LONGEST PALINDROMIC SUBSEQUENCE

Given a string S, find the longest palindromic substring in S.

A generalisation is the k-common substring problem. Given the set of strings;

$$S = \{S_1, \ldots S_k\}, \text{ where } |S_i| = n_i$$

and $\sum n_i = N$.

Find for each $2 \leq k \leq k$, the longest strings which occurs as substrings of atleast k-strings.

METHOD - 1 ($O(n^3)$)

VERY NAIVE BRUTE FORCE APPROACH

The simple Brute force approach is to check each substring whether the substring is a palindrome or not. It can be done by 3 loops - the outer 2 loops can pick a substring and then the 3rd loop can be used to check if the substring is a Palindrome or not (just check by reversing the string.

Overall Complexity - $O(n^3)$ ✗ not optimal at all

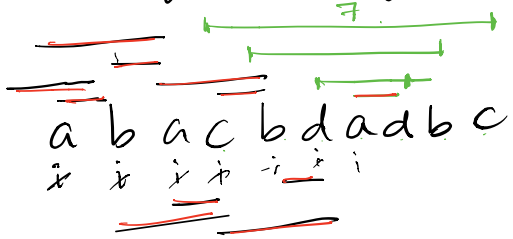
```
LONGEST_PALINDROMIC_SUBSTRING(S):
    max_length ← 0 , best_string = ""
    n ← s.length.
    for i ← 0 to n:                          n^2
        for j ← i to n:
            current_len = j - i + 1
            substring = s[j to current_len]
            if (IS_PALINDROME(substring)     n
                AND current_len > max_length):
                max_length = current_len
                best_string = substring.
    return best_string.

IS_PALINDROME(S):
    S_rev = reverse(S)
    return S_rev == S
```

$n^2 \times n = O(n^3)$

METHOD-2 (RUNTIME $O(n^2)$) Space Optimized — $O(1)$

CHECK SUBSTRINGS USING A PIVOT

The idea is to iterate over the string. And for every i^{th} iteration, check all possible even and odd length of substrings around i and keep count of max_length_palindrome.



PSEUDOCODE:

```
FIND_LONGEST_PALINDROMIC_SUBSTRING (S):

    max_length ← 0
    n ← S.length
    low ← 0 , high ← 0
    // one by one consider every character point
    of even and odd length palindrome.

    for i ← 1 to n:
        // Find the longest EVEN length palindrom
        with i as the pivot.
        low ← i - 1
        high ← i
```

```
WHILE low >= 0
  AND high < n
  AND s[low] == s[high] :

      IF high-low+1 > max_length :
            start = low
            max_length = high-low+1
      low = low - 1
      high = high + 1
```

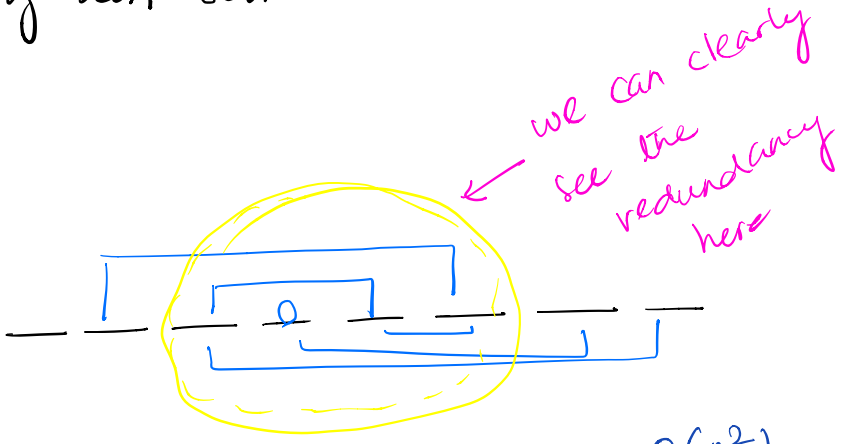
go even

can be put into one CHECK() function

```
// Now do the same with ODD length

low ← i-1
high ← i+1
WHILE low >= 0
   AND high < n
   AND s[low] == s[high] :

      IF high - low + 1 > max_length :
            start = 0
            max_length = high - low + 1

      low = low - 1
      high = high + 1

print "Longest Palindromic Substring is"
print s[start : start + max_length ]
return max_length
```

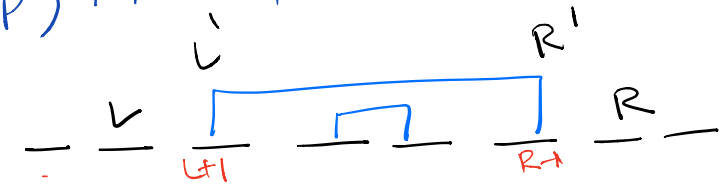
* This approach has some redundancy when it comes to checking substrings. There is an overlap of substrings that have already been searched in the previous iteration.

Ex:



METHOD - 3 (DP) Runtime optimized, space $O(n^2)$

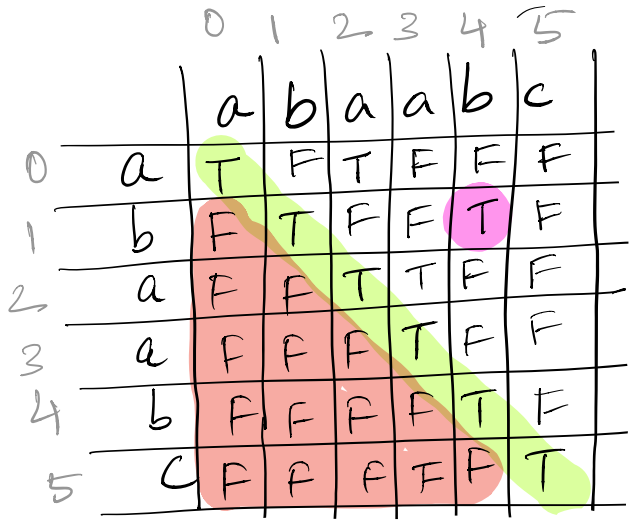
Optimization



So in the optimization using memoization (DP) we can use the previous information for (L', R') to compute if (L, R) is a palindromic substring.

To check if $[L \to R]$ is a good substring we check if:

$[L, R] \longrightarrow [L+1, R-1]$ AND $L == R$.
is a Palindrome



		0 a	1 b	2 a	3 a	4 b	5 c
0	a	T	F	T	F	F	F
1	b	F	T	F	F	T	F
2	a	F	F	T	T	F	F
3	a	F	F	F	T	F	F
4	b	F	F	F	F	T	F
5	c	F	F	F	F	F	T

Longest we can see here is baab ending at index 4.
Hence max_length = 4.

- First we mark DP[L+1][R+1] (all the diagonal elements) as True, because every element is a palindrome of itself.

- Then check for substrings of length 2.

  substring[0][1] = ab
  substring[1][2] = ba
  substring[2][3] = aa
  substring[3][4] = ab
  substring[4][5] = bc

  We can check the above just by comparing that str[L] == str[R].
  And fill in the next diagonal.

- Now for all other substrings of length >, 3 we can use memoized values from DP table for ranges DP[L+1][R-1] and check the following:

```
str[L] == str[R]
AND, DP_table[L+1][R-1] == True.
```

Ex: length = 3

[L+1, R-1]

substring [0][2] = aba

substring [1][3] = baa

substring [2][4] = aab

substring [3][5] = abc

k = 3

i = 0 j = 0+3-1 = 2

i = 1 j = 1+3-1 = 3

i = 2 j = 2+3-1 = 4

i = 3 j = 3+3-1 = 5

i < N-k+1

PSEUDOCODE:

```
T = [[False for i ← 0 to n] for j ← 0 to n]
start = 0
max_length = 1  // as every element is a palindrome of itself.

IF S is empty:  // IF string is empty
    return 0

// Now fill in the diagonals as True, as every element is a palindrom of itself.
For i 0 → N:
    T[i][i] = True.
```

// Now Fill in diagonal for substrings lengths == 2

```
For i in 0 → N-1
    if str[i] == str[i+1]:
        T[i][i+1] = True
        start = i
        max_length = 2
```

// Now Fill in diagonal for substring length >= 3 using the above formula.

```
For k in 3 → N:
    // Fix the starting index. to fill diagonal
    For L in 0 → N-k+1:
        R ← L+k-1

        // Check the substring Lth to Rth
        iff [L+1]th — (R-1)th is a Palindrome

        IF T[L+1][R-1]
        AND str[L] == str[R]:
            T[L][R] == True.

        IF k > max_length:
            start = L
```

max_length = k

return max_length