363. Max Sum of Rectangle No Larger Than K

Hard | Array | Binary Search | Matrix | Ordered Set | Prefix Sum | Leetcode Link

Problem Description

Given a 2D grid called `matrix`, with dimensions `m x n`, and an integer `k`, the problem asks to find the maximum sum of any rectangle within the matrix, where the sum must not exceed `k`. A rectangle in a matrix is defined by any contiguous block forming an area within the grid. The sum of a rectangle is simply the sum of all the elements it contains. We have the guarantee that there exists at least one such rectangle whose sum is no more than `k`.

Intuition

The significant complexity in solving this problem lies in dealing with two dimensions while trying to optimize for the sum condition. A common approach to problems involving subarray sums is to use a running total and consider differences of this cumulative sum to find subarrays that match certain criteria; however, that typically applies to one-dimensional arrays.

When extending this idea to two dimensions, one efficient method is to reduce the problem to a one-dimensional problem by fixing one dimension. We do this by selecting two rows at a time and then compressing those rows into a one-dimensional array where each element is the sum of the elements in the column between the two rows. This effectively converts the problem into the "Maximum Sum of Subarray No More Than K" for a one-dimensional array.

Once we have this one-dimensional array, we can use a cumulative sum array to store previous sums and a sorted set to efficiently check if there's a previous sum that, when subtracted from the current sum, gives a result that is as close to `k` as possible but not greater. The `SortedSet` in Python comes in handy because it maintains the elements in sorted order, allowing binary-search-like operations for finding the appropriate sums we're interested in. This is a classic case of using space to gain time: we're collecting possible sums in the sorted set at the expense of memory in order to save significant time on searching.

The used algorithm iterates over all possible pairs of rows, for each pair compresses them into a one-dimensional array, computes a running total for that array, and searches the sorted set to find the best match for the current running total that does not exceed `k`. The maximum of these matches is remembered as it represents the maximum sum rectangle for the chosen rows. This is repeated until all pairs of rows have been considered, and the overall maximum sum that does not exceed `k` is the solution to the problem.

Solution Approach

The solution uses a few techniques and data structures. Let's walk through the steps and the rationale behind them:

1. **Two-pointer Technique**: We iterate over all possible row pairs using two pointers: the outer loop variable `i` starting from the first row and the inner loop variable `j` ranging from `i` to the last row. This generates all combinations of row starts and ends for our rectangles.
2. **Cumulative Column Sums**: For each row pair `(i, j)`, we compute the cumulative sum for each column in this section of the matrix. We maintain a one-dimensional array `nums`, where `nums[h]` represents the sum of elements from row `i` to `j` in column `h`. This collapses our two-dimensional problem into a one-dimensional array problem.
3. **Prefix Sum and Sorted Set**: We create a variable `s` representing the cumulative sum as we iterate through `nums` and a `SortedSet` `ts` to store past cumulative sums. The sorted set allows us to effectively perform two crucial operations: adding a new cumulative sum and searching for a particular sum. We start with a zero in the set because that represents an empty rectangle, which conceptually aligns with having no rectangle at all (a sum of zero).
4. **Finding the Best Subarray Sum**: As we get the cumulative sum `s`, we're interested in finding some previous sum in our set `ts` such that the difference `s - previous_sum` is as close to `k` as possible without going over. To do this, we look for the smallest prefix sum in the set where the `s - prefix_sum` is still larger than `k`. We find this prefix using the function `ts.bisect_left(s - k)`, which does a binary search for the position where `s - k` could be inserted while maintaining order. This position points to the smallest number larger or equal to `s - k`.
5. **Maximizing the Subarray Sum**: After finding such a prefix sum (`ts[p]`), we check if the difference `s - ts[p]` is larger than our previously recorded answer but still not larger than `k` and update our answer `ans` accordingly.

This process repeats until all row combinations have been exhausted and the `ans` variable holds the maximum rectangle sum no larger than `k`.

The algorithm runs in O(`m^2 * n * log(n)`) time, where `m` is the number of rows and `n` is the number of columns. This complexity comes from the fact that there are O(`m^2`) row pairs, for each of which we go through `n` columns and perform a `log(n)` operation (binary search) in the sorted set for each one.

Example Walkthrough

Let's walk through the solution using a small example. Suppose we have the following `3x3` matrix and `k = 8`:

```
matrix = [
  [1, 0, 1],
  [0, -2, 3],
  [1, 2, -1]
]
```

Our goal is to find the maximum sum of any rectangle in this matrix that does not exceed `k`.

Step 1: Two-pointer Technique
We'll consider all possible pairs of rows in the matrix. For simplicity, let's look at the pair where the first row has index `i = 0` and the second row has index `j = 2`. This will cover the entire height of the matrix.

Step 2: Cumulative Column Sums
We combine rows `0` to `2` into a single one-dimensional array, summing them column-wise:

```
nums = [
  (matrix[0][0] + matrix[1][0] + matrix[2][0]),    // First column sum
  (matrix[0][1] + matrix[1][1] + matrix[2][1]),    // Second column sum
  (matrix[0][2] + matrix[1][2] + matrix[2][2])     // Third column sum
]

nums = [1, 0, 3]     // Columns summed between rows 0 and 2
```

Step 3: Prefix Sum and Sorted Set
We set `s = 0` and start with an empty `SortedSet ts = {0}` to store cumulatively the sums.

Step 4: Finding the Best Subarray Sum
As we go through the array `nums`, we add the element to `s` and search for the best subarray sum:

- For the first column, `s = 1`, we look for the smallest number in `{0}` such that when subtracted from `s` doesn't exceed `k`. The best we can do is `s - 0 = 1` which is less than `8` (our `k`).
- Now, we add `s` to our sorted set: `ts = {0, 1}`.
- Then for the second column, `s` becomes `1`. There is no change since `nums[1]` is `0`, so the potential sums are identical to the previous step.
- Next, for the third column, `s` becomes `1 + 3 = 4`. We look for the smallest number in `{0, 1}` such that `4 - num` is as close as possible to `k` without going over. In both cases, we are under `k`.

With every addition to `s`, we continue checking the sorted set `ts` for the best possible subarray sum, remembering to add `s` to `ts` each time to consider it for future subarrays.

Step 5: Maximizing the Subarray Sum
We continue the above process until all column sums for the current row pair are considered. The largest sum we can achieve without exceeding `k` is the one we aim for. In our small example, considering all the combinations, the maximum sum we should find under or equal to `k` is `4`.

By repeating Steps 1 to 5 for all row pairs `(i, j)`, we exhaust all possible rectangles in the matrix, and `ans` is updated each time we find a larger sum that is within our constraint of `k`. In this case, after considering all possible rectangles, we find that the maximum sum that does not exceed `k` is `4`, which comes from the sub-matrix:

```
[1, 0, 1]
[0, -2, 3]
[1, 2, -1]
```

This is just a simple example that demonstrates the method; in a larger matrix with more variable elements, the sorted set would play a more significant role in efficiently finding the possible rectangle sums.

Python Solution

```python
from sortedcontainers import SortedSet
from math import inf

class Solution:
    def max_sum_submatrix(self, matrix, k):
        # Number of rows and columns in the matrix
        num_rows, num_cols = len(matrix), len(matrix[0])
        # This variable will store the maximum sum of the submatrix that doesn't exceed 'k'
        max_sum = -inf

        # Iterate over the starting row of our submatrix
        for starting_row in range(num_rows):
            # A list to store row sums
            row_sums = [0] * num_cols

            # Iterate over the ending row of our submatrix
            for ending_row in range(starting_row, num_rows):
                # Update the row sums by adding corresponding values from the new row
                for col in range(num_cols):
                    row_sums[col] += matrix[ending_row][col]

                # Initialize current sum and create a sorted set to store prefix sums
                current_sum = 0
                sorted_prefix_sums = SortedSet([0])

                # Iterating over the cumulative sum of row elements
                for sum_value in row_sums:
                    current_sum += sum_value
                    # Find the index of the smallest prefix sum so that
                    # current_sum - prefix_sum <= k and we get the biggest current_sum possible
                    index = sorted_prefix_sums.bisect_left(current_sum - k)
                    # If such index exists in the sorted set, update the max_sum
                    if index != len(sorted_prefix_sums):
                        max_sum = max(max_sum, current_sum - sorted_prefix_sums[index])
                    # Add the current prefix sum to the sorted set
                    sorted_prefix_sums.add(current_sum)

        # Return the maximum sum found that is less than or equal to 'k'
        return max_sum

```

Java Solution

```java
class Solution {
    public int maxSumSubmatrix(int[][] matrix, int k) {
        int rows = matrix.length;
        int cols = matrix[0].length;
        final int infinity = Integer.MAX_VALUE;
        int maxSum = -infinity;
        // Iterating over the starting row for submatrix
        for (int startRow = 0; startRow < rows; ++startRow) {
            int[] columnSums = new int[cols];
            // Extending the submatrix to each possible end row
            for (int endRow = startRow; endRow < rows; ++endRow) {
                // Summing up each column element of the current submatrix
                for (int col = 0; col < cols; ++col) {
                    columnSums[col] += matrix[endRow][col];
                }
                int currentSum = 0;
                TreeSet<Integer> set = new TreeSet<>();
                // Adding a base to consider subarrays starting from the first element
                set.add(0);
                // Traversing the cumulative column sums to find the submatrix with the sum closest to k
                for (int sum : columnSums) {
                    currentSum += sum;
                    // Checking if there is a prefix sum that, if removed, leaves a sum close to k
                    Integer minSum = set.ceiling(currentSum - k);
                    if (minSum != null) {
                        maxSum = Math.max(maxSum, currentSum - minSum);
                    }
                    set.add(currentSum);
                }
            }
        }
        return maxSum;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int maxSumSubmatrix(vector<vector<int>>& matrix, int k) {
        // Get the number of rows and columns in the matrix
        int rowCount = matrix.size(), colCount = matrix[0].size();
        // Initialize the answer with the smallest integer value
        const int INF = INT_MIN; // INT_MIN is from climits and more standard than 1 << 30
        int maxSum = INF;

        // Outer loop - Start of the row to consider for submatrices
        for (int rowStart = 0; rowStart < rowCount; ++rowStart) {
            // Initialize an array to store the sum of elements in current submatrix columns
            vector<int> columnSums(colCount, 0);
            // Inner loop - End of the row to consider for submatrices
            for (int rowEnd = rowStart; rowEnd < rowCount; ++rowEnd) {
                // Sum up the column elements in the current submatrix
                for (int col = 0; col < colCount; ++col) {
                    columnSums[col] += matrix[rowEnd][col];
                }
                // Use a set to help us find the rectangle with max sum not exceed k
                set<int> accumulatedSums;
                // Start with zero as an accumulated sum to compare
                accumulatedSums.insert(0);
                // Initialize the sum for the current submatrix
                int currSum = 0;
                // Iterate through each column sum
                for (int sum : columnSums) {
                    currSum += sum;
                    // Find the smallest number in accumulatedSums such that: number >= currSum - k
                    auto it = accumulatedSums.lower_bound(currSum - k);
                    // If such a number is found, update maxSum if necessary
                    if (it != accumulatedSums.end()) {
                        maxSum = max(maxSum, currSum - *it);
                    }
                    // Add the current sum to the set of accumulated sums
                    accumulatedSums.insert(currSum);
                }
            }
        }
        // Return the maximum submatrix sum such that it does not exceed k
        return maxSum;
    }
};

```

Typescript Solution

```typescript
// Defines a comparator type that compares two items and returns a number.
type Compare<T> = (lhs: T, rhs: T) => number;

// Represents a node in a red-black tree with added functionality.
interface RBTreeNode<T = number> {
  data: T;
  count: number;
  left: RBTreeNode<T> | null;
  right: RBTreeNode<T> | null;
  parent: RBTreeNode<T> | null;
  color: number;
}

// Helper function to determine if a given node is on the left of its parent.
function isOnLeft<T>(node: RBTreeNode<T>): boolean {
  return node === node.parent!.left;
}

// Helper function to check if a node has a red child.
function hasRedChild<T>(node: RBTreeNode<T>): boolean {
  return !!((node.left && node.left.color === 0) || (node.right && node.right.color === 0));
}

// Helper function to rotate a given node to the left.
// ...

// Helper function to rotate a given node to the right.
// ...

// Helper function to swap colors between two nodes.
// ...

// Helper function to swap data between two nodes.
// ...

// ... Other necessary RBTree methods ...

// Implements the red-black tree.
function createRBTree<T>(compare: Compare<T>): RBTreeNode<T> | null {
  let root: RBTreeNode<T> | null = null;
  // Implement the methods of RBTree (insert, delete, find, rotations, etc.) here.
  // ...
  return root;
}

// ... Other necessary TreeSet and TreeMultiSet methods ...

// Represents a collection of unique elements with a defined order.
function createTreeSet<T>(collection: T[] = [], compare: Compare<T>): TreeSet<T> {
  const tree: RBTreeNode<T> | null = createRBTree(compare);
  // Functionality to support Tree operations like add, delete, find, etc.
  // ...
  return {} as TreeSet<T>; // Replace with actual TreeSet implementation.
}

// Represents a collection that allows duplicates and has a defined order.
function createTreeMultiSet<T>(collection: T[] = [], compare: Compare<T>): TreeMultiSet<T> {
  const tree: RBTreeNode<T> | null = createRBTree(compare);
  // Functionality to support MultiSet operations like add, delete, count, etc.
  // ...
  return {} as TreeMultiSet<T>; // Replace with actual TreeMultiSet implementation.
}

// Computes the max sum of a rectangle in the matrix that does not exceed `k`.
function maxSumSubmatrix(matrix: number[][], k: number): number {
  const m = matrix.length;
  const n = matrix[0].length;
  let maxSum = -Infinity;

  // Computes cumulative sum for submatrices and uses TreeSet to find
  // subrectangle with the largest sum that doesn't exceed `k`.
  for (let i = 0; i < m; ++i) {
    const cumulativeSum: number[] = new Array(n).fill(0);

    for (let j = i; j < m; ++j) {
      // Update the cumulative sum by adding the new row's elements.
      for (let h = 0; h < n; ++h) {
        cumulativeSum[h] += matrix[j][h];
      }

      // Use TreeSet to compute the maximum sum subrectangle within `k`.
      const treeSet: TreeSet<number> = createTreeSet<number>([], (lhs, rhs) => lhs - rhs);
      // ... Perform the necessary TreeSet operations ...

      // Update the maximum sum found so far.
      // ...
    }
  }

  return maxSum;
}

// Define additional methods and variables here that are necessary for the global implementation.
// ...

```

Time and Space Complexity

The given code implements an algorithm to find the maximum sum of a submatrix in a 2D array that is no larger than `k`. Here's the analysis of its time and space complexity:

Time Complexity

- There are two nested loops (loop over `i` and `j`) that iterate over the rows of the matrix. In the worst case, both loops iterate `m` times, where `m` is the number of rows. This part contributes `O(m^2)` to the time complexity.
- Inside the nested loops, there's another loop over `h` that iterates `n` times where `n` is the number of columns, to sum up the elements in the current rectangle (`nums`). This loop runs `n` times for each iteration of the outer two loops, adding an `O(m^2 * n)` factor.
- For each `nums` subarray created, we perform a binary search and insertion into a sorted set `ts`. The SortedSet operations are logarithmic with respect to the size of the set. The set could potentially contain up to `n` elements in the worst case, so each operation (`bisect_left` and `add`) will take `O(log n)`. As these sorted set operations are done for each element in `nums`, this results in an additional `O(n log n)` term.
- Combining the factors, the overall time complexity is `O(m^2 * n * n * log n) = O(m^2 * n^2 * log n)`.

Space Complexity

- The space complexity is mainly determined by the space needed to store the `nums` array and the sorted set `ts`.
- The array `nums` has a size of `n`, so it contributes `O(n)` to the space complexity.
- The sorted set `ts` can contain up to `n` prefix sums, contributing another `O(n)` to the space complexity.
- As these are the largest allocations and no other allocations grow with respect to `m` or `n`, the total space complexity is `O(n)`.

In conclusion, the time complexity of the code is `O(m^2 * n^2 * log n)` and the space complexity is `O(n)`.