ASD

DS2 - documents de cours, TD, TP autorisés - durée 2h - Éléments de correction

Vous pourrez utiliser toute fonction vue en cours, TD, TP à la condition expresse de spécifier correctement les entrées et les sorties.
Le barême et le temps sont donnés à titre indicatif.

Exercice 1 : Le tri insertion, encore [8 points, 45 minutes]

Compétence évaluée : manipuler des listes, comprendre une structure de données, évaluer des complexités, comparer des implantations.

On suppose dans cet exercice manipuler un ensemble ordonné de n entiers, tous différents, que l'on veut trier (ordonné signifie qu'il existe un suivant et un précédent).

L'objectif de l'exercice est d'étudier et de comparer la complexité en temps d'un tri sur place de cet ensemble suivant la structure de données utilisée pour stocker les n entiers. Le tri sur place choisi pour cette étude est le tri par insertion.

Partie A

Dans cette partie on suppose que les entiers à trier sont stockés dans une liste simplement chaînée, encapsulée dans une structure de donnée permettant l'accès à la première cellule et la longueur.

```
def cons (v,l):
    return { 'val' : v, 'next' : l }

l = cons (5, cons(3, cons(1, cons(2, cons(4,None)))))
s = {'first' : l, 'length':5}
```

Le tri insertion s'écrit alors :

```
def tri_insertion_a (s):
    n = s['length']
    for i in range(1,n):
        inserer_a(s,i)
```

On fournit un code de la fonction `inserer_a` dont la partie chaînage est **incorrecte** :

```
def inserer_a (s,i):
    """
    s : une liste simplement chainee
    i : le i-eme element a inserer dans s[0:i]
    """
    assert (i > 0 and i < s['length'])
    # recherche du i-eme element
    pp = None
    p = s['first']
    for j in range(i):
        pp = p
        p = p['next']
    pn = p['next']
    # recherche de la position d'insertion
    pq = None
    q = s['first']
    while q['val'] < p['val']:
        pq = q
        q = q['next']
    # chainage
```

```
    pq['next'] = p
    p['next'] = q
```

Q 1.1 Corriger la fonction `inserer_a` (s'aider d'un dessin pourra être utile).

Corrigé

```
    # chainage
    if p != q:
        pp['next'] = pn
        if pq == None:
            s['first'] = p
            p['next'] = q
        else:
            pq['next'] = p
            p['next'] = q
```

Q 1.2 Quel est le coût asymptotique (en nombre d'affections de dictionnaires) de l'insertion ordonnée du i-ème élément dans le début de la liste (jusqu'à la $(i-1)$-ème cellule) ? Justifier clairement la complexité de chacune des étapes.

Corrigé

Il faut trouver la i–ème cellule, c'est en $\Theta(i)$. Il faut trouver la position d'insertion, c'est en $\Omega(1)$ et $\mathcal{O}(i)$. Le chaînage de la i-ème cellule se fait en temps constant (3 références à modifier). Au final $\mathcal{O}(i)$.

Q 1.3 Quel est le coût du tri insertion ? Justifier.

Corrigé

$$\sum_{i=1}^{n-1} \mathcal{O}(i) = \mathcal{O}(n^2)$$

Partie B

Dans cette partie on suppose que les entiers à trier sont stockés sous la forme de couples dans une table de taille n permettant la simulation d'une liste chaînée. Les couples sont constitués des entiers à trier et de l'adresse de l'entier suivant dans la table. Par convention l'adresse du suivant du dernier entier sera `None`. La table ainsi que l'adresse du premier élément seront encapsulés dans une structure de donnée représentant l'ensemble s des entiers.

Par exemple, si l'on a 5 entiers à trier :42, 31, 15, 56, 27 ; l'ensemble pourrait être :

```
s = { 'table' : [ [31,2], [56,4], [15,1], [42,0], [27,None] ], 'first' : 3}
```

Le tri insertion s'écrit alors :

```
def tri_insertion_b (s):
    n = len(s['table'])
    for i in range(1,n):
        inserer_b(s,i)
```

Q 1.4 Ecrire en Python la fonction `inserer_b` prenant en entrée l'ensemble s et qui réalise l'insertion ordonnée du i–ème entier dans le début de la table (jusqu'au $(i-1)$–ème élément).

Corrigé

```
def inserer_b (s,i):
    # trouve le (i-1)-eme element
    p0 = s['p0']
    t = s['table']
    p = p0
    pp = None
    for j in range (i):
        pp = p
        p = t[p][1]
    pn = t[p][1]
    # trouve la position d'insertion
    r = None
    q = p0
    while t[q][0] < t[p][0]:
        r = q
        q = t[q][1]
    # chainage
    if p != q:
        t[pp][1] = pn
        if r == None:
            s['p0'] = p
            t[p][1] = q
        else:
            t[r][1] = p
            t[p][1] = q
```

Q 1.5 Donner les états successifs de la table pour réaliser le tri insertion sur l'exemple ci-dessus.

Corrigé

```
{'table': [[31, 3], [56, 4], [15, 1], [42, 2], [27, None]], 'first': 0}
{'table': [[31, 3], [56, 4], [15, 0], [42, 1], [27, None]], 'first': 2}
{'table': [[31, 3], [56, 4], [15, 0], [42, 1], [27, None]], 'first': 2}
{'table': [[31, 3], [56, None], [15, 4], [42, 1], [27, 0]], 'first': 2}
```

Q 1.6 Quel est le coût asymptotique (en nombre d'affectation d'indices de la table) de l'insertion ordonnée du i-ème entier dans le début de la table (jusqu'au $(i-1)$-ème entier) ? Justifier.

Corrigé

Même réponse que précédemment.

Q 1.7 Quel est le coût du tri insertion ? Justifier.

Corrigé

Même réponse que précédemment.

Conclusion

Q 1.8 Comparer les complexités trouvées. Expliquer pourquoi on aboutit à ce résultat ?

Corrigé

Dans les deux cas, on insère dans une liste, que le chaînage soit fait par des références à la cellule suivante ou par des indices donnant la cellule suivante.

Q 1.9 Quelle autre structure de données proposer, qui convienne au problème du tri par insertion.

Corrigé

Un tableau.

Exercice 2 : Compression d'image avec des arbres [7 points, 45 minutes]

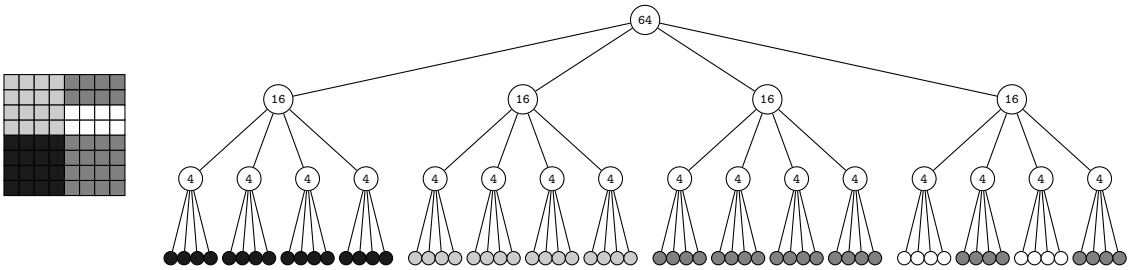
Compétence évaluée : comprendre une structure de données, réfléchir à son utilisation, manipuler des arbres.

On suppose manipuler des images comme des matrices de pixels. Chaque pixel étant un triplet d'entiers pour le codage des trois couleurs rouge, vert, bleu.

Pour simplifier on suppose ne manipuler que des images de taille $2^p \times 2^p$, p étant un entier supérieur ou égal à 1.

Pour représenter ces images on propose d'utiliser des arbres quaternaires (chaque nœud interne dispose de 4 fils). Les feuilles stockent les pixels. L'idée est qu'une image (la racine de l'arbre) est découpée en 4 parties, puis chaque partie redécoupée en 4 parties et ainsi de suite jusqu'à arriver à une image de taille 1 pixel (les feuilles).

La figure ci-dessous montre : à gauche une image de 64 pixels, au centre l'arbre représentant cette image.



On suppose que les arbres quaternaires peuvent être construits à partir des fonctions suivantes :

```
def nouveauNoeud (f1,f2,f3,f4):
    """
    Construit un nouveau noeud dont les 4 fils sont f1,f2,f3,f4.
    """
    taille = sum([f['taille'] for f in [f1,f2,f3,f4]])
    return { 'taille' : taille, 'couleur' : None, 'fils' : [f1,f2,f3,f4] }

def nouvelleFeuille (couleur,taille):
    """
    Construit une nouvelle feuille dont la couleur associee est
    couleur, taille representant le nombre de pixels de cette couleur.
    """
    assert(couleur != None)
    return { 'taille' : taille, 'couleur' : couleur, 'fils' : None }
```

Q 2.1 Etant donnée une image sous la forme d'une matrice de pixels (une liste de listes de pixels), donner le code d'une fonction `construitArbre` qui crée la représentation de l'image sous forme d'un arbre quaternaire. On suppose donnée une fonction `extraire(image,x,y,u,v)` qui retourne la partie de l'image comprise entre les pixels de coordonnées (x, y) et (u, v) (la colonne u et la ligne v étant exclues).

```
def construitArbre (image):
    """
    image[j][i] represente le pixel de coordonnees (i,j)
    0 <= i < len(t[0])
    0 <= j < len(t)
    """
```

Corrigé

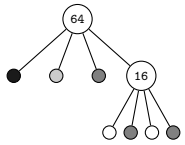
```
def construitArbre (image):
    """
    t[j][i] represente le pixel de coordonnees (i,j)
    0 <= i < len(t[0])
    0 <= j < len(t)
    """
    m = len(t)
    n = len(t[0])
    if n == m == 1:
        return nouvelleFeuille(t[0][0],1)
    else:
        n2 = n // 2
        m2 = m // 2
        f1 = construire(extraire(t,0,0,n2,m2))
        f2 = construire(extraire(t,0,m2,n2,m))
        f3 = construire(extraire(t,n2,0,n,m2))
        f4 = construire(extraire(t,n2,m2,n,m))
        return nouveauNoeud (f1,f2,f3,f4)
```

Q 2.2 Ecrire une fonction `nombreDeNoeuds(a,couleur)` qui étant donné un arbre représentant une image et une couleur (sous forme de triplet) compte le nombre de pixels de cette couleur dans l'image.

Corrigé

```
def nombreDeNoeuds (a, couleur):
    if a['fils'] == None:
        if a['val'] == couleur:
            return 1
        else:
            return 0
    else:
        return sum ( [ nombreDeNoeuds(f,couleur) for f in a['fils'] ] )
```

A partir d'un arbre quaternaire, on peut assez simplement compresser l'image. Si les 4 feuilles d'un nœud x sont de la même couleur alors on peut remplacer le nœud x par une feuille de ladite couleur. Et cela peut être fait de manière récursive. On réduit alors la taille de l'arbre : on compresse l'image. L'arbre ci-dessus montre une compression de l'arbre initial dessiné ci-dessus.



On suppose disposer d'un prédicat `deMemeCouleur(l)` qui étant donné une liste de 4 couleurs (4 triplets d'entiers) retourne vrai si les 4 couleurs sont dans identiques.

Q 2.3 Ecrire une fonction récursive `compresser(n)` qui retourne un arbre résultat de la compression telle que décrite ci-dessus (il peut être utile d'écrire des sous-fonctions).

Corrigé

```
def tous_des_feuilles (a):
    return [ est_feuille(f) for f in a['fils'] ] == [ True,True,True,True ]

def compresser (a):
    if a['fils'] != None:
        a['fils'] = [ compresser(f) for f in a['fils'] ]
        if tous_des_feuilles(a) and deMemeCouleur(couleurs(a)):
            taille = sum([f['taille'] for f in a['fils']])
            a = nouvelleFeuille (couleurs(a)[0],taille)
    return a
```

Dans les arbres construits on n'a pas conservé les coordonnées des pixels.

Q 2.4 Est-on capable de reconstruire l'image à partir de l'arbre non compressé ? Si oui expliquer brièvement comment, sinon expliquer pourquoi.

Corrigé

Oui. Comme on sait que l'image a été coupée en 4, on sait que les pixels de la feuille 1 de la racine correspondent au quart inférieur gauche, que les pixels de la feuille 2 de la racine correspondent au quart supérieur gauche, etc. Et cela récursivement. (bien sûr il faut savoir dans quel ordre l'agorithme a traité l'image)

Q 2.5 Est-on capable de reconstruire l'image à partir de l'arbre compressé ? Justifier.

Corrigé

Oui parce qu'on a gardé en mêmoire le nombre de pixels dans chaque feuille et qu'on sait que l'image est de taille $2^p \times 2^p$.

Exercice 3 : Stockage des mains de joueurs de cartes [5 points, 30 minutes]

Compétence évaluée : imaginer une structure de donnée en fonction de la complexité souhaitée des opérations.

On souhaite stocker les mains de joueurs de cartes (avec un jeu de 32 cartes). Pour chaque joueur on dispose donc de la liste de ses cartes. On ne connaît pas à l'avance le nombre (non borné) de joueurs.

L'objectif de l'exercice est de concevoir une structure de données permettant de stocker l'ensemble des cartes de tous les joueurs pour laquelle :

- la complexité en espace est en $\Theta(1)$,
- un prédicat permettant de savoir si un joueur donné dispose d'une carte donnée s'exécute en $\Theta(1)$,
- une fonction permettant d'obtenir la liste des n cartes d'un joueur donné s'exécute en $\Theta(n)$.

Q 3.1 Expliquer la structure de donnée choisie.

Corrigé

Q 3.2 Décrire en français comment se déroule l'ajout d'une carte d'un joueur.

Corrigé

Q 3.3 En dessiner une représentation pour les 4 mains suivantes :

- joueur 1 : valet de trèfle, as de cœur, 8 de pique, 9 de carreau,
- joueur 2 : roi de pique, as de carreau, 10 de pique
- joueur 3 : as de trèfle, as de pique, dame de cœur
- joueur 4 : 7 de pique, 10 de carreau, ~~dame de cœur~~, valet de pique, valet de cœur

Corrigé

Q 3.4 Justifier la complexité en espace.

Corrigé

Q 3.5 Justifier la complexité en temps du prédicat testant si un joueur possède une carte.

Q 3.6 Justifier la complexité en temps de la construction de la liste des cartes d'un joueur.