1901. Find a Peak Element II

Medium Array Binary Search Matrix

Problem Description

In this problem, we are given a 2D grid `mat` which is represented by an `n x m` matrix, where `n` is the number of rows and `m` is the number of columns. The task is to find a **peak element** in this grid. A peak element is defined as one that is strictly greater than all of its **adjacent** neighbors to the left, right, top, and bottom. The twist here is that this must be achieved under a strict time complexity of either `O(n log(m))` or `O(m log(n))`, which means we cannot simply iterate over each element of the grid.

To make the problem more challenging, there are a few constraints:

- The matrix uses **zero-based indexing**, which is a common representation for arrays and grids in programming.
- Adjacent cells in the matrix are guaranteed to have different values (no two adjacent cells are equal), simplifying the search for peak elements since a peak won't be 'flanked' by equal values.
- The entire matrix is imagined to be surrounded by an **outer perimeter** with the value `-1` in each cell, ensuring that edge elements can also be considered peaks if they are greater than all their in-bounds neighbors.

The output of the problem should be the coordinates `[i, j]` of any peak element, in the form of a two-element array.

Intuition

To find a peak element efficiently without checking every single element in the matrix, we can leverage the fact that a row's maximum element is greater than the elements directly to its left and right (since no two adjacent elements are equal). Then the idea is to identify if this maximum element is also a peak in the vertical direction. To exploit this property, binary search comes into play, as it allows us to significantly reduce the number of elements we inspect in order to find a peak.

We apply binary search with the following reasoning:

1. Pick the middle row of the current search interval and find the column index of its maximum element, say it's at column `j`.
2. Now check the element just above (if it exists) and below the maximum element found in the middle row. This is essentially checking whether this maximum element is also greater than its top and bottom adjacent cells.
3. If the element is greater than the element below (the element in the next row, same column), then a peak must exist in the upper part of the search interval, so we reduce the search interval to the upper half.
4. If the element is less than the element below, then a peak must exist in the lower part of the search interval, so we adjust the search interval to the lower half.
5. Continue this approach recursively until we narrow the search down to one row, then the maximum element in that row will be a peak element, because it is greater than its left and right neighbors by the definition of the maximum, and it is also greater than any top and bottom neighbor due to the binary search process.

This methodology guarantees that we find a peak in `O(n log(m))` time if the binary search is applied on rows (for every row picked, we find the max in `O(m)` time), or `O(m log(n))` time if applied on columns (again, finding the max in each column within `O(n)` time). The solution's effectiveness comes from never having to check all the elements and using the binary search to repeatedly halve the search space.

Solution Approach

To implement the solution to find a peak element in the matrix, we utilize binary search owing to its logarithmic time complexity, which fits the problem's requirement. The reference solution approach outlines a methodical strategy that uses binary search in a non-standard way, adapted to a 2D context.

Here's the step-by-step implementation breakdown:

1. **Initialize Search Space**: We start by defining the search space as the entire set of rows in the matrix with `l` (left) being the first row index (`0`) and `r` (right) being the last row index (`len(mat) - 1`).
2. **Binary Search on Rows**: We apply binary search on the row indices. The loop continues as long as `l < r`, implying that we have more than one row remaining in our search space.
3. **Find Local Maximum in Row**: In every iteration of the binary search, we calculate the middle row index `mid` between `l` and `r`. Then we find the column index `j` of the maximum value in the `mid`th row using `max(mat[mid])` and its associated index method `index()`. The column index `j` of this maximum value effectively serves as a candidate for the peak's column index.
4. **Compare with the Element Below**: We compare the found maximum value `mat[mid][j]` with the value directly below it `mat[mid + 1][j]` (since the matrix is surrounded by -1, we are assured that there are no out-of-bound issues).
5. **Adjust Search Space**: If `mat[mid][j]` is greater, then a peak must exist at or above this row, and we set `r = mid`. If not, a peak must be below it, and we update `l = mid + 1`.
6. **Convergence**: When `l` equals `r`, the while loop exits, indicating that we have narrowed down to a single row. Now, the maximum element of this row `mat[l]` is guaranteed to be a peak element, since it is greater than its left and right neighbors (as it is the maximum), and the binary search ensures it's greater than any top and bottom neighbors.
7. **Return Coordinates**: Finally, the coordinates of the peak element are the row index `l` and the column index of the maximum value in the `l`th row, found using `mat[l].index(max(mat[l]))`. The function then returns `[l, _]`.

The algorithm is efficient due to the binary search, and it effectively adheres to the required time complexity by limiting the elements it has to examine directly. No additional data structures are necessary, as we are simply manipulating indices and comparing elements directly within the given matrix.

Now, let's enclose key variables and code references within backticks to follow proper markdown convention for code formatting:

- Initialize search space: `l = 0, r = len(mat) - 1`
- Loop condition: `while l < r`
- Find local maximum: `j = mat[mid].index(max(mat[mid]))`
- Compare with element below: `if mat[mid][j] > mat[mid + 1][j]`
- Adjust search space: `r = mid` or `l = mid + 1`
- Convergence to single row: `when l == r`
- Return coordinates: `[l, mat[l].index(max(mat[l]))]`

Example Walkthrough

Let's illustrate the solution approach with a small example.

Suppose we have the following 3×4 matrix `mat`:

```
1  4  5  6
2  3  8  5
3  10 2  3
```

Step 1: *Initialize Search Space*

Here, `l = 0` and `r = 2` because we have three rows in total (0-indexed).

Step 2: *Binary Search on Rows*

We start with `l < r`, so we need to apply the binary search.

Step 3: *Find Local Maximum in Mid Row*

Calculate the middle row, `mid = (l + r) // 2` (using integer division). Initially, `l = 0` and `r = 2`, so `mid = 1`.

Now, we find the maximum in the middle row `mat[1]`, which is `8` at column index `j = 2`.

Step 4: *Compare with Element Below*

We compare the value `mat[mid][j]` with the value directly below it `mat[mid + 1][j]`.

In this case, `mat[mid][j] = 8` and `mat[mid + 1][j] = 2`. Since 8 is greater than 2, we are on the right path.

Step 5: *Adjust Search Space*

As our comparison shows that `mat[mid][j]` is greater than `mat[mid + 1][j]`, we set `r = mid`, which is now `r = 1`.

Step 6: *Convergence*

The loop continues, and now we find that `l = r`, meaning we are focused on a single row, which is the first one.

Step 7: *Return Coordinates*

We then find the maximum in the 1st row, `mat[1]`, which is `10` at column `j_l = 1`.

So finally, we return the coordinates `[l, j_l]`, which gives us `[1, 1]`, as the peak element's position.

Putting it into the code context:

- Updated search space: `r = 1`
- Loop termination: `l == r = 1`
- Final peak element's coordinates: `[1, 1]`

This example demonstrates the complete process from initialization to narrowing down the search interval using binary search until finding a peak element, without ever needing to check each element individually.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def findPeakGrid(self, mat: List[List[int]]) -> List[int]:
        # Define the row boundaries for the search
        low, high = 0, len(mat) - 1

        # Continue the binary search while the search space has more than 1 row
        while low < high:
            mid = (low + high) // 2  # Find the middle row
            max_col_index = mat[mid].index(max(mat[mid]))  # Find the column index of the maximum element in the middle row

            # Compare the max element of mid row with the element directly below it in the next row
            if mat[mid][max_col_index] > mat[mid + 1][max_col_index]:
                high = mid  # The peak lies on or before the mid row
            else:
                low = mid + 1  # The peak lies after the mid row

        # After exiting the loop, 'low' is the row where the peak element is located
        # Find the column index of the maximum element in the 'low' row
        peak_col_index = mat[low].index(max(mat[low]))

        # Return the position [row, column] of the peak element
        return [low, peak_col_index]

# Example usage:
# sol = Solution()
# print(sol.findPeakGrid([[1, 4], [3, 2]]))  # Output would be [0, 1], since 4 is the peak in this case
```

Java

```java
class Solution {
    public int[] findPeakGrid(int[][] mat) {
        // Initialize left and right pointers for the binary search on rows
        int left = 0, right = mat.length - 1;
        // Get the number of columns for later use
        int columns = mat[0].length;

        // Using binary search to find the peak row
        while (left < right) {
            int mid = left + (right - left) / 2; // Find the middle index between left and right
            int maxColumnIndex = findMaxPosition(mat[mid]); // Find the index of the peak in the middle row

            // Compare the peak element with the one right below it
            if (mat[mid][maxColumnIndex] < mat[mid + 1][maxColumnIndex]) {
                // If it's bigger, move the right pointer to the middle row
                right = mid;
            } else {
                // If it's not bigger, move the left pointer to one row below the middle row
                left = mid + 1;
            }
        }

        // After exiting the loop, 'left' is the index of the peak row. We find the column index of the peak in that row
        return new int[] {left, findMaxPosition(mat[left])};
    }

    // Helper function to find the index of the maximum element in a given array (row)
    private int findMaxPosition(int[] row) {
        int maxIndex = 0; // Assume the first element is the maximum initially
        // Iterate through the array
        for (int i = 1; i < row.length; ++i) {
            if (row[maxIndex] < row[i]) {
                // If we find a bigger element, update the index
                maxIndex = i;
            }
        }
        // Return the index of the largest element in the row
        return maxIndex;
    }
}
```

C++

```cpp
class Solution {
public:
    vector<int> findPeakGrid(vector<vector<int>>& matrix) {
        int left = 0; // Start index of the row search space
        int right = matrix.size() - 1; // End index of the row search space

        // Keep searching as long as the search space has more than one row
        while (left < right) {
            int mid = (left + right) / 2; // Find the mid index of the current search space

            // Find the column index of the maximum element in the middle row
            int maxColIndex = max_element(matrix[mid].begin(), matrix[mid].end()) - matrix[mid].begin();

            // If the maximum element in the middle row is also a peak element
            if (matrix[mid][maxColIndex] > matrix[mid + 1][maxColIndex]) {
                // Peak element must be in the upper half of the matrix
                right = mid;
            } else {
                // Peak element must be in the lower half of the matrix
                left = mid + 1;
            }
        }

        // After the while loop, left == right, and it points to the peak row
        // Find the column index of the maximum element in the peak row
        int peakColIndex = max_element(matrix[left].begin(), matrix[left].end()) - matrix[left].begin();

        // Return the position (row, column) of the peak element
        return {left, peakColIndex};
    }
};
```

TypeScript

```typescript
function findPeakGrid(mat: number[][]): number[] {
    let leftBound = 0; // Left boundary of the search space
    let rightBound = mat.length - 1; // Right boundary of the search space

    // Perform a binary search in the direction of the peak's row
    while (leftBound < rightBound) {
        let midRow = leftBound + Math.floor((rightBound - leftBound) / 2); // Middle index of the current search space
        let maxElementColIndex = mat[midRow].indexOf(Math.max(...mat[midRow])); // Column index of max element in the middle row

        // Compare the middle element with the one below it
        // If the middle element is greater, then a peak cannot be in the lower half, move the rightBound down
        if (mat[midRow][maxElementColIndex] > mat[midRow + 1][maxElementColIndex]) {
            rightBound = midRow;
        } else {
            // If the middle element is not greater, then the peak is in the lower half, move the leftBound up
            leftBound = midRow + 1;
        }
    }

    // leftBound will be the row where the peak element is present
    // Find the column index of the maximum element in the peak row
    let peakColIndex = mat[leftBound].indexOf(Math.max(...mat[leftBound]));

    // Return the coordinates of the peak
    return [leftBound, peakColIndex];
}
```

Python3

```python
from typing import List

class Solution:
    def findPeakGrid(self, matrix: List[List[int]]) -> List[int]:
        # Define the row boundaries for the search
        low, high = 0, len(matrix) - 1

        # Continue the binary search while the search space has more than 1 row
        while low < high:
            mid = (low + high) // 2  # Find the middle row
            max_col_index = matrix[mid].index(max(matrix[mid]))  # Find the column index of the maximum element in the middle row

            # Compare the max element of mid row with the element directly below it in the next row
            if matrix[mid][max_col_index] > matrix[mid + 1][max_col_index]:
                high = mid  # The peak lies on or before the mid row
            else:
                low = mid + 1  # The peak lies after the mid row

        # After exiting the loop, 'low' is the row where the peak element is located
        # Find the column index of the maximum element in the 'low' row
        peak_col_index = matrix[low].index(max(matrix[low]))

        # Return the position [row, column] of the peak element
        return [low, peak_col_index]

# Example usage:
# sol = Solution()
# print(sol.findPeakGrid([[1, 4], [3, 2]]))  # Output would be [0, 1], since 4 is the peak in this case
```

Time and Space Complexity

The time complexity of the provided code is `O(n * log m)`, where `n` is the number of rows and `m` is the number of columns in the matrix `mat`. This stems from using binary search on the rows, which has a complexity of `O(log n)`, combined with finding the maximum element in each middle row, which takes `O(m)`. Since these operations are done for each step of the binary search, they result in the mentioned time complexity.

The space complexity of the code is `O(1)`, indicating constant extra space usage irrespective of the input size. This is because the algorithm only uses a fixed amount of auxiliary space for variables like `l`, `r`, `mid`, and `j`, regardless of the dimensions of the input matrix.