ICC/Programmation — Série 4

GC/MX, 13 octobre 2023

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 chacun de ses éléments, et l'afficher en entier.

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

```
1  from typing import List
2
3  x: int = 10
4  fibo: List[int] = []
```

Complétez après la 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. Code tracing – `for` et `while`

Partie papier-crayon

Observez le programme suivant :

```
1  from typing import List
2
3  numbers: List[int] = [1, 1, 1, 3, 5, 5]
4  for index in range(len(numbers)):
5      while numbers.count(numbers[index]) > 1:
6          numbers.remove(numbers[index])
7  print(numbers)
```

a) Quelles sont les variables importantes à *tracer* pour comprendre ce que fait ce programme ?

b) Sans exécuter ce programme, construisez et remplissez le tableau de suivi des variables.

Partie sur ordinateur

a) Recopiez le programme précédent, puis exécutez-le. Une erreur se produit. Quel **type** d'erreur s'est produit selon le message écrit dans le terminal ? Quel pourrait être le sens d'un tel message d'erreur ?

b) Pour vérifier votre hypothèse, placez un *breakpoint* en face de l'instruction **`for`**, puis exécutez le programme pas à pas en mode debug.

c) Avez-vous trouvé le problème dans ce programme entraînant l'erreur ? Si oui, corrigez ce programme en remplaçant la boucle **`for`** par une boucle **`while`**. Si non, demandez de l'aide.

Exercice 3. Recherche de doublons dans une liste

a) Dans un nouveau fichier, reprenez le code suivant et modifiez-le pour compléter la définition de la fonction **`has_duplicates()`**. Cette fonction doit retourner **`True`** si la liste passée en argument contient des valeurs répétés (doublons) ; **`False`** sinon.

```python
from typing import List

def has_duplicates(l: List[int]) -> bool:
    # TO COMPLETE
    ...
```

Pour tester votre fonction, vous pouvez utiliser le code suivant :

```python
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) La fonction ci-après **`find_duplicates()`** prend une liste **`l`** en paramètre et retourne une nouvelle liste qui contient les doublons présents dans **`l`**. La liste retournée ne doit pas contenir de doublons. Pour tester votre fonction, vous pouvez utiliser le code suivant :

```python
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 4. Algorithmes de tris et complexité temporelle

Dans la partie théorique, vous avez rencontré plusieurs algorithmes de tri et évoqué la notion de complexité temporelle. La notion de complexité temporelle peut sembler quelque peu abstraite, pourtant elle a des illustrations très concrètes.

Dans cet *exercice*, il s'agira de rendre explicite les différences de temps d'exécution entre quelques algorithmes de tri, sur des jeux de données relativement grands.

Le code ci-après reprend le contenu du fichier mis à disposition sur Moodle (avec les commentaires et explications en moins pour rendre le document plus lisible).

a) À votre avis, quel sera l'ordre de grandeur du temps d'exécution pour ces deux tris ? Souvenez-vous également que cet ordre de grandeur dépend de la *puissance de calcul* de votre machine.

b) Exécutez le programme et armez-vous d'un peu de patience… *Note:* Si le temps d'exécution est trop long, réduisez la taille des listes créées automatiquement. Dans le programme proposé, les listes sont de taille 12'000.

```python
from typing import List
from time import time
from random import randint

def insertion_sort(l: List[int]) -> List[int]:
    i: int = 1
    while i < len(l):
        j: int = i
        while j > 0 and l[j] < l[j - 1]:
            temp = l[j - 1]
            l[j - 1] = l[j]
            l[j] = temp
            j -= 1
        i += 1
```

```
    return l


def fusion_sort(l: List[int]) -> List[int]:
    if len(l) == 1:
        return l
    m: int = len(l) // 2
    l1: List[int] = fusion_sort(l[:m])
    l2: List[int] = fusion_sort(l[m:])
    return fusion(l1, l2)

def fusion(l1: List[int], l2: List[int]) -> List[int]:
    j1: int = 0
    j2: int = 0
    fusion_list: List[int] = []
    while j1 < len(l1) and j2 < len(l2):
        if l1[j1] <= l2[j2]:
            fusion_list.append(l1[j1])
            j1 += 1
        else:
            fusion_list.append(l2[j2])
            j2 += 1
    if j1 == len(l1):
        fusion_list += l2[j2:]
    else:
        fusion_list += l1[j1:]
    return fusion_list


def create_random_list(size: int) -> List[int]:
    l: List[int] = []
    for i in range(size):
        l.append(randint(0, 3*size))
    return l


trials: int = 20
time_insertion_sort: List[float] = []
time_fusion_sort: List[float] = []

for i in range(trials):
    print(f"Trial number: {i}")
    insertion_list = create_random_list(12000)
    fusion_list = insertion_list.copy()
    print("Working on insertion sort...")
    start_time: float = time()
    insertion_sort(insertion_list)
    stop_time: float = time()
    time_insertion_sort.append(stop_time - start_time)
    print("Insertion sort done...")
    print("Working on fusion sort...")
    start_time = time()
    fusion_sort(fusion_list)
    stop_time = time()
    time_fusion_sort.append(stop_time - start_time)
    print("Fusion sort done...")

print(f"Temps moyen pris pour la tri par insertion: {(sum(time_insertion_sort) / trials):2f} secondes.")
print(f"Le minimum atteint par le tri par insertion est de : {(min(time_insertion_sort)):2f} secondes.")
print(f"Le maximum atteint par le tri par insertion est de : {(max(time_insertion_sort)):2f} secondes.")

print(f"Temps moyen pris pour la tri fusion: {(sum(time_fusion_sort) / trials):2f} secondes.")
print(f"Le minimum atteint par le tri fusion est de : {(min(time_fusion_sort)):2f} secondes.")
print(f"Le maximum atteint par le tri fusion est de : {(max(time_fusion_sort)):2f} secondes.")
```

Exercice 5. Tri par sélection

Suite à l'exercice précédent, nous allons voir encore un autre exemple d'algorithme de tri dit «lent» : le tri par sélection.

Ce tri consiste à parcourir une liste dans son ensemble et à trouver son plus petit élément. Cet élément permute sa place avec l'élément à l'indice 0. Ensuite, on cherche le deuxième plus petit élément, donc le plus petit élément à partir de la position 1, et on permute 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 aléatoire entre x et y (compris).

```
1  from typing import List
2  from random import randint
3
4  ints: List[int] = ...
```

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

```
1  def find_min_index(l: List[int], start_index: int) -> int:
2      ...
```

c) Modifiez légèrement la déclaration de la fonction **`find_min_index()`** 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):

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

d) Complétez la fonction **`selection_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.

```
1  def selection_sort(l: List[int]) -> None:
2      ...
3
4  print(ints)
5  selection_sort(ints)
6  print(ints)
```