

Solution: Valid Palindrome

Let's solve the Valid Palindrome problem using the Two Pointers pattern.

We'll cover the following

- Statement
- Solution
 - Naive approach
 - Optimized approach using two pointers
 - Step-by-step solution construction
 - Just the code
 - Solution summary
 - Time complexity
 - Space complexity

Statement

Write a function that takes a string, `s`, as an input and determines whether or not it is a palindrome.

Note: A **palindrome** is a word, phrase, or sequence of characters that reads the same backward as forward.

Constraints:

- $1 \leq s.length \leq 2 \times 10^5$
- The string `s` will not contain any white space and will only consist of ASCII characters.

Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

Naive approach

The naive approach to solve this problem is to reverse the string and then compare the reversed string with the original string. If they match, the original string is a valid palindrome. Although this solution has a linear time complexity, it requires extra space to store the reversed string, making it less efficient in terms of space complexity. Therefore, we can use an optimized approach to save extra space.

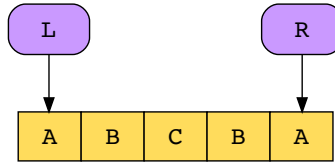
Optimized approach using two pointers

A palindrome is a word or phrase that reads the same way when it is reversed. This means that the characters at both ends of the word or phrase should be exactly the same.

The two-pointers approach would allow us to solve this problem in linear time, without any additional space complexity or the use of built-in functions. This is because we'll traverse the array from the start and the end simultaneously to reach the middle of the string.



Check both ends of the string.
They're both the same.



Sample example 1

1 of 5

Note: In the following section, we will gradually build the solution. Alternatively, you can skip straight to [just the code](#).

Step-by-step solution construction

We'll have two pointers, where the first pointer is at the starting element of our string, while the second pointer is at the end of the string. We move the two pointers towards the middle of the string and, at each iteration, we compare each element. The moment we encounter a nonidentical pair, we can return FALSE because our string can't be a palindrome.

We will construct the solution step by step and the first step is to set up two pointers and move them toward the middle of the string. We can do that with the following code snippet:

Java

```
1 class ValidPalindrome {
2
3     public static String isPalindrome(String s) {
4         System.out.println("String to check: " + s + ". Length of string: " + s.length());
5         int left = 0;
6         int right = s.length() - 1;
7         int i = 1;
8         // The terminating condition for the loop is when both the pointers reach the same element or when they
9         while (left < right) {
10             System.out.println("In iteration " + i + ", left = " + left + ", right = " + right);
11             System.out.println("The current element being pointed to by the left pointer is '" + s.charAt(left)
12             System.out.println("The current element being pointed to by the right pointer is '" + s.charAt(right)
13             left = left + 1; // Heading towards the right
14             right = right - 1; // Heading towards the left
15             i = i + 1;
16             System.out.println(new String(new char[100]).replace('\0', '-'));
17         }
18         System.out.println("Loop terminated with left = " + left + ", right = " + right);
19         return "The pointers have either reached the same index, or have crossed each other, hence we don't need to
20     }
21
22     //Driver code
23     public static void main(String[] arg) {
24         String[] testCases = {
25             "RACECAR",
26             "ABBA",
27             "TART"
28         };
```



Valid Palindrome





That's how we always traverse in opposite directions with the help of two pointers. The termination condition for our code is that the **left** pointer should always be less than the **right** pointer, because the moment they cross each other, we reach the middle of the string and don't need to go any further. We can check how this works with our example.

In the code sample above, we see that in the case of the palindromic strings, at each step in the traversal toward the middle of the string, the characters at both the left and the right indexes are identical. However, with the third test case, "TART" (which isn't a palindromic string), in the second iteration of the loop, we see that our output identifies that the characters at the left and right indexes aren't the same. This observation allows us to add a simple check to our code.

If we encounter a nonidentical pair, we can simply return FALSE, because the string isn't a palindrome, and we don't need to test any further. Otherwise, we're able to traverse to the middle of the string. In this case, we return TRUE, because each element has a match at the expected position in the string.

Java

```
1 class ValidPalindrome {
2
3     public static boolean isPalindrome(String s) {
4         int left = 0;
5         int right = s.length() - 1;
6         System.out.println("The element being pointed to by the left pointer is '" + s.charAt(left) + "'");
7         System.out.println("The element being pointed to by the right pointer is '" + s.charAt(right) + "'");
8         while (left < right) {
9             System.out.println("We check if the two elements are indeed the same, in this case...");
10            if (s.charAt(left) != s.charAt(right)) // If the elements at index left and index right are not equal
11            {
12                System.out.println("The elements aren't the same, hence we return False");
13                return false; // then the symmetry is broken, the string is not a palindrome
14            }
15            System.out.println("They are the same, thus we move the two pointers toward the middle to continue");
16            left = left + 1; // Heading towards the right
17            right = right - 1; // Heading towards the left
18            System.out.println("The new element at the left pointer is '" + s.charAt(left) + "'");
19            System.out.println("The new element at the right pointer is '" + s.charAt(right) + "'");
20        }
21        // We reached the middle of the string without finding a mismatch, so it is a palindrome.
22        return true;
23    }
24
25    //Driver code
26    public static void main(String[] arg) {
27        String[] testCase = {
28            "RACECAR"
29        }
30    }
31}
```






Valid Palindrome

Just the code

Here's the complete solution to this problem:

Java

```
1 class ValidPalindrome {
2
3     public static boolean isPalindrome(String s) {
4         int left = 0;
5         int right = s.length() - 1;
6         while (left < right) {
7             if (s.charAt(left) != s.charAt(right))
8             {
9                 return false;
10            }
11        }
12    }
13}
```



```

11     left = left + 1;
12     right = right - 1;
13 }
14 return true;
15 }
16
17 //Driver code
18 public static void main(String[] arg) {
19     String[] testCase = {
20         "RACEACAR",
21         "A",
22         "ABCDEFGFEDCBA",
23         "ABC",
24         "ABCBA",
25         "ABBA",
26         "RACEACAR"
27     };
28     for (int k = 0; k < testCase.length; k++) {

```



Valid Palindrome

Solution summary

- Initialize two pointers and move them from opposite ends.
- The first pointer starts at the beginning of the string and moves toward the middle, while the second pointer starts at the end and moves toward the middle.
- Compare the elements at each position to detect a nonmatching pair.
- If both pointers reach the middle of the string without encountering a nonmatching pair, the string is a palindrome.

Time complexity

The time complexity is $O(n)$, where n is the number of characters in the string. However, our algorithm will only run $(n/2)$ times, since two pointers are traversing toward each other.

Space complexity

The space complexity is $O(1)$, since we use constant space to store two indexes.

[← Back](#)

Valid Palindrome

[Next →](#)

Sum of Three Values

☒ Completed



