2566. Maximum Difference by Remapping a Digit

Easy Greedy Math

Problem Description

In this problem, we are given an integer `num`, and we are told that Danny will remap exactly one of the 10 digits (0 through 9) to another digit. Remapping a digit `d1` to `d2` means replacing all occurrences of `d1` in `num` with `d2`. We need to calculate the difference between the largest and smallest numbers that can be created by remapping exactly one digit. A couple of important rules to note:

- Danny can choose to change a digit to the same digit (effectively not changing the number).
- Danny may remap a digit differently when seeking to create the maximum value and minimum value.
- The remapped number is allowed to have leading zeros.
- The goal is to maximize and minimize the given number separately by changing only one digit and then find the difference between these two values.

Intuition

To find the minimum number, we should look to replace a non-zero digit with `0`. The best candidate here is the first non-zero digit, and it should be turned into a zero to minimize the number, taking into account the possibility of leading zeros.

For the maximum number, we aim to replace a digit with `9`. However, we must be strategic in choosing which digit to replace:

- We can ignore any `9`s in the number since changing them to `9` would be redundant.
- If there's any digit in the number other than `9`, we replace the first occurrence of such a digit with `9` to maximize the number. It's crucial to note that we don't need to look at the rest of the digits once we've performed this remapping since no further changes could create a larger number.

The solution presented checks each digit from the start. It first sets the minimum by replacing the first digit with `0`. Then, it iterates through each digit, and upon finding a digit that is not `9`, it replaces all occurrences of this digit with `9` to find the maximum number. Finally, it returns the difference between the max value obtained and the min value.

This approach works because of the constraints that we only need to remap a single digit, we want to maximize/minimize with one replacement, and we have the freedom to choose different digits when optimizing for the maximum and minimum values.

Solution Approach

The solution approach for this problem is straightforward and cleverly optimizes the search for the maximum and minimum values by remapping only the necessary characters.

1. **Conversion to String**: Initially, the integer `num` is converted into a string. This allows for easy access to its individual digits and simplifies the process of remapping them.
2. **Calculating Minimum**: To calculate the minimum possible number, the solution replaces the first digit in the string with `0` (using Python's `str.replace()` method). This remapped minimum number will be used later to compute the final difference.
3. **Iterative Search for Maximum**: The code then uses a for-loop to iterate through each character in the string representation of `num`. The digits are checked in order, from left to right:
    - If a character is found that's not `9`, the solution replaces all occurrences of this specific character with `9`. After this operation, the maximum possible number is found, and the algorithm stops searching further digits because we've achieved the largest increase possible by changing just one type of digit.
    - If it iterates through all digits and they are all `9`'s, it implies that no increase can occur, hence the maximum number is the same as `num`.
4. **Returning the Difference**: After finding the maximum and minimum values, the function subtracts the minimum from the maximum and returns the resulting difference.

This solution uses the following concepts:

- **String manipulation and processing**: It navigates through the string representation of the integer for both the minimum and maximum calculations, utilizing the fact that strings are easily iterable.
- **Greedy approach**: Remapping the first non-zero for minimum and the first non-nine digit for maximum value ensures we're making the locally optimal choice to minimize or maximize the number. Given the problem constraints, these local choices are also globally optimal.
- **Conditional logic**: By using if-conditions, the algorithm checks for the condition that ensures the maximum increase when replacing a digit with `9`.

The elegancy of this algorithm is its simplicity, as it avoids the need for complex data structures or patterns, and performs its task by the simple but efficient traversal of the string.

Example Walkthrough

Let's demonstrate how the solution approach works with a small example. Suppose we are given the integer `num = 682`. We need to find the largest and smallest numbers by remapping exactly one digit and then calculate the difference between these two remapped numbers.

Step 1: Conversion to String

First, we convert `num` to a string: `num_str = '682'`.

Step 2: Calculating Minimum

Next, to calculate the minimum number, we look for the first non-zero digit. In `num_str`, this is `6`. We then replace this digit with `0`, resulting in a new string: `min_str = '082'`. This is our minimum number with leading zeros allowed, which evaluates to `82`.

Step 3: Iterative Search for Maximum

Now, we iterate through `num_str` to find the first digit that is not `9` and replace all occurrences of this digit with `9`. Starting from the left, `6` is not a `9`, so we replace it: `max_str = '982'`. We stop after this replacement because we are only allowed to remap one digit, and we've created the largest number possible by doing so, which is `982`.

Step 4: Returning the Difference

Finally, we convert `min_str` and `max_str` back to integers and subtract to get the difference: `max_num = 982`, `min_num = 82`, so the difference is `900`.

In this example, by strategically remapping `6` to `0` for the minimum and `6` to `9` for the maximum, we've maximized the difference to `900`.

Solution Implementation

Python

```python
class Solution:
    def minMaxDifference(self, num: int) -> int:
        # Convert the given integer to a string for processing
        num_str = str(num)

        # Replace the first digit of the string with '0' to create the minimum number
        # This works under the assumption that the first digit is not '0'
        # Otherwise, 'mi' would become a different number of digits.
        min_num = int(num_str.replace(num_str[0], '0'))

        # Iterate over the digits of the string
        for digit in num_str:
            # If a digit is not '9', we can create the max number
            # by replacing the first occurrence of that digit with '9'
            if digit != '9':
                max_num = int(num_str.replace(digit, '9'))
                # Return the difference between the max number and the min number
                return max_num - min_num

        # If all digits are '9', return the difference between the original number and min_num
        return num - min_num
```

Java

```java
class Solution {
    public int minMaxDifference(int num) {
        String numStr = String.valueOf(num); // Convert the integer to a string for manipulation
        int minVal = Integer.parseInt(numStr.replace(numStr.charAt(0), '0')); // Replace first digit with '0' to get the minimum valu
        // Iterate over the characters in the string
        for (char digit : numStr.toCharArray()) {
            if (digit != '9') {
                // Replace the current digit with '9' to get the maximum value and return the difference
                return Integer.parseInt(numStr.replace(digit, '9')) - minVal;
            }
        }
        // If all digits are '9', return the difference between the original number and minVal
        return num - minVal;
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to calculate the minimum and maximum difference by altering numbers
    int minMaxDifference(int num) {
        // Convert the input number to a string
        string numStr = to_string(num);

        // Create a copy of the string for later modification
        string maxStr = numStr;

        // Get the first digit of the string
        char firstDigit = numStr[0];

        // Replace all occurrences of the first digit with '0' to create the minimum possible number
        for (char& c : numStr) {
            if (c == firstDigit) {
                c = '0';
            }
        }

        // Convert the modified string back to an integer to obtain the minimum number
        int minNum = stoi(numStr);

        // Iterate over the characters in the copy of the original number string
        for (int i = 0; i < maxStr.size(); ++i) {
            // If a character is not '9', it can be replaced with '9' to maximize the number
            if (maxStr[i] != '9') {
                char currentDigit = maxStr[i];

                // Replace all occurrences of the current digit with '9'
                for (int j = i; j < maxStr.size(); ++j) {
                    if (maxStr[j] == currentDigit) {
                        maxStr[j] = '9';
                    }
                }

                // Convert the maximized string back to an integer and return the difference
                return stoi(maxStr) - minNum;
            }
        }

        // If all characters were '9', return the difference between the original number and the minimum number
        return num - minNum;
    }
};
```

TypeScript

```typescript
function minMaxDifference(num: number): number {
    // Convert the number to a string.
    const numString = num.toString();

    // Replace the first digit of the number with '0's to find the minimum.
    const min = Number(numString.replace(new RegExp(numString[0], 'g'), '0'));

    // Iterate through the string representation of the number.
    for (const digit of numString) {
        // If the current digit is not '9', replace all occurrences of this digit
        // with '9's to get the maximum number and return the difference.
        if (digit !== '9') {
            const max = Number(numString.replace(new RegExp(digit, 'g'), '9'));
            return max - min;
        }
    }

    // In the case where all digits are '9's, the max is the number itself; return the difference.
    return num - min;
}
```

```python
class Solution:
    def minMaxDifference(self, num: int) -> int:
        # Convert the given integer to a string for processing
        num_str = str(num)

        # Replace the first digit of the string with '0' to create the minimum number
        # This works under the assumption that the first digit is not '0'
        # Otherwise, 'mi' would become a different number of digits.
        min_num = int(num_str.replace(num_str[0], '0'))

        # Iterate over the digits of the string
        for digit in num_str:
            # If a digit is not '9', we can create the max number
            # by replacing the first occurrence of that digit with '9'
            if digit != '9':
                max_num = int(num_str.replace(digit, '9'))
                # Return the difference between the max number and the min number
                return max_num - min_num

        # If all digits are '9', return the difference between the original number and min_num
        return num - min_num
```

Time and Space Complexity

Time Complexity

The time complexity of the function `minMaxDifference` can be analyzed by looking at the operations that are executed in sequence:

1. The conversion of `num` into a string `s` is `O(n)`, where `n` is the number of digits in `num`, because each digit has to be processed.
2. The replacement operation to create `mi`, which runs in `O(n)` since in the worst case, it has to check each character to perform the replacement.
3. The for-loop iterates over each character in the string `s` once, resulting in `O(n)` complexity.
4. Inside the loop, a replacement operation is done and can be considered `O(n)`. In the worst case, this runs only once because the function returns immediately after finding the first character that is not '9'.

Since the steps mentioned above are executed sequentially, and the loop has an early return condition, the time complexity of the code is `O(n)` – linear with respect to the number of digits in `num`.

Space Complexity

The space complexity of the function is determined by the extra space used by the variables `s` and `mi`, as well as the space for any intermediate strings created during the replace operations:

1. The space required to store the string `s` is `O(n)`.
2. The integer `mi` does not depend on `n` and is thus `O(1)`.
3. The replacement operation creates a new string each time it is called, but since these strings are not stored and only one exists at any time, the additional space complexity is `O(n)`.

Therefore, the overall space complexity of the function is `O(n)`, which again is linear with respect to the number of digits in `num`.