1086. High Five

Easy Array Hash Table Sorting Leetcode Link

Problem Description

In this problem, we are given a list of students' scores in the form `items`, where every element `items[i] = [ID_i, score_i]` corresponds to a score from a student with `ID_i`. Our task is to compute the average of the top five scores for each student. We must then return an array of pairs `result`, where `result[j] = [ID_j, topFiveAverage_j]` signifies the student with `ID_j` and their average of top five scores. It is important to note that if a student has fewer than five scores, we calculate the average of the scores they have. The `result` array should be ordered in ascending order by student ID.

To calculate the average, we sum up the top five scores of each student and then perform integer division by `5`. Integer division means that after dividing, if we have a decimal value, we drop the remainder and keep only the integer part.

Intuition

When tackling this problem, we can start by thinking about how we can pair students with their scores and how to efficiently compute the top five average. Here is a step-by-step approach to the problem:

1. **Organizing Scores by Students**: We can use a dictionary to map each student's ID to a list of their scores. A `defaultdict` from Python's `collections` module is perfect for this because it allows us to append scores to each student's list without initialization.
2. **Finding Top Scores**: We are only interested in the top five scores for each student. The `nlargest` function from Python's `heapq` module can help us to find the top five scores efficiently.
3. **Computing Averages**: Once we have the top five scores, we can simply sum them up and then divide by `5` to get the average. This must be integer division as specified by the problem.
4. **Creating the Result List**: We need the resulting list to be sorted by student ID. We start with the smallest ID and go up to the largest ID we encountered, checking if the ID has any scores mapped to it. If it has, we compute the average and add it to the result list in a `[ID_i, topFiveAverage_i]` format.
5. **Returning the Result**: After processing all student IDs, we end up with a list of ID and average pairs, which we return as the final answer.

By keeping these steps in mind, we can write a Python program that is both efficient and easy to understand, which is what the given solution code is doing.

Solution Approach

The implementation of the solution follows the intuition and can be understood in multiple steps:

1. **Creating a Dictionary with Default Values**: A `defaultdict` is created to store the scores for each student. The choice of `defaultdict` is crucial as it initializes a new list automatically if a key (student ID) is not found, avoiding key errors or the need for manual initialization.

   ```python
   d = defaultdict(list)
   ```

2. **Storing Scores by Student IDs**: The solution iterates over each score in the `items` list, appending the score to the list associated with the student's ID in the dictionary. During this process, we keep track of the maximum student ID encountered to know the range we need to consider for our final result.

   ```python
   for i, x in items:
       d[i].append(x)
       m = max(m, i)
   ```

3. **Iterating Over Possible Student IDs**: We iterate from `1` to the maximum student ID `m + 1` to ensure we cover all possible IDs. We check for each ID if there are scores associated with it.

   ```python
   for i in range(1, m + 1):
       if xs := d[i]:
   ```

4. **Calculating the Top Five Average**: For each student with scores, we use the `nlargest` function to fetch the top five scores efficiently (in O(n log 5) time where n is the number of scores per student). We then sum these scores and perform integer division by `5` to get the average.

   ```python
   avg = sum(nlargest(5, xs)) // 5
   ```

5. **Storing the Results**: The calculated averages along with the student IDs are stored in the result list `ans` using the `append` method. This ensures that the results are stored in the ascending order of the student IDs as the loop already iterates in ascending order.

   ```python
   ans.append([i, avg])
   ```

6. **Returning the Sorted Results**: Since our loop iteration ensures IDs are already in ascending order, we can directly return the `ans` list as our final sorted result.

   ```python
   return ans
   ```

The provided solution is efficient because it utilizes a dictionary to relate scores with students' IDs, and a heap-based `nlargest` method to retrieve the top scores. This method minimizes sorting overhead since it does not require sorting all the scores. Furthermore, by keeping a running maximum of encountered student IDs, the solution only looks at IDs that have scores without unnecessarily processing unused IDs, thereby optimizing the performance.

Example Walkthrough

Let's illustrate the solution approach with a small example. Assume our input `items` is:

```python
items = [[1, 91], [1, 92], [2, 93], [2, 97], [1, 60], [2, 77], [1, 65], [1, 87], [1, 100], [2, 100], [2, 76]]
```

This `items` list consists of scores for two students with IDs 1 and 2. Let's apply our solution approach step-by-step.

1. **Creating a Dictionary with Default Values**:

   First, we create a `defaultdict` to store the scores indexed by student IDs.

   ```python
   from collections import defaultdict
   d = defaultdict(list)
   ```

2. **Storing Scores by Student IDs**:

   We go through each item in the list. We add each score to the list of scores for the corresponding student ID. Suppose the `max_id` variable keeps track of the maximum ID encountered so far.

   ```python
   max_id = 0
   for i, x in items:
       d[i].append(x)
       max_id = max(max_id, i)
   # Now, d = {1: [91, 92, 60, 65, 87, 100], 2: [93, 97, 77, 100, 76]}
   ```

3. **Iterating Over Possible Student IDs**:

   Since we've seen that the maximum ID is 2, we will iterate over student IDs from 1 to 2.

   ```python
   for i in range(1, max_id + 1):
       if xs := d[i]:
   ```

4. **Calculating the Top Five Average**:

   If a student has scores, we will select the top five scores, sum them, and divide by 5 to get the average.

   ```python
   from heapq import nlargest
   ans = []
   for i in range(1, max_id + 1):
       if xs := d[i]:
           avg = sum(nlargest(5, xs)) // 5
           # For student 1: nlargest(5, [91, 92, 60, 65, 87, 100]) -> [100, 92, 91, 87, 65], sum -> 435, average -> 435//5 = 87
           # For student 2: nlargest(5, [93, 97, 77, 100, 76]) -> [100, 97, 93, 77, 76], sum -> 443, average -> 443//5 = 88
           ans.append([i, avg])
   ```

5. **Storing the Results**:

   We have calculated averages for both student IDs, so we add them to our answer list.

   ```python
   ans.append([i, avg])
   # After iteration, ans = [[1, 87], [2, 88]]
   ```

6. **Returning the Sorted Results**:

   Lastly, we return the list `ans`, which is already sorted by student IDs, as our final result.

   ```python
   return ans
   # Result: [[1, 87], [2, 88]]
   ```

In this small example, student 1 had six scores and student 2 had five. After taking the top five scores for each, the averages of their top five scores are 87 and 88, respectively. Thus, the final result reflecting each student's ID and their calculated top-five average is:

```python
[[1, 87], [2, 88]]
```

Python Solution

```python
from collections import defaultdict
from heapq import nlargest

class Solution:
    def highFive(self, items: List[List[int]]) -> List[List[int]]:
        # Dictionary to store scores for each ID
        scores_dict = defaultdict(list)
        # Variable to keep track of the highest ID number seen
        max_id = 0

        # Iterate through the items to populate the scores dictionary
        for student_id, score in items:
            # Append the score to the list of scores for the current student_id
            scores_dict[student_id].append(score)
            # Update max_id if the current student_id is greater
            max_id = max(max_id, student_id)

        # List to store the result
        result = []

        # Iterate through all possible IDs from 1 to max_id (inclusive)
        for i in range(1, max_id + 1):
            # Check if there are scores for the current ID
            if scores := scores_dict[i]:
                # Calculate the average of the top 5 scores
                average = sum(nlargest(5, scores)) // 5
                # Append [student_id, average_score] to the result list
                result.append([i, average])

        # Return the final list of averages
        return result

```

Java Solution

```java
class Solution {

    public int[][] highFive(int[][] items) {
        // Initialize an array of lists to store scores for each student
        List<Integer>[] scoresPerStudent = new List[1001];
        Arrays.setAll(scoresPerStudent, k -> new ArrayList<>());

        // Populate the lists with scores. Each student is represented at index i, and score is x
        for (var item : items) {
            int studentId = item[0];
            int score = item[1];
            scoresPerStudent[studentId].add(score);
        }

```

```
        // Sort the scores in descending order for each student
        for (var scores : scoresPerStudent) {
            scores.sort((a, b) -> b - a);
        }

        // Prepare a list to hold the answer
        List<int[]> averageTopFiveScores = new ArrayList<>();

        // Iterate through each student's list of scores
        for (int i = 1; i <= 1000; ++i) {
            var scores = scoresPerStudent[i];
            // Ensure that the student has at least one score
            if (!scores.isEmpty()) {
                int sum = 0;
                // Get the top five scores, or all scores if fewer than five, and calculate their sum
                for (int j = 0; j < Math.min(5, scores.size()); ++j) {
                    sum += scores.get(j);
                }
                // Compute and add the average of the top five scores for the student to the result list
                averageTopFiveScores.add(new int[] {i, sum / Math.min(5, scores.size())});
            }
        }

        // Convert the list of average scores to a 2D array and return
        return averageTopFiveScores.toArray(new int[0][]);
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to calculate the average of the top five scores for each student
    vector<vector<int>> highFive(vector<vector<int>>& items) {
        vector<int> scores[1001]; // Array of vectors to store scores for each student

        // Iterate through all the score items
        for (auto& item : items) {
            int studentId = item[0]; // Extract the student ID
            int score = item[1];     // Extract the score
            scores[studentId].push_back(score); // Add score to the corresponding student's list
        }

        vector<vector<int>> result; // Resultant vector to store the averages of top five scores

        // Process the scores for each student
        for (int i = 1; i <= 1000; ++i) { // Assuming student IDs are in the range 1 to 1000
            if (!scores[i].empty()) { // Ensure that the student has scores recorded
                sort(scores[i].begin(), scores[i].end(), greater<int>()); // Sort scores in descending order

                int sumTopFiveScores = 0; // Variable to store the sum of the top five scores
                for (int j = 0; j < 5; ++j) { // Calculate the sum of top 5 scores
                    sumTopFiveScores += scores[i][j];
                }
                int averageTopFiveScores = sumTopFiveScores / 5; // Compute the average of top five scores

                result.push_back({i, averageTopFiveScores}); // Add the student ID and average to the result
            }
        }

        return result; // Return the final result
    }
};

```

Typescript Solution

```typescript
// This function calculates the average of the top 5 scores for each unique ID.
function highFive(items: number[][]): number[][] {
    // Initialize an array to store the scores grouped by ID.
    const scoresById: number[][] = Array(1001)
        .fill(0)
        .map(() => Array());

    // Loop through each item (id, score) and group the scores by their ID.
    for (const [id, score] of items) {
        scoresById[id].push(score);
    }

    const averages: number[][] = []; // This will hold the final result.

    // Iterate through the potential range of IDs.
    for (let i = 1; i <= 1000; ++i) {
        // Check if the current ID has any scores.
        if (scoresById[i].length > 0) {
            // Sort the scores for the current ID in descending order.
            scoresById[i].sort((a, b) => b - a);

            // Calculate the sum of the top 5 scores.
            const sumTopFive = scoresById[i].slice(0, 5).reduce((a, b) => a + b, 0);

            // Calculate the average, floor it, and push the ID and average to the result array.
            averages.push([i, Math.floor(sumTopFive / 5)]);
        }
    }

    // Return the array containing ID and average of top 5 scores.
    return averages;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the function is determined by several factors:

1. The time complexity of iterating over `items`, which has `n` elements: `O(n)`.
2. Appending each item's score to the corresponding list in the dictionary: `O(1)` (amortized) for each insertion.
3. The `max` function inside the loop, which has constant time `O(1)` because it's just comparing two integers.
4. The final iteration over the range from `1` to `m + 1`, which is `O(m)` where `m` is the maximum id seen.
5. For each student, the `nlargest` function is used to obtain the top 5 scores which have a complexity of `O(k * log(m))` for each student (`k` is 5 here, and `m` is the number of scores for that student).
6. Summing the scores and calculating the average has constant time complexity `O(1)`.

Since we don't know the relationship between `m` and `n` exactly, we combine the terms to reflect the worst case. We can assume each student has at least 5 scores and at most `n` scores. Thus, the `nlargest` step ends up with a time complexity of `O(n * log(n))` due to at most `n` calls to `nlargest` with a list of maximum length `n`.

Combining these, our overall time complexity is `O(n + n * log(n))`, which simplifies to `O(n * log(n))`.

Space Complexity

For space complexity, we have:

1. A dictionary that holds up to `m` keys, with each key holding a list of scores. If every score is unique to a student, the space complexity is `O(n)`.
2. An output list `ans` that will hold `m` elements: `O(m)`.

Combining these, we get `O(n) + O(m)`. However, since `m` cannot exceed `n` (as there cannot be more student IDs than individual scores), the space complexity simplifies to `O(n)`.