Correction du DS 2

Informatique pour tous, deuxième année

Julien Reichert

Exercice 1

Question de cours, pas de correction.

Exercice 2

Bien entendu, il faut commencer par récupérer une version triée de l'entrée. Une astuce possible revient à prendre un tri non en place (tri fusion par exemple) et à remplacer l'opérateur de comparaison utilisé par une fonction `compare`, qui peut alors être mise en argument de la fonction de tri, avec pour valeur par défaut le test qu'on aurait employé.

Ensuite, vu qu'on a payé au moins du $n\log(n)$, on peut faire un parcours supplémentaire de la version triée et voir si on a à chaque fois des égalités.

```
def merge(l1, l2, comparaison):
    n1, n2, i1, i2 = len(l1), len(l2), 0, 0
    l = [None] * (n1+n2)
    while i1 < n1 and i2 < n2:
        if comparaison(l1[i1], l2[i2]):
            l[i1+i2] = l1[i1]
            i1 += 1
        else:
            l[i1+i2] = l2[i2]
            i2 += 1
    for i in range(i1, n1): # vide si i1 = n1, termine l
            l[i+i2] = l1[i]
    for i in range(i2, n2): # vide si i2 = n2, termine l
            l[i1+i] = l2[i]
    return l

def merge_sort(l, comparaison=lambda x, y : x < y):
    n = len(l)
    if n <= 1:
        return l
    l1 = merge_sort(l[:n//2], comparaison)
    l2 = merge_sort(l[n//2:], comparaison)
    return merge(l1, l2, comparaison)

def comp_class(a, b):
    return a[1] > b[1] # merge_sort trie dans l'ordre croissant !
```

```
def classement(liste):
    ordre = merge_sort(liste, comp_class)
    place = 1
    valeur_precedente = None
    for i in range(len(ordre)):
        (nom, valeur) = ordre[i]
        if valeur_precedente != valeur:
            place = i + 1 # On réinitialise la place qui ne peut être que l'indice actuel plus un
            valeur_precedente = valeur
        print(place, nom)
```

Exercice 3

Puisqu'il faut faire le travail en place, on évitera d'aplatir la matrice, de la trier et de la reconstituer. L'idée est de reprendre un algorithme classique et de bricoler les indices. Par exemple, le tri sélection sera en $\mathcal{O}(m^2)$ pour une liste de listes comprenant en tout m éléments (les listes peuvent être de tailles distinctes ici sans poser de problème). Vu la complexité, beaucoup de précalculs et de calculs a posteriori peuvent être faits sans se poser de question.

```
def les_indices(ll, ligne_debut=0, colonne_debut=0):
    for j in range(colonne_debut, len(ll[ligne_debut])):
        yield (ligne_debut, j) # Plus propre que d'utiliser une liste qu'on constitue
    for i in range(ligne_debut+1, len(ll)):
        for j in range(len(ll[i])):
            yield (i, j)

def tri_nity(ll):
    for i, j in les_indices(ll):
        imin, jmin = i, j # où est le minimum a priori
        for ii, jj in les_indices(ll, i, j+1):
            if ll[ii][jj] < ll[imin][jmin]: # mise à jour du minimum
                imin, jmin = ii, jj
        if (imin, jmin) != (i, j):
            ll[imin][jmin],ll[i][j] = ll[i][j],ll[imin][jmin]
```

Exercice 4

On peut recycler l'astuce de l'exercice 2 en créant une fonction de comparaison maison pour laquelle 42 est inférieur à toute autre valeur. Autre possibilité : commencer par mettre les `42` au début puis faire un tri en précisant la position de départ. Le tri rapide est particulièrement adapté ici.

```
def split(l,deb,fin):
    ind,ind_fin = deb+1,fin
    pivot = l[deb]
    while ind <= ind_fin:
      if l[ind] <= pivot:
          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): # La fonction qui était locale dans les notes de cours
    if deb < fin:
        ind = split(l,deb,fin)
        quick_sort_aux(deb,ind-1)
        quick_sort_aux(ind+1,fin)

def tri_llian(l):
    place_42, n = 0, len(l)
    for i in range(n):
        if l[i] == 42:
            if i > place_42:
                l[i], l[place_42] = l[place_42], 42 # pas besoin d'écrire l[i], et 42. est moche
            place_42 += 1
    quick_sort_aux(place_42,n-1)
```

Exercice 5

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]
l <- [1, 3, 2]
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 6

Le principe est simple : on cherche une position de notre élément par dichotomie, puis on lance deux recherches dichotomiques symétriques pour trouver les deux positions où notre élément cesse d'apparaître.

L'invariant de boucle est le suivant : on a toujours les inégalités `liste[ind_min] < element < liste[ind_max]` et `liste[ind_g] == element == liste[ind_d]` (sauf éventuellement à la fin où on vérifie de quel côté de la « frontière » les indices se retrouvent).

```
def double_dicho(liste, element):
    n = len(liste)
    ind_min, ind_max = 0, n-1
    while ind_min < ind_max:
        ind_mil = (ind_min + ind_max) // 2
        if liste[ind_mil] > element:
            ind_max = ind_mil-1
        elif liste[ind_mil] < element:
            ind_min = ind_mil+1
        else:
            ind_g = ind_mil
            ind_d = ind_mil
            while ind_min < ind_g:
                ind_mil_g = (ind_min + ind_g) // 2
                if liste[ind_mil_g] < element:
                    ind_min = ind_mil_g+1
                else:
                    ind_g = ind_mil_g-1
            while ind_max > ind_d:
                ind_mil_d = (ind_max + ind_d) // 2
                if liste[ind_mil_d] > element:
                    ind_max = ind_mil_d-1
                else:
                    ind_d = ind_mil_d+1
            if liste[ind_min] < element:
                ind_min += 1
            if liste[ind_max] > element:
                ind_max -= 1
            return (ind_min, ind_max)
    raise ValueError("Introuvable")
```