1183. Maximum Number of Ones

Greedy Heap (Priority Queue)

Problem Description

Hard

The problem presents us with a task to figure out the maximum number of 1s that can be placed in a matrix of dimensions width * height such that any square sub-matrix with size sideLength * sideLength contains at most max0nes 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 1s 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 1s within a section of the

2. Make sure that when the pattern is repeated, we still adhere to the constraints (with regards to max0nes) at the edges where two repeating

The intuition behind the solution is to:

patterns meet.

5. Sum up the most frequent positions that we filled with 1s to determine the maximum number of 1s in the whole matrix.

matrix that is the size of sideLength * sideLength, and then repeat this pattern throughout the entire matrix.

1. Create a pattern within a single square block of sideLength * sideLength that maximizes the 1s without exceeding max0nes.

- 3. Calculate the frequency of each position within the square block throughout the entire matrix and fill the positions with the highest frequency with 1s first. 4. After placing 1s in the positions with the highest frequency, we continue doing so in descending order until we reach the limit set by max0nes.
- 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

of 1s the matrix can have. **Solution Approach**

The implementation of the solution involves several steps that use basic programming constructs and straightforward arithmetic

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

prioritize positions with the highest frequency. Finally, we sum up the maxOnes highest values in cnt to find the maximum number

position within a sideLength x sideLength block can be filled throughout the entire matrix (M). The size of this array

operations. Here's how the solution is carried out:

corresponds to all possible positions in a block of the size sideLength * sideLength. Fill the counter array: We iterate over each cell in the matrix using nested loops. Each cell's position is determined using the

is calculated using (i % x) * x + (j % x), which effectively maps the 2D coordinates to a 1D array index. 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.

module operation to find its equivalent position within the block pattern. The index in cnt for any given cell at position (i, j)

we just take the first max0nes elements of the sorted cnt array. These elements indicate the highest possible frequency for the 1s that can be placed in those positions without violating the constraints. The sum of these top frequencies will give us the maximum count of 1s that can be placed in the whole matrix.

The solution employs the [greedy](/problems/greedy_intro) algorithm concept by always prioritizing the positions that will

appear most frequently across the matrix and filling those with 1s before considering positions with lower frequency. This

Sum up the top frequencies: Finally, since we need to place a maximum of maxOnes 1s in any sideLength x sideLength block,

ensures the optimal distribution of the 1s 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

(max0nes), and it returns the calculated maximum number of 1s accordingly.

frequency within such a block. The cnt list starts as [0, 0, 0, 0].

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 maximum number of 1s we can place, which is 2 + 2 = 4.

highest frequency, and the resulting matrix might look something like:

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.

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

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:

any 2×2 sub-matrix.

Python

Solution Implementation

Iterate over each cell in the grid

cell_count[cell_index] += 1

for j in range(height):

cell_count.sort(reverse=True)

return sum(cell_count[:max_ones])

for i in range(width):

 Continue this for all cells and cnt ends up as [2, 2, 1, 1]. 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.

Sum up the top frequencies: Since max0nes is 2, we take the largest 2 values from cnt, which are both 2. We sum these to get

By following these steps, the pattern within a single block would be to place ones in the first two positions because they have the

∘ 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]).

1 1 1 1 0 0 0 0 0

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

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

 $cell_count = [0] * (side_length * side_length) # Initialize a list to count occurrence of '1's in the grid cells$

Calculate the position of the cell in the side_length x side_length square

This identifies the repeating pattern within the submatrix

Increment the count for this position

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

// Calculates the maximum number of 1's that can be placed in a grid,

std::vector<int> frequency_count(sideLength * sideLength, 0);

// Iterate through the entire grid using modular arithmetic

// Calculate the position in the subgrid

++frequency_count[position_in_subgrid];

std::sort(frequency_count.rbegin(), frequency_count.rend());

// Sort the frequency count in descending order

// to get the most frequent positions first

max_ones_count += frequency_count[i];

for (int i = 0; i < max0nes; ++i) {</pre>

// Count frequency of 1s for each position in subgrid

// to map positions to subgrid positions

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

for (int i = 0; i < width; ++i) {

int max_ones_count = 0;

// with constraints on the number of 1's in any subgrid of a certain side length

// Increment the frequency count for this subgrid position

// Calculate the maximum number of 1's by adding up the highest frequencies

int position_in_subgrid = (i % sideLength) * sideLength + (j % sideLength);

int maximumNumberOfOnes(int width, int height, int sideLength, int maxOnes) {

cell_index = (i % side_length) * side_length + (j % side_length)

Sum the counts of the top `max_ones` cells to get the maximum number of '1's

Sort the cell counts in descending order to get the cells with the most '1's first

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

return answer;

C++

public:

#include <vector>

class Solution {

#include <algorithm>

Java

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) {</pre>
            // 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 < max0nes; ++i) {</pre>
        answer += count[count.length - i - 1]; // Take values from the end of the sorted array.
   // Return the calculated maximum number of ones.
```

```
// Return the maximum number of 1's
        return max_ones_count;
};
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) {</pre>
        for (let j = 0; j < height; ++j) {</pre>
           // 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
```

cell_count = [0] * (side_length * side_length) # Initialize a list to count occurrence of '1's in the grid cells

```
Time and Space Complexity
```

Iterate over each cell in the grid

for j in range(height):

cell_count.sort(reverse=True)

return sum(cell count[:max ones])

result = sol.maximumNumberOfOnes(3, 3, 2, 1)

for i in range(width):

class Solution:

Example Usage:

sol = Solution()

Time Complexity

log(sideLength^2)).

Space Complexity

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 0(width * height).

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 $0(n \log n)$, where n is the number of elements in the list. This step would have a complexity of $0(sideLength^2)$

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

- log(sideLength^2)). The overall time complexity would be the sum of these two parts, which would be: 0(width * height + sideLength^2
- 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 0(sideLength^2).

def maximumNumberOfOnes(self, width: int, height: int, side_length: int, max_ones: int) -> int:

Calculate the position of the cell in the side_length x side_length square

`side_length` is the length of one side of the square in the grid

Increment the count for this position

cell_count[cell_index] += 1

This identifies the repeating pattern within the submatrix

cell_index = (i % side_length) * side_length + (j % side_length)

Sum the counts of the top `max_ones` cells to get the maximum number of '1's

print(result) # Should print the maximum number of '1's possible with the given constraints

Sort the cell counts in descending order to get the cells with the most '1's first

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: 0(sideLength^2).