<https://leetcode.com/problems/evaluate-division/description/>

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.

Note: The variables that do not occur in the list of equations are undefined, so the answer cannot be determined for them.

**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 ]

note: x is undefined => -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.

**Attempt 1: 2024-12-01**

**Solution 1: Directed Graph (120 min)**

**Style 1: DFS method parameter contains 'distance'**

class Edge {

    String to;

    double weight;

    public Edge(String to, double weight) {

        this.to = to;

        this.weight = weight;

    }

}

class Solution {

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

        Map<String, List<Edge>> graph = buildGraph(equations, values);

        int len = queries.size();

        double[] result = new double[len];

        for(int i = 0; i < len; i++) {

            result[i] = getPathWeight(queries.get(i).get(0), queries.get(i).get(1), 1.0, graph, new HashSet<String>());

        }

        return result;

    }

    private double getPathWeight(String start, String end, double distance, Map<String, List<Edge>> graph, Set<String> visited) {

        if(!graph.containsKey(start)) {

            return -1.0;

        }

// Base cases

        // Found the target

        if(start.equals(end)) {

            return distance;

        }

        if(!visited.contains(start)) {

            visited.add(start);

            for(Edge e : graph.get(start)) {

                double result = getPathWeight(e.to, end, distance \* e.weight, graph, visited);

// Valid path found

                if(result != -1.0) {

                    return result;

                }

            }

        }

// No valid path

        return -1.0;

    }

    private Map<String, List<Edge>> buildGraph(List<List<String>> equations, double[] values) {

        Map<String, List<Edge>> graph = new HashMap<>();

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

            String u = equations.get(i).get(0);

            String v = equations.get(i).get(1);

            graph.putIfAbsent(u, new ArrayList<>());

            graph.putIfAbsent(v, new ArrayList<>());

            graph.get(u).add(new Edge(v, values[i]));

            graph.get(v).add(new Edge(u, 1.0 / values[i]));

        }

        return graph;

    }

}

Time Complexity:

Graph construction: O(E), where E is the number of equations.

Query processing: Each DFS is O(V + E), where V is the number of variables (nodes) and E is the number of edges.

Total: O(E + Q ⋅ (V + E)), where Q is the number of queries.

Space Complexity:

Graph storage: O(E).

DFS stack and visited set: O(V).

**Refer to**

<https://leetcode.com/problems/evaluate-division/solutions/171649/1ms-dfs-with-explanations/>

Binary relationship is represented as a graph usually.

Does the direction of an edge matters? -- Yes. Take a / b = 2 for example, it indicates a --2--> b as well as b --1/2--> a.

Thus, it is a directed weighted graph.

In this graph, how do we evaluate division?

Take a / b = 2, b / c = 3, a / c = ? for example,

a --2--> b --3--> c

We simply find a path using DFS from node a to node c and multiply the weights of edges passed, i.e. 2 \* 3 = 6.

Please note that during DFS,

Rejection case should be checked before accepting case.

Accepting case is (graph.get(u).containsKey(v)) rather than (u.equals(v)) for it takes O(1) but (u.equals(v)) takes O(n) for n is the length of the longer one between u and v.

public double[] calcEquation(String[][] equations, double[] values, String[][] queries) {

/\* Build graph. \*/

Map<String, Map<String, Double>> graph = buildGraph(equations, values);

double[] result = new double[queries.length];

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

result[i] = getPathWeight(queries[i][0], queries[i][1], new HashSet<>(), graph);

}

return result;

}

private double getPathWeight(String start, String end, Set<String> visited, Map<String, Map<String, Double>> graph) {

/\* Rejection case. \*/

if (!graph.containsKey(start))

return -1.0;

/\* Accepting case. \*/

if (graph.get(start).containsKey(end))

return graph.get(start).get(end);

visited.add(start);

for (Map.Entry<String, Double> neighbour : graph.get(start).entrySet()) {

if (!visited.contains(neighbour.getKey())) {

double productWeight = getPathWeight(neighbour.getKey(), end, visited, graph);

if (productWeight != -1.0)

return neighbour.getValue() \* productWeight;

}

}

return -1.0;

}

private Map<String, Map<String, Double>> buildGraph(String[][] equations, double[] values) {

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

String u, v;

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

u = equations[i][0];

v = equations[i][1];

graph.putIfAbsent(u, new HashMap<>());

graph.get(u).put(v, values[i]);

graph.putIfAbsent(v, new HashMap<>());

graph.get(v).put(u, 1 / values[i]);

}

return graph;

}

**Refer to**

<https://leetcode.com/problems/evaluate-division/solutions/88169/java-ac-solution-using-graph/comments/126499>

public double[] calcEquation(String[][] eq, double[] vals, String[][] q) {

Map<String, Map<String, Double>> m = new HashMap<>();

for (int i = 0; i < vals.length; i++) {

m.putIfAbsent(eq[i][0], new HashMap<>());

m.putIfAbsent(eq[i][1], new HashMap<>());

m.get(eq[i][0]).put(eq[i][1], vals[i]);

m.get(eq[i][1]).put(eq[i][0], 1 / vals[i]);

}

double[] r = new double[q.length];

for (int i = 0; i < q.length; i++)

r[i] = dfs(q[i][0], q[i][1], 1, m, new HashSet<>());

return r;

}

double dfs(String s, String t, double r, Map<String, Map<String, Double>> m, Set<String> seen) {

if (!m.containsKey(s) || !seen.add(s)) return -1;

if (s.equals(t)) return r;

Map<String, Double> next = m.get(s);

for (String c : next.keySet()) {

double result = dfs(c, t, r \* next.get(c), m, seen);

if (result != -1) return result;

}

return -1;

}

**Style 2: DFS method parameter doesn't contain 'distance'**

class Edge {

    String to;

    double weight;

    public Edge(String to, double weight) {

        this.to = to;

        this.weight = weight;

    }

}

class Solution {

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

        Map<String, List<Edge>> graph = buildGraph(equations, values);

        int len = queries.size();

        double[] result = new double[len];

        for(int i = 0; i < len; i++) {

            result[i] = getPathWeight(queries.get(i).get(0), queries.get(i).get(1), graph, new HashSet<String>());

        }

        return result;

    }

    private double getPathWeight(String start, String end, Map<String, List<Edge>> graph, Set<String> visited) {

        if(!graph.containsKey(start)) {

            return -1.0;

        }

        // Base cases

        // Found the target

        if(start.equals(end)) {

            return 1.0;

        }

        if(!visited.contains(start)) {

            visited.add(start);

            for(Edge e : graph.get(start)) {

                double result = getPathWeight(e.to, end, graph, visited);

                // Valid path found

                if(result != -1.0) {

                    return result \* e.weight;

                }

            }

        }

        // No valid path

        return -1.0;

    }

    private Map<String, List<Edge>> buildGraph(List<List<String>> equations, double[] values) {

        Map<String, List<Edge>> graph = new HashMap<>();

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

            String u = equations.get(i).get(0);

            String v = equations.get(i).get(1);

            graph.putIfAbsent(u, new ArrayList<>());

            graph.putIfAbsent(v, new ArrayList<>());

            graph.get(u).add(new Edge(v, values[i]));

            graph.get(v).add(new Edge(u, 1.0 / values[i]));

        }

        return graph;

    }

}

Time Complexity:

Graph construction: O(E), where E is the number of equations.

Query processing: Each DFS is O(V + E), where V is the number of variables (nodes) and E is the number of edges.

Total: O(E + Q ⋅ (V + E)), where Q is the number of queries.

Space Complexity:

Graph storage: O(E).

DFS stack and visited set: O(V).

**Refer to chatGPT**

**Solution Approach**

We can model this problem as a graph:

Each variable is a node.

Each equation represents a directed edge with a weight equal to the division value.

For example, a / b = 2.0 becomes an edge from a to b with weight 2.0 and an edge from b to a with weight 1/2.0.

To evaluate queries, we perform a graph traversal (DFS or BFS) to find a path from the numerator to the denominator and calculate the product of weights along the path.

import java.util.\*;

class Solution {

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

// Step 1: Build the graph

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

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

String a = equations.get(i).get(0);

String b = equations.get(i).get(1);

double value = values[i];

graph.putIfAbsent(a, new HashMap<>());

graph.putIfAbsent(b, new HashMap<>());

graph.get(a).put(b, value);

graph.get(b).put(a, 1.0 / value);

}

// Step 2: Process each query

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

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

String start = queries.get(i).get(0);

String end = queries.get(i).get(1);

if (!graph.containsKey(start) || !graph.containsKey(end)) {

results[i] = -1.0; // If either variable is not in the graph

} else {

results[i] = dfs(graph, start, end, new HashSet<>());

}

}

return results;

}

private double dfs(Map<String, Map<String, Double>> graph, String current, String target, Set<String> visited) {

// Base cases

if (current.equals(target)) return 1.0; // Found the target

visited.add(current);

// Explore neighbors

for (Map.Entry<String, Double> neighbor : graph.get(current).entrySet()) {

String next = neighbor.getKey();

double weight = neighbor.getValue();

if (!visited.contains(next)) {

double result = dfs(graph, next, target, visited);

if (result != -1.0) { // Valid path found

return result \* weight;

}

}

}

return -1.0; // No valid path

}

}

**Explanation**

Graph Construction:

Use a Map<String, Map<String, Double>> to represent the graph.

Each key is a node, and the value is a map of neighbors with edge weights.

DFS for Query Evaluation:

For each query (a, b), use DFS to find a path from a to b.

Track visited nodes with a Set<String> to avoid infinite loops.

Query Results:

If no path exists or either node is missing, return -1.0.

Otherwise, return the product of edge weights along the path.

**Complexity Analysis**

**Time Complexity:**

Graph construction: O(E), where E is the number of equations.

Query processing: Each DFS is O(V + E), where V is the number of variables (nodes) and E is the number of edges.

Total: O(E + Q ⋅ (V + E)), where Q is the number of queries.

**Space Complexity:**

Graph storage: O(E).

DFS stack and visited set: O(V).

**Test code**

import java.util.\*;

public class Solution {

class Edge {

String to;

double weight;

public Edge(String to, double weight) {

this.to = to;

this.weight = weight;

}

}

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

Map<String, List<Edge>> graph = buildGraph(equations, values);

int len = queries.size();

double[] result = new double[len];

for(int i = 0; i < len; i++) {

result[i] = getPathWeight(queries.get(i).get(0), queries.get(i).get(1), 1.0, graph, new HashSet<String>());

}

return result;

}

private double getPathWeight(String start, String end, double distance, Map<String, List<Edge>> graph, Set<String> visited) {

if(!graph.containsKey(start)) {

return -1.0;

}

if(start.equals(end)) {

return distance;

}

if(!visited.contains(start)) {

visited.add(start);

for(Edge e : graph.get(start)) {

double result = getPathWeight(e.to, end, distance \* e.weight, graph, visited);

if(result != -1.0) {

return result;

}

}

}

return -1.0;

}

private Map<String, List<Edge>> buildGraph(List<List<String>> equations, double[] values) {

Map<String, List<Edge>> graph = new HashMap<>();

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

String u = equations.get(i).get(0);

String v = equations.get(i).get(1);

graph.putIfAbsent(u, new ArrayList<>());

graph.putIfAbsent(v, new ArrayList<>());

graph.get(u).add(new Edge(v, values[i]));

graph.get(v).add(new Edge(u, 1.0 / values[i]));

}

return graph;

}

public static void main(String[] args) {

Solution so = new Solution();

List<List<String>> equations = new ArrayList<>();

List<String> e1 = new ArrayList<>();

e1.add("a");

e1.add("b");

List<String> e2 = new ArrayList<>();

e2.add("b");

e2.add("c");

equations.add(e1);

equations.add(e2);

double[] values = new double[] {2.0, 3.0};

List<List<String>> queries = new ArrayList<>();

List<String> q1 = new ArrayList<>();

q1.add("a");

q1.add("c");

List<String> q2 = new ArrayList<>();

q2.add("b");

q2.add("a");

List<String> q3 = new ArrayList<>();

q3.add("a");

q3.add("e");

List<String> q4 = new ArrayList<>();

q4.add("a");

q4.add("a");

List<String> q5 = new ArrayList<>();

q5.add("x");

q5.add("x");

queries.add(q1);

queries.add(q2);

queries.add(q3);

queries.add(q4);

queries.add(q5);

double[] result = so.calcEquation(equations, values, queries);

System.out.println(result);

}

}