***Check for Palindrome in Java***

A string is said to be a palindrome if it is the same if we start reading it from left to right or right to left. So let us consider a string “**str”**, now the task is just to find out with its reverse string is the same as it is.

**Example:**

**Input :** str = "abba"

**Output:** Yes

**Input :** str = "geeks"

**Output:** No

**Naive Approach:**By Reversing the given string and Comparing  
We can check if the given string is a palindrome by comparing the original string with its reversed version.   
Below is the implementation of the above approach:

Java

/\*package whatever //do not write package name here \*/

import java.io.\*;

class GFG {

public static boolean isPalindrome(String str)

{

// Initializing an empty string to store the reverse

// of the original str

String rev = "";

// Initializing a new boolean variable for the

// answer

boolean ans = false;

for (int i = str.length() - 1; i >= 0; i--) {

rev = rev + str.charAt(i);

}

// Checking if both the strings are equal

if (str.equals(rev)) {

ans = true;

}

return ans;

}

public static void main(String[] args)

{

// Input string

String str = "geeks";

// Convert the string to lowercase

str = str.toLowerCase();

boolean A = isPalindrome(str);

System.out.println(A);

}

}

**Output**

false

**Time Complexity:-**The time complexity of the given code is O(n), where n is the length of the input string. This is because the for loop iterates through each character in the string once to create the reverse string.

**Space Complexity:-**The space complexity of the code is O(n), where n is the length of the input string. This is because the reverse string is created and stored in a separate string variable, which takes up space in memory proportional to the length of the input string. In addition, the other variables used in the code (i, str, and ans) take up a constant amount of space that is independent of the input size.

In the above example, if we write **ABba** in place of **abba**, then also we should get the output as **yes**.  So, we must change the case of the string to either lowercase or uppercase before we check it for a palindrome. If we do not do this, we will get unexpected results. This is because the compiler checks the characters based on their **ASCII** value and the **ASCII** value of**A** is not the same as **a**.

**Approach:**Our approach will be that we will first convert the string to lowercase. Then, we will take two pointers **i** pointing to the start of the string and **j** pointing to the end of the string. Keep incrementing **i** and decrementing **j** while **i < j** and at every step check whether the characters at these pointers are the same or not. If not then the string is not a palindrome else it is.

**Example 1:**

Java

// Java program to check whether a

// string is a Palindrome

// Using two pointing variables

// Main class

public class GFG {

// Method

// Returning true if string is palindrome

static boolean isPalindrome(String str)

{

// Pointers pointing to the beginning

// and the end of the string

int i = 0, j = str.length() - 1;

// While there are characters to compare

while (i < j) {

// If there is a mismatch

if (str.charAt(i) != str.charAt(j))

return false;

// Increment first pointer and

// decrement the other

i++;

j--;

}

// Given string is a palindrome

return true;

}

// Method 2

// main driver method

public static void main(String[] args)

{

// Input string

String str = "geeks";

//Convert the string to lowercase

str = str.toLowerCase();

// passing bool function till holding true

if (isPalindrome(str))

// It is a palindrome

System.out.print("Yes");

else

// Not a palindrome

System.out.print("No");

}

}

**Output**

No

**Time Complexity:-**The time complexity of the given code is O(n), where n is the length of the input string. This is because the while loop iterates through half of the string to check if it is a palindrome. Since we only check half of the string, the number of iterations is proportional to n/2, which is O(n).

**Space Complexity:-**The space complexity of the code is O(1), because the code only uses a constant amount of additional space that is independent of the input size. The only variables used in the code are i, j, and str, which each take up a constant amount of space. No additional space is created in the code.

**Example 2:**

Java

// Java Program to check Whether the String is Palindrome

// or Not

// Main class

class GFG {

// Method 1

// Returns true if string is a palindrome

static boolean isPalindrome(String str)

{

// Pointers pointing to the beginning

// and the end of the string

int i = 0, j = str.length() - 1;

// While there are characters to compare

while (i < j) {

// If there is a mismatch

if (str.charAt(i) != str.charAt(j))

return false;

// Increment first pointer and

// decrement the other

i++;

j--;

}

// Given string is a palindrome

return true;

}

// Main driver method

public static void main(String[] args)

{

String str = "geeks";

String str2 = "RACEcar";

//Change strings to lowercase

str = str.toLowerCase();

str2 = str2.toLowerCase();

// For string 1

System.out.print("String 1 :");

if (isPalindrome(str))

System.out.print("It is a palindrome");

else

System.out.print("It is not a palindrome");

// new line for better readability

System.out.println();

// For string 2

System.out.print("String 2 :");

if (isPalindrome(str2))

System.out.print("It is a palindrome");

else

System.out.print("It is not a palindrome");

}

}

**Output**

String 1 :It is not a palindrome

String 2 :It is a palindrome

**Time Complexity:-**The time complexity of the given code is O(n), where n is the length of the input string. This is because the while loop in the isPalindrome method iterates through half of the string to check if it is a palindrome. Since we only check half of the string, the number of iterations is proportional to n/2, which is O(n).

**Space Complexity:-**The space complexity of the code is O(1), because the code only uses a constant amount of additional space that is independent of the input size. The only variables used in the code are i, j, str, and str2, which each take up a constant amount of space. No additional space is created in the code.

**Recursive approach:**The approach is very simple. Just like the two-pointer approach, we will check the first and the last value of the string but this time it will be through recursion.

1. We will take two pointers i pointing to the start of the string and j pointing to the end of the string.
2. Keep incrementing i and decrementing j while i < j and at every step
3. Check whether the characters at these pointers are the same or not. We are doing this through recursion – (i+1, j-1
4. If all the characters are the same on the ith and jth index till i>=j condition satisfies, print true else false

Below is the implementation of the above approach:

Java

//// Java program to check whether a

// string is a Palindrome using recursion

import java.io.\*;

class GFG {

public static boolean isPalindrome(int i, int j,

String A)

{

// comparing the two pointers

if (i >= j) {

return true;

}

// comparing the characters on those pointers

if (A.charAt(i) != A.charAt(j)) {

return false;

}

// checking everything again recursively

return isPalindrome(i + 1, j - 1, A);

}

public static boolean isPalindrome(String A)

{

return isPalindrome(0, A.length() - 1, A);

}

public static void main(String[] args)

{

// Input string

String A = "geeks";

// Convert the string to lowercase

A = A.toLowerCase();

boolean str = isPalindrome(A);

System.out.println(str);

}

}

**Output**

false

**Time Complexity:-**The time complexity of the given code is O(n), where n is the length of the input string. This is because the isPalindrome function recursively calls itself for the characters at positions (i+1, j-1) until the pointers i and j cross each other or the characters at the pointers are not equal. Since we compare each character in the string exactly once, the time complexity is O(n).

**Space Complexity:-**The space complexity of the code is O(n), where n is the length of the input string. This is because each recursive call creates a new stack frame that stores the current values of the function parameters and local variables. In the worst case, the function call stack may grow as large as n/2 (when the input string is a palindrome), so the space complexity is O(n).