1292. Maximum Side Length of a Square with Sum Less than or Equal to Threshold

Medium Array Binary Search Matrix Prefix Sum Leetcode Link

Problem Description

Given a matrix of integers `mat` with dimensions `m x n` and an integer `threshold`, the task is to find the maximum side length of a square submatrix where the sum of its elements is less than or equal to `threshold`. If no such square exists, we should return `0`. The problem challenges us to do this efficiently across all possible submatrices in a large matrix.

Intuition

To solve this problem, we use dynamic programming to precompute the sums of submatrices to quickly assess whether a square of a given size meets the threshold criteria. The intuition behind the approach is to use a binary search algorithm to efficiently find the maximum side length, combined with a sum lookup that leverages the precomputed sums of submatrices.

First, we create an auxiliary matrix `s` where each element `s[i][j]` represents the sum of elements in the rectangle from the top-left corner of the matrix to `(i, j)`. This auxiliary matrix allows us to compute the sum of any submatrix in constant time by using the inclusion-exclusion principle.

The binary search is done on the side length of the square. For each possible side length `k`, we use the auxiliary matrix to check if there exists at least one square submatrix of size `k x k` with a sum less than or equal to `threshold`. If such a square exists for a side length `k`, we know that any smaller square will also satisfy the condition, so we continue the search for larger squares. If there is no such square, we decrease the side length and continue the search. This process is repeated until we narrow down to the maximum square size that satisfies the sum condition.

By combining binary search for efficiency and dynamic programming for fast submatrix sum calculation, the algorithm efficiently solves the problem without checking every possible square individually, which would be computationally expensive.

Solution Approach

The provided solution leverages dynamic programming and binary search to find the maximum side-length of a square with a sum less than or equal to the provided `threshold`.

Dynamic Programming

To calculate the sum of any submatrix quickly, an auxiliary matrix `s` is created. This 2D array stores the cumulative sum for every index `(i, j)` representing the sum of elements from `(0, 0)` to `(i, j)`.

The sum for each element `s[i][j]` is obtained by:

```
s[i][j] = s[i - 1][j] + s[i][j - 1] - s[i - 1][j - 1] + mat[i - 1][j - 1]
```

This formulation is based on the inclusion-exclusion principle, where `s[i - 1][j] + s[i][j - 1]` adds the rectangle above and to the left of `(i, j)`, but since the intersection (`mat[0][0]` to `mat[i - 1][j - 1]`) is added twice, it is subtracted once with `- s[i - 1][j - 1]`. Finally, `mat[i - 1][j - 1]` is added to include the current cell.

Binary Search

Binary search is used to find the maximum valid square side length. The search range is from `0` to the minimum of the matrix's height and width, as this represents the largest potential square.

Within each iteration of binary search:

1. The middle of the current range (`mid`) is selected as the candidate side-length for the square.
2. A check function, `check(k)`, is called to verify if there exists at least one `k x k` square whose sum is within `threshold`.
3. If such a square exists for `mid`, the lower half (`l`) of the search is updated to `mid`, as we want to continue searching for possibly larger squares that satisfy the condition.
4. If no such square exists for `mid`, the upper half (`r`) is updated to `mid - 1`.
5. The binary search loop continues until `l` is no longer less than `r`.

The `check(k)` function iterates over all possible positions for the top-left corner of a `k x k` square. It uses the previously computed auxiliary matrix `s` to determine if the sum of the square is less than or equal to `threshold` by performing a simple subtraction operation that removes the unnecessary areas, similar to how `s` was populated.

```
v = s[i + k][j + k] - s[i][j + k] - s[i + k][j] + s[i][j]
```

Time Complexity

The overall time complexity of the solution is `O(m * n * log(min(m, n)))`, where `m` and `n` are the dimensions of the input matrix. This is due to the binary search varying between `1` and `min(m, n)` and the sum check performed within the `check(k)` function that takes `O(m * n)` time.

As a result, the solution is efficient and effective, providing a fast way to compute the largest square within `threshold` while avoiding the overhead of calculating every possible square sum independently.

Example Walkthrough

Let's illustrate the solution approach using a small example. Suppose we have the following 3×3 matrix `mat` and a `threshold` of 8:

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

We want to find the maximum side length of a square submatrix where the sum of its elements is less than or equal to `threshold`.

First, we build the auxiliary matrix `s` to store cumulative sums:

```
s = [
  [0, 0, 0, 0],
  [0, 1, 3, 6],
  [0, 5, 12, 21],
  [0, 12, 27, 45]
]
```

Each element `s[i][j]` represents the sum of elements from `(0, 0)` to `(i-1, j-1)` in the original matrix.

Next, we use binary search to find the maximum side length of the square. We start the search with `left = 0` and `right = min(m, n) = 3`.

1. The first `mid` value is `(0 + 3) / 2 = 1.5`, rounded down to `1`.
2. We call `check(1)` for each possible `1x1` square and see if any square's sum is less than or equal to the threshold. For `1x1` squares, all values from the original matrix qualify, as they are all less than `8`.
3. Since `check(1)` is true, we update `left = mid + 1`, and the new range becomes `2` to `3`.

We repeat this process:

1. Our next `mid` is `(2 + 3) / 2 = 2.5`, rounded down to `2`.
2. We call `check(2)` for each possible `2x2` square. For instance, for the top-left `2x2` square:
   - Calculate its sum using the auxiliary matrix `s`: `s[2][2] - s[0][2] - s[2][0] + s[0][0] = 12 - 0 - 0 + 0 = 12`, which is greater than the threshold.
   - Hence, no `2x2` square has a sum less than or equal to the threshold.
3. Since `check(2)` is false, we update `right = mid - 1`, and the range becomes `2` to `1`.

Since `left` is not less than `right`, the binary search terminates, and we find that the maximum side length of a square submatrix with a sum less than or equal to the threshold is `1`.

Python Solution

```python
from typing import List

class Solution:
    def maxSideLength(self, mat: List[List[int]], threshold: int) -> int:
        # Define a function to check if a square with side k fits the threshold
        def check_square_fits(k: int) -> bool:
            for i in range(rows - k + 1):
                for j in range(cols - k + 1):
                    # Compute the sum of the square using prefix sum technique
                    square_sum = prefix_sum[i + k][j + k] - prefix_sum[i][j + k] - prefix_sum[i + k][j] + prefix_sum[i][j]
                    # If the sum is within the threshold, then square of side k fits
                    if square_sum <= threshold:
                        return True
            return False

        # Matrix dimensions
        rows, cols = len(mat), len(mat[0])

        # Calculate the prefix sum matrix for efficient area sum calculation
        # Padding with an extra row and column filled with 0 for easy calculation
        prefix_sum = [[0] * (cols + 1) for _ in range(rows + 1)]
        for i in range(1, rows + 1):
            for j in range(1, cols + 1):
                prefix_sum[i][j] = prefix_sum[i - 1][j] + prefix_sum[i][j - 1] - prefix_sum[i - 1][j - 1] + mat[i - 1][j - 1]

        # Initialize binary search bounds
        left, right = 0, min(rows, cols)

        # Perform binary search to find the maximum square side length
        while left < right:
            mid = (left + right + 1) // 2 # Use integer division for Python 3
            # Check if there's a square with the current mid side length that fits the threshold
            if check_square_fits(mid):
                left = mid
            else:
                right = mid - 1

        # The left pointer now points to the maximum size square that fits the threshold
        return left

```

Java Solution

```java
class Solution {
    private int numRows;           // Number of rows in the matrix
    private int numCols;           // Number of columns in the matrix
    private int threshold;         // Threshold for the maximum sum of the square sub-matrix
    private int[][] prefixSum;     // Prefix sum matrix

    public int maxSideLength(int[][] mat, int threshold) {
        this.numRows = mat.length;
        this.numCols = mat[0].length;
        this.threshold = threshold;
        this.prefixSum = new int[numRows + 1][numCols + 1];

        // Build the prefix sum matrix
        for (int i = 1; i <= numRows; ++i) {
            for (int j = 1; j <= numCols; ++j) {
                prefixSum[i][j] = prefixSum[i - 1][j] + prefixSum[i][j - 1]
                                - prefixSum[i - 1][j - 1] + mat[i - 1][j - 1];
            }
        }

        int left = 0, right = Math.min(numRows, numCols);
        // Binary search for the maximum side length of a square sub-matrix
        while (left < right) {
            int mid = (left + right + 1) >> 1;
            if (canFindSquareWithMaxSum(mid)) {
                left = mid;
            } else {
                right = mid - 1;
            }
        }
        return left;
    }

    // Check if there's a square sub-matrix of side length k with a sum less than or equal to the threshold
    private boolean canFindSquareWithMaxSum(int sideLength) {
        for (int i = 0; i <= numRows - sideLength; ++i) {
            for (int j = 0; j <= numCols - sideLength; ++j) {
                int sum = prefixSum[i + sideLength][j + sideLength]
                        - prefixSum[i][j + sideLength]
                        - prefixSum[i + sideLength][j] + prefixSum[i][j];
                if (sum <= threshold) {
                    return true;
                }
            }
        }
        return false;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <cstring>

class Solution {
public:
    int maxSideLength(vector<vector<int>>& matrix, int threshold) {
        int rows = matrix.size();
        int cols = matrix[0].size();

        // Prefix sum array with extra row and column to handle boundary conditions.
        int prefixSum[rows + 1][cols + 1];
        memset(prefixSum, 0, sizeof(prefixSum));

        // Calculate prefix sum for the matrix.
        for (int i = 1; i <= rows; ++i) {
            for (int j = 1; j <= cols; ++j) {
                prefixSum[i][j] = prefixSum[i - 1][j] + prefixSum[i][j - 1] - prefixSum[i - 1][j - 1] + matrix[i - 1][j - 1];
            }
        }

        // Lambda function that checks if there's a square with side length k
        // that has a sum less than or equal to the threshold.
        auto isSquareValid = [&](int k) {
            for (int i = 0; i <= rows - k; ++i) {
                for (int j = 0; j <= cols - k; ++j) {
                    // Calculate the sum for the current square.
                    int squareSum = prefixSum[i + k][j + k] - prefixSum[i][j + k] -
                                    prefixSum[i + k][j] + prefixSum[i][j];

                    if (squareSum <= threshold) {
                        return true;
                    }
                }
            }
            return false;
        };

        // Binary search to find the maximum valid side length.
        int left = 0, right = min(rows, cols);
        while (left < right) {
            // Midpoint of the current search range, rounding up.
            int mid = (left + right + 1) / 2;
            if (isSquareValid(mid)) {
                left = mid; // Mid is possible, search higher.
            } else {
                right = mid - 1; // Mid is too large, search lower.
            }
        }

        return left; // Left will contain the maximum valid side length.
    }
};

```

Typescript Solution

```typescript
function maxSideLength(mat: number[][], threshold: number): number {
    const rows = mat.length; // Total number of rows in the matrix
    const cols = mat[0].length; // Total number of columns in the matrix
    const prefixSum: number[][] = Array.from({ length: rows + 1 }, () => Array(cols + 1).fill(0)); // Initialize prefix sum matrix

    // Compute the prefix sum of the matrix
    for (let i = 1; i <= rows; ++i) {
        for (let j = 1; j <= cols; ++j) {
            prefixSum[i][j] = prefixSum[i - 1][j] + prefixSum[i][j - 1] - prefixSum[i - 1][j - 1] + mat[i - 1][j - 1];
        }
    }

    // Check function to verify if there exists a square with side length k whose sum is below the threshold
    const hasValidSquare = (k: number): boolean => {
        for (let i = 0; i <= rows - k; ++i) {
            for (let j = 0; j <= cols - k; ++j) {
                const sum = prefixSum[i + k][j + k] - prefixSum[i + k][j] - prefixSum[i][j + k] + prefixSum[i][j];
                if (sum <= threshold) {
                    return true; // Found valid square
                }
            }
        }
        return false; // No valid square found
    };

    let left = 0; // Lower bound of the side length search space
    let right = Math.min(rows, cols); // Upper bound of the side length search space

    // Binary search to find the maximum side length of a square with a sum less than or equal to the threshold
    while (left < right) {
        const mid = Math.floor((left + right + 1) / 2); // Mid-point of the current search space
        if (hasValidSquare(mid)) {
            left = mid; // The square side length can be at least 'mid', continue searching right
        } else {
            right = mid - 1; // 'mid' is too large, continue searching left
        }
    }

    return left; // The largest valid side length found
}

```

Time and Space Complexity

Time Complexity

The provided code operates in several steps. The computation of the prefix sum matrix, `s`, takes `O(m * n)` time where `m` is the number of rows and `n` is the number of columns in the input matrix, `mat`. This is because it needs to iterate over every element in `mat` to compute the sum.

The check function, which checks if a square with side `k` has a sum less than or equal to the `threshold`, is `O(m * n)` as well for each individual `k`, since, in the worst case, it has to iterate over the entire matrix again, checking each possible top-left corner of a square.

Finally, the binary search performed in the last part of the code runs in `O(log(min(m, n)))` time. Since the binary search calls the `check` function at each iteration, the combined time complexity for the search routine with subsequent checks is `O(m * n * log(min(m, n)))`.

Hence, the overall time complexity of the code is `O(m * n + m * n * log(min(m, n)))` which simplifies to `O(m * n * log(min(m, n)))`.

Space Complexity

Regarding space complexity, the additional space is used for the prefix sum matrix, `s`, which has dimensions `(m + 1) x (n + 1)`. Therefore, the space complexity is `O((m + 1) * (n + 1))`, which simplifies to `O(m * n)` since the +1 is relatively negligible for large `m` and `n`.