Partie I. Stockage interne des données

Q1 Chaque ligne est constituée de 8 caractères donc occupe $8 octets$. 20 minutes correspondent à $1200s$. Comme il y a deux mesures par seconde, cela correspond à 2400 mesures au total, soit une taille de $8 \times 2400 = 19200 octets = 19,2 ko$.
Q2 La campagne de mesure fait état d'un fichier récolté toutes les demi-heures pendant 15 jours. Il y a donc $15 \times 24 \times 2 = 7,2.10^2$ fois les informations précédentes collectées, ce qui représente donc une taille totale de $13824000 octets$, soit $13,8 Mo$.
Une carte-mémoire de $1 Go$ est largement suffisante.
Q3 Ôter un chiffre revient à passer chaque ligne de 8 à seulement 7 octets, d'où une réduction de $1/8 = 12\%$ de la taille du fichier total.
Q4

```
f = open('donnees.txt')   # Ouverture du descripteur de fichier
L = f.readlines()         # Lecture effective du fichier
liste_niveaux = []        # Définition du conteneur
for i in range(1,len(L)): # On saute la première ligne de texte
    liste_niveaux.append(float(L[i])) # et on convertit le reste en flottants
f.close()                 # Fermeture du descripteur de fichier
```

Partie II. Analyse vague par vague

Q5 On utilise à chaque fois le maximum précédant Z_i et le minimum qui le suit. On a alors $H_1 \approx 6 - (-3) = 9\text{m}$, puis $H_2 \approx 7 - (-2) = 9\text{m}$ et $H_3 \approx 5 - (-1) = 6\text{m}$. Pour les mesures de périodes, on obtient
$T_1 \approx 15,5 - 3,5 = 12\text{s}$ et $T_2 \approx 28,5 - 15,5 = 13\text{s}$
Q6

```
def moyenne(liste_niveaux):
    m = 0
    for valeur in liste_niveaux:
        m = m + valeur
    return m/len(liste_niveaux)
```

Q7

```
def integrale_precise(liste_niveaux):
    dt = 0.5      # Pas d'intégration
    integrale = 0 # Initialisation de l'intégrale
    for i in range(len(liste_niveaux)-1):
        trapeze = (liste_niveaux[i] + liste_niveaux[i+1])/2 * dt
        integrale = integrale + trapeze
    return integrale
```

Alors, Moyenne $\eta(t) = \frac{1}{t_{tot}} \int_0^{t_{tot}} \eta(t)dt$

```
def moyenne_precise(liste_niveaux):
    dt = 0.5
    t_tot = len(liste_niveaux) * dt
    integrale = integrale_precise(liste_niveaux)
    return integrale / t_tot
```

Q8 Il s'agit de calculer la moyenne et passer en revue tous les points pour savoir s'il y en a un qui dépasse la moyenne en sens descendant. On sort alors directement de la fonction par le return.

```
def ind_premier_pzd(liste_niveaux):
    moyenne = moyenne_precise(liste_niveaux)
    for i in range(len(liste_niveaux)-1):
        if liste_niveaux[i] > moyenne and liste_niveaux[i+1] < moyenne:
            return i
    return -1 # Cas où on n'a pas réussi à vérifier la condition dans la boucle
```

Q9 Même chose avec le dernier, il suffirait de garder une mémoire et la renvoyer à la fin.

```
def ind_dernier_pzd(liste_niveaux):
    moyenne = moyenne_precise(liste_niveaux)
    position = -2 # Valeur par défaut si jamais on ne trouve rien
    for i in range(len(liste_niveaux)-1):
        if liste_niveaux[i] > moyenne and liste_niveaux[i+1] < moyenne:
            position = i
    return position
```

Néanmoins, cette implémentation a deux soucis : on doit nécessairement parcourir toute la liste (ce qui donne une complexité en $O(N)$ et non $O(1)$) *et* le calcul de la moyenne est lui-même déjà en $O(N)$. Pour avoir une méthode en $O(1)$ dans le meilleur des cas, on peut utiliser la même technique qu'à la question précédente mais en partant (de manière symétrique) de la fin. Le meilleur des cas est alors que le dernier indice cherché soit l'avant-dernier. Néanmoins, comme dit plus haut, il faut aussi s'assurer que le calcul de la moyenne a déjà été fait auparavant et passé en paramètre à la fonction sinon il imposera sa complexité linéaire en N.

```
# Ne pas oublier de donner la moyenne en argument
def ind_dernier_pzd(liste_niveaux, moyenne):
    # On parcourt la boucle avec un pas négatif de -1 pour remonter la liste
    for i in range(len(liste_niveaux)-2,-1,-1):
        if liste_niveaux[i] > moyenne and liste_niveaux[i+1] < moyenne:
```

```
            return i # Sortie rapide de la fonction dans le meilleur des cas
    return -2 # Si où on n'a pas réussi à vérifier la condition dans la boucle
```

Q10 Il s'agit à présent de récupérer tous les indices et les stocker dans une liste. La fonction complétée s'écrit

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

Il faut bien prendre le point en position $i+1$ qui succède juste à la traversée de la moyenne.
Q11 Pour construire la liste des vagues, on va itérer sur les successeurs et découper notre liste initiale en conséquence.

```
def decompose_vagues(liste_niveaux):
    successeurs = construction_successeurs(liste_niveaux)
    vagues = []
    for i in range(len(successeurs)-1):
        Zi   = successeurs[i]                # Position de Zi
        Zip1 = successeurs[i+1]              # Position de Zi+1
        vagues.append(liste_niveaux[Zi:Zip1]) # La vague associée
    return vagues
```

Q12 Maintenant que l'on dispose d'une liste de vagues, il suffit d'appliquer *max* et *min* sur cette liste pour obtenir les crêtes et compter le nombre d'éléments pour estimer la période. Attention tout de même au fait que l'on prend le *max avant* Z_i et le *min après* Z_i et que dans la fonction $decompose_vagues$, on n'ait pas pensé à garder la première demi-vague pourtant nécessaire pour estimer H_1 et T_1...

```
def proprietes(liste_niveaux):
    vagues = decompose_vagues(liste_niveaux)
    dt = 0.5 # Intervalle de temps entre deux points
    P  = []  # La future liste des propriétés que l'on veut obtenir
    # On traite à part le premier cas
    i0 = ind_premier_pzd(liste_niveaux)
    H1 = max(liste_niveaux[:i0]) - min(vagues[0])
    T1 = len(vagues[0])
    P.append([H1,T1])
    # Après, on peut prendre les vagues les unes après les autres.
    for i in range(len(vagues)-1):
```

```
        Hi = max(vagues[i]) - min(vagues[i+1])
        Ti = len(vagues[i+1]) * dt
        P.append([Hi,Ti])
    return P
```

Partie III. Contrôle des données

Q13 Pour obtenir H_{max}, il suffit de déterminer le maximum des H calculés par la fonction proprietes précédente.

```
def H_max(liste_niveaux):
    prop = proprietes(liste_niveaux) # Détermination des propriétés des vagues
    Hmax = 0          # Initialisation de la mémoire.
    for Pi in prop:   # On itère sur toutes les propriétés.
        Hi,Ti = Pi    # On récupère le couple de propriété pour la vague i.
        if Hi > Hmax: # Si on trouve mieux que la mémoire,
            Hmax = Hi # on le mémorise.
    return Hmax       # Renvoi du maximum observé
```

Q14 Au premier appel de la fonction de tri, on doit regarder la liste en totalité. Par conséquent, l'indice g (pour *gauche*) doit valoir initialement 0 alors que l'indice d (pour *droit*) doit valoir $len(liste) - 1$. Pour le choix du pivot, il faut juste se souvenir qu'on cherche à appliquer cette fonction sur une liste de listes de propriétés (dont seule la première nous intéresse). Comme dans la suite du code (ligne 6 par exemple), on voit que pivot correspond à la valeur stockée dans la liste de liste, on peut choisir $pivot = liste[g][0]$ par exemple comme pivot.
Q15 Il suffit de rajouter en toute première ligne de la fonction triRapide la vérification sur la taille $d - g$ restant à vérifier

```
def triRapide(liste,g,d):
    if d - g < 15:
        triInsertion(liste,g,d)
    else:
        # On ne réécrit pas le reste de la fonction qui est inchangé.
        # (à l'indentation près)
```

Remarquons que l'énoncé ne parle d'appliquer le triInsertion que sur des sous-listes de la liste initiale (ce qui fait que le test devrait plutôt avoir lieu aux environs des lignes 17 et 19) mais outre le fait que cela duplique inutilement du code, si on considère que le triInsertion est plus efficace que triRapide sur des petites listes, autant l'appliquer tout de suite sur la liste initiale si celle-ci est déjà suffisamment petite.
Q16 Pour procéder à un tri par insertion, on démarre à l'indice de gauche et on évolue sur la droite en prenant chaque nouvel élément et en décalant au fur et à mesure les élément dans la liste jusqu'à atteindre sa place dans la partie triée (un peu comme vous écartez vos cartes pour laisser passer la carte que vous voulez rajouter dans votre jeu en partant du côté droit)

```
def triInsertion(liste,g,d):
```

```
    for i in range(g+1,d+1):
        j = i-1        # On regarde le plus à droite du jeu de carte déjà trié
        tmp = liste[i] # On sort la carte de son jeu
        # Tant que la carte à placer n'est pas assez grande par rapport à la
        # carte courante et qu'on n'a pas parcouru toute la main,
        while j >= g and liste[j][0] > tmp[0]:
            liste[j+1] = liste[j] # on décale la carte courante vers la droite
            j = j - 1             # et on passe à la suivante.
        # Si on est sorti, c'est que sa place est à droite de la carte courante
        liste[j+1] = tmp
    return liste
```

À noter qu'on suppose toujours trier ici des listes de listes par rapport au premier élément, ce qui demande de comparer liste[j][0] à tmp[0] et pas seulement liste[j] à tmp.

Q17 Dans la fonction skewness proposée, on peut remarquer à la ligne 6 que la moyenne est recalculée pour chacune des hauteurs de la liste alors que sa valeur est fixe et pourrait n'être calculée qu'une unique fois. Comme un calcul de moyenne est forcément linéaire en la taille de la liste initiale, le faire pour chacune des n valeurs de hauteurs produit une fonction globalement quadratique en n, ce qui n'est guère optimal ici. Il suffit de rajouter le calcul $Hmoyen = moyenne(liste_h auteurs)$ entre les lignes 4 et 5 (juste avant la boucle for) et utiliser cette valeur calculée dans la ligne 6 pour s'affranchir de ce souci.

Q18 Les calculs de S et K font chacun intervenir une unique boucle sur les valeur de la liste (pour calculer la somme). Il ne devrait donc pas y avoir de différence de type de la complexité si les deux fonctions procédant à cette évaluation ont été programmées de la même façon [1].

Partie IV. Base de données relationnelle

Q19 La première requête est une simple sélection sur la localisation associée à une projection sur l'identifiant et le nom du site.

```
SELECT idBouee,nomSite FROM Bouee WHERE localisation = 'Mediterranee'
```

Pour la deuxième, il s'agit de vérifier que l'identifiant de la bouée n'est pas dans la table Tempete (soit que son nombre d'occurences dans la table est nul), ce qui demande un aliasing pour ne pas confondre le idBouee voulu.

```
SELECT idBouee AS id FROM Bouee
       WHERE (SELECT COUNT(*) FROM Tempete WHERE idBouee=id) = 0
```

Une autre possibilité revient à utiliser la notion d'appartenance (ou plutôt de non appartenance)

```
SELECT idBouee AS id FROM Bouee
       WHERE idBouee NOT IN (SELECT idBouee FROM Tempete)
```

On peut aussi faire une différence de deux ensembles (avec l'instruction MINUS ou EXCEPT selon le GDBD utilisé)

1. C'est-à-dire qu'il ne s'agit pas de calculer S en appliquant la correction de la questions Q17, mais ne pas faire de même pour K.

```
SELECT idBouee AS id FROM Bouee
EXCEPT
SELECT idBouee FROM Tempete
```

La dernière requête va demander à faire à la fois une jointure (à l'aide de JOIN) et un regroupement par régions (à l'aide d'un GROUP BY).

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

Partie V. Analyse spectrale

Q20 S'il suffit, pour calculer une TFD sur N éléments de calculer deux TFD sur $N/2$ éléments et faire les N calculs nécessaires à repasser des P_k et I_k aux X_k, alors si on note C_N le coût d'une TFD sur N éléments, on devrait avoir
$C_N = 2 \times C_{N/2} + \alpha\, N = 2\left(2\, C_{N/4} + \alpha\, N2\right) + \alpha\, N = 4\, C_{N/4} + 2\alpha\, N = 8\, C_{N/8} + 3\alpha\, N = \cdots$
Finalement, on obtient $C_N = NC_1 + \alpha\, N \times \log_2(N)$ puisqu'il y a $\log_2(N)$ divisions successives par 2, ce qui donne une complexité globale en $O(N \log_2 N)$.
Q21 Pour écrire cet algorithme sous forme récursive, il faut remarquer que P_k et I_k correspondent respectivement aux transformées de Fourier des éléments d'indices respectifs pairs et impairs de la liste initiale. On peut donc demander le calcul desdites TFD pour calculer chacun des X_k.

```
def TFD(x):
    N = len(x)      # Raccourci
    if N == 1:      # On traite le cas particulier d'un élément unique
        return x    # Dans ce cas, la TFD ne change rien
    # Préparation des coefficients pairs et impairs
    liste_pairs   = [x[i] for i in range(0,N,2)]
    liste_impairs = [x[i] for i in range(1,N,2)]
    # Transformées de Fourier pour les coefficients pairs et impairs
    P = TFD(liste_pairs)
    I = TFD(liste_impairs)
    X = [0]*N  # Préparation du réceptacle
    w = np.exp(-2*np.pi*1j/N)
    for k in range(N//2):
        X[k]      = P[k] + w**k * I[k]
        X[k+N//2] = P[k] - w**k * I[k]
    return X
```