auto &edge : edges) {  if (uf.IsUnion(edge[0], edge[1])) {  return { edge[0], edge[1] };  }  uf.Union(edge[0], edge[1]);  }  return {};  }  }; |

**839. 相似字符串组**

如果我们交换字符串 X 中的两个不同位置的字母，使得它和字符串 Y 相等，那么称 X 和 Y 两个字符串相似。如果这两个字符串本身是相等的，那它们也是相似的。

例如，"tars" 和 "rats" 是相似的 (交换 0 与 2 的位置)； "rats" 和 "arts" 也是相似的，但是 "star" 不与 "tars"，"rats"，或 "arts" 相似。

总之，它们通过相似性形成了两个关联组：{"tars", "rats", "arts"} 和 {"star"}。注意，"tars" 和 "arts" 是在同一组中，即使它们并不相似。形式上，对每个组而言，要确定一个单词在组中，只需要这个词和该组中至少一个单词相似。

我们给出了一个字符串列表 A。列表中的每个字符串都是 A 中其它所有字符串的一个字母异位词。请问 A 中有多少个相似字符串组？

示例：

输入：["tars","rats","arts","star"]

输出：2

思路：寻找连通子图的升级版。遍历所有节点，遇到没有归属的节点则用dfs或bfs查找其全部连通节点。 1.用变量cluster记录当前连通子图的个数； 2.用vector记录节点所属的连通子图； 3.遍历vector，遇到不为0的跳过（表示已有归属）；遇到0则cluster加一，并用dfs查找所有连通节点，并标记为cluster。

|  |
| --- |
| class Solution {  public:  vector<int> parents;  int roots;  int numSimilarGroups(vector<string>& A) {  roots = A.size();  parents = vector<int>(A.size());  for (int i = 0; i < parents.size(); i++) {  parents[i] = i;  }  for (int i = 0; i < A.size() - 1; i++) {  for (int j = i + 1; j < A.size(); j++) {  if (isSimilar(A[i], A[j])) {  int pi = find(i);  int pj = find(j);  if (pi != pj) {  merge(pi, pj);  roots--;  }  }  }  }  return roots;  }  int find(int x) {  return parents[x] == x ? x : parents[x] = find(parents[x]);  }  void merge(int x, int y) {  parents[x] = y;  }  bool isSimilar(string& a, string& b) {  int diff = 0;  for (int i = 0; i < a.length(); i++) {  if (a[i] != b[i]) diff++;  }  return diff <= 2;  }  }; |

**128. 最长连续序列**

给定一个未排序的整数数组，找出最长连续序列的长度。

要求算法的时间复杂度为 O(n)。

示例:

输入: [100, 4, 200, 1, 3, 2]

输出: 4

解释: 最长连续序列是 [1, 2, 3, 4]。它的长度为 4。

|  |
| --- |
| class UF {  public:  UF(const vector<int>& nums)  {  for (auto num : nums) {  hash\_[num] = num;  }  }  int Find(int id)  {  int parent = hash\_[id];  if (id != parent) {  hash\_[id] = Find(parent);  }  return hash\_[id];  }  void Union(int l, int r)  {  int lParent = Find(l);  int rParent = Find(r);  hash\_[lParent] = rParent;  }  int GetMaxParentNum()  {  map<int, int> counts;  for (const auto& item : hash\_) {  int parent = Find(item.first);  counts[parent]++;  }  int res = 0;  for (const auto& item : counts) {  res = max(res, item.second);  }  return res;  }  private:  map<int, int> hash\_;  };  class Solution {  public:  int longestConsecutive(vector<int>& nums)  {  set<int> record;  UF uf(nums);  for (auto num : nums) {  record.insert(num);  if (record.find(num + 1) != record.end()) {  uf.Union(num, num + 1);  }  if (record.find(num - 1) != record.end()) {  uf.Union(num, num - 1);  }  }  return uf.GetMaxParentNum();  }  }; |

|  |
| --- |
| 把相邻元素合并到一个集合中，取数量最大的那个集合作为结果。  用哈希表存储每个端点值对应连续区间的长度  若数已在哈希表中：跳过不做处理  若是新数加入：  取出其左右相邻数已有的连续区间长度 left 和 right  计算当前数的区间长度为：cur\_length = left + right + 1  根据 cur\_length 更新最大长度 max\_length 的值  更新区间两端点的长度值  class Solution(object):  def longestConsecutive(self, nums):  hash\_dict = dict()  max\_length = 0  for num in nums:  if num not in hash\_dict:  left = hash\_dict.get(num - 1, 0)  right = hash\_dict.get(num + 1, 0)  cur\_length = 1 + left + right  if cur\_length > max\_length:  max\_length = cur\_length  hash\_dict[num] = cur\_length  hash\_dict[num - left] = cur\_length  hash\_dict[num + right] = cur\_length  return max\_length |

|  |
| --- |
| int longestConsecutive(vector<int>& nums) {  int res = 0;  unordered\_set<int> s(nums.begin(), nums.end());  for (auto e : s) {  if (s.count(e-1) == 0) {  int c = 1;  while (s.count(++e) != 0) ++c;  res = std::max(c, res);  }  }  return res;  } |

**924. 尽量减少恶意软件的传播**

在节点网络中，只有当 graph[i][j] = 1 时，每个节点 i 能够直接连接到另一个节点 j。

一些节点 initial 最初被恶意软件感染。只要两个节点直接连接，且其中至少一个节点受到恶意软件的感染，那么两个节点都将被恶意软件感染。这种恶意软件的传播将继续，直到没有更多的节点可以被这种方式感染。

假设 M(initial) 是在恶意软件停止传播之后，整个网络中感染恶意软件的最终节点数。我们可以从初始列表中删除一个节点。如果移除这一节点将最小化 M(initial)， 则返回该节点。如果有多个节点满足条件，就返回索引最小的节点。

请注意，如果某个节点已从受感染节点的列表 initial 中删除，它以后可能仍然因恶意软件传播而受到感染。

输入：graph = [[1,1,0],[1,1,0],[0,0,1]], initial = [0,1]

输出：0

输入：graph = [[1,0,0],[0,1,0],[0,0,1]], initial = [0,2]

输出：0

输入：graph = [[1,1,1],[1,1,1],[1,1,1]], initial = [1,2]

输出：1

优先去除所在集合只包含它一个初始节点的初始节点，如果有多个这样的节点，取集合较大的。如果集合大小一样，或者该集合包含了多个初始节点，取下标最小的。

两种方法：bfs 、 并查集

首先是很容易想到的广度优先搜索，每次从初始数组里去掉一个后进行搜索，记录最终感染的个数，感染数越小则说明去除该节点效果最好 （同时需要记录去除的节点在所有节点中的索引，如果感染数相同，拿索引较小的那个节点）

|  |
| --- |
| class Solution {  public:  int minMalwareSpread(vector<vector<int>>& graph, vector<int>& initial) {  //暴力遍历initial  int n = graph.size(), m = initial.size();  vector<vector<int>> next\_to(n);  for (int i = 0; i < n; ++i) {  for (int j = 0; j < n; ++j) if (j != i && graph[i][j] == 1) {  next\_to[i].push\_back(j);  }  }  int maxInfect = INT\_MAX;  int index = -1;  for (int i = 0; i < m; ++i) {  queue<int> q;  vector<bool> visited(n, false);  for (int j = 0; j < m; ++j) if (j != i) {  q.push(initial[j]);  visited[initial[j]] = true;  }  int cnt = q.size();  while(!q.empty()) {  int top = q.front();  q.pop();  const vector<int>& temp = next\_to[top];  for (int k = 0; k < temp.size(); ++k) {  if (!visited[temp[k]]) {  visited[temp[k]] = true;  q.push(temp[k]);  ++cnt;  }  }  }  if (cnt < maxInfect || cnt == maxInfect && initial[i] < index) {  maxInfect = cnt;  index = initial[i];  }  }  return index;  }  }; |

并查集，将所有节点分成堆，然后判断堆中包含初始节点的个数，如果等于0或者大于1，则不作处理，如果等于1，删除该堆中初始节点能够减少的感染数等于堆大小（在我的代码中堆的尾部保留了堆中该初始节点的备份，便于直接拿取）。找到最大的堆大小即可（相同大小同样要取索引较小）

|  |
| --- |
| class Solution {  int[] size;  int[] parent;  public int minMalwareSpread(int[][] graph, int[] initial) {  int N = graph.length;  parent = new int[N];  for (int i = 0; i < N; i++) parent[i] = i; // 初始化parent  size = new int[N];  Arrays.fill(size, 1); // 初始化size  Arrays.sort(initial); // 排序保证如果有多个节点满足条件，就返回索引最小的节点。  // 合并  for (int i = 0; i < N; i++) {  for (int j = i + 1; j < N; j++) {  if (graph[i][j] == 1) union(i, j);  }  }  // 计数  Map<Integer, Integer> map = new HashMap<>();  for (int node : initial) {  int f = find(node);  map.put(f, map.getOrDefault(f, 0) + 1);  }  int cur\_node = initial[0];  int cur\_size = Integer.MIN\_VALUE;  for (int node : initial) {  int f = find(node);  if (map.get(f) == 1 && size[f] > cur\_size) { // 只出现一次才能说明移除是有用的  cur\_node = node;  cur\_size = size[f];  }  }  return cur\_node;  }  public int find(int i) {  while (parent[i] != i) { // 路径压缩,仅压缩1层  parent[i] = parent[parent[i]];  i = parent[i];  }  return i;  }  public void union(int i, int j) {  int f1 = find(i);  int f2 = find(j);  if (f1 != f2) {  parent[f2] = f1;  size[f1] += size[f2]; // 合并两个root的size到新的root  }  }  } |

第二种方法的效率相对高一些。 提醒一下，有一个坑：当删除初始节点数组中某两个节点的效果相同时，取索引较小的节点，这个索引指的是节点在总数组中的索引，不是在初始数组中的索引，比如【3,1】，应该取1而不是3