À faire individuellement ou par petits groupes de deux ou trois.

Exercice 1. Pangramme – Utilisation des ensembles

Un pangramme est une phrase comportant toutes les lettres de l'alphabet. Un exemple en français est: «Portez ce vieux whisky au juge blond qui fume». Un autre exemple couramment utilisé en anglais est: «The quick brown fox jumps over the lazy dog».

a) Recopiez le code ci-après.

```python
fr: str = 'Portez ce vieux whisky au juge blond qui fume'
en: str = 'The quick brown fox jumps over the lazy dog'
```

b) Définissez une fonction **`lower_str(s: str) -> str`** qui prend une chaîne de caractères en entrée et retourne cette même chaîne mais avec uniquement des lettres minuscules.

c) Définissez une fonction **`is_pangram(s: str) -> bool`** qui prend une chaîne de caractères et retourne **`True`** si cette chaîne de caractères est un pangramme, **`False`** sinon. Pour cela, vous pouvez:

- Utiliser un ensemble pour stocker les lettres de la chaîne de caractères.
- Vous assurer de ne pas ajouter les espaces ni les ponctuations dans cet ensemble.

Exercice 2. Ticket de caisse du supermarché – Utilisation des dictionnaires

Voici quelques produits que l'on peut trouver régulièrement en supermarché:

- Pommes: 3.30 CHF/kg
- Bananes: 3.25 CHF/kg
- Fraises: 6.95 CHF/kg
- Tomates: 2.90 CHF/kg
- Poivrons rouges: 3.50 CHF/kg

a) Reprenez le code suivant pour créer un dictionnaire **`products`** dont les clés sont les noms des produits et les valeurs sont les prix au kilogramme.

```python
products: Dict[str, float] = {
    "Pommes": 3.30,
    "Bananes": 3.25,
    "Fraises": 6.95,
    "Tomates": 2.90,
    "Poivrons rouges": 3.50
}
```

b) Définissez une fonction **`shopping_cart(products: Dict[str, float]) -> Dict[str, float]`** qui, pour chaque produit dans **`products`**, demande à l'utilisateur de saisir la quantité (en kilogramme) dans le panier et retourne le panier.

c) Définissez une fonction **`compute_total(...) -> ...`** qui calcule et retourne le prix final du panier. Pour cela, vous aurez besoin du dictionnaire **`products`** ainsi que celui retourné par la fonction **`shopping_cart()`**.

Exercice 3. Suite de Fibonacci – Version récursive et mémoïsation

L'exercice 1 de la série 4 vous a proposé de construire de façon itérative les n premiers éléments de la suite de Fibonacci. Puis, l'exercice 2 de la série 4 du cours de théorie vous a expliqué qu'il était possible de déclarer une fonction récursive calculant le n^{eme} élément de la suite, mais que la complexité de cet algorithme était de l'ordre de 2^n. Pour accélérer cet algorithme, nous pouvons faire appel à la programmation dynamique et à la *mémoïsation*, afin de ne pas recalculer des termes déjà calculés. Cela peut être implémenté grâce aux **`dict`**.

Pour rappel, voici comment la suite de Fibonacci est définie :

$$F(0) = 0$$
$$F(1) = 1$$
$$F(n) = F(n-1) + F(n-2)$$

a) En vous aidant de la solution de l'exercice 2 de la série 4 de la partie théorique, définissez une fonction **`rec_fibonacci(n: int) -> int`** qui calcule de façon récursive le n^{eme} élément de la suite de Fibonacci. Vous pouvez vous aider du code de départ ci-après:

```python
from typing import Dict
from time import time


def rec_fibonacci(n: int) -> int:
    ...

start = time()
print(rec_fibonacci(39))
stop = time()
print(f"Calculated in {stop - start} seconds.")
```

b) La version récursive avec *mémoïsation* va donc reposer sur l'utilisation d'un **`dict`** qui va stocker toutes les nouvelles valeurs. Observez le programme suivant, et en particulier l'utilisation du **`dict`**, et voyez la différence avec votre version sans mémoïsation.

```python
from typing import Dict
from time import time


def rec_fibonacci(n: int) -> int:
    ....

def memo_fibonacci(n: int, previous_terms: Dict[int, int]) -> int:
    if n == 0:
        return 0
    if n == 1:
        return 1
    # Si F(n) n'a jamais été calculé auparavant, alors on le calcule et on stocke sa valeur dans le dictionnaire
    if n not in previous_terms.keys():
        previous_terms[n] = memo_fibonacci(n-1, previous_terms) + memo_fibonacci(n-2, previous_terms)
    return previous_terms[n]

start = time()
print(rec_fibonacci(39))
stop = time()
print(f"Calculated in {stop - start} seconds.")

start = time()
print(memo_fibonacci(39, {})) # La fonction est appelée avec un dictionnaire vide initialement
stop = time()
print(f"Calculated in {stop - start} seconds.")
```