TD3 : Diviser pour mieux régner

La méthode "diviser pour mieux régner" consiste à diviser une tâche longue à réaliser en plusieurs (souvent deux) tâches plus courtes dont le traitement permettra de résoudre le problème de départ.

Préliminaire : écriture d'un nombre dans une base arbitraire

La façon usuelle d'écrire un nombre $n \in \mathbb{N}^*$ (avec des chifres entre 0 et 9) consiste à trouver des chiffres c_0, c_1, c_2, …, c_k tels que $\sum_{i=0}^{k} c_i 10^i = n$, avec $c_k \neq 0$.

De même, "écrire un nombre n en base b" consiste à trouver c_0, c_1, …, c_k tels que

$$\begin{cases} \forall i \in \{0,1,\ldots,k\}, c_i \in \{0,1,\ldots,b-1\} \\ c_k \neq 0 \\ \sum_{i=0}^{k} c_i b^i = n \end{cases} \quad (1)$$

Démontrer que pour tout $n \in \mathbb{N}^*$ et tout $b \in \mathbb{N} \setminus \{0,1\}$, il existe un unique multiplet c_0, c_1, …, c_k satisfaisant les conditions (1), et que c'est précisément le n-uplet renvoyé par le programme ci-dessous :

```
def en_base(n,b):
    q,r=(n//b),(n%b) # quotient et reste de la division
    if q==0: return tuple([r])#                euclidienne de n par b
    return tuple([r])+en_base(q,b)#"tuple(" convertit en n-uplet
```

Dans le cas particulier où $b = 2$, on parle d'"écriture en binaire"

Premier exemple de "division pour mieux reigner"

```
def p(x,n):
    if n==0:
        return 1
    else:
        return x*p(x,n-1)
```

*** Que calcule `p(x,n)`, lorsque `n`$\in\mathbb{N}$ et `x`$\in\mathbb{R}$?**
*** Combien de multiplications sont effectuées lors du calcul de `p(x,n)` ?**

La méthode "diviser pour mieux régner" permet de calculer la même quantité :
Si n est pair, avec n=2k, alors $p(x,n)$ se calcule aussi comme étant $p(x,k) \times p(x,k)$
De même si n est impaire est vaut 2k+1, alors p(x,n) se calcule aussi comme étant $x \times p(x,k) \times p(x,k)$
Le programme ci dessous définit donc une fonction qui calcule la même chose :

```
def p2(x,n):
    if n==0: return 1
    r,k=(n%2),(n//2)
    a=p2(x,k)
    if r==0: return a*a
    else: return x*a*a
```

*** Si le nombre (entier) `n` compte m chiffres quand on l'écrit en binaire, alors combien de multiplications au maximum sont effectuées lors du calcul de `p2(x,n)` ?**
*** Argumenter que les divisions effectuées ne vont pas impacter significativement le temps de calcul avec la fonction `p2`.**
*** Lorsque `n` est très grand, laquelle des deux méthodes est la plus rapide ?**

Tri d'une liste

Un autre exemple consiste à trier des listes, c'est à dire à en trier les éléments par ordre croissant.

Une façon simple de procéder consiste à chercher le minimum de la liste, le mettre en premier, puis à trier le reste de la liste.

Le programme ci-dessous met en oeuvre cette méthode, à condition que vous définissiez une fonction `trouve_min` qui trouve la position de la plus petite valeur d'une liste

```
def tri_basique(l):
    if len(l)<=1:
        return l
    else:
        k=trouve_min(l)
        return [l[k]]+tri_basique(l[:k]+l[k+1:])
```

*** Définissez une fonction `trouve_min` qui parcourt la liste afin de trouver sa plus petite valeur, et renvoi la position de la plus petite valeur.**

Par exemple `trouve_min([4,8,3,6,2,7])` doit renvoyer `4` car le plus petit élément de la liste est le cinquième (et en python, le cinquième élément porte le numéro `4`, c'est à dire que c'est `l[4]`)

*** Si la liste `l` contient n éléments, alors quand on calcule `trouve_min(l)`, combien de fois l'ordinateur compare-t-il des valeurs de la liste pour trouver le minimum ?**

*** En conséquence, quand on calcule `tri_basique(l)`, combien de fois l'ordinateur compare-t-il des valeurs de la liste au cours du calcul ?**

Tri par fusion

On peut utiliser la méthode de "diviser pour mieux régner" afin de trier plus efficacement des listes :

```
def tri_fusion(l):
    n=len(l)
    if len(l)<=1: return l
    k=n//2
    return f(tri_fusion(l[:k]),tri_fusion(l[k:]))
```

où f est la fonction définie par

```
def f(l1,l2):
    if l1==[]: return l2
    elif l2==[]: return l1
    elif l1[0]>l2[0]: return [l2[0]]+f(l1,l2[1:])
    else: return [l1[0]]+f(l2,l1[1:])
```

*** Que fait la fonction `f` ?**

*** On note F_m le nombre de comparaisons effectuées par `tri_fusion` lorsque la liste `l` contient 2^m éléments.**

Déterminer une relation de récurrence satisfaite par la suite F_m. Déduire que $\forall m \in \mathbb{N}$, $F_m = 2^m(m-1)+1$.

*** En supposant que le nombre de comparaisons effectuées soit représentatif du temps de calcul nécessaire, laquelle des deux façons de trier une liste est plus rapide lorsque que la liste contient beaucoup d'éléments ?**

Recherche d'un élément dans une liste

On cherche désormais à déterminer si un élément est présent ou pas dans une liste.

```
def appartient(element,liste):
    if liste==[]:
        return False
    if liste[0]==element:
        return True
    else:
        .........
```

Compléter la définition ci-dessus de la fonction "`appartient`" qui doit renvoyer "`True`" si "`element`" est présent dans "`liste`", et "`False`" s'il n'y est pas présent.

Recherche par dichotomie

Quand on cherche à déterminer si une liste contient un élément précis, on peut aller plus vite si cette liste est déjà triée.

Par exemple, quand on cherche un mot dans le dictionnaire, on gagne un temps considérable grâce au fait que le dictionnaire soit trié (dans l'ordre alphabétique en l'occurrence).

On suppose désormais que l'on cherche un élément parmi une liste qui a préalablement été triée.

En adoptant le principe de "diviser pour mieux régner", écrire une fonction plus rapide que la précédente pour déterminer si un élément appartient à une liste.

```
def appartient2(element,liste_triee):
    .......
    .......
```

Comparer les temps de calcul des deux méthodes.