2119. A Number After a Double Reversal

Easy Math

Leetcode Link

Problem Description

This problem asks us to implement a function that performs an operation on an input integer `num` and checks for a certain condition. The operation involves **reversing** the digits of `num`, which we can call `reversed1`, and then reversing the digits of `reversed1`, resulting in `reversed2`. The condition to check is whether `reversed2` is equal to the original integer `num`. If they are equal, the function should return `true`; otherwise, it should return `false`. It is important to note that when reversing an integer, any leading zeros that appear as a result of the reversal are not retained.

An example of the operation would be:

- If `num` is `123`, `reversed1` would be `321`, and then `reversed2` would be `123`, which is equal to the original `num`.
- If `num` is `1200`, `reversed1` would be `21` (since leading zeros are not retained), and `reversed2` would be `12`, which is not equal to the original `num`.

Intuition

Upon analyzing the problem, we realize that reversing a number twice will always give us back the original number with one exception: if the original number has trailing zeros. This is because, during the first reversal, the trailing zeros are not retained, leading to a smaller number. When reversing for the second time, there is no way to get those zeros back; hence, `reversed2` will not equal `num` if `num` has trailing zeros.

Therefore, the only time `reversed2` is not equal to `num` is when `num` has one or more trailing zeros. The exception to this would be the number 0 itself, which remains the same even after multiple reversals.

So the solution can be reduced to a simple check:

- If `num` is 0, return `true`, because reversing 0 will always yield 0.
- Otherwise, check if `num` is divisible by 10 (which means it has at least one trailing zero), and if so, return `false`.
- In all other cases, return `true`, because any number without trailing zeros will stay the same after two reversals.

This is the thinking process that brings us to the concise solution provided in the code.

Solution Approach

The solution uses a straightforward logical check to determine whether the input `num` will remain the same after two reversals. This solution does not perform the actual reversal of the number, which would require additional operations and could be less efficient, especially for large numbers.

Algorithmically, the approach can be broken down as follows:

1. **Zero Check**: Directly return `true` if `num` is 0, as reversing 0 any number of times will still yield 0. This is a quick check to handle this special case.
2. **Trailing Zero Check**: For any number other than 0, check if it has trailing zeros, which we do by checking if `num` is divisible by 10 (using the modulo operator `%`). If it is, this means `num` ends with at least one zero and therefore will not be the same after two reversals. In this case, we return `false`.
3. **Default Case**: If neither of the above conditions is met, it implies that the number `num` does not have any trailing zeros and will remain the same after two reversals. Hence, we return `true` by default.

This approach is efficient because it operates in constant time (O(1)), and there is no need for additional data structures (no extra space complexity). We are using the characteristics of the problem itself to come up with a logical shortcut that bypasses the need for computation-heavy string conversions or arithmetic operations associated with actual number reversal.

The Python code for the solution is straightforward and concise:

```python
class Solution:
    def isSameAfterReversals(self, num: int) -> bool:
        # Return true if num is 0 or it does not end with a zero (not divisible by 10)
        return num == 0 or num % 10 != 0
```

This code establishes the implementation of the logical checks discussed above in step-by-step fashion, resulting in an elegant and efficient solution to the problem.

Example Walkthrough

Let's consider an integer `num = 1200` to illustrate the solution approach.

1. **Zero Check**: The function first checks if `num` is zero. In our example, `num` is 1200, so this condition fails. We do not return `true` here.
2. **Trailing Zero Check**: Next, the function checks if `num` ends with at least one zero. This is done by checking if `num` is divisible by 10. For our number, `1200 % 10 == 0` is `true`, which means `num` ends with a zero. According to our analysis, if a number ends with a zero, it will not be the same after two reversals due to loss of trailing zeros during the first reversal. Thus the function should return `false`.
3. **Default Case**: This step would only be reached if the number were not zero and did not end with a zero. But since our example number fails the Trailing Zero Check, the Default Case does not apply.

Applying the solution approach from the provided content, the function should return `false` for the input `num = 1200`. This means that after reversing the digits of 1200 to form `reversed1 = 21`, and then reversing `reversed1` to get `reversed2 = 12`, `reversed2` is not equal to the original `num` as a result of the missing trailing zeros.

Putting the example to test with the implemented code:

```python
solution = Solution()
result = solution.isSameAfterReversals(1200)
print(result)  # Output: False
```

The function correctly identifies that `1200`, after two reversals, does not equal the original number due to the trailing zeros being dropped during the reversal process.

Python Solution

```python
class Solution:
    def is_same_after_reversals(self, num: int) -> bool:
        # Check if a number remains the same after a double reversal.
        # The function returns True if the number is 0 because reversing 0
        # will always result in 0. Additionally, if the number doesn't end in 0,
        # the function also returns True because reversing it twice will lead to
        # the original number. However, if a non-zero number ends in 0, reversing
        # it twice won't result in the original number, so it returns False.

        # Check for the special case where the number is 0.
        if num == 0:
            return True
        # Check if number ends in 0, as such numbers are not the same after two reversals.
        elif num % 10 != 0:
            return True
        # If the number ends in 0 and is not itself 0, it fails the condition.
        else:
            return False

```

Java Solution

```java
class Solution {

    // Method to determine if reversing a number twice results in the original number
    public boolean isSameAfterReversals(int num) {
        // If the number is 0, it remains the same after any number of reversals
        // Also, if the last digit of the number is not 0, it will retain its
        // last digit when reversed, so reversing it again will result in the original number
        // However, if the last digit is 0, the first reversal will trim the zero,
        // hence the second reversal won't give back the original number
        return num == 0 || num % 10 != 0;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Checks if reversing a number twice gives back the same number
    bool isSameAfterReversals(int num) {
        // A number will be the same after two reversals if:
        // 1. It's zero (since reversing 0 gives 0),
        // 2. Its last digit is not zero (since otherwise, the first reversal will remove the last zero).
        return num == 0 || num % 10 != 0;
    }
};

```

Typescript Solution

```typescript
// Checks if reversing a number twice gives back the same number
function isSameAfterReversals(num: number): boolean {
    // A number will be the same after two reversals if:
    // 1. It's zero (since reversing 0 gives 0),
    // 2. Its last digit is not zero (since otherwise, the first reversal would
    // remove the trailing zero and the second reversal wouldn't restore it).
    return num === 0 || num % 10 !== 0;
}

```

Time and Space Complexity

The given code checks whether a number is the same after reversing it twice. The operation involves checking if the number is zero or if the last digit of the number is not zero.

Time Complexity

The operation `num % 10` is a modulo operation, which has a constant time complexity, i.e., `O(1)`. Checking if a number is equal to zero or not (`num == 0`) is also a constant time operation. Since there are no loops or recursive calls, the overall time complexity of the function is `O(1)`.

Space Complexity

The function uses a fixed amount of space, with only one input integer and no additional data structures or recursive calls that would use more space. As a result, the space complexity is constant as well, denoted as `O(1)`. The function does not allocate any additional memory that grows with the input size, so the space used remains consistent regardless of the input.