CCMP IPT 2018 - Corrigé - Q1 à Q14.

P. Haslé

Remarques générales:

- Tous les résultats doivent être **justifiés** !
- 1 octect = 8 bits
- Ne pas inventer de syntaxe Python, cela donne très mauvaise impression.
- Pour les opérations sur les listes, il est important de ne pas confondre, indices et valeurs.

- `import ...` sert pour utiliser un module pas lire un fichier `import donnees.txt` n'a donc aucun sens.

Q1 En te tenant, pas compte de la prmière ligne, avec un échantillonage de 2hz, il y a 2400 enregistrements en 20 minutes ($2 \times 60 \times 20$). Avec 8 octets par enregistrements, on a donc une taille des données de 20 minutes qui est de 19,2Ko.

Q2 Pour une campage de 15 jours, avec 48 plages de 20 minutes par jour, nous avons $(15 \times 48 \times 19,2ko)$. Soit un peu moins de 14000Ko donc envion 14Mo. Une carte de capacité de 1Go est amplement suffisante.

Q3 Chaque ligne nécessite 8 octets avec la précision actuelle, en retirant un chiffre par ligne, on retire 1 octet. L'économie de mémoire sera donc de 1/8, un peu moins de 2Mo sur les 15 jours.

Q4

```
with open("donnees.txt") as f:
    lignes = f.readlines() #  lecture des lignes, placées dans la liste lignes

    liste_niveaux = [ float(ligne) for ligne u=i lignes[1:]]
    # la tranche [1:] écarte la première ligne,
    # float(ligne) permet de convertir la chaine de caractères en flottants
```

Q5 Par lecture graphique on considère :

- $H_1 \approx 9$
- $H_2 \approx 8,9$
- $H_3 \approx 6$
- $T_1 \approx 12$
- $T_2 \approx 13$

Q6

```
def moyenne(liste_niveaux: list) -> float:
    sum_niveaux = 0
    for niveau in liste_niveaux:
        sum_niveaux += niveau
    return sum_niveaux / len(liste_niveaux)
```

remarque: Il est possible d'utiliser la fonction `sum` de Python sauf quand cela est clairement indiqué. Dans ce cas il n'y a pas de boucle et la fonction peut devenir :

```
def moyenne(liste_niveaux: list) -> float:
    return sum(liste_niveaux) / len(liste_niveaux)
```

Q7

On utilise la methode des trapèzes avec $\Delta_t = 0.5$.

```
def integrale_precise(liste_niveaux: list) -> float:
    delta_t = 0.5
    somme_trapezes = 0
    for i in range(len(liste_niveaux)i -1):
        trapeze = (liste_niveaux[i] + liste_niveaux[i + 1]) / 2 * delta_t
```

```
        somme_trapezes += trapeze
    return somme_trapezes
```

Et pour la moyenne précise:

```
def moyenne_precise(liste_niveaux: list) -> float:
    duree = (len(liste_niveaux) - 1) / 2 # le nombre d'intervalles est égal au nombre de valeurs, moins 1
    return integrale_precise(liste_niveaux) / duree
```

Q8

```
def ind_premier_pzd(liste_niveaux: list) -> int:
    moy = moyenne(liste_niveaux)
    for i in range(len(liste_niveaux) - 1):
        if liste_niveaux[i] > moy and liste_niveaux[i + 1] < moy:
            return i
    return -1
```

remarque on a ici un parcourt séquentiel de la liste en compléxité $\mathcal{O}(n)$.Un code plus efficace fonctionnerait par recherche dichotomique.

Q9 Pour obtenir la compléxité en $\mathcal{O}(1)$ dans le meilleur des cas on parcourt la liste en commençant par la fin. Il faut aussi bien sûr que la moyenne soit déjà calculée, sinon la complexité sera en $\mathcal{O}(n)$ même dans le meilleur des cas.

```
def ind_dernier_pzd(liste_niveaux: list, moy: float)-> int:
    inf = False # indique quand les points sont en dessous de la moyenne
    for i in range(len(liste_niveaux) - 1, 0, -1):
        if liste_niveaux[i] < moyenne and liste_niveaux[i - 1] > moyenne:
            return i - 1

    # non trouvé, malgré la recherche itérative
    return -2
```

Q10

```
def construction_successeurs(liste_niveaux: list)->list:
    n = len(liste_niveaux)
    successeurs =[]
    m = moyenne(liste_niveaux)
    for i in range (n - 1):
        if liste_niveaux[i] > m and liste_niveaux[i + 1] < m:
            successeurs.append(i + 1) # ajout du successeur immédiat
    return successeurs
```

Q11

```
def decompose_vagues(liste_niveaux: list)-> [list]:
    successeurs = construction_successeurs(liste_niveaux)
    decomposition = []
    for i in range(len(successeurs) - 1):
        suc1 = successeurs[i]
        suc2 = successeurs[i + 1]

        decomposition.append(liste_niveaux[suc1:suc2])

    return decomposition
```

Q12

```
def proprietes(liste_niveaux: list) -> [[float,float]]:
    decomposition = decompose_vagues(liste_niveaux)
    p = [] # pour stocker les listes
    for vague in decomposition:
        mini = min(vague)
        maxi = max(vague)
        hi = maxi - mini
        t1 = (len(vague) - 1)* 0.5 # -1 toujours le décalage longueur, nombres valeurs et 0.5 car 2Hz

        p.append([hi,ti])
```

Q13

```
def h_max(liste_niveaux: list) -> float:
    return max([h for h,t in proprietes(liste_niveaux)])
```

Q14

Pour trier toute la liste, les arguments `g` et `d` doivent prendre les valeurs respectivement `0` et `len(l) - 1` (indices des premier et dernier élément de la liste).

Voici le code complété :

```
def triRapide(liste, g, d):
    pivot = liste[(g + d) // 2][0]
    i = g
    j = d

    while i <= j:
        while liste[i][0] < pivot:
            i += 1
        while liste[j][0] > pivot:
            j -= 1

        if i <= j:
            liste[i], liste[j] = liste[j], liste[i]
            i += 1
            j -= 1

    if g < j:
        triRapide(liste, g, j)
    if i < d:
        triRapide(liste, i, d)
```

Remarque

Si on initialise `pivot` à g, come cela se fait souvent pour le tri rapide, la condition `liste[i][0] < pivot` du deuxième `while` ne sera jamais vraie. En effet comme i est inialisé à `g` il y a égalité entre les deux termes comparés et donc pas d'infériorité stricte. La boucle du troisième while va décrémenter `j` jusuq'à l'indice le plus à droite du terme dont liste[j][0] est inférieur au pivot. Si j reste supérieur à i, alors ce terme sera échangé avec le pivot, et on appellera donc `triRapide(liste, i, d)` ce qui va permettre de trier la liste.

Et les deux dernières lignes

```
    if i < d:
        triRapide(liste, i, d)
```

ne seront donc jamais exécutées, c'est ce que l'on appelle du **code mort**.

La liste sera pourtant bien triée mais ce n'est pas un **tri rapide** où on utilise **diviser pour régner** en triant les deux parties de la partition autour du pivot.