Diviser pour régner

Introduction :

La méthode « diviser pour régner » est une approche de résolution de problèmes qui consiste à décomposer un problème complexe en sous-problèmes plus faciles à traiter afin de résoudre le problème initial.

Nous allons dans un premier temps définir et caractériser la méthode « diviser pour régner ». Nous étudierons ensuite son application avec deux algorithmes mettant en œuvre ce paradigme : la recherche dichotomique et le tri-fusion.

1 Principes généraux

a. Origine et applications

- La **méthode du tri-fusion**, appliquant le principe « diviser pour régner », a été inventée en 1945 par le mathématicien et informaticien américano-hongrois John von Neumann.
- La **méthode algorithmique** « diviser pour régner » a été formalisée en 1946 pour la recherche dichotomique par le physicien américain John William Mauchly.

Mais les principes algorithmiques de type « diviser pour régner » sont eux-mêmes beaucoup plus anciens. Plusieurs siècles avant notre ère, les Babyloniens avaient conceptualisé cette méthode pour faciliter la recherche parmi un ensemble de dates ordonnées chronologiquement. L'algorithme d'Euclide constitue un autre exemple d'approche « diviser pour régner ».

Définition

Méthode « diviser pour régner » :

Le paradigme de programmation « diviser pour régner » consiste à décomposer un problème général en petits sous-problèmes plus simples à résoudre, permettant par recomposition d'aboutir à la résolution du problème général.

Cette méthode est appelée ainsi en référence à sa désignation anglaise : *divide and conquer*.

b. Principales caractéristiques

Le paradigme de programmation « diviser pour régner » nécessite que le problème à résoudre soit décomposable en sous-problèmes, cette décomposition pouvant être récursive. Il s'agit d'une approche de haut en bas, également appelée **approche descendante**.

⚠ Attention

La décomposition du problème en sous-problèmes est également pratiquée dans le cadre du paradigme de programmation dynamique qui fait l'objet d'un cours distinct.
La principale différence entre les deux approches réside dans l'indépendance ou non des sous-problèmes :

- dans le paradigme « diviser pour régner » les problèmes sont différents les uns des autres, ils ne se répètent pas ;
- dans le paradigme de programmation dynamique, les sous-problèmes ne sont pas tous, ils se chevauchent.

L'approche « diviser pour régner » comporte trois étapes : **diviser**, **régner** et **combiner**. Seules les deux premières étapes sont explicitées dans la dénomination du paradigme « diviser pour régner », mais une étape de combinaison des solutions aux sous-problèmes est nécessaire pour aboutir à la résolution du problème général.
Étudions plus précisément chacune de ces trois étapes.

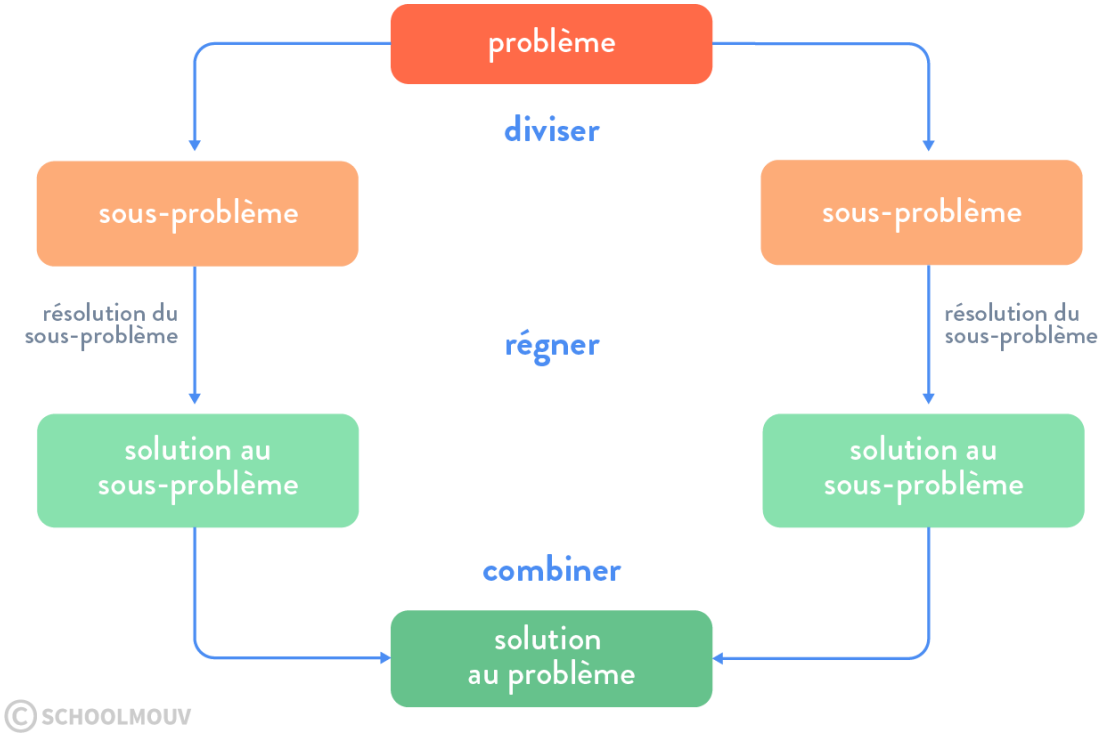
c. Étapes

À retenir

L'ordre des étapes est le suivant :

1. diviser
2. régner
3. combiner

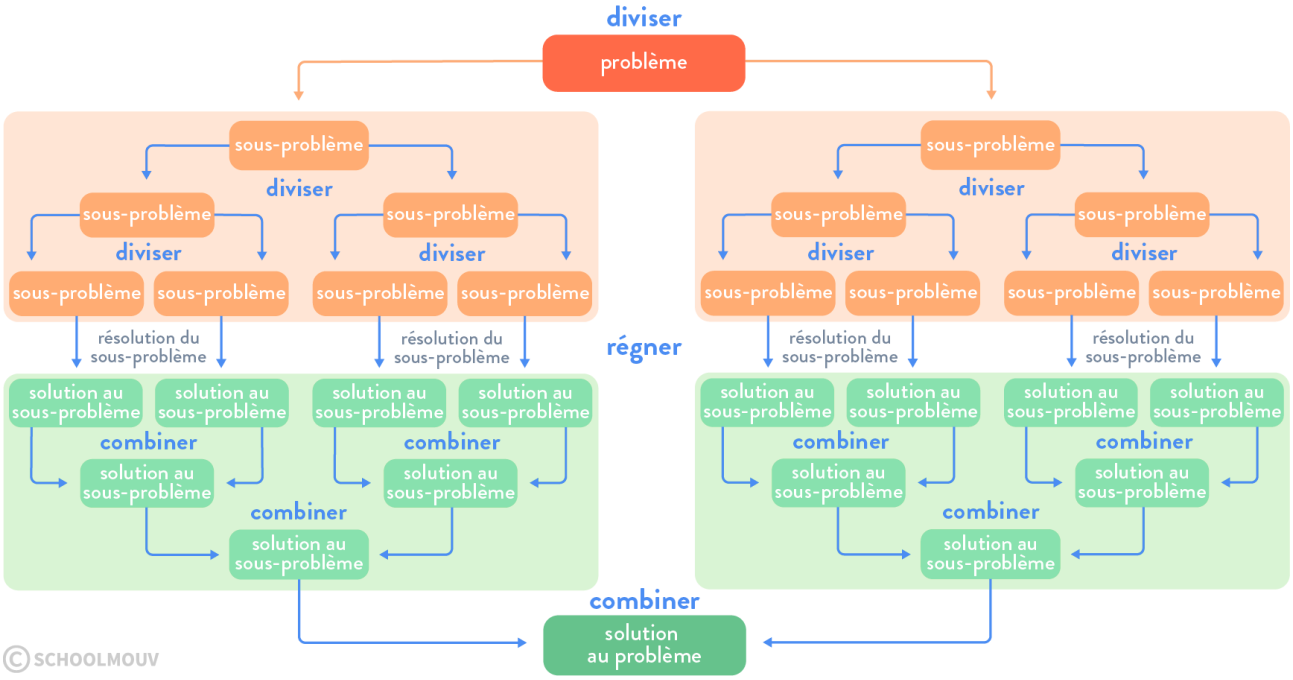
Le schéma de principe ci-après permet de visualiser l'articulation des étapes successives.



1. L'étape « diviser » (*divide* en anglais) consiste à décomposer le problème principal en sous-problèmes.
2. L'étape « régner » (ou « conquérir » pour *conquer* en anglais) consiste à résoudre individuellement chacun des sous-problèmes.
3. L'étape « combiner » (ou « recombiner ») consiste à fusionner l'ensemble des résultats obtenus pour chacun des sous-problèmes afin d'obtenir en résultat final la solution du problème général initialement posé.

Ces sous-problèmes peuvent eux-mêmes être décomposés en sous-problèmes selon une logique récursive. En cas de divisions multiples, il y aura, après les résolutions des sous-problèmes individuels, autant d'étapes de recombinaison qu'il y a eu d'étapes de division.

Le schéma ci-après illustre le processus avec plusieurs niveaux de divisions et de recombinaisons.



→ Le schéma montre que chaque problème ou sous-problème est décomposé en deux sous-problèmes. Cette division par deux est la plus fréquente, mais la méthode pourrait s'appliquer avec une décomposition en un nombre plus important de sous-problèmes d'un même niveau.

À retenir

> Lors de la décomposition de problèmes en sous-problèmes, on fait en sorte que les sous-problèmes soient, dans la mesure du possible, de tailles comparables.

L'**implémentation récursive** est généralement privilégiée dans ce contexte, mais il est également possible de réaliser des **implémentations itératives** de la méthode « diviser pour régner ».

Maintenant que nous avons défini les principales caractéristiques de cette méthode algorithmique, nous allons en étudier deux applications, d'abord la **recherche dichotomique**, puis le **tri-fusion**.

2 Recherche dichotomique

a. Principes généraux

Définition

> **Recherche dichotomique :**
>
> La recherche dichotomique consiste à vérifier si un élément est présent dans un ensemble ordonné. Cette recherche par dichotomie est parfois appelée « fouille dichotomique » ou encore « recherche binaire ».

On utilise couramment cette méthode, sans nécessairement en connaître le nom : chaque fois que l'on cherche un mot dans un dictionnaire par exemple, ou lorsqu'on nous propose de deviner un nombre.

Exemple

> Si on nous demande de deviner un nombre entre 1 et 100, on va commencer par proposer 50. Si c'est moins, on sait que le nombre est à chercher entre 1 et 49 et on proposera 25. Si c'est plus de 25, on proposera 37, là encore pour être à peu près à la moitié de l'intervalle entre 26 et 49. Et ainsi de suite jusqu'à trouver le nombre mystère.

Ainsi, on tend vers l'élément **médian** de l'ensemble considéré. Si ce n'est pas l'élément recherché, on poursuit la recherche dans la première ou dans la seconde partie de l'ensemble, en fonction du résultat de la comparaison entre l'élément recherché et l'élément médian auquel on l'a comparé :

- si l'élément médian est supérieur à l'élément recherché, l'élément recherché est nécessairement dans la première partie de l'ensemble ;

- si l'élément médian est inférieur à l'élément recherché, l'élément recherché est nécessairement dans la seconde partie de l'ensemble.

→ On applique cette même logique à chaque sous-partie jusqu'à trouver l'élément recherché, s'il est présent.

Cette méthode très simple est aussi très efficace. On élimine en effet chaque fois la moitié de l'ensemble des possibles. La réduction est donc très rapide, même pour de grands ensembles. La complexité de l'algorithme de recherche dichotomique est $O(log(n))$.

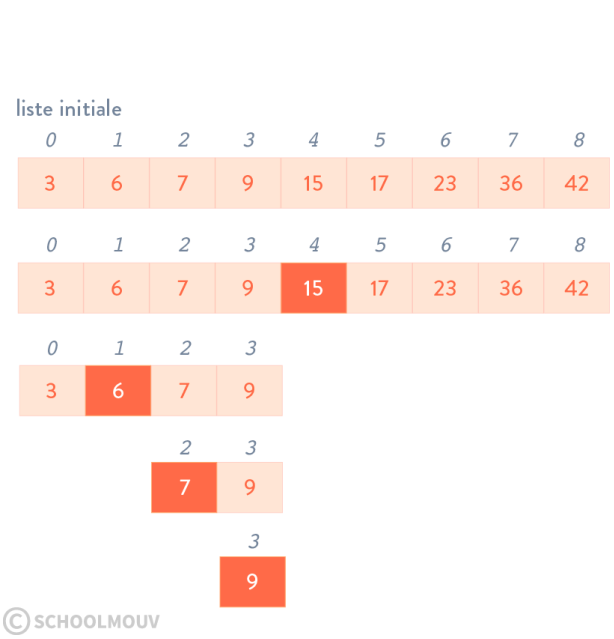
⚠ Attention

> La méthode de recherche dichotomique s'effectue nécessairement sur un ensemble de données préalablement **triées**, conventionnellement par ordre croissant. La méthode s'appuie en effet sur l'ordre de classement des éléments pour pouvoir en éliminer la moitié à chaque étape.

Nous allons réaliser une implémentation de recherche dichotomique pour indiquer si un nombre est, ou non, présent dans une liste triée de nombres entiers.
Il convient ici de préciser que la recherche dichotomique est généralement rattachée au paradigme « diviser pour régner », mais avec la particularité de ne pas résoudre tous les sous-problèmes, puisque la moitié du problème est éliminée à chaque étape, ainsi que nous allons l'illustrer en décomposant ces étapes avec un exemple.

b. Décomposition des étapes

Nous disposons d'une séquence de nombre entiers triés par ordre croissant : $3, 6, 7, 9, 15, 17, 23, 36, 42$. Nous souhaitons vérifier si le nombre 9 est présent dans cette liste en utilisant la recherche dichotomique.



À chaque étape du processus nous déterminons le milieu de la liste, qui est comparé au nombre recherché pour éliminer la demi-liste qui ne peut pas le contenir. On continue jusqu'à le trouver, s'il est présent, ou à constater son absence.

Dans notre illustration, le nombre 9 est présent. Il devient l'élément milieu lors de l'étape où la liste est réduite à un seul élément. Si nous avions recherché le nombre 8, nous aurions constaté à ce stade son absence.

Nous avons présenté les étapes aboutissant aux deux cas de base possibles :

- la présence du nombre recherché ;
- l'absence du nombre recherché.

Constater l'absence du nombre recherché nécessite d'aller jusqu'au bout des divisions possibles de la liste de départ. En revanche la découverte du nombre recherché peut s'avérer beaucoup plus rapide :

- il suffit d'une seule division pour ensuite découvrir le nombre 6 figurant au milieu de la demi-liste conservée ;
- aucune division n'est nécessaire pour découvrir le nombre 15 puisqu'il est situé au milieu de la liste de départ.

C. Implémentation

L'implémentation récursive est une transposition assez littérale des étapes que nous venons d'illustrer.
Les deux cas de base possibles sont :

- la découverte du nombre cherché, qui peut survenir à tout moment ;
- une sous-liste devenue vide au terme des divisions successives.

Nous créons la fonction correspondante :

```
def recherche_dichotomique(valeur, sequence, debut, fin):
    if debut > fin:
        return False
    else:
        milieu = (debut + fin) // 2
        if sequence[milieu] == valeur:
            return True
        else:
            if valeur < sequence[milieu]:
                return recherche_dichotomique(valeur, sequence, debut, milieu - 1)
            else:
                return recherche_dichotomique(valeur, sequence, milieu + 1, fin)
```

Nous vérifions son bon fonctionnement :

```
sequence = [3, 6, 7, 9, 15, 17, 23, 36, 42]
print(recherche_dichotomique(9, sequence, 0, len(sequence)-1))
# affiche True

print(recherche_dichotomique(19, sequence, 0, len(sequence)-1))
# affiche False
```

Cette implémentation oblige à spécifier manuellement les valeurs initiales de début et de fin de liste. Nous pouvons faire évoluer notre fonction, pour éviter à l'utilisateur·rice d'avoir à le faire, en insérant des valeurs par défaut.

```
def recherche_dichotomique(valeur, sequence, debut=None, fin=None):
    if debut is None:
```

```
        debut = 0
    if fin is None:
        fin = len(sequence) - 1
    if debut > fin:
        return False
    else:
        milieu = (debut + fin) // 2
        if sequence[milieu] == valeur:
            return True
        else:
            if valeur < sequence[milieu]:
                return recherche_dichotomique(valeur, sequence, debut, milieu-1)
            else:
                return recherche_dichotomique(valeur, sequence, milieu+1, fin)
```

Avec cette modification, l'appel de fonction s'effectue en spécifiant uniquement le nombre à trouver et la liste dans laquelle le chercher.

```
print(recherche_dichotomique(9, [3, 6, 7, 9, 15, 17, 23, 36, 42]))
# affiche True
```

En restant dans une implémentation récursive, on pourrait aussi passer en argument la liste réduite, au lieu d'en spécifier des bornes plus restreintes par le biais des autres arguments. Néanmoins cela entraînerait la création d'autant de sous-listes qu'il y a d'appels récursifs.

On peut également implémenter une version itérative de la recherche dichotomique, avec une boucle *while* en remplacement des appels récursifs.

```
def recherche_dichotomique(valeur, sequence):
        debut = 0
        fin = len(sequence) - 1
        present = False
        while debut <= fin and not present:
            milieu = (debut + fin) // 2
            if sequence[milieu] == valeur:
                present = True
            else:
```

```
            if valeur < sequence[milieu]:
                fin = milieu - 1
            else:
                debut = milieu + 1
        return present

print(recherche_dichotomique(9, [3, 6, 7, 9, 15, 17, 23, 36, 42]))
# affiche True

print(recherche_dichotomique(41, [3, 6, 7, 9, 15, 17, 23, 36, 42]))
# affiche False
```

On notera que l'élimination des sous-listes non pertinentes à chaque étape de la recherche dichotomique, nous a dispensé d'un traitement de recombinaison puisque nous aboutissons à un **résultat unique**. Nous allons montrer la nécessité de la recombinaison avec le cas d'étude suivant, celui du tri-fusion.

3 Tri-fusion

a. Principe général

Définition

> **Tri-fusion :**
>
> Le principe du tri-fusion consiste à trier une liste de la manière suivante : on la divise récursivement en deux sous-listes. Les sous-listes sont ensuite triées puis fusionnées entre elles.

Le **tri-fusion** (ou tri par fusion) est parfois aussi appelé « tri par interclassement ». La comparaison des algorithmes de tri sort du cadre de ce cours, mais on retiendra que le tri-fusion :

- présente une complexité de $O(nlog(n))$;
- fait partie des algorithmes de tri dont la complexité est constante dans tous les cas de figure, c'est-à-dire qu'elle reste du même ordre en moyenne, dans

le meilleur et dans le pire des cas.

b. Décomposition des étapes

Commençons par un cas très simple pour bien comprendre la décomposition des étapes.
Nous considérons une liste de quatre éléments `[4, 2, 1, 3]`, que nous souhaitons trier en appliquant le tri-fusion s'appuyant sur la méthode « diviser pour régner ».

Les trois étapes sont :

1. diviser ;
2. régner,
3. combiner.

1. Nous effectuons dans un premier temps des divisions successives :
   - la liste `[4, 2, 1, 3]` est divisée en deux listes distinctes : `[4, 2]` et `[1, 3]` ;
   - chacune des listes obtenues précédemment est encore divisée ;
   - pour obtenir quatre listes distinctes : `[4]`, `[2]`, `[1]` et `[3]`.
2. Quand les listes ne contiennent plus qu'un élément, nous atteignons l'étape « régner » qui nous permet de résoudre facilement les sous-problèmes : en effet une liste comprenant un seul élément est nécessairement triée.
3. Nous devons maintenant combiner les résultats obtenus pour chaque niveau de division.
   Les listes individuelles sont fusionnées deux par deux, en ordonnant leurs contenus :
   - `[4]` et `[2]` sont fusionnés en `[2,4]`, et parallèlement `[1]` et `[3]` sont fusionnés en `[1,3]` ;
   - les listes `[2,4]` et `[1,3]` sont finalement fusionnées en `[1,2,3,4]`.

c. Implémentation

La division en sous-listes s'effectue sur la même base que pour la recherche dichotomique, à ceci près que tous les sous-problèmes seront traités à l'issue des divisions.

La partie « régner » correspond au cas de base ou cas terminal des appels récursifs, et repose sur le fait qu'une liste composée d'un seul élément est nécessairement triée. C'est également vrai pour une liste vide, et nous devrons prévoir ce cas de figure pour le traitement de listes comprenant un nombre impair de nombres.

La recombinaison des solutions individuelles par fusion de ces dernières constitue le point-clé qui a donné son nom à l'algorithme tri-fusion. Les listes triées doivent, en effet, être rassemblées **deux à deux** pour chaque niveau de division. La fusion s'applique à deux listes triées, qui doivent être fusionnées de manière à former une liste unique triée.

Commençons par implémenter une fonction réalisant cette fusion entre deux listes individuellement triées par ordre croissant :

```
def fusionne(liste1, liste2):
    liste_fusionnee = []
    i, j = 0, 0
    while i < len(liste1) and j < len(liste2):
        if liste1[i] <= liste2[j]:
            liste_fusionnee.append(liste1[i])
            i += 1
        else:
            liste_fusionnee.append(liste2[j])
            j += 1
    while i < len(liste1):
        liste_fusionnee.append(liste1[i])
        i += 1
    while j < len(liste2):
        liste_fusionnee.append(liste2[j])
        j += 1
    return liste_fusionnee
```

On utilise des index, matérialisés par les variables `i` et `j`, pour repérer la progression dans le parcours séquentiel des listes. Les éléments courants de chaque liste sont comparés et seul le plus petit des deux est ajouté à la liste fusionnée.

La première boucle `while` est exécutée tant qu'aucune liste n'a été parcourue en totalité. Les deux boucles suivantes ont pour objet l'ajout à la liste fusionnée des valeurs non encore utilisées de la seule des deux listes qui n'a pas été parcourue en totalité à ce stade.

Nous testons cette fonction de tri avec deux listes individuelles triées.

```
print(fusionne([1, 3, 5, 8], [2, 4, 6, 7]))
# affiche [1, 2, 3, 4, 5, 6, 7, 8]
```

Nous vérifions que la fonction est également en mesure de fusionner des listes comprenant des doublons.

```
print(fusionne([1, 3, 5, 7], [1, 2, 5, 7]))
# affiche [1, 1, 2, 3, 5, 5, 7, 7]
```

Nous vérifions, par ailleurs, que la fonction est également en mesure de fusionner des listes de longueurs inégales, afin d'être en mesure de traiter des paires de listes issues de divisions d'une liste de longueur impaire.

```
print(fusionne([1, 3, 6, 8], [2, 5, 7]))
# affiche [1, 2, 3, 5, 6, 7, 8]
```

Définissons maintenant la fonction principale de tri-fusion. Elle effectue des appels récursifs jusqu'à atteindre des listes contenant au plus un seul élément. Elle combine ensuite les résultats issus des divisions récursives à l'aide de la fonction de fusion créée précédemment.

```
def tri_fusion(liste):
    if len(liste) < 2:
        return liste
    milieu = len(liste) // 2
    premiere_sous_liste =
    tri_fusion(liste[:milieu])
```

- Diviser : création des deux sous-listes (« premiere_sous_liste » et

```
    seconde_sous_liste = tri_fusion(liste[milieu:])
    return fusionne(premiere_sous_liste, seconde_sous_liste)
```

« seconde_sous_liste »)

- Régner : tri fusion sur chaque sous-liste (appel fonction « tri_fusion »)
- Combiner : fusion des résultats issus du tri-fusion de chaque sous-liste (appel fonction « fusionne »

Testons notre fonction avec une liste non triée :

```
print(tri_fusion([3, 2, 5, 9, 7, 8]))
# affiche [2, 3, 5, 7, 8, 9]
```

Nous obtenons bien une liste triée par ordre croissant. Effectuons un autre test avec une liste de longueur impaire :

```
print(tri_fusion([3, 2, 5, 10, 9, 7, 8]))
# affiche [2, 3, 5, 7, 8, 9, 10]
```

Notre fonction de tri-fusion gère correctement les listes de longueur impaire.

> Conclusion :
>
> Nous avons présenté et caractérisé la méthode « diviser pour régner », une approche de résolution de problèmes basée sur la décomposition d'un problème complexe en sous-problèmes indépendants plus faciles à résoudre, dont les solutions individuelles sont ensuite recombinées pour résoudre le problème général.
> Nous avons ensuite étudié deux algorithmes rattachés à la méthode "diviser pour régner" : d'abord la recherche dichotomique dans un ensemble déjà trié, puis le tri-fusion, un algorithme fusionnant des listes individuelles obtenues par divisions successives d'une liste initiale pour trier celle-ci.