1 Diviser pour régner : principe général

Le principe algorithmique « Diviser pour régner » consiste à résoudre un problème en le **subdivisant récursivement en sous-problèmes** plus petits et plus simples à résoudre, puis à combiner les solutions des sous-problèmes pour obtenir la solution au problème global.

Diviser pour régner :

1. Diviser
2. Régner
3. Combiner

2 Recherche dichotomique dans un tableau trié

Nous avons déjà rencontré ce principe en 1ère lors de la recherche dichotomique, en version itérative.

```
Fonction recherche_dichotomique
  Entrées :
  -  tab : tableau trié.
  -  val : valeur recherchée (numérique).
  Sortie : Renvoie indice de la position de val dans tab (ou -1).

  gauche = 0
  droite = taille(tab) - 1
  TANT QUE droite - gauche >= 0 :
    milieu = (gauche + droite) // 2
    SI tab[milieu] == val:
      RENVOYER milieu           # val est dans le tableau à l'indice milieu
    SINON SI tab[milieu] > val: # recherche entre gauche et (milieu - 1)
      droite = milieu - 1
    SINON :                     # recherche entre (milieu + 1) et droite
      gauche = milieu + 1
  RENVOYER -1                   # sortie de boucle sans trouver val
```

Le principe *diviser pour régner* repose plutôt sur des versions récursives de résolution.
Le problème global consiste à trouver une valeur dans un intervalle de la taille totale du tableau. On va diviser cette recherche en se concentrant sur des intervalles plus petits.

1. Diviser : séparer l'intervalle de recherche en un intervalle plus petit.
2. Régner : chercher la valeur dans un petit intervalle.
3. Combiner : solution obtenue automatiquement si on a trouvé la valeur dans l'intervalle réduit.

```
Fonction recherche(tab, val, gauche, droite)
  Entrées :
  -  tab : tableau trié.
  -  val : valeur recherchée (numérique).
  - gauche : indice du début de la recherche (par défaut = 0)
  - droite : indice de fin de la recherche (par défaut = taille(tab) - 1)
  Sortie : Renvoie indice de la position de val dans tab entre les indices gauche et
  ↪  droite (ou -1).

    SI gauche > droite:      # intervalle vide : valeur absente du tableau
       RENVOYER -1
    milieu = (gauche + droite) // 2
    SI tab[milieu] == val:
       RENVOYER milieu         # val est dans le tableau à l'indice milieu
    SINON SI tab[milieu] > val: # recherche entre gauche et (milieu - 1)
       RENVOYER recherche(tab, val, gauche, milieu-1)
    SINON:                     # recherche entre (milieu + 1) et droite
       RENVOYER recherche(tab, val, milieu+1, droite)
```

Rappelons que **cette recherche a un coût logarithmique (en $\log N$)** : càd que si on double la taille du tableau, cela n'augmente le temps de l'algorithme que de une « étape ».
Le logarithme en base 2 (celui manipulé classiquement en informatique) donne la taille en bits d'un nombre entier.

n	n_{bin}	$\log_2 n$
2	10	1
4	100	2
8	1000	3

3 Tri fusion (merge sort)

En classe de 1ère, nous avons étudié deux algorithmes de tri de tableau : **tri par sélection** et **tri par insertion**. Ces 2 algorithmes ont un **coût quadratique** (dans le pire des cas, et en moyenne), càd que si on double la taille du tableau, cela multiplie par 4 le temps de l'algorithme.

Nous allons voir un **algorithme de tri reposant sur le principe *diviser pour régner* qui a un coût en $N.\log(N)$** (on dit aussi quasi-linéaire : on peut démontrer que c'est un coût optimum pour un tri).

On peut retrouver une animation sur les tris sur ce site :
`https://professeurb.github.io/articles/tris/`

Le principe de l'algorithme du tri fusion est le suivant :

1. **Diviser** : séparer le tableau à trier en 2 sous-tableaux (on coupe au milieu).
2. **Régner** : trier chacun des sous-tableaux.
3. **Combiner** : fusionner les 2 sous-tableaux triés.

Cet algorithme nécessite donc une fonction de fusion de 2 tableaux (chacun étant trié). La fusion se fait en comparant le 1er terme de chaque tableau jusqu'à ce qu'un tableau soit vide. Le plus petit des 2 éléments comparés est ajouté au fur et à mesure dans un nouveau tableau.

```
Fonction Fusion
  Entrées :
  -  tab1, tab2 : 2 tableaux triés.
  Sortie : Renvoie un seul tableau trié fusionnant tab1 et tab2

   tab = creer_tableau_vide
   indice1, indice2 = 0, 0
   TANT QUE indice1 < taille(tab1) OU QUE indice2 < taille(tab2):
      SI indice1 >= taille(tab1):
         POUR i DE indice2 À taille(tab2)-1:
             AJOUTER tab2[i] À tab
         RENVOYER tab
      SI indice2 >= taille(tab2):
         POUR i DE indice1 À taille(tab1)-1:
             AJOUTER tab1[i] À tab
         RENVOYER tab
      SI tab1[indice1] < tab2[indice2]
         AJOUTER tab1[indice1] À tab
         INCREMENTER indice1
      SINON
         AJOUTER tab2[indice2] À tab
         INCREMENTER indice2
   RENVOYER tab
```

Remarque : il existe d'autres façons d'écrire la fonction de fusion. On présente par exemple ci-dessous une version récursive.

```
Fonction Fusion(tab1, tab2)
   SI est_vide(tab1):
      RENVOYER tab2
   SI est_vide(tab2):
      RENVOYER tab1
   SINON SI tab1[0] < tab2[0]:
      RENVOYER tab1[0] + Fusion(tab1[1 à fin], tab2)
   SINON
      RENVOYER tab2[0] + Fusion(tab1, tab2[1 à fin])
```

Ayant à notre disposition une fonction de fusion, on peut désormais programmer l'algorithme récursif du tri fusion :

```
Fonction Tri_Fusion
  Entrée : tab : tableau non trié.
  Sortie : Trie le tableau en place

   SI taille(tab) == 1:
      RENVOYER tab
   SINON:
      tab1, tab2 = premiere_moitie(tab), deuxième_moitie(tab)
      tab1 = Tri_Fusion(tab1)
      tab2 = Tri_Fusion(tab2)
      RENVOYER Fusion(tab1, tab2)
```

Exemple de tri du tableau de 7 nombres [15, 14, 17, 5, 7, 19, 10] :

1. Compléter le schéma suivant présentant les étapes de tri fusion sur le tableau.
2. Identifier les étapes de division, fusion (et règne).
3. Combien y a t-il d'étapes jusqu'au règne (cela correspond au nombre d'appels récursifs) ?
4. Quelle doit être la taille du tableau pour avoir une étape de plus ? Puis encore une étape de plus... ?