TP 11 : Complexité

Exercice 1. Écrire une fonction `liste_pos(x,L)` qui renvoie la liste (éventuellement vide) des positions où `x` apparait dans `L`. Par exemple `liste_pos(3,[1,3,5,2,3])` doit renvoyer `[1,4]`. Donner un ordre de grandeur du temps d'exécution de cette fonction en fonction de n, nombre d'éléments de la liste `L`.

Exercice 2 *(Sujet d'oral fourni par la banque PT).* À chaque question, les instructions ou les fonctions écrites devront être testées. Soit un entier naturel n non nul et une liste `t` de longueur n dont les termes valent 0 ou 1. Le but de cet exercice est de trouver le nombre maximal de 0 contigus dans `t` (c'est-à-dire figurant dans des cases consécutives). Par exemple, le nombre maximal de zéros contigus de la liste `t1` suivante vaut 4 :

`i`	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
`t1[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 paramètres une liste `t`, de longueur n, et un indice i compris entre 0 et $n-1$, et renvoyant :

$$\begin{cases} 0 & \text{si } \texttt{t}[i]=1 \\ \text{le nombre de zéros consécutifs dans } \texttt{t} \text{ à partir de } \texttt{t}[i] \text{ inclus, si } \texttt{t}[i]=0 \end{cases}$$

Par exemple, les appels

`nombreZeros(t1,4)`, `nombreZeros(t1,1)`, `nombreZeros(t1,8)`

renvoient respectivement les valeurs 3, 0 et 1.

(2) Comment obtenir le nombre maximal de zéros contigus d'une liste `t` connaissant la liste des `nombreZeros(t,i)` pour $0 \leqslant i \leqslant n-1$? Utiliser ceci pour rédiger la fonction `nombreZerosMax(t)`, de paramètre `t`, renvoyant le nombre maximal de 0 contigus d'une liste `t` non vide. On utilisera la fonction `nombreZeros`.

(3) Quelle est la complexité de la fonction `nombreZerosMax` construite à la question précédente ?

(4) Trouver un moyen simple, toujours en utilisant la fonction `nombreZeros`, d'obtenir un algorithme plus performant.

Exercice 3 *Suite de Syracuse.* On rappelle la définition de la suite de Syracuse : u_0 est un entier positif et

$$\begin{aligned} \forall n \in \mathbb{N},\ u_{n+1} &= \frac{u_n}{2} \text{ si } u_n \text{ est pair} \\ &= 3u_n + 1 \text{ si } u_n \text{ est impair} \end{aligned}$$

(a) Lorsque $u_0 = 5$, calculer u_1, u_2, u_3, u_4.

(b) Écrire une fonction `syracuse(u0,n)` qui construit la liste $[u_0, u_1, \ldots, u_n]$ correspondant à cette suite.

(c) Tester cette fonction sur des exemples pertinents.

(d) Déterminer un ordre de grandeur du temps d'exécution de cette fonction, en fonction de n.

Exercice 4 *Recherche dichotomique (recherche dans une liste triée).* On a vu en classe que la recherche d'un élément dans une liste L de taille n a un temps d'exécution en $O(n)$. On suppose dans cet exercice que la liste L est triée et on va utiliser cette hypothèse pour rendre l'algorithme de recherche plus efficace.

(a) On considère une liste L de nombres triée en ordre croissant ainsi qu'un nombre x. Le but de l'algorithme de recherche dichotomique est de déterminer et renvoyer un entier i tel que $L[i] = x$. Lorsqu'un tel entier i n'existe pas, on conviendra de renvoyer la valeur -1. Le principe de l'algorithme est le suivant :
- Utiliser deux variables g et d qui délimitent la partie de la liste dans laquelle on effectue la recherche ;
- Prendre l'indice i situé au milieu de la partie considérée et comparer $L[i]$ à x, ceci permet de décider dans quelle moitié de la liste poursuivre la recherche ;
- Utiliser une boucle *while* et réfléchir soigneusement à la condition d'arrêt.

Écrire précisément l'algorithme de recherche dichotomique.

(b) Programmer cet algorithme sous la forme d'une fonction **`rech_dicho(x,L)`** et la tester sur plusieurs exemples. Ne pas oublier de faire le test sur des exemples très simples tels que :

```
rech_dicho(2,[]), rech_dicho(2,[2]), rech_dicho(2,[3])
```

mais on ne se limitera pas non plus à ces exemples.

(c) Pour étudier la complexité, on supposera pour simplifier que la liste L est de taille $n = 2^k$ avec un entier k. Démontrer que le corps de la boucle *while* est répété au plus k fois. En déduire un ordre de grandeur du temps d'exécution de la fonction de recherche en fonction de k puis en fonction de n.

Exercice 5 *(Sujet d'oral fourni par la banque PT).* À chaque question, les instructions ou les fonctions écrites devront être testées.

(1) Écrire une fonction **`somcube`**, d'argument n, renvoyant la somme des cubes des chiffres du nombre entier naturel n. Donner un ordre de grandeur du temps d'exécution de cette fonction en fonction de n.

(2) Trouver tous les nombres entiers naturels inférieurs à 1000 égaux à la somme des cubes de leurs chiffres.

(3) Écrire une fonction **`egaux_somcubes`**, d'argument N, renvoyant la liste des nombres entiers naturels inférieurs à N qui sont égaux à la somme des cubes de leurs chiffres. Donner un ordre de grandeur de son temps d'exécution en fonction de N.

Corrections

Ex 1. On a vu en classe la fonction suivant qui teste si **x** apparait dans **L** :

```
def appartient(x,L):
    n = len(L)
    trouvé = False
    for i in range(0,n):
        if L[i]==x:
            trouvé = True
    return trouvé
```

On modifie cette fonction pour qu'elle enregistre dans une liste **R** les indices **i** pour lesquels **L[i]==x**.

```
def liste_pos(x,L):
    n = len(L)
    R = []
    for i in range(0,n):
        if L[i]==x:
            R.append(i)
    return R

print(liste_pos(3,[1,3,5,2,3]))
```

```
[1, 4]
```

Le temps d'exécution est $O(n)$.

Ex 2.

```
t1 = [0,1,1,1,0,0,0,1,0,1,1,0,0,0,0]

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

print(nombreZeros(t1,4))
print(nombreZeros(t1,1))
print(nombreZeros(t1,8))
print(nombreZeros(t1,12))
```

```
3 0 1 3
```

```
def nombreZerosMax(t):
    n = len(t)
    m = 0
    for i in range(n):
        nz = nombreZeros(t,i)
        if nz>m:
            m = nz
    return m

print(nombreZerosMax(t1))
```

```
4
```

La complexité est en $O(n^2)$.

```
def nombreZerosMax(t):
    n = len(t)
    m = 0
    i = 0
    while i<n:
        nz = nombreZeros(t,i)
        if nz>m:
            m = nz
        i = i+nz+1
    return m

print(nombreZerosMax(t1))
```

```
4
```

La complexité est cette fois en $O(n)$. Autre méthode.

```
def nombreZerosMax(t):
    n = len(t)
    m = 0
    nz = 0
    for i in range(n):
        if t[i]==0:
            nz = nz+1
        else:
            nz = 0
        if nz>m:
            m = nz
    return nz

print(nombreZerosMax(t1))
```

```
4
```

Ex 3. Pour $u0 = 5$, on trouve $u_1 = 16$ puis $u_2 = 8$, $u_3 = 4$, $u_4 = 2$, $u_5 = 1$, $u_6 = 4$, etc.

```
def syracuse(u0,n):
    """
    Construit la liste [u0,u1,...,un]
    constituée des (n+1) premiers termes de la suite de Syracuse
    partant de u0
    """
    u=u0
    L=[u0]
    for k in range(n):
        if u%2==0:
            u=u/2
        else:
            u=3*u+1
        L.append(u)
    return L
print(syracuse(5,6))
```

```
[5, 16, 8.0, 4.0, 2.0, 1.0, 4.0]
```

Le temps d'exécution est un $O(n)$. Remarque : si on définit une fonction qui calcule u_n puis on l'utilise ensuite pour construire la liste, on obtient un temps d'exéction en $O(n^2)$.

```
def rech_dicho(x,L):
    n = len(L)
    g = 0
    d = n-1
    while d>g:
        m = (d+g)//2
        if L[m]==x:
            g = m
            d = m
        elif L[m]<x:
            g = m+1
        else:
            d = m-1
    if d<g:
        return -1
    elif L[g]==x:
        return g
    else:
        return -1

print(rech_dicho(2,[]))
print(rech_dicho(2,[2]))
print(rech_dicho(2,[3]))
print(rech_dicho(2,[2,3,4]))
print(rech_dicho(2,[1,2,3]))
print(rech_dicho(2,[0,1,2]))
```

```
-1
0
```

```
-1
0
1
2
```

Ex 5. Les chiffres d'un entier n s'obtiennent comme les restes de divisions euclidiennes successives par 10. On utilise une variable s initialisée à 0 à laquelle on ajoute successivement les cubes des chiffres de n obtenus ainsi.

```
def somcubes(n):
    s = 0
    while n>0:
        s = s+(n%10)**3
        n = n//10
    return s

print(somcubes(12))
```

```
9
```

On affiche alors les nombres entiers n compris entre 0 et 1000 pour lesquels **n==somcubes(n)** :

```
for n in range(0,1001):
    # 0<=n<=1000
    if n==somcubes(n):
        print(n)
```

```
0 1 153 370 371 407
```

Sur le même principe :

```
def egaux_somcubes(N):
    L = []
    for n in range(0,N+1):
        # 0<=n<=N
        if n==somcubes(n):
            L.append(n)
    return L

print(egaux_somcubes(2000))
```

```
[0, 1, 153, 370, 371, 407]
```

Le temps d'exécution de **somcubes(n)** est $O(p)$ où p est le nombre de chiffres de n dans l'écriture en base 10. Or si n s'écrit avec p chiffres en base 10, on a $10^{p-1} \leqslant n \leqslant 10^p$ donc $(p-1)\ln(10) \leqslant \ln(n) \leqslant p \leqslant 10$ et ainsi le temps d'exécution en fonction de n est $O(\ln(n))$. L'appel de **somcubes(n)** à l'intérieur de **egaux_somcubes** a un temps d'exécution de l'ordre de $O(\ln(n))$ et c'est également $O(\ln(N))$ car on réalise cet appel pour tout n tel que $0 \leqslant n \leqslant N$. Le temps d'exécution de **somcubes(N)** est donc de l'ordre de $O(N\ln(N))$.