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 chaines de caractères et qui renvoie `True` si ces deux chaines 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 chaine `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"`.

```python
def indice_premier(car, chaine):
    # On parcourt la chaine par indice
    # On renvoie l'indice dès qu'on rencontre car
    for i in range(len(chaine)):
        if chaine[i]==car:
            return i
    # On renvoie -1 puisque car n'est pas dans chaine
    return -1
```

2. Ecrire une fonction `supprime_premier` qui prend en argument un caractère `car` et une chaine `chaine` et renvoie la chaine obtenue en supprimant la premier 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 chaine 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 chaine en supprimant ce caractère (par exemple en utilisant des tranches).

```python
def supprime_premier(car, chaine):
    ind = indice_premier(car, chaine)
    if ind==-1:
        return chaine
    else:
        return chaine[:ind]+chaine[ind+1:]
```

3. En utilisant la fonction précédente, écrire une fonction récursive `anagrammes_rec` qui prend en argument deux chaines 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 chaines restantes.

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

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

   A chaque appel de `anagrammes_rec` la taille des chaines 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 chaine 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 chaine de caractères et renvoie un dictionnaire dont les clés sont les caractères composant la chaine 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.

```
def cree_dico(chaine):
    dico = {}
    for c in chaine:
        if c in dico:
            #caractere déjà présent, on ajoute 1 à son nombre d'occurence
            dico[c] = dico[c]+1
        else:
            #caractere qui apparait pour la première fois
            dico[c] = 1
    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`
   ⚠ on s'interdit ici d'utiliser le test d'égalité `==` entre deux dictionnaires et on écrira un parcours de dictionnaire.

```
def egaux(dico1,dico2):
    if len(dico1)!=len(dico2):
        return False
    for cle in dico1:
        if (cle not in dico2 or dico1[cle]!=dico2[cle]):
            return False
    return True
```

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

```
def anagrammes_iter(chaine1,chaine2):
    dico1 = cree_dico(chaine1)
    dico2 = cree_dico(chaine2)
    return egaux(dico1,dico2)
```

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

> 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
Oceanie
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.

```
SELECT population, pib
FROM pays
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.

```sql
SELECT nom
FROM pays
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 ».

```sql
SELECT nom
FROM pays
WHERE region="Amérique latine"
ORDER BY nom;
```

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

```sql
SELECT nom
FROM pays
ORDER BY population DESC
LIMIT 5;
```

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

```sql
SELECT nom, MAX(pib)
FROM pays
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)

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

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

```sql
SELECT region, SUM(pib) as total_pib
FROM pays
GROUP BY region
ORDER BY total_pib DESC;
```