CORRIGÉ DM INFO n°1 : MINES 2015

Q1. En complément à 2, un bit sert à représenter le signe, et il en reste 9 pour représenter la valeur absolue ; cette valeur absolue peut donc varier entre 0 et $2^9 - 1 = 511$, donc on peut représenter tous les entiers de l'intervalle $[\![-512\,;511]\!]$ (soit 1024 valeurs).

Q2. On peut ici représenter 1024 valeurs de l'intervalle réel $[-5\,;5]$, c'est-à-dire 1023 intervalles chacun de longueur $\dfrac{10}{1023} \approx 0.0098$. La résolution est donc environ de $1.10^{-2}\ V$.

Q3. Le programme demandé est simple :

```
def lect_mesures():
    res = []
    correct = False
    while not correct:
        car = com.read(1)
        if car in ['U', 'I', 'P']:
            correct = True
            res.append(car)
    NbDonnees = int(com.read(3))
    donnees = []
    for in in range(NbDonnees):
        donnees.append(int(com.read(4)))
    res.append(donnees)
    res.append(int(com.read(4))
    return res
```

Q4. *L'énoncé n'est pas très clair : faut-il utiliser une liste* `mesures` *issue de la procédure précédente, ou seulement une liste* `mesure`*, égale à* `mesures[1]`*, contenant les mesures ? C'est cette dernière hypothèse que nous ferons.*

Là encore, le programme est simple et ne demande pas d'explications :

```
def check(mesure, CheckSum):
    somme = sum([abs(x) for x in mesure])
    ''' on peut aussi faire une boucle:
    somme = 0
    for i in range(len(mesure)):
        somme += abs(mesure[i])
    '''
    ''' ou aussi:
    somme = 0
    for x in mesure:
        somme += abs(x)
    '''
    return somme % 10000 == CheckSum
```

Q5. On suppose ici que l'on a déjà écrit : `import matplotlib.pyplot as plt`

```
def affichage(mesure):
    # temps en millisecondes
    T = [2*x for x in range(len(mesure))]
    # intensités en Ampères
    Y = [0.004*x for x in mesure]

    plt.plot(T,Y)
    plt.xlabel('Temps (ms)')
    plt.ylabel('Intensité (A)')
    plt.title('Courant moteur')
    plt.show()
```

Q6. Quelques rappels : soit f une fonction continue par morceaux sur $[a\,;b]$, à valeurs dans $\mathbb{K}$.

Soit $n \in \mathbb{N}^*$ et, pour tout $k \in [\![1\,;n]\!]$, $c_k = a + k\dfrac{b-a}{n}$.

On peut alors approcher $\displaystyle\int_a^b f$ par les sommes de Riemann suivantes.

- *Méthode des rectangles à gauche*

$$S_{g,n}(f) = \frac{b-a}{n}\sum_{k=0}^{n-1} f(c_k).$$

- *Méthode des rectangles à droite*

$$S_{d,n}(f) = \frac{b-a}{n}\sum_{k=1}^{n} f(c_k).$$

- *Méthode des trapèzes*

$$T_n(f) = \frac{1}{2}\left(S_{g,n}(f) + S_{d,n}(f)\right) = \frac{b-a}{n}\left(\frac{f(a)+f(b)}{2} + \sum_{k=1}^{n-1} f(c_k)\right).$$

On sait ici que les mesures sont stockées dans une liste `mesure` ; la longueur N de cette liste correspond au nombre de points de la subdivision, c'est-à-dire, avec les notations ci-dessus, $N = n+1$. L'intervalle d'intégration est ici $[a\,;b]$ avec $a = 0$ et $b = t_{final} = (N-1) * dt = n * dt$. Ainsi, la valeur moyenne (égale à l'intégrale divisée par $t_{final} = b - a$) sera simplement approchée par :

$$\frac{1}{n}\left(\frac{f(a)+f(b)}{2} + \sum_{k=1}^{n-1} f(c_k)\right).$$

D'où le programme :

```
def moyenne(mesure):
    N = len(mesure)
    S = (mesure[0] + mesure[-1])/2
    for i in range (1, N-1):
        S += mesure[i]
    return S / (N-1)
```

Q7.

```
def ecart_type(mesure):
    Imoy = moyenne(mesure)
    mesure2 = [(x  - Imoy)**2 for x in mesure]
    return moyenne(mesure2)**0.5
```

Q8. Il faut ici supposer que les constantes `Imin` et `Imax` ont été préalablement définies.
La première requête est simple :

```
SELECT nSerie from testfin WHERE Imoy>Imin AND Imoy<Imax
```

Q9. Pour obtenir la moyenne d'une liste de valeurs, il faut faire une requête intermédiaire :

```
SELECT AVG(Iec) FROM testfin
```

donc la requête demandée s'écrira :

```
SELECT nSerie, Iec, fichierMes FROM testfin WHERE Iec < (
SELECT AVG(Iec) FROM testfin
)
```

Q10. Dire qu'une imprimante n'a pas été validée signifie qu'elle n'apparaît pas dans la table `production`.
On peut donc écrire, en utilisant une sous-requête :

```
SELECT nSerie, fichierMes FROM testfin
WHERE nSerie NOT IN (SELECT nSerie FROM production)
```

ou bien, en utilisant une jointure :

```
SELECT nSerie, fichierMes from testfin AS t
WHERE NOT EXISTS (SELECT * FROM production AS p WHERE t.nSerie = p.nSerie)
```

On peut aussi utiliser l'ordre EXCEPT (ou MINUS, selon les implémentations de SQL), qui fait une différence ensembliste des tables, à condition qu'elles aient exactement les mêmes colonnes :

```
SELECT nSerie, fichierMes from testfin
EXCEPT (SELECT nSerie, fichierMes FROM testfin AS t, production AS p
                WHERE t.nSerie = p.nSerie)
```

Q11. *La notion d'arbre n'est pas au programme d'IPT, néanmoins je trouve que l'énoncé permettait de comprendre de quoi il s'agit.*

La fonction `test(x)` renvoie `True` si et seulement si `x` est un couple (= tuple de longueur 2), formé d'une chaîne de caractères (type `str`) et d'un entier (type `int`).

Les fonctions `get1(x)` et `get2(x)` renvoient alors les éléments de ce couple `x`, c'est-à-dire (dans le cas où `x` est de la forme précédente) respectivement une chaîne de caractères et un entier.

Q12. *On peut noter un certain nombre d'erreurs dans le programme fourni : il apparaît une mystérieuse fonction `get2q` (on comprend qu'il s'agit de `get2`), et vers la fin du programme l'indentation est incorrecte.*

Notons aussi l'emploi de la structure `while...else` qui n'est pas au programme, et dont l'utilisation est loin d'être indispensable.

Enfin, je trouve que le programme qui est proposé pour construire l'arbre de Huffman est particulièrement laid...

On comprend aisément que la fonction `make_huffman_tree` renvoie une liste de 4 éléments, les deux premiers étant les arguments d'entrée, et les deux suivants correspondant au nœud obtenu à partir de ces deux éléments, le premier étant formé des caractères et le second étant égal au poids total. On a donc :

```
node1 = [ ('F', 1), ('E', 1), ['F', 'E'], 2 ]
node2 = [ ('D', 1), ('B', 1), ['D', 'B'], 2 ]
```

Q13. Même principe, sauf qu'ici les arguments d'entrée ne sont plus des feuilles, mais les nœuds construits auparavant :

```
node3 = [ [ ('F', 1), ('E', 1), ['F', 'E'], 2 ], [ ('D', 1), ('B', 1), ['D', 'B'], 2 ], ['F', 'E', 'D', 'B'], 4]
```

Q14. La fonction `freq_table` renvoie un dictionnaire avec la fréquence de chaque caractère :

```
f= {'A':2, 'B':3, 'C':1}
```

Q15. Que dire ? La fonction est récursive...

Q16. Il s'agit ici de prouver la validité d'un invariant de programme dans le cas d'une procédure récursive ; c'est le même principe qu'un invariant de boucle.

On peut remarquer que cette procédure suit le même principe que celui du tri par insertion *; on aurait donc pu s'en inspirer et éviter la récursivité...*

Pour répondre à la question, on va démonter « par récurrence » sur `pos` la propriété :

$\mathcal{P}$: les éléments de `lst[0:pos]` ont un poids inférieur à celui de la variable `item`.

- *Initialisation* : la propriété est vrai pour `pos=0` puisque la liste est alors vide ;
- *Hérédité* : Supposons la propriété $\mathcal{P}$ vérifiée au début de la procédure. Il y a 3 cas :
  - si `pos=len(lst)`, on a atteint la fin de la liste et tous les éléments précédents ont un poids inférieur ; on ajoute alors l'objet `item` en fin de liste, et évidemment la propriété $\mathcal{P}$ reste vraie ;
  - sinon, on teste si `item` a un poids inférieur à celui de `lst[pos]` ; dans ce cas on l'insère à cet endroit ; puisque tous les éléments entre `0` et `pos-1` ont un poids inférieur d'après $\mathcal{P}$, et que les éléments suivants ont un poids supérieur car la liste est déjà triée, la propriété $\mathcal{P}$ reste vraie ;
  - dans le dernier cas, `item` a un poids strictement supérieur à celui de `lst[pos]` : on incrémente alors `pos` de 1, de sorte que $\mathcal{P}$ reste vraie, et on réappelle la procédure.

Q17. *J'ai déjà signalé l'indentation incorrecte de ce morceau de programme...*

Une réponse à la question pourrait être :

```
## création de l'arbre lorsqu'il y a plus de deux feuilles

## 5- tant qu'il y a plus de deux feuilles ou nœuds

## 6- on construit un sous-arbre avec les deux éléments de plus petits poids

## 7- on retire les deux éléments précédemment utilisés de la liste

## 8- on place le sous-arbre ainsi créé à la bonne place
```

Q18. Dans le meilleur des cas, l'élément à insérer est de poids inférieur à tous les autres ; il sera donc inséré en tête après avoir fait 2 tests.

Dans le pire des cas, il est de poids supérieur à tous les autres ; la procédure `insert_item` sera donc appelée récursivement n fois, avec 2 tests à chaque fois, puis l'élément sera directement inséré en fin de liste, d'où au total $2n+1$ tests.

Q19. La variable passée à la procédure `build_huffman_tree` est `txt`, à partir de laquelle la procédure construit une liste `lst` ; supposons donc que cette dernière liste est de longueur n (*c'est ce que voulait dire l'énoncé je pense...*).

La boucle `while` est toujours effectuée $n-2$ fois (à chaque fois on supprime deux éléments de la liste mais on en rajoute un !). À la k-ième itération, on appelle la procédure `insert_item` avec comme paramètre une liste de longueur $n-1-k$.
Donc dans le meilleur des cas il y aura $2(n-2)$ tests, et dans le pire des cas il y en aura

$$\sum_{k=1}^{n-2}\big(2(n-1-k)+1\big) = (n-2)+2\frac{(n-2)(n-3)}{2} \underset{n\to+\infty}{\sim} n^2\,.$$

On ne tient pas compte ici du tri de la liste, qui est dans le meilleur des cas de complexité $O(n\ln n)$, et qui est donc logiquement négligeable.

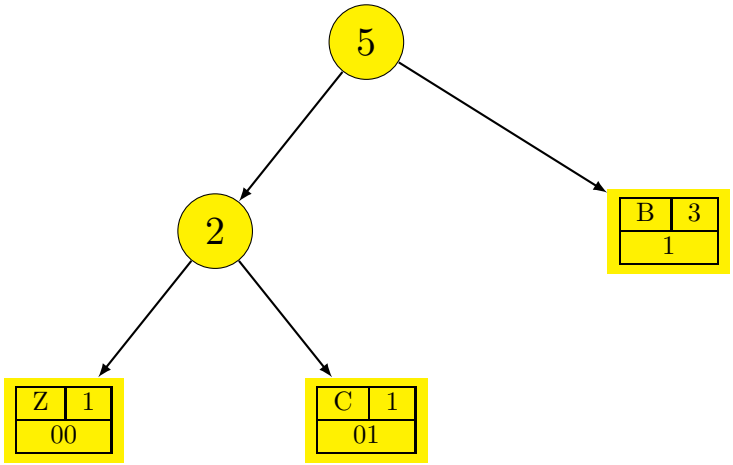
Remarque : je ne vois guère l'intérêt de ce calcul de complexité, car quel que soit le texte `txt`*, la liste* `lst` *ne comportera jamais plus de* 255 *caractères ! La construction de l'arbre a donc un temps fixe, négligeable en général devant la longueur du texte et les autres opérations de compression !*

Q20. Il est impossible de répondre de manière exacte à cette question, car on ne sait pas *a priori* comment sont rangés les caractères dans le dictionnaire donc comment sont ordonnés des caractères de mêmes poids...

Une solution possible est :

```
[ [('Z', 1), ('C',1), ['Z', 'C'], 2], ('B', 3), ['Z', 'C', 'B'], 5]
```

Q21. Là encore deux solutions sont possibles (selon l'ordre des caractères 'C' et 'Z'). Par exemple :



Q22. Pour résoudre de façon approchée une équation différentielle linéaire du 1er ordre, on utilise la méthode d'Euler explicite, qui consiste à construire l'estimation demandée $s(t)$ de proche en proche en des points espacés de $\mathrm{d}t$, en utilisant l'approximation :

$$s(t+\mathrm{d}t) \approx s(t) + s'(t)\,\mathrm{d}t\,,$$

la valeur de $s'(t)$ étant fournie par l'équation différentielle $s' = -k\dfrac{s-e}{10}$.
Cela donne le programme suivant (on suppose que l'on a écrit auparavant : `from math import sin`) :

```python
def solution(k):
    # conditions initiales
    t = 0
    s = 0

    T = [t] # liste des abscisses
    S = [s] # liste des ordonnées
    dt = 0.1 # pas de la simulation
    t_final = 10
    while t <= t_final:
        s += -k*(s-sin(t))/10*dt
        t += dt
        S.append(s)
        T.append(t)
    return S # dans ce cas, nul besoin de la variable T
    # ou bien: plot(T,S)
```

Q23. On calcule la distance entre la solution trouvée précédemment dans les trois cas indiqués par l'énoncé et la sortie obtenue réellement ; si cette distance dans les trois cas dépasse un seuil fixé, l'imprimante sera déclarée défectueuse.

```python
def convient(S, seuil = 1e-3):
    # renvoie True si l'enregistrement S est suffisamment proche d'une des
    # valeurs théoriques obtenues pour k=0.5, 1.1 et 2
    # seuil représente l'erreur maximale admissible

    def distance(l1, l2):
        # distance entre deux listes supposées de mêmes longueurs
        # pour la norme euclidienne usuelle
        return sum([(l1[i] - l2[i])**2 for i in range(len(l1))])**0.5

    S1 = solution(0.5)
    S2 = solution(1.1)
    S3 = solution(2)

    return distance(S, S1)<seuil or  distance(S,S2)<seuil or distance(S,S3)<seuil
```