TP1

Avec corrigé

Introduction

Le but de ce TP est de mettre en œuvre des techniques revues dans le premier cours. On mettra l'accent sur les calculs de complexité.

1 Premier exemple : nombre de 0 contigus

Soit n un entier naturel non nul est `t` un tableau de taille n (représenté en pratique par une liste de longueur n) dont les termes valent 0 ou 1. Le but de cet exercice est de trouver le nombre maximal de zéros contigus dans `t` . Par exemple, le nombre maximal de zéros contigus dans de la liste `t1` suivante vaut 4 :

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
t[i]	0	1	1	1	0	0	0	1	0	1	1	0	0	0	0

1. Écrire une fonction `nombreZeros(t,i)` , prenant en paramètre une liste `t` (de longueur n) et un indice `i` compris entre 0 et $n-1$, et renvoyant :
   - `0` , si `t[i]==1`
   - le nombre de zéros consécutifs dans `t` à partir de `t[i]` inclus, si `t[i]==0`.
2. Comment obtenir le nombre maximal de zéros contigus d'une liste `t` connaissant la liste des `nombreZeros(t,i)` pour $0 \leq i \leq n-1$?
3. En déduire une fonction `nombreZerosMax(t)` renvoyant le nombre maximal de zéros contigus de la liste `t`.
   Tester avec `t1`.
4. Quelle est, en fonction de n, la complexité de la fonction `nombreZerosMax(t)` ?
5. On peut écrire un algorithme plus performant pour obtenir le même résultat (complexité linéaire) : compléter la fonction `nombreZerosMax2(t)` ci-dessous.

```
def nombreZerosMax2(t):
    n=len(t)
    N=0#destin\'e \'a contenir le r\'esultat
    k=0
    nb=0#nombre de z\'eros contigus \'a gauche de t[k]
    while k<n:
        if t[k]==0:
            ...
            ...
                ...
        elif t[k]==1 :
            ...
        k+=1
    return N
```

6. Démontrer la correction de cet algorithme en en dégageant un invariant.

Corrigé

1.
```
def nombreZeros(t,i):
    n=len(t)
    k=0
    while i+k<n and t[i+k]==0:
        k+=1
    return k
```

```
def nombreZerosMax(t):
    n=len(t)
    N=0#destin\'e \'a contenir le r\'esultat
    for i in range(n):
        nb=nombreZeros(t,i)
        if nb>N :
            N=nb
    return N
```

3. Chaque appel à `nombreZeros(t,i)` nécessite de l'ordre de $n-i$ opérations élémentaires. Lorsque l'on fait la somme pour $0 \leq i \leq n-1$, on obtient une complexité de en $\mathcal{O}(n^2)$.
Remarquons que le pire des cas pour cet algorithme est celui où le tableau ne contient que des 0.

4. On va parcourir une seule fois le tableau, en gardant en mémoire le nombre de zéros contigus que l'on vient de parcourir.

```
def nombreZerosMax2(t):
    n=len(t)
    N=0#destin\'e \'a contenir le r\'esultat
    k=0
    nb=0#nombre de z\'eros contigus \'a gauche de t[k]
    while k<n:
        if t[k]==0:
            nb+=1
            if nb>N:
                N=nb
        elif t[k]==1 :
            nb=0
        k+=1
    return N
```

5. L'invariant est le suivant :

   - `nb` est le nombre de zéros contigus à gauche (strictement) de la case d'indice `k`.
   - `N` est le nombre maximum de zéros contigus dans `T[:k]`.

2 Tri insertion

Principe : À chaque étape, le sous-tableau $T[:k]$ étant supposé trié, on insère l'élément $T[k]$ dans $T[:k]$ de sorte que $T[:k+1]$ soit trié ; ceci jusqu'à ce que T soit trié.
Remarquons que la structure de tableau ne permet pas l'insertion. On procèdera donc ainsi : on stocke $T[k]$ dans une variable temporaire ; puis à partir de $i = k-1$ (en décrémentant i) on décale $T[i]$ d'une case vers la droite tant que $T[i] > T[k]$. Enfin, on place $T[k]$ à la place du dernier $T[i]$ décalé.

1. Compléter la fonction suivante afin qu'elle trie le tableau `T` selon ce principe.

```
def triInsert(T):
    n = len(T)
    k = 1 # On ins\'ere T[k] dans T[:k] tri\'e
    while k < n:
        tmp = ...
        j = ...
        while ..... and ....:
            T[j] = ....
            j-=1
        T[j] = ....
        k+=1
```

2. Montrer la correction de cet algorithme en donnant un invariant de la boucle externe ainsi qu'un invariant de la boucle interne.

3. Déterminer la complexité de cet algorithme (dans le pire puis dans le meilleurs des cas).

Corrigé :

1.
```
def triInsert(T):
    n = len(T)
    k = 1#On ins\'ere T[k] dans T[:k] tri\'e
    while k < n:
        tmp = T[k]
        j = k
        while j > 0 and T[j-1] > tmp:
            T[j] = T[j-1]
            j-=1
        T[j] = tmp
        k+=1
```

2. Invariant de la boucle externe : $T[:k]$ trié.

   Invariant de la boucle interne : `T[:j]` trié ; `T[j+1:k+1]` trié contenant des éléments supérieur ou égaux à `tmp`.
   La condition d'arrêt est ici importante : `tmp>=T[j-1]` ; ce qui assure que `T[:k+1]` est trié à l'arrêt de la boucle.

3. La boucle interne nécessite dans le pire des cas de l'ordre de k opérations. Donc, comme dans le tri sélection, on aboutit à une complexité en $\mathcal{O}(n^2)$.
   Dans le meilleurs des cas (tableau déjà trié), on voit que la boucle interne ne nécessite qu'une comparaison. On obtient donc une complexité linéaire.

3 Tri fusion

3.1 Principe

Le principe du tri fusion (merge sort) est le suivant : on divise le tableau à trier en deux parties égales (à une unité près), on trie chacune de ces parties, puis on reconstitue le tableau (trié) en fusionnant les deux sous-tableaux de sorte à respecter l'ordre.
Dès l'énoncé de ce principe, on voit que l'on aura intérêt à programmer cet algorithme de manière récursive, en appelant la fonction elle-même pour trier les sous-tableaux.

3.2 Programme

Compléter dans le fichier `Tri_Fusion_A_Completer` :

- la fonction `fusion(t1,t2)` afin qu'elle effectue la fusion de deux tableaux triés et un nouveau tableau `t` trié.
- la fonction `tri_fusion(L)` qui réalise un tri selon le principe énoncé ci-dessus.

Cette fonction a l'inconvénient de nécessiter à chaque itération une copie complète du tableau, d'où une utilisation importante d'espace mémoire. L'algorithme de tri fusion ne peut pas être traité totalement " en place ".
Corrigé : Voir le fichier `Tri_fusion.py`

3.3 Correction et complexité

La terminaison est assurée par le fait que les appels récursifs se font sur des tableaux de taille strictement inférieure à la taille du tableau initial. On atteint donc bien le cas de base $n = 1$.
La correction du tri fusion programmé ci-dessus est assurée :

- par la correction de la fusion ;
- par la démonstration par une récurrence forte de la propriété suivante :
  $\mathcal{P}_n$: lorsque $len(L) = n$, le résultat de `tri_Fusion(L)` est la liste `L` triée.
  La récurrence est claire dès que l'on s'est assuré de la correction de la fusion.

Complexité :

1. Établir la relation de récurrence que vérifie le nombre d'opérations $C(n)$ néssaire au traitement d'un tableau de taille n. (On rappelle qu'il existe une constante K telle que la fusion de deux tableaux de tailles n nécessite au plus $K \times n$ opérations).
2. En déduire la complexité $C(n)$ du tri fusion (on pourra raisonner dans le cas où $n = 2^k$ en posant $u(k) = C(2^k)$ puis $v(k) = \dfrac{u(k)}{2^k}$).

Corrigé :

1. $C(n) = 2C(n//2) + Kn$
2. On étudie la suite $u(k) = C(2^k)$. La relation qui précède se traduit par $u(k+1) = 2u(k) + K \times 2^{k+1}$.
   Posons alors $v(k) = \dfrac{u(k)}{2^k}$.
   On a $v(k+1) = v(k) + K$; d'où $v(k) = v(0) + k \times K$ et donc $u(k) = \mathcal{O}(k \times 2^k)$.
   Conclusion : $C(n) = \mathcal{O}(n \log n)$.

4 Tri rapide

4.1 Principe

Le principe du tri rapide (quicksort) est le suivant : on choisit un pivot `p` dans le tableau `T` (par exemple son premier élément), puis on dispose tous les éléments inférieurs à `p` à gauche de `p` (tableau `Tg`) et les autres à sa droite (tableau `Td`). Puis on recommence avec `Tg` et `Td`, jusqu'à ce que l'on tombe sur des tableaux de taille un. Dans cette description, le caractère récursif de l'algorithme apparaît.
Remarquons qu'après la première étape, `p` est à sa place définitive. La correction de l'algorithme est donc la conséquence d'une récurrence immédiate : la construction assure que si l'on sait trier des tableaux de tailles strictement inférieures à celle de `T` (`Tg` et `Td`), alors on sait trier `T`.
Exemple : Écrire une trace de cet algorithme (encore un peu imprécis car on n'a pas précisé comment se faisait le "pivotage") avec `T=[5,7,3,4,8,1,2,9]` .

4.2 Première implémentation

1. Écrire une fonction `pivot(l)` qui renvoie une liste de trois éléments :
   - le pivot (premier élément) ;
   - une liste constituée des éléments inférieurs ou égaux au pivot ;
   - une liste constituée des éléments strictement supérieurs au pivot.

   On s'autorise l'extraction de listes et l'usage de listes intermédiaires.
2. Écrire à partir de la fonction précédente une fonction récursive `triRapide(l)` qui trie `l` selon le principe expliqué ci-dessus.

Corrigé : voir fichier `Tri_Rapide`

4.3 Implémentation en place

Cet algorithme a l'avantage de pouvoir être programmé en place en n'effectuant que des échanges.
Dans le fichier `Tri_Rapide_A_Completer`

1. Compléter la fonction `pivotage(T,g,d)` qui prend comme pivot `p` un élément (par exemple `T[g]`) de `T[g:d]` et place les éléments de `T[g:d]` inférieurs (ou égaux) à `p` à sa gauche et ceux supérieurs à sa droite ; ceci tout en laissant le reste du tableau inchangé.
   On aura intérêt à laisser provisoirement `p` à gauche de `T[g:d]` .
   On demande également que la fonction retourne l'indice final du pivot.
2. Compléter la fonction récursive `triRapideEnPlacePart(T,g,d)` qui trie la partie `T[g:d]` selon le principe décrit ci-dessus ; puis écrire une fonction `triRapideEnPlace(T)` .

Corrigé : voir fichier `Tri_Rapide`

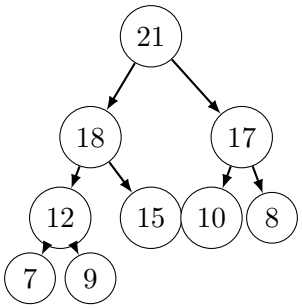
4.4 Complexité

Dans le pire des cas, le pivot choisi à chaque étape est le plus petit élément du tableau restant à trier et rien ne bouge (cela correspond à un tableau déjà trié). La taille du tableau à trier diminue alors seulement d'une unité à chaque étape. Et comme le traitement du pivotage est de l'ordre de la taille du tableau restant à trier, cela nous amène à une relation de récurrence pour la complexité du type : $C(n) = C(n-1) + n$. Ceci entraîne une complexité en $\mathcal{O}(n^2)$.
Dans le meilleur des cas, le tableau est coupé en deux parties (presque) égales à chaque appel récursif. On se retrouve exactement dans le cas (du point de vue complexité) du tri fusion, ce qui donne une complexité en $\mathcal{O}(n \log n)$
On admettra que le tri rapide a une complexité moyenne en $\mathcal{O}(n \log n)$, ce qui rend son utilisation en pratique intéressante.

5 Complément : tri par tas

5.1 Principe

On commence par quelques définitions (données de manière succintes) conduisant à la structure de données de tas.
Un arbre binaire parfait est une structure d'arbre binaire où seule la dernière ligne (constituée de "feuilles") est éventuellement incomplète, chaque nœud portant une étiquette (ci-dessous un entier).
Un arbre binaire est dit partiellement ordonné lorsque l'étiquette de chaque nœud est supérieure ou égale aux étiquettes de ses fils.
Un arbre binaire parfait partiellement ordonné est appelé tas.
On appelle hauteur h d'un arbre binaire le nombre de "lignes" constituant le tas. Remarquons que, si l'on note n le nombre de nœuds de l'arbre binaire, on a $2^{h-1} \leq n \leq 2^h - 1$.
Exemple :



On numérote chacun des nœuds et des feuilles en attribuant le numéro 1 au sommet puis en parcourant l'arbre de haut en bas et de gauche à droite. Sur l'exemple précédent, le nœud numéro 6 porte l'étiquette 10. Cette numérotation permet de représenter un tas par une liste python d'entiers (éventuellement en laissant le premier élément de la liste libre si l'on veut respecter la numérotation des listes en python).

Le principe général du tri par tas est le suivant :

- On part d'un tableau d'entier de taille n (sous forme d'une liste `l` de taille $n+1$; `l[0]` n'entre pas en jeu et sera pris égal à `None`).
- On transforme `l` (c'est à dire `l[1:]`) en tas.
- Dans ce tas, le plus grand élément est en position 1 ; on peut donc le mettre dans sa position finale en l'échangeant avec `l[n]` . Considérons alors `l[1:n]`. Les deux arbres fils du premier nœud sont toujours des tas ; seul `l[1]` n'est éventuellement pas conforme à cette définition. On rétablit alors la structure de tas en faisant descendre `l[1]` dans le tas (les détails seront donnés plus loin). Une fois cette structure de tas rétablie, on échange la nouvelle valeur de `l[1]` avec `l[n-1]` .
  On applique ensuite cette même procédure à `l[1:n-1]` .... jusqu'à ce que tous les éléments de `l` soient en bonne position.

5.2 Fonctions parent et fils

Afin de travailler sur cette structure, on aura tout d'abord besoin de fonctions donnant la position des fils et du parent d'un nœud.

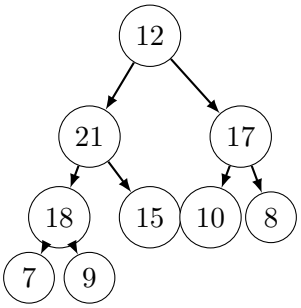
1. Écrire une fonction `parent(i)` au nœud numéro i du tas le numéro de son parent.
2. Écrire des fonctions `filsG(i)` et `filsD(i)` qui associe au nœud numéro i du tas le numéro de son fils gauche et de son fils droit. On pourra tolérer que les fonctions `fils` renvoient un indice qui n'est pas dans la liste (en tenant compte de cette convention ensuite).

5.3 Rétablissement de la structure de tas

Dans la description de l'algorithme, on a vu que l'on aurait besoin :

- de transformer notre liste initiale en tas ;
- de rétablir la structure de tas après échange de son plus grand élément avec un élément inconnu.

La deuxième procédure va servir pour la première, on commence donc par le rétablissement de la structure de tas.
Lorsque le plus grand élément d'un tas à été échangé, on se retrouve avec un (sous)-arbre dont le sommet ne respecte pas la structure de tas tandis que les deux sous-arbres partant de ce sommet sont des tas, comme dans l'exemple suivant :



Pour rétablir la structure de tas, le principe est d'échanger la valeur du sommet avec la valeur maximale de ses fils ; et ce jusqu'à ce que l'arbre soit à nouveau un tas.

1. Commencer par rétablir la structure de tas selon le principe décrit ci-dessus à la main sur l'exemple précédent.
2. Écrire une fonction `entasser(l,s,k)` (`l` de longueur $n+1$, $1 \leq s < k \leq n$) qui rétablit la structure de tas dans le sous-arbre de `l[s:k+1]` de sommet `s` , sous les hypothèses faites ci-dessus.
   Remarque : les paramètres `s` et `k` anticipent le fait que l'on souhaite programmer cet algorithme en place.
   Tester par exemple avec la liste représentant l'exemple ci-dessus.

5.4 Construction d'un tas à partir d'un arbre quelconque

Pour transformer la liste initiale `l` en une liste représentant un tas, on procède de la manière suivante.
Cette liste représente, avec les conventions ci-dessus, un arbre binaire qui n'est pas un tas.
On part cette fois du dernier niveau de l'arbre. Les feuilles qui le constituent sont des tas. On considère ensuite les sous-arbres constituant les deux derniers niveaux. Ce sont des tas sauf si leur sommet ne respecte pas cette structure. On peut donc en faire des tas en utilisant la même technique que ci-dessus. On continue à remonter ensuite dans l'arbre en faisant en sorte qu'après l'étape i, les sous-arbre constituant les i dernières lignes soient des tas.

1. Prenons en exemple la liste `l=[None,18,9,17,10,8,12,15,7,21]` .
   Dessiner l'arbre binaire correspondant à cette liste puis le transformer à la main en tas selon la procédure décrite ci-dessus.
2. Écrire une fonction `transformerEnTas(l)` qui transforme une liste quelconque `l` de taille $n+1$ (toujours avec `l[0]=None`) en une liste représentant un tas.
   Tester, par exemple avec la liste précédente.

5.5 Implémentation du tri

1. Écrire enfin la fonction `triParTas(l)` complète.
   Tester sur la liste précédente.

2. Montrer que la complexité de ce tri est en $\mathcal{O}(n\log(n))$.

Corrigé :

- Pour les fonctions : voir le fichier `Tri_tas.py`.

- Pour la complexité : reprenons le principe de ce tri.
  Il est constitué d'une boucle externe de longueur n (la répétition $n-1$ fois de l'insertion d'une valeur dans le tas). À l'intérieur de cette boucle, on insère un élément dans un tas de taille (nombre d'éléments) inférieure à n. Cette insertion nécessite deux comparaisons pour chaque passage de l'élément à insérer au niveau inférieur. Le nombre d'itérations est donc au maximum la hauteur du tas ; et celle-ci est majorée par $h = \log_2(n)$.
  On obtient ainsi une complexité dans le pire des cas de l'ordre de $n\log(n)$.
  Remarquons que dans le meilleur des cas (liste déjà triée) la boucle interne est de longueur 1 ; on obtient donc une complexité en $\mathcal{O}(n)$.
  Il faut également tenir compte de la mise sous forme de tas de la liste initiale. Par les mêmes arguments, on obtient également une complexité en $n\log(n)$.