440. K-th Smallest in Lexicographical Order

Hard Trie [Leetcode Link]

Problem Description

Given two numbers `n` and `k`, the task is to find the `kth` smallest number in the range `[1, n]` when the numbers are arranged in lexicographic order. Lexicographic order means the numbers are arranged like they would be in a dictionary, and not in numerical order. For example, with `n = 13`, the sequence in lexicographic order is `1, 10, 11, 12, 13, 2, 3, 4, 5, 6, 7, 8, 9`.

Intuition

To solve this problem, a key observation is understanding how numbers are arranged in lexicographic order, particularly focusing on the tree structure that this order implies. Every number can be seen as the root of a sub-tree, with its immediate children being obtained by appending a digit from `0-9` to the number.

For example, for the range `[1, 13]`, at the first level of this tree, we have `1`, `2`, ..., `9`. If we look closer at the sub-tree under `1`, the second level would contain `10`, `11`, `12`, `13`. No further numbers can be under `1` because that would exceed the range.

To find the `kth` lexicographic number, we can perform a sort of "guided" depth-first search through this tree.

The key insight is to skip over entire sub-trees that we don't need to count. The `count` function aids with this, as it counts how many numbers are present in the sub-tree that starts with the prefix `curr`. It essentially tells us how many steps we can jump directly without visiting each node. If there are enough steps to skip the current sub-tree and still have some part of `k` remaining, we move to the next sibling by adding `1` to `curr`. Otherwise, we go to the next depth of the tree by changing `curr` to `curr * 10` (appending a `0`).

Keep in mind that we subtract one from `k` initially because we start counting from `1` (the first number) and each time we either skip a sub-tree or go deeper, we need to decrement `k` accordingly until we traverse `k` steps.

Solution Approach

The `findKthNumber` method begins the search with the current number set to `1` and decrements `k` by `1` since we start from the first number.

The `count` function calculates the number of integers under the current prefix within the range up to `n`. It achieves this by iterating through the range, counting by moving on to child nodes in the sub-tree. For the given `curr` prefix, the loop steps through each level in its sub-tree by appending zeros (multiplying by `10`) to the current prefix and to the next smallest number with the same number of digits (`next`). The range from `curr` to `next - 1` represents a full set of children in lexicographic order under the current prefix.

The `min(n - curr + 1, next - curr)` calculation determines how many numbers to count at the current level (between the `curr` and the `next` prefix). It uses the smaller of either the total number in the range or the number up to the next prefix (not inclusive) to avoid going out of bounds of the range `[1, n]`.

In the main while loop of `findKthNumber`, we repeatedly decide whether to skip the current prefix's sub-tree or go deeper into the sub-tree:

- If `k >= cnt` (where `cnt` is the count of numbers under the current prefix), it means that the `kth` number is not under the current prefix's sub-tree. Therefore, we should skip to the next sibling by incrementing `curr` and decrementing `k` by `cnt`.
- If `k < cnt`, we go into the current sub-tree by moving to the next level (making `curr` ten times larger, that is, appending a '0'). Since we are still looking inside this sub-tree, we decrement `k` by `1` to account for moving one level deeper. Now, `k` represents the relative order of the search inside the sub-tree.

We repeat this process until `k` becomes `0`, which means we've found our `kth` number, represented by the current `curr` value.

This algorithm effectively skips large chunks of the search space, rapidly homing in on the `kth` smallest number in lexicographic order without having to examine every number individually.

Example Walkthrough

Let's illustrate the solution approach using `n = 15` and `k = 4`. We want to find the 4th smallest number in the range `[1, 15]` when the numbers are arranged in lexicographic order.

1. Begin with the current number set to `1` and decrement `k` by `1`, as we start from the first number. Now, `curr = 1` and `k = 3`.
2. Calculate the count of numbers under the current prefix using the `count` function. For `curr = 1`, this count includes `1, 10, 11, 12, 13, 14, 15`, which equals 7. Since `k = 3` is less than `cnt = 7`, we don't skip the current prefix's sub-tree.
3. Move to the next level in this sub-tree by making `curr = 1 * 10 = 10` and decrement `k` by `1` to account for the first number we currently at (which is `1`). Now, `curr = 10` and `k = 2`.
4. Since `k` is still positive, we again compute the count for the current `curr = 10`. The count includes `10` only since `11` would go to the next level of this sub-tree. So, `cnt = 1`.
5. Since `k >= cnt`, we skip the number `10` by incrementing `curr` to `11` and reducing `k` by `cnt` (which is `1`). Now, `curr = 11` and `k = 1`.
6. Now, with `curr = 11`, we again compute the count. It contains `11` only (since `12` goes to the next level), so `cnt = 1`.
7. Since `k >= cnt`, we skip the number `11` by incrementing `curr` to `12` and reducing `k` by `cnt`. Now, `curr = 12` and `k = 0`.

Since `k` is now `0`, we've reached the 4th smallest number in the lexicographic order, which is `12`.

This is how the solution approach helps in finding the kth smallest number in lexicographic order without having to list out or visit each number, by making use of the tree structure and skipping over the entire sub-trees when possible.

Python Solution

```python
class Solution:
    def find_kth_number(self, n: int, k: int) -> int:
        # Helper function to count the number of elements less than or equal to current prefix within n
        def count_prefix(prefix):
            next_prefix = prefix + 1
            total = 0
            # Keep expanding the current prefix by a factor of 10 (moving to the next level in the trie)
            while prefix <= n:
                # Calculate the number of elements between the current prefix and the next prefix
                total += min(n - prefix + 1, next_prefix - prefix)
                next_prefix *= 10
                prefix *= 10
            return total

        # Starting with the first prefix
        current = 1
        # Since we start from 1, we subtract 1 from k to match 0-based indexing
        k -= 1

        while k:
            count = count_prefix(current)
            # If the remaining k is larger than the count,
            # it means the target is not in the current subtree
            if k >= count:
                k -= count   # Skip the current subtree
                current += 1  # Move to the next sibling
            else:
                # The target is within the current subtree
                k -= 1      # Move to the next level in the trie
                current *= 10
        # The kth number has been found
        return current
```

Java Solution

```java
class Solution {
    // Class level variable to store the upper limit.
    private int upperLimit;

    // Method to find k-th smallest number in the range of 1 to n inclusive.
    public int findKthNumber(int n, int k) {
        // Assign the upper limit.
        this.upperLimit = n;
        // Start with the smallest number 1.
        long current = 1;
        // Decrement k as we start with number 1 already considered.
        k--;

        // Loop until we find the k-th number.
        while (k > 0) {
            // Determine the count of numbers prefixed with 'current'.
            int count = getCount(current);
            // If k is larger than or equal to the count, skip this prefix.
            if (k >= count) {
                k -= count; // Decrement k by the count of numbers.
                current++; // Move to the next number.
            } else {
                // If k is smaller than the count, go deeper into the next level.
                k--; // We're considering a number (current * 10) in next step, hence decrement k.
                current *= 10; // Move to the next level by multiplying by 10.
            }
        }
        // The current number is the k-th smallest number.
        return (int) current;
    }

    // Helper method to count the numbers prefixed with 'current' within 'n'.
    public int getCount(long current) {
        // Find the next sibling in the tree (e.g., from 1 to 2, 10 to 20 etc.).
        long next = current + 1;
        // Initialize the count of numbers.
        long count = 0;

        // Loop until 'current' exceeds 'upperLimit'.
        while (current <= upperLimit) {
            // Add the delta between n and current to the count.
            // We take the minimum of delta and (next - current) to handle cases where
            // 'n' is less than 'next - 1' (e.g., when 'n' is 12 and range is 10 to 20).
            count += Math.min(upperLimit - current + 1, next - current);

            // Move both 'current' and 'next' one level down the tree.
            next *= 10;
            current *= 10;
        }
        // Return the count after casting it to an integer.
        return (int) count;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int upperLimit; // This will store the upper limit upto which we need to find the numbers.

    // This method will find the kth smallest integer in the lexicographical order for numbers from 1 to n.
    int findKthNumber(int n, int k) {
        upperLimit = n; // Set the upper limit for the numbers
        --k; // Decrement k as we start from 0 to make calculations easier
        long long currentPrefix = 1; // We start with 1 as the smallest lexicographical number

        // Keep looking for the kth number until k reaches 0
        while (k) {
            int countOfNumbers = getCountOfNumbersWithPrefix(currentPrefix);

            if (k >= countOfNumbers) {
                // If k is larger than the count, skip the entire subtree
                k -= countOfNumbers;
                ++currentPrefix; // Move to the next prefix
            } else {
                // If k is within the range, go deeper into the subtree
                k--; // Decrement k as we have found another smaller number
                currentPrefix *= 10; // Append a 0 to dive into the next level of the tree
            }
        }
        return static_cast<int>(currentPrefix); // Cast and return the current number
    }

    // This helper method will count the numbers with a given prefix up to the upper limit.
    int getCountOfNumbersWithPrefix(long long prefix) {
        long long nextPrefix = prefix + 1; // Get the next prefix by adding one
        int count = 0; // Initialize the count for the numbers

        // Calculate the count of the numbers with the current prefix within the bounds of 'n'
        while (prefix <= upperLimit) {
            // Add the minimum of the range to the current prefix or the numbers until 'n'
            count += min(static_cast<long long>(upperLimit) - prefix + 1, nextPrefix - prefix);
            nextPrefix *= 10; // Move to the next level by appending a 0 to nextPrefix
            prefix *= 10; // Move to the next level by appending a 0 to prefix
        }
        return count; // Return the count of numbers with the current prefix
    }
};
```

Typescript Solution

```typescript
let upperLimit: number; // This stores the upper limit up to which numbers are found.

/**
 * Finds the kth smallest integer in lexicographical order for numbers from 1 to n.
 * @param {number} n - The upper limit of the number range.
 * @param {number} k - The position of the number to find.
 * @returns {number} - The kth smallest lexicographical number.
 */
function findKthNumber(n: number, k: number): number {
    upperLimit = n; // Set the upper limit for the range of numbers.
    k -= 1; // Decrement k as we start from 0 to simplify calculations.
    let currentPrefix: number = 1; // Start with 1 as the smallest lexicographical number.

    while (k > 0) {
        let countOfNumbers: number = getCountOfNumbersWithPrefix(currentPrefix);

        if (k >= countOfNumbers) {
            // The ith number is beyond the current subtree.
            k -= countOfNumbers;
            currentPrefix += 1; // Go to the next sibling prefix.
        } else {
            // The ith number is within the current subtree.
            k -= 1; // We step over a number in lexicographical order.
            currentPrefix *= 10; // Go down to the next level in the tree.
        }
    }

    return currentPrefix; // Return the found number.
}

/**
 * Counts the numbers with a specific prefix within the upper limit.
 * @param {number} prefix - The current prefix to count within.
 * @returns {number} - The count of numbers with the specified prefix.
 */
function getCountOfNumbersWithPrefix(prefix: number): number {
    let nextPrefix: number = prefix + 1; // Next prefix sequence.
    let count: number = 0; // Count of numbers with the given prefix.

    while (prefix <= upperLimit) {
        // Calculate minimum of the remaining numbers with current prefix vs whole next level.
        count += Math.min(upperLimit - prefix + 1, nextPrefix - prefix);
        nextPrefix *= 10; // Prepare the nextPrefix for the next level.
        prefix *= 10; // Prepare the prefix for the next level.
    }
    return count; // Return the count.
}
```

Time and Space Complexity

The provided code is designed to find the `k`-th smallest number in the lexicographical order of numbers from `1` to `n`.

Time Complexity

The time complexity of the function depends on two main factors: the number of iterations required to decrement `k` to `0`, and the time taken by the `count` function, which calculates the number of lexicographic steps from the current prefix to the next.

1. The `count` function has a while loop that runs as long as `curr <= n`. In each iteration, the `curr` is multiplied by `10`. This loop could theoretically run up to `O(log(n))` times because the maximum "distance" from `1` to `n` in terms of 10x increments is logarithmic with base `10` to `n`.
2. The outer while loop that decrements `k` runs until `k` becomes `0`. However, every time we update `curr` (either by `curr += 1` or `curr *= 10`), we make a relatively large jump in terms of lexicographical order, especially when we multiply by `10`. Despite `k` starting with a maximum of `n`, because of the exponential nature of the jumps, the total number of times this loop will iterate is also `O(log(n))` in practice.

The result is that the overall time complexity is `O(log(n)^2)`. Each decrement of `k` in the worst case can call `count`, and there can be up to `O(log(n))` such operations.

Space Complexity

The space complexity of the solution is `O(1)`. Outside of `count`'s recursive calls, no additional memory that scales with `n` or `k` is used, as we're only maintaining constant space variables like `curr`, `next`, `cnt`, and `k`.