2443. Sum of Number and Its Reverse

Medium Math Enumeration

Problem Description

In the context of this LeetCode problem, one is tasked with determining whether a non-negative integer `num` can be represented as a sum of any non-negative integer and its reverse. The reverse of an integer is the number obtained by reversing the order of its digits. For instance, the reverse of `123` is `321`. The problem requires the function to return `true` if such a representation is possible for the given `num`, or `false` otherwise.

Intuition

The straightforward approach to solve this problem relies on the simple brute force method. We consider all non-negative integers from `0` up to `num` inclusive, because the sum of a number and its reverse cannot be greater than `num` itself. For each of these integers, named `k`, we calculate its reverse by converting `k` to a string, reversing the string, and converting it back to an integer. Then, we check whether the original number `k` plus its reverse equals `num`. If we find any such `k` that satisfies this condition, the function returns `true`. If the loop completes without finding any valid `k`, the function returns `false`. This approach ensures that all possibilities are checked.

This process is efficiently executed in the solution code using a generator expression within the `any` function, which iteratively checks each number until a match is found, and returns `true` as soon as a satisfying number is encountered.

Solution Approach

For this problem, the implementation is fairly straightforward and does not involve complex data structures or advanced algorithms. It's a direct translation of the brute force approach into Python code.

The solution defines a method `sumOfNumberAndReverse` within the `Solution` class. This method takes one parameter, `num`, which is the non-negative integer that we want to examine.

The core of the implementation is the expression:

```
any(k + int(str(k)[::-1]) == num for k in range(num + 1))
```

Here's a step-by-step walk-through of what's happening:

1. `range(num + 1)`: We create an iterable sequence of numbers starting from `0` up to and including `num`. This is because the largest number that, when added to its reverse, could potentially equal `num` is `num` itself.
2. `str(k)[::-1]`: For each number `k` in the range, we convert `k` into a string using `str(k)` then reverse the string by applying the slicing operation `[::-1]`. This slice notation is a Python idiom for reversing sequences.
3. `int(...)`: The reversed string is converted back to an int because we need to perform arithmetic with it.
4. `k + int(str(k)[::-1]) == num`: We add the integer `k` to its reverse and check if the sum is equal to `num`.
5. `any(...)`: This is a built-in Python function that takes an iterable and returns `True` if at least one element in the iterable is `True`. In this case, it iterates over the generator expression, which yields `True` or `False` for each `k` in the sequence based on whether `k` plus its reverse equals `num`.
6. `return ...`: Finally, the method returns the result of the `any` function. If any value of `k` found satisfies the condition, `True` is returned; otherwise, `False` is returned.

This solution is elegant and concise thanks to Python's high-level abstractions but comes with an O(n) time complexity, as it might need to check all integers from `0` to `num`. The space complexity, on the other hand, is O(1) since there are no additional data structures consuming memory based on the input size; the integers are generated one by one.

Example Walkthrough

Let's consider a small example using the number `num = 121` to illustrate the solution approach. We want to determine if there exists a non-negative integer `k` such that when `k` is added to its reverse, the sum equals `121`.

1. First, we generate a sequence of numbers from `0` to `121` inclusive, because these are all the potential candidates for `k` that, when combined with their reverse, could equal `num`.
2. We then iterate through these numbers one by one. For each iteration, let's denote our current number as `k`.
3. We reverse the digits of `k`. If `k` was `12`, the reversed version would be `21` which is obtained by converting `k` to a string (`"12"`), reversing the string (`"21"`), and converting it back to an integer (`21`).
4. We add the original `k` to its reversed version. For `k = 12`, this would be `12 + 21` which equals `33`.
5. We check if this sum equals `num` (`121` in this case). Since `33` is not equal to `121`, we continue this process with the next number in the sequence.
6. If at any point the sum of `k` and its reverse equals `121`, we will return `True`. For example, if `k` were `112`, its reverse would be `211`, and adding those together yields `112 + 211 = 323` which is not `121`. So we move on.
7. If we have gone through all numbers up to `121` and have not found a sum that equals `121`, we will return `False`.
8. Luckily, when `k = 29`, we find that its reverse is `92`, and adding them together yields `29 + 92 = 121`. Since this satisfies our condition, the `any` function will immediately return `True`, indicating that `num = 121` can indeed be expressed as the sum of a number and its reverse.

Solution Implementation

Python

```python
class Solution:
    def sum_of_number_and_reverse(self, num: int) -> bool:
        # Iterate over all numbers from 0 to num, inclusive
        for integer in range(num + 1):
            # Calculate the reverse of the current number by converting it to a string,
            # reversing it, and then casting it back to an integer
            reverse_integer = int(str(integer)[::-1])

            # Check if the sum of the current number and its reverse equals the input number
            if integer + reverse_integer == num:
                # If a match is found, return True
                return True

        # If no match is found in the iteration, return False
        return False
```

Java

```java
class Solution {
    /**
     * Checks whether a given number can be expressed as
     * the sum of a number and its reverse.
     *
     * @param num The number to check.
     * @return true if the number can be expressed as the sum
     *         of a number and its reverse, otherwise false.
     */
    public boolean sumOfNumberAndReverse(int num) {
        // Loop from 0 to the given number (inclusive)
        for (int originalNumber = 0; originalNumber <= num; ++originalNumber) {
            int reversedNumber = 0;
            int temp = originalNumber;

            // Reverse the current number
            while (temp > 0) {
                int lastDigit = temp % 10;
                reversedNumber = reversedNumber * 10 + lastDigit;
                temp /= 10;
            }

            // Check if the sum of the original and reversed number is equal to the input number
            if (originalNumber + reversedNumber == num) {
                return true; // Found a pair that satisfies the condition
            }
        }

        // If no such pair is found in the loop, return false
        return false;
    }
}
```

C++

```cpp
class Solution {
public:
    // Checks if a number can be expressed as the sum of a number and its reverse
    bool sumOfNumberAndReverse(int num) {
        // Loop through all numbers starting from 0 up to the given number
        for (int original_number = 0; original_number <= num; ++original_number) {
            int remaining = original_number;
            int reversed_number = 0;
            // Reverse the original_number
            while (remaining > 0) {
                reversed_number = reversed_number * 10 + remaining % 10; // Append the last digit of remaining to reversed_number
                remaining /= 10; // Remove the last digit from remaining
            }
            // Check if the sum of the original number and its reverse equals the given number
            if (original_number + reversed_number == num) {
                return true; // If the condition is met, return true
            }
        }
        return false; // If no such pair is found, return false
    }
};
```

TypeScript

```typescript
// Checks if a given number is equal to the sum of another number and its reverse.
// num: Number to check for this property.
// Returns a boolean value indicating whether such a pair exists.

function sumOfNumberAndReverse(num: number): boolean {
    // Iterate over all numbers from 0 to the input number
    for (let i = 0; i <= num; i++) {
        // Calculate the reverse of the current number 'i'
        const reversedNumber = Number([...i.toString()].reverse().join(''));

        // Check if the current number plus its reverse equals the input number
        if (i + reversedNumber === num) {
            // Return true if the condition holds for the current number
            return true;
        }
    }
    // After checking all numbers, return false if no suitable pair was found
    return false;
}
```

```python
class Solution:
    def sum_of_number_and_reverse(self, num: int) -> bool:
        # Iterate over all numbers from 0 to num, inclusive
        for integer in range(num + 1):
            # Calculate the reverse of the current number by converting it to a string,
            # reversing it, and then casting it back to an integer
            reverse_integer = int(str(integer)[::-1])

            # Check if the sum of the current number and its reverse equals the input number
            if integer + reverse_integer == num:
                # If a match is found, return True
                return True

        # If no match is found in the iteration, return False
        return False
```

Time and Space Complexity

Time Complexity

The given function `sumOfNumberAndReverse` performs a linear search from `0` to `num` inclusive. For each value `k` in this range, the function calculates the reverse of the number by converting it to a string, reverse the string (this is done using `str(k)[::-1]`), and then converting it back to an integer. After this, it checks if the sum of the number `k` and its reverse equals the input number `num`.

Therefore, we can say the time complexity of the function is O(n * m), where `n` is the value of `num` and `m` represents the time taken to reverse the number. Since the number of digits `d` in the number `k` can be represented as O(log(k)), the reverse operation is O(d) which simplifies to O(log(n)) for the worst case when `k` is close to `num`. Thus, the overall time complexity is O(n * log(n)).

Space Complexity

The space complexity of the function is O(1). The reason for this constant space complexity is because aside from a few variables (`k` and the reversed number), the function does not use any additional space that scales with the input size `num`. The inputs and variables are of a fixed size, so it doesn't matter how large `num` is, the space used by the function remains constant.