

In this file, I'll explain my thought process for each method to accomplish each level of the coding test.

Palindrome:

Program.cs

```
private static string reverseString(String s)
{
    char[] array = Regex.Replace(s, @"\s+", "").ToUpper().ToCharArray();
    Array.Reverse(array);
    return new string(array);
}
```

- The shortest way to reverse an input string array is to convert the string into a char array, since the Array class already has a method to reverse elements in an array.
- However, since we don't have anyway to determine if each character will be uppercase or lowercase, we will just convert them all to uppercase.
- The **Regex.Replace** method trims the input string of its whitespaces for the cases of palindrome phrases ("i.e. Step on no pets").
- Method returns the reversed string input

```
public Boolean checkPalindrome(String s)
{
    return Regex.Replace(s, @"\s+", "").ToUpper().Equals(reverseString(s));
}
```

- Since we already have the reversed string from the previous method, the output will just be compared using the **Equals()** method. The **Regex.Replace()** and **ToUpper()** method is also called before comparison since the reversed string has been uppercased and removed its whitespaces.
- This method achieves the Level 1 of the coding test.

```

private String findPalindrome(String original)
{
    left_list = new List<String>();
    right_list = new List<String>();
    int index_of_longest_left = 0, index_of_longest_right = 0;
    String reversed_left = reverseString(original);
    String reversed_right = reverseString(original);
    original = Regex.Replace(original, @"\s+", "").ToUpper();
    String palindrome_left = original.Substring(0, 1);
    String palindrome_right = original.Substring(original.Length - 1, 1);
    for (int a = 1; a <= original.Length; a++)
    {
        if (a == original.Length)
        {
            palindrome_left += " ";
            palindrome_right = " " + palindrome_right;
        }
        else
        {
            palindrome_left += original[a];
            palindrome_right = original[original.Length - 1 - a] + palindrome_right;
        }
        if (!reversed_left.Contains(palindrome_left))
        {
            left_list.Add(palindrome_left.Substring(0, palindrome_left.Length - 1));
            reversed_left = reversed_left.Remove(reversed_left.IndexOf(palindrome_left.Substring(0, palindrome_left.Length - 1)),
                palindrome_left.Substring(0, palindrome_left.Length - 1).Length);
            palindrome_left = palindrome_left.Substring(palindrome_left.Length - 1, 1);
            if (left_list.Count > 1)
            {
                if (left_list[left_list.Count - 1].Length > left_list[index_of_longest_left].Length)
                    index_of_longest_left = left_list.Count - 1;
            }
        }
    }

    if (!reversed_right.Contains(palindrome_right))
    {
        right_list.Add(palindrome_right.Substring(1, palindrome_right.Length - 1));
        reversed_right = reversed_right.Remove(reversed_right.IndexOf(palindrome_right.Substring(1, palindrome_right.Length - 1)),
            palindrome_right.Substring(1, palindrome_right.Length - 1).Length);
        palindrome_right = palindrome_right.Substring(0, 1);
        if (right_list.Count > 1)
        {
            if (right_list[right_list.Count - 1].Length > right_list[index_of_longest_right].Length)
                index_of_longest_right = right_list.Count - 1;
        }
    }

    if (left_list[index_of_longest_left].Length >= right_list[index_of_longest_right].Length)
        return left_list[index_of_longest_left];
    else
        return right_list[index_of_longest_right];
}

```

In this method, for every iteration, we are going to compare palindromes reading from both sides. Since there are possible cases, i.e. the string **aballa**, when reading from the left will have the longest palindrome to be **aba**, while reading from the right the longest palindrome will be **alla**. To get the actual longest string, we will compare from which side produced the longest palindrome.

Since reading from either side will have the same algorithm, I will only explain the one side to be efficient:

- The **left_list** variable of datatype List<String> will store the palindromes/individual letters found on the input string.
- **Index_of_longest_from_left** will be the reference for which element in the list is the longest palindrome. This is the return value of the whole method.
- The **palindrome_left** variable takes the first element of the original string after being removed of the whitespaces.
- The for loop begins at the 2nd character of the original string until it reaches equal to the length of the original string. In the last iteration, we will reach a point where **a** will be equal to **original.length**, and an IndexOutOfRangeException will be thrown and the

rest of the code will not run, and the final iteration and comparison will not take place. To avoid this, we will be adding a whitespace (**palindrome_left += " "**) to go through the final iteration.

- Inside the loop, we will be appending the **palindrome_left** string with the next element in the original string.
- Since we are going through all characters in a string, it is of **linear time complexity**.
- Since there are possibilities of having multiple palindromes in one string, we want to take the following steps:
 1. Continue the loop while the **palindrome_left** string exists in the **reversed_left** string.
 2. When the **palindrome_left** string does not exist in the **reversed_left** string, that would mean that the last added element makes the **palindrome_left** string NOT a palindrome. Then, we would substring the **palindrome_left**, removing the last character, and adding it to the **left_list**.
 3. From the **reversed_left** string, remove the part equal to the **palindrome_left** (not including the last element).
 4. Reinitialize the **palindrome_left** string with the last element.
 5. Add the conditional statement that checks when the **left_list** is not empty.
- After the loop function, we will compare which side produced a longer palindrome, and returns it.

```
- references | 0/1 passing
public String longestPalindrome(String s)
{
    if (findPalindrome(s).Length == 1)
        return "None";
    else
        return findPalindrome(s);
}
```

- This method makes use of the **findPalindrome()** method explained previously. The earlier method returns the index of the list that contains the longest palindrome in the **list**, therefore this method only returns that string element.
- This method achieves the second level of the coding test.

Here is an example with an input of **noonxyzzyxabbad**:

```
Microsoft Visual Studio Debug Console
Iteration [1]: Palindrome Value: NO || Reversed Value: DABBAXYZZYXNOON
Iteration [2]: Palindrome Value: NOO || Reversed Value: DABBAXYZZYXNOON
Iteration [3]: Palindrome Value: NOON || Reversed Value: DABBAXYZZYXNOON
Iteration [4]: Palindrome Value: X || Reversed Value: DABBAXYZZYX
Iteration [5]: Palindrome Value: XY || Reversed Value: DABBAXYZZYX
Iteration [6]: Palindrome Value: XYZ || Reversed Value: DABBAXYZZYX
Iteration [7]: Palindrome Value: XYZZ || Reversed Value: DABBAXYZZYX
Iteration [8]: Palindrome Value: XYZZY || Reversed Value: DABBAXYZZYX
Iteration [9]: Palindrome Value: XYZZYX || Reversed Value: DABBAXYZZYX
Iteration [10]: Palindrome Value: A || Reversed Value: DABBA
Iteration [11]: Palindrome Value: AB || Reversed Value: DABBA
Iteration [12]: Palindrome Value: ABB || Reversed Value: DABBA
Iteration [13]: Palindrome Value: ABBA || Reversed Value: DABBA
Iteration [14]: Palindrome Value: D || Reversed Value: D
Iteration [15]: Palindrome Value:  || Reversed Value:
The longest palindrome in the string is: XYZZYX

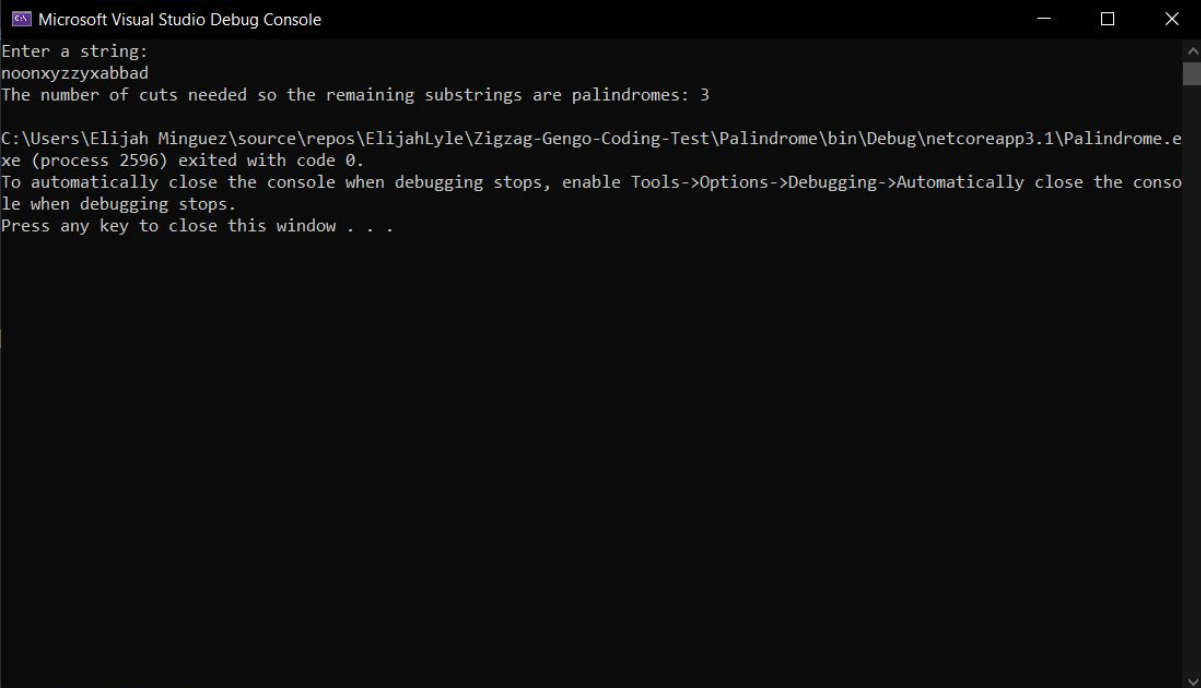
C:\Users\Elijah Minguez\source\repos\ElijahLyle\Zigzag-Gengo-Coding-Test\Palindrome\bin\Debug\netcoreapp3.1\Palindrome.exe (process 2004) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

- If we try to manually find the longest palindrome in this string, we will also come up with the same answer.

```
5 references | 0/1 passing
public int cutPalindrome(String s)
{
    String index = findPalindrome(s);
    int number_of_cuts_from_left = 0, number_of_cuts_from_right = 0;
    foreach (String element in left_list)
        if (element.Length > 1)
            number_of_cuts_from_left++;
    foreach (String element in right_list)
        if (element.Length > 1)
            number_of_cuts_from_right++;
    if (number_of_cuts_from_left <= number_of_cuts_from_right)
        return number_of_cuts_from_left;
    else
        return number_of_cuts_from_right;
}
```

- For the third level of the coding test, since we already have a method of finding all the palindromes in a string, we only need to count how many elements in the **list** is not a letter.
- Since we are also iterating for each element in the list, it also has linear time complexity. However, it is twice the complexity since we are looping two lists.

Using the same example **noonxyzzyxabba**d:



```
Microsoft Visual Studio Debug Console
Enter a string:
noonxyzzyxabba
The number of cuts needed so the remaining substrings are palindromes: 3
C:\Users\Elijah Minguéz\source\repos\ElijahLyle\Zigzag-Gengo-Coding-Test\Palindrome\bin\Debug\netcoreapp3.1\Palindrome.exe (process 2596) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```


Palindrome Tests:

Currently, the best way I thought of testing the methods was to compare manual results to program results, which is why I used the method **Assert.AreEqual** for each validation per level of the coding test.

Here are the input strings we will be using:

```
private string input_1 = "abcdcba";
private string input_2 = "abaxyzzyxf";
private string input_3 = "noonabbad";
private string input_4 = "Step on no pets";
```

Level 1 Validation:

```
[TestMethod]
public void Palindrome_Level1()
{
    Assert.AreEqual(true, palindrome.checkPalindrome(input_1));
    Assert.AreEqual(false, palindrome.checkPalindrome(input_2));
    Assert.AreEqual(false, palindrome.checkPalindrome(input_3));
    Assert.AreEqual(true, palindrome.checkPalindrome(input_4));
}
```

Level 2 Validation:

```
[TestMethod]
public void Palindrome_Level2()
{
    Assert.AreEqual("abcdcba".ToUpper(), palindrome.longestPalindrome(input_1));
    Assert.AreEqual("xyzyx".ToUpper(), palindrome.longestPalindrome(input_2));
    Assert.AreEqual("noon".ToUpper(), palindrome.longestPalindrome(input_3));
}
```

Level 3 Validation:

```
[TestMethod]
public void Palindrome_Level3()
{
    Assert.AreEqual(1, palindrome.cutPalindrome(input_1));
    Assert.AreEqual(2, palindrome.cutPalindrome(input_2));
    Assert.AreEqual(2, palindrome.cutPalindrome(input_3));
    Assert.AreEqual(1, palindrome.cutPalindrome(input_4));
}
```