Correction du DS 2

Informatique pour tous, deuxième année

Julien Reichert

Exercice 1

Question de cours, pas de correction.

Exercice 2

Une façon simple est de localiser à chaque étape le minimum parmi les valeurs non encore définitivement placées et de faire deux renversements : un qui place le minimum en question en fin de liste, et un qui le place au bon endroit. On a donc un algorithme de tri par insertion avec les opérations élémentaires restreintes.

```
def renversement(l, debut):
    if debut == 0:
        l[:] = l[::-1] # ne pas oublier [:] sinon l ne mute pas !
    else:
        l[debut:] = l[:debut-1:-1] # attention, bug si debut vaut 0

def minimum(l, debut):
    rep = debut
    for i in range(debut+1, len(l)):
        if l[i] < l[rep]:
            rep = i
    return rep

def tri_crepes(l):
    for i in range(len(l)):
        ind = minimum(l, i)
        if ind != i: # gain de temps
            renversement(l, ind)
            renversement(l, i)
```

La complexité se voit facilement (mais on a encore un nombre quadratique de comparaisons et d'affectations d'un élément), et la preuve de correction découle du tri par insertion.

Exercice 3

```
import math

def tri(l, debut=0, fin=None):
    if fin == None:
        fin = len(l)-1
    if debut >= fin-1:
        if l[debut] > l[fin]:
            l[debut], l[fin] = l[fin], l[debut]
        return
    largeur = fin-debut+1
    tri(l, debut, debut+math.ceil((2*largeur)/3-1))
    tri(l, fin-math.ceil((2*largeur)/3-1), fin)
    tri(l, debut, debut+math.ceil((2*largeur)/3-1))
```

Pour la terminaison, on se sert du variant `fin - debut`, qui décroît strictement sur chaque appel récursif imbriqué, car la « largeur » est réduite d'un tiers, avec arrêt quand elle passe à `2`. Pour la correction, on peut montrer par récurrence que `tri(l, debut, fin)` trie la zone de `l` entre `debut` et `fin` inclus, et pour établir l'hérédité on prouve que les appels récursifs ont les effets successifs suivants :

- Remarque préliminaire : on peut considérer l'intervalle (de taille fixée à $3k+r$ avec $0 \leq r < 3$) comme coupé en trois zones, de tailles respectives k, $k+r$ et k et notées A, B et C.
- Premier appel : Considérons un élément parmi le tiers des éléments les plus grands de l'intervalle. Soit il déjà dans la zone C, soit il a été éjecté de la zone A, car il y a au plus $k-1$ éléments qui lui sont supérieurs et donc pas assez pour occuper toute la zone B.
- Deuxième appel : Considérons toujours le tiers des éléments les plus grands. Ils sont tous dans les zones B et C, qui subissent un tri par hypothèse de récurrence. Alors désormais, la zone C contient le tiers des éléments les plus grands dans l'ordre croissant.
- Troisième appel : Les autres éléments sont triés eux aussi dans l'ordre croissant, CQFD.

La complexité se calcule à l'aide de la formule $c_n = 3c_{\frac{2n}{3}} + \mathcal{O}(1)$, ce qui se résout en $c_n = \mathcal{O}(n^{\log_{\frac{3}{2}} 3})$, ce qui n'est vraiment pas efficace (d'où le nom anglais de *Stoogesort*, en référence à un très ancien film).

Exercice 4

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 5

```
# Ma proposition :
def tritom(l):
    elts = {}
    for i in range(len(l)):
        assert l[i] not in elts, "Liste avec doublon"
        elts[l[i]] = i
    res = [0] * len(l)
    i = 0
    for whatever in sorted(elts.keys()):
        res[elts[whatever]] = i
        i += 1
    return res

# Proposition de Tom :
def tri_tom(l):
    n = len(l)
    ops = []
    res = list(range(1, n+1))
    for i in range(n-1, 0, -1):
        for j in range(i):
            if l[j+1] < l[j]:
                l[j+1], l[j] = l[j], l[j+1]
                ops.append([j+1, j])
    ops.reverse()
    for k in ops:
        x, y = k
        res[x], res[y] = res[y], res[x]
    return res
```

Exercice 6

Question 6a : `SELECT COUNT(*) FROM Examens WHERE Matiere = "Informatique"`

Question 6b : `SELECT MIN(Date) FROM Examens`
ou (hors-programme) `SELECT Date FROM Examens ORDER BY Date LIMIT 1`

Question 6c : `SELECT COUNT(DISTINCT Matiere) FROM Examens`

Question 6d : `SELECT COUNT(*) FROM Etudiants JOIN Notes ON Id = Etudiant`
`WHERE Note >= 10 AND Classe="MPSI2"` (pas de pénalité en cas de $>$).

Question 6e : `SELECT AVG(Note) FROM Etudiants JOIN Notes ON Etudiants.Id = Etudiant JOIN Examens ON Examens.Id = Examen WHERE Matiere="Informatique" AND Nom="Dupont" AND Prenom="Théo"`

Question 6f : `SELECT Prenom, Nom FROM Etudiants JOIN Notes ON Etudiants.Id = Etudiant`
`GROUP BY Id ORDER BY AVG(Note) DESC LIMIT 1` (hors-programme c'est plus facile)
ou

```
SELECT Prenom, Nom FROM Etudiants JOIN Notes ON Etudiants.Id = Etudiant
GROUP BY Id HAVING AVG(Note) =
(SELECT MAX(Moyenne) FROM (SELECT AVG(Note) AS Moyenne FROM Notes GROUP BY Etudiant) as td)
```