402. Remove K Digits

Medium Stack Greedy String Monotonic Stack

Problem Description

This LeetCode problem asks you to find the smallest possible integer after removing exactly `k` digits from a string `num` that represents a non-negative integer. The goal is to reduce the size of the number while keeping the remaining digits in the same order as they were in the original number.

Intuition

The intuition behind the solution is to use a greedy algorithm. If we want the resulting number to be the smallest possible, we should ensure that the higher place values (like tens, hundreds etc.) have the smallest possible digits. Therefore, while parsing the string from left to right, if we encounter a digit that is larger than the digit following it, we remove the larger digit (which is at a higher place value). This decision is greedy because it makes the best choice at each step, aiming to keep the smallest digits at higher place values.

To efficiently perform removals and keep track of the digits, a stack is an excellent choice. Each time we add a new digit to the stack, we compare it to the element on top of the stack (which represents the previous digit in the number). If the new digit is smaller, it means we can make the number smaller by popping the larger digit off the stack. This process is repeated up to `k` times as required by the problem statement.

The stack represents the digits of the resulting number with the smallest digits at the bottom (higher place values). When `k` removals are done, or the string is fully parsed, we take the bottom `n - k` digits from the stack (where `n` is the length of `num`), since `k` digits have been removed, and that forms our result. Leading zeroes are removed as they do not affect the value of the number. If all digits are removed, we must return '0', which is the smallest non-negative integer.

Solution Approach

The implementation of this algorithm is straightforward once you understand the intuition:

1. We create an empty list called `stk`, which we will use as a stack to keep track of the valid digits of the smallest number we are constructing.
2. We need to retain `len(num) - k` digits to form the new number after we have removed `k` digits. The variable `remain` holds this value.
3. We iterate over each character `c` in the string `num`:
    - While we still have more digits `k` to remove, and the stack `stk` is not empty, and the digit at the top of the stack `stk[-1]` is greater than the current digit `c`, we pop the top of the stack. This is because keeping `c`, which is smaller, will yield a smaller number.
    - We also decrement `k` by 1 each time we pop a digit off the stack since that counts as one removal.
    - After the check (and potential removal), we append the current digit `c` to the stack. This digit is now part of the new number.
4. After we finish iterating over `num`, the stack contains the digits of the resulting number, but it might have more digits than necessary if we didn't need to remove `k` digits. Thus, we slice the stack up to `remain` digits.
5. Next, we need to convert the list of digits into a string. We join the digits in `stk` up to the `remain` index and then we remove any leading zeros with `.lstrip('0')`.
6. The last step is to handle the case where all digits are removed, resulting in an empty string. If that happens, we return `'0'` because we must return a valid number and `0` is the smallest non-negative integer. In any other case, we return the joined string of digits that now represents the smallest possible integer after the removal of `k` digits.

This algorithm makes use of a stack, which is a classic data structure that operates on a Last In, First Out (LIFO) principle. It's an ideal choice to store the digits of the new number because it allows for easy removal of the last added digit when a smaller digit comes next. The process is greedy and makes local optimum choices by preferring smaller digits in the higher place values.

Remember, in Python, a list can act as a stack with the `append` method to push elements onto the stack and the `pop` method to remove the top element of the stack.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Suppose the input string `num` is "1432219" and `k` is 3. We want to remove 3 digits to make the number as small as possible.

Here's the step-by-step process:

1. Initialize an empty list `stk` to represent the stack. The number of digits we want to remain in the final number is `remain = len(num) - k = 7 - 3 = 4`.
2. Iterate over each digit in "1432219":
    - Start with the first digit '1'. Since the stack is empty, we add '1' to `stk`.
    - Next is '4'. '4' is greater than '1', so we keep it and push '4' to `stk`.
    - Then comes '3'. '3' is smaller than '4' and `k > 0`, so we pop '4' out of the stack. Now `stk = ['1']` and `k = 2`.
    - Now we have '2'. '2' is smaller than '3', so we pop '3'. Now, `stk = ['1']` and `k = 1`.
    - Add '2' to the stack. `stk = ['1', '2']`.
    - Another '2' comes, which is the same as the last digit, so we push '2' to `stk`. `stk = ['1', '2', '2']`.
    - Finally, '1' is smaller than '2', so we pop the last '2' from `stk`. `stk = ['1', '2']`, and `k = 0` (no more removals allowed).
    - Since we've already removed 3 digits, just push '1' and then '9' to the stack. Now, `stk = ['1', '2', '1', '9']`.
3. We've finished processing each digit and our stack `stk` represents the smallest number we could make. However, we need to make sure we have the right number of digits, which should be `remain = 4`. Since `stk` already contains 4 digits, there's no need to slice.
4. Join the stack to form a number and strip leading zeros (if any). `result = ''.join(stk).lstrip('0')`. In this case, '1219'.
5. We return '1219', which is the smallest number possible after removing 3 digits from "1432219".

This example illustrates how the stack helps efficiently manage the digits of the new number, ensuring that smaller digits remain at the higher place values whenever possible.

Python Solution

```python
class Solution:
    def removeKdigits(self, num: str, k: int) -> str:
        # Initialize a stack to keep track of the digits
        stack = []

        # Number of digits to remain in the final number
        remaining_digits_count = len(num) - k

        # Iterate over each character in the input string
        for digit in num:
            # While we can still remove digits, and the stack is not empty,
            # and the current digit is smaller than the last digit in the stack:
            while k and stack and stack[-1] > digit:
                # Remove the last digit from the stack as it's greater than the current one
                stack.pop()
                # Decrease the count of digits we can remove
                k -= 1
            # Add the current digit to the stack
            stack.append(digit)

        # Build the final number string from the stack up to the remaining digits
        final_number = ''.join(stack[:remaining_digits_count])

        # Strip leading zeros from the final number and return it, or return '0' if empty
        return final_number.lstrip('0') or "0"
```

Java Solution

```java
class Solution {
    public String removeKdigits(String num, int k) {
        // Create a StringBuilder to use as a stack to keep track of digits.
        StringBuilder stack = new StringBuilder();

        // Iterate through each character in the input string.
        for (char digit : num.toCharArray()) {
            // While the current digit is smaller than the last digit in the stack
            // and we still have digits to remove (k > 0), remove the last digit.
            while (k > 0 && stack.length() > 0 && stack.charAt(stack.length() - 1) > digit) {
                stack.deleteCharAt(stack.length() - 1);
                k--;
            }
            // Append the current digit to the stack (StringBuilder).
            stack.append(digit);
        }

        // If after the iteration we still need to remove more digits, remove from the end.
        while (k > 0) {
            stack.deleteCharAt(stack.length() - 1);
            k--;
        }

        // Remove leading zeros by finding the index of the first non-zero digit.
        int nonZeroIndex = 0;
        while (nonZeroIndex < stack.length() && stack.charAt(nonZeroIndex) == '0') {
            nonZeroIndex++;
        }
        // Create a new string starting from the first non-zero digit.
        String result = stack.substring(nonZeroIndex);

        // If the resulting string is empty, return "0" instead; otherwise, return the string.
        return result.isEmpty() ? "0" : result;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to remove 'k' digits from the string 'num' to get the smallest possible number.
    string removeKdigits(string num, int k) {
        string stack; // Using 'stack' to store the characters representing the smallest number

        // Iterate through each character in the input number
        for (char& digit : num) {
            // Check if the current digit is smaller than the last digit in 'stack'
            // and whether we have still digits to remove
            while (k > 0 && !stack.empty() && stack.back() > digit) {
                stack.pop_back(); // Remove the last digit from 'stack' to maintain the smallest number
                --k; // Decrement the count of digits to remove
            }
            stack += digit; // Add the current digit to 'stack'
        }

        // Further remove digits from the end if we haven't removed enough 'k' digits
        // This is necessary when the sequence was initially increasing
        while (k > 0) {
            stack.pop_back(); // Remove the last digit from 'stack'
            --k; // Decrement the count of digits to remove
        }

        // Remove leading zeros from the 'stack'
        int startIndex = 0; // Index to keep track of leading zeros
        while (startIndex < stack.size() && stack[startIndex] == '0') {
            ++startIndex; // Increment index to skip the leading zero
        }

        string result = stack.substr(startIndex); // Extract the non-zero starting substring as result
        return result.empty() ? "0" : result; // If result is empty, return "0"; otherwise, return the result
    }
};
```

Typescript Solution

```typescript
function removeKdigits(numString: string, k: number): string {
    // Convert the string to an array of characters for easier manipulation
    let digitArray = [...numString];

    // Keep removing digits until we have removed k digits
    while (k > 0) {
        let indexToDelete = 0; // Initialize deletion index

        // Find where the digit is greater than the one following it; that's our deletion target
        while (indexToDelete < digitArray.length - 1 && digitArray[indexToDelete + 1] >= digitArray[indexToDelete]) {
            indexToDelete++;
        }

        // Remove the digit at the identified deletion index
        digitArray.splice(indexToDelete, 1);

        // Decrement the count of digits we still need to remove
        k--;
    }

    // Join the array back into a string and strip leading zeroes, if any
    let result = digitArray.join('').replace(/^0+/g, '');

    // If the result is an empty string, return '0', otherwise return the processed number string
    return result || '0';
}
```

Time and Space Complexity

Time Complexity

The time complexity of the given code can be analyzed based on the operations performed. The code iterates over each character in the string `num` which has a length of `n`. In the worst case, each character may be pushed to and popped from the stack `stk` once. Pushing and popping from the stack are `O(1)` operations, but since the inner while loop could run up to `k` times for each character, it might appear at first as if the complexity is `O(nk)`. However, each element is pushed and popped at most once, resulting in a time complexity of `O(n)` overall because the while loop can't execute more than `n` times over the course of the entire function.

Therefore, the total time complexity of the algorithm is:

```
O(n)
```

Space Complexity

The space complexity is determined by the space used by the stack `stk`, which in the worst case could contain all characters if `k` is zero or if all characters are in increasing order. Therefore, the space complexity is proportional to the length of the input string `num`.

Thus, the space complexity of the algorithm is:

```
O(n)
```