2428. Maximum Sum of an Hourglass

Prefix Sum

Problem Description

<u>Array</u>

Matrix

Medium

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:

X X XX 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 Xs. 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.

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

ntuition

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.

The problem is then approached by considering each element of the matrix that could potentially be the center of an

Here's a step-by-step breakdown of the solution:

and column ((m - 2, n - 2)).

returned as the final answer.

the size of an hourglass is constant and small.

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

We first initialize a variable ans to store the maximum sum of any hourglass found during the traversal of the matrix.

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.

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.

We then compare this sum with our current maximum value stored in ans and update ans if the current sum is greater.

After we have completed the traversal, the ans variable contains the highest sum of any hourglass in the matrix. This value is

- 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
- 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

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 0 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 0

the block is:

of the hourglass:

Solution Implementation

max_hourglass_sum = 0

class Solution:

1 + 1 + 1

1 + 1 + 1

0 C 0 0

1 1 1 0

Example Walkthrough

elements.

1 1 1 0 1 0 1 1 1

This illustrates the solution approach where we systematically explore each potential center and calculate the corresponding

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.

hourglass sum to find the maximum. In this case, the answer is straightforward since there's only one possible hourglass.

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

Consider the 3×3 block centered at (1,1), which includes all the cells adjacent to it directly or diagonally. For this hourglass,

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.

The sum of the hourglass is 1 + 1 + 1 + 1 + 1 + 1 + 1 = 7. Since there is only one feasible hourglass in this example, the maximum sum is 7.

def maxSum(self, grid: List[List[int]]) -> int:

for row in range(1, num_rows - 1):

Return the maximum hourglass sum found

return max_hourglass_sum

public int maxSum(int[][] grid) {

// Dimensions of the grid

int columns = grid[0].length;

for (int i = 1; i < rows - 1; ++i) {

for (int j = 1; j < columns - 1; ++j) {

int rows = grid.length;

int maxHourglassSum = 0;

Start at the center position (1,1).

Python

We can conclude that the maximum hourglass sum in the given matrix is 7.

Get the dimensions of the grid num_rows, num_columns = len(grid), len(grid[0]) # Initialize the maximum hourglass sum to 0

Traverse the grid, avoiding the borders since hourglasses extend beyond a single point

Update the maximum hourglass sum if the current one is greater

max_hourglass_sum = max(max_hourglass_sum, hourglass_sum)

// Computes the maximum sum of any hourglass-shaped subset in a 2D grid.

// Variable to hold the maximum sum of hourglass found so far

// Compute the sum of the current hourglass

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

for (int y = j - 1; $y \le j + 1$; ++y) {

hourglassSum += grid[x][y];

// Return the maximum sum of an hourglass found in the grid

// Update the maximum hourglass sum found so far

// Return the maximum hourglass sum

// Get the row count (m) and column count (n) of the grid.

for (let row = 1; row < rowCount - 1; ++row) {</pre>

for (let col = 1; col < colCount - 1; ++col) {</pre>

let hourglassSum = -grid[row][col];

// Initialize the variable to store the maximum sum of the hourglass.

// excluding the borders because an hourglass shape cannot fit there.

// Starting with the center value negation, as it's added twice in the nested loop.

// Sum values of the hourglass, three rows and three columns at a time.

// Loop over the internal cells where an hourglass can be formed,

// Initialize the sum of the current hourglass.

for (let y = col - 1; y <= col + 1; ++y) {

for (let x = row - 1; $x \le row + 1$; ++x) {

hourglassSum += grid[x][y];

return maxHourglassSum;

function maxSum(grid: number[][]): number {

const rowCount = grid.length;

let maxHourglassSum = 0;

const colCount = grid[0].length;

maxHourglassSum = max(maxHourglassSum, hourglassSum);

// Loop through each cell, but avoid the edges where hourglass cannot fit

// Add the sum of the entire 3x3 block around the current cell

maxHourglassSum = Math.max(maxHourglassSum, hourglassSum);

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]

```
// 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[i][j - 1] - grid[i][j + 1];
```

Java

class Solution {

```
return maxHourglassSum;
This code snippet is for a class named `Solution` containing a method `maxSum` that calculates the maximum sum of an "hourglass"
a b c
  d
e f g
C++
#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) {</pre>
            for (int col = 1; col < numCols - 1; ++col) {</pre>
                // 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 sum if the current hourglass sum is greater than the maximum sum found so far

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

};

TypeScript

```
// 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 hourglas
  a b c
    d
  e f g
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
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,

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 0((m-2)*(n-2)*9), which simplifies to 0(m*n)since the constant factor of 9 can be dropped in big O notation.

excluding the outermost rows and columns. For each element grid[i][j] located inside these bounds (thus (m-2)*(n-2)

Space Complexity

The space complexity of the code is 0(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 s, i, j, x, 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.