892. Surface Area of 3D Shapes

Easy Geometry Array Math Matrix

Leetcode Link

Problem Description

In this problem, we have a square `n x n` grid representing an overhead view of a collection of 3D cubes where each grid cell `(i, j)` contains `v` cubes, indicated by the value `v = grid[i][j]`. Each cube is of size `1 x 1 x 1`. After placing the cubes, we glue any cubes that are directly next to each other on the grid vertically or horizontally. Our task is to determine the total surface area of the joined 3D shapes created.

Each cube on its own has a surface area of 6 units, but when cubes are adjacent, they share walls and reduce the overall surface area. We must account for shared walls between adjacent cubes and also remember that there is an additional bottom surface for the shape formed by cubes at each grid cell.

Intuition

The intuition behind the solution is to do a two-fold computation:

- First, for each tower of cubes, calculate the surface area if it were isolated, i.e., not adjacent to any other cubes. This involves the surface area of the sides and top of the tower.
- Second, subtract the area of the surfaces shared between adjacent towers of cubes.

To break it down:

1. We iterate over each cell in the grid:
    - For a cell with `v` cubes (where `v > 0`), it has 4 sides, and each will have an area of `v`, plus the top and bottom, each having an area of `1`. Hence the total surface area of an isolated tower is `2 * v + 4`.
    - We then check if there is a neighboring tower to the left (west) and above (north):
        - If there is a neighbor to the left, we find the minimum of `v` and the height of this neighboring tower. The shared wall's area is this minimum times two (since each shared face between two cubes is equal to `1 * 1 = 1`, and we're subtracting from both towers). So we subtract this value from the total surface area.
        - If there is a neighbor above, perform the same calculation for the shared wall and subtract accordingly.
2. Sum these adjusted surface area values to get the total surface area of the 3D shape.

This approach gives us the correct total surface area of the resulting irregular 3D shape after accounting for all internal and external surfaces.

Solution Approach

The provided solution to finding the total surface area of the resulting 3D shapes formed by gluing `1 × 1 × 1` cubes together in a grid follows an efficient step-by-step approach. Here is a detailed explanation:

1. Initialize a variable `ans` to hold the total surface area count, starting at 0.
2. Loop through each cell in the grid with a nested for-loop, using `i` to index rows and `j` to index columns:
    - For each cell at `(i, j)` with a value `v`:
        - If `v` is greater than 0 (indicating there are cubes),
            - Add the surface area of an isolated tower of `v` cubes to `ans`. This is done by adding `2` (for the top and bottom of the tower) plus `4 * v` for the four vertical sides of the tower.
3. Next comes the subtraction of shared walls to account for gluing:
    - If the current tower has a neighbor to the left (at `(i, j - 1)`), the shared wall's area to subtract is twice the minimum height of these two towers: `min(v, grid[i][j - 1]) * 2`.
    - If the current tower has a neighbor above (at `(i - 1, j)`), similarly, the shared wall's area is also twice the minimum height of these towers: `min(v, grid[i - 1][j]) * 2`.
    - Subtract these shared wall areas from the `ans`.
4. We ensure the shared walls are only subtracted when there is actually a neighbor. We check that `i` or `j` is not at the first row or column by conditionally executing the subtraction only if `i` or `j` is greater than 0.
5. Continue this process for all cells in the grid.
6. After the loops complete, `ans` contains the correct total surface area. This value is then returned.

In terms of data structures, the solution uses the given `grid` directly and an accumulator for the surface area count. There is no need for additional data structures.

The algorithmic pattern demonstrated in this solution is straightforward iteration through grid elements, with conditionals within the nested loops to handle edge cases (literally the edges of the grid). The mathematics behind the area calculations is simple arithmetic involving addition, multiplication, and minimum values.

The result is a solution with time complexity O(n^2), where `n` is the dimension of the grid, which is optimal since we must visit each cell at least once, and constant space complexity, as no additional space proportional to the input size is used.

Example Walkthrough

Let's go through a small example to illustrate the solution approach with an `n x n` grid where `n = 2`. Consider the following grid configuration:

```
grid = [
  [1, 2],
  [3, 4]
]
```

Step 1:

- Initialize `ans` to 0.

Step 2:

- Loop over each cell `(i, j)` in `grid`.

Step 3:

- At `(0, 0)`, value `v` is `1`. There are no neighbors above or to the left. Surface area added to `ans` is `2 (top and bottom) + 4 * v = 2 + 4 * 1 = 6`.

- At `(0, 1)`, value `v` is 2. No neighbors above but there is one to the left. Surface area with sides is `2 + 4 * v = 2 + 4 * 2 = 10`. It shares a wall with `(0, 0)`, so we subtract the shared surface: `min(2, 1) * 2 = 2`. The surface area contributed by this cell is `10 - 2 = 8`.

Step 4:

- At `(1, 0)`, `v` is 3. There's a neighbor above but not to the left. Surface area with sides is `2 + 4 * v = 2 + 4 * 3 = 14`. Shared wall with `(0, 0)` must be subtracted: `min(3, 1) * 2 = 2`. The contribution is `14 - 2 = 12`.

- At `(1, 1)`, v is 4. There are neighbors both above and to the left. Surface area with sides is `2 + 4 * v = 2 + 4 * 4 = 18`. It shares walls with `(0, 1)` and `[1, 0]`. Subtractions are `min(4, 2) * 2 = 4` and `min(4, 3) * 2 = 6`. Contribution is `18 - 4 - 6 = 8`.

Step 5:

- Sum the adjusted surface areas for each cell: `6 (from (0, 0)) + 8 (from (0, 1)) + 12 (from (1, 0)) + 8 (from (1, 1)) = 34`.

After the final calculation, `ans` is `34`, so the total surface area for the shape formed by the cubes in this `2 x 2` grid is `34` units.

Python Solution

```python
class Solution:
    def surfaceArea(self, grid: List[List[int]]) -> int:
        # Initialize the surface area to 0
        surface_area = 0

        # Loop through each cell of the grid
        for i, row in enumerate(grid):
            for j, height in enumerate(row):
                # If the height of the current voxel is not zero
                if height:
                    # Add the surface area of the current voxel
                    # Each voxel contributes to 2 base/top faces and 4 side faces
                    surface_area += 2 + height * 4

                # If we are not on the first row, subtract the overlapping area
                # with the voxel directly behind (in the i-1 row)
                if i:
                    surface_area -= min(height, grid[i - 1][j]) * 2

                # If we are not in the first column, subtract the overlapping area
                # with the voxel directly on the left (in the j-1 column)
                if j:
                    surface_area -= min(height, grid[i][j - 1]) * 2

        # Return the total calculated surface area
        return surface_area
```

Please note that the `List` type hint assumes that you have imported `List` from `typing` module at the beginning of your script like this:

```python
from typing import List
```

Java Solution

```java
class Solution {

    // Function to calculate the total surface area of 3D shapes
    public int surfaceArea(int[][] grid) {
        // Length of the grid (number of rows/columns in the grid)
        int n = grid.length;
        // Initialize surface area to 0
        int totalSurfaceArea = 0;

        // Iterate over each cell in the grid
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                // If the cell has at least one cube
                if (grid[i][j] > 0) {
                    // Add surface area (2) + 4 sides for each cube in the cell
                    totalSurfaceArea += 2 + grid[i][j] * 4;

                    // Subtract the area of the sides that are shared with the adjacent cube on the left
                    if (i > 0) {
                        totalSurfaceArea -= Math.min(grid[i][j], grid[i - 1][j]) * 2;
                    }

                    // Subtract the area of the sides that are shared with the adjacent cube behind
                    if (j > 0) {
                        totalSurfaceArea -= Math.min(grid[i][j], grid[i][j - 1]) * 2;
                    }
                }
            }
        }

        // Return the total surface area
        return totalSurfaceArea;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    // Calculate the total surface area of 3D shapes represented by a grid
    int surfaceArea(std::vector<std::vector<int>>& grid) {
        // Get the size of the grid (which is n x n)
        int gridSize = grid.size();
        // Initialize the answer to 0
        int totalSurfaceArea = 0;

        // Iterate through the grid
        for (int i = 0; i < gridSize; ++i) {
            for (int j = 0; j < gridSize; ++j) {
                // If the current cell is non-zero (a block exists)
                if (grid[i][j]) {
                    // Each block has a top and bottom face
                    totalSurfaceArea += 2;
                    // Each block also contributes 4 side faces
                    totalSurfaceArea += grid[i][j] * 4;
                    // If there is a block to the north, subtract the hidden area
                    if (i > 0) {
                        totalSurfaceArea -= std::min(grid[i][j], grid[i - 1][j]) * 2;
                    }
                    // If there is a block to the west, subtract the hidden area
                    if (j > 0) {
                        totalSurfaceArea -= std::min(grid[i][j], grid[i][j - 1]) * 2;
                    }
                }
            }
        }

        // Return the totalSurfaceArea calculated
        return totalSurfaceArea;
    }
};
```

Typescript Solution

```typescript
function surfaceArea(grid: number[][]): number {
    // Get the size of the grid (which is n x n)
    const gridSize: number = grid.length;
    // Initialize the total surface area to 0
    let totalSurfaceArea: number = 0;

    // Iterate through the grid
    for (let i = 0; i < gridSize; ++i) {
        for (let j = 0; j < gridSize; ++j) {
            // If the current cell is non-zero (a block exists)
            if (grid[i][j]) {
                // Each block has a top and bottom face
                totalSurfaceArea += 2;
                // Each block also contributes 4 side faces
                totalSurfaceArea += grid[i][j] * 4;
                // If there is a block to the north, subtract the hidden area
                if (i > 0) {
                    totalSurfaceArea -= Math.min(grid[i][j], grid[i - 1][j]) * 2;
                }
                // If there is a block to the west, subtract the hidden area
                if (j > 0) {
                    totalSurfaceArea -= Math.min(grid[i][j], grid[i][j - 1]) * 2;
                }
            }
        }
    }

    // Return the calculated total surface area
    return totalSurfaceArea;
}
```

Time and Space Complexity

Time Complexity

The given code iterates over each cell in the `grid`, which has a size `n * n` where `n` is the length of a side of the `grid`. We can assume that the grid is square-shaped since no other shape is indicated. For each cell, a constant amount of work is done: checking the cell value, adding the surface area of the top and bottom, and subtracting the areas that are shared with adjacent blocks (if any).

This leads to a time complexity of $O(n^2)$, as it's necessary to visit each cell exactly once, and the amount of work per cell does not depend on the size of the grid.

Space Complexity

The space complexity of the code is $O(1)$ because no additional space proportional to the size of the input grid is being used. The solution uses a fixed amount of space, as all calculations are done in place, with a single integer `ans` holding the accumulated surface area, aside from the input data structure itself.