Correction

Python - SQL

Festival de Musique

Gestion par dictionnaire

1) Une fonction `heure_list(concert:dict) -> dict`

```python
def heure_list(concert:dict) -> dict:
    dico = {12+i: [] for i in range(12)}
    for nom, heure in concert.items():
        dico[heure[0]] += [nom]
        if heure[1] - heure[0] > 1: # si plus de 1h on déborde
            dico[heure[0]+1] += [nom]
    return dico
```

2) Une fonction `heure_nb(concert:dict) -> dict`

```python
def heure_nb(concert:dict) -> dict:
    dico = {}
    for heure, liste in heure_list(concert).items():
        dico[heure] = len(liste)
    return dico
```

3) Une fonction `maxi_heure(concert:dict) -> list`

```python
def maxi_heure(concert:dict) -> list:
    dico = heure_nb(concert)
    maxi = 0
    for heure, nb in dico.items():
        if nb > maxi:
            maxi = nb
    liste = []
    for heure, nb in dico.items():
        if nb == maxi:
            liste.append(heure)
    return liste
```

Questions préliminaires

4) Une fonction `valeur(concerts:list)` :

```python
def valeur(concerts):
    total = 0
    for info in concerts:
        total += info[2]
    return total
```

5) Le tri fusion :

a) Une fonction <u>récursive</u> `fusion(gauche, droite)` :

```python
def fusion(gauche, droite):
    if gauche == [] or droite == []: return gauche + droite
    if gauche[0][1] < droite[0][1] :
```

```
        return [gauche[0]] + fusion(gauche[1:], droite)
    else :
        return [droite[0]] + fusion(gauche, droite[1:])
```

b) Une fonction récursive `tri(concerts)` :

```
def tri(concerts):
    if len(concerts) <= 1:
        return concerts
    gauche = tri(concerts[:len(concerts)//2])
    droite = tri(concerts[len(concerts)//2:])
    resultat = fusion(gauche, droite)
    return resultat
```

c) Si C_n compte le nombre d'opérations pour un tableau de taille n, alors la fonction `tri_fusion` appelle différentes actions (diviser en deux sous-tableaux, régner = appel récursif 2 fois, la recombinaison selon la fonction `fusion` en $\mathcal{O}(n)$).
La division et la fusion des tableaux prennent un temps linaire, la complexité K_n est (dans le pire des cas) en $\mathcal{O}(n)$ donc $\exists\Omega \in \mathbb{R}/K_n \leq \Omega \cdot n$ (à partir d'un certain rang).
Les appels récursifs de la fonction s'effectuent sur des sous-tableaux de taille identique $n/2$ et ajoute donc les complexités $C_{n/2}$. On se ramène ainsi à la récurrence favorable du tri rapide $C_n = 2C_{n/2} + \Omega \cdot n$.
Pour simplifier (et sans perte de généralité), n est une puissance de 2 et $\exists k \in \mathbb{N}/n = 2^k$ (alors $k = \log_2 n$) :

L_0: $C_n = 2C_{n/2} + \Omega \cdot n$
L_1: $C_{n/2} = 2C_{n/4} + \Omega \cdot \frac{n}{2}$
L_2: $C_{n/4} = 2C_{n/8} + \Omega \cdot \frac{n}{4}$
...
L_{k-1}: $C_2 = 2C_{1=n/2^k} + \Omega \cdot \frac{n}{2^{k-1}}$

L'opération $L_0 + 2^1 L_1 + 2^2 L_2 + ... + 2^{k-1} L_{k-1}$ permet d'isoler C_n et il vient :
$C_n = 2^k C_1 + k \cdot \Omega \cdot n$ soit $C_n = nC_1 + \Omega \cdot n\log_2 n \Rightarrow \boxed{\mathcal{O}(n \log n)}$
Dans ce cas, l'algorithme présente une complexité optimale en $\mathcal{O}(n \log n)$.

Une autre démonstration : $C_n = 2C_{n/2} + \Omega \cdot n \Rightarrow \frac{C_n}{n} = \frac{C_{n/2}}{n/2} + \Omega$
Avec $n = 2^k$ et $u_k = \frac{C_{2^k}}{2^k}$, on a : $\frac{C_{2^k}}{2^k} = \frac{C_{2^{k-1}}}{2^{k-1}} + \Omega \Rightarrow u_k = u_{k-1} + \Omega$
Donc $u_k = \Omega k + u_0$ donc $\frac{C_n}{n} = \Omega \log(n) + u_0$, on retrouve la complexité en $\mathcal{O}(n \log n)$.

d) Comparaison :

	en place (oui/non)	complexité au pire	complexité au mieux
Tri fusion	non	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
Tri insertion	oui	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$

6) Pour une liste triée de concerts, la durée de fin d'un concert est inférieure à la durée de début du concert suivant.

```
def compatible(concerts):
    concerts_tri = tri(concerts)
    n = len(concerts_tri)
    for i in range(n-1):
        if concerts_tri[i][1] > concerts_tri[i+1][0]:
            return False
    return True
```

Méthode gloutonne

7) L'algorithme glouton. trie les concerts par instant de fin décroissant et choisit le concert se terminant le plus tard, puis le concert précédent compatible se terminant au plus tard, et ainsi de suite.

a) Un code `glouton(concerts)`

```python
def glouton(concerts):
    concerts_tri = tri(concerts)[::-1] # à l'envers car plus tard d'abord
    dernier = concerts_tri[0]
    liste = [dernier]
    for concert in concerts_tri:
        if concert[1] <= dernier[0]:
            dernier = concert[:]
            liste.append(dernier)
    return liste
```

b) La complexité de ce code est celle du tri rapide : $\mathcal{O}(n \log n)$.

c) Cette méthode est rapide mais donne un solution approchée (pas forcément optimale).

Force brute

8) On teste toutes les combinaisons de concerts.

a) Une fonction récursive `tousous(concerts:list)`

```python
def tousous(concerts):
    if len(concerts) == 0:
        return [[]]
    liste = []
    for sous_liste in tousous(concerts[1:]):
        liste.append(sous_liste)
        liste.append(sous_liste + [concerts[0]])
    return liste
```

b) On compte $2^n = \sum_{k=1}^{n} \binom{n}{p}$ combinaisons de concerts.
On compte le nombre de façon de choisir k concerts parmi n.

9) On parcourt l'ensemble des sous-listes d'une liste de concerts tout en vérifiant la compatibilité des listes.

a) Une fonction naïve `toucompatible(concerts:list)`

```python
def toucompatible(concerts):
    liste = []
    for sous_liste in tousous(concerts):
        if compatible(sous_liste):
            liste.append(sous_liste)
    return liste
```

b) On parcourt le 2^n combinaisons de concerts et on vérifie la compatibilité dont le coût est $\mathcal{O}(n \log n)$, l'opération globale présente une complexité $\mathcal{O}(2^n n \log n)$

10) Un dernier parcours de liste permet enfin de conclure.

a) Une fonction `maxichoix(concerts:list)->(float, list)`

```python
def maxichoix(concerts:list)->(float, list):
    maxi = 0
    for concert in toucompatible(concerts):
        if valeur(concert) > maxi:
            maxi = valeur(concert)
            choix = concert
    return maxi, choix
```

b) La complexité est celle de la fonction `toucompatible` : $\mathcal{O}(2^n n \log n)$

c) Cette méthode présente une complexité trop grande qui ne permet pas de travailler sur de grandes listes. En revanche, elle assure de trouver la solution.