246. Strobogrammatic Number

Easy Hash Table Two Pointers String

Leetcode Link

Problem Description

The problem requires us to determine if a given string `num` represents a strobogrammatic number. A strobogrammatic number is one that appears the same when flipped 180 degrees. Imagine looking at certain numbers on a digital clock; when rotated upside down, they still read as valid numbers. Notably, '6' becomes '9', '9' becomes '6', '8' remains '8', '1' remains '1', and '0' remains '0'. Numbers like '2', '3', '4', '5', and '7' do not form valid numbers when flipped, so they cannot be part of a strobogrammatic number.

The goal is to return `true` if the number represented by the string `num` is strobogrammatic and `false` otherwise. It is important to consider how the number looks when each of its digits is rotated, and the entire string needs to be a valid number after the rotation.

Intuition

To determine if a number is strobogrammatic, we can map each digit to its corresponding digit when flipped 180 degrees. We initialize a list, `d`, where the index corresponds to a digit and the value at that index corresponds to what the digit would look like after being flipped 180 degrees. In the list `d`, a value of `-1` means the digit does not form a valid number when flipped (these are the numbers that cannot contribute to a strobogrammatic number).

A strobogrammatic number should be symmetrical around its center. Therefore, we only need to compare the first half of the string to the second half. We set up two pointers, `i` starting at the beginning of the string and `j` at the end, and move them towards the center. At each step we:

1. Convert the digits at indices `i` and `j` to integers `a` and `b`.
2. Check whether the strobogrammatic counterpart of `a` is equal to `b`. If it's not, we immediately return `false` since the number is not strobogrammatic.
3. Move the pointers inward, `i` going up and `j` going down.

If we successfully traverse the string without mismatches, then the number is strobogrammatic and we return `true`.

The solution is elegant because it only requires a single pass, giving us an efficient way to solve the problem with O(n) complexity, where n is the length of the string `num`.

Solution Approach

The provided Python implementation uses a simple approach leveraging a list to check for strobogrammatic pairs and two-pointer technique to validate symmetry.

Below is a step-by-step explanation of the algorithm:

1. **Data Structure**: We use a list called `d` where each index `i` corresponds to the digit `i` in integer form, and the value at `d[i]` represents what the digit would look like if flipped 180 degrees. For digits that are invalid upon flipping (2, 3, 4, 5, 7), we assign a value of `-1`.

2. **Two-Pointer Technique**: To validate that the `num` is strobogrammatic, we use two pointers, `i` and `j`. The pointer `i` starts at the beginning of the string `num` (index 0), and `j` starts at the end of the string (index `len(num) - 1`).

3. **Iteration and Validation**: The algorithm iterates over the string by moving `i` from the start toward the end, and `j` from the end toward the start. At each iteration, it:

   - Converts characters at current indices `i` and `j` to integers `a` and `b`, respectively.
   - Uses `d[a]` to obtain the flipped counterpart of `a`.
   - If `d[a]` does not equal `b`, then `num` cannot be strobogrammatic, hence the function returns `false`.

4. **Termination Condition**: The iteration stops when the pointers `i` and `j` meet or cross each other (`i > j`), indicating the entire string has been checked.

5. **Returning the Result**: If the function hasn't returned `false` by the end of the iteration, it means `num` is strobogrammatic and the function returns `true`.

This algorithm employs an array to emulate a direct mapping, which offers an efficient lookup time of O(1) for each digit's strobogrammatic counterpart, and it runs in linear time relative to the length of the string `num`, resulting in O(n) time complexity. The space complexity is constant O(1), only requiring storage for the list `d`, which has a fixed length of 10, and the two index variables `i` and `j`.

Example Walkthrough

Let's illustrate the solution approach using a small example where `num` is `"69"`. We want to determine if this string represents a strobogrammatic number.

1. **Data Structure Initialization**: According to the algorithm, we set up a list `d` mapping digits to their flipped counterparts:

   ```
   d = [-1, 1, -1, -1, -1, -1, 9, -1, 8, 6]
   ```

2. **Initializing Pointers**: We have two pointers `i` and `j`. In our case, `i` starts at index `0` and `j` starts at index `1` (since the string `"69"` has a length of 2).

3. **Iteration and Validation**:

   - For the first iteration (`i = 0`, `j = 1`):
     - We convert the character at index `i` to an integer `a` (`a = 6`).
     - We convert the character at index `j` to an integer `b` (`b = 9`).
     - We then use `d[a]` to find the flipped counterpart of `a` (`d[6] = 9`).
     - We check if `d[a]` equals `b`. In this case, `d[6]` (which is `9`) is equal to `b` (which is also `9`), so we proceed.
   - We then move the pointers inward: `i` goes up to `1` and `j` goes down to `0`.
   - Since `i` now meets `j`, our loop terminates.

4. **Termination Condition**:

   - The pointers `i` and `j` have met, indicating we've checked all necessary digits.

5. **Returning the Result**:

   - Since there were no mismatches during the iteration, we conclude that the string `"69"` is strobogrammatic.
   - The function would return `true`.

By following this procedure with the given example, we demonstrate how the algorithm successfully identifies that the string `"69"` is a strobogrammatic number using the two-pointer technique and a fixed mapping list for valid strobogrammatic pairs.

Python Solution

```python
class Solution:
    def isStrobogrammatic(self, num: str) -> bool:
        # Mapping of strobogrammatic numerals where the key is the numeral as int
        # and the value is its 180-degree rotated equivalent (also as an int).
        # Any numeral that doesn't have a strobogrammatic equivalent is mapped to -1.
        rotated_numerals = [0, 1, -1, -1, -1, -1, 9, -1, 8, 6]

        left, right = 0, len(num) - 1  # Pointers to traverse from both ends of the string.

        # Loop to check the strobogrammatic property of the number.
        while left <= right:
            # Convert the left and right pointed numerals in the string to integers.
            left_numeral, right_numeral = int(num[left]), int(num[right])

            # If the rotated numeral of left doesn't match the right numeral,
            # or if the numeral at the current position doesn't have a valid rotation,
            # then the number is not strobogrammatic.
            if rotated_numerals[left_numeral] != right_numeral:
                return False

            # Move the pointers closer to the center.
            left += 1
            right -= 1

        # The entire number has been checked and is strobogrammatic.
        return True
```

Java Solution

```java
class Solution {

    /**
     * Checks if a number is strobogrammatic.
     * A number is strobogrammatic if it looks the same when rotated 180 degrees.
     *
     * @param num the string representing the number to check
     * @return true if the number is strobogrammatic, false otherwise
     */
    public boolean isStrobogrammatic(String num) {
        // An array to represent the 180-degree rotation mapping. For digits that don't
        // have a valid rotation (2,3,4,5,7), we set the value to -2.
        int[] digitMapping = new int[] {0, 1, -1, -1, -1, -1, 9, -1, 8, 6};

        // Two pointers approach - starting from the first and last digit of the string.
        for (int left = 0, right = num.length() - 1; left <= right; ++left, --right) {
            // Convert the characters at the pointers to their corresponding integer values.
            int digitLeft = num.charAt(left) - '0';
            int digitRight = num.charAt(right) - '0';

            // Check if the rotation of digitLeft is equal to digitRight. If not, return false.
            if (digitMapping[digitLeft] != digitRight) {
                return false;
            }
        }
        // If all pair of digits satisfy the strobogrammatic condition, return true.
        return true;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function checks if a given number is strobogrammatic
    bool isStrobogrammatic(string num) {
        // Define a map from digit to its strobogrammatic counterpart
        vector<int> digit_map = {0, 1, -1, -1, -1, -1, 9, -1, 8, 6};

        // Initialize two pointers, one starting from the beginning (left) and the other from the end (right) of the string
        int left = 0, right = num.size() - 1;

        // Loop through the string with both pointers moving towards the center
        while (left <= right) {
            // Convert characters to their corresponding integer values
            int left_digit = num[left] - '0';
            int right_digit = num[right] - '0';

            // Check if the current digit has a valid strobogrammatic counterpart
            // and whether it matches the counterpart of its mirror position
            if (digit_map[left_digit] != right_digit) {
                // If not, then the number isn't strobogrammatic
                return false;
            }

            // Move the pointers towards the center
            ++left;
            --right;
        }

        // All checks passed, the number is strobogrammatic
        return true;
    }
};
```

Typescript Solution

```typescript
// Function checks if a given number is strobogrammatic
function isStrobogrammatic(num: string): boolean {
    // Define an array from digit to its strobogrammatic counterpart
    const digitMap: number[] = [0, 1, -1, -1, -1, -1, 9, -1, 8, 6];

    // Initialize two pointers, one starting from the beginning (left) and the other
    // from the end (right) of the string
    let left: number = 0,
    let right: number = num.length - 1;

    // Loop through the string with both pointers moving towards the center
    while (left <= right) {
        // Convert characters to their corresponding integer values
        const leftDigit: number = parseInt(num[left], 10);
        const rightDigit: number = parseInt(num[right], 10);

        // Check if the current digit has a strobogrammatic counterpart
        // and whether it matches the counterpart of its mirror position
        if (digitMap[leftDigit] !== rightDigit) {
            // If not, the number isn't strobogrammatic
            return false;
        }

        // Move the pointers towards the center
        left++;
        right--;
    }

    // All checks passed, the number is strobogrammatic
    return true;
}
```

Time and Space Complexity

Time Complexity

The given Python function `isStrobogrammatic` checks if a number is strobogrammatic, which means the number looks the same when rotated 180 degrees.

To analyze the time complexity, we consider the number of times the while loop runs with respect to the length of the string `num`. The loop runs until the pointers `i` (starting from the beginning) and `j` (starting from the end) meet or cross each other. For a string of length `n`, this loop will run approximately `n/2` times, since each iteration checks two digits (one at the beginning, one at the end).

Therefore, the time complexity of the code is `O(n/2)`, which simplifies to `O(n)`, where `n` is the length of the input string `num`.

Space Complexity

The space complexity involves analyzing the additional space used by the algorithm, excluding the input itself. In this function, the space used includes:

- A list `d` of length 10, containing the strobogrammatic equivalents.
- Two integer variables `i` and `j`.
- Two temporary integer variables `a` and `b` in the loop.

The space taken by the list `d` is constant, regardless of the length of the input string. Therefore, it doesn't scale with `n`. Similarly, the variables `i`, `j`, `a`, and `b` are a fixed number of additional space used, regardless of the size of the input.

Thus, the space complexity of the function is `O(1)`, which denotes constant space complexity.