Exercice 2 (6 points)

Cet exercice porte sur les listes, les dictionnaires, les fonctions et la récursivité.

Nous souhaitons créer en langage Python un dictionnaire contenant un grand nombre de mots de façon à ce qu'une recherche dans ce dictionnaire soit la plus rapide possible.

Pour cela nous allons créer des groupes de mots, chaque groupe sera une liste Python associée à une clé unique dans le dictionnaire.

Voici un extrait du dictionnaire que nous souhaitons créer (les `...` indiquent des éléments non listés dans cet extrait) :

```
d = {
    44  : ['ABAISSEMENT', 'ADMINISTRATEUR', ..., 'VERSETS'],
    74  : ['ABAISSER', 'ABLATION', ..., 'TROU'],
    243 : ['ABANDON', 'ALLEGRETTO', ..., 'ZIP'],
    36  : ['ABANDONNANT', 'ABOLITIONNISTE', ..., 'VOULAIT'],
    134 : ['ABANDONNE', 'AGNOSTICISME', ..., 'VOIES'],
    40  : ['ABANDONNENT', 'ACCOUCHEUSE', ..., 'YACK'],
    ...
}
```

Chaque clé sera un nombre entier positif et chaque valeur sera une liste de mots.

Partie A

Pour générer ces valeurs de clés, nous utiliserons une fonction dite de *hachage*, c'est-à-dire une fonction qui pour un mot donné calculera la clé qui lui sera associée. Nous choisissons une fonction de *hachage* simple qui consiste à additionner sur un octet les codes ASCII de chaque lettre de ce mot.

Nous n'utiliserons que des lettres majuscules de l'alphabet, nous rappelons ici le code ASCII (en hexadécimal, c'est-à-dire en base 16) de chacune de ces lettres. Chaque valeur en hexadécimal est notée avec le préfixe `0x`, par exemple `0x15` correspond en décimal à $1 \times 16 + 5 \times 1 = 21$.

Table des codes ASCII					
'A' : 0x41	'F' : 0x46	'K' : 0x4B	'P' : 0x50	'U' : 0x55	'Z' : 0x5A
'B' : 0x42	'G' : 0x47	'L' : 0x4C	'Q' : 0x51	'V' : 0x56	
'C' : 0x43	'H' : 0x48	'M' : 0x4D	'R' : 0x52	'W' : 0x57	
'D' : 0x44	'I' : 0x49	'N' : 0x4E	'S' : 0x53	'X' : 0x58	
'E' : 0x45	'J' : 0x4A	'O' : 0x4F	'T' : 0x54	'Y' : 0x59	

Ainsi le mot `'NSI'` aura pour clé `0xEA` puisque : `0x4E + 0x53 + 0x49 = 0xEA`, ou `234` en décimal.

Si la somme des codes ASCII ne tient pas sur un octet, seul l'octet de poids faible est conservé.

Par exemple : la somme des codes `ASCII` des lettres du mot `'ADMINISTRATEUR'` est de `0x42C`, la clé qui lui sera associée sera donc `0x2C`, égale à `44` en décimal.

1. Trouver la valeur exprimée en hexadécimal de la clé qui sera associée au mot `'EW'`.

2. Comparer, sans les calculer, les valeurs des clés qui seront associées aux mots `'SAC'` et `'CAS'`. Justifier la réponse.

Voici le code de la fonction qui calcule cette clé pour un mot donné (nous rappelons que la fonction `ord` renvoie le code ASCII d'un caractère) :

```
1 def code_hachage(mot):
2     somme = ...
3     for caractere in ...:
4         somme = ...
5     return somme % 0x100
```

3. Recopier et compléter les lignes 2, 3 et 4 de la fonction `code_hachage`.

4. Expliquer l'expression `somme % 0x100` dans le code ci-dessus et donner les valeurs possibles pour la clé.

Partie B

Les listes de mots associées à chaque clé sont rangées dans l'ordre alphabétique, ce qui facilitera la recherche d'un mot par la suite.

Nous allons maintenant voir l'écriture d'une fonction permettant l'ajout d'un mot dans une liste de mots en maintenant un ordre alphabétique dans cette liste.

Nous rappelons qu'en langage Python la comparaison de chaînes de caractères est possible à l'aide des opérateurs classiques de comparaison : <, >, ==, >= et <= . Ces opérateurs utilisent l'ordre alphabétique.

Par exemple, l'expression booléenne `'ANNEE' < 'BATEAU'` vaut `True` puisque le mot `'ANNEE'` est placé avant le mot `'BATEAU'` dans l'ordre alphabétique.

Voici le code d'une fonction `ajouter_mot_liste(liste, mot)` qui a pour paramètres `liste`, une liste de chaînes de caractères (ce sont les mots) et `mot`, une chaîne de caractères correspondant au mot que l'on souhaite ajouter à `liste`. Cette fonction modifie `liste` en y ajoutant `mot` selon l'ordre alphabétique. Elle renvoie de plus `liste` après cet ajout.

```
 1 def ajouter_mot_liste(liste, mot):
 2     i = 0
 3     while i < len(liste):
 4         if mot < liste[i]:
 5             # La méthode "insert" permet d'ajouter un
 6             # élément à un indice donné dans "liste",
 7             # les éléments suivants sont décalés.
 8             liste.insert(i, mot)
 9             return liste
10         i = i + 1
11     # La méthode "append" permet d'ajouter un nouvel
12     # élément à la fin de "liste".
13     liste.append(mot)
14     return liste
```

Voici des exemples d'utilisation de cette fonction :

```
>>> ajouter_mot_liste([],'NSI')
['NSI']
>>> ajouter_mot_liste(['NSI'],'PYTHON')
['NSI','PYTHON']
>>> ajouter_mot_liste(['NSI','PYTHON'],'OBJET')
['NSI','OBJET','PYTHON']
>>> ajouter_mot_liste(['NSI','OBJET','PYTHON'],'RAM')
['NSI','OBJET','PYTHON','RAM']
```

5. Déterminer, dans le pire cas, l'ordre de grandeur du nombre de comparaisons entre chaînes de caractères effectuées par un appel à `ajouter_mot_liste`. On exprimera le résultat en fonction de n, le nombre de mots présents dans `liste`.

Nous souhaitons écrire une fonction qui permette l'ajout d'un mot dans le dictionnaire décrit au début de l'exercice (qui associe à chaque clé entière possible une liste de mots).

6. Rappeler avec quelle expression Python on peut tester si une clé c est déjà présente dans un dictionnaire `dico`.

7. Écrire un code pour la fonction `ajouter_mot_dict(dict_mots, mot)` qui a pour paramètres `dict_mots`, un dictionnaire qui associe à chaque clé entière possible une liste de mots, et `mot` le mot à ajouter dans `dict_mots`, et qui réalise cet ajout (mais ne renvoie aucune valeur). On utilisera les fonctions `code_hachage` et `ajouter_mot_liste`.

Partie C

Nous allons maintenant voir comment faire la recherche d'un mot dans notre dictionnaire.

Nous souhaitons une recherche rapide et nous allons profiter du fait que les mots sont classés dans l'ordre alphabétique dans les listes du dictionnaire.

Nous prendrons une approche dichotomique pour écrire une fonction `est_present(liste, mot, debut, fin)` qui a pour paramètres :

- `liste`, une liste de mots classés dans l'ordre alphabétique ;
- `mot`, le mot à rechercher ;
- `debut` et `fin`, les indices entre lesquels la recherche se fait dans `liste`.

Cette fonction renverra `True` si `liste` contient `mot` entre les indices `debut` (inclus) et `fin` (exclus), et renverra `False` sinon.

Voici son code :

```
 1 def est_present(liste, mot, debut, fin):
 2     if debut > fin:
 3         return False
 4     milieu = (debut + fin) // 2
 5     if liste[milieu] > mot:
 6         return est_present(liste, mot, 0, milieu - 1)
 7     elif liste[milieu] < mot:
 8         return est_present(liste, mot, milieu + 1, fin)
 9     else:
10         return True
```

Nous testons cette fonction ainsi :

```
>>> liste_mots = ['FONCTION', 'NSI', 'PYTHON', 'OBJET', 'RAM']
>>> est_present(liste_mots, 'NSI', 0, len(liste_mots))
True
```

8. Donner, lors de l'exécution de l'exemple précédent, les valeurs des paramètres `debut` et `fin` prises lors de chaque appel de la fonction `est_present`.

9. Expliquer pourquoi la méthode dichotomique permet d'effectuer, en général, moins d'opérations qu'une recherche simple qui consisterait à comparer un par un les mots de la liste avec le mot cherché.

10. Donner l'ordre de grandeur du nombre de comparaisons de mots effectuées par la méthode dichotomique sur une liste de longueur n ?

11. Écrire un code pour la fonction `mot_present(dict_mots, mot)` qui a pour paramètres `dict_mots` , un dictionnaire qui associe à chaque clé entière possible une liste de mots, et `mot`, le mot à rechercher dans `dict_mots`, et qui renvoie `True` si le mot est présent et `False` sinon. On utilisera les fonctions `code_hachage` et `est_present`.