883. Projection Area of 3D Shapes

Easy Geometry Array Math Matrix Leetcode Link

Problem Description

In this problem, we are given an `n x n` grid where each cell in the grid contains a non-negative integer value representing the height of a tower of cubes placed in that cell. The task is to calculate the total surface area of all three orthogonal projections (top-down, front, and side) of the arrangement of the cubes.

To understand this better, imagine looking at the cubes from three different viewpoints: from directly above (the xy-plane), from the front (the yz-plane), and from the side (the zx-plane). The area of the projection is simply the shadow of these cubes on the respective planes. The top-down view counts all the cubes that are visible from above, the front view counts the tallest tower in each row, and the side view counts the tallest tower in each column.

Intuition

For the solution, we can tackle each projection separately and sum up their areas:

- **xy-plane (Top-Down View)**: To compute the area of the projection on the xy-plane, we count the number of cubes that are visible from the top. Each cube contributes to the area if its height is greater than zero. We simply iterate over the grid and count the number of cells with a value greater than zero.
- **yz-plane (Front View)**: For the yz-plane, we are looking at the grid from the front, so we need to consider the tallest tower in each row since only the tallest tower is visible from this viewpoint. We iterate through each row and take the maximum value of the row, representing the height of the tallest tower.
- **zx-plane (Side View)**: Similarly, for the zx-plane projection, which represents the view from the side, we need to find the tallest tower in each column. We can achieve this by zipping the columns together and finding the maximum height of each column.

The final area of all projections is the sum of the three calculated areas: the top-down, the front, and the side projection areas.

Solution Approach

The implementation of the solution uses simple yet effective techniques to calculate the projection areas on all three planes. The choice of data structures is straightforward - since the input is a grid (a list of lists in Python), we use this same structure to operate on the data. Here's how the solution is structured:

- **xy-plane (Top-Down View)**: The `xy` variable is calculated by using a nested list comprehension. The outer loop iterates over each row, and the inner loop iterates over each value in the row, checking if `v > 0`. This check effectively counts the number of cells with cubes visible from the top view, contributing to the `xy` area. The use of list comprehension makes this part compact and efficient.

  ```python
  xy = sum(v > 0 for row in grid for v in row)
  ```

- **yz-plane (Front View)**: The `yz` projection is calculated by iterating over each row in the grid and selecting the maximum value found in that row using the `max` function. This represents the highest tower when looking from the front, and the sum of these maximum values across all rows gives the `yz` area.

  ```python
  yz = sum(max(row) for row in grid)
  ```

- **zx-plane (Side View)**: To find the `zx` projection, we need the tallest tower in each column. Python's `zip` function is used here, which effectively transposes the grid, grouping all elements at similar column indices together. By iterating over these transposed columns and selecting the maximum value found in each, we compute the `zx` area.

  ```python
  zx = sum(max(col) for col in zip(*grid))
  ```

After computing the areas individually, we sum them to get the total area of all three projections, which is returned as the final result. The algorithm's complexity is linear in the number of cells in the grid, making it O(n^2), where `n` is the dimension of the grid. The solution is not only intuitive but also showcases the power and readability of Python's list comprehensions and built-in functions.

No complex patterns or algorithms are necessary for this problem, as it's more about understanding what each projection looks like and selecting the appropriate elements from the grid to count. The solution, therefore, combines basic iteration and aggregation methods to achieve an efficient and understandable result.

```python
class Solution:
    def projectionArea(self, grid: List[List[int]]) -> int:
        xy = sum(v > 0 for row in grid for v in row)
        yz = sum(max(row) for row in grid)
        zx = sum(max(col) for col in zip(*grid))
        return xy + yz + zx
```

Example Walkthrough

Let's go through an example to illustrate the solution approach using a `3 x 3` grid as an example:

Suppose we have the following grid representing the heights of the towers of cubes:

```python
grid = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
```

We will walk through the solution by calculating the area of each orthogonal projection.

Top-Down View (xy-plane):

First, we look at how many cubes contribute to the top view. Since all cubes are visible from the top, we count all cells with a height greater than zero.

```
1 2 3
4 5 6
7 8 9
```

From this, we can see that all 9 cells contribute to the top view (as all heights are non-zero). Thus, the `xy` area is `9`.

Front View (yz-plane):

Now, we look at each row to find the tallest tower, as this would be visible from the front.

```
1 2 3  <- Tallest in 1st row is 3
4 5 6  <- Tallest in 2nd row is 6
7 8 9  <- Tallest in 3rd row is 9
```

The heights of the tallest towers are `3`, `6`, and `9`, and the `yz` area is the sum of these which is `3 + 6 + 9 = 18`.

Side View (zx-plane):

For the side view, we need the tallest tower in each column. By looking at each column:

```
1
4
7 <- Tallest in 1st column is 7

2
5
8 <- Tallest in 2nd column is 8

3
6
9 <- Tallest in 3rd column is 9
```

The tallest towers from each column have heights `7`, `8`, and `9`. Thus, the `zx` area is the sum of these which is `7 + 8 + 9 = 24`.

Finally, add up all the areas to get the total surface area of all projections:

```
xy + yz + zx = 9 + 18 + 24 = 51
```

So according to our solution, the final result for the given `3 x 3` grid is `51`.

Python Solution

```python
class Solution:
    def projectionArea(self, grid: List[List[int]]) -> int:
        # Calculate the XY projection area by counting all non-zero values in the grid
        xy_projection = sum(value > 0 for row in grid for value in row)

        # Calculate the YZ projection area by summing up the tallest building in each row
        yz_projection = sum(max(row) for row in grid)

        # Calculate the ZX projection area by summing up the tallest building in each column.
        # zip(*grid) creates an iterator that aggregates elements from each row into columns.
        zx_projection = sum(max(column) for column in zip(*grid))

        # Return the sum of the three projection areas.
        return xy_projection + yz_projection + zx_projection
```

Java Solution

```java
class Solution {

    // Method to calculate projection area of a 3D shape
    // represented by a grid onto the xy, yz, and zx planes.
    public int projectionArea(int[][] grid) {
        int areaXY = 0; // Area projection onto the XY plane
        int areaYZ = 0; // Area projection onto the YZ plane
        int areaZX = 0; // Area projection onto the ZX plane

        // We assume grid is a square (n x n matrix), so we use grid.length to get 'n'.
        for (int i = 0, n = grid.length; i < n; ++i) {
            int maxYForYZ = 0; // Max height in y-direction for YZ projection
            int maxXForZX = 0; // Max height in x-direction for ZX projection

            // Iterate over each row and column to calculate heights
            for (int j = 0; j < n; ++j) {

                // Increment area counter for XY plane if height > 0
                if (grid[i][j] > 0) {
                    areaXY++;
                }

                // Determine the tallest point in the current row for YZ projection
                maxYForYZ = Math.max(maxYForYZ, grid[i][j]);

                // Determine the tallest point in the current column for ZX projection
                maxXForZX = Math.max(maxXForZX, grid[j][i]);
            }

            // Accumulate the maximum height of the grid for YZ and ZX projections
            areaYZ += maxYForYZ;
            areaZX += maxXForZX;
        }

        // The total projection area is the sum of all three individual areas
        return areaXY + areaYZ + areaZX;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <algorithm>
using namespace std;

class Solution {
public:
    int projectionArea(vector<vector<int>>& grid) {
        int areaXY = 0; // Initialize the area of the projection onto the XY plane
        int areaYZ = 0; // Initialize the area of the projection onto the YZ plane
        int areaZX = 0; // Initialize the area of the projection onto the ZX plane

        // Iterate over the grid to calculate the areas of the projections
        for (int i = 0, n = grid.size(); i < n; ++i) {
            int maxRowValue = 0; // The maximum value in the current row for YZ projection
            int maxColValue = 0; // The maximum value in the current column for ZX projection

            for (int j = 0; j < n; ++j) {
                // Count the number of cells with a value greater than 0 for the XY projection
                areaXY += grid[i][j] > 0 ? 1 : 0;

                // Find the maximum value in the i-th row for YZ projection
                maxRowValue = max(maxRowValue, grid[i][j]);

                // Find the maximum value in the j-th column for ZX projection
                maxColValue = max(maxColValue, grid[j][i]);
            }

            // Add the maximum row value to the YZ projection's area
            areaYZ += maxRowValue;

            // Add the maximum column value to the ZX projection's area
            areaZX += maxColValue;
        }

        // Return the total projection area by summing up the individual projections
        return areaXY + areaYZ + areaZX;
    }
};
```

Typescript Solution

```typescript
function projectionArea(grid: number[][]): number {
    // Get the size of the grid.
    const size = grid.length;
    // Calculate the top-view projection area, counting the cells that are non-zero.
    let totalArea = grid.reduce((total, row) => total + row.reduce((count, cell) => count + (cell > 0 ? 1 : 0), 0), 0);

    // Iterate over each row and each column to find the maximum heights
    // and calculate the side-view projection areas.
    for (let i = 0; i < size; i++) {
        let maxRowHeight = 0; // Max height in the current row.
        let maxColHeight = 0; // Max height in the current column.
        for (let j = 0; j < size; j++) {
            // Find the maximum height of the current row by comparing heights of the cells.
            maxRowHeight = Math.max(maxRowHeight, grid[i][j]);
            // Find the maximum height of the current column by comparing heights of the cells.
            maxColHeight = Math.max(maxColHeight, grid[j][i]);
        }
        // Add the maximum heights to the total projection area.
        totalArea += maxRowHeight + maxColHeight;
    }

    // Return the total projection area.
    return totalArea;
}
```

Time and Space Complexity

The time complexity of the given code is `O(N^2)` where `N` is the number of rows (or columns, assuming the grid is square). This is due to the three separate loops that iterate over all N*N elements of the grid or its transposition. The first loop iterates over all elements for the `xy` shadow, the second loop iterates over each row to find the maximum for `yz` shadow, and the third loop iterates over each transposed column to find the maximum for the `zx` shadow.

The space complexity of the code is `O(N)` because of the space required for the transposition of the grid (`zip(*grid)`). This creates a list of tuples where each tuple is a column from the original grid. The space is linear with the number of columns N.