Devoir - DS 1 : En route pour les JO

I - Traitement de données d'une course d'athlétisme

1 Analyse du déroulement de la course

Q1. Donner la relation entre x_{i+1}, x_i et $\int_{t_i}^{t_{i+1}} v(t)dt$.

$$x_{i+1} = x_i + \int_{t_i}^{t_{i+1}} v(t)dt$$

Q2. À l'aide de la FIGURE **??**, donner l'expression approchée de $\int_{t_i}^{t_{i+1}} v(t)\mathrm{d}t$ en fonction de v_i, v_{i+1}, t_i et t_{i+1}. En déduire une estimation de x_{i+1} en fonction de x_i, v_i, v_{i+1}, t_i et t_{i+1}.

$$\int_{t_i}^{t_{i+1}} v(t)dt \simeq \frac{v_{i+1}+v_i}{2}(t_{i+1}-t_i)$$

$$x_{i+1} = x_i + \frac{v_{i+1}+v_i}{2}(t_{i+1}-t_i)$$

Q3. Écrire une fonction `inte(LV,LT)` prenant pour arguments une liste `LV` de vitesses et une liste `LT` d'instants de mesure, et renvoyant la liste des positions estimées.

```
def inte(LV, LT):
    """renvoie la liste des positions estimées, à l'aide des listes LV et LT"""
    LX = [0] # Initialisation de la position de départ prise comme nulle.
    for i in range(len(LT)-1):
        LX.append(LX[-1] + (LV[i + 1] + LV[i]) / 2 * (LT[i + 1] - LT[i]))
    return LX
LXexp = inte(LVexp, LTexp)
```

Q4. À l'aide de la documentation en **annexe**, donner la suite d'instructions permettant d'obtenir le tracé de la FIGURE **??** : positions estimées repérées par les marqueurs points, droite horizontale à 100 m allant de 0 à 10 s en pointillés ("dotted line"), étiquettes sur les deux axes, légende. On rappelle que pour tracer une droite, il suffit de deux points.

```
import matplotlib.pyplot as plt
plt.figure()
plt.plot(LTexp, LXexp, '.')
plt.plot([0, 10], [100, 100], ':')
plt.xlabel('Temps (s)')
```

```
plt.ylabel('Position (m)')
plt.legend(['Position estimee', 'Arrivee'], loc=4)
plt.show()
```

Q5. Donner l'expression de l'instant d'arrivée T en fonction de x_{iA}, x_{iA-1}, t_{iA}, t_{iA-1} et d.

$$d = X_{iA-1} + \frac{X_{iA} - X_{iA-1}}{t_{iA} - t_{iA-1}} \cdot (T - t_{iA-1})$$

$$\Leftrightarrow T = t_{iA-1} + (d - X_{iA-1}) \cdot \frac{t_{iA} - t_{iA-1}}{X_{iA} - X_{iA-1}}$$

Q6. Écrire une fonction `arrivee(LX,LT,d)` prenant pour arguments la liste `LX` des positions estimées et la liste `LT` des instants de mesure, et renvoyant l'instant d'arrivée à la distance d, déterminé selon le principe ci-dessus. Si d n'est jamais atteinte, la fonction renverra `False`. Votre recherche de l'intervalle de position ($[x_{iA-1}\,,\,x_{iA}]$) où la distance d a été atteinte devra être linéaire.

```
def arrivee(LX, LT, d):
    """calcule l'instant d'arrivée à la distance d, par interpolation linéaire, avec while"""
    if LX[-1] < d: # Si le dernier élément de la liste est plus petit que d, c'est que cette distance n'a pas été atteinte
        return False
    i = 0 # Initialisation de la recherche linéaire
    while LX[i] < d: # Tant que la distance d n'a pas été atteinte, on incrémente l'index de recherche
        i += 1
    return LT[i-1] + (d-LX[i-1])*(LT[i]-LT[i-1])/(LX[i]-LX[i-1]) # Voir question précédente
```

Q7. Pour le plaisir!!! Reprendre la question précédente, mais en faisant une recherche dichotomique de l'intervalle de position où la distance d a été atteinte.

```
def arrivee(LX, LT, d):
    """calcule l'instant d'arrivée à la distance d, par interpolation linéaire, avec while"""
    if LX[-1] < d: # Si le dernier élément de la liste est plus petit que d, c'est que cette distance n'a pas été atteinte
        return False
# Mise en place de l'algorithme de recherche dichotomique
    borne_gauche= 0
    borne_droite=len(LX)-1
    while borne_droite-borne_gauche!=1 # L'algorithme se poursuit tant que l'encadrement n'a pas été trouvé
        borne_milieu=int((borne_gauche+borne_droite)/2) # On utilise int car la division par 2 ne renvoie pas un nombre entier (soit x.0 soir x.5, avec x la partie entière)
        if LX[borne_milieu]==d:
            return LT[borne_milieu]
        elif LX[borne_milieu]<d:
            borne_gauche=borne_mileu
        else:
            borne_droite=borne_milieu
```

```
16      return LT[borne_gauche] + (d-LX[borne_gauche])*(LT[borne_droite]-LT[
            borne_gauche])/(LX[borne_droite]-LX[borne_gauche]) # Voir question précé
            dente
```

Q8. Ici, quel est l'intérêt de la recherche dichotomique par rapport à une recherche linéaire ?

La complexité dans le pire des cas est en $\mathcal{O}(\log(n))$ pour une recherche dichotomique contre une complexité en $\mathcal{O}(n)$ pour une recherche linéaire avec $n =$`len(LX)`.

Q9. Donner l'instruction affichant à l'écran l'instant de l'arrivée à 100 m à partir des listes `LXexp` et `LTexp`.

```
1  print(arrivee(LXexp,LTexp,100))
```

Q10. Lorsqu'on applique cette fonction aux listes `LVexp` et `LTexp`, quelle grandeur physique cela permet-il d'estimer ? Justifier la réponse. Quelle est l'approximation utilisée par cette fonction pour réaliser cette estimation ?

Lorsqu'on applique la fonction `f` aux listes LVexp et LTexp on estime l'accélération par dérivation numérique de la vitesse. La dérivée $\frac{dv}{dt}$ est approximée par le taux d'accroissement :

$$\frac{dv(t_i)}{dt} \simeq \frac{v_{i+1} - v_i}{t_{i+1} - t_i}$$

On procède donc par interpolation linéaire : on suppose que la vitesse varie linéairement entre deux instants de mesure.

Q11. Si la longueur de `LVexp` et `LTexp` vaut `n`, que vaut la longueur de `f(LVexp,LTexp)` ?

La longueur de f(LVexp,LTexp) vaut $(n-1)$.

Q12. On rappelle que les deux listes passées en argument de la fonction `plot` doivent être de même longueur. Donner une instruction permettant de tracer `f(LVexp,LTexp)` en fonction du temps (on demande la courbe seule, sans format particulier ni légendes).

```
1  plt.figure()
2  plt.plot(LTexp[:-1], f(LVexp, LTexp), '.') # on a enlevé la dernière valeur de
       LTexp
3  '''Pour avoir la figure du sujet''' # pour avoir le même nombre de points
4  LY = f(LVexp, LTexp)
5  moyenne = sum(LY) / len(LY)
6  borne = 0.5 * moyenne
7  plt.plot([0, 10], [borne, borne], '--')
8  plt.plot([0, 10], [-borne, -borne], '--')
9  plt.xlabel('Temps (s)')
10 plt.ylabel('f(LVexp,LTexp) (unites SI)')
11 plt.show()
```

Q13. Écrire une fonction `instants(LY,LT)` prenant pour arguments une liste `LY` obtenue par `f` et une liste `LT` d'instants de mesure, et renvoyant le couple d'instants définis ci-dessus. Si un instant (voire les deux !) n'est pas trouvé, on lui attribuera la valeur -1.

Pour déterminer l'instant où l'on sort de la bande et qu'on reste en dessous, il faut parcourir la liste dans le sens inverse et trouvé le moment où l'on rentre dans la bande. C'est plus simple comme cela. On calcule une fonction `calcul_moyenne` pour nous aider à établir le code voulu.

```
def calcul_moyenne(liste):
    somme=0
    for i in range(len(liste)):
        somme+=liste[i]
    return somme/len(liste)

def instants(LY, LT):
    """renvoie le temps de début de la phase à vitesse constante, et le temps de début de la phase de décélération"""
    i = 0
    moyenne = calcul_moyenne(LY)
    borne = 0.5 * moyenne
    while (abs(LY[i]) > borne) and (i < len(LY)): # tant qu'on est pas rentré dans la bande ni allé à la fin
        i += 1
    if i == len(LY): # on a été à la fin sans rentrer dans la bande
        vcons = -1
    else: # on est rentré dans le tube avant d'avoir été à la fin
        vcons = LT[i] # i est le premier indice pour lequel on est dedans

    j = len(LY) - 1 # on prend le dernier indice
    while (LY[j] < -borne) and (j > 0): # Tant qu'on est au dessus de la borne inf de la bande et qu'on est pas remonté au début
        j = j - 1
    if j == len(LY) - 1: # on est arrivé au début sans trouver
        vdec = -1
    else: # on est sorti par en dessous de la bande
        vdec = LT[j + 1]

    return vcons, vdec

instants(f(LVexp, LTexp), LTexp[:-1])
```

Q14. Donner, en justifiant la réponse, la complexité de `instants(LY,LT)` en fonction de la longueur des données si l'on suppose la liste `LY` quelconque.

L'algorithme est constitué de 2 boucles successives qui s'exécutent au maximum n fois. Chacune des boucles effectue des opérations à coût constant. La complexité est donc linéaire, en $\mathcal{O}(n)$ si la liste `LY` est de longueur n.

2 Modélisation dynamique de la course

Q15. Quel principe mathématique est utilisé par `polyfit` pour déterminer la liste des coefficients polynomiaux ? Expliquer rapidement ce principe.

Les coefficients polynomiaux sont déterminés à l'aide de la méthode des moindres carrées.

On dispose d'un ensemble de données $\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_p\}$ avec $\mathbf{x}_i \in \mathbb{R}^n$. $\mathbf{x}_i$ peut donc être considéré comme un vecteur de dimension n. On vérifie alors :

$$\mathbf{x}_i = [x_{i_1}, x_{i_2}, \cdots, x_{i_n}]$$

Pour les données de l'ensemble $\mathcal{X}$, on dispose de leurs évaluations qui sont stockées dans l'ensemble $\mathcal{Y} = \{y_1, y_2, \cdots, y_p\}$. On vérifie $y_i \in \mathbb{R}$. (On peut généraliser à $\mathbf{y}_i \in \mathbb{R}^k$, mais nous ne le ferons pas ici).

Dans ce cours, quel que soit le modèle utilisé pour établir une régression des données, on s'appuie sur **la minimisation de l'erreur quadratique moyenne**. On dispose d'un modèle $\tilde{f}$ dont on cherche à optimiser les paramètres `param`. (`param` est un vecteur des paramètres que l'on chercher à optimiser). On pose alors :

$$\tilde{y}_i = \tilde{f}(\mathbf{x}_i, \texttt{param})$$

L'erreur quadratique moyenne (*mean square error* en anglais) est donnée par :

$$\begin{aligned} MSE(\texttt{param}) &= \frac{1}{p}\sum_{i=1}^{p} \parallel y_i - \tilde{y}_i \parallel^2 \\ &= \frac{1}{p}\sum_{i=1}^{p} \parallel y_i - \tilde{f}(\mathbf{x}_i, \texttt{param}) \parallel^2 \end{aligned}$$

Afin de trouver le vecteur `param` le plus adapté aux données $\mathcal{X}$ et leurs évaluation $\mathcal{Y}$, il faut résoudre le problème suivant :

$$\underset{\texttt{param}}{\arg\min} \left(\frac{1}{p}\sum_{i=1}^{p} \parallel y_i - \tilde{f}(\mathbf{x}_i, \texttt{param}) \parallel^2 \right)$$

Q16. Donner l'instruction permettant d'identifier les coefficients du modèle à partir des listes `LAexp` et `LVexp` et de les placer dans une liste `P`. Préciser, pour chaque terme de `P`, s'il s'agit de A, B ou C tels que ces coefficients sont définis dans l'équation ci-dessus.

```
1 import numpy as np
2 LAexp=f(LVexp,LTexp)
3 P=np.polyfit(LVexp,LAexp,2) # L'équation que l'on cherche à approximer est de
      degré 2 : a=A+B*v(t)+C*v(t)**2
4 C,B,A=P[0],P[1],P[2]
```

Q17. En appliquant la méthode d'Euler explicite, déterminer la relation de récurrence permettant de calculer v_{i+1} en fonction de v_i, t_{i+1}, t_i et les coefficients A, B et C.

On a, par définition de l'accélération : $a = \dfrac{\mathrm{d}v}{\mathrm{d}t}$ et $a = A + B \cdot v(t) + C \cdot v(t)^2$. Ainsi :

$$\frac{v_{i+1} - v_i}{t_{i+1} - t_i} = A + B \cdot v_i + C \cdot {v_i}^2$$

$$v_{i+1} = v_i + \left(A + B \cdot v_i + C \cdot {v_i}^2\right) \cdot (t_{i+1} - t_i)$$

Q18. Écrire une fonction `simu(LT,P)` prenant pour argument une liste `LT` d'instants et la liste `P` des coefficients précédemment obtenue, et renvoyant la liste des vitesses simulées aux instants de `LT`. La vitesse initiale v_0 sera choisie nulle.

```python
def simu(LT, P):
    """Renvoie la liste des vitesse simulées instant de LT"""
    v0 = 0
    v = v0
    V = [v0]
    for i in range(len(LT) - 1):
        v += (P[2] + P[1] * v + P[0] * v**2) * (LT[i + 1] - LT[i])
        V.append(v)
    return V
```

Q19. Donner l'expression de la quantité affichée à l'écran par ces instructions et expliquer en quoi cette quantité mesure, quantitativement, l'écart entre les mesures et la simulation.

L'expression de la quantité affichée à l'écran est (avec n le nombre de mesures) :

$$\sqrt{\frac{\sum_{i=0}^{n-1}(v_{isim}-v_{iexp})^2}{\sum_{i=0}^{n-1}(v_{iexp})^2}}$$

Cette quantité mesure, quantitativement, l'écart entre les mesures et la simulation. En effet, cette grandeur adimensionnée permet d'évaluer l'écart entre la série des v_{isim} (simulation) et la série des v_{iexp} (mesures) :

- Si les deux séries sont identiques : $(v_{isim} - v_{iexp}) = 0\,\forall i$. La quantité affichée vaut 0.
- Si les deux séries sont très différentes : les $(v_{isim} - v_{iexp})$ sont grands. La quantité affichée est grande.

On évalue la racine du rapport entre l'écart au carré entre les deux estimations et la mesure expérimentale au carré prise pour référence.

Q20. Calculer la valeur finale de la vitesse en faisant la moyenne des 10 dernières mesures de la liste `LVexp`. Cette moyenne sera noté `V_finale`.

```python
V_finale=0
for i in range(10):
    V_finale+=LVexp[-i-1]
V_finale=V_finale/10
```

Q21. On suppose disponible une fonction `simu_temps(P,t)` renvoyant la vitesse du coureur à l'instant t. Cette fonction `simu_temps` est très similaire à celle que vous avez écrite en question 18. À l'aide de la méthode de Newton, déterminez l'instant estimé où la vitesse a atteint 95% de la vitesse finale. Prévoir le cas où l'algorithme ne convergerait pas.

Tout d'abord, il faut définir la fonction dont on cherche le 0.

```python
def fonction_recherche_zero(f,P,V_finale,t):
return f(P,t)-0.95*V_finale
```

La méthode de Newton, s'appuie sur le calcul d'une suite convergente vers un point fixe (quand cette suite converge !) de la façon suivante

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

La fonction f joue le rôle de la fonction dont on cherche le 0, c'est-à-dire `simu-V_finale`. Il faut donc déterminer la dérivée de la fonction `simu-V_finale`. Pour cela, on choisit une dérivée numérique.

```
def derivee_fonction(f,P,V_finale,t,h=10**(-6)):
    return (fonction_recherche_zero(f,P,V_finale,t+h)-fonction_recherche_zero(f,
        P,V_finale,t))/h
```

Le code de la méthode de Newton est alors :

```
def Newton(t_0,P,f,derivee_f,epsilon,simu_temps,V_finale):
    '''
    t_0 : est l'instant initial avec lequel on initialise l'algorithme de Newton

    P : tuple des coefficients de la fonction polynomiales
    f : fonction dont on cherche le 0
    derivee_fonction : la fonction derivée de f
    epsilon : Critère d'arrêt de la méthode de Newton
    simu_temps : fonction simu_temps
    V_finale : Vitesse finale
    '''
    t=t_0
    compt=0 # Compteur en cas de non convergence de la méthode
    while abs(f(simu_temps,P,V_finale,t))>epsilon and compt<40:
        t=t-f(simu_temps,P,V_finale,t)/derivee_fonction(simu_temps,P,V_finale,t)

    if compt<40: # Si la boucle while s'est stoppée car on est arrivé à
        convergence, on retourne le temps estimé
        return t
    else: # Sinon on renvoie le booléen False
        return False

t_95=Newton(0,P,fonction_recherche_zero,derivee_fonction,10**(-5),simu_temps,
    V_finale) # On choisit d'initialiser la méthode de Newton à t_0=0.
```

2.1 Exploitation du modèle pour estimer les efforts sur le coureur

Q22. Expliquer quelle est l'erreur dans ce programme. Proposer une modification de la/des ligne(s) incriminées afin de corriger l'erreur. Indiquer alors dans quel ordre sont renvoyées les trois composantes recherchées.

Modification de l'algorithme :

```
def composantes(LV, LA, P):
    a, b = P[0], P[1]
    LA0, LA1, LA2 = [], [], []
    for k in range(len(LA)):
        LA2.append(a * LV[k]**2)
        LA1.append(b * LV[k])
        LA0.append(LA[k] - LA1[k] - LA2[k])
return (LA0, LA1, LA2)
```

Il y a deux erreurs dans l'algorithme proposé :

- Attention, on augmente la taille de la liste à chaque itération (on ajoute terme après terme). Exemple : `for k=2` ⇒ `LA2[k]` n'est pas défini car `len(LA2)=2`
- `LA1 + b * LV[k]` ? ? ? ?`LA1` est une liste et `b * LV[k]` est un flottant ! ! il faut que ce soit une liste.

Ordre dans lequel sont renvoyées les trois composantes recherchées :

- En 1er : LA0 : liste des f_i ⇒ propulsion
- En 2ème : LA1 : liste des $P_1 \cdot v_i = b \cdot v_i = B \cdot vi$ ⇒ traînée de frottement
- En 3ème : LA2 : liste des $P_0 \cdot {v_i}^2 = a \cdot {v_i}^2 = C{v_i}^2$ ⇒ traînée de pression (ou de forme)

Q23. Écrire une fonction `travail`, dont les arguments d'entrée sont à définir mais incluent au moins une force massique `LA`, qui retourne un flottant égal au travail massique de cette « force » tout au long de la course. Cette fonction pourra appeler n'importe quelle fonction définie dans ce sujet conformément à son en-tête, qu'elle ait été codée ou non.

Avant propos : par la méthode des trapèzes : $\int_{t_i}^{t_{i+1}} a(t)v(t)\mathrm{d}t = \left(\frac{a_{i+1}\cdot v_{i+1} + a_i \cdot v_i}{2}\right)\cdot(t_{i+1}-t_i)$

```
def travail(LA, LV, LT, d):
    LX = inte(LV, LT)
    T = arrivee(LX, LT, d)
    k= 0
    W= 0
    while LT[k] < T:
        W += (LA[k+1] * LV[k+1] + LA[k] * LV[k]) * (LT[k+1] - LT[k]) / 2
        k += 1
    return W
```

3 Stockage et mise en forme des données

Q24. Justifier que ce couple d'identifiants constitue bien une clé primaire pour cette table.

Chaque coureur ne participe qu'une fois à chaque épreuve, le couple {id_coureur, id_epreuve} est unique et peut donc constituer une clé primaire.

Q25. Écrire une requête SQL renvoyant les noms et dates de toutes les épreuves de « 100 m » enregistrées dans la base.

```
SELECT nom, date FROM epreuves WHERE distance = 100
```

Q26. Que fait la requête suivante ?

```
SELECT p.temps, p.inst_cte, p.inst_dec, p.travail
FROM performances AS p
JOIN epreuves AS e ON e.id = p.id_epreuve
WHERE e.distance = 100 AND temps <= 12
```

Cette requête donne, pour tous les « 100 m » enregistrés dans la base parcourus en moins de 12 s, les quadruplets :

- le temps, ou instant d'arrivée, en s
- l'instant initial de la phase à vitesse constante, en s
- l'instant initial de la phase de décélération, en s
- le travail massique de la force de propulsion, en J/kg.

Q27. Écrire une requête SQL renvoyant les noms et prénoms des coureurs, les noms et dates des épreuves, et les temps réalisés pour tous les « 100 m » enregistrés dans la base.

```sql
SELECT c.nom, c.prenom, e.nom, e.date, p.temps
FROM performances AS p
JOIN epreuves AS e ON e.id = p.id_epreuve
JOIN coureurs AS c ON c.id = p.id_coureur
WHERE e.distance = 100
```

Q28. Proposer une requête SQL donnant le nombre total d'épreuves s'étant déroulées durant l'année 2010 ?

```sql
SELECT COUNT(*) FROM epreuves WHERE date LIKE '2010%'
```

Q29. Proposer une requête SQL donnant le nombre total d'athlètes **distincts** ayant participé à des épreuves d'athlétisme durant l'année 2010.

```sql
SELECT COUNT(*) FROM (SELECT DISTINCT c.id
FROM performances AS p
JOIN epreuves AS e ON e.id = p.id_epreuve
JOIN coureurs AS c ON c.id = p.id_coureur
WHERE date LIKE '2010%' )
```

On utilise ici des requêtes imbriquées. On détermine tout d'abord tous les athlètes distincts ayant participé à des épreuves durant l'année 2010 et on fait le compte du nombre d'enregistrements alors retourné.

On peut faire plus simplement :

```sql
SELECT COUNT(DISTINCT c.id )
FROM performances AS p
JOIN epreuves AS e ON e.id = p.id_epreuve
JOIN coureurs AS c ON c.id = p.id_coureur
WHERE date LIKE '2010%'
```

Q30. Proposer une requête SQL donnant l'identifiant de l'athlète détenant le record du monde du 1500 m.

```sql
SELECT c.id FROM performances AS p
JOIN epreuves AS e ON e.id = p.id_epreuve
JOIN coureurs AS c ON c.id = p.id_coureur
WHERE e.distance=1500 and p.temps=(SELECT MIN(p.temps) FROM performances AS p
JOIN epreuves AS e ON e.id = p.id_epreuve
JOIN coureurs AS c ON c.id = p.id_coureur
WHERE e.distance=1500)
```

Là encore, on utilise des requêtes imbriquées avec l'opérateur d'agragéation `MIN` pour sélectionner le temps minimal sur 1500 m. Il est important de préciser dans chaque requête la distance de 1500 m, sinon on pourra se retrouver avec l'identifiant d'un athlète ayant couru par exemple un 800 m dans le même temps que le record du monde du 1500 m.

On peut également faire :

```sql
SELECT c.id FROM performances AS p
JOIN epreuves AS e ON e.id = p.id_epreuve
JOIN coureurs AS c ON c.id = p.id_coureur
WHERE e.distance=1500
ORDER BY p.temps ASC LIMIT 1
```

Q31. Proposer une requête SQL donnant l'identifiant de l'athlète et le nombre d'épreuve appartenant à l'athlète ayant le plus participé à des épreuves à la même date. Il n'y a pas de distinctions à faire car un athlète peut avoir participer à 2 épreuves identiques dans la même journée (deux 200 m dans la même journée par exemple).

```sql
SELECT c.id,COUNT(*) AS nombre_epreuves
FROM performances AS p
JOIN epreuves AS e ON e.id = p.id_epreuve
JOIN coureurs AS c ON c.id = p.id_coureur
GROUP BY e.date,c.id ORDER BY nombre_epreuves DESC LIMIT 1
```

On utilise la fonction `COUNT` pour compter le nombre d'épreuves par athlète et par date, puis utiliser `ORDER BY` et `LIMIT` pour obtenir l'athlète avec le plus grand nombre d'épreuves. Cette requête regroupe les données par date et par id, compte le nombre d'épreuves pour chaque athlète par année, les ordonne par ordre décroissant du nombre d'épreuves, puis utilise `LIMIT 1` pour obtenir le résultat correspondant à l'athlète ayant participé au plus grand nombre d'épreuves la même année.

Q32. Écrire une fonction `performances(nom,prenom,T)` prenant en arguments deux chaînes `nom` et `prenom` et le tableau `T`, qui recherche dans `T` les données relatives au coureur identifié par `nom` et `prenom` puis renvoie un couple de listes (`jours,temps`) de même longueur telles que, pour tout indice `i` strictement inférieur à cette longueur :

- `jours[i]` contienne le nombre entier de jours séparant la date de l'épreuve du 1er janvier 2000 (on utilisera la fonction `nb_jours` pour réaliser le calcul) ;
- et `temps[i]` contienne le temps en secondes, au format flottant.

On ne demande pas de trier les résultats. On rappelle que le coureur doit pouvoir être identifié même si la casse de `nom` et/ou `prenom` ne correspond pas à celle qui est enregistrée dans `T`.

```python
def performances(nom, prenom, T):
    L = [] # cette liste ne contiendra que les listes du coureur qui nous inté
        resse
    for k in range(len(T)):
        if T[k][0].lower() = nom.lower() and T[k][1].lower() = prenom.lower():
            L.append(T[k])
    date1 = [2000, 1, 1]
    jours = []
    temps = []
    for i in range(len(L)):
        date2 = L[i][3].split('-')
        for k in range(3):
            date2[k] = int(date2[k])
        jours.append(nb_jours(date1, date2))
        temps.append(L[i][4])
    return (jours, temps)
```

Q33. Écrire une fonction `top10(T)` prenant comme argument le tableau `T` défini ci-dessus et retournant un tableau au même format, mais limité aux données relatives aux 10 meilleurs temps classés par ordre croissant. On adaptera l'algorithme de tri présenté ci-dessus (il est permis de changer l'ordre des lignes de `T`) et on veillera à éviter toute opération inutile.

```python
def top10(T):
```

```
    for k in range(10):
        mini = T[k][4]
        for i in range(k + 1, len(T)):
            if T[i][4] < mini:
                temp = T[k]
                T[k] = T[i]
                T[i] = temp
                mini = T[k][4]
    return T[:10]
```

Q34. En justifiant la réponse, donner la complexité de la fonction `top10` en fonction du nombre N de performances enregistrées dans le tableau `T`.

La complexité est en $\mathcal{O}(N)$, on réalise 10 boucles de N boucles, soit $10N$ boucles.

Q35. Si l'on suppose N très grand, quel est l'intérêt de la technique utilisée ici par rapport à l'approche "naïve" consistant à trier entièrement le tableau grâce à un algorithme performant (tri rapide, tri fusion...) puis à en extraire les 10 meilleurs temps ?

Les algorithmes de tri performants ont une complexité en $\mathcal{O}(N \log N)$, pour un N grand ils sont donc moins rapides qu'une recherche des 10 plus petites valeurs.

Q36. Le tri par sélection serait-il toujours avantageux si l'on souhaitait trier entièrement `T` par temps croissants au lieu de se limiter aux 10 meilleurs temps ? Pourquoi ?

Le tri par sélection de l'ensemble de la liste a une complexité en $\mathcal{O}(N^2)$. Il est donc moins performant que les tris proposés si l'on trie toute la liste.