À rendre mercredi 6 novembre 2024

Devoir Maison n°3 – *Corrigé*

REMARQUES

- + + + Ne coupez pas une fonction ou une requête SQL sur deux pages.
- Comme dans toutes les matières, les réponses doivent être JUSTIFIÉES. On ne peut se contenter d'une réponse balancée sans explication.
- Python :
  - attention à l'**indentation**,
  - attention aux limites dans le `range` pour les parcours de listes,
  - les fonctions doivent être **commentées** clairement : variables, rôles des variables, explication du fonctionnement des différentes étapes (sans enfoncer les portes ouvertes !),
- SQL :
  - attention à la syntaxe des **jointures**,
  - attention à l'ambiguïté possible lors d'une jointure entre deux tables ayant des attributs de même nom,
  - attention à la syntaxe `nom_table.nom_attribut` (si ambiguïté lors d'une jointure),
  - attention `WHERE / HAVING`

I Base de données

On considère la base de données des jeux olympiques et paralympiques de Paris 2024 constituée des tables suivantes. On donne quelques exemples du contenu de chaque table pour illustrer leur constitution.

id	nom	prénom	ddn	pays	oly	para
1	Marchand	Léon	20020517	France	1	0
2	Aubert	Aurélie	19970609	France	0	1

Table 1 – Table `athletes`

id	ida	ide
1	2	150

Table 2 – Table `participation`
ida est une clé étrangère en lien avec la clé primaire de la table `athletes`, et ide est une clé étrangère en lien avec la clé primaire de la table `epreuves`

id	sport	nom	sexe	jeux
1	athlétisme	finale 100 m	femme	oly
2	natation	finale 100 m nage libre	femme	para
3	triathlon	équipe	mixte	oly

Table 3 – Table `epreuves`

id	ide	or	argent	bronze
1	45	1	50	46
20	150	2	60	70

Table 4 – Table `medailles`
`ide` est une clé étrangère en lien avec la clé primaire de la table `epreuves`. `or`, `argent` et `bronze` sont des clés étrangères en lien avec la clé primaire de la table `athletes` et indique qui a obtenu la médaille.

R1. Écrire la requête en langage SQL qui donne les noms et prénoms des athlètes nés à partir de 2000.

Solution:

```sql
SELECT nom, prenom
FROM athletes
WHERE ddn>=20000101
```

R2. Écrire la requête en langage SQL qui donne les noms et prénoms des athlètes ayant participé à la fois aux jeux olympiques et aux jeux paralympiques.

Solution:

```sql
SELECT nom,prenom
FROM athletes
WHERE oly=1 AND para=1
```

R3. Écrire la requête en langage SQL qui donne le nombre d'athlètes envoyé par chaque pays aux jeux olympiques.

Solution:

```sql
SELECT pays, COUNT(id)
FROM athletes
WHERE oly=1
GROUP BY pays
```

R4. Écrire la requête en langage SQL qui donne le nom du pays ayant envoyé le plus d'athlètes aux jeux paralympiques, et le nom du pays ayant envoyé le moins d'athlètes aux jeux olympiques.

Solution:

```sql
SELECT pays
FROM athletes
WHERE para=1
GROUP BY pays
ORDER BY COUNT(*) DESC
LIMIT 1
UNION
SELECT pays
FROM athletes
WHERE oly=1
GROUP BY pays
ORDER BY COUNT(*) ASC
LIMIT 1
```

R5. Écrire la requête en langage SQL qui donne le nombre d'épreuves organisées en "natation".

Solution:

```sql
SELECT COUNT(id)
JOIN epreuves
WHERE sport="natation"
```

R6. Écrire la requête en langage SQL qui donne le nombre d'athlètes paralympiques (on ne devra compter chaque athlète qu'une fois) ayant participé à des épreuves de triathlon.

Solution:

```
SELECT COUNT(DISTINCT ida)
FROM participation
JOIN epreuves
ON epreuves.id=ide
WHERE sport="triathlon" AND jeux="para"
```

R7. Écrire la requête en langage SQL qui donne les épreuves auxquelles ont participé Léon Marchand.

Solution:

```
SELECT E.nom
FROM epreuves AS E
JOIN athletes AS A
JOIN participation
ON ida=A.id AND ide=E.id
WHERE A.nom="Marchand" AND A.prenom="Leon"
```

R8. Écrire la requête en langage SQL qui donne le nom et prénom de l'athlète paralympique ayant obtenu la médaille d'or à l'épreuve féminine "finale individuelle BC1" dans le sport "boccia".

Solution:

```
SELECT A.prenom, A.nom
FROM epreuves AS E
JOIN athletes AS A
JOIN medailles AS M
ON ide=E.id AND or=A.id
WHERE E.nom="finale individuelle BC1" AND sport="boccia" AND sexe="
   femme" AND jeux="para"
```

R9. Écrire la requête en langage SQL qui donne la liste des pays ayant envoyé 4 athlètes ou moins aux jeux olympiques et paralympiques réunis.

Solution:

```
SELECT pays
FROM athletes
GROUP BY pays
HAVING COUNT(id)<=4
```

II Programme de rêve

Vous vous rendez pour une journée aux jeux olympiques ou paralympiques (oui... c'est un peu tard). Vous souhaitez vous faire votre programme de rêve.

L'épreuve d'indice i est définie par l'instant de début (d_i), l'instant de fin (f_i) et son intérêt (à qui vous lui avez attribué une valeur v_i) : $[d_i, f_i, v_i]$.

Ainsi la liste des n épreuves ayant lieu la journée où vous y êtes est notée $E = [[d_0, f_0, v_0], ..., [d_{n-1}, f_{n-1}, v_{n-1}]]$.

Vous souhaitez assister à un maximum d'épreuves tout en optimisant les intérêts que vous y portez. On cherche à déterminer le sous-ensemble S de E tel que $\sum_{i \in S} v_i$ est maximal.

Vous ne pouvez bien évidemment assister qu'à une épreuve à la fois, et vous n'assistez qu'à des épreuves entières. L'intersection entre les intervalles de deux épreuves auxquelles vous assistez est l'ensemble vide ou limité à un nombre $(d_i = f_j)$.

II.1 Questions préliminaires

R10. Écrire une fonction d'en-tête `valeur(E:list[list])->int` qui prend en argument une liste `E` du format précédent (représentant un ensemble d'épreuves) et qui renvoie la valeur totale de l'ensemble des épreuves.

Solution:

```
def valeur(E):
    """
    Entrée : liste E [[d,f,v]...]
    Sortie : valeur totale  de E
    """
    V=0
    for e in E: # parcourt des différents épreuves
        V=V+e[2] # on somme les valeurs des différents épreuves
    return V
```

II.2 Algorithme glouton

On envisage un algorithme glouton. Pour cela, on trie les épreuves par instant de fin croissant et on applique l'algorithme glouton suivant :

— on choisit l'épreuve se terminant le plus tard,

— puis on choisit l'épreuve se terminant au plus tard parmi celles qui sont compatibles avec l'épreuve choisie précédemment,

— et ainsi de suite.

On suppose qu'il n'y a pas deux épreuves qui finissent à la même heure.

II.2.a) Tri rapide

On souhaite trier les épreuves par **ordre d'instant de fin croissant**. On souhaite écrire l'algorithme de tri rapide appliqué à une liste de listes `E` de trois éléments, que l'on classe par ordre croissant de l'élément de rang 2 de chaque sous-liste :

— choisir le premier élément de la liste, que l'on notera `p`, appelé pivot,

— on partitionne la liste en deux sous listes :

  — une liste `E1` contenant les épreuves d'heure de fin strictement inférieure à l'heure de fin de `p`,

  — une liste `E2` contenant les épreuves d'heure de fin supérieure à celle de `p`. On trie récursivement chacune des deux listes et on rassemble le tout.

Par ex. `tri_rapide([[0,3,2],[3,1,4],[5,5,1],[2,2,3]])` renvoie `[[3,1,4],[2,2,3],[0,3,2],[5,5,1]]`

R11. Compléter la la fonction d'en-tête `tri_rapide(E:list[list])->list[list]` qui prend en argument une liste `E` que l'on souhaite trier par ordre croissant du deuxième élément de chaque sous-liste.

Solution: Il faut bien comparer les heures de fin (rang 1 dans les sous-listes) mais trier les sous-listes)

```
def tri_rapide(E):
    """
    entrées : liste des épreuves [[d,f,v]...]
    sortie : liste des épreuves triée par ordre de fin croissant
    """
    if len(E)<=1:
        return E
    p=E[0] # pivot
    E1,E2=[],[]
    for e in E: # parcourt de la liste des épreuves
        if e[1]<p[1]: # on compare les heures de fin du épreuve avec l'
   heure de fin du pivot
            E1.append(e)
        else:
            E2.append(e)
    return tri_rapide(E1)+tri_rapide(E2) # appels récursifs et
   concaténation
# ou
def tri_rapide(E):
    if len(E)<=1:
        return E
    p=E[0] # pivot
    E1,E2=[],[]
    for e in E[:1]:
        if e[1]<p[1]:
            E1.append(e)
        else:
            E2.append(e)
    return tri_rapide(E1)+[p]+tri_rapide(E2)
```

Pour la suite, on suppose que la liste `E` est triée par ordre d'instant de fin croissant.

II.2.b) Glouton

On donne l'exemple de la liste d'épreuves suivantes :
`epreuves=[[9,10,3],[9,13,2],[11,14,1],[11,15,3],[17,21,3],[19,22,2]]`.

R12. Mettre en œuvre l'algorithme glouton à la main pour déterminer le programme optimal des épreuves à aller voir. Quelle est la valeur totale de ce programme ? Est-ce que la solution trouvée avec l'algorithme glouton est optimale ? Peut-on trouver une meilleure solution ?

Solution: On choisit l'épreuve qui termine le plus tard : celui qui fini à 22h, et qui commence à 19h.
On remonte dans les épreuves pour trouver celui qui finit le plus tard mais avant 19h. On choisit l'épreuve qui commence à 11h et termine à 15h.
Puis on remonte, on choisit l'épreuve qui commence à 9h et qui termine à 10h.
La solution trouvée par l'algorithme glouton est `[[19,22,2],[11,15,3],[9,10,3]]`, d'une valeur totale 8.
On peut trouver une meilleure solution, au sens maximiser la valeur totale : `[[9,10,3],[11,15,3],[17,21,3]` pour une valeur totale de 9.

R13. Écrire une fonction d'en-tête `compatible(E:list[list],k:int)->int` qui détermine l'indice d'épreuve (parmi la liste `E`) compatible avec l'épreuve d'indice `k` et qui se termine au plus près de cette épreuve `k`.
La fonction renvoie -1 si aucune épreuve n'est compatible avec l'épreuve k.

Solution:

```
def compatible(E,k):
    """
    Entrées :
    E : liste des épreuves
    k : entier, n° de l'épreuve
    Sortie : i entier, qui est l'épreuve qui finit au plus près du
   début de l'épreuve k
    """
    assert k<=len(E)-1
    d=E[k][0] # début de l'épreuve k
    i=-1
    while E[i+1][1]<=d: # épreuve termine avant le début de la k
        i=i+1 # on cherche celle qui termine au plus près du début de
   la k
    return i
```

R14. Compléter la fonction d'en-tête `glouton(E:list[list])->list[int]` qui prend en argument une liste `E` des épreuves sous le format défini précédemment, et renvoie la liste `Eavoir` des épreuves à aller voir en utilisant l'algorithme glouton décrit précédemment.
On pourra avantageusement utiliser la fonction `compatible` écrite précédemment.
Pour la liste donnée en exemple, `glouton(epreuves)` renvoie `[[19,22,2],[11,15,3],[9,10,3]]`

Solution:

```
def glouton(E):
    """
    E : liste des épreuves
    Eavoir : liste des épreuves à aller voir pour maximiser le nombre
   de épreuve vu
    """
    Eavoir=[] # liste des épreuves à voir
    i=len(E)-1 # parcours de épreuves à partir de celui qui termine au
   plus tard
    Eavoir.append(E[i]) # ajout du épreuve finissant le plus tard
    i=compatible(E,i) # on cherche l'épreuve suivant compatible
    while i>=0: # s'il y a un épreuve compatible
        Eavoir.append(E[i])
        i=compatible(E,i) # on cherche l'épreuve compatible précédent
    return Eavoir
```

R15. Écrire les instructions nécessaires (on n'attend pas une fonction, mais un appel aux fonctions écrites précédemment) pour obtenir la somme des valeurs du programme élaboré grâce à l'algorithme glouton.

Solution: ATTENTION à l'utilisation d'une fonction !

```
>>> valeur(glouton(épreuves))
```

III Programmation dynamique

On cherche toujours à déterminer le sous-ensemble D de E tel que $\sum_{i\in D} v_i$ est maximal.

On suppose qu'on a une liste de n épreuves du type `E=[[d0,f0,v0],...]` triée par heure de fin croissante.

On note $S(E,k)$ la somme maximale des valeurs des épreuves qu'on peut aller voir, quand on considère les épreuves jusqu'au rang k inclus : $E_0...E_k$.

R16. Pour quelle valeur de k obtient-on la solution à notre problème ?

Solution: On obtient la solution à notre problème pour $k = n-1$, c'est-à-dire en considérant toutes les épreuves.

R17. Quelle est la valeur de $S(E,-1)$?

Solution: Pour $k=-1$, on ne considère aucune épreuve, donc $S(E,-1)=0$.

On cherche une relation de récurrence entre $S(E,k)$ (=la valeur totale en considérant les épreuves jusqu'au rang k) et $S(E,k-1)$ (=la valeur totale en considérant les épreuves jusqu'au rang $k-1$).

Pour comparer ces deux valeurs, il faut se demander si aller voir l'épreuve k est plus intéressant que de ne pas aller le voir.

R18. (a) Exprimer $S(E,k)$ en fonction de $S(E,k-1)$ si vous n'allez pas voir l'épreuve k.

Solution: Si on ne va pas voir l'épreuve k, la valeur totale est la valeur maximale obtenue en ne considérant que les épreuves $E_0...E_{k-1}$: $\boxed{S(E,k) = S(E,k-1)}$

(b) Exprimer $S(E,k)$ en fonction de $S(E,.....)$ et v_k si vous allez voir l'épreuve k. On pourra utiliser la fonction `compatible(E,k)` qui renvoie le rang de l'épreuve compatible avec l'épreuve `k` et qui se termine au plus près de cette épreuve `k`.

Solution: Si on va voir l'épreuve k, la valeur totale est la valeur du épreuve k (v_k), plus la valeur maximale obtenue en ne considérant que les épreuves compatibles avec l'épreuve k, donc d'indice inférieur à celui donné par la fonction `compatible(E,k)` :

$$\boxed{S(E,k) = v_k + S(E, \text{compatible}(E,k))}$$

(c) Comment exprimer $S(E,k)$ en fonction des deux cas précédents ?

Solution: Parmi les deux possibilités précédentes, il faut choisir celle de valeur maximale :

$$\boxed{S(E,k) = \max(v_k + S(E, \text{compatible}(E,k)), S(E,k-1))}$$

R19. Recopier et compléter la relation de récurrence suivante :

Solution: On regroupe les questions précédentes :

$$S(E,k) = \begin{cases} 0 & \text{si } k=-1 \\ \max\Big(v_k + S\big(E, \text{compatible}(E,k)\big), S(E,k-1)\Big) & \text{sinon} \end{cases}$$

R20. En quoi est-ce un problème à sous-structure optimale ? Pourquoi est-il intéressant d'utiliser la programmation dynamique ?

Solution: D'après la question R19, la solution optimale peut être construite à partir des solutions optimales à ses sous-problèmes.
De plus, les sous problèmes se chevauchent car $S\Big(E, \text{compatible}(E,k)\Big)$ et $S(E,k-1)$ font appel aux mêmes calculs. Lors de l'utilisation de la fonction récursive précédente, de nombreux calculs vont être refaits plusieurs fois.
La programmation dynamique est pertinente d'utilisation ici car elle évitera de recalculer plusieurs fois les mêmes solutions.

R21. En traduisant la relation de récurrence (R19), compléter la fonction : `sol_mem(E:list[list])->list`.
On définit le dictionnaire `dico` qui stocke les solutions successives, ses clés sont les valeurs de k, et les valeurs associées $S(E,k)$, c'est-à-dire la valeur totale maximale des épreuves auxquelles vous pouvez assister en considérant les épreuves $E_0 ... E_k$.
`dico` est une variable locale pour `sol_mem`, et une variable globale pour la fonction auxiliaire `S(E:list[list],k:int)` qui renvoie la valeur maximale quand on considère les épreuves jusqu'à celle de rang k.

Solution:

```
def sol_mem(E):
    """
    Renvoie la valeur maximale des épreuves pouvant être vues parmi les
    épreuves E
    """
    dico={-1:0} # dictionnaire pour mémoïser, pour k=-1 (aucune épreuve
), alors valeur totale =0
    def S(E,k):
        """ renvoie la valeur totale maximale en considérant les
épreuves jusqu'au rang k """
        if k in dico: # la solution a déjà été calculée
          return dico[k]
        else:
            a=S(E,k-1) # on ne choisit pas l'épreuve k
            vk=E[k][2] # valeur du épreuve k
            b=vk+S(E,compatible(E,k)) # on choisit l'épreuve k de
valeur vk
            dico[k]=max(a,b) # on cherche le maximum du nombre de
épreuve entre avoir choisi l'épreuve k et ne pas l'avoir choisi
        return dico[k]
    return S(E,len(E)-1) # il faut appliquer la fonction S pour k=len(E)
    -1 : considérer toutes les épreuves pour trouver les solutions au
    problème.
```