

Devoir surveillé d'informatique

⚠️ Consignes

- On pourra toujours librement utiliser une fonction demandée à une question précédente même si cette question n'a pas été traitée.
- Veillez à présenter vos idées et vos réponses partielles même si vous ne trouvez pas la solution complète à une question.
- La clarté et la lisibilité de la rédaction et des programmes sont des éléments de notation.

□ Exercice 1 : anagrammes

Deux mots *de même longueur* sont anagrammes l'un de l'autre lorsque l'un est formé en réarrangeant les lettres de l'autre. Par exemples :

- *niche* et *chien* sont des anagrammes.
- *epele* et *pelle*, ne sont pas des anagrammes, en effet bien qu'ils soient formés avec les mêmes lettres, la lettre *l* ne figure qu'à un seul exemplaire dans *epele* et il en faut deux pour écrire *pelle*.

Le but de l'exercice est d'écrire une fonction `anagrammes` qui prend en argument deux chaînes de caractères et qui renvoie `True` si ces deux chaînes sont des anagrammes et `False` sinon.

■ Partie I : Une approche récursive

Dans cette partie, on utilise une approche récursive en se ramenant à chaque étape à des mots plus petits.

1. Ecrire une fonction `indice_premier` qui prend en argument un caractère `car` et une chaîne `chaine` et renvoie l'indice de la première occurrence de `car` dans `chaine`. Dans le cas où `car` n'est pas dans `chaine` on renvoie `-1`. Par exemples,

- `indice_premier("c", "niche")` renvoie 2 car le premier `c` de `"niche"` se trouve à l'indice 2.
- `indice_premier("e", "epele")` renvoie 0 car le premier `e` de `"epele"` se trouve à l'indice 0.
- `indice_premier("t", "chien")` renvoie `-1` car il n'y a pas de `t` dans `"chien"`.

```

1 def indice_premier(car, chaine):
2     # On parcourt la chaîne par indice
3     # On renvoie l'indice dès qu'on rencontre car
4     for i in range(len(chaine)):
5         if chaine[i]==car:
6             return i
7     # On renvoie -1 puisque car n'est pas dans chaîne
8     return -1

```

2. Ecrire une fonction `supprime_premier` qui prend en argument un caractère `car` et une chaîne `chaine` et renvoie la chaîne obtenue en supprimant la première occurrence de `car` dans `chaine`. Si `car` n'est pas dans `chaine` alors on renvoie `chaine` sans modification. Par exemples :

- `supprime_premier("c", "niche")` renvoie `"nihe"`.
- `supprime_premier("c", "chien")` renvoie `"hien"`.
- `supprime_premier("l", "Python")` renvoie `"Python"` car comme la lettre `l` n'apparaît pas dans `"Python"` la chaîne n'est pas modifiée.

Indication : on pourra utiliser la question précédente afin de trouver l'indice `i` d'apparition de la première occurrence du caractère à supprimer puis reconstruire la chaîne en supprimant ce caractère (par exemple en utilisant des tranches).

```

1 def supprime_premier(car, chaine):
2     ind = indice_premier(car, chaine)
3     if ind==-1:
4         return chaine
5     else:
6         return chaine[:ind]+chaine[ind+1:]

```

3. En utilisant la fonction précédente, écrire une fonction récursive `anagrammes_rec` qui prend en argument deux chaînes de caractères `chaine1` et `chaine2` et renvoie `True` si ce sont des anagrammes l'une de l'autre et `False` sinon.

Par exemple, `anagrammes_rec("niche","chien")` renvoie `True`.

Indication : on pourra par exemple supprimer le premier caractère de `chaine1` dans `chaine2` puis faire un appel récursif sur les chaînes restantes.

```

1 def anagrammes_rec(chaine1,chaine2):
2     if len(chaine1)==0 and len(chaine2)==0:
3         return True
4     nchaine2 = supprime_premier(chaine1[0],chaine2)
5     # Si la suppression n'a rien donné, alors ce ne sont pas des anagrammes
6     if len(nchaine2)==len(chaine2):
7         return False
8     return anagrammes_rec(chaine1[1:],nchaine2)

```

4. Donner (en la justifiant) la complexité de cette fonction en notant n la taille (commune) des deux chaînes.

A chaque appel de `anagrammes_rec` la taille des chaînes décroît de 1, il y a donc au plus n appels récursifs. Lors de chacun de ces appels récursifs, la boucle de parcourt de la chaîne pour y supprimer un élément fait au plus n tours. La complexité est donc quadratique.

■ Partie II : Une approche itérative

Dans cette partie, on utilise une approche itérative en manipulant les dictionnaires de Python.

1. Ecrire une fonction `cree_dico` qui prend en argument une chaîne de caractères et renvoie un dictionnaire dont les clés sont les caractères composant la chaîne et les valeurs leur nombre d'apparitions.

Par exemple, `cree_dico("epele")` renvoie le dictionnaire `{ 'e':3, 'p':1, 'l':1 }` en effet dans le mot 'epele', 'e' apparaît à trois reprises et 'l' et 'p' chacun une fois.

```

1 def cree_dico(chaine):
2     dico = {}
3     for c in chaine:
4         if c in dico:
5             #caractere déjà présent, on ajoute 1 à son nombre d'occurrence
6             dico[c] = dico[c]+1
7         else:
8             #caractere qui apparait pour la première fois
9             dico[c] = 1
10    return dico

```

2. Ecrire une fonction `egaux` qui prend en argument deux dictionnaires et renvoie `True` si ces deux dictionnaires sont égaux (c'est-à-dire contiennent exactement les mêmes clés avec les mêmes valeurs) et `False` sinon.

Par exemple, `egaux({'e':3, 'p':1, 'l':1 },{'p':1,'e':2,'l':2})` renvoie `False`

A on s'interdit ici d'utiliser le test d'égalité `==` entre deux dictionnaires et on écrira un parcours de dictionnaire.

```

1 def egaux(dico1,dico2):
2     if len(dico1)!=len(dico2):
3         return False
4     for cle in dico1:
5         if (cle not in dico2 or dico1[cle]!=dico2[cle]):
6             return False
7     return True

```

3. Ecrire une fonction `anagrammes_iter` qui prend en argument deux chaînes de caractères et renvoie `True` si ce sont des anagrammes et `False` sinon.

```

1 def anagrammes_iter(chaine1, chaine2):
2     dico1 = cree_dico(chaine1)
3     dico2 = cree_dico(chaine2)
4     return egaux(dico1, dico2)

```

4. Donner (en la justifiant) la complexité de cette fonction en notant n la taille commune des deux chaînes.

La boucle `for` de la fonction `cree_dico` s'exécute n fois mais elle ne contient que des opérations élémentaires, en effet le test d'appartenance à un dictionnaire est une opération élémentaire. Donc `cree_dico` a une complexité en $\mathcal{O}(n)$. Il en est de même pour la fonction testant l'égalité des deux dictionnaires. Donc cette méthode a une complexité linéaire.

□ **Exercice 2** : requête SQL sur une seule table

On considère la base de données `pays_du_monde` contenant une seule table `pays` dont le schéma est donné ci-dessous :

pays	
nom	: TEXT
region	: TEXT
population	: INT
surface	: INT
cotes	: INT
pib	: INT

D'autre part, on précise la signification des champs suivants :

- `population` : le nombre d'habitants du pays.
- `region` : la région du pays (par exemple "Europe de l'ouest")
- `area` : la surface du pays (en km carré).
- `coastline` : la surface côtière du pays, cette valeur vaut 0 lorsque le pays n'a pas d'ouverture sur la mer
- `pib` : le produit intérieur brut par habitant, c'est une mesure de la richesse du pays.

Et on indique que la requête `SELECT DISTINCT region FROM pays` a renvoyé le résultat suivant :

region
Asie
Afrique du nord
Europe de l'est
Europe de l'ouest
Océanie
Afrique sub saharienne
Proche orient
Amérique latine
Amérique du nord

Ecrire les requêtes permettant de :

1. Trouver la population et le produit intérieur brut de la France.

```

1 SELECT population, pib
2 FROM pays
3 WHERE nom = "France";

```

2. Trouver les pays d'Europe (région « Europe de l'est » ou « Europe de l'ouest ») n'ayant pas d'ouverture sur la mer.

```
1 SELECT nom
2 FROM pays
3 WHERE cotes=0 AND (region="Europe de l'est" or region = "Europe de l'ouest");
```

3. Classer par ordre alphabétique les pays de la région « Amérique latine ».

```
1 SELECT nom
2 FROM pays
3 WHERE region="Amérique latine"
4 ORDER BY nom;
```

4. Lister par ordre décroissant du nombre d'habitants les cinq pays les plus peuplés

```
1 SELECT nom
2 FROM pays
3 ORDER BY population DESC
4 LIMIT 5;
```

5. Trouver le pays de la région « Proche orient » le plus riche (ayant le pib le plus élevé).

```
1 SELECT nom, MAX(pib)
2 FROM pays
3 WHERE region="Proche orient";
```

6. Classer le pays de la région « Afrique du nord » par densité décroissante de population (la densité est le rapport entre le nombre d'habitant et la surface)

```
1 SELECT nom, population/surface AS densite
2 FROM pays
3 WHERE region="Afrique du nord"
4 ORDER BY densite DESC;
```

7. Classer les régions par somme du pib décroissante des pays qui les composent.

```
1 SELECT region, SUM(pib) as total_pib
2 FROM pays
3 GROUP BY region
4 ORDER BY total_pib DESC;
```