Devoir surveillé d'informatique

Remarques et consignes importantes

- On pourra toujours librement utiliser une fonction demandée à une question précédente même si cette question n'a pas été traitée.
- Veillez à présenter vos idées et vos réponses partielles même si vous ne trouvez pas la solution complète à une question.
- La clarté et la lisibilité de la rédaction et des programmes sont des éléments de notation.

❑ Exercice 1 : *Randonnée*

🎓 d'après CCMP 2021 - PC, PC PSI (Partie 1)

Lors de la préparation d'une randonnée, une accompagnatrice doit prendre en compte les exigences des participants. Elle dispose d'informations ressemblées dans deux tables d'une base de données :

— La table `Rando` décrit les randonnées possibles : la clef primaire **`rid`**, son nom, le niveau de difficulté du parcours (entier entre 1 et 5), le dénivelé en mètres, la durée moyenne en minutes :

rid	rnom	diff	deniv	duree
1	La belle des champs	1	20	30
2	Lac de Castellagne	4	650	150
3	Le tour du mont	2	200	120
4	Les crêtes de la mort	5	1200	360
...	...	...	...	...

— La table `Participant` décrit les randonneurs : la clef promaire **`pid`**, le nom du randonneur, son année de naissance, le niveau de difficulté maximum de ses randonnées.

pid	pnom	ne	diff_max
1	Calvin	2014	2
2	Hobbes	2015	2
3	Susie	2014	2
4	Rosalyn	2001	4
...	...	...	...

Donner une requête SQL sur cette base pour :

Q1 – Compter le nombre de participants nés entre 1999 et 2003 inclus.

```
SELECT COUNT(*)
FROM Participant
WHERE ne>=199 and ne<=2003;
```

Q2 – Calculer la durée moyenne des randonnées pour chaque niveau de difficulté.

```
SELECT diff, AVG(duree)
FROM Rando
GROUP BY diff
```

Q3 – Extraire le nom des participants pour lesquels la randonnée n°42 est trop difficile

```
SELECT pnom
FROM Participant
WHERE diff_max < (SELECT diff from Rando WHERE rid=42);
```

Q4 – Extraire les clés primaires des randonnées qui ont un ou des homonymes (nom identique et clé primaire distincte), sans redondance.
Solution utilisant un sc grou by et un HAVING

```sql
SELECT DISTINCT rid FROM Rando
WHERE rnom in (SELECT rnom FROM Rando GROUP BY rnom HAVING COUNT(*)>1)
```

Solution avec une auto jointure

```sql
SELECT DISTINCT R1.rid
FROM Rando AS R1
JOIN Rando AS R2 ON R1.nom = R2.nom
WHERE R1.rid <> R2.rid
```

L'accompagnatrice a activé le suivi d'une randonnée par géolocalisation satellitaire et souhaite obtenir quelques propriétés de cette randonnée une fois celle-ci effectuée. Elle a exporté les données au format texte CSV (*comma-separated-values* – valeurs séparées par des virgules) dans un fichier nommé `suivi_rando.csv` : la première ligne annonce le format, les suivantes donnent les positions dans l'ordre chronologique.
Voici le début de ce fichier pour une randonnée partant de Valmorel, en Savoie, un bel après-midi d'été :

```
lat(°),long(°),height(m),time(s)
45.461516,6.44461,1315.221,1597496966
45.461448,6.444426,1315.702,1597496970
45.461383,6.444239,1316.182,1597496973
45.461641,6.444035,1316.663,1597496979
45.461534,6.443879,1317.144,1597496984
45.461595,6.4437,1317.634,1597496989
45.461562,6.443521,1318.105,1597496994
...
```

Le module `math` de Python fournit les fonctions `asin`, `sin`, `cos`, `sqrt` et `radians`. Cette dernière convertit des degrés en radians, unité des fonctions trigonométriques. La documentation donne aussi des éléments de manipulation de fichiers textuels :

- `fichier = open(NOM_FICHIER, MODE)` ouvre le fichier, en lecture si `MODE` est `"r"`, en écriture si `"w"`.
- `ligne = fichier.readline()` récupère la ligne suivante de `fichier` ouvert en lecture avec `open`.
- `lignes = fichier.readlines()` donne la liste des lignes suivantes.
- `fichier.close()` ferme `fichier`, ouvert avec `open`, après son utilisation.
- `ligne.split(SEP)` découpe la chaîne de caractères `ligne` selon le séparateur `SEP` : si ligne vaut `"42,43,44"` alors `ligne.split(",")` renvoie la liste `["42", "43", "44"]`.

On souhaite exploiter le fichier de suivi d'une randonnée – supposé préalablement placé dans le répertoire de travail – pour obtenir une liste `coords` des listes de 4 flottants (latitude, longitude, altitude, temps) représentant les points de passage collectés lors de la randonnée.
À partir du canevas fourni en annexe, et en ajoutant les `import` nécessaire :

Q5 – Implémenter la fonction `importe_rando` qui réalise cette importation en renvoyant la liste souhaitée, par exemple en utilisant certaines des fonctions ci-dessus, ou une autre approche de votre choix.

```python
def import_rando(nom_fichier):
    reader = open(nom_fichier, "r")
    premiere_ligne = reader.readline()
    lignes = reader.read().strip().split('\n')
    reader.close()
    coords = []
    for l in lignes:
        lat, long, h, t = l.split(',')
        coords.append([float(lat), float(long), float(h), float(t)])
    return coords
```

On peut proposer une version plus condensée mais qui produit le même résultat :

```python
def import_rando(nom_fichier):
    reader = open(nom_fichier, "r")
    lignes = reader.read().strip().split('\n')[1:]
    reader.close()
    coords = [list(map(float, l.split(','))) for l in lignes]
    return coords
```

On suppose maintenant l'importation effectuée dans `coords`, avec au moins deux points d'instants distincts.

Q6 – Implémenter la fonction `plus_haut` qui renvoie le liste (latitude, longitude) du point le plus haut de la randonnée.

```python
def plus_haut(coords):
    premier_point = coords[0]
    phaut = premier_point[0], premier_point[1]
    max_haut = premier_point[2]
    for i in range(1, len(coords)):
        if coords[i][2] > max_haut:
            max_haut = coords[i][2]
            phaut = coords[i][0], coords[i][1]
    return phaut
```

Q7 – Implémenter la fonction `deniveles` qui calcule les dénivelés cumulés positif et négatif en mètres de la randonnée, sous forme d'une liste de deux flottants. Le dénivelé positif est la somme des variations d'altitudes positives sur le chemin et inversement pour le dénivelé négatif.

```python
def deniv(coords):
    dpositif, dnegatif = 0, 0
    for i in range(1, len(coords)):
        alt = coords[i][2]
        prec = coords[i-1][2]
        if alt > prec:
            dpositif += (alt-prec)
        else:
            dnegatif += (alt-prec)
    return dpositif, dnegatif
```

On souhaite évaluer de manière approchée la distance parcourue lors d'une randonnée. On suppose la Terre parfaitement sphérique de raron $R_T = 6371$ km au niveau de la mer. On utilise la formule de haversine pour calculer la distance d du grand cercle sur une sphère de rayon r entre deux points de coordonnées (latitude, longitude) (φ_1, λ_1) et (φ_2, λ_2)

$$d = 2\,r \arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos(\varphi_1)\cos(\varphi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right)$$

On prendra en compte l'altitude moyenne de l'arc, que l'on complètera pour la variation d'altitude par la formule de Pythagore, en assimilant la portion de ce cercle à un segment de droite perpendiculaire à la verticale.

Q8 – Implémenter la fonction `distance` qui calcule avec cette approche la distance en mètres entre deux points de passage. On décomposera obligatoirement les formules pour en améliorer la lisibilité.

```python
from math import asin, sin, cos, sqrt, radians
RT = 6371
def distance(p1,p2):
    phi1, lambda1, alt1 = radians(p1[0]), radians(p1[1]), p1[2]
    phi2, lambda2, alt2 = radians(p2[0]), radians(p2[1]), p2[2]
    alt_moyenne = (alt1+alt2)/2 + RT*1000
    m1 = sin((phi2-phi1)/2)**2
    m2 = sin((lambda2-lambda1)/2)**2
    darc = 2*alt_moyenne*asin(sqrt(m1+cos(phi1)*cos(phi2)*m2))
    return sqrt(darc**2+(alt2-alt1)**2)
```

Q9 – Implémenter la fonction `distance_totale` qui calcule la distance en mètres parcourue au cours d'une randonnée.

```python
def distance_totale(coords):
    dt = 0
    for i in range(1,len(coords)):
        dt += distance(coords[i-1],coords[i])
    return dt
```

Annexe pour l'exercice 1 : canevas de codes Python

```python
# import Python à compléter...

# importation du fichier d'une randonnée
def importe_rando(nom_fichier):
    # À compléter...

coords = importe_rando("suivi_rando.csv")

# donne le point (latitude, longitude) le plus haut de la randonnée
def plus_haut(coords):
    # À compléter...

print("point le plus haut", plus_haut(coords))
# exemple : point le plus haut [45.461451, 6.443064]

# calcul des dénivelés positif et négatif de la randonnée
def deniveles(coords):
    # À compléter...

print("denivelés", deniveles(coords), "m")
# exemple : denivelés [4.059999999999945, -1.1759999999999309] m

RT = 6371 # rayon moyen volumétrique de la Terre en km

# distance entre deux points
def distance(coord1, coord2):
    # À compléter...
```

❑ **Exercice 2** : *Programmes divers et saut de taille maximale* 🎓 d'après CAPES 2023 (Partie 1)

Notes de programmation : Vous disposez pour répondre aux questions de cet exercice des fonctions Python de manipulation de listes suivantes :

- On peut créer une liste de taille n remplie avec la valeur x avec `li = [x] * n`
- On peut obtenir la taille d'une liste `li` avec `len(li)`.
- Si `li` est une liste de n éléments, on peut accéder au k-ème élément (pour $0 \leq k <$ `len(li)`) avec li[k]. On peut définir sa valeur avec li[k] = x.
- On peut concaténer deux listes `li1` et `li2` en utilisant l'opération `li1 + li2`. On utilisera aussi cette opération dans des expressions mathématiques.
- `li[a:b]` désigne la liste des éléments d'indice compris entre a et $b-1$ dans `li`. On utilisera aussi cette opération dans des expressions mathématiques.

Les autres fonctions sur les listes (`sort`, `index`, `max`, etc.) sont interdites à moins de les réécrire explicitement. L'opérateur `in` d'appartenance à une liste est interdit, mais on peut utiliser ce mot-clé dans les autres contextes (par exemple dans une boucle `for`).

Complexité : Par *complexité* d'un algorithme, on entend le nombre d'opérations élémentaires nécessaires à l'exécution de cet algorithme dans le pire cas. Lorsque cette complexité dépend d'un ou plusieurs paramètres $k_0, \ldots, k_{r-1}$, on dit que la complexité est $O(f(k_0, \ldots, k_{r-1}))$ s'il existe une constante $C > 0$ telle que, pour toutes les valeurs $k_0, \ldots, k_{r-1}$ suffisamment grandes, ce nombre d'opérations élémentaires est majoré par $C \times f(k_0, \ldots, k_{r-1})$.

✧ **Partie I** : Programmes divers

Q1 – Ecrire une fonction `fibonacci` qui prend en argument un entier `n` supérieur ou égal à 2 et renvoie la liste des `n` premiers termes de la suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$ définie par $F_0 = 0$, $F_1 = 1$ et $\forall n \geq 2, F_n = F_{n-1} + F_{n-2}$ (chaque terme est la somme des deux précédents).

```
def fibonacci(n):
    termes = [0, 1]
    for i in range(n-2):
        termes.append(termes[-1]+termes[-2])
    return termes
```

Q2 – Ecrire une fonction `indice_min` qui prend en argument une liste d'entiers `li` et renvoie l'indice d'un de ses minimums.

```
def indice_min(entiers):
    mini, indice_min = entiers[0], 0
    for i in range(1, len(entiers)):
        if entiers[i] < mini:
            mini, indice_min = entiers[i], i
    return indice_min
```

Q3 – Que renverra `indice_min([1, 0, 2, 0])` avec votre programme ?

> Dans la fonction précédente le minimum (et son indice) ne sont mis à jour qu'en cas d'infériorité strict, donc la fonction affichera l'indice du premier minimum c'est à dire 1.

Q4 – Ecrire une fonction `lettre_majoritaire` qui prend en argument une chaîne de caractères non vide et renvoie le caractère qui apparait le plus fréquemment. Ainsi, `lettre_majoritaire('abcdedde')` devrait renvoyer `'d'`.
Note : l'utilisation efficace d'un dictionnaire sera valorisée. On pourra alors utiliser l'opérateur `in`

```
def lettre_majoritaire(chaine):
    occurrences = {}
    for c in chaine:
        if c in occurrences:
            occurrences[c] += 1
        else:
            occurrences[c] = 1
    maxl = 0
    for c in occurrences:
        if occurrences[c] > maxl:
            maxl,lm = occurrences[c], c
    return lm
```

✧ **Partie II** : Saut de valeur maximale

Dans une liste de flottants `li`, on appelle *saut* un couple (i, j) avec $0 \leq i \leq j <$ `len(li)` et la *valeur* d'un saut est la valeur `li[j]-li[i]`. On va ici programmer plusieurs manières de trouver un saut de valeur maximale dans une liste. Par exemple, dans la liste `[2.0, 0.2, 3.0, 5.3, 2.0]`, un tel saut est `(1, 3)` (car `0.2` et `5.3` sont aux indices 1 et 3 respectivement).

Q1 – Ecrire une fonction `valeur` qui prend en argument une liste et un saut et renvoie la valeur de ce saut. Par exemple `valeur([2.0, 0.2, 3.0, 5.3, 2.0],(0,2))` renvoie `1.0` (car `li[2]-li[0] = 1.0`).

```
def valeur(li,saut):
    i,j = saut
    return li[j]-li[i]
```

Q2 – Donner un exemple de liste avec exactement deux sauts de valeur maximale et préciser ces sauts.

La liste `[2, 6, 1, 5]` possède deux sauts de valeurs maximale : `(0,1)` et `(2,3)` (ces deux sauts ont une valeur de 4)

Q3 – À l'aide d'un contre-exemple, montrer qu'on ne peut pas se contenter de chercher le minimum et le maximum d'une liste pour trouver un saut de valeur maximale.

Dans la liste `[2, 6, 1, 5]` le minimum est à l'indice 2 (c'est 1) et le maximum à l'indice 1 (c'est 3) et comme le minimum est après le maximum ce n'est pas le saut maximal.

Q4 – Écrire une fonction `saut_max_naif` qui renvoie un saut de valeur maximale en testant tous les couples (i, j) tels que $0 \leq i \leq j <$ `len(li)`.

```
def saut_max_naif(li):
    vsaut, imax, jmax = 0, 0, 0
    for i in range(len(li)):
        for j in range(i, len(li)):
            if valeur(li, (i,j)) > vsaut:
                vsaut, imax, jmax = valeur(li,(i,j)), i, j
    return (imax,jmax)
```

On décrit ici un algorithme utilisant le paradigme de la programmation dynamique pour résoudre ce problème : pour chaque k entre 1 et `len(li)`, on va calculer m_k l'indice du minimum de `li[0:k]`, et le couple (i_k, j_k) un saut de valeur maximale dans `li[0:k]`. Ainsi, on aura $m_1 = i_1 = j_1 = 0$ car `li[0:1]` ne comporte qu'un seul élément.

Q5 – Pour $k <$`len(li)`, expliquer comment on peut calculer efficacement m_{k+1} à partir de m_k et des valeurs dans `li`.

Par définition $m_k = \min(\texttt{li[0]}, \dots \texttt{li}[k-1])$ et donc on a immédiatement
$m_{k+1} = \begin{cases} \texttt{li}[k] \text{ si } \texttt{li}[k] < m_k \\ m_k \text{ sinon} \end{cases}$

Q6 – Justifier que la relation suivante est correcte.

$$(i_{k+1}, j_{k+1}) = \begin{cases} (i_k, j_k) \text{ si } \texttt{li}[k]\texttt{-li}[m_k] \ < \ \texttt{li}[j_k]\texttt{-li}[i_k] \\ (m_k, k) \text{ sinon} \end{cases}$$

On sait que $(i_k; j_k)$ est le saut maximal de `li[0:k]` pour déterminer le saut de valeur maximal dans `li[0:k+1]` on doit prendre le saut maximal entre : (i_k, j_k) et le nouveau saut maximal disponible qui est (mk, k) ce dernier a pour valeur li[k]-li[m_k]. On doit donc comparer la valeur de ce saut à la valeur du saut (i_k, j_k) ce qui correspond bien à la relation précédente.

Q7 – Ecrire une fonction `saut_max_dynamique` qui prend en argument une liste `li` et renvoie un saut de valeur maximale en utilisant la relation de la question 6.

```
def saut_max_dynamique(li):
    mk, ik, jk = 0, 0, 0
    for k in range(1,len(li)):
        if li[k]<li[mk]:
            mk = k
        if valeur(li,(ik,jk))<li[k]-li[mk]:
            ik, jk = mk, k
    return (ik, jk)
```

Q8 – Déterminer la complexité de votre programme dans le pire cas, puis comparer cette complexité avec celle du programme donnée en question 4 En notant n la longueur de la liste, la boucle est executée $n-1$ fois et ne contient que des opérations élémentaires donc la complexité est en $O(n)$. Pour le programme de la question 4, par contre on a deux boucles imbriquées et donc une complexité en $O(n^2)$.