TP Informatique n° 3
Tris, récursivité, complexité

1 Chaînes de caractères

Les chaînes de caractères (type `string`) sont des listes de caractères (lettres de l'alphabet, chiffres, symboles). On les note entre guillements ou apostrophes :

```
>>> ch1='blablabla'
>>> ch2="toto"
```

On accède à chacun des caractères comme pour une liste : `ch1[0]` renvoie `'b'`, et on dipose des fonctions de concaténation (`ch1+ch2`), de longueur (`len`) et d'extraction de sous-chaîne (`ch1[3:6]`).

■ **Exercice 1** Ecrire une fonction `recherche_mot(m,t)` qui recherche si le mot m est présent dans le texte t, et qui renvoie la position de la première occurence du mot s'il est présent, et None sinon. ■

2 Récursivité

Une fonction est dite récursive lorsqu'elle s'appelle elle-même. En mathématiques, on rencontre cette notion lors de la définition d'une suite récurrente, ou de $n!$ par exemple.

Listing 1 – Fonction factorielle récursive

```
def factrec(n):
    assert(n>=0 and type(n)==int)
    if n == 0:
        return 1
    else:
        return n*factrec(n-1)
```

⚠ On ne peut faire appel à la fonction qu'avec des valeurs du paramètre **strictement inférieures** à la valeur du paramètre en entrée de la fonction ! Sinon, la fonction risque de ne pas terminer ...De même, on examinera soigneusement les cas de base (comme pour une récurrence mathématique).

■ **Exercice 2** Ecrire une fonction récursive `u(n)` qui calcule le n ème terme de la suite définie par $u_0 = 1, u_1 = 0$ et pour tout $n \in \mathbb{N}$ $u_{n+2} = u_{n+1} + 2u_n$. ■

■ **Exercice 3** Le jeu des tours de Hanoï est constitué de trois piquets et de sept disques de tailles différentes, et consiste à déplacer les disques d'une tour de départ à une tour d'arrivée en passant par une tour intermédiaire, avec les règles suivantes :

– On ne peut déplacer qu'un disque à la fois
– On ne peut placer un disque que sur un emplacement vide ou sur un disque plus grand
– C'est le cas dans la configuration initiale

Pour résoudre le problème, on procède de la façon suivante :

1. On a n disques sur le piquet (la tour) de départ
2. On déplace les $n-1$ premiers disques sur la tour intermédiaire
3. On déplace le dernier disque sur la tour d'arrivée
4. On déplace les disques de la tour intermédiaire vers la tour d'arrivée

Ecrire une fonction `hanoi(n,D,I,A)` qui déplace n disques depuis le piquet D jusqu'au piquet A en utilisant le piquet I pour intermédiaire. Exemple avec $n = 2$:

```
>>> hanoi(2,"gauche","milieu","droite")
gauche-->milieu
gauche-->droite
milieu-->droite
```

■

3 Complexité et preuve de programmes

On cherche ici à faire l'étude théorique d'un algorithme, c'est à dire de démontrer (à l'aide d'un raisonnement par récurrence) qu'il donne le bon résultat, et d'évaluer le nombre d'opérations nécéssaires pour y arriver.

3.1 Terminaison et correction d'une fonction

On dit qu'un algorithme termine lorsqu'il conduit à un résultat en un nombre fini d'opérations (ce qui n'est pas forcément garanti lorsqu'on utilise une boucle while ou une fonction récursive) et qu'il est correct lorsqu'il renvoie la valeur attendue.

■ **Exercice 4** Montrer par récurrence que `factrec(n)` termine et renvoie la valeur $n!$. ■

■ **Exercice 5** Montrer que l'algorithme de la division euclidienne termine et est correct.

Listing 2 – Division euclidienne pour les entiers naturels

```
a=int(input("entrer a: "))
b=int(input("entrer b: "))
q=0
r=a
```

```
while r>=b:
    q=q+1
    r=r-b
print("q=",q)
print("r=",r)
```

■

3.2 Complexité

Un problème peut avoir plusieurs solutions : en informatique, plusieurs algorithmes peuvent conduire au même résultat. On les différentie souvent en considérant leur complexité :

1. en terme d'opérations élémentaires (additions, multiplications, comparaisons, échanges dans un tableau) : c'est la complexité temporelle (liée au temps d'exécution de l'algorithme) ;
2. en terme d'occupation en mémoire (pour trier un tableau de n nombres, a-t-on besoin de recopier ce tableau ou peut-on le trier en place ?) : c'est la complexité spatiale.

■ **Exemple 1** Pour calculer 2^n de façon naïve, on a besoin de $n-1$ multiplications. ■

Listing 3 – Exponentiation rapide

```
def expo_rapide(n):
    assert (n >= 0 and type(n)==int)
    if n==0:
        return 1
    else:
        r=expo_rapide(n//2)
        if n%2 == 0:
            return r*r
        else:
            return 2*r*r
```

Expliquer l'algorithme d'exponentiation rapide et calculer sa complexité.

■ **Exercice 6** En remarquant que le polynôme $P(X) = \sum_{k=0}^{n} a_k X^k$ s'écrit aussi

$$P(X) = (\dots(a_n X + a_{n-1})X + a_{n-2})X + \dots)X + a_0$$

ecrire une fonction `horner(P,x)` prenant en paramètre la liste P des coefficients du polynôme et un réel x et qui calcule $P(x)$ (Algorithme de Horner). Calculer le nombre de multiplications nécessaires pour obtenir le résultat, et comparer avec l'algorithme naif. ■

4 Tris

L'étude des tris est une partie incontournable de l'algorithmique, car elle permet de mettre en oeuvre différentes stratégies (récursivité, diviser pour régner), elle amène à devoir choisir entre plusieurs structures de données (tableaux ou listes, mais aussi arbres), et amène à devoir étudier la complexité temporelle et spatiale de ces algorithmes.

4.1 Tri par insertion

Considérons le tri par insertion (similaire au classement d'un jeu de cartes) :

Listing 4 – Tri par insertion

```
def tri_insertion(L):
    n=len(L)
    for i in range(n):
        x=L[i]
        j=i
        while (j>0 and L[j-1]>x):
            L[j] = L[j-1]
            j=j-1
        L[j]=x
    return None
```

Trier une liste de cinq nombres à la main à l'aide de cet algorithme, justifier sa terminaison et sa correction, et calculer sa complexité.

4.2 Tri à bulles

Que fait la fonction suivante ? Justifier le nom de « tri à bulles » !

Listing 5 – Tri à bulles

```
def tri_bulles(L):
    n = len(L)
    echange_ok = False
    while echange_ok == False:
        echange_ok = True
        for j in range(0, n-1):
            if L[j] > L[j+1]:
                echange_ok = False
                L[j],L[j+1] = L[j+1],L[j]
        n = n-1
    return None
```

4.3 Tri rapide

Le tri rapide est un algorithme récursif basé sur le principe « diviser pour régner ». Etant donnée une liste L, on commence par choisir le premier élément comme *pivot*, et on sépare la liste entre deux sous-listes : la première contient des éléments inférieurs ou égaux au pivot, et la seconde contient des éléments supérieurs (strictement) au pivot. Puis on applique le tri rapide aux deux sous-listes obtenues.

Ecrire une fonction `echange(L,i,j)` qui échange les éléments d'indices i et j de la liste L et une fonction `partition(L,gauche,droite)` qui considère la liste `L[gauche:droite+1]` et qui effectue la partition. Cette dernière fonction renverra la position du pivot dans la liste partitionnée.

Exemple 2

```
>>> L=[3,1,4,2,5,3,8]
>>> partition(L,0,6)
3
>>> L
[3, 1, 2, 3, 5, 4, 8]
```

Ecrire alors une fonction récursive `quicksort(L,gauche,droite)`.

Remarque 1 La complexité du tri rapide est en moyenne (et à une constante près) de l'ordre de $n\log(n)$ comparaisons/ affectations, et de n^2 comparaisons/affectations dans le pire cas. On notera qu'il est possible de démontrer que l'ordre de grandeur optimal d'un tri est de $n\log(n)$ comparaisons.

4.4 Tri fusion

Le tri fusion est un tri également basé sur la méthode « diviser pour régner ». La différence avec le tri à bulles est qu'il n'est pas en place, mais les deux sous-listes à trier à chaque appel récursif sont sensiblement de même taille. C'est un tri dit optimal en terme de comparaisons, car il nécéssite de l'ordre de $n\log(n)$ comparaisons/affectations (à une constante près) pour trier un tableau de n nombres.

■ **Exercice 7** Analyser l'algorithme du tri fusion présenté ci-dessous et le faire fonctionner à la main sur la liste `[7,6,3,5,4,2,1,8]`.

Listing 6 – Tri à bulles

```python
def fusion(L1,L2,g,m,d):
    """ fusionne les listes L1[g:m] et L1[m,d] dans la liste L2"""
    i,j =g,m #i parcourt  L1[g:m] et j parcourt L1[m,d]
    for k in range(g,d):
        if i<m and (j==d or L1[i] <=L1[j]):
            L2[k] = L1[i] # on ajoute un élément de L1[g:m] à L2
            i = i+1
        else:
            L2[k] = L1[j] # on ajoute un élément de L1[m:d] à L2
            j = j+1
    return None

def tri_fusion(L):
    tmp = L[:] # on recopie la liste pour la modifier
    def tri_rec(g,d): # fonction recursive locale
        if g >= d-1: return
        m=(g+d)//2 # on determine le milieu
        tri_rec(g,m) #on trie recursivement les deux sous-tableaux
        tri_rec(m,d)
        tmp[g:d] = L[g:d]
        fusion(tmp,L,g,m,d) # on effectue la fusion
    tri_rec(0,len(L))
```

■

4.5 Recherche dichotomique

■ **Exercice 8** Etant donnée une liste L **triée en ordre croissant**, écrire une fonction `recherche_dicho(x,L)` qui renvoie, si elle existe, la position d'une occurence de x dans la liste L, et None sinon. On pourra s'appuyer sur la méthode « diviser pour régner »en coupant le tableau en deux par le milieu, et en déterminant si x doit être cherché dans la partie gauche ou droite. ■