<https://github.com/doocs/leetcode/blob/main/solution/1100-1199/1152.Analyze%20User%20Website%20Visit%20Pattern/README_EN.md>

You are given two string arrays username and website and an integer array timestamp. All the given arrays are of the same length and the tuple [username[i], website[i], timestamp[i]] indicates that the user username[i] visited the website website[i] at time timestamp[i].

A **pattern** is a list of three websites (not necessarily distinct).

For example, ["home", "away", "love"], ["leetcode", "love", "leetcode"], and ["luffy", "luffy", "luffy"] are all patterns.

The **score** of a **pattern** is the number of users that visited all the websites in the pattern in the same order they appeared in the pattern.

For example, if the pattern is ["home", "away", "love"], the score is the number of users x such that x visited "home" then visited "away" and visited "love" after that.

Similarly, if the pattern is ["leetcode", "love", "leetcode"], the score is the number of users x such that x visited "leetcode" then visited "love" and visited "leetcode" one more time after that.

Also, if the pattern is ["luffy", "luffy", "luffy"], the score is the number of users x such that x visited "luffy" three different times at different timestamps.

Return ***the pattern with the largest score***. If there is more than one pattern with the same largest score, return the lexicographically smallest such pattern.

**Example 1:**

Input:

username = ["joe","joe","joe","james","james","james","james","mary","mary","mary"],

timestamp = [1,2,3,4,5,6,7,8,9,10],

website = ["home","about","career","home","cart","maps","home","home","about","career"]

Output: ["home","about","career"]

Explanation:

The tuples in this example are:

["joe", 1, "home"]

["joe", 2, "about"]

["joe", 3, "career"]

["james", 4, "home"]

["james", 5, "cart"]

["james", 6, "maps"]

["james", 7, "home"]

["mary", 8, "home"]

["mary", 9, "about"]

["mary", 10, "career"]

The 3-sequence ("home", "about", "career") was visited at least once by 2 users.

The 3-sequence ("home", "cart", "maps") was visited at least once by 1 user.

The 3-sequence ("home", "cart", "home") was visited at least once by 1 user.

The 3-sequence ("home", "maps", "home") was visited at least once by 1 user.

The 3-sequence ("cart", "maps", "home") was visited at least once by 1 user.

**Example 2:**

Input:

username = ["ua","ua","ua","ub","ub","ub"],

timestamp = [1,2,3,4,5,6],

website = ["a","b","a","a","b","c"]

Output: ["a","b","a"]

Explanation:

The tuples in this example are:

["ua",1,"a"]

["ua",2,"b"]

["ua",3,"a"]

["ub",4,"a"]

["ub",5,"b"]

["ub",6,"c"]

The 3-sequence ("a", "b", "a") was visited at least once by 1 user.

The 3-sequence ("a", "b", "c") was visited at least once by 1 user.

Since both visited by 1 user, the lexicographically smallest pattern is ("a", "b", "a")

**Constraints:**

3 <= username.length <= 50

1 <= username[i].length <= 10

timestamp.length == username.length

1 <= timestamp[i] <= 10^9

website.length == username.length

1 <= website[i].length <= 10

username[i] and website[i] consist of lowercase English letters.

**It is guaranteed that there is at least one user who visited at least three websites.**

All the tuples [username[i], timestamp[i], website[i]] are unique.

**Attempt 1: 2024-7-16**

**Solution 1: Hash Table + Sorting (30min)**

import java.util.\*;

public class Solution {

class Visit {

int timestamp;

String website;

Visit(int timestamp, String website) {

this.timestamp = timestamp;

this.website = website;

}

}

public List<String> mostVisitedPattern(String[] username, int[] timestamp, String[] website) {

// Step 1: Group the website visits by user

Map<String, List<Visit>> userVisits = new HashMap<>();

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

userVisits.putIfAbsent(username[i], new ArrayList<>());

userVisits.get(username[i]).add(new Visit(timestamp[i], website[i]));

}

// Step 2: Generate 3-sequences for each user and count their occurrences

Map<String, Integer> sequenceCount = new HashMap<>();

for (String user : userVisits.keySet()) {

List<Visit> visits = userVisits.get(user);

// It is guaranteed that there is at least one user who visited at least three websites

Collections.sort(visits, (a, b) -> a.timestamp - b.timestamp);

Set<String> seenSequences = new HashSet<>();

// Iterate through combinations to form unique 3-sequence keys for this user

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

for (int j = i + 1; j < visits.size(); j++) {

for (int k = j + 1; k < visits.size(); k++) {

String sequence = visits.get(i).website + "," + visits.get(j).website + "," + visits.get(k).website;

if (!seenSequences.contains(sequence)) {

sequenceCount.put(sequence, sequenceCount.getOrDefault(sequence, 0) + 1);

seenSequences.add(sequence);

}

}

}

}

}

// Step 3: Find the most frequent sequence

String mostVisited = "";

int maxCount = 0;

for (String sequence : sequenceCount.keySet()) {

int count = sequenceCount.get(sequence);

if (count > maxCount || (count == maxCount && sequence.compareTo(mostVisited) < 0)) {

mostVisited = sequence;

maxCount = count;

}

}

// Step 4: Convert the most visited sequence to a list

return Arrays.asList(mostVisited.split(","));

}

public static void main(String[] args) {

Solution so = new Solution();

//String[] username = new String[]{"ua","ua","ua","ub","ub","ub"};

//int[] timestamp = new int[]{1,2,3,4,5,6};

//String[] website = new String[]{"a","b","a","a","b","c"};

String[] username = new String[]{"joe","joe","joe","james","james","james","james","mary","mary","mary"};

int[] timestamp = new int[]{1,2,3,4,5,6,7,8,9,10};

String[] website = new String[]{"home","about","career","home","cart","maps","home","home","about","career"};

List<String> result = so.mostVisitedPattern(username, timestamp, website);

System.out.println(result);

}

}

Time Complexity: O(n^3 log n)

Space Complexity: O(n^3)

**Refer to chatGPT**

**Problem Breakdown**

Store the Visits: Store each visit along with its timestamp for each user.

Generate Sequences: Generate all possible 3-sequence combinations for each user.

Count Sequences: Count the occurrences of each 3-sequence across all users.

Find the Most Frequent Sequence: Determine the most frequent 3-sequence. If there are ties, return the lexicographically smallest one.

**Explanation**

Grouping Visits:

We store each visit in a map where the key is the username and the value is a list of Visit objects containing timestamp and website information.

Sorting Visits:

For each user, sort their visits by timestamp to ensure the correct sequence of websites.

Generating and Counting Sequences:

For each user, generate all possible sequences of three websites. Use a set to avoid counting duplicate sequences for the same user multiple times.

Store and update the count of each 3-sequence in a map.

Finding the Most Frequent Sequence:

Traverse the map to find the sequence with the highest count. In case of ties, choose the lexicographically smallest sequence.

Returning the Result:

Split the most frequent sequence by commas to return it as a list of strings.

**Complexity Analysis**

Time Complexity: O(n^3), where n is the number of websites a user visits. This is due to generating all possible 3-sequences for each user.

Space Complexity: O(n), for storing the visits and sequences.

This solution ensures that we accurately find and return the most frequent 3-sequence of website visits while adhering to the problem constraints.

**Refer to**

<https://algo.monster/liteproblems/1152>

**Problem Description**

In this problem, we are given the browsing history of multiple users stored in three arrays: username, timestamp, and website. Each index i in these arrays represents a single browsing event, where username[i] is the user who visited website[i] at time timestamp[i]. The goal is to find a sequence of three websites, which we will refer to as a "pattern," that is most popular among the users. A pattern here means any three websites visited in a specific order, not necessarily different. The popularity or "score" of a pattern is determined by counting how many unique users visited the websites in the same order as the pattern. The task is to return the pattern with the highest score, and in case of a tie, return the lexicographically smallest pattern.

**Intuition**

The solution to this problem involves several steps. First, we need to organize the browsing events for each user in chronological order. This is essential because we are interested in patterns which depend on the sequence of website visits.

To accomplish this, we start by combining the given arrays into a single list of tuples, each consisting of a username, a timestamp, and a website. We then sort this list by timestamp to ensure that each user's visits are in the correct order.

With the data sorted, we can now create a mapping from each user to their list of visited websites in chronological order. This is done using a dictionary, where each key is a username, and the corresponding value is the list of websites they visited.

We then calculate the score for every possible pattern. To ensure we count each pattern only once per user, we use a set to collect all unique patterns visited by a user. After processing all users' data, we will have the total count of unique users for each pattern across all users.

Finally, we sort the patterns by their score in descending order and lexicographically. The first pattern in this sorted list is our answer, as it has the highest score, and in the case of a tie, it is the lexicographically smallest among those with the highest scores.

The provided solution performs these steps and correctly identifies the most visited pattern as required.

**Solution Approach**

The solution uses a combination of sorting, dictionary usage, sets, and the Counter class from the collections module in Python to implement an efficient approach toward finding the most visited pattern. Here's a step-by-step breakdown:

Sorting by Timestamp: First, we create a list of tuples with the username, timestamp, and website. We sort this list based on the timestamp to ensure that we consider the websites in the order they were visited by each user. This is achieved by the expression:

sorted(zip(username, timestamp, website), key=lambda x: x[1])

Mapping Users to Websites: We use a defaultdict of lists from the collections module to map each user to their visited websites in chronological order. This data structure is useful for aggregating the list of website visits per user without having to check if the user key already exists in the dictionary. We do this by:

d = defaultdict(list)

for user, \_, site in sorted\_events:

d[user].append(site)

Generating Patterns: As we're interested in patterns of three websites, we iterate through each user's websites to generate all possible patterns using a nested loop structure. Since we want unique patterns per user, a set named s is used to store these combinations without duplicates:

s = set()

for i in range(m - 2):

for j in range(i + 1, m - 1):

for k in range(j + 1, m):

s.add((sites[i], sites[j], sites[k]))

Counting Patterns' Scores: We use a Counter object to keep track of how many times each pattern occurs across all users. For each user, we add the patterns from the set s to the counter, which increments the count for each pattern:

cnt = Counter()

for t in s:

cnt[t] += 1

Sorting and Selecting the Top Pattern: Finally, we sort the items in the counter to find the most visited pattern. The sort criteria are the pattern's score in descending order and the pattern itself in lexicographic order for tie-breaking. The top pattern is then selected as the answer:

sorted(cnt.items(), key=lambda x: (-x[1], x[0]))[0][0]

The combination of sorting, dictionary and set usage to eliminate duplicates, and a frequency counter (Counter class) makes this approach efficient and effective for solving the problem.

**Example Walkthrough**

Let's illustrate the solution approach with a small example.

Suppose we have the following input data:

username: ["Jane", "Jane", "Jane", "Alex", "Alex", "Alex"]

timestamp: [1, 2, 3, 4, 5, 6]

website: ["A", "B", "C", "A", "B", "D"]

Following the algorithm's steps:

Sorting by Timestamp: We combine username, timestamp, and website into a list of tuples and sort it. The sorted result will be:

[("Jane", 1, "A"), ("Jane", 2, "B"), ("Jane", 3, "C"), ("Alex", 4, "A"), ("Alex", 5, "B"), ("Alex", 6, "D")]

Mapping Users to Websites: We create a dictionary (defaultdict) to map each user to the list of websites they visited in order:

d["Jane"] = ["A", "B", "C"]

d["Alex"] = ["A", "B", "D"]

Generating Patterns: We generate patterns for each user. For "Jane", the pattern would be:

s["Jane"] = {("A", "B", "C")}

And for "Alex":

s["Alex"] = {("A", "B", "D")}

Note that both users have the pattern ("A", "B") in common but have different third websites visited.

Counting Patterns' Scores: We count the occurrences of patterns across all users. For this example:

cnt[("A", "B", "C")] = 1

cnt[("A", "B", "D")] = 1

Each pattern has been visited by one unique user.

Sorting and Selecting the Top Pattern: We sort patterns based on their scores and lexicographically. After sorting:

patterns\_sorted = [("A", "B", "C"), ("A", "B", "D")]

Since the scores are the same, we pick the lexicographically smallest pattern, which is ("A", "B", "C").

Hence, the most visited pattern in this example would be ("A", "B", "C"), as it is the lexicographically smallest among the highest scoring patterns.

**Solution Implementation**

import java.util.\*;

// Definition of Node as a custom data structure to hold user visit information.

class Node {

String user;

int timestamp;

String website;

Node(String user, int timestamp, String website) {

this.user = user;

this.timestamp = timestamp;

this.website = website;

}

}

class Solution {

public List<String> mostVisitedPattern(String[] usernames, int[] timestamps, String[] websites) {

// Map to hold data for each user and their list of Node objects (timestamps and websites visited).

Map<String, List<Node>> userData = new HashMap<>();

int visitCount = usernames.length; // Total number of website visits

// Constructing the user data map from usernames, timestamps, and websites.

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

String user = usernames[i];

int ts = timestamps[i];

String site = websites[i];

userData.computeIfAbsent(user, k -> new ArrayList<>()).add(new Node(user, ts, site));

}

// Map to hold the count of each unique 3-sequence pattern.

Map<String, Integer> patternFrequency = new HashMap<>();

// Process each user's site visit history to calculate the pattern frequency.

for (List<Node> sites : userData.values()) {

int visitSize = sites.size();

Set<String> sequences = new HashSet<>();

// Check if there are at least 3 sites visited to form a valid pattern.

if (visitSize > 2) {

// Sort the user's visit by timestamp

Collections.sort(sites, (a, b) -> a.timestamp - b.timestamp);

// Iterate through combinations to form unique 3-sequence keys for this user

for (int i = 0; i < visitSize - 2; ++i) {

for (int j = i + 1; j < visitSize - 1; ++j) {

for (int k = j + 1; k < visitSize; ++k) {

sequences.add(sites.get(i).website + "," +

sites.get(j).website + "," +

sites.get(k).website);

}

}

}

}

// Count frequency of each 3-sequence pattern.

for (String seq : sequences) {

patternFrequency.put(seq, patternFrequency.getOrDefault(seq, 0) + 1);

}

}

// Variables to track the maximum frequency and the corresponding pattern.

int maxFrequency = 0;

String topPattern = "";

// Iterate over the pattern frequencies to determine the most visited pattern.

for (Map.Entry<String, Integer> entry : patternFrequency.entrySet()) {

// Compare the count or lexicographical order if counts are equal.

if (maxFrequency < entry.getValue() ||

(maxFrequency == entry.getValue() && entry.getKey().compareTo(topPattern) < 0)) {

maxFrequency = entry.getValue();

topPattern = entry.getKey();

}

}

// Return the top pattern as a list of sites.

return Arrays.asList(topPattern.split(","));

}

}

**Time and Space Complexity**

**Time Complexity**

The time complexity can be split into a few different parts:

Sorting the combined list of (username, timestamp, website) by timestamp. If we say n is the number of events, this operation has a complexity of O(n log n).

Constructing the dictionary (d). The for-loop runs through each of the n events once, so this step has a complexity of O(n).

Constructing the 3-sequence set and counting. For each user, we generate all possible combinations of website visit sequences of length 3. In the worst case, every user visits m websites, and the number of possible combinations is O((m choose 3)), which is O(m^3/6) when expanded. Summing this up for all users, if u is the number of users, this step would have complexity of O(u \* m^3/6). However, since m = n in the worst case (a single user visiting all websites), it simplifies to O(u \* n^3/6).

Sorting the count dictionary cnt items which, in the worst case, can have as many as O(n^3/6) unique sequences. The complexity of this sort is O((n^3/6) log (n^3/6)), which simplifies to O(n^3 log n).

Considering all steps, the overall time complexity would be dominated by O(n log n) + O(n) + O(u \* n^3/6) + O(n^3 log n) which simplifies to O(u \* n^3/6) + O(n^3 log n) due to the higher powers of n.

**Space Complexity**

For space complexity, we consider the data structures used:

The dictionary d with the list of sites for each user. In the worst case, this can store up to n elements, giving O(n).

The set s which, in the worst case, can contain O(n^3/6) different sequences per user (again considering that a user visits all websites). Therefore, across all users, this could potentially be O(u \* n^3/6).

The counter cnt, which will hold the same number of unique sequences as in set s, so it also has a possible space complexity of O(u \* n^3/6).

Combining these observations, the overall space complexity would also be dictated by the counter and set, giving an upper bound of O(u \* n^3/6). If u is not significantly large compared to n, we can approximate the space complexity to O(n^3) to reflect the worst-case scenario.