2544. Alternating Digit Sum

Easy Math Leetcode Link

Problem Description

You are provided with a positive integer `n`. The integer `n` contains digits, and each digit is meant to be given a sign. The rule for assigning signs to the digits of `n` is as follows:

- The leftmost digit (most significant digit) gets a positive sign.
- Every other digit after the first one has a sign that is the opposite to the one of its immediately preceding digit. This means the sign alternates between positive and negative as you move from the most significant digit to the least significant digit.

Your task is to calculate the sum of these digits considering their signs and return the result.

Example: If `n` is `321`, according to the rules, the sums are: `3 * (+1) + 2 * (-1) + 1 * (+1) = 3 - 2 + 1 = 2`.

You need to implement a function that takes the integer `n` and returns the sum with consideration of the alternating signs on the digits.

Intuition

The solution approach revolves around applying the rules of alternating signs while iterating through the digits of the given integer. The process can be broken down into the following steps:

1. Convert the integer `n` into a string to easily access individual digits.
2. Iterate over each character in the string, which represents a digit of `n`.

The implementation uses the `enumerate` function to get both the index and the character during the iteration. The index `i` starts at `0` for the most significant digit which implies:

- If `i` is even, the sign is positive.
- If `i` is odd, the sign is negative.

3. Convert the character back to an integer, multiply it by the sign determined by its index (`(-1) ** i`, which is `1` for even and `-1` for odd indices).
4. Sum up these adjusted values as you iterate.

The approach is efficient and clever because it avoids the need for explicitly checking the parity of each index or maintaining an additional variable to keep track of the sign. Instead, it takes advantage of the properties of exponents, where raising `-1` to an even power yields `+1` and to an odd power yields `-1`, thus naturally alternating the sign according to the index.

In summary, the code uses a string conversion, enumeration, and a mathematical trick to apply the alternating signs and sum the digits in a concise and elegant manner.

Solution Approach

The implementation of the solution follows several key programming concepts. Here's a walkthrough that includes the algorithms, data structures, or patterns used:

1. **String Conversion**: The first step is to take the integer `n` and convert it to a string using `str(n)`. This is crucial because it provides a way to iterate over the digits individually. In Python, strings are iterable, meaning we can loop through its characters (digits, in this case) one by one.

2. **Enumeration**: The next step employs the `enumerate` function. This built-in function in Python adds a counter to an iterable. So, when we use `enumerate(str(n))`, we get back two pieces of information at each step: the index (`i`) and the character (`x`) representing the digit.

3. **Comprehension and Mathematical Operation**: We use a generator comprehension to process each digit within a single line. The expression `(-1) ** i * int(x)` computes the value of each digit with its correct sign. The exponentiation `(-1) ** i` determines the sign: it's `+1` when `i` is even (including `0` for the most significant digit), and `-1` when `i` is odd.

4. **Summation**: Finally, the `sum()` function takes the generator comprehension and computes the sum of all processed digits. Since the generator comprehension yields the digit with the right sign, the `sum()` function adds them up correctly to produce the final answer.

These combined steps result in a solution that is elegant and efficient. The use of string conversion and comprehension makes the code compact, while the mathematical operation ensures that the alternating sign rule is correctly applied without any need for conditional logic.

The algorithm has a time complexity of O(d), where d is the number of digits in `n`, because it must process each digit exactly once.

In the reference solution provided:

```python
class Solution:
    def alternateDigitSum(self, n: int) -> int:
        return sum((-1) ** i * int(x) for i, x in enumerate(str(n)))
```

The class `Solution` has a method `alternateDigitSum` that follows the aforementioned steps to provide an efficient resolution to the problem.

Example Walkthrough

Let's go through a small example to illustrate the solution approach. Suppose we are given the integer `n = 314`.

According to our rules, we want to calculate the following:

- The first digit ('3') has a positive sign, so its value is `+3`.
- The second digit ('1') has a negative sign, so its value is `-1`.
- The third digit ('4') has a positive sign again, so its value is `+4`.

Applying the alternation rule to get the sum: `3 * (+1) + 1 * (-1) + 4 * (+1) = 3 - 1 + 4 = 6`.

Here's how we use the solution approach to get this answer:

1. **Convert Integer to String**: We start by converting the integer `n` into a string with `str(n)`, which gives us `'314'`. This allows us to individually access each digit as a character in a string.

2. **Enumerate Digits**: By using `enumerate(str(n))`, we get an index and the character value for each digit. So during the iteration, we will get `(0, '3')`, `(1, '1')`, and `(2, '4')`.

3. **Calculate Digit Values with Signs**: We compute the value of each digit by determining its sign. For the first digit, the index `i` is 0, which is even, so the sign is positive. Thus, `(-1) ** i * int(x)` yields `+3`. For the second digit, `i` is 1, which is odd, so we get a negative sign and `(-1) ** i * int(x)` yields `-1`. For the third digit, `i` is 2 (even), resulting in a positive sign once more and `(-1) ** i * int(x)` yields `+4`.

4. **Compute the Sum**: When we sum these up using `sum((-1) ** i * int(x) for i, x in enumerate(str(n)))`, we add `+3`, `-1`, and `+4` which results in `6`.

Thus, when implementing this in the `alternateDigitSum` method of the `Solution` class, the answer returned for the input `314` would be `6`, matching our manual calculation.

Python Solution

```python
class Solution:
    def alternate_digit_sum(self, n: int) -> int:
        # Convert the number to a string to iterate over each digit.
        digits = str(n)

        # Use a list comprehension to calculate the alternating sum.
        # The enumerate function provides the index (i) and the digit (x),
        # which we cast to an integer. We alternate the sign of the sum
        # by raising -1 to the power of i (even indices add, odd indices subtract).
        alternating_sum = sum((-1) ** i * int(x) for i, x in enumerate(digits))

        # Return the final alternating sum.
        return alternating_sum

# Example usage:
# Create an instance of the Solution class.
solution = Solution()

# Call the alternate_digit_sum method with an example number.
result = solution.alternate_digit_sum(12345)  # Example input number

# The calculated result would then be printed, if needed.
print(result)  # The output for 12345 would be 1-2+3-4+5 = 3

```

Java Solution

```java
class Solution {
    // Function to calculate the alternating sum of digits of a number
    public int alternateDigitSum(int number) {
        // Initialize the answer to 0
        int alternatingSum = 0;
        // Initialize the sign to 1; this will be alternated between 1 and -1
        int sign = 1;

        // Convert the number to its string representation
        // then iterate over each character in the string
        for (char digitChar : String.valueOf(number).toCharArray()) {
            // Convert char digit to its integer value
            int digit = digitChar - '0';
            // Update alternating sum by adding current digit times the sign
            alternatingSum += sign * digit;
            // Alternate sign for next iteration (-1 if it was 1, 1 if it was -1)
            sign *= -1;
        }

        // Return the final alternating sum
        return alternatingSum;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // This function takes an integer 'n' and returns the alternating sum
    // of its digits, where the first digit is added, the second is
    // subtracted, the third is added again, and so on.
    int alternateDigitSum(int n) {
        int answer = 0;       // This will hold the final alternating sum.
        int sign = 1;         // This represents the sign for addition/subtraction.

        // Convert the integer 'n' into a string for easy iteration
        // over its digits.
        string numStr = to_string(n);

        // Loop through each digit character in the string 'numStr'.
        for (char digitChar : numStr) {
            // Convert the digit character to its integer value.
            int digit = digitChar - '0';

            // Add the digit to 'answer' with the appropriate 'sign'.
            answer += sign * digit;

            // Flip the sign to alternate between addition and subtraction.
            sign *= -1;
        }

        // Return the calculated alternating sum.
        return answer;
    }
};

```

Typescript Solution

```typescript
function alternateDigitSum(n: number): number {
    // Initialize the accumulator for the alternating sum of digits
    let accumulatedSum = 0;

    // Variable to track whether to add or subtract the current digit
    let sign = 1;

    // Continue the process until all digits have been processed
    while (n) {
        // Add or subtract the current rightmost digit based on the sign and update the accumulated sum
        accumulatedSum += (n % 10) * sign;

        // Switch the sign for the next iteration
        sign = -sign;

        // Remove the rightmost digit to process the next one
        n = Math.floor(n / 10);
    }

    // Return the final accumulated sum after adjusting the sign back
    // The multiplication by `-sign` is used to negate the sum correctly based on the last iteration's sign
    return accumulatedSum * -sign;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(k)`, where `k` is the number of digits in the integer `n`. This is because the code involves converting the integer `n` into a string, which takes `O(k)` time, and then enumerating over each digit, resulting in a total linear complexity with respect to the number of digits.

Space Complexity

The space complexity of the code is `O(k)`, since the largest amount of memory is used when the integer is converted to a string, which will take up space proportional to the number of digits `k`. The generator used in the `sum()` function does not create a list of all digits, so it does not add to the space complexity.