1540. Can Convert String in K Moves

Medium Hash Table String

Leetcode Link

Problem Description

In this problem, you are given two strings `s` and `t`, and an integer `k`. The goal is to determine if you can transform string `s` into string `t` by performing up to `k` moves. A move consists of picking an index `j` in string `s` and shifting the character at that index a certain number of times. Each character in `s` may only be shifted once, and the shift operation wraps around the alphabet (e.g., shifting 'z' by 1 results in 'a').

The key constraints are:

- You can make no more than `k` moves in total.
- Each move involves shifting one character of `s` by `i` positions in the alphabet, where `1 <= i <= k`.
- The index chosen for shifting in each move should not have been used in a previous move.
- You can only shift a character to the next one in the alphabet, with 'z' changing to 'a'.

The task is to return `true` if `s` can be transformed into `t` using at most `k` moves under these rules; otherwise, return `false`.

Intuition

The solution approach starts by realizing that in order to change `s` into `t`, for each position where `s` and `t` differ, we need to calculate the number of shifts required to convert the character in `s` to the corresponding character in `t`. This can be done by finding the difference in their ASCII values and taking the modulus with 26 to handle the wrapping around the alphabet.

The next step is to keep track of the number of times each shift amount is needed. We use a list, `cnt`, to record the frequency of each required shift amount from 0 to 25 (since there are 26 letters in the alphabet). A shift amount of 0 means the characters are already the same, and no move is needed.

Once we have this frequency array, we check if any of these shifts can be performed within the move limit `k`. For each non-zero shift amount, we calculate when the last shift can occur. Since each shift amount can be used every 26 moves, we determine the maximum moves needed for each shift (`i + 26 * (cnt[i] - 1)`). If this exceeds `k`, it's impossible to transform `s` into `t`, and we return `false`.

The reason for subtracting 1 from `cnt[i]` is that the first occurrence of each shift amount does not have to wait for a full cycle—it can happen immediately. Hence, only subsequent occurrences (if any) need to be delayed by 26 moves each.

If none of the shifts exceeds the move limit `k`, then it is possible to transform `s` into `t`, and we return `true`.

Solution Approach

The implementation of the solution involves a few key steps that use simple data structures and algorithms.

1. **Length Check**: First, we check if `s` and `t` are of equal length. If not, return `False` immediately because the problem states that only characters in `s` can be shifted, implying both strings must be of the same length to be convertible.

2. **Frequency Array**: We create an array, `cnt`, of length 26 initialized to zero, which will hold the frequency of each shift amount required.

3. **Calculate Shifts**: We loop over the characters of `s` and `t` simultaneously using Python's `zip` function. For each corresponding pair of characters `(a, b)`, we calculate the shift required to turn `a` into `b`. The shift is calculated using the formula `(ord(b) - ord(a) + 26) % 26`, where `ord()` is a Python function that returns the ASCII value of a character. The addition of 26 before the modulus operation is to ensure a positive result for cases where `b` comes before `a` in the alphabet.

4. **Count the Shifts**: We increment the count of the appropriate shift amount in the `cnt` array. If no shift is needed, the increment occurs at `cnt[0]`.

5. **Move Validation**: After calculating all the necessary shifts, we iterate over the `cnt` array (starting from index 1 since index 0 represents no shift). For each shift amount `i`, we find out how late this shift can occur by multiplying the number of full cycles (`cnt[i] - 1`) by the cycle length `26`, and adding the shift amount `i` itself. The result tells us at which move number the last shift could feasibly happen. If this move number is greater than `k`, the transformation is not possible within the move limit, so we return `false`.

6. **Result**: If none of the calculated shift timings exceed `k`, then it is verified that all characters from `s` can be shifted to match `t` within the move limit. Thus, the function returns `True`.

By using a frequency array and calculating the maximum move number needed for each shift, the algorithm efficiently determines the possibility of transformation without actually performing the moves, resulting in a less complex and time-efficient solution.

Example Walkthrough

Let's consider a simple example:

- `s = "abc"`
- `t = "bcd"`
- `k = 2`

Step 1: Length Check

- Both strings `s` and `t` are of equal length, which is 3. We can proceed with the transformation process.

Step 2: Frequency Array

- We create an array `cnt` with length 26, initialized to zero: `cnt = [0] * 26`.

Step 3: Calculate Shifts

- We loop over `s` and `t` in parallel and calculate the shift required:
  - For s[0]: 'a' to t[0]: 'b' requires 1 shift.
  - For s[1]: 'b' to t[1]: 'c' requires 1 shift.
  - For s[2]: 'c' to t[2]: 'd' requires 1 shift.
- The shift calculations:
  - Shift for 'a' to 'b' is `(ord('b') - ord('a') + 26) % 26 = 1`.
  - Shift for 'b' to 'c' is `(ord('c') - ord('b') + 26) % 26 = 1`.
  - Shift for 'c' to 'd' is `(ord('d') - ord('c') + 26) % 26 = 1`.

Step 4: Count the Shifts

- For each shift calculated, increment the corresponding index in `cnt`:
  - For shift 1: `cnt[1]` becomes 3 because we need to shift by 1 three times.

Step 5: Move Validation

- We iterate through `cnt` starting from index 1:
  - For shift amount `i = 1` and frequency `cnt[1] = 3`, calculate maximum move needed: `i + 26 * (cnt[1] - 1) = 1 + 26 * (3 - 1) = 53` which exceeds `k = 2`, implying that it's impossible to conduct all required shifts within `k` moves.

As we can see, for the sample `s` and `t`, it would not be possible to transform `s` into `t` with only `k` moves since the latest move needed exceeds `k`. Thus, the `false` outcome would be the correct answer for this example.

Python Solution

```python
class Solution:
    def can_convert_string(self, s: str, t: str, k: int) -> bool:
        # If the lengths of the input strings differ, conversion is not possible
        if len(s) != len(t):
            return False

        # Initialize an array to keep count of the number of shifts for each character
        shift_counts = [0] * 26

        # Iterate over the characters of both strings
        for char_s, char_t in zip(s, t):
            # Calculate the shift difference and take modulo 26 to wrap around the alphabet
            shift = (ord(char_t) - ord(char_s) + 26) % 26
            # Increment the count of the corresponding shift
            shift_counts[shift] += 1

        # Check if the shifts can be achieved within the allowed operations 'k'
        for i in range(1, 26):
            # Calculate the maximum number of operations needed for this shift
            # For multiple shifts 'i', we need to wait 26 more for each additional use
            max_operations = i + 26 * (shift_counts[i] - 1)

            # If max_operations exceeds 'k', then it's not possible to convert the string
            if max_operations > k:
                return False

        # If all shifts are possible within the operations limit, return True
        return True

# Example usage:
# solution_instance = Solution()
# result = solution_instance.can_convert_string("input1", "input2", 10)
# print(result) # Output: True or False depending on whether the conversion is possible
```

Java Solution

```java
class Solution {

    // Method to determine if it's possible to convert string s to string t
    // by shifting each character in s, at most k times.
    public boolean canConvertString(String s, String t, int k) {
        // Return false if the lengths of the strings are different.
        if (s.length() != t.length()) {
            return false;
        }

        // Array to keep track of how many shifts for each letter are required.
        int[] shiftCounts = new int[26];

        // Calculate the shift required for each character to match the target string.
        for (int i = 0; i < s.length(); ++i) {
            int shift = (t.charAt(i) - s.charAt(i) + 26) % 26;
            ++shiftCounts[shift];
        }

        // Check for each shift if it's possible within the allowed maximum of k shifts.
        for (int i = 1; i < 26; ++i) {
            // If the maximum shift needed for any character is more than k, return false.
            if (i + 26 * (shiftCounts[i] - 1) > k) {
                return false;
            }
        }

        // If all shifts are possible within the allowed maximum, return true.
        return true;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    bool canConvertString(string source, string target, int maxShifts) {
        // If the lengths of source and target are not the same, then conversion is not possible
        if (source.size() != target.size()) {
            return false;
        }

        // Initialize an array to count the number of times a particular shift is needed
        int shiftCount[26] = {}; // There are 26 possible shifts (for 26 letters)

        // Iterate over the characters of the strings
        for (int i = 0; i < source.size(); ++i) {
            // Calculate the shift needed to convert source[i] to target[i]
            // % 26 ensures the shift is in the range [0, 25]
            // +26 before % 26 takes care of negative shifts, turning them positive
            int shift = (target[i] - source[i] + 26) % 26;
            // Increment the count of the shift needed
            ++shiftCount[shift];
        }

        // Iterate over all possible shifts except 0 (no shift needed)
        for (int i = 1; i < 26; ++i) {
            // Check if the number of shifts 'i' can be performed within the maxShifts
            // This considers every 26th shift because the same letter can't be shifted until 26 others have occurred
            if (i + 26 * (shiftCount[i] - 1) > maxShifts) {
                return false; // If not possible, return false
            }
        }

        // If all shifts can be performed, return true
        return true;
    }
};
```

Typescript Solution

```typescript
function canConvertString(source: string, target: string, maxShifts: number): boolean {
    // If the lengths of source and target are not the same, then conversion is not possible
    if (source.length !== target.length) {
        return false;
    }

    // Initialize an array to count the number of times a particular shift is needed
    // There are 26 possible shifts (for 26 letters in the alphabet)
    let shiftCount: number[] = new Array(26).fill(0);

    // Iterate over the characters of the strings
    for (let i = 0; i < source.length; i++) {
        // Calculate the shift needed to convert source[i] to target[i]
        // % 26 ensures the shift is in the range [0, 25]
        // Adding 26 before % 26 takes care of negative shifts by making them positive
        let shift = (target.charCodeAt(i) - source.charCodeAt(i) + 26) % 26;
        // Increment the count of the shift needed
        shiftCount[shift]++;
    }

    // Iterate over all possible shifts except 0 (no shift needed)
    for (let i = 1; i < 26; i++) {
        // Check if the number of shifts 'i' can be performed within the maxShifts
        // This considers every 26th shift because the same letter can't be shifted again until 26 other shifts have occurred
        if (i + 26 * (shiftCount[i] - 1) > maxShifts) {
            return false; // If not possible, return false
        }
    }

    // If all shifts can be performed given the constraints, return true
    return true;
}
```

Time and Space Complexity

The time complexity of the code is `O(n)`, where `n` is the length of the strings `s` and `t`. This is because there is a single loop that iterates over the characters of `s` and `t` only once, and the operations within the loop are of constant time. The second loop is not dependent on `n` and iterates up to a constant value (26), which does not affect the time complexity in terms of `n`.

The space complexity of the code is `O(1)`, as there is a fixed-size integer array `cnt` of size 26, which does not scale with the input size. The rest of the variables use constant space as well.