1432. Max Difference You Can Get From Changing an Integer

Medium Greedy Math

Leetcode Link

Problem Description

In the given problem, we have an integer `num`. We need to perform a series of steps twice in order to generate two different numbers, `a` and `b`. The steps are as follows:

- Select a digit `x` within the range of 0 to 9 from the number.
- Select another digit `y` within the range of 0 to 9. Note that `y` can be the same as `x`.
- Replace all instances of `x` with `y` in the number to create a new variant.
- Ensure that the resulting number doesn't have any leading zeros and isn't zero itself.

After performing these steps twice, we want to find the maximum difference between the two resulting numbers `a` and `b`. The underlying challenge is to decide which digits to replace in order to maximize this difference.

Intuition

The intuition behind the solution involves two primary goals: first, maximize the value of `a`, and second, minimize the value of `b`. To increase `a` as much as possible, we should replace the first non-nine digit (from left to right) with a nine. This ensures the greatest increase in the number's value.

Conversely, to minimize `b`, we should reduce the value of the first digit if it is not already a one, by replacing it with one. This is because the first digit has the largest weight in determining the value of the number. If the first digit is already a one, we search for the first non-zero and non-one digit to change to zero, since one cannot be replaced with zero (as it would not decrease the number's value). This approach is used because the digit zero gives the lowest possible value, and we want `b` to be as small as it can be.

Making these replacements produces two numbers, `a` and `b`, that are on the opposite ends of the possible range, thus maximizing the difference between them. The code correctly implements this strategy, and by converting the resulting strings back to integers and subtracting `b` from `a`, we yield the desired maximum difference.

Solution Approach

The implemented solution follows a straightforward approach:

1. Convert the original number `num` to a string, so we can conveniently access and replace digits.
2. To compute `a`, iterate over the string representation of `num` to find the first digit that is not `9`. Once such a digit is found, replace all instances of this digit with `9` in the string. This guarantees the largest possible increase for `a`, as replacing any digit with `9` will yield the highest number possible, given the constraint that we change all occurrences of the chosen digit.
3. For computing `b`, we look at the first digit of the string representation. If it is not `1`, we replace all instances of this first digit with `1`. We do this because the leftmost digit carries the most weight in determining the size of the number, and changing it to `1` gives us the largest possible decrease, without violating the constraint that the new integer cannot have leading zeros or be zero.
4. If the first digit is already `1`, we skip it and then iterate through the rest of the digits to find the first digit that isn't `0` or `1` and replace all instances of this digit with `0`. This ensures that `b` becomes the lowest possible number without resulting in a leading zero or turning the number into zero.
5. After replacement, we convert `a` and `b` back to integers and calculate the difference `a - b`, which represents the maximum possible difference obtained by applying the given operations twice.

The solution effectively uses Python's string manipulation capabilities to replace characters. No complex data structures or algorithms are required, as the problem boils down to making the right choices on which digits to replace during each step.

By making the optimal replacements, the algorithm ensures that the difference between the maximum and minimum possible values after transformation is as large as possible.

Example Walkthrough

Let's illustrate the solution approach with a small example where `num = 2736`. Our goal is to perform the steps mentioned in the solution to maximize the difference between the two resulting numbers `a` and `b`.

Step 1: Convert `num` to a string to handle each digit individually.

- `num_str = "2736"`

Step 2: Maximize the number `a`.

- Iterate over `num_str` from left to right and find the first digit that is not `9`. In this case, it is `2`.
- Replace all instances of `2` with `9`. Our new string for `a` is `"9736"`.

Step 3: Minimize the number `b`.

- Check the first digit of `num_str`. It is `2`, which is not `1`, so we replace it with `1`.
- However, since we have already replaced `2` with `9` for calculating `a`, we can't change `b` the same way.
- So, for `b`, we start from the original `num_str` which is `"2736"` and perform replacement:
  - The first digit is `2` and not `1`, so for `b`, all instances of `2` are replaced with `1`. The new string for `b` is `"1736"`.

Step 4: Address the case where the first digit is already `1`

- This step is not applicable to our example, as we have already replaced the first digit with `1` since it was not `1` to begin with.

Step 5: Calculate the difference between `a` and `b`.

- Convert the new strings `"9736"` for `a` and `"1736"` for `b` back to integers.
- `a = 9736`
- `b = 1736`
- The maximum difference is `a - b = 9736 - 1736 = 8000`.

By following these steps, the solution has correctly and efficiently maximized the difference between `a` and `b`, resulting in a difference of `8000`. The ability to identify which digits to replace is key to achieving the optimal result in this scenario.

Python Solution

```python
class Solution:
    def maxDiff(self, num: int) -> int:
        # Convert the given number to a string for character manipulation
        str_num = str(num)

        # Find the maximum number (replace one non '9' digit with '9')
        max_num = str_num
        for digit in str_num:
            if digit != '9':
                # Replace all instances of the first non '9' digit with '9'
                max_num = max_num.replace(digit, '9')
                break  # Break after the first replacement

        # Find the minimum number
        # If the first digit is not '1', replace all instances of it with '1'
        min_num = str_num
        if str_num[0] != '1':
            min_num = min_num.replace(str_num[0], '1')
        else:
            # Otherwise, for the rest of the digits, find the first digit that is not '0' or '1' and replace all instances with '0'
            for digit in str_num[1:]:
                if digit not in '01':
                    min_num = min_num.replace(digit, '0')
                    break  # Break after the first replacement

        # Return the difference between the maximum and minimum number
        return int(max_num) - int(min_num)

```

Java Solution

```java
class Solution {

    // This method calculates the maximum difference between two numbers
    // that can be obtained by changing the digits of the original number.
    public int maxDiff(int num) {
        // Convert the integer to a String for easier manipulation.
        String numStr = String.valueOf(num);
        // Create two copies of the string, one for the maximum value and one for the minimum.
        String maxNumStr = numStr;
        String minNumStr = numStr;

        // Find the first non-'0' digit and replace all its occurrences with '9' to get the maximum number.
        for (int i = 0; i < numStr.length(); ++i) {
            if (numStr.charAt(i) != '9') {
                maxNumStr = numStr.replace(numStr.charAt(i), '9');
                break;
            }
        }

        // For minimum number, if the first digit is not '1', replace all its occurrences with '1'.
        if (minNumStr.charAt(0) != '1') {
            minNumStr = minNumStr.replace(minNumStr.charAt(0), '1');
        } else {
            // If the first digit is '1', find the first digit that is not '0' or '1' from the second digit onwards
            // and replace all its occurrences with '0'.
            for (int i = 1; i < minNumStr.length(); ++i) {
                if (minNumStr.charAt(i) != '0' && minNumStr.charAt(i) != '1') {
                    minNumStr = minNumStr.replace(minNumStr.charAt(i), '0');
                    break;
                }
            }
        }

        // Parse the max and min strings back to integers and return the difference.
        return Integer.parseInt(maxNumStr) - Integer.parseInt(minNumStr);
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to replace all occurrences of a character 'from' with 'to' in a string 's'
    void replaceAll(std::string& s, char from, char to) {
        for (char& c : s) {
            if (c == from) {
                c = to;
            }
        }
    }

    // Function to calculate the maximum difference between two numbers you can get
    // by changing digits of the original number 'num'
    int maxDiff(int num) {
        // Convert the number to a string for easy manipulation
        std::string highestNumStr = std::to_string(num);
        std::string lowestNumStr = highestNumStr;

        // Crete the highest possible number by replacing the first non '9' digit with '9'
        for (int i = 0; i < highestNumStr.size(); ++i) {
            if (highestNumStr[i] != '9') {
                replaceAll(highestNumStr, highestNumStr[i], '9');
                break;
            }
        }

        // Create the lowest possible number
        if (lowestNumStr[0] != '1') {
            // If the first digit is not '1', replace it with '1'
            replaceAll(lowestNumStr, lowestNumStr[0], '1');
        } else {
            // If the first digit is '1', find the next digit that is not '0' or '1' and replace it with '0'
            for (int i = 1; i < lowestNumStr.size(); ++i) {
                if (lowestNumStr[i] != '0' && lowestNumStr[i] != '1') {
                    replaceAll(lowestNumStr, lowestNumStr[i], '0');
                    break;
                }
            }
        }

        // Convert the modified strings back to integers and return the difference
        return std::stoi(highestNumStr) - std::stoi(lowestNumStr);
    }
};

```

Typescript Solution

```typescript
// Function to calculate the maximum difference between two numbers you can get
// by altering characters of the original number 'num'
function maxDiff(num: number): number {
    // Convert the number to a string for easy manipulation
    let highestNumStr: string = num.toString();
    let lowestNumStr: string = highestNumStr;

    // Create the highest possible number by replacing the first non '9' digit with '9'
    for (let i: number = 0; i < highestNumStr.length; ++i) {
        if (highestNumStr[i] !== '9') {
            highestNumStr = highestNumStr.replaceAll(highestNumStr[i], '9');
            // After replacement is done, break out of the loop
            break;
        }
    }

    // Create the lowest possible number
    if (lowestNumStr[0] !== '1') {
        // If the first digit is not '1', replace it with '1'
        lowestNumStr = lowestNumStr.replaceAll(lowestNumStr[0], '1');
    } else {
        // If the first digit is '1', find the next digit that is not '0' or '1' and replace it with '0'
        for (let i: number = 1; i < lowestNumStr.length; ++i) {
            if (lowestNumStr[i] !== '0' && lowestNumStr[i] !== '1') {
                lowestNumStr = lowestNumStr.replaceAll(lowestNumStr[i], '0');
                // After replacement is done, break out of the loop
                break;
            }
        }
    }

    // Convert the modified strings back to numbers and compute the difference
    // The difference is returned as the result
    return parseInt(highestNumStr, 10) - parseInt(lowestNumStr, 10);
}

```

Time and Space Complexity

The given Python code defines a method `maxDiff` that computes the maximum difference by transforming the input integer into its greatest and smallest possible values by changing its digits.

Time Complexity:

To determine the time complexity, we consider the length of the string representation of the input number `n` as `d`.

- The method converts the number to a string twice, which takes $O(d)$ time.
- The first `for` loop iterates over each character in the string `s`, and in the worst case, this would be `d` iterations. The `replace` operation inside the loop can potentially replace `d - 1` characters in the worst case, taking $O(d)$ time. However, since the loop breaks after the first replacement, this loop runs at most once, so it is $O(d)$.
- There's a similar `for` loop for string `s`. The worst-case scenario would be checking each character until the second to last, and performing one replacement operation which also takes $O(d)$ time.

Hence, the overall time complexity of the function is $O(d)$ due to the string manipulation operations being based on the length of the number's string representation.

Space Complexity:

For space complexity, we consider the extra space used by the algorithm besides the input.

- Two new string variables `a` and `b` are created based on the number's string representation, which consume $O(d)$ space together.
- No additional data structures are used that grow with the length of the string representation of the input.

Thus, the space complexity is $O(d)$, where `d` is the length of the string representation of the number since the space required depends only on the length of the string copies `a` and `b`.