Élus, heureux et collés !

INFORMATIQUE COMMUNE - Devoir n° 2 - Olivier Reynet

Consignes :

1. utiliser une copie différente pour chaque partie du sujet,
2. écrire son nom sur chaque copie,
3. écrire de manière lisible et intelligible,
4. préparer une réponse au brouillon avant de la reporter sur la feuille.

R Les parties A,B et C sont indépendantes. Le langage Python est le seul langage informatique autorisé dans les réponses. On s'appliquera à bien respecter les indentations. Il est souvent judicieux, sauf mention contraire dans la question, d'utiliser les fonctions programmées dans les questions précédentes. Les questions marquées ★ sont plus difficiles.

A First To The Post

Le système électoral «First To The Post» (FTTP) est un mode de scrutin utilisé notamment par le Royaume-Uni pour élire les membres du parlement. Les électeurs votent pour un candidat dans leur circonscription. Le candidat ayant obtenu le plus de voix est déclaré vainqueur.

Chaque candidat possède un numéro, c'est-à-dire un entier. S'il y a n candidats dans une circonscription, alors les candidats sont numérotés de 0 à $n-1$.

Un scrutin FTTP est modélisé par une liste d'entiers dont chaque élément représente la voix d'un électeur.

■ **Exemple 1 — Un scrutin FTTP.** Supposons qu'il y a trois candidats pour un poste à pourvoir. Le résultat d'un scrutin FTTP pourrait être `[2, 2, 0, 1, 1, 0, 0, 1, 2, 1, 2, 1]`.

A1. Dans l'exemple 1, combien d'électeurs ont-ils voté ?

> **Solution :** Il suffit de compter le nombre d'éléments de la liste du scrutin puisque chaque élément de cette liste est un voix. Donc 12.

A2. Écrire une fonction de signature `suffrages_exprimes(scrutin: list[int])−> int` qui renvoie le nombre d'électeurs qui se sont exprimés lors de ce scrutin.

Solution :

```python
def suffrages_exprimes(scrutin: list[int]) -> int:
    return len(scrutin)
```

A3. Écrire une fonction de signature `generer_alea_FTTP(n: int,m: int)-> list[int]` qui renvoie une liste d'entiers représentant un scrutin FTTP à n candidats et m électeurs qui s'expriment. Le choix du candidat se fera aléatoirement en utilisant la fonction `randrange` du module `random`. On rappelle que `randrange(n)` renvoie un entier aléatoire tiré **uniformément** entre `0` et `n-1`. Ne pas oublier d'importer correctement la fonction.

Solution :

```python
from random import randrange

def generer_alea_FTTP(n,m):
    scrutin = []
    for j in range(m):
        scrutin.append(randrange(n))
    return scrutin
```

A4. Écrire une fonction de signature `decompter_candidat(scrutin: list[int], c: int)-> int` qui renvoie le nombre de voix qu'a obtenu le candidat `c` pour le scrutin. Par exemple, `decompter_candidat([0, 2, 1, 2, 1, 2, 2, 0, 2, 2],2)` renvoie `6`.

Solution :

```python
def decompter_candidat(scrutin: list[int], c: int) -> int:
    nb = 0
    for suffrage in scrutin:
        if suffrage == c:
            nb += 1
    return nb
```

A5. Écrire une fonction de signature `vmax(L)` qui renvoie l'élément maximum d'une liste s'il existe, `None` sinon. L'usage de la fonction `max` de Python n'est pas autorisé.

Solution :

```python
def vmax(L):
    if len(L) > 0:
        M = L[0]
        for elem in L:
            if elem > M:
                M = elem
        return M
    else:
        return None
```

A6. Écrire une fonction de signature `decompter(scrutin: list[int])-> list[int])` qui renvoie une liste d'entiers représentant les résultats d'un scrutin FTTP. Par exemple, `decompter([2, 2, 0, 1, 1, 0, 0, 1, 2, 1, 2, 1])` renvoie `[3, 5, 4]`, ce qui signifie que le candidat 0 a obtenu 3 voix, le candidat 1 a obtenu 5 voix et que le candidat 2 a obtenu 4 voix. S'inspirer de vos connaissances sur le tri par comptage.

Solution :

```python
def decompter(scrutin: list[int]) -> list[int]:
    n = vmax(scrutin) + 1 # nombre de candidats
    h = [0 for _ in range(n)] #  pour le décompte des occurences
    for k in range(len(scrutin)):
        h[scrutin[k]] += 1
    return h

def decompter(scrutin: list[int]) -> list[int]:
    n = vmax(scrutin) + 1 # nombre de candidats
    h = [0 for _ in range(n)] #  pour le décompte des occurences
    for voix in scrutin:
        h[voix] += 1
    return h
```

A7. Écrire une fonction de signature `vainqueur(scrutin: list[int])-> int` qui renvoie le numéro du candidat vainqueur de l'élection. On tiendra compte de l'efficacité de la fonction : l'objectif est de n'effectuer qu'une seule fois le parcours de la liste `scrutin`. On supposera par ailleurs qu'il n'y a pas de candidats exaequo et que le scrutin n'est pas une liste vide.

Solution :

```python
def vainqueur(scrutin: list[int]) -> int:
    d = decompter(scrutin)
    imax = 0
    for k in range(1, len(d)):
        if d[k] > d[imax]:
            imax = k
    return imax
```

B Heureux

Un nombre est heureux si la fonction `est_heureux` décrite par l'algorithme 1 renvoie la valeur booléenne vrai. L'objectif de cette partie est de coder cette fonction.

Algorithme 1 Vérifier si un nombre est heureux

```
Fonction EST_HEUREUX(n)
    dejà_vu ← ∅                                  ▷ Ensemble des nombres déjà visités
    tant que n ≠ 1 répéter
        si n ∈ dejà_vu alors
            renvoyer faux                        ▷ Le nombre n'est pas heureux
        Ajouter n à dejà_vu
        n ← SOMME_DES_CARRES_DES_CHIFFRES_DE(n)
    renvoyer vrai                                ▷ Le nombre est heureux
```

■ **Exemple 2 — 2008 est heureux.** Appliquons l'algorithme 1 à 2008 :

- Prenons le nombre 2008.
- La somme des carrés de ses chiffres vaut $4+64$, soit 68.
- La somme des carrés des chiffres de 68 vaut $36+64$, soit 100.
- La somme des carrés des chiffres de 100 vaut 1, donc 2008 est un nombre heureux.

R Il existe une infinité de nombres heureux et le problème de savoir si un nombre est heureux est décidable, c'est-à-dire on peut toujours répondre à cette question et on peut le démontrer.

B1. Écrire une fonction de signature `inverser(L)` qui renvoie une liste qui contient les mêmes éléments que `L` mais dans l'ordre inverse. Par exemple, `inverser([3,47,0, 12])` renvoie `[12, 0, 47, 3]`. L'usage de la méthode `L.reverse()` n'est pas autorisé.

Solution :

```python
def inverser(L):
    R = []
    n = len(L)
    for k in range(n):
        R.append(L[n - k - 1])
    return R
```

R Soit n un nombre écrit en base 10. Les divisions euclidiennes successives de n par 10 permettent de trouver la décomposition de n en base 10, c'est-à-dire la liste de ses chiffres.
Tant que le quotient de la division de n par 10 n'est pas nul, on recommence la division euclidienne en prenant le quotient comme dividende et 10 comme diviseur. Cet algorithme permet donc de construire la liste des chiffres qui compose un nombre entier.

■ **Exemple 3 — Trois centaines, six dizaines et sept unités.** Par exemple, prenons $n=367$. On a :

- $367 = 36\times 10+7$, donc 7 est le chiffre des unités.
- $36 = 3\times 10+6$, donc 6 est le chiffre des dizaines.
- $3 = 0\times 10+3$, donc 3 est le chiffre des centaines.

B2. Écrire une fonction de signature `decomposition_b10(n: int)-> list[int]` qui renvoie la liste des chiffres d'un nombre entier en base dix. Par exemple, `decomposition_b10(2307)` renvoie `[2, 3, 0, 7]`.

Solution :

```python
def decomposition_b10(n: int) -> list[int]:
    d = []
    while n != 0:
        d.append(n % 10)
        n = n // 10
    return inverser(d)
```

B3. Écrire une fonction de signature `somme_carres(n: int)-> int` qui renvoie la somme des carrés des chiffres qui compose le nombre entier `n`. Par exemple, `somme_carres(203)` renvoie `13`.

Solution :

```python
def somme_carres(n: int) -> int:
    d = decomposition_b10(n)
    s = 0
    for elem in d:
        s += elem * elem
    return s
```

B4. Le tri par insertion est composé de deux boucles dont l'une permet d'insérer à la bonne place un élément dans une sous-liste bien ordonnée. En vous inspirant de cette boucle, écrire une fonction de signature `inserer(elem: int, L: list[int])` qui insère un élément `elem` dans une liste `L` bien ordonnée dans l'ordre ascendant. Cette fonction travaille en place, directement sur la liste `L`, à laquelle on ajoute donc un élément. Par exemple, si `L` est la liste `[1,3,7,9]`, alors `inserer(4,L)` modifie la liste `L` en `[1, 3, 4, 7, 9]`.

Solution :

```python
def inserer(elem: int, L: list[int]):
    n = len(L)
    L.append(elem)
    j = n
    while L[j - 1] > elem and j > 0:
        L[j] = L[j - 1]
        j = j - 1
    L[j] = elem
```

B5. Écrire une fonction de signature `rech_dicho(L: list[int], elem: int)-> bool` qui renvoie `True` si `elem` appartient à la liste `L` triée dans l'ordre ascendant, `False` sinon. Cette fonction utilise le principe de la recherche par dichotomie.

Solution :

```python
def rech_dicho(L: list[int], elem: int) -> bool:
    n = len(L)
    g = 0
    d = n - 1
    while g <= d:
        m = (g + d) // 2
        if L[m] == elem:
            return True
        elif L[m] < elem:
            g = m + 1
        else:
```

```python
            d = m - 1
    return False
```

B6. 7 est-il un nombre heureux ?

Solution : 7 → 49 → 97 → 130 → 10 → 1. Oui, 7 est heureux ! C'est le plus petit des nombres heureux après 1.

B7. 42 est-il un nombre heureux ?

Solution : 42 → 20 → 4 → 16 → 37 → 58 → 89 → 145 → 42. Non, 42 n'est pas heureux ! On voit qu'on retombe sur 42 et donc, si on continue l'algorithme on retrouvera indéfiniment la même séquence périodique sans rencontrer 1.

B8. Écrire une fonction de signature `est_heureux(n)` qui implémente l'algorithme 1. La variable `deja_vu` sera de type `list[int]`. On veillera à la maintenir triée en utilisant la fonction `inserer` dans le but d'utiliser `rech_dicho`.

Solution :

```python
def est_heureux(n):
    deja_vu = []
    while n != 1:
        if rech_dicho(deja_vu, n):
            return False
        inserer(n, deja_vu)
        n = somme_carres(n)
    return True
```

B9. ★Expliquer ce que calcule la fonction `mystère` : que renvoie-t-elle ? Comment procède-t-elle ?

```python
def mystere(L):
    if len(L) == 0:
        return L
    else:
        return [L[-1]] + mystere(L[:-1])
```

Solution : La fonction mystère inverse la liste L. Elle procède récursivement, en plaçant le dernier élément en première place puis en concaténant le reste de la liste inversée. La condition d'arrêt est atteinte lorsque le reste de la liste est vide, l'inverse d'une liste vide étant la liste vide, pas besoins d'appel récursif.

B10. ★Écrire une fonction de signature `decomposition_b10(n: int)-> list[int]` récursive. On pourra utiliser la concaténation de listes `+`.

Solution :

```python
def decomposition_b10(n: int) -> list[int]:
    if n == 0:
        return []
    else:
        return decomposition_b10(n // 10) + [n % 10]
```

C Collés

L'objectif de cette partie est de modéliser le colloscope afin de le compléter automatiquement moyennant certaines hypothèses simplificatrices.

Une classe de CPGE compte `n` groupes de colle. Le colloscope de la classe comporte `n` créneaux à l'emploi du temps. Un semestre comporte `n` semaines de colles.

Un colloscope est modélisé par une liste de listes. Chaque sous-liste représente un créneau dans l'emploi du temps et contient, dans l'ordre des semaines, les numéros des groupes collés. On suppose qu'il y a **toujours** autant de groupes que de créneaux et on désignera ce nombre par `n`.

C1. Écrire une fonction de signature `colloscope_vide(n: int)-> list[list]` qui renvoie un colloscope vide comportant `n` sous-listes vides.

Solution :

```python
def colloscope_vide(n):
    return [[] for _ in range(n)]
```

C2. Écrire une fonction de signature `groupes_depart(n: int)-> list[int]` qui génère la séquence des entiers de 0 à $n-1$. Par exemple, `groupes_depart(7)` renvoie `[0, 1, 2, 3, 4, 5, 6]`.

Solution :

```python
def groupes_depart(n):
    return [i for i in range(n)]
```

C3. Écrire une fonction de signature `decalage(sequence: list[int])-> list[int]` qui renvoie la permutation circulaire de la séquence passée en paramètre décalée d'un élément vers la gauche. Par exemple, `decalage([0, 1, 2, 3, 4, 5, 6])` renvoie `[1, 2, 3, 4, 5, 6, 0]`.

Solution :

```python
def decalage(sequence: list[int]) -> list[int]:
    L = []
    for i in range(1,len(sequence)):
        L.append(sequence[i])
    L.append(sequence[0])
    return L
```

C4. Écrire une fonction de signature `colloscope(n: int)-> list[list[int]]` qui renvoie un colloscope tel que chaque créneau se voit attribuer un décalage différent de la séquence initiale des groupes. Par exemple, `colloscope(4)` renvoie `[[0, 1, 2, 3], [1, 2, 3, 0], [2, 3, 0, 1], [3, 0, 1, 2]]`.

Solution :

```
def colloscope(n):
    seq = groupes_depart(n)
    C = []
    for i in range(n):
        C.append(seq)
        seq = decalage(seq)
    return C
```

C5. ★Écrire une fonction de signature `grouposcope(colloscope: list[list[int]])-> list[list[int]]` qui renvoie le grouposcope, c'est-à-dire la liste des listes des créneaux pour chaque groupe. Par exemple, `grouposcope(colloscope(4))` renvoie `[[0, 3, 2, 1], [1, 0, 3, 2], [2, 1, 0, 3], [3, 2, 1, 0]]`.

Solution :

```
def grouposcope(colloscope: list[list[int]]) -> list[list[int]]:
    n = len(colloscope)
    G = [[None for _ in range(n)] for _ in range(n)]
    for kc in range(len(colloscope)):
        for kg in range(len(colloscope[kc])):
            groupe = colloscope[kc][kg]
            G[groupe][kc] = kg
    return G
```

On cherche maintenant à vérifier qu'un grouposcope est conforme avant d'en informer les étudiants. Un grouposcope est composé de sous-listes dont les tailles sont toutes les mêmes. Une sous-liste d'un grouposcope vérifie de plus les propriétés suivantes :

- tous ses créneaux sont différents,
- ses créneaux sont numérotés de 0 à $n-1$ si n est la longueur de la liste.

C6. Écrire une fonction de signature `memes_longueurs(g: list[list[int]])-> bool` qui renvoie `True` si toutes les sous-listes d'un grouposcope ont même longueur, `False` sinon.

Solution :

```
def memes_longueurs(g: list[list[int]]) -> bool:
    n = len(g)
    for i in range(n):
        if len(g[i]) != n:  # longueurs identiques = n
            return False
    return True
```

C7. ★Écrire une fonction de signature `tous_differents(sl: list[int])-> bool` qui renvoie `True` si tous les créneaux de colle sont différents pour une sous-liste `sl` d'un grouposcope, `False` sinon. On pourra utiliser un tableau de booléen `deja_vus` pour mémoriser les éléments déjà rencontrés.

Solution :

```
def tous_differents(sl: list[int]) -> bool:
    n = len(sl)
    deja_vu = [False for _ in range(n)]  # mémoriser les éléments déjà vus
    for j in range(n):
        if deja_vu[sl[j]] or sl[j] >= n:
            return False
        else:
            deja_vu[sl[j]] = True
    return True
```

C8. ★Écrire une fonction de signature `est_grouposcope(g: list[list[int]])-> bool` qui renvoie `True` si `g` est un grouposcope, c'est-à-dire s'il vérifie toutes les propriétés ci-dessus, et `False` sinon. On garantira par une assertion que le grouposcope fourni en paramètre n'est pas vide.

Solution :

```
def est_grouposcope(g: list[list[int]]) -> bool:
    assert len(g) > 0
    n = len(g)
    for i in range(n):
        if len(g[i]) != n:  # longueurs identiques = n
            return False
        deja_vu = [False for _ in range(n)] # mémoriser les éléments déjà vus
        for j in range(len(g[i])):
            if deja_vu[g[i][j]] or g[i][j] >= n:
                return False
            else:
                deja_vu[g[i][j]] = True
    return True
```
