2337. Move Pieces to Obtain a String

Medium Two Pointers String

Leetcode Link

Problem Description

The given problem is about transforming one string into another using specific movement rules for the characters within the strings. The strings `start` and `target` are of the same length and consist only of characters 'L', 'R', and '_'. *An 'L' can move only left if there is a blank space (*'_'*) directly to its left, and an 'R' can move only right if there is a blank space directly to its right.* The goal is to determine if the `start` string can be transformed into the `target` string by applying these movements any number of times. The output should be `true` if the transformation is possible, otherwise `false`.

Intuition

The intuition behind the solution involves two key insights:

1. The relative order of the pieces 'L' and 'R' cannot change because 'L' can only move left and 'R' can only move right. Therefore, if in the `target` string, an 'R' appears to the left of an 'L' which was to its right in the `start` string, the transformation is not possible.
2. Given the same relative order, an 'L' can only move to the left, so its position in the `target` string must not be to the right of its position in the `start` string. Similarly, an 'R' can only move to the right, so its position in the `target` string must not be to the left of its position in the `start` string.

With these insights in mind, the solution approach is straightforward:

- Ignore the blank spaces and compare the positions of the non-blank characters in both strings. If there are a different number of 'L' and 'R' characters or they are in a different relative order, the target cannot be reached.
- If the characters are in the same order, check if 'L' in `target` is never to the right of its position in `start`, and 'R' is never to the left. If this is true for all characters, then the transformation is possible and return `true`. Otherwise, return `false`.

Solution Approach

The Python code provided earlier implements the solution approach effectively. It uses the following steps and *Python*-specific data structures and functions:

1. **Filtering and Pairing**: The comprehension lists `a` and `b` filter out the blank spaces `'_'` and create lists of tuples that contain the character and its index (position) from the `start` and `target` strings, respectively. This is done using the `enumerate` function which gives the index along with each character as you iterate over the string:

```
1 a = [(c, i) for i, c in enumerate(start) if c != '_']
2 b = [(c, i) for i, c in enumerate(target) if c != '_']
```

2. **Checking the Length**: This step checks if the lengths of the filtered lists are equal. If they are not, it is not possible to get `target` from `start` since the number of movable pieces is different, so the function returns `False`.

```
1 if len(a) != len(b):
2     return False
```

3. **Comparing Corresponding Pairs**: The use of the `zip` function takes pairs from `a` and `b` in a parallel manner to ensure we're comparing pieces from the `start` and `target` strings that are supposed to correspond to each other.
4. **Piece Type and Position Validation**: For each pair of tuples `(c, i)` from `a`, and `(d, j)` from `b`, the following checks are performed:
   - If the piece type is not the same (`c != d`), it cannot be a valid transformation since 'L' cannot become 'R' and vice versa, hence `False` is returned.
   - If the piece is 'L', it should not be to the right in `target` as compared to `start` (`i < j` case), since 'L' can only move left.
   - If the piece is 'R', it should not be to the left in `target` as compared to `start` (`i > j` case), since 'R' can only move right.

   If any of these conditions are violated, the function returns `False`.

```
1 for (c, i), (d, j) in zip(a, b):
2     if c != d or (c == 'L' and i < j) or (c == 'R' and i > j):
3         return False
```

5. **Returning True**: If none of the above checks fail, it means the transformation from `start` to `target` is possible, hence `True` is returned at the end of the function.

By using tuples for storing character and index pairs, and the `zip` function to iterate over them, we avoid the need for more complex data structures. This simplifies the algorithm and improves its readability and performance.

Example Walkthrough

Let's consider two strings as an example:

- `start`: `"R_L"`
- `target`: `"_RL"`

Following the steps of the solution approach:

1. **Filtering and Pairing**:

   For `start`:

```
1 a = [(c, i) for i, c in enumerate("R_L") if c != '_']
2 # a = [('R', 0), ('L', 2)]
```

   For `target`:

```
1 b = [(c, i) for i, c in enumerate("_RL") if c != '_']
2 # b = [('R', 1), ('L', 2)]
```

2. **Checking the Length**:

   Length of `a` 2 Length of `b` 2

   Since both lengths are equal, we proceed.

3. **Comparing Corresponding Pairs**:

   We utilize `zip` to compare each pair from `a` and `b`.

   Pairs to compare:

```
1 ('R', 0) from 'start' and ('R', 2) from 'target'
2 ('L', 3) from 'start' and ('L', 3) from 'target'
```

4. **Piece Type and Position Validation**:

   For the first pair:

   - Piece type is the same, both are 'R'.
   - 'R' is in position 0 in `start` and position 2 in `target`, which is a valid move for 'R', since it can move to the right.

   For the second pair:

   - Piece type is the same, both are 'L'.
   - 'L' is in position 3 in `start` and position 3 in `target`, indicating no movement, which is also valid since 'L' cannot move to the right.

   Since both pairs are valid according to the movement rules, we can continue.

5. **Returning True**:

   Because none of the validation checks failed, the function would return `True`, meaning it is possible to transform the `start` string `"R_L"` into the `target` string `"_RL"` by moving the 'R' two spaces to the right.

Python Solution

```
1  class Solution:
2      def canChange(self, start: str, target: str) -> bool:
3          # Create pairs of (character, index) for non-'_' characters in start string.
4          start_positions = [(char, index) for index, char in enumerate(start) if char != '_']
5
6          # Create pairs of (character, index) for non-'_' characters in target string.
7          target_positions = [(char, index) for index, char in enumerate(target) if char != '_']
8
9          # If the number of non-'_' characters in start and target are different, return False.
10         if len(start_positions) != len(target_positions):
11             return False
12
13         # Iterate over the pairs of start and target together.
14         for (start_char, start_index), (target_char, target_index) in zip(start_positions, target_positions):
15             # If the characters are not the same, the transformation is not possible.
16             if start_char != target_char:
17                 return False
18             # A 'L' character in start should have an index greater than or equal to that in target.
19             if start_char == 'L' and start_index < target_index:
20                 return False
21             # A 'R' character in start should have an index less than or equal to that in target.
22             if start_char == 'R' and start_index > target_index:
23                 return False
24
25         # If all conditions are met, return True indicating the transformation is possible.
26         return True
27
```

Java Solution

```
1  class Solution {
2      // Main method to check if it's possible to transform the start string to the target string
3      public boolean canChange(String start, String target) {
4          // Parse the strings to obtain the positions and types of 'L' and 'R' characters
5          List<int[]> startPosList = parseString(start);
6          List<int[]> targetPosList = parseString(target);
7
8          // If the number of 'L' and 'R' characters in both strings is different, transformation is not possible
9          if (startPosList.size() != targetPosList.size()) {
10             return false;
11         }
12
13         // Compare the positions and types of 'L' and 'R' characters in the two lists
14         for (int i = 0; i < startPosList.size(); ++i) {
15             int[] startPos = startPosList.get(i);
16             int[] targetPos = targetPosList.get(i);
17
18             // If the types of characters (L or R) are different at any point, transformation is not possible
19             if (startPos[0] != targetPos[0]) {
20                 return false;
21             }
22             // If 'L' in start is to the right of 'L' in target, transformation is not possible as 'L' only moves left
23             if (startPos[0] == 'L' && startPos[1] < targetPos[1]) {
24                 return false;
25             }
26             // If 'R' in start is to the left of 'R' in target, transformation is not possible as 'R' only moves right
27             if (startPos[0] == 'R' && startPos[1] > targetPos[1]) {
28                 return false;
29             }
30         }
31         // All checks passed, transformation is possible
32         return true;
33     }
34
35     // Helper method to parse a string to a list of positions and types for 'L' and 'R'
36     private List<int[]> parseString(String s) {
37         List<int[]> result = new ArrayList<>();
38         for (int i = 0; i < s.length(); ++i) {
39             char currentChar = s.charAt(i);
40             // If the current character is 'L', add to the list with type 1
41             if (currentChar == 'L') {
42                 result.add(new int[] {1, i});
43             }
44             // If the current character is 'R', add to the list with type 2
45             else if (currentChar == 'R') {
46                 result.add(new int[] {2, i});
47             }
48         }
49         return result;
50     }
51 }
52
```

C++ Solution

```
1  #include <vector>
2  #include <string>
3  #include <utility>
4
5  using namespace std;
6
7  // Define "pii" as an alias for "pair<int, int>".
8  using pii = pair<int, int>;
9
10 class Solution {
11 public:
12     // Main function to determine if one string can transition to another.
13     bool canChange(string start, string target) {
14         // Extract positions and directions of 'L' and 'R' from both strings.
15         auto startPositions = extractPositions(start);
16         auto targetPositions = extractPositions(target);
17
18         // If the number of 'L' and 'R' characters are different, return false.
19         if (startPositions.size() != targetPositions.size()) return false;
20
21         // Check each corresponding 'L' and 'R' character from start and target.
22         for (int i = 0; i < startPositions.size(); ++i) {
23             auto startPosition = startPositions[i], targetPosition = targetPositions[i];
24             // If the direction is different, the change is not possible.
25             if (startPosition.first != targetPosition.first) return false;
26             // If an 'L' in start is to the right of the 'L' in target, change is not possible.
27             if (startPosition.first == 1 && startPosition.second < targetPosition.second) return false;
28             // If an 'R' in start is to the left of the 'R' in target, change is not possible.
29             if (startPosition.first == 2 && startPosition.second > targetPosition.second) return false;
30         }
31
32         // If all 'L' and 'R' can be moved to their target positions, return true.
33         return true;
34     }
35
36     // Helper function to extract the positions and directions of 'L' and 'R'.
37     vector<pii> extractPositions(string s) {
38         vector<pii> positions;
39         for (int i = 0; i < s.size(); ++i) {
40             // If the character is 'L', associate it with direction 1.
41             if (s[i] == 'L') {
42                 positions.push_back({1, i});
43             // If the character is 'R', associate it with direction 2.
44             } else if (s[i] == 'R') {
45                 positions.push_back({2, i});
46             }
47         }
48         return positions;
49     }
50 };
51
```

Typescript Solution

```
1  function canChange(start: string, target: string): boolean {
2      const length = start.length; // The length of the start and target strings
3      let startLow = 0; // Start index for iterating through the start string
4      let targetLow = 0; // Start index for iterating through the target string
5
6      while (true) {
7          // Skip all the underscores in the start string
8          while (startLow < length && start[startLow] === '_') {
9              ++startLow;
10         }
11         // Skip all the underscores in the target string
12         while (targetLow < length && target[targetLow] === '_') {
13             ++targetLow;
14         }
15
16         // If both indices have reached the end, the strings can be changed to each other
17         if (startLow === length && targetLow === length) {
18             return true;
19         }
20
21         // If one index reaches the end before the other, or the characters at the current indices do not match,
22         // the strings cannot be changed to each other
23         if (startLow === length || targetLow === length || start[startLow] !== target[targetLow]) {
24             return false;
25         }
26
27         // If the character is 'L' and the start index is ahead of the target index, or
28         // if the character is 'R' and the start index is behind the target index,
29         // it is not possible to change the strings according to the rules
30         if ((start[startLow] === 'L' && startLow < targetLow) || (start[startLow] === 'R' && startLow > targetLow)) {
31             return false;
32         }
33
34         // Move both indices forward
35         ++startLow;
36         ++targetLow;
37     }
38 }
39
```

Time and Space Complexity

Time Complexity

The given Python function `canChange` checks whether it is possible to transform the `start` string into the `target` string under certain conditions. Analyzing the time complexity involves a few steps:

1. Creation of list `a`: The list comprehension iterates over each character in the `start` string and includes only non-underscore characters. Therefore, this operation has a time complexity of `O(n)` where `n` is the length of the `start` string.
2. Creation of list `b`:Similarly, the creation of list `b` iterates over each character in the `target` string and has a time complexity of `O(n)`.
3. Comparison of lengths: Checking if `len(a)` is equal to `len(b)` takes constant time, `O(1)`.
4. Zipping and iterating: Zipping the two lists `a` and `b` and iterating over them to compare elements has a time complexity of `O(m)`, where `m` is the number of non-underscore characters in the strings, which is at most `n`.

Given these steps occur sequentially, the overall time complexity is dominated by the terms with `O(n)`, leading to a total time complexity of `O(n)`.

Space Complexity

1. List `a` and `b` store the non-underscore characters and their respective indices from `start` and `target`. These take space proportional to the number of non-underscore characters, which is `O(n)` where `n` is the number of such characters.
2. The space taken by the zip object and iteration is negligible since they don't store values but only reference the elements in `a` and `b`.

Hence, the overall space complexity of the function `canChange` is `O(n)`, where `n` is the number of non-underscore characters and `n <= n`. If we consider that all characters could be non-underscore, the space complexity would also be `O(n)` in the worst case.