TNSI	Algorithmique Diviser pour régner	Cours – La stratégie « **Diviser pour régner** »

Objectifs :

- ⇨ Découvrir une nouvelle stratégie algorithmique
- ⇨ Comprendre l'intérêt de l'approche « diviser pour régner »
- ⇨ Voir quelques exemples d'application

D'après l'album La Zizanie de Goscinny et Uderzo

« **Diviser pour régner** :
Diviser les forces de l'adversaire pour le contrôler, le manipuler, le soumettre.
Du latin "*divide ut regnes*" attribuée à Philippe II de Macédoine au IV° siècle avant JC »

I - Principe

L'idée directrice du principe[1] de programmation « Diviser pour régner » ou DPR (« *Divide and conquer* » en anglais) est de simplifier un problème complexe à résoudre en le divisant en sous-problèmes qui soient suffisamment simple à traiter puis en recombinant les solutions des sous-problèmes afin de reconstituer la solution du problème original.

Ainsi les algorithmes de type « Diviser pour régner » procèdent en trois étapes :

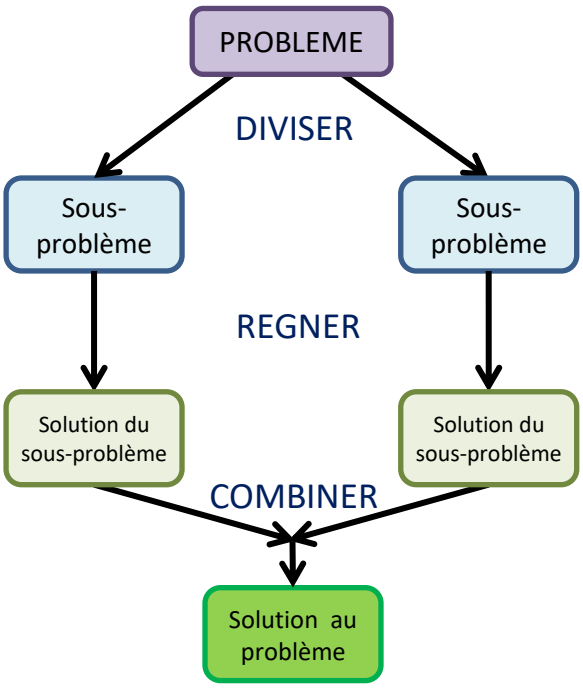
⇨ **DIVISER** :

⇨ **RÉGNER** :

⇨ **COMBINER** :

Algorithmiquement, cela donne :

```
def diviser_pour_regner(a, i, j):
  if est_petit(a, i, j):
    return solution(a, i, j)
  else:
    milieu = diviser(a, i, j)
    p1 = diviser_pour_regner(a,i,milieu)
    p2 = diviser_pour_regner(a,milieu+1,j)
    s = combiner(p1, p2)
  return s
```



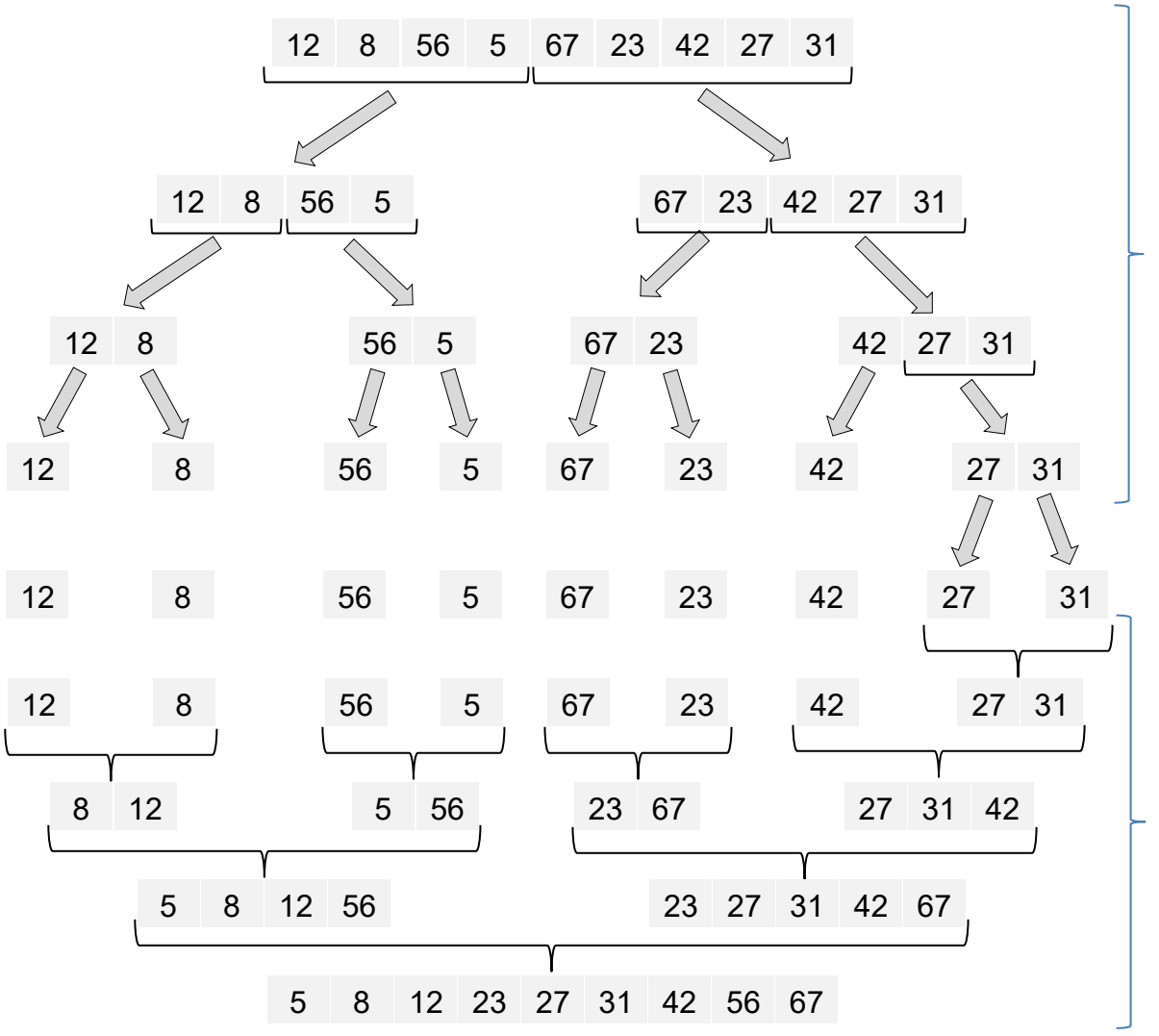
[1] On parle de *principe* de programmation, de *stratégie* ou de *méthode* de résolution ou encore de *paradigme*.

II – Un exemple classique : le tri fusion

Lorsqu'on veut trier une grande quantité d'information et qu'on est nombreux, on peut vouloir se répartir la tâche et que chacun tri une partie des données dans son coin pour qu'on ait ensuite plus qu'à combiner les données triées par chaque personne afin d'obtenir le résultat final trié.
Cette méthode est effectivement bien plus rapide et on peut se dire à première vue que cela est dû à la parallélisation du travail de trie. En fait nous allons voir qu'une personne seule qui procède ainsi en divisant la tâche de tri en plusieurs petites parties faciles à trier gagne également du temps par rapport à une méthode de trie classique comme le tri par insertion ou par sélection.

Le principe du tri fusion suit exactement le schéma « Diviser pour régner » :

1. On divise récursivement par 2 la liste à trier jusqu'à ce que chaque sous-liste ne contienne qu'un seul élément.	DIVISER
2. Vu qu'il n'y a qu'un seul élément dans chaque sous-liste, celle-ci est donc directement triée.	REGNER
3. On fusionne ensuite les listes deux à deux en s'arrangeant pour que la fusion des deux listes triées soit elle-même triée. On procède ainsi jusqu'à ce qu'on ait fusionné toutes les sous-listes. On obtient alors la liste complète triée.	COMBINER



Application 1 :

1) Ecrire en pseudo-code l'algorithme d'une fonction `tri_fusion(t)` qui renvoie une version triée du tableau `t`. Cette fonction pourra s'appuyer sur une fonction `combiner` dont on définira la signature.
2) Ecrire en pseudo-code l'algorithme de la fonction `combiner` utilisée précédemment.
3) Ecrire une version non récursive de l'algorithme précédent (ou récursive si vous avez écrit un algorithme récursif à la question précédente).

Application 2 :

1) Ecrire un programme python implémentant le tri fusion (version récursive) et le tester.
2) Quelle est la complexité du tri fusion ? Comment pourrait-on le vérifier expérimentalement ?

Pour mesurer le temps d'exécution d'une portion de code, on peut utiliser le module `timeit` et sa fonction `timeit` :

```
import timeit
temps = timeit.timeit('t.sort()', 't = [i*2 for i in range(10)]', number=25,
                      globals=globals())
```

Le premier argument est une chaîne contenant le code à exécuter, le deuxième argument (optionnel) est une fonction préparatoire qui est exécutée une seule fois avant de mesurer le temps d'exécution du code, l'argument `number` indique le nombre de répétition du code et `globals` indique l'espace de nommage dans lequel on doit exécuter le code.
La fonction renvoi le temps d'exécution (en seconde) des `number` exécution du code.
Ce module permet une mesure assez précise du temps d'exécution d'une portion de code.

3) Utiliser le module `timeit` pour mesurer le temps d'exécution de `tri_fusion` sur des tableaux de taille 10, 100, 1000, 10000 et 1000000 éléments. Quel problème rencontre-t-on pour des tableaux de grande taille ? Comment le solutionner ?
4) Comment montrer à partir des résultats bruts de mesure de temps que la complexité temporelle du tri fusion est bien de n·log(n) ? Faire le nécessaire pour que votre programme mette en évidence cette complexité.

III – Intérêts et limites de la stratégie « Diviser pour régner »

1) Avantages et inconvénients

La stratégie « diviser pour régner » permet de :

⇒ ..... de problème en se limitant à des sous-cas simples à gérer.
⇒ ..... Sur des machines multi-processeurs (ou multi-thread), le gain peut alors être très important. Ceci n'est toutefois possible que si les différents threads peuvent fonctionner de manière indépendante (c'est le cas pour le tri fusion).
⇒ Bien souvent d'..... (spatiale et/ou temporelle) que l'algorithme naïf.

En revanche :

⇒ Il y a des ..... (constitution des sous-problèmes dans la phase « diviser ») qui peuvent être non-négligeables.
⇒ Elle peut être ..... (en particulier la phase de recombinaison peut être parfois délicate à mettre en œuvre).
⇒ Elle ..... par rapport à l'algorithme naïf et peut même dans certains cas la dégrader

2) Exemple de la recherche de minimum

Dans cette partie on va chercher à écrire un algorithme de recherche de minimum dans un tableau utilisant la méthode DPR.

a. Algorithme naïf

Application 3 :

1) Ecrire une fonction python `minimum(t)` qui renvoie la valeur minimale se trouvant dans le tableau `t` en utilisant un algorithme très simple.
2) Quelle est la complexité de cet algorithme ?

b. Amélioration avec la méthode DPR

Application 4 :

1) Ecrire en pseudo-code l'algorithme d'une fonction `minimum_DPR(t)` qui renvoie la valeur minimale se trouvant dans le tableau `t` en utilisant un algorithme du type « Diviser pour régner ».
2) Programmer cette fonction en python et valider son fonctionnement.
3) Quelle est la complexité de la fonction ? Vérifier à l'aide du module `timeit`.
4) Comparer le temps d'exécution des fonctions `minimum` et `minimum_DPR`. Que remarquez-vous ?

Références :

Vidéo Lumni : https://www.lumni.fr/video/la-methode-laquo-diviser-pour-regner-raquo
En anglais : https://www.geeksforgeeks.org/divide-and-conquer-algorithm-introduction/

Exercices – **Diviser pour régner**

Exercice 1 : Deuxième plus grand élément

On s'intéresse dans cet exercice à la complexité dans le pire des cas et en nombre de comparaisons des algorithmes.

1) Pour rechercher le plus grand et deuxième plus grand élément de n entiers, donner un algorithme naïf et sa complexité.

Pour améliorer les performances, on se propose d'envisager la solution consistant à calculer le maximum suivant le principe d'un tournoi (tournoi de tennis par exemple). Pour simplifier, on se place dans le cas où il y a $n = 2^k$ nombres qui s'affrontent dans le tournoi.

2) Comment retrouve-t-on, une fois le tournoi terminé, le deuxième plus grand ? Quelle est la complexité de l'algorithme ?
3) Ecrire le pseudo-code de cet algorithme. Pour les plus avancés, vous pouvez essayer de le programmer en python.

Exercice 2 : Goûts musicaux

Un site internet cherche à regrouper ses membres en fonction des goûts musicaux de chacun. Pour cela, chaque membre doit classer par ordre de préférence une liste d'artistes. On dit que deux membres, Arthur et Béatrice, ont des goûts musicaux proches lorsque qu'il y a peu d'inversions dans leurs classements : une inversion est une paire d'artiste {L, M} telle qu'Arthur préfère L à M et Béatrice préfère M à L. On cherche donc à compter le nombre d'inversion dans les classements d'Arthur et Béatrice.

1) Compter le nombre d'inversion les classements suivants :

   Arthur : Britney Spears, Lady Gaga, Michael Jackson, Madonna, Céline Dion ;
   Béatrice : Lady Gaga, Madonna, Britney Spears, Michael Jackson, Céline Dion.

2) Proposer un algorithme naïf qui résout le problème. Quelle est sa complexité ?

On cherche maintenant à améliorer l'algorithme précédent en utilisant la stratégie DPR. Pour cela, on coupe le classement de chaque membre en deux sous-classements de même taille, celui des artistes préférés (classement supérieur) et celui des autres artistes (classement inférieur). On compte alors les inversions (L, M) qui peuvent être de deux types : soit L et M apparaissent dans le même sous-classement de Béatrice, soit L et M apparaissent dans deux différents sous-classements de Béatrice (inversions mixtes).

3) On suppose que les deux sous-classements de Béatrice sont triés en fonction du classement d'Arthur (Si L et M apparaissent dans le même sous-classement de Béatrice, alors ils apparaissent dans le même ordre que dans le classement d'Arthur). Montrer qu'on peut alors compter les inversions mixtes en temps linéaire.
4) Donner un algorithme de type Diviser-Pour-Régner qui fonctionne en temps $O(n \cdot \log(n))$.

Exercice 3 : Sous-séquence maximale

Le but de cet exercice est de s'intéresser au problème de la sous-séquence maximale d'une liste de nombre et des différents algorithmes qui ont été proposés pour le résoudre. Le problème s'énonce ainsi : on dispose d'un tableau T de nombres et on cherche la plus grande somme de nombres consécutifs dans le tableau. Par exemple pour le tableau [-1, 2, 0, 5, -6, 7, -2, 1], la plus grande somme possible est 8 qu'on obtient entre les indices 1 et 5. Si notre tableau ne contient que des nombres négatifs, la plus grande somme ne prend aucun élément et est donc égale à 0. Ce problème a été posé par Ulf Grenander vers la fin des années 70.

Pour un tableau t et deux indices $i \leq j$, on définit $S(i,j) = \sum_i^j t[k]$. Pour tout $i > j$ on a $S(i,j) = 0$.

1) Proposer un algorithme de type force brute qui teste toutes les paires d'indices possibles. Quelle est sa complexité ?
2) Proposer une amélioration pour arriver à une complexité quadratique.

Michael Shamos propose alors un algorithme de type diviser pour régner pour résoudre ce problème. Pour ce faire, il remarque que lorsqu'on divise un tableau en deux, la sous-séquence maximale se trouve soit dans la partie gauche, soit dans la partie droite, soit à cheval sur la séparation.

3) Écrire un algorithme de type diviser pour régner pour le problème de la sous-séquence maximale et prouver sa correction.
4) Quelle est sa complexité ?

Enfin, Jay Kadane entend parler du problème et écrit alors un algorithme de programmation dynamique.

5) On suppose T[0] < 0. Que peut-on dire sur la sous-séquence maximale ?
6) Soit j > 1 tel que pour tout i < j on a S(0,i) ≥ 0 et S(0,j) < 0. Que peut-on dire de la position de la sous-séquence maximale ?
7) En déduire un algorithme pour résoudre notre problème et prouver sa correction.
8) Calculer sa complexité et prouver qu'elle est optimale.