Diviser pour régner et recherche dichotomique

1. Définition et idée générale

La stratégie **diviser pour régner** consiste à découper un problème en sous-problèmes *plus petits*, à résoudre ces sous-problèmes, puis à combiner leurs réponses.

> ⚠️ **Attention !**
>
> Cette méthode fonctionne bien quand les sous-problèmes sont *vraiment* plus simples, et que leur combinaison est *« facile »*.

2. Utilisation

On rencontre cette technique dans :

- la recherche dichotomique ;
- le tri-fusion.

2.1. Exemple simple : la recherche dichotomique

La recherche dichotomique se fait sur une liste *triée* par ordre croissant :

- on « vise au milieu » de la liste ;
- soit on a trouvé la valeur cherchée : c'est gagné !
- soit la valeur cherchée est plus petite que la valeur au milieu : on relance une recherche dichotomique sur la première moitié de la liste ;
- soit la valeur cherchée est plus grande que la valeur au milieu : on relance une recherche dichotomique sur la deuxième moitié de la liste ;
- on réitère le processus jusqu'à ce qu'on ait trouvé la valeur cherchée, ou jusqu'à ce qu'on ne puisse plus « couper en deux » la liste (auquel cas, la valeur cherchée n'est pas présente dans la liste).

Version récursive

```
def recherche_dichotomique(tab, x):
    if tab == []:
        return False
    milieu = len(tab) // 2
    if tab[milieu] == x:
        return True
    if x < tab[milieu]:
        return recherche_dichotomique(tab[:milieu], x)
    return recherche_dichotomique(tab[milieu+1:] , x)
```

💭 Remarque : toujours exclure la position `milieu` !

En effet, si `x` n'est pas à la position `milieu`, on cherche soit dans `tab[:milieu]`, soit dans `tab[milieu+1:]`. Dans les deux cas, la position `milieu` est exclue (dans le *slicing* Python, la valeur finale est *toujours* ***exclue***).

⚠️ Attention au coût ! ▼

Dans le code précédent, le *slicing* `tab[:milieu]` ou `tab[milieu+1:]` créent des **copies** de (portions de) la liste `tab` d'origine. Cela induit un surcoût *important* en temps d'exécution : en effet, si l'on tient compte du coût des copies, la complexité peut atteindre $O(n \log n)$ au lieu de $O(\log n)$ au mieux. Ici, les facilités offertes par Python se payent au prix fort en termes d'efficacité... Pour éviter ces copies, on utilise des indices dans la fonction. Cela donne :

```
def recherche_dichotomique(tab, x, gauche, droite):
    if gauche > droite:
        return False
    milieu = (gauche + droite) // 2
    if tab[milieu] == x:
        return True
    if x < tab[milieu]:
        return recherche_dichotomique(tab, x, gauche, milieu - 1)
    else:
        return recherche_dichotomique(tab, x, milieu + 1, droite)

lst = [10, 20, 30, 50, 70, 90]
recherche_dichotomique(lst, 42, 0, len(lst)-1)     # Passer les bornes initiales, ici
```

Version itérative

```python
def recherche_dichotomique(tab, x):
    deb, fin = 0, len(tab) - 1
    while deb <= fin:
        milieu = (deb + fin) // 2
        if tab[milieu] == x:
            return True
        elif x < tab[milieu]:
            fin = milieu - 1
        else:
            deb = milieu + 1
    return False
```

> ⚠️ **Pièges classiques de ce code**
>
> La recherche dichotomique itérative présente deux pièges.
>
> - Si `tab[milieu]` n'est pas la valeur cherchée, inutile de continuer à garder la position `milieu` dans l'intervalle de recherche. Comme on sait déjà qu'elle ne peut pas convenir, il faut la retirer, avec `milieu - 1` ou `milieu + 1` selon le cas. De cette façon, on évite le risque de retomber sur la « même case » et de bloquer l'algorithme. Dit autrement, on s'assure ainsi que la *condition d'arrêt* de la boucle `while` sera bien atteinte.
> - Le calcul de la position moyenne, en raison de la (nécessaire) division entière (ou *euclidienne*) par `2`, peut amener à ce que `deb` et `fin` aient la *même valeur*. Le point précédent garantit alors que, si la valeur cherchée n'est ni en `deb` (ni en `fin`, qui est égal à `deb`), on aura au tour suivant `deb > fin`, et la boucle s'arrêtera.

2.2. Exemple plus avancé : le tri-fusion

Voir la fiche dédiée.

3. Points d'attention complémentaires

- Diviser ne suffit pas : il faut aussi savoir recombiner.
- Le gain de complexité vient essentiellement de la division répétée en deux parties (à peu près) égales.
- Le tri-fusion est l'exemple classique à connaître.
- Attention aux facilités de Python qui, si elles permettent un code concis et simple, masquent la complexité de certaines opérations.

4. Liens avec d'autres notions

- Le **tri-fusion** est l’exemple central.
- La **récursivité** est très souvent utilisée pour l’implémentation.
- La **complexité** permet d’évaluer le gain.
- La **recherche dichotomique** repose elle aussi sur une division par deux.