TP n°10 : palindromes et anagrammes

Introduction

L'objectif de ce TP est de manipuler des chaînes de caractères. Nous allons nous intéresser aux problématiques des palindromes et des anagrammes :

- Un *palindrome* est un mot qui se lit aussi bien par la gauche que par la droite, par exemple `'rotor'`
- Un *anagramme* d'un mot m est un mot obtenu par permutation des lettres de m. .Par exemple, «neige» et «genie» sont des anagrammes.

Ce TP est largement inspiré par un TP de Mickaël Péchaud, ainsi que par un sujet de concours sur les palindromes. Il sera l'occasion d'aborder les points suivants : manipulation de chaînes de caractères, programmation dynamique, utilisation d'un dictionnaire des occurrences, programmation récursive des permutations d'un mot, utilisation de la méthode de tri `list.sort`.

1 Palindromes

1.1 Tester si un mot est un palindrome

1. Écrire une fonction booléenne `mots_egaux(mot1: str, mot2: str)-> bool` qui teste l'égalité de deux chaînes de caractère `mot1` et `mot2`.

   Dans cette question, on s'interdit d'utiliser l'opérateur `==` sur les chaînes de caractère.

2. Considérons une chaîne de caractères, par exemple `'amour'`, et inversons l'ordre de ses lettres : on obtient `'ruoma'`. Nous dirons que le mot `'ruoma'` est le miroir du mot `'amour'` Écrire une fonction `miroir(mot:str)-> str` qui prend en argument une chaîne de caractère `mot` et qui renvoie son miroir.

3. En déduire une fonction booléenne `est_palindrome(mot: str)-> bool` qui teste si `mot` est un palindrome (on attend un code avec une seule ligne).

4. Proposer une nouvelle version de la fonction précédente, un peu plus efficace, qui comparerait au pire $n/2$ lettres où n est la longueur du mot.

1.2 Palindromes contenus dans un mot, méthode naïve

On cherche les palindromes constitués de lettres consécutives d'un mot. Par exemple, dans le mot `'decalages'`, le palindrome le plus long est `'ala'` qui correspond aux lettres du mot d'indice $4, 5, 6$. On notera `mot[4:7]` cette tranche. Plus généralement, si `mot` est une chaîne de caractères, on notera `mot[i:j]` la tranche dont les indices sont compris entre i et j, j étant exclu. On pourra utiliser librement si besoin les tranches en Python (slicing).

5. Écrire une fonction `plus_long_palindrome(mot:str)-> (int, [str])` qui cherche le ou les plus longs palindromes contenus dans la chaîne `mot`. On demandera qu'elle renvoie un couple (l, t) où t est la liste du ou des plus longs palindromes et l leur longueur. Si n est la longueur de `mot`, on pourra parcourir toutes les tranches `mot[i:j]` pour $i, j \in [\![0, n-1]\!]$ avec $j \geqslant i+1$, et tester si elles sont des palindromes. Par exemple, si `mot = 'baccarrrre'`, cela renverra le couple `(4,['acca','rrrr'])`.

6. Déterminer la complexité de cette fonction.

1.3 Une autre méthode par programmation dynamique

On souhaite améliorer cette complexité. On va pour cela utiliser le paradigme de la programmation dynamique. Étant donné un mot m, on désire programmer une matrice A telle que le coefficient a_{ij} soit égal à 1 si la tranche $m[i : j+1]$ est un palindrome et soit égal à 0 sinon. Cette matrice est donc carrée de taille n où n est la longueur du mot. Par exemple, si `'mot = bacca'`, `mot[2:4] = 'cc'`, c'est un palindrome, donc $a_{2,3} = 1$. En revanche `mot[0:3] = 'bac'` n'est pas un palindrome, donc $a_{0,2} = 0$.

7. Compléter la matrice A suivante correspondante au mot `'bacca'`. On ne remplira que la partie triangulaire supérieure.

	b	a	c	c	a
b			0		
a					
c				1	
c					
a					

Table 1 – La carte du mot 'bacca'

8. Soit m un mot et $m[i : j+1]$ une tranche de longueur au moins 3. A quelles conditions sur $m[i], m[j]$ et $m[i+1 : j]$, le mot $m[i : j+1]$ est-il un palindrome ?

9. En déduire une fonction `carte(mot:str)->[[int]]` qui prend en entrée une chaîne `mot` et renvoie une matrice A de taille $n \times n$ où n est la longueur de `mot` telle que le coefficient a_{ij} soit égal à 1 si la tranche `mot[i:j+1]` est un palindrome et 0 sinon. On pourra compléter la fonction du fichier élève. Attention, il faudra parcourir la matrice du bas vers le haut et de la gauche vers la droite.

10. Écrire une fonction `indice_dernier_un(t:[int])-> int` qui prend en entrée une liste d'entiers qui contient au moins une fois le nombre 1. Elle doit renvoyer l'indice du dernier 1 dans t. Par exemple, si $t = [0, 1, 0, 0, 1, 0, 0]$, cela renverra 4.

11. En déduire une deuxième fonction `plus_long_palindrome_bis(mot:str)-> (int, [str])` qui cherche le ou les plus longs palindromes contenus dans la chaîne `mot`.

12. Déterminer la complexité de cette fonction.

2 Anagrammes

2.1 Une méthode par comptage des occurrences des lettres

13. Écrire une fonction `mot_to_dico(mot: str)-> dict`, qui prend en argument un mot m et renvoie un dictionnaire
    - dont les clés sont les lettres de m
    - et la valeur correspondant à une clé k est le nombre d'occurrences de k dans m.
14. En déduire une fonction booléenne `sont_anagrammes(mot1:str, mot2:str)-> bool` qui teste si le mot `mot1` est un anagramme du mot `mot2` (on pourra utiliser `==` pour comparer l'égalité de deux dictionnaires, avec une complexité temporelle $O(n)$ où n est la taille commune des dictionnaires, et $O(1)$ s'ils sont de tailles différentes).
15. Quelle est sa complexité ?

2.2 Une deuxième méthode en triant les mots

16. Quelle est en pratique la principale différence entre une chaîne de caractères (i.e. un objet de type `str`) et une liste de caractères ?

    Écrire les fonctions de conversions entre ces deux types `string_to_list(mot : str) -> list` et `list_to_string(l : list) -> str`.
17. Écrire une fonction `tri_mot(mot: str)-> str`, qui prend en argument un mot, et renvoie la version triée de ce mot. On pourra utiliser librement la méthode `sort` qui permet de trier une liste de caractère en place. Par exemple, si `t = ['t', 'e', 's']`, après exécution de `t.sort()`, la liste est modifiée et égale à `['e', 's', 't']`.
18. En déduire une nouvelle fonction booléenne `sont_anagrammes2(mot1:str, mot2:str)-> bool` qui teste si le mot `mot1` est un anagramme du mot `mot2`
19. Quelle est sa complexité ?

2.3 Génération de tous les anagrammes d'un mot

On désire programmer tous les anagrammes d'un mot.

20. Écrire une fonction `echange_deux_lettres(mot:str, i:int, j:int) -> str` qui prend en argument un chaîne de caractère `mot` et renvoie une nouvelle chaîne de caractère où les lettres de `mot` d'indices i et j ont été échangées. Par exemple

    ```
    >>> echange_deux_lettres('amour', 1, 3)
    'auomr'
    ```
21. Écrire une fonction *récursive* `permutation_aux(mot:str, i:int)-> [str]` qui prend en argument une chaîne de caractère `mot` de longueur n et un indice $i \in [\![0, n]\!]$ et renvoie la liste des permutations de `mot` laissant fixes les lettres de `mot` d'indices strictement inférieurs à i. Par exemple :

```
>>> permutations_aux('abcd', 1)
['abcd', 'abdc', 'acbd', 'acdb', 'adcb', 'adbc']
```

On ne se souciera pas des éventuels doublons dans le résultat renvoyé.

22. En déduire une fonction booléenne `permutation(mot:str)-> [str]` qui prend en argument une chaîne de caractère `mot` et renvoie la liste de ses permutations.

23. En déduire une fonction `sont_anagrammes3(mot1:str, mot2:str)-> bool`, qui prend en argument deux mots et teste s'ils sont anagrammes l'un de l'autre.

24. Combien d'anagrammes possède un mot de n lettres ? Que pensez-vous de la complexité de cette fonction ?

Annexe

Voici les fonctions et méthodes sur les listes, dictionnaires et deque dont l'usage est autorisé – ainsi que les complexités que vous utiliserez pour les calculs de complexité (en fonction de la longueur de la liste n). Tout autre fonction dont vous auriez besoin doit être implémentée !

Merci à Mickael Péchaud pour son code Latex.

Fonctions et méthodes sur les listes

Opération	Exemple	Complexité
Création d'une liste vide	`l=[]`	$O(1)$
Accès direct	`l[0]`	$O(1)$
Longueur	`len(l)`	$O(1)$
Concaténation	`l1+l2`	$O(n1+n2)$
Ajout en fin de liste	`l.append(1)`	$O(1)$
Suppression en fin de liste	`l.pop()`	$O(1)$
Extraction de tranche	`l[1:10]`	$O(n)$, où n est la longueur de la tranche.
Répétition	`[0]*k`	$O(n)$, où n est la longueur de la liste créée.
Création par compréhension	`[k**2 for k in range(n)]`	$O(n)$ si l'expression est évaluée en temps constant
Copie	`copy(l)`	$O(n)$
Test d'égalité	`l1 == l2`	$O(1)$ si les longueurs sont différentes, $O(n)$ sinon

Fonctions et méthodes sur les chaînes de caractères

Opération	Exemple	Complexité
Création	`s = 'Ma chaîne'`	$O(n)$
Accès direct	`s[0]`	$O(1)$
Longueur	`len(s)`	$O(1)$
Concaténation	`s1+s2`	$O(n1+n2)$
Extraction de tranche	`s[1:10]`	$O(n)$, où n est la longueur de la tranche.

Fonctions et méthodes sur les dictionnaires

Opération	Exemple	Complexité
Création	`d = {cle : valeur}`	$O(1)$
Test d'appartenance d'une clé	`cle in d`	$O(1)$
Ajout d'un couple clé/valeur	`d[cle] = valeur`	$O(1)$
Valeur correspondant à une clé	`d[cle]`	$O(1)$
Égalité de deux dictionnaires	`d1 == d2`	$O(n)/O(1)$ si de tailles différentes
Ensemble des clés	`d.keys()`	$O(n)$