2358. Maximum Number of Groups Entering a Competition

Medium | Greedy | Array | Math | Binary Search Leetcode Link

Problem Description

The problem deals with organizing a set of university students with given grades into groups to enter a competition. Each group should be ordered based on the following two conditions:

1. The sum of the grades in the `i`th group should be less than the sum of the grades in the `(i + 1)`th group for all the groups except the last one.
2. The number of students in the `i`th group should be less than the number of students in the `(i + 1)`th group for all the groups except the last one.

With these conditions in mind, the task is to determine the maximum number of groups that can be formed with the given grades array. It's important to note that each group must be non-empty and the students should be ordered in a way that the conditions mentioned above are respected.

Intuition

To arrive at the solution for this problem, we should start by understanding that the conditions imply a pattern of groups where both the sum of grades and the number of students per group increase progressively.

A natural approach could involve trying to create groups by starting with a single student in the first group and adding more students to subsequent groups while ensuring that the sum of grades increases. However, this approach may not always give us the maximum number of groups.

Let's observe that the minimum sum that a group of `k` students can have is when the group has the smallest `k` grades, and it would be best to assign the smallest grades to the earlier groups to meet the increasing sum condition efficiently.

By looking at the pattern where the sum and the size of each subsequent group need to be larger than the previous group, we can infer that there is a quadratic relationship in the way we assign students to groups to maximize the total number of groups.

The actual solution uses binary search to find the maximum `k` such that the first `k` triangular numbers (the sum of the first `k` positive integers, which is `k(k + 1)/2`) does not exceed the length of the grades array. This aligns with the observation that adding one student to each new group (starting from 1 student in the first group, 2 in the second, and so on) keeps increasing both the sum and the size of the groups.

The `bisect_right` function in Python finds the insertion point to the right of the search space. Essentially, it finds the point where `k(k + 1)/2` would be inserted in the sequence from 0 to `n*2` (where `n` is the number of grades) while ensuring that `k(k + 1)/2` is less than or equal to `n` (the length of the `grades` list). By subtracting 1 from the result, we get the largest `k` where the groups can be formed according to the rules specified in the problem.

Therefore, by looking for the largest `k` such that the sum of the first `k` natural numbers is less than or equal to the size of the grades array, we effectively find the maximum number of groups that can be formed.

Solution Approach

The solution provided makes use of both a mathematical insight and the `bisect_right` binary search algorithm from Python's standard library.

The key mathematical insight here is recognizing that we're essentially trying to find the maximal `k` for which the sum of the first `k` numbers (which is given by `k(k + 1)/2`, the `k`th triangular number) does not exceed the total number of grades `n`. This sum has to be less than or equal to `n` because we can only arrange `n` students into these groups, and the size of each group increases linearly (1 for the first group, 2 for the second, and so on).

The `bisect_right` function is a binary search algorithm that assumes the input is already sorted (which indeed `range(n + 1)` is). It searches for the point where an element (in this case the `k`th triangular number) would be inserted while keeping the list sorted. We thus provide it a key function `lambda x: x * x + x` that, for each `x` in `range(n + 1)`, calculates twice the sum of the first `x` numbers (since `2(k(k + 1)/2) = k(k + 1)`), and the `bisect_right` function finds the point where this value would exceed `n * 2`.

The `range(n + 1)` represents potential group sizes from 0 to `n`, which we use as the search space for the binary search. We multiply `n` by 2 in both the range and the lambda function to avoid dealing with fractions (which are slower and less accurate in binary computations). After finding the insertion point, we subtract 1 to find the actual maximum number of groups `k` that can be formed.

Let's break down the code:

- `n = len(grades)`: we get the number of grades to know how many students we have.
- `bisect_right(range(n + 1), n * 2, key=lambda x: x * x + x)`: we apply `bisect_right`.
  - `range(n + 1)` is our sorted list from 0 to `n`.
  - `n * 2` is the value that should not be exceeded by twice the triangular number.
  - `key=lambda x: x * x + x` is the calculation that mimics twice the `k`th triangular number.
- After finding the insertion point for `k(k + 1)`, we subtract 1 to get the maximum number of groups that can be formed under the described constraints.

This approach efficiently computes the result without having to actually simulate the group formation, leading to an algorithm that is far more efficient than brute-forcing would be, especially as `n` gets large.

Here's the python code enclosed in the markdown syntax:

```python
class Solution:
    def maximumGroups(self, grades: List[int]) -> int:
        n = len(grades)
        return bisect_right(range(n + 1), n * 2, key=lambda x: x * x + x) - 1
```

Example Walkthrough

Let's use a small set of grades to illustrate the solution approach: `[70, 60, 80, 40, 30]`.

There are 5 students. According to the problem, we should form groups where each subsequent group has a higher sum of grades and more students than the previous one.

Initially, we might start with the first group consisting of the student with the lowest grade, which is 30. The next group should have a higher sum and more students. Let's try to keep the groups as small as possible to maximize the number of groups. The second group could then include the next lowest two grades: 40 and 60, for a sum of 100. So far, our groups would be `[30]` and `[40, 60]` with sums of 30 and 100, respectively.

Following the pattern, the third group should have at least 3 students (since the second has 2), and their sum should be more than 100. We are left with grades 70 and 80, so we cannot form such a group.

In the intuitive approach, we managed to form only 2 groups, but we aim to find the maximum number of groups possible.

Our mathematical insight tells us that the number of groups is limited by the requirement that each group should have one more student than the previous one and that the sum of the members of each group follows the pattern of triangular numbers (1, 3, 6, 10, 15, etc.).

In Python, we would use the `bisect_right` function as follows:

1. The number `n` is set to the length of the grades list, which is 5.
2. We call `bisect_right(range(6), 10, key=lambda x: x * x + x)`. The `range(6)` represents our search space (0 to 5, representing possible values of `k`). We're comparing against `10` because it's `n * 2` (5 * 2). The key function computes `x(x + 1)`, which represents twice the value of the `k`th triangular number.
3. The `bisect_right` function will find the point where inserting a value would exceed `10`. In this case, the insertion point is `4`, as `3(3 + 1)` is 12 when doubled, and `2(2 + 1)` is 6, which does not exceed 10.
4. Subtracting 1 from this insertion point gives us `3`, indicating we cannot form more than 3 groups satisfying both conditions.

So, according to our method, we can form a maximum of 3 groups with the provided grades. Note that the algorithm does not require us to specify which grades exactly go into each group because it is working on the principle that smaller groups come first and each subsequent group will naturally have more than the sum of any smaller group before it by virtue of needing more members.

In this example, however, even though our algorithm tells us we can form 3 groups, the given grades only allow for 2 groups that meet both conditions. This is a result of the specific grades given and the constraints. The purpose of the algorithm is to find the upper limit of possible groups, which may not always be attainable with the given specific set of grades.

Python Solution

```python
from typing import List
from bisect import bisect_right

class Solution:
    def maximumGroups(self, grades: List[int]) -> int:
        # The total number of grades given
        total_grades = len(grades)

        # Use binary search to find the right position in the range [0, total_grades + 1)
        # The key function calculates the sum of the first x natural numbers, which is used to
        # determine if there's enough grades to form x groups
        # The formula for the sum of the first x natural numbers is (x * (x + 1)) // 2
        # For the condition to be satisfied, it should be less than or equal to the total number of grades
        # Therefore, the range for bisect_right is squared and added to x to maintain the inequality
        # Subtracting 1 at the end provides the maximum number of groups that can be formed
        max_groups = bisect_right(range(total_grades + 1), total_grades * 2, key=lambda x: x * (x + 1) // 2) - 1

        return max_groups
```

Java Solution

```java
class Solution {
    public int maximumGroups(int[] grades) {
        // The length of the grades array
        int length = grades.length;

        // Variables to define the search range, initialized to the entire range of possible group numbers
        int left = 0, right = length;

        // Binary search to find the maximum number of groups
        while (left < right) {
            // Calculate the middle point. We calculate it this way to avoid integer overflow.
            int mid = (left + right + 1) >>> 1;

            // Check if the total number of students fits the condition for 'mid' groups
            // The condition is derived from the requirements of forming groups with an increasing number of students.
            // mid * (mid + 1) / 2 is the sum of the first 'mid' integers, which is the minimum number of students needed for 'mid' g
            // We use long to avoid integer overflow when evaluating the condition.
            if (1L * mid * (mid + 1) > 2L * length) {
                // If total students are insufficient, we decrease the 'right' boundary
                right = mid - 1;
            } else {
                // If total students are sufficient, we increase the 'left' boundary
                left = mid;
            }
        }

        // When the while loop exits, 'left' will be the maximum number of groups that can be created
        return left;
    }
}
```

C++ Solution

```cpp
#include <vector>
using namespace std;

class Solution {
public:
    // Function to find the maximum number of groups with different sizes
    int maximumGroups(vector<int>& grades) {
        int n = grades.size(); // Store the total number of grades
        int left = 0; // Initialize left boundary of binary search
        int right = n; // Initialize right boundary of binary search

        // Execute binary search to find the maximum number of groups
        while (left < right) {
            int mid = (left + right + 1) / 2; // Calculate the middle index, leaning to the right

            // Check if the sum of group sizes exceeds the total amount of grades
            // as the sum of first k numbers (1+2+...+k) is given by k*(k+1)/2
            if (1LL * mid * (mid + 1) > 2LL * n) {
                // If it exceeds, we need to reduce our search space to lower half
                right = mid - 1;
            } else {
                // If it does not exceed, we need to explore upper half
                // to find if we can fit more groups
                left = mid;
            }
        }

        // The left index now contains the maximum number of possible groups
        return left;
    }
};
```

Typescript Solution

```typescript
function maximumGroups(grades: number[]): number {
    // Total number of grades available
    const totalGrades = grades.length;

    // Initialize the search bounds for the smallest and largest possible groups
    let minimumGroups = 1; // Lower bound of search
    let maximumGroups = totalGrades; // Upper bound of search

    // Perform binary search to find the maximum number of groups
    while (minimumGroups < maximumGroups) {
        // Calculate the midpoint (using right shift by 1 for integer division by 2)
        // and add 1 to ensure the left part of the sandwich is smaller
        const mid = (minimumGroups + maximumGroups + 1) >> 1;

        // Check if the current midpoint's number of groups is too large to form using the triangular number formula
        // Triangular number formula: mid * (mid + 1) / 2 must be <= totalGrades
        if (mid * (mid + 1) > totalGrades * 2) {
            // If too large, adjust the upper bound of the search
            maximumGroups = mid - 1;
        } else {
            // Otherwise, adjust the lower bound to mid
            minimumGroups = mid;
        }
    }

    // After the loop, minimumGroups contains the maximum number of groups we can form
    return minimumGroups;
}
```

Time and Space Complexity

The given Python code is aimed at finding the maximum number of groups with distinct lengths that the list of `grades` can be divided into. The key part of this code is the use of `bisect_right`, a binary search algorithm from the Python `bisect` module, to efficiently find the point where a quadratic equation's result surpasses a certain value, `n * 2`.

Time Complexity

The binary search performed using `bisect_right` operates on a range of numbers from `0` to `n+1`. The time complexity of a binary search is `O(log k)`, where `k` is the size of the range we're searching within. In our case, because the search is within a numerical range up to `n+1`, the time complexity can be expressed as `O(log n)`.

The lambda function is applied on each iteration of the binary search to compute the sum of squares and a linear term of the current middle value `x`. This operation is `O(1)` as it involves simple arithmetic operations. Since this lambda is called for each step of the binary search, it does not change the overall time complexity, maintaining it as `O(log n)`.

Space Complexity

`bisect_right` operates within the given range and does not require additional space proportional to the input size. The variables used to store the results of the lambda operation and the lengths of `grades` (`n`) are constant with respect to the input size, leading to `O(1)` space complexity.

Therefore, the space complexity of this code is `O(1)`, signifying constant space usage regardless of the input size.