Documentation on K-th Smallest in Lexicographical Order

Table of Contents

1. Problem Statement
2. Intuition
3. Key Observations
4. Approach
5. Edge Cases
6. Complexity Analysis
   - Time Complexity
   - Space Complexity
7. Alternative Approaches
8. Test Cases
9. Final Thoughts

1. Problem Statement

Given two integers n and k, return the k-th lexicographically smallest integer in the range [1, n].

Example 1

- Input: n = 13, k = 2
- Output: 10
- Explanation: The lexicographical order is [1, 10, 11, 12, 13, 2, 3, 4, 5, 6, 7, 8, 9], so the second smallest number is 10.

Example 2

- Input: n = 1, k = 1
- Output: 1

Constraints

- $1 <= k <= n <= 10^9$

2. Intuition

Instead of generating the lexicographical order explicitly (which is inefficient), we use a trie-like approach. The numbers can be structured in a 10-ary tree, where each node represents a prefix.

For example, numbers from 1 to 13 in lexicographical order form this structure:

```
   1
  / | \
 10 11 12 13
2 3 4 5 6 7 8 9
```

To find the k-th smallest number:

1. Count the numbers under a given prefix.
2. Navigate through the tree efficiently without generating all numbers.

3. Key Observations

i. Lexicographical order follows a DFS-like structure:
   a. If 1 is first, then 10, 11, ..., 19 follow before 2.

ii. Counting numbers with a given prefix efficiently:
   a. Instead of listing numbers, count them mathematically.
   b. count_nodes(prefix, n) function determines the count.

iii. Skipping prefixes:
   a. If k is larger than the count of numbers under curr, move to the next prefix.
   b. Otherwise, go deeper in the current prefix.

4. Approach

Step 1: Define a Counting Function

Define count_nodes(prefix, n), which counts numbers starting with a given prefix up to n:

- Start from prefix, iterate by multiplying by 10 to cover subtrees.
- Stop when exceeding n.

Step 2: Traverse the Lexicographical Order

i. Start at curr = 1 (smallest lexicographical number).
ii. While k > 1:
    a. Compute count_nodes(curr, n).
    b. If k falls within the subtree, move deeper (curr *= 10).
    c. Else, move to the next prefix (curr += 1) and subtract count from k.
iii. Return curr when k = 0.

5. Edge Cases

i. Minimum Input (n = 1, k = 1)
    a. Output is always 1.
ii. Exact Power of 10 (n = 1000, k = 500)
    a. Ensures correct traversal across different levels.
iii. Large n Values (n = 10^9)
    a. Tests efficiency, ensuring O(log n) complexity.
iv. Last Element (k = n)
    a. Ensures traversal correctly moves to the last number.

6. Complexity Analysis

Time Complexity

- count_nodes takes O(log n).
- We traverse O(log n) steps.
- Overall complexity: O(log n) (much better than sorting O(n log n)).

Space Complexity

- O(1), since no extra storage is needed apart from variables.

7. Alternative Approaches

1. Brute Force Sorting (O(n log n))

- Generate numbers, sort them lexicographically, and return k-th element.
- Inefficient for large n (10^9 elements is infeasible).

2. DFS-based Lexicographical Traversal (O(n))

- A full DFS traversal to find k.
- Still too slow for large n.

8. Test Cases

```
solution = Solution()

# Basic Cases
print(solution.findKthNumber(13, 2))  # Expected: 10
print(solution.findKthNumber(1, 1))   # Expected: 1

# Large Cases
print(solution.findKthNumber(100, 10))   # Expected: 17
print(solution.findKthNumber(1000, 100)) # Expected: 91
print(solution.findKthNumber(100000, 1000)) # Expected: 9001
```

9. Final Thoughts

- This approach optimally finds the k-th lexicographical number in O(log n) time.
- Trie-based counting helps skip large chunks of numbers, making it scalable.
- Useful for problems involving lexicographical ordering without full enumeration.