DS 2

Informatique pour tous, deuxième année

Julien Reichert

Durée : 2 heures maximum.

L'exercice 4 ne sera pas corrigé pour les étudiants choisissant de faire l'exercice 5, qui porte sur les bases de données.

Exercice 1 (question de cours) : Écrire un des algorithmes de tri parmi le tri fusion et le tri rapide, au choix dans sa version en place ou non en place. Prouver sa terminaison ; calculer sa complexité dans le pire des cas par une analyse rigoureuse. Prouver aussi la correction de l'algorithme. En plus de cela, l'algorithme doit avoir un argument booléen précisant si le tri doit se faire dans l'ordre croissant ou non. La fonction `reversed` et la méthode `reverse` sont interdites, de même que la méthode `sort`, bien évidemment.

Exercice 2 : Écrire une version du tri par insertion procédant par dichotomie pour trouver l'endroit où placer chaque élément successif de la liste à trier. L'algorithme peut agir au choix en place ou sur de la mémoire extérieure. Déterminer la complexité asymptotique du programme écrit, d'abord en nombre d'affectations, puis en nombre de comparaisons. Il suffit de donner la formule de récurrence et d'utiliser le cours du chapitre précédent.

Exercice 3 : Écrire une fonction `very_bad_tri` qui agit en place sur une liste, l'effet étant que les éléments aux indices pairs sont réarrangés pour figurer dans l'ordre croissant, et que les éléments aux indices impairs sont réarrangés pour figurer dans l'ordre décroissant. Écrire aussi une fonction `very_bad_tri_2` qui agit en place sur une liste d'entiers, l'effet étant que les entiers pairs sont envoyés dans l'ordre croissant au début de la liste, et les entiers impairs sont envoyés dans l'ordre décroissant à la fin de la liste.

Pour illustrer les tris à écrire, on pourra consulter le code suivant :

```
l = [23, 7, 25, 76, 34, 52, 56, 7, 19]
very_bad_tri(l)
# l devient [19, 76, 23, 52, 25, 7, 34, 7, 56]
very_bad_tri_2(l)
# l devient [34, 52, 56, 76, 25, 23, 19, 7, 7]
```

Exercice 4 : Soient les trois algorithmes suivants, agissant sur une liste. Déterminer pourquoi les deux premiers ne sont pas des tris corrects, prouver que le troisième en est un (en particulier qu'il termine) et calculer sa complexité.

On suppose déjà écrites une fonction `engendrer_transpositions(n)` qui retourne en temps quadratique en `n` la liste des couples `(i,j)` pour $0 \leq$ `i < j < n`, une fonction `melange(l)` qui mélange en temps linéaire en la taille de `l` son argument de façon pseudo-aléatoire et une fonction `croissante(l)` qui détermine en temps linéaire en la taille de `l` si son argument est une liste croissante.

```
def petit_tri_ah_non(liste):
    transpo = engendrer_transpositions(len(liste))
    melange(transpo)
    while not (croissante(liste)):
        (i,j) = transpo.pop()
        liste[i], liste[j] = liste[j], liste[i]
```

```
def grand_tri_ah_non(liste):
    transpo = engendrer_transpositions(len(liste))
    melange(transpo)
    while not (croissante(liste)):
        (i,j) = transpo.pop()
        if liste[i] > liste[j]:
            liste[i], liste[j] = liste[j], liste[i]

def tri_castin(liste):
    transpo = engendrer_transpositions(len(liste))
    melange(transpo)
    k = 0
    while not (croissante(liste)):
        (i,j) = transpo[k]
        if liste[i] > liste[j]:
            liste[i], liste[j] = liste[j], liste[i]
            k = 0
        else:
            k += 1
```

Exercice 5 : On considère la base de données suivante, utilisée par une auto-école afin de gérer les sessions d'exercices au code. Les tables sont les suivantes, avec des attributs intuitifs :

- `CLIENTS`, avec les attributs `Id` (entier, clé primaire), `NomPrenom` (chaîne de caractères) et d'autres informations qui ne nous intéressent pas ici ;
- `SEANCES`, avec les attributs `Id_seance` (entier, clé primaire), `Date` (format spécifique ordonné), `Id_client` (entier, clé extérieure référant au client), `NbFautes` (entier) ;
- `REPONSES`, avec les attributs `Id_seance` (entier, clé extérieure référant à la séance), `Id_client` (entier, clé extérieure référant au client), `Question` (entier), `Reponses` (chaîne de caractères) ;
- `SOLUTIONS`, avec les attributs `Id_seance` (entier, clé extérieure référant à la séance), `Question` (entier), `Reponses` (chaîne de caractères) ;

Question 1 : Quelle clé primaire peut-on envisager pour les tables `REPONSES` et `SOLUTIONS` ?

Question 2 : Au vu de la structure, on peut considérer qu'une table n'était pas nécessaire. Comment procéder pour s'en passer ?

Question 3 : Écrire des requêtes permettant de récupérer les informations suivantes :

- Le nombre de clients enregistrés.
- L'ensemble des dates où une séance a eu lieu.
- Le nombre moyen de fautes d'un client précisé (on considère que son identifiant est disponible dans la valeur `n`).
- Le nombre minimal de fautes sur l'ensemble des clients et des séances.
- Le nombre maximal de questions ayant la même solution lors d'une séance précisée (on considère que son identifiant est disponible dans la valeur `s`).
- Le nombre de fautes qu'un client précisé (on considère que son identifiant est disponible dans la valeur `n`) a faites lors d'une séance précisée (de même, l'identifiant sera `s`). Pour cette dernière requête, on s'interdira d'utiliser la table `SEANCES` (l'idée est de faire une requête d'insertion à partir de données élémentaires).

Question 4 : Un client est considéré comme prêt à passer l'examen du code s'il a toujours fait moins de cinq fautes lors de ses trois précédentes séances. Écrire une requête qui détermine si c'est le cas d'un client précisé.

Question 5 : L'auto-école a décidé d'une offre de lancement promotionnelle pour fêter les deux mois de son activité : au client qui a fait le moins de fautes au total et dont l'assiduité est parfaite, dix heures de conduite sont offertes. Écrire une requête qui a permis de le trouver. On pourra supposer qu'il y a existence et unicité.