661. Image Smoother

Matrix

Problem Description

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 task is to write an algorithm for an "image smoother," which is a filter operation applied to each cell of a grayscale image. The

The solution approach involves a nested loop to traverse each cell in the image. For each cell, a sub-loop considers 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.

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

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.

used (// in Python), which gives us a rounded down result as required.

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

- 8. Return the resulting image matrix after the smoother has been applied to all cells.
- The implementation of the image smoother involves tackling the problem with a brute-force approach, where each cell's value is

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.

Solution Approach

updated based on its neighbours.

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.

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

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

Here's a detailed breakdown of the solution approach:

- (including the cell itself).
- and j-1 to j+1, attempting to cover all nine cells in the block including the center cell. For every neighbouring cell, the algorithm checks if its coordinates (x, y) are within the image's boundaries using 0 <= x <

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

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

Once all valid neighbours are considered, the algorithm computes the average value by dividing s by cnt. Integer division is

- range' error. Only valid neighbours are included in the average calculation by adding their values to the s and incrementing the cnt.
- 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 $O(m \times n \times k)$, 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.
- Let's illustrate the solution approach using a small example. Imagine we have the following 3×3 grayscale image matrix: [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. We determine the dimensions of the image, which here are m = 3 and n = 3. We create an output matrix ans with the same dimensions, all initialized to zero:

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

end up with:

[0, 0, 0],

[0, 166, 0],

[0, 0, 0]]

[150, 183, 150],

• cnt = 4.

Python

Java

class Solution {

Initially, let s = 0 and cnt = 0.

[[0, 0, 0],

[0, 0, 0],

[0, 0, 0]]

Example Walkthrough

[200, 300, 200],

[100, 200, 100]]

it covers all nine cells in the block. We check each cell in this range to see if they are within the boundaries.

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

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

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

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

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

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

• The average is then 600 // 4 = 150. The value 150 is placed in the top-left cell of the output matrix.

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.

smoothed_image = [[0] * num_cols for _ in range(num_rows)]

for neighbor row in range(row - 1, row + 2):

pixel count += 1

Initialize the sum and count of neighboring pixels

for neighbor col in range(col - 1, col + 2):

Calculate the smoothed value for the current pixel

smoothed_image[row][col] = pixel_sum // pixel_count

Check all 9 positions in the neighborhood (including the pixel itself)

pixel_sum += img[neighbor_row][neighbor_col]

by taking the average of the sum of the neighborhood pixels

Ensure the neighbor is within image boundaries before including it

if 0 <= neighbor row < num_rows and 0 <= neighbor_col < num_cols:</pre>

Iterate through each pixel in the image

pixel_sum = pixel_count = 0

for col in range(num cols):

for row in range(num rows):

Return the smoothed image

public int[][] imageSmoother(int[][] img) {

// Initialize the smoothed image array

int[][] smoothedImg = new int[rows][cols];

for (int j = 0; j < cols; ++j) {

for (int x = i - 1; $x \le i + 1$; ++x) {

count++:

smoothedImage[i][j] = sum / count;

// Define the relative positions of the neighbors around a pixel

const resultImage = new Array(rows).fill(0).map(() => new Array(cols).fill(0));

let sum = 0; // Sum of the pixel values in the smooth box

Calculate the smoothed value for the current pixel

smoothed_image[row][col] = pixel_sum // pixel_count

by taking the average of the sum of the neighborhood pixels

let count = 0; // Number of pixels in the smooth box

// Initialize the result image with the same dimensions

for (let col = 0: col < cols: col++) {</pre>

// Return the resulting smooth image

function imageSmoother(img: number[][]): number[][] {

return smoothedImage;

// Get the dimensions of the image

[-1, -1], [-1, 0], [-1, 1],

[0, -1], [0, 0], [0, 1],

[1, -1], [1, 0], [1, 1]

const rows = img.length;

const neighborOffsets = [

const cols = img[0].length;

for (int v = i - 1; $v \le i + 1$; ++v) {

int sum = 0: // Sum of the surrounding cell values including itself

int count = 0; // Counter for the number of cells included in the sum

// Check if the neighboring cell (x, y) is within the bounds of the image

sum += image[x][y]; // Add the cell value to the sum

// Increment the counter for each valid cell

// Compute the average value of the surrounding cells and assign to the corresponding cell in the result

// Iterate through the neighboring cells centered at (i, j)

if $(x >= 0 \&\& x < rows \&\& v >= 0 \&\& v < cols) {$

// Get the dimensions of the image

return smoothed_image

int rows = img.length;

int cols = img[0].length;

returned as the result of the smoothed image.

Solution Implementation

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

```
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
```

The same approach is repeated for other edge cells, considering the valid neighbors. Once all cells are processed, ans matrix is

// Iterate through each pixel in the image for (int i = 0; i < rows; ++i) { for (int i = 0; i < cols; ++i) {

```
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 \le i + 1; ++x) {
                    for (int y = i - 1; y \le i + 1; ++y) {
                        // Check if the neighbor is within the image boundaries
                        if (x >= 0 \&\& x < rows \&\& v >= 0 \&\& v < 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++
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 (let row = 0; row < rows; row++) {</pre>
```

1;

};

TypeScript

```
// 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 += imq[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;
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:</pre>
                           pixel count += 1
                           pixel_sum += img[neighbor_row][neighbor_col]
```

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

Time Complexity

Return the smoothed image

return smoothed_image

Time and Space Complexity

most 3 times each. Thus, the total time complexity is 0(m * n * 9), which simplifies to 0(m * n) since 9 is a constant factor and does not affect the asymptotic complexity. **Space Complexity**

grid. This nested iteration contributes a constant factor of 9 for each cell, because the innermost two loops (over x and y) run at

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 0(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 0(m * n).