Chapitre 19 - Diviser pour régner

Objectifs :

▷ Ecrire un algorithme en utilisant la méthode *"diviser pour régner"*.

1 Introduction

Nous avons vu en première deux algorithmes de tris assez naturels, mais peu efficaces : le **tri par insertion** et le **tri par sélection**.

Cette année, nous allons étudier un algorithme beaucoup plus efficace et très utilisé inventé par *John Von Neumann* en 1945 : le **tri par fusion**.

Cet algorithme nous permettra d'illustrer la méthode **diviser pour régner** que nous avions déjà vue lors de la **recherche dichotomique**.

2 La dichotomie (rappels)

Nous avons vu en classe de *Première* l'algorithme de **dichotomie**. Ce mot, qui vient du grec *dikhotomia*, signifie *« division en deux parties »*.

Notre but ici est la recherche de la présence (ou non) d'un élément dans une liste triée. Notre fonction renverra donc un booléen.

La **recherche naïve** (élément par élément) est naturellement de **complexité linéaire**. Nous allons voir que la méthode dichotomique est plus efficace.

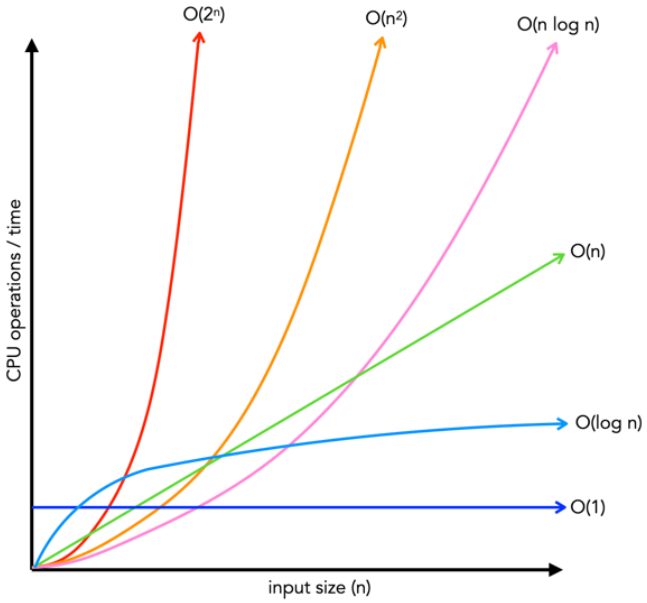
2.1 Version itérative

```
1  def recherche_dichotomique(tab, val) :
2      ''' renvoie True ou False suivant la presence de la valeur val dans le
           tableau deja trie tab.
3      '''
4      i_debut = 0
5      i_fin = len(tab) - 1
6      while i_debut <= i_fin :
7          i_centre = (i_debut + i_fin) // 2     # on prend l'indice central
8          val_centrale = tab[i_centre]          # on prend la valeur centrale
9          # si la valeur centrale est la valeur cherchee
10         if val_centrale == val:
11             return True
12         # si la valeur centrale est trop petite
13         if val_centrale < val:
14             i_debut = i_centre + 1
15         # on ne prend pas la valeur centrale qui a deja ete testee
16         else :
17             i_fin = i_centre - 1
18     return False
```

Exemple d'utilisation :

```
>>> tab = [1, 5, 7, 9, 12, 13]
>>> recherche_dichotomique(tab,12)
True
>>> recherche_dichotomique(tab,17)
False
```

À chaque tour de la boucle `while`, la taille de la liste est divisée par 2. Ceci confère à cet algorithme une **complexité logarithmique** (bien meilleure qu'une **complexité linéaire**).



2.2 Version récursive

2.2.1 Le *slicing*

Pour écrire simplement la version récursive de cet algorithme, nous allons avoir besoin de faire du *slicing* (découpage) de listes.

Exemples de *slicing* :

```
>>> lst = ['a', 'b', 'c', 'd', 'e']
>>> lst[:2]
['a', 'b']
>>> lst[2:]
['c', 'd', 'e']
```

On comprend que :

- `lst[:k]` va renvoyer la sous-liste composée du premier élément jusqu'à celui d'indice k non inclus.
- `lst[k:]` va renvoyer la sous-liste composée du *k-ième* élément (inclus) jusqu'au dernier.
- plus généralement, `lst[k:p]` va renvoyer la sous-liste composée du *k-ième* élément (inclus) jusqu'au *p-ième* (non inclus).

2.2.2 Dichotomie récursive avec slicing

```
1  def dichotomie_rec(tab, val):
2      if len(tab) == 0:
3          return False
4      i_centre = len(tab) // 2
5      if tab[i_centre] == val:
6          return True
7      if tab[i_centre] < val:
8          return dichotomie_rec(tab[i_centre + 1:], val)
9          # On prend la partie droite de liste, juste apres l'indice central
10     else:
11         return dichotomie_rec(tab[:i_centre], val)
12         #On prend la partie gauche de liste, juste avant l'indice central
```

Exemple d'utilisation :

```
>>> tab = [1, 5, 7, 9, 12, 13]
>>> dichotomie_rec(tab,12)
True
>>> dichotomie_rec(tab,17)
False
```

2.2.3 Dichotomie récursive sans slicing

Il est possible de programmer de manière récursive la recherche dichotomique sans toucher à la liste, et donc en jouant uniquement sur les indices :

```
def dicho_rec_2(tab, val, i=0, j=None):
# Pour pouvoir appeler simplement la fonction sans preciser les indices
# on leur donne des parametres par defaut
    if j is None:
        j = len(tab)-1
    if i > j :
        return False
    m = (i+j) // 2
    if tab[m] < val :
        return dicho_rec_2(tab,val,m+1,j)
    elif tab[m] > val :
        return dicho_rec_2(tab,val,i,m-1)
    else :
        return True
```

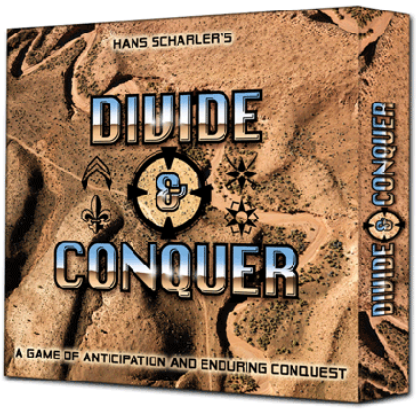
Exemple d'utilisation :

```
>>> tab = [1, 5, 7, 9, 12, 13]
>>> dicho_rec_2(tab,12)
True
>>> dicho_rec_2(tab,17)
False
```

3 Diviser pour régner

Les algorithmes de dichotomie ont tous en commun de **diviser par deux** la taille des données de travail **à chaque étape**.

Cette méthode de résolution d'un problème est connue sous le nom de **diviser pour régner**, ou *divide and conquer* en anglais.



A retenir !

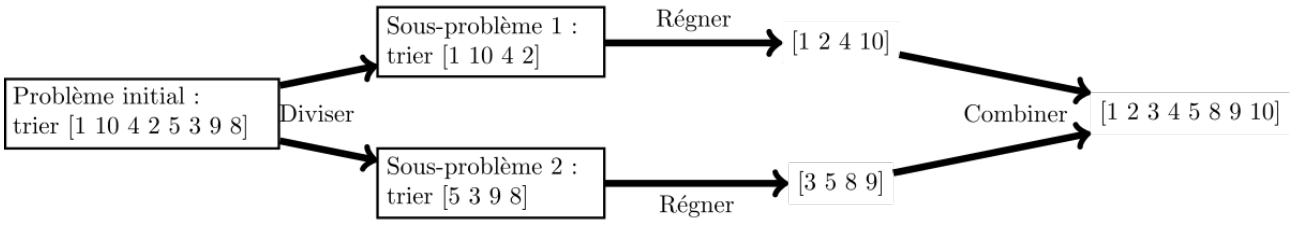
Le principe de **diviser pour régner** consiste à ramener la résolution d'un problème sur n données à la résolution d'un problème sur la moitié des données et poursuivre ce découpage jusqu'à ce que le problème devienne évident (par exemple trier un tableau d'une seule donnée).

Une fois que les solutions des sous problèmes ont été trouvées, on les **combine** pour obtenir la solution du problème complet.

- ▷ **Diviser** : découper un problème initial en sous-problèmes ;
- ▷ **Régner** : résoudre les sous-problèmes (récursivement ou directement s'ils sont assez petits) ;
- ▷ **Combiner** : calculer une solution au problème initial à partir des solutions des sous-problèmes.

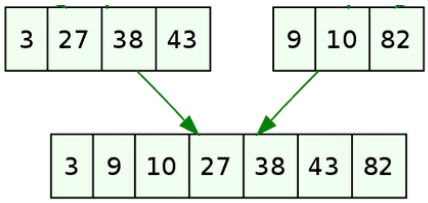
4 Le tri fusion

Ainsi l'algorithme de **tri fusion** qui s'appuie sur le principe *"diviser pour régner"* permet de fusionner deux tableaux déjà triés en un tableau trié.



Exemple :

Pour fusionner ces deux tableaux triés :



Etape par étape

Il suffit d'une itération sur les deux listes en même temps donc ici 5 itérations pour une liste de 7 éléments :

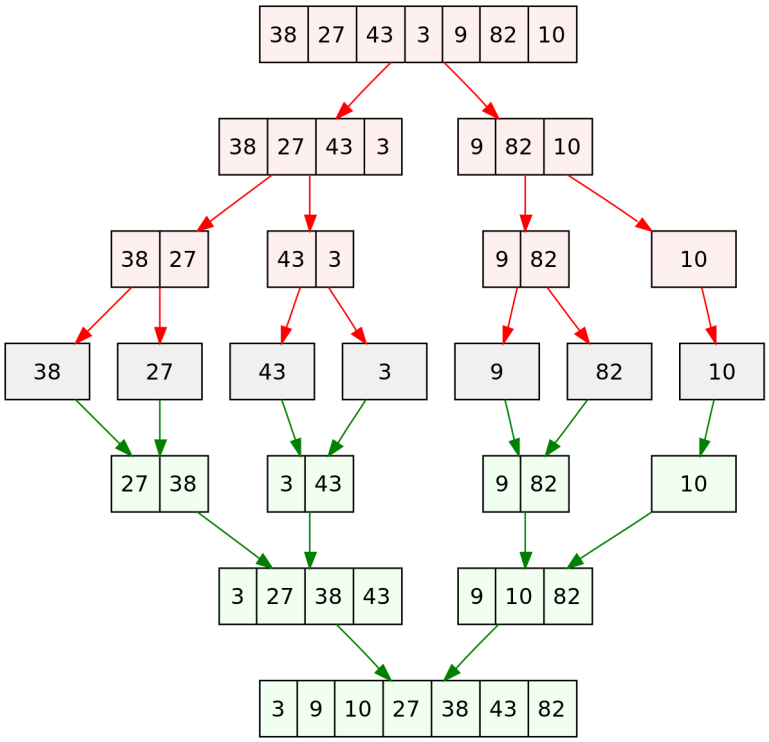
1. On considère `3` et `9`, on place `3`, et on avance sur la 1ère liste.
2. On considère `27` et `9`, on place `9`, et on avance sur la 2e liste.
3. On considère `27` et `10`, on place `10`, ...
4. On considère `27` et `82`, on place `27`, ...
5. On considère `38` et `82`, on place `38`, ...
6. On considère `43` et `82`, on place `43`, et on voit qu'on est arrivé au bout de la première liste.
7. On place maintenant tous les éléments restants de la deuxième liste.

Le **découpage récursif** d'un tableau jusqu'à arriver au cas terminal. On va donc séparer notre algorithme en deux fonctions :

— une qui réalise la **fusion**
— l'autre qui réalise la récursion du tri c'est-à-dire le **découpage**.

Ces deux opérations sont symbolisées sur l'illustration ci-dessous :

— **rouge :** division
— **vert :** fusion



4.1 L'algorithme de fusion

Voici l'**algorithme de fusion** de deux tableaux triés en un seul écrit en pseudo-code :

Algorithme 1 : Fonction fusion de 2 tableaux d'entiers triés

Entrées : $T1$, $T2$: tableaux d'entiers triés
Sortie : T : tableau

$i1 \leftarrow 0$ `// indice du 1er tableau`
$i2 \leftarrow 0$ `// indice du 2e tableau`
$T \leftarrow []$ `// liste vide destinée à accueillir les éléments triés`

tant que *non atteint la fin d'un des tableaux* **faire**
  si $T1[i1] \leq T2[i2]$ **alors**
    Insérer $T1[i1]$ à la fin de tableau
    Incrémenter $i1$
  fin
  sinon
    Insérer $T2[i2]$ à la fin du tableau
    Incrémenter $i2$
  fin
fin

si $i1 <$ *longueur de* $T1$ **alors**
  Insérer tous les éléments restants de $T1$ à la fin du tableau
fin
sinon si $i2 <$ *longueur de* $T2$ **alors**
  Insérer tous les éléments restants de $T2$ à la fin de tableau
fin

return T

4.2 L'algorithme de tri fusion

Voici l'algorithme récursif de tri fusion qui utilise la fonction `fusion` définie précédemment.

En pseudo-code, on retrouve des techniques de découpage du tableau en deux avec des divisions entières `//` vues dans la recherche dichotomique.

Algorithme 2 : Fonction tri_fusion

Entrées : T : tableau
Sortie : T : tableau trié

$n \leftarrow$ Longueur de T;

`// Cas terminal: une liste de un élément est forcément triée`
si $n == 1$ **alors**
  return T;
fin

`// Recursion sur les deux demi-tableaux sinon`
$T1 \leftarrow$ liste des $n//2$ premiers éléments de T;
$T2 \leftarrow$ liste des $n//2$ derniers éléments de T;

`// Renvoi de la fusion des deux tableaux triés`
return *fusion(tri_fusion(T1), tri_fusion(T2))*;

5 Conclusion

Nous avons vu dans ce chapitre un algorithme particulièrement **élégant** et **efficace** pour trier des éléments. Bien sûr, dans la pratique des contraintes de mémoire peuvent intervenir, et là au contraire cet algorithme se révélera **peu performant**, car l'utilisation de la **récursivité** et du tableau intermédiaire le rend très **gourmand** en mémoire.

La méthode *«diviser pour régner»* est une méthode très **efficace** pour résoudre des problèmes **complexes** en les **découpant** en sous problèmes indépendants.

En revanche, elle devient inefficace si les sous-problèmes se chevauchent, et il conviendra alors d'utiliser la technique de la *« Programmation dynamique »* vue précédemment au chapitre 17.

6 Exercices

Exercice 1 : *Dichotomie*

Compléter le code incomplet ci dessous :

```
def recherche_dichotomique(tab, val) :
'''
    renvoie True ou False suivant la presence de la valeur val
    dans le tableau trie tab
'''
    i_debut = ...
    i_fin = ...
    while ... <= ... :
        i_centre = ...
        val_centrale = ...
        if ... == ... :
            return ...
        if ... < ...:
            i_debut = ...
        else :
            i_fin = ...
    return False
```

Pour tester :

```
>>> tab = [1,5,7,9,12,13]
>>> recherche_dichotomique(tab,12)
True
>>> recherche_dichotomique(tab,17)
False
```

Exercice 2 : *L'algorithme de fusion*

Implémenter en Python l'**algorithme de fusion** du cours (*algorithme 1*).

Pour tester :

```
>>> fusion([3,6,8],[2,5,7,12])
[2,3,5,6,7,8,12]
```

Exercice 3 : *L'algorithme de tri fusion*

Implémenter en Python l'**algorithme de tri fusion** du cours (*algorithme 2*).

Pour tester :

```
>>> tri_fusion([0,25,36,41,1,465,2,3,987])
[0,1,2,3,25,36,41,465, 987]
```

Exercice 4 : *L'algorithme de tri par insertion*

Le tri par insertion est un algorithme efficace qui s'inspire de la façon dont on peut trier une poignée de cartes. On commence avec une seule carte dans la main gauche (les autres cartes sont en tas sur la table) puis on pioche la carte suivante et on l'insère au bon endroit dans la main gauche.

Voici une implémentation en Python de cet algorithme :

```
1 def tri_insertion(liste):
2 """ tri par insertion la liste en parametre """
3     for indice_courant in range(1,len(liste)):
4         element_a_inserer = liste[indice_courant]
5         i = indice_courant - 1
6         while i >= 0 and liste[i] > .... :
7             liste[...] = liste[...]
8             i = i-1
9             liste[i+1] = element_a_inserer
```

1. Compléter les lignes 6 et 7 du code ci-dessus pour faire fonctionner le tri par insertion.
2. Tester la fonction avec la liste `notes = [8, 7, 18, 14, 12, 9, 17, 3]`

Exercice 5 : *Sujet Bac - épreuve pratique : sujet n°28 - session 2021*

On considère l'algorithme de tri de tableau suivant : à chaque étape, on parcourt depuis le début du tableau tous les éléments non rangés et on place en dernière position le plus grand élément.

Exemple avec le tableau : `t = [41, 55, 21, 18, 12, 6, 25]`

— Étape 1 : on parcourt tous les éléments du tableau, on permute le plus grand élément avec le dernier. Le tableau devient `t = [41, 25, 21, 18, 12, 6, 55]`
— Étape 2 : on parcourt tous les éléments sauf le dernier, on permute le plus grand élément trouvé avec l'avant-dernier. Le tableau devient : `t = [6, 25, 21, 18, 12, 41, 55]`
— Et ainsi de suite...

Le code de la fonction `tri_iteratif` qui implémente cet algorithme est donné ci- dessous :

```
1 def tri_iteratif(tab):
2     for k in range(..., 0 ,-1):
3         imax = ...
4         for i in range(0, ...):
5             if tab[i] > ... :
6                 imax = i
7             if tab[imax] > ... :
8                 ..., tab[imax] = tab[imax], ...
9     return tab
```

Compléter le code qui doit donner :

```
>>> tri_iteratif([41, 55, 21, 18, 12, 6, 25])
[6, 12, 18, 21, 25, 41, 55]
```

On rappelle que l'instruction `a, b = b, a` échange les contenus de `a` et `b`.

Exercice 6 : *L'algorithme de tri par sélection*

Comme dans la plupart des algorithmes de tri étudiés, nous allons travailler **en place**. Cela signifie que nous ne travaillons que sur la liste initiale, sans en créer de nouvelles. Le tri par sélection sera fait en permutant des éléments.

L'idée de l'algorithme est la suivante (vu en *Première*) :

— on cherche le minimum de toute la liste, et on le place au tout début de la liste.
— on cherche maintenant le minimum de toute la liste SAUF le 1er terme, et on le place en 2ème position.
— on continue ainsi jusqu'à la fin.

Implémenter le tri par sélection en ne manipulant les indices des éléments de la liste.

Pour tester :

```
>>> ma_liste = [7, 5, 2, 8, 1, 4]
>>> tri_selection(ma_liste)
>>> ma_liste
[1, 2, 4, 5, 7, 8]
```

Exercice 7 : *Sujet Bac - épreuve pratique : sujet n°43 - session 2023*

La fonction `tri_bulles` prend en paramètre une liste `T` d'entiers non triés et renvoie la liste triée par ordre croissant.

Le tri à bulles est un tri en place qui commence par placer le plus grand élément en dernière position en parcourant la liste de gauche à droite et en échangeant au passage les éléments voisins mal ordonnés (si la valeur de l'élément d'indice `i` a une valeur strictement supérieure à celle de l'indice `i+1`, ils sont échangés). Le tri place ensuite en avant-dernière position le plus grand élément de la liste privée de son dernier élément en procédant encore à des échanges d'éléments voisins. Ce principe est répété jusqu'à placer le minimum en première position.

Exemple : pour trier la liste `[7, 9, 4, 3]` :

— première étape : `7` et `9` ne sont pas échangés, puis `9` et `4` sont échangés, puis `9` et `3` sont échangés, la liste est alors `[7, 4, 3, 9]`
— deuxième étape : `7` et `4` sont échangés, puis `7` et `3` sont échangés, la liste est alors `[4, 3, 7, 9]`
— troisième étape : `4` et `3` sont échangés, la liste est alors `[3, 4, 7, 9]`

Compléter le code Python ci-dessous qui implémente la fonction `tri_bulles`.

```
1  def tri_bulles(T):
2  ''' Renvoie le tableau T trie par ordre croissant '''
3      n = len(T)
4      for i in range(...,...,-1):
5          for j in range(i):
6              if T[j] > T[...]:
7                  ... = T[j]
8                  T[j] = T[...]
9                  T[j+1] = temp
10     return T
```

Pour tester :

```
>>> tri_bulles([])
[]
>>> tri_bulles([9, 3, 7, 2, 3, 1, 6])
[1, 2, 3, 3, 6, 7, 9]
>>> tri_bulles([9, 7, 4, 3])
[3, 4, 7, 9]
```