2274. Maximum Consecutive Floors Without Special Floors

Medium Array Sorting

Leetcode Link

Problem Description

Alice is managing a company with office space spanning several floors of a building. She has rented floors from a certain `bottom` floor to a certain `top` floor. Not every floor is for work; some special floors are intended for relaxation. The problem provides us with the range of floors Alice has rented by specifying two integers `bottom` and `top`, indicating that all floors between and including these two are rented. We must also account for the array `special`, which contains the specific floors designated for relaxation.

Our objective is to determine the maximum number of consecutive floors that are not dedicated to relaxation. In other words, we want to find the longest stretch of floors that are uninterrupted by special floors where employees can work without encountering a relaxation space.

Intuition

To solve this, we can approach the problem by focusing on the gaps between the special floors, as well as the beginning and the end of the total floor range. Consecutive floors without a special floor can only exist within these gaps.

Here's the step-by-step intuition behind the solution:

1. Sort the `special` array. Sorting helps us to easily process the special floors in ascending order, allowing efficient comparison between adjacent special floors and to easily identify the maximum gap between them.
2. Find the maximum gap at the beginning. The first gap is between the `bottom` floor Alice rented and the first special floor. This is simply the difference between the first element in the sorted `special` array and the `bottom` variable (subtracting one since the special floor itself cannot be included).
3. Find the maximum gap at the end. Similarly, the last potential gap is between the last special floor and the `top` floor Alice rented. This is the difference between the `top` variable and the last element in the sorted `special` array.
4. Find the maximum gap between the consecutive special floors. We iterate over the sorted `special` array and calculate the difference between each pair of adjacent special floors, subtracting one to exclude the special floor itself.
5. The answer is the largest of the gaps found in steps 2, 3, and 4. This maximum number represents the longest stretch of floors available without any special floor interruptions, thus fulfilling Alice's requirement.

The provided Python function `maxConsecutive` implements this thought process using a sorted list of special floors and comparisons to find and return the maximum number of consecutive floors without a special floor.

Solution Approach

The solution utilizes a sorting algorithm and a single pass iteration to determine the longest sequence of non-special floors encompassed between `bottom` and `top`, given the `special` floors.

Here's a comprehensive walk-through of the solution's implementation:

1. **Sorting the Special Floors**: The algorithm begins by sorting the array `special`. This is critical, as it ensures that the floors are evaluated in sequential order, which is necessary for identifying the consecutive gaps between them optimally. Python's sort method on arrays (`list.sort()`) is employed here, which typically provides (O(n \log n)) time complexity, where `n` is the number of elements in the `special` array.
2. **Initializing Maximum Gaps**: We initialize `ans` with the maximum gap possible at the beginning or end of the range. This is done by checking the first and last item of the sorted `special` array:
   - To check the gap at the beginning, the algorithm computes `special[0] - bottom`. This gives the count of floors between the first rented floor (`bottom`) and the first special floor.
   - To check the gap at the end, the algorithm calculates `top - special[-1]`. This gives the count of floors between the last special floor and the last rented floor (`top`).
   - The larger of these two values becomes the initial `ans`, as it represents the longest currently known stretch without a special floor.
3. **Iterating and Comparing Consecutive Special Floors**: We iterate through the sorted `special` array starting from the second element to the end of the array, comparing each pair of adjacent special floors:
   - The difference between each pair of adjacent special floors `special[i] - special[i - 1]` is computed, and then one is subtracted from this value (`- 1`). The subtraction accounts for the fact that we exclude the starting special floor of each gap.
   - The calculated value represents the number of consecutive non-special floors between the current special floor and the one preceding it.
4. **Updating Maximum Gaps**: After each comparison of consecutive special floors, we update `ans` if the number of floors between the current pair of special floors is greater than the current value of `ans`, effectively keeping track of and updating the maximum stretch of consecutive non-special floors.
5. **Returning the Result**: After iterating through all the special floors and identifying the maximum gaps both at the start, end, and between the special floors, `ans` will hold the maximum number of consecutive floors without a special floor. This value is returned as the final result.

The algorithm's overall complexity is dominated by the sorting step, with (O(n \log n)) time complexity for sorting and (O(n)) for the single pass through the list, resulting in a time complexity of (O(n \log n)) overall.

Example Walkthrough

Let's illustrate the solution approach using a small example. Suppose Alice rents the floors 1 through 10 (`bottom = 1, top = 10`) and the special floors for relaxation is `[2, 5, 9]`.

Here's the step-by-step application of the solution:

1. **Sorting the Special Floors**: First, we sort the array `special`, which after sorting looks like `[2, 5, 9]`. Sorting isn't necessary in this case because the array is already sorted, but it's an essential step in the general case.
2. **Initializing Maximum Gaps**: We initialize `ans` with the maximum gap at the beginning or the end. The maximum gap at the beginning is `special[0] - bottom` which is `2 - 1 = 1`. The maximum gap at the end is `top - special[-1]` which is `10 - 9 = 1`. The larger of these values is `1`, so `ans` is set to `1`.
3. **Iterating and Comparing Consecutive Special Floors**: We iterate through the sorted special floors and calculate the gaps: a. The gap between the first and the second special floors is `5 - 2 - 1 = 2`. This is the number of floors from floor 3 to 4. b. The gap between the second and the third special floors is `9 - 5 - 1 = 3`. This is the number of floors from floor 6 to 8.
4. **Updating Maximum Gaps**: We compare the found gaps with `ans`. The consecutive gaps we found are `2` and `3`. Since `3` is greater than the current `ans` value of `1`, we update `ans` to `3`.
5. **Returning the Result**: After evaluating all gaps, we find that the largest gap is `3`. Therefore, the function `maxConsecutive(bottom, top, special)` would return `3`, indicating the maximum number of consecutive floors without a relaxation floor is a stretch from floor 6 to floor 8.

By following this approach, we've arrived at the solution using an example scenario by using the sorted special floors `[2, 5, 9]` and the provided `bottom` and `top` values.

Python Solution

```python
class Solution:
    def maxConsecutive(self, bottom: int, top: int, special: list[int]) -> int:
        # First we sort the 'special' list to find the consecutive gaps efficiently
        special.sort()

        # The maximum consecutive number starts with either the gap before the first
        # special number or after the last special number because beyond these
        # points we only have consecutive numbers without interruption.
        max_consecutive = max(special[0] - bottom, top - special[-1])

        # Now we find the gap between each pair of special numbers and update the
        # maximum consecutive numbers. We subtract one because two 'special' numbers
        # can never be part of the consecutive sequence.
        for i in range(1, len(special)):
            gap = special[i] - special[i - 1] - 1
            max_consecutive = max(max_consecutive, gap)

        # Return the overall maximum consecutive numbers found.
        return max_consecutive
```

Java Solution

```java
class Solution {
    public int maxConsecutive(int bottom, int top, int[] special) {
        // Sort the special array to find consecutive ranges easily
        Arrays.sort(special);

        // Obtain the size of the special array
        int n = special.length;

        // The maximum consecutive numbers can start from the bottom or end at the top,
        // initialize it by considering the gaps between the bottom and the first special,
        // and the last special and the top.
        int maxConsecutive = Math.max(special[0] - bottom, top - special[n - 1]);

        // Iterate through the sorted special numbers to find the largest gap between them
        for (int i = 1; i < n; ++i) {
            // Calculate the gap between current and previous special, excluding both
            int gap = special[i] - special[i - 1] - 1;

            // Update maxConsecutive if the current gap is larger
            maxConsecutive = Math.max(maxConsecutive, gap);
        }

        // Return the maximum number of consecutive numbers not included in special
        return maxConsecutive;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <algorithm>

int maxConsecutive(int bottom, int top, std::vector<int>& special) {
    // Copy the 'special' vector and sort it in ascending order
    std::vector<int> sortedSpecial(special);
    std::sort(sortedSpecial.begin(), sortedSpecial.end());

    // Add boundary elements to the sorted vector to simplify edge cases.
    // One less than the bottom value and one more than the top value.
    sortedSpecial.insert(sortedSpecial.begin(), bottom - 1);
    sortedSpecial.push_back(top + 1);

    // Initialize a variable to keep track of the maximum consecutive gap
    int maxGap = 0;

    // Calculate the length of the sortedSpecial vector for iteration
    int sortedSpecialLength = sortedSpecial.size();

    // Iterate through the sortedSpecial vector to find the maximum gap
    for (int i = 1; i < sortedSpecialLength; i++) {
        // Calculate the gap between consecutive elements, subtract 1 since the endpoints are not included
        int currentGap = sortedSpecial[i] - sortedSpecial[i - 1] - 1;

        // Update maxGap if currentGap is greater than the previously recorded maximum
        maxGap = std::max(maxGap, currentGap);
    }

    // Return the largest gap found
    return maxGap;
}
```

Typescript Solution

```typescript
function maxConsecutive(bottom: number, top: number, special: number[]): number {
    // Copy the 'special' array and sort it in ascending order
    let sortedSpecial = special.slice().sort((a, b) => a - b);

    // Add boundary elements to the sorted array to simplify edge cases.
    // One less than the bottom value and one more than the top value.
    sortedSpecial.unshift(bottom - 1);
    sortedSpecial.push(top + 1);

    // Initialize variable to keep track of the maximum consecutive gap
    let maxGap = 0;

    // Calculate the length of the sortedSpecial array for iteration
    const sortedSpecialLength = sortedSpecial.length;

    // Iterate through the sortedSpecial array to find the maximum gap
    for (let i = 1; i < sortedSpecialLength; i++) {
        // Calculate the gap between consecutive elements, subtract 1 since the endpoints are not included
        const currentGap = sortedSpecial[i] - sortedSpecial[i - 1] - 1;

        // Update maxGap if currentGap is greater than the previously recorded maximum
        maxGap = Math.max(maxGap, currentGap);
    }

    // Return the largest gap found
    return maxGap;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the sorting operation is `O(n log n)`, where `n` is the number of elements in the `special` list. After sorting, the function iterates through the `special` list exactly once, which has a time complexity of `O(n)`. As the sorting operation dominates, the overall time complexity of the code is `O(n log n)`.

Space Complexity

The space complexity of the algorithm is `O(1)` (under the assumption that the sort is done in-place). The space used by the algorithm does not grow with the size of the input, as it uses a fixed amount of extra space (just a few variables to keep track of the maximum consecutive floors: `ans`, `i` and the input `special` that is sorted in place).