# 1585. Check If String Is Transformable With Substring Sort Operations

Hard Queue String Sorting

## Problem Description

The goal is to determine if it's possible to convert string `s` into string `t` by performing several operations. In each operation, you can select a non-empty substring from `s` and rearrange its characters in ascending order. A substring is defined as a contiguous block of characters within the string. The operation can be applied any number of times to any substring within `s`. The problem asks to return `true` if it can be transformed into `t` using these operations, otherwise return `false`.

## Intuition

To solve the problem, the solution uses the intuition that each character in string `s` can only move towards the left in each operation. This is due to the fact that sorting a substring in an ascending order will not move any character to a position greater than its original position.

Given that constraint, we check if we can transform `s` into `t` by considering the characters in `t` from left to right. We make use of a data structure `pos`, which is a dictionary of deques, where the keys are the digits (characters represented as integers) found in `s`, and the values are deques holding the indices of these digits.

As we iterate through each character in `t`, we check if the character can be moved to its correct position in `s`. This involves checking if there is an occurrence of the character left in `s` (`pos[x]` is not empty), and ensuring that there are no smaller characters (digits) to the left of the character's current position in `s` (`any(pos[i] and pos[i][0] < pos[x][0] for i in range(x))` returns `False`). If these conditions are not met, it means that it is not possible to transform `s` into `t`, and we return `False`. Otherwise, we "move" the character in `s` to its next position (which is simulated by popping from the left of the deque corresponding to that character) and continue with the next character in `t`.

If we successfully go through the entire string `t` without returning `False`, it means the transformation is possible and we return `true`.

## Solution Approach

The implementation of the solution takes a strategic approach using the following algorithmic concepts and data structures:

- **Deque Data Structure**: The solution uses a `defaultdict` to create a mapping of `deques` which are essentially double-ended queues to store the indices of corresponding characters. The `deque` allows efficient removal of elements from both ends which is essential since we are trying to simulate the movement of characters within the string.
- **Index Tracking**: Each character's position in `s` is tracked by storing its indices in order in the corresponding `deque`. This helps to determine the possibility of moving a character to a certain position.
- **Greedy Approach**: A greedy approach is applied by attempting to match characters from `t` with characters in `s` from left to right. This checks the feasibility of reaching the desired pattern.

To understand the step-by-step algorithm:

1. **Initialize** `pos`: A `defaultdict(deque)` called `pos` is created to store the indices of characters in `s`.
2. **Store Indices**: Iterating through `s`, the indices are stored in `pos`. For example, if `s = '3421'` then `pos[1]` will be `deque([3])`, `pos[2]` will be `deque([2])`, and so on.
3. **The Transformation Check**: The algorithm runs through each character `c` in `t` and performs the following:
   - It converts `c` to an integer `x`.
   - It checks if `pos[x]` is not empty, which means there's still an occurrence of that character in `s`.
   - It uses a combination of list comprehension and the `any` function to determine if there's any smaller character to the left of the current one in `s`. This is done with `any(pos[i] and pos[i][0] < pos[x][0] for i in range(x))`. If true, then it's not possible to move `c` to the desired position without violating the sorting constraint (since a smaller number cannot be passed).
4. **Simulate Character Movement**: If the checks pass, the character's index is moved (deque is popped from the left) as if the character has been sorted to its place in `s`. This is effectively simulating the operation defined in the problem.
5. **Return the Result**: If no inconsistencies are found during the whole iteration, the transformation is possible, and therefore we return `true`. If at any point an inconsistency arises, the function returns `false` immediately.

The algorithm's correctness hinges on the property that you can only sort substrings in ascending order, which consequently only allows characters in `s` to move leftward (smaller index) and not rightward.

### Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we have `s = "4321"` and `t = "1234"`. We want to find out if it's possible to convert `s` into `t` using the defined operations.

Following the solution steps:

1. **Initialize** `pos`: We create a `defaultdict(deque)` to hold the indices of characters from string `s`. The `pos` will look like this after we've finished:

```
pos[4] = deque([0])
pos[3] = deque([1])
pos[2] = deque([2])
pos[1] = deque([3])
```

2. **Store Indices**: As we've initialized `pos`, each character from `s` has its index stored in the corresponding deque.
3. **Transformation Check**: Now, we check if we can transform `s` into `t` character by character.
   - We start with `t[0]` which is `'1'`, convert it into an integer `x = 1`, and see if `pos[x]` is not empty (which means '1' is still in `s`). `pos[1]` is `deque([3])`, so it's not empty.
   - We check if there's a smaller character to the left of the '1' in `s` using the `any` statement. Since '1' is the smallest character, no smaller character can be to the left, so we pass this check.
   - We then "move" the '1' by popping from `pos[1]`, simulating the sorting operation. `pos` now looks like this:

```
pos[4] = deque([0])
pos[3] = deque([1])
pos[2] = deque([2])
pos[1] = deque([]) // '1' has been moved
```

4. **Repeat for the Next Character**: We repeat the same check for the next character in `t`, which is `'2'`.
   - `x` becomes 2 and `pos[2]` is `deque([2])`, so we have a '2' to work with.
   - With the `any` statement, we check for smaller characters. Since '1' has already been moved, and there is no '0' in `s`, the check passes.
   - We "move" the '2' by popping from `pos[2]`, and `pos` now looks like:

```
pos[4] = deque([0])
pos[3] = deque([1])
pos[2] = deque([]) // '2' has been moved
pos[1] = deque([])
```

5. **Continue With Remaining Characters**: We continue this process for '3' and '4'. Each character passes the condition checks and gets moved.
6. **Return the Result**: Since we were able to move all characters from `s` to match the sorted order of `t` without encountering any blocking condition, we conclude that it's possible to transform `s` into `t`. The function would return `true`.

This walkthrough demonstrates that given these operations and their constraints, we can determine the possibility of transforming one string into another algorithmically, using `deque`s for index tracking and a series of checks to simulate the transformation.

## Solution Implementation

### Python

```python
from collections import defaultdict

class Solution:
    def isTransformable(self, s: str, t: str) -> bool:
        # Initialize a dictionary to store queues of indices for each digit in s
        digit_positions = defaultdict(deque)

        # Fill the dictionary with the positions of each digit in s
        for index, digit in enumerate(s):
            digit_positions[int(digit)].append(index)

        # Iterate over each digit in t
        for digit in t:
            digit_value = int(digit)

            # If there are no positions left for the current digit or if there is any digit with a smaller value
            # in front of the current digit's earliest position, the transformation is not possible
            if not digit_positions[digit_value] or any(digit_positions[smaller_digit] and digit_positions[smaller_digit][0] < digit_positions[digit_value][0] for smaller_digit in range(digit_value)):
                return False

            # If the current digit can be transformed, remove its earliest occurrence from the queue
            digit_positions[digit_value].popleft()

        # If all digits in t can be reached by transforming s successfully, return True
        return True
```

### Java

```java

class Solution {

    public boolean isTransformable(String s, String t) {
        // Initialize an array of queues representing each digit from 0 to 9
        Deque<Integer>[] positions = new Deque[10];
        Arrays.setAll(positions, x -> new ArrayDeque<>());

        // Fill each queue with indices of digits in string s
        for (int i = 0; i < s.length(); ++i) {
            int digit = s.charAt(i) - '0'; // Get digit at char position i
            positions[digit].offer(i); // Add index to corresponding digit queue
        }

        // Process the target string to see if it is transformable from source string
        for (int i = 0; i < t.length(); ++i) {
            int targetDigit = t.charAt(i) - '0'; // Target digit to search for

            // If there are no positions left for this digit, s cannot be transformed into t
            if (positions[targetDigit].isEmpty()) {
                return false;
            }

            // Check if any smaller digit appears after the current digit in the source
            for (int j = 0; j < targetDigit; ++j) {
                // If a smaller digit is found and it comes before the current digit in source s
                if (!positions[j].isEmpty() && positions[j].peek() < positions[targetDigit].peek()) {
                    return false; // s cannot be transformed into t
                }
            }

            // If position is valid, remove it as it is 'used' for transformation
            positions[targetDigit].poll();
        }

        // If all checks pass, s is transformable into t
        return true;
    }
}
```

### C++

```cpp

class Solution {
public:
    // Function to check if the string 's' can be transformed into the string 't'.
    bool isTransformable(string s, string t) {
        // Queue to store the positions of digits 0-9 in the string 's'.
        queue<int> digit_positions[10];

        // Fill the queues with the positions of each digit in 's'.
        for (int i = 0; i < s.size(); ++i) {
            digit_positions[s[i] - '0'].push(i);
        }

        // Iterate through each character in the target string 't'.
        for (char& c : t) {
            int digit = c - '0'; // Convert character to digit

            // If there is no such digit in 's', transformation is impossible.
            if (digit_positions[digit].empty()) {
                return false;
            }

            // Check if any smaller digit is positioned before our digit in 's'.
            for (int j = 0; j < digit; ++j) {
                // If there is a smaller digit in front of our digit, it's not transformable.
                if (!digit_positions[j].empty() && digit_positions[j].front() < digit_positions[digit].front()) {
                    return false;
                }
            }

            // Since this digit can be safely transformed, we dequeue it from the corresponding queue.
            digit_positions[digit].pop();
        }

        // If all characters can be transformed without violating the constraints, return true.
        return true;
    }
};
```

### TypeScript

```typescript

// Import the Queue implementation if you don't have one.
// This is just a placeholder, as TypeScript does not have a built-in queue.
// You can replace this with your own Queue class or any library implementation.
import { Queue } from 'some-queue-library';

// Function to check if the string 'src' can be transformed into the string 'target'.
function isTransformable(src: string, target: string): boolean {
    // Array to store the positions of digits 0-9 in the string 'src'.
    const digitPositions: Queue<number>[] = [];

    // Initialize queues for each digit.
    for (let i = 0; i < 10; i++) {
        digitPositions[i] = new Queue<number>();
    }

    // Fill the queues with the positions of each digit in 'src'.
    for (let i = 0; i < src.length; i++) {
        const digit = parseInt(src[i], 10);
        digitPositions[digit].enqueue(i);
    }

    // Iterate through each character in the target string 'target'.
    for (const char of target) {
        const digit = parseInt(char, 10); // Convert character to digit

        // If there is no such digit in 'src', transformation is impossible.
        if (digitPositions[digit].isEmpty()) {
            return false;
        }

        // Check if any smaller digit is positioned before this digit in 'src'.
        for (let j = 0; j < digit; j++) {
            // If there is a smaller digit in front of this digit, it's not transformable.
            if (!digitPositions[j].isEmpty() && digitPositions[j].front() < digitPositions[digit].front()) {
                return false;
            }
        }

        // Since this digit can be safely transformed, we dequeue it from the corresponding queue.
        digitPositions[digit].dequeue();
    }

    // If all characters can be transformed without violating the constraints, return true.
    return true;
}
```

```python
from collections import defaultdict

class Solution:
    def isTransformable(self, s: str, t: str) -> bool:
        # Initialize a dictionary to store queues of indices for each digit in s
        digit_positions = defaultdict(deque)

        # Fill the dictionary with the positions of each digit in s
        for index, digit in enumerate(s):
            digit_positions[int(digit)].append(index)

        # Iterate over each digit in t
        for digit in t:
            digit_value = int(digit)

            # If there are no positions left for the current digit or if there is any digit with a smaller value
            # in front of the current digit's earliest position, the transformation is not possible
            if not digit_positions[digit_value] or any(digit_positions[smaller_digit] and digit_positions[smaller_digit][0] < digit_positions[digit_value][0] for smaller_digit in range(digit_value)):
                return False

            # If the current digit can be reached by transforming s successfully, return True
            digit_positions[digit_value].popleft()

        # If all digits in t can be reached by transforming s successfully, return True
        return True
```

## Time and Space Complexity

The given Python function `isTransformable` checks if it's possible to transform the string `s` into the string `t` by repeatedly moving a digit to the leftmost position if there are no smaller digits to its left.

### Time Complexity:

To determine the time complexity, let us analyze the various operations being performed in the function.

1. **Initializing** `pos`: Building the `pos` dictionary takes $O(n)$ time, where $n$ is the length of `s`, as we iterate over all characters in `s` once.
2. **Checking Transformability**: We iterate over each character in `t`, and for each character, perform a check that could iterate over a maximum of 10 possible values (since the digits range from 0 to 9) - this is the `any()` function call.

   The inner check `pos[i] and pos[i][0] < pos[x][0]` is $O(1)$ for every digit `i` for each character of `t`, because it's merely indexing and comparison.

   Therefore, the total time for transforming, in the worst case, would be $O(10 * n)$ which simplifies to $O(n)$, since we don't count constants in Big O notation.
3. **Popping from** `pos`: The `popleft()` operation is $O(1)$ for a deque.

Therefore, the overall time complexity of the function is $O(n)$.

### Space Complexity:

1. **Space for** `pos`: The `pos` dictionary stores deque objects for each unique digit found in `s`, and each deque can grow to the size of the number of occurrences of that digit. The total size of all deques combined will not exceed $n$. Therefore, the space complexity contributed by `pos` is $O(n)$.
2. **Miscellaneous Space**: Constant additional space used by iterators and temporary variables.

Thus, the total space complexity is also $O(n)$.

In conclusion, the time complexity is $O(n)$ and the space complexity is $O(n)$.