Correction du DS 2

Informatique pour tous, deuxième année

Julien Reichert

Exercice 1

Question de cours, pas de correction.

Exercice 2

La version non en place est légèrement plus facile à écrire.

```
def tri_insertion_dicho(l):
    ll = []
    for element in l:
        ind_deb, ind_fin = 0, len(ll)
        while ind_deb <= ind_fin:
            ind_mil = (ind_deb + ind_fin) // 2
            if ll[ind_mil] > element:
                ind_fin = ind_mil # pas - 1 pour savoir où placer l'élément
            elif ll[ind_mil] < element:
                ind_deb = ind_mil + 1
            else:
                ind_deb, ind_fin = ind_mil + 1, ind_mil # on sort tout de suite
        ll.insert(ind_fin, element)
    return ll
```

On note n la taille de la liste à trier. Dans le pire des cas, on applique une insertion au début de `ll`, ce qui coûte sa taille en nombre d'affectations (sans compter les affectations des indices, mais celles-ci sont en nombre négligeable) pour chaque élément de `l`, soit une complexité en $\mathcal{O}(n^2)$. En termes de comparaisons, la dichotomie fait qu'on a de l'ordre du logarithme de la taille de `ll` comparaisons pour chaque élément de `l`, soit une complexité en $\mathcal{O}(n \log n)$.

Exercice 3

```
def very_bad_tri(l): # c'est parti pour les bulles !
    for i in range((len(l) - 1) // 2, 0, -1):
        for j in range(i):
            if l[2*j] > l[2*j+2]:
                l[2*j], l[2*j+2] = l[2*j+2], l[2*j]
            if 2*j + 3 < len(l) and l[2*j+1] < l[2*j+3]:
                l[2*j+1], l[2*j+3] = l[2*j+3], l[2*j+1]

def apartheid(l):
    ind_deb, ind_fin = 0, len(l)-1
    while ind_deb <= ind_fin:
        if l[ind_deb] % 2 == 0:
            ind_deb += 1
```

```
        else:
            l[ind_deb], l[ind_fin] = l[ind_fin], l[ind_deb]
            ind_fin -= 1
    return ind_deb # suivant le dernier résultat du test, d'un côté ou de l'autre de la frontière

def split(l, deb, fin, croissant):
    ind, ind_fin = deb+1, fin
    pivot = l[deb]
    while ind <= ind_fin:
      if (l[ind] <= pivot) == croissant: # ne pas oublier les parenthèses !
          ind += 1
      else:
          l[ind], l[ind_fin] = l[ind_fin], l[ind]
          ind_fin -= 1
    l[deb], l[ind-1] = l[ind-1], l[deb]
    return ind-1

def quick_sort_aux(deb, fin, croissant=True):
    if deb < fin:
        ind = split(l, deb, fin, croissant)
        quick_sort_aux(deb, ind-1, croissant)
        quick_sort_aux(ind+1, fin, croissant)

def very_bad_tri_2(l): # tri rapide
    separation = apartheid(l)
    if l[separation] % 2 == 0:
        separation += 1
    quick_sort_aux(0, separation-1)
    quick_sort_aux(separation, len(l)-1, False)
```

Exercice 4

Le premier algorithme teste toutes les transpositions dans un ordre arbitraire et vérifie après chaque transposition si la liste est triée. Il est facile de trouver un contre-exemple minimal : une liste de trois éléments qui n'est pas triée après chacune des trois transpositions quand l'ordre est malheureux. On note que dans ce cas, on entre dans la boucle avec une liste de transpositions restantes vide, ce qui provoquera une erreur.

```
l = [3, 2, 1]
transpo = [(0, 2), (0, 1), (1, 2)] # après mélange (et le pop prend l'élément de droite)
l <- [3, 1, 2] puis l <- [1, 3, 2] puis l <- [2, 3, 1]
```

En pratique, c'était normal qu'un contre-exemple se trouve facilement : en faisant trois transpositions, on engendre au plus quatre des six permutations existantes (en comptant celle dont on dispose au début), parfois celle qu'on cherche n'est pas parmi ces quatre-là.

Le deuxième algorithme ne fait pas les transpositions qui créent une inversion (rappel : une inversion est un couple de positions pour lesquelles les éléments sont dans le mauvais ordre). Tant pis pour lui, le résultat est le même, car avec ce contre-exemple on produit les mêmes étapes, sauf la dernière qui n'est pas faite, mais quoi qu'il arrive la liste n'est toujours pas triée.

Le troisième algorithme est un tri valide mais encore plus stupide que le tri à bulles. Il repose sur l'invariant de boucle suivant : après un passage dans la boucle conditionnelle (qui coûte un $\mathcal{O}(n)$ dans la foulée), soit on a effectué une transposition qui a fait disparaître au moins une inversion dans la liste, sans en créer d'autres (éventuellement quelques inversions en sont devenues des autres, mais le nombre total a strictement décru), soit le nombre `k` a augmenté. Dans le premier cas, on progresse en direction d'une liste sans inversion, ce qui est le cas si, et seulement si, la liste est triée, provoquant l'arrêt de l'algorithme. Dans le deuxième cas, on finira par parcourir l'ensemble des

transpositions possibles, mais dès que les indices concernés correspondent à une inversion, on arrive dans le premier cas, de sorte qu'on ne déclenche jamais d'erreur de débordement de la liste des transpositions dans la mesure où après chaque transposition effectivement faite on remet `k` à zéro et surtout on ne supprime pas la transposition de la liste.

Ainsi, on a réussi à combiner une preuve de terminaison et de correction, et la complexité se lit ainsi : $\mathcal{O}(n^2)$ passages entre deux transpositions effectuées, $\mathcal{O}(n^2)$ transpositions éventuellement nécessaires plus la vérification que la liste est triée avant de faire n'importe quelle étape, ce qui veut dire que chacun des $\mathcal{O}(n^4)$ tours de boucle a un coût en $\mathcal{O}(n)$. On ne rêve pas : le tri est en $\mathcal{O}(n^5)$. Ceci étant, il s'appuie sur un tri qui frôle l'imbécillité (pour ne pas dire qui est en plein dedans) : tester toutes les permutations, et ce dernier algorithme serait en $\mathcal{O}(n!)$.

Exercice 5

Question 1 : Si on cherche une clé primaire, il faut permettre à un client de répondre à toutes les questions d'une séance, mais aussi de participer à plusieurs séances, dont on numérotera toujours les questions de `1` à `40`, et bien entendu il faut permettre à plusieurs clients de participer à une séance. Ceci étant, une réponse ne peut être donnée qu'une fois, ce qui impose pour la table `REPONSES` la clé primaire `(Id_seance, Id_client, Question)`, et de manière analogue pour la table `SOLUTIONS` la clé primaire `(Id_seance, Question)`.

Question 2 : Pour donner les solutions aux questions d'une séance, on peut les affecter à un identifiant de client fictif (par exemple `-1`) et donc se servir de la table `REPONSES` uniquement.

Question 3 : Dans l'ordre...

- `SELECT COUNT(*) FROM CLIENTS;`
- `SELECT DISTINCT Date FROM SEANCES`
  ou `SELECT Date FROM SEANCES GROUP BY Date;`
- `SELECT AVG(NbFautes) FROM SEANCES WHERE Id_client = n;`
- `SELECT MIN(NbFautes) FROM SEANCES;`
- `SELECT MAX(NbQuestions) FROM`
  `(SELECT COUNT(*) AS NbQuestions FROM SOLUTIONS GROUP BY Reponses) AS tablederivee;`
- `SELECT COUNT(*) FROM REPONSES AS R JOIN SOLUTIONS AS S`
  `ON R.Id_seance = S.Id_seance AND R.Question = S.Question`
  `WHERE Id_client = n AND R.Id_seance = s AND R.Reponses <> S.Reponses;`

Question 4 : Le mieux est de construire les requêtes étape par étape.

On récupère donc les dates des trois dernières séances pour un client particulier d'identifiant noté `n` :

```
SELECT Date FROM SEANCES WHERE Id_client = n ORDER BY Date DESC LIMIT 3
```

Ensuite, pour les nombres de fautes associés, il suffit de voir si le maximum est inférieur ou égal à cinq, en vérifiant aussi qu'il y a au moins trois séances, donc on injecte :

```
SELECT COUNT(*), MAX(NbFautes) FROM
(SELECT * FROM SEANCES WHERE Id_client = n ORDER BY Date DESC LIMIT 3) AS tablederivee
```

À présent, on passe à la jointure pour se servir du nom :

```
SELECT COUNT(*) = 3 AND MAX(NbFautes) <= 5 AS Pret FROM
(SELECT * FROM SEANCES JOIN CLIENTS ON Id = Id_client
WHERE NomPrenom = nom ORDER BY Date DESC LIMIT 3) AS tablederivee
```

En pratique, pour obtenir la liste des tels clients, le mieux est de faire une boucle dans un langage externe grâce auquel on peut faire des requêtes, et on peut même faire le traitement avec ce langage pour ne pas avoir à écrire

des requêtes trop compliquées (suivant le degré de maîtrise dans les deux langages...).

Question 5 : Pour obtenir le nombre de séances au total, on peut par exemple écrire

```
SELECT COUNT(DISTINCT Id_seance) FROM SOLUTIONS.
```

Pour filtrer les clients ayant effectué toutes les séances, on fait un regroupement et on utilise `HAVING` :

```
SELECT Id_client FROM SEANCES GROUP BY Id_client
HAVING COUNT(*) = (SELECT COUNT(DISTINCT Id_seance) FROM SOLUTIONS)
```

De même, pour obtenir le nombre minimal de fautes au total parmi de tels clients, on récupère le nombre total de fautes pour chacun de ces clients :

```
SELECT SUM(NbFautes) FROM SEANCES GROUP BY Id_client
HAVING COUNT(*) = (SELECT COUNT(DISTINCT Id_seance) FROM SOLUTIONS)
```

On obtient une table dérivée dont on recherche le minimum, qui doit être le nombre total de fautes du client sélectionné :

```
SELECT Id_client FROM SEANCES GROUP BY Id_client
HAVING COUNT(*) = (SELECT COUNT(DISTINCT Id_seance) FROM SOLUTIONS)
AND SUM(NbFautes) = (SELECT MIN(Total) FROM
(SELECT SUM(NbFautes) AS Total FROM SEANCES GROUP BY Id_client
HAVING COUNT(*) = (SELECT COUNT(DISTINCT Id_seance) FROM SOLUTIONS)) AS tablederivee)
```