293. Flip Game

Easy String

Problem Description

In the given problem, we are playing a game called Flip Game with a friend. The game is played with a string `currentState` that can consist of only two characters: `'+'` and `'-'`. In each turn, a player must flip exactly two consecutive `'+'` characters into `'--'` characters. This is the only valid move in the game. The game proceeds with each player taking turns until no more valid moves can be made, at which point the last player to have made a valid move wins.

Our objective is to find all possible states of the `currentState` after one valid move has been made by the current player. We must provide a list of these possible states, or an empty list if no valid moves are available. The resulting states can be returned in any order.

Intuition

The solution to this problem is straightforward and involves iterating through the string to look for pairs of consecutive `'+'` characters because that's the only condition where a flip is possible. Here's the logical approach:

1. Convert the string `currentState` into a list of characters, so we can easily modify individual characters during our iteration.
2. Initialize an empty list `ans` to store the possible next states.
3. Loop through the list starting from index 0 and going to the second to last index (since we are always looking for pairs, we don't need to check the last character by itself).
4. At each iteration, check if the current character and the one following it are both `'+'`.
   - If they are, we flip them to `'-'` by setting them to `'-'`.
   - Then, we take the modified list, convert it back into a string, and add this state to our list of possible next moves `ans`.
   - After this operation, reverting the characters back to `'+'` is important because we are exploring all possible moves from the original state, one at a time.
5. Continue this process until we have checked all possible pairs of `'+'`.
6. Return the `ans` list containing all the possible next states. If no pairs of `'+'` were found, the `ans` list would be empty.

Solution Approach

The solution takes advantage of the linear nature of the `currentState` to perform a single-pass scan. Here is how the algorithm unfolds:

1. **Data Structure**:
   - The function converts the `currentState` string into a list of characters, referred to as `s` in the code, to enable easy modification of individual characters.
   - An empty list, `ans`, is created for storing the possible states after making valid moves.
2. **Algorithm**:
   - The function iterates through the list `s` with a loop, starting from the first character and stops at the second-last character using `enumerate(s[:-1])`. This range is used because the last character cannot start a pair of two `'+'`.
   - At each iteration, the function checks if the current character and the one immediately following it form a pair of consecutive `'+'` using `if c == "+" and s[i + 1] == "+":`.
   - Then it checks for a pair is found:
     - Both characters are flipped from `'+'` to `'-'` by setting `s[i]` and `s[i + 1]` to `'-'`.
     - The updated list `s` is then joined to form a string representing the new state and is added to the `ans` list.
     - Immediately after recording the new state, the characters at index `i` and `i + 1` are reset back to `'+'` to restore the original state before moving to the next iteration. This is crucial as it allows us to explore all possible moves independently.
3. **Return Value**:
   - After the end of the loop, the list `ans` will contain all the unique states that result from performing valid moves on the `currentState`.
   - The function returns the `ans` list. If no valid moves exist, `ans` will be returned as an empty list.

The key pattern used here is **iteration** over the list, combined with simple **character manipulation** to explore different move outcomes. This solution is efficient as it runs in *O(n)* time, where `n` is the length of the `currentState`, and operates with constant space complexity beyond the input and output lists.

Example Walkthrough

Let's use a small example to illustrate the solution approach.

Suppose the `currentState` string is `"++++"`. Our task is to find all possible states after one valid move. A valid move entails flipping two consecutive `'+'` characters into `'--'`.

Following the solution approach:

1. Convert the string into a list of characters: `s = ['+', '+', '+', '+']`
2. Initialize an empty list to store possible next states: `ans = []`
3. Iterate over the list (ignoring the last character as a start since there can't be a pair):
   - Check `s[0]` and `s[1]`: Both are `'+'`, so flip them to get `['-', '-', '+', '+']`, convert back to string `"--++"`, and add to `ans`.
   - Revert `s[0]` and `s[1]` back to `'+'`, so `s` is now `['+', '+', '+', '+']` again.
   - Move to the next pair.
   - Check `s[1]` and `s[2]`: Both are `'+'`, so flip them to get `['+', '-', '-', '+']`, convert back to string `"+--+"`, and add to `ans`.
   - Revert `s[1]` and `s[2]` back to `'+'`.
   - Continue in this fashion.
   - The next pair `s[2]` and `s[3]` are not both `'+'`, so no flip.
   - Check `s[3]` and `s[4]`: Not both `'+'`, so no flip.
   - Finally, check `s[4]` and `s[5]`: Both are `'+'`, so flip to get `['+', '+', '+', '-', '-', '-']`, convert back to string `"+++---"`, and add to `ans`.
4. We're left with `ans` containing the strings: `"--++"`, `"+--+"`, and `"+++---"`.
5. Return `ans`, which is `["--++", "+--+", "+++---"]`.

The players continue the game with each one using the states from `ans` until no more moves can be played. The last move before the game state has no more consecutive `'+'` is the winning move.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def generatePossibleNextMoves(self, currentState: str) -> List[str]:
        # Convert the `currentState` string into a list of characters for manipulation
        state_list = list(currentState)

        # Initialize an empty list to store all the possible next moves
        possible_moves = []

        # Loop through the state_list until the second-to-last character
        for i in range(len(state_list) - 1):
            # Check if two consecutive characters are both '+'
            if state_list[i] == '+' and state_list[i + 1] == '+':
                # Flip them to '-' to generate a new possible move
                state_list[i] = state_list[i + 1] = '-'
                # Convert the updated list back to string and add to our list of possible moves
                possible_moves.append("".join(state_list))
                # Flip the characters back to '+' to reset the state for the next iteration
                state_list[i] = state_list[i + 1] = '+'

        # Return the list of all possible moves
        return possible_moves
```

Java

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

public class Solution {
    // Method to generate all possible next moves by flipping two consecutive '+' to '--'
    public List<String> generatePossibleNextMoves(String currentState) {
        char[] charArray = currentState.toCharArray(); // Convert the string to a character array for easy manipulation
        List<String> possibleMoves = new ArrayList<>(); // List to store all the possible next moves

        // Iterate over the character array to find consecutive '+'
        for (int i = 0; i < charArray.length - 1; ++i) {
            // Check if the current and next character are both '+'
            if (charArray[i] == '+' && charArray[i + 1] == '+') {
                charArray[i] = '-'; // Flip the current '+' to '-'
                charArray[i + 1] = '-'; // Flip the next '+' to '-'

                // Add the new state to the list of possible moves. Convert the character array back to a string.
                possibleMoves.add(new String(charArray));

                // Reset the characters back to '+' for the next iteration
                charArray[i] = '+';
                charArray[i + 1] = '+';
            }
        }
        // Return the list of all possible moves
        return possibleMoves;
    }
}
```

C++

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

class Solution {
public:
    // Function to generate all possible next moves by flipping two consecutive '+' to '--'
    std::vector<std::string> generatePossibleNextMoves(std::string currentState) {
        std::vector<std::string> possibleMoves; // Vector to hold all possible next moves

        // Iterate through the string to find two consecutive '+'
        for (size_t i = 0; i < currentState.size() - 1; ++i) { // Using size_t for index
            // Check if the current and next characters are '+'
            if (currentState[i] == '+' && currentState[i + 1] == '+') {
                currentState[i] = '-';  // Flip the current '+' to '-'
                currentState[i + 1] = '-';  // Flip the next '+' to '-'

                // Add the new state to the list of possible moves
                possibleMoves.push_back(currentState);

                // Flip the characters back to '+' to restore the original state for the next iteration
                currentState[i] = '+';
                currentState[i + 1] = '+';
            }
        }

        // Return all possible moves
        return possibleMoves;
    }
};

// Note: You generally include the necessary header files and use of "std::"
// before standard library constructs is a good practice in C++.
// This ensures that you are using elements from the standard namespace.
```

TypeScript

```typescript
// Function to generate all possible next moves, from flipping two consecutive '+' to '--'
function generatePossibleNextMoves(currentState: string): string[] {
  const possibleMoves: string[] = []; // Array to hold all possible next moves

  // Iterate through the string to find two consecutive '+'
  for (let i = 0; i < currentState.length - 1; ++i) {
    // Check if the current and next characters are '+'
    if (currentState[i] === '+' && currentState[i + 1] === '+') {
      const newState = currentState.substring(0, i) + '--' + currentState.substring(i + 2);
      // Add the new state to the list of possible moves
      possibleMoves.push(newState);
    }
  }

  // Return all possible moves
  return possibleMoves;
}
```

```python
from typing import List

class Solution:
    def generatePossibleNextMoves(self, currentState: str) -> List[str]:
        # Convert the `currentState` string into a list of characters for manipulation
        state_list = list(currentState)

        # Initialize an empty list to store all the possible next moves
        possible_moves = []

        # Loop through the state_list until the second-to-last character
        for i in range(len(state_list) - 1):
            # Check if two consecutive characters are both '+'
            if state_list[i] == '+' and state_list[i + 1] == '+':
                # Flip them to '-' to generate a new possible move
                state_list[i] = state_list[i + 1] = '-'
                # Convert the updated list back to string and add to our list of possible moves
                possible_moves.append("".join(state_list))
                # Flip the characters back to '+' to reset the state for the next iteration
                state_list[i] = state_list[i + 1] = '+'

        # Return the list of all possible moves
        return possible_moves
```

Time and Space Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the input string `currentState`. This is due to the fact that we traverse the string once, with each iteration checking a pair of adjacent characters and modifying the string if a match is found. The replacement operation for a single pair of characters is `O(1)`, therefore the overall time taken is proportional to the length of the string.

The space complexity of the code is also `O(n)`. This is because:

1. We convert the input string into a list, which takes `O(n)` space.
2. A new string is created in each iteration when a valid move is found (the `''.join(s)` operation), and in the worst case, this could happen for every pair of adjacent characters. However, these strings are not all stored in memory at the same time; they are added to the list `ans` and then the original list `s` is restored to its previous state. Since the longest string appended to `ans` has the same length as the input string, `ans` will also take up `O(n)` space.
3. The auxiliary space for the loop and variables is `O(1)` (constant).

It's important to note that, while we do create multiple strings in the process, they are not all in memory at the same time. Each generated result is added to `ans` then the original list `s` is modified back. Nevertheless, in the worst case, where you can flip every pair, you'd have `n/2` results stored in `ans` which constitute `O(n^2)` space if you consider all potential flips together. But considering space complexity in terms of additional space required by the program, we disregard the space taken by the input and output. Hence, we typically consider the space complexity of the additional space required, which remains `O(n)`.