793. Preimage Size of Factorial Zeroes Function

Hard Math Binary Search Leetcode Link

Problem Description

The problem asks us to find the number of non-negative integers `x` such that the number of trailing zeroes in `x!` (x factorial) is equal to a given integer `k`. Factorials of integers grow very rapidly, so evaluating them directly to count trailing zeroes isn't practical for large values of `x`. A trailing zero is created with a combination of `2` and `5`. Since there are more `2`s than `5`s in a factorial, the problem is effectively about finding how many times `5` is a factor in the numbers up to `x`.

For example:

- `f(5) = 1` because `5!` (1 * 2 * 3 * 4 * 5) ends in one zero.
- `f(10) = 2` because `10!` has two zeroes at the end.

The challenge is finding the function `f(x)` that counts trailing zeroes without calculating the actual factorial, and then determining how many such `x` satisfy `f(x) = k`.

Intuition

We need an efficient way to compute the number of trailing zeroes without calculating the factorial. We can do this with the following observation: the number of trailing zeroes in `x!` is equivalent to the number of times `5` appears as a factor in `1 * 2 * 3 * ... * x`.

So we define a helper function `f(x)` that returns the count of how many times `5` can divide `x!`. We don't want to compute `x!` directly, so instead, we divide `x` by `5` and recursively apply `f` to `x // 5`. This takes advantage of the fact that every multiple of `5` contributes at least one `5` to the factorial, every multiple of `25` contributes at least two, and so on.

Then, we apply a binary search strategy to find the value of `x` for which `f(x) = k`. Since `f(x)` is an increasing function, we can define a function `g(k)` that finds the smallest `x` such that `f(x)` is greater than or equal to `k`. The range we search over is from `0` to `5 * k`, because `5 * k` will always have at least `k` factors of `5`.

Using `g(k)`, we find the starting and ending range of `x` values that satisfy `f(x) = k` by finding the difference `g(k + 1) - g(k)`. Since the number of zeroes can jump over some numbers (like how there are no numbers that result in exactly one trailing zero), this difference will give us the count of `x` values that have exactly `k` trailing zeroes; it will be either `0` or `5`, never in between.

Solution Approach

The solution provided uses the helper function `f(x)` to identify how many times `5` can divide `x!`, which provides the number of trailing zeroes in `x!`. This computation is based on integer division and recursion, both of which are efficient and prevent the need for calculating large factorials.

The `g(k)` function uses a binary search through the built-in `bisect_left` function from Python's `bisect` module. This function is utilized to perform the binary search in an efficient manner, searching within a specified range. This range starts from `0` and ends at `5 * k` since, as mentioned earlier, we know that there must be at least `k` number of zeroes by `5 * k`.

Thus, the `bisect_left` takes three arguments:

- The range `range(5 * k)` which is the sequence of numbers where we apply the binary search.
- The value `k` that we are searching for.
- The key function `f` that transforms the current midpoint in the binary search into the number of trailing zeroes, deciding if we need to search left or right.

We call `g(k)` to locate the smallest `x` for which `f(x) >= k`, and then call `g(k + 1)` to find the smallest `x` for which `f(x) >= k + 1`. By subtracting these two results, we obtain the count of `x` values such that `f(x) = k`. If there is a set of numbers whose factorial has `k` trailing zeroes, the result will be `5`, otherwise it's `0`.

The reason why the count is `5` is that between two consecutive powers of `5` (for example, `25` and `125`), there are exactly `5` multiples of `5` that contribute to one additional zero (at `30`, `35`, `40`, `45`, and `50` for the range between `25` to `125`). Thus, when we find a number `x` where `f(x) = k`, there are `5` consecutive numbers maintaining the same number of trailing zeroes. If there is an increase in the count before we reach five values, then it means we've hit a power of `25`, `125`, etc., which adds more zeroes and skips certain `k` values, leading to `0` results for those `k`.

In conclusion, the solution combines the efficiency of binary search with the mathematical insight into factorials to solve the problem in log-linear time complexity, avoiding factorial calculations.

Example Walkthrough

Let's assume we want to find the number of non-negative integers `x` such that the number of trailing zeroes in `x!` equals `k`, where `k = 6`.

We'll begin by understanding that we're not looking for the factorial of `x` but the number of times `5` appears as a factor in `x!`. Every multiple of `5` contributes to a trailing zero in the factorial.

First, we need to determine how many times `5` divides into `x!`, which we do with the helper function `f(x)`. This function works as follows:

- `f(25) = 6` because `25` contributes `5` twice (25 and 20), and there are four more multiples of `5` (15, 10, 5, and the contribution from `20` again) up to 25.

Next, we use a binary search to find the smallest `x` that results in at least `k` trailing zeroes. We need to perform binary search on the interval from `0` to `5*k` (30 in this case), since `5*k` will have at least `k` trailing zeroes.

- We implement the `g(k)` function, which calls `bisect_left` to find the smallest `x` such that `f(x) >= k`.

Using the `f(x)` function, we perform a binary search starting with the left bound `l = 0` and the right bound `r = 5*k`:

- Midpoint `m = (l + r) // 2`.
- We check if `f(m) >= k`. If it is, we move our right bound to `m`; otherwise, we move our left bound to `m + 1`.
- We repeat until `l >= r`, at which point `l` will be the smallest integer for which `f(l) >= k`.

To find the number of values of `x` that give us exactly `k` trailing zeroes, we use `g(k + 1) - g(k)`:

- `g(6)` might give us 25 (that's when f(x) first reaches 6 zeros).
- `g(7)` gives us the value of `x` when `f(x)` reaches 7 trailing zeroes, let's say it's 30.

Subtracting these two values, `g(7) - g(6) = 30 - 25 = 5`, tells us there are 5 integers (`25, 26, 27, 28, 29`) whose factorials result in exactly 6 trailing zeroes.

Here is how the count is often `5`: Between `25` and `30`, there are increments of `1` in the number of factors of `5` (meaning one more trailing zero), but when we reach `30`, we're effectively reaching a multiple of `5^2` (since 30 is divisible by 5 two times), which results in one more trailing zero, causing the count to not exactly reach `k = 6` until `f(x)` hits the next factorial number that has `5` as a factor twice.

Consequently, we concluded that there are exactly five integers whose factorials end up with six trailing zeroes.

Python Solution

```python
from bisect import bisect_left

class Solution:
    def preimageSizeFZF(self, k: int) -> int:
        # Recursive function to count trailing zeroes in factorial of x
        def count_trailing_zeroes(x):
            # Base case: when x is 0, it contributes no trailing zeroes
            if x == 0:
                return 0
            # Recursive case: x contributes x // 5 trailing zeroes
            # plus count of trailing zeroes from all multiples of 5 less than x
            return x // 5 + count_trailing_zeroes(x // 5)

        # Function to find the left boundary of K's preimage using binary search
        # It finds the smallest number whose factorial has exactly k trailing zeroes
        def left_boundary_of_k(k):
            # Perform a binary search on the range [0, 5*k)
            # Since any number have at most k trailing zeroes in its factorial
            return bisect_left(range(5 * k), k, key=count_trailing_zeroes)

        # The size of the preimage for any valid k is always 5
        # This is because factorial(n) for values of n between two consecutive
        # multiples of 5 have the same number of trailing zeroes
        # We can use this by checking the difference between the left boundaries
        # of k and k+1. If there's a valid preimage size, it will always be 5.
        # If it is zero, then there is no number whose factorial ends with k zeroes.
        return left_boundary_of_k(k + 1) - left_boundary_of_k(k)

```

Java Solution

```java
class Solution {

    // This function calculates how many trailing zeroes a factorial number has.
    private int trailingZeroesInFactorial(long x) {
        if (x == 0) {
            return 0;
        }
        // The recursive call ensures calculation of trailing zeroes by summing
        // all occurrences of 5 factors which contribute to trailing zeroes.
        return (int) (x / 5) + trailingZeroesInFactorial(x / 5);
    }

    // This is the main function of the solution, which returns the size of the set
    // of numbers that contain exactly 'k' trailing zeroes in their factorial representation.
    public int preimageSizeFZF(int k) {
        // The size of the preimage is the difference between the first number that has
        // more than 'k' trailing zeroes and the first number that has more than 'k-1' trailing zeroes.
        // 'g(k+1)' finds the smallest number that has more than 'k' zeroes,
        // while 'g(k)' does the same for 'k-1'. Their difference is the size of the preimage.
        return firstNumberGreaterTrailingZeroes(k + 1) - firstNumberGreaterTrailingZeroes(k);
    }

    // This helper function calculates the smallest number that has
    // more trailing zeroes than the given 'k'.
    private int firstNumberGreaterTrailingZeroes(int k) {
        long left = 0;
        // A number can have at most k trailing zeroes if it is k factorial, which is a very large number
        // so we set a safe upper bound for binary search as '5 * k', because every 5 numbers there's
        // at least one number which contributes to another trailing zero.
        long right = 5L * k;

        // Perform a binary search to find the smallest number with more than 'k' trailing zeroes
        while (left < right) {
            long mid = (left + right) >> 1;

            // If the number of trailing zeroes for 'mid' is greater or equal to 'k',
            // we need to search to the left (decreasing the range).
            if (trailingZeroesInFactorial(mid) >= k) {
                right = mid;
            } else {
                // Otherwise, search to the right (increasing the range).
                left = mid + 1;
            }
        }
        // The left boundary of the search range is the first number with more
        // than 'k' trailing zeroes.
        return (int) left;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to calculate the number of numbers with k trailing zeroes.
    int preimageSizeFZF(int k) {
        return getCountOfNumbers(k + 1) - getCountOfNumbers(k);
    }

    // Helper function to find the count of numbers that have
    // at most k trailing zeroes in their factorial representation.
    int getCountOfNumbers(int k) {
        long long left = 0;
        long long right = static_cast<long long>(5) * k; // 5 * k is the initial upper bound for binary search.
        while (left < right) {
            long long mid = left + (right - left) / 2; // Safely calculate the mid to avoid overflow.
            if (getTrailingZeroes(mid) >= k) {
                right = mid; // If mid has at least k trailing zeroes, move the right bound to mid.
            } else {
                left = mid + 1; // Else, move the left bound to mid + 1.
            }
        }
        return static_cast<int>(left); // Narrowing conversion of long long to int.
    }

    // Function to calculate the number of trailing zeroes in x factorial.
    int getTrailingZeroes(long long x) {
        int result = 0;
        while (x > 0) {
            x /= 5; // Each division by 5 adds to the count of trailing zeroes.
            result += x;
        }
        return result;
    }
};

```

Typescript Solution

```typescript
// Global function to calculate the number of numbers with k trailing zeros.
function preimageSizeFZF(k: number): number {
    return getCountOfNumbers(k + 1) - getCountOfNumbers(k);
}

// Helper global function to find the count of numbers that have
// at most k trailing zeros in their factorial representation.
function getCountOfNumbers(k: number): number {
    let left: number = 0;
    let right: number = 5 * k; // Set the initial upper bound for the binary search.

    while (left < right) {
        let mid: number = left + Math.floor((right - left) / 2); // Safely calculate the mid to avoid overflow.

        if (getTrailingZeroes(mid) >= k) {
            right = mid; // If mid has at least k trailing zeros, move the right boundary to mid.
        } else {
            left = mid + 1; // Otherwise, move the left boundary to mid + 1.
        }
    }

    return left; // The narrowing conversion of long long to int is not necessary in TypeScript.
}

// Global function to calculate the number of trailing zeros in x factorial.
function getTrailingZeroes(x: number): number {
    let result: number = 0;

    while (x > 0) {
        x = Math.floor(x / 5); // Each division by 5 adds to the count of trailing zeros.
        result += x;
    }

    return result;
}

```

Time and Space Complexity

The given code defines a class `Solution` with a method `preimageSizeFZF` that calculates the number of integers which factorial ends with exactly `k` trailing zeroes. The method `preimageSizeFZF` calls two helper functions, `f` and `g`.

Time Complexity:

- The `f(x)` function is a recursive function that computes the number of trailing zeroes in the factorial of `x`. The time complexity of `f(x)` is `O(log_5(x))` because the recursive call is made `(log_5(x))` times, reducing `x` by a factor of 5 each time until `x` becomes 0.

- The `g(k)` function uses `bisect_left` from Python's standard library to find the leftmost value in the range `[0, 5 * k)` that makes `f(x)` equal to `k`. The bisect_left function performs a binary search which has a time complexity of `O(log(n))` where `n` is the length of the sequence being searched. In this case, `n` is `5 * k` making the complexity `O(log(5k))`.

- The `g(k)` function is called twice inside `preimageSizeFZF` (once with `k` and once with `k + 1`), so these two calls do not change the overall time complexity of the binary search.

- Combining the complexities, the total time complexity for each call to `g(k)` can be considered as `O(log(5k) * log_5(5k))` due to the bisect_left looping log(5k) times and each time calling f(x) with the complexity of `O(log_5(x))`. Since `log(5k)` and `log_5(5k)` are linearly dependent, we can simplify this to `O((log(k))^2)`.

- Finally, the overall time complexity of `preimageSizeFZF` is `O((log(k))^2)`.

Space Complexity:

- The space complexity of the `f(x)` function is `O(log_5(x))` because it involves recursive calls, and each call adds to the call stack until the base case is reached.

- The `g(k)` function's space complexity is `O(1)` as it only involves variables for the binary search and doesn't use any additional space that scales with the input size.

- Since `g(k)` is the dominant function and it does not use any significant extra space, the overall space complexity of `preimageSizeFZF` is the space complexity of the recursive calls of `f(x)`, which is `O(log_5(k))` (it's called with values close to `5 * k`, which does not change the logarithmic relationship).

In summary:

- Time Complexity: `O((log(k))^2)`
- Space Complexity: `O(log_5(k))`