Correction du DS 3

Julien Reichert

Partie 1

La réponse aux questions de cours... figure dans le cours !

Partie 2

Exercice 2.1

Il existe $n!$ permutations d'un ensemble de taille n.

Exercice 2.2

```
def engendre_peen(n):
    if n == 0:
        return [[]]
    peenm1 = engendre_peen(n-1)
    reponse = []
    for l in peenm1:
        for i in range(n):
            reponse.append(l[:i] + [n-1] + l[i:])
    return reponse
```

Exercice 2.3

```
def filtre_peen(liste, i, j, b):
    reponse = []
    for peen in liste:
        if (peen[i] < peen[j]) == b:
            reponse.append(peen)
    return reponse
```

Exercice 2.4

```
def majorite_filtre_peen(liste, i, j):
    oui = 0
    non = 0
    for peen in liste:
        if peen[i] < peen[j]:
            oui += 1
        else:
            non += 1
    return (oui > non)
```

Exercice 2.5

On commence par adapter la fonction précédente (qu'on appellera `majorite_filtre_peen2`) en modifiant la dernière ligne en `return max(oui, non)`.

```
def meilleur_filtre_peen(liste):
    assert(len(liste) != 0 and len(liste[0]) > 1)
    minmax = len(liste)
    for i in range(len(liste[0])):
        for j in range(i):
            taillemax = majorite_filtre_peen2(liste, i, j)
            if taillemax < minmax:
                minmax = taillemax
                imin = i
                jmin = j
    return imin, jmin
```

Exercice 2.6

```
def peen_tri_minmax(l):
    n = len(l)
    liste = engendre_peen(n)
    while len(liste) > 1:
        i, j = meilleur_filtre_peen(liste)
        liste = filtre_peen(liste, i, j, l[i] < l[j])
    return liste[0]

def applique_permutation(l, peen):
    reponse = [None for _ in l]
    for i in range(len(peen)):
        reponse[peen[i]] = l[i]
    return reponse

def tri_minmax(l):
    return applique_permutation(l, peen_tri_minmax(l))
```

Exercice 2.7

La formule de Stirling donne un équivalent en $+\infty$ de $n!$, à savoir $\sqrt{2\pi n}(\frac{n}{e})^n$. En faisant passer le logarithme aux équivalents et en négligeant ce qui est négligeable, on obtient que $\log_2 n! \sim_{n\to+\infty} n\log_2 n$ (en pratique valable pour toutes les bases de logarithme). Le nombre de comparaisons effectué est en pratique de l'ordre du logarithme en base deux de la factorielle de n, car dans le minmax on peut s'arranger pour que chaque comparaison sépare la liste des peen possibles en deux listes de même taille à un près. **On notera qu'il s'agit d'une heuristique, et plus précisément d'un algorithme glouton, car un minmax complet anticiperait les comparaisons suivantes pour être sûr de minimiser la profondeur de l'arbre.**

Exercice 2.8

Au vu des dépendances, il faut calculer la complexité de toutes les fonctions écrites. On note n la taille de la liste à trier.

`engendre_peen` : Espace de l'ordre de $n!$ donc temps supérieur ou égal. C'est déjà mal parti !

`filtre_peen` : Temps et espace linéaires en la taille de la liste en entrée (mais qui sera là aussi de l'ordre de $n!$, oups !).

`majorite_filtre_peen` et sa version adaptée : identique à `filtre_peen`.

`meilleur_filtre_peen` : avec la double boucle, c'est de l'ordre de n^2 fois la complexité des fonctions précédentes, donc on en est à $\mathcal{O}(n^2 n!)$.

`peen_tri_minmax` : D'après la question précédente, le nombre de tours de boucle est de l'ordre de $n\log_2 n$, ce qui est encore l'occasion de faire une multiplication et mène à $\mathcal{O}(n^3 n!\log_2 n)$. Le tri lui-même ne coûte que n à la fin.

Partie 3

Exercice 3.1

```
def arene_batonnets():
    reponse = dict()
    for i in range(22):
        for b in [True, False]:
            reponse[(b, i)] = []
            for j in range(max(0, i-3), i):
                reponse[(b, i)].append((not b, j))
    return reponse
```

Exercice 3.2

Il s'agit de la construction de l'attracteur telle qu'écrite dans le TP 6, à ceci près que l'arène est désormais un dictionnaire plutôt qu'une liste donc les clés sont à gérer un peu différemment.

```
def arene_renversee(arene):
    rep = { sommet : [] for sommet in arene } # initialisation accélérée plutôt qu'une boucle
    for sommet in arene:
        for successeur in arene[sommet]: # plus [1], l'information du propriétaire est dans la clé
            rep[successeur].append(sommet)
    return rep

def attracteur(arene, gain_eve):
    L = arene.copy()
    L_rev = arene_renversee(arene)
    attracteur = gain_eve[:]
    a_traiter = gain_eve[:] # idéalement sous forme de file, mais une pile convient aussi
    while len(a_traiter) > 0:
        en_cours = a_traiter.pop()
        for x in L_rev[en_cours]:
            if not x[0]: # la clé est un couple, comme dit ci-avant
                if x not in attracteur :
                    attracteur.append(x)
                    a_traiter.append(x)
            else :
                L[x].remove(en_cours)
                if L[x] == [] :
                    attracteur.append(x)
                    a_traiter.append(x)
    return(attracteur)
```

Exercice 3.3

En testant sur une console Python...

```
>>> attracteur(arene_batonnets(), [(True, 0)])
[(True, 0), (False, 1), (False, 2), (False, 3), (True, 4), (False, 5), (False, 6), (False, 7),
(True, 8), (False, 9), (False, 10), (False, 11), (True, 12), (False, 13), (False, 14), (False, 15),
(True, 16), (False, 17), (False, 18), (False, 19), (True, 20), (False, 21)]
```

Exercice 3.4

À chaque fois, Ève n'a qu'un seul coup gagnant et Adam peut jouer ce qu'il veut, le sommet est gagnant pour Ève, donc on admettra qu'il se ne retire qu'un bâtonnet, ce qui donne comme exemple de partie `[(False, 21), (True, 20), (False, 19), (True, 16), (False, 15), (True, 12), (False, 11), (True, 8), (False, 7), (True, 4), (False, 3), (True, 0)]` et une victoire d'Ève.

Exercice 3.5

On s'appuie sur la construction de l'attracteur pour créer un dictionnaire annonçant pour tous les sommets s'ils sont gagnants pour Ève (valeur `True` associée à la clé) ou pour Adam (valeur `False`). Dans ce cas, le nombre de mauvais coups est le nombre d'alternances entre une valeur `True` et une valeur `False` sur les sommets au cours de la partie.

```
def nombre_mauvais_coups(partie):
    arene = arene_batonnets()
    attr = attracteur(arene, [(True, 0)])
    gagnants = dict() # pour éviter un test d'appartenance en temps linéaire
    for b, sommet in attr:
        gagnants[(b, sommet)] = True
        gagnants[(not b, sommet)] = False # astuce !
    reponse = 0
    for i in range(1, len(partie)):
        if gagnants[partie[i]] != gagnants[partie[i-1]]:
            reponse += 1
    return reponse
```

Partie 4

Question 4.1

La ligne manquante triait la liste `distances` en place, par exemple `distances.sort()`.

Question 4.2

```
def dist_eucl_dim_d(pt1, pt2):
    assert len(pt1) == len(pt2) # cela ne peut pas faire de mal
    rep = 0
    for i in range(len(pt1)):
        rep += (pt1[i] - pt2[i]) ** 2
    return rep ** .5 # ou rep si on prend la distance au carré (le tri sera identique)
```

Un élément de `les_points` sera alors par exemple `((4, 2), "rouge")`, donc un point dans un espace de dimension deux, de coordonnées $(4, 2)$ et de couleur rouge.

Question 4.3

La variable `etiq` correspond à sa valeur au dernier tour de boucle, donc à l'étiquette du `k`-ième plus proche voisin (avec la gestion des éventuelles égalités faite par le tri).

Question 4.4

Changement au niveau de la boucle, avec une indentation relative :

```
for d, etiq in distances[:k]:
    if d == 0:
        return etiq
    if etiq in etiquettes:
        etiquettes[etiq] += 1 / d
    else:
        etiquettes[etiq] = 1 / d
```

Partie 5

Exercice 5.1

Des parties infinies sont possibles, par exemple en commençant en 0, 4 ou 5 et si Ève choisit toujours d'aller en 0 depuis 4 et Adam choisit toujours d'aller en 5 depuis 0 (où Ève doit aller en 4).

Exercice 5.2

La somme des étiquettes des sommets visités est 12 donc comme ce n'est pas un nombre premier la condition de victoire d'Adam, qui a déclenché le coup vers `fin`, n'est pas remplie, et celle d'Ève non plus en lisant attentivement sa description. Donc c'est un match nul (en pratique, si Adam n'était pas allé en `fin`, Ève aurait pu gagner au coup suivant).

Exercice 5.3

Le jeu n'est pas déterminé car si une partie commence en 5, on peut remarquer que quand on atteint un sommet d'Adam la somme des étiquettes rencontrées est forcément un multiple de 3 qui n'est pas 3, donc jamais un nombre premier, donc Adam ne peut jamais gagner. Dans ce cas, à la lumière de ce fait, pour ne pas risquer qu'Ève gagne, Adam a tout intérêt à arrêter la partie dès qu'il le peut (car il ne peut pas être seul à décider de rendre la partie infinie). Par ailleurs, au vu de la condition de gain, une stratégie positionnelle ne peut pas tenir compte de la somme des étiquettes rencontrées et n'a aucune chance d'être gagnante quoi qu'il se soit passé avant que le sommet ne soit atteint.

Exercice 5.4

Par définition, si un sommet est gagnant pour un joueur, ce joueur a une stratégie lui permettant de gagner la partie quoi que fasse son adversaire, mais à condition de jouer selon sa stratégie, sinon rien n'est plus garantie. La réponse est donc non dans le cas général mais en pratique comme les joueurs jouent au mieux de leur intérêt ce sera oui (pour la deuxième question).

Exercice 5.5

Dessiner les arborescences étant pénibles sur ce fichier, le choix retenu sera un exposé bien verbeux.

Sommet de départ : 0

Adam va en 3 et gagne immédiatement.

Sommet de départ : 1

Ève peut arrêter la partie sur un match nul ou tenter d'aller en 2 puis en 3 et Adam a deux choix : aller en 1 et laisser Ève avoir un coup gagnant ou aller en `fin` pour faire un match nul. Autant arrêter la partie tout de suite.

Sommet de départ : 2

Ève doit aller en 3 où Adam gagne au coup suivant.

Sommet de départ : 3

Adam gagne trivialement dès son premier coup.

Sommet de départ : 4

Ève peut aller en 0 ou en 3, mais même dans le premier cas Adam peut aller en 3 et gagner en un coup.

Sommet de départ : 5

Comme déjà signalé précédemment, après 5 et 4, Adam est amené à jouer depuis 0 ou 3 et va directement (ou indirectement après être allé de 0 à 3) sur `fin` pour ne pas prolonger une partie qu'il ne pourra jamais gagner.

Sommet de départ : `fin`

(En pratique, c'est un sommet de départ possible à ne pas oublier, mais on a immédiatement un match nul par définition.)

Exercice 5.6

En conclusion : les sommets gagnants en début de partie pour Adam sont 0, 2, 3 et 4, et les autres ne sont gagnants pour personne.

Exercice 5.7

On peut coder le graphe sur lequel l'arène repose (structure de dictionnaire quasiment incontournable) en tant que variable globale ou locale à la fonction, voire en tant qu'argument mais comme la condition de gain est spéciale la pertinence est limitée...

```
arene = {}
arene[0] = [1, 3, 5]
arene[1] = [2, 4, "fin"]
arene[2] = [3]
arene[3] = [1, "fin"]
arene[4] = [0, 3]
arene[5] = [4]
arene["fin"] = []

def premier(n):
    for i in range(2, int(n ** 0.5)):
        if n % i == 0:
            return False
    return True

def resultat_partie(partie):
    if partie == [] or partie[-1] != "fin":
        assert False
    if partie == ["fin"]:
        return 0
    total = 0
    i = 0
    while partie[i] != "fin":
        total += partie[i]
        if partie[i+1] not in arene[partie[i]]:
            assert False
        i += 1
    if i < len(partie)-1:
        assert False
    if not premier(total):
        return 0
    if partie[i-1] == 1:
        return 1
    else:
        return -1
```