1792. Maximum Average Pass Ratio

Medium | Greedy | Array | Heap (Priority Queue) | Leetcode Link

Problem Description

In the given LeetCode problem, we have a list of classes, each represented by a list of two integers where the first integer indicates the number of students that will pass the final exam and the second integer represents the total number of students in that class. Additionally, we have a certain number of extra brilliant students who are guaranteed to pass the exam in whatever class they join.

The goal is to distribute these extra students among the classes in a way that maximizes the overall average pass ratio. The pass ratio for a class is calculated as the fraction of students passing the exam over the total number of students in the class. The overall average pass ratio is the sum of individual class pass ratios divided by the total number of classes.

We are tasked with assigning the extra students and returning the highest possible average pass ratio across all classes, where answers within `10^-5` of the actual answer will be considered acceptable.

Intuition

To maximize the overall average pass ratio, we need to consider where adding an extra student would make the most significant impact. Intuitively, adding a student to a class where it will cause a more substantial increase in the pass ratio would be more beneficial than adding them to a class where the increase would be less significant. This is because the marginal gain in the pass ratio decreases as the number of students in the class increases.

To efficiently determine where to add an extra student, we can use a max heap data structure. The heap will help us prioritize the classes based on which of them would give us the highest incremental increase in the pass ratio if an extra student were added.

Here's the intuition behind the solution steps:

1. First, we calculate the potential increase in the pass ratio for each class if we add one extra student and store this along with the current pass and total counts in a max heap. The potential increase is represented as the difference between the current pass ratio and the pass ratio if we added one student, which is `(a / b - (a + 1) / (b + 1))`, where `a` is the number of students passing and `b` is the total number of students.

2. We then iterate for each extra student, popping off the class from the heap with the highest potential increase and updating that class's pass and total counts to reflect adding one extra student. We then calculate the new potential increase in the pass ratio for that class and push it back into the heap.

3. Now with each extra student assigned to a class, we calculate the final average pass ratio by summing up the pass ratios of each class and dividing by the number of classes.

In summary, by continuously allocating extra students to the classes where they lead to the most significant relative improvement in the pass ratio, we can achieve the maximum average pass ratio across all classes.

Solution Approach

The implementation of the solution uses a priority queue, represented in Python as a min heap (due to Python's heapq library natively implementing min heaps). Since we want to use this as a max heap, we store tuples with the first element being the negative of the potential increase in the pass ratio (to invert the min heap into a max heap), along with the current number of students passing and the total students for that class.

Here's how each part of the solution is implemented:

1. **Building the Max Heap:** The max heap is initialized with the negative potential increase in the pass ratio of adding an extra student to each class, along with the current passing and total student counts for each class. The reason for storing the negative is that Python's heapq module provides a min heap; negating the values allows us to simulate a max heap. Thus, the line `(a / b - (a + 1) / (b + 1), a, b)` for each class `[a, b]` in `classes` captures the marginal gain from adding an extra student, and `heapify(h)` turns this list into a heap in-place for efficient access to the class with the maximum potential increase.

2. **Allocating Extra Students:** We then iterate the number of `extraStudents` times. In each iteration, we pop from the heap to get the class with the current highest potential pass ratio increase (`heappop(h)`), increment the passing and total student counts (`a + 1, b + 1`), and push the updated class back to the heap (`heappush(h, ...)`) with its new potential increase.

3. **Calculating the Final Average Pass Ratio:** Finally, once all the extra students have been allocated, we calculate the sum of the actual pass ratios of each class (`sum(v[1] / v[2] for v in h)`) and divide by the number of classes to get the average pass ratio.

The algorithm is particularly efficient because it uses a priority queue to always select the class that will benefit the most from an additional student, rather than having to recalculate and compare the pass ratio increases for all classes upon each student's assignment. This allows the solution to run in O((n + m) * log(n)) time complexity, where `n` is the number of classes and `m` is the number of extra students since each heap operation (pop and push) takes O(log(n)) time, and there are `m` such operations plus `n` operations to build the initial heap.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach outlined above. Suppose we have two classes and two extra brilliant students to distribute. The first class has 1 student who will pass out of a total of 2 students, and the second class has 2 students who will pass out of a total of 5 students. Our classes list is therefore `[[1, 2], [2, 5]]`.

1. **Building the Max Heap:**
   - For the first class, the potential increase in the pass ratio from adding an extra student is calculated by the expression ((1/2 - (1+1)/(2+1))), which equals approximately -0.1667 when taking the negative (remember we are using a min heap to simulate a max heap).
   - For the second class, the potential increase is ((2/5 - (2+1)/(5+1))), which equals approximately -0.0333.
   - The max heap, therefore, initially contains the elements [(-0.1667, 1, 2), (-0.0333, 2, 5)] after it has been heapified.
2. **Allocating Extra Students:**
   - We now distribute the two extra students. For the first student, we pop the class with the highest potential increase, which is the first class, and update the counts to reflect the addition of a student. The updated class now has a passing count of `1+1` and a total count of `2+1`.
   - We now recalculate the potential increase for this class, which becomes ((2/3 - (2+1)/(3+1))), or approximately -0.0833 after taking the negative, and push it back onto the heap.
   - The heap currently contains [(-0.0833, 2, 3), (-0.0333, 2, 5)].
   - For the second student, we again pop the class with the highest potential increase, which is now the first class again, and update it similarly. The updated class now has a passing count of `2+1` and a total of `3+1`.
   - We recalculate the potential increase for this class, which is ((3/4 - (3+1)/(4+1))), or approximately -0.05 when taking the negative, and push it back onto the heap.
   - The heap now contains [(-0.05, 3, 4), (-0.0333, 2, 5)].
3. **Calculating the Final Average Pass Ratio:**
   - With no more extra students to allocate, we calculate the final average pass ratio.
   - We sum the actual pass ratios of each updated class. The final pass ratios are (3/4) for the first class and (2/5) for the second class.
   - The sum is (3/4 + 2/5), which is 1.95/2.0.
   - We divide this sum by the total number of classes (2) to get the average pass ratio, which comes out to 0.975.

This is the highest possible average pass ratio that can be achieved by distributing the two extra students and illustrates how the max heap is used to prioritize class updates to maximize the overall average pass ratio.

Python Solution

```python
from heapq import heapify, heappop, heappush
from typing import List

class Solution:
    def maxAverageRatio(self, classes: List[List[int]], extra_students: int) -> float:
        # Create a max-heap based on the change in average by adding an extra student to each class
        # The data in the heap is a tuple containing the difference in the ratio,
        # the number of pass students and then total number of students in that class.
        max_heap = [(-(a / b - (a + 1) / (b + 1)), a, b) for a, b in classes]
        # Convert the array into a heap data structure (in-place)
        heapify(max_heap)

        # Allocate extra students to the classes
        for _ in range(extra_students):
            # Pop the class with the maximum potential ratio increase
            neg_delta, pass_students, total_students = heappop(max_heap)
            # Update the student counts for that class
            pass_students += 1
            total_students += 1
            # Calculate the new potential increase in average and push it back into the heap
            new_neg_delta = -(pass_students / total_students - (pass_students + 1) / (total_students + 1))
            heappush(max_heap, (new_neg_delta, pass_students, total_students))

        # After all extra students have been allocated, calculate the total average ratio
        total_ratio = sum(pass_students / total_students for _, pass_students, total_students in max_heap)
        # Return the average ratio across all classes
        return total_ratio / len(classes)

```

Java Solution

```java
import java.util.PriorityQueue;

class Solution {
    public double maxAverageRatio(int[][] classes, int extraStudents) {
        // Priority queue to store each class using a custom comparator based on the improvement
        // in pass ratio by adding one extra student to the class.
        PriorityQueue<double[]> priorityQueue = new PriorityQueue<>((a, b) -> {
            double improvementA = (a[0] + 1) / (a[1] + 1) - a[0] / a[1];
            double improvementB = (b[0] + 1) / (b[1] + 1) - b[0] / b[1];
            return Double.compare(improvementB, improvementA); // Max-heap, so we invert the comparison
        });

        // Populate the priority queue with the pass ratio of each class
        for (int[] cls : classes) {
            priorityQueue.offer(new double[] {cls[0], cls[1]});
        }

        // Distribute the extra students to the classes where they would cause the highest improvement
        while (extraStudents-- > 0) {
            double[] currentClass = priorityQueue.poll();
            double passes = currentClass[0] + 1, totalStudents = currentClass[1] + 1;
            priorityQueue.offer(new double[] {passes, totalStudents});
        }

        // Calculate the total average ratio after all extra students have been distributed
        double totalAverageRatio = 0;
        while (!priorityQueue.isEmpty()) {
            double[] classRatio = priorityQueue.poll();
            totalAverageRatio += classRatio[0] / classRatio[1];
        }

        // Return the final average ratio by dividing by the total number of classes
        return totalAverageRatio / classes.length;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <queue>

class Solution {
public:
    // Function to calculate the maximum average ratio after adding extra students
    double maxAverageRatio(vector<vector<int>>& classes, int extraStudents) {
        // Create a priority queue to store the potential increase in ratio and class counts
        priority_queue<tuple<double, int, int>> potentialQueue;

        // Calculate the initial potential increase for each class and push it to the priority queue
        for (const auto& cls : classes) {
            int passCount = cls[0], totalCount = cls[1];
            double potentialIncrease = static_cast<double>(passCount + 1) / (totalCount + 1) - static_cast<double>(passCount) / tot
            potentialQueue.push({potentialIncrease, passCount, totalCount});
        }

        // Distribute extra students to classes where they would maximally increase the average ratio
        while (extraStudents--) {
            // Get the class with the highest potential increase
            auto [potential, passCount, totalCount] = potentialQueue.top();
            potentialQueue.pop();

            // Add an extra student to the class
            passCount++;
            totalCount++;

            // Recalculate the potential increase for the updated class and push back into the queue
            potentialIncrease = static_cast<double>(passCount + 1) / (totalCount + 1) - static_cast<double>(passCount) / totalCount
            potentialQueue.push({potentialIncrease, passCount, totalCount});
        }

        // Calculate the final average ratio
        double totalRatio = 0;
        while (!potentialQueue.empty()) {
            auto [_, passCount, totalCount] = potentialQueue.top();
            potentialQueue.pop();
            totalRatio += static_cast<double>(passCount) / totalCount;
        }

        // Return the average ratio across all classes
        return totalRatio / classes.size();
    }
};

```

Typescript Solution

```typescript
// Import statement for PriorityQueue, assuming there's a library or custom implementation for this
// As TypeScript does not have a built-in PriorityQueue, an external library should be used,
// or one should implement a custom PriorityQueue that supports custom comparator.

import { PriorityQueue } from 'some-priority-queue-library';

// Class pair structure to store classes' passCount and totalCount.
interface ClassPair {
    passCount: number;
    totalCount: number;
}

// Function to calculate the potential increase in ratio
function calculatePotentialIncrease(passCount: number, totalCount: number): number {
    return (passCount + 1) / (totalCount + 1) - passCount / totalCount;
}

// Function to calculate the maximum average ratio after adding extra students
function maxAverageRatio(classes: number[][], extraStudents: number): number {
    // Create a priority queue to store the potential increase in ratio and class counts.
    // The comparator should ensure the highest potential increase comes first.
    const potentialQueue = new PriorityQueue<(number | ClassPair)[]>({
        comparator: function(a, b) {
            return a[0] > b[0];
        }
    });

    // Calculate the initial potential increase for each class and push it to the priority queue.
    classes.forEach(([passCount, totalCount]) => {
        const potentialIncrease = calculatePotentialIncrease(passCount, totalCount);
        potentialQueue.push([potentialIncrease, { passCount, totalCount }]);
    });

    // Distribute extra students to classes where they would maximally increase the average ratio.
    while (extraStudents > 0) {
        const [potential, classPair] = potentialQueue.pop();
        const { passCount, totalCount } = classPair as ClassPair;

        // Add an extra student to the class.
        classPair.passCount++;
        classPair.totalCount++;

        // Recalculate the potential increase for the updated class and push back into the queue.
        const newPotentialIncrease = calculatePotentialIncrease(classPair.passCount, classPair.totalCount);
        potentialQueue.push([newPotentialIncrease, classPair]);

        extraStudents--;
    }

    // Calculate the final average ratio.
    let totalRatio = 0;
    while (potentialQueue.length > 0) {
        const [, classPair] = potentialQueue.pop();
        const { passCount, totalCount } = classPair as ClassPair;
        totalRatio += passCount / totalCount;
    }

    // Return the average ratio across all classes.
    return totalRatio / classes.length;
}

```

Time and Space Complexity

Time Complexity

The primary operations in this code are:

1. The initial creation and heapification of a list of tuples, which takes `O(N)` time, where `N` is the length of the `classes` list.
2. Extra iterations equal to the number of extra students `extraStudents`. In each iteration:
   - `heappop()` operation, which has a time complexity of `O(log N)`.
   - Simple arithmetic operations, followed by a `heappush()` operation, which also has a time complexity of `O(log N)`.

Therefore, the time complexity for the loop that runs `extraStudents` times is `O(extraStudents * log N)`.

The final computation to sum the ratios has a time complexity of `O(N)`.

Adding all these up, the total time complexity is `O(N + extraStudents * log N)`.

Space Complexity

The space complexity of the code is due to:

1. The space required for heap `h`, which stores `N` tuples, so it's `O(N)`.
2. The space for the outputs of the arithmetic operations, which is constant `O(1)`, as they don't depend on the size of the input and are reused in each iteration.

Therefore, the total space complexity is `O(N)`.