La méthode « Diviser pour Régner »

Plan

- Présentation de la méthode
- Exemple 1 : Les tours de Hanoï
- Exemple 2 : Le tri-fusion

Méthodologie de résolution de problème

On veut résoudre le problème **A**.

Si on sait :

1. transformer le problème **A** en un problème **B**;
2. résoudre le problème **B**;
3. transformer la solution du problème **B** en une solution du problème **A**,

alors on sait résoudre le problème **A**.

Méthode « Diviser pour Régner »

Situation

- On veut résoudre un type de problème pour une taille n ;
- On peut le faire en se basant sur la résolution du même problème pour de plus petits tailles.

Méthode « Diviser pour Régner »

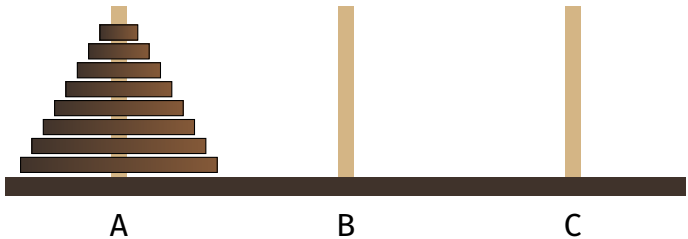
Situation

- On veut résoudre un type de problème pour une taille n ;
- On peut le faire en se basant sur la résolution du même problème pour de plus petits tailles.

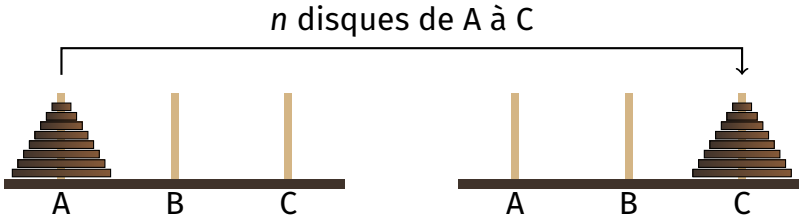
Résolution en 3 étapes

1. **Diviser** pour faire apparaître les sous-problèmes à résoudre ;
2. **Régner** pour résoudre effectivement les sous-problèmes ;
3. **Combiner** pour obtenir une solution du problème initial.

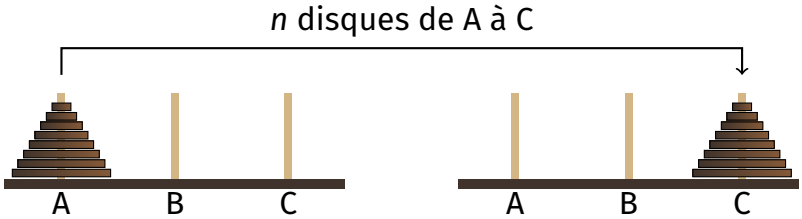
Les tours de Hanoï



Les tours de Hanoï

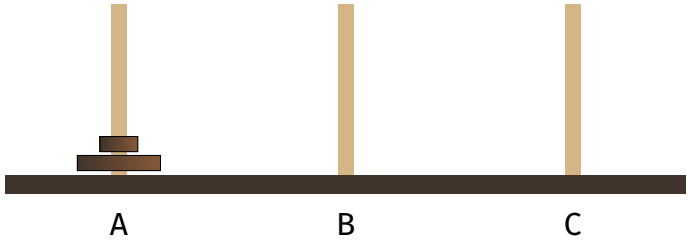


Les tours de Hanoï

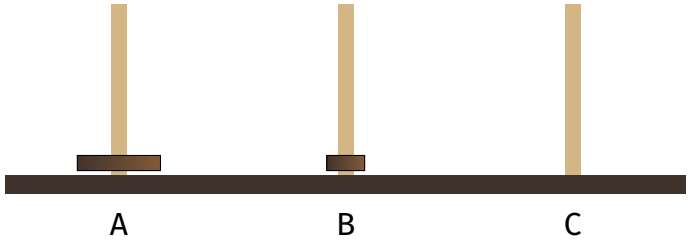


- On ne peut déplacer qu'un disque à la fois ;
- On ne peut poser un disque que sur un disque plus grand ou une tige vide.

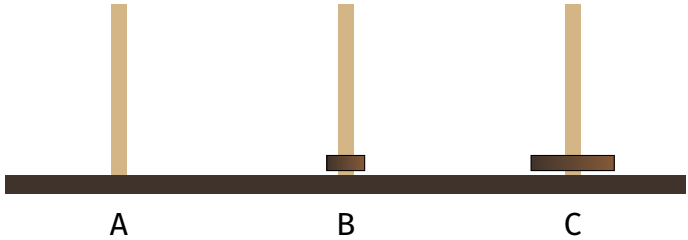
Exemples



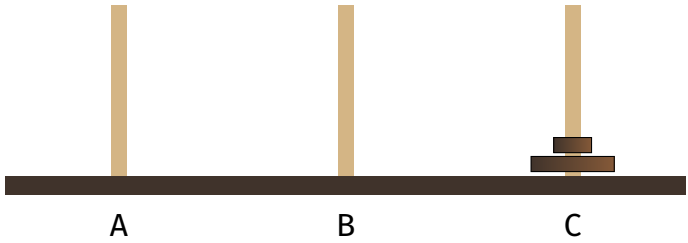
Exemples



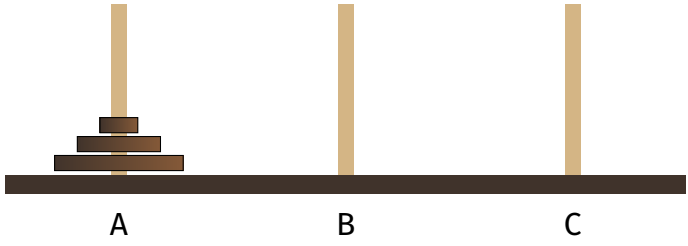
Exemples



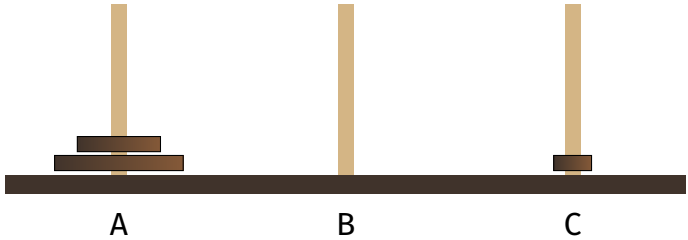
Exemples



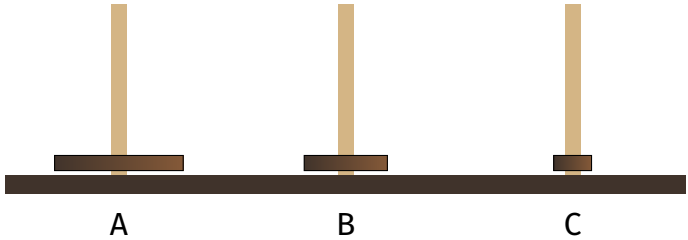
Exemples



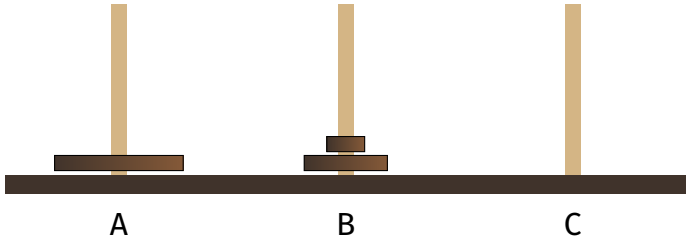
Exemples



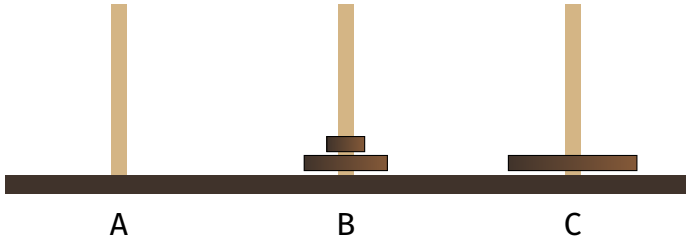
Exemples



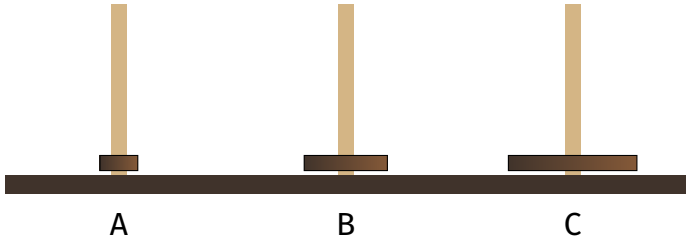
Exemples



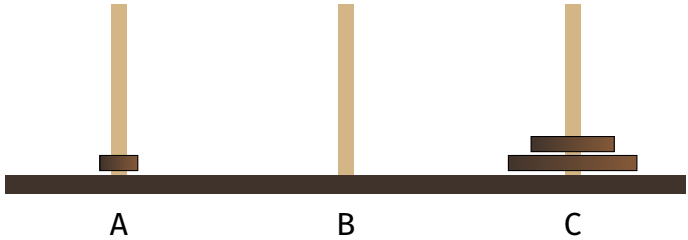
Exemples



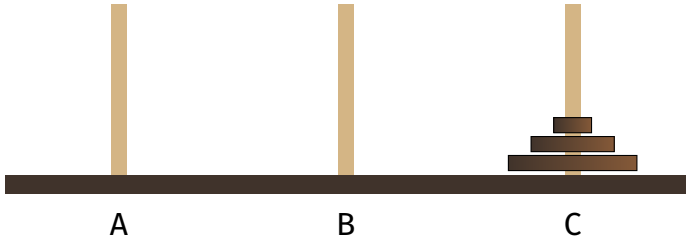
Exemples



Exemples



Exemples

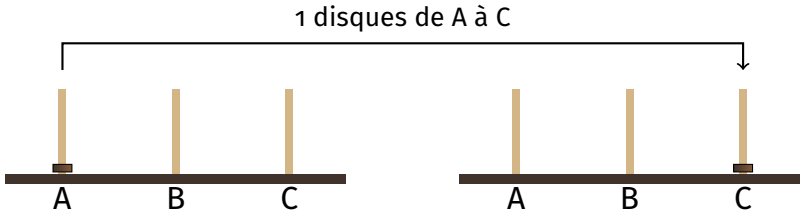


Questions pour diviser et régner

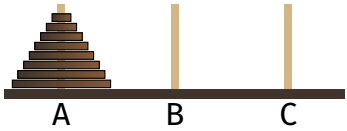
- **Diviser** Quels problèmes « plus petits » considérer ?
- **Cas de base** Comment résoudre les problèmes les plus petits ?
- **Combiner** À partir d'une solution du problème plus petit, comment obtenir une solution du problème plus grand ?

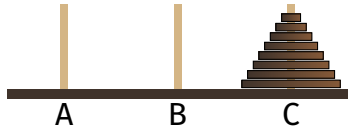
Cas de base

Cas de base

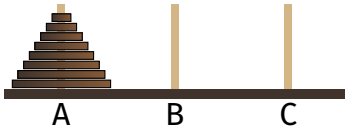


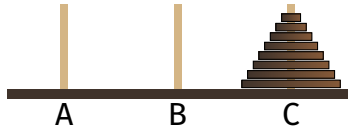
Diviser et régner



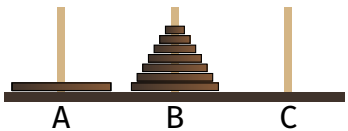


Diviser et régner

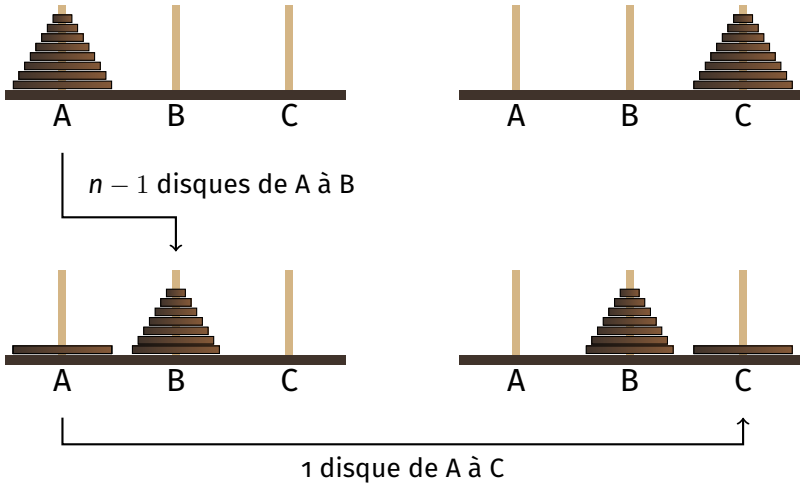




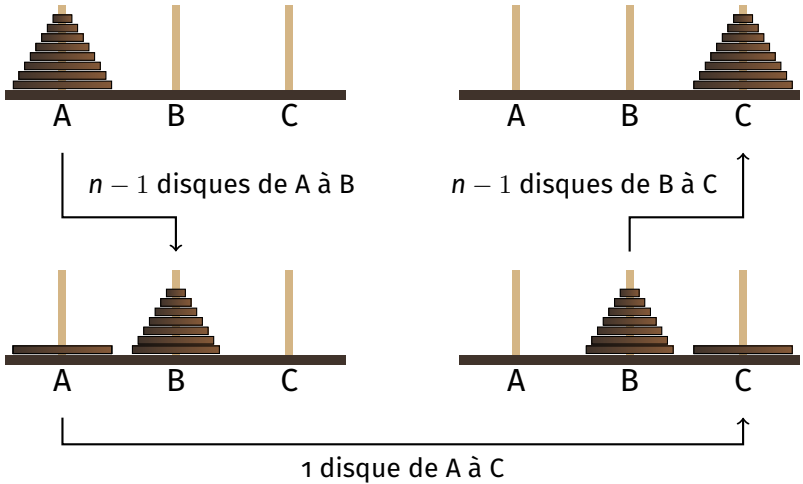
$n-1$ disques de A à B



Diviser et régner



Diviser et régner



Les tours de Hanoï, une solution

```
def hanoï(n, départ, arrivée, troisième):
```

Les tours de Hanoï, une solution

```
def hanoï(n, départ, arrivée, troisième):
    if n == 1: # Cas de base
        print("Disque", n, ":", départ, "->", arrivée)
```

Les tours de Hanoï, une solution

```
def hanoï(n, départ, arrivée, troisième):
    if n == 1: # Cas de base
        print("Disque", n, ":", départ, "->", arrivée)
    elif n >= 2: # Diviser et régner
```

Les tours de Hanoï, une solution

```
def hanoï(n, départ, arrivée, troisième):
    if n == 1: # Cas de base
        print("Disque", n, ":", départ, "->", arrivée)
    elif n >= 2: # Diviser et régner
        hanoï(n - 1, départ, troisième, arrivée)
```

Les tours de Hanoï, une solution

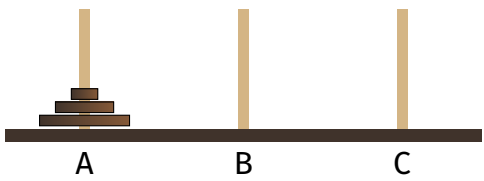
```
def hanoï(n, départ, arrivée, troisième):
    if n == 1: # Cas de base
        print("Disque", n, ":", départ, "->", arrivée)
    elif n >= 2: # Diviser et régner
        hanoï(n - 1, départ, troisième, arrivée)
        print("Disque", n, ":", départ, "->", arrivée)
```

Les tours de Hanoï, une solution

```
def hanoï(n, départ, arrivée, troisième):
    if n == 1: # Cas de base
        print("Disque", n, ":", départ, "->", arrivée)
    elif n >= 2: # Diviser et régner
        hanoï(n - 1, départ, troisième, arrivée)
        print("Disque", n, ":", départ, "->", arrivée)
        hanoï(n - 1, troisième, arrivée, départ)
```

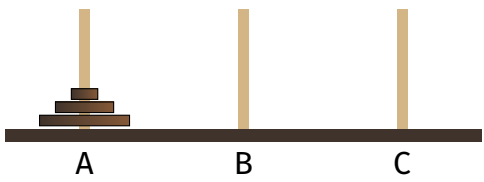
Exemple

```
>>> hanoi(3, "A", "C", "B")
Disque 1 : A -> C
Disque 2 : A -> B
Disque 1 : C -> B
Disque 3 : A -> C
Disque 1 : B -> A
Disque 2 : B -> C
Disque 1 : A -> C
```



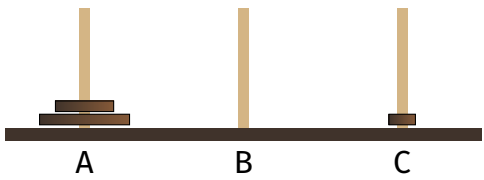
Exemple

```
>>> hanoi(3, "A", "C", "B")
Disque 1 : A -> C    *
Disque 2 : A -> B
Disque 1 : C -> B
Disque 3 : A -> C
Disque 1 : B -> A
Disque 2 : B -> C
Disque 1 : A -> C
```



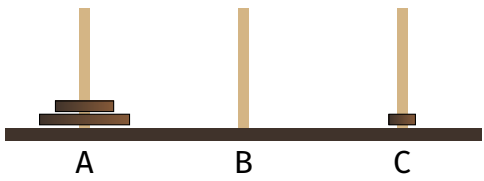
Exemple

```
>>> hanoi(3, "A", "C", "B")
Disque 1 : A -> C    *
Disque 2 : A -> B
Disque 1 : C -> B
Disque 3 : A -> C
Disque 1 : B -> A
Disque 2 : B -> C
Disque 1 : A -> C
```



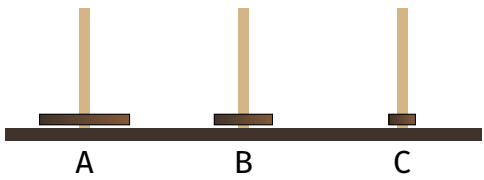
Exemple

```
>>> hanoi(3, "A", "C", "B")
Disque 1 : A -> C
Disque 2 : A -> B    *
Disque 1 : C -> B
Disque 3 : A -> C
Disque 1 : B -> A
Disque 2 : B -> C
Disque 1 : A -> C
```



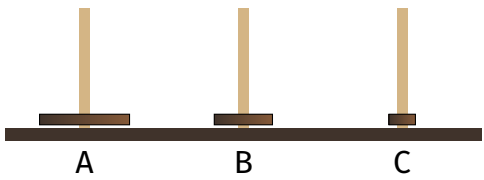
Exemple

```
>>> hanoi(3, "A", "C", "B")
Disque 1 : A -> C
Disque 2 : A -> B    *
Disque 1 : C -> B
Disque 3 : A -> C
Disque 1 : B -> A
Disque 2 : B -> C
Disque 1 : A -> C
```



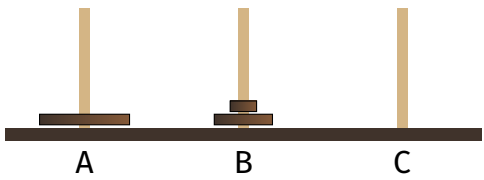
Exemple

```
>>> hanoi(3, "A", "C", "B")
Disque 1 : A -> C
Disque 2 : A -> B
Disque 1 : C -> B    *
Disque 3 : A -> C
Disque 1 : B -> A
Disque 2 : B -> C
Disque 1 : A -> C
```



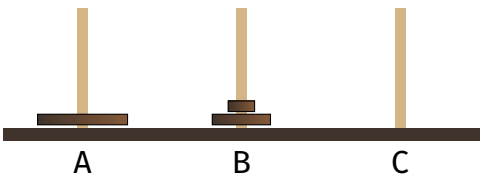
Exemple

```
>>> hanoi(3, "A", "C", "B")
Disque 1 : A -> C
Disque 2 : A -> B
Disque 1 : C -> B    *
Disque 3 : A -> C
Disque 1 : B -> A
Disque 2 : B -> C
Disque 1 : A -> C
```



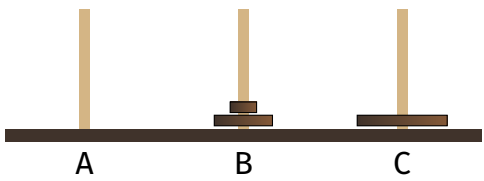
Exemple

```
>>> hanoi(3, "A", "C", "B")
Disque 1 : A -> C
Disque 2 : A -> B
Disque 1 : C -> B
Disque 3 : A -> C    *
Disque 1 : B -> A
Disque 2 : B -> C
Disque 1 : A -> C
```



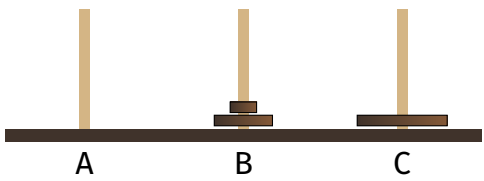
Exemple

```
>>> hanoi(3, "A", "C", "B")
Disque 1 : A -> C
Disque 2 : A -> B
Disque 1 : C -> B
Disque 3 : A -> C    *
Disque 1 : B -> A
Disque 2 : B -> C
Disque 1 : A -> C
```



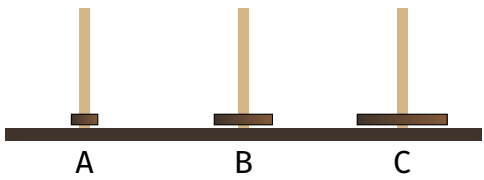
Exemple

```
>>> hanoi(3, "A", "C", "B")
Disque 1 : A -> C
Disque 2 : A -> B
Disque 1 : C -> B
Disque 3 : A -> C
Disque 1 : B -> A    *
Disque 2 : B -> C
Disque 1 : A -> C
```



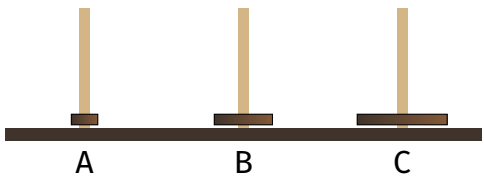
Exemple

```
>>> hanoi(3, "A", "C", "B")
Disque 1 : A -> C
Disque 2 : A -> B
Disque 1 : C -> B
Disque 3 : A -> C
Disque 1 : B -> A    *
Disque 2 : B -> C
Disque 1 : A -> C
```



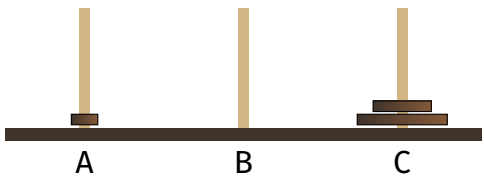
Exemple

```
>>> hanoi(3, "A", "C", "B")
Disque 1 : A -> C
Disque 2 : A -> B
Disque 1 : C -> B
Disque 3 : A -> C
Disque 1 : B -> A
Disque 2 : B -> C    *
Disque 1 : A -> C
```



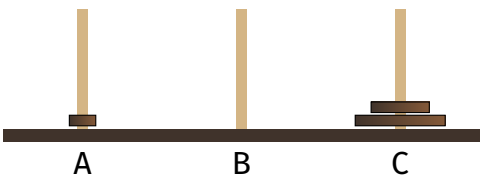
Exemple

```
>>> hanoi(3, "A", "C", "B")
Disque 1 : A -> C
Disque 2 : A -> B
Disque 1 : C -> B
Disque 3 : A -> C
Disque 1 : B -> A
Disque 2 : B -> C    *
Disque 1 : A -> C
```



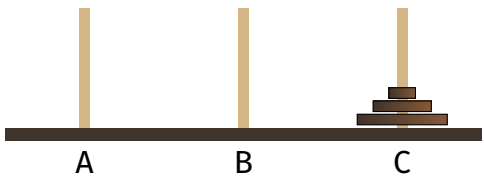
Exemple

```
>>> hanoi(3, "A", "C", "B")
Disque 1 : A -> C
Disque 2 : A -> B
Disque 1 : C -> B
Disque 3 : A -> C
Disque 1 : B -> A
Disque 2 : B -> C
Disque 1 : A -> C    *
```



Exemple

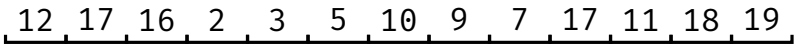
```
>>> hanoi(3, "A", "C", "B")
Disque 1 : A -> C
Disque 2 : A -> B
Disque 1 : C -> B
Disque 3 : A -> C
Disque 1 : B -> A
Disque 2 : B -> C
Disque 1 : A -> C    *
```



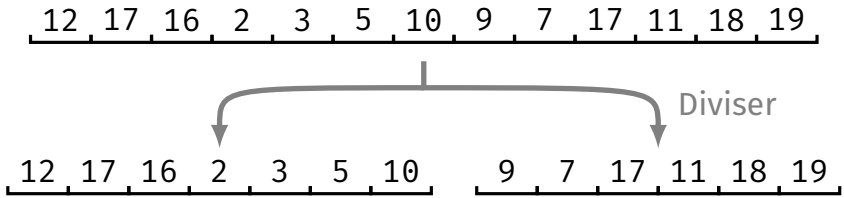
Le tri-fusion

Nous allons appliquer la méthode « diviser pour régner » à l'élaboration d'un algorithme de tri.

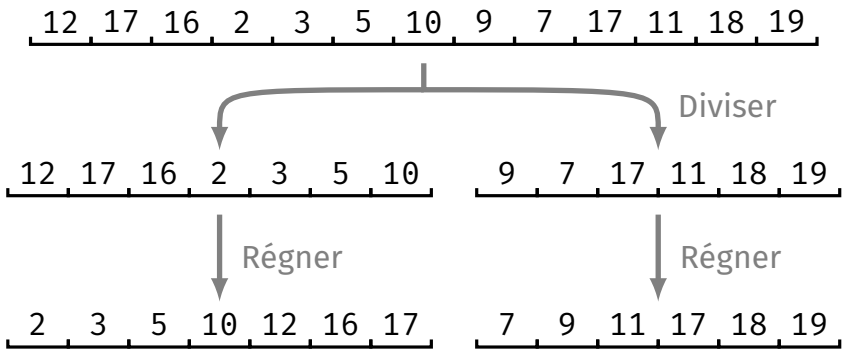
Diviser pour régner



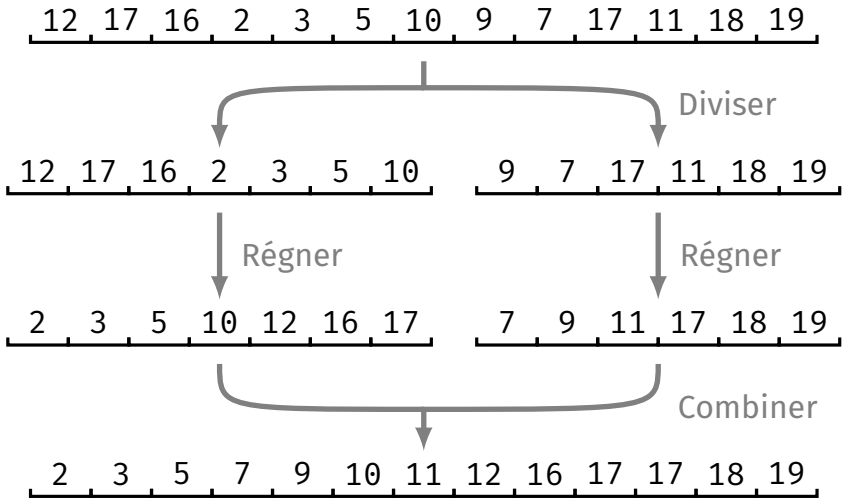
Diviser pour régner



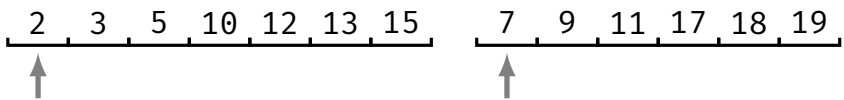
Diviser pour régner



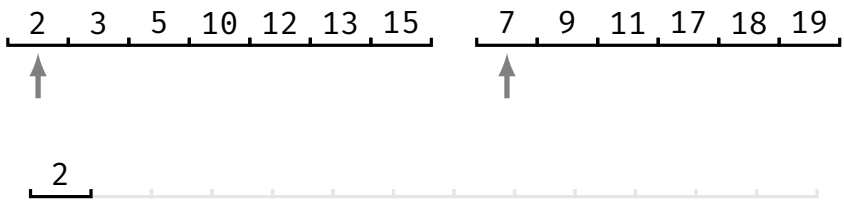
Diviser pour régner



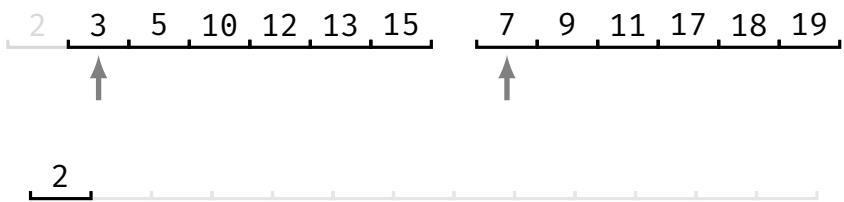
Fusion de deux tableaux ordonnés



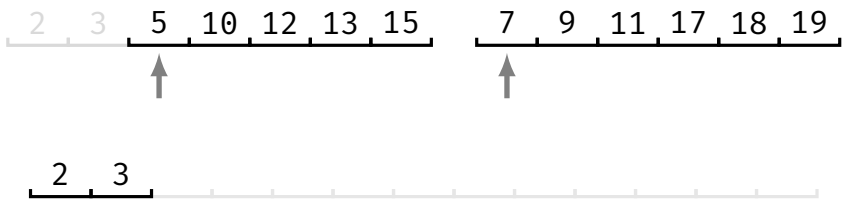
Fusion de deux tableaux ordonnés



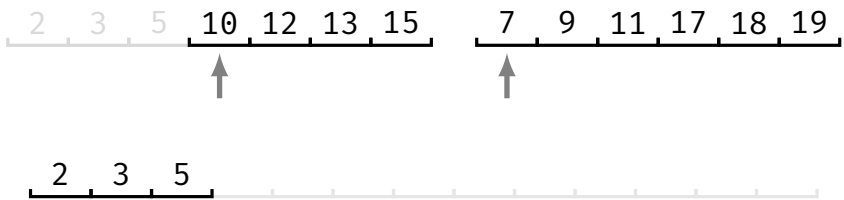
Fusion de deux tableaux ordonnés



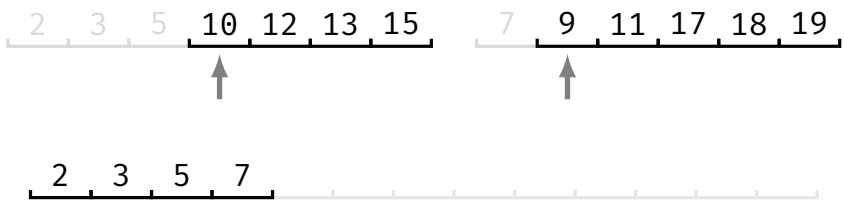
Fusion de deux tableaux ordonnés



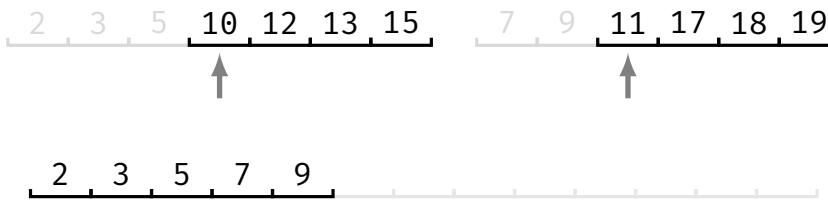
Fusion de deux tableaux ordonnés



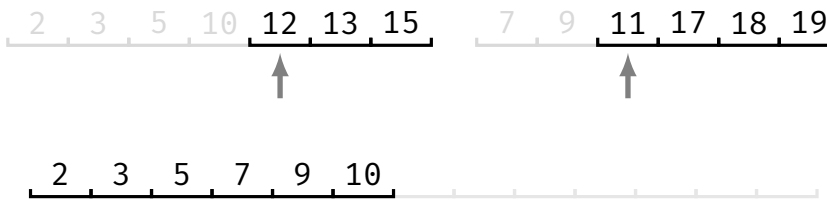
Fusion de deux tableaux ordonnés



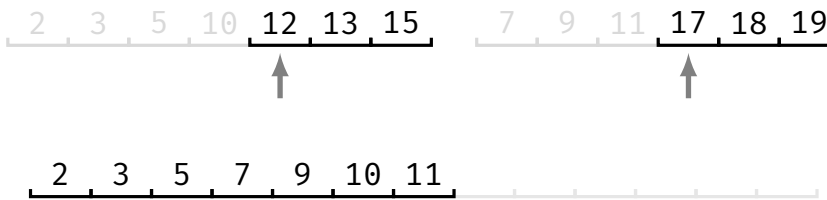
Fusion de deux tableaux ordonnés



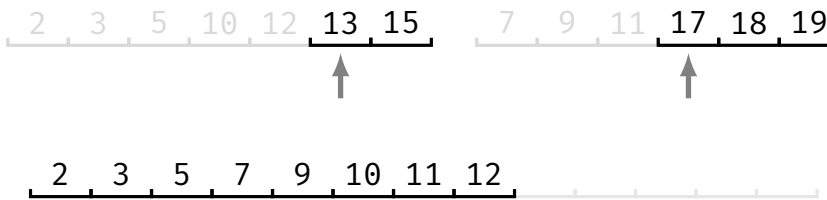
Fusion de deux tableaux ordonnés



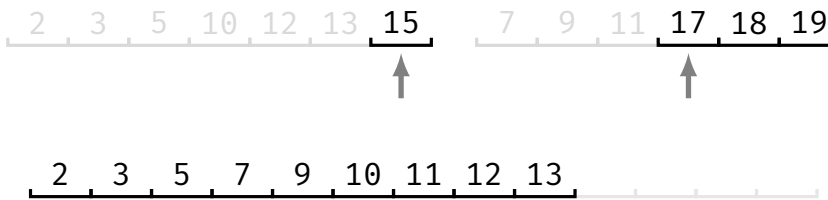
Fusion de deux tableaux ordonnés



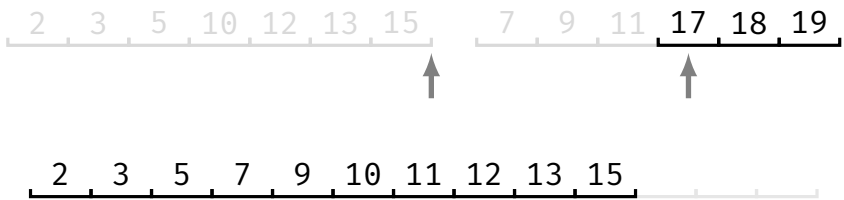
Fusion de deux tableaux ordonnés



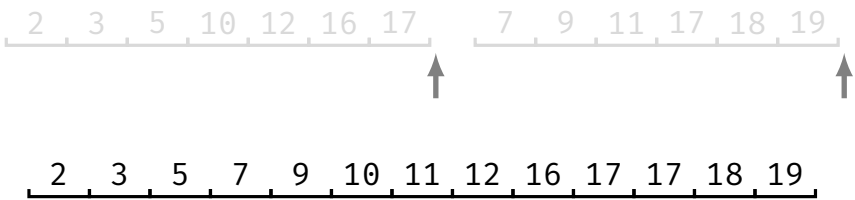
Fusion de deux tableaux ordonnés



Fusion de deux tableaux ordonnés



Fusion de deux tableaux ordonnés



Fusion de deux tableaux ordonnés

```
def fusion(tab1, tab2):
    résultat = []
    pos1, pos2 = 0, 0
```

Fusion de deux tableaux ordonnés

```
def fusion(tab1, tab2):
    résultat = []
    pos1, pos2 = 0, 0
    while pos1 < len(tab1) and pos2 < len(tab2):
```

Fusion de deux tableaux ordonnés

```
def fusion(tab1, tab2):
    résultat = []
    pos1, pos2 = 0, 0
    while pos1 < len(tab1) and pos2 < len(tab2):
        if tab1[pos1] < tab2[pos2]:
```

Fusion de deux tableaux ordonnés

```
def fusion(tab1, tab2):
    résultat = []
    pos1, pos2 = 0, 0
    while pos1 < len(tab1) and pos2 < len(tab2):
        if tab1[pos1] < tab2[pos2]:
            résultat.append(tab1[pos1])
            pos1 = pos1 + 1
```

Fusion de deux tableaux ordonnés

```
def fusion(tab1, tab2):
    résultat = []
    pos1, pos2 = 0, 0
    while pos1 < len(tab1) and pos2 < len(tab2):
        if tab1[pos1] < tab2[pos2]:
            résultat.append(tab1[pos1])
            pos1 = pos1 + 1
        else:
            résultat.append(tab2[pos2])
            pos2 = pos2 + 1
```

Fusion de deux tableaux ordonnés

```
def fusion(tab1, tab2):
    résultat = []
    pos1, pos2 = 0, 0
    while pos1 < len(tab1) and pos2 < len(tab2):
        if tab1[pos1] < tab2[pos2]:
            résultat.append(tab1[pos1])
            pos1 = pos1 + 1
        else:
            résultat.append(tab2[pos2])
            pos2 = pos2 + 1
    if pos1 < len(tab1):
        résultat = résultat + tab1[pos1:]
    else:
        résultat = résultat + tab2[pos2:]
    return résultat
```

Fusion de deux tableaux, complexité

La fusion de deux tableaux de longueurs n_1 et n_2 est en $n_1 + n_2$.

Tri-fusion

```
def tri_fusion(tab):
    n = len(tab)
    if n <= 1:
```

Tri-fusion

```
def tri_fusion(tab):
    n = len(tab)
    if n <= 1:
        # cas de base
        return tab
    else:
```

Tri-fusion

```
def tri_fusion(tab):
    n = len(tab)
    if n <= 1:
        # cas de base
        return tab
    else:
        # diviser
        t1 = tab[:n // 2]
        t2 = tab[n // 2:]
```

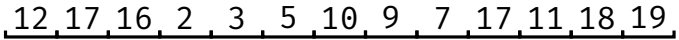
Tri-fusion

```
def tri_fusion(tab):
    n = len(tab)
    if n <= 1:
        # cas de base
        return tab
    else:
        # diviser
        t1 = tab[:n // 2]
        t2 = tab[n // 2:]
        # régner
        t1_trié = tri_fusion(t1)
        t2_trié = tri_fusion(t2)
```

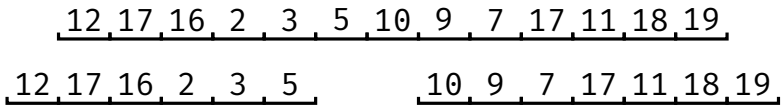
Tri-fusion

```
def tri_fusion(tab):
    n = len(tab)
    if n <= 1:
        # cas de base
        return tab
    else:
        # diviser
        t1 = tab[:n // 2]
        t2 = tab[n // 2:]
        # régner
        t1_trié = tri_fusion(t1)
        t2_trié = tri_fusion(t2)
        # combiner
        res = fusion(t1_trié, t2_trié)
        return res
```

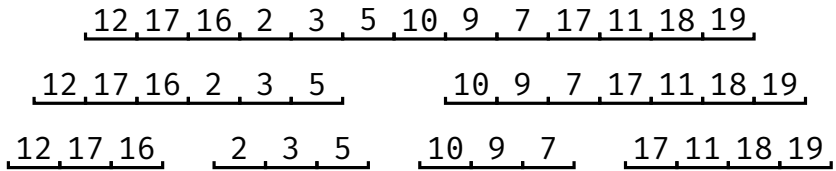
Tri-fusion, complexité



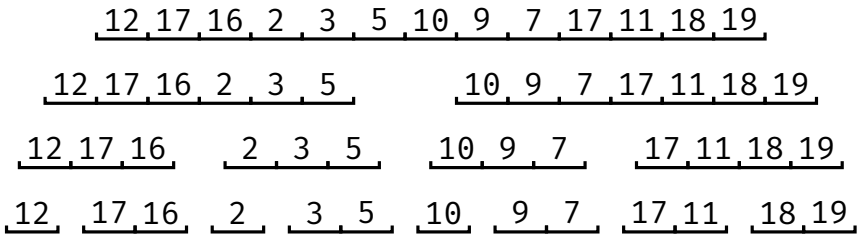
Tri-fusion, complexité



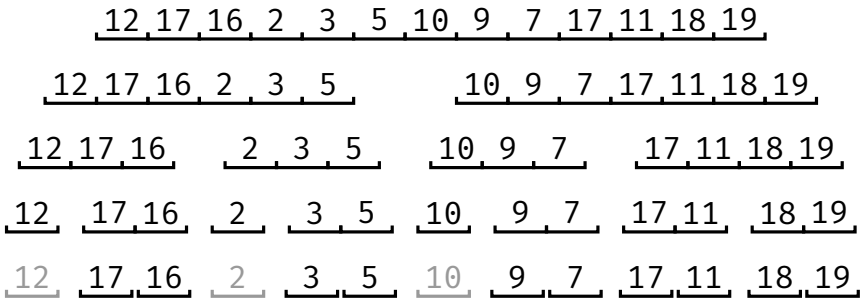
Tri-fusion, complexité



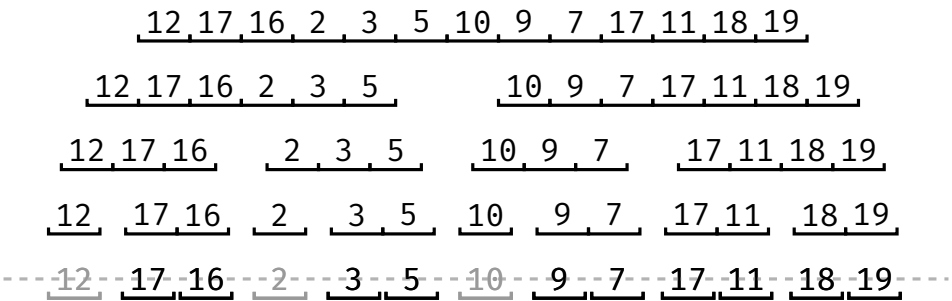
Tri-fusion, complexité



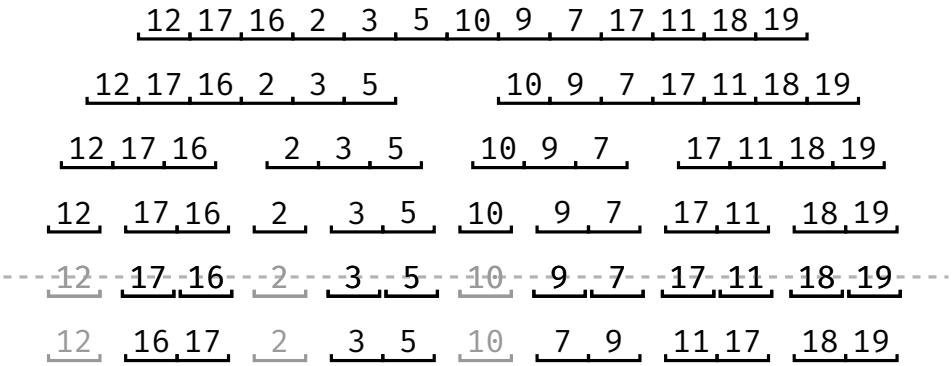
Tri-fusion, complexité



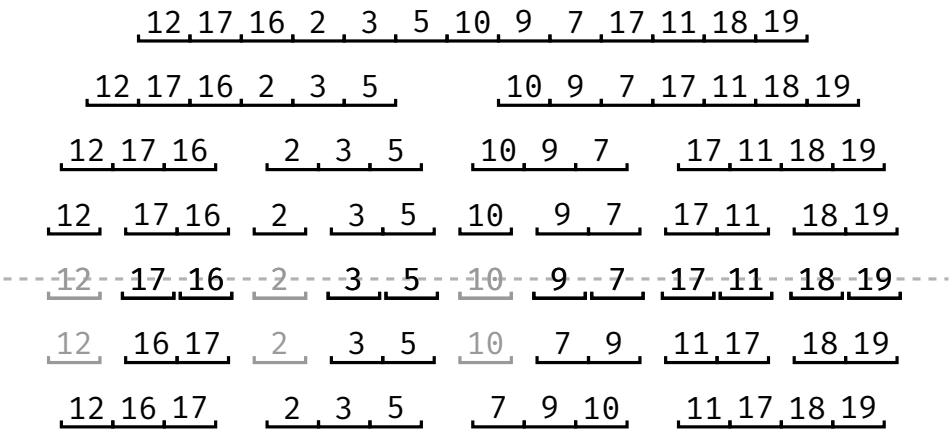
Tri-fusion, complexité



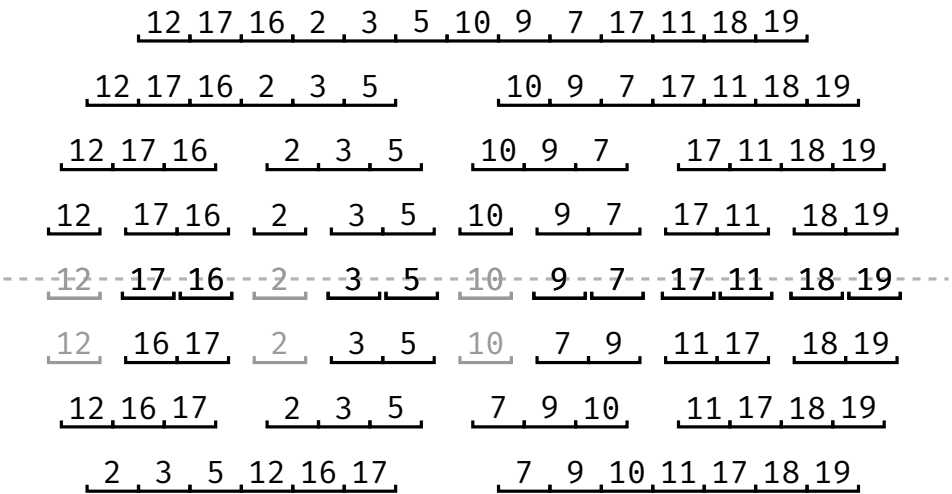
Tri-fusion, complexité



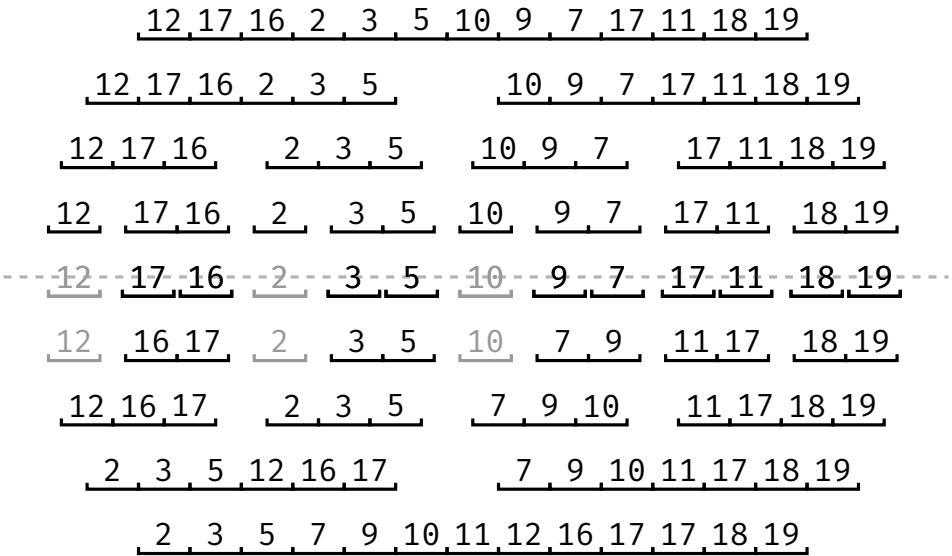
Tri-fusion, complexité



Tri-fusion, complexité



Tri-fusion, complexité



Tri-fusion, complexité

- Il y a de l'ordre de $\log_2 n$ étapes de division/combinaison.
- Chaque étape a un coût de l'ordre de n.

Le **tri-fusion** est en $n \log_2 n$.