1881. Maximum Value after Insertion

Medium Greedy String

Problem Description

The problem presents us with two input values: a large integer `n` represented as a string and an integer digit `x` which range from 1 to 9. The goal is to insert the digit `x` into the string representation of `n` in such a way that the resulting number is maximized. If `n` is negative, `x` can be inserted anywhere except before the minus sign.

Here are some key points to keep in mind:

- We can insert `x` in between any two digits of `n`.
- The challenge is to figure out the most optimal position for `x` to achieve the largest possible value.
- The comparison of where to insert `x` differs based on whether `n` is negative or positive.

Intuition

The intuition behind the solution stems from understanding how numerical value is affected by the placement of digits. For positive numbers, placing a larger digit towards the left increases the number's value. Thus, for a positive `n`, we look for the first instance where `x` is greater than a digit in `n`, and insert `x` before this digit to maximize the value.

Conversely, for negative numbers, we want to minimize the value of the negative magnitude to maximize `n`'s value. Therefore, we look for the first digit in `n` that is smaller than `x`, and insert `x` after this digit to ensure the negative number is as small as possible (which makes `n` as large as possible in the negative domain).

Here's the step-by-step breakdown of our approach:

1. If `n` is positive, iterate through its digits.
    - Find the position where the current digit is smaller than `x`.
    - Insert `x` before this digit and return the modified string.
2. If `n` is negative, skip the minus sign and iterate through the remaining digits.
    - Find the position where the current digit is larger than `x`.
    - Insert `x` after this digit, accounting for the skipped minus sign, and return the modified string.
3. If no such position is found in the above iterations (all digits of `n` are either larger than `x` for positive `n`, or smaller/equal to `x` for negative `n`), insert `x` at the end of the string.

By following this rule, the solution ensures that `n`'s value is maximized according to the problem's constraints.

Solution Approach

The implementation of the solution follows the intuition previously explained. The algorithm is straightforward and can be summarized into the following steps:

1. **Check the Sign of `n`:**
    - Determine if the number `n` is negative or positive by checking the first character of the string representation of `n`.
2. **Positive Case (`n` is not negative):**
    - Loop through each digit in `n` using a `for` loop.
    - In every iteration, compare the current digit with the given digit `x`.
    - If the current digit is found to be less than `x`, use string slicing to insert `x` before this digit.
    - Return the modified string immediately.
3. **Negative Case (`n` is negative):**
    - Similar to the positive case, use a `for` loop but start from the second character (skipping the negative sign).
    - Compare each digit with `x`.
    - If the current digit is greater than `x`, insert `x` before this digit.
    - Use string slicing while keeping in mind the negative sign's position (which is at index 0).
    - Return the modified string immediately.
4. **Appending to the End:**
    - If the loop completes without finding a suitable position for insertion (which means `x` is less or equal to all digits in a positive `n`, or `x` is less than or equal to all digits in a negative `n` after the sign), append `x` at the end of `n`.

The algorithm does not use any additional data structures, and it relies solely on the properties of string slicing and built-in comparison operators in Python. The solution pattern is iterative and can be classified as a simple linear search, which finds the insertion point by comparing each digit of `n` with `x`.

The Python code handles the insertion and string manipulation tasks using concatenation, which adds the string representation of `x` to the appropriate slice of `n`. The solution effectively applies the basic concepts of string manipulation with time complexity `O(n)` where `n` represents the number of digits in the input string `n`.

Example Walkthrough

Let's illustrate the solution approach with a small example where we have the integer `n` represented as the string `"273"` and the integer digit `x` as `4`. The goal is to insert `4` into `"273"` such that the resulting number is as large as possible.

Following the solution steps:

1. **Check the Sign of `n`:**
    - `"273"` does not start with a minus sign; therefore, the number is positive.
2. **Positive Case (`n` is not negative):**
    - We start iterating through each digit in `"273"`.
    - Compare first digit `2` with `x` (`4`).
    - `2` is less than `4`, so this is where `4` should be inserted to maximize the number.
    - Use string slicing to insert `4` before `2` to get `"4273"`.
    - Return modified string `"4273"` as the answer.

The resulting number is `"4273"`, which is the largest number that can be formed by inserting `4` into `"273"`.

Now, let's consider a negative example with `n` as `"-456"` and `x` as `3`.

1. **Check the Sign of `n`:**
    - `"-456"` starts with a minus sign; therefore, the number is negative.
2. **Negative Case (`n` is negative):**
    - Skip the minus sign and start the iteration from the second character.
    - Compare each digit with `x` (`3`).
    - `4` is greater than `3`, so we continue to the next digit.
    - `5` is also greater than `3`, so we continue.
    - `6` is greater than `3`, so we could keep going, but since there are no more digits, `3` will be inserted at the end.
    - Use string slicing while keeping in mind the negative sign's position.
    - The final string will be `"-4536"`.

In the negative case, the largest number we can form by adding `3` to `"-456"` is `"-4536"`. Here `3` is inserted at the end because each digit in `"-456"` after the minus sign is larger than `3`, making `"-4536"` the best possible outcome.

Solution Implementation

Python

```python
class Solution:
    def maxValue(self, n: str, x: int) -> str:
        # Check if the number n is positive
        if n[0] != "-":
            # Loop through each character in the number
            for i, char in enumerate(n):
                # If the current digit is less than x, insert x before it
                if int(char) < x:
                    return n[:i] + str(x) + n[i:]
            # If not inserted, add x to the end of the number
            return n + str(x)
        else:
            # The number n is negative, skip the '-' sign and start from the next digit
            for i, char in enumerate(n[1:]):
                # If the current digit is greater than x, insert x before it
                if int(char) > x:
                    return n[:i + 1] + str(x) + n[i + 1:]
            # If x has not been inserted yet, add it to the end of the number
            return n + str(x)
```

Java

```java
class Solution {
    public String maxValue(String n, int x) {
        // Initialize the index variable i to 0
        int i = 0;

        // If the first character is not a '-'
        if (n.charAt(0) != '-') {
            // Loop through the string until we find a digit less than x
            for (; i < n.length(); ++i)
                // No body for this for loop as it's just used to find the breakpoint
                ;
        } else {
            // If the first character is '-', start with the second character
            // Loop through the string until we find a digit greater than x
            for (i = 1; i < n.length() && n.charAt(i) - '0' <= x; ++i) {
                // No body for this for loop as it's just used to find the breakpoint
            }
        }

        // Concatenate the string parts and the integer x:
        // 1. Substring from the start to i (the breakpoint)
        // 2. The integer x, converted to a string
        // 3. The remaining substring from i to the end of the string
        return n.substring(0, i) + x + n.substring(i);
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to insert the digit 'x' into the string 'n' to achieve the highest possible value.
    string maxValue(string n, int x) {
        int position = 0; // Initialize the insertion position

        // If the number is positive
        if (n[0] != '-') {
            // Iterate over the string until we find a digit less than 'x'
            for (; position < n.size() && (n[position] - '0') >= x; ++position)
                ; // The loop body is empty since all work is done in the condition
        } else { // If the number is negative
            // Start from position 1 to skip the minus sign
            for (position = 1; position < n.size() && (n[position] - '0') <= x; ++position)
                ; // Again, the loop body is empty
        }

        // Insert 'x' into the found position and construct the new string
        return n.substr(0, position) + to_string(x) + n.substr(position);
    }
};
```

TypeScript

```typescript
/**
 * Function to insert the maximum value.
 * Inserts an integer digit `x` into the string representation of a non-negative integer `n`,
 * at such a position that the new integer is as large as possible.
 *
 * @param {string} n - The string representation of the number into which `x` has to be inserted.
 * @param {number} x - The integer digit to insert into `n`.
 * @return {string} - The resulting string after insertion of `x`.
 */
function maxValue(n: string, x: number): string {
    // Convert the string `n` into an array of characters
    let numberArray: string[] = [...n];

    // Determine the sign of the number (positive by default)
    let sign: number = 1;

    // Starting index for the iteration, adjusted if the number is negative
    let index: number = 0;

    // If the first character is a minus sign, update the `sign` and `index`
    if (numberArray[0] === '-') {
        sign = -1;
        index++;
    }

    // Find the position to insert `x` by iterating over the characters of `n`
    // For a positive number, it stops before the first digit smaller than `x`
    // For a negative number, it stops before the first digit larger than `x`
    while (index < n.length && (parseInt(numberArray[index]) - x) * sign >= 0) {
        index++;
    }

    // Insert `x` into the correct position in the array of characters
    numberArray.splice(index, 0, x.toString());

    // Join the array back into a string and return the result
    return numberArray.join('');
}

// The maxValue function can now be called with TypeScript syntax
// Example usage: let result: string = maxValue('123', 5);
```

```python
class Solution:
    def maxValue(self, n: str, x: int) -> str:
        # Check if the number n is positive
        if n[0] != "-":
            # Loop through each character in the number
            for i, char in enumerate(n):
                # If the current digit is less than x, insert x before it
                if int(char) < x:
                    return n[:i] + str(x) + n[i:]
            # If not inserted, add x to the end of the number
            return n + str(x)
        else:
            # The number n is negative, skip the '-' sign and start from the next digit
            for i, char in enumerate(n[1:]):
                # If the current digit is greater than x, insert x before it
                if int(char) > x:
                    return n[:i + 1] + str(x) + n[i + 1:]
            # If x has not been inserted yet, add it to the end of the number
            return n + str(x)
```

Time and Space Complexity

The time complexity of the given code is `O(n)` where `n` is the length of the input string. The for-loop iterates over the string characters at most once. In each iteration, it performs constant time operations (comparisons and integer conversions). Therefore, the total time taken is linear with respect to the length of the input string.

The space complexity of the code is `O(1)` (disregarding the input and the output). The reason is that the code only uses a fixed number of variables (`i`, `c`, `x`), and creating the final string output doesn't count towards additional space since the output is required and does not contribute to the space used by the algorithm itself.