# Leet Code Training Day 18 Union Find

Union Find is another interesting problem in graph. We have a array of of elements, and we have a relationship which can be transmitted, for example if x and y are in one group, and y and z in one group we can say x and z are in same group too.

The implementation is that we have a array of parents for each element, from 0 to n-1, at very beginning, each parement point itself, which means every one is in its own group. When we try to merge two items. We will point **the parent of the parent** of one item to the parent of another item.

parent[parent[i]] = parent[j];

During the process, the parent relation can become a long list, to reduce the process time, we will compress the relationship by pointing the father to its great father if they are different, by doing so we will compress the relationship to 2 layers.

while (parent[parent[i]] != parent[i]) parent[i] = parent[parent[i]]

You can also write the above logic like this:

auto p = parent[self];

while (parent[p] != p) p = parent[p];

parent[self] = p;

Another thing you need to know is that Union Find pattern some times can be replaced by a BFS pattern. Consider this scenario: assume a makes some friends with b and c, b makes some friends with d and e and c make some friends f and g. You can put a in queue, get friends b and c put in queue, then get friends d,e, f,g and put in queue. One process loop will generate a group id.

## 323. Number of Connected Components in an Undirected Graph

Medium

You have a graph of n nodes. You are given an integer n and an array edges where edges[i] = [ai, bi] indicates that there is an edge between ai and bi in the graph.

Return *the number of connected components in the graph*.

**Example 1:**

A picture containing text, clock, clipart

Description automatically generated

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

**Output:** 2

**Example 2:**

A picture containing text, clock, clipart

Description automatically generated

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

**Output:** 1

**Constraints:**

* 1 <= n <= 2000
* 1 <= edges.length <= 5000
* edges[i].length == 2
* 0 <= ai <= bi < n
* ai != bi
* There are no repeated edges.

### Analysis:

Typical Union Find Problem, we merge the nodes with connection and finally we calculate groups. However this problem can also be solved differently, we simply start with any node, BFS to get all the neighbors, thus we process one group first, then move to next group.

/// <summary>

/// Leet Code 323. Number of Connected Components in an Undirected Graph

///

/// Medium

///

/// You have a graph of n nodes. You are given an integer n and an array

/// edges where edges[i] = [ai, bi] indicates that there is an edge

/// between ai and bi in the graph.

///

/// Return the number of connected components in the graph.

///

/// Example 1:

/// Input: n = 5, edges = [[0,1],[1,2],[3,4]]

/// Output: 2

///

/// Example 2:

/// Input: n = 5, edges = [[0,1],[1,2],[2,3],[3,4]]

/// Output: 1

/// Constraints:

/// 1. 1 <= n <= 2000

/// 2. 1 <= edges.length <= 5000

/// 3. edges[i].length == 2

/// 4. 0 <= ai <= bi < n

/// 5. ai != bi

/// 6. There are no repeated edges.

/// </summary>

int LeetCodeGraph::countComponents(int n, vector<pair<int, int>>& edges)

{

vector<vector<int>> neighbors(n);

vector<int> visited(n);

queue<int> process\_queue;

int count = 0;

for (size\_t i = 0; i < edges.size(); i++)

{

neighbors[edges[i].first].push\_back(edges[i].second);

neighbors[edges[i].second].push\_back(edges[i].first);

}

for (int i = 0; i < n; i++)

{

if (visited[i] == 0)

{

process\_queue.push(i);

visited[i] = 1;

while (!process\_queue.empty())

{

int current = process\_queue.front();

process\_queue.pop();

for (int target : neighbors[current])

{

if (visited[target] == 0)

{

process\_queue.push(target);

visited[target] = 1;

}

}

}

count++;

}

}

return count;

}

## 547. Number of Provinces

Medium

There are n cities. Some of them are connected, while some are not. If city a is connected directly with city b, and city b is connected directly with city c, then city a is connected indirectly with city c.

A **province** is a group of directly or indirectly connected cities and no other cities outside of the group.

You are given an n x n matrix isConnected where isConnected[i][j] = 1 if the ith city and the jth city are directly connected, and isConnected[i][j] = 0 otherwise.

Return *the total number of****provinces***.

**Example 1:**

A picture containing text, clock, clipart

Description automatically generated

**Input:** isConnected = [[1,1,0],[1,1,0],[0,0,1]]

**Output:** 2

**Example 2:**

A picture containing diagram

Description automatically generated

**Input:** isConnected = [[1,0,0],[0,1,0],[0,0,1]]

**Output:** 3

**Constraints:**

* 1 <= n <= 200
* n == isConnected.length
* n == isConnected[i].length
* isConnected[i][j] is 1 or 0.
* isConnected[i][i] == 1
* isConnected[i][j] == isConnected[j][i]

### Analysis:

We can use union find to merge the cities. However, this problem can also be resolved in BFS pattern as above. Here we apply union find pattern.

/// <summary>

/// Leet Code 547. Number of Provinces

///

/// Medium

///

/// There are n cities. Some of them are connected, while some are not. If

/// city a is connected directly with city b, and city b is connected

/// directly with city c, then city a is connected indirectly with city c.

///

/// A province is a group of directly or indirectly connected cities and

/// no other cities outside of the group.

///

/// You are given an n x n matrix isConnected where isConnected[i][j] = 1

/// if the ith city and the jth city are directly connected, and

/// isConnected[i][j] = 0 otherwise.

///

/// Return the total number of provinces.

///

/// Example 1:

/// Input: isConnected = [[1,1,0],[1,1,0],[0,0,1]]

/// Output: 2

///

/// Example 2:

/// Input: isConnected = [[1,0,0],[0,1,0],[0,0,1]]

/// Output: 3

///

/// Constraints:

/// 1. 1 <= n <= 200

/// 2. n == isConnected.length

/// 3. n == isConnected[i].length

/// 4. isConnected[i][j] is 1 or 0.

/// 5. isConnected[i][i] == 1

/// 6. isConnected[i][j] == isConnected[j][i]

/// </summary>

int LeetCodeGraph::findCircleNum(vector<vector<int>>& M)

{

vector<int> parent(M.size());

for (size\_t i = 0; i < M.size(); i++)

{

parent[i] = i;

}

for (size\_t i = 0; i < M.size(); i++)

{

for (size\_t j = 0; j < M[i].size(); j++)

{

if (i == j) continue;

if (M[i][j] == 1)

{

// fine the root of both source and target and union them by

// pointing target to the source

int source = i;

int target = j;

while (parent[parent[source]] != parent[source]) parent[source] = parent[parent[source]];

while (parent[target] != parent[parent[target]]) parent[target] = parent[parent[target]];

parent[parent[target]] = parent[source];

}

}

}

int count = 0;

for (size\_t i = 0; i < parent.size(); i++)

{

if (parent[i] == i) count++;

}

return count;

}

## 721. Accounts Merge

Medium

Given a list of accounts where each element accounts[i] is a list of strings, where the first element accounts[i][0] is a name, and the rest of the elements are **emails** representing emails of the account.

Now, we would like to merge these accounts. Two accounts definitely belong to the same person if there is some common email to both accounts. Note that even if two accounts have the same name, they may belong to different people as people could have the same name. A person can have any number of accounts initially, but all of their accounts definitely have the same name.

After merging the accounts, return the accounts in the following format: the first element of each account is the name, and the rest of the elements are emails **in sorted order**. The accounts themselves can be returned in **any order**.

**Example 1:**

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

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

**Explanation:**

The first and second John's are the same person as they have the common email "johnsmith@mail.com".

The third John and Mary are different people as none of their email addresses are used by other accounts.

We could return these lists in any order, for example the answer [['Mary', 'mary@mail.com'], ['John', 'johnnybravo@mail.com'],

['John', 'john00@mail.com', 'john\_newyork@mail.com', 'johnsmith@mail.com']] would still be accepted.

**Example 2:**

**Input:** accounts = [["Gabe","Gabe0@m.co","Gabe3@m.co","Gabe1@m.co"],["Kevin","Kevin3@m.co","Kevin5@m.co","Kevin0@m.co"],["Ethan","Ethan5@m.co","Ethan4@m.co","Ethan0@m.co"],["Hanzo","Hanzo3@m.co","Hanzo1@m.co","Hanzo0@m.co"],["Fern","Fern5@m.co","Fern1@m.co","Fern0@m.co"]]

**Output:** [["Ethan","Ethan0@m.co","Ethan4@m.co","Ethan5@m.co"],["Gabe","Gabe0@m.co","Gabe1@m.co","Gabe3@m.co"],["Hanzo","Hanzo0@m.co","Hanzo1@m.co","Hanzo3@m.co"],["Kevin","Kevin0@m.co","Kevin3@m.co","Kevin5@m.co"],["Fern","Fern0@m.co","Fern1@m.co","Fern5@m.co"]]

**Constraints:**

* 1 <= accounts.length <= 1000
* 2 <= accounts[i].length <= 10
* 1 <= accounts[i][j] <= 30
* accounts[i][0] consists of English letters.
* accounts[i][j] (for j > 0) is a valid email.

### Analysis:

Find the root of each email and merge them in the same account. Path compression is optional here.

/// <summary>

/// Leet code #721. Accounts Merge

///

/// Given a list accounts, each element accounts[i] is a list of strings,

/// where the first element accounts[i][0] is a name, and the rest of the

/// elements are emails representing emails of the account.

///

/// Now, we would like to merge these accounts. Two accounts definitely

/// belong to the same person if there is some email that is common to

/// both accounts. Note that even if two accounts have the same name, they

/// may belong to different people as people could have the same name. A

/// person can have any number of accounts initially, but all of their

/// accounts definitely have the same name.

///

/// After merging the accounts, return the accounts in the following

/// format: the first element of each account is the name, and the rest of

/// the elements are emails in sorted order. The accounts themselves can be

/// returned in any order.

///

/// Example 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:

/// The first and third John's are the same person as they have the common

/// email "johnsmith@mail.com".

/// The second John and Mary are different people as none of their email

/// addresses are used by other accounts.

/// We could return these lists in any order, for example the answer

/// [['Mary', 'mary@mail.com'], ['John', 'johnnybravo@mail.com'],

/// ['John', 'john00@mail.com', 'john\_newyork@mail.com',

/// 'johnsmith@mail.com']] would still be accepted.

/// Note:

///

/// 1.The length of accounts will be in the range [1, 1000].

/// 2.The length of accounts[i] will be in the range [1, 10].

/// 3.The length of accounts[i][j] will be in the range [1, 30].

/// </summary>

vector<vector<string>> LeetCodeGraph::accountsMergeII(vector<vector<string>>& accounts)

{

unordered\_map<string, int> email\_map;

unordered\_map<int, int> parent\_map;

unordered\_map<int, set<string>> merged\_account;

vector<vector<string>> result;

for (size\_t i = 0; i < accounts.size(); i++)

{

string name = accounts[i][0];

int self = i;

parent\_map[i] = i;

for (size\_t j = 1; j < accounts[i].size(); j++)

{

string email = accounts[i][j];

if (email\_map.count(email) == 0)

{

email\_map[email] = i;

}

else

{

// union merge with root

int parent = email\_map[email];

while (parent\_map[parent] != parent) parent = parent\_map[parent];

if (parent != self)

{

parent\_map[self] = parent;

self = parent;

}

}

}

}

for (auto itr : parent\_map)

{

for (size\_t j = 1; j < accounts[itr.first].size(); j++)

{

int parent = itr.second;

while (parent\_map[parent] != parent) parent = parent\_map[parent];

merged\_account[parent].insert(accounts[itr.first][j]);

}

}

for (auto itr : merged\_account)

{

vector<string> account;

account.push\_back(accounts[itr.first][0]);

for (string str : itr.second) account.push\_back(str);

result.push\_back(account);

}

return result;

}

## 737. Sentence Similarity II

Medium

We can represent a sentence as an array of words, for example, the sentence "I am happy with leetcode" can be represented as arr = ["I","am",happy","with","leetcode"].

Given two sentences sentence1 and sentence2 each represented as a string array and given an array of string pairs similarPairs where similarPairs[i] = [xi, yi] indicates that the two words xi and yi are similar.

Return true*if sentence1 and sentence2 are similar, or*false*if they are not similar*.

Two sentences are similar if:

* They have **the same length** (i.e., the same number of words)
* sentence1[i] and sentence2[i] are similar.

Notice that a word is always similar to itself, also notice that the similarity relation is transitive. For example, if the words a and b are similar, and the words b and c are similar, then a and c are **similar**.

**Example 1:**

**Input:** sentence1 = ["great","acting","skills"], sentence2 = ["fine","drama","talent"], similarPairs = [["great","good"],["fine","good"],["drama","acting"],["skills","talent"]]

**Output:** true

**Explanation:** The two sentences have the same length and each word i of sentence1 is also similar to the corresponding word in sentence2.

**Example 2:**

**Input:** sentence1 = ["I","love","leetcode"], sentence2 = ["I","love","onepiece"], similarPairs = [["manga","onepiece"],["platform","anime"],["leetcode","platform"],["anime","manga"]]

**Output:** true

**Explanation:** "leetcode" --> "platform" --> "anime" --> "manga" --> "onepiece".

Since "leetcode is similar to "onepiece" and the first two words are the same, the two sentences are similar.

**Example 3:**

**Input:** sentence1 = ["I","love","leetcode"], sentence2 = ["I","love","onepiece"], similarPairs = [["manga","hunterXhunter"],["platform","anime"],["leetcode","platform"],["anime","manga"]]

**Output:** false

**Explanation:** "leetcode" is not similar to "onepiece".

**Constraints:**

* 1 <= sentence1.length, sentence2.length <= 1000
* 1 <= sentence1[i].length, sentence2[i].length <= 20
* sentence1[i] and sentence2[i] consist of lower-case and upper-case English letters.
* 0 <= similarPairs.length <= 2000
* similarPairs[i].length == 2
* 1 <= xi.length, yi.length <= 20
* xi and yi consist of English letters.

### Analysis:

All the similar words should have be merged as same root.

/// <summary>

/// Leet code #737. Sentence Similarity II

///

/// Given two sentences words1, words2 (each represented as an array of

/// strings), and a list of similar word pairs pairs, determine if two

/// sentences are similar.

///

/// For example, words1 = ["great", "acting", "skills"] and words2 =

/// ["fine", "drama", "talent"] are similar, if the similar word pairs

/// are pairs = [["great", "good"], ["fine", "good"], ["acting","drama"],

/// ["skills","talent"]].

///

/// Note that the similarity relation is transitive. For example, if

/// "great" and "good" are similar, and "fine" and "good" are similar,

/// then "great" and "fine" are similar.

///

/// Similarity is also symmetric. For example, "great" and "fine" being

/// similar is the same as "fine" and "great" being similar.

///

/// Also, a word is always similar with itself. For example, the sentences

/// words1 = ["great"], words2 = ["great"], pairs = [] are similar, even

/// though there are no specified similar word pairs.

///

/// Finally, sentences can only be similar if they have the same number of

/// words. So a sentence like words1 = ["great"] can never be similar to

/// words2 = ["doubleplus","good"].

///

/// Note:

///

/// The length of words1 and words2 will not exceed 1000.

/// The length of pairs will not exceed 2000.

/// The length of each pairs[i] will be 2.

/// The length of each words[i] and pairs[i][j] will be in the range [1, 20].

/// </summary>

bool LeetCodeGraph::areSentencesSimilarTwo(vector<string>& words1, vector<string>& words2,

vector<pair<string, string>> pairs)

{

if (words1.size() != words2.size()) return false;

unordered\_map<string, string> similar\_words;

for (auto itr : pairs)

{

string first = itr.first;

// insert the first word if not exist

if (similar\_words.count(first) == 0) similar\_words[first] = first;

// find the root of the first word

while (similar\_words[first] != first) first = similar\_words[first];

string second = itr.second;

// insert the second word if not exist

if (similar\_words.count(second) == 0) similar\_words[second] = second;

// find the root of the second word

while (similar\_words[second] != second) second = similar\_words[second];

// point the second word to the first

similar\_words[second] = first;

}

for (size\_t i = 0; i < words1.size(); i++)

{

string first = words1[i];

// find the root of first word

while (similar\_words[first] != first) first = similar\_words[first];

string second = words2[i];

// find the root of second word

while (similar\_words[second] != second) second = similar\_words[second];

if (first != second) return false;

}

return true;

}

## 1061. Lexicographically Smallest Equivalent String

Medium

You are given two strings of the same length s1 and s2 and a string baseStr.

We say s1[i] and s2[i] are equivalent characters.

* For example, if s1 = "abc" and s2 = "cde", then we have 'a' == 'c', 'b' == 'd', and 'c' == 'e'.

Equivalent characters follow the usual rules of any equivalence relation:

* **Reflexivity:** 'a' == 'a'.
* **Symmetry:** 'a' == 'b' implies 'b' == 'a'.
* **Transitivity:** 'a' == 'b' and 'b' == 'c' implies 'a' == 'c'.

For example, given the equivalency information from s1 = "abc" and s2 = "cde", "acd" and "aab" are equivalent strings of baseStr = "eed", and "aab" is the lexicographically smallest equivalent string of baseStr.

Return *the lexicographically smallest equivalent string of*baseStr*by using the equivalency information from*s1*and*s2.

**Example 1:**

**Input:** s1 = "parker", s2 = "morris", baseStr = "parser"

**Output:** "makkek"

**Explanation:** Based on the equivalency information in s1 and s2, we can group their characters as [m,p], [a,o], [k,r,s], [e,i].

The characters in each group are equivalent and sorted in lexicographical order.

So the answer is "makkek".

**Example 2:**

**Input:** s1 = "hello", s2 = "world", baseStr = "hold"

**Output:** "hdld"

**Explanation:** Based on the equivalency information in s1 and s2, we can group their characters as [h,w], [d,e,o], [l,r].

So only the second letter 'o' in baseStr is changed to 'd', the answer is "hdld".

**Example 3:**

**Input:** s1 = "leetcode", s2 = "programs", baseStr = "sourcecode"

**Output:** "aauaaaaada"

**Explanation:** We group the equivalent characters in s1 and s2 as [a,o,e,r,s,c], [l,p], [g,t] and [d,m], thus all letters in baseStr except 'u' and 'd' are transformed to 'a', the answer is "aauaaaaada".

**Constraints:**

* 1 <= s1.length, s2.length, baseStr <= 1000
* s1.length == s2.length
* s1, s2, and baseStr consist of lowercase English letters.

### Analysis:

Union-find the similar character and mapping to the smallest one.

/// <summary>

/// Leet code 1061. Lexicographically Smallest Equivalent String

///

/// Given strings A and B of the same length, we say A[i] and B[i] are

/// equivalent characters. For example, if A = "abc" and B = "cde", then

/// we have 'a' == 'c', 'b' == 'd', 'c' == 'e'.

///

/// Equivalent characters follow the usual rules of any equivalence relation:

///

/// Reflexivity: 'a' == 'a'

/// Symmetry: 'a' == 'b' implies 'b' == 'a'

/// Transitivity: 'a' == 'b' and 'b' == 'c' implies 'a' == 'c'

/// For example, given the equivalency information from A and B above,

/// S = "eed", "acd", and "aab" are equivalent strings, and "aab" is the

/// lexicographically smallest equivalent string of S.

///

/// Return the lexicographically smallest equivalent string of S by using

/// the equivalency information from A and B.

///

///

/// Example 1:

///

/// Input: A = "parker", B = "morris", S = "parser"

/// Output: "makkek"

/// Explanation: Based on the equivalency information in A and B, we can

/// group their characters as [m,p], [a,o], [k,r,s], [e,i]. The characters

/// in each group are equivalent and sorted in lexicographical order. So the

/// answer is "makkek".

///

/// Example 2:

///

/// Input: A = "hello", B = "world", S = "hold"

/// Output: "hdld"

/// Explanation: Based on the equivalency information in A and B, we can

/// group their characters as [h,w], [d,e,o], [l,r]. So only the second

/// letter 'o' in S is changed to 'd', the answer is "hdld".

///

/// Example 3:

///

/// Input: A = "leetcode", B = "programs", S = "sourcecode"

/// Output: "aauaaaaada"

/// Explanation: We group the equivalent characters in A and B as

/// [a,o,e,r,s,c], [l,p], [g,t] and [d,m], thus all letters in S except 'u'

/// and 'd' are transformed to 'a', the answer is "aauaaaaada".

///

/// Note:

///

/// 1. String A, B and S consist of only lowercase English letters from

/// 'a' - 'z'.

/// 2. The lengths of string A, B and S are between 1 and 1000.

/// 3. String A and B are of the same length.

/// </summary>

string LeetCodeGraph::smallestEquivalentString(string A, string B, string S)

{

vector<int> letters(26);

for (size\_t i = 0; i < 26; i++) letters[i] = i;

for (size\_t i = 0; i < A.size(); i++)

{

int a = A[i] - 'a';

int b = B[i] - 'a';

while (letters[a] != a)

{

letters[a] = letters[letters[a]];

a = letters[a];

}

while (letters[b] != b)

{

letters[b] = letters[letters[b]];

b = letters[b];

}

if (a < b) letters[b] = a;

else letters[a] = b;

}

string result = S;

for (size\_t i = 0; i < result.size(); i++)

{

int a = result[i] - 'a';

while (letters[a] != a) a = letters[a];

result[i] = a + 'a';

}

return result;

}

## 1101. The Earliest Moment When Everyone Become Friends

Medium

There are n people in a social group labeled from 0 to n - 1. You are given an array logs where logs[i] = [timestampi, xi, yi] indicates that xi and yi will be friends at the time timestampi.

Friendship is **symmetric**. That means if a is friends with b, then b is friends with a. Also, person a is acquainted with a person b if a is friends with b, or a is a friend of someone acquainted with b.

Return *the earliest time for which every person became acquainted with every other person*. If there is no such earliest time, return -1.

**Example 1:**

**Input:** logs = [[20190101,0,1],[20190104,3,4],[20190107,2,3],[20190211,1,5],[20190224,2,4],[20190301,0,3],[20190312,1,2],[20190322,4,5]], n = 6

**Output:** 20190301

**Explanation:**

The first event occurs at timestamp = 20190101 and after 0 and 1 become friends we have the following friendship groups [0,1], [2], [3], [4], [5].

The second event occurs at timestamp = 20190104 and after 3 and 4 become friends we have the following friendship groups [0,1], [2], [3,4], [5].

The third event occurs at timestamp = 20190107 and after 2 and 3 become friends we have the following friendship groups [0,1], [2,3,4], [5].

The fourth event occurs at timestamp = 20190211 and after 1 and 5 become friends we have the following friendship groups [0,1,5], [2,3,4].

The fifth event occurs at timestamp = 20190224 and as 2 and 4 are already friends anything happens.

The sixth event occurs at timestamp = 20190301 and after 0 and 3 become friends we have that all become friends.

**Example 2:**

**Input:** logs = [[0,2,0],[1,0,1],[3,0,3],[4,1,2],[7,3,1]], n = 4

**Output:** 3

**Constraints:**

* 2 <= n <= 100
* 1 <= logs.length <= 104
* logs[i].length == 3
* 0 <= timestampi <= 109
* 0 <= xi, yi <= n - 1
* xi != yi
* All the values timestampi are **unique**.
* All the pairs (xi, yi) occur at most one time in the input.

### Analysis:

Union-find the similar character and mapping to the smallest one.

/// <summary>

/// Leet code #1101. The Earliest Moment When Everyone Become Friends

///

/// In a social group, there are N people, with unique integer ids from 0 to

/// N-1.

/// We have a list of logs, where each logs[i] = [timestamp, id\_A, id\_B]

/// contains a non-negative integer timestamp, and the ids of two different

/// people.

/// Each log represents the time in which two different people became friends.

/// Friendship is symmetric: if A is friends with B, then B is friends with A.

/// Let's say that person A is acquainted with person B if A is friends with

/// B, or A is a friend of someone acquainted with B.

/// Return the earliest time for which every person became acquainted with

/// every other person. Return -1 if there is no such earliest time.

///

/// Example 1:

/// Input: logs = [[20190101,0,1],[20190104,3,4],[20190107,2,3],[20190211,1,5],

/// [20190224,2,4],[20190301,0,3],[20190312,1,2],[20190322,4,5]], N = 6

/// Output: 20190301

/// Explanation:

/// The first event occurs at timestamp = 20190101 and after 0 and 1 become

/// friends we have the following friendship groups [0,1], [2], [3], [4], [5].

/// The second event occurs at timestamp = 20190104 and after 3 and 4 become

/// friends we have the following friendship groups [0,1], [2], [3,4], [5].

/// The third event occurs at timestamp = 20190107 and after 2 and 3 become

/// friends we have the following friendship groups [0,1], [2,3,4], [5].

/// The fourth event occurs at timestamp = 20190211 and after 1 and 5 become

/// friends we have the following friendship groups [0,1,5], [2,3,4].

/// The fifth event occurs at timestamp = 20190224 and as 2 and 4 are already

/// friend anything happens.

/// The sixth event occurs at timestamp = 20190301 and after 0 and 3 become

/// friends we have that all become friends.

///

/// Note:

/// 1. 1 <= N <= 100

/// 2. 1 <= logs.length <= 10^4

/// 3. 0 <= logs[i][0] <= 10^9

/// 4. 0 <= logs[i][1], logs[i][2] <= N - 1

/// 5. It's guaranteed that all timestamps in logs[i][0] are different.

/// 6. Logs are not necessarily ordered by some criteria.

/// 7. logs[i][1] != logs[i][2]

/// </summary>

int LeetCodeGraph::earliestAcq(vector<vector<int>>& logs, int N)

{

vector<int> count(N, 1);

vector<int> parent(N, -1);

sort(logs.begin(), logs.end());

for (size\_t i = 0; i < logs.size(); i++)

{

int first = logs[i][1];

int second = logs[i][2];

if (parent[first] == -1) parent[first] = first;

if (parent[second] == -1) parent[second] = second;

while (parent[first] != first)

{

parent[first] = parent[parent[first]];

first = parent[first];

}

while (parent[second] != second)

{

parent[second] = parent[parent[second]];

second = parent[second];

}

if (first != second)

{

parent[second] = first;

count[first] += count[second];

if (count[first] == N) return logs[i][0];

}

}

return -1;

}

## 1202. Smallest String With Swaps

Medium

You are given a string s, and an array of pairs of indices in the string pairs where pairs[i] = [a, b] indicates 2 indices(0-indexed) of the string.

You can swap the characters at any pair of indices in the given pairs **any number of times**.

Return the lexicographically smallest string that s can be changed to after using the swaps.

**Example 1:**

**Input:** s = "dcab", pairs = [[0,3],[1,2]]

**Output:** "bacd"

**Explaination:**

Swap s[0] and s[3], s = "bcad"

Swap s[1] and s[2], s = "bacd"

**Example 2:**

**Input:** s = "dcab", pairs = [[0,3],[1,2],[0,2]]

**Output:** "abcd"

**Explaination:**

Swap s[0] and s[3], s = "bcad"

Swap s[0] and s[2], s = "acbd"

Swap s[1] and s[2], s = "abcd"

**Example 3:**

**Input:** s = "cba", pairs = [[0,1],[1,2]]

**Output:** "abc"

**Explaination:**

Swap s[0] and s[1], s = "bca"

Swap s[1] and s[2], s = "bac"

Swap s[0] and s[1], s = "abc"

**Constraints:**

* 1 <= s.length <= 10^5
* 0 <= pairs.length <= 10^5
* 0 <= pairs[i][0], pairs[i][1] < s.length
* s only contains lower case English letters.

### Analysis:

Group the indexes with pairs, and add all characters there then pick the characters from smallest heap.

/// <summary>

/// Leet code #1202. Smallest String With Swaps

///

/// You are given a string s, and an array of pairs of indices in the string

/// pairs where pairs[i] = [a, b] indicates 2 indices(0-indexed) of the string.

/// You can swap the characters at any pair of indices in the given pairs any

/// number of times.

/// Return the lexicographically smallest string that s can be changed to

/// after using the swaps.

///

/// Example 1:

/// Input: s = "dcab", pairs = [[0,3],[1,2]]

/// Output: "bacd"

/// Explaination:

/// Swap s[0] and s[3], s = "bcad"

/// Swap s[1] and s[2], s = "bacd"

///

/// Example 2:

/// Input: s = "dcab", pairs = [[0,3],[1,2],[0,2]]

/// Output: "abcd"

/// Explaination:

/// Swap s[0] and s[3], s = "bcad"

/// Swap s[0] and s[2], s = "acbd"

/// Swap s[1] and s[2], s = "abcd"

///

/// Example 3:

/// Input: s = "cba", pairs = [[0,1],[1,2]]

/// Output: "abc"

/// Explaination:

/// Swap s[0] and s[1], s = "bca"

/// Swap s[1] and s[2], s = "bac"

/// Swap s[0] and s[1], s = "abc"

///

/// Constraints:

/// 1. 1 <= s.length <= 10^5

/// 2. 0 <= pairs.length <= 10^5

/// 3. 0 <= pairs[i][0], pairs[i][1] < s.length

/// 4. s only contains lower case English letters.

/// </summary>

string LeetCodeGraph::smallestStringWithSwaps(string s, vector<vector<int>>& pairs)

{

vector<int> parent(s.size());

unordered\_map<int, priority\_queue<char>> char\_set;

for (size\_t i = 0; i < parent.size(); i++) parent[i] = i;

for (size\_t i = 0; i < pairs.size(); i++)

{

int a = pairs[i][0];

int b = pairs[i][1];

while (parent[a] != a)

{

parent[a] = parent[parent[a]];

a = parent[a];

}

while (parent[b] != b)

{

parent[b] = parent[parent[b]];

b = parent[b];

}

parent[a] = b;

}

for (size\_t i = 0; i < s.size(); i++)

{

int a = i;

while (parent[a] != a)

{

parent[a] = parent[parent[a]];

a = parent[a];

}

char\_set[a].push('z' - s[i]);

}

string result;

for (size\_t i = 0; i < s.size(); i++)

{

int a = i;

while (parent[a] != a) a = parent[a];

if (!char\_set.empty())

{

result.push\_back('z' - char\_set[a].top());

char\_set[a].pop();

}

}

return result;

}

## 1319. Number of Operations to Make Network Connected

Medium

There are n computers numbered from 0 to n - 1 connected by ethernet cables connections forming a network where connections[i] = [ai, bi] represents a connection between computers ai and bi. Any computer can reach any other computer directly or indirectly through the network.

You are given an initial computer network connections. You can extract certain cables between two directly connected computers, and place them between any pair of disconnected computers to make them directly connected.

Return *the minimum number of times you need to do this in order to make all the computers connected*. If it is not possible, return -1.

**Example 1:**

Icon

Description automatically generated with low confidence

**Input:** n = 4, connections = [[0,1],[0,2],[1,2]]

**Output:** 1

**Explanation:** Remove cable between computer 1 and 2 and place between computers 1 and 3.

**Example 2:**

Diagram

Description automatically generated with low confidence

**Input:** n = 6, connections = [[0,1],[0,2],[0,3],[1,2],[1,3]]

**Output:** 2

**Example 3:**

**Input:** n = 6, connections = [[0,1],[0,2],[0,3],[1,2]]

**Output:** -1

**Explanation:** There are not enough cables.

**Constraints:**

* 1 <= n <= 105
* 1 <= connections.length <= min(n \* (n - 1) / 2, 105)
* connections[i].length == 2
* 0 <= ai, bi < n
* ai != bi
* There are no repeated connections.
* No two computers are connected by more than one cable.

### Analysis:

Union find with connections then connect groups.

/// <summary>

/// Leet code #1319. Number of Operations to Make Network Connected

///

/// Medium

///

/// There are n computers numbered from 0 to n-1 connected by ethernet

/// cables connections forming a network where connections[i] = [a, b]

/// represents a connection between computers a and b. Any computer can

/// reach any other computer directly or indirectly through the network.

///

/// Given an initial computer network connections. You can extract

/// certain cables between two directly connected computers, and place

/// them between any pair of disconnected computers to make them directly

/// connected. Return the minimum number of times you need to do this in

/// order to make all the computers connected. If it's not possible,

/// return -1.

///

/// Example 1:

/// Input: n = 4, connections = [[0,1],[0,2],[1,2]]

/// Output: 1

/// Explanation: Remove cable between computer 1 and 2 and place between

/// computers 1 and 3.

///

/// Example 2:

/// Input: n = 6, connections = [[0,1],[0,2],[0,3],[1,2],[1,3]]

/// Output: 2

///

/// Example 3:

/// Input: n = 6, connections = [[0,1],[0,2],[0,3],[1,2]]

/// Output: -1

/// Explanation: There are not enough cables.

///

/// Example 4:

/// Input: n = 5, connections = [[0,1],[0,2],[3,4],[2,3]]

/// Output: 0

///

/// Constraints:

/// 1. 1 <= n <= 10^5

/// 2. 1 <= connections.length <= min(n\*(n-1)/2, 10^5)

/// 3. connections[i].length == 2

/// 4. 0 <= connections[i][0], connections[i][1] < n

/// 5. connections[i][0] != connections[i][1]

/// 6. There are no repeated connections.

/// 7. No two computers are connected by more than one cable.

/// </summary>

int LeetCodeGraph::makeConnected(int n, vector<vector<int>>& connections)

{

vector<int> p(n);

for (int i = 0; i < n; i++)

{

p[i] = i;

}

int cables = 0;

for (size\_t i = 0; i < connections.size(); i++)

{

int a = connections[i][0];

int b = connections[i][1];

while (p[a] != a)

{

p[a] = p[p[a]];

a = p[a];

}

while (p[b] != b)

{

p[b] = p[p[b]];

b = p[b];

}

if (a == b) cables++;

else p[a] = b;

}

int result = -1;

for (int i = 0; i < n; i++)

{

if (p[i] == i) result++;

}

if (cables < result) return -1;

else return result;

}

## 1722. Minimize Hamming Distance After Swap Operations

Medium

You are given two integer arrays, source and target, both of length n. You are also given an array allowedSwaps where each allowedSwaps[i] = [ai, bi] indicates that you are allowed to swap the elements at index ai and index bi **(0-indexed)** of array source. Note that you can swap elements at a specific pair of indices **multiple** times and in **any** order.

The **Hamming distance** of two arrays of the same length, source and target, is the number of positions where the elements are different. Formally, it is the number of indices i for 0 <= i <= n-1 where source[i] != target[i] **(0-indexed)**.

Return *the****minimum Hamming distance****of*source*and*target*after performing****any****amount of swap operations on array*source*.*

**Example 1:**

**Input:** source = [1,2,3,4], target = [2,1,4,5], allowedSwaps = [[0,1],[2,3]]

**Output:** 1

**Explanation:** source can be transformed the following way:

- Swap indices 0 and 1: source = [2,1,3,4]

- Swap indices 2 and 3: source = [2,1,4,3]

The Hamming distance of source and target is 1 as they differ in 1 position: index 3.

**Example 2:**

**Input:** source = [1,2,3,4], target = [1,3,2,4], allowedSwaps = []

**Output:** 2

**Explanation:** There are no allowed swaps.

The Hamming distance of source and target is 2 as they differ in 2 positions: index 1 and index 2.

**Example 3:**

**Input:** source = [5,1,2,4,3], target = [1,5,4,2,3], allowedSwaps = [[0,4],[4,2],[1,3],[1,4]]

**Output:** 0

**Constraints:**

* n == source.length == target.length
* 1 <= n <= 105
* 1 <= source[i], target[i] <= 105
* 0 <= allowedSwaps.length <= 105
* allowedSwaps[i].length == 2
* 0 <= ai, bi <= n - 1
* ai != bi

### Analysis:

Union find all the positions can be swappable and put it in a common hash map.

/// <summary>

/// Leet code 1722. Minimize Hamming Distance After Swap Operations

///

/// Medium

///

/// You are given two integer arrays, source and target, both of length n.

/// You are also given an array allowedSwaps where each allowedSwaps[i] =

/// [ai, bi] indicates that you are allowed to swap the elements at index

/// ai and index bi (0-indexed) of array source. Note that you can swap

/// elements at a specific pair of indices multiple times and in any order.

///

/// The Hamming distance of two arrays of the same length, source and

/// target, is the number of positions where the elements are different.

/// Formally, it is the number of indices i for 0 <= i <= n-1 where

/// source[i] != target[i] (0-indexed).

///

/// Return the minimum Hamming distance of source and target after

/// performing any amount of swap operations on array source.

///

/// Example 1:

/// Input: source = [1,2,3,4], target = [2,1,4,5],

/// allowedSwaps = [[0,1],[2,3]]

/// Output: 1

/// Explanation: source can be transformed the following way:

/// - Swap indices 0 and 1: source = [2,1,3,4]

/// - Swap indices 2 and 3: source = [2,1,4,3]

/// The Hamming distance of source and target is 1 as they differ in 1

/// position: index 3.

///

/// Example 2:

/// Input: source = [1,2,3,4], target = [1,3,2,4], allowedSwaps = []

/// Output: 2

/// Explanation: There are no allowed swaps.

/// The Hamming distance of source and target is 2 as they differ in 2

/// positions: index 1 and index 2.

///

/// Example 3:

/// Input: source = [5,1,2,4,3], target = [1,5,4,2,3],

/// allowedSwaps = [[0,4],[4,2],[1,3],[1,4]]

/// Output: 0

/// Constraints:

/// 1. n == source.length == target.length

/// 2. 1 <= n <= 10^5

/// 3. 1 <= source[i], target[i] <= 10^5

/// 4. 0 <= allowedSwaps.length <= 10^5

/// 5. allowedSwaps[i].length == 2

/// 6. 0 <= ai, bi <= n - 1

/// 7. ai != bi

/// </summary>

int LeetCodeGraph::minimumHammingDistance(vector<int>& source,

vector<int>& target, vector<vector<int>>& allowedSwaps)

{

vector<int> parent(source.size());

for (size\_t i = 0; i < parent.size(); i++) parent[i] = i;

for (size\_t i = 0; i < allowedSwaps.size(); i++)

{

int a = allowedSwaps[i][0];

int b = allowedSwaps[i][1];

while (parent[parent[a]] != parent[a]) parent[a] = parent[parent[a]];

while (parent[parent[b]] != parent[b]) parent[b] = parent[parent[b]];

parent[parent[a]] = parent[parent[b]];

}

unordered\_map<int, unordered\_map<int, int>> exch\_grp;

for (size\_t i = 0; i < source.size(); i++)

{

while (parent[parent[i]] != parent[i]) parent[i] = parent[parent[i]];

exch\_grp[parent[i]][source[i]]++;

}

int result = 0;

for (size\_t i = 0; i < target.size(); i++)

{

if (exch\_grp[parent[i]].count(target[i]) > 0)

{

exch\_grp[parent[i]][target[i]]--;

if (exch\_grp[parent[i]][target[i]] == 0)

{

exch\_grp[parent[i]].erase(target[i]);

}

}

else

{

result++;

}

}

return result;

}

## 886. Possible Bipartition

Medium

Given a set of N people (numbered 1, 2, ..., N), we would like to split everyone into two groups of **any** size.

Each person may dislike some other people, and they should not go into the same group.

Formally, if dislikes[i] = [a, b], it means it is not allowed to put the people numbered a and b into the same group.

Return true if and only if it is possible to split everyone into two groups in this way.

**Example 1:**

**Input:** N = 4, dislikes = [[1,2],[1,3],[2,4]]

**Output:** true

**Explanation**: group1 [1,4], group2 [2,3]

**Example 2:**

**Input:** N = 3, dislikes = [[1,2],[1,3],[2,3]]

**Output:** false

**Example 3:**

**Input:** N = 5, dislikes = [[1,2],[2,3],[3,4],[4,5],[1,5]]

**Output:** false

**Note:**

1. 1 <= N <= 2000
2. 0 <= dislikes.length <= 10000
3. 1 <= dislikes[i][j] <= N
4. dislikes[i][0] < dislikes[i][1]
5. There does not exist i != j for which dislikes[i] == dislikes[j].

### Analysis

This can be a union find, but consider there are only two groups, we can start from one person give itself and the dislike one as different ids, and then use BFS to traverse all its dislike people and then traverse the people disliked by the dislike people.

If the whole group is split, we can traverse the remaining ones.

/// <summary>

/// Leet code #886. Possible Bipartition

///

/// Given a set of N people (numbered 1, 2, ..., N), we would like to split

/// everyone into two groups of any size.

///

/// Each person may dislike some other people, and they should not go into the

/// same group.

///

/// Formally, if dislikes[i] = [a, b], it means it is not allowed to put the

/// people numbered a and b into the same group.

///

/// Return true if and only if it is possible to split everyone into two groups

/// in this way.

///

/// Example 1:

/// Input: N = 4, dislikes = [[1,2],[1,3],[2,4]]

/// Output: true

/// Explanation: group1 [1,4], group2 [2,3]

///

/// Example 2:

/// Input: N = 3, dislikes = [[1,2],[1,3],[2,3]]

/// Output: false

///

/// Example 3:

/// Input: N = 5, dislikes = [[1,2],[2,3],[3,4],[4,5],[1,5]]

/// Output: false

///

/// Note:

///

/// 1 <= N <= 2000

/// 0 <= dislikes.length <= 10000

/// 1 <= dislikes[i][j] <= N

/// dislikes[i][0] < dislikes[i][1]

/// There does not exist i != j for which dislikes[i] == dislikes[j].

/// </summary>

bool LeetCode::possibleBipartition(int N, vector<vector<int>>& dislikes)

{

vector<int> group(N + 1, -1);

vector<vector<int>> dislike\_list(N + 1);

for (size\_t i = 0; i < dislikes.size(); i++)

{

int a = dislikes[i][0];

int b = dislikes[i][1];

dislike\_list[a].push\_back(b);

dislike\_list[b].push\_back(a);

}

for (int i = 1; i <= N; i++)

{

if (group[i] != -1) continue;

queue<int> search;

group[i] = 0;

search.push(i);

while (!search.empty())

{

int person = search.front();

search.pop();

for (auto dislike : dislike\_list[person])

{

if (group[dislike] == group[person]) return false;

else if (group[dislike] == -1)

{

group[dislike] = 1 - group[person];

search.push(dislike);

}

}

}

}

return true;

}

## 990. Satisfiability of Equality Equations

Medium

You are given an array of strings equations that represent relationships between variables where each string equations[i] is of length 4 and takes one of two different forms: "xi==yi" or "xi!=yi".Here, xi and yi are lowercase letters (not necessarily different) that represent one-letter variable names.

Return true*if it is possible to assign integers to variable names so as to satisfy all the given equations, or*false*otherwise*.

**Example 1:**

**Input:** equations = ["a==b","b!=a"]

**Output:** false

**Explanation:** If we assign say, a = 1 and b = 1, then the first equation is satisfied, but not the second.

There is no way to assign the variables to satisfy both equations.

**Example 2:**

**Input:** equations = ["b==a","a==b"]

**Output:** true

**Explanation:** We could assign a = 1 and b = 1 to satisfy both equations.

**Constraints:**

* 1 <= equations.length <= 500
* equations[i].length == 4
* equations[i][0] is a lowercase letter.
* equations[i][1] is either '=' or '!'.
* equations[i][2] is '='.
* equations[i][3] is a lowercase letter.

### Analysis

Group the equal items in same group, then check the not equal items happened to be in same group or not.

/// <summary>

/// Leet code #990. Satisfiability of Equality Equations

///

/// Given an array equations of strings that represent relationships

/// between variables, each string equations[i] has length 4 and takes

/// one of two different forms: "a==b" or "a!=b". Here, a and b are

/// lowercase letters (not necessarily different) that represent

/// one-letter variable names.

///

/// Return true if and only if it is possible to assign integers to

/// variable names so as to satisfy all the given equations.

///

/// Example 1:

///

/// Input: ["a==b","b!=a"]

/// Output: false

/// Explanation: If we assign say, a = 1 and b = 1, then the first

/// equation is satisfied, but not the second. There is no way to

/// assign the variables to satisfy both equations.

///

/// Example 2:

///

/// Input: ["b==a","a==b"]

/// Output: true

/// Explanation: We could assign a = 1 and b = 1 to satisfy both equations.

///

/// Example 3:

///

/// Input: ["a==b","b==c","a==c"]

/// Output: true

///

/// Example 4:

///

/// Input: ["a==b","b!=c","c==a"]

/// Output: false

///

/// Example 5:

///

/// Input: ["c==c","b==d","x!=z"]

/// Output: true

///

/// Note:

/// 1. 1 <= equations.length <= 500

/// 2. equations[i].length == 4

/// 3. equations[i][0] and equations[i][3] are lowercase letters

/// 4. equations[i][1] is either '=' or '!'

/// 5. equations[i][2] is '='

/// </summary>

bool LeetCodeGraph::equationsPossible(vector<string>& equations)

{

unordered\_map<char, char> same;

vector<pair<char, char>> diff;

for (size\_t i = 0; i < equations.size(); i++)

{

char a = equations[i][0];

char b = equations[i][3];

if (equations[i][1] == '!')

{

diff.push\_back(make\_pair(a, b));

}

else

{

if (same.count(a) == 0) same[a] = a;

if (same.count(b) == 0) same[b] = b;

while (same[a] != a) a = same[a];

while (same[b] != b) b = same[b];

if (a != b) same[a] = b;

}

}

for (size\_t i = 0; i < diff.size(); i++)

{

char a = diff[i].first;

char b = diff[i].second;

if (same.count(a) == 0) same[a] = a;

if (same.count(b) == 0) same[b] = b;

while (same[a] != a) a = same[a];

while (same[b] != b) b = same[b];

if (a == b) return false;

}

return true;

}

## 947. Most Stones Removed with Same Row or Column

Medium

On a 2D plane, we place n stones at some integer coordinate points. Each coordinate point may have at most one stone.

A stone can be removed if it shares either **the same row or the same column** as another stone that has not been removed.

Given an array stones of length n where stones[i] = [xi, yi] represents the location of the ith stone, return *the largest possible number of stones that can be removed*.

**Example 1:**

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

**Output:** 5

**Explanation:** One way to remove 5 stones is as follows:

1. Remove stone [2,2] because it shares the same row as [2,1].

2. Remove stone [2,1] because it shares the same column as [0,1].

3. Remove stone [1,2] because it shares the same row as [1,0].

4. Remove stone [1,0] because it shares the same column as [0,0].

5. Remove stone [0,1] because it shares the same row as [0,0].

Stone [0,0] cannot be removed since it does not share a row/column with another stone still on the plane.

**Example 2:**

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

**Output:** 3

**Explanation:** One way to make 3 moves is as follows:

1. Remove stone [2,2] because it shares the same row as [2,0].

2. Remove stone [2,0] because it shares the same column as [0,0].

3. Remove stone [0,2] because it shares the same row as [0,0].

Stones [0,0] and [1,1] cannot be removed since they do not share a row/column with another stone still on the plane.

**Example 3:**

**Input:** stones = [[0,0]]

**Output:** 0

**Explanation:** [0,0] is the only stone on the plane, so you cannot remove it.

**Constraints:**

* 1 <= stones.length <= 1000
* 0 <= xi, yi <= 104
* No two stones are at the same coordinate point.

### Analysis

Consider this problem in the way, if some stones have same row or column they are in same group. Use union find to group them and check how many groups are there.

/// <summary>

/// Leet code #947. Most Stones Removed with Same Row or Column

///

/// On a 2D plane, we place stones at some integer coordinate points.

/// Each coordinate point may have at most one stone.

///

/// Now, a move consists of removing a stone that shares a column or

/// row with another stone on the grid.

/// What is the largest possible number of moves we can make?

///

///

/// Example 1:

/// Input: stones = [[0,0],[0,1],[1,0],[1,2],[2,1],[2,2]]

/// Output: 5

///

/// Example 2:

/// Input: stones = [[0,0],[0,2],[1,1],[2,0],[2,2]]

/// Output: 3

///

/// Example 3:

///

/// Input: stones = [[0,0]]

/// Output: 0

///

/// Note:

/// 1. 1 <= stones.length <= 1000

/// 2. 0 <= stones[i][j] < 10000

/// </summary>

int LeetCodeGraph::removeStones(vector<vector<int>>& stones)

{

unordered\_map<int, int> row\_map;

unordered\_map<int, int> col\_map;

unordered\_map<int, int> union\_map;

for (size\_t i = 0; i < stones.size(); i++)

{

int index = i;

int row = stones[i][0];

int col = stones[i][1];

if (row\_map.count(row) == 0) row\_map[row] = index;

if (col\_map.count(col) == 0) col\_map[col] = index;

int source = row\_map[row];

int target = col\_map[col];

if (union\_map.count(source) == 0) union\_map[source] = source;

if (union\_map.count(target) == 0) union\_map[target] = target;

while (source != union\_map[source]) source = union\_map[source];

while (union\_map.count(target) > 0 && target != union\_map[target]) target = union\_map[target];

union\_map[source] = target;

}

int result = 0;

for (auto r : union\_map)

{

if (r.first == r.second) result++;

}

return stones.size() - result;

}

## 1135. Connecting Cities With Minimum Cost

Medium

There are n cities labeled from 1 to n. You are given the integer n and an array connections where connections[i] = [xi, yi, costi] indicates that the cost of connecting city xi and city yi (bidirectional connection) is costi.

Return *the minimum****cost****to connect all the*n*cities such that there is at least one path between each pair of cities*. If it is impossible to connect all the n cities, return -1,

The **cost** is the sum of the connections' costs used.

**Example 1:**

Diagram, polygon

Description automatically generated

**Input:** n = 3, connections = [[1,2,5],[1,3,6],[2,3,1]]

**Output:** 6

**Explanation:** Choosing any 2 edges will connect all cities so we choose the minimum 2.

**Example 2:**

Diagram

Description automatically generated

**Input:** n = 4, connections = [[1,2,3],[3,4,4]]

**Output:** -1

**Explanation:** There is no way to connect all cities even if all edges are used.

**Constraints:**

* 1 <= n <= 104
* 1 <= connections.length <= 104
* connections[i].length == 3
* 1 <= xi, yi <= n
* xi != yi
* 0 <= costi <= 105

### Analysis

This problem can also be resolved as union find. Sort the edge by distance first, select the shortest edge, if they connect to different group add to result, otherwise, discard it.

/// <summary>

/// Leet code #1135. Connecting Cities With Minimum Cost

///

/// There are N cities numbered from 1 to N.

///

/// You are given connections, where each connections[i] =

/// [city1, city2, cost] represents the cost to connect city1

/// and city2 together. (A connection is bidirectional:

/// connecting city1 and city2 is the same as connecting city2

/// and city1.)

///

/// Return the minimum cost so that for every pair of cities,

/// there exists a path of connections (possibly of length 1)

/// that connects those two cities together. The cost is the

/// sum of the connection costs used. If the task is impossible,

/// return -1.

///

/// Example 1:

/// Input: N = 3, connections = [[1,2,5],[1,3,6],[2,3,1]]

/// Output: 6

/// Explanation:

/// Choosing any 2 edges will connect all cities so we choose the minimum 2.

///

/// Example 2:

/// Input: N = 4, connections = [[1,2,3],[3,4,4]]

/// Output: -1

/// Explanation:

/// There is no way to connect all cities even if all edges are used.

///

/// Note:

/// 1. 1 <= N <= 10000

/// 2. 1 <= connections.length <= 10000

/// 3. 1 <= connections[i][0], connections[i][1] <= N

/// 4. 0 <= connections[i][2] <= 10^5

/// 5. connections[i][0] != connections[i][1]

/// </summary>

int LeetCodeGraph::minimumCost(int N, vector<vector<int>>& connections)

{

priority\_queue<vector<int>> paths;

vector<int> parent(N + 1);

int result = 0;

for (size\_t i = 0; i < connections.size(); i++)

{

int x = connections[i][0] - 1;

int y = connections[i][1] - 1;

int d = connections[i][2];

paths.push({ -d, x, y });

}

int n = N;

while (!paths.empty())

{

vector<int> pos = paths.top();

paths.pop();

int x = pos[1];

int y = pos[2];

while (parent[x] != 0) x = parent[x];

while (parent[y] != 0) y = parent[y];

if (x != y)

{

result += -pos[0];

parent[x] = y;

n--;

}

}

if (n == 1) return result;

else return -1;

}

# Advance Problems

## 305. Number of Islands II

Hard

You are given an empty 2D binary grid grid of size m x n. The grid represents a map where 0's represent water and 1's represent land. Initially, all the cells of grid are water cells (i.e., all the cells are 0's).

We may perform an add land operation which turns the water at position into a land. You are given an array positions where positions[i] = [ri, ci] is the position (ri, ci) at which we should operate the ith operation.

Return *an array of integers* answer *where* answer[i] *is the number of islands after turning the cell* (ri, ci) *into a land*.

An **island** is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

**Example 1:**

Shape

Description automatically generated

**Input:** m = 3, n = 3, positions = [[0,0],[0,1],[1,2],[2,1]]

**Output:** [1,1,2,3]

**Explanation:**

Initially, the 2d grid is filled with water.

- Operation #1: addLand(0, 0) turns the water at grid[0][0] into a land. We have 1 island.

- Operation #2: addLand(0, 1) turns the water at grid[0][1] into a land. We still have 1 island.

- Operation #3: addLand(1, 2) turns the water at grid[1][2] into a land. We have 2 islands.

- Operation #4: addLand(2, 1) turns the water at grid[2][1] into a land. We have 3 islands.

**Example 2:**

**Input:** m = 1, n = 1, positions = [[0,0]]

**Output:** [1]

**Constraints:**

* 1 <= m, n, positions.length <= 104
* 1 <= m \* n <= 104
* positions[i].length == 2
* 0 <= ri < m
* 0 <= ci < n

**Follow up:** Could you solve it in time complexity O(k log(mn)), where k == positions.length?

### Analysis:

With every land added, we will merge the land to its neighbors, we will assign an island id (group id) to each new land, and try to merge with the neighbors.

/// <summary>

/// Leet Code 305. Number of Islands II

///

/// Hard

///

/// You are given an empty 2D binary grid grid of size m x n. The grid

/// represents a map where 0's represent water and 1's represent land.

/// Initially, all the cells of grid are water cells (i.e., all the

/// cells are 0's).

///

/// We may perform an add land operation which turns the water at

/// position into a land. You are given an array positions where

/// positions[i] = [ri, ci] is the position (ri, ci) at which we

/// should operate the ith operation.

///

/// Return an array of integers answer where answer[i] is the

/// number of islands after turning the cell (ri, ci) into a land.

///

/// An island is surrounded by water and is formed by connecting

/// adjacent lands horizontally or vertically. You may assume all

/// four edges of the grid are all surrounded by water.

/// Example 1:

/// Input: m = 3, n = 3, positions = [[0,0],[0,1],[1,2],[2,1]]

/// Output: [1,1,2,3]

/// Explanation:

/// Initially, the 2d grid is filled with water.

/// - Operation #1: addLand(0, 0) turns the water at grid[0][0] into a

/// land. We have 1 island.

/// - Operation #2: addLand(0, 1) turns the water at grid[0][1] into a

/// land. We still have 1 island.

/// - Operation #3: addLand(1, 2) turns the water at grid[1][2] into a

/// land. We have 2 islands.

/// - Operation #4: addLand(2, 1) turns the water at grid[2][1] into a

/// land. We have 3 islands.

///

/// Example 2:

/// Input: m = 1, n = 1, positions = [[0,0]]

/// Output: [1]

///

/// Constraints:

/// 1. 1 <= m, n, positions.length <= 10^4

/// 2. 1 <= m \* n <= 10^4

/// 3. positions[i].length == 2

/// 4. 0 <= ri < m

/// 5. 0 <= ci < n

///

/// Follow up: Could you solve it in time complexity O(k log(mn)), where

/// k == positions.length?

/// </summary>

vector<int> LeetCodeGraph::numIslands2(int m, int n, vector<vector<int>>& positions)

{

vector<int> result;

vector<int> parent(m \* n + 1);

for (size\_t i = 0; i < parent.size(); i++) parent[i] = i;

vector<vector<int>> grid\_map(m, vector<int>(n));

int count = 0;

for (size\_t i = 0; i < positions.size(); i++)

{

vector<int> pos = positions[i];

// already set before

if (grid\_map[pos[0]][pos[1]] == 0)

{

int island\_id = i + 1;

grid\_map[pos[0]][pos[1]] = island\_id;

count++;

vector<vector<int>> directions = { {-1, 0}, {1, 0}, {0, -1}, {0, 1} };

for (size\_t d = 0; d < directions.size(); d++)

{

vector<int> next = pos;

next[0] += directions[d][0];

next[1] += directions[d][1];

if ((next[0] < 0) || (next[0] >= (int)grid\_map.size()) ||

(next[1] < 0) || (next[1] >= (int)grid\_map[0].size()))

{

continue;

}

int neighbor = grid\_map[next[0]][next[1]];

// water

if (neighbor == 0)

{

continue;

}

while (parent[parent[neighbor]] != parent[neighbor]) parent[neighbor] = parent[parent[neighbor]];

if (parent[neighbor] != island\_id)

{

parent[parent[neighbor]] = island\_id;

count--;

}

}

}

result.push\_back(count);

}

return result;

}

## 952. Largest Component Size by Common Factor

Hard

You are given an integer array of unique positive integers nums. Consider the following graph:

* There are nums.length nodes, labeled nums[0] to nums[nums.length - 1],
* There is an undirected edge between nums[i] and nums[j] if nums[i] and nums[j] share a common factor greater than 1.

Return *the size of the largest connected component in the graph*.

**Example 1:**

Diagram

Description automatically generated

**Input:** nums = [4,6,15,35]

**Output:** 4

**Example 2:**

A picture containing shape

Description automatically generated

**Input:** nums = [20,50,9,63]

**Output:** 2

**Example 3:**

Diagram

Description automatically generated

**Input:** nums = [2,3,6,7,4,12,21,39]

**Output:** 8

**Constraints:**

* 1 <= nums.length <= 2 \* 104
* 1 <= nums[i] <= 105
* All the values of nums are **unique**.

### Analysis:

Union Find with common factor

/// <summary>

/// Leet code #952. Largest Component Size by Common Factor

///

/// Given a non-empty array of unique positive integers A, consider the

/// following graph:

///

/// There are A.length nodes, labelled A[0] to A[A.length - 1];

/// There is an edge between A[i] and A[j] if and only if A[i] and A[j]

/// share a common factor greater than 1.

/// Return the size of the largest connected component in the graph.

///

/// Example 1:

/// Input: [4,6,15,35]

/// Output: 4

///

/// Example 2:

/// Input: [20,50,9,63]

/// Output: 2

///

/// Example 3:

/// Input: [2,3,6,7,4,12,21,39]

/// Output: 8

///

/// Note:

/// 1. 1 <= A.length <= 20000

/// 2. 1 <= A[i] <= 100000

/// </summary>

int LeetCodeGraph::largestComponentSize(vector<int>& A)

{

vector<vector<int>> factors(A.size());

unordered\_map<int, int> prime\_map;

vector<int> union\_map(A.size());

vector<int> count(A.size(), 1);

for (size\_t i = 0; i < A.size(); i++)

{

int x = A[i];

for (size\_t d = 2; d <= sqrt(x); d++)

{

if (x % d == 0)

{

while (x % d == 0) x /= d;

factors[i].push\_back(d);

}

}

if (x > 1) factors[i].push\_back(x);

}

int result = 1;

for (size\_t i = 0; i < A.size(); i++)

{

union\_map[i] = i;

count[i] = 1;

for (auto p : factors[i])

{

if (prime\_map.count(p) == 0)

{

prime\_map[p] = i;

}

else

{

int source = i;

int target = prime\_map[p];

while (union\_map[source] != source) source = union\_map[source];

while (union\_map[target] != target) target = union\_map[target];

if (source != target)

{

union\_map[source] = target;

count[target] += count[source];

result = max(count[target], result);

}

}

}

}

return result;

}

## 839. Similar String Groups

Hard

Two strings X and Y are similar if we can swap two letters (in different positions) of X, so that it equals Y. Also two strings X and Y are similar if they are equal.

For example, "tars" and "rats" are similar (swapping at positions 0 and 2), and "rats" and "arts" are similar, but "star" is not similar to "tars", "rats", or "arts".

Together, these form two connected groups by similarity: {"tars", "rats", "arts"} and {"star"}.  Notice that "tars" and "arts" are in the same group even though they are not similar.  Formally, each group is such that a word is in the group if and only if it is similar to at least one other word in the group.

We are given a list strs of strings where every string in strs is an anagram of every other string in strs. How many groups are there?

**Example 1:**

**Input:** strs = ["tars","rats","arts","star"]

**Output:** 2

**Example 2:**

**Input:** strs = ["omv","ovm"]

**Output:** 1

**Constraints:**

* 1 <= strs.length <= 300
* 1 <= strs[i].length <= 300
* strs[i] consists of lowercase letters only.
* All words in strs have the same length and are anagrams of each other.

### Analysis:

This can be resolved in Union Find, but it can also be resolved by BFS which find the similar string in one group.

/// <summary>

/// Leet code #839. Similar String Groups

///

/// Two strings X and Y are similar if we can swap two letters (in

/// different positions) of X, so that it equals Y.

///

/// For example, "tars" and "rats" are similar (swapping at positions

/// 0 and 2), and "rats" and "arts" are similar, but "star" is not

/// similar to "tars", "rats", or "arts".

///

/// Together, these form two connected groups by similarity: {"tars",

/// "rats", "arts"} and {"star"}. Notice that "tars" and "arts" are

/// in the same group even though they are not similar. Formally, each

/// group is such that a word is in the group if and only if it is similar

/// to at least one other word in the group.

///

/// We are given a list A of unique strings. Every string in A is an

/// anagram of every other string in A. How many groups are there?

///

/// Example 1:

///

/// Input: ["tars","rats","arts","star"]

/// Output: 2

/// Note:

///

/// 1.A.length <= 2000

/// 2.A[i].length <= 1000

/// 3.A.length \* A[i].length <= 20000

/// 4.All words in A consist of lowercase letters only.

/// 5.All words in A have the same length and are anagrams of each other.

/// 6.The judging time limit has been increased for this question.

/// </summary>

int LeetCode::numSimilarGroups(vector<string>& A)

{

vector<int> visited(A.size());

int result = 0;

for (size\_t i = 0; i < A.size(); i++)

{

if (visited[i] == 1) continue;

queue<string> search;

search.push(A[i]);

visited[i] = 1;

result++;

while (!search.empty())

{

string str = search.front();

search.pop();

for (size\_t j = 0; j < A.size(); j++)

{

if (visited[j] == 1) continue;

int diff\_count = 0;

for (size\_t k = 0; k < str.size(); k++)

{

if (str[k] != A[j][k]) diff\_count++;

if (diff\_count > 2) break;

}

if (diff\_count == 2)

{

search.push(A[j]);

visited[j] = 1;

}

}

}

}

return result;

}

## USACO 2013 January, Bronze, Liars and Truth Tellers

/// <summary>

/// USACO 2013 January Contest, Bronze

///

/// Problem 3: Liars and Truth Tellers [Brian Dean, 2013]

///

/// After spending so much time around his cows, Farmer John has started to

/// understand their language. Moreover, he notices that among his N cows

/// (2 <= N <= 1000), some always tell the truth while others always lie.

///

/// FJ carefully listens to M statements (1 <= M <= 10,000) from his cows, each

/// of the form "x y T", meaning that "cow x claims cow y always tells the

/// truth" or "x y L", meaning that "cow x claims cow y always tells lies".

/// Each statement involves a pair of different cows, and the same pair of cows

/// may appear in multiple statements.

///

/// Unfortunately, FJ believes he might have written down some entries in his

/// list incorrectly, so there may not be a valid way to designate each cow as

/// a truth teller or a liar that is consistent with all the M statements on

/// FJ's list. To help FJ salvage as much of his list as possible, please

/// compute the largest value of A such that there exists a valid way to

/// designate each cow as a truth teller or a liar in a manner that is

/// consistent with the first A entries in FJ's list.

///

/// PROBLEM NAME: truth

///

/// INPUT FORMAT:

///

/// \* Line 1: Two space-separated integers, N and M.

///

/// \* Lines 2..1+M: Each line is of the form "x y L" or "x y T",

/// describing a statement made by cow x about cow y.

///

/// SAMPLE INPUT (file truth.in):

///

/// 4 3

/// 1 4 L

/// 2 3 T

/// 4 1 T

///

/// INPUT DETAILS:

///

/// There are 4 cows and 3 statements. Cow 1 says that cow 4 lies, cow 2 says

/// that cow 3 tells the truth, and cow 4 says that cow 1 tells the truth.

///

/// OUTPUT FORMAT:

///

/// \* Line 1: The maximum value of A such that the first A entries in FJ's

/// list can be consistent with some assignment of "truth teller"

/// or "liar" to the N cows.

///

/// SAMPLE OUTPUT (file truth.out):

///

/// 2

///

/// OUTPUT DETAILS:

///

/// Statements 1 and 3 cannot both be satisfied at the same time, but

/// statements 1 and 2 can be, if we let cows 1..3 tell the truth and cow 4 be

/// a liar.

/// </summary>

class Truth

{

private:

static int find\_root(unordered\_map<int, int> &component, int id)

{

if (component.count(id) == 0)

{

component[id] = id;

component[-id] = -id;

}

else

{

while (component[id] != id) id = component[id];

}

return id;

}

static void merge(unordered\_map<int, int> &component, int x, int y)

{

component[y] = x;

component[-y] = -x;

}

public:

static void process(void)

{

const string m\_Folder = "2013\\Bronze\\Truth\\";

ifstream fin(m\_Folder + "truth.in");

int N, M;

fin >> N >> M;

vector<pair<pair<int, int>, char>> truths(M);

for (int i = 0; i < M; i++)

{

fin >> truths[i].first.first >> truths[i].first.second >> truths[i].second;

}

fin.close();

int result = 0;

unordered\_map<int, int> component;

for (int i = 0; i < M; i++)

{

int first = truths[i].first.first;

int second = truths[i].first.second;

int x = find\_root(component, first);

int y = find\_root(component, second);

if ((truths[i].second == 'T' && x == -y) ||

(truths[i].second == 'L' && x == y))

{

break;

}

else if (truths[i].second == 'T')

{

merge(component, x, y);

}

else

{

merge(component, x, -y);

}

result++;

}

ofstream fout(m\_Folder + "truth.out", std::ofstream::out | std::ofstream::trunc);

fout << to\_string(result) << endl;

fout.close();

}

};