1183. Maximum Number of Ones

Hard · Greedy · Heap (Priority Queue) · Leetcode Link

Problem Description

The problem presents us with a task to figure out the maximum number of `1`s that can be placed in a matrix of dimensions `width * height` such that any square sub-matrix with size `sideLength * sideLength` contains at most `maxOnes` ones. We need to consider that the value of each cell in the matrix can either be `0` or `1`. The main goal is to place the `1`s in such a way that we can maximize their count without violating the condition placed on the square sub-matrices.

Intuition

To solve this problem, we need to recognize that the constraints on sub-matrices of size `sideLength * sideLength` create a repeating pattern across the larger matrix. The idea is that we only need to figure out the placement of `1`s within a section of the matrix that is the size of `sideLength * sideLength`, and then repeat this pattern throughout the entire matrix.

The intuition behind the solution is to:

1. Create a pattern within a single square block of `sideLength * sideLength` that maximizes the `1`s without exceeding `maxOnes`.
2. Make sure that when the pattern is repeated, we still adhere to the constraints (with regards to `maxOnes`) at the edges where two repeating patterns meet.
3. Calculate the frequency of each position within the square block throughout the entire matrix and fill the positions with the highest frequency with `1`s first.
4. After placing `1`s in the positions with the highest frequency, we continue doing so in descending order until we reach the limit set by `maxOnes`.
5. Sum up the most frequent positions that we filled with `1`s to determine the maximum number of `1`s in the whole matrix.

The solution code implements this approach effectively by first initializing a counter array `cnt` to keep track of how many times each position within a `sideLength` square would appear in the whole matrix. It then iterates over every cell in the matrix, calculating the position's index in the `cnt` array by using modulo arithmetic. After populating `cnt`, we sort it in reverse order to prioritize positions with the highest frequency. Finally, we sum up the `maxOnes` highest values in `cnt` to find the maximum number of `1`s the matrix can have.

Solution Approach

The implementation of the solution involves several steps that use basic programming constructs and straightforward arithmetic operations. Here's how the solution is carried out:

1. **Initialize a counter array**: A list `cnt` of size `x * x` (where `x` is `sideLength`) is created to keep track of the number of times each position within a `sideLength` x `sideLength` block can be filled throughout the entire matrix (`M`). The size of this array corresponds to all possible positions in a block of the size `sideLength * sideLength`.

2. **Fill the counter array**: We iterate over each cell in the matrix using nested loops. Each cell's position is determined using the module operation to find its equivalent position within the block pattern. The index in `cnt` for any given cell at position `(i, j)` is calculated using `(i % x) * x + (j % x)`, which effectively maps the 2D coordinates to a 1D array index.

3. **Sort the counter array**: Once the entire matrix has been scanned and the `cnt` array has been filled with the frequency of each position, we sort `cnt` in descending order. This ensures that the positions with the highest frequency (i.e., positions that will be included in the maximum number of `sideLength` x `sideLength` squares) are at the start of the `cnt` array.

4. **Sum up the top frequencies**: Finally, since we need to place a maximum of `maxOnes` `1`s in any `sideLength` x `sideLength` block, we just take the first `maxOnes` elements of the sorted `cnt` array. These elements indicate the highest possible frequency for the `1`s that can be placed in those positions without violating the constraints. The sum of these top frequencies will give us the maximum count of `1`s that can be placed in the whole matrix.

The solution employs the `greedy algorithm` concept by always prioritizing the positions that will appear most frequently across the matrix and filling those with `1`s before considering positions with lower frequency. This ensures the optimal distribution of the `1`s within the constraints given.

The data structure used is a simple list to keep track of the frequency counts. The sorting algorithm, which could be any efficient in-place sorting algorithm, is utilized to arrange the counts in descending order. Finally, the modulo operation and arithmetic are used to map positions and calculate frequencies.

The reference solution code provided above encapsulates this approach within the `maximumNumberOfOnes` method of the `Solution` class. This method is parameterized by the matrix dimensions (`width` and `height`), the block size (`sideLength`), and the constraint (`maxOnes`), and it returns the calculated maximum number of `1`s accordingly.

Example Walkthrough

Let's use a small example to illustrate the solution approach. Consider a matrix of dimensions `width = 3`, `height = 3`, `sideLength = 2`, and `maxOnes = 2`. This means we want to fill a 3×3 matrix with ones and zeros, but any 2×2 sub-matrix within it can contain at most 2 ones.

1. **Initialize counter array**: Since our `sideLength` is 2, we create a list `cnt` of size `2 * 2` or 4 to keep track of each position's frequency within such a block. The `cnt` list starts as `[0, 0, 0, 0]`.

2. **Fill the counter array**: We iterate over the 3×3 matrix's cells. For each cell at position `(i, j)`, we use the formula `(i % sideLength) * sideLength + (j % sideLength)` to increment the corresponding index in `cnt`. After this step, `cnt` looks like this:

   - The cell at `(0, 0)` maps to `cnt[0]`, increment it (cnt: `[1, 0, 0, 0]`).
   - The cell at `(0, 1)` maps to `cnt[1]`, increment it (cnt: `[1, 1, 0, 0]`).
   - The cell at `(0, 2)` also maps to `cnt[0]` (because `(0 % 2)*2 + (2 % 2)` equals `0`), increment it (cnt: `[2, 1, 0, 0]`).
   - Continue this for all cells and `cnt` ends up as `[2, 2, 1, 1]`.
3. **Sort the counter array**: We sort `cnt` in descending order, resulting in `[2, 2, 1, 1]`. This tells us the frequency of each position across the whole matrix.

4. **Sum up the top frequencies**: Since `maxOnes` is 2, we take the largest 2 values from `cnt`, which are both `2`. We sum these to get the maximum number of `1`s we can place, which is `2 + 2 = 4`.

By following these steps, the pattern within a single block would be to place ones in the first two positions because they have the highest frequency, and the resulting matrix might look something like:

```
1 1 1
1 0 0
0 0 0
```

This matrix has the maximum number of `1`s (which is 4 in this case) while adhering to the constraint of a maximum of 2 ones in any 2×2 sub-matrix.

Python Solution

```python
class Solution:
    def maximumNumberOfOnes(self, width: int, height: int, side_length: int, max_ones: int) -> int:
        # `side_length` is the length of one side of the square in the grid
        cell_count = [0] * (side_length * side_length)  # Initialize a list to count occurrence of '1's in the grid cells

        # Iterate over each cell in the grid
        for i in range(width):
            for j in range(height):
                # Calculate the position of the cell in the side_length x side_length square
                # This identifies the repeating pattern within the submatrix
                cell_index = (i % side_length) * side_length + (j % side_length)
                # Increment the count for this position
                cell_count[cell_index] += 1

        # Sort the cell counts in descending order to get the cells with the most '1's first
        cell_count.sort(reverse=True)

        # Sum the counts of the top `max_ones` cells to get the maximum number of '1's
        return sum(cell_count[:max_ones])

# Example Usage:
# sol = Solution()
# result = sol.maximumNumberOfOnes(3, 3, 2, 1)
# print(result)  # Should print the maximum number of '1's possible with the given constraints
```

Java Solution

```java
import java.util.Arrays; // Import necessary for the Arrays.sort() method.

class Solution {
    // Method calculates the maximum number of ones that can be placed
    // in a width by height grid, with a maximum of maxOnes ones per
    // sideLength by sideLength subgrid.
    public int maximumNumberOfOnes(int width, int height, int sideLength, int maxOnes) {
        // Initialize the counter array.
        // Each element represents the number of times a cell is visited in the tiling process.
        int[] count = new int[sideLength * sideLength];

        // Iterate over each cell in the grid.
        for (int i = 0; i < width; ++i) {
            for (int j = 0; j < height; ++j) {
                // Calculate the position in the subgrid (modular arithmetic).
                int index = (i % sideLength) * sideLength + (j % sideLength);
                // Increment the count for this position in subgrid.
                ++count[index];
            }
        }

        // Sort the count array in ascending order to find the positions
        // with the highest counts.
        Arrays.sort(count);

        // Initialize the answer variable, which will hold the maximum number of ones.
        int answer = 0;

        // Add the highest values from the sorted count array. The highest values correspond
        // to the positions most frequently visited, and thus should be prioritized to
        // place the ones.
        for (int i = 0; i < maxOnes; ++i) {
            answer += count[count.length - i - 1]; // Take values from the end of the sorted array.
        }

        // Return the calculated maximum number of ones.
        return answer;
    }
}
```

C++ Solution

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

class Solution {
public:
    // Calculates the maximum number of 1's that can be placed in a grid,
    // with constraints on the number of 1's in any subgrid of a certain side length
    int maximumNumberOfOnes(int width, int height, int sideLength, int maxOnes) {
        // Count frequency of 1s for each position in subgrid
        std::vector<int> frequency_count(sideLength * sideLength, 0);

        // Iterate through the entire grid using modular arithmetic
        // to map positions to subgrid positions
        for (int i = 0; i < width; ++i) {
            for (int j = 0; j < height; ++j) {
                // Calculate the position in the subgrid
                int position_in_subgrid = (i % sideLength) * sideLength + (j % sideLength);
                // Increment the frequency count for this subgrid position
                ++frequency_count[position_in_subgrid];
            }
        }

        // Sort the frequency count in descending order
        // to get the most frequent positions first
        std::sort(frequency_count.rbegin(), frequency_count.rend());

        // Calculate the maximum number of 1's by adding up the highest frequencies
        int max_ones_count = 0;
        for (int i = 0; i < maxOnes; ++i) {
            max_ones_count += frequency_count[i];
        }

        // Return the maximum number of 1's
        return max_ones_count;
    }
};
```

Typescript Solution

```typescript
/**
 * This function calculates the maximum number of ones that can be distributed
 * in a sub-matrix pattern within a larger matrix while ensuring that
 * each sub-matrix has no more than the specified maximum number of ones.
 *
 * @param {number} width - The width of the main matrix
 * @param {number} height - The height of the main matrix
 * @param {number} sideLength - The side length of the sub-matrix
 * @param {number} maxOnes - The maximum number of ones allowed in each sub-matrix
 * @returns {number} - The maximum number of ones that can be placed in the main matrix
 */
function maximumNumberOfOnes(width: number, height: number, sideLength: number, maxOnes: number): number {
    // Create an array to store the count of potential ones for each position in the sub-matrix
    const subMatrixCounts: number[] = new Array(sideLength * sideLength).fill(0);

    // Iterate over each position in the main matrix
    for (let i = 0; i < width; ++i) {
        for (let j = 0; j < height; ++j) {
            // Determine the corresponding position within the sub-matrix
            const positionIndex: number = (i % sideLength) * sideLength + (j % sideLength);
            // Increment the count for this position
            ++subMatrixCounts[positionIndex];
        }
    }

    // Sort the counts in descending order
    subMatrixCounts.sort((a, b) => b - a);

    // Sum up to the maxOnes most frequent positions to find the maximum number of ones
    const maxOnesSum: number = subMatrixCounts.slice(0, maxOnes).reduce((sum, count) => sum + count, 0);

    return maxOnesSum;
}

// Example usage
const maxWidth: number = 4;
const maxHeight: number = 4;
const subMatrixSideLength: number = 1;
const maxSubMatrixOnes: number = 2;
const result: number = maximumNumberOfOnes(maxWidth, maxHeight, subMatrixSideLength, maxSubMatrixOnes);

console.log(result); // Output the result
```

Time and Space Complexity

Time Complexity

The given code has mainly two parts that contribute to the total time complexity: the nested `for` loop and the sorting operation.

1. **Nested `for` Loop:** The nested loop runs once for every cell in the `width x height` grid. A single iteration of the inner loop results in a constant-time operation. Therefore, the total time taken by the nested loop would be `O(width * height)`.

2. **Sorting Operation:** The `sort()` method is called on the `cnt` list, which here can have at most `sideLength * sideLength` elements. The sorting operation using the default TimSort algorithm implemented in Python has a worst-case time complexity of `O(n log n)`, where `n` is the number of elements in the list. This step would have a complexity of `O(sideLength^2 log(sideLength^2))`.

The overall time complexity would be the sum of these two parts, which would be: `O(width * height + sideLength^2 log(sideLength^2))`.

Space Complexity

The space complexity of the given code is determined principally by the storage required for the `cnt` list.

- **cnt List:** The list `cnt` has `sideLength * sideLength` elements, so the space required is `O(sideLength^2)`.

Additionally, a fixed amount of extra space is used by counters and indices in the `for` loops, but this does not depend on the input size and thus contributes only a constant factor.

Hence, the total space complexity is: `O(sideLength^2)`.