Task Description

A palindrome is a string that reads the same from the left and from the right. For example, *mom* and *tacocat* are palindromes, as are any single-character strings. Given a string, determine the number of its substrings that are palindromes.

Example
The string is *s = 'tacocat'*. Palindromic substrings are *['t', 'a', 'c', 'o', 'c', 'a', 't', 'coc', 'acoca', 'tacocat']*. There are *10* palindromic substrings.

Function Description
Complete the *countPalindromes* function in the editor.

countPalindromes has the following parameter:
string s: the string to analyze

Returns:
int: an integer that represents the number of palindromic substrings in the given string

Constraints

- $1 \le |s| \le 5 \times 10^3$
- *each character of s, s[i] ∈ ['a'-'z']*.

▸ **Input Format For Custom Testing**

▾ **Sample Case 0**

Sample Input

```
STDIN       Function
-----       -----
aaa     →   s = 'aaa'
```

Sample Output

```
6
```

Explanation
There are *6* possible substrings: *'a', 'a', 'a', 'aa', 'aa', 'aaa'*. All of them are palindromes, so return *6*.

▾ **Sample Case 1**

Sample Input

```
STDIN       Function
-----       -----
abccba  →   s = 'abccba'
```

Sample Output

```
9
```

Explanation
There are *21* possible substrings of *s*, the following *9* of which are palindromes: *{'a', 'a', 'b', 'b', 'c', 'c', 'cc', 'bccb', 'abccba'}*.

▸ **Sample Case 2**

Interviewer Guidelines Private

Interviewer guidelines are a set of hints and follow up questions to help you guide and evaluate the candidate.

▾ **Hint 1**

Checkout the constraints on length of string. It seems that $O(n^2)$ should work here.

▾ **Hint 2**

For each substring, try evaluating whether it is palindromic or not. Then you can easily count the number of palindromic substrings. Try dynamic programming!

▾ **Solution**

Concepts covered: Strings, Dynamic Programming
Optimal Solution:
Let's first find out for each substring, whether it is palindromic or not using Dynamic Programming. Create a subproblem $dp_{i,j}$ which is equal to:

- 1, if the substring from i to j is palindromic
- 0, if the substring from i to j is non-palindromic

The base cases here is trivial:

- $dp_{i,i} = 1$, since every string of length 1 is palindromic.
- $dp_{i,i+1} = 1$ if $s_i = s_{i+1}$ otherwise 0.

Now, to evaluate every other state we could use the following recurrence:

$dp_{i,j} = (s_i == s_j)$ & $dp_{i+1,j-1}$

This is because for the substring *s[i..j]* to be palindromic it must follow that s_i is equal to s_j and also the substring *s[i+1..j-1]* is palindromic.

```
def countPalindromes(s):
    n = len(s)
    dp = [[False] * n for _ in range(n)]
    for i in range(n):
        dp[i][i] = True
        if i < n-1 and s[i] == s[i+1]:
            dp[i][i+1] = True
```

Java 8

```
10  import static java.util.stream.Collectors.joining;
11  import static java.util.stream.Collectors.toList;
12
13  class Result {
14
15      public static int countPalindromes(String s) {
16          Map<Character, TreeSet<Integer>> map = new HashMap<>();
17          for(int i =0; i< s.length(); i++){
18              if(!map.containsKey(s.charAt(i)) ){
19                  TreeSet<Integer> indexes = new TreeSet<>();
20                  indexes.add(i);
21                  map.put(s.charAt(i), indexes);
22              }
23              else{
24                  map.get(s.charAt(i)).add(i);
25              }
26
27          }
28          int count = 0;
29          for(Map.Entry<Character,TreeSet<Integer>> entry: map.entrySet()){
30              if(entry.getValue().size()> 1){
31                 Integer[] indeces =  entry.getValue().toArray(new Integer[entry.getValue().size()]);
32                 System.out.println();
33                 for(int i = 0; i< indeces.length; i++){
34                     for(int j = i+1; j< indeces.length; j++){
35                         if( isPalindrome(s.substring(indeces[i], indeces[j] +1))) count++;
36
37                     }
38                 }
39              }
40          }
41          return count + s.length();
42
43      }
44      public static boolean isPalindrome(String s){
45          int len = s.length();
46          int i = 0;
47          int j = len-1;
48          for(; i<len/2;i++,j-- ){
49              if(s.charAt(i) != s.charAt(j) && i!=j){
50                  return false;
51              }
52          }
53          return true;
54      }
55
56  }
```

Line: 26 Col: 13

Chat Window

Compiled Successfully. 12/13 testcases have been passed.

Interviewer Guidelines

SCORECARD

```
    for gap in range(2, n):
        for i in range(n):
            j = i + gap
            if j >= n:
                continue
            dp[i][j] = (s[i] == s[j]) and dp[i+1][j-1]
    ans = 0
    for i in range(n):
        for j in range(i, n):
            ans += int(dp[i][j])
    return ans
```

Brute Force Approach: Passes 12 of 13 test cases

```
def countPalindromes(s):
    n = len(s)
    ans = 0
    for i in range(n):
        for j in range(i, n):
            valid = True
            for k in range(i, (i + j)//2 + 1):
                if s[k] != s[j - (k - i)]:
                    valid = False
                    break
            ans += int(valid)
    return ans
```

Error Handling: There are no edge cases in this problem

▼ Complexity Analysis

Time Complexity - $O(n^2)$.
Since there are $O(n^2)$ subproblems and each subproblems takes constant amount of time for computation, the overall time complexity is $O(n^2)$.

Space Complexity - $O(n^2)$.
For each subproblem we require constant space, hence the space requirement is $O(n^2)$.

▼ Follow up Question 1

Can you find the number of palindromic subsequences instead?
We would again use dynamic programming to find the answer. Let's suppose $dp_{i,j}$ denote the number of palindromic subsequences of the substring $s[i..j]$. We could build $dp_{i,j}$ using the following recurrence: $dp_{i,j} = (s_i == s_j) * dp_{i+1,j-1} + dp_{i+1,j} + dp_{i,j-1} - dp_{i+1,j-1}$

▸ Follow up Question 2