REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
Ministère de l'enseignement supérieur et de la recherche scientifique
Université Larbi Ben M'Hidi Oum El Bouaghi

Cours Méthodes algorithmiques

1ere année master AD

# Chapitre 2

# Diviser pour Régner

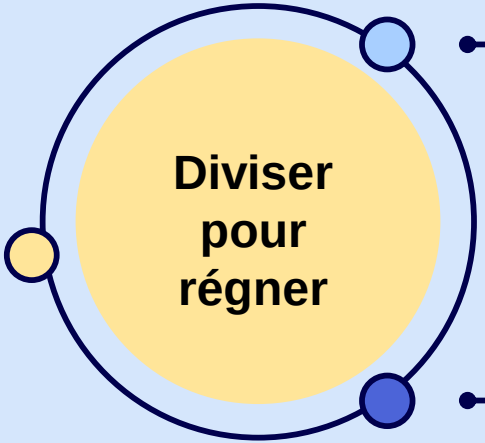


# Introduction au Paradigme "Diviser pour Régner"



En décomposant un problème complexe en sous-problèmes plus simples, on parvient à maîtriser chaque partie pour mieux dominer l'ensemble

# Introduction au Paradigme "Diviser pour Régner"




**Optimise** les performances

**Concept clé en algorithmique** utilisé pour résoudre des problèmes complexes.

Une méthode cruciale lorsque les problèmes sont **volumineux**.

# Diviser pour régner

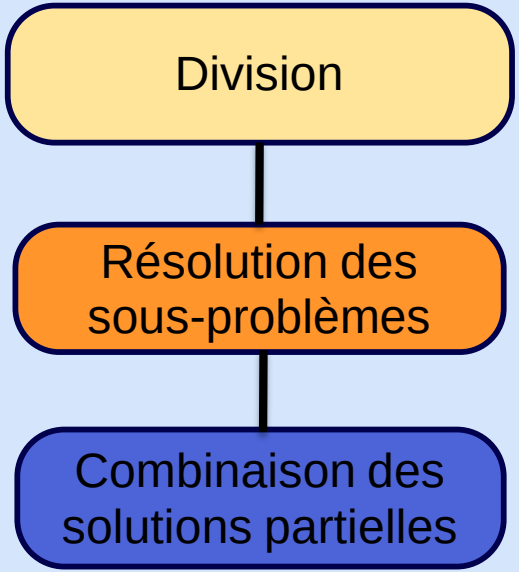
## Définition

Le paradigme **"Diviser pour régner"** est une technique algorithmique qui consiste à:

- **Diviser** un problème complexe en sous-problèmes plus simples
- Résoudre chaque sous-problème de manière indépendante (souvent de façon **récursive**)
- **Combiner** les solutions pour obtenir la solution du problème initial.



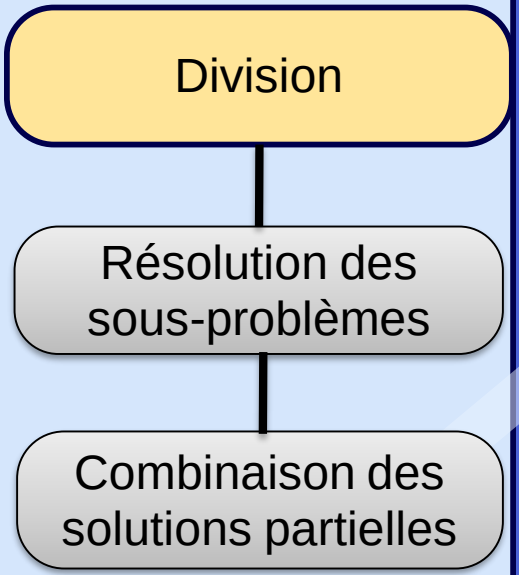
# Principe de Diviser pour régner



# Principe de Diviser pour régner

- Dans cette première étape, le problème de départ est divisé en sous-problèmes plus petits et indépendants, qui sont de même nature que le problème initial.

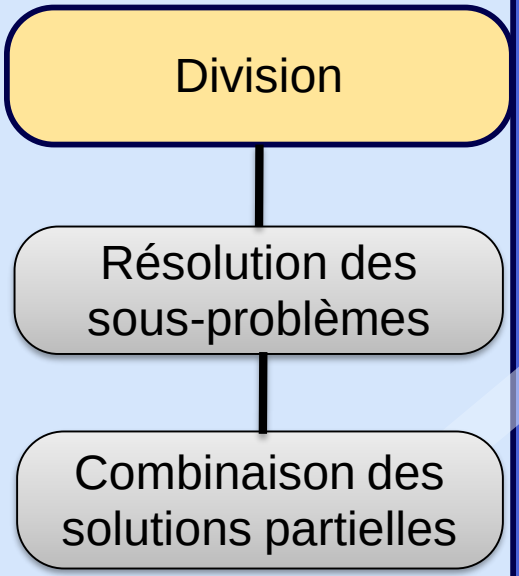
- La division doit être effectuée de manière à ce que les sous-problèmes soient plus faciles ou plus rapides à résoudre que le problème global.



# Principe de Diviser pour régner

**Nature de la division** :

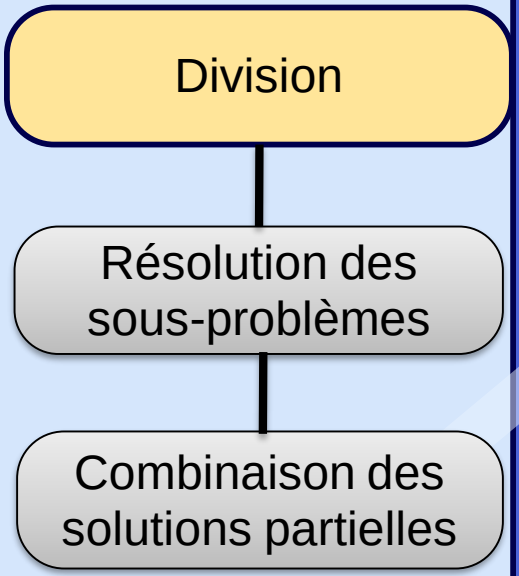
Selon la structure du problème, cette division peut se faire de manière simple en deux parties, ou en plusieurs sous-problèmes



# Principe de Diviser pour régner

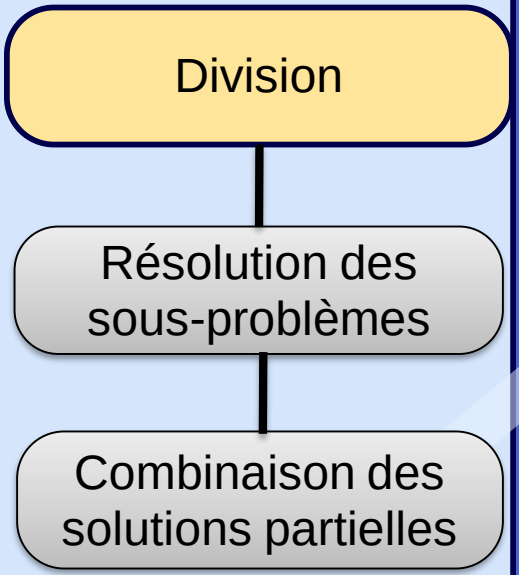
**Objectif de la division** :

- L'objectif est de réduire la complexité du problème à chaque étape
- en décomposant celui-ci en sous-problèmes plus petits jusqu'à atteindre un point où la solution devient évidente ou triviale



# Principe de Diviser pour régner

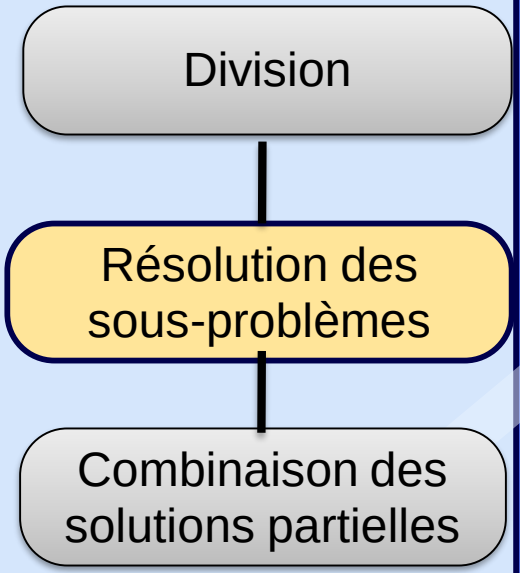
- Après avoir divisé le problème, chaque sous-problème est résolu de manière récursive.
- Si le sous-problème est encore trop grand pour être résolu directement, il est à son tour divisé suivant le même schéma.



# Principe de Diviser pour régner

**Résolution récursive** :

chaque sous-problème est résolu en appliquant la même stratégie de division, résolution et combinaison jusqu'à atteindre une condition de terminaison appelée : cas de base

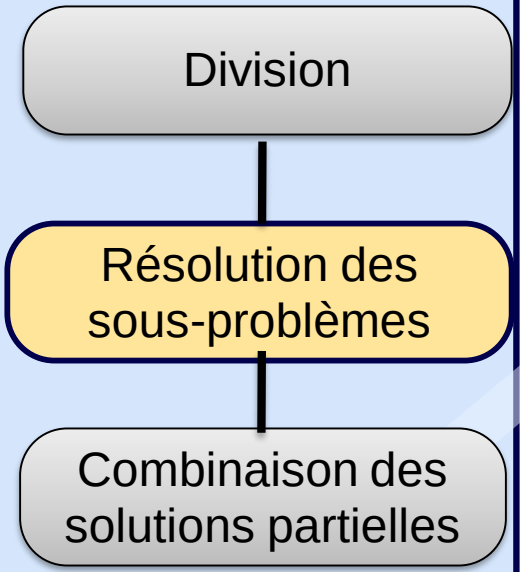


# Principe de Diviser pour régner

**Cas de base** :

- La récursion continue jusqu'à ce que les sous-problèmes soient suffisamment petits pour être résolus directement sans besoin de division supplémentaire.

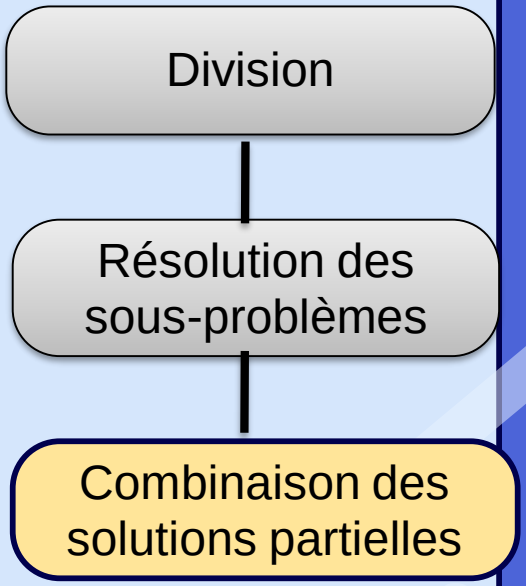
- Le cas de base est généralement trivial, comme un tableau de taille 1 dans un algorithme de tri ou un cas où la division ne produit plus de sous-problèmes significatifs.



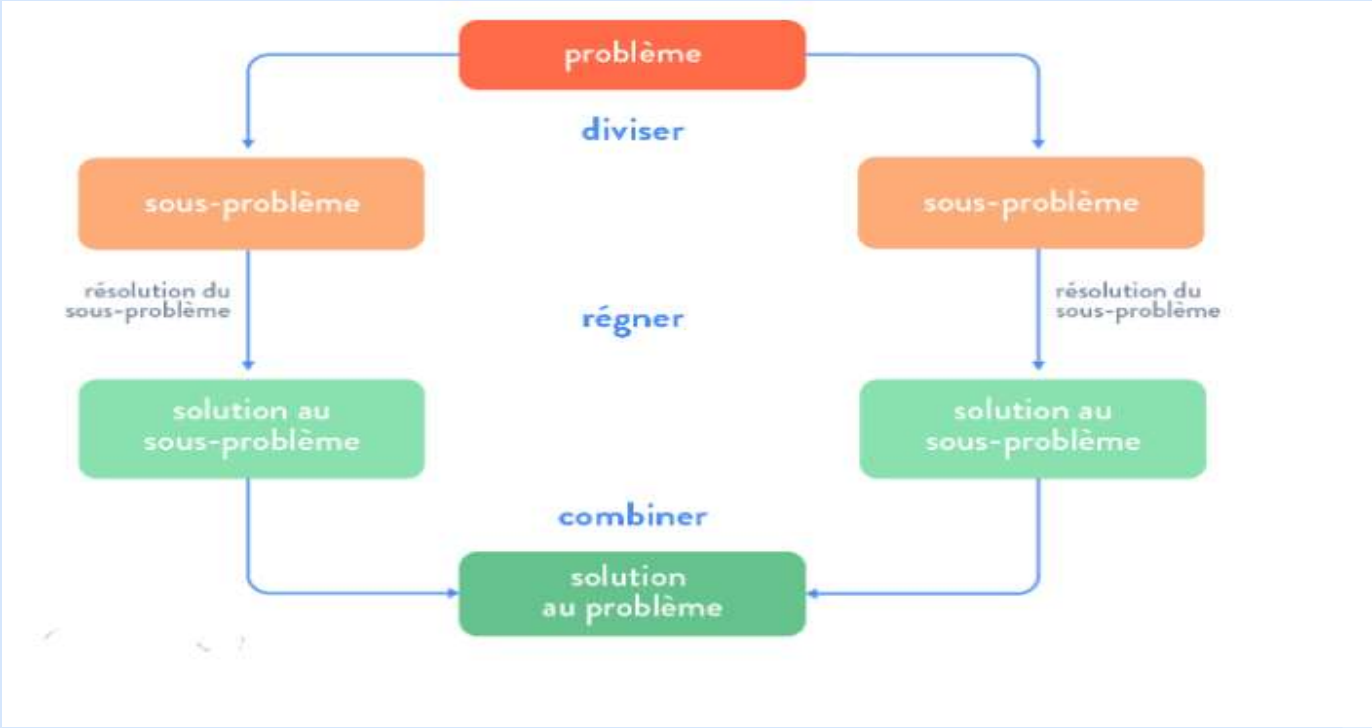
# Principe de Diviser pour régner

- Une fois les sous-problèmes résolus, les résultats obtenus sont combinés pour former la solution du problème initial.

- Cette étape est cruciale, car elle doit se faire efficacement pour ne pas annuler les gains obtenus lors des étapes de division et de résolution.



# Strategie de Diviser pour régner



# Stategie de Diviser pour régner

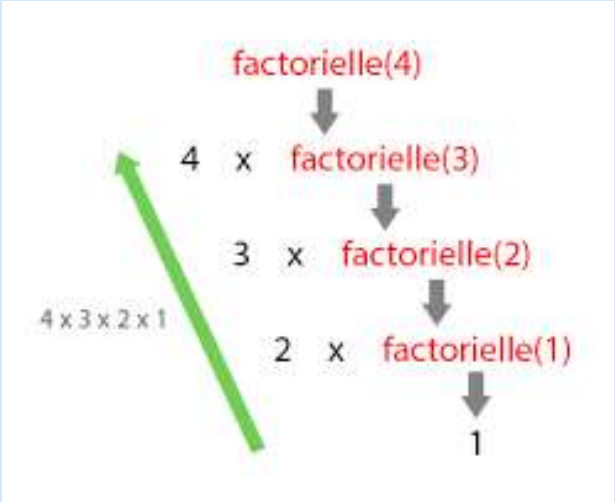
- La première stratégie est la récursion sur les données : où les données sont directement décomposées en un certain nombre de partitions.
- Cela donne lieu à des sous-problèmes qui sont résolus récursivement à l'aide de la même fonction.
- À la fin, les résultats obtenus sont combinés afin de calculer la solution complète.

# Stategie de Diviser pour régner

- La deuxième stratégie est la récursion sur les résultats, qui effectue d'abord un prétraitement avant de couper les données en partitions.
- Ensuite, elle résout récursivement les sous-problèmes résultants en utilisant la même fonction.
- Enfin, elle combine les résultats obtenus afin de calculer la solution complète..

# Stategie de Diviser pour régner

- Les exemples incluent des fonctions de calcul de la factorielle ou de la suite de Fibonacci, où chaque appel récursif retourne un résultat basé sur les résultats des appels récursifs précédents.



# La récursivité

- La récursivité est une technique de programmation et un concept algorithmique où une fonction s'appelle elle-même pour résoudre un problème.

- Elle est souvent utilisée pour simplifier la conception et l’implémentation d'algorithmes, surtout pour les problèmes qui peuvent être divisés en sous-problèmes similaires.



# La récursivité

**Concepts de Base de la Récursivité**

- **Définition :** Une fonction est dite récursive si elle s'appelle elle-même directement ou indirectement. La récursivité est un outil puissant pour résoudre des problèmes complexes en les décomposant en parties plus simples.

- **Cas de Base :** Chaque fonction récursive doit avoir un cas de base, qui est une condition dans laquelle la fonction renvoie une valeur sans appeler elle-même. Cela empêche la récursion de continuer indéfiniment.

- **Appels Récursifs :** Si le cas de base n'est pas atteint, la fonction s'appelle elle-même avec des paramètres modifiés. Cela doit réduire progressivement la taille du problème, menant finalement au cas de base.

# La récursivité

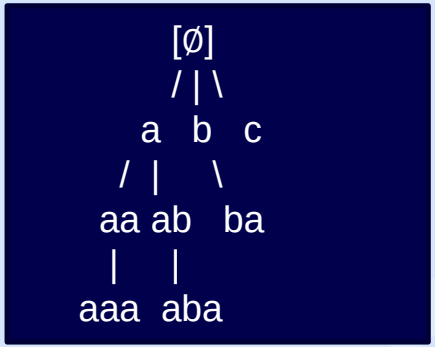
**Structure d'une Fonction Récursive**
Une fonction récursive typique comprend les éléments suivants :

```
def fonction_recursive(paramètres):
    if condition_de_base:
        # Cas de base
        return valeur
    else:
        # Appel récursif
        return
fonction_recursive(nouveaux_paramètres)
```

# Représentation du problème : l'arbre d'exploration

Le processus de backtracking peut être visualisé comme un arbre d'exploration :

- Chaque nœud représente une solution partielle.
- Chaque branche représente un choix possible.
- Une feuille représente soit :
  - ✓ une solution complète valide, soit
  - ✓ une impasse :solution invalide qu'on abandonne



# Composants essentiels du Backtracking

Pour concevoir un algorithme de retour sur trace, il faut identifier trois éléments clés :

| Élément | Description | Exemple (N-reines) |
| --- | --- | --- |
| Choix | Ensemble des décisions possibles à un instant donné. | Position possible d'une reine sur la ligne courante. |
| Contraintes | Conditions que chaque solution partielle doit respecter. | Aucune reine ne doit être dans la même colonne ou diagonale. |
| Solution complète | État dans lequel tous les choix ont été faits et respectent les contraintes. | Les N reines sont placées sans conflit. |

# Avantages et limites du principe

Les avantages

- Général et modulaire : applicable à de nombreux problèmes combinatoires.
- Simple à implémenter avec la récursion.
- Peut être optimisé grâce à des tests de contraintes précoces

Les limites

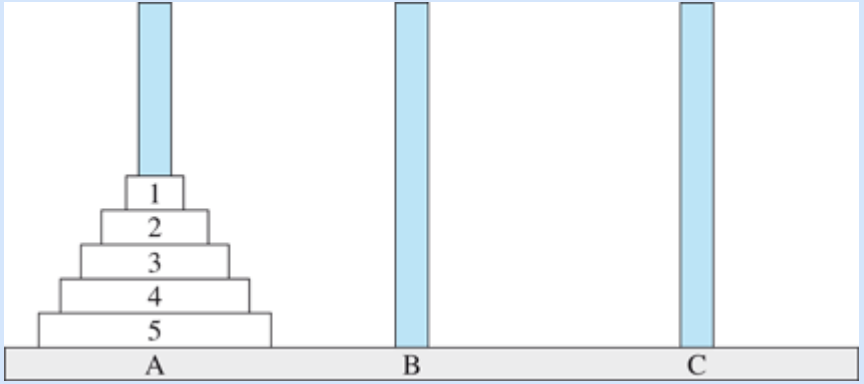
- Complexité exponentielle dans le pire des cas
- Peut devenir lent pour de très grands espaces de recherche sans élagage

# Exemples Illustratifs

01

## Le problème de Hanoi

- Le problème de Hanoi est un célèbre casse-tête mathématique et algorithmique qui met en jeu trois tours et un certain nombre de disques de tailles différentes.



# Exemples Illustratifs

01

## Le problème de Hanoi

- **Objectif :** Déplacer tous les disques d'une tour source (A) vers une tour cible (C), en respectant **certaines règles**.

- **Tours :** Trois tours sont disponibles :
  Tour A (source)
  Tour B (auxiliaire)
  Tour C (cible)

  **Disques :** Les disques sont empilés sur la tour source, avec le plus grand disque au bas et le plus petit au sommet.

# Exemples Illustratifs

01

## Le problème de Hanoi

**Règles**

- **Un disque à la fois :** On ne peut déplacer qu'un seul disque à la fois.
- **Règle de taille :** Un disque ne peut être placé que sur un disque plus grand ou sur une tour vide.

# Exemples Illustratifs

01

## Le problème de Hanoi

**Stratégie de Résolution**

Le problème de Hanoi peut être résolu de manière **récursive**:

- **Déplacer n-1 disques :** Déplacez les n-1 disques du sommet de la tour A vers la tour B, en utilisant la tour C comme auxiliaire.
- **Déplacer le disque restant :** Déplacez le disque restant (le plus grand) de la tour A vers la tour C.
- **Déplacer les n-1 disques :** Déplacez les n-1 disques de la tour B vers la tour C, en utilisant la tour A comme auxiliaire.

# Le problème de Hanoi

| Mouvement | Position |
| --- | --- |
| Position initiale |   |
| 1 : T1 vers T3 |   |
| 2 : T1 vers T2 |   |
| 3 : T3 vers T2 |   |
| 4 : T1 vers T3 |   |
| 5 : T2 vers T1 |   |
| 6: T2 vers T3 |   |
| 7 : T1 vers T3 |   |

# Exemples Illustratifs

01

## Le problème de Hanoi

```
function tour_de_hanoi(n, source, destination, intermédiaire):
    if n == 1:
        print("Déplacez le disque 1 de", source, "à", destination)
    else:
        # Étape 1 : Déplacer les n-1 disques de la source vers l'intermédiaire
        tour_de_hanoi(n - 1, source, intermédiaire, destination)

        # Étape 2 : Déplacer le disque n de la source vers la destination
        print("Déplacez le disque", n, "de", source, "à", destination)

        # Étape 3 : Déplacer les n-1 disques de l'intermédiaire vers la destination
        tour_de_hanoi(n - 1, intermédiaire, destination, source)
```

# Exemples Illustratifs

02

## Tri par Fusion (Merge Sort)

- Le tri par fusion est un excellent exemple de la technique "diviser pour régner". Il fonctionne de la manière suivante :

  **Diviser** : Divisez le tableau en deux moitiés.
  **Récursivité** : Appliquez le tri par fusion de manière récursive à chaque moitié jusqu'à ce que chaque sous-tableau contienne un seul élément (un tableau de taille un est déjà trié)
  **Combiner** : Fusionnez les deux sous-tableaux triés pour former un tableau trié.

# Exemples Illustratifs

02

## Tri par Fusion (Merge Sort)

```
tri_fusion(tab) :
    si longueur(tab) > 1 :
        milieu = longueur(tab) // 2
        gauche, droite = tab[0:milieu-1], tab[milieu:]
        tri_fusion(gauche)
        tri_fusion(droite)
        fusion(tab, gauche, droite)

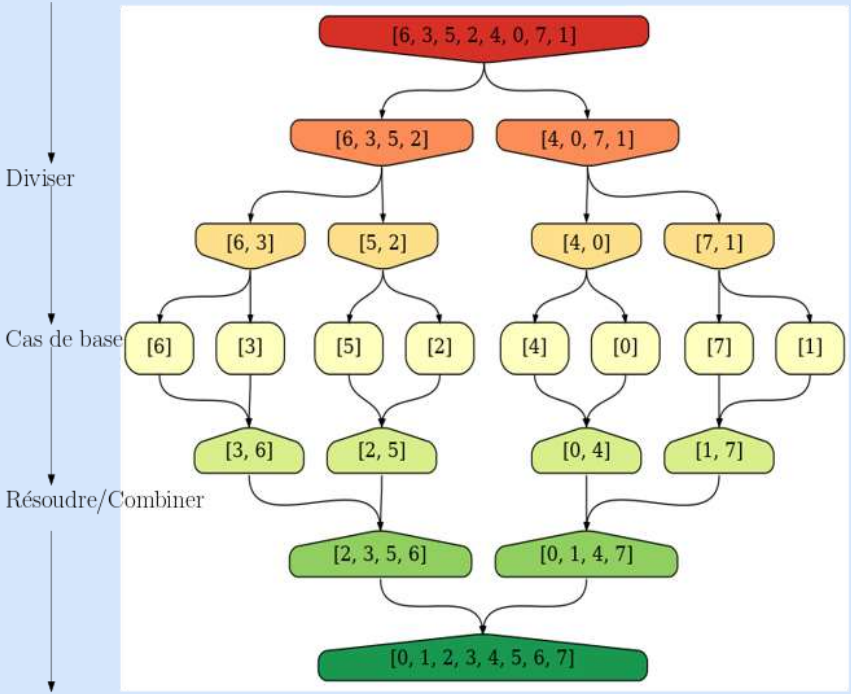
fusion(tab, g, d) :
    i, j, k = 0
    tant que i < longueur(g) et j < longueur(d) :
        si g[i] < d[j] : tab[k] = g[i]; i += 1
        sinon : tab[k] = d[j]; j += 1
        k += 1
Tant que i< longueur(g) faire
tab[k] = g[i]; k += 1; i += 1

Tant que j< longueur(d) faire
tab[k] = d[j]; k += 1; j += 1
```

02

# Exemples Illustratifs

## Tri par Fusion (Merge Sort)



# Exemples Illustratifs

03

## Recherche Binaire

- La recherche binaire est un algorithme efficace utilisé pour trouver un élément dans un tableau trié. Elle fonctionne en divisant de manière répétée l'intervalle de recherche en deux, réduisant ainsi le nombre d'éléments à examiner à chaque étape.

- Le tableau doit être trié au préalable.

- Si le tableau n'est pas trié, la recherche binaire ne fonctionnera pas correctement.

# Exemples Illustratifs

03

## Recherche Binaire

```
Algorithme Recherche_Binaire(A, debut, fin, x)
    A est un tableau trié d'entiers distincts
    debut et fin sont les indices de la portion considérée
    x est la valeur à rechercher
    Si debut > fin alors
        retourner -1  → Élément non trouvé
    fin si
    milieu ← (debut + fin) // 2
    Si A[milieu] = x alors
        retourner milieu
    Sinon si A[milieu] > x alors
        retourner Recherche_Binaire(A, debut, milieu - 1, x) → Cherche dans la moitié gauche

    Sinon
        retourner Recherche_Binaire(A, milieu + 1, fin, x) Cherche dans la moitié droite
    fin si
Fin Algorithme
```

# Exemples Illustratifs

03

## Recherche Binaire




Chercher le nombre 7

# Exemples Illustratifs

03

## Recherche Binaire




Chercher le nombre 10

# Exemples Illustratifs

04

## Trouver le maximum dans un tableau

- **Problème :** Trouver le plus grand élément dans un tableau.

**Algorithme :** Divise le tableau en deux sous-tableaux, trouve le maximum dans chaque sous-tableau, puis retourne le maximum des deux.

# Exemples Illustratifs

04

## Trouver le maximum dans un tableau

```
Algorithme Max_Diviser_Pour_Regner(A, debut, fin)
    A est le tableau, debut et fin les indices considérés
    Si debut = fin alors
        retourner A[debut] →Cas de base : un seul élément
fin si

    milieu ← (debut + fin) // 2

    max_gauche ← Max_Diviser_Pour_Regner(A, debut, milieu) Maximum dans la moitié gauche
    max_droite ← Max_Diviser_Pour_Regner(A, milieu + 1, fin) Maximum dans la moitié droite

    Si max_gauche > max_droite alors
        retourner max_gauche
    Sinon
        retourner max_droite
    fin si
Fin Algorithme
```

# Exemples Illustratifs

04

## Trouver le maximum dans un tableau

