Algorithmique : principe du Diviser pour Régner

Définition : **C'est une classe d'algorithme où l'on découpe un problème en sous-problèmes qui s'énoncent de la même manière et qu'on recompose à la fin pour former une solution. C'est une approche "du haut vers les bas".**

Principe :

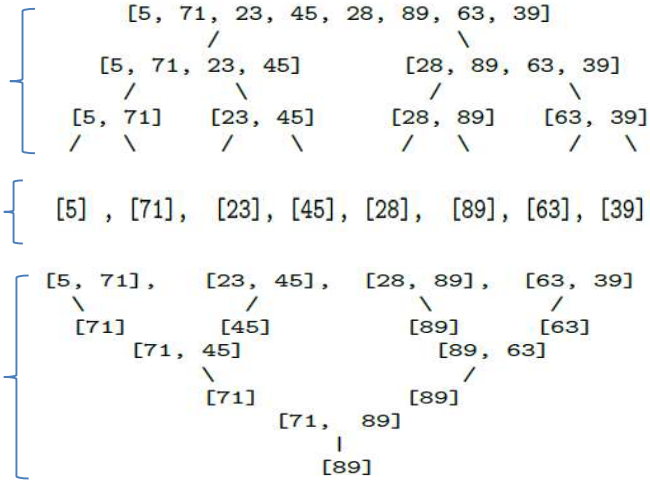
- **Diviser** : partager le problème en sous-problèmes de même nature (souvent de taille *n* /2)
- **Régner** : résoudre ces différents sous-problèmes (généralement récursivement)
- **Combiner** : fusionner les solutions pour obtenir la solution du problème initial

Exemple 1 : Recherche du maximum dans un tableau :

➢ **version classique** itérative avec un parcours de la liste et un test. algo vu en 1ere	**Début** Fonction maxi_iter(tab) maxi ← tab[0] Pour chaque élément elt de tab, Si elt > max alors max ← elt fin si Fin pour Retourner maxi **Fin**	Python :

➢ **Version : diviser pour régner**

1. **Diviser :** On sépare le tableau en deux parties de même taille à un près
2. **Régner :** pour un tableau de taille 1, son maximum est son élément
3. **Combiner** : on ne garde que le plus grand que chaque paire



Fonction maxi_rec (tab) :

début

Si ……..

alors retourner ……

Sinon

n ← taille(tab)//2

L1 ← ……

L2 ← ……..

Retourner ……..

Fin si

Fin

Implémenter les 2 fonctions en python

Remarque : les 2 algorithmes ont une complexité **linéaire** mais la version diviser pour régner est moins efficace ! , cela est du aux appels liés à la récursivité (pile)

Exemple 2 : Recherche d'un élément dans une liste non triée :

En adoptant le paradigme "diviser pour régner", l'idée pour résoudre cette question est de :

- rechercher récursivement l'élément dans la première moitié de la liste et dans la seconde,
- puis de combiner les résultats via l'opérateur logique or.

En effet, l'élément recherché sera dans la liste s'il est dans la première moitié ou dans la seconde.
La condition d'arrêt à la récursivité sera l'obtention d'une liste à un seul élément, car il est alors immédiat de conclure si l'élément recherché appartient à une telle liste ou non.

Voici donc les trois étapes de la résolution de ce problème via la méthode "diviser pour régner" :

- Diviser la liste en deux sous-listes en la “coupant” par la moitié.
- Rechercher la présence de l’élément dans chacune de ces sous-listes. Arrêter la récursion lorsque les listes n’ont plus qu’un seul élément. (=cas de base)
- Combiner avec l’opérateur logique or les résultats obtenus.

Fonction recherche_rec (x,tab) : ‘’ parametre : tab list d’élément , x du type element Sortie : booleen vrai si est dans tab et faux sinon’’ Début : Si taille de tab est 1 alors Retourner ….. ==……] Sinon n ← taille(L)//2 L1 ← …… L2 ← ……. Retourner ……. ou …….. Fin si Fin	Ecrire le programme en python et le comparer à la version itérative.

Remarque : si la recherche d’un élément se fait dans une liste triée, on retrouve le principe de la dichotomie.

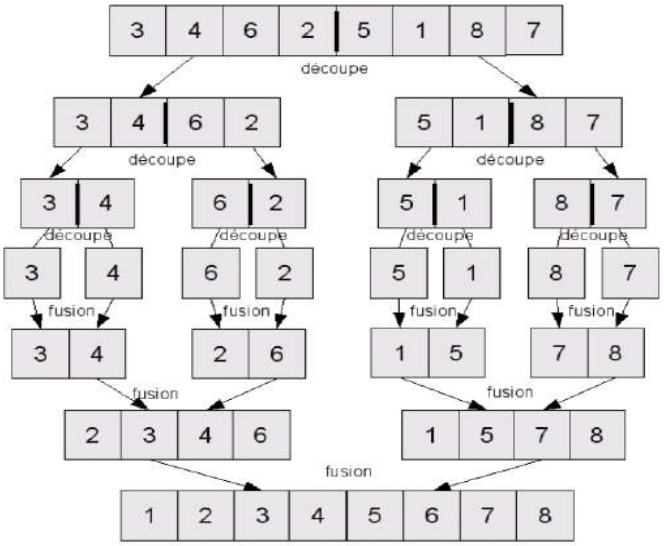
La liste étant triée, après comparaison avec l’élément du "milieu" il est en effet facile de voir dans quelle moitié peut éventuellement se trouver l’élément cherché. On aura plus alors qu’à recommencer récursivement la recherche. *Le terme dichotomie provient du grec ancien "dikhotomia", signifiant "division en deux parties".*

Le tri Fusion : (à connaitre pour le bac)

Définition : Le tri fusion consiste à **trier récursivement** les deux moitiés de la liste, puis à **fusionner** ces deux sous-listes triées en une seule. La condition d’arrêt à la récursivité sera l’obtention d’une liste à un seul élément, car une telle liste est évidemment déjà triée.

Voici donc les trois étapes (diviser, régner et combiner) de cet algorithme :

- **Diviser** la liste en deux sous-listes de même taille (à un élément près) en la "coupant" par la moitié.
- **Trier récursivement** chacune de ces deux sous-listes. Arrêter la récursion lorsque les listes n’ont plus qu’un seul élément.
- **Fusionner** les deux sous-listes triées en une seule.



Commençons par écrire la fonction fusion :

- **paramètres d'entrée :**
  L1 et L2 deux listes triées
- **sortie** une liste triée avec les éléments de L1 et L2.

Compléter ci-contre le code de la version **itérative** de cette fonction fusion

Attention : ** c'est un exercice de la banque de sujets de l'épreuve pratique**

```
def fusion(L1,L2):
    n1 = len(L1)
    n2 = len(L2)
    L12 = [0]*(n1+n2)
    i1 = 0
    i2 = 0
    i = 0
    while i1 < n1 and ... :
        if L1[i1] < L2[i2]:
            L12[i] = ...
            i1 = ...
        else:
            L12[i] = L2[i2]
            i2 = ...
        i += 1
    while i1 < n1:
        L12[i] = ...
        i1 = i1 + 1
        i = ...
    while i2 < n2:
        L12[i] = ...
        i2 = i2 + 1
        i = ...
    return L12
```

- version **récursive**

Fonction **fusion_rec** (L1,L2) :
Début
  Si L1 est vide alors
    Retourner ……
  Si L2 est vide alors
    Retourner ……

  Si …… <= …… alors

    Retourner ……

  Sinon

    Retourner ……

  Fin si
Fin

En utilisant cette fonction fusion, on écrit alors facilement la fonction **Tri_Fusion** pour une liste L

Algorithme Fonction Tri_Fusion (L) : **Début** **Si …… alors** **Retourner ……** **Sinon** **n ← taille(L)//2** **retourner ……** **fin si** **fin**	*D'une manière générale, la complexité du tri par fusion est de l'ordre de nlog(n) tout en comptant le temps de calcul nécessaire pour réaliser les différents découpages. C'est mieux que le tri par sélection ou insertion qui sont en $O(n^2)$ Néanmoins, malgré cette rapidité d'exécution, cet algorithme n'est que très rarement utilisé. En effet, en raison des découpages répétés, l'algorithme nécessite de l'espace mémoire supplémentaire qui peut être un facteur critique lorsque le jeu de données à trier est conséquent.*