2428. Maximum Sum of an Hourglass

Problem Description

You are given a matrix of integers with dimensions `m x n`. The task is to find the maximum sum of an "hourglass" shape within this matrix. An hourglass in this context is defined as a subset of the matrix with the following form:

```
1 X X X
2   X
3 X X X
```

Here, `X` represents the elements which are part of the hourglass. To get the sum of an hourglass, you add up all the `X`s. The goal is to find the hourglass with the highest sum within the given matrix. Take note that the hourglass should be fully contained within the matrix and cannot be rotated.

Intuition

The intuition behind the solution is based on systematic exploration. Since the hourglass has a fixed shape, we can deduce that we need to scan through the matrix by moving the center of the hourglass from one feasible position to another. The 'feasible positions' are those where an hourglass can be fully contained in the matrix. This means that the center cannot be on the border; it has to be at least one row and one column away from the edges.

Once we have determined where the center of an hourglass can be, we can calculate the sum of the elements for each hourglass configuration. To avoid redundant calculations, we calculate the sum of the full block of 3×3 and then subtract the values that do not belong to the hourglass shape (the corners).

We keep track of the maximum sum we find while iterating over all possible positions for the center of the hourglass. This method ensures that we check every possible hourglass and find the one with the maximum sum, which is our final answer.

Solution Approach

The algorithm for solving this problem is straightforward and doesn't require complex data structures or advanced patterns. Here's a step-by-step breakdown of the solution:

1. We first initialize a variable `ans` to store the maximum sum of any hourglass found during the traversal of the matrix.
2. The problem is then approached by considering each element of the matrix that could potentially be the center of an hourglass. We must ensure that these centers are not on the border of the matrix because an hourglass cannot fit there. Thus, we start iterating from the second row and column (`(1,1)` considering 0-based indexing) up to the second-to-last row and column (`(m - 2, n - 2)`).
3. For each possible center of the hourglass at position `(i, j)`, we compute the sum of the hourglass by adding up all the elements in the 3×3 block centered at `(i, j)` by using a nested loop or a sum with a comprehension list.
4. After getting the sum of the entire 3×3 block, we need to subtract the elements that don't belong to the hourglass. These are the elements at positions `(i, j - 1)` and `(i, j + 1)`. So we subtract the values at these positions from the sum of the 3×3 block to get the correct hourglass sum.
5. We then compare this sum with our current maximum value stored in `ans` and update `ans` if the current sum is greater.
6. After we have completed the traversal, the `ans` variable contains the highest sum of any hourglass in the matrix. This value is returned as the final answer.

The code provided makes use of list comprehensions for summing elements and simple for-loops for traversal. The Python `max` function is utilized to keep track of the maximum value, and the nested loops along with indexing allow for accessing matrix elements.

In essence, the algorithm is O(m*n), where `m` is the number of rows and `n` is the number of columns in the matrix, because we need to check each potential center for the hourglass. This is an example of a brute-force approach since it checks all possible hourglass configurations in the matrix. Despite the brute-force nature, it is efficient enough for the problem's constraints because the size of an hourglass is constant and small.

Example Walkthrough

Let's walk through a simple example to illustrate the solution approach. Suppose we are given the following 3×4 matrix of integers:

```
1 1 1 1 0
2 0 1 0 0
3 1 1 1 0
```

According to our solution approach, we must find the maximum sum of an "hourglass" shape within this matrix. Here, the potential centers for the hourglass are marked with `C` in the matrix:

```
1 1 1 1 0
2 0 C 0 0
3 1 1 1 0
```

We can see that there is only one feasible center at position `1,1` (0-based indexing), since it is the only position that allows a full hourglass to be contained within the matrix. We will now compute the sum of the hourglass centered at this position.

1. Start at the center position `(1,1)`.
2. Consider the 3×3 block centered at `(1,1)`, which includes all the cells adjacent to it directly or diagonally. For this hourglass, the block is:

   ```
   1 1 1
   2 0 1 0
   3 1 1
   ```

3. Now, to find the sum of the hourglass, we add up the elements within this 3×3 block, except for the corners that are not part of the hourglass:

   ```
   1 1 + 1 + 1
   2   + 1 +
   3 1 + 1 + 1
   ```

   The sum of the hourglass is `1 + 1 + 1 + 1 + 1 + 1 + 1 = 7`.

4. Since there is only one feasible hourglass in this example, the maximum sum is `7`.
5. We can conclude that the maximum hourglass sum in the given matrix is `7`.

This illustrates the solution approach where we systematically explore each potential center and calculate the corresponding hourglass sum to find the maximum. In this case, the answer is straightforward since there's only one possible hourglass. However, in a larger matrix, we would iterate over every element that could be the center of an hourglass while avoiding the border elements. After iterating through all potential centers, we would compare the sums and return the maximum one found.

Python Solution

```python
class Solution:
    def maxSum(self, grid: List[List[int]]) -> int:
        # Get the dimensions of the grid
        num_rows, num_columns = len(grid), len(grid[0])

        # Initialize the maximum hourglass sum to 0
        max_hourglass_sum = 0

        # Traverse the grid, avoiding the borders since hourglasses extend beyond a single point
        for row in range(1, num_rows - 1):
            for col in range(1, num_columns - 1):
                # Calculate the sum of the current hourglass
                hourglass_sum = (
                    grid[row - 1][col - 1] +
                    grid[row - 1][col] +
                    grid[row - 1][col + 1] +
                    grid[row][col] + # The center of the hourglass
                    grid[row + 1][col - 1] +
                    grid[row + 1][col] +
                    grid[row + 1][col + 1]
                )

                # Update the maximum hourglass sum if the current one is greater
                max_hourglass_sum = max(max_hourglass_sum, hourglass_sum)

        # Return the maximum hourglass sum found
        return max_hourglass_sum
```

Java Solution

```java
class Solution {
    // Computes the maximum sum of any hourglass-shaped subset in a 2D grid.
    public int maxSum(int[][] grid) {
        // Dimensions of the grid
        int rows = grid.length;
        int columns = grid[0].length;

        // Variable to hold the maximum sum of hourglass found so far
        int maxHourglassSum = 0;

        // Loop through each cell, but avoid the edges where hourglass cannot fit
        for (int x = 1; x < rows - 1; ++x) {
            for (int y = 1; y < columns - 1; ++y) {
                // Compute the sum of the current hourglass
                // Initialize the sum with the negative value of the center elements to the left and right
                // Since we are going to add all nine cells, subtracting the unwanted cells in advance
                // prevents them from being included in the final hourglass sum.
                int hourglassSum = -grid[x][y - 1] - grid[x][y + 1];

                // Add the sum of the entire 3x3 block around the current cell
                for (int i = x - 1; i <= x + 1; ++i) {
                    for (int j = y - 1; j <= y + 1; ++j) {
                        hourglassSum += grid[i][j];
                    }
                }

                // Update the maximum sum if the current hourglass sum is greater than the maximum sum found so far
                maxHourglassSum = Math.max(maxHourglassSum, hourglassSum);
            }
        }

        // Return the maximum sum of an hourglass found in the grid
        return maxHourglassSum;
    }
}
```

This code snippet is for a class named `Solution` containing a method `maxSum` that calculates the maximum sum of an "hourglass" in a 2-dimensional grid. An hourglass shape is defined by a pattern of cells from the grid in the following form:

```
a b c
  d
e f g
```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // For std::max

using namespace std;

class Solution {
public:
    int maxSum(vector<vector<int>>& grid) {
        // Get the number of rows 'm' and columns 'n' from the grid
        int numRows = grid.size(), numCols = grid[0].size();

        // Initialize the variable to hold the maximum sum of an hourglass
        int maxHourglassSum = 0;

        // Iterate over each potential center of an hourglass
        for (int row = 1; row < numRows - 1; ++row) {
            for (int col = 1; col < numCols - 1; ++col) {
                // Calculate the sum of the current hourglass, which includes:
                // - The sum of the top row (3 cells)
                // - The center cell
                // - The sum of the bottom row (3 cells)
                int hourglassSum = grid[row - 1][col - 1] + grid[row - 1][col] + grid[row - 1][col + 1]
                                   + grid[row][col]
                                   + grid[row + 1][col - 1] + grid[row + 1][col] + grid[row + 1][col + 1];

                // Update the maximum hourglass sum found so far
                maxHourglassSum = max(maxHourglassSum, hourglassSum);
            }
        }

        // Return the maximum hourglass sum
        return maxHourglassSum;
    }
};
```

Typescript Solution

```typescript
function maxSum(grid: number[][]): number {
    // Get the row count (m) and column count (n) of the grid.
    const rowCount = grid.length;
    const colCount = grid[0].length;

    // Initialize the variable to store the maximum sum of the hourglass.
    let maxHourglassSum = 0;

    // Loop over the internal cells where an hourglass can be formed,
    // excluding the borders because an hourglass shape cannot fit there.
    for (let row = 1; row < rowCount - 1; ++row) {
        for (let col = 1; col < colCount - 1; ++col) {
            // Initialize the sum of the current hourglass.
            // Starting with the center value negation, as it's added twice in the nested loop.
            let hourglassSum = -grid[row][col];

            // Sum values of the hourglass, three rows and three columns at a time.
            for (let x = row - 1; x <= row + 1; ++x) {
                for (let y = col - 1; y <= col + 1; ++y) {
                    hourglassSum += grid[x][y];
                }
            }

            // Update the maximum sum with the higher of the two values: the current max and the current hourglass sum.
            maxHourglassSum = Math.max(maxHourglassSum, hourglassSum);
        }
    }

    // Return the maximum hourglass sum found.
    return maxHourglassSum;
}
```

In this code, the function `maxSum` calculates the maximum sum of any hourglass-shaped sum in a 2D grid array where each hourglass shape is formed by a subset of values in a pattern that resembles an hourglass:

```
a b c
  d
e f g
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is primarily determined by the two nested loops that iterate over the elements of the grid, excluding the outermost rows and columns. For each element `grid[i][j]` located inside these bounds (thus `(m-2)*(n-2)` elements), we are computing the sum of the 3×3 hourglass centered at `grid[i][j]`. This involves adding up 9 values for each valid `i` and `j`.

Therefore, we are performing a constant amount of work (specifically, 9 additions and 2 subtractions) for each hourglass shape. Given that there are `(m-2)*(n-2)` such hourglasses, the overall time complexity is `O((m-2)*(n-2)+9)`, which simplifies to `O(m*n)` since the constant factor of 9 can be dropped in big O notation.

Space Complexity

The space complexity of the code is `O(1)`, as it only uses a fixed number of variables and does not allocate any additional space that grows with the input size. The variable `ans` is used to keep track of the maximum sum, and temporary variables like `x`, `i`, `j`, `v`, and `y` are of a constant number as well. This code makes in-place calculations and does not require any extra space for data structures like arrays or lists that would depend on the input size.