1727. Largest Submatrix With Rearrangements

Medium Greedy Array Matrix Sorting

Leetcode Link

Problem Description

In this problem, we are given a binary matrix `matrix` with dimensions `m x n`, meaning it has `m` rows and `n` columns, where each element can either be `0` or `1`. The task is to rearrange the columns in any order such that we can find the largest submatrix composed entirely of `1`s. The output should be the area of this largest submatrix, i.e., the number of `1`s it contains.

Intuition

The intuition behind the solution relies on transforming the problem into a histogram problem, where each row of the matrix can be seen as the base of a histogram with heights denoting the number of consecutive `1`s above. To get this histogram from our binary matrix, we iterate over each element, and if it is a `1`, we add the value of the element above it, thereby accumulating the number of continuous `1`s in the column up to that row.

Once we have the histogram representation for each row, the next step is to sort each row in descending order. This sorting step helps us to ensure that, when calculating the potential area of the submatrix formed by `1`s, we always start with the tallest bar in the histogram (which corresponds to the longest consecutive sequence of `1`s in that reordered column).

After sorting, we determine the area of the largest rectangle we can form with the sorted sequence of bars in the histogram. Essentially, we iterate through each value in the sorted row, calculating the area as `current_height * current_width`, where `current_height` is the value of the bar (the sequence of `1`s in this column), and `current_width` is the current index in the sorted array plus one (since the array is zero-indexed). We keep a running maximum of these calculated areas, which will give us the area of the largest submatrix that can be formed by rearranging the columns optimally.

Solution Approach

The provided solution approach uses dynamic programming and sorting to find the area of the largest submatrix composed of `1`s after columns are optimally reordered. Here's a step-by-step analysis:

1. **Dynamic Programming:** The first part of the solution is to convert the binary matrix into a form that represents the continuous vertical `1`s as the height of histograms. We do this by updating the matrix in-place. For each cell that contains a `1`, we look directly above it (the cell in the previous row, same column). If the above cell is also a `1`, we accumulate the value. In other words, `matrix[i][j]` becomes `matrix[i][j] + matrix[i - 1][j]` if `matrix[i][j]` equals `1`.

   This first step is critical because it allows us to apply a histogram-based reasoning to the problem. Each row in the transformed matrix will represent a "base" and the numbers will represent the heights of "bars" that make up a histogram. In this transformed histogram, a rectangle of height `h` means that there are `h` continuous `1`s from the current row upwards in that column.

2. **Sorting:** After transforming each row into a histogram, we sort each row in descending order. This enables us to treat each row as a series of potentially "stackable" rectangles where each rectangle's height is the value of the cell and the width is how many cells we've counted so far.

3. **Calculating Maximum Area:** For each sorted row, we calculate the area that the rectangle would occupy if we used the cell as the smallest height in the rectangle (which is also the furthest left side of the rectangle). The area is calculated by multiplying the height of the bar (`i`, the value of the cell) by the width (`j`, the index of the value in the row plus one, because indices are zero-based but width counts are one-based).

   We iterate through each element in the row, and hence, for each element, we calculate the potential area of the submatrix if we were to stop at that particular column. Taking the maximum of these calculated areas gives us the maximum rectangular area for that particular permutation of rows.

The final answer is the largest area found after considering all such maximal rectangles for every row. This is a very efficient solution as it handles the problem with a complexity that is approximately O(m * n * log(n)), because the most expensive operation is sorting each row of the matrix, which takes O(n * log(n)) and is done `m` times.

Example Walkthrough

Let's illustrate the solution approach with a small example of a binary matrix. Consider the following `3 x 4` matrix:

```
matrix = [
    [1, 0, 1, 1],
    [1, 1, 1, 1],
    [0, 1, 1, 0]
]
```

Now, let's walk through the steps:

1. **Dynamic Programming:**

   - We start from the first row. Since there's no row above it, the values stay as they are.
   - Moving onto the second row, we add the value from the above cell if the current cell is `1`. After doing this for the second row, our matrix becomes:

   ```
   matrix = [
       [1, 0, 1, 1],
       [2, 1, 2, 2],
       [0, 1, 1, 0]
   ]
   ```

   - For the third row, we apply the same method. However, since the first and the last cells in the third row are `0`, they don't get accumulated and reset any potential stack of `1`s. Our matrix now looks like:

   ```
   matrix = [
       [1, 0, 1, 1],
       [2, 1, 2, 2],
       [0, 2, 3, 0]
   ]
   ```

2. **Sorting:**

   - Next, we sort each row in descending order to make 'bars' of 'histogram' aligned by their heights. This process will give us:

   ```
   matrix = [
       [1, 1, 1, 0],  // Sorted row 1
       [2, 2, 2, 1],  // Sorted row 2
       [3, 2, 0, 0]   // Sorted row 3
   ]
   ```

3. **Calculating Maximum Area:**

   - We now calculate the areas for each row by treating each cell as the height of a histogram bar, with the width being the index of the cell +1 (since indices are zero-based but widths are one-based).
   - For the first sorted row: `(1*1), (1*2), (1*3)`; max area = 3.
   - For the second sorted row: `(2*1), (2*2), (2*3), (1*4)`; max area = 6.
   - For the third sorted row: `(3*1), (2*2)`; max area = 4.

The largest area obtained from our calculations is `6`, and this is the solution to our problem. This means the largest submatrix composed entirely of `1`s that can be obtained by rearranging the columns of the original matrix has an area of `6`.

Python Solution

```python
from typing import List

class Solution:
    def largestSubmatrix(self, matrix: List[List[int]]) -> int:
        # Update the matrix such that each cell in the matrix represents the height
        # of the histogram that can be formed with the base at that cell
        for row in range(1, len(matrix)):
            for col in range(len(matrix[0])):
                # Increment the cell value by one if the top cell is 1;
                # otherwise, leave it as it is (i.e., 0).
                if matrix[row][col]:
                    matrix[row][col] = matrix[row - 1][col] + 1

        # Initialize the maximum area to 0
        max_area = 0

        # Iterate over each row to find the largest rectangular area possible
        for row in matrix:
            # Sort each row in descending order to facilitate the calculation
            # of the maximum rectangle in the histogram
            row.sort(reverse=True)

            # Iterate over each value in the sorted row to calculate the maximum area
            # Each value in the sorted row corresponds to the height of a bar in the histogram,
            # and its index represents potential width of the rectangle at that height
            for index, height in enumerate(row):
                # Calculate the width by adding 1 to the index since 'enumerate' is 0-based
                width = index + 1

                # Calculate the area of the rectangle that can be formed with this height
                area = width * height

                # Update the maximum area found so far
                max_area = max(max_area, area)

        # Return the maximum area found
        return max_area
```

Java Solution

```java
class Solution {
    public int largestSubmatrix(int[][] matrix) {
        // Get the dimensions of the matrix
        int rows = matrix.length;
        int cols = matrix[0].length;

        // Preprocess the matrix to count continuous ones vertically
        for (int i = 1; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                // If the current cell has a '1', add to the count from the cell above
                if (matrix[i][j] == 1) {
                    matrix[i][j] += matrix[i - 1][j];
                }
            }
        }

        // Initialize the variable to track the largest area of the submatrix
        int maxArea = 0;

        // Iterate over each row
        for (int[] row : matrix) {
            // Sort the row to group the column heights together
            Arrays.sort(row);

            // Iterate from the end of the row to find the maximum area
            for (int j = cols - 1, height = 1; j >= 0 && row[j] > 0; --j, ++height) {
                // Calculate the area of the submatrix ending at (i,j)
                int area = row[j] * height;
                // Update the maximum area
                maxArea = Math.max(maxArea, area);
            }
        }

        // Return the largest submatrix area found
        return maxArea;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int largestSubmatrix(vector<vector<int>>& matrix) {
        int numRows = matrix.size(); // Store the number of rows in the matrix
        int numCols = matrix[0].size(); // Store the number of columns in the matrix

        // Preprocess the matrix to compute the height of each column
        for (int i = 1; i < numRows; ++i) {
            for (int j = 0; j < numCols; ++j) {
                // If the current cell has a 1, add the value from the cell above.
                // This will end up with each cell containing the number of consecutive 1's above it + 1 for itself.
                if (matrix[i][j]) {
                    matrix[i][j] += matrix[i - 1][j];
                }
            }
        }

        int largestArea = 0; // Initialize the largest area found to be 0
        // Iterate through each preprocessed row to calculate the max area for each row
        for (auto& row : matrix) {
            // Sort the row in non-ascending order to maximize the potential area
            sort(row.begin(), row.end());

            // Iterate over each element of the row to calculate the area
            for (int j = 0; j < numCols; ++j) {
                // Update the largest area if the current area (height * width) is greater
                largestArea = max(largestArea, (j + 1) * row[j]);
            }
        }

        // Return the largest area found
        return largestArea;
    }
};
```

Typescript Solution

```typescript
function largestSubmatrix(matrix: number[][]): number {
    const numRows: number = matrix.length; // Store the number of rows in the matrix
    const numCols: number = matrix[0].length; // Store the number of columns in the matrix

    // Preprocess the matrix to compute the height of each column
    for (let i = 1; i < numRows; ++i) {
        for (let j = 0; j < numCols; ++j) {
            // If the current cell has a 1, add the value from the cell above.
            // Each cell stores the count of consecutive 1's above it + 1 (itself, if it's 1).
            if (matrix[i][j]) {
                matrix[i][j] += matrix[i - 1][j];
            }
        }
    }

    let largestArea: number = 0; // Initialize the largest area found to be 0

    // Iterate over each preprocessed row to calculate the max area for each row
    for (const row of matrix) {
        // Sort the row in non-ascending order to maximize the potential area
        // based on the previous calculated heights
        row.sort((a, b) => b - a); // Sorts the row in descending order

        // Calculate the area considering each column's height
        for (let j = 0; j < numCols; ++j) {
            // Calculate the area (height * width) and update the largest area if the current area is greater
            largestArea = Math.max(largestArea, (j + 1) * row[j]);
        }
    }

    // Return the largest area found
    return largestArea;
}
```

Time and Space Complexity

The given Python code first preprocesses the input matrix by computing the maximum height of a stack of `1`s ending at each cell. It then sorts these stack heights row by row and calculates the maximum area of a rectangle formed by these stacks. Now, let's break down the time and space complexities:

Time Complexity

1. The first `for` loop runs through the rows of the matrix, starting at the second row, taking `O(m)` time where `m` is the number of rows.
2. Inside this loop, the nested `for` loop runs through all the columns, `n` times for each row, resulting in `O(n)` time per row.
3. Multiplying the row iterations by the column iterations, we get `O(m * n)` for this preprocessing part.
4. The second part of the code, where each row is sorted in reverse order, takes `O(n log n)` time for each row, as the `.sort()` operation takes `O(n log n)` time.
5. As we apply the sorting to each of `n` rows, the total time for this step is `O(m * n log n)`.
6. After sorting, there is another nested loop structure, which runs through all the elements of the matrix (in their sorted row form) once again. However, the complexity of this part is `O(m * n)` times, since for each of `n` rows, we are iterating over `n` columns.
7. Combining all the parts, the dominant part of the time complexity is `O(m * n log n)`, since this term will outweigh `O(m * n)` for larger values of `n`.

Hence, the total time complexity of the algorithm is `O(m * n log n)`.

Space Complexity

1. The space complexity of the preprocessing step does not require additional space beyond the input matrix as it is performed in place, so this part is `O(1)`.
2. The sorting step does not use any extra space either, aside from the negligible internal space used by the sorting algorithm (which can typically be considered as constant space for purposes of complexity analysis).

Considering the above points, the space complexity of the code is `O(1)` additional space, with `matrix` itself taking `O(m * n)` but not counting towards additional space as it is the input.