You are given an array of variable pairs equations and an array of real numbers values, where equations[i] = [Ai, Bi] and values[i] represent the equation Ai / Bi = values[i]. Each Ai or Bi is a string that represents a single variable.

You are also given some queries, where queries[j] = [Cj, Dj] represents the jth query where you must find the answer for Cj / Dj = ?.

Return *the answers to all queries*. If a single answer cannot be determined, return -1.0.

**Note:** The input is always valid. You may assume that evaluating the queries will not result in division by zero and that there is no contradiction.

**Example 1:**

**Input:** equations = [["a","b"],["b","c"]], values = [2.0,3.0], queries = [["a","c"],["b","a"],["a","e"],["a","a"],["x","x"]]

**Output:** [6.00000,0.50000,-1.00000,1.00000,-1.00000]

**Explanation:**

Given: *a / b = 2.0*, *b / c = 3.0*

queries are: *a / c = ?*, *b / a = ?*, *a / e = ?*, *a / a = ?*, *x / x = ?*

return: [6.0, 0.5, -1.0, 1.0, -1.0 ]

**Example 2:**

**Input:** equations = [["a","b"],["b","c"],["bc","cd"]], values = [1.5,2.5,5.0], queries = [["a","c"],["c","b"],["bc","cd"],["cd","bc"]]

**Output:** [3.75000,0.40000,5.00000,0.20000]

**Example 3:**

**Input:** equations = [["a","b"]], values = [0.5], queries = [["a","b"],["b","a"],["a","c"],["x","y"]]

**Output:** [0.50000,2.00000,-1.00000,-1.00000]

**Constraints:**

* 1 <= equations.length <= 20
* equations[i].length == 2
* 1 <= Ai.length, Bi.length <= 5
* values.length == equations.length
* 0.0 < values[i] <= 20.0
* 1 <= queries.length <= 20
* queries[i].length == 2
* 1 <= Cj.length, Dj.length <= 5
* Ai, Bi, Cj, Dj consist of lower case English letters and digits.

/\*\*

\* @param {string[][]} equations

\* @param {number[]} values

\* @param {string[][]} queries

\* @return {number[]}

\*/

var calcEquation = function(equations, values, queries) {

};

Solution

Overview

As revealed by the hints, the problem can be solved with two important data structures, namely *Graph* and *Union-Find*.

In the following sections, we will explain how to solve the problem respectively with regards to the data structures.

Approach 1: Path Search in Graph

**Intuition**

First, let us look at the example given in the problem description. Given two equations, namely \frac{a}{b} = 2, \space \frac{b}{c} = 3*ba*​=2, *cb*​=3, we could derive the following equations:

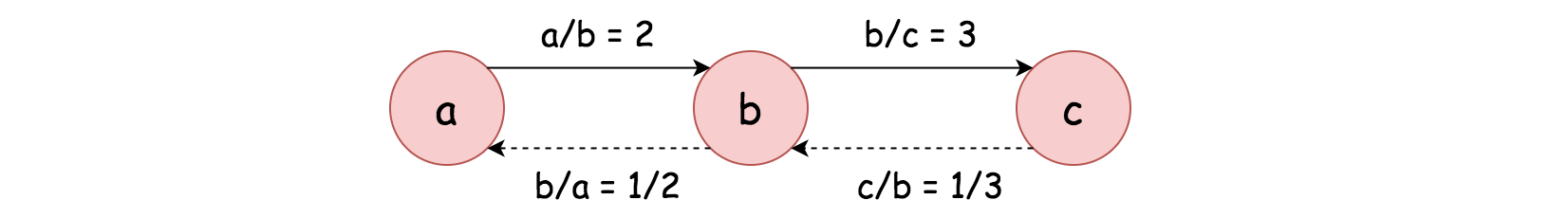
* 1). \frac{b}{a} = \frac{1}{2}, \space \frac{c}{b} = \frac{1}{3}*ab*​=21​, *bc*​=31​
* 2). \frac{a}{c} = \frac{a}{b} \cdot \frac{b}{c} = 2 \cdot 3 = 6*ca*​=*ba*​⋅*cb*​=2⋅3=6

Each division implies the reverse of the division, which is how we derive the equations in **(1)**. While by ***chaining up*** equations, we could obtain new equations in **(2)**.

We could reformulate the equations with the *graph* data structure, where each variable can be represented as a **node** in the graph, and the division relationship between variables can be modeled as **edge** with direction and weight.

The direction of edge indicates the order of division, and the weight of edge indicates the result of division.

With the above formulation, we then can convert the initial equations into the following graph:



To evaluate a query (*e.g.* \frac{a}{c}=?*ca*​=?) is equivalent to performing two tasks on the graph: 1). find if there exists a path between the two entities. 2). if so, calculate the cumulative products along the paths.

In the above example (\frac{a}{c}=?*ca*​=?), we could find a path between them, and the cumulative products are 66. As a result, we can conclude that the result of \frac{a}{c}*ca*​ is 2 \cdot 3 = 62⋅3=6.

**Algorithm**

As one can see, we just transform the problem into a **path searching** problem in a graph.

More precisely, we can reinterpret the problem as "given two nodes, we are asked to check if there exists a path between them. If so, we should return the cumulative products along the path as the result.

Given the above problem statement, it seems intuitive that one could apply the *backtracking* algorithm, or sometimes people might call it *DFS* (Depth-First Search).

Essentially, we can break down the algorithm into two steps overall:

* Step 1). we build the graph out of the list of input equations.
  + Each equation corresponds to two edges in the graph.
* Step 2). once the graph is built, we then can evaluate the query *one by one*.
  + The evaluation of the query is done via searching the path between the given two variables.
  + Other than the above searching operation, we need to handle two *exceptional* cases as follows:
  + Case 1): if either of the nodes does not exist in the graph, *i.e.* the variables did not appear in any of the input equations, then we can assert that no path exists.
  + Case 2): if the origin and the destination are the same node, *i.e.* \frac{a}{a}*aa*​, we can assume that there exists an invisible self-loop path for each node and the result is one.

Here we give one sample implementation on the backtracking algorithm.

class Solution {

public double[] calcEquation(List<List<String>> equations, double[] values,

List<List<String>> queries) {

HashMap<String, HashMap<String, Double>> graph = new HashMap<>();

// Step 1). build the graph from the equations

for (int i = 0; i < equations.size(); i++) {

List<String> equation = equations.get(i);

String dividend = equation.get(0), divisor = equation.get(1);

double quotient = values[i];

if (!graph.containsKey(dividend))

graph.put(dividend, new HashMap<String, Double>());

if (!graph.containsKey(divisor))

graph.put(divisor, new HashMap<String, Double>());

graph.get(dividend).put(divisor, quotient);

graph.get(divisor).put(dividend, 1 / quotient);

}

// Step 2). Evaluate each query via bactracking (DFS)

// by verifying if there exists a path from dividend to divisor

double[] results = new double[queries.size()];

for (int i = 0; i < queries.size(); i++) {

List<String> query = queries.get(i);

String dividend = query.get(0), divisor = query.get(1);

if (!graph.containsKey(dividend) || !graph.containsKey(divisor))

results[i] = -1.0;

else if (dividend == divisor)

results[i] = 1.0;

else {

HashSet<String> visited = new HashSet<>();

results[i] = backtrackEvaluate(graph, dividend, divisor, 1, visited);

}

}

return results;

}

private double backtrackEvaluate(HashMap<String, HashMap<String, Double>> graph, String currNode, String targetNode, double accProduct, Set<String> visited) {

// mark the visit

visited.add(currNode);

double ret = -1.0;

Map<String, Double> neighbors = graph.get(currNode);

if (neighbors.containsKey(targetNode))

ret = accProduct \* neighbors.get(targetNode);

else {

for (Map.Entry<String, Double> pair : neighbors.entrySet()) {

String nextNode = pair.getKey();

if (visited.contains(nextNode))

continue;

ret = backtrackEvaluate(graph, nextNode, targetNode,

accProduct \* pair.getValue(), visited);

if (ret != -1.0)

break;

}

}

// unmark the visit, for the next backtracking

visited.remove(currNode);

return ret;

}

}

Note: with the built graph, one could also apply the **BFS** (*Breadth-First Search*) algorithm, as opposed to the DFS algorithm we employed.

However, the essence of the solution remains the same, *i.e.* we are searching for a path in a graph.

**Complexity Analysis**

Let N*N* be the number of input equations and M*M* be the number of queries.

* Time Complexity: \mathcal{O}(M \cdot N)O(*M*⋅*N*)
  + First of all, we iterate through the equations to build a graph. Each equation takes \mathcal{O}(1)O(1) time to process. Therefore, this step will take \mathcal{O}(N)O(*N*) time in total.
  + For each query, we need to traverse the graph. In the worst case, we might need to traverse the entire graph, which could take \mathcal{O}(N)O(*N*). Hence, in total, the evaluation of queries could take M \cdot \mathcal{O}(N) = \mathcal{O}(M \cdot N)*M*⋅O(*N*)=O(*M*⋅*N*).
  + To sum up, the overall time complexity of the algorithm is \mathcal{O}(N) + \mathcal{O}(M \cdot N) = \mathcal{O}(M \cdot N)O(*N*)+O(*M*⋅*N*)=O(*M*⋅*N*)
* Space Complexity: \mathcal{O}(N)O(*N*)
  + We build a graph out the equations. In the worst case where there is no overlapping among the equations, we would have N*N* edges and 2N2*N* nodes in the graph. Therefore, the sapce complexity of the graph is \mathcal{O}(N + 2N) = \mathcal{O}(3N) = \mathcal{O}(N)O(*N*+2*N*)=O(3*N*)=O(*N*).
  + Since we employ the recursion in the backtracking, we would consume additional memory in the function call stack, which could amount to \mathcal{O}(N)O(*N*) space.
  + In addition, we used a set visited to keep track of the nodes we visited during the backtracking. The space complexity of the visited set would be \mathcal{O}(N)O(*N*).
  + To sum up, the overall space complexity of the algorithm is \mathcal{O}(N) + \mathcal{O}(N) + \mathcal{O}(N) = \mathcal{O}(N)O(*N*)+O(*N*)+O(*N*)=O(*N*).
  + Note that we did not take into account the space needed to hold the results. Otherwise, the total space complexity would be \mathcal{O}(N + M)O(*N*+*M*).

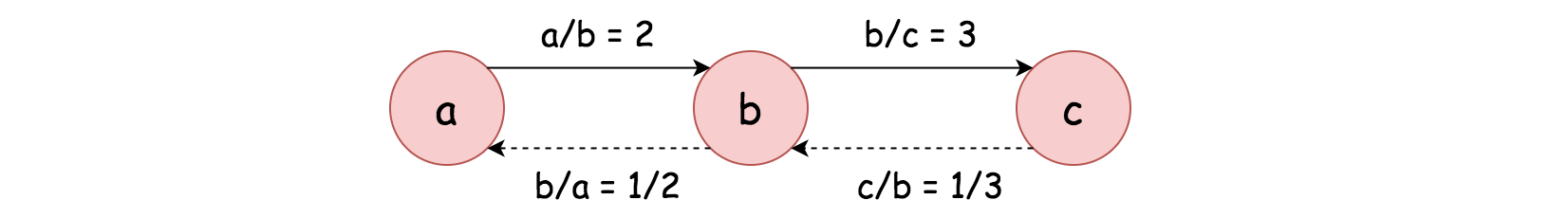
Approach 2: Union-Find with Weights

**Intuition**

As we mentioned before, the problem can also be solved with the **Union-Find** data structure and algorithm.

As a reminder, the [Union-Find](https://en.wikipedia.org/wiki/Disjoint-set_data_structure) data structure, also known as *Disjoint Set*, is used to track a set of elements partitioned into a number of disjoint (non-overlapping) subsets. The Union-Find data structure is often applied to solve the **graph partition** problem, where we partition a graph into a set of inter-connected subgraph.

Given the above description, it is not immediately evident that how we can apply the algorithm in this problem. To get familiar with the Union-Find data structure, we would recommend one to check out another problem called [Largest Component Size by Common Factor](https://leetcode.com/problems/largest-component-size-by-common-factor/), where we can apply the Union-Find algorithm in a more ***canonical*** way.



One thing is clear though. Thanks to the characteristic of the Union-Find data structure, we can easily determine whether the nodes of a and c belong to the same *group* in the above graph, which accomplishes the first task that we need to perform, *i.e.* determining if there exists a path between two nodes.

However, one important task is missing, which is how can we calculate the cumulative product along the path, with the Union-Find data structure.

As a spoiler alert, it suffices to adapt the Union-Find data structure and algorithm a little bit.

**Customized Union-Find Data Structure**

The name of Union-Find data structure is originated from the fact that it mainly consists of two operations: Union() and Find() defined as follows:

* Find(x): get the identity of the group that the element x belongs to.
* Union(x, y): merge the two groups that the two elements belong to respectively.

Now, here are the adaptions that we will do. Or more precisely, here are a few **behaviors** that our *customized* Union-Find data structure would possess at the end.

First of all, essentially we will build a *table* which contains an entry for each node in the graph, with the help of Union-Find.

The entry is defined as key -> (group\_id, weight). For example, initially, given a node a, its entry in the table would look like 'a' -> ('a', 1), where the first 'a'indicates the id of the node, the second 'a' indicates the id of the group that the node belongs to, and finally the value 1 indicates the weight of the node.

With the above definitions, the tasks become simple. Given two nodes (variables a and b) with entries as (a\_group\_id, a\_weight) and (b\_group\_id, b\_weight) respectively, to evaluate the query of \frac{a}{b} = ?*ba*​=?, we only need to perform the following two calculations:

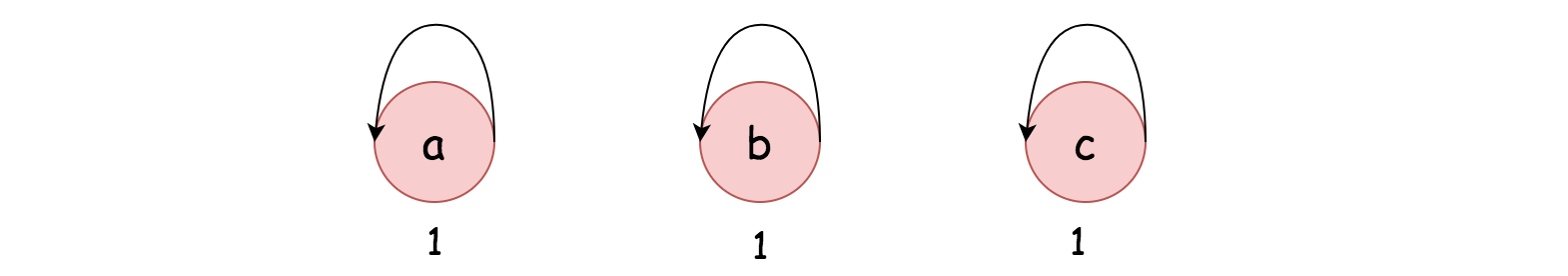
* a\_group\_id == b\_group\_id: If so, they belong to the same group, *i.e.* there exists a path between them.
* a\_weight / b\_weight: If the above condition holds, by dividing over the ***relative*** weights that are assigned to the variables, we then can obtain the result of \frac{a}{b}*ba*​ at the end.

Now it all boils down to how we can build the above table with the help of Union-Find algorithm.

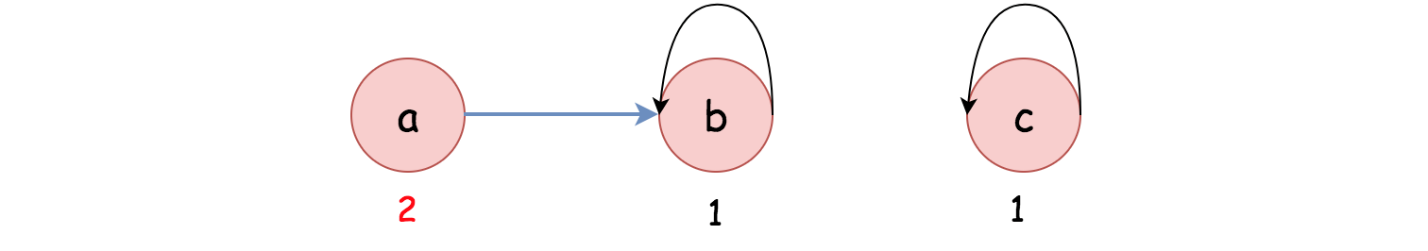
Again, let us look at the same example we presented before.

We have two equations as input, namely \frac{a}{b} = 2, \space \frac{b}{c} = 3*ba*​=2, *cb*​=3.

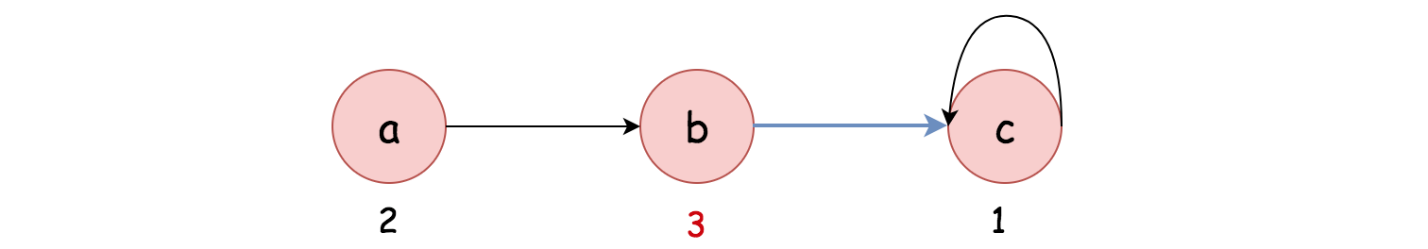
* Initially, the entries for each variable would look like the following, where the group\_id of each variable is the variable itself and the weight of each variable is 1. Each variable forms a group on its own, since there is no relationship among them at the moment.



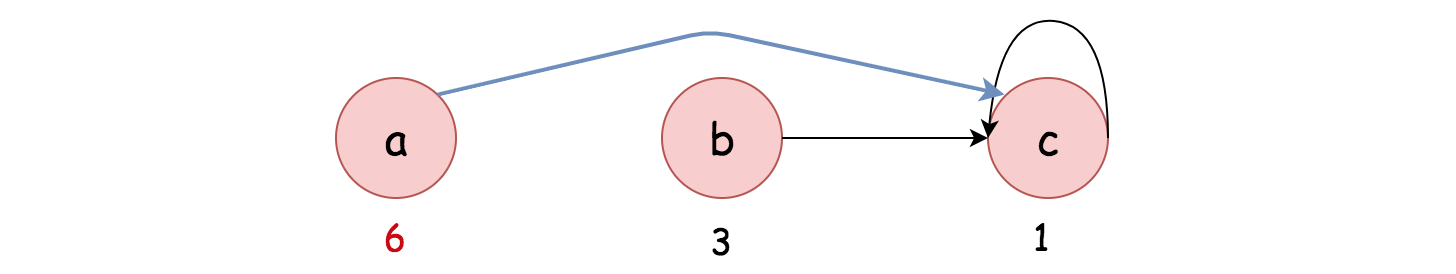
* Now if we process the equation \frac{a}{b}=2*ba*​=2, by joining (**Union** operation) the two groups that the variables a and b belong to, we would obtain the results as shown in the following graph. More precisely, we attach the group of dividend a to the one of the divisor b. Meanwhile, we would also update the *relative* weight of the group a to reflect the ratio between the two variables.



* Similarly, we continue to process the equation of \frac{b}{c}=3*cb*​=3, by joining (**Union** operation) the groups of b and c together. Similarly, we attach the group of dividend b to the one of divisor c. And also we update the weight of the group b to reflect the ratio between the two variables.



* As one might notice, there is some inconsistency in the above graph, *i.e.* the group\_id of the variable of a should then be c and the weight of the variable a should be 6 rather than 2. Indeed, these inconsistencies are expected. The **magic** happens when we invoke the **Find** operation on the variable a, where a *chain* reaction would be triggered to update the group\_id and weight along the chain, as follows:



Once the **lazy** evaluation of find() is triggered, the states of the nodes along the chain would then be updated, and eventually they become consistent.

The mechanism of update is fairly similar with the DFS traversal, as one will see more in detail in the implementation later.

**Algorithm**

Now that we have defined the behaviors for the desired Union-Find data structure, let us put them down into implementation.

The overall interfaces of our Union-Find data structure remain the same. We will implement two functions: find(variable) and union(dividend, divisor, quotient).

* find(variable): the function will return the group\_id that the variable belongs to. Moreover, the function will update the states of variables along the chain, if there is any discrepancy.
* union(dividend, divisor, quotient): this function will attach the group of dividend to that of the divisor, if they are not already the same group. In addition, it needs to update the weight of the dividend variable accordingly, so that the ratio between the dividend and divisor is respected.

We present a sample implementation of the above two functions in the later section, which is inspired from the post of [WangQiuc](https://leetcode.com/problems/evaluate-division/discuss/270993/Python-BFS-and-UF(detailed-explanation)) in the discussion forum. Concise the implementation might be, it might be tricky to wrap one's head around it. One might want to refer to the step-wise example we showed before.

Once we implement the above two functions, we then solve the problem in two steps:

* Step 1): we iterate through each input equation, and invoke the union(dividend, divisor, quotient) on each of them, in order to build the Union-Find data structure.
* Step 2): we evaluate the query one by one. The evaluation is just as intuitive as our first approach, which can be broken down into the following cases:
  + case 1): Either of the variables did not appear in the input equations. The query is not valid. We then return -1.0 as the result.
  + case 2): If both variables are valid, we then apply the find(variable) to obtain the tuple of (group\_id, weight) for each variable. If they are not of the same group, *i.e.* there is no chain of division between them, we then return -1.0 as the result.
  + case 3): Finally if both variables are of the same group, then we simply perform the division between their weights as the result.

class Solution {

public double[] calcEquation(List<List<String>> equations, double[] values,

List<List<String>> queries) {

HashMap<String, Pair<String, Double>> gidWeight = new HashMap<>();

// Step 1). build the union groups

for (int i = 0; i < equations.size(); i++) {

List<String> equation = equations.get(i);

String dividend = equation.get(0), divisor = equation.get(1);

double quotient = values[i];

union(gidWeight, dividend, divisor, quotient);

}

// Step 2). run the evaluation, with "lazy" updates in find() function

double[] results = new double[queries.size()];

for (int i = 0; i < queries.size(); i++) {

List<String> query = queries.get(i);

String dividend = query.get(0), divisor = query.get(1);

if (!gidWeight.containsKey(dividend) || !gidWeight.containsKey(divisor))

// case 1). at least one variable did not appear before

results[i] = -1.0;

else {

Pair<String, Double> dividendEntry = find(gidWeight, dividend);

Pair<String, Double> divisorEntry = find(gidWeight, divisor);

String dividendGid = dividendEntry.getKey();

String divisorGid = divisorEntry.getKey();

Double dividendWeight = dividendEntry.getValue();

Double divisorWeight = divisorEntry.getValue();

if (!dividendGid.equals(divisorGid))

// case 2). the variables do not belong to the same chain/group

results[i] = -1.0;

else

// case 3). there is a chain/path between the variables

results[i] = dividendWeight / divisorWeight;

}

}

return results;

}

private Pair<String, Double> find(HashMap<String, Pair<String, Double>> gidWeight, String nodeId) {

if (!gidWeight.containsKey(nodeId))

gidWeight.put(nodeId, new Pair<String, Double>(nodeId, 1.0));

Pair<String, Double> entry = gidWeight.get(nodeId);

// found inconsistency, trigger chain update

if (!entry.getKey().equals(nodeId)) {

Pair<String, Double> newEntry = find(gidWeight, entry.getKey());

gidWeight.put(nodeId, new Pair<String, Double>(

newEntry.getKey(), entry.getValue() \* newEntry.getValue()));

}

return gidWeight.get(nodeId);

}

private void union(HashMap<String, Pair<String, Double>> gidWeight, String dividend, String divisor, Double value) {

Pair<String, Double> dividendEntry = find(gidWeight, dividend);

Pair<String, Double> divisorEntry = find(gidWeight, divisor);

String dividendGid = dividendEntry.getKey();

String divisorGid = divisorEntry.getKey();

Double dividendWeight = dividendEntry.getValue();

Double divisorWeight = divisorEntry.getValue();

// merge the two groups together,

// by attaching the dividend group to the one of divisor

if (!dividendGid.equals(divisorGid)) {

gidWeight.put(dividendGid, new Pair<String, Double>(divisorGid,

divisorWeight \* value / dividendWeight));

}

}

}

**Complexity Analysis**

Since we applied the Union-Find data structure in our algorithm, we would like to start with a statement on the time complexity of the data structure, as follows:

**Statement**: If M*M* operations, either Union or Find, are applied to N*N* elements, the total run time is \mathcal{O}(M \cdot \log^{\*}{N})O(*M*⋅log∗*N*), where \log^{\*}log∗ is the [iterated logarithm](https://en.wikipedia.org/wiki/Iterated_logarithm).

One can refer to the [proof of Union-Find complexity](https://en.wikipedia.org/wiki/Proof_of_O(log*n)_time_complexity_of_union%E2%80%93find) for more details.

In our case, the maximum number of elements in the Union-Find data structure is equal to twice of the number of equations, i.e. each equation introduces two new variables.

Let N*N* be the number of input equations and M*M* be the number of queries.

* Time Complexity: \mathcal{O}\big( (M+N) \cdot \log^{\*}N\big)O((*M*+*N*)⋅log∗*N*).
  + First of all, we iterate through each input equation and invoke union() upon it. As a result, the overall time complexity of this step is \mathcal{O}\big(N \cdot \log^{\*}N\big)O(*N*⋅log∗*N*).
  + As the second step, we then evaluate the query one by one. For each evaluation, we invoke the find() function at most twice. Therefore, the overall time complexity of this step is \mathcal{O}\big(M \cdot \log^{\*}N\big)O(*M*⋅log∗*N*).
  + To sum up, the total time complexity of the algorithm is \mathcal{O}\big( (M+N) \cdot \log^{\*}N\big)O((*M*+*N*)⋅log∗*N*).
  + Note, as compared to the DFS/BFS search approach, Union-Find data structure is more **efficient** for the repetitive/redundant query scenario.
  + Once we evaluate a query with Union-Find, all the subsequent repetitive queries or any query that has the overlapping with the previous query in terms of variable group could be evaluated in **constant time**. For instance, in the above example, once the query of \frac{a}{c}*ca*​ is evaluated, if later we want to evaluate \frac{a}{b}*ba*​, we could instantly obtain the states of variables a and b without triggering the chain update again. While for DFS/BFS approaches, the cost of evaluating each query is independent for each other.
* Space Complexity: \mathcal{O}(N)O(*N*)
  + The space complexity of our Union-Find data structure is \mathcal{O}(N)O(*N*) where we maintain a state for each variable.
  + Since the find() function is implemented with recursion, there would be some additional memory consumption in function call stack, which could amount to \mathcal{O}(N)O(*N*).
  + To sum up, the total space complexity of the algorithm is \mathcal{O}(N) + \mathcal{O}(N) = \mathcal{O}(N)O(*N*)+O(*N*)=O(*N*).
  + Again, we did not take into account the space needed to hold the results. Otherwise, the total space complexity would be \mathcal{O}(N + M)O(*N*+*M*).