3071. Minimum Operations to Write the Letter Y on a Grid

```
Hash Table
            Counting
```

part of the grid, these values can't be the same.

this specific combination of values i and j.

<u>Array</u>

Problem Description

We have to transform this grid such that it forms the letter 'Y'. A grid forms the letter 'Y' when:

Matrix

center down to the bottom) contain the same value. 2. All cells that are not part of the letter 'Y' also contain the same value, which must be distinct from the value of the 'Y' cells.

In this problem, we are provided with an odd-sized, 0-indexed n x n grid containing three possible values in each cell: 0, 1, or 2.

1. All cells that are part of the letter 'Y' (the two diagonals converging at the center cell from the top left and top right and the vertical line from the

3. A cell's value can be changed to either 0, 1, or 2 in one operation.

Our goal is to find the *minimum* number of such operations needed to write the letter 'Y' on the grid according to these rules.

Intuition

The solution approach leverages the concept of counting and enumeration. Knowing that all cells of the 'Y' must share one value

(let's call it 'i'), and all cells not a part of 'Y' must share a different value ('j'), there are limited possibilities since we have only three different values to choose from (0, 1, or 2). Moreover, when choosing a value for the 'Y' part and a different value for the non-'Y'

Medium

We build two count dictionaries: cnt1 keeps track of how many times each value appears within the cells that make up the letter 'Y', and cnt2 does the same for the cells outside of the letter 'Y'. Then, for each pair of values (i for the 'Y', j for the rest) we calculate the number of operations that would be needed if we chose this pair of values. This is done simply by subtracting the number of already correctly valued cells from the total cell count, which gives us the number of cells we need to change.

Solution Approach The implementation of the solution follows a counting and enumeration approach as indicated in the Reference Solution Approach.

Firstly, two Counter objects from Python's collections module, cnt1 and cnt2, are used. They serve as the data structures to

The solution iterates over each cell in the grid using nested loops. For each cell at position (i, j), it determines if the cell belongs to the letter 'Y'. This classification is done using three conditions, indicating the cell is part of one of the three parts of

the 'Y': • a = i == j and i <= n // 2: Checks if the cell is on the diagonal from the top-left to the center cell.

keep track of the frequency of each value in the cells that belong to the 'Y' (cnt1) and those that do not (cnt2).

• b = i + j == n - 1 and $i \ll n // 2$: Checks if the cell is on the diagonal from the top-right to the center cell. • c = j == n // 2 and i >= n // 2: Checks if the cell is part of the vertical line from the center down.

Using logical OR (a or b or c), we can ascertain if a cell is part of the 'Y'. If it is, increment the count for the value in cnt1,

otherwise in cnt2. In the minimizing step, we generate permutations of two different values, i and j (with i != j), where i is a candidate value for 'Y' cells, and j is for the non-'Y' cells. We subtract the sum of the count of already correct 'Y' cells (cnt1[i]) and the count of

As we want to minimize the number of operations, we take the minimum over all (i, j) pairs using a generator expression:

already correct non-'Y' cells (cnt2[j]) from the total number of cells in the grid (n * n). This gives us the count of operations for

min(n * n - cnt1[i] - cnt2[j] for i in range(3) for j in range(3) if i != j By iterating over all value combinations and taking the minimum of the computed operations, we determine the least amount of

changes needed to form the letter 'Y' on the grid according to the problem's conditions.

Counting Values: We use two Counter objects, cnt1 for the 'Y' and cnt2 for the non-'Y'.

Let's illustrate the solution approach using a small 3×3 grid example. Our grid looks like this: 1 0 1 0 2 0 1 0 1

Initially, we determine the 'Y' cells are located at indices (0,0), (0,2), (1,1), (2,1). The rest of the cells are not part of the 'Y'.

Value 1 at indices (0,0) and (0,2), so cnt1[1] = 2. Value 2 at index (1,1), so cnt1[2] = 1.

Example Walkthrough

Now, following the solution approach:

For cnt1 ('Y' cells), the grid has:

For cnt2 (non-'Y' cells), the grid has:

No other values are present in the non-'Y' cells.

Proceeding similarly for other combinations:

Value 0 at indices (0,1), (1,0), (1,2), and (2,0), (2,2), so cnt2[0] = 5.

• Operations needed: 3 * 3 - cnt1[1] - cnt2[0] = 9 - 2 - 5 = 2.

• i = 2, j = 0: Operations = 9 - cnt1[2] - cnt2[0] = <math>9 - 1 - 5 = 3.

```
ensuring i != j. For each combination, we calculate the needed operations.
   For i = 1 (value for 'Y') and j = 0 (value for non-'Y'):
```

• i = 1, j = 2: Operations = 9 - cnt1[1] - cnt2[2] = 9 - 2 - 0 = 7 (since cnt2[2] is not present).

• i = 2, j = 1: Operations = 9 - cnt1[2] - cnt2[1] = 9 - 1 - 0 = 8 (since cnt2[1] is not present).

Calculate the minimum number of operations required to write the letter Y on the grid.

Iterate over the grid to count the occurrences in the "Y" shape and elsewhere

Calculate which part of the grid is the 'Y' shape

is_vert_or_first_diag = i == j and i <= n // 2</pre>

is horiz middle = j == n // 2 and i >= n // 2

is_second_diag = i + j == n - 1 and i <= n // 2

Determining Operations: We enumerate the permutations of different values for 'Y' (i) and non-'Y' (j) from the set {0, 1, 2},

```
\circ min(2, 7, 3, 8) = 2.
Therefore, the minimum number of operations needed to transform this 3×3 grid into the letter 'Y' is 2. We can achieve this by
changing the middle element (1,1) from 2 to 1, and any of the non-'Y' elements from 0 to 1.
```

Minimizing Operations: We take the smallest number of operations from the calculations. In this case, it is:

Args: grid (List[List[int]]): A square grid of integers. Returns: int: The minimum number of operations needed.

Count the occurrences if is_vert_or_first_diag or is_second_diag or is_horiz_middle: vert_horiz_counter[x] += 1

Solution Implementation

from collections import Counter

n = len(grid)

def minimumOperationsToWriteY(self, grid):

Get the size of the grid

vert_horiz_counter = Counter()

for i, row in enumerate(grid):

for j, x in enumerate(row):

non_Y_counter = Counter()

Initialize counters for each part of the "Y"

// Return the minimum number of operations found

int minimumOperationsToWriteY(vector<vector<int>>& grid) {

// Iterate through the grid and categorize elements

for (int j = 0; j < gridSize; ++j) {</pre>

for (int i = 0; i < gridSize; ++i) {</pre>

} else {

int gridSize = grid.size(); // total size of the grid (n x n)

return minOperations;

C++

public:

};

/**

TypeScript

class Solution {

Python

class Solution:

```
else:
                    non_Y_counter[x] += 1
       # Compute the minimum operations by finding max occurrence in 'Y' and outside 'Y', excluding the same number in both plac
       min_operations = min(
            n * n - vert_horiz_counter[i] - non_Y_counter[j]
            for i in range(3) for j in range(3) if i != j
        return min_operations
Java
class Solution {
   // Method to calculate the minimum number of operations needed to write 'Y' on the grid
    public int minimumOperationsToWriteY(int[][] grid) {
       int gridSize = grid.length; // Dimension of the grid (since it's n x n)
       // Arrays to store the count of each number (0, 1, 2) in the regions that form 'Y'
        int[] countRegionY = new int[3];
        int[] countOutsideRegionY = new int[3];
       // Loop through each cell in the grid to separate the cells that will form 'Y'
       // and those that will not
        for (int i = 0; i < gridSize; ++i) {</pre>
            for (int j = 0; j < gridSize; ++j) {</pre>
                // Conditions to detect cells that are part of 'Y'
                boolean isDiagonalFromTopLeft = i == j && i <= gridSize / 2;</pre>
                boolean isDiagonalFromTopRight = i + j == gridSize - 1 && i <= gridSize / 2;</pre>
                boolean isVerticalMiddleLine = j == gridSize / 2 && i >= gridSize / 2;
                if (isDiagonalFromTopLeft || isDiagonalFromTopRight || isVerticalMiddleLine) {
                    // Increment the count for the region that forms 'Y'
                    ++countRegionY[grid[i][j]];
                } else {
                    // Increment the count for the region outside 'Y'
                    ++countOutsideRegionY[grid[i][j]];
       // We'll assume the worst case where we need to change every cell initially
       int minOperations = gridSize * gridSize;
       // Evaluate all combinations where the number for region 'Y' and outside are different
        for (int numberForY = 0; numberForY < 3; ++numberForY) {</pre>
            for (int numberOutsideY = 0; numberOutsideY < 3; ++numberOutsideY) {</pre>
                if (numberForY != numberOutsideY) {
                    // Calculate the number of operations needed if these two numbers were used
                    int operations = gridSize * gridSize - countRegionY[numberForY] - countOutsideRegionY[numberOutsideY];
                    // Check if the current operation count is lower than the minimum found so far
                    minOperations = Math.min(minOperations, operations);
```

```
int minimumOperations = gridSize * gridSize; // maximum number of operations (each cell made to write an element not alre
// Find the minimum number of operations by checking all combinations of colours between the groups
for (int i = 0; i < 3; ++i) { // iterate over possible colours for group one
    for (int j = 0; j < 3; ++j) { // iterate over possible colours for group two
        // Ensure we are not counting the same colour for both groups
        if (i != j) {
           // Calculate operations needed to paint the current combination
           // And update minimumOperations if this count is lower
           minimumOperations = min(minimumOperations, gridSize * gridSize - countGroupOne[i] - countGroupTwo[j]);
```

return minimumOperations; // Return the minimum number of operations found

* Calculates the minimum number of operations to write the integer 'Y'.

* @return {number} The minimum number of operations to write 'Y'.

* @param {number[][]} grid - A square grid of numbers.

for (let i = 0; i < 3; ++i) {

if (i !== j) {

return minimumOperations;

Get the size of the grid

non_Y_counter = Counter()

vert_horiz_counter = Counter()

for i, row in enumerate(grid):

for j, x in enumerate(row):

from collections import Counter

n = len(grid)

for (let j = 0; j < 3; ++j) {

minimumOperations = Math.min(

minimumOperations,

// Return the calculated minimum operations.

Initialize counters for each part of the "Y"

int countGroupOne[3] = $\{0\}$; // frequency count of numbers in the first group (Y shape)

// Increase the correct frequency counter based on the booleans above

if (isDiagonalOne || isDiagonalTwo || isVerticalLine) {

++countGroupTwo[grid[i][j]]; // for non-Y shape

++countGroupOne[grid[i][j]]; // for Y shape

int countGroupTwo[3] = $\{0\}$; // frequency count of numbers in the second group (not Y shape)

bool isDiagonalOne = (i == j) && (i <= gridSize / 2); // condition for main diagonal and upper half

bool isDiagonalTwo = (i + j == gridSize - 1) && (i <= gridSize / 2); // condition for anti-diagonal and upper hall

bool isVerticalLine = (j == gridSize / 2) && (i >= gridSize / 2); // condition for the vertical middle line and i

```
function minimumOperationsToWriteY(grid: number[][]): number {
   // Get the size of the grid.
   const gridSize = grid.length;
   // Initialize counters for the number collections.
   const crossCounters: number[] = Array(3).fill(0);
   const otherCounters: number[] = Array(3).fill(0);
    // Loop through the grid to count occurrences.
    for (let i = 0; i < gridSize; ++i) {</pre>
        for (let j = 0; j < gridSize; ++j) {</pre>
           // Check if the current cell is part of the cross (Y shape).
            const onFirstDiagonal = i === j && i <= gridSize >> 1;
            const onSecondDiagonal = i + j === gridSize - 1 && i <= gridSize >> 1;
            const onMiddleColumn = j === gridSize >> 1 && i >= gridSize >> 1;
            // Update the corresponding counters based on the cell's position.
            if (onFirstDiagonal || onSecondDiagonal || onMiddleColumn) {
                ++crossCounters[grid[i][j]];
            } else {
                ++otherCounters[grid[i][j]];
   // Initialize the answer with the maximum possible number of operations.
    let minimumOperations = gridSize * gridSize;
```

```
class Solution:
   def minimumOperationsToWriteY(self, grid):
       Calculate the minimum number of operations required to write the letter Y on the grid.
       Args:
       grid (List[List[int]]): A square grid of integers.
       Returns:
        int: The minimum number of operations needed.
```

Iterate over the grid to count the occurrences in the "Y" shape and elsewhere

Calculate which part of the grid is the 'Y' shape

is_vert_or_first_diag = i == j and i <= n // 2</pre>

 $is_second_diag = i + j == n - 1 and i <= n // 2$

is_horiz_middle = j == n // 2 and i >= n // 2

// Determine the minimum operations required by trying all combinations of numbers.

// We want to use different numbers for the cross and the other cells.

gridSize * gridSize - crossCounters[i] - otherCounters[j]

// Calculate the minimum operations by subtracting the already correct cells.

```
# Count the occurrences
               if is_vert_or_first_diag or is_second_diag or is_horiz_middle:
                   vert_horiz_counter[x] += 1
               else:
                   non_Y_counter[x] += 1
       # Compute the minimum operations by finding max occurrence in 'Y' and outside 'Y', excluding the same number in both places
       min_operations = min(
           n * n - vert_horiz_counter[i] - non_Y_counter[j]
           for i in range(3) for j in range(3) if i != j
       return min_operations
Time and Space Complexity
```

The time complexity of the given code is $0(n^2)$. This is because the nested for-loops iterate over the entire grid, which is of size n x n. The operations inside the for-loops are constant time operations, causing the overall time to be proportional to the number of elements in the grid.

Time Complexity

Space Complexity

The space complexity of the code is higher than 0(1) mentioned in the reference answer. It actually depends on the range of the elements in the grid. The two counters cnt1 and cnt2 are used to count occurrences of elements that satisfy certain conditions. Because Python's Counter is essentially a hash map, the space used by these counters will increase with the variety of elements in the grid. Assuming the range of numbers in the grid is k, the space complexity would be 0(k). If we were to consider k as being a constant because the problem might define a limit on the values of grid elements, then we could consider the space complexity to be 0(1). Without such a constraint being specified, the space complexity is not strictly constant.