

1585. Check If String Is Transformable With Substring Sort Operations

Hard Greedy String Sorting

Leetcode Link

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 `s` 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:

- Initialize `pos`:** A `defaultdict(deque)` called `pos` is created to store the indices of characters in `s`.
- 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.
- 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 `x` to the desired position without violating the sorting constraint (since a smaller number cannot be passed).
- 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.
- 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:

- 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:

```
1 pos[4] = deque([0])
2 pos[3] = deque([1])
3 pos[2] = deque([2])
4 pos[1] = deque([3])
```
- Store Indices:** As we've initialized `pos`, each character from `s` has its index stored in the corresponding deque.
- 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:

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

```
1 pos[4] = deque([0])
2 pos[3] = deque([1])
3 pos[2] = deque([]) // '2' has been moved
4 pos[1] = deque([])
```
- Continue With Remaining Characters:** We continue this process for `'3'` and `'4'`. Each character passes the condition checks and gets moved.
- 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 `deques` for index tracking and a series of checks to simulate the transformation.

Python Solution

```
1 from collections import defaultdict
2
3 class Solution:
4     def isTransformable(self, s: str, t: str) -> bool:
5         # Initialize a dictionary to store queues of indices for each digit in s
6         digit_positions = defaultdict(deque)
7
8         # Fill the dictionary with the positions of each digit in s
9         for index, digit in enumerate(s):
10             digit_positions[int(digit)].append(index)
11
12         # Iterate over each digit in t
13         for digit in t:
14             digit_value = int(digit)
15
16             # If there are no positions left for the current digit or if there is any digit with a smaller value
17             # in front of the current digit's earliest position, the transformation is not possible
18             if not digit_positions[digit_value] or any(digit_positions[smaller_digit] and digit_positions[smaller_digit][0] < digit_value for smaller_digit in range(digit_value)):
19                 return False
20
21             # If the current digit can be transformed, remove its earliest occurrence from the queue
22             digit_positions[digit_value].popleft()
23
24         # If all digits in t can be reached by transforming s successfully, return True
25         return True
26
```

Java Solution

```
1 class Solution {
2
3     public boolean isTransformable(String s, String t) {
4         // Initialize an array of queues representing each digit from 0 to 9
5         Deque<Integer>[] positions = new Deque[10];
6         Arrays.setAll(positions, k -> new ArrayDeque<>());
7
8         // Fill each queue with indices of digits in string s
9         for (int i = 0; i < s.length(); i++) {
10             int digit = s.charAt(i) - '0'; // Get digit at char position i
11             positions[digit].offer(i); // Add index to corresponding digit queue
12         }
13
14         // Process the target string to see if it is transformable from source string
15         for (int i = 0; i < t.length(); i++) {
16             int targetDigit = t.charAt(i) - '0'; // Target digit to search for
17
18             // If there are no positions left for this digit, s cannot be transformed into t
19             if (positions[targetDigit].isEmpty()) {
20                 return false;
21             }
22
23             // Check if any smaller digit appears after the current digit in the source
24             for (int j = 0; j < targetDigit; j++) {
25                 // If there is a smaller digit and it comes before the current digit in source s
26                 // if (positions[j].isEmpty() && positions[i].peek() < positions[targetDigit].peek()) {
27                     return false; // s cannot be transformed into t
28                 }
29             }
30
31             // If position is valid, remove it as it is 'used' for transformation
32             positions[targetDigit].poll();
33         }
34
35         // If all checks pass, s is transformable into t
36         return true;
37     }
38 }
39
```

C++ Solution

```
1 class Solution {
2 public:
3     // Function to check if the string 'src' can be transformed into the string 't'.
4     bool isTransformable(string s, string t) {
5         // Queue to store the positions of digits 0-9 in the string 's'.
6         queue<int> digit_positions[10];
7
8         // Fill the queues with the positions of each digit in 's'.
9         for (int i = 0; i < s.size(); i++) {
10             digit_positions[s[i] - '0'].push(i);
11         }
12
13         // Iterate through each character in the target string 't'.
14         for (char& c : t) {
15             int digit = c - '0'; // Convert character to digit
16
17             // If there is no such digit in 's', transformation is impossible.
18             if (digit_positions[digit].empty()) {
19                 return false;
20             }
21
22             // Check if any smaller digit is positioned before our digit in 's'.
23             for (int j = 0; j < digit; j++) {
24                 // If there is a smaller digit in front of our digit, it's not transformable.
25                 if (!digit_positions[j].isEmpty() && digit_positions[j].front() < digit_positions[digit].front()) {
26                     return false;
27                 }
28             }
29
30             // Since this digit can be safely transformed, we pop it from the queue.
31             digit_positions[digit].pop();
32         }
33
34         // If all characters can be transformed without violating the constraints, return true.
35         return true;
36     }
37 };
38
```

Typescript Solution

```
1 // Import the Queue implementation if you don't have one.
2 // This is just a placeholder, as TypeScript does not have a built-in queue.
3 // You can replace this with your own Queue class or any library implementation.
4 import { Queue } from 'some-queue-library';
5
6 // Function to check if the string 'src' can be transformed into the string 'target'.
7 function isTransformable(src: string, target: string): boolean {
8     // Array to store the positions of digits 0-9 in the string 'src'.
9     const digitPositions: Queue<number>[] = [];
10
11     // Initialize queues for each digit.
12     for (let i = 0; i < 10; i++) {
13         digitPositions[i] = new Queue<number>();
14     }
15
16     // Fill the queues with the positions of each digit in 'src'.
17     for (let i = 0; i < src.length; i++) {
18         const digit = parseInt(src[i], 10);
19         digitPositions[digit].enqueue(i);
20     }
21
22     // Iterate through each character in the target string 'target'.
23     for (const char of target) {
24         const digit = parseInt(char, 10); // Convert character to digit
25
26         // If there is no such digit in 'src', transformation is impossible.
27         if (digitPositions[digit].isEmpty()) {
28             return false;
29         }
30
31         // Check if any smaller digit is positioned before this digit in 'src'.
32         for (let j = 0; j < digit; j++) {
33             // If there is a smaller digit in front of this digit, it's not transformable.
34             if (!digitPositions[j].isEmpty() && digitPositions[j].front() < digitPositions[digit].front()) {
35                 return false;
36             }
37         }
38
39         // Since this digit can be safely transformed, we dequeue it from the corresponding queue.
40         digitPositions[digit].dequeue();
41     }
42
43     // If all characters can be transformed without violating the constraints, return true.
44     return true;
45 }
46
```

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.

- 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.
- 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.

- 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:

- 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 $O(n)$.

- 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 $O(n)$.