2TSI. TD PYTHON 20. CORRECTION

MESURE DE HOULE

Q1. La fréquence d'échantillonnage étant de 2 Hz, il y a 2 mesures par seconde. Chacune est stockée sur 8 caractères et chaque caractère nécessite 1 octet de stockage. Ainsi, en 20 mn (donc en $60*20$ secondes), il est nécessaire de stocker :

$$8*2*60*20 = 19\,200$$

octets c'est-à -dire $19,2$ Ko.

Commentaires : La question est élémentaire ... mais s'avère que qu'une petite minorité des candidats fournit une réponse convenable ! N'hésitez pas à vous entrainez à ces petits calculs présents dans de nombreuses épreuves.

Q2. Il est nécessaire de faire un enregistrement toutes les 30 mn donc 2 par heure durant 15 jours. Chaque jour comptant 24 h et un enregistrement nécessitant un stockage de $19,2$ Ko, il est nécessaire de stocker

$$19,2*2*24*15 = 13\,824$$

Ko soit environ 14 Mo donc une carte de 1 Go est amplement suffisante.

Commentaires : Question élémentaire mais une part importante des candidats confondent Méga et Giga.

Q3. Oter un chiffre revient à une réduction de 1 chiffre par rapport aux 8 chiffres initiaux revient à une réduction de $\frac{1}{8}*100 = 12,5$ % de l'espace mémoire nécessaire.

Commentaires : À nouveau une question élémentaire mais qui pose manifestement de grosses difficultés à une fraction importante des candidats.

Q4. On ouvre le fichier souhaité et on en extrait la liste de ses lignes avec la méthode *readlines()*.

```
import os

nom = "donnees.txt"
f = open(nom, 'r')
liste_lignes = f.readlines()
liste_niveaux = []
for i in range(1,len(liste_lignes)):
liste_niveaux.append(liste_lignes[i])
f.close()
```

Si le fichier est très lourd, on peut parcourir une à une ses lignes et les stocker puis en extraire toutes les lignes sauf la première via la méthode *readline()*. Chaque activation de cette méthode permet de changer de ligne d'où les deux appels à *readline()* au début du fichier pour se positionner sur la deuxième ligne du fichier. On poursuit tant que la ligne courante n'est pas vide (on préume que la ligne vide n'apparait changement entre deux enregistrements).

```
nom = "donnees.txt"
f = open(nom, 'r')
ligne = f.readline()
ligne = f.realine()
liste_niveau = []
while ligne != "":
liste_niveaux.append(ligne)
ligne = f.readline()
```

Commentaires : Cette question est totalement hors-programme (le rapport le concède) et, en conséquent, très peu de candidats ont pu donner une réponse correcte, ce qui est tout à fait normal même si le rapport le regrette ! Au final, cette question n'a sélectionné aucun candidat et c'est tant mieux (même s'il eut été préferable que le concours place une question dans le cadre du programme).

Q5. D'après les lectures graphiques, on a :

- $\max\limits_{[0,Z_1]} \eta \simeq 6$ m, $\min\limits_{[Z_1,Z_2]} \eta \simeq -3$ m donc $H_1 \simeq 6-(-3) = 9$ m ;
- $\max\limits_{[Z_1,Z_2]} \eta \simeq 7$ m, $\min\limits_{[Z_2,Z_3]} \eta \simeq -2$ m donc $H_2 \simeq 7-(-2) = 9$ m ;
- $\max\limits_{[Z_2,Z_3]} \eta \simeq 5$ m, $\min\limits_{[Z_3,Z_4]} \eta \simeq -1$ m donc $H_3 \simeq 5-(-1) = 6$ m ;
- $T_1 = Z_2 - Z_1 \simeq 15 - 5 = 10$ s ;
- $T_2 = Z_3 - Z_2 \simeq 28 - 15 - 13$ s.

Commentaires : Question sans difficulté particulière qui fut bien traitée par une large part des candidats.

Q6. Voici une version via les itérations sur les éléments de la liste (car une liste est un itérateur en Python).

```
def moyenne(liste_niveaux):
s = 0
for mesure in liste_niveaux:
s = s + mesure
return s/len(liste_niveaux)
```

Voici une version en s'aidant des indices des éléments de la liste

```
def moyenne(liste_niveaux):
n = len(liste_niveaux)
s = 0
for i in range(n):
s = s + liste_niveaux[i]
return s/n
```

Commentaires : Question sans difficulté particulière qui fut bien traitée par une large part des candidats.

Q7. Rappelons la mèthode des trapèzes. Si l'intervalle $[a, b]$ est découpé en segments $[a_0, a_1]$, $[a_1, a_2]$, .., $[a_{n-1}, a_n]$ de même longueur h avec $a_0 = a$ et $a_n = b$ alors

$$\int_a^b f(t)\,dt \simeq \sum_{i=0}^{n-1} \left(\frac{f(a_i) + f(a_{i+1})}{2} \right) h.$$

Si L est la liste stockant les valeurs successives de f alors, pour tout indice i, $L[i] = f(a_i)$. Comme l'échantillonnage est de 2 Hz, la longueur de chaque intervalle est de $\frac{1}{2}$ seconde d'où le code suivant :

```
def integrale_precise(liste_niveaux):
n = len(liste_niveaux)
dt = 1/2
s = 0
for i in range(n-1):
s = s + (liste_niveaux[i] + liste_niveaux[i+1])/2
return s*dt
```

Rappelons que la valeur moyenne d'une fonction f sur un intervalle $[a, b]$ est le nombre $\frac{1}{b-a}\int_a^b f$. Comme ici la longueur de l'intervalle est de 20 mn, on obtient le code suivant.

```
def moyenne_precise(liste_niveaux):
return(integrale_precise(liste_niveaux)/20)
```

Commentaires : Il s'agit d'une question de cours sans difficulté particulière. Néanmoins cette question s'est avérée discriminante.

Q8. Nous allons parcourir tous les éléments de la liste *liste_niveaux* et trouver le premier indic i tel que *liste_niveaux[i]* est supérieur à la moyenne de la liste et l'indice suivant ne l'est pas. Si aucun tel indice n'existe, la boucle s'achève et on revoit -1.
Voici une version avec la boucle while

```
ef ind_premier_pzd(liste_niveaux):
n=len(liste_niveaux)
m=moyenne_precise(liste_niveaux)
i=0
while i < n-1 and liste_niveaux[i] > m and liste_niveaux[i+1] < m:
i = i +1
if i == n-1:
return -1
else :
return i-1
```

Il faut renvoyer l'indice $i - 1$ car, lorsque le test a échoué, l'indice avait augmenté de 1 précédemment.
Voici la version avec la boucle for et une sortie anticipée

```
def ind_premier_pzd(liste_niveaux):
n=len(liste_niveaux)
m=moyenne_precise(liste_niveaux)
for i in range(n-1):
if liste_niveaux[i] > m and liste_niveaux[i+1] < m:
return i
return -1
```

Commentaires : Pour simplifier le codage, il ne faut pas hésiter à utiliser une boucle *for* et à faire des sorties prématurées via *return*. La gestion des booléens (les tests $L[i] >$ par exemple) s'est avérée discrimante. Attention à l'indentation. Par exemple, placer *return -1* au même niveau que le *if* entraine une sortie prématurée de la fonction, ce qui fut lourdement sanctionné à cette épreuve.
La question est la première qui teste une maîtrise convenable de la syntaxe Python (donc il faut y être rigoureux et soigneux).

Q9. Il suffit d'utiliser un code voisin du code précédent en utilisant une boucle descendante du type :
for i in range$(n-1, 0, -1)$:
puis on teste si *liste_niveaux*$[i] > m$ et *liste_niveaux*$[i-1] > m$. Si c'est le cas, on *return* i sinon on *return* -2.

```
def ind_dernier_pzd(liste_niveaux):
n=len(liste_niveaux)
m=moyenne_precise(liste_niveaux)
for i in range(n-1,0,-1):
    if liste_niveaux[i] < m and liste_niveaux[i-1] > m:
        return i
return -2
```

Q10. Cette fonction est juste une modification de la fonction *ind_premier_pzd* (de la question 8). Au lieu de retourner le premier indice, on ajoute chaque indice convenable à la liste (ou plus précisèment, son successeur strict comme le demande l'énoncé). Voici la fonction correspondante.

```
def construction_successeurs( liste_niveaux ):
n = len( liste_niveaux )
successeurs =[]
m = moyenne_precise(liste_niveaux)
for i in range(n-1):
if liste_niveaux[i] > m and liste_niveaux[i+1] < m:
successeurs.append(i+1)
```

Commentaires : La fonction demandée ne peut être en $O(1)$ dans le meilleur des cas car elle nécessite le calcul de la moyenne de la liste qui est en $O(n)$ si n est la longueur de la liste ! Par contre, si la moyenne est prés-calculée, la fonction proposée ci-dessus a bien une complexité en $O(1)$ si la liste alterne entre 2 valeurs, l'une au dessus de la moyenne l'autre en dessous (par exemple $[-1, 1, -1, 1, -1, 1, ..]$ qui est de moyenne nulle). Le rapport confirme cette erreur d'énoncé.
La question est élémentaire et fut globalement bien réussie.

Q11. On tape : *succ = construction_successeurs(liste_niveaux)*
Puis on initialise *vagues* = [ ] au départ. On travaille dans une boucle :
for i in range(len(succ) − 1) :
Si Z_i et Z_{i+1} sont deux successeurs consécutifs, il crée la vague allant de l'indice Z_i (inclus) à l'indice Z_{i+1} (non inclus). Rappelons que si L est une liste et i, j deux entiers avec $i < j$ alors $L[i : j]$ est la liste formée des éléments de L allant de l'indice i à l'indice j (non inclus). Puis on nourrit *vagues* et quand *vagues* est bien remplie, on la *return*.

```
def decompose_vagues(liste_niveaux):
succ= construction_successeurs(liste_niveaux)
vagues = []
for i in range(len(succ)-1):
    Zi = succ[i] \, ; Zi1 = succ[i+1]
    vagues.append(liste_niveaux[Zi:Zi1])
return vagues
```

Q12. On tape d'abord :
def proprietes(listes_niveaux) :
On utilise la fonction décomposant en vague, en tapant :
vagues = decompose_vagues(liste_niveaux)
On affecte $Z1$ avec *ind_premier_pzd(liste_niveaux)*.
Il faut distinguer le premier intervalle $[0, Z_1]$ des autres. Nous utilisons la commande $L[0 : Z_1 + 1]$ pour obtenir tous les éléments ments de L situés entre l'indice 0 et l'indice Z_1 (inclus). On affecte alors $H1$ puis $T1$. Enfin, dans *prop*, on rentre$[[H1, T1]]$.
On déroule ensuite une boucle *for* pour calculer les différentes hauteurs et temps (qu'il faut multiplier par 0.5 car le pas de temps est d'une demi-seconde, fréquence de 2 Hz). On affecte dans la boucle les H_i et les T_i. Puis, on nourrit *prop* et on le *return*

```
def proprietes(liste_niveaux):
vagues = decompose_vagues(liste_niveaux)
dt = 1/2 \, ; \, Z1 = ind_premier_pzd(liste_niveaux)
H1 = max(liste_niveaux[0:Z1+1]) - min(vagues[0])
T1 = len(vagues[0])\, ; \, prop = [[H1, T1]]
for i in range(1,len(vagues)):
    Hi = max(vagues[i-1]) - min(vagues[i])
 Ti = len(vagues[i]) * dt \, ; \, prop.append([Hi,Ti])
return prop
```

Q13. On crée la liste des propriétés *prop* associées par la commande *proprietes(liste_niveaux)*.
Puis, en nommant *Hmax* la fonction cherchée, on tape :
def Hmax(liste_niveaux) :
Par les listes en compréhension, on crée la liste des hauteurs associé en extrayant de chaque élément p de la liste *prop* sa première composante *p[0]* (qui donne la hauteur de la vague).
Ainsi, la syntaxe *L[i] for L in prop* extrait de chaque élément L de la liste *prop* sa composante *L[i]* Pour finir, on applique la fonction *max* sur cette liste (puisque le sujet l'autorise à la question précédente).

```
def Hmax(liste_niveaux):
        return max([ p[0] for p in proprietes(liste_niveaux)])
```

On peut aussi la recoder comme suit (cela évite la création d'une liste inutile).

```
def Hmax(liste_niveaux):
prop = proprietes(liste_niveaux)
m = -1
for p in prop :
if p[0] > m :
m = p[0]
return m
```

Commentaires : Question de cours qui fut bien traitée par une large part des candidats.

Q14. Dans le tri rapide, on peut choisir tout élément de la liste *liste* comme pivot. Par exemple, celui d'indice g ou bien d'indice d ou tout autre de votre choix (même au hasard entre g et d). Voici un exemple en n'oubliant pas que *liste[i]* est une liste à deux éléments et c'est le premier qui contient la hauteur de la vague.

```
pivot = liste[g][0]
```

Commentaires : Le tri rapide proposé ici est en place, c'est-à-dire qu'il modifie la liste *liste* sans en créer une autre (et la renvoyer) d'où l'absence de *return*.
Question sans difficulté particulière ... pour les candidats connaissant leur cours (notamment celui sur les tris).

Q15. Si l'on appelle *triRapide_bis* la modification et on conserve comme nom *triRapide* à la fonction proposée par l'énoncé,

```
def triRapide_bis (liste,g,d):
if d-g <15:
tri_insertion(liste , g,d)
else:
triRapide(liste,g,d)
```

Commentaires : La question est potentiellement élémentaire mais il s'est avéré que la plupart des candidats n'ont saisi la signification de la phrase du sujet : « lorsque le nombre de données dans une sous-liste devient inférieur ou égal à 15, la fonction triInsertion soit appelé pour terminer le tri. ». Au final, peu de candidats ont réussi à traiter correctement cette question.

Q16. Dans le tri insertion d'une liste L, on se place en un indice $j = i$. On teste si $L[j] < L[j-1]$. Si c'est le cas, on échange $L[j]$ et $L[j-1]$ puis on recommence avec l'indice $j = i - 1$. On poursuit ainsi jusqu'à l'indice minimal (ici g). On commence par $i = g$ puis on arrête lorsque $i = d$ d'où le code suivant (en tenant compte que les éléments de la liste sont ici des listes que l'on tri selon la première composante, correspondant à la hauteur de vague).

```
def triInsertion (liste ,g ,d):
for i in range(g+1,d+1):
j = i-1
tmp = liste[i]
while g <= j and tmp[0] < liste[j][0]:
liste[j+1] = liste[j]
j = j - 1
liste[j+1]=tmp
```

Commentaires : Le tri insertion proposé ici est en place, c'est-à-dire qu'il modifie la liste *liste* sans en créer une autre (et la renvoyer) d'où l'absence de *return*.
Question sans difficulté particulière ... pour les candidats connaissant leur cours (notamment celui sur les tris).

Q17. Si m désigne la moyenne de la haute des vagues et n la liste de la liste *liste_hauteurs*, la ligne 6 montre que m est recalculé à chaque itération de la boucle *for* donc elle est recalculer n fois. Comme la fonction *moyenne* a une complexité temporelle en $O(n)$, cela fait une complexité totale en $nO(n) = O\left(n^2\right)$ fois. On contourne le problème en stockant initialement m. La complexité est alors de $O(n)$ (côut du calcul de m) $+O(n)$ (les n itérations de la boucle *for*) soit une complexité temporelle en $2O(n) = O(n)$. Voici le code correspondant.

```
def skewness ( liste_hauteurs ):
n=len( liste_hauteurs )
et3=(ecartType ( liste_hauteurs ))**3
S=0
m = moyenne( liste_hauteurs )
for i in range(n):
S+=(liste_hauteurs [ i] -m )**3
S=n/(n-1)/(n-2)*S/et3
return S
```

Q18. Les complexités sont identiques (remplacer les puissances troisième par des puissance quatrième, modifier quelques divisions et multiplications ainsi qu'une soustraction) .. asymptotiquement.

Commentaires : La question est bien réussie par une large partie des candidats.

Q19. **Première requête**. Il suffit de sélectionner au sein de la table *Bouee* les enregistrements ayant la bonne localisation et de projeter les résultats selon les champs d'identification et de nom de site.

```
SELECT idBouee, nomSite F
ROM Bouee
WHERE localisation = "Mediterranee"
```

Deuxième requête.Il suffit de sélectionner tous les identifiants de bouées dans la table *Bouee* en éliminant celles apparaissant dans la table *Tempete*. Miracle, la commande *EXCEPT* le permet !

```
SELECT idBouee FROM Bouee
EXCEPT
SELECT idBouee FROM Tempete
```

Troisième requête. Il faut croiser les tables *Bouee* et *Tempete* selon l'identifiant de la bouée. On regroupe alors les bouées selon le site où elles se situent. Ensuite, au sein de chacun de ces regroupements, on extrait la plus grande des hauteurs maximales. Voici le code correspondant.

```
SELECT nomSite, max(Hmax)
FROM Bouee JOIN Tempete
ON Bouee.idBouee=Tempete.idBouee
GROUP BY nomSite
```

Q20. Le tri fusion consiste à diviser un tableau de taille N en deux sous-tableaux de taille $\frac{N}{2}$, de les trier par tri fusion, puis de recombiner les deux sous-tableaux obtenus par interclassement (ce qui a un coût en $O(N)$). Or, le cours affirme que ce tri a une complexité en $O(N\log_2(N))$ donc la TFD rapide a aussi une complexité en $O(N\log_2(N))$. En effet, il suffit de remplacer les mots *tableau* par *liste*, *tri fusion* par *TFD rapide* et *interclassement* par *recombinaison* (les opérations ($P_k + w^k I_k$ et $P_k - w^k I_k$, $0 \leqslant k \leqslant \frac{N}{2}$) alors les processus sont identiques entre le tri fusion et la TFD rapide ainsi que la complexité de l'interclassement et de la recombinaison de chaque étape.

Commentaires : Question difficile.

Q21. Comme tout programme récursif, on pense à écrire en priorité les tests d'arrêts. Ici, il s'agit de savoir si le tableau est de taille au plus 1 ou pas. Ensuite, il suffit de créer les sous-tableaux associés aux indices pairs et impairs, effectuer leurs TFD respectives puis recombiner ces deux tableaux.

```
def TFD(x):
N = len(x)
if N <= 1:
return x
n = N//2
x_pair = [x[2*i] for i in range(n)]
x_impair = [x[2*i+1] for i in range(n)]
P = TFD(x_pair)
I = TFD(x_impair)
w = np.exp(2*np.pi*1j/N)
X = [ 0 for k in range(N) ]
for k in range(n):
X[k] = P[k] + I[k]*w**k
X[k+n] = P[k] - I[k]*w**k
return(X)
```

Commentaires : Au final, cette question n'a eu de reponse satisfaisante que dans les meilleures copies.