738. Monotone Increasing Digits

Medium Greedy Math

Problem Description

The problem statement presents a challenge where you need to find the largest number that is less than or equal to a given integer `n` and also has monotone increasing digits. A number is said to have monotone increasing digits if every digit is less than or equal to the digit immediately to its right. The objective is to transform `n` in such a way that it satisfies this condition but still remains the largest possible number under the given constraints.

Intuition

To solve this problem, the intuition is to process the number from the left to right because the leftmost digits have a higher value. Therefore, adjusting them will have a greater impact on the overall value of the number. As we iterate through the digits, we're looking for the point where the monotone condition is violated (a digit that is greater than the next digit). When this violation is found, we must adjust the number to restore the monotone condition.

The key insight is that once we find a pair of digits where the left digit is greater than the right digit, we can reduce the left digit by one and change all subsequent digits to 9 to get the largest possible monotone increasing number. However, this reduction can cause a violation of the monotone condition to the left, so we must check and fix those cases too.

The solution approach involves turning the integer into a string to work with individual digits more conveniently, then walking through the string to find the first descending point. We decrement the digit at that point and then check to the left to ensure we still have a monotone sequence, adjusting as needed. Lastly, we set all digits to the right of the adjusted digit to 9, since that will give us the largest possible number that maintains our condition.

Solution Approach

The implementation starts by converting the integer `n` to a string. This allows us to access each digit character easily to compare and modify them.

The algorithm proceeds in the following steps:

1. **Initiate Search for Non-Monotone Point:** We initialize `i` to 1 and start iterating over the digits from left to right, comparing each digit with its immediate predecessor to ensure that each digit is greater than or equal to the previous one. The goal is to identify the index where the monotone increasing property fails.
2. **Detect Decreasing Sequence:** As we proceed, if we find a digit that is smaller than the digit before it (`s[i] < s[i - 1]`), this is where the property of monotone increasing digits is violated. This is the point where we need to take action to correct the sequence.
3. **Reduce and Correct the Sequence:** To fix the violation, we move one position to the left (start decrementing i) and reduce that digit by 1 (`s[i - 1] = str(int(s[i - 1]) - 1)`). We continue correcting the sequence by moving leftward until we reach a point where the sequence is increasing again.
4. **Maximize Remaining Digits:** After correcting the sequence, we replace all digits to the right of our correction point with '9'. This is because '9's will maximize the resulting number while preserving the monotone property from that point forward.
5. **Return the Result:** Finally, we convert the list of characters back into a string, and then into an integer, which is the largest number less than or equal to `n` with monotone increasing digits.

Algorithmic Concepts:

- **Greedy Approach:** The idea of changing only the first violating digit and replacing subsequent digits with '9's is greedy because it makes the most significant digits as large as possible without violating the monotone condition.
- **String Manipulation:** By treating the number as a string, we are able to manipulate individual digits more directly compared to doing so with arithmetic operations.
- **Simple Iteration:** The solution performs linear scans from left to right, and in some cases from right to left, showcasing a good use of simple iterative logic to solve the problem.

Data Structures Used:

- **List of Characters:** A list is used to represent the digits of `n`, because string in Python is immutable, and a list allows us to make changes to individual digits conveniently.

The solution exploits the property that a number is always greater when its leftmost digits are greater, hence the left-to-right and then right-to-left scans to achieve a correct and efficient approach to finding the answer.

Example Walkthrough

Let's illustrate the solution approach with a small example. Consider the integer `n = 3214`.

Following the steps of the solution:

1. **Initiate Search for Non-Monotone Point:** We will convert the number into a string for easy manipulation, `s = "3214"`, and start looking for the point where the digits stop being monotone increasing. We initialize `i` to 1 and start comparing adjacent digits.
2. **Detect Decreasing Sequence:** While comparing, we notice that `s[1] > s[2]` (which is `2 > 1`), violating the monotone increasing condition at index `i = 2`.
3. **Reduce and Correct the Sequence:** Now, we need to decrement `s[i - 1]` from `2` to `1`. But this alone makes the string "3114", which still violates the condition at `i = 1` (since `3 > 1`). So, we need to also decrement `s[i - 2]` from `3` to `2`, giving us "2114".
4. **Maximize Remaining Digits:** After the correction, we know that `s[i]` to `s[len(s) - 1]` can all be set to '9' to maximize the number without violating the monotone condition. So, '2114' becomes '2999'.
5. **Return the Result:** We convert the string "2999" back into an integer, which gives us `2999`. This is the largest number less than or equal to `3214` that has monotone increasing digits.

With the original number `3214`, the solution finds that at the index `2`, the monotone condition is violated. It then modifies the digits in such a way to maintain the maximum values while correcting the sequence, resulting in `2999`. This effectively demonstrates the greedy approach of the solution, which optimizes the leftmost digits first to get the largest possible number with the constraint of monotone increasing digits.

Solution Implementation

Python

```python
class Solution:
    def monotoneIncreasingDigits(self, n: int) -> int:
        # Convert the number to a list of characters for easy manipulation
        digits = list(str(n))

        # Find the first digit that breaks the monotony
        index = 1
        while index < len(digits) and digits[index - 1] <= digits[index]:
            index += 1

        # If a non-monotone digit was found, modify the number
        if index < len(digits):
            # Decrease the previous digit by 1 until the number becomes monotone
            while index > 0 and digits[index - 1] > digits[index]:
                digits[index - 1] = str(int(digits[index - 1]) - 1)
                index -= 1

            # Setting the rest of the digits to '9' to guarantee the largest monotone number
            index += 1
            while index < len(digits):
                digits[index] = '9'
                index += 1

        # Convert the list of characters back to an integer and return
        return int(''.join(digits))
```

Java

```java
class Solution {
    public int monotoneIncreasingDigits(int n) {
        // Convert the input integer n to a character array to process each digit individually.
        char[] digits = String.valueOf(n).toCharArray();

        // Initialize index i for iterating through the digits.
        int i = 1;

        // Find the point where digits are no longer increasing (i.e., digits[i-1] > digits[i]).
        for (; i < digits.length && digits[i - 1] <= digits[i]; i++) {
            // This loop will keep iterating until it finds a decreasing point or reaches the end.
        }

        // If the monotone increasing property is violated,
        // work backwards to find the digit which can be decremented.
        if (i < digits.length) {
            // Decrement the violating digit and find the position from which to replace with 9s.
            for (; i > 0 && digits[i - 1] > digits[i]; i--) {
                // Reduce the previous digit by 1 to maintain the monotone increasing property.
                digits[i - 1]--;
            }

            // Reset remaining digits to '9' beginning from the current index to the end of the array.
            for (; i < digits.length; i++) {
                digits[i] = '9';
            }
        }

        // Convert the updated character array back to an integer and return the result.
        return Integer.parseInt(new String(digits));
    }
}
```

C++

```cpp
class Solution {
public:
    // This function finds the largest monotone increasing number that is less than or equal to n.
    int monotoneIncreasingDigits(int n) {
        // Convert the integer n to a string representation for digit manipulation
        string numStr = to_string(n);
        int marker = 1;

        // Traverse the digits of the number until we find a digit that is less than the previous digit,
        // indicating that the sequence is no longer monotone increasing
        while (marker < numStr.size() && numStr[marker - 1] <= numStr[marker]) {
            ++marker;
        }

        // If we found a position where the monotone property is violated
        if (marker < numStr.size()) {
            // Go back and find the position where we need to decrement to restore the monotone property
            while (marker > 0 && numStr[marker - 1] > numStr[marker]) {
                numStr[marker - 1]--;
                --marker;
            }
            // Fill the rest of the digits after marker with '9' to maximize the resultant number
            for (int i = marker + 1; i < numStr.size(); ++i) {
                numStr[i] = '9';
            }
        }

        // Convert the modified string back to an integer and return it
        return stoi(numStr);
    }
};
```

TypeScript

```typescript
// This function finds the largest monotone increasing number that is less than or equal to n.
function monotoneIncreasingDigits(n: number): number {
    // Convert the integer n to a string representation for digit manipulation
    let numStr: string = n.toString();

    // Initialize marker as a pointer to traverse the digits
    let marker: number = 1;

    // Traverse the digits of the number to find where the monotone increasing sequence is broken
    while (marker < numStr.length && numStr[marker - 1] <= numStr[marker]) {
        marker++;
    }

    // Check if we found a position where the sequence is no longer monotone increasing
    if (marker < numStr.length) {
        // Step back to find the correct digit to decrement in order to keep the number monotone increasing
        while (marker > 0 && numStr[marker - 1] > numStr[marker]) {
            // Decrement the digit at marker - 1
            let prevDigit = parseInt(numStr[marker - 1], 10) - 1;
            numStr = numStr.substring(0, marker - 1) + prevDigit.toString() + numStr.substring(marker);

            // Move the marker back to the left as needed
            marker--;
        }

        // Fill the rest of the string with '9' after the marker to maximize the resulting number
        for (let i = marker + 1; i < numStr.length; i++) {
            numStr = numStr.substring(0, i) + '9' + numStr.substring(i + 1);
        }
    }

    // Convert the modified string back to an integer and return it
    return parseInt(numStr, 10);
}
```

```python
class Solution:
    def monotoneIncreasingDigits(self, n: int) -> int:
        # Convert the number to a list of characters for easy manipulation
        digits = list(str(n))

        # Find the first digit that breaks the monotony
        index = 1
        while index < len(digits) and digits[index - 1] <= digits[index]:
            index += 1

        # If a non-monotone digit was found, modify the number
        if index < len(digits):
            # Decrease the previous digit by 1 until the number becomes monotone
            while index > 0 and digits[index - 1] > digits[index]:
                digits[index - 1] = str(int(digits[index - 1]) - 1)
                index -= 1

            # Setting the rest of the digits to '9' to guarantee the largest monotone number
            index += 1
            while index < len(digits):
                digits[index] = '9'
                index += 1

        # Convert the list of characters back to an integer and return
        return int(''.join(digits))
```

Time and Space Complexity

The provided code aims to find the largest monotone increasing number that is less than or equal to a given number `n`. The time complexity and space complexity are analyzed as follows:

Time Complexity

- **Best-case scenario**: When the digits are already in a monotone increasing order, the algorithm would traverse the number's digits once. Thus, in the best case, the time complexity is `O(d)`, where `d` is the number of digits in `n`.
- **Worst-case scenario**: The algorithm will need to traverse the digits twice if a decrease is found:
  - Once to identify the first position where `s[i - 1]` is greater than `s[i]`.
  - And a second pass to correct the number by decrementing the digit at position `i - 1` and setting all subsequent digits to `'9'`.

  Since each of these operations is linear with respect to the number of digits in `n`, the worst-case time complexity will also be `O(d)`.

In conclusion, the overall time complexity is `O(d)`, with `d` being the number of digits in `n`.

Space Complexity

- The input number `n` is converted to a list `s` of its digits in string format. This is the only significant additional memory used, which depends on the number of digits `d` in `n`.
- No other additional data structures are used that grow with the input size. The variables used for iteration and comparison are constant in size.

Therefore, the space complexity is `O(d)`, where `d` is the number of digits in `n`.