1450. Number of Students Doing Homework at a Given Time

Easy Array Leetcode Link

Problem Description

The problem presents a scenario where multiple students have a record of when they started (`startTime`) and finished (`endTime`) their homework. These times are captured in two integer arrays where each element corresponds to an individual student. There's also a specific time called `queryTime`, which is the time we are interested in examining.

The task is to determine how many students were in the process of doing their homework at that exact `queryTime`. In other words, for a student to be counted, the `queryTime` must be between their `startTime` (inclusive) and `endTime` (inclusive). If the `queryTime` is equal to a student's `startTime` or `endTime`, that student is also counted.

Therefore, our primary objective is to scan through each student's start and end times and count how many students are within the range that includes the `queryTime`.

Intuition

The intuitive approach to solving this problem involves iterating through the set of students' start and end times, checking whether the `queryTime` falls between the two times.

Given that we have pairs of start and end times, we can use the `zip` function in Python that conveniently merges the two arrays together. This means we will get a combined iterable where each element is a tuple `(a, b)`, where `a` comes from `startTime` and `b` comes from `endTime`.

We check for each pair `(a, b)` whether `queryTime` is greater than or equal to `a` (startTime) and less than or equal to `b` (endTime). The expression `a <= queryTime <= b` will return `True` if `queryTime` is within the range; otherwise, it will return `False`.

By iterating over all these pairs and summing up the number of `True` results, we determine how many students were doing their homework at `queryTime`. The Python built-in function `sum` can be used to add up `True` values (each treated as `1`) easily.

This approach does not necessitate explicit loops or conditionals; it can be executed as a one-liner within the `busyStudent` method, making it both efficient and concise.

Solution Approach

The solution uses a simple but effective algorithm that involves the following steps and components:

1. **Zipping the Arrays:** The `zip` function takes two lists, `startTime` and `endTime`, and combines them into a single iterable. Each element of this iterable is a tuple consisting of corresponding elements from the two lists (i.e., the start and end time for a single student).
2. **List Comprehension and Conditional Expressions:** A list comprehension is used to iterate through each tuple of start and end times. It applies a conditional expression `a <= queryTime <= b` for each tuple `(a, b)`, where `a` and `b` are the start and end times, respectively. This expression returns `True` if `queryTime` is within the inclusive range `[a, b]`; otherwise, it returns `False`.
3. **Summation of True Instances:** Python treats `True` as `1` and `False` as `0`. The `sum` function is used to add up all the results of the conditional expressions within the list comprehension. Effectively, this adds up all `1`s (whenever the condition is `True`), which corresponds to counting the number of students who are busy at `queryTime`.
4. **Return the Count:** The result of the `sum` is the total number of students doing their homework at the `queryTime`, which is then returned by the function.

By leveraging Python's built-in functions and language features, this solution is concise and eliminates the need for explicit loops or if-else conditional statements. The list comprehension elegantly handles the iteration and conditional checks in a single line of code, making it an exemplary demonstration of Python's capabilities for compact code that is still readable and efficient.

Example Walkthrough

Let's consider the following simple example to illustrate the solution approach:

Suppose we have three students with the following `startTime` and `endTime` to represent when each student started and finished their homework:

```
startTimes = [1, 2, 3]
endTimes = [3, 2, 7]
queryTime = 2
```

We want to determine how many students were doing their homework at the `queryTime` of `2`.

Step by Step Walkthrough:

1. **Zipping the Arrays:** We zip `startTimes` and `endTimes` to produce a list of tuples:

   ```
   zippedTimes = [(1, 3), (2, 2), (3, 7)]
   ```

2. **List Comprehension and Conditional Expressions:** We use a list comprehension to iterate through `zippedTimes` and evaluate whether `queryTime` falls within each student's interval:

   ```
   homeworkStatus = [(1 <= 2 <= 3), (2 <= 2 <= 2), (3 <= 2 <= 7)]
   ```

   This gives us:

   ```
   homeworkStatus = [True, True, False]
   ```

3. **Summation of True Instances:** Using the `sum` function, we count the `True` values in the `homeworkStatus` list, which represent the students who were doing their homework at the `queryTime`:

   ```
   busyStudents = sum([True, True, False])
   ```

   This results in `busyStudents = 2`.

4. **Return the Count:** The final result, which is `2`, is the answer to how many students were doing their homework at `queryTime`.

So, for our example, at `queryTime = 2`, there were 2 students actively doing their homework. The one-liner corresponding to this process in the actual implementation would be:

```
busyStudents = sum(a <= queryTime <= b for a, b in zip(startTimes, endTimes))
```

And when we insert our example values, the function call would be:

```
busyStudents = sum(1 <= 2 <= 3, 2 <= 2 <= 2, 3 <= 2 <= 7)  # Evaluates to 2
```

Thus, this simple and effective algorithm finds the solution with an elegant and concise approach.

Python Solution

```python
from typing import List

class Solution:
    def busy_student(self, start_time: List[int], end_time: List[int], query_time: int) -> int:
        # Initialize the count of busy students.
        busy_students_count = 0

        # Iterate over paired start and end times using zip.
        for start, end in zip(start_time, end_time):
            # Check if the query_time is between any start and end time.
            if start <= query_time <= end:
                busy_students_count += 1  # If so, increment the count.

        # Return the total count of busy students at query_time.
        return busy_students_count

# The function can be used as follows:
# solution = Solution()
# result = solution.busy_student([1, 2, 3], [3, 2, 7], 4)
# print(result)  # Output: 1

```

Java Solution

```java
class Solution {
    // Method to count how many students are 'busy' at a given queryTime.
    // A student is considered 'busy' if the queryTime falls between their startTime and endTime inclusive.
    public int busyStudent(int[] startTimes, int[] endTimes, int queryTime) {
        int busyCount = 0; // Counter for the number of busy students

        // Iterate over the array of start times.
        // Assuming startTimes and endTimes arrays are of the same length.
        for (int i = 0; i < startTimes.length; i++) {
            // Check if the queryTime is between the startTime and endTime for each student.
            if (startTimes[i] <= queryTime && queryTime <= endTimes[i]) {
                busyCount++; // Increment the count if the student is busy at queryTime.
            }
        }

        // Return the total count of busy students.
        return busyCount;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function that counts the number of students who are busy at a specific queryTime
    int busyStudent(vector<int>& startTimes, vector<int>& endTimes, int queryTime) {
        int count = 0; // Initialize the counter to 0

        // Loop over all students
        for (int i = 0; i < startTimes.size(); ++i) {
            // If queryTime is between the startTime and endTime for a student, increase the count
            if (startTimes[i] <= queryTime && queryTime <= endTimes[i]) {
                count++;
            }
        }

        // Return the total count of students who are busy at queryTime
        return count;
    }
};

```

Typescript Solution

```typescript
// This function calculates the number of students who are busy at a given time.
// @param {number[]} startTimes - The array of start times for student study sessions.
// @param {number[]} endTimes - The array of end times for student study sessions.
// @param {number} queryTime - The specific time at which we want to know how many students are busy.
// @returns {number} - The count of students who are busy at the queryTime.
function busyStudent(startTimes: number[], endTimes: number[], queryTime: number): number {
    // Retrieve the total number of students by checking the length of the startTimes array.
    const studentCount = startTimes.length;

    // Initialize a variable to keep track of the number of busy students.
    let busyCount = 0;

    // Loop over each student's session times.
    for (let i = 0; i < studentCount; i++) {
        // If the current query time falls within the start and end times of a student's session,
        // increment the count of busy students.
        if (startTimes[i] <= queryTime && endTimes[i] >= queryTime) {
            busyCount++;
        }
    }

    // Return the total count of busy students at the query time.
    return busyCount;
}

```

Time and Space Complexity

Time Complexity:

The time complexity of the code is `O(n)`, where `n` is the number of students. This is because the code uses a generator expression within the `sum` function that iterates over each student once to check if the `queryTime` is between their `startTime` and `endTime`.

Each comparison `a <= queryTime <= b` is done in constant time `O(1)`, and since there are `n` such comparisons (assuming `startTime` and `endTime` lists are both of length `n`), the total time complexity of the loop is linear with respect to the number of students.

Space Complexity:

The space complexity of the code is `O(1)`. The generator expression does not create an additional list in memory; it computes the sum on-the-fly, and thus there is no significant additional space usage that depends on the input size. The only space used is for the variables and the input lists themselves, which are not counted towards space complexity as they are considered to be input to the function.