1604. Alert Using Same Key-Card Three or More Times in a One Hour Period

Medium | Array | Hash Table | String | Sorting   Leetcode Link

Problem Description

The problem provides data on the use of key-cards by LeetCode company workers for unlocking office doors. The security system logs the name of the worker and the time when the key-card was used. An alert is triggered if any worker uses the key-card three or more times within any one-hour period during a single day.

The task is to write a program that analyzes this data. You're given two lists: `keyName`, containing the names of the workers, and `keyTime`, containing the corresponding times when their key-cards were used. The program must output a list of unique names of workers who triggered an alert due to frequent use of their key-card within any given one-hour period. This output list should be sorted in alphabetical order.

Access times in `keyTime` are in a 24-hour format, denoted as "HH:MM". It is important to note that overlapping hours, such as from "10:00" to "11:00", are considered a one-hour period, but times from "22:51" to "23:52" are not since they span into a new one-hour period.

Intuition

To solve this problem, the primary goal is to track the usage frequency of key-cards by each worker within a one-hour period. Here, we can follow these steps as part of our intuition towards the solution:

1. Parse each time entry in `keyTime` from "HH:MM" format to minutes only. This makes it easier to calculate the difference between times.
2. Group all the time entries by worker names using a dictionary where the key is the worker's name and the value is a list of their corresponding times in minutes.
3. Iterate over each worker's list of time entries:
   - If a worker's list contains less than three time entries, no alert would be triggered for this worker, so they can be skipped.
   - Sort the list of times for each worker, because to find a period of one hour where the card was used at least three times, the times need to be in order.
   - Look for any three consecutive time entries (after sorting) within a 60-minute window. If there are such entries, add the worker's name to the list of those who triggered an alert.
4. Finally, sort the list of workers who triggered an alert in alphabetical order and return it.

This approach is efficient since it minimizes sorting to only the necessary times (those belonging to the same worker), instead of sorting all time entries globally. The lookup to check for the one-hour frequency is optimized by only comparing between three consecutive time entries due to the sorted nature of the lists.

Solution Approach

The implementation of the solution given applies several common programming patterns and data structures to efficiently address the problem. Let's break down how the provided Python solution accomplishes this task:

1. The `defaultdict` from Python's `collections` module is used to create a dictionary that will automatically generate a list for each new key. This data structure helps group all the key-card use times by worker name.

2. The solution iterates through the combined `keyName` and `keyTime` lists using `zip`, which allows us to process the corresponding elements of these two lists in pairs.

3. Each time in the `keyTime` list is converted from the "HH:MM" string format to a total number of minutes using integer arithmetic. This conversion allows for much simpler comparisons since we can now just subtract the times to get the difference in minutes:

   ```
   t = int(t[:2]) * 60 + int(t[3:])
   ```

   Here, `t[:2]` extracts the hours component and `t[3:]` extracts the minutes component from the string, which are then converted to an integer total number of minutes.

4. For each worker, we sort their list of time entries, necessary to find the sequence of three time entries within a one-hour window (60 minutes). The sorting assures us that no three consecutive time entries can be more than one hour apart if there is no such trio within the sorted list.

5. The algorithm then iterates over each sorted time list for the workers, only if there are at least three time entries (`n := len(ts)`). Python's Walrus operator `:=` is used for assignment and comparison in a single expression.

6. To check for a one-hour period alert, a window of three consecutive times `ts[i]`, `ts[i + 1]`, and `ts[i + 2]` is examined. If the difference between `ts[i + 2]` and `ts[i]` is less than or equal to 60 minutes, we confirm that there were at least three uses within a one-hour period:

   ```
   if ts[i + 2] - ts[i] <= 60:
   ```

   If this condition is met, the worker's name is added to the `ans` list, and the loop breaks since one alert is sufficient to flag the worker.

7. After all the workers have been checked, the list of workers who triggered an alert (`ans`) is sorted alphabetically to meet the specified output requirement.

8. The sorted list `ans` is returned, containing all the worker names that triggered an alert, sorted in ascending alphabetical order.

By using these steps, the algorithm is able to efficiently check each worker against the alert condition while maintaining the order of names as required for the output.

Example Walkthrough

Let's use a small example to illustrate the solution approach.

Suppose `keyName` and `keyTime` are as follows:

```
keyName = ["Alice", "Bob", "Alice", "Alice", "Bob"]
keyTime = ["10:00", "10:20", "11:00", "12:00", "10:40"]
```

1. Convert `keyTime` entries to minutes:

   ```
   ["10:00", "10:20", "11:00", "12:00", "10:40"]
   becomes
   [600, 620, 660, 720, 640]  # after conversion
   ```

2. Group by worker names using `defaultdict`:

   ```
   {"Alice": [600, 660, 720], "Bob": [620, 640]}
   ```

3. The times associated with Alice are [600, 660, 720], and for Bob, they are [620, 640]. Alice's times contain more than two entries, but Bob's do not, so Bob can be skipped.

4. Sort the times for each worker. In this case, Alice's times are already in order.

5. For Alice, check three consecutive time entries:

   ```
   Between 600, 660 - Not within a one-hour window (difference is 60 minutes).
   Between 660, 720 - Not within a one-hour window (difference is 60 minutes).
   ```

   Since there are no three consecutive entries within a 60-minute window for Alice, she does not trigger an alert.

6. No alerts are triggered for any worker, so the final step of sorting the `ans` list is trivial as it is empty.

Hence, the output would be an empty list `[]`, indicating that no worker triggered an alert in this example.

Python Solution

```python
from collections import defaultdict

class Solution:
    def alertNames(self, keyNames: List[str], keyTimes: List[str]) -> List[str]:
        # Create a dictionary to store times associated with each name
        name_times_dict = defaultdict(list)

        # Convert time strings to minutes and store them in the dictionary
        for name, time_str in zip(keyNames, keyTimes):
            time_minutes = int(time_str[:2]) * 60 + int(time_str[3:])
            name_times_dict[name].append(time_minutes)

        # Initialize a list to hold the names of individuals who should be alerted
        alert_names = []

        # Iterate through the dictionary to check for each person
        for name, times in name_times_dict.items():
            times_count = len(times)
            # Only proceed if there are more than two time points
            if times_count > 2:
                # Sort the times in ascending order
                times.sort()
                # Check for any three consecutive times within a 60 minutes window
                for i in range(times_count - 2):
                    if times[i + 2] - times[i] <= 60:
                        # If the condition is met, add the person's name to the alert list
                        alert_names.append(name)
                        break  # No need to check further for this person

        # Sort the list of names alphabetically before returning
        alert_names.sort()
        return alert_names
```

Java Solution

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Map;
import java.util.HashMap;

class Solution {
    public List<String> alertNames(String[] keyNames, String[] keyTimes) {
        // Map to track the time (converted to minutes) at which each keyName logs in
        Map<String, List<Integer>> nameToTimesMap = new HashMap<>();

        // Iterate over the input arrays to populate the nameToTimesMap
        for (int i = 0; i < keyNames.length; ++i) {
            String name = keyNames[i];
            String time = keyTimes[i];
            // Convert the time string to total minutes for easier comparison later
            int totalMinutes
                = Integer.parseInt(time.substring(0, 2)) * 60 + Integer.parseInt(time.substring(3));
            // If the name is not in the map, put it with a new ArrayList; add the time otherwise
            nameToTimesMap.computeIfAbsent(name, k -> new ArrayList<>()).add(totalMinutes);
        }

        // List to hold the names of individuals who should receive an alert
        List<String> alerts = new ArrayList<>();

        // Iterate through the map entries
        for (Map.Entry<String, List<Integer>> entry : nameToTimesMap.entrySet()) {
            List<Integer> times = entry.getValue();
            int n = times.size();
            // We only care about individuals with more than 2 time logs
            if (n > 2) {
                // Sort the times to check the sequence
                Collections.sort(times);
                // Go through the times to find a sequence of 3 within 60 minutes
                for (int i = 0; i < n - 2; ++i) {
                    if (times.get(i + 2) - times.get(i) <= 60) {
                        // If such a sequence is found, add the name to the alerts list and break
                        alerts.add(entry.getKey());
                        break;
                    }
                }
            }
        }

        // Sort the names alphabetically before returning
        Collections.sort(alerts);
        return alerts;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <unordered_map>
#include <algorithm>

class Solution {
public:
    vector<string> alertNames(vector<string>& keyName, vector<string>& keyTime) {
        // Map to store user names and their swipe times in minutes
        unordered_map<string, vector<int>> nameTimeMap;

        // Convert times into minutes and store them in the map
        for (size_t i = 0; i < keyName.size(); ++i) {
            string name = keyName[i];
            string time = keyTime[i];
            int hours, minutes;
            // Parse the time string to extract hours and minutes
            sscanf(time.c_str(), "%d:%d", &hours, &minutes);
            // Calculate total time in minutes
            int totalMinutes = hours * 60 + minutes;
            // Add the total time to the corresponding name in the map
            nameTimeMap[name].emplace_back(totalMinutes);
        }

        // List to hold the names of users who trigger the alert condition
        vector<string> alertNamesList;

        // Iterate over each entry in the name time map
        for (auto& [name, timeStamps] : nameTimeMap) {
            // Sort the timestamps for each user
            sort(timeStamps.begin(), timeStamps.end());
            // Check if the user has at least three timestamps
            if (timeStamps.size() > 2) {
                // Iterate through the timestamps
                for (size_t i = 0; i < timeStamps.size() - 2; ++i) {
                    // Check if three consecutive swipes are within 60 minutes
                    if (timeStamps[i + 2] - timeStamps[i] <= 60) {
                        // Add the name to the alert list and break the loop
                        alertNamesList.emplace_back(name);
                        break;
                    }
                }
            }
        }

        // Sort the list of names that trigger the alert condition
        sort(alertNamesList.begin(), alertNamesList.end());

        // Return the final list of alert names
        return alertNamesList;
    }
};
```

Typescript Solution

```typescript
// Represents a map from user names to an array of their swipe times in minutes
const nameTimeMap: { [name: string]: number[] } = {};

/**
 * Converts a time string to minutes.
 * @param time - A string representing the time in HH:MM format.
 * @returns The number of minutes since midnight.
 */
function convertTimeToMinutes(time: string): number {
    const [hours, minutes] = time.split(':').map(Number);
    return hours * 60 + minutes;
}

/**
 * Checks if the alert condition is met for a user.
 * @param timeStamps - An array of swipe times in minutes for a user.
 * @returns A boolean indicating if the alert condition is met.
 */
function checkAlertCondition(timeStamps: number[]): boolean {
    for (let i = 0; i < timeStamps.length - 2; i++) {
        if (timeStamps[i + 2] - timeStamps[i] <= 60) {
            return true;
        }
    }
    return false;
}

/**
 * Adds swipe times to the name-time mapping.
 * @param keyName - An array of key names.
 * @param keyTime - An array of corresponding key times.
 */
function processSwipeTimes(keyName: string[], keyTime: string[]): void {
    for (let i = 0; i < keyName.length; ++i) {
        const name = keyName[i];
        const time = keyTime[i];
        const totalMinutes = convertTimeToMinutes(time);
        if (!nameTimeMap[name]) {
            nameTimeMap[name] = [];
        }
        nameTimeMap[name].push(totalMinutes);
    }
}

/**
 * Generates a list of names who triggered the alert condition.
 * @param keyName - An array of key names.
 * @param keyTime - An array of corresponding key times.
 * @returns An array of names for those who have triggered the alert.
 */
function alertNames(keyName: string[], keyTime: string[]): string[] {
    processSwipeTimes(keyName, keyTime);

    const alertNamesList: string[] = [];

    // Iterate over each entry in the name-time map
    for (const name in nameTimeMap) {
        // Sort the timestamps for each user
        nameTimeMap[name].sort((a, b) => a - b);
        // Check if the alert condition is met for the user
        if (checkAlertCondition(nameTimeMap[name])) {
            alertNamesList.push(name);
        }
    }

    // Sort the list of names that triggered the alert condition
    alertNamesList.sort();

    return alertNamesList;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the provided code can be analyzed in the following steps:

1. **Zipping and transforming times:** The `zip(keyName, keyTime)` operation goes through all the keys and times once, which is `O(N)` where `N` is the number of entries in the `keyName` and `keyTime` lists.

2. **Time conversion:** The conversion of times from strings to minutes is done for each element once and takes constant time, so for N elements, this is also `O(N)`.

3. **Sorting the times:** The `sort()` operation on the list of times within the dictionary `d` can vary in its time complexity depending on the number of time stamps per name, which is less than or equal to `N`. In the worst case, if every name has `N` timestamps associated with it and we assume `M` to represent the number of unique names, then the sorting can take `O(N log N)`. However, since every timestamp can only belong to one name, and assuming `K` average number of timestamps per name, sorting would be `O(K log K)` for each name and `O(M * K log K)` in total.

4. **Checking for 60-minute frames:** The for loop runs a maximum of `K - 2` times per name, which is `O(K)`, regarding the inner operation as constant time. Multiplying by the number of unique names `M`, this gives us `O(M * K)` for all names.

5. **Sorting the answer:** Finally, sorting the list of names `ans`, which has at most `M` elements, will take `O(M log M)`.

Combining these steps, the dominating factor in time complexity is the sorting of timestamps, because `K` could be much larger than log `K`. Therefore, if every person has many timestamps, the time complexity approaches `O(N log N)`. Otherwise, it is more appropriately represented as `O(M * K log K + M log M)`.

Space Complexity

The space complexity can be analyzed as follows:

1. **Storing timestamps:** The dictionary `d` can have at most `M` unique names with `N` total timestamps, so in the worst case, space complexity for `d` is `O(N)`.

2. **Answer List:** The `ans` list will contain at most `M` elements (as it only stores names), so the space needed is `O(M)`.

Hence, the total space complexity for the algorithm is `O(N)` because the space needed for the dictionary will be larger than or equal to the space for the list with all unique names.