Correction du DS2

1. Il y a 4 palindromes de longueur 1, 1 palindrome de longueur 2 (`['b','b']`) et 1 palindrome de longueur 3 (`['b','a','b']`) donc au total $\boxed{\text{6 palindromes}}$

2. a) La comparaison de listes est possible ; ne pas oublier le cas particulier des listes vides

```python
def estPalindrome(L) :
    return (len(L)>0) and (L == L[::-1])
```

b) Comme le cas des listes vides est pris en compte, on n'est pas obligé de faire attention à ne pas tester les listes vides (donc une double boucle avec i et j quelconques)

```python
def nbPalindromes(L) :
    n = len(L)
    nb = 0
    for i in range(n) :
        for j in range(n) :
            if estPalindrome(L[i:j]) :
                nb += 1
    return nb
```

c) La fonction `estPalindrome` a une complexité en $O(n)$ car l'opération de slicing et la comparaison des listes sont en $O(n)$. On utilise cette fonction sur la liste `L[i:j]`, donc un nouveau slicing de complexité $O(n)$ qui s'ajoute à celle de `estPalindrome` ($O(n) + O(n) = O(n)$), dans deux boucles imbriquées de longueurs n donc la complexité de `nbPalindromes` est $\boxed{O\left(n^3\right)}$

3. a) Si $j \leqslant i$, la liste `L[i:j]` est vide donc $\boxed{\texttt{P[i][j] = False}}$

b) Un mot est un palindrome si et seulement si le premier et le dernier caractère sont identiques (`L[i] == L[j-1]`) et le mot extrait en supprimant ces deux caractères est aussi un palindrome (`P[i+1][j-1] == True`). Comme $j - i > 2$, le mot `L[i+1:j-1]` n'est pas vide.

c) Il faut donc faire un cas particulier pour $1 \leqslant j - i \leqslant 2$: si $j = i + 1$, le mot est de longueur 1 donc est un palindrome alors que si $j = i+1$, le mot, de longueur 2, est un palindrome si et seulement si `L[i] == L[i+1]`. On compte ensuite le nombre de fois où `P[i][j]` contient la valeur `True`

```python
def palindromes(L) :
    n = len(L)
    P = [[False for j in range(n+1)] for i in range(n+1)]
    for i in range(n) :
        P[i][i+1] = True
    for i in range(n-1) :
        P[i][i+2] = (L[i]==L[i+1])
    for i in range(n) :
        for j in range(i+3,n+1) :
            P[i][j] = (L[i]==L[j-1]) and (P[i+1][j-1])
    nb = 0
    for L in P :
        for l in L :
            if l :
                nb += 1
    return nb
```

d) L'initialisation de `P` est en $O(n^2)$, les deux premières boucles `for` en $O(n)$ puis la double boucle imbriquée en $O(n^2)$. Le parcours de `P` final est aussi en $O(n^2)$ donc la complexité de `palindromes` est en $\boxed{O\left(n^2\right)}$

e) On utilise la mémoïsation : la fonction `f` interne renvoie un plus long palindrome de la liste `L[i:j]`.

```python
def PLP(L) :
    d = {}
    def f(L,i,j) :
        if (i,j) not in d :
            if i == j :
                r = []
            else :
                if estPalindrome(L[i:j]) :
                    r = L[i:j]
```

```python
                    else :
                        r1,r2 = f(L,i,j-1),f(L,i+1,j)
                        if len(r1) > len(r2) :
                            r = r1
                        else :
                            r = r2
                d[(i,j)] = r
            return d[(i,j)]
    return f(L,0,len(L))
```

Même si la question n'était pas posée, on peut évaluer la complexité de cette fonction : elle remplit un dictionnaire dont les clefs sont des couples de $[\![0,n]\!]^2$ donc $O(n^2)$ couples et pour chacun, elle utilise la fonction linéaire `estPalindrome`, la fonction `PLP` a donc une complexité en $O(n^3)$.

4. a) Un palindrome de $\texttt{L}^{\#}$ peut commencer par le symbole # et finit donc aussi par ce symbole, il contient donc $k+1$ fois le symbole # et k lettres. Sinon, il commence et finit par une lettre donc contient k lettres et $k-1$ fois le symbole #. Il est donc toujours de longueur impaire.

b) Si $\ell_p, \dots, \ell_{p+k-1}$ est un palindrome de `L` de longueur k, les deux palindromes de $\texttt{L}^{\#}$ sont $\ell_p, \#, \ell_{p+1}, \dots, \#, \ell_{p+k-1}$, qui est de longueur $(k-1)+k = 2k-1$ et $\#, \ell_p, \#, \ell_{p+1}, \dots, \#, \ell_{p+k-1}, \#$ qui est de longueur $(k+1)+k = 2k+1$.

c) Tout palindrome de `L` est associé à 2 palindromes de $\texttt{L}^{\#}$ et inversement (sauf les $n+1$ `'#'`) donc $\boxed{N^{\#} = 2N + n + 1}$

d) Le nombre de palindromes centré en i est $1 + \hat{\rho}_i$ (ne pas oublier de compter le palindrome de longueur 0)

```python
def NbPalindromes(L) :
    T = Manacher(L)
    nb = 0
    for r in T :
        nb += r+1
    return nb
```

e) Il reste à rajouter la boucle qui va tenter de trouver un palindrome plus grand (boucle `while`) et le test qui permet de mettre à jour la variable `MC` en fonction de la valeur du rayon maximal calculé :

```python
def Manacher(L) :
    n = len(L)
    T = [0 for i in range(n)]
    MC = 0
    for i in range(n) :
        j = i - MC
        if T[MC] >= j :
            T[i] = min(T[MC - j],T[MC] - j)
        r = T[i]+1
        while i-r>=0 and i+r<n and L[i-r]==L[i+r] :
            T[i] = r
            r += 1
        if i + T[i] > MC + T[MC] :
            MC = i
    return T
```

f) Les deux boucles imbriquées semblent donner une complexité en $O(n^2)$ mais un calcul plus précis donne une complexité en $O(n)$: cela vient du fait que si la boucle `while` est exécutée un grand nombre de fois alors le plus long palindrome centré en `MC` sera grand donc ne nécessitera pas d'effectuer la boucle `while` ensuite.

Plus précisément, si k_i est le nombre de fois où la boucle `while` est exécutée pour un entier $i \in [\![1, n-1]\!]$:

— on a $j = i - \texttt{MC}_{i-1}$

— si $j > \texttt{T[MC}_{i-1}\texttt{]}$ donc si $i > t_{i-1}$ (le centre i n'est plus dans le plus grand palindrome centré en $\texttt{MC}_{i-1}$), `T[i]` contient la valeur 0 en entrant dans la boucle `while`, qui est exécutée k_i fois, donc `T[i]` contient la valeur k_i à la fin de la boucle. On a trouvé un meilleur centre donc $\texttt{MC}_i = i$ puis $t_i = i + k_i$ et $t_i - t_{i-1} \geqslant k_i$.

— Si $j \leqslant \texttt{T[MC}_{i-1}\texttt{]}$ et si $\texttt{T[MC}_{i-1}\texttt{-j]} \neq \texttt{T[MC}_{i-1}\texttt{]-j}$ alors la boucle `while` n'est pas exécutée car la condition `L[i-r]==L[i+r]` ne peut pas être satisfaite : selon les cas, si cette condition était remplie, cela contredirait la définition de $\hat{\rho}_{i-j}$ ou de $\hat{\rho}_i$. Dans ce cas, $k_i = 0$, puis $\texttt{MC}_i = \texttt{MC}_{i-1}$ ou $\texttt{MC}_i = \texttt{MC}_{i-1} + 1$ et $t_i - t_{i-1} \geqslant k_i$.

— Dans le dernier cas, $\texttt{T[MC}_{i-1}\texttt{-j]} = \texttt{T[MC}_{i-1}\texttt{]-j}$, on a $\texttt{T[i]} = k_i\texttt{+T[MC}_{i-1}\texttt{]-j} = k_i - i + t_{i-1}$ donc $t_i \geqslant i + \texttt{T[i]} = k_i + t_{i-1}$. Donc à nouveau $t_i - t_{i-1} \geqslant k_i$.

On en déduit, avec $t_0 = 0$, $t_{n-1} = \sum_{i=1}^{n-1}(t_i - t_{i-1}) \geqslant \sum_{i=0}^{n-1} k_i$ qui est le nombre total de boucles effectuées. Comme $t_{n-1} = \texttt{MC}_{n-1} + \texttt{T[MC}_n\texttt{]} \leqslant 2n$, le nombre de boucles est majoré par $2n$ donc une complexité en $\boxed{O(n)}$