1817. Finding the Users Active Minutes

Hash Table

Problem Description

<u>Array</u>

Medium

element represents a user's action with two pieces of information: the user's ID and the time the action occurred, recorded by the minute. We are instructed to compute a user's active minutes (UAM), which are the unique minutes during which they performed any actions. Importantly, even if a user performs multiple actions within a single minute, that minute is only counted once for their UAM. Our goal is to create an array of size k, where each entry j indicates the number of users whose UAM is exactly j. To clarify, the

In this problem, we are dealing with logs of user actions on LeetCode. The logs are given as a 2-dimensional array where each

array we are constructing is 1-indexed, meaning that the index of each item starts at 1, not 0. Therefore, for every number of active minutes that a user could have (from 1 to k), we want to count how many users have that amount of UAM.

The natural approach to solve this problem relies on representing each user's actions without duplications and counting their active minutes efficiently. Since a user can perform many actions, possibly at the same times, we need a way to store these

Intuition

actions such that we can easily determine the unique minutes. Using a hash table, or in Python, a dictionary that maps each user ID to a set of unique times, provides an effective solution. Sets

are useful here because they naturally eliminate duplicate entries, which aligns perfectly with counting unique active minutes. After populating the hash table with each user's unique active minutes, we then need to determine the number of users for each possible UAM value up to k. We accomplish this by initiating an answer array ans filled with zeroes to correspond to counts for

each UAM from 1 to k. We loop through the hash table and increment the count in ans at the index that corresponds to the user's UAM (adjusted by subtracting 1 for 1-indexing). The reason behind this approach is two-fold:

2. By using an array indexed by active minutes, we can simply tally up the number of users for each UAM, which the problem requires us to report.

Here's the detailed implementation flow:

Solution Approach The solution's strategy is based on the hash table data structure which is very common for storing relationships between key-

value pairs. In Python, this is typically implemented using a defaultdict which is a subclass of the dictionary (dict). A

1. By mapping users to sets of timestamps, we efficiently record user activity without duplication, directly giving us the UAM for each user.

defaultdict allows us to specify a default value for the dictionary – in this case, a set. The set is chosen as the default value because we are interested in keeping a collection of unique active minutes for each user.

a set that contains all unique minutes during which the user performed actions. Populate the defaultdict: • Iterate over each entry in the logs array.

Initialize the defaultdict: We create d as a defaultdict of sets. Each user ID will be a key, and the corresponding value will be

• For each [user_id, timestamp] pair, add timestamp to the set mapped to by user_id. The set ensures that if the same timestamp is inserted

multiple times, it will only be stored once, preserving the unique nature of the active minutes.

•

array.

- Initiate the ans array: We create an array (list in Python) called ans with length k, filled with zeros. Each index i in this array represents the number of users whose UAM is i + 1, because the problem asks for a 1-indexed array.
- Loop through the sets in the defaultdict values, which each represent the set of unique active minutes for a particular user ID. • The size of each set is the UAM count for that user. Subtracting 1 from the set size gives the index that corresponds to that UAM in the ans
- By following this solution approach, we effectively tabulate the distribution of UAMs across all users. Finally, we return the

for i, t in logs:

return ans

Example Walkthrough

Increment the value at the calculated index in ans by 1.

d = defaultdict(set) # Create defaultdict of sets

Count the UAM for each user:

populated ans array as our result, which contains the count of users for each UAM value from 1 to k. Here's an illustration of the reference solution approach in a concise form: class Solution:

d[i].add(t) # Populate the defaultdict ans = [0] * k # Initialize ans list for ts in d.values(): ans[len(ts) - 1] += 1 # Count UAM and populate ans

```
logs = [[1, 1], [2, 2], [2, 2], [2, 3], [3, 4], [3, 4], [3, 4], [3, 5]], k = 3
  With this data, we'll walk through the implementation steps:
Initialize the defaultdict
  We create a defaultdict named d. Each key will be unique user IDs, and each value will be a set of unique timestamps:
```

We iterate through each log entry and update d with unique active minutes for each user:

• Lastly, user 3 has entries at minutes 4 and 5, all duplicates, so user 3's set will only contain {4, 5}.

Let's consider an example to illustrate the solution approach with the logs array and k value as follows:

def findingUsersActiveMinutes(self, logs: List[List[int]], k: int) -> List[int]:

• First log entry is [1, 1]—we add minute 1 to user 1's set. • Next, we process three log entries for user 2. They are all in minutes 2 and 3, resulting in user 2's set containing {2, 3}.

Initiate the ans array

each user:

[1, 2, 0]

Python

Java

class Solution {

d = defaultdict(set)

Populate the defaultdict

Initially, d will be empty.

*{*1: *{*1*}*, 2: *{*2, 3*}*, 3: *{*4, 5*}}*

K is 3, so we create an answer list ans of length k, with all elements initialized to 0:

• User 2 has a UAM count of 2, so we increment ans [1] to 1.

ans = [0] * 3 # Which is <math>[0, 0, 0]

User 1 has a UAM count of 1, so we increment ans [0] from 0 to 1 (because index 0 corresponds to UAM 1).

The final output of the reference solution for our given logs and k values would be [1, 2, 0].

Count the UAM for each user We'll iterate through the sets in defaultdict values and update the ans list based on the number of unique active minutes for

After processing, d looks like this:

```
• User 3 also has a UAM of 2, so we increase ans [1] again, from 1 to 2.
After processing, the ans array is:
```

Solution Implementation

This array means there is one user with a UAM of 1, two users with a UAM of 2, and zero users with a UAM of 3.

user_active_minutes[user_id].add(minute) # Initialize an answer array to store counts of users with X active minutes # where X is the index + 1

// If the user ID doesn't exist in the map, create a new HashSet for that user.

userActiveTimes.computeIfAbsent(userId, key -> new HashSet<>()).add(timestamp);

// Initialize an array to store the UAM count for each possible count 1 through k

def findingUsersActiveMinutes(self, logs: List[List[int]], k: int) -> List[int]:

Iterate through the logs, adding unique minutes to each user's set

Iterate through the user_active_minutes to update the answer array

public int[] findingUsersActiveMinutes(int[][] logs, int k) {

// A hashmap to store unique time stamps for each user

int userId = log[0]; // Extract the user ID

int timestamp = log[1]; // Extract the timestamp

Map<Integer, Set<Integer>> userActiveTimes = new HashMap<>();

// Then, add the timestamp to the user's set of active times.

Dictionary mapping user IDs to a set of their active minutes

```
for minutes in user_active_minutes.values():
   # Increase the count for the number of active minutes
   # We do -1 because the index is 0-based and minutes is 1-based
   answer[len(minutes) - 1] += 1
```

return answer

answer = [0] * k

from collections import defaultdict

user_active_minutes = defaultdict(set)

for user_id, minute in logs:

// Iterate over each log entry

// where x ranges from 1 to k.

return activeMinutesCounts;

vector<int> activeMinutesCounts(k, 0);

if (uniqueActiveMinutes <= k) {</pre>

// Return the resulting vector of counts.

for (const auto& userToTimeEntry : userToActiveMinutesMap) {

int uniqueActiveMinutes = userToTimeEntry.second.size();

activeMinutesCounts[uniqueActiveMinutes - 1]++;

// This function calculates the number of unique users who have exactly 1 to k active minutes.

// Logs are represented as an array of [userId, minute] and k is the upper limit of active minutes.

// The size of the set for each user gives the total number of unique active minutes.

for (int[] log : logs) {

from typing import List

class Solution:

```
int[] answer = new int[k];
       // Iterate over each user's set of timestamps
        for (Set<Integer> timeStamps : userActiveTimes.values()) {
           // Count the number of unique timestamps for the user and
           // increment the respective count in the answer array.
           // Subtract 1 from the size to convert the count into a zero-based index.
            answer[timeStamps.size() - 1]++;
       // Return the filled answer array
       return answer;
C++
#include <vector>
#include <unordered_map>
#include <unordered_set>
using namespace std;
class Solution {
public:
   // This function takes a vector of vector of integers which contain user ID and
   // login time, along with an integer k representing the total number of minutes.
   // It returns a vector with the count of users who have a specific number of
    // active minutes ranging from 1 to k.
    vector<int> findingUsersActiveMinutes(vector<vector<int>>& logs, int k) {
       // Create a map that associates each user (identified by their ID) with a set of unique active minutes.
       unordered_map<int, unordered_set<int>> userToActiveMinutesMap;
        for (const auto& logEntry : logs) {
            int userID = logEntry[0];
            int time = logEntry[1];
           // Insert the time into the set for this user, duplicates are automatically handled by the set.
            userToActiveMinutesMap[userID].insert(time);
       // Initialize a vector to store the counts of users having x number of active minutes,
```

// Increment the count for the appropriate number of active minutes (decrement by 1 to match 0-indexing).

```
function findingUsersActiveMinutes(logs: number[][], k: number): number[] {
   // Initialize a map to store unique minutes for each user.
   const userActivityMap: Map<number, Set<number>> = new Map();
```

TypeScript

};

```
// Iterate through all the log entries.
      for (const [userId, minute] of logs) {
          // If the user does not have a set yet, initialize it.
          if (!userActivityMap.has(userId)) {
              userActivityMap.set(userId, new Set<number>());
          // Add the minute to the set corresponding to the user.
          userActivityMap.get(userId)!.add(minute);
      // Initialize an array to count users with exactly 1 to k active minutes.
      const userCount: number[] = Array(k).fill(0);
      // Iterate through the map to count the number of active minutes for each user.
      for (const userMinutesSet of userActivityMap.values()) {
          if (userMinutesSet.size > 0 && userMinutesSet.size <= k) {</pre>
              // Increment the count for the number of users with a specific count of active minutes.
              // (size - 1) is used because indices are 0-based and our minute counts are 1-based.
              ++userCount[userMinutesSet.size - 1];
      // Return the array with counts of users having 1 to k active minutes.
      return userCount;
from collections import defaultdict
from typing import List
class Solution:
   def findingUsersActiveMinutes(self, logs: List[List[int]], k: int) -> List[int]:
       # Dictionary mapping user IDs to a set of their active minutes
        user_active_minutes = defaultdict(set)
       # Iterate through the logs, adding unique minutes to each user's set
        for user_id, minute in logs:
            user_active_minutes[user_id].add(minute)
       # Initialize an answer array to store counts of users with X active minutes
       # where X is the index + 1
       answer = [0] * k
       # Iterate through the user_active_minutes to update the answer array
        for minutes in user_active_minutes.values():
           # Increase the count for the number of active minutes
            # We do -1 because the index is 0-based and minutes is 1-based
            answer[len(minutes) - 1] += 1
       return answer
```

the logs array exactly once, adding the timestamp to a set in the dictionary. Since set operations such as ladd() have a time complexity of O(1), iterating over the logs array contributes O(n) to the time complexity.

Time and Space Complexity

Further, we also iterate over the values of the dictionary, with the number of values being at most n. For each of the values, we increment the appropriate count in the ans array, which is another 0(1) operation. Hence, the total time complexity remains 0(n). The space complexity of the code is also 0(n) because in the worst-case scenario, each log entry could be for a unique user and a unique timestamp, leading to n unique entries in the dictionary. As such, both the dictionary and the set of timestamps could

The time complexity of the code is O(n), where n is the length of the logs array. This is because we iterate over each element of

potentially grow linearly with the number of log entries. Therefore, the space used by the dictionary is proportional to the size of the logs array, n.