ICC/Programmation — Série 4

CGC/SIE, 10 mars 2025

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

Exercice 1. Création et parcours de listes

Dans cet exercice, nous allons créer une liste, la remplir, puis afficher chacune de ses «cases».

(a) Dans un nouveau fichier, insérez ce début de code:

```
x: int = 10
fibo: list[int] = []
```

Complétez après ligne 4 pour remplir la liste **`fibo`** avec les nombres de la suite de Fibonacci jusqu'à la position **`x`** (exclue). À la position i, on doit trouver la i^{e} valeur de la suite de Fibonacci, soit $F(i)$. La suite de Fibonacci $F(n)$ est définie comme suit:

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

Indice: Initialisez les deux premières valeurs directement. Pour les autres, vous devrez écrire une boucle.

(b) Affichez la liste une fois remplie directement avec un seul **`print()`**.

(c) Avec un **`for-in`** simple, affichez la liste des valeurs avec un **`print()`** par valeur.

(d) Avec un **`for-in`** et un **`enumerate()`**, affichez la liste des valeurs selon ce format:

```
F(0) = 0
F(1) = 1
...
F(9) = 34
```

Exercice 2. Recherche de doublons

(a) Dans un nouveau fichier, complétez le code suivant, trouvable sur Moodle:

```
from random import randint

def has_duplicates(l: list[int]) -> bool:
    ...
```

La fonction **`has_duplicates()`** doit retourner **`True`** si la liste passée en argument contient de valeurs répétées (doublons); **`False`** sinon. Vous pouvez utiliser ce code pour tester:

```
print(has_duplicates([]))                  # False
print(has_duplicates([9, 3, 2, 5, 1]))     # False
print(has_duplicates([1, 1]))              # True
print(has_duplicates([1, 2, 2, 3, 1, 5, 1])) # True
```

(b) Complétez la fonction suivante afin qu'elle retourne la liste complète des doublons observés, et ce, sans doublons dans la liste retournée elle-même. Vous pouvez utiliser le code de test en dessous.

```
def find_duplicates(l: List[int]) -> List[int]:
    ...

print(find_duplicates([]))                  # []
print(find_duplicates([1, 2, 3, 5, 9]))     # []
print(find_duplicates([1, 1]))              # [1]
print(find_duplicates([1, 2, 2, 3, 1, 5, 1])) # [1, 2] ou [2, 1]
```

(c) Écrivez une variante de la fonction **`has_duplicates()`** qui fait appel à la fonction **`find_duplicates()`** pour faire son travail.

Exercice 3. Tri par sélection

L'organisation des données est centrale pour écrire des programmes efficaces: choisir les structures de données adaptées est souvent crucial. En Python, une liste normale n'est pas forcément triée, mais avoir une liste triée a de nombreux avantages — notamment, pour la recherche d'éléments à l'intérieur. Ici, nous allons implémenter un algorithme de tri certes pas très efficace, mais facile à comprendre, le **Selection Sort**, ou tri par sélection.

Il consiste à parcourir la liste dans son ensemble et à trouver le plus petit élément. Cet élément est ensuite déplacé à la position 0, et l'élément à la position 0 est déplacé à la position qu'occupait précédemment le plus petit élément. Ensuite, on cherche le deuxième plus petit élément, donc le plus petit élément à partir de la position 1, et on échange sa position avec celle de l'élément à la position 1. Idem de façon répétée avec les positions 3, 4, ..., $n-2$. Le dernier élément, à la position $n-1$, se retrouve automatiquement à être le plus grand.

Faisons ceci étape par étape.

(a) Complétez le code suivant pour construire une liste de 20 nombres entiers aléatoirement choisis entre 1 et 100 (compris). Indice: grâce à l'**`import`** à la ligne 2, l'expression **`randint(x, y)`** retourne un entier alétoire entre x et y (compris).

```
from random import randint

ints: list[int] = ...
```

(b) Complétez le code suivant de façon à ce qu'un appel à la fonction **`find_min_index()`** retourne l'index de l'élément le plus petit de la liste, en commençant la recherche non pas forcément à 0, mais à l'index passé en paramètre dans **`start_index`**.

```
def find_min_index(l: List[int], start_index: int) -> int:
    ...
```

(c) Modifiez légèrement la déclaration de votre fonction de manière à ce que, lors d'un appel de fonction, l'indication d'une valeur pour **`start_index`** soit optionnelle et que la valeur par défaut **`0`** soit utilisée.

Vous pouvez utiliser ce code pour tester les parties (c) et (d):

```
print(find_min_index([3, 1, 4, 1, 18]))     # 1
print(find_min_index([3, 1, 4, 1, 18], 0))  # 1, équivalent
print(find_min_index([3, 1, 4, 1, 18], start_index=2)) # 3
print(find_min_index([3, 1, 4, 1, 18], start_index=4)) # 4
```

(d) Complétez la fonction **`sort()`** de manière à ce que, par des appels répétés à **`find_min_index()`** et des déplacements d'éléments, elle trie la liste selon le principe expliqué dans le texte introductif de cet exercice.

```
def sort(l: List[int]) -> None:
    ...

print(ints)
sort(ints)
print(ints)
```

Le prochain exercice est sur la page suivante.

Exercice 4. Algorithmique

Cet exercice vous demande faire preuve d'«imagination computationnelle» pour résoudre un problème où vous êtes peu guidés. Essayez de le résoudre sans faire de recherches Google.

Écrivez une fonction **`binary_to_decimal`** qui convertit un string contenant un nombre binaire positif en sa valeur décimale sous forme de **`int`**.

Exemples:

- `"101010"` doit donner **`42`**
- `"10010110"` doit donner **`150`**
- `"11110111110"` doit donner **`1982`**
- `"10000000000000000000000000000000"` doit donner **`1073741824`**

Indice: Quel est la valeur représentée par le dernier caractère ? Et l'avant-dernier ? Et l'avant-avant-dernier ?