1433. Check If a String Can Break Another String

Medium Greedy String Sorting Leetcode Link

Problem Description

The problem involves determining if there is a permutation of one string (`s1`) that can "break" a permutation of another string (`s2`), or vice-versa. The concept of one string breaking another is defined such that, for each index `i` from 0 to `n-1`, where `n` is the size of the strings, the character from the first string at index `i` is greater than or equal to the character from the second string at the same index when both strings are arranged in alphabetical order.

For example, if `s1` is "abc" and `s2` is "xya", one valid permutation of `s1` that can break `s2` is "cba" (since "cba" ≥ "axy"), and hence the output would be True.

To solve this problem, we need to determine if such permutations exist for `s1` and `s2`.

Intuition

A logical approach to solving this problem is to first sort both strings alphabetically. Once we sort them, we would have the smallest characters located at the start and the largest at the end in each string. If `s1` can break `s2`, after sorting, every character in `s1` should be greater than or equal to the corresponding character in `s2` at each index. Similarly, if `s2` can break `s1`, then every character in `s2` should be greater than or equal to the corresponding character in `s1` at each index.

The intuition behind sorting is that we are aligning characters in the order of their ranks (based on their alphabetical order), and we are doing a one-to-one comparison between the characters of both strings at corresponding positions. If all characters in `s1` are greater than or equal to all corresponding characters in `s2`, then we conclude `s1` can break `s2`. Otherwise, if all characters in `s2` are greater than or equal to all corresponding characters in `s1`, we conclude that `s2` can break `s1`. If neither of these conditions hold, then no permutation can break the other.

The reason we can use sorting and then linear comparison in this problem is because of the transitive property of the "can break" relationship, which states that if `a >= b` and `b >= c`, then `a >= c`. Thus, we can get away with just sorting the strings and comparing them once rather than looking at every possible permutation of both strings, which would be very inefficient.

Solution Approach

The implementation of the solution follows a simple yet efficient approach, utilizing the Python language's native libraries and features. Here's a step-by-step breakdown:

1. **Sort Both Strings**: The `sorted()` function is used to sort `s1` and `s2` alphabetically. This is a critical step that gets the strings ready for comparison. Sorting is done using an efficient sorting algorithm, typically Timsort in Python, which has a complexity of O(n log n), where n is the number of elements in the string.

   ```
   cs1 = sorted(s1)
   cs2 = sorted(s2)
   ```

2. **Compare Sorted Strings**: The `zip()` function is used to aggregate elements from both sorted lists into pairs, which makes it convenient to compare corresponding characters of `cs1` and `cs2`.

3. **Use List Comprehension with `all()` Function**: This approach uses a list comprehension paired with the `all()` function to check the break condition. The `all()` function returns `True` if all elements in an iterable are `True`. There are two conditions checked here:

   - **`s1` can break `s2`**: This is checked by `all(a >= b for a, b in zip(cs1, cs2))`. It compares each corresponding pair of characters between the two sorted strings. If every character `a` from `cs1` is greater than or equal to character `b` from `cs2` for all indexed pairs, then `s1` can indeed break `s2`.

   - **`s2` can break `s1`**: Similarly, `all(a <= b for a, b in zip(cs1, cs2))` checks if `s2` can break `s1`. For this to be true, every character `a` from `cs1` must be less than or equal to character `b` from `cs2`.

4. **Return the Result**: The final return statement combines the two conditions with an `or` operator. If either `s1` can break `s2` or `s2` can break `s1`, it returns `True`; otherwise, it returns `False`.

   ```
   return all(a >= b for a, b in zip(cs1, cs2)) or all(
       a <= b for a, b in zip(cs1, cs2)
   )
   ```

This solution effectively leverages Python's built-in functions and language constructs to produce a concise and readable piece of code. By using sorting and linear comparison, it avoids unnecessary complexity and delivers an optimal solution with a time complexity of O(n log n) due to the sorting step (which is the most time-consuming operation here) and a space complexity of O(n) due to the storage requirements for the sorted lists `cs1` and `cs2`.

Example Walkthrough

Let's take a small example to illustrate the solution approach described above. Suppose we have two strings `s1 = "abe"` and `s2 = "acd"`. We want to determine if a permutation of `s1` can break `s2` or if a permutation of `s2` can break `s1`.

Following the solution steps:

1. **Sort Both Strings**:

   - We sort both strings using Python's `sorted()` function.
   - `cs1` after sorting `s1`: `"abe"` becomes `"abe"`.
   - `cs2` after sorting `s2`: `"acd"` becomes `"acd"`.

   Since both the strings are already in alphabetical order, the sorted versions remain the same.

2. **Compare Sorted Strings**:

   - We use `zip()` to pair up characters from `cs1` and `cs2`.
   - This gives us the following pairs: `('a', 'a')`, `('b', 'c')`, `('e', 'd')`.
3. **Use List Comprehension with `all()` Function**:

   - **`s1` can break `s2`**:

     - We check if all characters in `cs1` are greater than or equal to `cs2` using the first condition of the solution: `all(a >= b for a, b in zip(cs1, cs2))`.
     - Comparing the pairs: `'a' >= 'a'` (True), `'b' >= 'c'` (False), `'e' >= 'd'` (True).
     - Since not all comparisons are `True`, the first condition evaluates to `False`.
   - **`s2` can break `s1`**:

     - Similarly, we check if all characters in `cs2` are greater than or equal to `cs1` using the second condition: `all(a <= b for a, b in zip(cs1, cs2))`.
     - Comparing the pairs: `'a' <= 'a'` (True), `'b' <= 'c'` (True), `'e' <= 'd'` (False).
     - Since not all comparisons are `True`, the second condition also evaluates to `False`.
4. **Return the Result**:

   - The final result is obtained by combining the two conditions with an `or`: `False or False`.
   - Since both conditions are `False`, the final result is `False`.
   - Therefore, for the strings `s1 = "abe"` and `s2 = "acd"`, no permutation of `s1` can break `s2` and no permutation of `s2` can break `s1`.

In this example, we can see how the solution approach methodically determines whether one string can break another by leveraging Python's powerful built-in functions and efficient list comprehension. The result is obtained without having to manually check every possible permutation, thus optimizing the time and space complexity of the solution.

Python Solution

```python
class Solution:
    def checkIfCanBreak(self, s1: str, s2: str) -> bool:
        # Sort both strings to compare them lexicographically
        sorted_s1 = sorted(s1)
        sorted_s2 = sorted(s2)

        # Check if sorted_s1 can "break" sorted_s2. This is true if for every index 'i',
        # the character in sorted_s1 at index 'i' is greater than or equal to the character in sorted_s2 at the same index.
        can_s1_break_s2 = all(char_s1 >= char_s2 for char_s1, char_s2 in zip(sorted_s1, sorted_s2))

        # Check if sorted_s2 can "break" sorted_s1. This is similar to the previous check
        # but in the opposite direction: for every index 'i',
        # the character in sorted_s2 at index 'i' is greater than or equal to the character in sorted_s1.
        can_s2_break_s1 = all(char_s2 >= char_s1 for char_s1, char_s2 in zip(sorted_s1, sorted_s2))

        # Return True if either sorted_s1 can break sorted_s2 or sorted_s2 can break sorted_s1.
        return can_s1_break_s2 or can_s2_break_s1

```

Java Solution

```java
class Solution {

    /**
     * Checks if one of the strings can "break" the other by comparing characters
     * after sorting both strings.
     *
     * @param s1 The first input string.
     * @param s2 The second input string.
     * @return True if one string can break the other, false otherwise.
     */
    public boolean checkIfCanBreak(String s1, String s2) {
        // Convert the strings to character arrays for sorting.
        char[] sortedS1 = s1.toCharArray();
        char[] sortedS2 = s2.toCharArray();

        // Sort the character arrays.
        Arrays.sort(sortedS1);
        Arrays.sort(sortedS2);

        // Check if sortedS1 can break sortedS2 or if sortedS2 can break sortedS1.
        return canBreak(sortedS1, sortedS2) || canBreak(sortedS2, sortedS1);
    }

    /**
     * Helper method to check if the first character array can "break" the second.
     *
     * @param array1 The first character array.
     * @param array2 The second character array.
     * @return True if array1 can break array2, false otherwise.
     */
    private boolean canBreak(char[] array1, char[] array2) {
        // Iterate through the arrays and compare characters.
        for (int i = 0; i < array1.length; ++i) {
            // If any character of array1 is smaller than its counterpart in array2,
            // then array1 cannot break array2.
            if (array1[i] < array2[i]) {
                return false;
            }
        }

        // If all characters in array1 are greater than or equal to their counterparts
        // in array2, array1 can break array2.
        return true;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function that checks if one string can break another after sorting
    bool checkIfCanBreak(string s1, string s2) {
        // Sort both strings
        sort(s1.begin(), s1.end());
        sort(s2.begin(), s2.end());

        // Check if either string can break the other
        return canBreak(s1, s2) || canBreak(s2, s1);
    }

private:
    // Helper function to check if s1 can break s2
    bool canBreak(const string& s1, const string& s2) {
        // Iterate through both strings
        for (int i = 0; i < s1.size(); ++i) {
            // If any character in s1 is less than the character in s2 at the same position,
            // s1 cannot break s2
            if (s1[i] < s2[i]) {
                return false;
            }
        }
        // If all characters in s1 are greater than or equal to those in s2 at the same positions,
        // s1 can break s2
        return true;
    }
};

```

Typescript Solution

```typescript
/**
 * This function checks if one string can "break" another by comparing their sorted characters.
 * A string s1 can break s2 if in all indices i, the character from s1 is greater than or equal to the character from s2, after both
 * @param s1 First input string
 * @param s2 Second input string
 * @returns true if either s1 can break s2 or s2 can break s1, false otherwise
 */
function checkIfCanBreak(s1: string, s2: string): boolean {
    // Convert both strings to arrays of characters and sort them alphabetically
    const sortedChars1: string[] = Array.from(s1).sort();
    const sortedChars2: string[] = Array.from(s2).sort();

    /**
     * This helper function checks if characters of the first array can "break" the second array.
     * @param chars1 Array of sorted characters from the first string
     * @param chars2 Array of sorted characters from the second string
     * @returns true if all characters in chars1 are greater than or equal to chars2, false otherwise
     */
    const canBreak = (chars1: string[], chars2: string[]): boolean => {
        for (let i = 0; i < chars1.length; i++) {
            if (chars1[i] < chars2[i]) {
                return false;
            }
        }
        return true;
    };

    // Return true if either s1 can break s2 or s2 can break s1
    return canBreak(sortedChars1, sortedChars2) || canBreak(sortedChars2, sortedChars1);
}

```

Time and Space Complexity

The given Python code defines a method `checkIfCanBreak`, which takes two strings `s1` and `s2` as input and checks if one string can "break" the other by comparing the sorted versions of both strings. Here's the complexity analysis of the code:

Time Complexity:

The function performs the following operations:

1. Sorts string `s1` - This has a time complexity of `O(n log n)`, where `n` is the length of `s1`.
2. Sorts string `s2` - Similarly, this also has a time complexity of `O(n log n)`, where `n` is the length of `s2` (assuming `s1` and `s2` are of approximately equal length for simplicity).
3. Two calls to the `all()` function with generator expressions containing `zip(cs1, cs2)` - Each call iterates over the zipped lists and compares the elements, which takes `O(n)` time as there are `n` pairs to check.

Assuming `n` is the length of the longer string, the total time complexity should be `2 * O(n log n) + 2 * O(n)`. However, since `O(n)` is dominated by `O(n log n)` in terms of asymptotic complexity, it is ignored in the final complexity expression. Thus, the time complexity of the function is `O(n log n)`.

Space Complexity:

The function uses extra space for the following:

1. Storing the sorted version of `s1` - This takes `O(n)` space.
2. Storing the sorted version of `s2` - This also takes `O(n)` space.

Therefore, the space complexity of the function is `O(n)`, where `n` is the length of the longer string.

In the given code, no additional space complexity is incurred apart from storing the sorted strings since the generator expressions used in `all()` functions don’t create an additional list, but rather computes the expressions on the fly.