296. Best Meeting Point

Hard Array Math Matrix Sorting   Leetcode Link

Problem Description

In this problem, you are given a `m x n` binary grid, where each cell in the grid can either contain a `1` (which represents the home of a friend) or a `0` (which signifies an empty space). The goal is to find the minimal total travel distance to a single meeting point.

The "total travel distance" is defined as the sum of all distances from the homes of each friend (cells with a `1`) to the meeting point. When calculating distances, we use the Manhattan Distance, which means the distance between two points is the sum of the absolute differences of their coordinates. For two points `p1` and `p2`, the Manhattan Distance is `|p2.x - p1.x| + |p2.y - p1.y|`.

The challenge of the problem lies in finding the optimal meeting point that minimizes the total travel distance for all friends.

Intuition

The intuition behind the solution is based on properties of the Manhattan Distance in a grid. When considering the distance along one axis (either horizontal or vertical), the best meeting point minimizes the distance to all points along that axis. In a one-dimensional space, this is the median of all points because it ensures that the sum of distances to the left and right (or up and down) are minimized.

The solution consists of two steps:

1. Separately find the best meeting point for rows and columns. By treating the rows and columns independently and finding the medians, we effectively split the problem into two one-dimensional issues.
2. Calculate the sum of Manhattan Distances from all friends' homes to the found meeting point.

To find the medians:

- First, we iterate through the grid and store the row and column numbers of all cells with a '1' into separate lists - `rows` and `cols`.
- The median for the rows is simply the middle element of the `rows` list. This works because the grid rows are already indexed in sorted order.
- For the columns, since we collect them in the order they appear while iterating over the rows, we need to sort the `cols` list before finding the middle element, which serves as the median.

Finally, the function `f(arr, x)` calculates the sum of distances of all points in `arr` to point `x`, the median. The `minTotalDistance` function returns the sum of `f(rows, i)` and `f(cols, j)`, where `i` and `j` are the median row and column indices, representing the optimal meeting point.

Solution Approach

The implementation of the solution starts by defining a helper function `f(arr, x)` which is responsible for calculating the total Manhattan Distance for a list of coordinates, `arr`, to a given point `x`. Using the Manhattan Distance formula, we sum up `abs(v - x)` for all values `v` in `arr`, representing either all the x-coordinates or y-coordinates of friends' homes.

In the main function `minTotalDistance`, we first create two lists, `rows` and `cols`. These lists will store the x and y coordinates, respectively, of all friends' homes. We achieve this by iterating through every cell in the grid with nested loops. When we find a cell with a `1`, we append the row index `i` to `rows` and the column index `j` to `cols`.

The next step is to sort `cols`. The `rows` are already in sorted order because we've collected them by iterating through each row in sequence. However, `cols` are collected out of order, so we must sort them to determine the median accurately.

Once we have our sorted lists, we find the medians by selecting the middle elements of the `rows` and `cols` lists. These are our optimal meeting points along each axis. Here, the bitwise right shift operator `>>` is used to find the index of the median quickly. It's equivalent to dividing the length of the list by 2. We calculate `i = rows[len(rows) >> 1]` for rows and `j = cols[len(cols) >> 1]` for columns.

Finally, we sum up the total Manhattan Distance from all friends' homes to the median points by calling `f(rows, i) + f(cols, j)`. This sum represents the minimal total travel distance that all friends have to travel to meet at the optimal meeting point, and it is returned as the solution to the problem.

This implementation utilizes basic algorithm concepts, such as iteration, sorting, and median selection coupled with mathematical insight specific to the problem context—the Manhattan Distance formula. The approach is efficient as it breaks down a seemingly complex two-dimensional problem into two easier one-dimensional problems by exploiting properties of the Manhattan Distance in a grid.

Example Walkthrough

Let's illustrate the solution approach using a small 3×3 binary grid example:

```
Grid:
1 0 0
0 1 0
0 0 1
```

The grid shows that we have three friends living in different homes, each marked by `1`. They are trying to find the best meeting point.

Following the approach:

1. We first iterate over the grid row by row, and whenever we find a `1`, we store the row and column indices in separate lists:
   - `rows = [0, 1, 2]` (Already in sorted order, since we are going row by row).
   - `cols = [0, 1, 2]` (These happen to be in sorted order, but in general, this wouldn't be the case, and we would need to sort this list).
2. Since both `rows` and `cols` lists are already sorted, we find the medians directly. Length of both lists is `3`:
   - Median row index `i = rows[3 >> 1] = rows[1] = 1`.
   - Median column index `j = cols[3 >> 1] = cols[1] = 1`.

We have determined that the best meeting point is at the cell with coordinates (1,1), which is also the home of one of the friends, in the center of the grid.

3. Next, we calculate the Manhattan Distance for each friend to the meeting point using the helper function `f(arr, x)`:
   - `f(rows, i) = abs(0 - 1) + abs(1 - 1) + abs(2 - 1) = 1 + 0 + 1 = 2`.
   - `f(cols, j) = abs(0 - 1) + abs(1 - 1) + abs(2 - 1) = 1 + 0 + 1 = 2`.
4. The minimal total travel distance is the sum of the distances calculated, which is `2 + 2 = 4`.

This total distance of `4` represents the minimum travel distance for all friends to meet at the (1,1) cell in the grid. This conclusion is reached following the properties of Manhattan Distance and the strategy of using medians to minimize the travel distance.

Python Solution

```python
class Solution:
    # Method to calculate the minimum total distance
    def minTotalDistance(self, grid: List[List[int]]) -> int:
        # Helper function to calculate the distance to the median element 'median' from all elements in 'elements'
        def calculate_distance(elements, median):
            return sum(abs(element - median) for element in elements)

        # List to record the positions of '1's in rows and columns
        row_positions, col_positions = [], []

        # Loop through the grid to find positions of '1's
        for row_index, row in enumerate(grid):
            for col_index, cell in enumerate(row):
                if cell:  # If the cell is '1', record its position
                    row_positions.append(row_index)
                    col_positions.append(col_index)

        # Sort the column positions to easily find the median
        col_positions.sort()

        # Find medians of rows and columns positions for '1'',s
        # since the list is sorted/constructed in order, the median is the middle value
        row_median = row_positions[len(row_positions) // 2]
        col_median = col_positions[len(col_positions) // 2]

        # Calculate the total distance using the median of rows and columns
        total_distance = calculate_distance(row_positions, row_median) + calculate_distance(col_positions, col_median)

        return total_distance

```

Java Solution

```java
class Solution {
    public int minTotalDistance(int[][] grid) {
        // The grid dimensions
        int rows = grid.length;
        int cols = grid[0].length;

        // Lists to store the coordinates of all 1s in the grid
        List<Integer> iCoordinates = new ArrayList<>();
        List<Integer> jCoordinates = new ArrayList<>();

        // Iterate through the grid to populate the lists with the coordinates of 1s
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                if (grid[i][j] == 1) {
                    iCoordinates.add(i);
                    jCoordinates.add(j);
                }
            }
        }

        // Sort the columns' coordinates as row coordinates are already in order because of the way they are added
        Collections.sort(jCoordinates);

        // Find the median of the coordinates, which will be our meeting point
        int medianRow = iCoordinates.get(iCoordinates.size() >> 1);
        int medianCol = jCoordinates.get(jCoordinates.size() >> 1);

        // Calculate the total distance to the median points
        int totalDistance = calculateDistance(iCoordinates, medianRow) + calculateDistance(jCoordinates, medianCol);
        return totalDistance;
    }

    // Helper function to calculate the total distance all 1s to the median along one dimension
    private int calculateDistance(List<Integer> coordinates, int median) {
        int sum = 0;
        for (int coordinate : coordinates) {
            sum += Math.abs(coordinate - median);
        }
        return sum;
    }
}

```

C++ Solution

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

class Solution {
public:
    int minTotalDistance(std::vector<std::vector<int>>& grid) {
        int numRows = grid.size();  // Row count of the grid
        int numCols = grid[0].size();  // Column count of the grid
        std::vector<int> rowPositions;  // Stores the row positions of 1s in the grid
        std::vector<int> colPositions;  // Stores the column positions of 1s in the grid

        // Collect positions of 1s (people) in each dimension
        for (int row = 0; row < numRows; ++row) {
            for (int col = 0; col < numCols; ++col) {
                if (grid[row][col]) {
                    rowPositions.emplace_back(row);
                    colPositions.emplace_back(col);
                }
            }
        }

        // Sort the positions of the columns as they may not be in order
        sort(colPositions.begin(), colPositions.end());

        // Find the median position for persons in grid for rows and columns separately
        int medianRow = rowPositions[rowPositions.size() / 2];
        int medianCol = colPositions[colPositions.size() / 2];

        // Lambda function to calculate the total distance for 1 dimension
        auto calculateDistance = [](const std::vector<int>& positions, int median) {
            int sumDistances = 0;
            for (int position : positions) {
                sumDistances += std::abs(position - median);
            }
            return sumDistances;
        };

        // Calculate total distance to the median row and median column
        return calculateDistance(rowPositions, medianRow) + calculateDistance(colPositions, medianCol);
    }
};

```

Typescript Solution

```typescript
function minTotalDistance(grid: number[][]): number {
    const numRows = grid.length; // Row count of the grid
    const numCols = grid[0].length; // Column count of the grid
    const rowPositions: number[] = []; // Stores the row positions of 1s in the grid
    const colPositions: number[] = []; // Stores the column positions of 1s in the grid

    // Collect positions of 1s (people) in each dimension
    for (let row = 0; row < numRows; ++row) {
        for (let col = 0; col < numCols; ++col) {
            if (grid[row][col] === 1) {
                rowPositions.push(row);
                colPositions.push(col);
            }
        }
    }

    // Sort the positions of the columns as they may not be in order
    colPositions.sort((a, b) => a - b);

    // Find the median position for people in grid for rows and columns separately
    const medianRow = rowPositions[Math.floor(rowPositions.length / 2)];
    const medianCol = colPositions[Math.floor(colPositions.length / 2)];

    // Function to calculate the total distance for one dimension
    const calculateDistance = (positions: number[], median: number): number => {
        return positions.reduce((sumDistances, position) => {
            return sumDistances + Math.abs(position - median);
        }, 0);
    };

    // Calculate total distance to the median row and median column
    return calculateDistance(rowPositions, medianRow) + calculateDistance(colPositions, medianCol);
}

```

Time and Space Complexity

The time complexity of the given code is primarily determined by two factors: the traversal of the grid once and the sorting of the column indices.

1. **Traversing the Grid:** Since every cell of the grid is visited once, this operation has a time complexity of `O(mn)` where `m` is the number of rows and `n` is the number of columns in the grid.
2. **Sorting the Columns:** The sorting operation is applied to the list containing the column indices, where the worst case is when all the cells have a 1, and hence the list contains `mn` elements. Sorting a list of `n` elements has a time complexity of `O(n log n)`. Thus, the complexity for sorting the column array is `O(mn log(mn))`.

With these operations, the total time complexity is the sum of individual complexities. However, since `O(mn log(mn))` dominates `O(mn)`, the overall time complexity is `O(mn log(mn))`.

For space complexity:

1. **Storing the Rows and Columns:** In the worst case, we store the row index for `mn` ones and the same for the column index. Thus, the space complexity is `O(2mn)` which simplifies to `O(mn)` because constant factors are neglected in Big O notation.

Therefore, the final time complexity is `O(mn log(mn))` and the space complexity is `O(mn)`.