2387. Median of a Row Wise Sorted Matrix

Medium Array Binary Search Matrix Leetcode Link

Problem Description

The problem presents us with a matrix `grid` of size `m x n` (where 'm' represents the number of rows and 'n' represents the number of columns), which contains an **odd** number of integers. It's stated that the integers in each row of the grid are sorted in **non-decreasing** order. Our task is to find the **median** of all the integers in the matrix. But there's a catch: we must find the median in less than `O(m * n)` time complexity. This condition tells us that we cannot simply sort all the elements of the matrix and pick the middle one, as this would exceed the allowed time complexity.

Intuition

The intuition behind the solution lies in binary search and the understanding of how the median is positioned within a sorted list of numbers. In a sorted list of odd numbers, the median is the middle element. In a matrix form, however, we cannot directly access the middle element since the numbers are spread across rows, but we know that exactly half of the numbers are less than or equal to it, and half are greater than it.

Since each row is sorted, we can make use of this property to apply binary search, but not on the elements directly. Instead, we use binary search on a range of possible values (since we don't have a flat sorted list). We use `bisect` library in Python which offers a way to determine the position where a number would be inserted to maintain order, which is `bisect_right`, and `bisect_left` to return the smallest number such that a given condition is met.

Here's how it works:

1. **Define a search space**: The problem hints at the range of elements by suggesting the use of `range(10**6 + 1)`. This means each element can be in the range from `0` to `10**6`.
2. **Define the target**: Because there's an odd number of elements in the matrix, the median is the `(m*n + 1) / 2`th smallest element (since Python uses 0-based indexing, we use `target = (m * n + 1) >> 1` to get the index).
3. **Counting function**: We create a function `count(x)` that tells us the number of elements in the matrix which are less than or equal to `x`. This uses `bisect_right` which counts how many elements in each row are less than or equal to `x`, and we sum these counts for all rows.
4. **Binary search**: We perform a binary search across the possible values of elements and use our `count(x)` function to determine if a value is too low or too high. We're looking for the smallest number `x` where `count(x)` is at least the target number. This `x` is the median.

In the code:

- `bisect_left(range(10**6 + 1), target, key=count)` is the binary search command.
- It looks through the sorted search space (`range(10**6 + 1)`) for the value such that at least `target` numbers in the matrix are less than or equal to it.
- `count` function is used as a key to guide the binary search using the count of the elements.

The intuition for why this works is that `bisect_left` will hone in on the value with enough smaller or equal elements in the rows, thanks to the sorted property of the rows, and the odd number of total elements ensures there's a definite middle element which is the median.

Solution Approach

The implementation relies heavily on the `bisect` module from Python's standard library, which is designed for binary searching within a sorted list. By conceptualizing the problem in terms of finding the median via binary search, we avoid having to sort or merge all of the matrix's elements, which enables us to achieve a solution that meets the required time complexity bound.

Here are the key components of the implementation:

1. **Binary Search Space**: The range of possible values each element in the matrix can take is from `0` to `10**6`. This is established as the binary search space, where `bisect_left` will search the number that is the median.
2. **Count Function**: A helper function `count(x)` is designed to compute the count of elements that are less than or equal to `x`. This uses `bisect_right`, which, when used on each row of the matrix, returns the index at which `x` should be inserted to keep the row sorted. Summing these indices across all rows gives the total count of elements less than or equal to `x`.
3. **Target**: Since there is an odd total number of elements in the matrix, the median is at the `((m * n) + 1) / 2`th smallest element. We right-shift by one bit to find the target index `target = (m * n + 1) >> 1` more efficiently (equivalent to integer division by 2).
4. **bisect_left**: The binary search is performed by `bisect_left(range(10**6 + 1), target, key=count)`. The `range(10**6 + 1)` gives us the space in which to look for our median value, `target` is the number of elements that must be smaller than or equal to our median, and `count` as the key-function verifies whether the current middle value in our binary search meets the condition of having `target` number of elements less than or equal to it.

To get a better understanding of how the `bisect_left` method operates here:

- `bisect_left` searches for the insertion point so as to maintain the sorted order. In this case, we are trying to find the smallest number such that there would be `target` numbers less than or equal to it in the sorted array representation of the matrix.
- It will go through the whole `range(10**6 + 1)` one interval at a time, zeroing in on where the median would be if we had one giant sorted list of the elements (but without actually creating this list). At each step of the binary search, `bisect_left` looks at the count of numbers less than or equal to the middle number it's currently considering and determines whether to search higher or lower.

The combination of these components leads to a solution that efficiently finds the median without directly sorting or flattening the matrix into a list, which would result in a time complexity greater than the allowed `O(m * n)`.

Example Walkthrough

Let's illustrate the solution approach with an example. We are given a `3x3` matrix with non-decreasing rows:

```
grid = [
  [1, 3, 5],
  [2, 4, 6],
  [3, 5, 7]
]
```

In this grid, `m = 3` and `n = 3`. There are `m * n = 9` elements, and we are looking for the median element, which is the `(9 + 1) / 2 = 5`th smallest element due to the odd number of elements.

1. **Define a search space:** The possible values for elements range from `0` to `10**6`. In practice, we can use the smallest and largest elements of the matrix as the bounds for binary search. Here, the search space is from `1` to `7`.
2. **Define the target:** We calculate the target as the median's position. Since there are 9 elements, the median will be the 5th smallest element, so `target = 5`.
3. **Counting function:** We need to count the number of elements less than or equal to some value `x`. If `x = 4`, then:
   - In the first row, there are 2 elements (`1, 3`) less than or equal to `4`.
   - In the second row, there are 2 elements (`2, 4`) less than or equal to `4`.
   - In the third row, there are 2 elements (`3, 4`) less than or equal to `4`. The total count for `x = 4` is `2 + 2 + 2 = 6`.
4. **Binary search:** We use `bisect_left` to perform a binary search within our search space (from `1` to `7`), looking for the smallest number `x` such that the count of numbers less than or equal to `x` is at least 5 (our `target`). We start the binary search:
   - Midpoint of `1` and `7` is `4`. Counting using our previous counting function, we find 6 elements less than or equal to `4`. Therefore, `4` may be too high—we need at least 5 elements, and we found 6—so we will now look for lower numbers.
   - Next we check the interval from `1` to `4`, with a midpoint of `2`. With `x = 2`, the total count of elements less than or equal to `2` is `1 + 1 + 1 = 3`. This is too low, so we need to search higher.
   - Now we check the interval from `3` to `4`, with midpoint `3`. With `x = 3`, the total count of elements less than or equal to `3` is `2 + 1 + 2 = 5`. Now we've found exactly `5` elements, which meets our target, and since we're looking for the smallest such number, `3` is our median.

Thus, using the solution approach, we've found that the median of the matrix is `3` without having to flatten and sort the entire matrix. The time complexity of this approach is `O(m * log(max-min))`, which is significantly less than `O(m * n)` for large matrices.

Python Solution

```python
import bisect

class Solution:
    def matrixMedian(self, grid: List[List[int]]) -> int:
        # Helper function to count numbers less or equal to x using binary search
        def count_less_or_equal(x):
            # Using bisect_right to find the insertion point which gives us
            # the count of elements less than or equal to 'x'
            return sum(bisect.bisect_right(row, x) for row in grid)

        # Get the dimensions of the matrix
        num_rows, num_cols = len(grid), len(grid[0])

        # Calculate the target position for median in the sorted array
        target = (num_rows * num_cols + 1) >> 1  # Equivalent to // 2

        # Function to determine the median by binary searching over the range of values
        # which could range from 1 to 10**6 considering constraints from the problem
        def find_median(left, right, target):
            while left < right:
                mid = left + (right - left) // 2
                if count_less_or_equal(mid) < target:
                    left = mid + 1
                else:
                    right = mid
            return left

        # Initialize the search range between the smallest and largest possible values
        # based on the problem constraints, then use binary search to find the median
        return find_median(1, 10**6, target)

```

Java Solution

```java
class Solution {
    private int[][] grid; // Initialize a private grid variable to store the input grid.

    // Method to find the median in a row-wise sorted matrix.
    public int matrixMedian(int[][] grid) {
        this.grid = grid; // Assign the passed grid to the class's grid variable.
        int numRows = grid.length; // The number of rows in the grid.
        int numCols = grid[0].length; // The number of columns in the grid.
        int target = (numRows * numCols + 1) / 2; // The median's position in the sorted order of elements.
        int minVal = 0; // Lower bound for the value of median.
        int maxVal = 1000010; // Upper bound for the value of median.

        // Binary search to find the median value in the matrix.
        while (minVal < maxVal) {
            int midVal = (minVal + maxVal) / 2; // Calculate mid value.
            if (countLessEqual(midVal) >= target) {
                // If count of elements less than or equal to midVal
                // is greater than or equal to target, narrow down the upper half.
                maxVal = midVal;
            } else {
                // If not, narrow down the lower half.
                minVal = midVal + 1;
            }
        }
        return minVal; // minVal is the median after the end of binary search.
    }

    // Helper method to count the number of elements less than or equal to x in the matrix.
    private int countLessEqual(int x) {
        int count = 0; // Counter for the number of elements less than or equal to x.
        // Iterate over each row in the grid.
        for (int[] row : grid) {
            int left = 0; // Left pointer of the binary search.
            int right = row.length; // Right pointer of the binary search.

            // Binary search to find the count of elements less than or equal to x in each row.
            while (left < right) {
                int mid = (left + right) / 2; // Calculate mid index.
                if (row[mid] > x) {
                    // If the current element is greater than x, narrow down the right part.
                    right = mid;
                } else {
                    // Else, narrow down the left part.
                    left = mid + 1;
                }
            }
            count += left; // Add the numbers less than or equal to x found in the current row to the count.
        }
        return count; // Return the total count.
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // Include algorithm for using upper_bound

class Solution {
public:
    // Function to find the median of the matrix
    int matrixMedian(vector<vector<int>>& matrix) {
        int rows = matrix.size();  // Number of rows in the matrix
        int cols = matrix[0].size();  // Number of columns in the matrix

        // Initializing search space for median
        int minValue = 0;
        int maxValue = 1000001; // Assuming 1e6 + 1 as the upper bound of the matrix values

        // Calculate the 'target' as the index after which we have the median
        int target = (rows * cols + 1) >> 1;  // Use bitwise shift for division by 2

        // Lambda function to count numbers less than or equal to 'x' using 'upper_bound'
        auto countLessOrEqual = [&](int x) {
            int count = 0;
            for (const auto& row : matrix) {
                // 'upper_bound' returns an iterator to the first element greater than 'x'
                count += (upper_bound(row.begin(), row.end(), x) - row.begin());
            }
            return count;
        };

        // Binary search to find the median
        while (minValue < maxValue) {
            int midValue = (minValue + maxValue) >> 1;  // Finding the mid value for binary search

            // If count of elements less than or equal to 'midValue' is at least 'target'
            // then median is at 'midValue' or before it.
            if (countLessOrEqual(midValue) >= target) {
                maxValue = midValue;  // Continue to search in the left half
            } else {
                minValue = midValue + 1;  // Continue to search in the right half
            }
        }

        // 'minValue' will hold the median value of the matrix after binary search
        return minValue;
    }
};

```

Typescript Solution

```typescript
// Importing array utilities from lodash for operations like 'upperBound'
import { upperBound } from 'lodash';

// Define a function to find the median of the matrix
function matrixMedian(matrix: number[][]): number {
    const rows: number = matrix.length; // Number of rows in the matrix
    const cols: number = matrix[0].length; // Number of columns in the matrix

    // Initializing the search space for the median
    let minValue: number = 0;
    let maxValue: number = 1000001; // Assuming 1e6 + 1 as the upper limit for matrix values

    // Calculate the 'target' index that represents the median position
    const target: number = ((rows * cols) + 1) >> 1; // Use bitwise shift for division by 2

    // Define a function to count how many numbers in the matrix are less than or equal to 'x'
    const countLessOrEqual = (x: number): number => {
        return matrix.reduce((count, row) => {
            // Find the index of the first element that is greater than 'x'
            // upperBound from lodash acts similarly to 'upper_bound' in C++
            count += upperBound(row, x);
            return count;
        }, 0);
    };

    // Perform a binary search to find the median
    while (minValue < maxValue) {
        const midValue: number = (minValue + maxValue) >> 1; // Calculate the mid-value

        // If the count of elements less than or equal to 'midValue' is at least 'target'
        // then the median must be 'midValue' or smaller.
        if (countLessOrEqual(midValue) >= target) {
            maxValue = midValue; // Search in the left half
        } else {
            minValue = midValue + 1; // Search in the right half
        }
    }

    // 'minValue' will hold the median value of the matrix after finishing the binary search
    return minValue;
}

```

Time and Space Complexity

The time complexity of the provided code is `O(log(MaxElement) * m * n)` where `MaxElement` is the maximum value that an element can take in the grid, `m` is the number of rows and `n` is the number of columns in the grid.

The reason for this complexity is that the binary search is performed over a range of `[0, 10^6]` which requires `log(10^6)` time. During each step of the binary search, a count of elements less than the current value (`x`) is performed across all rows, which takes `O(m * n)` time (`m` calls to `bisect_right`, each potentially iterating over up to `n` elements).

The space complexity of the code is `O(1)`, since no additional space is used that's based on the input size. The only variables in use are for counters and indices, which require a constant amount of space.