ALG. Diviser pour régner

I/ Diviser pour régner

L'expression « **diviser pour régner »** (*divide and conquer en anglais*) est très ancienne. Déjà, au temps des Romains, elle était employée au Sénat à des fins politiques ou par les chefs des armées à des fins militaires.

Le paradigme « **diviser pour régner »** repose sur 3 étapes :

- **DIVISER** : le problème d'origine est divisé en un certain nombre de sous-problèmes.
- **RÉGNER** : on résout chaque sous-problème une seule fois (les sous-problèmes sont plus faciles à résoudre que le problème d'origine).
- **COMBINER** : les solutions des sous-problèmes sont combinées afin d'obtenir la solution du problème d'origine.

Les algorithmes basés sur le paradigme "diviser pour régner" sont très souvent des **algorithmes récursifs**.
Voici quelques exemples d'algorithmes l'utilisant :

- La recherche d'un palindrome.
- La recherche dichotomique.
- Le tri-fusion.

II/ Principe d'un palindrome

Un **palindrome** est une chaîne de caractères qui peut se lire dans les deux sens, de gauche à droite mais aussi de droite à gauche.

La fonction *palindrome(ch)* ci-dessous prend en argument une chaîne de caractères et renvoie *True* s'il s'agit d'un palindrome et *False* sinon.

```
def palindrome(ch) :
    # Cas d'arrêt : chaîne vide ou à un seul caractère :
    # c'est un palindrome
    if len(ch) <= 1 :
        return True

    # Cas général : égalité des extrémités et test de
    # la nouvelle chaîne privées des extrémités
    else :
        return ch[0] == ch[len(ch)-1] and palindrome(ch[1:len(ch)-1])


# Jeu de tests
print(palindrome("ressasser"))   # Attendu : True
print(palindrome("radar"))       # Attendu : True
print(palindrome("toto"))        # Attendu : False
```

```
True
True
False
```

Remarque : les instructions *ch[len(ch)-1]* et *ch[-1]* sont identiques en langage Python.

Le problème initial est bien découpé en deux sous-problèmes : l'un très simple dont la réponse est immédiate et l'autre qui est une inversion plus simple du problème initial (chaîne de caractères plus courte).

Pour éviter la recopie de la chaîne, on peut utiliser les indices :

```python
def palindrome(ch,deb,fin) :
    # Cas d'arrêt : chaîne vide ou à un seul caractère :
    # c'est un palindrome
    if fin - deb < 1 :
        return True

    # Cas général : égalité des extrémités et test de
    # la nouvelle chaîne privées des extrémités
    else :
        return ch[deb] == ch[fin] and palindrome(ch,deb+1,fin-1)


# Jeu de tests
print(palindrome("ressasser",0,len("ressasser")-1))   # Attendu : True
print(palindrome("radar",0,len("radar")-1))           # Attendu : True
print(palindrome("toto",0,len("toto")-1))             # Attendu : False
```

```
True
True
False
```

Cet algorithme a un coût linéaire, la chaîne de caractères n'étant parcourue qu'une fois et chacun d'entre eux n'est comparé qu'une seule fois à un autre.

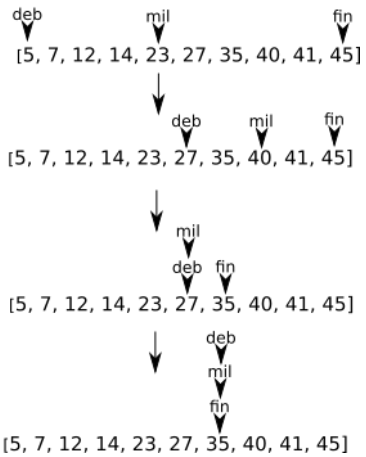
III/ Principe de la recherche dichotomique

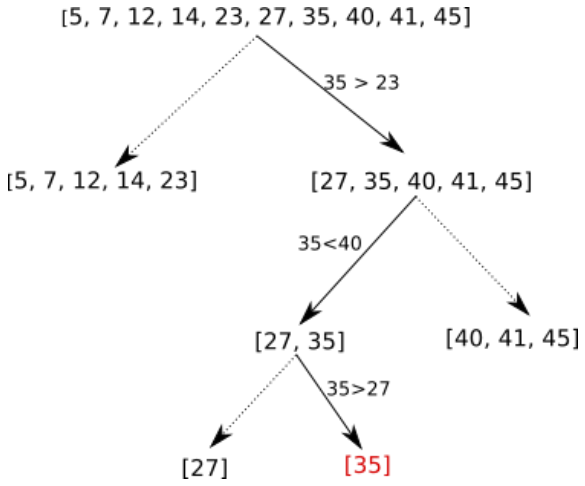
Le coût d'une recherche d'un élément dans un tableau de taille *n* est de l'ordre de *n* (complexité linéaire). En effet, dans le pire des cas, il faut parcourir toute la liste jusqu'au dernier élément.

L'idée est d'abord de **trier** ce tableau en vue de faciliter les recherches ultérieures d'éléments. Trier des nombres est une activité ancienne, on a trouvé une tablette babylonienne datant de l'an -200 avec une longue liste de nombres qui avait été triée dans ce but (Donald Knuth, *The Art of Computer Programming*).
C'est au début des années 1960 qu'a été publié un algorithme de recherche dichotomique fonctionnant pour toute taille de tableau par D.H Lehmer, qui travaillait autour du projet ENIAC.

En savoir plus, projet ENIAC, lien ici : http://www.espace-turing.fr/Le-projet-ENIAC-sur-les-rails.html

Exemple : dans le tableau *t = [5,7,12,14,23,27,35,40,41,45]*, on recherche l'élément *35*.
Deux représentations possibles :





L'idée est donc de définir le milieu du tableau (variable "mil") et de couper le tableau en 2, on se retrouve avec 2 tableaux. On garde uniquement le tableau qui peut contenir la valeur recherchée. On recommence le processus jusqu'au moment où l'on "tombe" sur la valeur recherchée ou que l'on se retrouve avec un tableau contenant un seul élément : si l'élément unique du tableau n'est pas l'élément recherché, on renverra *False*.

La **complexité** de la recherche dichotomique est **logarithmique** puisqu'à chaque passage, on divise le nombre d'éléments dans lesquels chercher la valeur souhaitée par 2.

Au départ, cet algorithme a été utilisé pour des tableaux comportant un nombre d'éléments selon la formule $n = 2^p-1$ tout simplement car leurs partages successifs sont évidents.

Ainsi, pour une liste de 127 éléments :

Nbre de tests	1	2	3	4	5	6	7
Nbre d' éléments	127	63	31	15	7	3	1
Ecriture en binaire	1111111	111111	11111	1111	111	11	1

On peut en déduire qu'en sept tests au maximum, on peut prouver l'existence d'une valeur dans un tableau comportant 127 données.
On remarque aussi qu'il s'agit du **nombre de bits** composant **l'écriture binaire** de ce **nombre d'éléments**.

Voici deux méthodes pour programmer l'algorithme de recherche dichotomique (itérative / récursive).

```python
# Paradigme itératif
def dichotomie(liste, val) :
    # Indices minimum et maximum de la liste.
    iMin = 0
    iMax = len(liste) - 1

    while iMin <= iMax :
        # Indice du centre de la liste.
        iMid = (iMin + iMax)//2

        # Si on a trouvé l'élément, c'est bon.
        if val == liste[iMid] :
            return True
        else :
            # La valeur est dans la première
            # moitié de la liste.
            if val < liste[iMid] :
                iMax = iMid - 1
            # La valeur est dans la deuxième
            # moitié de la liste.
            else :
                iMin = iMid + 1

    return False

# Jeu de tests
tab = [5,7,10,12,15,18,19,22,25,35,41]
print(dichotomie(tab,22))    # Attendu : True
print(dichotomie(tab,27))    # Attendu : False
```

```
True
False
```

```python
# Paradigme récursif
def dichotomie(liste, val, iMin, iMax) :

    # Cas d'arrêt (un seul élément).
    if iMin == iMax :
        return liste[iMin] == val

    else :
        iMid = (iMin + iMax)//2

        # Si on a trouvé l'élément, c'est bon.
        if val == liste[iMid] :
            return True
        elif val < liste[iMid] :
            # Si iMid, iMin et iMax sont égaux
            # sans avoir trouvé la valeur, on renvoie
            # la valeur `False`
            if iMid == iMin :
                return False
            else :
                return dichotomie(liste, val, iMin, iMid-1)
        else :
            return dichotomie(liste, val, iMin+1, iMax)


# Jeu de tests
tab = [5,7,10,12,15,18,19,22,25,35,41]
print(dichotomie(tab,22,0,len(tab)-1))    # Attendu : True
print(dichotomie(tab,27,0,len(tab)-1))    # Attendu : False
```

```
True
False
```

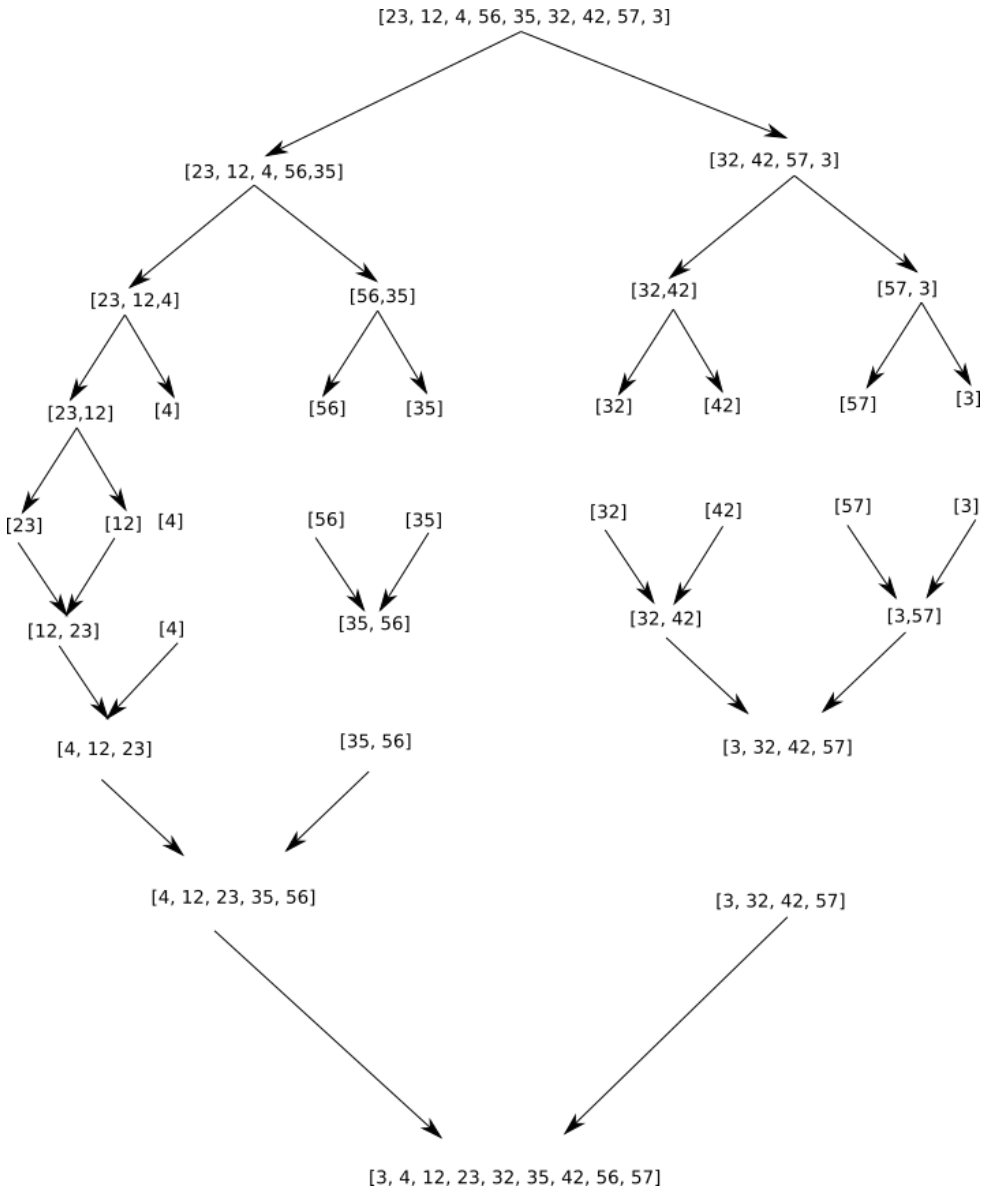
Important : le tableau doit évidemment être trié dans l'ordre croissant pour pouvoir utiliser la recherche dichotomique !

III/ Principe du tri-fusion

En Première, deux algorithmes de tri ont été étudiés : le **tri par insertion** et le **tri par sélection** qui ont une complexité quadratique.
Attention : ils doivent être maîtrisés, y compris leur programmation.

Voici la présentation d'une nouvelle méthode de tri, le tri-fusion. Comme pour les algorithmes déjà étudiés, cet algorithme de tri fusion prend en entrée un tableau non trié et donne en sortie, le même tableau, mais trié.

Voici un exemple d'application de cet algorithme sur le tableau A = [23, 12, 4, 56, 35, 32, 42, 57, 3] :



On remarque que dans le cas du tri-fusion, la phase "RÉGNER" se réduit à sa plus simple expression, en effet, à la fin de la phase "DIVISER", Il n'y a à trier des tableaux comportant un seul élément, ce qui est évidemment trivial.

Voici un exemple de programme pour l'algorithme de tri-fusion :

```python
def fusion(liste1, liste2) :
    # Liste triée finale
    liste = []
    i,j = 0,0

    # Tant qu'il reste des éléments à trier dans les deux liste
    while i < len(liste1) and j < len(liste2) :
        # Cas du plus petit élément dans `liste1` : on le met
        # dans `liste`
        if liste1[i] <= liste2[j] :
            liste.append(liste1[i])
            i += 1
        else :
            liste.append(liste2[j])
            j += 1

    # S'il reste des éléments dans liste1 (et pas dans liste2)
    if i < len(liste1) :
        liste += liste1[i:]
    # Sinon dans `liste2`
    elif j < len(liste2) :
        liste += liste2[j:]

    return liste


# Jeu de test (pour la fonction fusion())
liste1 = [3,5,8,9,12,15]
liste2 = [1,2,6,10]
# Attendu : [1, 2, 3, 5, 6, 8, 9, 10, 12, 15]
print(fusion(liste1,liste2))
```

```
[1, 2, 3, 5, 6, 8, 9, 10, 12, 15]
```

```python
def tri_fusion(liste) :
    # Cas d'arrêt
    if len(liste) < 2 :
        return liste

    # Cas général
    # Division par deux successives
    # de chaque sous-liste.
    iMid = len(liste)//2
    liste1 = tri_fusion(liste[:iMid])
    liste2 = tri_fusion(liste[iMid:])

    return fusion(liste1, liste2)

# Jeu de test
tab = [5,7,3,5,9,6,8,1,4,6]
# Attendu : [1, 3, 4, 5, 5, 6, 6, 7, 8, 9]
print(tri_fusion(tab))
```

```
[1, 3, 4, 5, 5, 6, 6, 7, 8, 9]
```

Pour la complexité, on peut -de manière intuitive- raisonner ainsi, en supposant qu'un tableau possède *n* éléments :

- Par divisions par deux successives, il faut $log_2(n)$ opérations pour parvenir à un seul élément (recherche dichotomique).
- Il y aura en tout *n* listes composés de singletons soit *n x log2(n)* opérations en tout.
- La fusion a un coût linéaire, à chaque comparaison entre deux éléments, un est enlevé d'une des deux listes.

On peut en déduire une complexité de **n x log2(n)** pour le **tri-fusion**, ce qui est bien plus performant qu'un **tri par sélection ou insertion** dans le cas général.
A noter tout de même que le tri par insertion est efficace dans des listes pratiquement triées (coût linéaire) !