2037. Minimum Number of Moves to Seat Everyone

Easy | Array | Sorting

Leetcode Link

Problem Description

In this problem, we have `n` seats and `n` students, with each being given an initial position. The goal is to find the minimum number of moves required to place each student in a seat such that no two students end up in the same seat. The array `seats` contains the positions of the seats and the array `students` contains the initial positions of the students. A move consists of increasing or decreasing the position of a student by 1. This problem becomes a matter of optimally matching each student to a seat.

Intuition

The key to solving this problem is realizing that the most optimal way to match students to seats with the minimum moves is to match the closest available seat to each student. To achieve this, we can sort both the `seats` and `students` arrays. Sorting brings pairs of students and seats that are closest to each other in correspondence.

Once sorted, the position of the i-th student should match with the i-th seat. Since each move changes a position by 1, the total moves for each student to reach their seat is the absolute difference between their position and their allocated seat's position. By sorting, we ensure that we are not crossing pairs, which could potentially lead to a higher number of moves if we were to move students past each other.

When the pairs are matched one-to-one in a sorted manner, we can sum the absolute differences of corresponding pairs to find the minimum number of moves required for all students to be seated.

Solution Approach

The solution involves sorting algorithms and the concept of absolute difference to calculate the minimum number of moves. The following steps depict the implementation:

1. **Sorting**: We use a sorting algorithm (such as quicksort, mergesort, etc., typically `O(n log n)` complexity) to sort both the `seats` and `students` arrays. In Python, this is done using the `sort()` method directly on the lists, which sorts the list in-place.

2. **Pairing Students with Seats**: After sorting, the students and seats are paired up by their indices. That is, the first student (`students[0]`) will go to the first seat (`seats[0]`), and so on.

3. **Calculating Moves**: Since the solution only requires the total number of moves, we go through each pair `(student, seat)` and calculate the absolute difference between their positions. The `abs()` function is used for this purpose, which is a built-in function in Python to calculate the absolute value of a number.

   For example, if a student is at position 2 (`students[i] = 2`) and their paired seat is at position 5 (`seats[i] = 5`), the moves required to get the student to that seat would be `abs(2 - 5) = 3`. The number of moves is then added to a running total.

4. **Summing Up Moves**: As we calculate the absolute difference for each student and seat pair, we sum up these differences using the `sum()` function. This sum represents the total minimum number of moves required to seat all students. The `zip()` function in Python is helpful here to iterate over corresponding elements of `seats` and `students` simultaneously.

The code for this would look like this:

```python
class Solution:
    def minMovesToSeat(self, seats: List[int], students: List[int]) -> int:
        seats.sort()     # Sort the seats array.
        students.sort()  # Sort the students array.
        # Calculate the sum of absolute differences between paired students and seats.
        return sum(abs(a - b) for a, b in zip(seats, students))
```

This approach guarantees an optimal solution with a time complexity of `O(n log n)` because of the sorting step, which is the most time-consuming part of the algorithm. The pairing and summing steps are linear, making it `O(n)`. Therefore, the sorting step dictates the overall time complexity.

Example Walkthrough

Let's consider an example with `n = 3` seats and `n = 3` students with the following initial positions:

- `seats = [4, 1, 5]`
- `students = [3, 2, 6]`

First, using the solution approach, we need to sort both `seats` and `students` arrays to organize them in ascending order:

- Sorted `seats`: `[1, 4, 5]`
- Sorted `students`: `[2, 3, 6]`

Now, we pair the students with their seats by index:

- 1st pair: seat at position 1 and student at position 2 (student needs to move 1 position to the left)
- 2nd pair: seat at position 4 and student at position 3 (student needs to move 1 position to the right)
- 3rd pair: seat at position 5 and student at position 6 (student needs to move 1 position to the left)

Next, we calculate the absolute differences for each pair to find out the number of moves required:

- Moves for 1st pair: `abs(seats[0] - students[0]) = abs(1 - 2) = 1`
- Moves for 2nd pair: `abs(seats[1] - students[1]) = abs(4 - 3) = 1`
- Moves for 3rd pair: `abs(seats[2] - students[2]) = abs(5 - 6) = 1`

Finally, we sum up the number of moves for all pairs to find the total minimum number of moves:

- Total moves: `(1 + 1 + 1) = 3`

Thus, it takes a minimum of 3 moves to seat all the students.

Python Solution

```python
class Solution:
    def min_moves_to_seat(self, seats: List[int], students: List[int]) -> int:
        # First, sort the lists of seats and students to organize them in ascending order.
        # This aligns each student to the closest available seat.
        seats.sort()
        students.sort()

        # Next, calculate the total number of moves by iterating over the paired sorted lists.
        # For each pair, find the absolute difference between a student's position and
        # a seat's position. This represents the number of moves a student must make to reach the seat.
        moves = sum(abs(seat - student) for seat, student in zip(seats, students))

        # Return the total number of moves needed for all students to sit in the seats.
        return moves

```

Java Solution

```java
import java.util.Arrays; // Import Arrays class for sorting

class Solution {
    // Method to find the minimum number of moves required to seat all students.
    // Each student moves to a seat, and the total moves should be minimized.
    public int minMovesToSeat(int[] seats, int[] students) {
        // Sort the arrays to match student-seat pairs by their order.
        Arrays.sort(seats);
        Arrays.sort(students);

        // Initialize the count of moves to zero.
        int totalMoves = 0;

        // Iterate over the arrays to match each student to a seat
        for (int i = 0; i < seats.length; ++i) {
            // Add the absolute difference of positions to total moves.
            // Absolute difference indicates the number of moves for each student.
            totalMoves += Math.abs(seats[i] - students[i]);
        }

        // Return the total number of moves required to seat all students.
        return totalMoves;
    }
}

```

C++ Solution

```cpp
#include <vector> // Include header for vector
#include <algorithm> // Include header for sort

class Solution {
public:
    // Function to find the minimum number of moves to seat everyone
    int minMovesToSeat(vector<int>& seats, vector<int>& students) {
        // Sort the 'seats' vector in non-decreasing order.
        sort(seats.begin(), seats.end());
        // Sort the 'students' vector in non-decreasing order.
        sort(students.begin(), students.end());

        // Initialize a variable 'minMoves' to store the minimum number of moves needed.
        int minMoves = 0;
        // Iterate over the sorted vectors to find the total moves needed.
        for (int i = 0; i < seats.size(); ++i) {
            // Calculate the absolute difference between corresponding seats and students
            // and add it to the 'minMoves' counter.
            minMoves += abs(seats[i] - students[i]);
        }
        // Return the total minimum number of moves required.
        return minMoves;
    }
};

```

Typescript Solution

```typescript
// Function to calculate the minimum number of moves required to seat students.
// Each student moves to their seat with the least possible amount of individual moves.
// Parameters `seats` and `students` are arrays of numbers representing seat positions and students' current positions respectively.
function minMovesToSeat(seats: number[], students: number[]): number {
    // Sort the seats array in ascending order.
    seats.sort((a, b) => a - b);
    // Sort the students array in ascending order.
    students.sort((a, b) => a - b);

    // Length of the seats array, which is also the number of students.
    const numberOfSeats = seats.length;
    // Initialize the accumulator for the total moves required.
    let totalMoves = 0;

    // Iterate over each seat/student pair to calculate the total moves.
    for (let index = 0; index < numberOfSeats; index++) {
        // Increase the total moves by the distance the current student needs to move to reach the current seat.
        totalMoves += Math.abs(seats[index] - students[index]);
    }

    // Return the computed total moves.
    return totalMoves;
}

```

Time and Space Complexity

The given code sorts two lists, calculates the absolute difference between corresponding elements from each sorted list, and then sums the differences.

Time Complexity:

1. Sorting both `seats` and `students` arrays has a time complexity of `O(n log n)` where `n` is the number of elements in each list. Since both lists are sorted independently, the combined time complexity is `O(2 * n log n)` which simplifies to `O(n log n)`.

2. The `zip()` function pairs the elements of both lists, which is `O(n)` where `n` is the length of the shorter list. Here we can assume both lists are the same length.

3. The list comprehension with `sum()` iterates over the zipped lists to compute the absolute differences and sum them up. This operation is linear with respect to the number of seats/students, so it is `O(n)`.

Thus, the overall time complexity of the provided code is `O(n log n) + O(n)`. Since `O(n log n)` dominates `O(n)`, the final time complexity is `O(n log n)`.

Space Complexity:

1. The sorting is done in-place, so it doesn't require additional space proportional to the input size. Thus, the space complexity of the sorting operation is `O(1)`.

2. The use of `zip()` and the list comprehension within `sum()` does not create a new list of size `n` but instead creates an iterator that generates tuples one by one. Therefore, it only requires constant extra space.

3. The final expression computes the sum in constant space, adding the absolute difference one pair at a time.

So, the overall space complexity of the provided code is `O(1)`, constant space, since it only uses a fixed amount of additional memory aside from the input lists.