<https://leetcode.com/problems/valid-palindrome/description/>

A phrase is a **palindrome** if, after converting all uppercase letters into lowercase letters and removing all non-alphanumeric characters, it reads the same forward and backward. Alphanumeric characters include letters and numbers.

Given a string s, return true**if it is a palindrome, or**false**otherwise**.

**Example 1:**

**Input:** s = "A man, a plan, a canal: Panama"

**Output:** true

**Explanation:** "amanaplanacanalpanama" is a palindrome.

**Example 2:**

**Input:** s = "race a car"

**Output:** false

**Explanation:** "raceacar" is not a palindrome.

**Example 3:**

**Input:** s = " "

**Output:** true

**Explanation:** s is an empty string "" after removing non-alphanumeric characters.Since an empty string reads the same forward and backward, it is a palindrome.

**Constraints:**

1 <= s.length <= 2 \* 10^5

s consists only of printable ASCII characters.

**Attempt 1: 2024-01-03**

**Solution 1: Two Pointers (720min)**

class Solution {

    public boolean isPalindrome(String s) {

        int i = 0;

        int j = s.length() - 1;

        while(i < j) {

            char a = Character.toLowerCase(s.charAt(i));

            char b = Character.toLowerCase(s.charAt(j));

            // Must use 'isLetterOrDigit' to check

            // Test out by:

            // s = "0P", expect return false, if only use 'isLetter' will return true

            if(!Character.isLetterOrDigit(a) && Character.isLetterOrDigit(b)) {

                i++;

            } else if(Character.isLetterOrDigit(a) && !Character.isLetterOrDigit(b)) {

                j--;

            } else if(Character.isLetterOrDigit(a) && Character.isLetterOrDigit(b)) {

                if(a != b) {

                    return false;

                }

                i++;

                j--;

            } else {

                i++;

                j--;

            }

        }

        return true;

    }

}

Time Complexity: O(N)

Space Compelxity: O(1)

**Refer to**

<https://algo.monster/liteproblems/125>

**Problem Description**

The problem requires us to determine whether a given string is a palindrome or not. A phrase is considered a palindrome if it reads the same forwards and backwards, once it has been transformed by turning all uppercase letters into lowercase letters and removing all non-alphanumeric characters (non-alphanumeric characters are anything other than letters and numbers). The input string s needs to be evaluated, and the output should be true if s is a palindrome, false otherwise. The challenge lies in the handling of the string preprocessing and palindrome checking efficiently.

**Intuition**

The intuition behind the solution stems from the definition of a palindrome. To check if a string is a palindrome, one has to compare the characters starting from the ends towards the center, ensuring symmetry on both sides. If at any point the characters do not match, we can immediately conclude that the string is not a palindrome.

However, considering the conditions of this specific problem, we must ignore non-alphanumeric characters and case differences between letters. Implementing this in a solution involves two pointers, one starting from the beginning (i) and the other from the end (j) of the string. We move these pointers inward, skipping any non-alphanumeric characters we encounter.

The key steps include:

Convert characters to lower case before comparison to ignore case differences.

Ignore all non-alphanumeric characters by adjusting pointers and not considering these characters in the comparison.

Move the pointers towards the center (i moving right and j moving left) to inspect each remaining character, comparing them for equality.

If any pair of alphanumeric characters does not match, return false immediately, as it is not a palindrome.

If all the compared characters are equal until the pointers cross or meet, return true because the preprocessed string is a palindrome.

**Solution Approach**

The implementation adheres to a two-pointer technique, which is a common pattern used in problems involving arrays or strings that require scanning or comparing elements from opposite ends towards the center. The functions isalnum() and lower() are used to preprocess characters according to the problem's requirements.

Here is a detailed breakdown of the implementation steps in the reference solution:

Initialize two pointers, i and j, at the beginning and the end of the string s respectively.

Use a while loop to iterate as long as i is less than j. This loop will run until the entire string has been checked or once the characters meet or cross over (which would signify that the checked characters so far are symmetric and the string is a palindrome).

Inside the loop, perform the following steps:

Check if the character at position i is non-alphanumeric using the isalnum() method; if it is, increment i to skip over it.

Check if the character at position j is non-alphanumeric; if it is, decrement j to skip over it.

If both characters at positions i and j are alphanumeric, convert them to lowercase using the lower() method for a case-insensitive comparison.

Compare the preprocessed characters at i and j:

If they are not equal, return false because the string cannot be a palindrome if any two corresponding characters do not match.

If they are equal, move i one position to the right and j one position to the left to continue the symmetric comparison towards the center of the string.

If all alphanumeric characters are symmetric around the center after considering the whole string, return true as the string is a palindrome.

The algorithm's time complexity is O(n), where n is the length of the string since it requires a single pass through the string. The space complexity is O(1), as no additional structures are required; the input string's characters are checked in place using the two pointers.

**Example Walkthrough**

Let's use the string s = "A man, a plan, a canal: Panama" as our example to illustrate the solution approach.

First, we initialize two pointers, i at the beginning (position 0) and j at the end (position 29) of the string.

Our while loop begins. We check if i < j which is true (0 < 29), so we enter the loop.

Inside the loop, we use the following steps:

Check if the character at position i (s[0] = 'A') is alphanumeric. It is alphanumeric, so we don't increment i.

Check if the character at position j (s[29] = 'a') is alphanumeric. It is, so we don't decrement j.

We convert both characters to lowercase and compare them: tolower('A') is equal to tolower('a').

Since they match, we move both pointers: i becomes 1, and j becomes 28.

Repeat the previous step:

The character at the new position i (s[1] = ' ') is not alphanumeric, so we increment i.

The character at the new position j (s[28] = 'm') is alphanumeric, so we do nothing with j.

Now i is 2, and j is 28. Check again for the new i position, which is s[2] = 'm', an alphanumeric character. We don't increment i.

We compare the lowercase versions of characters at i (s[2] = 'm') and j (s[28] = 'm'): they match, so we move i to 3 and j to 27.

We continue this process, skipping spaces, commas, and the colon, until our pointers meet near the center of the string or cross over, which means we would have compared all alphanumeric characters from both ends.

If a mismatch is found before the pointers meet or cross, we return false.

In this example, when i and j finally meet/cross, all characters were symmetrical ignoring spaces and punctuation, so we return true.

Following this approach with our example string "A man, a plan, a canal: Panama", we would find that it is indeed a palindrome according to the given problem description, so the function would correctly output true.

**Java Solution**

class Solution {

    /\*\*

     \* Check if a given string is a palindrome, considering alphanumeric characters only and ignoring cases.

     \*

     \* @param s The input string to be checked for palindrome.

     \* @return A boolean indicating if the input string is a palindrome.

     \*/

    public boolean isPalindrome(String s) {

        // Initialize two pointers

        int leftIndex = 0;

        int rightIndex = s.length() - 1;

        // Continue comparing characters while left index is less than right index

        while (leftIndex < rightIndex) {

            // If the character at the left index is not alphanumeric, move the left pointer to the right

            if (!Character.isLetterOrDigit(s.charAt(leftIndex))) {

                leftIndex++;

            }

            // If the character at the right index is not alphanumeric, move the right pointer to the left

            else if (!Character.isLetterOrDigit(s.charAt(rightIndex))) {

                rightIndex--;

            }

            // If the characters at both indexes are alphanumeric, compare them ignoring case

            else if (Character.toLowerCase(s.charAt(leftIndex)) != Character.toLowerCase(s.charAt(rightIndex))) {

                // If characters do not match, it's not a palindrome

                return false;

            } else {

                // If characters match, move both pointers

                leftIndex++;

                rightIndex--;

            }

        }

        // If all alphanumeric characters were matched successfully, it is a palindrome

        return true;

    }

}

**Time and Space Complexity**

**Time Complexity**

The time complexity of the given code can be analyzed by looking at the number of operations performed in relation to the length of the input string s. The main part of the function is a while loop that continues until the two pointers i and j meet in the middle. Both i and j move at most n/2 steps, where n is the length of s. There are a constant number of operations within each loop iteration (checking whether characters are alphanumeric and whether they are equal). Therefore, the time complexity of this function is O(n).

**Space Complexity**

The space complexity is determined by the amount of extra space used by the algorithm as the input size scales. The given code uses a constant amount of extra space: two integer variables i and j. Regardless of the input size, no additional space that scales with input size is used. Thus, the space complexity of the function is O(1).