Problem Name: Palindrome Number

Link: https://leetcode.com/problems/palindrome-number/

Solution Language(s): Java

Initial Approach

In this problem, we are checking if a given integer, *x*, is a palindrome or not. I had two ideas for this challenge right off the bat: one was to use a two pointer approach and check if the characters at the beginning and end matched, and then narrowing the pointers until they met or overlapped.

Initial Solution

```
class Solution {
    public boolean isPalindrome(int x) {
        String y = Integer.toString(x);
        int start = 0;
        int end = y.length() - 1;
        while (start < end) {
            if (y.charAt(start) != (y.charAt(end))) {
                return false;
            }
            start++;
            end--;
        }
        return true;
    }
}
```

Time Complexity: O(n)

The time complexity in this approach is O(n) since you are checking up to *n* characters, where *n* is the number of digits in *x*.

Space Complexity: O(n)

The space complexity of the two pointer approach is O(n) due to the fact that storing the string representation of *x* takes O(n), where n is the number of digits in *x*.

Final Approach

My other approach idea was to create the reversed integer and then check if the reversed and initial integers were equal to one another. I start with a negative check in order to speed up the runtime of the algorithm, as no negative number can be a palindrome. Secondly, I use a long to represent reversed because the reversed integer *x* may be larger than the integer range.

Final Solution

```
class Solution {
    public boolean isPalindrome(int x) {
        if (x < 0) {
            return false;
        }
        long reversed = 0;
        int y = x;
        while (y != 0) {
            reversed = reversed * 10 + (y % 10);
            y /= 10;
        }
        if (reversed == x) {
            return true;
        } else {
            return false;
        }
    }
}
```

Time Complexity: O(n)

The time complexity in this approach is O(n), where *n* is the number of digits in the integer *x*.

Space Complexity: O(1)

The space complexity of this method is simply O(1) as it is constant, with just one reversed integer that is created.

Conclusion

In conclusion, this problem had a number of solutions. The first two ideas I had both had an optimal time complexity of O(n), where n is the number of digits in the parameter *x*. We can see that in the first solution I used a string representation of *x*, while in the final solution I managed to solve the problem with only numeric variables. The second solution also