K-th Smallest in Lexicographical Order

Given integers `n` and `k`, find the lexicographically k-th smallest integer in the range from `1` to `n`.

Note: $1 \le k \le n \le 10^9$.

Example:

```
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 se
cond smallest number is 10.
```

Solution 1

This solution is inspired by @mgch's code.

General ideas:

1. Calculate the number of numbers that prefix with `result`.

Numbers prefixed by `result` are the union of the following intervals:

[`result` , `result` +1)

[`result` *10, (`result` +1)*10)

[`result` *100, (`result` +1)*100)

...

But they also shall belong to [1, n]

Therefore, we can easily calculate the number of elements prefixed by `result` using the following code (not considering overflow):

```
            int count = 0;
            for (int first = result, last = result + 1; // the first interval con
tains only one element: {result}
                first <= n; // the interval is not empty
                first *= 10, last *= 10) // increase a digit
            {
                // valid interval = [first, last) union [first, n]
                count += (min(n + 1, last) - first); // add the length of interva
l
            }
```

2. Search the next prefix, or search more detailedly.

- If the number of numbers that prefixed by `result` is smaller than the remaining `k`, we do not need to consider the numbers prefixed by `result` any more, and march to the next prefix: `result` +1;
- Otherwise, we need to search more detailedly, by appending another digit to the prefix. Such search shall start with `result` *10.

C++

```cpp
class Solution {
public:
    int findKthNumber(int n, int k)
    {
        int result = 1;
        for(--k; k > 0; )
        {
            // calculate #|{result, result*, result**, result***, ...}|
            int count = 0;
            for (long long first = static_cast<long long>(result), last = first +
1;
                first <= n; // the interval is not empty
                first *= 10, last *= 10) // increase a digit
            {
                // valid interval = [first, last) union [first, n]
                count += static_cast<int>((min(n + 1LL, last) - first)); // add t
he length of interval
            }

            if (k >= count)
            {   // skip {result, result*, result**, result***, ...}
                // increase the current prefix
                ++result;
                k -= count;
            }
            else
            {   // not able to skip all of {result, result*, result**, result***,
...}
                // search more detailedly
                result *= 10;
                --k;
            }
        }
        return result;
    }
};
```

Python

```
class Solution(object):
    def findKthNumber(self, n, k):
        result = 1;
        k -= 1
        while k > 0:
            count = 0
            interval = [result, result+1]
            while interval[0] <= n:
                count += (min(n+1, interval[1]) - interval[0])
                interval = [10*interval[0], 10*interval[1]]

            if k >= count:
                result += 1
                k -= count
            else:
                result *= 10
                k -= 1
        return result
```

Complexities:

- **Time Complexity:** $O((\log n) \hat{} 2)$
  Here, $\log n$ is the number of digits in n, and it is also the number of replications of appending zero to search detailedly. Each such appending introduces the increasement of prefix at most 10 times, and each increasement may require $O(\log n)$ time to calculate the number of numbers that prefixed by the `result`.
- **Space Complexity:** $O(1)$

written by zhiqing_xiao original link here

Solution 2

Original idea comes from
http://bookshadow.com/weblog/2016/10/24/leetcode-k-th-smallest-in-lexicographical-order/

Actually this is a denary tree (each node has 10 children). Find the kth element is to do a k steps preorder traverse of the tree.

Initially, image you are at node 1 (variable: curr),
the goal is move (k - 1) steps to the target node x. (substract steps from k after moving)
when k is down to 0, curr will be finally at node x, there you get the result.

we don't really need to do a exact k steps preorder traverse of the denary tree,**the idea is to calculate the steps between curr and curr + 1 (neighbor nodes in same level), in order to skip some unnecessary moves.**

Main function
Firstly, calculate how many steps curr need to move to curr + 1.

1. if the steps <= k, we know we can move to curr + 1, and narrow down k to k - steps.
2. else if the steps > k, that means the curr + 1 is actually behind the target node x in the preorder path, we can't jump to curr + 1. What we have to do is to move forward only 1 step (curr * 10 is always next preorder node) and repeat the iteration.

calSteps function

1. how to calculate the steps between curr and curr + 1?
   Here we come up a idea to calculate by level.
   Let n1 = curr, n2 = curr + 1.
   n2 is always the next right node beside n1's right most node (who shares the same ancestor "curr")
   (refer to the pic, 2 is right next to 1, 20 is right next to 19, 200 is right next to 199).
2. so, if n2 <= n, what means n1's right most node exists, we can simply add the number of nodes from n1 to n2 to steps.
3. else if n2 > n, what means n (the biggest node) is on the path between n1 to n2, add (n + 1 - n1) to steps.
4. organize this flow to "steps += Math.min(n + 1, n2) - n1; n1 *= 10; n2 *= 10;"

Here is the code snippet:

```java
public int findKthNumber(int n, int k) {
    int curr = 1;
    k = k - 1;
    while (k > 0) {
        int steps = calSteps(n, curr, curr + 1);
        if (steps <= k) {
            curr += 1;
            k -= steps;
        } else {
            curr *= 10;
            k -= 1;
        }
    }
    return curr;
}
//use long in case of overflow
public int calSteps(int n, long n1, long n2) {
    int steps = 0;
    while (n1 <= n) {
        steps += Math.min(n + 1, n2) - n1;
        n1 *= 10;
        n2 *= 10;
    }
    return steps;
}
```

written by NathanNi original link here

Solution 3

There is a good explanation at there.

The solution used DFS to search in a trie tree. The trick is to skip the sub-tree if the (current index + node number of the sub-tree) is smaller than the k.

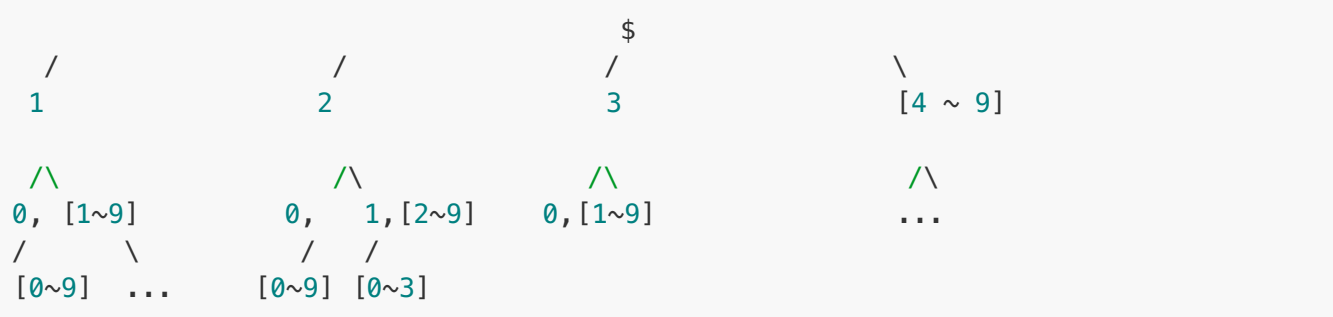
The key problem is - How to count the nodes in a sub-tree?

The trie tree was made up of two kinds of sub-tree, the complete sub-tree with each node either has ten or zero children and the incomplete sub-tree with some inner nodes can has 1 to 9 children.

The complete tree's nodes number is easy to get, it will be something like 1, 11, 111, 1111...(each node has ten children plus the root)

The incomplete tree can be calculated like the flowing example:

for n = 213, the tree will be something like this:



the sub-trees start with char '1' and '3'~'9' are complete trees, which '1' sub-tree with ranges in 1,10~19, 100~199 has 111 nodes and '3'~'9' each has 11 nodes (e.g. 3 with ranges in 3, 30~39).

the sub-tree starts with '2' is not full, i.e. the ranges are 2, 20~29, 200~213. But the '2' sub-tree still has a 'complete' tree like '3'~'9' with range in 2, 20~29, and we just need to add the remained leaf nodes which is (213%200)+1 or (213-200)+1.

The 200 is the most left node in the '2' sub-tree and 299 is the most right node in '2' sub-tree if it is complete.
it also showed the idea about how to judge a given sub-three is complete or incomplete:

We first **assume all sub-trees are complete** and test by following:

```
{
if (the most right node in a sub-tree is not greater than the number n)

    the sub-tree is complete with nodes number  = 111..(log10(n)+1 1s)

else if (the most left node in a sub-tree is greater than the number n)

    the sub-tree is complete with nodes number  = 11...  (log10(n) 1s)

else

    the sub-tree is incomplete with nodes number =
n - (the most left node in the subree) + 1 + 11... (log10(n) 1s)
}
```

The nodes number in a complete tree can be cached.

Then we just need to recursively decrease k by the nodes number of the skipped sub-tree until k reached 1.

Also be carefully the first layer is start from 1 while other layers are start from 0.

```
public class Solution {
    int countNum(int n){
        int i=0;
        while(n>0){
            n/=10;
            i++;
        }
        return i;
    }
    int getFullTreeNum(int depth){
        int sum=0, children=1;
        while(depth>0){
            sum+=children;
            children*=10;
            depth--;
        }
        return sum;
    }
    int getMax(int prefix, int depth){
        while(depth>0){
            prefix*=10;
            prefix+=9;
            depth--;
        }
        return prefix;
    }
    int getMin(int prefix, int depth){
        while(depth>0){
            prefix*=10;
            depth--;
        }
        return prefix;
    }
```

```
    int helper(int n, int k, int prefix, int depth){
        int lowNum=getFullTreeNum(depth), highNum=getFullTreeNum(depth-1);
        for(int i=(prefix==0?1:0);i<=9;i++){
            int nodeNum=0;
            if(getMax(prefix*10+i, depth-1)<=n){
                nodeNum=lowNum;
            }
            else if(getMin(prefix*10+i, depth-1)>n){
                nodeNum=highNum;
            }
            else{
                nodeNum=highNum+((n-getMin(prefix*10+i, depth-1))+1);
            }
            k-=nodeNum;
            if(k<=0){
                k+=nodeNum;
                if(k==1){
                    return prefix*10+i;
                }
                else {
                    return helper(n, k-1, prefix*10+i, depth-1);
                }
            }
        }
        return 0;
    }

    public int findKthNumber(int n, int k) {
        int depth=countNum(n);
        int index=0;
        return helper(n, k, 0, depth);
    }
}
```

written by pureklkl original link here

From Leetcoder.