247. Strobogrammatic Number II

Medium Topics Array String

Problem Description

The task is to generate all strobogrammatic numbers of a given length n. A number is considered strobogrammatic if it looks the same when rotated 180 degrees. This means that when you flip the number upside down, it should appear identical to its original appearance. To understand this better, imagine how the numbers would look on a seven-segment display; the numbers 0, 1, and 8 look the same when flipped, while 6 and 9 look like each other when turned upside down.

It's important to note that only the digits 0, 1, 6, 8, and 9 can be used to form strobogrammatic numbers, and when forming numbers of more than 1 digit, the number cannot start with 0 (since that would not be considered a valid number format).

When presented with an integer n, we are required to return a list of strings, where each string represents a strobogrammatic number of length n. The ordering of the output numbers is not important, meaning any order in the returned list is acceptable.

Intuition

The solution to this problem can be visualized as a recursive process where we build the strobogrammatic numbers from the inside out. At each step of the recursion, we're focused on adding a layer of digits to the current number.

When the length n we are building is 0, the base case is reached, and we return an empty list since there can't be a number with zero digits. Similarly, when n is 1, the possible strobogrammatic numbers are those that consist of only a single digit that looks the same when flipped, namely 0, 1, and 8.

For lengths greater than 1, we need to form numbers by placing strobogrammatic pairs at the beginning and end of the numbers we have formed so far. The pairs are (1, 1), (8, 8), (6, 9), and (9, 6), since these are the digits that maintain their strobogrammatic property when placed on opposite ends.

The recursive function `dfs(u)` deals with the construction of these numbers. For any given u, it reduces the problem to finding strobogrammatic numbers of length $u - 2$ and adds the strobogrammatic pairs around them, effectively increasing their length by 2.

An additional rule is set for the outermost layer when u is equal to the original n. We do not add the pair (0, 0) as that would lead to numbers starting with 0, which is not allowed. So, the $0 + v + 0$ combination is only added when we're not at the outermost layer.

Using this recursive strategy, when we reach the full length n, we will have created all possible combinations of strobogrammatic numbers. The function `dfs(n)` initiates this process and the list returned from this function is the final answer.

Solution Approach

The provided solution approach uses a depth-first search (DFS) as the primary algorithm for constructing strobogrammatic numbers. It leverages recursion to build the numbers layer by layer, beginning from the middle and expanding outwards. The fundamental data structure used is a list to collect and return the possible strobogrammatic numbers.

Recursive Strategy

The `dfs` function is at the core of the strategy and is recursive in nature. The parameter u indicates the current length of the strobogrammatic number being built.

- **Base Cases:**
  - When u equals 0, the function returns a list containing an empty string `['']`, which symbolically represents the middle of the strobogrammatic number.
  - When u equals 1, the function returns the list `['0', '1', '8']`, as these are the only digits that remain unchanged when rotated 180 degrees individually.
- **Recursive Step:** For any u greater than 1, `dfs(u)` calls itself with $u - 2$, aiming to get the strobogrammatic numbers of a smaller length that will form the middle part of the new, larger numbers.

  During the recursive step, the function iterates over each string v returned by the `dfs(u - 2)` call. It then adds each strobogrammatic pair (`'11'`, `'88'`, `'69'`, `'96'`) around v to the `ans` list, thus forming a new, longer strobogrammatic number. This process is how the solution builds the numbers from the inside out.

  The condition `if u != n` ensures that we do not add the pair `'00'` at the outermost layer when constructing numbers of length n. Zeros are added in the middle layers because they are allowed there but not as the leading digit of the entire number.
- **Efficient Construction:** By using the above pairs and ensuring that `'0'` is not added as the outermost digits, the algorithm efficiently constructs the correct strobogrammatic numbers while maintaining the necessary constraints.

Backtracking

A subtle form of backtracking is exhibited in this DFS function. When the function appends a new pair to a smaller strobogrammatic number, it explores that path entirely, and upon completion, it backtracks to try the next pair. This way, it explores all possible number formations of length n.

Conclusion

The recursive building process and intelligent pair placement combine to ensure each constructed number is strobogrammatic. This approach is effective because it expands upon smaller already-valid numbers, ensuring that the overall structure of the number retains the strobogrammatic property.

Once the recursion unwinds back to the initial call with `dfs(n)`, all strobogrammatic numbers of length n are returned, thus solving the problem.

Example Walkthrough

Let's say we want to find all strobogrammatic numbers of length $n = 2$. We can execute the solution approach to see how it works.

1. We call the recursive function `dfs(2)`.
2. Since $n = 2$ is not equal to 0 or 1, we proceed with the recursive step and call `dfs(0)` to find strobogrammatic numbers of length $n - 2$.
3. `dfs(0)` returns `['']` since the base case $n = 0$ simply means the middle of the strobogrammatic number, which is the empty string in this case.
4. Now, with $u = 2$, we need to surround this empty string with strobogrammatic pairs to form numbers of length 2.
5. We iterate through each strobogrammatic pair (`'11'`, `'88'`, `'69'`, `'96'`) and surround the string from `dfs(0)`, which is the middle part. We don't add `'00'` because $u = n$ and we can't start with a zero.
   - Add `'11'` to the empty string: `'' + '11' + ''`
   - Add `'88'` to the empty string: `'' + '88' + ''`
   - Add `'69'` to the empty string: `'' + '69' + ''`
   - Add `'96'` to the empty string: `'' + '96' + ''`
6. The `ans` list is now `['11', '88', '69', '96']`.
7. Since we are now done with adding pairs to the length 2 strobogrammatic number and $u = n$, we conclude the function and return `['11', '88', '69', '96']` as the final result.

As we can see from this example, the recursive function builds the strobogrammatic numbers from the center outwards, and only pairs that satisfy the strobogrammatic property are added in each step. For numbers with lengths greater than 2, this process continues inward, building up from shorter strobogrammatic strings to the desired length.

This walkthrough exemplifies the mechanics of the provided recursive DFS solution in constructing valid strobogrammatic numbers of a specified length.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def findStrobogrammatic(self, n: int) -> List[str]:
        # Helper function to perform depth-first search.
        def dfs(length):
            # Base case for recursion: when no more characters can be placed.
            if length == 0:
                return ['']
            # When we have space for one character, only 0, 1, and 8 are strobogrammatic by themselves.
            if length == 1:
                return ['0', '1', '8']

            # Recurse with two fewer spaces to place the rest.
            subresults = dfs(length - 2)
            full_results = []
            # Iterate through each subresult from the recursive call.
            for subresult in subresults:
                # Add strobogrammatic pairs around the subresult.
                for left, right in ['11', '88', '69', '96']:
                    full_results.append(left + subresult + right)
                # If we are not at the outermost layer, we can use '0' as a strobogrammatic pair as well.
                if length != n:
                    full_results.append('0' + subresult + '0')
            return full_results

        # Start the recursion with the full length n.
        return dfs(n)
```

Java

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

class Solution {
    // Pairs of digits that are strobogrammatic, i.e. they look the same when rotated 180 degrees
    private final List[] STROBOGRAMMATIC_PAIRS = {{1, 1}, {8, 8}, {6, 9}, {9, 6}};
    private int n;

    // Main method to find all strobogrammatic numbers of length n
    public List<String> findStrobogrammatic(int n) {
        this.n = n;
        return buildStrobogrammaticNumbers(n);
    }

    // Helper method to recursively build strobogrammatic numbers of the current length
    private List<String> buildStrobogrammaticNumbers(int currentLength) {
        // Base case for length 0: If current length is 0, return a list with an empty string
        if (currentLength == 0) {
            return Collections.singletonList("");
        }

        // Base case for length 1: Only digits 0, 1, and 8 are strobogrammatic, return them as strings
        if (currentLength == 1) {
            return Arrays.asList("0", "1", "8");
        }

        // List to store the strobogrammatic numbers of the current call
        List<String> strobogrammaticNumbers = new ArrayList<>();

        // Get the strobogrammatic numbers for (currentLength - 2)
        List<String> subNumbers = buildStrobogrammaticNumbers(currentLength - 2);

        // Loop over each strobogrammatic number of smaller length
        for (String number : subNumbers) {
            for (int[] pair : STROBOGRAMMATIC_PAIRS) { // Loop over each strobogrammatic pair
                // Surround the smaller-length number with a strobogrammatic pair
                strobogrammaticNumbers.add(pair[0] + number + pair[1]);
            }

            // If we are not building the outermost layer, add '0's to the list as well
            // This is because '0's cannot be at the beginning and end of a strobogrammatic number
            if (currentLength != n) {
                strobogrammaticNumbers.add("0" + number + "0");
            }
        }

        return strobogrammaticNumbers;
    }
}
```

C++

```cpp
#include <vector>
#include <string>
#include <functional>
using namespace std;

class Solution {
public:
    // Define pairs of strobogrammatic numbers where each pair is symmetrical around the vertical axis.
    const vector<pair<char, char>> kPairs = {{'1', '1'}, {'8', '8'}, {'6', '9'}, {'9', '6'}};

    // Main method to find all strobogrammatic numbers of length n.
    vector<string> findStrobogrammatic(int n) {
        // Define a recursive function using lambda that takes the length of the number to be constructed.
        function<vector<string>(int)> findNumbersOfLength = [&](int length) {
            // Base case for recursion: when the length is 0, return an empty string.
            if (length == 0) return vector<string>{""};
            // If the length is 1, the strobogrammatic numbers are one of these characters.
            if (length == 1) return vector<string>{"0", "1", "8"};

            // Recursive call, which reduces the length of the number by two.
            vector<string> ans; // This will store the resulting strobogrammatic numbers.
            for (auto& v : findNumbersOfLength(length - 2)) {
                // Add each pair of numbers to both ends of each string returned for length-2.
                for (auto& [left, right] : kPairs) ans.push_back(left + v + right);
                // If we are not at the outermost level of the original number, add '0' to both ends as well.
                // Adding '0' at the beginning or end of the number, hence it is only added when length != n.
                if (length != n) ans.push_back('0' + v + '0');
            }
            // Return the constructed list of strobogrammatic numbers.
            return ans;
        };

        // Start the recursive function with the initial length n.
        return dfs(n);
    }
};
```

TypeScript

```typescript
// Import the necessary modules for utility functions
import { Vector, Pair } from "./path_to_utility"; // Replace with the correct path to utility modules if available

// Define pairs of strobogrammatic numbers where each pair is symmetrical around the vertical axis.
const kPairs: Array<Pair<char, char>> = [['1', '1'], ['8', '8'], ['6', '9'], ['9', '6']];

// Define a recursive function using lambda that takes the length of the number to be constructed.
const findStrobogrammaticNumbers = (length: number, initialLength: number): Array<string> => {
  // Base case for recursion: when the length is 0, return an empty string.
  if (length === 0) return [""];
  // If the length is 1, the strobogrammatic numbers can only be one of these characters.
  if (length === 1) return ["0", "1", "8"];

  // Recursive call, which reduces the length of the number by two.
  const results: Array<string> = [];
  for (const v of findStrobogrammaticNumbers(length - 2, initialLength)) {
    // Add each pair of numbers to both ends of each string returned for length-2.
    for (const [left, right] of kPairs) results.push(left + v + right);
    // If we are not at the outermost level of the original number, add '0' to both ends as well.
    // "0" cannot be at the beginning or end of the number, hence it is only added when length != initialLength.
    if (length !== initialLength) results.push('0' + v + '0');
  }
  // Return the constructed list of strobogrammatic numbers.
  return results;
};

// Main method to find all strobogrammatic numbers of length n.
const findStrobogrammatic = (n: number): Array<string> => {
  // Start the recursive function with the initial length n.
  return findStrobogrammaticNumbers(n, n);
};

// Note that helper types like Pair and Vector are not native TypeScript types and would typically not be necessary.
// They are used here as stand-ins for whatever utility types you have to represent a pair and a list.
// If those utilities are not available, TypeScript's native types (e.g., tuples and arrays) can directly be used.
```

Python

```python
from typing import List

class Solution:
    def findStrobogrammatic(self, n: int) -> List[str]:
        # Helper function to perform depth-first search.
        def dfs(length):
            # Base case for recursion: when no more characters can be placed.
            if length == 0:
                return ['']
            # When we have space for one character, only 0, 1, and 8 are strobogrammatic by themselves.
            if length == 1:
                return ['0', '1', '8']

            # Recurse with two fewer spaces to place the rest.
            subresults = dfs(length - 2)
            full_results = []
            # Iterate through each subresult from the recursive call.
            for subresult in subresults:
                # Add strobogrammatic pairs around the subresult.
                for left, right in ['11', '88', '69', '96']:
                    full_results.append(left + subresult + right)
                # If we are not at the outermost layer, we can use '0' as a strobogrammatic pair as well.
                if length != n:
                    full_results.append('0' + subresult + '0')
            return full_results

        # Start the recursion with the full length n.
        return dfs(n)
```

Time and Space Complexity

The given Python code generates all strobogrammatic numbers of length n. Here is an analysis of its time complexity and space complexity:

- **Time Complexity:**

  We can analyze the time complexity by considering the number of recursive calls and the work done in each call. The `dfs` function is called recursively with an argument reduced by 2 in each call until it reaches 0 or 1.

  For each recursive call to `dfs(u - 2)`, we iterate over the results and append four different pairs to each of the strings (three pairs if u is the original n to avoid leading zeros). The number of such string concatenations doubles with every additional recursive pair (`'00'`, `'11'`, `'88'`, `'69'`, `'96'`), minus the case `'00'` at the top level to prevent leading zeros in the final numbers.

  If we start with $n = 0$ or $n = 1$, the function ends after one call. However, for $n > 1$, every two levels of the recursion approximately double the results.

  Therefore, the time complexity can be approximated as $O(5^{n/2})$.

- **Space Complexity:**

  Space complexity is driven by two factors: the space to store the intermediate results and the depth of the recursive call stack.

  For the intermediate results, the maximum space is used when the final list of strobogrammatic numbers is generated. Similar to time complexity, the space usage doubles for every additional level. Since we are generating all possible combinations, the space used to store these combinations will be proportional to the number of combinations, which is $O(5^{n/2})$.

  Meanwhile, the recursive call stack will have a depth of at most $n/2$ (since we reduce n by 2 with each recursive call). Each recursive call requires a constant amount of space, so the total space used by the call stack will be $O(n)$.

  Thus, when accounting for both factors, the space complexity is dominated by the storage for the list of strobogrammatic numbers. Hence, the space complexity is $O(5^{n/2})$.