668. Kth Smallest Number in Multiplication Table

Hard Math Binary Search Leetcode Link

Problem Description

In this problem, we are dealing with a multiplication table that is conceptualized as a 2D matrix with `m` rows and `n` columns. The value in each cell of the matrix is the product of its row and column indices (assuming 1-indexed, i.e., indexing starting from 1). For example, value at `mat[i][j]` would be `i * j`. The task is to find the `k`th smallest element in this multiplication table.

To put it more concretely, if we flatten this table into a sorted single-dimensional array, we want to find the `k`th element in that array. However, constructing this array explicitly would be inefficient, especially for large `m` and `n`, which is why we need a more clever approach to solve this problem.

Intuition

We use binary search to find the `k`th smallest number in a more efficient way than sorting the entire multiplication table. Instead of constructing the multiplication table, we leverage its sorted properties indirectly.

The intuition is that every row in the multiplication table is a multiple of the first row. Given a number `x`, we can easily calculate how many numbers in the table are less than or equal to `x` by summing up the number of such elements in each row. This is because each row `i` contains the multiples of `i` up to `i*n`, and each multiple less than or equal to `x` counts towards our total. In other words, for each row `i`, the number of elements that do not exceed `x` is `min(x // i, n)`.

We utilize binary search on the range of possible values – from 1 to `m*n` which are all the possible values in the multiplication table. We look for the smallest number such that there are at least `k` numbers less than or equal to it in the multiplication table. The key observation here is that if there are more than `k` numbers less than or equal to a guess `mid` in the table, then our answer must be less than or equal to `mid`; otherwise, it must be greater.

By repeatedly narrowing our search range based on the count of elements up to the midpoint, we zero in on the `k`th smallest number. When the `left` and `right` pointers in our binary search meet, we have found the least number for which there are `k` or more numbers smaller or equal to it in the table, which by the properties of the search, will be exactly the `k`th smallest number.

Solution Approach

The solution utilizes a binary search algorithm to efficiently find the `k`th smallest number in the multiplication table. The binary search operates not on the multiplication table directly but on the value range from `1` to `m*n`, which are the potential candidates for our answer. The midpoint of this range in each iteration gives us a guess to validate.

Here is a walk-through of the solution step by step:

- We initially set `left` to `1` and `right` to `m*n`, covering the entire range of values in the multiplication table.
- While `left` is less than `right`, we calculate `mid` as the midpoint between `left` and `right`. In the provided code, the operation `(left + right) >> 1` efficiently computes the midpoint by adding `left` and `right` together and then doing a right bitwise shift by 1 (which is equivalent to integer division by 2).
- We need to count how many numbers in the table are less than or equal to `mid`. We initialize `cnt` to `0` for this purpose.
- We then iterate over each row `i` from `1` to `m`. For each row, we increment `cnt` by `min(mid // i, n)`. This gives us the total count of numbers not exceeding `mid` in the multiplication table.
- If `cnt` is at least `k`, it means we have at least `k` numbers in the multiplication table that are less than or equal to `mid`, and thus our `k`th smallest number is `mid` or lower. We then set our new `right` to `mid`.
- If `cnt` is less than `k`, it means we need a larger number to reach our target of `k` numbers less than or equal to it in the table. We then set our new `left` to `mid + 1`.
- This process is repeated, narrowing the search range until `left` and `right` converge, at which point `left` will have the value of the `k`th smallest number in the multiplication table.

By repeatedly halving the search space, the binary search ensures an efficient approach with a time complexity of O(m * log(m*n)), which is much more efficient than generating the multiplication table and sorting it, especially for large values of `m` and `n`.

Example Walkthrough

Let's illustrate the solution approach with an example. Suppose we have a multiplication table with `m = 3` rows and `n = 3` columns, and we want to find the `k`th smallest number in this table where `k = 5`.

The multiplication table looks like this:

```
1 1  2  3
2 2  4  6
3 3  6  9
```

The flattened sorted version of this table would be `[1, 2, 2, 3, 3, 4, 6, 6, 9]`, and clearly, the 5th number is `3`. But since we are not creating this array, let's proceed with the binary search approach.

1. We set `left` = 1 and `right` = 3*3 = 9.
2. Now we perform the binary search. We calculate `mid` by averaging `left` and `right`, so initially, `mid` would be (1 + 9) / 2 = 5.
3. With `mid` = 5, we count the numbers less than or equal to 5 in the multiplication table. For the first row, all three numbers (1, 2, 3) are ≤ 5, so that's 3 counts. For the second row, two numbers (2, 4) are ≤ 5. For the third row, one number (3) is ≤ 5. So our count `cnt` = 3 + 2 + 1 = 6.
4. Since `cnt` = 6 is greater than `k` = 5, we have more numbers than needed. Hence, `right` becomes `mid` = 5.
5. Repeat the binary search with `left` = 1 and `right` = 5. Our new `mid` is (1 + 5) / 2 = 3.
6. With `mid` = 3, we count again. In all three rows, there's exactly one number less than or equal to 3. The counts are 3, 1 and 1, respectively, for a total of `cnt` = 3+1+1 = 5.
7. Since `cnt` = 5 is exactly `k`, we could settle for this `mid`. However, the binary search algorithm doesn't stop until `left` and `right` converge.
8. We don't update `right` since `cnt` is not greater than `k`. We would update `left` to `mid + 1`, which now equals 4.
9. Now `left` = 4 and `right` = 5. Since `left` is not less than `right`, the while loop terminates.
10. We have found our `k`th smallest number which is `left` = 4.

However, we know from the hand-calculated array that the 5th smallest number is `3`, not `4`. This discrepancy is because the binary search approach narrows down to the number for which there are at least `k` numbers smaller or equal to it in the table, and it doesn't stop until the `left` boundary overtakes or meets the `right` boundary, which implies that the `left` boundary, in this case, may give us the first number that allows us to reach `k` count, but not necessarily the `k`th distinct number.

Thus, to correct the final output of our binary search case, we would instead take the `right` boundary as our answer when the loop terminates, because it was the last number that gave us a count that was exactly `k` before `left` was incremented past it.

For our example, the 5th smallest number in the multiplication table is indeed `3`, which would be the correct final value of `right` at the end of our binary search process.

Python Solution

```python
class Solution:
    def findKthNumber(self, m: int, n: int, k: int) -> int:
        # Initialize the search range between 1 and m*n
        left, right = 1, m * n

        # Binary search to find the k-th smallest number
        while left < right:
            mid = (left + right) // 2  # Use floor division for Python3
            count = 0

            # Count the number of values less than or equal to mid in the 2D multiplication table
            for i in range(1, m + 1):
                count += min(mid // i, n)

            # If the count is greater than or equal to k, search the left half
            if count >= k:
                right = mid
            # If the count is less than k, search the right half
            else:
                left = mid + 1

         # The left pointer will be at the k-th smallest number after exiting the loop
        return left

```

Java Solution

```java
class Solution {

    /**
     * Finds the kth smallest number in a multiplication table of size m x n.
     *
     * @param m The number of rows in the multiplication table.
     * @param n The number of columns in the multiplication table.
     * @param k The kth smallest number to find.
     * @return The value of the kth smallest number in the multiplication table.
     */
    public int findKthNumber(int m, int n, int k) {
        // Initialize the range of possible values for the kth number.
        int left = 1; // The smallest number in the multiplication table.
        int right = m * n; // The largest number in the multiplication table.

        // Perform a binary search to find the kth number.
        while (left < right) {
            // Midpoint of the current search range.
            int mid = left + (right - left) / 2;

            // Counter for the number of elements less than or equal to mid.
            int count = 0;

            // Iterate through each row to count numbers less than or equal to mid.
            for (int i = 1; i <= m; i++) {
                // In row i, since the numbers are i, 2i, 3i, ..., ni,
                // the count of numbers <= mid is min(mid / i, n).
                count += Math.min(mid / i, n);
            }

            // Check if the count of numbers <= mid is at least k.
            if (count >= k) {
                // If there are at least k numbers <= mid, the kth number is
                // in the left half of the search range.
                right = mid;
            } else {
                // If there are fewer than k numbers <= mid, the kth number is
                // in the right half of the search range.
                left = mid + 1;
            }
        }

        // The left pointer converges to the kth smallest number.
        return left;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int findKthNumber(int m, int n, int k) {
        // Initialize the binary search boundaries.
        int left = 1, right = m * n;

        // Execute the binary search.
        while (left < right) {
            // Calculate the middle point of the search space.
            int mid = left + (right - left) / 2;
            int count = 0; // This will hold the count of numbers less than or equal to 'mid'.

            // Count how many numbers are less than or equal to 'mid' for each row.
            for (int i = 1; i <= m; ++i) {
                // In the i-th row, numbers are i, 2i, 3i, ..., ni.
                // We determine how many of them are less than or equal to 'mid' by dividing 'mid' by i.
                // However, the count cannot exceed 'n' (the number of columns).
                count += std::min(mid / i, n);
            }

            // If the count is at least 'k', the desired value is at 'mid' or to the left of 'mid'.
            if (count >= k) {
                right = mid; // Narrow the search to the left half.
            } else {
                // If count is less than 'k', the desired value is to the right of 'mid'.
                left = mid + 1; // Narrow the search to the right half.
            }
        }

        // 'left' is now equal to the desired k-th number.
        return left;
    }
};

```

Typescript Solution

```typescript
function findKthNumber(m: number, n: number, k: number): number {
  // Initialize the binary search boundaries.
  let left = 1;
  let right = m * n;

  // Execute the binary search.
  while (left < right) {
    // Calculate the middle point of the search space.
    let mid = left + Math.floor((right - left) / 2);
    let count = 0; // This will hold the count of numbers less than or equal to 'mid'.

    // Count how many numbers are less than or equal to 'mid' for each row.
    for (let i = 1; i <= m; ++i) {
      // In the i-th row, numbers are i, 2i, 3i, ..., ni.
      // We determine how many of them are less than or equal to 'mid' by dividing 'mid' by i.
      // However, the count cannot exceed 'n' (the number of columns).
      count += Math.min(Math.floor(mid / i), n);
    }

    // If the count is at least 'k', the desired value is at 'mid' or to the left of 'mid'.
    if (count >= k) {
      right = mid; // Narrow the search to the left half.
    } else {
      // If the count is less than 'k', the desired value is to the right of 'mid'.
      left = mid + 1; // Narrow the search to the right half.
    }
  }

  // 'left' is now equal to the desired k-th number.
  return left;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code hinges on the binary search algorithm and the loop that calculates the count within each iteration. The binary search runs in `O(log(m*n))` time, as it is applied to a range from 1 to `m*n`. For each middle point `mid`, we run a loop from 1 to `m`, which results in a time complexity of `O(m)` for that segment of the code. Therefore, the total time complexity of the algorithm, which is a combination of these two operations, is `O(m * log(m*n))`.

Space Complexity

The space complexity of the code is `O(1)`. This is because we only use a constant amount of additional space, namely variables for `left`, `right`, `mid`, and `cnt`, regardless of the size of the input parameters `m`, `n`, and `k`.