378. Kth Smallest Element in a Sorted Matrix

Medium | Array | Binary Search | Matrix | Sorting | Heap (Priority Queue)

Problem Description

In this problem, you are given an `n x n` matrix, with both rows and columns sorted in ascending order. The task is to find the `kth` smallest element in the entire matrix, considering the elements as if they were in a single, sorted list. Note that we are looking for the `kth` smallest in terms of order, not the `kth` unique value.

The challenge specifies that we must identify this `kth` smallest element efficiently, especially concerning memory usage. The solution should use an approach that is more memory-efficient than simply using a flat list of all matrix elements, which would require `O(n^2)` space.

Intuition

The brute force method of tackling this problem would be to create an array from all elements in the matrix, sort it, and then pick the `kth` element. However, this approach is not efficient, especially in terms of space complexity. The first hint towards an optimal solution comes from the fact that the rows and columns are individually sorted. This property suggests that a more nuanced method of searching could be applied.

Binary search comes to the rescue here, but instead of the regular binary search on a list, we apply it to a value range from the smallest element in the matrix (the top-left corner) to the largest element (the bottom-right corner). The crucial observation is that every element less than or equal to a value `mid` within this range qualifies as a candidate for being one of the `k` smallest elements.

The `check` function essentially counts how many elements in the matrix are less than or equal to `mid`. This count is compared with `k` to decide if we need to search higher or lower. Each time, we adjust the value of `mid` and repeat the process, converging on the value of the `kth` smallest element. The logic being, if there are at least `k` numbers less than or equal to `mid`, then the `kth` smallest number is at most `mid`.

Through the binary search approach, we gradually narrow down the range until `left` and `right` converge, at which point `left` will be our `kth` smallest element. The binary search approach ensures that we only need constant extra space (for the variables used), and the time complexity is much more favorable than sorting all elements.

Solution Approach

The problem is efficiently solved using a binary search algorithm. However, instead of performing the binary search on the matrix elements directly, we apply it to the range of possible element values within the matrix.

Here's a step-by-step breakdown of the implementation process:

- We define a helper function `check` that takes the matrix, a `mid` value, the integer `k`, and the matrix size `n`. Its goal is to determine if there are at least `k` elements smaller than or equal to `mid`. It initializes `count` to 0, starting from the bottom-left corner of the matrix (`i = n - 1, j = 0`) and moving upwards or rightwards depending on the comparison between the matrix element and `mid`.
- For each `j`, if `matrix[i][j]` is less than or equal to `mid`, the code increments `count` by `i + 1` since all elements above in the same column are also less than or equal to `mid` due to the matrix's sorting property. Then, it moves to the next column by increasing `j`.
- If `matrix[i][j]` is greater than `mid`, it means that element and the ones below it in the same row cannot be part of the `k` smallest elements, so we move one row up by decrementing `i`.
- The binary search initializes `left` as the smallest element in the matrix (`matrix[0][0]`) and `right` as the largest (`matrix[n - 1][n - 1]`).
- In a loop, `mid` is calculated as the average of `left` and `right`. If the `check` function returns `true`, meaning there are at least `k` elements smaller than or equal to `mid`, the `right` boundary is brought down to `mid` because one of the `k` smallest elements could be `mid` or some smaller number.
- Otherwise, if the `check` function returns `false`, there are fewer than `k` elements smaller than `mid`, and we must search in the larger half by updating `left` to `mid + 1`.
- The loop continues until `left` and `right` converge. When `left` equals `right`, this value is the `kth` smallest element in the matrix, as there are `k - 1` elements smaller than it, and every element larger than `right` is not within the `k` smallest.
- We return `left` as the final result.

The algorithm takes advantage of the sorted rows and columns property of the matrix, which allows us to count the number of elements less than or equal to any given `mid` value in `O(n)` time. Combined with binary search running in `O(log(max-min))` time, where `max` and `min` are the largest and smallest numbers in the matrix, the overall time complexity is `O(n * log(max-min))`. Since this method only requires constant extra space, it efficiently meets the memory complexity challenge in the problem description.

Example Walkthrough

Let's walk through an example with a `3x3` matrix and find the `5th` smallest element. The matrix is as follows:

```
 1  5  9
10 11 13
12 13 15
```

According to the description, both rows and columns are sorted in ascending order. We are looking for the `5th` smallest element.

1. Initialize our binary search with `left` as the smallest element (`matrix[0][0]` which is `1`) and `right` as the largest element (`matrix[2][2]` which is `15`).
2. Calculate `mid` as the average of `left` and `right`. In the first instance, `mid = (1 + 15) / 2 = 8`.
3. The `check` function will count how many elements are less than or equal to `mid` (8). Starting from the bottom-left of the matrix (`matrix[2][0]` which is `12`):
   - Since `12 > 8`, move one row up to `matrix[1][0]` which is `10`.
   - `10` is also greater than `8`, move up again to `matrix[0][0]` which is `1`.
   - `1` is less than `8`, so we count all elements in this column (`i+1`), giving us `1`. Move to the next column.
   - In the second column, `matrix[0][1]` is `5`, which is less than `8`. We count all elements in this column (`i+1`), now our total is `2 + 1 = 3`. Move to the next column.
   - In the third column, `matrix[0][2]` is `9`, which is greater than `8`. Since we can't move up, we stop.
4. With `3` counted elements less than or equal to `mid`, and we are looking for the `5th` element, our `mid` value is too low. Adjust `left` to `mid + 1`, making `left = 9`.
5. Repeat the binary search steps:
   - Update `mid` to `(9 + 15) / 2 = 12`.
   - Using the `check` function, we now find that the elements `1`, `5`, `9`, `10`, and `11` are all less than or equal to `12`, giving us `5` elements.
   - Since we found exactly `5` elements, and we are looking for the `5th` smallest, our `mid` value might be the answer, but there could be smaller values we haven't yet considered. Update `right` to `mid`.
6. `left` is now `9` and `right` becomes `12`. We now look for `mid` which is `(9 + 12) / 2 = 10.5`, we use `10` for integer division.
7. Reapply the `check` function:
   - `1`, `5`, and `9` are less than or equal to `10`. That's `3` elements in total.
   - `10` itself is less than or equal to `10`, so count all elements in that column (`3` total), giving us `3 + 3 = 6` elements.

   However, we only need `5` elements, and since we have more than `5`, we bring `right` down to `mid`, making `right = 10`.
8. As `left` and `right` are now both `10`, we have converged to the answer.

The `5th` smallest element in this `3x3` matrix is `10`.

Through this process, we only kept track of the range of potential `kth` smallest values and never stored or sorted the entire set of matrix elements. This way, the algorithm efficiently adheres to the constraints of optimizing space complexity.

Python Solution

```python
from typing import List

class Solution:
    def kthSmallest(self, matrix: List[List[int]], k: int) -> int:
        # Helper function to count the number of elements smaller or equal to mid
        def count_less_equal(mid, k, size):
            count = 0
            row, col = size - 1, 0  # Start with the bottom-left corner of the matrix

            while row >= 0 and col < size:
                if matrix[row][col] <= mid:
                    count += row + 1  # Add all the elements of the current column
                    col += 1  # Move to the next column
                else:
                    row -= 1  # Move to the previous row

            return count >= k  # Check if the count is greater than or equal to k

        size = len(matrix)
        # Set initial binary search bounds
        left, right = matrix[0][0], matrix[size - 1][size - 1]

        # Perform binary search
        while left < right:
            mid = (left + right) // 2  # Choose the middle value
            # If the count of numbers less than or equal to mid is k or more
            if count_less_equal(mid, k, size):
                right = mid  # Narrow down the search space to the lower half
            else:
                left = mid + 1  # Narrow down the search space to the upper half

        return left  # The kth smallest number

```

Java Solution

```java
class Solution {

    // This method finds the kth smallest element in a sorted matrix
    public int kthSmallest(int[][] matrix, int k) {
        int dimension = matrix.length; // The dimension of the matrix
        int low = matrix[0][0], high = matrix[dimension - 1][dimension - 1]; // Initialize the binary search bounds

        // Perform binary search
        while (low < high) {
            int mid = (low + high) >>> 1; // Calculate the middle value
            // If the count of numbers less than or equal to 'mid' is at least 'k', adjust the high bound
            if (countLessOrEqual(matrix, mid, k, dimension)) {
                high = mid;
            } else { // Else, adjust low bound
                low = mid + 1;
            }
        }
        // After the loop, 'low' is the kth smallest number
        return low;
    }

    // This helper method counts how many numbers are less than or equal to 'mid'
    private boolean countLessOrEqual(int[][] matrix, int mid, int k, int dimension) {
        int count = 0; // To store the count of elements
        int row = dimension - 1, col = 0; // Start from the bottom-left corner of the matrix

        // Iterate over the matrix
        while (row >= 0 && col < dimension) {
            // If current element is less than or equal to mid, move to the right and add the row count to the total
            if (matrix[row][col] <= mid) {
                count += (row + 1);
                col++;
            } else { // If current element is greater than mid, move upwards
                row--;
            }
        }
        // If the count is at least 'k', return true
        return count >= k;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Returns the kth smallest element in a sorted matrix.
    int kthSmallest(vector<vector<int>>& matrix, int k) {
        int n = matrix.size(); // Size of the matrix (since matrix is n x n)
        int left = matrix[0][0]; // Initialize 'left' to the smallest element
        int right = matrix[n - 1][n - 1]; // Initialize 'right' to the largest element

        // Binary search in the range of values of the matrix
        while (left < right) {
            int mid = left + (right - left) / 2; // Avoid potential overflow
            // If there are at least k elements less than or equal to mid, go to the left half
            if (check(matrix, mid, k, n)) {
                right = mid; // Move right towards the mid
            } else { // Otherwise, go to the right half
                left = mid + 1; // Move left towards the mid+1
            }
        }
        // 'left' now points to the kth smallest element
        return left;
    }

private:
    // Helper function to check if there are at least k elements less than or equal to 'mid'
    bool check(vector<vector<int>>& matrix, int mid, int k, int n) {
        int count = 0; // Count of elements less than or equal to 'mid'
        int i = n - 1; // Start from the bottom-left corner of the matrix
        int j = 0; // Start from the first column

        while (i >= 0 && j < n) {
            // If the current element is less than or equal to mid, move to the next column
            if (matrix[i][j] <= mid) {
                count += (i + 1); // Add all the elements of current column (since columns are sorted)
                ++j; // Move to the next column
            } else { // If the current element is greater than mid, move up
                --i; // Move to the previous row
            }
        }
        // Return true if count is greater than or equal to k
        return count >= k;
    }
};

```

Typescript Solution

```typescript
// Type declaration for a matrix of numbers.
type Matrix = number[][];

// Returns the kth smallest element in a sorted matrix.
function kthSmallest(matrix: Matrix, k: number): number {
    const n = matrix.length; // Size of the matrix (since matrix is n x n)
    let left = matrix[0][0]; // Initialize 'left' to the smallest element
    let right = matrix[n - 1][n - 1]; // Initialize 'right' to the largest element

    // Binary search in the range of values of the matrix
    while (left < right) {
        const mid = left + Math.floor((right - left) / 2); // Avoid potential overflow
        // If there are at least k elements less than or equal to mid, go to the left half
        if (check(matrix, mid, k, n)) {
            right = mid; // Move right towards the mid
        } else { // Otherwise, go to the right half
            left = mid + 1; // Move left towards mid+1
        }
    }
    // 'left' now points to the kth smallest element
    return left;
}

// Helper function to check if there are at least k elements less than or equal to 'mid'.
function check(matrix: Matrix, mid: number, k: number, n: number): boolean {
    let count = 0; // Count of elements less than or equal to 'mid'
    let i = n - 1; // Start from the bottom-left corner of the matrix
    let j = 0; // Start from the first column

    while (i >= 0 && j < n) {
        // If the current element is less than or equal to mid, move to the next column
        if (matrix[i][j] <= mid) {
            count += (i + 1); // Add all the elements of the current column (since columns are sorted)
            ++j; // Move to the next column
        } else { // If the current element is greater than mid, move up
            --i; // Move to the previous row
        }
    }
    // Return true if count is greater than or equal to k
    return count >= k;
}

```

Time and Space Complexity

Time Complexity

The time complexity of this algorithm can be determined by analyzing two main parts: the binary search operation and the check function that is called within the binary search.

1. **Binary Search**: The binary search is performed on the range of possible values in the matrix (from the smallest element `matrix[0][0]` to the largest element `matrix[n - 1][n - 1]`). Since these values are in the order of the matrix elements and not the size of the matrix, the number of iterations is determined by the number of bits in the numerical range (`log(max - min)`), where `max` is the maximum value and `min` is the minimum value in the matrix. The range here is from `matrix[0][0]` to `matrix[n - 1][n - 1]`, so the binary search will take `O(log(matrix[n-1][n-1] - matrix[0][0]))`.
2. **Check Function**: During each iteration of the binary search, the check function iterates through the matrix in a diagonal fashion starting from `(n-1, 0)` until either the first row or last column is reached. This function has a worst-case time complexity of `O(n)` since it could potentially go through each row once.

Multiplying the number of binary search iterations by the complexity of the check function yields a total time complexity of `O(n * log(matrix[n-1][n-1] - matrix[0][0]))`.

Space Complexity

The algorithm only uses a constant amount of extra space for variables used in the binary search and check functions (`left`, `right`, `mid`, `i`, `j`, `count`). There are no data structures used that grow with the size of the input. Thus, the space complexity is `O(1)`.