


Première NSI

Thème 3	Algorithmique
Séance 1	Complexité
Séance 2	Tri par insertion
Séance 3	Tri par sélection
Séance 4	Dichotomie
Séance 5	Algorithme glouton
Séance 6	Algorithme KNN

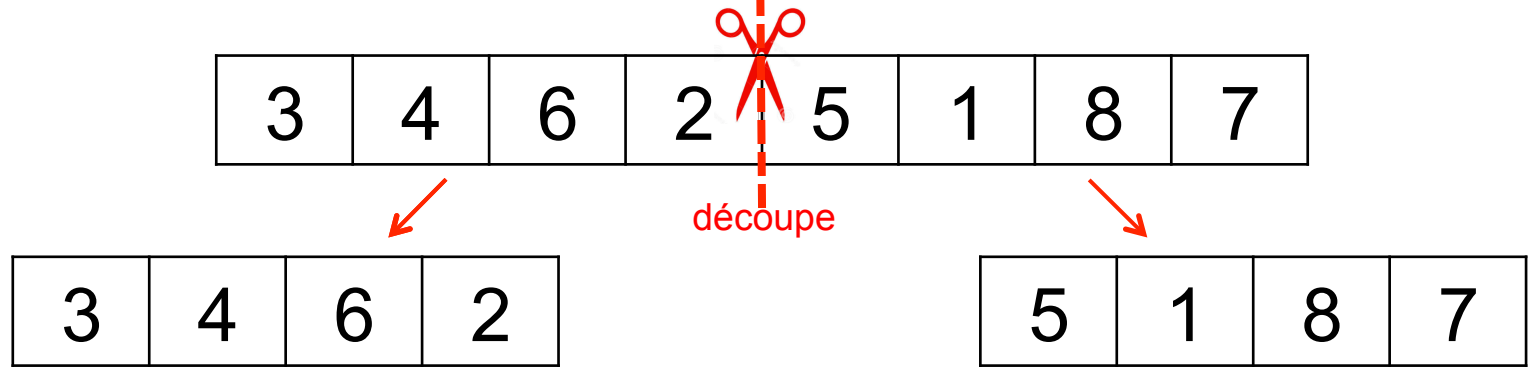
Terminale NSI

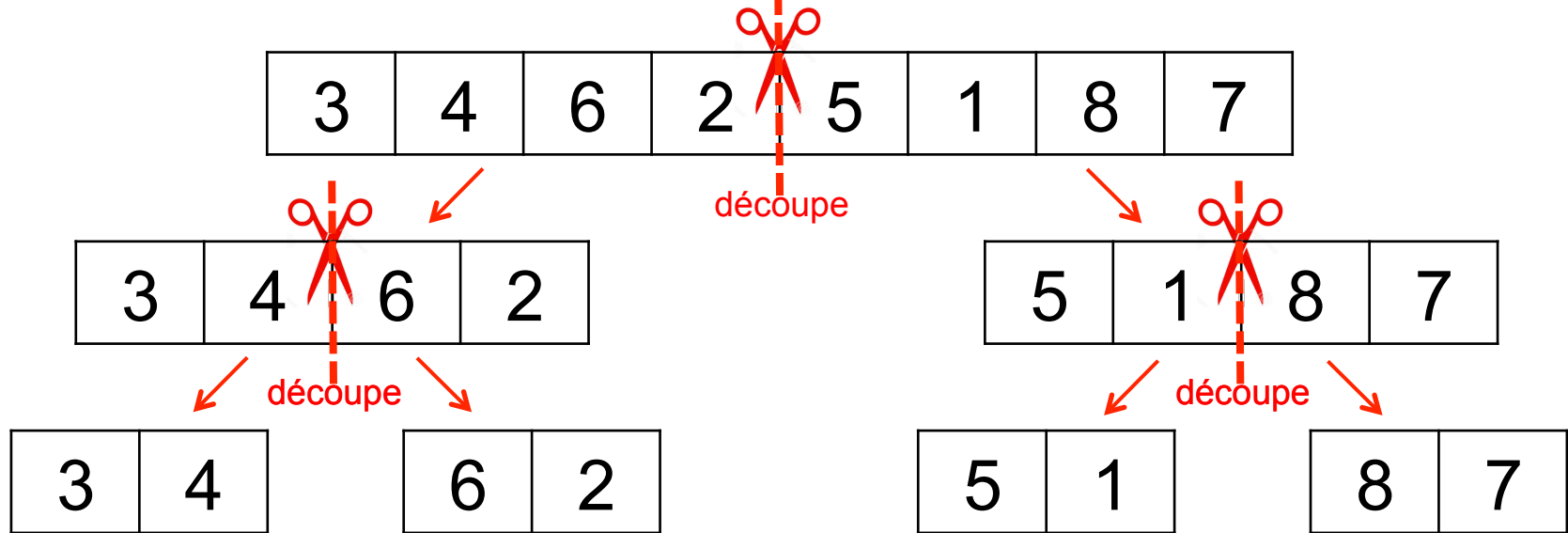


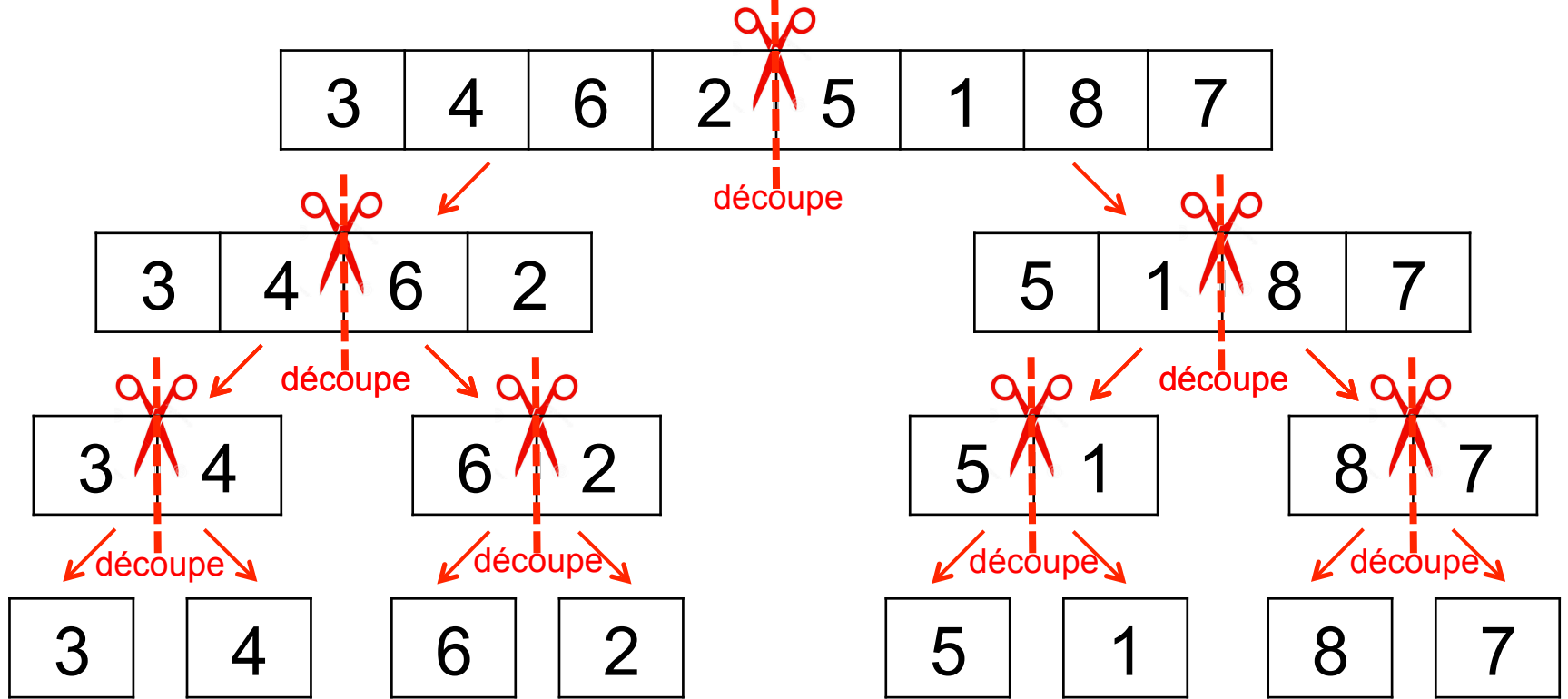
Chapitre 6	Diviser pour régner
Séance 1	Tri fusion
Séance 2	TP Joconde

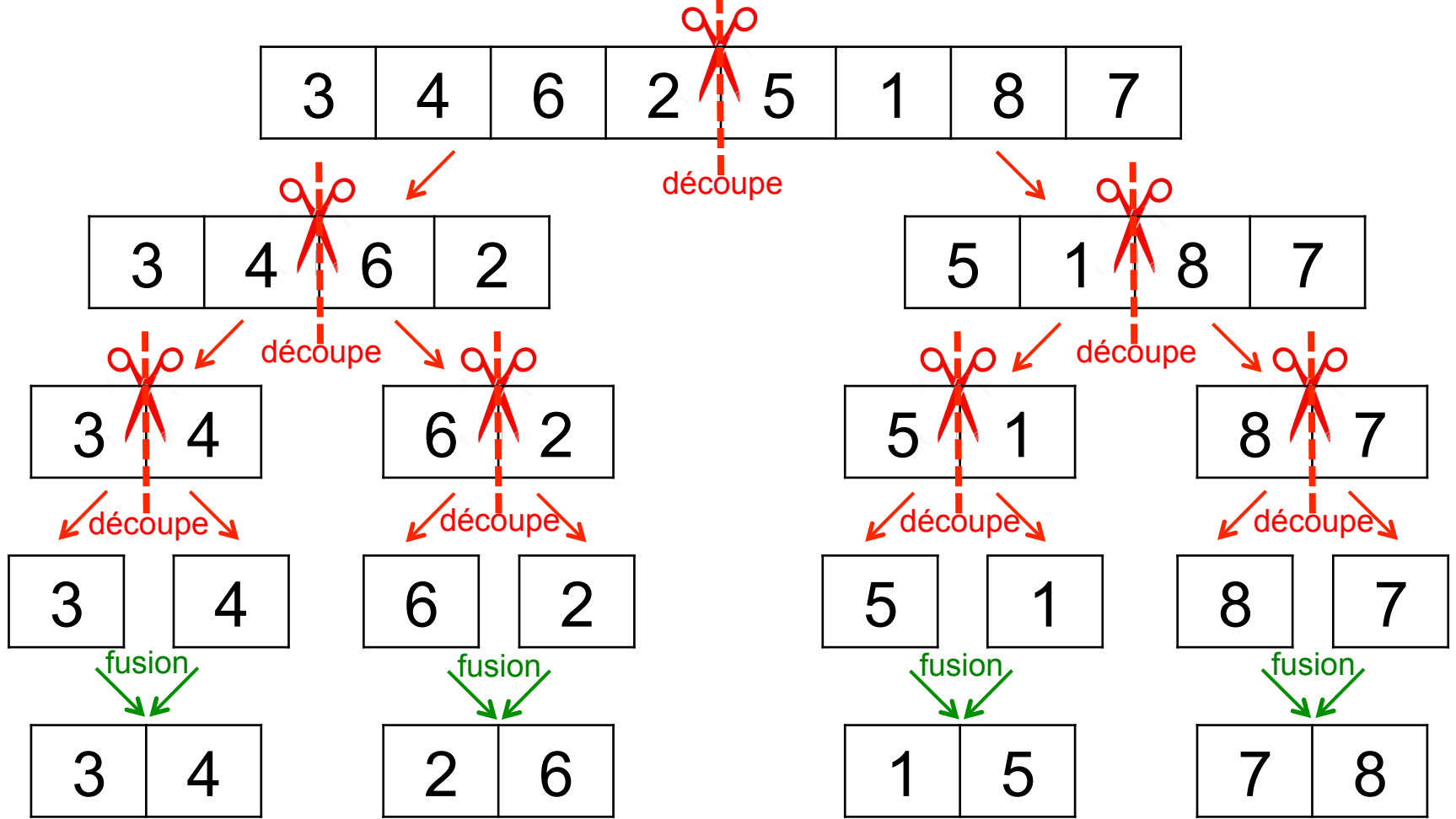
Nous allons maintenant étudier un algorithme de tri basé sur le principe **diviser pour régner** : **le tri fusion** (*en anglais Merge sort*, inventé par John von Neumann en 1945)

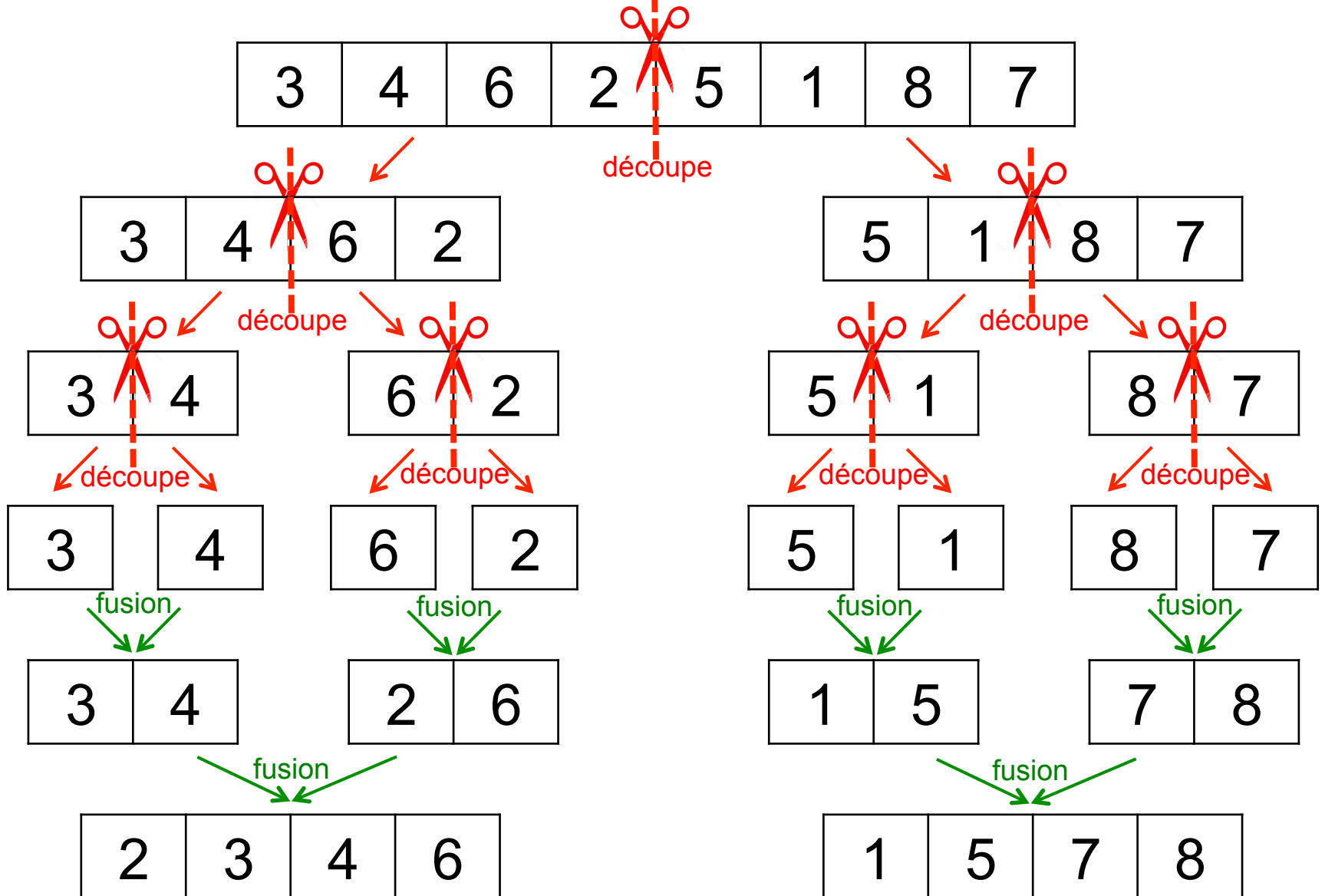
3	4	6	2	5	1	8	7
---	---	---	---	---	---	---	---

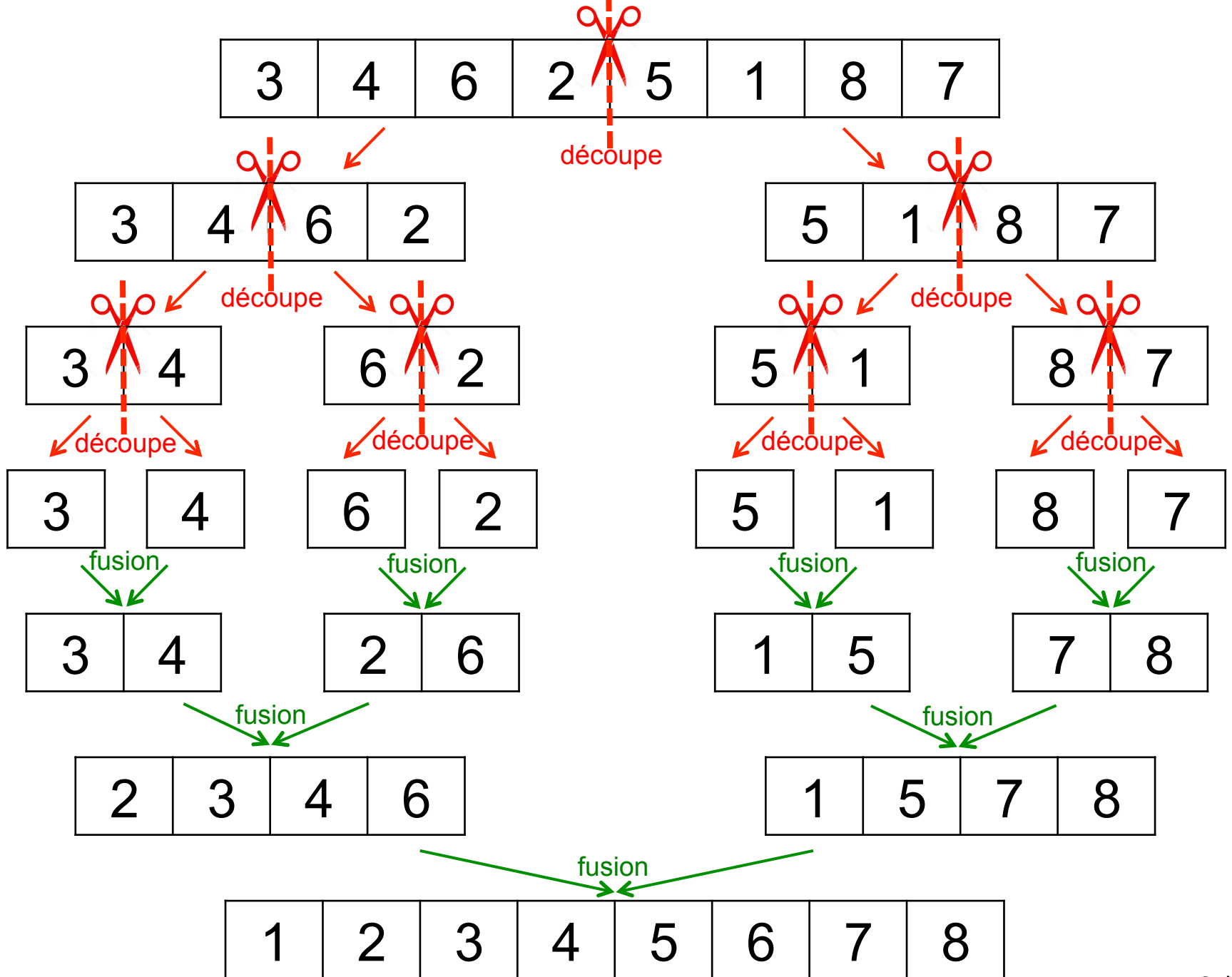




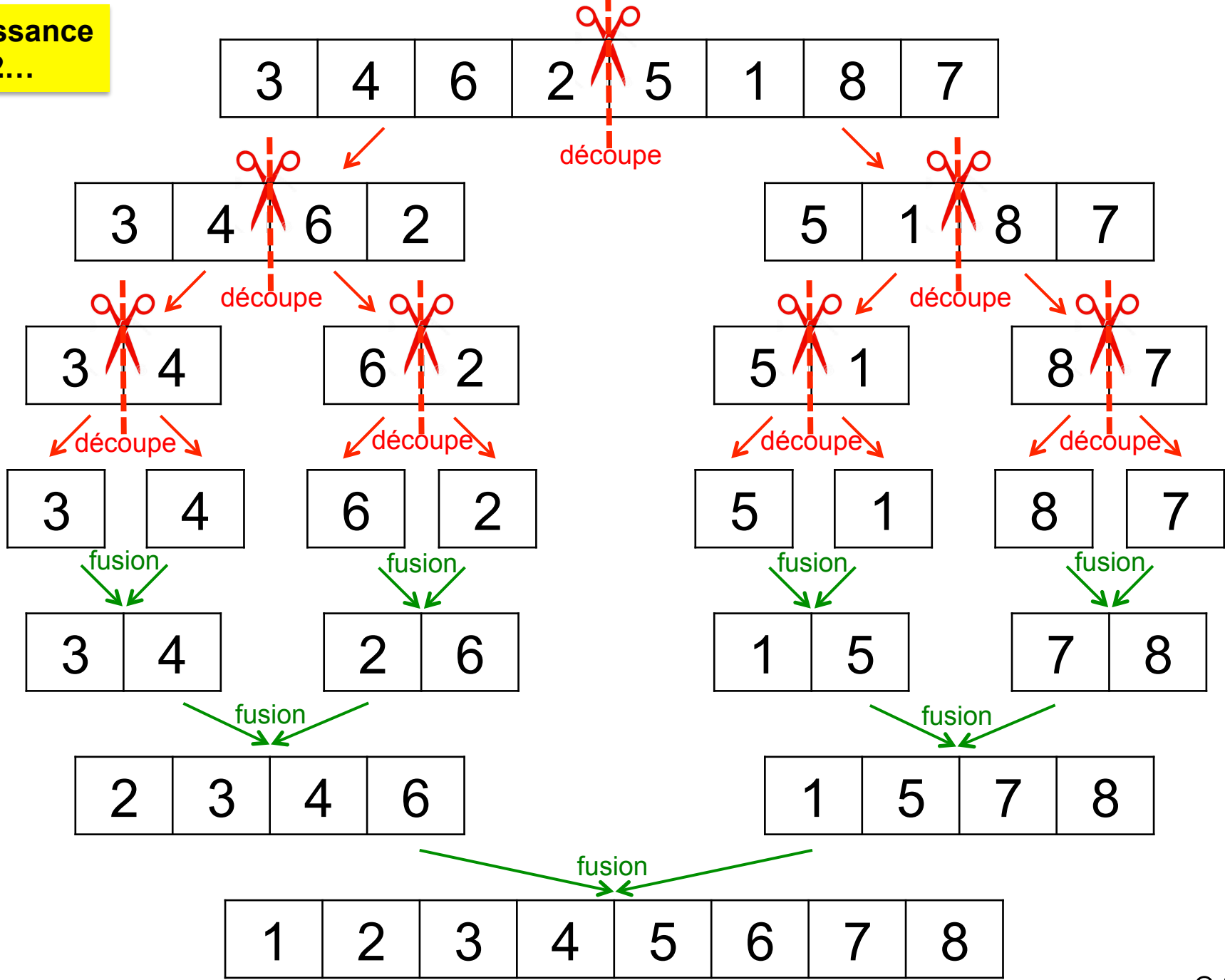




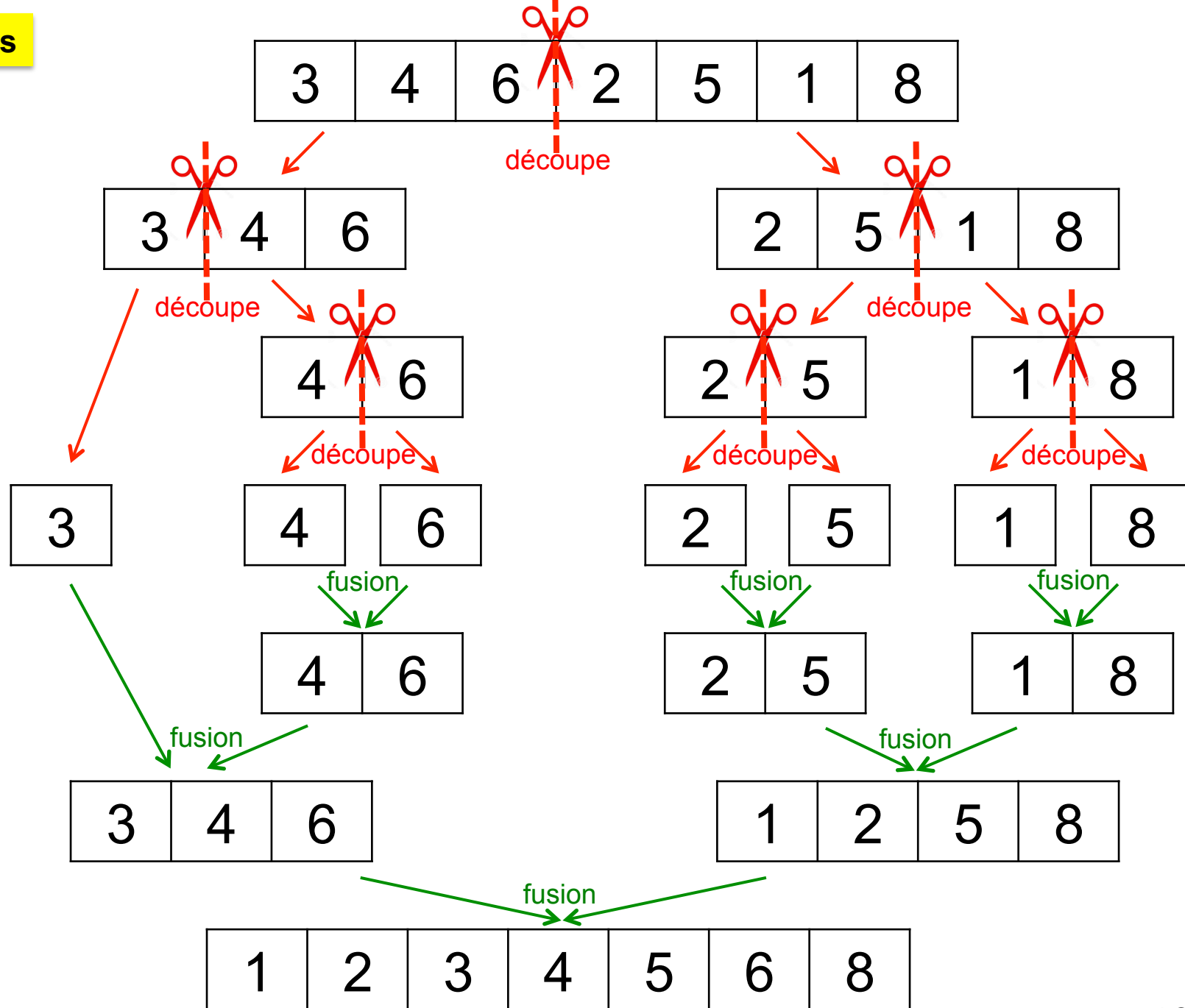




Puissance
de 2...



... ou pas



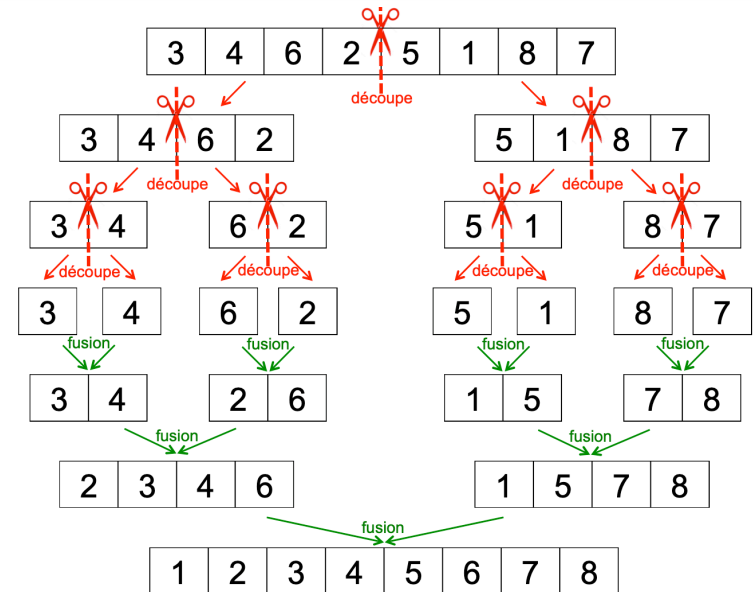
Chapitre 6 - Diviser pour régner

Séance 1 - Tri fusion

Principe du tri fusion

Le tri fusion consiste à **trier récursivement** les deux moitiés de la liste, puis à **fusionner** ces deux sous-listes triées en une seule.

La **condition d'arrêt** à la récursivité sera l'obtention d'une liste à un **seul élément**, car une telle liste est évidemment déjà triée.



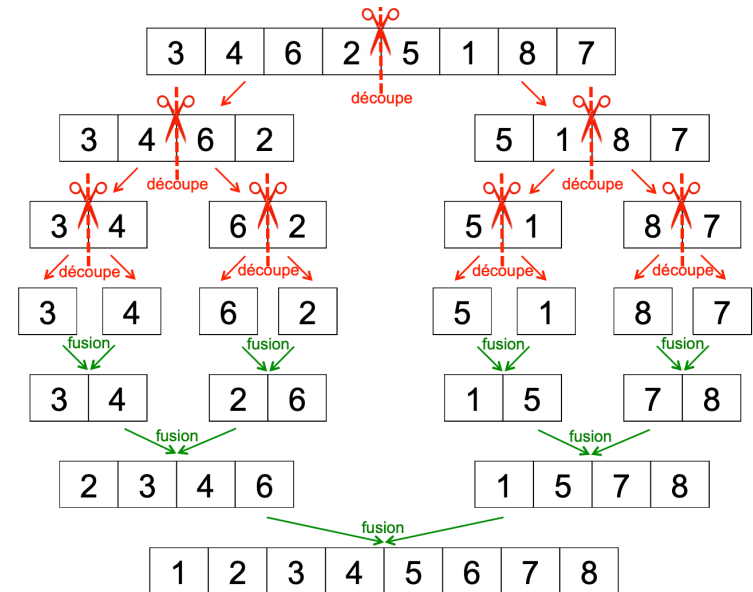
Chapitre 6 - Diviser pour régner

Séance 1 - Tri fusion

Principe du tri fusion

Le tri fusion consiste à **trier récursivement** les deux moitiés de la liste, puis à **fusionner** ces deux sous-listes triées en une seule.

La **condition d'arrêt** à la récursivité sera l'obtention d'une liste à un **seul élément**, car une telle liste est évidemment déjà triée.



On retrouve les trois étapes du **diviser pour régner** :

- **DIVISER** Diviser la liste en deux sous-listes de même taille (à un élément près) en la "**coupant**" par la moitié
- **REGNER** Trier **récursivement** chacune de ces deux sous-listes et arrêter la récursion lorsque les listes n'ont plus qu'un **seul élément**
- **COMBINER** **Fusionner** les deux sous-listes triées en une seule

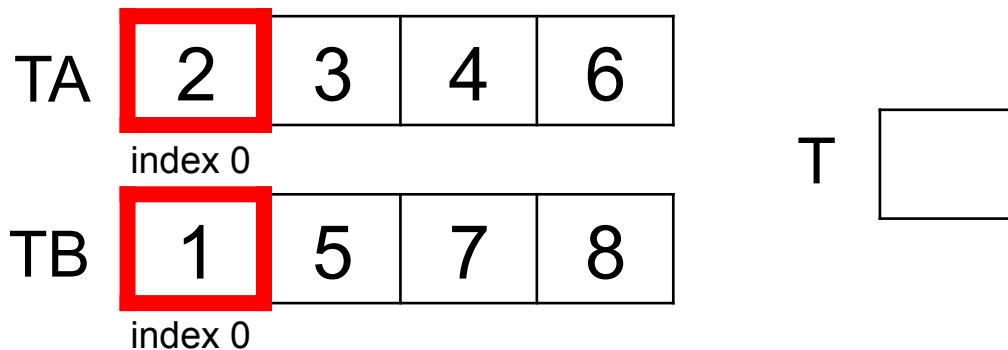
Principe de la fusion

L'idée qui permet d'avoir une **fusion** efficace repose sur le fait que les deux listes TA et TB sont **triées**.

Il suffit en fait de les parcourir dans l'ordre : on sait que les plus petits éléments des deux listes sont au début, et le plus petit élément de la liste globale est forcément soit le plus petit élément de TA, soit le plus petit élément de TB (c'est le plus petit des deux).

Une fois qu'on l'a déterminé, on le retire de la demi-liste dans laquelle il se trouve, et on recommence à regarder les éléments (restants) du début.

Une fois qu'on a « vidé » TA et TB, on a bien effectué la fusion.



TA

2	3	4	6
---	---	---	---

index 0

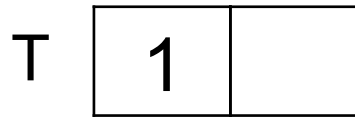
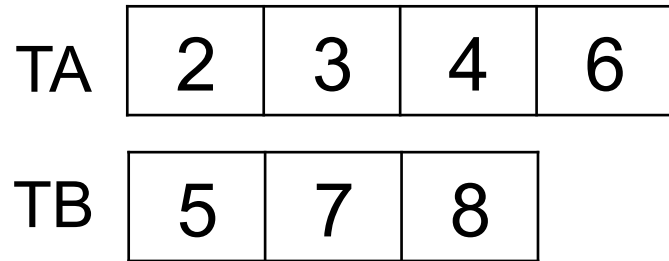
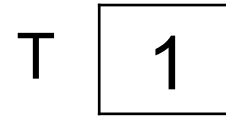
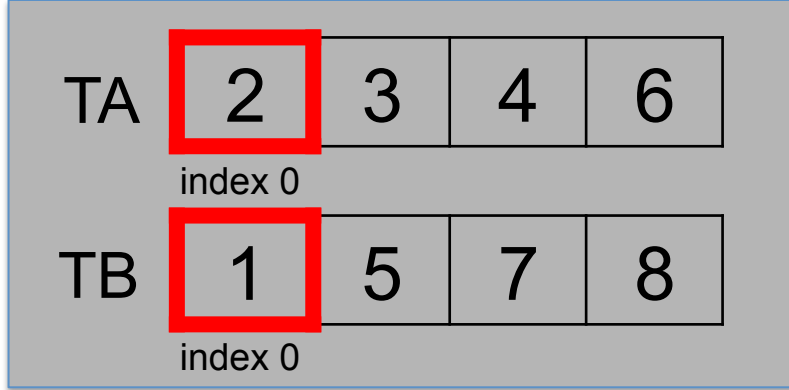
TB

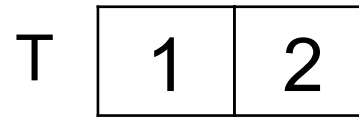
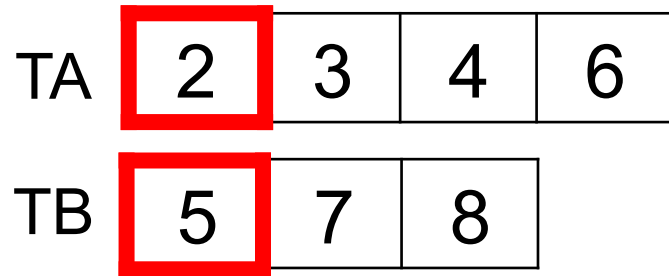
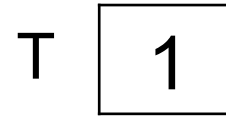
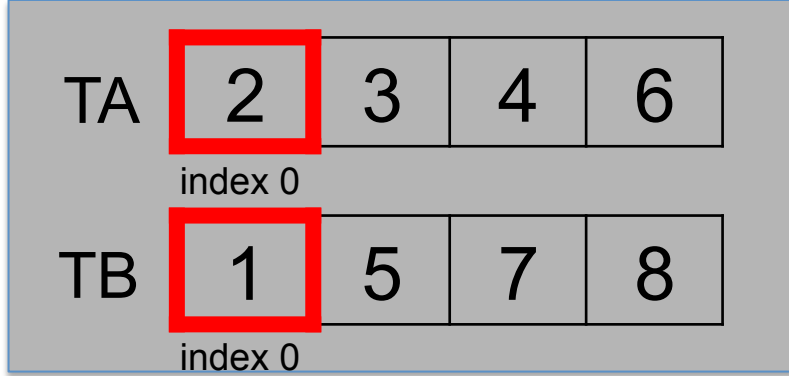
1	5	7	8
---	---	---	---

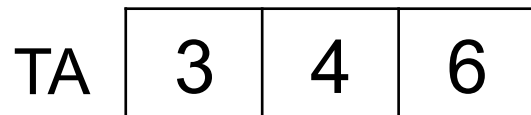
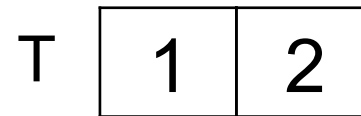
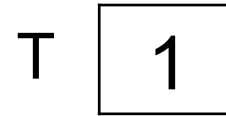
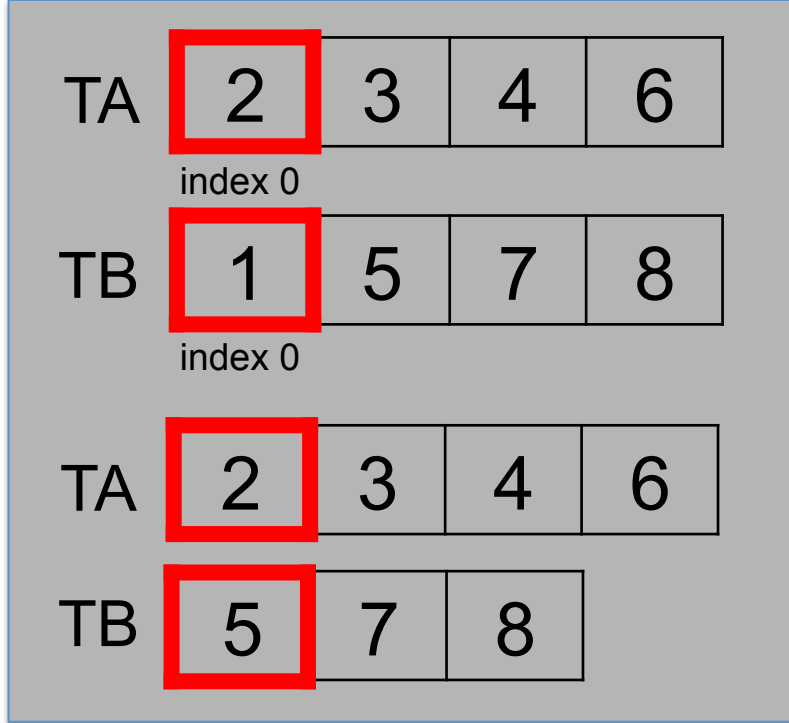
index 0

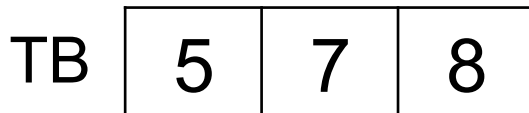
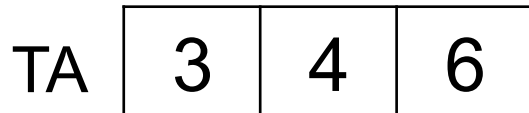
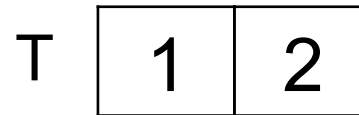
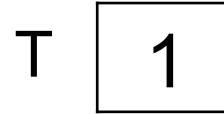
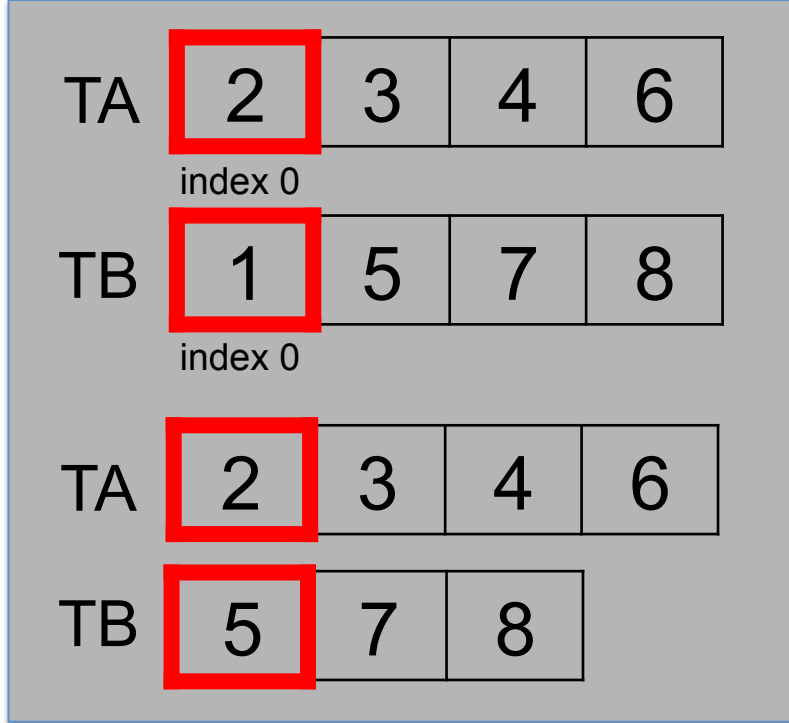
T

1









**Et ainsi de suite... jusqu'à l'un des deux soit vide,
pour ensuite concaténer le reste.**



Exercice

Proposez un algorithme et une implémentation pour **fusion**.
(Vous pouvez utiliser la méthode **pop** des tableaux python)





Exercice

Proposez un algorithme et une implémentation pour **fusion**.
(Vous pouvez utiliser la méthode **pop** des tableaux)

Algorithme

fonction **fusion**(TA, TB) :

- $T \leftarrow$ tableau vide
- Tant que TA non vide **ET** TB non vide :
 - Si le 1^{er} élément de TA est strictement inférieur au 1^{er} élément de TB :
 - Ajouter le 1^{er} élément de TA dans T
 - Supprimer le 1^{er} élément dans TA
 - Sinon :
 - Ajouter le 1^{er} élément de TB dans T
 - Supprimer le 1^{er} élément dans TB
- On renvoie T + TA + TB



Exercice

Proposez un algorithme et une implémentation pour **fusion**.
(Vous pouvez utiliser la méthode **pop** des tableaux)

Algorithme

fonction **fusion**(TA, TB) :

- $T \leftarrow$ tableau vide
- Tant que TA non vide **ET** TB non vide :
 - Si le 1^{er} élément de TA est strictement inférieur au 1^{er} élément de TB :
 - Ajouter le 1^{er} élément de TA dans T
 - Supprimer le 1^{er} élément dans TA
 - Sinon :
 - Ajouter le 1^{er} élément de TB dans T
 - Supprimer le 1^{er} élément dans TB
- On renvoie T + TA + TB

Implémentation Python

```
def fusion(TA, TB):  
    T = []  
    while TA != [] and TB != []:  
        if TA[0] < TB[0]:  
            T.append(TA.pop(0))  
        else:  
            T.append(TB.pop(0))  
    return T + TA + TB
```



Python 3.6
([known limitations](#))

```
1 def fusion(TA, TB):  
2     T=[]  
→ 3     while TA!=[] and TB!=[]:  
4         if TA[0] < TB[0]:  
→ 5             T.append(TA.pop(0))  
6         else:  
7             T.append(TB.pop(0))  
8     return T + TA + TB  
9 fusion([2,5], [8,9])  
10 fusion([2,7], [3,5])
```

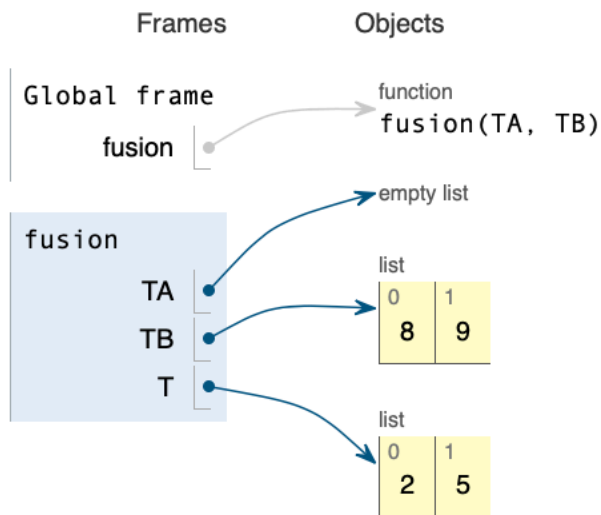
[Edit this code](#)

→ line that just executed
→ next line to execute

<< First < Prev Next > Last >>

Step 11 of 28

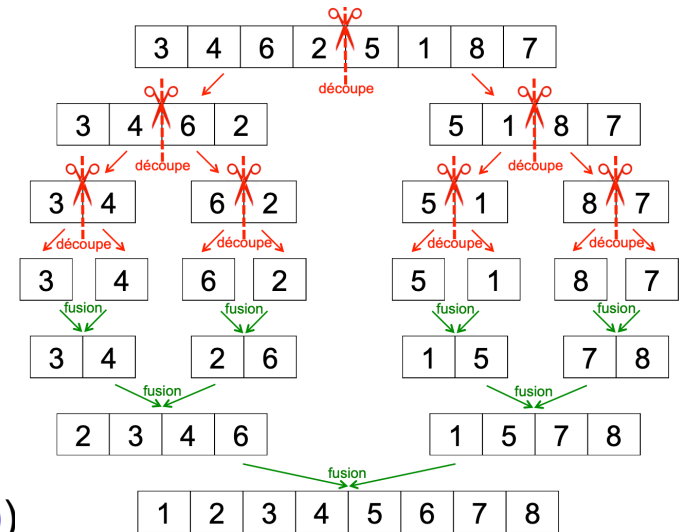
[customize visualization](#) (NEW!)



Algorithme du tri fusion

fonction **tri_fusion(T)** :

- Si la taille du tableau T est inférieure ou égale à 1 :
 - On renvoie le tableau (un seul élément)
- TA ← **la première moitié de T**
- TB ← **la seconde moitié de T**
- On renvoie la **fusion**(**tri_fusion(TA)**, **tri_fusion(TB)**)



Implémentation

```
def tri_fusion(T):
```

```
    if len(T) <= 1 :                # fin de la récursion si vide ou à
        return T                    un seul élément
```

```
    else:
```

```
        m = len(T) // 2             # position pour couper au milieu
```

```
        TA = tri_fusion(T[:m])      # on trie la 1ère moitié récursivement
```

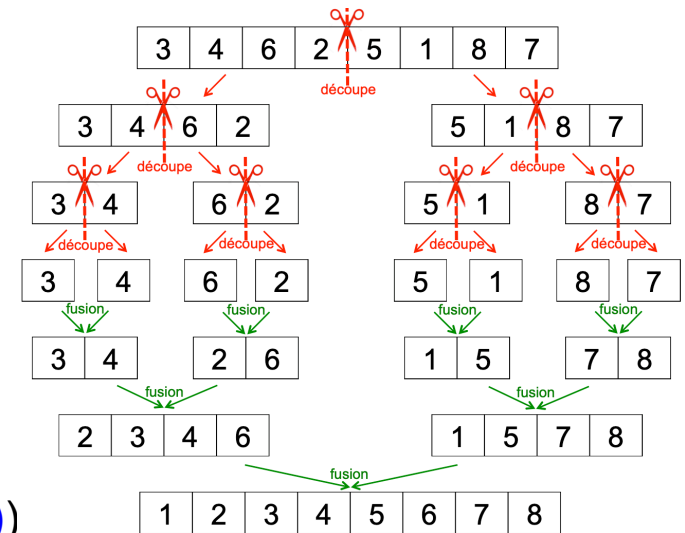
```
        TB = tri_fusion(T[m:])      # on trie la 2nde moitié récursivement
```

```
        return fusion(TA, TB)       # on renvoie la fusion des moitiés triées
```

Algorithme du tri fusion

fonction **tri_fusion(T)** :

- Si la taille du tableau T est inférieure ou égale à 1 :
 - On renvoie le tableau (un seul élément)
- TA ← **la première moitié de T**
- TB ← **la seconde moitié de T**
- On renvoie la **fusion(tri_fusion(TA), tri_fusion(TB))**



Implémentation

def **tri_fusion(T)**:

```
    if len(T) <= 1 :                # fin de la récursion si vide ou à  
        F return T                  un seul élément
```

```
    else:
```

```
        1 m = len(T) // 2           # position pour couper au milieu  
        2 TA = tri_fusion(T[:m])    # on trie la 1ère moitié récursivement  
        3 TB = tri_fusion(T[m:])    # on trie la 2nde moitié récursivement  
        4 return fusion(TA, TB)     # on renvoie la fusion des moitiés triées
```


Appels successifs

```
tri_fusion([3, 4, 6, 2, 5, 1, 8, 7])
```

```
1 m=4
```

```
2 TA = tri_fusion([3, 4, 6, 2])
```

```
3 TB = tri_fusion([5, 1, 8, 7])
```

```
4 fusion(TA, TB)
```



Exercice

Poursuivre le déroulement du programme.

Appels successifs

```
tri_fusion([3, 4, 6, 2, 5, 1, 8, 7])
```

```
  m=4
```

```
  TA = tri_fusion([3, 4, 6, 2])
```

```
    1 m = 2
```

```
    2 TA = tri_fusion([3, 4])
```

```
    3 TB = tri_fusion([6, 2])
```

```
    4 fusion(TA, TB)
```

```
  TB = tri_fusion([5, 1, 8, 7])
```

```
fusion(TA, TB)
```

Appels successifs

```
tri_fusion([3, 4, 6, 2, 5, 1, 8, 7])
```

```
    m=4
```

```
    TA = tri_fusion([3, 4, 6, 2])
```

```
        m = 2
```

```
        TA = tri_fusion([3, 4])
```

```
            1 m = 1
```

```
            2 TA = tri_fusion([3])
```

```
            3 TB = tri_fusion([4])
```

```
            4 fusion(TA, TB)
```

```
        TB = tri_fusion([6, 2])
```

```
    fusion(TA, TB)
```

```
    TB = tri_fusion([5, 1, 8, 7])
```

```
fusion(TA, TB)
```

Appels successifs

```
tri_fusion([3, 4, 6, 2, 5, 1, 8, 7])
  m=4
  TA = tri_fusion([3, 4, 6, 2])
    m = 2
    TA = tri_fusion([3, 4])
      m = 1
      TA = tri_fusion([3]) F -> return 3
      TB = tri_fusion([4]) F -> return 4
      fusion(TA, TB)      -> return [3, 4]
    TB = tri_fusion([6, 2])
```

```
      fusion(TA, TB)
TB = tri_fusion([5, 1, 8, 7])
```

```
fusion(TA, TB)
```

Appels successifs

```
tri_fusion([3, 4, 6, 2, 5, 1, 8, 7])  
  m=4  
  TA = tri_fusion([3, 4, 6, 2])  
    m = 2  
    TA = tri_fusion([3, 4])  
      m = 1  
      TA = tri_fusion([3]) F -> return 3  
      TB = tri_fusion([4]) F -> return 4  
      fusion(TA, TB)      -> return [3, 4]  
    TB = tri_fusion([6, 2])
```

Et ainsi de suite ...

```
  fusion(TA, TB)  
TB = tri_fusion([5, 1, 8, 7])
```

```
fusion(TA, TB)
```

Appels successifs

```
tri_fusion([3, 4, 6, 2, 5, 1, 8, 7])
  m=4
  TA = tri_fusion([3, 4, 6, 2])
    m = 2
    TA = tri_fusion([3, 4])
      m = 1
      TA = tri_fusion([3]) -> return 3
      TB = tri_fusion([4]) -> return 4
      fusion(TA, TB) -> return [3, 4]
    TB = tri_fusion([6, 2])
      m = 1
      TA = tri_fusion([6]) -> return 6
      TB = tri_fusion([2]) -> return 2
      fusion(TA, TB) -> return [2, 6]
    fusion(TA, TB) -> return [2, 3, 4, 6]
  TB = tri_fusion([5, 1, 8, 7])
    m = 2
    TA = tri_fusion([5, 1])
      m = 1
      TA = tri_fusion([5]) -> return 5
      TB = tri_fusion([1]) -> return 1
      fusion(TA, TB) -> return [1, 5]
    TB = tri_fusion([8, 7])
      m = 1
      TA = tri_fusion([8]) -> return 8
      TB = tri_fusion([7]) -> return 7
      fusion(TA, TB) -> return [7, 8]
    fusion(TA, TB) -> return [1, 5, 7, 8]
  fusion(TA, TB) -> return [1, 2, 3, 4, 5, 6, 7, 8]
```

..... la suite dans le notebook
tri_fusion.ipynb



Complexité du tri fusion

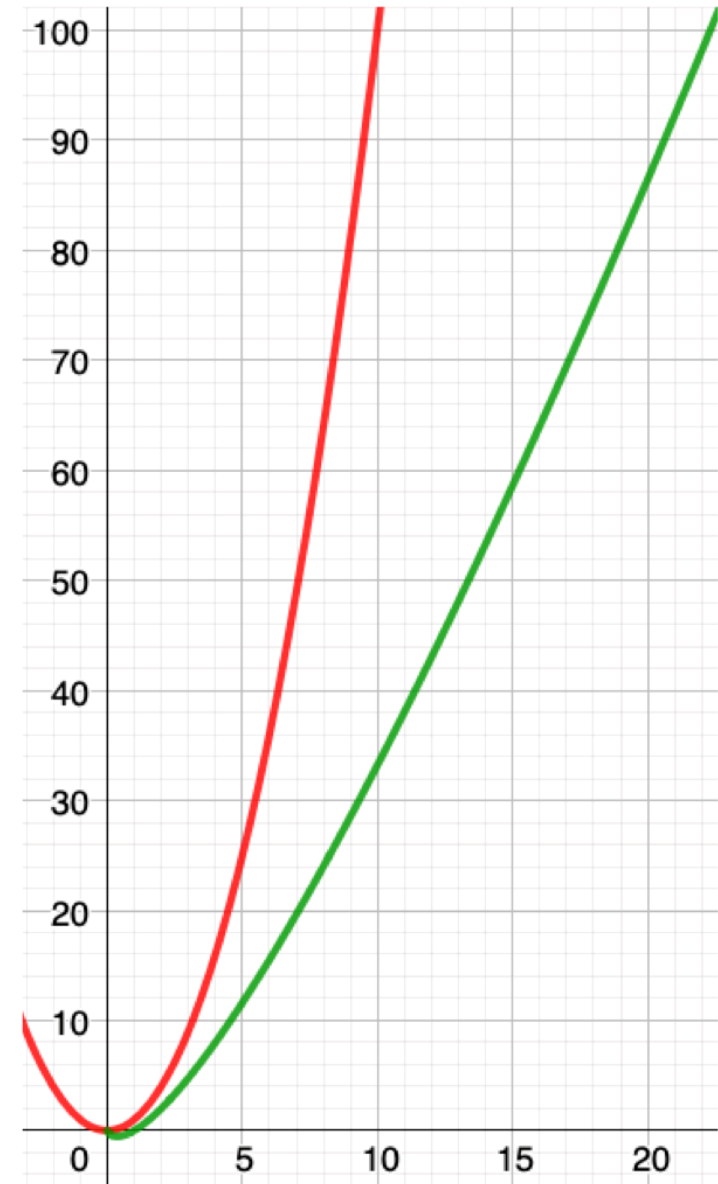
Le calcul rigoureux de la complexité de cet algorithme sort du cadre de ce cours.

Mais, en remarquant que la première phase (DIVISER) consiste à **"couper"** les tableaux en deux plusieurs fois de suite, intuitivement, on peut dire qu'un **logarithme base 2** doit intervenir.

La deuxième phase consiste à faire des **comparaisons** entre les premiers éléments de chaque tableau à fusionner, on peut donc supposer que pour un tableau de **n éléments**, on aura n comparaisons.

En combinant ces 2 constatations on peut donc dire que la complexité du tri fusion est en $O(n \cdot \log n)$.

La comparaison des courbes de la fonction n^2 (en rouge) et $n \cdot \log n$ (en vert) nous montre que l'algorithme de tri fusion est plus "efficace" que l'algorithme de tri par insertion ou que l'algorithme de tri par sélection.



Les différentes complexités

La complexité en temps est notée $O(\text{fonction})$, où *fonction* est réduite «au plus simple», en prenant uniquement la partie à la croissance la plus rapide, et en supprimant éventuellement les constantes multiplicatives.

Les complexités les plus fréquentes sont :

- **Complexité constante** $O(1)$:

le temps d'exécution est constant quelle que soit la taille des données.

- **Complexité logarithmique** $O(\log n)$:

schématiquement, si on multiplie la taille des données par 2, alors on augmente le nombre d'opérations de 1.

- **Complexité linéaire** $O(n)$:

si on augmente la taille des données de n , on augmente le temps de traitement de n .

- **Complexité log-linéaire ou quasi-linéaire**

$O(n \cdot \log n)$: augmente un peu plus vite que linéaire, mais beaucoup moins que quadratique.

- **Complexité quadratique** $O(n^2)$:

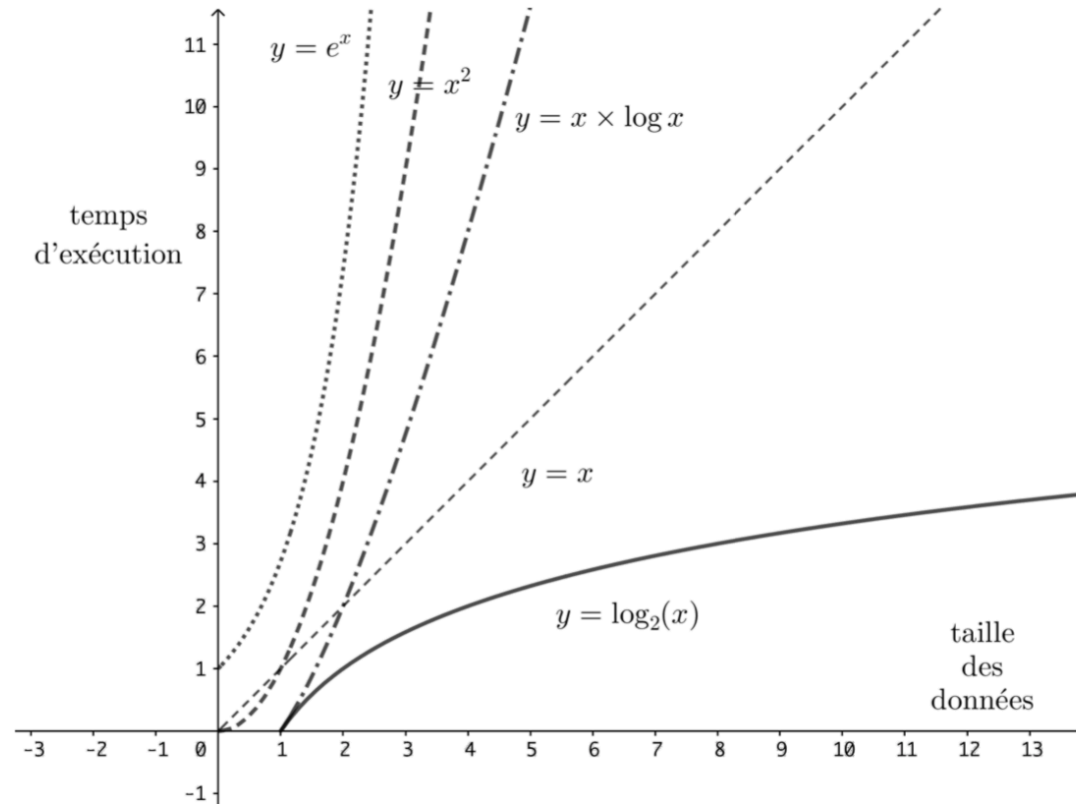
si on multiplie la taille des données par 2, on multiplie le temps par 4.

- **Complexité cubique** $O(n^3)$:

si on multiplie la taille des données par 2, on multiplie le temps par 8.

- **Complexité exponentielle** $O(2^n)$ ou $O(e^n)$:

algorithmes très lents.



Pour aller plus loin

- Tri rapide (*Quicksort*)
- Démonstrations animées d'algorithmes de tri, et implémentations en plusieurs langages sur le site <http://lwh.free.fr/pages/algo/tri/tri.htm>