TP1

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 : plus grande liste d'entiers consécutifs

Soit n un entier naturel non nul est `t` une liste d'entier de longueur n. Le but de cet exercice est de trouver le nombre maximal d'entiers consécutifs dans `t` . Par exemple, le nombre maximal d'entiers consécutifs 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]`	2	3	10	4	5	6	7	1	11	12	13	0	9	14	15

1. Écrire une fonction `nombreCons(t,i)`, prenant en paramètre une liste `t` (de longueur n) et un indice `i` compris entre 0 et $n-1$, et renvoyant le nombre d'entiers consécutifs dans `t` à partir de `t[i]` inclus.
   Tester avec une liste représentant le tableau précédent et successivement `i` égal à 0, 1, 3, 13.

2. Comment obtenir le nombre maximal d'entiers consécutifs d'une liste `t` connaissant la liste des `nombreCons(t,i)` pour $0 \leq i \leq n-1$?

3. En déduire une fonction `nombreConsMax(t)` renvoyant le nombre maximal d'entiers consécutifs de la liste `t`.
   Tester avec `t1`.

4. Quelle est, en fonction de n, la complexité de la fonction `nombreConsMax(t)` ?

5. On peut écrire un algorithme plus performant pour obtenir le même résultat (complexité linéaire) : compléter la fonction `nombreConsMax2(t)` ci-dessous.

```
def nombreConsMax2(t):
    n=len(t)
    i=0
    l=1 # On maintient : t[i,i+l] est une liste d'entiers consécutifs
    m=1 # Nombre max d'entiers consécutifs dans t[:i+l]
    while ...:
        if t[i+l]==t[i+l-1]+1:
            ...
            if l>m:
                ...
        else:
            i=...
            l=...
    return m
```

6. Démontrer la correction de cet algorithme.

2 Palyndrome

Un palyndrôme est un mot qui peut se lire dans les deux sens (comme `'laval'`).
On manipulera dans cette partie des chaînes de caractères. On cherche à écrire une fonction qui teste si une chaîne de caractères est un palyndrome.

1. Écrire une fonction `estPalyndrome(c)` qui prend en entrée une chaîne de caractères `c` et qui renvoie `True` si `c` est un palyndrome et `False` sinon.

2. On souhaite écrire une version récursive de cette fonction. Pour cela, on remarque qu'une chaîne de caractères (contenant au moins deux caractères) est un palyndrome si et seulement si

   - sa première lettre est la même que sa dernière ;
   - la chaîne obtenue en enlevant la première et la dernière lettre est un palyndrome.

   Écrire une fonction récursive `estPalyndromeRec(c)` qui fait la même chose que la précédente de manière récursive en tenant compte de cette remarque.
   On rappelle que pour une chaîne de caractères `c`, comme pour les listes, la commande `c[i:j]` renvoie la chaîne de caractères extraite de `c` commençant à l'indice `i` (inclu) et l'indice `j` (exclu).

3. Dans la première version de cette fonction récursive, on s'est autorisé à recopier des morceaux de la chaîne de caractères initiale. On va maintenant construire une version qui évite cet inconvénient. Pour cela, on construit une fonction auxiliaire qui renvoie `True` si `c[i:j]` est un palyndrome et `False` sinon.
   Compléter :

```
def estPalyndromeRecV2(c):
    n = len(c)
    def estPalyndromeRecV2_Aux(c,i,j):
        """ 0 <= i < j <=n
            renvoie True si c[i:j]  est un palyndrome """
        if ... :  # cas de base
            return ..
        else:
            return (...  ) and ...
    return estPalyndromeRecV2_Aux(c,...,...)
\end{enumerate}
```

   Tester ces fonctions (avec `'esoperesteicietserepose'` par exemple).

3 Tri insertion avec dichotomie

Rappelons tout d'abord le principe général du tri insertion. Partant d'un tableau de nombres de taille n (représenté ici par une liste python) noté `T`, on souhaite le trier, sans créer de nouveau tableau, dans l'ordre croissant.

- On insère tout d'abord `T[1]` dans `T[:1]` de sorte que `T[:2]` soit trié (en fait on échange `T[0]` et `T[1]` si `T[0]>T[1]`).

- À la deuxième étape, on insère `T[2]` dans `T[:2]` de sorte que `T[:3]` soit trié.
- À la k-ème étape ($k < n$), `T[:k]` est déjà trié. On insère `T[k]` dans `T[:k]` de sorte que `T[:k+1]` soit trié.

Le principe pour insérer `T[k]` dans `T[:k]` est, après avoir enregistré la valeur de `T[k]` dans une variable, de décaler les éléments de `T[:k]` vers la droite tant qu'ils sont strictement supérieurs à `T[k]` (on peut aussi s'arrêter car on est arrivé au bout du tableau). Puis on place `T[k]` à la droite du premier élément inférieur ou égal à `T[k]` que l'on a détecté (ou en position 0 si on n'en a détecté aucun).
Prenons un exemple pour fixer les idées. Partant de la liste `[4,1,5,0,2,3]`, on obtient après 3 étapes la liste `[0,1,4,5,2,3]`. On doit alors insérer `2` dans la partie de la liste située à sa gauche. Pour cela, on va décaler `5` puis `4` d'un cran vers la droite ; on tombe alors sur le nombre `1` (inférieur à `2`) ce qui nous indique que l'on doit arrêter et placer `2` en position `2`.
Le but de l'exercice est de programmer un tri insertion en modifiant la recherche de l'endroit où l'on doit insérer `T[k]` dans `T[:k]` : cette recherche ne se fera plus en descendant `T[:k]` mais par dichotomie.

1. Compléter la fonction suivante afin qu'elle renvoie, étant donné `k` , l'indice de `T` auquel on doit insérer `T[k]` dans `T[:k]` supposé trié. La recherche de cet indice se fera par dichotomie. La fonction ne doit pas modifier le tableau.

```
def indice(T,k):#renvoie l'indice auquel il faut ins\'erer T[k]
                #dans T[:k] suppos\'e tri\'e
    if T[0] >= T[k]:
        return ...
    elif T[k-1] <= T[k]:
        return ...
    else :
        a=0;b=k-1  # on maintient T[k]>T[a] et T[k]<T[b]
        while ....
            c=(b+a)//2
            if T[c] == T[k]:
                ....
            elif ....
                ....
            elif ...
                ....
        return ....
```

2. Quelle est la complexité (en fonction de k) de l'algorithme de la fonction précédente.
3. Écrire une fonction qui trie le tableau `T` par insertion en utilisant la fonction `indice(T,k)` pour déterminer à chaque étape la place à laquelle il faut insérer `T[k]` (on ne doit bien sûr utiliser aucune fonction d'insertion prédéfinie).
4. Donner un invariant de la boucle externe de cet algorithme.
5. Montrer que si l'on ne compte que les comparaisons la complexité de ce tri est en $\mathcal{O}(n \log n)$.
6. Quelle est sa complexité si l'on tient compte de toutes les opérations élémentaires (y compris les affectations) ?

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 de `l[1:]` 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.

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.
   Durant cette étape :

   - On aura intérêt à laisser provisoirement `p` à gauche de `T[g:d]` ;
   - Avec les notations du fichier, on fera en sorte que `T[g+1:indp+1]` soit constitué d'éléments inférieurs (ou égaux) à `p`, `T[indp+1:i]` soit constitué d'éléments strictement supérieurs à `p` et `T[i:d]` reste à traiter.

   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)` .

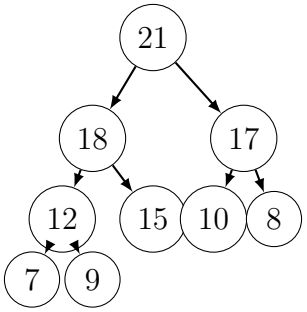
4.4 Complexité

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.

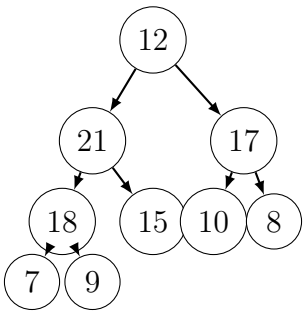
1. Écrire une fonction `parent(i)` au nœud numéro i du tas le numéro de son parent.
2. Écrire des fonctions `filsG(i)` et `filsD(i)` qui associe au nœud numéro i du tas le numéro de son fils gauche et de son fils droit. On pourra tolérer que les fonctions `fils` renvoient un indice qui n'est pas dans 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 $\mathcal{O}(n\log(n))$.