<https://leetcode.com/problems/move-pieces-to-obtain-a-string/description/>

You are given two strings start and target, both of length n. Each string consists **only** of the characters 'L', 'R', and '\_' where:

The characters 'L' and 'R' represent pieces, where a piece 'L' can move to the **left** only if there is a **blank** space directly to its left, and a piece 'R' can move to the **right** only if there is a **blank** space directly to its right.

The character '\_' represents a blank space that can be occupied by **any** of the 'L' or 'R' pieces.

Return true *if it is possible to obtain the string* target*by moving the pieces of the string*start***any****number of times*. Otherwise, return false.

**Example 1:**

**Input:** start = "\_L\_\_R\_\_R\_", target = "L\_\_\_\_\_\_RR"

**Output:** true

**Explanation:** We can obtain the string target from start by doing the following moves:

- Move the first piece one step to the left, start becomes equal to "**L**\_\_\_R\_\_R\_".

- Move the last piece one step to the right, start becomes equal to "L\_\_\_R\_\_\_**R**".

- Move the second piece three steps to the right, start becomes equal to "L\_\_\_\_\_\_**R**R".

Since it is possible to get the string target from start, we return true.

**Example 2:**

**Input:** start = "R\_L\_", target = "\_\_LR"

**Output:** false

**Explanation:** The 'R' piece in the string start can move one step to the right to obtain "\_**R**L\_".

After that, no pieces can move anymore, so it is impossible to obtain the string target from start.

**Example 3:**

**Input:** start = "\_R", target = "R\_"

**Output:** false

**Explanation:** The piece in the string start can move only to the right, so it is impossible to obtain the string target from start.

**Constraints:**

n == start.length == target.length

1 <= n <= 105

start and target consist of the characters 'L', 'R', and '\_'.

**Attempt 1: 2025-08-02**

**Solution 1: Two Pointers (30 min)**

class Solution {

    public boolean canChange(String start, String target) {

        int n = start.length();

        int i = 0;

        int j = 0;

        while(i < n || j < n) {

            // Skip '\_'

            while(i < n && start.charAt(i) == '\_') {

                i++;

            }

            // Skip '\_'

            while(j < n && target.charAt(j) == '\_') {

                j++;

            }

            // When one pointer reach the end, the other must reach

            // the end also, otherwise you cannot

            if(i == n || j == n) {

                return i == n && j == n;

            }

            // If any mismatch

            if(start.charAt(i) != target.charAt(j)) {

                return false;

            }

            // 'L' cannot move to right

            if(start.charAt(i) == 'L' && i < j) {

                return false;

            }

            // 'R' cannot move to left

            if(start.charAt(i) == 'R' && i > j) {

                return false;

            }

            i++;

            j++;

        }

        return true;

    }

}

Time Complexity: O(n)

Space Complexity: O(1)

**Refer to Deepseek**

Given two strings start and target, determine if you can transform start into target by moving any number of 'L' or 'R' characters under the following rules:

'L' can only move **left** (decreasing index).

'R' can only move **right** (increasing index).

A character cannot jump over another character.

**Example:**

start = "\_L\_\_R\_\_R\_", target = "L\_\_\_\_\_\_RR" → **True** ('L' moves left, 'R's move right).

start = "R\_L\_", target = "\_LR\_" → **False** ('R' cannot move left, 'L' cannot move right).

**Intuition**

**Relative Order of 'L' and 'R' Must Match**:

If we remove all '\_', start and target must have the same sequence of 'L' and 'R'.

**Example**: start = "R\_L" → "RL", target = "\_LR" → "LR" → Mismatch → **False**.

**Position Constraints**:

For each 'L' in start, its position in target must be **≤** its original position (since 'L' moves left).

For each 'R' in start, its position in target must be **≥** its original position (since 'R' moves right).

**Algorithm**

**Check Character Sequence**:

Compare the sequences of non-'\_' characters in start and target. If they differ, return false.

**Two-Pointer Validation**:

Use pointers i and j to traverse start and target.

Skip '\_' in both strings.

For each non-'\_' character:

If start[i] != target[j] → **False** (order mismatch).

If start[i] == 'L' and i < j → **False** ('L' cannot move right).

If start[i] == 'R' and i > j → **False** ('R' cannot move left).

**Solution Code**

class Solution {

public boolean canChange(String start, String target) {

int n = start.length();

int i = 0, j = 0;

while (i < n || j < n) {

// Skip underscores in 'start'

while (i < n && start.charAt(i) == '\_') i++;

// Skip underscores in 'target'

while (j < n && target.charAt(j) == '\_') j++;

// If one pointer reaches end before the other

if (i == n || j == n) {

return i == n && j == n; // Both must end together

}

// Mismatch in characters

if (start.charAt(i) != target.charAt(j)) {

return false;

}

// Check movement constraints

// If missing this check, test case:

// start = "\_LL\_\_R\_\_R\_"

// target = "L\_\_\_L\_\_\_RR"

// Output = true, Expected = false

if (start.charAt(i) == 'L' && i < j) { // 'L' cannot move right

return false;

}

// If missing this check, test case:

// start = "\_R"

// target = "R\_"

// Output = true, Expected = false

if (start.charAt(i) == 'R' && i > j) { // 'R' cannot move left

return false;

}

i++;

j++;

}

return true;

}

}

**Explanation**

**Two-Pointer Technique**:

i traverses start, j traverses target.

Both pointers skip '\_' to focus on 'L' and 'R'.

**Early Termination Checks**:

**Sequence Mismatch**: If start[i] != target[j] at any point, return false.

**Movement Validity**:

'L' in start must **not** be to the left of its position in target (cannot move right).

'R' in start must **not** be to the right of its position in target (cannot move left).

**Final Check**:

If both pointers reach the end simultaneously, the transformation is possible.

**Complexity**

**Time**: **O(n)** (single pass over both strings).

**Space**: **O(1)** (no extra space used).

This approach efficiently checks feasibility by leveraging the constraints of 'L' and 'R' movements while ensuring their relative order is preserved. 🚀

**Refer to**

[L301.Remove Invalid Parentheses (Ref.L22)](note://EAF368E1345B4DBFB47321E244D8E03F)