Devoir ITC n°1

Recherche d'un élément unique

On dispose d'une liste de $2n-1$ entiers naturels non nuls. Chaque élément de la liste est représenté exactement 2 fois à l'exception d'un entier présent une unique fois. Le but de ce problème est de trouver l'élément de la liste présent une unique fois.

I. Création de la liste

La seule importation autorisée est la fonction `random` du module `random`. L'instruction `random()` renvoie un réel tiré aléatoirement selon une distribution uniforme dans l'ensemble $]0;1[$.

1 – Rédiger une fonction `randentier(a:int, b:int)->int` qui renvoie un entier tiré aléatoirement dans l'ensemble $[\![a;b]\!]$.

```
from random import random
def randentier(a, b):
    x = random()
    y = a + (b-a+1)*x
    return int(y)
```

On crée une liste de n éléments de la façon suivante : le premier élément est tiré dans l'ensemble $[\![1;p]\!]$ avec $p \in \mathbb{N}$; le deuxième dans l'ensemble $[\![1+p;2p]\!]$, et ainsi de suite jusqu'à l'ensemble $[\![1+(n-1)\,p;np]\!]$.

2 – Quelle liste obtient-on si $n=4$ et $p=1$?

```
[1, 2, 3, 4]
```

3 – Proposer une fonction `alea(n:int, p:int)->list` renvoyant une liste de n éléments créés selon ce principe.

```
def alea(n, p):
    liste = []
    for i in range(n):
        a, b = 1 + i*p, (i*1)*p
        liste.append(randentier(a, b))
    return liste
```

Le mélange de Knuth est un algorithme pour générer une permutation aléatoire d'un ensemble fini, c'est-à-dire pour mélanger un ensemble d'objets. Dans le cas d'une liste de n éléments (indicés de 0 à $n-1$), l'algorithme s'écrit en pseudo-code :

```
pour i allant de 0 à n-1 faire :
|    j = entier aléatoire entre 0 et i
|    échanger liste[i] et liste[j]
```

4 – Proposer une procédure `permute(liste:list)` qui permute la liste donnée en argument selon le principe de Knuth. L'application de cette procédure modifie la liste donnée en argument sans avoir besoin d'un `return`.

```python
def permute(liste):
    n = len(liste)
    for i in range(n):
        j = randentier(0, i)
        liste[i], liste[j] = liste[j], liste[i]
```

5 – Sans utiliser la méthode `pop` , proposer une fonction `suppr1list(liste:list)->list` qui renvoie une liste identique à la liste d'entrée mais avec un élément supprimé, choisi aléatoirement.

```python
def suppr1list(liste):
    n = len(liste)
    k = randentier(0, n-1)
    return liste[0:k] + liste[k+1:n]
```

6 – À l'aide des fonctions précédentes, écrire une fonction `creation(n:int, p:int)->list` qui renvoie une liste mélangée de $2n-1$ éléments. Les entiers obtenus sont représentés exactement 2 fois à l'exception d'un entier présent une unique fois.

```python
def creation(n, p):
    liste = alea(n, p)
    liste *= 2
    permute(liste)
    return suppr1list(liste)
```

II. Recherche naïve

7 – À l'aide d'un parcours de liste à double boucle imbriquées, proposer une fonction python naïve d'en-tête `uniquenaif(liste:list)->int` permettant de trouver l'entier unique dans la liste des entiers donnée en argument.

```python
def uniquenaif(liste):
    n = len(liste)
    for i in range(n):
        j = i + 1
        while j < n and liste[i] != liste[j]:
            j += 1
        if j == n:
            return liste[i]
```

8 – Exprimer la complexité de ce code en fonction de n et/ou p

On effectue dans le pire cas $n-i$ tours de la boucle while, qui ne contient que des opérations élémentaires, donc

$$C_{n,p} = \sum_{i=0}^{n-1}(n-i) = n^2 - \frac{n(n-1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n = \mathcal{O}(n^2)$$

III. Recherche par analyse des fréquences

9 – Donner une fonction `maximum(liste:list)->int` renvoyant le plus grand entier de la liste d'entiers naturels donnée en argument.

```
def maximum(liste):
    maxi = liste[0]
    for x in liste:
        if x > maxi:
            maxi = x
    return maxi
```

10 – Rédiger une fonction `indice(liste:list, x:int)->int` qui renvoie l'indice (de la première occurrence) de l'entier `x` dans la liste en argument. On pourra supposer que la présence de x est garanti dans la liste au moins une fois.

```
def indice(liste, x):
    n = len(liste)
    for i in range(n):
        if liste[i] == x:
            return i
```

11 – Écrire une fonction `zeros(taille:int)->list` qui renvoie une liste de taille taille contenant uniquement des zéros.

```
def zeros(taille):
    return [0]*taille
```

12 – Proposer une fonction `freq(liste:list)->list` qui prend en argument d'entrée une liste d'entiers $[a_0, a_1, \dots a_{n-1}]$ et renvoie la liste Fq des fréquences de ces entiers. La liste Fq est de taille $M+1$ où M est le maximum de liste. L'idée de cette liste Fq est de considérer les valeurs a_k comme des indices (entre 0 et M), ainsi $Fq[a_k]$ représente le nombre d'apparitions de l'élément a_k dans la liste `liste`.

Par exemple pour la liste `L = [2, 2, 5, 3, 3, 1, 1]` la fonction `freq` renvoie la liste des fréquences d'apparitions `Fq = [0, 2, 2, 2, 0, 1]`. L'élément d'indice 5 indique 1 ce qui signifie que la valeur 5 n'apparaît que 1 fois dans la liste `L`.

```
def freq(liste):
    m = maximum(liste)
    fq = zeros(m+1)
    for val in liste:
        fq[val] += 1
    return fq
```

13 – À l'aide des fonctions précédentes, en déduire une fonction `uniquef(liste:list)->int` qui renvoie l'élément unique de la liste donnée en argument.

```
def uniquef(liste):
    fq = freq(liste)
    return indice(fq, 1)
```

14 – Exprimer la complexité de ce code en fonction de n et/ou p

La valeur maximale contenue dans la liste est comprise dans l'intervalle $[\![1+(n-1)\,p;np]\!]$; on en déduit dans le pire cas que $M = np$, donc dans la fonction `freq`

- l'appel à `maximum` coûte $\mathcal{O}(n)$
- l'appel à `zeros` coûte $\mathcal{O}(np)$
- la boucle for coûte $\mathcal{O}(n)$

et enfin l'appel à `indice` coûte à nouveau $\mathcal{O}(np)$, donc au total la complexité est de $\mathcal{O}(n) + \mathcal{O}(np) = \mathcal{O}(n(p+1)) \simeq \mathcal{O}(np)$ si $p \gg 1$.

IV. Recherche avec un tri

15 – À l'aide de la fonction `sorted(liste)` qui renvoie une liste triée de la liste donnée en argument, proposer une fonction `uniquetri(liste:list)->int` permettant de renvoyer l'entier unique dans la liste des entiers donnée en argument.

```
def uniquetri(liste):
    liste_triee = sorted(liste)
    n = len(liste)
    for i in range(0, n-1, 2):  # avancée de deux en deux
        if liste_triee[i] != liste_triee[i+1]:
            return liste_triee[i]
    return liste_triee[n-1]  # cas ou l'unique est le dernier
```

16 – Quelle est la complexité de cette fonction ? On indique que la complexité de la fonction `sorted` est en $\mathcal{O}(n\log n)$ pour une liste de n éléments.

Le tri coûte $n\log n$ et le parcours coûte $n/2$ tours de boucle contenant des opértations élémentaires, donc la complexité totale est $\mathcal{O}(n\log n)$.

V. Recherche avec XOR

On utilise habituellement l'écriture en base 10 des entiers : par exemple, 2985 représente le nombre $2\times10^3 + 9\times10^2 + 8\times10^1 + 5\times10^0$. Mais plus généralement, pour tout entier $b \geq 2$ on peut définir la représentation en base b d'un entier.

On définit la représentation en base b d'un entier, en convenant que l'écriture $(a_p a_{p-1}\dots a_0)_b$ représente le nombre :

$$a_p \times b^p + a_{p-1}\times b^{p-1} + \dots + a_1 \times b^1 + a_0 \times b^0$$

Pour s'assurer de l'unicité de l'écriture d'un entier dans une base donnée, il est nécessaire en outre d'imposer : $\forall k \in [\![0;p]\!], a_k \in [\![0;b-1]\!]$ et $a_p \neq 0$.

Ainsi, en base 2 (représentation binaire) seuls les chiffres 0 et 1 sont utilisés, par exemple le nombre $(101)_2$ représente l'entier $1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$, c'est à dire 5. Lorsqu'un nombre est écrit en base 10, par convention on omet de le préciser : $(x)_{10} = x$.

On rappelle ci-dessous les tables logiques du « et » (&), du « ou » (|) et du « ou exclusif » ($\wedge$) :

$a \,\&\, b$	$a=0$	$a=1$
$b=0$	0	0
$b=1$	0	1

$a \mid b$	$a=0$	$a=1$
$b=0$	0	1
$b=1$	1	1

$a \wedge b$	$a=0$	$a=1$
$b=0$	0	1
$b=1$	1	0

On considère par exemple :

- $a = 92$ dont la décomposition en binaire est $(1011100)_2$
- $b = 21$ d'écriture binaire $(10101)_2$

En opérant bit par bit, les différentes opérations donnent :

- $a \,\&\, b = (10100)_2 = 20$
- $a \mid b = (1011101)_2 = 93$
- $a \wedge b = (1001001)_2 = 73$

La fonction « ou exclusif » se nomme également XOR, est une opération commutative, et on peut l'obtenir en python avec `a^b` .

17 – On considère un entier naturel N, que renvoie :

- $0 \wedge N$?
- $N \wedge N$?
- $N \wedge N \wedge N$?

N, 0 et N

---

18 – En se servant des propriétés du XOR évoquées ci-dessus, donner une fonction itérative `uniquexor(liste:list)->int` permettant de trouver l'entier unique dans la liste des entiers donnée en argument. Sa complexité doit être optimale.

```python
def uniquexor(liste: list):
    somme = 0
    for entier in liste:
        somme ^= entier
    return somme
```

---

19 – En admettant que l'opération logique XOR s'effectue en temps constant, quelle est la complexité de la fonction `uniquexor` ? Conclusion.

Il n'y a qu'une boucle for de taille n et contenant des opérations en temps constant, donc sa complexité est $\mathcal{O}(n)$, linéaire. C'est la méthode la plus efficace.
