1616. Split Two Strings to Make Palindrome

Medium

Two Pointers

String

Problem Description

b into two substrings each. For string a, this produces a_prefix and a_suffix, and for string b, this results in b_prefix and b_suffix. After the split, the goal is to check whether concatenating a_prefix with b_suffix, or b_prefix with a_suffix, results in a palindrome. A palindrome is a string that reads the same forward as it does backward, like "racecar" or "madam".

Keep in mind that when you split a string into a prefix and suffix, either part can be empty. So, you can end up with a prefix

You are provided with two strings a and b that are of equal length. Your task is to pick an index at which you will split both a and

being an empty string and a_suffix being the entire string a, or vice versa. The challenge is to return true if you can create a

Intuition

palindrome by doing such a split and concatenation, or false otherwise.

1. If either a or b is already a palindrome, you don't need to split them; the answer is true. 2. If the beginning of a matches the end of b (and vice versa) until a certain point, then there's a chance that the remainder of the string is a

3. If we can't form a palindrome from the start, perhaps we could if we split the strings later, so we can check from the opposite direction as well;

palindrome by itself; we must check this.

Let's try to simplify the problem with the following insights:

start from the ends and move towards the middle.

Combine the checks: If any check returns true, a palindrome can be formed.

A couple of scenarios can lead to a successful palindrome formation:

palindromic considering the parts we are checking.

b_suffix or b_prefix + a_suffix forms a palindrome.

Step 1: Check for Palindrome Using the Two-Pointer Approach

Step 2: Invoking check2 for non-matching characters

Now we call check2 for substring a[i : j + 1], which is "ay".

def checkPalindromeFormation(self, a: str, b: str) -> bool:

def check_palindrome(a: str, b: str) -> bool:

or b[i:j+1] is a palindrome, return True

return check_palindrome(a, b) or check_palindrome(b, a)

public boolean checkPalindromeFormation(String a, String b) {

return checkCombination(a, b) || checkCombination(b, a);

// If indices have crossed, the resulting string is already a palindrome

private boolean isSubStrPalindrome(String str, int startIndex, int endIndex) {

// Method to check if the substring from startIndex to endIndex is a palindrome in string str

// Check both combinations, a with b and b with a

- Now, to the solution approach: Check if the current prefix of a and suffix of b can form a palindrome: Iterate through string a from the start and string b
- from the end simultaneously. As soon as the characters at the current indices don't match, stop. Check the inner substrings: When the characters don't match, you have to check if the inner substring a[i:j+1] or

because the previous characters on both ends were matching.

prefixes and suffixes of two strings a and b.

Check the opposite case: Repeat steps 1 and 2, but this time try to match the beginning of b and the end of a.

b[i:j+1] is a palindrome. If either of these substrings is a palindrome, it means that the whole string can be a palindrome,

- This approach works because starting from the ends and moving towards the middle ensures that by the time a mismatch occurs, if the remaining unchecked substring is a palindrome itself, the previous matching parts would make the whole string a palindrome. This holds true for both beginning from the start and checking towards the end, as well as the opposite.
- **Solution Approach** The solution makes use of two helper functions check1 and check2 to determine if a palindrome can be formed by combining the

The implementation involves the following steps: The check1 function is called twice, first with a and b in their original order and then with their order switched. This ensures that we check for palindromes by making a split before and also after the midpoint of the strings.

Inside check1, a two-pointer approach is used. This approach involves iterating over the strings with two indices: one (i)

starting from the beginning of a and the other (j) starting from the end of b. This continues as long as the characters at

these indices are equal. If they're not equal, this is where we might have a chance to split and form a palindrome, and we then

Once a mismatch is found, the check2 function comes into play. It takes the substring from a or b starting at index i and

Example Walkthrough

move on to step 3.

ending at j, and checks if it is a palindrome by comparing it to its reverse (a[i : j + 1][::-1]). This checks if the middle part, left unchecked by check1, forms a palindrome.

• The looping in check1 proceeds until i is greater than or equal to j, meaning all characters matched properly, so the strings are already

• The check2 function returns true, which means that even though there was a mismatch, the remaining unmatched part is a palindrome

- Finally, the checkPalindromeFormation function returns true if either call to check1 returns true. Otherwise, it returns false, indicating that no palindromic combination could be found. The solution efficiently uses the two-pointer technique to iterate and compare characters, which is a common pattern used in string manipulation algorithms. The method of checking the remaining inner substring separately helps to stop early when a
- Let us walk through an example to illustrate the solution approach.

palindrome cannot be formed, leading to an efficient algorithm both in terms of time and space complexity.

itself, thereby allowing for a successful palindrome formation when the outer matched parts are included.

We begin by using check1 to compare a and the reverse of b using a two-pointer technique. • Pointers i and j start at the start of a and the end of b respectively. We compare a[i] with b[j]. For i=0 and j=3, a[0] is 'x' and b[3] is 'x' as well, they match, so we continue.

• Increment i and decrement j, and compare a[1] with b[2]. We have a[1] is 'a' and b[2] is 'z'. They do not match, so we proceed to step 3.

• Check if "ay" is a palindrome by comparing it to its reverse "ya". It is not, so we go back to check1 and also invoke check2 on b[i : j + 1]

• Now we switch a and b and compare b and the reverse of a using a two-pointer technique initiated by calling check1 again.

• This means that in this example, we cannot form a palindrome from any split index by concatenating substrings from a and b.

Thus, the final result returned by the checkPalindromeFormation function for strings a = "xayb" and b = "abzx" would be

false. No palindromic combination is possible with the given strings following the checks performed according to the method

• This time b[0] is 'a' and the reversed a[3] is 'b', they do not match. Directly proceed to check the inner substrings.

Assume we are given the strings a = "xayb" and b = "abzx". The goal is to determine if there is a split index where $a_prefix +$

• Check if "bz" is a palindrome by comparing it to its reverse "zb". It is not a palindrome either.

Step 3: Check the Opposite Case

Step 4: Combine the checks

Solution Implementation

a suffix from the other string.

i, j = 0, len(b) - 1

while i < i and a[i] == b[j]:

and suffix from a, forms a palindrome

i, j = i + 1, j - 1

described.

class Solution:

which is "bz".

 Calling check2 for b[0:4], checking if "abzx" is a palindrome, which it is not. • Similarly, calling check2 for a[0:4], checking if "xayb" is a palindrome, which it is not.

- The checks from check1 and check2 have all returned false.
- **Python**

Check if the strings a and b can form a palindrome by switching prefix and suffix at any point.

If we've scanned the entire string and it's a palindrome, or if the substring a[i:j+1]

return i >= j or is_palindrome_substring(a, i, j) or is_palindrome_substring(b, i, j)

Return True if either combination of a prefix from a and suffix from b, or a prefix from b

// Move from both ends towards the center comparing characters of prefixString and suffixString

while (startIndex < endIndex && prefixString.charAt(startIndex) == suffixString.charAt(endIndex)) {</pre>

// or check if substring of prefixString or suffixString from startIndex to endIndex is a palindrome

return startIndex >= endIndex || isSubStrPalindrome(prefixString, startIndex, endIndex) || isSubStrPalindrome(suffixString, s

// Main method to check if a palindrome can be formed by replacing a prefix of one string with a suffix of another

Check if it is possible to create a palindrome by taking a prefix from one string and

Loop until the characters at i and j are equal and i is less than j

def is palindrome_substring(s: str, start: int, end: int) -> bool: Check if the substring s[start:end+1] is a palindrome. # Compare the substring with its reverse to check for a palindrome return s[start:end+1] == s[start:end+1][::-1]

// Helper method to check if a palindrome can be formed with a particular combination private boolean checkCombination(String prefixString, String suffixString) { int startIndex = 0; int endIndex = suffixString.length() - 1;

Java

class Solution {

startIndex++;

return left >= right;

let leftIndex = 0;

leftIndex++:

rightIndex--;

leftIndex++;

rightIndex--;

return leftIndex >= rightIndex;

a suffix from the other string.

let rightIndex = b.length - 1;

function checkPalindromeFormation(a: string, b: string): boolean {

function isPalindromeAfterSwitch(a: string, b: string): boolean {

// Helper function to check if a substring of 'a' is a palindrome.

// Check from the outside towards the center if characters match

// Check both combinations of strings to see if a palindrome can be formed.

// Check from the outside towards the center if characters match

// Helper function to check if switching between 'a' and 'b' strings can form a palindrome.

while (leftIndex < rightIndex && a.charAt(leftIndex) === b.charAt(rightIndex)) {</pre>

function isPalindromeSubstring(a: string, leftIndex: number, rightIndex: number): boolean {

while (leftIndex < rightIndex && a.charAt(leftIndex) === a.charAt(rightIndex)) {</pre>

// If true, then the substring from 'leftIndex' to 'rightIndex' is a palindrome

Check if it is possible to create a palindrome by taking a prefix from one string and

// If true, the substring is already a palindrome or can form a palindrome with the rest of 'a' or 'b'

return leftIndex >= rightIndex || isPalindromeSubstring(a, leftIndex, rightIndex) || isPalindromeSubstring(b, leftIndex, right

};

TypeScript

endIndex--;

```
while (startIndex < endIndex && str.charAt(startIndex) == str.charAt(endIndex)) {</pre>
            startIndex++;
            endIndex--;
        // If indices have crossed, substring is a palindrome
        return startIndex >= endIndex;
C++
class Solution {
public:
    // Public method to check if a palindrome can be formed from two strings
    bool checkPalindromeFormation(string a, string b) {
        // It checks both combinations: a|b and b|a
        return checkPalindrome(a, b) || checkPalindrome(b, a);
private:
    // Helper method to check if a palindrome can be formed by prefixing and suffixing substrings from two strings
    bool checkPalindrome(string& prefix, string& suffix) {
        int left = 0;
        int right = suffix.size() - 1;
        // Move towards the middle from both ends if corresponding characters match
        while (left < right && prefix[left] == suffix[right]) {</pre>
            ++left;
            --right;
        // Check if we have reached the middle or if either half is a palindrome
        return left >= right || isPalindrome(prefix, left, right) || isPalindrome(suffix, left, right);
    // Helper method to check if the substring of a is a palindrome within the given interval [left, right]
    bool isPalindrome(string& str, int left, int right) {
        // Continue checking for palindrome within the interval
        while (left < right && str[left] == str[right]) {</pre>
            ++left;
            --right;
        // If we reached the middle, then it's a palindrome
```

return isPalindromeAfterSwitch(a, b) || isPalindromeAfterSwitch(b, a); class Solution: def checkPalindromeFormation(self, a: str, b: str) -> bool:

check1 and check2.

Time Complexity:

between the pointers in either string is a palindrome.

```
def check_palindrome(a: str, b: str) -> bool:
           Check if the strings a and b can form a palindrome by switching prefix and suffix at any point.
           i, i = 0, len(b) - 1
           # Loop until the characters at i and j are equal and i is less than j
           while i < i and a[i] == b[j]:</pre>
               i, j = i + 1, j - 1
           # If we've scanned the entire string and it's a palindrome, or if the substring a[i:j+1]
           # or b[i:j+1] is a palindrome, return True
           return i >= j or is_palindrome_substring(a, i, j) or is_palindrome_substring(b, i, j)
       def is palindrome_substring(s: str, start: int, end: int) -> bool:
           Check if the substring s[start:end+1] is a palindrome.
           # Compare the substring with its reverse to check for a palindrome
           return s[start:end+1] == s[start:end+1][::-1]
       # Return True if either combination of a prefix from a and suffix from b, or a prefix from b
        # and suffix from a, forms a palindrome
        return check_palindrome(a, b) or check_palindrome(b, a)
Time and Space Complexity
  The given Python code defines a function checkPalindromeFormation which takes two strings a and b and checks if a
```

palindrome can be formed by taking a prefix of one string and the suffix of the other string. There are two helper functions,

check1 is a function that uses two-pointers to compare the characters from the beginning of one string and the end of the other

string. If the characters match, it moves the pointers inwards. When they stop matching, it calls check2 to check if the substring

1. The while loop in check1 runs in O(n) time in the worst case, where n is the length of the strings, since each character is compared at most once. 2. The check2 function, when called, takes O(k) time, where k is the length of the substring being checked. In the worst case, k is n (the entire

check2 checks if a substring is a palindrome by comparing the substring to its reverse.

string), so check2 has a worst-case time complexity of O(n). 3. The slicing a[i:j+1] and reversing a[i:j+1][::-1] operations in check2 both take O(n) time. 4. The checkPalindromeFormation function calls check1 twice, once with (a, b) and once with (b, a).

The worst-case overall time complexity is the sum of these operations, which is 0(n) + 0(n) + 0(n) + 0(n), simplifying to 0(n).

- **Space Complexity:**
- 1. The two-pointer approach in check1 only uses a constant amount of extra space for the pointers and indices, so it is 0(1). 2. The check2 function uses additional space for the substring and its reverse, which in the worst case could be the entire string, so it is 0(n). 3. No additional data structures are used.

Therefore, the space complexity of the code is primarily determined by the space needed to store the substring and its reverse in check2, which is O(n).