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.

I. Écriture d'un nombre dans une base arbitraire

La façon usuelle d'écrire un nombre $n \in \mathbb{N}^*$ (avec des chiffres 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

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

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

La fonction suivante détermine le n-uplet $(c_0, c_1, \ldots n)$ associé au nombre n pour la base b :

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

Question Exprimer (en fonction de n et de b) le nombre k de chiffres utilisés pour écrire le nombre n en base b.

II. Exemple de "division pour mieux régner" : Calcul de la puissance d'un nombre

1. Algorithme naïf

La fonction ci-dessous calcule `x`$^{\texttt{n}}$, lorsque `n`$\in\mathbb{N}$ et `x`$\in\mathbb{R}$.

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

Question Combien cette fonction fait elle de multiplications pour calculer `x`$^{\texttt{n}}$?

2. Algorithme "d'exponentiation rapide"

La méthode "diviser pour mieux régner" permet de calculer la même quantité en utilisant le fait que, si $n = 2k$ est pair, alors $x^n = x^k \times x^k$, tandis que si $n = 2k+1$ est impaire, alors on a $x^n = x \times x^k \times x^k$.

Le programme ci dessous calcule donc les puissances de n en utilisant cette observation :

```
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
```

Questions

1. 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)` ?
2. Argumenter que les divisions effectuées ne vont pas impacter significativement le temps de calcul avec la fonction `p2`.
3. Lorsque `n` est très grand, laquelle des deux méthodes est la plus rapide ?

III. Exercices

1. Unicité de l'écriture d'un nombre dans une base arbitraire

Démontrer que pour tout $n \in \mathbb{N}^*$ et tout $b \in \mathbb{N} \setminus \{0,1\}$, il existe un unique multiplet $(c_0, c_1, \ldots, c_k)$ satisfaisant les conditions de la partie I., et que c'est précisément le n-uplet renvoyé par la fonction `en_base`.

2. Tri d'une liste

2.1) Algorithme naïf

On cherche à trier (par ordre croissant) les éléments d'une liste.

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:])
```

1. Définissez une fonction `trouve_min` qui parcourt la liste afin de trouver sa plus petite valeur, et renvoie 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]`)
2. 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 ?
3. 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 ?

2.2) 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(l1,l2[1:])
```

1. Déterminer ce que fait la fonction `f`. (On pourra par exemple commencer par l'exécuter avec différents exemples de liste `l1` et `l2` triées, pour constater son effet.)
2. 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 $\forall m{\in}\mathbb{N}$, $F_m = 2^m(m-1)+1$.
3. 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 ?

3. Recherche d'un élément dans une liste

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

3.1) Algorithme naïf pour une liste arbitraire

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.

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

3.2) Recherche par dichotomie dans une liste triée

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.