« Diviser pour régner »

- La méthode « diviser pour régner » consiste en 3 parties :
  - **Diviser** : Découper un problème initial en plusieurs sous-problèmes plus faciles à résoudre.
  - **Régner** : Résoudre les sous-problèmes, en général de manière récursive.
  - **Combiner** : Calculer une solution au problème initial en combinant les solutions des sous-problèmes.
- Cette méthode ramène la résolution d'un problème dépendant d'un entier **n** à la résolution d'un ou de plusieurs sous-problèmes indépendants dont la taille des entrées passe de **n** à **n / 2** ou une fraction de **n**.
- Les algorithmes utilisant cette méthode s'écrivent souvent de manière **récursive**, la taille du problème étant divisée à chaque appel récursif plutôt que seulement réduite d'une unité.

L'exemple du tri fusion :

On souhaite écrire une fonction **récursive** fusion qui retourne la **fusion** de **deux listes triées**.

- Quels sont les **deux** cas de base ?
  Rappel : Les cas de base sont nécessaires pour que l'algorithme se termine.

Cas 1 : La liste 1 est vide, auquel cas on retourne la liste 2.

Cas 2 : La liste 2 est vide, auquel cas on retourne la liste 1.

- Quels sont les **deux autres cas** à identifier ?

Cas 3 : Le premier élément de la liste 1 est inférieur au premier élément de la liste 2. On retourne la concaténation du premier élément de la liste 1 et de la fusion du reste de la liste 1 avec la liste 2.

Cas 4 : Dernier cas, le premier élément de la liste 1 est supérieur au premier élément de la liste 2. On retourne la concaténation du premier élément de la liste 2 et de la fusion de la liste 1 avec le reste de la liste 2.

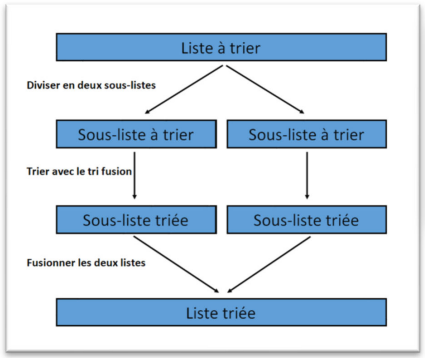
Voici tout d'abord la fonction Python **récursive** fusion qui retourne la **fusion** de **deux listes triées** :

```python
def fusion(L1, L2):
    """ Fonction retournant une liste triée, fusion de deux listes triées.
    :param L1: (list d'int) Liste d'entiers triée
    :param L2: (list d'int) Liste d'entiers triée
    :return: (list d'int) Fusion des deux listes triées """

    if L1 == []:
        return L2
    elif L2 == []:
        return L1
    elif L1[0] < L2[0]:
        return [L1[0]] + fusion(L1[1:], L2)
    else:
        return [L2[0]] + fusion(L1, L2[1:])
```

On souhaite à présent écrire la fonction Python **tri_fusion**, faisant appel à la fonction **fusion**, et qui effectue le **tri de deux listes**.

Voici un schéma représentant le problème, ainsi que la fonction **tri_fusion** récursive en Python.




```python
def tri_fusion(Liste):
    """ Effectue le tri fusion de la liste passée en entrée.
    :param Liste: (list d'int) Liste d'entiers
    :return: (list d'int) Une liste d'entiers triée """

    n = len(Liste)
    if n <= 1:
        return Liste
    else:
        m = n // 2
        return fusion(tri_fusion(Liste[:m]), tri_fusion(Liste[m:]))
```

- On souhaite analyser la complexité du **tri fusion**, en comparaison avec le tri **insertion** et le tri **sélection**.
- Le nombre de comparaisons du **tri fusion** dans le **pire des cas** est de :

$comp(n) = comp(\frac{n}{2}) + comp(\frac{n}{2}) + n - 1$ si **n** est **pair**

$comp(n) = comp(\frac{n}{2}) + comp(\frac{n}{2}) + n - 2$ si **n** est **impair**

n – 1 et **n – 2** sont d'ordre linéaire, on peut donc écrire la complexité :

$$C(n) = C(\frac{n}{2}) + C(\frac{n}{2}) + O(n)$$

À partir de cette écriture, on peut en déduire que la complexité temporelle du tri fusion est en $O(n \log n)$ dans tous les cas.

On parle de complexité **quasi linéaire**, donc un peu moins efficace qu'un coût **linéaire O(n)**.

En comparaison avec les tris sélection et insertion qui sont quadratiques (dans tous les cas pour le tri sélection, dans le pire cas pour le tri insertion), on peut en conclure que le tri fusion est, **en comparaison, plus efficace dans l'ensemble**.

Le tri insertion est toutefois **plus efficace** que le tri fusion lorsque la liste est **déjà triée**.

Complexité de la recherche dichotomique :

- La recherche d'un élément dans une liste triée peut se faire avec une recherche **dichotomique**. Il s'agit d'un algorithme (itératif ou récursif) utilisant la méthode « diviser pour régner ».
  1. On regarde si l'élément est au milieu de la liste.
  2. Si pas, on regarde s'il est dans la partie gauche de la liste
  3. Sinon, on regarde dans la partie droite.

Dans le cas de l'algorithme récursif, chaque fois qu'on **double n**, le nombre de comparaisons avec l'élément cherché **augmente de 2**, et on peut donc écrire la complexité :

$C(n) = C(\frac{n}{2}) + 2$ ⇨ **une telle complexité est dite logarithmique, d'ordre** $O(\log n)$