556. Next Greater Element III

Medium Math Two Pointers String Leetcode Link

Problem Description

Given a positive integer `n`, the goal is to find the smallest integer greater than `n` which contains exactly the same digits as `n`. This number is essentially the next permutation in a lexicographically greater order of the digits. If there is no such number that is greater than `n` with the same digits, the function returns `-1`. Additionally, the answer must fit within a signed 32-bit integer. If the smallest integer is greater than what a 32-bit integer can represent, the function should also return `-1`.

Intuition

The intuition behind the solution involves thinking about what makes one number bigger than another when they have the same digits. The key insight here is that for a number to be just bigger than the given number, the change must be made rightmost such that the digits to the right of this change-point are reversed to form the smallest sequence possible, while creating a number that is bigger than the original number.

1. Identify the rightmost digit that can be swapped to make the number bigger. We start from the right and look for the first digit that is smaller than the digit immediately to its right. If no such digit exists, the digits are in descending order and we cannot form a greater number, hence return `-1`.
2. Swap the identified digit with the smallest digit to its right that is greater than itself. This guarantees that the increase in the number is as small as possible while still being greater than the original number.
3. Reverse the sequence to the right of the original change point. After the swap, the digits to the right of the change point are still in descending order. Reversing them will give us the smallest possible sequence to ensure that the overall number is the smallest possible number greater than `n`.

Implementing this approach, we can navigate through the digit sequence obtained from the integer `n` and incrementally build up our solution step by step until we determine the next greatest element or lack thereof.

Solution Approach

The solution uses a single-pass approach and the concept of rearranging the digits to form the next permutation. To walk through the algorithm:

1. Convert the integer `n` to a list of characters, `cs`, to facilitate the swapping of digits.
2. First, we iterate through `cs` from right to left, trying to find the first pair of digits where the left digit is less than the right digit. This step essentially finds the digit (`cs[i]`) that can be swapped to create a number larger than `n`. The loop variable `i` starts from the second to last index (`n-2`) and decreases.
3. If such a digit `cs[i]` can't be found and `i` becomes less than 0, it means that the digits are in descending order, and no greater number can be formed, hence we return `-1`.
4. Now that we have found the swap position at index `i`, we need to find the smallest digit greater than `cs[i]` to its right. We do this by iterating from the end of `cs` until we find a digit `cs[j]` that is greater.
5. Swap `cs[i]` with `cs[j]` to form a larger number. However, the digits to the right of index `i` are still in descending order, which would make the number larger than necessary.
6. Reverse the sublist `cs[i+1:]` so that we get the smallest possible number that is larger than `n` after the swap.
7. Combine the characters back to form an integer `ans` and check if it is within the 32-bit integer range. If `ans` is larger than `2**31 - 1`, which is the maximum value for a 32-bit signed integer, return `-1`.
8. Finally, if `ans` is within the range, return it. This result is now the smallest integer larger than `n` using its digits.

This solution uses the pattern of finding the next permutation which is a well-known technique that applies to not just numbers but also arrays and sequences where you want to find a lexicographically ordered element.

Example Walkthrough

Let's use the integer `n = 1423` as a small example to illustrate the solution approach.

1. Convert the integer `n` to a list of characters `cs`, so `n = 1423` becomes `cs = ['1', '4', '2', '3']`.
2. Iterate from right to left to find the first digit that is smaller than the digit immediately to its right. In our example, starting from the right, we compare `3` to `2`, `2` to `4`, and then `4` to `1`. The first digit that is smaller than the digit to its right is `2`.
3. The swap position is found at index `i` where `cs[i] = '2'`. Now, find the smallest digit greater than `'2'` to its right. This would be `'3'`.
4. Swap `cs[i]` with `cs[j]` where `cs[j]` is `'3'`. The list `cs` now becomes `['1', '3', '2', '4']`.
5. The digits to the right of index `i` are `['2', '4']`. We reverse this sublist to get the smallest sequence, which will result in `['1', '3', '4', '2']`.
6. Combine the characters to form the new integer `ans = 1342`.
7. Check if `ans` fits within a signed 32-bit integer range. Since `ans = 1342` is less than `2**31 - 1`, it is within the range.
8. Return `ans` as the result. So the smallest integer greater than `1423` containing the same digits is `1342`.

Following this algorithm provides a simple and efficient method to determine the next permutation of an integer with the same set of digits.

Python Solution

```python
class Solution:
    def nextGreaterElement(self, num: int) -> int:
        # Convert the number to a list of characters for easy manipulation
        char_list = list(str(num))
        length = len(char_list)

        # Start from the second last element and move backward to find the first element which is smaller than its next element
        i = length - 2
        while i >= 0 and char_list[i] >= char_list[i + 1]:
            i -= 1

        # If we can't find such an element, it means the number cannot be greater in its permutation
        if i < 0:
            return -1

        # Now find an element which is greater than the found element at i
        j = length - 1
        while char_list[i] >= char_list[j]:
            j -= 1

        # Swap these two elements
        char_list[i], char_list[j] = char_list[j], char_list[i]

        # Reverse the sublist after the index i to get the next greater element in terms of permutation
        char_list[i + 1 :] = char_list[i + 1:][::-1]

        # Convert the char list back to an integer
        next_greater = int(''.join(char_list))

        # If the next greater number is bigger than the largest 32-bit integer, return -1
        return next_greater if next_greater <= 2**31 - 1 else -1

```

Java Solution

```java
class Solution {
    public int nextGreaterElement(int n) {
        // Convert the integer to a char array for easy manipulation of digits
        char[] digits = String.valueOf(n).toCharArray();
        int length = digits.length;

        // Start from the second last digit and move left until you find a digit
        // that is less than its immediate right digit.
        int i = length - 2;
        while (i >= 0 && digits[i] >= digits[i + 1]) {
            i--;
        }

        // If no such digit is found, no greater permutation exists.
        if (i < 0) {
            return -1;
        }

        // Start from the end of the array and move left until you find a digit
        // that is greater than the digit found above.
        int j = length - 1;
        while (digits[i] >= digits[j]) {
            j--;
        }

        // Swap the two digits found above.
        swap(digits, i, j);

        // Reverse the digits to the right of the position where the swap was made
        // to get the next greater permutation with the smallest increase.
        reverse(digits, i + 1, length - 1);

        // Convert the resultant char array back to long to check for integer overflow.
        long result = Long.parseLong(new String(digits));

        // If the result is greater than the max value for int,
        // return -1, otherwise, cast to int and return.
        return result > Integer.MAX_VALUE ? -1 : (int) result;
    }

    // Helper method to swap two elements in a char array.
    private void swap(char[] array, int i, int j) {
        char temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }

    // Helper method to reverse a part of the char array between two indices.
    private void reverse(char[] array, int start, int end) {
        while (start < end) {
            swap(array, start, end);
            start++;
            end--;
        }
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int nextGreaterElement(int n) {
        // Convert the given integer to a string to manipulate individual digits.
        string numStr = to_string(n);

        // Get the length of the string representation of the integer.
        int length = numStr.size();

        // Initialize i for scanning the string from the right, starting at the second last digit.
        int i = length - 2;

        // Initialize j to point at the last digit of the number.
        int j = length - 1;

        // Find the first digit (scanning from right to left) that is smaller than the digit to its right.
        while (i >= 0 && numStr[i] >= numStr[i + 1]) {
            --i;
        }

        // If no such digit is found, the number cannot be made greater, return -1.
        if (i < 0) return -1;

        // Find the rightmost digit that is greater than numStr[i], starting from the end of the string.
        while (numStr[i] >= numStr[j]) {
            --j;
        }

        // Swap the digits at index i and j.
        swap(numStr[i], numStr[j]);

        // Reverse the substring starting from the digit next to index i to the end of the string.
        // This gives us the next greater permutation of the number.
        reverse(numStr.begin() + i + 1, numStr.end());

        // Convert the modified string back to a long integer.
        long ans = stol(numStr);

        // Return the answer if it's within the range of a 32-bit integer,
        // otherwise return -1.
        return ans > INT_MAX ? -1 : static_cast<int>(ans);
    }
};

```

Typescript Solution

```typescript
// TypeScript function to find the next greater permutation of a given integer.
function nextGreaterElement(n: number): number {
  // Convert the given integer 'n' to a string to manipulate individual digits.
  let numStr: string = n.toString();

  // Initialize 'i' for scanning the string from the right, starting at the second last digit.
  let i: number = numStr.length - 2;

  // Find the first digit (scanning from right to left) that is smaller than the digit to its right.
  while (i >= 0 && numStr.charAt(i) >= numStr.charAt(i + 1)) {
    i--;
  }

  // If no such digit is found, the number cannot be made greater, return -1.
  if (i < 0) return -1;

  // Initialize 'j' to point at the last digit of the number.
  let j: number = numStr.length - 1;

  // Find the rightmost digit that is greater than numStr[i], starting from the end of the string.
  while (numStr.charAt(i) >= numStr.charAt(j)) {
    j--;
  }

  // Swap the digits at index 'i' and 'j'. We create a character array for swapping.
  let numChars: string[] = numStr.split('');
  [numChars[i], numChars[j]] = [numChars[j], numChars[i]]; // Perform the swap.
  numStr = numChars.join(''); // Convert the character array back to a string.

  // Reverse the substring starting from the digit next to index 'i' to the end of the string.
  let toReverse: string = numStr.substring(i + 1);
  let reversed: string = toReverse.split('').reverse().join('');
  numStr = numStr.substring(0, i + 1) + reversed; // Concatenate the non-reversed and reversed parts.

  // Convert the modified string back to an integer.
  let ans: number = parseInt(numStr);

  // Return the answer if it's within the range of a 32-bit integer; otherwise, return -1.
  return ans > 2**31 - 1 ? -1 : ans;
}

```

Time and Space Complexity

Time Complexity

The time complexity of this code is `O(n)`, where `n` is the number of digits in the given integer. Here's the breakdown of the time complexity:

1. Converting the integer to a list of characters: This is done in `O(n)` time.
2. Finding the pivot point (`i`) such that `cs[i]` is just less than the element to its right (`cs[i + 1]`): In the worst case, we may have to scan the entire list of digits from right to left, which is done in `O(n)` time.
3. Finding the smallest digit greater than `cs[i]` to its right to swap with (`j`): Again, in the worst-case scenario, this is done in `O(n)` time.
4. Swapping `cs[i]` and `cs[j]`: This is done in constant time, `O(1)`.
5. Reversing the sublist after the pivot point: The sublist is at most `n - 1` elements long and reversing it is done in `O(n)`.
6. Joining the list into a string and converting back to an integer: This operation is performed in `O(n)` time.
7. The final check to ensure the answer is within a 32-bit integer range is performed in constant time, `O(1)`.

All other operations are constant with respect to the size of the input and, as such, do not change the overall `O(n)` complexity.

Space Complexity

The space complexity of the code is also `O(n)`, where 'n' is the number of digits in the given integer. This is because:

1. We store the list of characters, `cs`, which takes `O(n)` space.
2. The additional space used when creating the sublist and reversing it also does not exceed `O(n)`.

There are no recursive calls or additional data structures that depend on the size of the input which would increase the space complexity further.