9. Palindrome Number

Easy Math

Leetcode Link

Problem Description

The given problem asks us to determine if an integer `x` is a **palindrome**. An integer is a palindrome when it reads the same backward as forward. For example, the integers `121` and `12321` are palindromes, while `123` and `-121` are not.

Intuition

The key idea to solving this problem is to reverse half of the number and compare it with the other half. If both halves are the same, then the number is a palindrome. We can do this comparison without using extra space, which improves the efficiency of the algorithm. To achieve this, we follow these steps:

- If a number is negative or if it is a multiple of 10 (except for 0), it cannot be a palindrome; such cases return `False` immediately. For example, `-121` cannot be a palindrome because the minus sign does not reverse, and `10` is not a palindrome because no integer starting with `0` can be a palindrome (except for `0` itself).
- We can then divide the number into two halves. However, instead of dividing the number explicitly, we construct the half reverse incrementally. We initialize a variable `y` to zero and then repeatedly take the last digit of `x` (by `x % 10`) and add it to `y` after first multiplying `y` by 10 (to shift its digits left).
- We do this process while `y` is less than `x`. Eventually, `x` will be reduced to either the first half or just short of the half of the original number (in the case of an odd number of digits), while `y` will contain the reversed second half. By stopping when `y` becomes greater than or equal to `x`, we prevent the need to handle special cases for odd digit numbers.
- Finally, we check if `x` is equal to `y` (which happens when the length of the number is even) or if `x` is equal to `y` divided by 10 (to remove the middle digit in the case of an odd length number). If either condition is true, the original number is a palindrome, and we return `True`; otherwise, we return `False`.

The given Python solution implements this logic within the `isPalindrome` function.

Solution Approach

The implementation of the palindrome check for an integer `x` in Python uses a straightforward approach without the need for additional data structures like arrays or strings. Here is the step-by-step walkthrough of the code and the logic behind the implementation:

- First, the code checks for the edge cases where the number cannot be a palindrome. This includes any negative numbers and any positive numbers that end in `0` – except for `0` itself. In Python, the expression `(x and x % 10 == 0)` will return `True` for all multiples of 10 except 0 because, in Python, 0 is considered as `False`. Thus, `if x < 0 or (x and x % 10 == 0):` checks if `x` is a negative number or a non-zero multiple of 10, returning `False` immediately if either is true.
- The algorithm initializes a variable `y` to zero. This variable serves as a reversed version of the second half of `x`.
- Then, the algorithm enters a `while` loop that continues to iterate as long as `y` is less than `x`. Inside this loop:
  - The last digit of `x` is appended to `y` using `y = y * 10 + x % 10`. The multiplication by 10 moves the current digits of `y` to the left before adding the new digit.
  - The last digit is removed from `x` using `x //= 10`, which is integer division by 10 in Python.
- When the loop ends, two possible scenarios exist:
  - If the original number `x` had an even number of digits, `x` and `y` would now be equal.
  - If the original number `x` had an odd number of digits, `x` would be `y` with the middle digit removed from `y`.
- To check if `x` is a palindrome, the condition `return x in (y, y // 10)` is used, which returns `True` if `x` matches either `y` or `y` divided by 10 (discarding the middle digit for the odd digit case).

The implementation utilizes the modulus operator `%` for extracting digits, integer division `//` for truncating digits, and a while loop to construct the reversed half incrementally. The advantage of this solution is that it operates in place and requires constant space, as it doesn't use extra memory proportional to the number of digits in `x`.

Example Walkthrough

Let's illustrate the solution approach with the example of `x = 1221`. We want to determine if `x` is a palindrome using the steps provided.

1. First, we check if `x` is less than 0 or if `x` is a multiple of 10, but not 0 itself. Since `1221` is a positive number and not a multiple of `10`, we proceed to the next step.
2. We initialize `y` to `0`. This `y` variable will be used to store the reversed number.
3. We enter the while loop and iterate as long as `y < x`. Here's how the loop progresses:
   - *First iteration:*
     - `y = 0 * 10 + 1221 % 10` results in `y = 1`
     - `x = 1221 // 10` leads to `x = 122`
   - *Second iteration:*
     - `y = 1 * 10 + 122 % 10` results in `y = 12`
     - `x = 122 // 10` leads to `x = 12`
4. At this point, `y` is no longer less than `x`. The loop ends, and we have two halves `x = 12` and `y = 12`.
5. Finally, we check if `x == y` or `x == y // 10`. Since `12` is equal to `12`, the function will return `True`, confirming that 1221 is indeed a palindrome.

This walkthrough demonstrates how the algorithm processes an input number through a series of steps to determine if it is a palindrome without requiring additional space for storing reversed numbers or strings.

Python Solution

```python
class Solution:
    def is_palindrome(self, number: int) -> bool:
        # Negative numbers and numbers that end with 0 (and are not 0 itself)
        # cannot be palindromes
        if number < 0 or (number != 0 and number % 10 == 0):
            return False

        reversed_half = 0
        # Build the reversed half of the number to reduce the number of operations
        while reversed_half < number:
            reversed_half = reversed_half * 10 + number % 10
            number //= 10

        # The number is a palindrome if it is the same as the reversed half or
        # the same as the reversed half without the last digit (for odd-length numbers)
        return number == reversed_half or number == reversed_half // 10
```

Java Solution

```java
class Solution {
    public boolean isPalindrome(int number) {
        // Any negative number cannot be a palindrome
        // Additionally, if the last digit is 0, the number cannot be a palindrome
        // unless the number is 0 itself.
        if (number < 0 || (number % 10 == 0 && number != 0)) {
            return false;
        }

        int reversedHalf = 0;

        // We only need to reverse half of the number to compare with the other half.
        // When the original number is less than the reversed number, it means we've processed half of the digits.
        while (number > reversedHalf) {
            // Extract the last digit of the number and move it to the tens place of the reversed half.
            reversedHalf = reversedHalf * 10 + number % 10;
            // Drop the last digit from the original number.
            number /= 10;
        }

        // At the end of the loop, we have two cases:
        // 1. The length of the number is odd, and we need to discard the middle digit by reversedHalf / 10.
        // 2. The length of the number is even, and the reversed half should be equal to the number.
        return number == reversedHalf || number == reversedHalf / 10;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to check if a number is a palindrome
    bool isPalindrome(int number) {
        // A negative number cannot be a palindrome.
        // Also, if the last digit is 0, then for the number to be a palindrome,
        // the first digit must also be 0 which means the number is 0.
        if (number < 0 || (number != 0 && number % 10 == 0)) {
            return false;
        }
        int reversedHalf = 0;
        // Reconstruct half of the number by reversing the digits
        while (reversedHalf < number) {
            // Extract the last digit of 'number' and add it to 'reversedHalf'
            reversedHalf = reversedHalf * 10 + number % 10;
            // Remove the last digit from 'number'
            number /= 10;
        }
        // Check if the original 'number' and 'reversedHalf' are the same
        // for odd-digit numbers, the middle digit is discarded by 'reversedHalf / 10'
        return number == reversedHalf || number == reversedHalf / 10;
    }
};
```

Typescript Solution

```typescript
// This function checks if a given number is a palindrome.
// A palindrome number is the same forward and backward, e.g., 121.
function isPalindrome(x: number): boolean {
    // A negative number or a number that ends in 0 (but is not 0 itself) cannot be a palindrome.
    if (x < 0 || (x > 0 && x % 10 === 0)) {
        return false;
    }
    // Initialize reversed half of the number to 0.
    let reversedHalf = 0;
    // Reverse the second half of the number and compare with the first half.
    // Stop when the reversed half is greater or equal to the remaining half.
    while (reversedHalf < x) {
        // Get the last digit of the number and add it to the reversed half.
        reversedHalf = reversedHalf * 10 + (x % 10);
        // Remove the last digit from the original number.
        x = Math.floor(x / 10);
    }
    // Check if the number is a palindrome considering both even and odd lengths.
    return x === reversedHalf || x === Math.floor(reversedHalf / 10);
}
```

Time and Space Complexity

The time complexity of the given code is `O(n)`, where `n` is the number of digits in the input integer `x`. This is because the loop runs until the reversed number `y` is greater than or equal to the input number `x`, which happens after `n/2` iterations in the worst case (when the whole number is a palindrome).

Each iteration involves a constant number of operations: a multiplication, an addition, a modulus operation, and a division, none of which depend on the size of `x`. Therefore, these operations do not affect the linear time complexity with respect to the number of digits in `x`.

The space complexity is `O(1)`, as the amount of extra space used does not grow with the size of the input. The variables `x` and `y` are the only extra space used, and they store integers, which take constant space regardless of the length of the integer `x`.