Fonction Python `sorted`
Diviser pour régner

Christophe Viroulaud

Terminale - NSI

Algo 01

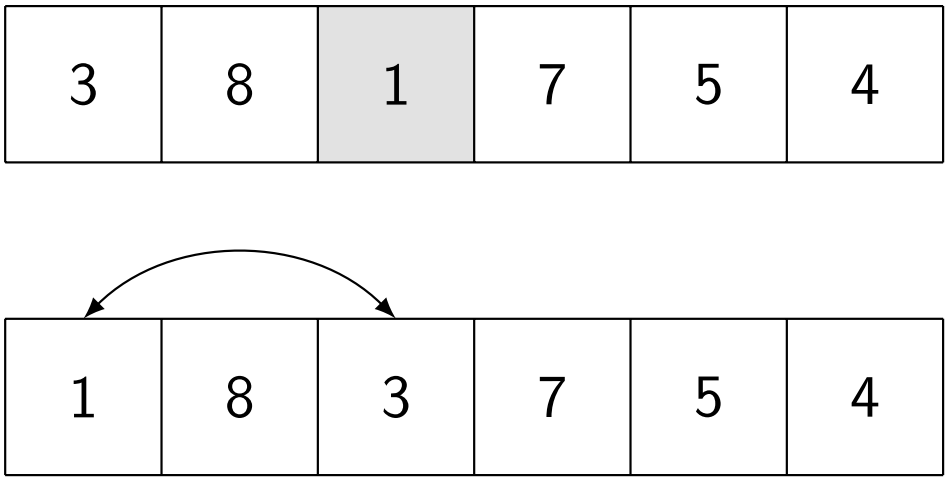
En tant que langage de haut niveau, Python offre des méthodes permettant d'effectuer efficacement certaines tâches courantes.
La méthode `sort` trie, en place, un tableau. .

Déterminer les algorithmes de tri implémentés dans la méthode `sort`.

Sommaire

Tri par sélection (en place)

```
1 Pour chaque élément du tableau
2     Trouver le plus petit élément dans la partie non triée.
3     Échanger cet élément avec le premier de la partie non triée.
```



Tri par sélection (en place)

```
1 Pour chaque élément du tableau
2     Trouver le plus petit élément dans la partie non triée.
3     Échanger cet élément avec le premier de la partie non triée.
```

Activité 1 :

1. Construire par compréhension un tableau de 10 entiers aléatoires compris entre 1 et 100.
2. Écrire la fonction `tri_selection(tab: list)` → `None` qui trie le tableau en place.

Correction

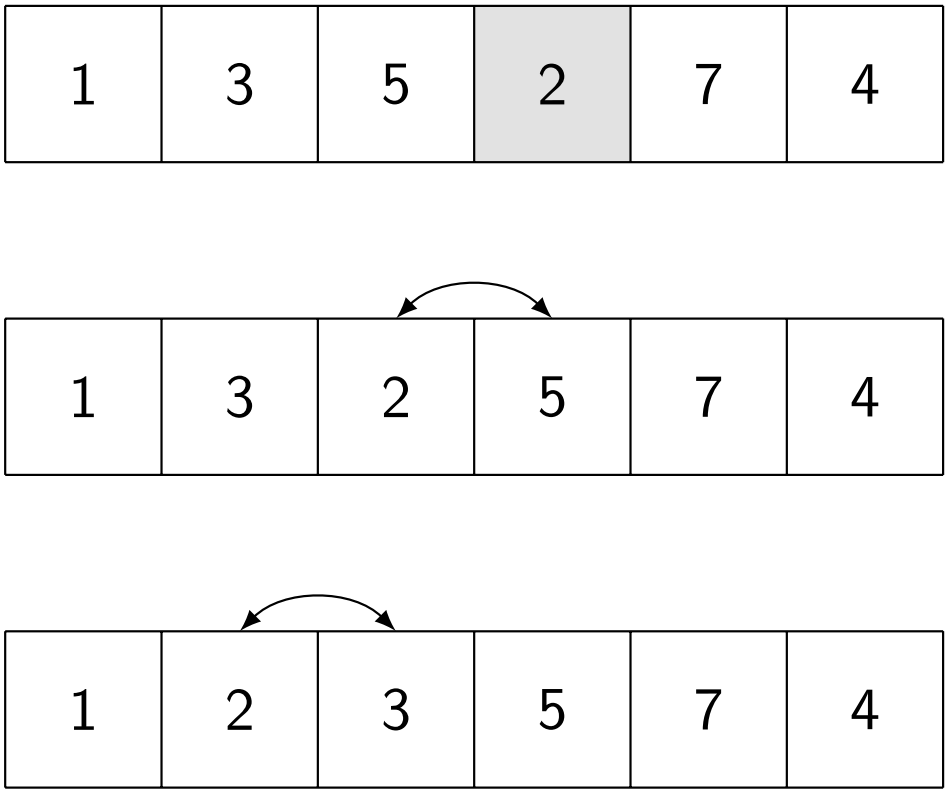
```
1  def tri_selection(tab: list) -> None:
2      for i in range(len(tab)):
3          # trouver le mini
4          i_mini = i
5          for j in range(i+1, len(tab)):
6              if tab[j] < tab[i_mini]:
7                  i_mini = j
8          # échanger
9          tab[i], tab[i_mini] = tab[i_mini], tab[i]
```

```
1  tab = [randint(1, 100) for _ in range(10)]
2  print(tab)
3  tri_selection(tab)
4  print(tab)
```

Sommaire

Tri par insertion (en place)

```
1 Pour chaque élément du tableau
2     Tant que l'élément précédent est inférieur
3         Permuter les deux éléments
```



Tri par insertion (en place)

```
1 Pour chaque élément du tableau
2     Tant que l'élément précédent est inférieur
3             Permuter les deux éléments
```

Activité 2 :

1. Construire par compréhension un tableau de 10 entiers aléatoires compris entre 1 et 100.
2. Écrire la fonction `tri_insertion(tab: list)` $\rightarrow$ `None` qui trie le tableau en place.

Correction

```
1  def tri_insertion(tab: list) -> None:
2      for i in range(len(tab)-1):
3          j = i+1
4          # tant que le précédent est inférieur
5          while j > 0 and tab[j] < tab[j-1]:
6              # permuter
7              tab[j], tab[j-1] = tab[j-1], tab[j]
8              j = j - 1
```

Sommaire

Comparaison des performances

Activité 3 :

1. Construire par compréhension un tableau de 10000 entiers compris entre 1 et 10000.
2. Mesurer la durée d'exécution de la méthode `sort` et des deux fonctions précédentes.

Correction

```
1 tab = [randint(1, 10000) for _ in range(10000)]
2 deb = time()
3 tri_selection(tab)
4 fin = time()
5 print(fin-deb)
```

Code 1 – tri par sélection

```
1 >>> sélection  4.557838678359985
2 >>> insertion 3.959839105606079
3 >>> sort  0.0019469261169433594
```

Code 2 – Résultats

Sommaire

Complexité

À retenir

La complexité temporelle estime le nombre d'opérations que doit réaliser le processeur pour exécuter le programme.

```python
def tri_selection(tab: list) -> None:
    for i in range(len(tab)): # n itérations
        # trouver le mini
        i_mini = i
        for j in range(i+1, len(tab)): # n itérations
            if tab[j] < tab[i_mini]:
                i_mini = j
        # échanger
        tab[i], tab[i_mini] = tab[i_mini], tab[i]
```

Code 3 – La complexité dépend de n^2

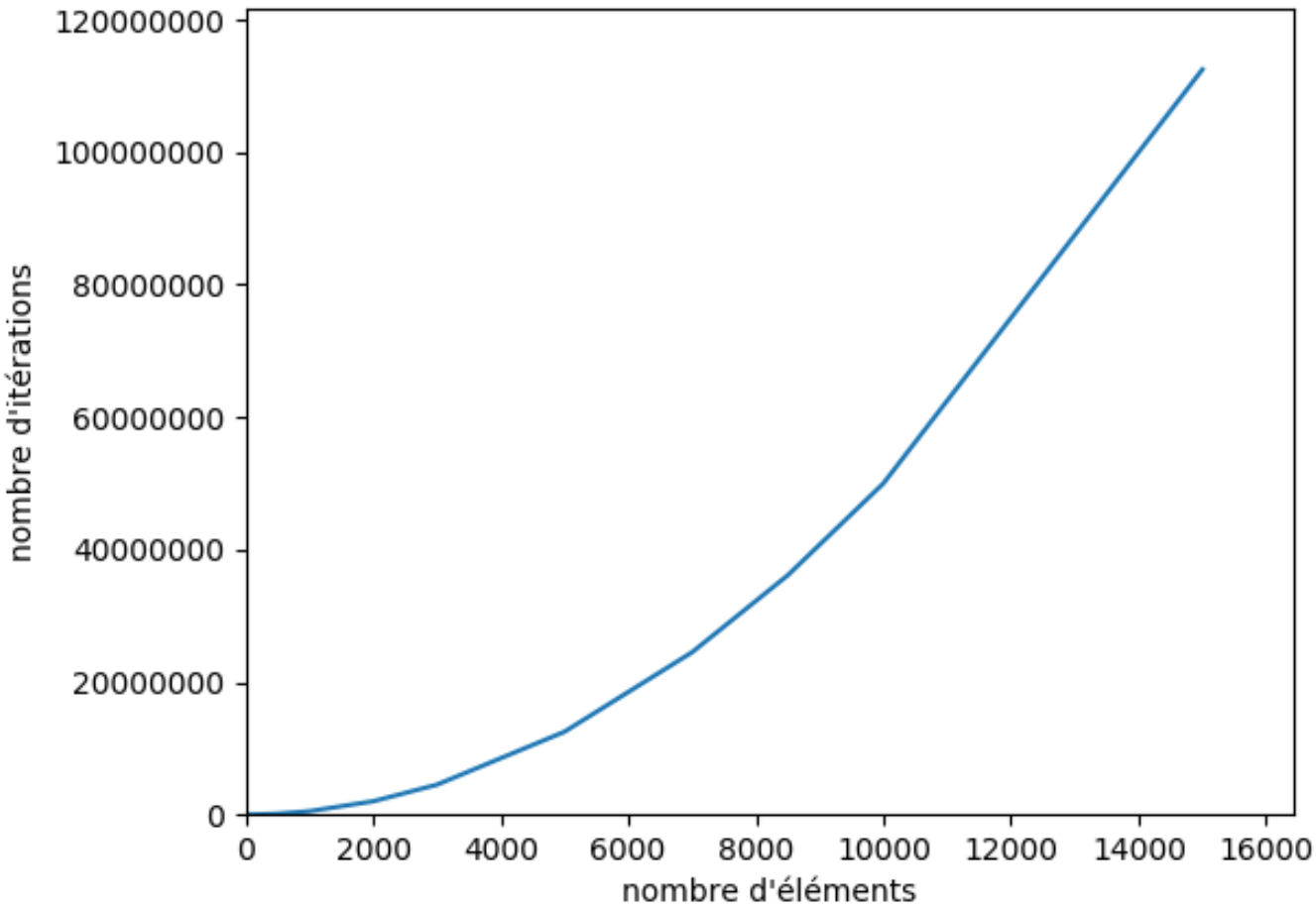


Figure 1 – Tri par sélection : complexité **quadratique**

Observation

- Les tris par sélection et insertion ont une complexité qui dépend de n^2.
- La méthode `sort` de Python semble avoir une complexité bien meilleure.

Sommaire

Résoudre de petits problèmes

Observation

La propriété triviale suivante permet de construire une nouvelle méthode de tri : **une liste qui contient 0 ou 1 élément est triée**.

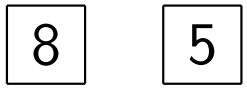


Figure 2 – Deux listes triées

Observation

Deux listes d'un élément peuvent être fusionnées facilement en une liste triée de deux éléments.

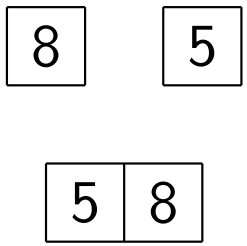


Figure 3 – Fusionner 2 listes de 1 élément

Principe

En résolvant des petits problèmes, nous pouvons remonter à des problèmes plus importants en appliquant le même principe.

Sommaire

...pour solutionner un gros problème

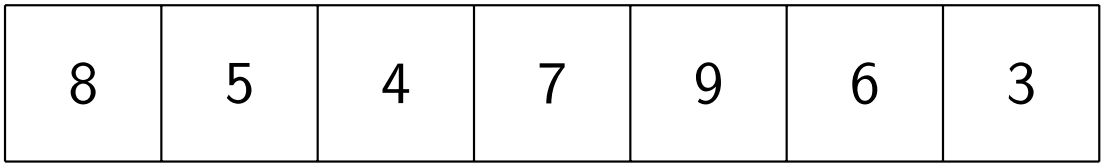


Figure 4 – Un gros problème : trier une liste

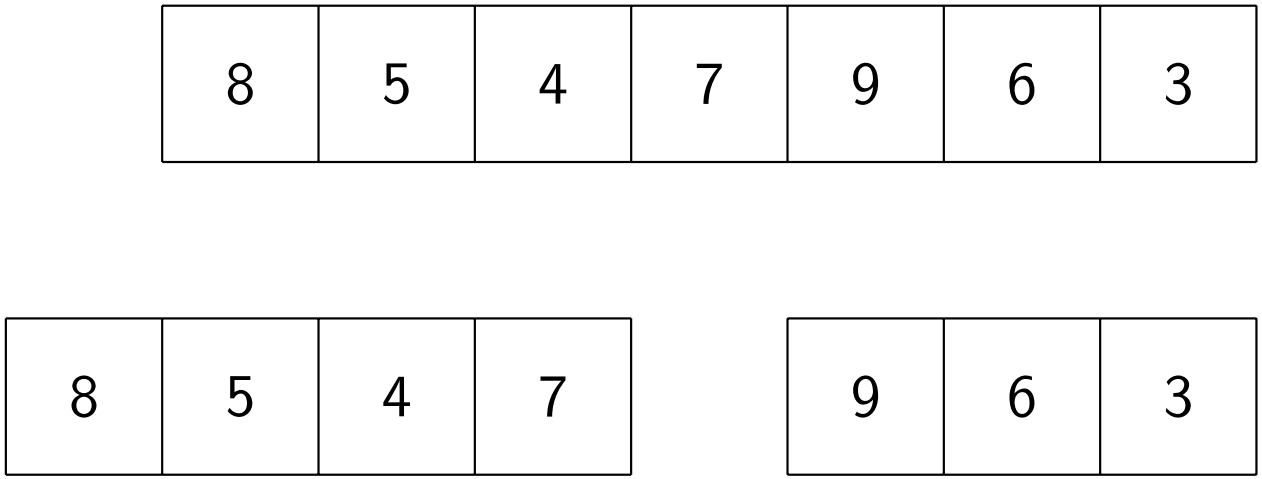
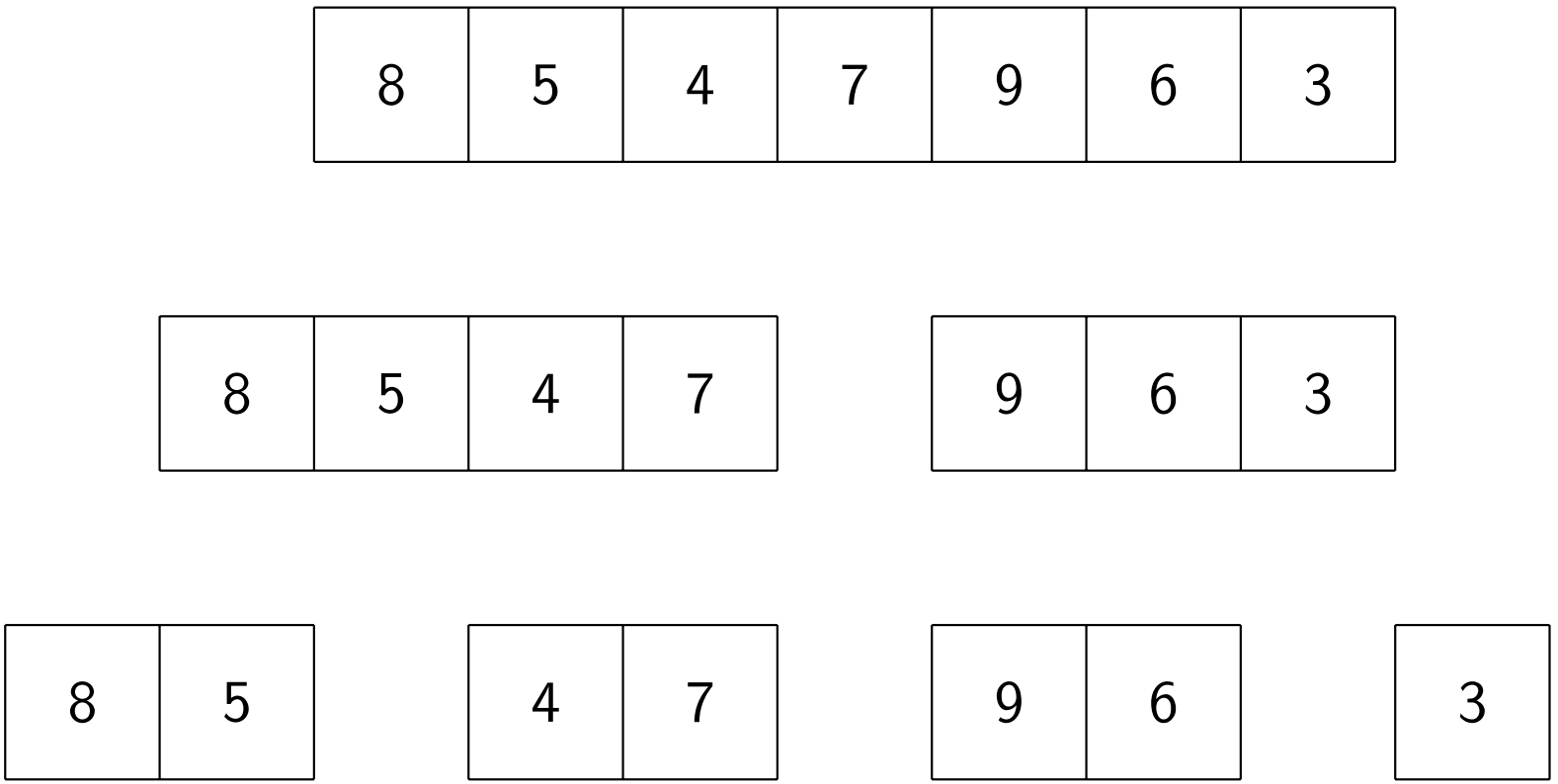


Figure 5 – Séparer la liste en deux listes plus petites



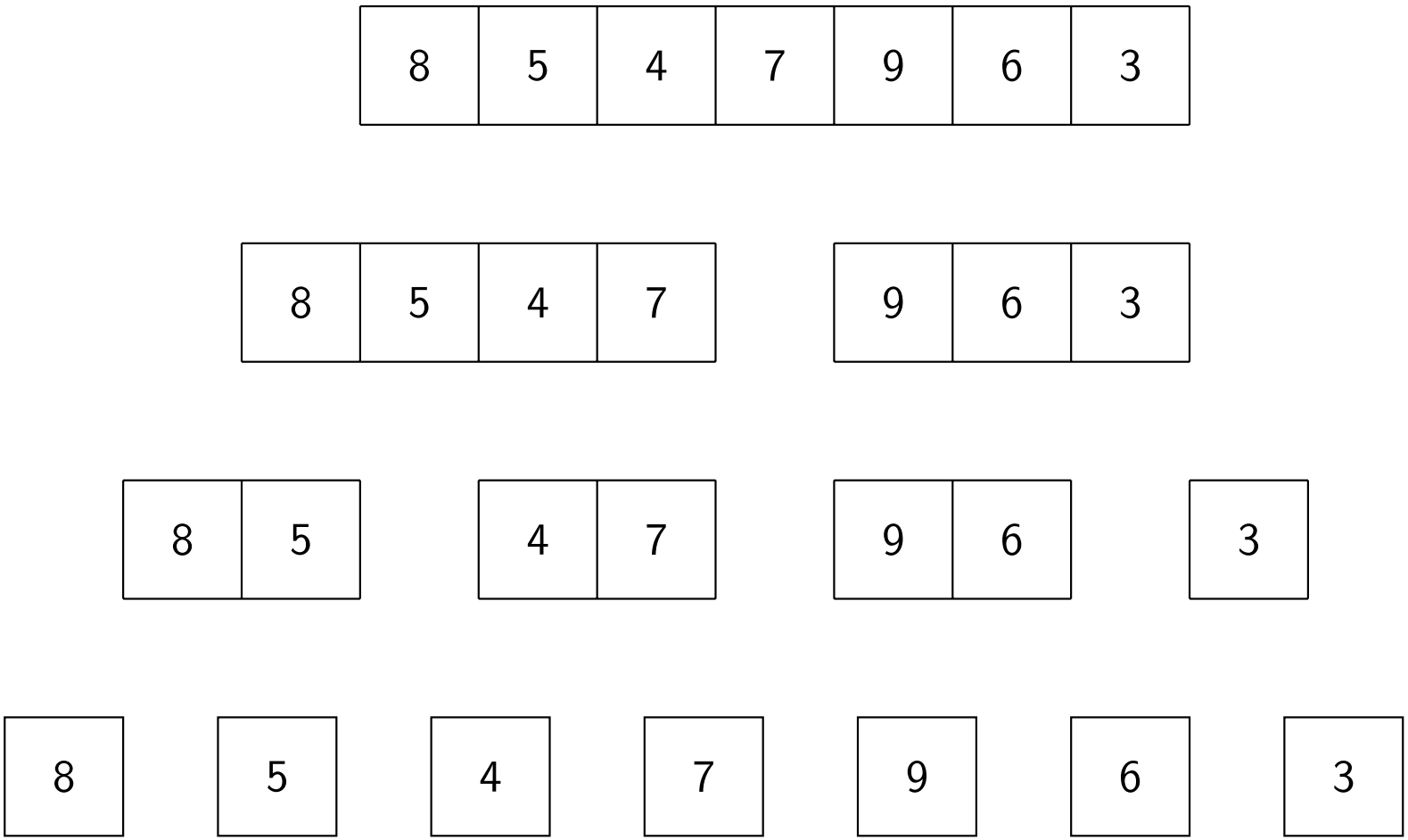


Figure 6 – Obtenir de petits problèmes

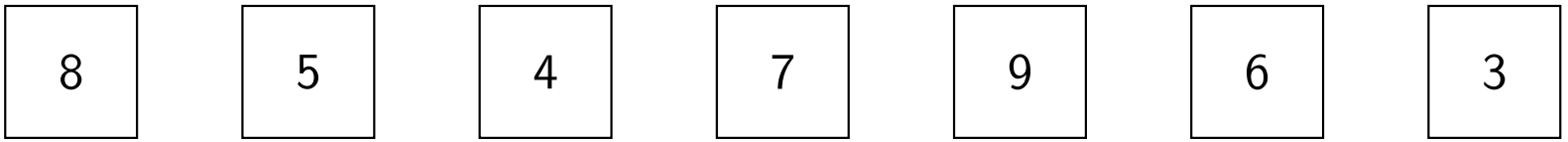


Figure 7 – Résoudre les petits problèmes

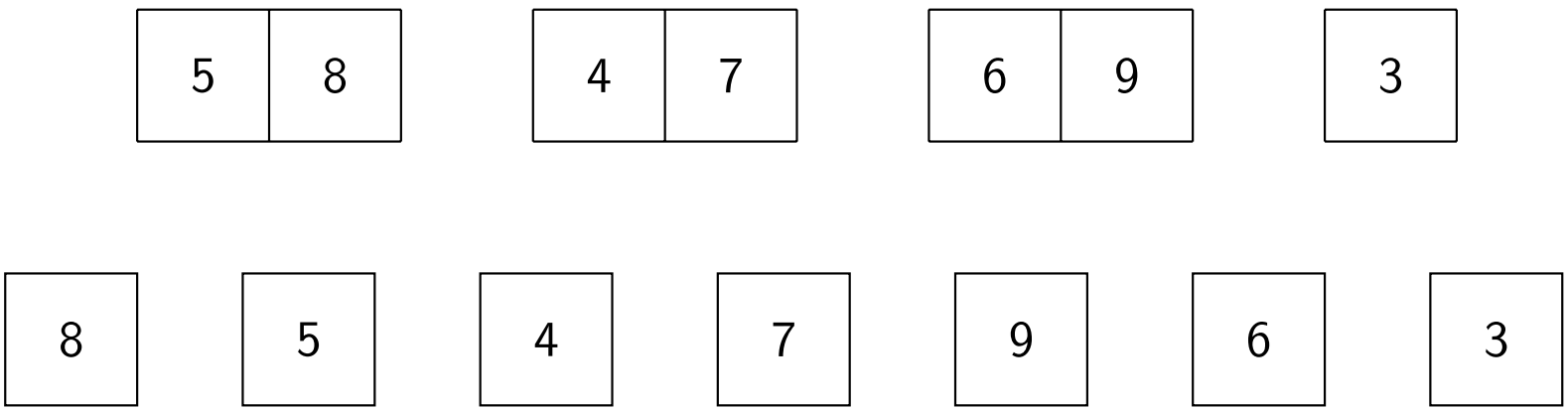


Figure 8 – Trier...

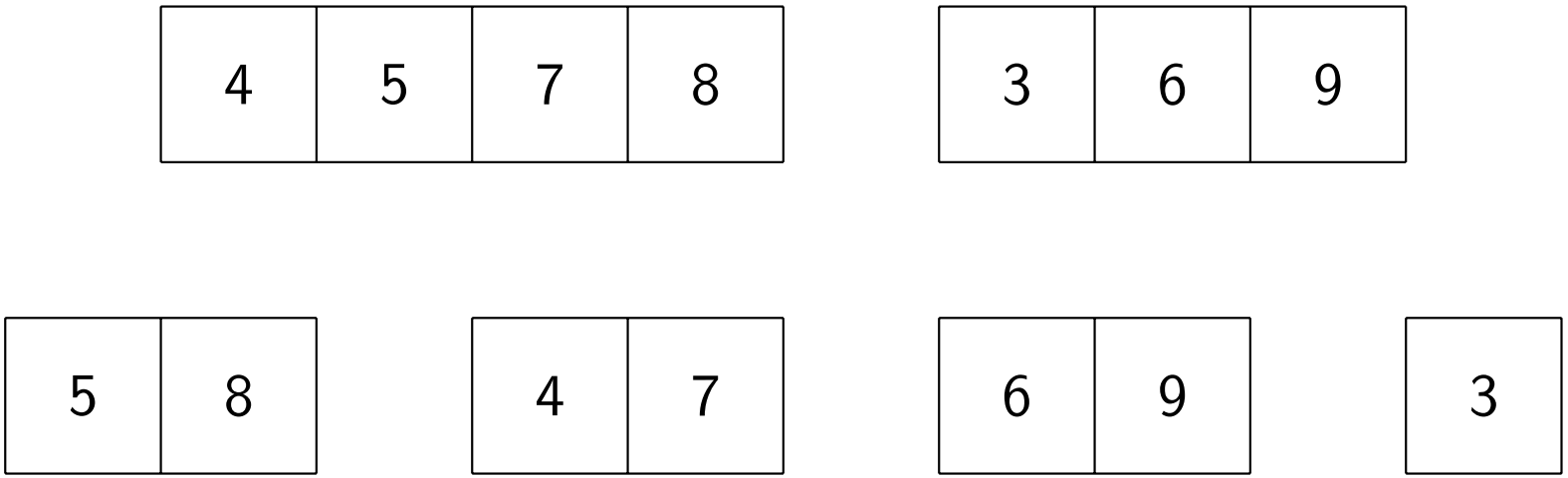


Figure 9 – ...et remonter

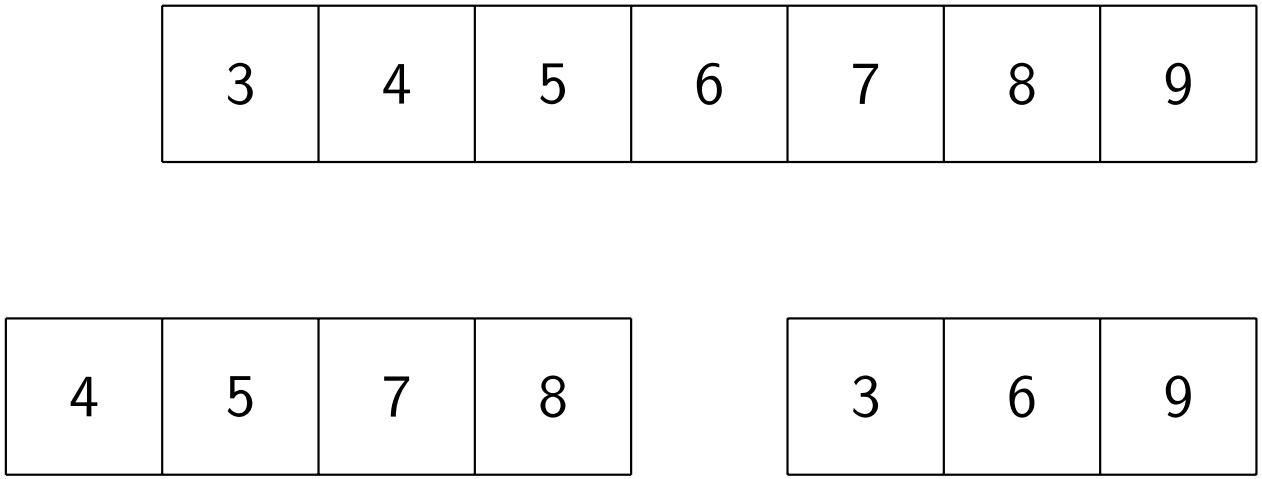


Figure 10 – Le tri se termine

Sommaire

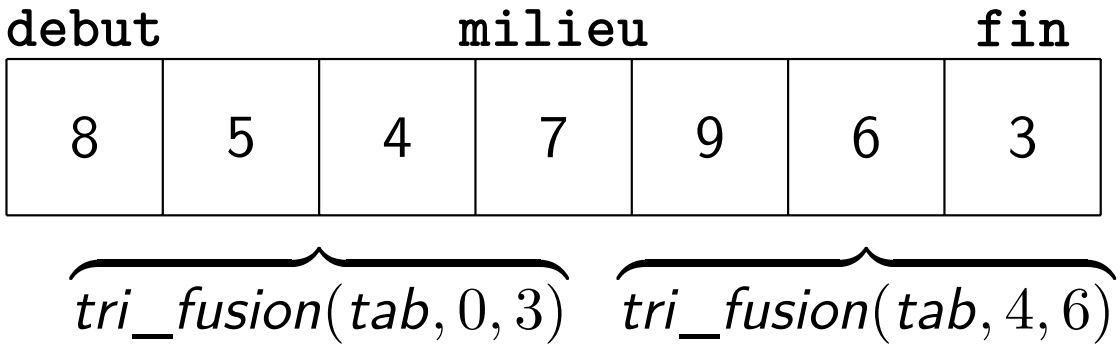
Diviser pour régner : un algorithme récursif

- **cas limite :** le tableau est de taille minimale
- sinon
  - rechercher le **milieu** du tableau,
  - **appel récursif** sur chaque sous-tableau
  - **fusionner** les tableaux lors de la remontée d'appel

Code 4 – Tri fusion

Implémentation

En pratique, nous effectuons un tri en place : les sous-tableaux sont repérés par des indices dans le tableau principal `tab`



Observation

Chaque appel récursif partage la liste en deux.

- **cas limite :** le tableau est de taille minimale (le début est supérieur ou égal à la fin).
- sinon
  - rechercher le **milieu** du tableau,
  - **appel récursif** sur chaque sous-tableau
  - **fusionner** les tableaux lors de la remontée d'appel

Activité 4 : Écrire la fonction `tri_fusion(tab: list, deb: int, fin: int)` → `None` qui trie le tableau :

- en appliquant l'algorithme ci-dessus,
- en utilisant la fonction `fusionner`.

Indication

On suppose que l'on dispose d'une fonction `fusionner(tab: list, deb: int, fin: int)` → `None` qui trie les éléments de `tab` entre les indices `deb` et `fin`.

Correction

```
1 def tri_fusion(tab: list, deb: int, fin: int) -> None:
2     if deb < fin:
3         milieu = (deb+fin)//2
4         tri_fusion(tab, deb, milieu)
5         tri_fusion(tab, milieu+1, fin)
6         fusionner(tab, deb, fin)
```

Code 5 – Tri fusion

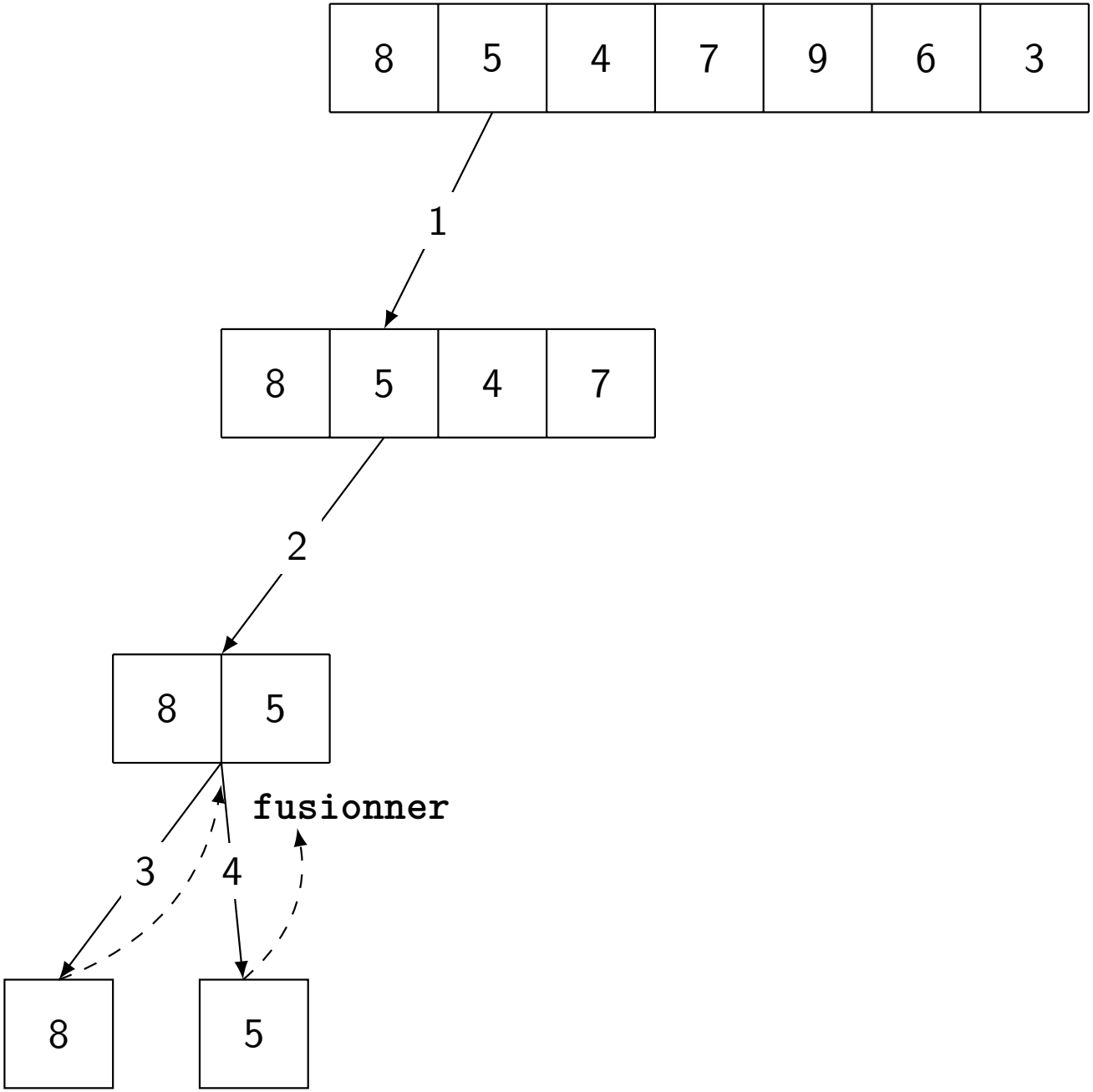


Figure 11 – Ordre des appels récursifs

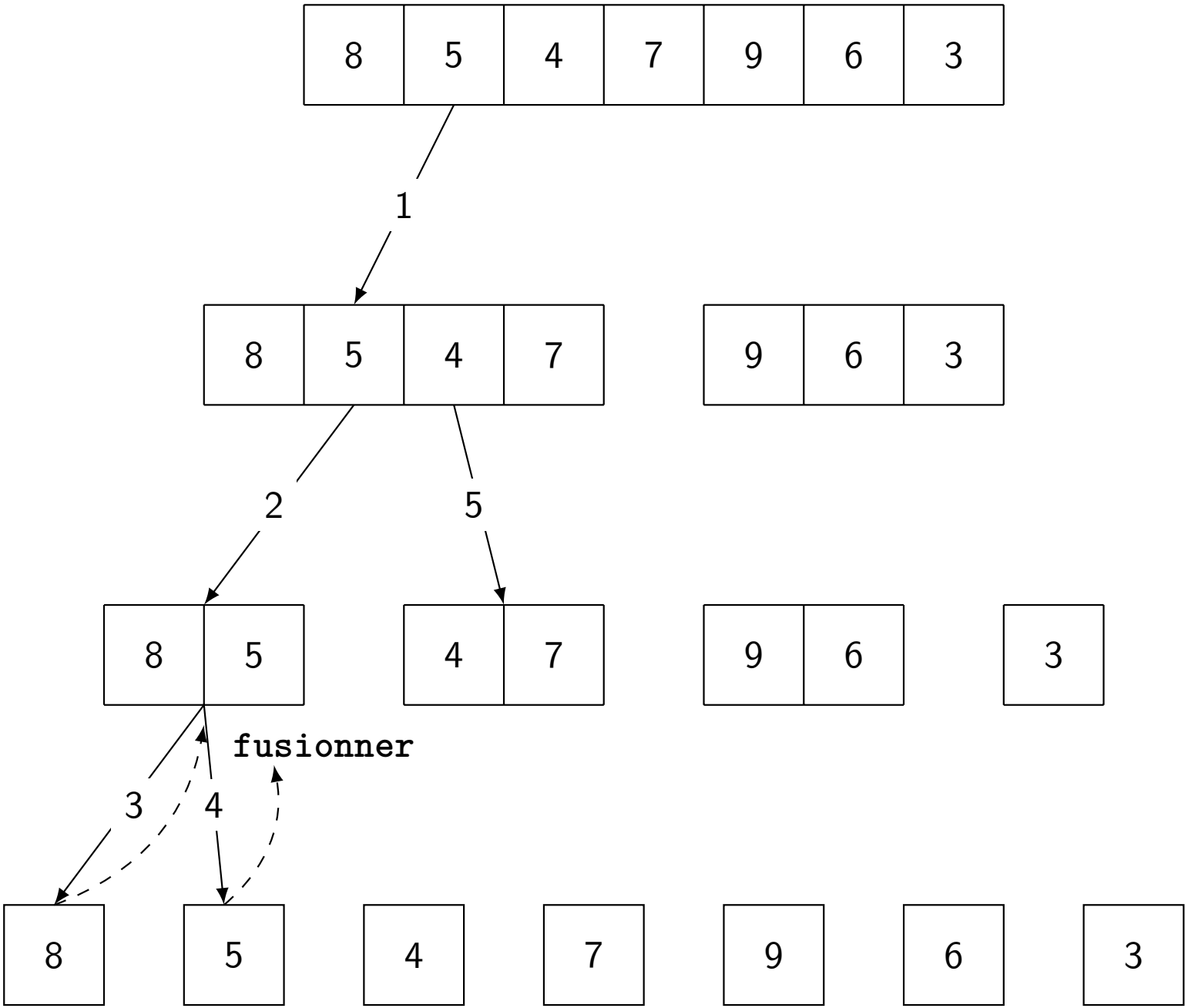


Figure 12 – Ordre des appels récursifs

Sommaire

Étape de la fusion

Lors de la remontée d'appel, la *fusion* assemble deux tableaux triés, en un seul.

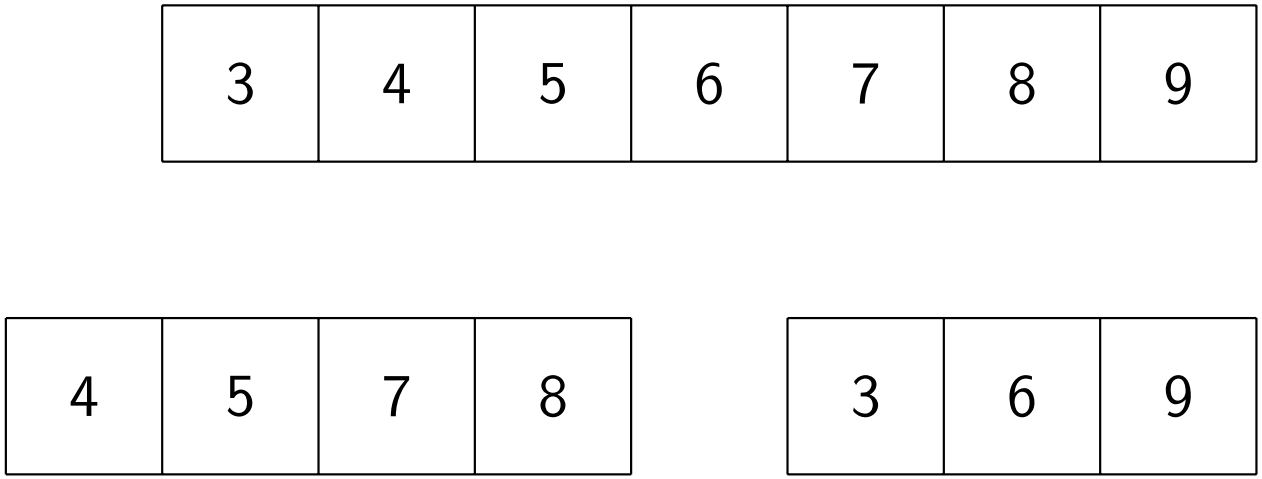


Figure 13 – Fusionner

En pratique, nous effectuons un tri *place*.

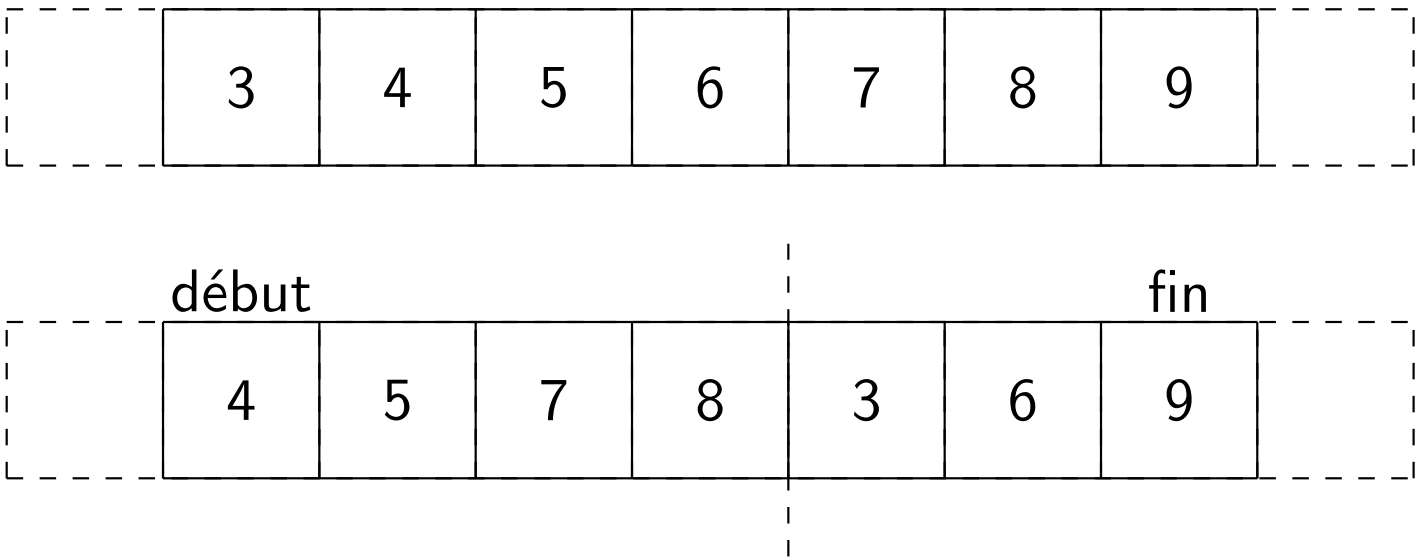


Figure 14 – Fusionner

Pour assembler les deux parties du tableau, il faut prendre le plus petit élément, jusqu'à vider un des deux blocs. Puis on complète avec les éléments restants.

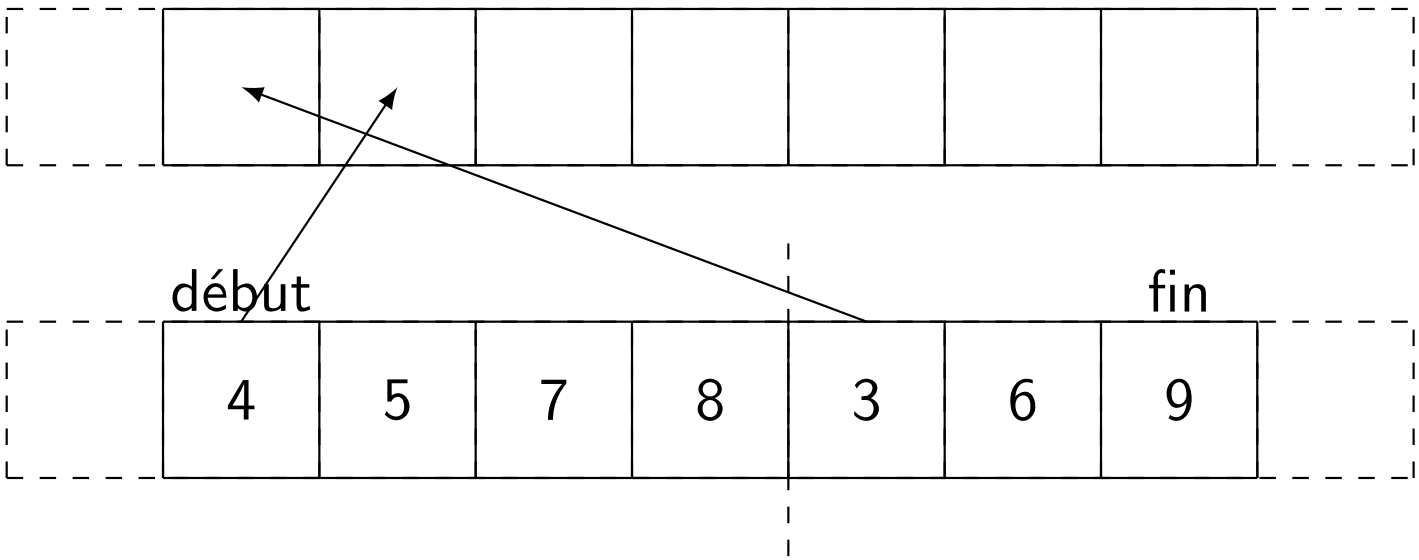


Figure 15 – Fusionner

Puis on complète avec les éléments restants.

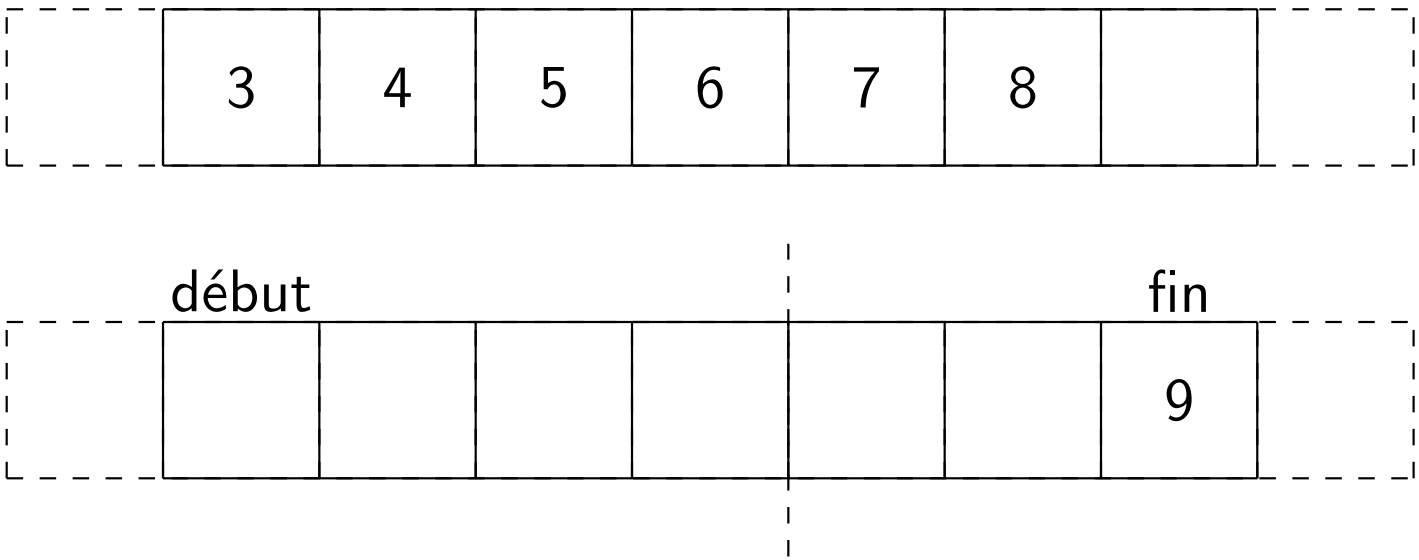


Figure 16 – Compléter

Activité 5 : Écrire la fonction `fusionner(tab: list, deb: int, fin: int)` → `None` qui assemble les éléments de `tab` entre les indices `deb` et `fin`. La fonction respectera l'algorithme suivant :

- Créer un tableau temporaire `res` vide.
- Trouver le milieu de la partie à trier.
- Tant que les parties gauches et droites n'ont pas été entièrement parcourues :
  - Si l'élément en cours de gauche est inférieur à celui de droite :
    - insérer l'élément en cours de gauche dans `res`
  - sinon
    - insérer l'élément de droite dans `res`
- Ajouter les éléments restants de la partie non parcourue entièrement.
- Recopier `res` dans la partie de `tab`.

Correction

```
1  def fusionner(tab: list, deb: int, fin: int) -> None:
2      res = []
3      milieu = (deb+fin)//2
4      i = deb
5      j = milieu+1
```

Code 6 – initialiser

Correction

```
1  while i <= milieu and j <= fin:
2      if tab[i] <= tab[j]:
3          res.append(tab[i])
4          i += 1
5      else:
6          res.append(tab[j])
7          j += 1
```

Code 7 – assembler

Correction

```
1 for i1 in range(i, milieu+1):
2     res.append(tab[i1])
3 for j1 in range(j, fin+1):
4     res.append(tab[j1])
```

Code 8 – compléter

Correction

```
1 # remplacement tab par res
2 for k in range(len(res)):
3     tab[deb+k] = res[k]
```

Code 9 – remplacer

Sommaire

Mesure de la durée d'exécution

Activité 6 :

1. Construire par compréhension un tableau de 10000 entiers compris entre 1 et 10000.
2. Mesurer la durée d'exécution de la fonction `tri_fusion` et des deux fonctions précédentes.

Correction

```
1 tab = [randint(1, 10000) for _ in range(10000)]
2 deb = time()
3 tri_fusion(tab, 0, len(tab)-1)
4 fin = time()
5 print("fusion ", fin-deb)
```

```
1 >>> sélection  4.557838678359985
2 >>> insertion 3.959839105606079
3 >>> sort  0.0019469261169433594
4 >>> fusion  0.1485743522644043
```

Sommaire

Complexité du découpage

Activité 7 : À chaque appel de la fonction `tri_fusion` nous divisons la liste en deux. Combien de fois faut-il couper la liste en deux pour obtenir des listes d'un élément ?

Mathématiquement nous cherchons a tel que :

$$\frac{n}{2^a} = 1$$

Le logarithme base 2 noté log_2 se définit : $log_2(2^x) = x$.

$$\frac{n}{2^a} = 1$$

$$\iff n = 2^a$$

$$\iff \log_2 n = \log_2(2^a)$$

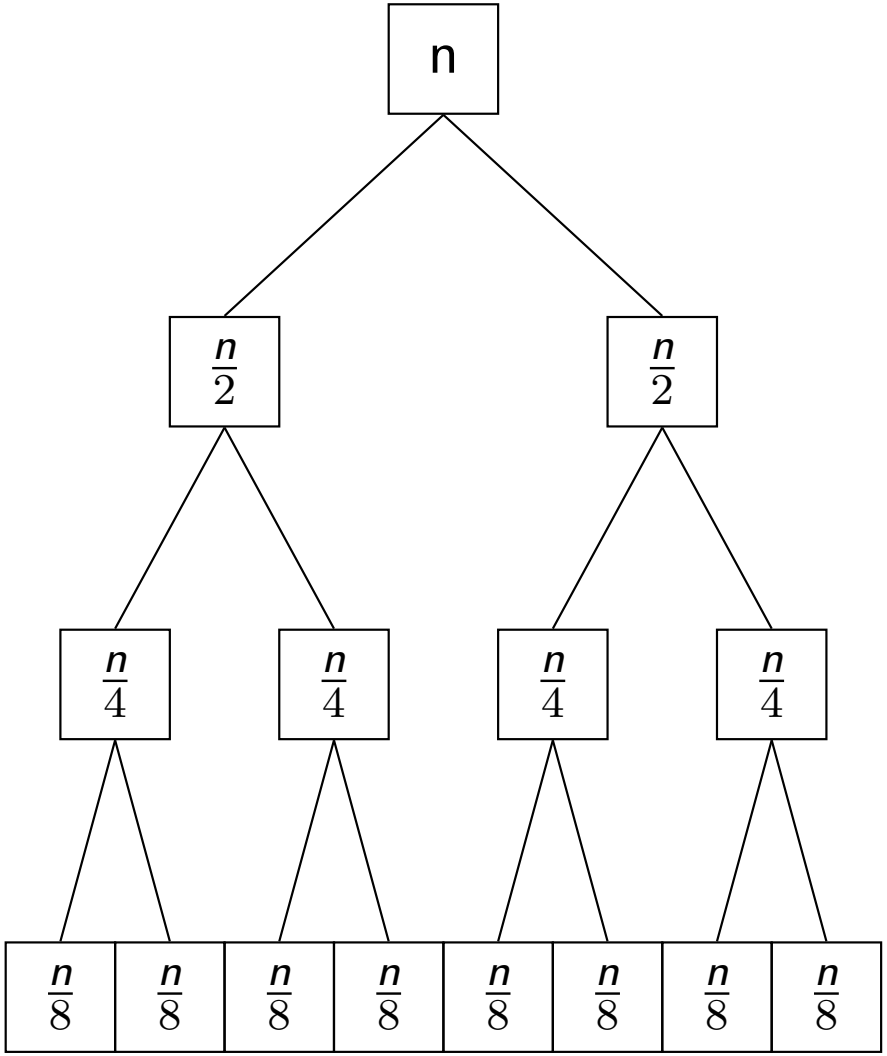
$$\iff \log_2 n = a$$

À retenir

La complexité du découpage en sous-listes est **logarithmique**.

Complexité de la fusion

La fonction **`fusionner`** réalise n comparaisons pour assembler deux listes de taille $\frac{n}{2}$.



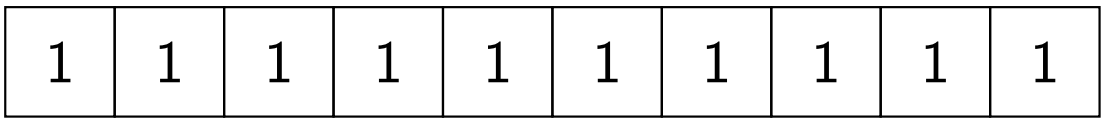
n

$2 \times (\frac{n}{2}) = n$

$2^2 \times (\frac{n}{2^2}) = n$

$2^3 \times (\frac{n}{2^3}) = n$

.....



$2^{log_2(n)} \times (\frac{n}{2^{log_2(n)}}) = n$

Complexité du tri fusion

À retenir

Chaque niveau de fusion a un coup de n et il y a $\log_2(n)$ niveaux. La complexité globale du tri fusion dépend de :

$$n \times log_2(n)$$

Activité 8 : Sachant que la complexité du tri par sélection est quadratique, la comparer au tri fusion pour un tableau de 100, 1000, 10000, 100000 éléments.

Rappel mathématique :

$$\log_2(n) = \frac{\ln n}{\ln 2}$$

Correction

éléments	tri par sélection	tri fusion
100	10^4	664
1000	10^6	9966
10000	10^8	132877
100000	10^{10}	1660964

Sommaire

Timsort

La méthode native `sort` implémente l'algorithme **Timsort** mis au point par Tim Peters en 2002. C'est un algorithme hybride de plusieurs tris :

- tri fusion
- tri par insertion