1415. The k-th Lexicographical String of All Happy Strings of Length n

Medium String Backtracking

Problem Description

The problem requires us to define a "happy string" as a string that fulfills two conditions: (1) it is composed only of the characters 'a', 'b', and 'c', and (2) it does not have any consecutive identical characters. Examples of happy strings are "abc", "ac", and "b". The challenge is to generate a list of all possible happy strings of a specified length `n`, sort this list in lexicographical (dictionary) order, and then find the `k`th string in this sorted list. If the number of happy strings of length `n` is less than `k`, then we must return an empty string, indicating that the `k`th happy string does not exist.

Intuition

To solve the problem, the intuition is to use depth-first search (DFS) to generate all possible happy strings. DFS is effective here since it allows us to explore all possible character combinations for the strings of length `n` by appending one character at a time and only continuing from those combinations that remain happy (i.e., no consecutive characters are the same).

The key to the DFS approach is that once we've appended a character to the string, all subsequent choices must differ from the last character added. If we reach a length of `n`, that means we've constructed a valid happy string, which we then add to our list of answers.

We initiate DFS with an empty string and explore every possibility by adding 'a', 'b', or 'c', keeping in mind the last character we appended. We continue this process recursively until we either reach the string length `n` or there are no valid characters left to append.

Once we've generated all happy strings, we check if the total count is less than `k`. If it is, this means there is no `k`th happy string, and we should return an empty string. Otherwise, we return the string that is at the `k-1` index in the list (as arrays are 0-indexed in Python, and our requirement is 1-indexed).

Solution Approach

The solution uses a recursive Depth-First Search (DFS) algorithm to build up the "happy strings" one character at a time. Here's how the solution works, breaking it down to the key components:

- A helper function `dfs(t)` is defined to perform the DFS on the current string `t`. The core idea is to keep appending characters to `t` until it reaches the desired length `n`.
- Within the `dfs` function, we first check if the length of `t` has reached `n`. If it has, we add `t` to the `ans` list. This signifies that we have successfully created a happy string of the required length.
- For each recursive call to `dfs`, we iterate through the characters 'a', 'b', and 'c'. For each character `c`, if the last character of `t` (if `t` is not empty) is the same as `c`, we skip to the next iteration, as we cannot have repeating characters.
- If the last character is not the same as `c`, or if `t` is empty, we make a recursive call to `dfs` with `t + c`, effectively probing this path in the search space.
- The recursion continues until we either reach strings of length `n` or exhaust all possibilities of constructing a happy string from the current substring.
- The `ans` list is a data structure used to store all the happy strings we discover while performing DFS. It is declared outside the `dfs` function to retain its value across multiple recursive calls.
- Once the DFS is complete, we look at the `ans` list. If the length of `ans` is less than `k`, implying there aren't enough happy strings, we return an empty string. Otherwise, we return the `k-1`th string in the list, since lists in Python have zero-based indexing.

The DFS approach is efficient for this problem because it systematically explores all valid combinations without constructing invalid strings. The solution is also space-optimized since it only saves the valid happy strings found and does not store any intermediate invalid states.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach using `n = 3` and `k = 5`. We are to generate all possible happy strings of length 3 and find the 5th string in lexicographical order.

1. We start with an empty string, `s`, and initiate our recursive `dfs(t)` function.
2. In the first level of the recursive tree, we have three choices for the first character: 'a', 'b', or 'c'. Let's start with `t = "a"`.
3. For the second character, we have two choices ('b' or 'c') since it can't be identical to the last character 'a'. We thus explore `t = "ab"` and `t = "ac"`.
4. Assuming we now have `t = "ab"`, for the third character we again have two choices ('a' or 'c'). This yields `t = "aba"` and `t = "abc"`.
5. We repeat the same process starting from `t = "ac"` and get `t = "aca"` and `t = "acb"`. Since we are building up in lexicographical order, we can see that "abc" comes before "acb". Therefore, "acb" will be considered later.
6. Continuing this process, starting with `t = "b"` and then `t = "c"`, and ensuring no consecutive characters are the same, we construct the following strings: `["aba", "abc", "aca", "acb", "bac", "bca", "cab", "cba"]`.
7. We have generated all possible happy strings of length 3. Now we sort them (although they were constructed in sorted order due to our approach): `["aba", "abc", "aca", "acb", "bac", "bca", "cab", "cba"]`.
8. To find the `k = 5`th string, we look at index `k-1 = 4` in the list (as the indexing starts from 0), which gives us "bac".

Here, the 5th happy string of length 3 in lexicographical order is "bac". If the value of `k` had been greater than the number of happy strings generated, our function would return an empty string, indicating that there's no valid answer.

Solution Implementation

Python

```python
class Solution:
    def getHappyString(self, length: int, index: int) -> str:
        # Helper function for depth-first search to find all happy strings
        def dfs(current_str):
            # Base condition: if current string reaches the desired length, add to results
            if len(current_str) == length:
                happy_strings.append(current_str)
                return
            # Iterate through all possible characters
            for char in 'abc':
                # Avoid repeating characters
                if current_str and current_str[-1] == char:
                    continue
                # Recursively build the string
                dfs(current_str + char)

        # List to store all possible happy strings
        happy_strings = []
        # Start the DFS with an empty string
        dfs('')
        # If there are fewer than k happy strings, return an empty string
        # Otherwise, return the k-th happy string (1-indexed)
        return '' if len(happy_strings) < index else happy_strings[index - 1]

# The code has been rewritten using Python 3 syntax.
# Variable names have been standardized for clarity, following Python naming conventions.
# Comments in English have been added to explain the intent of the code.
```

Java

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

class Solution {
    // List to hold all happy strings
    private List<String> happyStrings = new ArrayList<>();

    // Function to return the kth happy string of length n
    public String getHappyString(int n, int k) {
        // Generate all happy strings of length n
        generateHappyStrings("", n);
        // If the list size is less than k, no such k-th happy string exists. Return an empty string.
        return happyStrings.size() < k ? "" : happyStrings.get(k - 1);
    }

    // Helper function to generate happy strings using Depth First Search (DFS)
    private void generateHappyStrings(String current, int n) {
        // If the current string's length is n, add it to the list of happy strings
        if (current.length() == n) {
            happyStrings.add(current);
            return;
        }

        // Loop through the characters 'a', 'b', and 'c'
        for (char c : "abc".toCharArray()) {
            // If the last character of the current string is the same as 'c', skip to the next iteration
            // to ensure we do not have consecutive same characters, which makes the string unhappy
            if (current.length() > 0 && current.charAt(current.length() - 1) == c) {
                continue;
            }
            // Otherwise, add 'c' to current and continue to search for the rest of the string
            generateHappyStrings(current + c, n);
        }
    }
}
```

C++

```cpp
#include <vector>
#include <string>

class Solution {
public:
    // Vector to store the valid "happy strings".
    std::vector<std::string> happyStrings;

    // Main function to return the k-th happy string of length n.
    // If there aren't k happy strings, an empty string is returned.
    std::string getHappyString(int length, int k) {
        // Initiate a depth-first search to generate all happy strings of length 'n'
        generateHappyStrings("", length);
        // Check if there are less than 'k' happy strings, return "" if true.
        // Otherwise, return the k-th string (0-indexed, so we use k-1).
        return happyStrings.size() < k ? "" : happyStrings[k - 1];
    }

private:
    // Helper function to generate happy strings using DFS.
    void generateHappyStrings(std::string current, int length) {
        // If the current string has reached the target length 'n',
        // add it to the list of happy strings and return.
        if (current.size() == length) {
            happyStrings.push_back(current);
            return;
        }
        // Iterate over each character from 'a' to 'c'.
        for (char nextChar = 'a'; nextChar <= 'c'; ++nextChar) {
            // If the last character is the same as the next character,
            // skip to maintain the "happy" property.
            if (!current.empty() && current.back() == nextChar) continue;
            // Append the new character and continue the search.
            current.push_back(nextChar);
            generateHappyStrings(current, length);
            // Backtrack by removing the last character added.
            current.pop_back();
        }
    }
};
```

TypeScript

```typescript
// Array to store the valid "happy strings".
let happyStrings: string[] = [];

// Main function to return the k-th happy string of length n.
// If there aren't k happy strings, an empty string is returned.
function getHappyString(length: number, k: number): string {
    // Initiate a depth-first search to generate all happy strings of the given length
    generateHappyStrings("", length);
    // Check if there are fewer than k happy strings, return an empty string if so.
    // Otherwise, return the k-th string (0-indexed, so we use k-1).
    return happyStrings.length < k ? "" : happyStrings[k - 1];
}

// Helper function to generate happy strings using DFS.
function generateHappyStrings(current: string, length: number) {
    // If the current string has reached the target length,
    // add it to the list of happy strings and return.
    if (current.length === length) {
        happyStrings.push(current);
        return;
    }
    // Iterate over each character from 'a' to 'c'.
    for (let nextChar = 'a'.charCodeAt(0); nextChar <= 'c'.charCodeAt(0); nextChar++) {
        let nextCharStr = String.fromCharCode(nextChar);
        // If the last character is the same as the next character,
        // skip to maintain the "happy" property.
        if (current !== "" && current[current.length - 1] === nextCharStr) continue;
        // Append the new character and continue the search.
        generateHappyStrings(current + nextCharStr, length);
        // No need to backtrack explicitly since strings are immutable in JavaScript/TypeScript,
        // and each recursive call gets its own copy of the current string.
    }
}

// Use the functions in your code:
let length = 5; // The desired length for happy strings
let k = 3;      // The position of the happy string to return
let happyString = getHappyString(length, k); // Function call
console.log(happyString); // Log the result

class Solution:
    def getHappyString(self, length: int, index: int) -> str:
        # Helper function for depth-first search to find all happy strings
        def dfs(current_str):
            # Base condition: if current string reaches the desired length, add to results
            if len(current_str) == length:
                happy_strings.append(current_str)
                return
            # Iterate through all possible characters
            for char in 'abc':
                # Avoid repeating characters
                if current_str and current_str[-1] == char:
                    continue
                # Recursively build the string
                dfs(current_str + char)

        # List to store all possible happy strings
        happy_strings = []
        # Start the DFS with an empty string
        dfs('')
        # If there are fewer than k happy strings, return an empty string
        # Otherwise, return the k-th happy string (1-indexed)
        return '' if len(happy_strings) < index else happy_strings[index - 1]

# The code has been rewritten using Python 3 syntax.
# Variable names have been standardized for clarity, following Python naming conventions.
# Comments in English have been added to explain the intent of the code.
```

Time and Space Complexity

The provided Python code defines a class `Solution` with a method `getHappyString` that generates the k-th lexicographically smallest "happy" string of length `n`. A "happy" string is one in which no two adjacent characters are the same.

Time Complexity

The time complexity of the code is O(3*n).

Here's why:

- The algorithm uses a depth-first search (DFS) to generate all possible "happy" strings of length `n`.
- At each step of the DFS, there are at maximum 3 different characters ('a', 'b', 'c') that you can choose from, minus any character that is the same as the last character of the current string (`t`).
- In the worst case, the DFS would generate all possible "happy" strings, and since each position can have 2 choices (after the first character), the number of possibilities is `3 * 2^(n-1)`, which simplifies to `3 * 2 * 2 * ... * 2` (with `n-1` two's in the multiplication), which is O(3 * 2^(n-1)) = O(3*n).

Space Complexity

The space complexity of the code is O(3*n + n).

Here's why:

- `O(3*n)` is for storing the `ans` list, which in the worst case might contain all possible "happy" strings.
- `O(n)` is for the recursion stack depth, which goes as deep as `n` because we're generating strings of length `n`.
- The space used by the `ans` list is the dominating factor, hence the space complexity is mainly dictated by the number of happy strings generated.

In summary, the brute-force DFS approach used in this code might be feasible for small `n`, but is impractical for large `n` due to its exponential time and space complexity.