Chapitre 20 - Diviser pour régner

On voit dans ce chapitre une technique algorithmique qui porte le nom de « ***Diviser pour régner*** ». Elle permet dans certains cas, d'obtenir des algorithmes dont la complexité est bien meilleure à celle des solutions naïves.

Paradigme de cette méthode :

- **Diviser :** découper un problème initial en sous-problème (souvent en deux parties plus ou moins égales)
- **Régner :** résoudre les sous problèmes (généralement récursivement)
- **Fusionner :** calculer une solution au problème initial à partir des solutions des sous-problèmes

On se contente ici d'aborder plusieurs exemples qui vous permettront de mettre en œuvre cette méthode afin de mieux vous l'approprier.

1- Calcul d'une puissance en mathematiques :

Exemples :

Calcul de 3^{14}		Calcul de 3^{200}	
$3^{14} = 3^7 \times 3^7$	1 multiplication	$3^{200} = 3^{100} \times 3^{100}$	1 multiplication
$3^7 = 3 \times 3^3 \times 3^3$	2 multiplications	$3^{100} = 3^{50} \times 3^{50}$	1 multiplication
$3^3 = 3 \times 3 \times 3$	2 multiplications	$3^{50} = 3^{25} \times 3^{25}$	1 multiplication
Au total, pour calculer 3^{14}, on a réalisé 4 multiplications. La méthode naïve $3^{14} = 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3$ nécessite elle 13 multiplications.		$3^{25} = 3 \times 3^{12} \times 3^{12}$	2 multiplications
		$3^{12} = 3^6 \times 3^6$	1 multiplication
		$3^6 = 3^3 \times 3^3$	1 multiplication
		$3^3 = 3 \times 3 \times 3$	2 multiplications
		Au total, pour calculer 3^{200}, on a réalisé 9 multiplications. La méthode naïve $3^{200} = 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times \ldots \ldots$ nécessite elle 199 multiplications.	

Codes :

```
def puissanceDR(a, n):
    if n == 0 :
        return 1
    else :
        facteur = puissanceDR(a, n//2)
        if n % 2 == 0: return facteur * facteur
        elif n % 2 == 1 : return a * facteur * facteur
```

```
def puissanceNaif(a,n) :
    produit = 1
    for i in range(n) :
        produit *= a
    return produit
```

Calcul de $1.0000001^{1000000000}$:

L'opération ** de python comme la fonction puissanceDR, calculent cette puissance de manière instantanée. La méthode naïve prend elle un temps très long.

```
>>> puissanceDR(1.0000001,1000000000)
2.688103817159271e+43

>>> 1.0000001**1000000000
2.6881038582144647e+43
```

Compléments : Fonction qui donne aussi le nombre de multiplications :

```
def puissanceDR(a, n , nb = 0):
    if n == 1 :
        print(f"Nombre de multiplications : {nb}" )
        return a

    if n % 2 == 0:
        nb += 1
        facteur = puissanceDR(a, n//2,nb)
        return facteur * facteur
    else:
        nb += 2
        facteur = puissanceDR(a, (n-1)//2,nb)
        return a * facteur * facteur
```

```
>>> puissanceDR(3,200)
Nombre de multiplications : 9
265613988875874769338781322035779626829233452653394
95974574961739092490901302182994384699044001
```

2- RECHERCHE DU MAXIMUM D'UNE LISTE :

Exemples : Recherche du maximum de la liste $\ell = [23\,,12\,,4\,,56\,,35\,,57\,,3\,,11\,,6\,]$

On utilise le paradigme « diviser pour régner » qui nous conduit à diviser la liste en 2 parties de même taille. On détermine par récurrence le maximum de la partie gauche et celui de la partie droite. On renvoie parmi ces 2 valeurs, la plus grande. Cela donne :

<table>
<tr><td></td><td colspan="8">[23 , 12 , 4 , 56 , 35 , 57 , 3 , 11 , 6]</td></tr>
<tr><td>Diviser</td><td colspan="5">[23 , 12 , 4 , 56 , 35]</td><td colspan="4">[57 , 3 , 11 , 6]</td></tr>
<tr><td>Diviser</td><td colspan="3">[23 , 12 , 4]</td><td colspan="2">[56 , 35]</td><td colspan="2">[57 , 3]</td><td colspan="2">[11 , 6]</td></tr>
<tr><td rowspan="2">Diviser</td><td colspan="2">[23 , 12]</td><td rowspan="2">[4]</td><td rowspan="2">[56]</td><td rowspan="2">[35]</td><td rowspan="2">[57]</td><td rowspan="2">[3]</td><td rowspan="2">[11]</td><td rowspan="2">[6]</td></tr>
<tr><td>[23]</td><td>[12]</td></tr>
<tr><td>Maximum</td><td>23</td><td>12</td><td rowspan="2">4</td><td rowspan="2">56</td><td rowspan="2">35</td><td rowspan="2">57</td><td rowspan="2">3</td><td rowspan="2">11</td><td rowspan="2">6</td></tr>
<tr><td>Maximum</td><td colspan="2">23</td></tr>
<tr><td>Maximum</td><td colspan="3">23</td><td colspan="2">56</td><td colspan="2">57</td><td colspan="2">11</td></tr>
<tr><td>Maximum</td><td colspan="5">56</td><td colspan="4">57</td></tr>
<tr><td>Maximum</td><td colspan="9">57</td></tr>
</table>

Codes : Pour diviser une liste en 2 parties égales, il existe 2 techniques :

Technique propre à python, qui utilise l'opérateur [:] pour segmenter la liste.	Technique générale qui utilise la même liste mais en s'intéressant uniquement aux valeurs comprises entre 2 indices iD et iF (i départ et i fin)

`def maxi(l) :`
`  if len(l) == 1 : return l[0]`
`  else :`
`  iM = len(l)//2`
`  maxG = maxi(l[:iM])`
`  maxD = maxi(l[iM:])`
`  if maxG > maxD : return maxG`
`  else : return maxD`
	`def maxi(l,iD,iF) :`
	`  if iD == iF : return l[iD]`
	`  else :`
	`  iM = (iD + iF) //2`
	`  maxG = maxi(l,iD,iM)`
	`  maxD= maxi(l,iM+1,iF)`
	`  if maxG > maxD : return maxG`
	`  else : return maxD`

Compléments : Fonction qui donne aussi le nombre de comparaisons :

```
def maxi(l,nb=0) :
    if len(l) == 1 : return l[0],nb
    else :
        nb = nb+1
        iM = len(l)//2
        maxG,nb = maxi(l[:iM],nb)
        maxD,nb= maxi(l[iM:],nb)
        if maxG > maxD : return maxG,nb
        else : return maxD,nb
```

Exécution de ce script pour une liste de taille 10 :

```
l = [randint(0,100) for _ in range(10)]
m = maxi(l)
print(m)
```

```
>>> (executing file "maximum.py")
(86, 9)
```

donne 9 comparaisons.

Exécution de ce script pour une liste de taille 1 000 000 :

```
l = [randint(0,100) for _ in range(1000000)]
m = maxi(l)
print(m)
```

```
>>> (executing file "maximum.py")
(100, 999999)
```

donne 999 999 comparaisons.

Conclusion : Le paradigme « Diviser pour régner » n'apporte ici aucun gain en termes d'efficacité, par rapport à une méthode naïve.

3- PRESENCE D'UNE VALEUR PARTICULIERE DANS UNE LISTE NON TRIEE :

Exemples : Recherche de la valeur 5 dans la liste $\ell = [23\,,12\,,4\,,56\,,35\,,57\,,3\,,11\,,6\,]$

La fonction à définir renvoie *True* si 5 se trouve dans ℓ et renvoie *False* sinon. On utilise le paradigme « diviser pour régner » qui nous conduit à diviser la liste en 2 parties de même taille. On détermine par récurrence la présence de la valeur 5 dans la partie gauche et dans la partie droite. On renvoie le résultat « présence gauche » **OR** « présence droite ».

<table>
<tr><td></td><td colspan="8">Présence de la valeur 5 dans la liste [23 , 12 , 4 , 56 , 35 , 57 , 3 , 11 , 6]</td></tr>
<tr><td>Diviser</td><td colspan="4">[23 , 12 , 4 , 56 , 35]</td><td colspan="4">[57 , 3 , 11 , 6]</td></tr>
<tr><td>Diviser</td><td colspan="2">[23 , 12 , 4]</td><td colspan="2">[56 , 35]</td><td colspan="2">[57 , 3]</td><td colspan="2">[11 , 6]</td></tr>
<tr><td rowspan="2">Diviser</td><td colspan="2">[23 , 12]</td><td rowspan="2">[4]</td><td rowspan="2">[56]</td><td rowspan="2">[35]</td><td rowspan="2">[57]</td><td rowspan="2">[3]</td><td rowspan="2">[11]</td><td rowspan="2">[6]</td></tr>
<tr><td>[23]</td><td>[12]</td></tr>
<tr><td>Recherche</td><td>False</td><td>False</td><td rowspan="2">False</td><td rowspan="2">False</td><td rowspan="2">False</td><td rowspan="2">False</td><td rowspan="2">False</td><td rowspan="2">False</td><td rowspan="2">False</td></tr>
<tr><td>Recherche</td><td colspan="2">False</td></tr>
<tr><td>Recherche</td><td colspan="3">False</td><td colspan="2">False</td><td colspan="2">False</td><td colspan="2">False</td></tr>
<tr><td>Recherche</td><td colspan="5">False</td><td colspan="4">False</td></tr>
<tr><td>Recherche</td><td colspan="9">False</td></tr>
</table>

Codes : Pour diviser une liste en 2 parties égales, il existe 2 techniques. On donne ainsi 2 codes différents :

Technique propre à python, qui utilise l'opérateur [:] pour segmenter la liste.

```
def recherche(l,val) :
    if len(l) == 1 :
        if l[0] != val : return False
        else : return True
    else :
        iM = len(l)//2
        if val == l[iM] : return True
        else :
            return recherche(l[iM:],val) or recherche(l[:iM],val)
```

Technique générale qui utilise la même liste mais en s'intéressant uniquement aux valeurs comprises entre 2 indices iD et iF (i départ et i fin)

```
def recherche(l,iD,iF,val) :
    if iD == iF :
        return l[iD] == val
    else :
        iM = (iD + iF) //2
        return recherche(l,iD,iM,val) or  recherche(l,iM+1,iF,val)
```

Efficacité : Comme pour le script de recherche de maximum, le paradigme « Diviser pour régner » n'apporte aucun gain en termes d'efficacité, par rapport à une méthode naïve. Elle nécessite autant de comparaisons que dans une recherche naïve.

4- PRESENCE D'UNE VALEUR PARTICULIERE DANS UNE LISTE TRIEE :

Exemples : Recherche de la valeur 5 dans la liste $\ell = [3\,,4\,,6\,,11\,,12\,,23\,,35\,,56\,,57\,]$

La fonction à définir renvoie *True* si 5 se trouve dans ℓ et renvoie *False* sinon. On utilise le paradigme « diviser pour régner » qui nous conduit à diviser la liste en 2 parties de même taille. Comme la liste est ici triée, on détermine par récurrence, uniquement la présence de la valeur 5 dans la partie gauche ou dans la partie droite. On utilise ainsi le principe de la DICHOTOMIE déjà vu dans un chapitre précédent, mais en version itérative. On voit ici une version récursive.

Codes : Pour diviser une liste en 2 parties égales, il existe 2 techniques. On donne ainsi 2 codes différents :

Technique propre à python, qui utilise l'opérateur [:] pour segmenter la liste.

```
def recherche(l,val) :
    if len(l) == 1 :
        return l[0] == val
    else :
        iM = len(l)//2
        if val > l[iM] :
            return recherche(l[iM:],val)
        elif val < l[iM] :
            return recherche(l[:iM],val)
        elif val == l[iM] :
            return True
```

Technique générale qui utilise la même liste mais en s'intéressant uniquement aux valeurs comprises entre 2 indices iD et iF (i départ et i fin)

```
def recherche(l,iD,iF,val) :
    if iD > iF :return False
    else :
        iM = (iD + iF) //2
        if val < l[iM] :
            return recherche(l,iD,iM-1,val)
        elif val > l[iM] :
            return recherche(l,iM+1,iF,val)
        elif val == l[iM] :
            return True
```

Efficacité : Cette technique de dichotomie est extrêmement efficace. Une recherche naïve a une complexité en $\mathcal{O}(n)$ car elle demande autant de comparaisons que d'éléments présents dans la liste. Avec la dichotomie, on ne retient qu'une des 2 moitiés. Plus on progresse dans cette logique de division, plus les moitiés de listes conservées sont petites. On ne réalise d'autre part qu'une seule comparaison par moitié de liste conservée. La complexité est en $\mathcal{O}(log_2(n))$.

Par exemple, pour une liste de taille $n = 1\,000\,000$, on a $log_2(1\,000\,000) \approx 20$. Ainsi une recherche naïve demande environ 1 000 000 de comparaisons alors qu'avec une dichotomie, on a uniquement 20 comparaisons.

$$\log_2(1000000)$$

$$6 \cdot \log_2(5)+6 \approx 19.93157$$

5- TRI FUSION D'UNE LISTE :

Exemples : Tri de la liste $\ell = [23, 12, 4, 56, 35, 57, 3, 11, 6]$

On utilise le paradigme « diviser pour régner » qui nous conduit à diviser la liste en 2 parties de même taille. Par récurrence, on continue cette logique jusqu'à arriver à des listes qui ne contiennent qu'un seul élément. On reconstitue ensuite les 2 moitiés de liste en classant les valeurs par ordre croissant. On utilise pour cela une fonction dite de fusion. Cette fonction prend en argument 2 listes triées et renvoie la liste comprenant toutes les valeurs de ces 2 listes, rangées par ordre croissant. Par exemple :

```
>>> fusion([1,7,22,88] , [-8,4])
[-8, 1, 4, 7, 22, 88]
```

```
>>> fusion([1,88],[4])
[1, 4, 88]
```

<table>
<tr><td></td><td colspan="8">[23 , 12 , 4 , 56 , 35 , 57 , 3 , 11 , 6]</td></tr>
<tr><td>Diviser</td><td colspan="4">[23 , 12 , 4 , 56 , 35]</td><td colspan="4">[57 , 3 , 11 , 6]</td></tr>
<tr><td>Diviser</td><td colspan="2">[23 , 12 , 4]</td><td colspan="2">[56 , 35]</td><td colspan="2">[57 , 3]</td><td colspan="2">[11 , 6]</td></tr>
<tr><td rowspan="2">Diviser</td><td colspan="2">[23 , 12]</td><td rowspan="2">[4]</td><td rowspan="2">[56]</td><td rowspan="2">[35]</td><td rowspan="2">[57]</td><td rowspan="2">[3]</td><td rowspan="2">[11]</td><td rowspan="2">[6]</td></tr>
<tr><td>[23]</td><td>[12]</td></tr>
<tr><td>Fusion</td><td colspan="2">[12 , 23]</td><td colspan="2">[4 , 12 , 23]</td><td colspan="2">[35 , 56]</td><td colspan="2">[3 , 57]</td><td colspan="2">[6 , 11]</td></tr>
<tr><td>Fusion</td><td colspan="4">[4 , 12 , 23 , 35 , 56]</td><td colspan="4">[3 , 6 , 11 , 57]</td></tr>
<tr><td>Fusion</td><td colspan="8">[3 , 4 , 6 , 11 , 12 , 23 , 35 , 56 , 57]</td></tr>
</table>

Codes : Pour diviser une liste en 2 parties égales, il existe 2 techniques. On donne ainsi 2 codes différents :

Code de la fonction fusion().

```
def fusion(l1, l2) :
    i1 = 0
    i2 = 0
    l = []
    while i1 < len(l1) and i2 < len(l2) :
        if l1[i1] < l2[i2] :
            l.append(l1[i1])
            i1 = i1 + 1
        else :
            l.append(l2[i2])
            i2 = i2 + 1
    if i1 == len(l1) :
        for i in range(i2,len(l2)) :
            l.append(l2[i])
    if i2 == len(l2) :
        for i in range(i1,len(l1)) :
            l.append(l1[i])
    return l
```

Peut être remplacé par ces 2 lignes (écriture « pythonesque »)

```
if i1 == len(l1) : l = l + l2[i2:]
if i2 == len(l2) : l = l + l1[i1:]
```

Code de la fonction récursive tri().

Technique propre à python, qui utilise l'opérateur [:] pour segmenter la liste.

```
def tri(l) :
    if len(l) == 1 : return l
    else :
     iM = len(l)//2
     return fusion(tri(l[:iM]) , tri(l[iM:]))
l = [23 ,12 ,4 ,56 ,35 ,57 ,3 ,11 ,6 ]
>>> tri(l)
[3, 4, 6, 11, 12, 23, 35, 56, 57]
```

Code de la fonction récursive tri().

Technique générale qui utilise la même liste mais en s'intéressant uniquement aux valeurs comprises entre 2 indices iD et iF (i départ et i fin)

```
def tri(l,iD,iF) :
    if iD == iF :
        return [l[iD]]
    else :
        iM = (iD + iF) //2
        return fusion(tri(l,iD,iM) , tri(l,iM+1,iF))
l = [23 ,12 ,4 ,56 ,35 ,57 ,3 ,11 ,6 ]
```

```
>>> tri(l,0,len(l)-1)
[3, 4, 6, 11, 12, 23, 35, 56, 57]
```

Efficacité : Le paradigme « *Diviser pour régner* » apporte ici un gain en termes d'efficacité, par rapport aux méthodes naïves de tri par ***sélection*** ou par ***insertion*** qui ont une complexité en $\mathcal{O}(n^2)$.

Pour réaliser un tri ***fusion*** sur une liste de taille n, on réalise $log_2(n)$ divisions. Pour chacune d'elles, on a environ n opérations de comparaisons, car il s'agit de fusionner les n éléments de la liste. On peut ainsi dire que la complexité de cet algorithme de tri fusion est au final en $O(n \times log_2(n))$.

Par exemple, pour une liste de taille $n = 1\,000\,000$, on a $log_2(1\,000\,000) \approx 20$ et donc $n \times log_2(1\,000\,000) \approx 20\,000\,000 \approx 20 . 10^6$. Par contre, $n^2 = (10^6)^2 = 10^{12}$.

Ainsi une recherche naïve demande environ $\frac{10^{12}}{20 . 10^6} = 50\,000$ fois plus de comparaisons qu'un tri fusion.

$$\log_2(1000000)$$

$$6 \cdot \log_2(5)+6 \approx 19.93157$$

6- TRI RAPIDE UNE LISTE :

Cette méthode de tri consiste à placer un élément de la liste, appelé pivot, à sa place définitive en permutant tous les éléments de telle sorte que tous ceux qui sont inférieurs au pivot soient à sa gauche et que tous ceux qui sont supérieurs soient à sa droite. Pour chacune des sous-listes gauche et droite obtenues, on répète alors le processus : on y définit un nouveau pivot et on répartit les éléments autour de celui-ci. Le processus est répété récursivement jusqu'à ce que l'ensemble des éléments soient triés.

La position du pivot peut être définit aléatoirement.

Dans cette méthode :

1. **Diviser** : on divise le problème (liste gauche et liste droite autour d'un pivot)
2. **Régner** : On trie les valeurs des sous-listes en répartissant les éléments autour du pivot
3. **Combine**r : On reconstitue la liste entière en ajoutant chaque morceau trié

Pour l'illustrer, on utilise cette méthode sur la liste suivante :

	$[23, 12, 4, 56, 35, 57, 3, 11, 6]$
Etape 1	$[4, 3, 11, 6] + [12] + [23, 56, 35, 57]$
Etape 2	$[3] + [4] + [11, 6] + [12] + [23, 35] + [56] + [57]$
Etape 3	$[3] + [4] + [6] + [11] + [\ ] + [12] + [\ ] + [23] + [35] + [56] + [57]$
Reconstitution	$[3, 4, 6, 11, 12, 23, 35, 56, 57]$

Code de la fonction récursive triRapide().

```
from random import randint

def triRapide(l) :
    if len(l) < 2 : return l
    else :
        pivot = randint(0,len(l)-1)
        l1 = []
        l2 = []
        lm = []
        for i in range(len(l)) :
            if l[i] < l[pivot] :
                l1.append(l[i])
            elif l[i] > l[pivot] :
                l2.append(l[i])
            else : lm.append(l[i])
        return triRapide(l1) + lm + triRapide(l2)
```

Travail à faire :

1- Ecrire dans un fichier nommé triRapide.py, le code complet de la fonction triRapide(). Cette fonction a comme paramètre une liste de nombre. Elle renvoie une liste avec ces mêmes nombres triés dans l'ordre croissant. On donne ci-dessous 2 exemples d'exécution :

```
>>> triRapide([4 , 23 ,12 ,4 ,56 ,35 ,57 ,3 ,11 ,6 ])
[3, 4, 4, 6, 11, 12, 23, 35, 56, 57]

>>> triRapide([])
[]
```

2- Ecrire dans un fichier nommé comparaison.py, les codes des fonction triSelection() , triFusion() et triRapide(). Exécuter le programme principal ci-dessous qui permet de tester les temps de calcul pour des listes de taille N.

```
# programme principal
N =1000
liste = [randint(-5*N , 5*N) for i in range(N)]
if N < 100 : print('liste non triée : \n',liste)

nb = 0
print("TRI FUSION :")
deb = time()
l = triFusion(liste)
fin = time()
if N < 100 : print('liste triée : \n', l)
print(f"""liste de taille {N} tri fusion = {fin-deb} s avec {nb} comparaisons """)

nb = 0
print("TRI RAPIDE :")
deb = time()
l = triRapide(liste)
fin = time()
if N < 100 : print('liste triée : \n', l)
print(f"""liste de taille {N} tri rapide = {fin-deb} s avec {nb} comparaisons """)

nb = 0
print("TRI SELECTION :")
deb = time()
l = triSelection(liste)
fin = time()
if N < 100 : print('liste triée : \n', liste)
print(f"""liste de taille {N} tri sélection = {fin-deb} s avec {nb} comparaisons """)
```

On obtient l'affichage suivant dans la console :

```
>>> (executing file "comparaison des temps calculs.py")
TRI FUSION :
liste de taille 1000 tri fusion = 0.006346702575683594 s avec 9976 comparaisons
TRI RAPIDE :
liste de taille 1000 tri rapide = 0.0030188560485839844 s avec 11724 comparaisons
TRI SELECTION :
liste de taille 1000 tri sélection = 0.09268760681152344 s avec 499500 comparaisons
```

Compléter le tableau suivant qui donne les temps de calculs pour des tailles de liste différentes :

Taille liste	$N = 10$	$N = 100$	$N = 1\,000$	$N = 10\,000$	$N = 100\,000$	$N = 1\,000\,000$
Temps tri Fusion	0.0 s	0.002 s	0.006 s	0.0937 s	1.11 s	12.2 s
Temps tri Rapide	0.0 s	0.0 s	0.007 s	0.0816 s	0.916 s	9.4 s
Temps tri Selection	0.0 s	0.002 s	0.086 s	8.41 s	$\approx 8.41 \times 100$	$\approx 8.41 \times 10\,000$

3- Modifier les scripts des fonctions afin de rajouter un compteur de comparaisons nb. Compléter le même tableau en donnant la valeur de ces compteurs :

Taille liste	$N = 10$	$N = 100$	$N = 1\,000$	$N = 10\,000$	$N = 100\,000$	$N = 1\,000\,000$
Nb avec tri Fusion	34	672	9976	133 616	1 668 928	19 951 424
Nb avec tri Rapide	27	637	11 032	157 706	2 037 361	28 176 061
Nb avec tri Selection	45	4950	499 500	49 995 000	$\approx 49\,995\,000 \times 100$	$\approx 49\,995\,000 \times 10\,000$