TP1

Introduction

Le but de ce TP est de mettre est de revoir la manipulation des listes, chaînes de caractères. On mettra s'intéressera également aux calculs de complexité.

1 Premier exemple : nombre maximal 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. En déduire une fonction `nombreZerosMax(t)`, utilisant la fonction de la questio précédente, renvoyant le nombre maximal de zéros contigus de la liste `t`.
   Tester avec `t1`.
3. Quelle est, en fonction de n, la complexité de la fonction `nombreZerosMax(t)` ?
4. 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
```

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

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,...,...)
```

   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 : Problème

Un capteur mesure pendant plusieurs années le rayonnement gamma émis par un lointain pulsar. Pour chaque gamma reçu, repéré par son rang i, $0 \leq i < n$ ($n > 0$), on mesure son énergie a_i et l'instant de sa détection t_i. L'unité d'énergie est le kilo électron-Volt (keV) ; l'unité de temps est le dixième de seconde ; l'origine des temps est le début de la campagne de mesure. Pour tout i, la quantité a_i est un entier naturel. Les valeurs a_i sont rangées dans un tableau `a` de longueur n. Ces mesures n'ont pas lieu à intervalles réguliers. On note les dates t_i dans un tableau `t` de n éléments de type entier. Pour tout i et j tels que $0 \leq i < j < n$, on a donc $t_i < t_j$.

Question 5.1 – Écrire une fonction `compte(x,a)` qui retourne, en temps linéaire en n, le nombre de fois où `x` apparaît dans le tableau `a`.

Question 5.2 – En déduire une fonction `occurence(a)` , qui retourne, en temps quadratique en n, un tableau `r` tel que pour tout i, $0 \leq i < n$, `r[i]` soit le nombre de fois où `a[i]` apparaît dans `a`.

Partie I

On cherche à calculer les périodes T pendant lesquelles le rayonnement reste d'énergie constante.

$$T = t_j - t_i \qquad \text{avec} \qquad a_i = a_k = a_j \qquad \text{pour tout } k \text{ tel que } i \leq k \leq j \ .$$

Question 5.3 – Écrire une fonction `maxconstant(a,t)` qui retourne, en temps linéaire en n, la période la plus grande T pendant laquelle le rayonnement reste constant.

Partie II

On notera `occ` le tableau calculé à la question 2.

Question 5.4 – Écrire une fonction `maxoccurence(a,occ)` qui retourne sous forme d'une liste, en temps linéaire en n, les indices i_1 et i_2 des deux rayonnements m_1 et m_2 qui apparaissent le plus grand nombre de fois dans le tableau des mesures `a`. Si le tableau `a` ne contient que des valeurs identiques, on posera $i_2 = m_2 = -1$.

On veut maintenant réorganiser les tableaux de mesures `a` et de date `t` pour mettre en tête toutes les mesures valant m_1, puis celles valant m_2, puis toutes les autres (on supposera que `a`

contient au moins deux valeurs différentes).
La réorganisation des tableaux demandée est donc telle que :

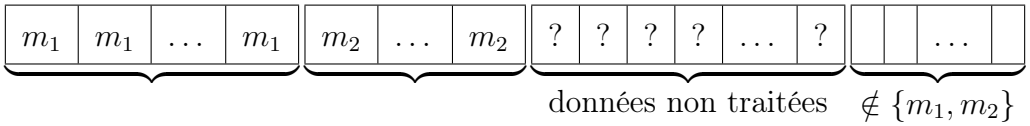
$$0 \leq i < b \quad \Rightarrow \quad a_i = m_1$$

$$b \leq i < r \quad \Rightarrow \quad a_i = m_2$$

$$r \leq i < n \quad \Rightarrow \quad a_i \notin \{m_1, m_2\}$$

Après réorganisation, le tableau `t` vérifie toujours que t_i est la date à laquelle s'est produit le rayonnement a_i.

Question 5.5 – Écrire une fonction `trier(a,t,m1,m2)` qui réordonne, en temps linéaire en n, les tableaux `a` et `t` pour regrouper en tête les deux mesures les plus fréquents, comme indiqué précédemment.
Indication : on parcourra les tableaux `a` et `t` en maintenant dans `a` une décomposition en quatre parties de la forme suivante :



Au début, la troisième zone occupe tout le tableau ; à la fin la troisième zone est vide.

Question 5.6 – La fonction précédente conserve-t-elle la croissance des dates à l'intérieur de chaque zone (c'est à dire $i < j$ implique $t_i < t_j$ pour i et j dans la même zone) ? Justifier.

On pourra tester les fonctions par exemple avec :

```
a=[1,6,8,8,8,5,9,5,5,3,3,6,1,12,13,2,10,8,9,10]
n=len(a)
t=[i for i in range(n)]
```