# Union Find

Union find is to group many items into one group as long as it has some relationship with another item in that group. Such relationship may be transitive, for example, if A and B are in same group and B and C are in same group then A and C must be in same group.

We need to point the nodes in the same group to one root. Make it something like a tree. A hashtable or vector mapping is normally the data structure what we need. At very beginning every node points to itself, when we group two nodes, we will point the root of one node to the root of another.

// find the root of the first word

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

// 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;

During the processing, we can compress path in the tree.

// find the root of the first word

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

{

similar\_words[first] = similar\_words[similar\_words[first]];

first = similar\_words[first];

}

## 305. Number of Islands II

Hard

A 2d grid map of m rows and n columns is initially filled with water. We may perform an *addLand* operation which turns the water at position (row, col) into a land. Given a list of positions to operate, **count the number of islands after each *addLand* operation**. 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:**

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

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

**Explanation:**

Initially, the 2d grid grid is filled with water. (Assume 0 represents water and 1 represents land).

0 0 0

0 0 0

0 0 0

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

1 0 0

0 0 0 Number of islands = 1

0 0 0

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

1 1 0

0 0 0 Number of islands = 1

0 0 0

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

1 1 0

0 0 1 Number of islands = 2

0 0 0

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

1 1 0

0 0 1 Number of islands = 3

0 1 0

**Follow up:**

Can you do it in time complexity O(k log mn), where k is the length of the positions?

### Analysis

We can map the neighbor to the root, when we merge two island, we deduct one.

/// <summary>

/// Leet code #305. Number of Islands II

/// </summary>

int LeetCode::checkIslands(int island\_id, int row, int col, vector<vector<int>>&grid\_map, vector<int>& island\_map)

{

if ((row < 0) || (row >= (int)grid\_map.size()) ||

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

{

return 0;

}

if ((grid\_map[row][col] == 0) || (island\_id == grid\_map[row][col]))

{

return 0;

}

return mergeIslands(island\_id, grid\_map[row][col], grid\_map, island\_map);

}

/// <summary>

/// Leet code #305. Number of Islands II

/// </summary>

int LeetCode::mergeIslands(int island1, int island2, vector<vector<int>>&grid\_map, vector<int>& island\_map)

{

int count = 0;

while (island\_map[island1] != 0) island1 = island\_map[island1];

while (island\_map[island2] != 0) island2 = island\_map[island2];

if (island1 == island2) return count;

count++;

island\_map[island2] = island1;

return count;

}

/// <summary>

/// Leet code #305. Number of Islands II

///

/// A 2d grid map of m rows and n columns is initially filled with water. We

/// may perform an addLand operation which turns the water at position (row, col)

/// into a land.

/// Given a list of positions to operate, count the number of islands after each

/// addLand operation.

/// 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:

///

/// Given m = 3, n = 3, positions = [[0,0], [0,1], [1,2], [2,1]].

/// Initially, the 2d grid grid is filled with water. (Assume 0 represents water

/// and 1 represents land).

/// 0 0 0

/// 0 0 0

/// 0 0 0

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

/// 1 0 0

/// 0 0 0 Number of islands = 1

/// 0 0 0

///

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

/// 1 1 0

/// 0 0 0 Number of islands = 1

/// 0 0 0

///

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

/// 1 1 0

/// 0 0 1 Number of islands = 2

/// 0 0 0

///

///

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

/// 1 1 0

/// 0 0 1 Number of islands = 3

/// 0 1 0

///

/// We return the result as an array: [1, 1, 2, 3]

///

/// Challenge:

/// Can you do it in time complexity O(k log mn), where k is the length of the

/// positions?

/// </summary>

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

{

vector<int> result;

vector<int> island\_map(m\*n);

vector<vector<int>> grid\_map(m, vector<int>(n, 0));

int count = 0;

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

{

pair<int, int> pos = positions[i];

int island\_id = i + 1;

grid\_map[pos.first][pos.second] = island\_id;

island\_map[island\_id] = 0;

count++;

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

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

{

count -= checkIslands(

grid\_map[pos.first][pos.second],

pos.first + directions[i][0],

pos.second + directions[i][1],

grid\_map,

island\_map);

}

result.push\_back(count);

}

return result;

}

## 547. Friend Circles

Medium

There are **N** students in a class. Some of them are friends, while some are not. Their friendship is transitive in nature. For example, if A is a **direct** friend of B, and B is a **direct** friend of C, then A is an **indirect** friend of C. And we defined a friend circle is a group of students who are direct or indirect friends.

Given a **N\*N** matrix **M** representing the friend relationship between students in the class. If M[i][j] = 1, then the ith and jth students are **direct** friends with each other, otherwise not. And you have to output the total number of friend circles among all the students.

**Example 1:**

**Input:**

[[1,1,0],

[1,1,0],

[0,0,1]]

**Output:** 2

**Explanation:**The 0th and 1st students are direct friends, so they are in a friend circle.   
The 2nd student himself is in a friend circle. So return 2.

**Example 2:**

**Input:**

[[1,1,0],

[1,1,1],

[0,1,1]]

**Output:** 1

**Explanation:**The 0th and 1st students are direct friends, the 1st and 2nd students are direct friends,   
so the 0th and 2nd students are indirect friends. All of them are in the same friend circle, so return 1.

**Note:**

1. N is in range [1,200].
2. M[i][i] = 1 for all students.
3. If M[i][j] = 1, then M[j][i] = 1.

### Analysis

We can map a circle of friends into one leader. So when we found two people in different circle become friend just map one leader to another

/// <summary>

/// Leet code #547. Friend Circles

///

/// There are N students in a class. Some of them are friends, while some

/// are not. Their friendship is transitive in nature. For example, if A

/// is a direct friend of B, and B is a direct friend of C, then A is an

/// indirect friend of C. And we defined a friend circle is a group of

/// students who are direct or indirect friends.

/// Given a N\*N matrix M representing the friend relationship between

/// students in the class. If M[i][j] = 1, then the ith and jth students

/// are direct friends with each other, otherwise not. And you have to

/// output the total number of friend circles among all the students.

/// Example 1:

/// Input:

/// [[1,1,0],

/// [1,1,0],

/// [0,0,1]]

/// Output: 2

/// Explanation:The 0th and 1st students are direct friends, so they are

/// in a friend circle.

/// The 2nd student himself is in a friend circle. So return 2.

/// Example 2:

/// Input:

/// [[1,1,0],

/// [1,1,1],

/// [0,1,1]]

/// Output: 1

/// Explanation:The 0th and 1st students are direct friends, the 1st and

/// 2nd students are direct friends,

/// so the 0th and 2nd students are indirect friends. All of them are in

/// the same friend circle, so return 1.

///

/// Note:

/// N is in range [1,200].

/// M[i][i] = 1 for all students.

/// If M[i][j] = 1, then M[j][i] = 1.

/// </summary>

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

{

vector<int> circle\_map(M.size());

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

{

circle\_map[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 (circle\_map[source] != source) source = circle\_map[source];

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

circle\_map[target] = source;

}

}

}

int count = 0;

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

{

if (circle\_map[i] == i) count++;

}

return count;

}

## 737. Sentence Similarity II

Medium

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].

### Analysis

We can map two similar sentences into same leader.

/// <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 LeetCode::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;

}

## 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.

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 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.

### Analysis

We can traverse from any node, use BFS to find all the similar string and we are sure the remaining are not similar.

/// <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;

}

## 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;

}

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

Medium

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

### Analysis

If merge the stones in same row and same column.

/// <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 LeetCode::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;

}

## 952. Largest Component Size by Common Factor

Hard

Given a non-empty array of unique positive integers A, consider the following graph:

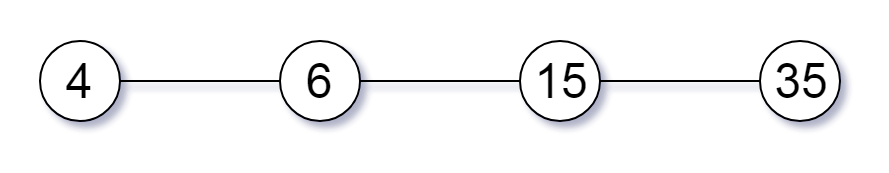
* There are A.length nodes, labelled A[0] to A[A.length - 1];
* There is an edge between A[i] and A[j] if and only if A[i] and A[j] share a common factor greater than 1.

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]

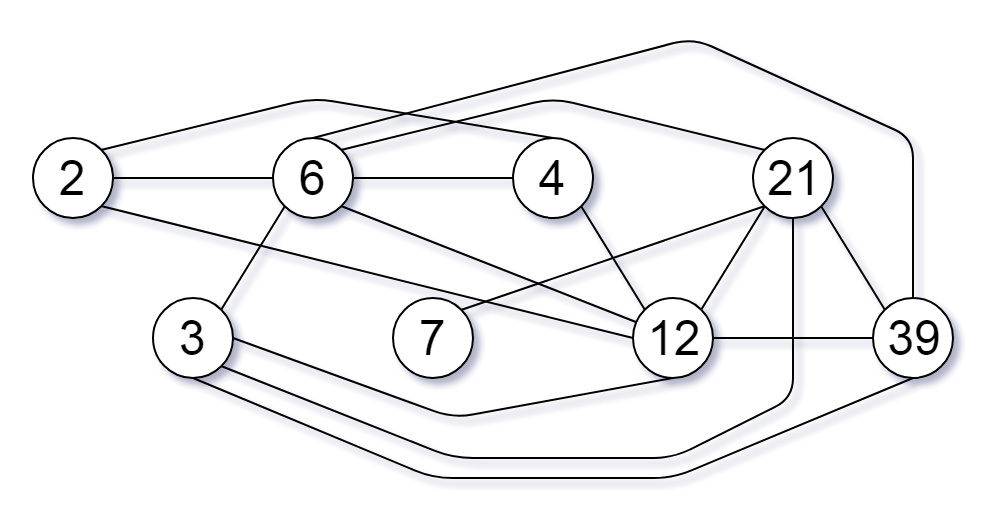
**Output:** 2



**Example 3:**

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

**Output:** 8



**Note:**

1. 1 <= A.length <= 20000
2. 1 <= A[i] <= 100000

### Analysis

If any two numbers have the common factor, they should be merged into the same group and add up count.

/// <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 LeetCode::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;

}

## 990. Satisfiability of Equality Equations

Medium

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 '='

### Analysis

We will assign each variable an id and merge the equal ones and assign an opposite number to the unequal ones.

/// <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 LeetCode::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;

}

## 1061. Lexicographically Smallest Equivalent String

Medium

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.

### Analysis

We merge the characters in same group and select 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 LeetCode::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

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. 2 <= 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>

/// 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 LeetCode::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;

}

## 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();

}

};