661. Image Smoother

Easy Array Matrix

Leetcode Link

Problem Description

The task is to write an algorithm for an "image smoother," which is a filter operation applied to each cell of a grayscale image. The image is represented as a 2D integer matrix, where each cell contains a grayscale value. The smoother considers the value of a cell and the values of the eight surrounding cells to calculate an average. This average is then rounded down and placed into the corresponding cell in the resulting image. When cells on the edges or corners of the image do not have eight surrounding cells, the smoother only averages over the existing neighbouring cells. The challenge is to apply this filter correctly, ensuring that boundary conditions are handled and only valid neighbours are included in the average calculation.

Intuition

The solution approach involves a nested loop to traverse each cell in the image. For each cell, a sub-loop considers the surrounding cells within a distance of 1 cell in all directions. To manage cells at the edges and corners that have fewer neighbours, the algorithm checks if each potential neighbouring cell is within the image boundaries before including it in the average calculation.

The steps are:

1. Loop over each cell in the input image.
2. Initialize sum (`s`) and count (`cnt`) variables for each cell. The sum will hold the total grayscale value sum of the neighbours, and count will keep track of the number of valid neighbours considered.
3. Use a nested loop to go through each of the surrounding cells, including the cell itself.
4. Check if the neighbouring cell is within the image boundaries (its indexes are neither less than 0 nor exceeding the image dimensions).
5. If the cell is within bounds, add its value to the sum and increment the count.
6. After considering all valid neighbours, calculate the average by dividing the sum by the count.
7. Assign the rounded down average to the corresponding cell in the output image matrix.
8. Return the resulting image matrix after the smoother has been applied to all cells.

The solution effectively handles the averaging with the correct denominator for each cell, taking image boundary conditions into account.

Solution Approach

The implementation of the image smoother involves tackling the problem with a brute-force approach, where each cell's value is updated based on its neighbours.

Here's a detailed breakdown of the solution approach:

1. The algorithm starts by determining the dimensions of the image matrix, which are stored in variables `m` and `n`, representing the number of rows and columns, respectively.

2. An output matrix `ans` of the same dimensions as the input (`img`) is created to store the smoothed values. This is initialized to zero for all cells.

3. A nested `for` loop is used to traverse through each cell of the image matrix `img` using row index `i` and column index `j`.

4. For each cell (i,j), we initialize `s` and `cnt` to zero. Here `s` will accumulate the sum of the grayscale values of the current cell and its valid neighbours, while `cnt` will count the number of valid neighbouring cells considered in the smoothing operation (including the cell itself).

5. Another nested `for` loop iterates over the cells in the 3×3 block centered at (i, j). The ranges of these loops are `i-1` to `i+1` and `j-1` to `j+1`, attempting to cover all nine cells in the block including the center cell.

6. For every neighbouring cell, the algorithm checks if its coordinates `(x, y)` are within the image's boundaries using `0 <= x < m` and `0 <= y < n`. This ensures we do not attempt to access cells outside the array, which would result in an 'index out of range' error.

7. Only valid neighbours are included in the average calculation by adding their values to the `s` and incrementing the `cnt`.

8. Once all valid neighbours are considered, the algorithm computes the average value by dividing `s` by `cnt`. Integer division is used (`//` in Python), which gives us a rounded down result as required.

9. The calculated average is then assigned to the corresponding cell in the `ans` matrix.

10. After the completion of both nested loops, the matrix `ans` is fully populated with the smoothed values and is returned as the final result.

The simplicity of the brute-force approach makes it an uncomplicated solution that is easy to understand and implement. However, it does have a time complexity of O(m x n x k), where `k` is the number of neighbouring cells to consider (in this case, 9), because we are visiting each cell surrounding every cell in the matrix. Using brute force is acceptable here because the size of the smoothing filter is fixed and small.

Example Walkthrough

Let's illustrate the solution approach using a small example. Imagine we have the following 3×3 grayscale image matrix:

```
[ [100, 200, 100],
  [200, 300, 200],
  [100, 200, 100] ]
```

This matrix represents the grayscale values for a 3×3 image. Now, let's walk through the steps to apply the image smoother for the central cell (1,1) with the value 300.

1. We determine the dimensions of the image, which here are `m = 3` and `n = 3`.

2. We create an output matrix `ans` with the same dimensions, all initialized to zero:

```
[ [0, 0, 0],
  [0, 0, 0],
  [0, 0, 0] ]
```

3. Using a nested `for` loop, we traverse each cell. We'll focus on the central cell (i=1, j=1).

4. Initially, let `s = 0` and `cnt = 0`.

5. The nested `for` loop runs over the neighbours of the central cell. The loop ranges from `i-1` to `i+1` and `j-1` to `j+1`, meaning it covers all nine cells in the block.

6. We check each cell in this range to see if they are within the boundaries.

7. As all cells are within the boundary for the central cell, we add each cell's value to `s` and increment `cnt` for each cell. So we end up with:

```
  s = 100+200+100+200+300+200+100+200+100 = 1500
  cnt = 9 (including the central cell itself)
```

8. We calculate the average grayscale value by integer division of `s` by `cnt`, which is `1500 // 9 = 166`.

9. We place the average value in the central cell of the output:

```
[ [0,   0, 0],
  [0, 166, 0],
  [0,   0, 0] ]
```

10. After repeating steps 3-9 for all other cells and adjusting the boundary cases accordingly, the final smoothed image matrix becomes:

```
[ [150, 183, 150],
  [183, 166, 183],
  [150, 183, 150] ]
```

Here's how the boundary cases are averaged for top-left corner cell (i=0, j=0):

- `s = 100+200+200+300`, as only 4 cells (including itself) are valid neighbors.
- `cnt = 4`.
- The average is then `600 // 4 = 150`.
- The value `150` is placed in the top-left cell of the output matrix.

The same approach is repeated for other edge cells, considering the valid neighbors. Once all cells are processed, `ans` matrix is returned as the result of the smoothed image.

Python Solution

```python
class Solution:
    def imageSmoother(self, img: List[List[int]]) -> List[List[int]]:
        # Get the number of rows and columns in the image
        num_rows, num_cols = len(img), len(img[0])

        # Initialize an output image with the same dimensions, filled with zeros
        smoothed_image = [[0] * num_cols for _ in range(num_rows)]

        # Iterate through each pixel in the image
        for row in range(num_rows):
            for col in range(num_cols):
                # Initialize the sum and count of neighboring pixels
                pixel_sum = pixel_count = 0

                # Check all 9 positions in the neighborhood (including the pixel itself)
                for neighbor_row in range(row - 1, row + 2):
                    for neighbor_col in range(col - 1, col + 2):
                        # Ensure the neighbor is within image boundaries before including it
                        if 0 <= neighbor_row < num_rows and 0 <= neighbor_col < num_cols:
                            pixel_count += 1
                            pixel_sum += img[neighbor_row][neighbor_col]

                # Calculate the smoothed value for the current pixel
                # by taking the average of the sum of the neighborhood pixels
                smoothed_image[row][col] = pixel_sum // pixel_count

        # Return the smoothed image
        return smoothed_image

```

Java Solution

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

        // Initialize the smoothed image array
        int[][] smoothedImg = new int[rows][cols];

        // Iterate through each pixel in the image
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                int sum = 0; // Sum of pixel values in the smoothing window
                int count = 0; // Number of pixels in the smoothing window

                // Iterate through the neighboring pixels including the current pixel
                for (int x = i - 1; x <= i + 1; ++x) {
                    for (int y = j - 1; y <= j + 1; ++y) {
                        // Check if the neighbor is within the image boundaries
                        if (x >= 0 && x < rows && y >= 0 && y < cols) {
                            count++; // Increment the pixel count
                            sum += img[x][y]; // Add the pixel value to the sum
                        }
                    }
                }

                // Compute the average pixel value and assign it to the smoothed image
                smoothedImg[i][j] = sum / count;
            }
        }

        // Return the smoothed image
        return smoothedImg;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    vector<vector<int>> imageSmoother(vector<vector<int>>& image) {
        int rows = image.size();        // Number of rows in the image
        int cols = image[0].size();     // Number of columns in the image
        // Create a 2D vector with the same dimensions as the input image to store the result
        vector<vector<int>> smoothedImage(rows, vector<int>(cols));

        // Iterate through each cell in the image
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                int sum = 0;   // Sum of the surrounding cell values including itself
                int count = 0; // Counter for the number of cells included in the sum

                // Iterate through the neighboring cells centered at (i, j)
                for (int x = i - 1; x <= i + 1; ++x) {
                    for (int y = j - 1; y <= j + 1; ++y) {
                        // Check if the neighboring cell (x, y) is within the bounds of the image
                        if (x >= 0 && x < rows && y >= 0 && y < cols) {
                            count++;            // Increment the counter for each valid cell
                            sum += image[x][y];  // Add the cell value to the sum
                        }
                    }
                }
                // Compute the average value of the surrounding cells and assign to the corresponding cell in the result
                smoothedImage[i][j] = sum / count;
            }
        }
        // Return the resulting smooth image
        return smoothedImage;
    }
};

```

Typescript Solution

```typescript
function imageSmoother(img: number[][]): number[][] {
    // Get the dimensions of the image
    const rows = img.length;
    const cols = img[0].length;

    // Define the relative positions of the neighbors around a pixel
    const neighborOffsets = [
        [-1, -1], [-1, 0], [-1, 1],
        [0, -1],  [0, 0],  [0, 1],
        [1, -1],  [1, 0],  [1, 1]
    ];

    // Initialize the result image with the same dimensions
    const resultImage = new Array(rows).fill(0).map(() => new Array(cols).fill(0));

    for (let row = 0; row < rows; row++) {
        for (let col = 0; col < cols; col++) {
            let sum = 0; // Sum of the pixel values in the smooth box
            let count = 0; // Number of pixels in the smooth box

            // Iterate through all neighbors including the current pixel
            for (const [offsetRow, offsetCol] of neighborOffsets) {
                const neighborRow = row + offsetRow;
                const neighborCol = col + offsetCol;

                // Check if the neighbor is within the image boundaries
                if (neighborRow >= 0 && neighborRow < rows && neighborCol >= 0 && neighborCol < cols) {
                    sum += img[neighborRow][neighborCol]; // Add the neighbor's value to the sum
                    count++; // Increase the pixel count
                }
            }

            // Calculate the smoothed value and assign it to the result image
            resultImage[row][col] = Math.floor(sum / count);
        }
    }

    // Return the smoothed image
    return resultImage;
}

```

Time and Space Complexity

Time Complexity

The given code iterates through every cell of the `img` matrix, represented by dimensions `m` x `n`, where `m` is the number of rows and `n` is the number of columns. For each cell `(i, j)`, it considers up to 9 neighboring cells (including the cell itself) in a 3×3 grid. This nested iteration contributes a constant factor of 9 for each cell, because the innermost two loops (over `x` and `y`) run at most 3 times each.

Thus, the total time complexity is `O(m * n * 9)`, which simplifies to `O(m * n)` since `9` is a constant factor and does not affect the asymptotic complexity.

Space Complexity

The space complexity comes from the additional matrix `ans` that is created to store the smoothed values. It's the same size as the input matrix `img`, so the space complexity is `O(m * n)`.

No additional space is used that grows with the size of the input, as the variables `s` (sum of the neighbor values) and `cnt` (the count of neighbors considered) use constant space. Therefore, the total space complexity remains `O(m * n)`.