AL03 – Diviser pour régner

1. L'algorithme de dichotomie

1.1. La version impérative

Exercice 1: Dichotomie impérative

Ecrire la fonction `recherche_dicho_imperative(tab, val)` qui renvoie `True` si `val` est dans `tab` et `False` sinon.

Code Python

```python
def recherche_dicho_imperative(tab, val):
    pass

tab = [1, 5, 7, 9, 12, 13]
assert recherche_dichotomique(tab, 12) == True
assert recherche_dichotomique(tab, 17) == False
```

À chaque tour de la boucle while, la taille de la liste est divisée par 2. Ceci confère à cet algorithme une complexité logarithmique (bien meilleure qu'une complexité linéaire).

1.2. Version récursive

Remarque 1: Le slicing permet de ne garder qu'un bout de liste

Code Python

```python
L = [3, 2, 5, 6, 9, 8, 5]
assert L[:] == L
assert L[2:] == [5, 6, 9, 8, 5]
assert L[:4] == [3, 2, 5, 6]
assert L[2:4] == [5, 6]
```

Exercice 2: Écrire la fonction `dichotomie_rec` avec slicing et la tester

2. Diviser pour régner

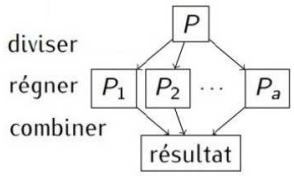
Définition 1: Diviser pour régner

Un problème peut se résoudre en employant le paradigme **Diviser pour Régner** lorsque:

- il est possible de décomposer ce problème en sous problèmes indépendants;
- la taille de ces sous problèmes est une fraction du problème final.

La méthode est composée de trois phases:

- la `division`: le problème initial est divisé en sous problème de taille inférieure;
- le `règne`: chaque sous-problème est résolu soit récursivement, soit directement si trivial;
- la `combinaison`: les solutions des sous problèmes sont combinés pour obtenir la solution du problème initial.



P est un problème de taille n
$P_1, P_2, \ldots, P_a$ sont des problèmes de taille $\frac{n}{a}$
Approche récursive : chaque P_i est à son tour décomposé

Exercice 3: Algorithme de recherche dans une liste non triée

```
si la liste est vide alors l'élément n'appartient pas à la liste
si la liste ne comporte qu'un seul élément alors l'élément est présent s'il est égal à L[0]
sinon on recherche l'élément dans la liste gauche ou dans la liste droite
```

Écrire cette fonction `recherche(L, val)` en python

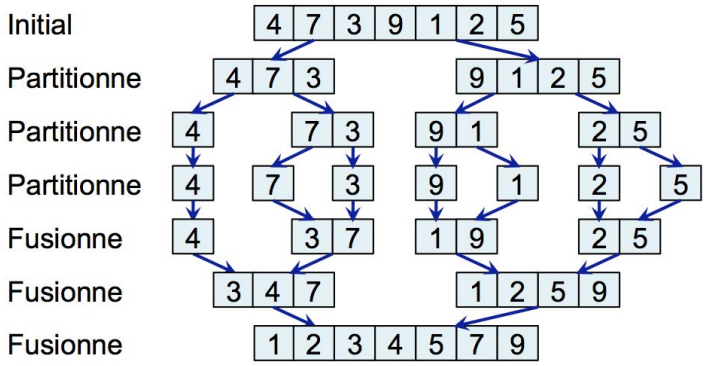
3. Le tri fusion

Pour trier une paquet de copie dans l'ordre alphabétique, il est possible de scinder le paquet en deux sous-paquets de même taille puis de trier chacun de ces sous-paquets avant de les fusionner pour obtenir le paquet trié.

Pour trier ce sous-paquet, on peut utiliser la même technique...

On peut scinder les sous-paquets jusqu'à avoir une certitude: si le paquet n'a qu'une seule copie alors ce paquet est trié.

```
si la liste ne contient qu'un seul élément:
    la liste est triée
sinon:
    on fusionne les deux sous listes triées
```



Exercice 4: Écrire la fonction `division(L)` qui renvoie un tuple contenant les deux sous-listes

Code Python

```python
def division(L:list)->tuple:

L1 = [1, 4, 6, 15]
L2 = [3, 9, 10]
assert division (L1) == ([1, 4], [6, 15])
assert division (L2) == ([3], [9, 10])
assert division ([1]) == ([] , [1])
```

Exercice 5: Écrire la fonction `fusion(L1, L2)` qui renvoie une liste fusionnée

```
fonction fusion(L1, L2, L)
    si L1 OU L2 est vide:
        on renvoie L1 + L2
    si le premier élément de L1 et plus petit que le premier élément de L2:
        on insère le premier élément de L1 dans la liste fusionnée
    si le premier élément de L2 et plus petit que le premier élément de L1:
        on insère le premier élément de L2 dans la liste fusionnée
    On appelle récursivement fusion avec le nouveau L1, le nouveau L2 et le nouveau L
```

Code Python

```python
def fusion(L1, L2, L = None):
    if L = None:
        L = []
    return fusion_rec(L1, L2, L)
def fusion_rec(L1, L2, L):
    if L1 == [] or L2 == []:
        return ....
    if ..... < .........:
        ......................
    else:
        ......................
    L = ......................
    return fusion_rec(........, ........, ..........)

assert fusion(L1, L2) == [1, 3, 4, 6, 9, 10, 15]
```

Exercice 6: Ecrire alors la fonction récursive `tri_fusion(L1, L2)`

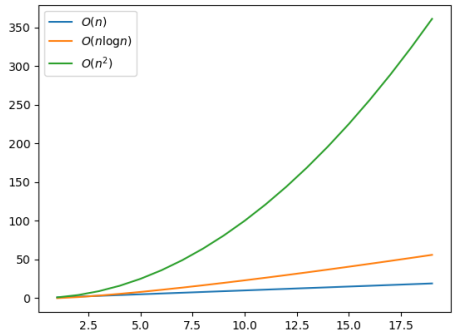
Code Python

```python
def tri_fusion(L:list)->list:

L = [5, 7, 2, 1, 9, 8, 6]
assert tri_fusion(L) == [1, 2, 5, 6, 7, 8, 9]
```

Propriété 1: Complexité de l'algorithme

L'instruction finale
`fusion(tri_fusion(L1), tri_fusion(L2))`
lance deux appels à la fonction `tri_fusion` (avec certes des données d'entrée deux fois plus petites).
On peut alors montrer que la complexité du tri fusion est en `O(n log n)`



Exercice 7: Dessiner de manière schématisé le tri fusion de `[4, 8, 3, 1, 7, 6]`

Exercice 8: Ecrire la trace de cet algorithme lors de l'appel de `tri_fusion([4, 8, 3, 1, 7, 6])`

Appel	n	IF	L1	L2	appel récursif fusion	valeur
trifusion([4,8,3,1,7,6])	6	F	[4,8,3]	[1,7,6]	trifusion([4,8,3]), trifusion([1,7,6])	

Exercice 9: Application à la recherche de maximum dans une liste

```
si la liste ne contient qu'un seul élément:
    c'est le plus grand
on coupe la liste en deux
on récupère le plus grand de la liste de gauche
on récupère le plus grand de la liste de droite
on renvoie le plus grand des deux
```

On pourra utiliser la fonction `max` pour trouver le maximum entre deux valeurs.

Exercice 10:
Application au calcul de l'exponentiation rapide
Il est facile de vérifier que:

- $a^0 = 1$
- si n est pair, $a^n = (a \times a)^{\frac{n}{2}}$
- si n est impair: $a^n = a \times (a \times a)^{\frac{n-1}{2}}$