1337. The K Weakest Rows in a Matrix

Easy · Array · Binary Search · Matrix · Sorting · Heap (Priority Queue) · Leetcode Link

Problem Description

In this problem, we are given a matrix `mat` that is composed of binary values - `1`'s and `0`'s. Every `1` in the matrix represents a soldier, and every `0` represents a civilian. One of the key elements of the setup is the arrangement of the soldiers and civilians in each row; all the soldiers (1's) come before any civilians (0's). This ordering makes it visually similar to a sorted binary array where all `1`'s are at the start of the array, followed by all `0`'s.

We are asked to evaluate the "strength" of each row based on the number of soldiers (`1`'s) in it. A row is considered "weaker" if it has fewer soldiers in it than another row, or if it has the same number of soldiers but comes earlier in the matrix (i.e., it has a smaller row index).

The problem requires us to return the indices of the `k` weakest rows in the matrix ordered from the weakest to the strongest. It's essentially like constructing a "leaderboard" of rows, with the least number of soldiers making a row rank higher (weaker) on this board.

Intuition

Approaching this problem, we observe two tasks: counting the number of soldiers in each row and then sorting the rows according to their "strength." Since the soldiers (1's) are all positioned to the left, the count of soldiers in a row is equal to the number of continuous `1`'s starting from the first column until the first `0` appears. This can be thought of as finding the first occurrence of `0` in the row.

Since the rows are sorted in the non-increasing order with all the soldiers at the beginning, we can use a binary search technique to quickly find the position of the first civilian (`0`) which indicates the number of soldiers in the row. The binary search here drastically reduces the time complexity over a linear scan, especially when the rows are long.

We construct a result `ans` that maps each row to the number of soldiers it has, by applying the binary search on the reversed row. We reverse the row before applying the binary search with `bisect_right` because this function is typically used to find the insertion point in a sorted array to maintain the order. By reversing, our soldiers `1` come at the end, and the insertion point for `0`tells us the number of `1`'s.

After we have the counts for each row, we create an index list `idx` which initially is just a list of row indices `[0, 1, 2, ..., m-1]`. We sort this index list based on the corresponding soldier counts from the `ans` array. Finally, we use slicing to get the first `k` elements from this sorted index list which represent the indices of the `k` weakest rows.

Solution Approach

The solution makes effective use of Python's built-in `bisect` library, which provides support for maintaining a list in sorted order without having to sort the list after each insertion. In particular, the `bisect_right` function is utilized to implement a binary search through a row to find the count of soldiers in that row.

Here's a step-by-step breakdown of the solution:

1. Determine the dimensions of the matrix with `m` as the number of rows and `n` as the number of columns.
2. An array `ans` is created to hold the number of soldiers (1's) for each row.
3. We iterate over each row in the matrix, and for each row, we apply `bisect_right` to the reversed row: `bisect_right(row[::-1], 0)`. This reversed row is a trick used to turn our sorted array of 1's followed by 0's into an array where a binary search can find the insertion point of 0 effectively. The result of this operation is the count of civilians (0's) in the reversed row, which is subtracted from `n` to get the count of soldiers.
4. Create an index list `idx` containing the indices of the rows, which starts from 0 up to `m-1`.
5. The index list `idx` is then sorted using a lambda function as the key. This lambda function maps each index `i` to the number of soldiers `ans[i]`, ensuring the sort operation arranges the indices according to the strength of the rows (with ties broken by row index, as per problem statement).
6. We use list slicing to retrieve the first `k` indices from the sorted `idx`, giving us the indices of the `k` weakest rows.

Code Snippet Deconstructed

Let's look at the critical sections of the code:

- `ans = [n - bisect_right(row[::-1], 0) for row in mat]` This line computes the strength of each row and stores it in `ans`. It calculates the number of soldiers as the total length `n` minus the number of civilians at the end of each reversed row.

- `idx.sort(key=lambda i: ans[i])` This line sorts the `idx` list according to the number of soldiers in each corresponding row. The lambda function returns the number of soldiers for each row using `ans[i]` as the key, ensuring that rows with fewer soldiers come first.

- `return idx[:k]` Returns the first `k` elements from the sorted list of indices, which corresponds to the `k` weakest rows in the matrix ordered from weakest to strongest.

In terms of algorithms and data structures, this solution primarily uses the binary search algorithm (through `bisect_right`) and basic list operations in Python (list comprehension, sorting, and slicing).

Example Walkthrough

Let's consider a matrix `mat` and an integer `k`:

```
mat = [
  [1, 1, 0, 0],
  [1, 1, 1, 0],
  [1, 0, 0, 0],
  [1, 1, 0, 0],
  [1, 1, 1, 1]
]
k = 3
```

In this matrix:

- Row 0 has 2 soldiers.
- Row 1 has 3 soldiers.
- Row 2 has 1 soldier (making it the weakest).
- Row 3 has 2 soldiers.
- Row 4 has 4 soldiers (making it the strongest).

We need to find the indices of the `k` weakest rows in the matrix, which are `k = 3` in our example.

Following the solution approach:

1. The dimensions of the matrix are `m = 5` rows and `n = 4` columns.
2. We'll create an array `ans` to hold counts of soldiers for each row.

For each row's count of soldiers using `bisect_right`:

- `ans[0] = 4 - bisect_right([0, 0, 1, 1][::-1], 0) = 4 - 2 = 2`
- `ans[1] = 4 - bisect_right([0, 1, 1, 1][::-1], 0) = 4 - 3 = 1`
- `ans[2] = 4 - bisect_right([0, 0, 0, 1][::-1], 0) = 4 - 1 = 3`
- `ans[3] = 4 - bisect_right([0, 0, 1, 1][::-1], 0) = 4 - 2 = 2`
- `ans[4] = 4 - bisect_right([1, 1, 1, 1][::-1], 0) = 4 - 4 = 0`

After applying the count for each row, `ans` looks like this: `[2, 1, 3, 2, 0]`.

3. Now we have `ans = [2, 1, 3, 2, 0]` and our index list `idx = [0, 1, 2, 3, 4]`.

4. Sorting the index list `idx` using the `ans` array:

After sorting `idx` based on `ans` values, we get the order `[4, 1, 0, 3, 2]` since `ans[4]` corresponds to the strongest row and `ans[2]` corresponds to the weakest.

5. Since we want the `k` weakest rows, we take the first `k` elements of the sorted index `idx` list: `[4, 1, 0]`.

The array `[4, 1, 0]` is then sorted to maintain the row order: `[0, 1, 4]`.

6. The final answer, which represents the indices of the `k` weakest rows ordered from weakest to stronger, is `[0, 1, 4]`.

This method allows for efficiently determining the weakest rows due to the use of the `bisect_right` to perform binary searches, rather than linear searches, and thus reduces the average complexity of the solution.

Python Solution

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

class Solution:
    def kWeakestRows(self, mat: List[List[int]], k: int) -> List[int]:

        # Determine the dimensions of the matrix
        num_rows = len(mat)
        num_cols = len(mat[0])

        # Calculate the soldier count in each row. Since rows are sorted, the count of 1s is found by
        # subtracting the first position of a 0 from the end of the reversed row, from the row width.
        soldier_counts = [num_cols - bisect_right(row[::-1], 0) for row in mat]

        # Create a list of indices from 0 to the number of rows - 1.
        indices = list(range(num_rows))

        # Sort the indices based on the number of soldiers in the row, using the soldier_counts as keys.
        indices.sort(key=lambda i: soldier_counts[i])

        # Slice the list of indices to return only the first 'k' elements, corresponding to the k weakest rows.
        return indices[:k]

```

Java Solution

```java
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;

class Solution {
    public int[] kWeakestRows(int[][] mat, int k) {
        int rowCount = mat.length;       // Number of rows in the matrix
        int colCount = mat[0].length;   // Number of columns in the matrix
        int[] soldierCount = new int[rowCount]; // Array to store the count of soldiers in each row
        List<Integer> indices = new ArrayList<>(); // List to store the indices of the rows

        // Populate the list with indices and the soldierCount array with the number of soldiers in each row
        for (int i = 0; i < rowCount; ++i) {
            indices.add(i);
            int left = 0, right = colCount;
            // Use binary search to find the number of soldiers in the row
            while (left < right) {
                int mid = (left + right) >> 1; // Find the middle index
                if (mat[i][mid] == 0) { // If mid element is 0, search in the left half
                    right = mid;
                } else { // Else search in the right half
                    left = mid + 1;
                }
            }
            soldierCount[i] = left; // Store the soldier count (index of the first 0)
        }

        // Sort the indices based on the number of soldiers (strength of the row)
        indices.sort(Comparator.comparingInt(i -> soldierCount[i]));

        int[] weakestRows = new int[k]; // Array to store the k weakest rows
        // Fill the weakestRows array with the first k indices from the sorted list
        for (int i = 0; i < k; ++i) {
            weakestRows[i] = indices.get(i);
        }
        return weakestRows; // Return the result
    }
}

```

C++ Solution

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

class Solution {
public:
    // Helper function to find the number of soldiers in the row.
    // Soldiers are represented by 1s and are always to the left of civilians (0s).
    int countSoldiers(vector<int>& row) {
        int left = 0;
        int right = row.size() - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (row[mid] == 0)
                right = mid - 1; // No soldier at the mid, look left.
            else
                left = mid + 1;  // Soldier found, look right.
        }
        return left; // left will point to the first civilian (0), which is equal to the soldier count.
    }

    // Function to find the k weakest rows in the matrix.
    vector<int> kWeakestRows(vector<vector<int>>& matrix, int k) {
        vector<pair<int, int>> strengthIndexPairs; // Pairs of soldier count and original row index.
        vector<int> weakestRows;  // Vector to store indices of the k weakest rows.

        // Compute the soldier count for each row and store along with the row index.
        for (int i = 0; i < matrix.size(); i++) {
            int soldierCount = countSoldiers(matrix[i]);
            strengthIndexPairs.push_back({soldierCount, i});
        }

        // Sort the pairs by the number of soldiers and use the index for tie-breaking.
        sort(strengthIndexPairs.begin(), strengthIndexPairs.end());

        // Collect the indices of the k weakest rows.
        for (int i = 0; i < k; i++) {
            weakestRows.push_back(strengthIndexPairs[i].second);
        }

        return weakestRows; // Return the result.
    }
};

```

Typescript Solution

```typescript
function kWeakestRows(mat: number[][], k: number): number[] {
    let rowCount = mat.length;

    // Create a map of sums of the rows along with their original indices
    let rowCountSumMap = mat.map((row, index) => [row.reduce((accumulator, currentValue) => accumulator + currentValue, 0), index]);

    let weakestRows = []; // This array will store the indices of the k weakest rows

    // Perform a modified bubble sort to find the k weakest rows. Note: this sorting is not the most efficient method for large datas
    for (let i = 0; i < k; i++) {
        for (let j = i + 1; j < rowCount; j++) { // Start with j = i + 1 as we already have i in position
            // Compare row sum, and then index if sums are equal
            if (
                rowCountSumMap[j][0] < rowCountSumMap[i][0] ||
                (rowCountSumMap[j][0] === rowCountSumMap[i][0] && rowCountSumMap[j][1] < rowCountSumMap[i][1])
            ) {
                // Swap with destructuring if current row is 'weaker' or has a smaller index than i-th
                [rowCountSumMap[i], rowCountSumMap[j]] = [rowCountSumMap[j], rowCountSumMap[i]];
            }
        }
        // Add the index of the i-th weakest row to our answer
        weakestRows.push(rowCountSumMap[i][1]);
    }

    return weakestRows; // Return the k weakest rows' indices
}

```

Time and Space Complexity

The time complexity of the code primarily consists of two parts: the computation of the soldiers in each row and the sorting of the index based on the number of soldiers.

1. Computing the number of soldiers in each row: This uses a binary search (`bisect_right`) for each of the `m` rows. Since the binary search operates on a row of size `n`, the complexity for this part is `O(m * log(n))`.

2. Sorting the index list: After having computed the number of soldiers, we sort `m` indices based on the computed values. The worst-case time complexity for sorting in Python is `O(m * log(m))`.

Combining both parts, the total time complexity is `O(m * log(n) + m * log(m))`.

In terms of space complexity:

- The `ans` list is `O(m)` because it contains the number of soldiers for each of the `m` rows.
- The `idx` list is also `O(m)` because it includes an index for each row.
- No additional significant space is used.

Hence, the space complexity is `O(m)` as we only need space proportional to the number of rows.