1629. Slowest Key

Easy Array String

Leetcode Link

Problem Description

In this problem, we are simulating the operation of a keypad being tested. The tester pressed keys in sequence, and we're given two pieces of information: the sequence of keys that were pressed and the times when each key was released.

The sequence of keys is represented by a string called `keysPressed`, where each character corresponds to a key that was pressed. The release times are given in an array `releaseTimes`, which includes the time when each key was released. Note that the array is sorted; the keys have been pressed in the order given by the string `keysPressed`, starting at time 0.

The duration of a keypress is defined as the time difference between the release of the current key and the release of the previous key. For the first key (index 0), its duration is simply its release time.

The aim is to find the key that had the longest keypress duration. If there are several keys with the same longest duration, we need to return the key with the highest lexicographical order (the one that appears last in the alphabet).

Intuition

To find the solution, we iterate through the `keysPressed` string and `releaseTimes` array to calculate the durations of all keypresses. To do this, we subtract the release time of the previous key from the release time of the current key.

We maintain two variables, `mx` for the maximum duration we have encountered so far and `ans` for the key that corresponds to this maximum duration. As we loop through the keys:

- If we find a keypress duration longer than the current maximum duration (`mx`), we update both `mx` and `ans` with the new values.
- If we find a keypress duration equal to the maximum duration (`mx`), we compare the current key with the key in `ans` lexicographically, and if the current key is lexicographically larger (i.e., it comes later in the alphabet), we update `ans`.

The logic above ensures that we will end with the key that has the longest duration, and if there is a tie, we will have the key with the highest lexicographical order.

Solution Approach

The solution uses a straightforward iteration and comparison approach without the need for complex data structures or algorithms. The variables `mx` and `ans` hold the information of the maximum duration encountered and the corresponding key respectively.

Here is a step-by-step breakdown:

1. Initialize `mx` with the release time of the first key, as there is no previous key to calculate the duration with, so the duration is releaseTimes[0].
2. Initialize `ans` with the first key itself from the `keysPressed` string.
3. Loop through indices from `1` to `len(keysPressed) - 1` (since we have already used the `0`th index for initialization).
   - Calculate the duration `d` for the `i`th keypress as `releaseTimes[i] - releaseTimes[i - 1]`.
   - Compare this duration with the current maximum duration `mx`.
   - If the current duration `d` is greater than `mx`, update both `mx` and `ans` with the current duration and key.
   - If the current duration `d` is equal to `mx`, compare the keys lexicographically.
     - We compare keys by their ASCII values using `ord()`. If the ASCII value of the current key `keysPressed[i]` is greater than that of `ans` (which means it is lexicographically larger), then update `ans` with the current key.
4. Continue this process until the loop finishes.
5. Return `ans`, which contains the key of the longest keypress duration or the lexicographically largest key if there are ties.

By using this direct method, we ensure a time complexity of O(n), where n is the length of the `keysPressed`, which is optimal since we have to examine each keypress to find the answer.

Example Walkthrough

Let's walk through an example to illustrate the solution approach.

Consider the input where `keysPressed` is `"cbcd"` and `releaseTimes` is `[1, 2, 4, 7]`.

Following the step-by-step solution approach:

1. Initialize `mx` with the release time of the first key, which is `releaseTimes[0] = 1`. There is no previous key, so the duration for this keypress is just its release time.

2. Initialize `ans` with the first key from the `keysPressed` string, which is `"c"`.

3. Now, we will loop through the indices from `1` to `len(keysPressed) - 1`, which in this case is from `1` to `3`.

   - For index `1`:

     - Calculate the duration `d` as `releaseTimes[1] - releaseTimes[0]`, which is `2 - 1 = 1`.
     - Compare `d` to `mx`. Here, `d` is equal to `mx`, which is `1`.
     - Since `d` is equal to `mx`, we compare `keysPressed[1]` with `ans` lexicographically. Here, `"b"` is less than `"c"`, so we don't update `ans`.

   - For index `2`:

     - Calculate `d` as `releaseTimes[2] - releaseTimes[1]`, which equals `4 - 2 = 2`.
     - Now `d` is greater than `mx` (`2 > 1`), so we update `mx` to `2` and `ans` to `keysPressed[2]` which is `"c"`.

   - For index `3`:

     - Calculate `d` as `releaseTimes[3] - releaseTimes[2]`, which equals `7 - 4 = 3`.
     - Again, `d` is greater than `mx` (`3 > 2`), so we update `mx` to `3` and `ans` to `keysPressed[3]` which is `"d"`.

4. After completing the loop, the maximum duration `mx` is `3` and the corresponding key `ans` is `"d"`.

5. We return `ans`, which is `"d"`. This is the key with the longest keypress duration. If there had been any ties, the solution would have returned the key with the highest lexicographical order.

There you have the walkthrough using the provided solution approach on a straightforward example. This shows how we can find the key with the longest duration or the lexicographically largest one in case of ties.

Python Solution

```python
from typing import List

class Solution:
    def slowestKey(self, release_times: List[int], keys_pressed: str) -> str:
        # Initialize the slowest key with the first key pressed
        slowest_key = keys_pressed[0]
        # Initialize the maximum duration with the duration of the first key
        max_duration = release_times[0]

        # Iterate over the keys pressed except the first one
        for i in range(1, len(keys_pressed)):
            # Calculate the duration of the current key pressed
            duration = release_times[i] - release_times[i - 1]

            # If current duration is greater than max_duration
            # or it's equal but the key is lexicographically greater,
            # update slowest_key and max_duration
            if duration > max_duration or (duration == max_duration and
                                           ord(keys_pressed[i]) > ord(slowest_key)):
                max_duration = duration
                slowest_key = keys_pressed[i]

        # Return the slowest key after iterating all keys
        return slowest_key

```

Java Solution

```java
class Solution {
    public char slowestKey(int[] releaseTimes, String keysPressed) {
        // Initialize the slowest key to the first key pressed
        char slowestKey = keysPressed.charAt(0);
        // Initialize the maximum duration to the release time of the first key
        int maxDuration = releaseTimes[0];

        // Iterate through the release times starting from the second element
        for (int i = 1; i < releaseTimes.length; ++i) {
            // Calculate the duration the key was held down
            int duration = releaseTimes[i] - releaseTimes[i - 1];

            // Compare the current duration to the max duration
            // Update if the current duration is greater, or if it's equal and the key is lexicographically larger
            if (duration > maxDuration || (duration == maxDuration && keysPressed.charAt(i) > slowestKey)) {
                maxDuration = duration; // Update the max duration
                slowestKey = keysPressed.charAt(i); // Update the slowest key
            }
        }

        // Return the slowest key with the longest duration
        return slowestKey;
    }
}

```

C++ Solution

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

class Solution {
public:
    // Function to determine the slowest key
    char slowestKey(vector<int>& releaseTimes, string keysPressed) {
        // Initialize slowest key with the first key and set its duration
        char slowestKeyChar = keysPressed[0];
        int longestDuration = releaseTimes[0];

        // Loop through the rest of the keys
        for (int i = 1; i < releaseTimes.size(); ++i) {
            // Calculate the duration for each key press
            int currentDuration = releaseTimes[i] - releaseTimes[i - 1];

            // Update the longest duration and slowest key if we find a longer duration
            // or if the duration is equal and the key character is lexically greater
            if (currentDuration > longestDuration || (currentDuration == longestDuration && keysPressed[i] > slowestKeyChar)) {
                longestDuration = currentDuration;
                slowestKeyChar = keysPressed[i];
            }
        }
        // Return the slowest key character found
        return slowestKeyChar;
    }
};

```

Typescript Solution

```typescript
// Function to determine the slowest key
function slowestKey(releaseTimes: number[], keysPressed: string): string {
    // Initialize slowest key with the first key and set its duration
    let slowestKeyChar: string = keysPressed[0];
    let longestDuration: number = releaseTimes[0];

    // Loop through the rest of the key release times
    for (let i = 1; i < releaseTimes.length; i++) {
        // Calculate the duration for each key press
        const currentDuration: number = releaseTimes[i] - releaseTimes[i - 1];

        // Update the longest duration and slowest key if we find a longer duration,
        // or if the duration is equal and the key character is lexically greater
        if (currentDuration > longestDuration ||
           (currentDuration === longestDuration && keysPressed[i] > slowestKeyChar)) {
            longestDuration = currentDuration;
            slowestKeyChar = keysPressed[i];
        }
    }

    // Return the slowest key character found
    return slowestKeyChar;
}

```

Time and Space Complexity

The provided Python code defines a function `slowestKey` that determines the character in `keysPressed` that has the longest duration between key presses. The code iterates through the `keysPressed` string and uses the `releaseTimes` list to find that character. Here's a breakdown of the time complexity and space complexity:

- **Time Complexity**: The time complexity of the function is `O(n)`, where `n` is the length of the `keysPressed` string (and also the length of the `releaseTimes` list). This is because the code iterates through the `keysPressed` string exactly once.

- **Space Complexity**: The space complexity of the function is `O(1)`, which is constant space complexity. No additional space is used that scales with the input size. Only a fixed number of single-value variables are used (`ans` and `mx`), and their space usage does not depend on the size of the input.

The time complexity is derived from the single loop running through the input lists, and the space complexity is based on the fixed number of variables used in the function.