TP NOTÉ: PLUS LONG PALINDROME

On veut calculer la position (indice de la case de début) et la longueur du plus long palindrome d'une chaîne de caractères T de longueur N>0.

Par exemple :

- ↙ pour T= « **ada**babcdca » les fonctions à écrire doivent renvoyer le tuple (0,3)
- ↙ pour T= « acb**aa**cba » le résultat est (3,2)

Il s'agit de coder en Python 3 les algorithmes associés aux différentes stratégies exposées ci-dessous.

Dans chaque cas, on calculera la complexité temporelle et spatiale (précisément il s'agit de la complexité asymptotique et dans le pire des cas).

Enfin on mesurera le temps d'exécution des diverses fonctions pour les chaîne de caractères présente dans les fichier plp.txt.

Vous pouvez écrire toutes les fonctions que vous jugerez nécessaires.

Barème :

- fonctions : V1: 2 V2 : 4 V3 : 5 V4 : 1
- Tests de validation : 3 vous générez un grand nombre de chaînes et comparez les divers résultats
- Calculs de complexité temporelle : 3 sur papier, déposer un pdf
- Mesure de temps et comparaison : 2 avec les chaînes des fichiers plp.txt

Vous rendrez un fichier py dans lequel vous noterez les temps d'exécution pour la chaîne du fichier plp1000.txt avec une machine Ubuntu de la salle de TP.

Attention : L'usage de la fonction python reversed() est interdit dans votre code. Vous pouvez seulement l'utiliser à l'intérieur d'assert.

V1 : Force brute

On parcourt tous les tronçons du tableau, avec deux boucles imbriquées. Chaque tronçon est testé pour établir s'il est un palindrome ou non et éventuellement mettre à jour la longueur du plus grand palindrome rencontré.

V2 : Programmation dynamique

On va diminuer la complexité en mémorisant des résultats intermédiaires. On crée une matrice booléenne m et on la remplit de telle sorte que :
Pour $0 \leq i \leq j \leq N-1$, on a :
la case m[i][j] = 1 si le slice T[i:j+1] est un palindrome (de longueur j - i + 1),
et sinon m[i][j] = 0.

Les autres cases de la matrice ne seront pas utilisées.

Pour i<j+1, la valeur de m[i][j] se déduit aisément de la valeur de m[i + 1][j -1] :

- m[i][j]= 1 si m[i + 1][j -1]=1 et T[i]=T[j]
- sinon, m[i][j] = 0.

L'algorithme consiste à :

1. Initialiser la diagonale de D par des 1 (tous les slices de longueur 1 sont des palindromes)
2. Initialiser ensuite toutes les cases de D qui concernent les tronçons de longueur 2, c'est à dire tels que T[i] == T[i + 1] (D[i][i + 1] recevra 1 dans ce cas et 0 sinon).
3. Remplir les autres cases de D (i<j-1), selon le principe exposé ci-dessus , en utilisant deux boucles imbriquées .

V3 : Encore de la force brute

Au lieu de s'intéresser aux extrémités des palindromes, on parcourt cette fois tous les centres possibles d'éventuels palindromes. Pour un indice de centre donné c , on progresse vers la gauche et le droite afin d'explorer tous les palindromes centrés en c et on garde le plus long palindrome ainsi rencontré.
Il faut distinguer entre les palindromes de longueurs paires et impaires...

V4 : Manacher

Une version python de l'algorithme de Manacher est ici :
http://www.enseignement.polytechnique.fr/informatique/INF474A/poly/poly_complement.pdf

Pour des explications, vous pouvez aller ici :
https://www.geeksforgeeks.org/manachers-algorithm-linear-time-longest-palindromic-substring-part-1/