TD n° 3 : dictionnaires

Exercice 1. *Comptage d'éléments*

1. Écrire une fonction d'en-tête
   ```
   def comptage(texte: str) -> dict:
   ```
   qui renvoie le dictionnaire qui à chaque caractère présent dans `texte` associe le nombre d'occurrences de ce caractère dans la chaîne `texte`.
2. Quelle est la complexité de `comptage` en fonction de la longueur n de la chaîne `texte` ?

⇝ Faire l'épreuve 3.14 (*Compter les caractères d'un texte*) du challenge.

Exercice 2. *Marchandises disponibles*

Vous êtes un vendeur dont le stock de marchandises est représenté par un dictionnaire dont les clés sont les noms des produits (type `str`) et les valeurs associées le nombre d'exemplaires de ce produit (type `int`). Un client vous passe commande d'une certaine `quantite` d'un `produit` donné.

1. Écrire une fonction d'entête
   ```
   def possible(stock: dict, produit: str, quantite: int) -> bool:
   ```
   qui renvoie `True` si le stock contient assez du `produit` demandé et `False` sinon.
2. Donner une liste de cas qui formeraient un jeu de tests pertinent pour cette fonction.

Exercice 3. *Scrabble*

Le Scrabble est un jeu de lettres où l'on forme des mots et l'on marque des points en faisant la somme des valeurs de chaque lettre composant le mot. Dans la version française, les points sont les suivants :

- 1 points pour A, E, I, L, N, O, R, S, T et U ;
- 2 points pour D, G et M ;
- 3 points pour B, C et P ;
- 4 points pour F, H et V ;
- 8 points pour J et Q ;
- 10 points pour K, W, X, Y, Z.

1. Donner la commande permettant de créer un dictionnaire dont les clés sont les lettres (type `str`) et les valeurs sont les points correspondant suivant le barème ci-dessus. (On pourra bien sûr ne pas le donner en totalité).
2. Écrire une fonction `score` qui prend en argument une chaîne de caractères `mot` (en majuscules sans accent) et qui renvoie la valeur de ce mot. On utilisera bien sûr le dictionnaire défini dans la question précédente.

Exercice 4. *Alphabet radio*

L'alphabet radio permet de transmettre un message sans risque d'ambiguïté en l'épelant et en représentant chaque lettre de l'alphabet par un mot donné. Par exemple le mot BAC sera épelé via « Bravo Alpha Charlie ». On stocke ces correspondances dans une variable globale `CODE` dont voici le début [1] :

```
CODE = {'A': 'Alpha', 'B': 'Bravo', 'C': 'Charlie', 'D': 'Delta', ...}
```

Vous pouvez considérer cette variable globale comme donnée et l'utiliser dans votre fonction.
Écrire une fonction `codage_radio` qui prend en argument une chaîne de caractères `texte` et qui renvoie une chaîne de caractères représentant le codage du `texte` dans cet alphabet radio. Pour simplifier les choses, on suppose que le `texte` en argument n'est constitué que de majuscules et d'espaces.

1. Voir `https://fr.wikipedia.org/wiki/Alphabet_phon%C3%A9tique_de_l%27OTAN` pour la liste complète.

Exercice 5. *Avec des chaînes de caractères*
Dans cet exercice, on pourra réutiliser la fonction `comptage` définie dans l'exercice 1.

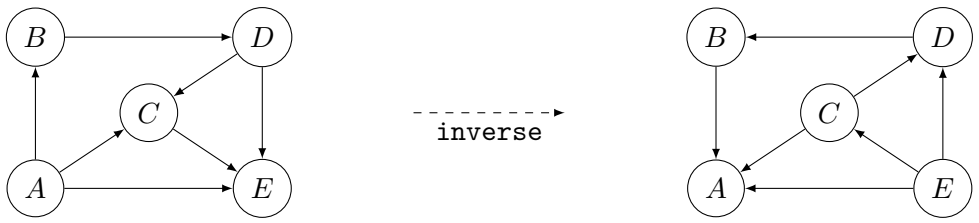
1. Écrire une fonction d'en-tête
   ```
   def plus_frequent(texte: str) -> (str, int):
   ```
   qui renvoie le caractère le plus fréquent dans `texte` ainsi que son nombre d'occurrences (s'il y en a plusieurs, renvoyer n'importe lequel).
2. ★ Modifier la fonction précédente afin qu'elle renvoie la liste (éventuellement réduite à un élément) de tous les caractères les plus fréquents, ainsi que leur nombre commun d'occurrences.
3. Écrire une fonction d'en-tête
   ```
   def compare(chaine1: str, chaine2: str) -> bool:
   ```
   qui renvoie `True` si les deux chaînes de caractères en arguments contiennent les mêmes caractères, pas nécessairement dans le même ordre, et `False` sinon.

Exercice 6. *Créer un dictionnaire à partir de deux listes*

1. Écrire une fonction `creation` qui prend en arguments deux listes `cles` et `valeurs` *supposées de même longueur* et qui renvoie un dictionnaire dont les clés sont les éléments de `cles` et les valeurs correspondantes, celles de `valeurs` de même indice. Par exemple `creation(['a', 'b', 'c'], [4, 5, 6])` doit renvoyer `{'a': 4, 'b': 5, 'c': 6}`.
2. Même question en ne supposant plus maintenant que les deux listes sont de même longueur. S'il n'y a pas assez de clés, simplement ignorer le reste des valeurs. S'il n'y a pas assez de valeurs, mettre `None` comme valeur associée.

Exercice 7. *Graphes*
Dans cet exercice, on considère des graphes orientés que l'on représente à l'aide d'un dictionnaire : les clés sont les sommets du graphe et la valeur associée à une clé s est la liste des sommets qui sont l'extrémité d'une arête partant de s.

1. Donner le dictionnaire représentant le graphe de gauche ci-dessous.



2. Écrire une fonction `inverse` qui prend en paramètre un graphe et qui renvoie le graphe obtenu par inversion du sens de chaque arête. Cette transformation est illustrée ci-dessus.

Exercice 8. *Des paires de chaussettes*
Le tiroir des chaussettes est sens dessus dessous : elles ne sont pas appariées et il en manque peut-être certaines pour faire des paires. On représente les chaussettes dans une liste de chaînes de caractères représentant leur couleur, par exemple :

```
tiroir = ['blanche', 'noire', 'noire', 'bleue', 'noire', 'blanche']
```

Écrire une fonction d'entête

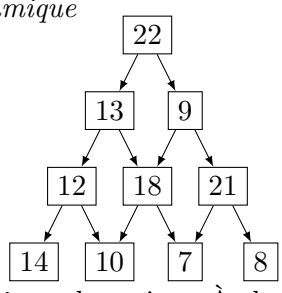
```
def paires(tiroir: [str]) -> int:
```

qui renvoie le nombre de paires de chaussettes de même couleur que l'on peut former. Par exemple, avec la composition du tiroir donnée précédemment, le résultat sera 2 (une paire noire et une paire blanche, les deux restantes ne s'apparient pas).

Exercice 9. ★ *Une fonction à comprendre*
Que fait la fonction suivante ?

```
def truc(nombres: [int], a: int) -> (int, int):
    d = {}
    for i in range(len(nombres)):
        n = nombres[i]
        diff = a - n
        if diff in d:
            return (d[diff], i)
        d[n] = i
```

Exercice 10. *Somme dans une pyramide, introduction à la programmation dynamique*
On considère une pyramide de nombres comme illustrée ci-contre.
On choisit de la représenter par une liste de listes : chaque liste représente un niveau de la pyramide, en commençant par la base. L'exemple ci-contre est ainsi donné par `p = [[14, 10, 7, 8], [12, 18, 21], [13, 9], [22]]`.
En particulier un nombre est repéré par sa position (i, j) où i désigne le numéro de la ligne (en partant de 0 en bas) et j son rang dans la ligne (en partant de 0 à gauche). Par exemple `p[2][1]` est le nombre 9.



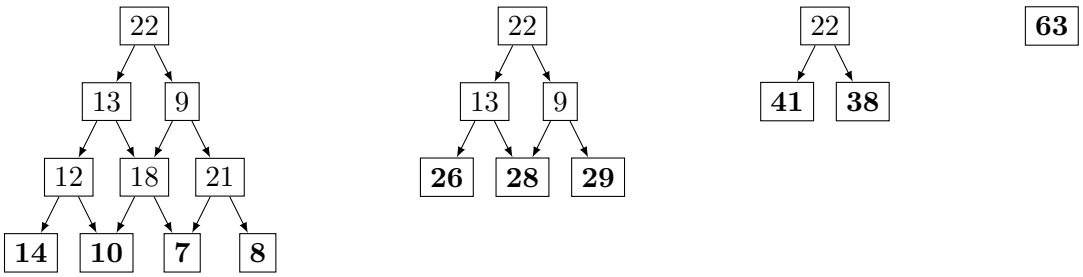
Le but est de trouver la somme maximale possible en partant du sommet et en suivant les arêtes. À chaque étape de « descente » on a donc deux choix : le nombre situé en bas à gauche ou celui en bas à droite du précédent choisi.
Avec la pyramide ci-contre, le maximum vaut 63 et est atteint par le chemin $22 \to 13 \to 18 \to 10$.

1. Donner un exemple de pyramide à 3 étages pour laquelle il ne suffit pas de prendre le maximum à chaque étape pour obtenir le maximum global.
2. Afin de résoudre le problème, on considère la fonction suivante.

```
def somme(p: [[int]], i: int, j: int) -> int:
    """ Renvoie la somme maximale obtenue dans la pyramide p en partant
    de la position (i, j). """
    if i == 0:
        return p[i][j]
    else:
        gauche = somme(p, i-1, j)
        droite = somme(p, i-1, j+1)
        if gauche < droite:
            return p[i][j] + droite
        else:
            return p[i][j] + gauche
```

   a) À quoi correspond le cas de la ligne 5 ?
   b) Expliquer les lignes 8 à 13.
   c) Avec quels arguments doit-on appeler cette fonction pour résoudre le problème posé ?
   d) Quel est l'inconvénient de cette fonction ?

3. Pour y remédier, on va procéder de bas en haut : on va d'abord calculer les valeurs maximales atteignables en démarrant de chaque élément de la ligne du bas, puis de chaque élément du niveau juste au-dessus, et ainsi de suite de proche en proche jusqu'au sommet. Illustrons la méthode :



Afin de ne pas modifier p et de stocker efficacement les calculs intermédiaires, on va utiliser un dictionnaire. On considère la fonction suivante.

```
1  def bas_en_haut(p: [[int]]) -> int:
2      """ Renvoie la somme maximale dans la pyramide p. """
3      n = len(p)
4      memo = {(i,j): p[i][j] for i in range(n) for j in range(n-i)}
5      for i in range(1, n):
6          for j in range(n-i):
7              gauche = ...
8              droite = ...
9              if gauche < droite:
10                 memo[(i,j)] = memo[(i,j)] + droite
11             else:
12                 memo[(i,j)] = memo[(i,j)] + gauche
13     return memo[(len(p)-1, 0)]
```

a) Expliquer ce que contient `memo` après l'exécution de la ligne 4.

b) Compléter les lignes 7 et 8.

c) Donner la complexité de `bas_en_haut` en fonction de la hauteur n de la pyramide.