Introduction à l'informatique

pour les mathématiques, la physique et les sciences computationnelles

Yann Thorimbert

Partie 2 | Chapitre 3
Tri fusion et récursivité

Yann Thorimbert

Chapitres du cours (seconde partie du cours)

0. Introduction à la complexité algorithmique

1. Algorithmes de tri naïfs

2. Structures de données : tableaux, listes et dictionnaires

3. Tri fusion et récursivité ←

4. Algorithmes de recherche au sein d'une séquence

5. Algorithmes sur graphes

Tri à bulles | **Rappel**

- Le tri à bulle est en moyenne $O(n^2)$.
- Contrairement au tri sélection, le meilleur des cas est $O(n)$.
- En pratique, le tri à bulles est également peu utilisé.
- Spoiler : il existe des tris dont la complexité dans le pire cas est plus faible que $O(n^2)$. Pour le comprendre, nous allons devoir nous intéresser à une stratégie algorithmique différente de celles considérées jusqu'ici.

Algorithmes récursifs

- Le tri que nous nous apprêtons à voir se formule bien **récursivement**.
  Il est basé sur une **division des tâches**, comme bien d'autres algorithmes d'une famille de stratégies nommée "Divide & Conquer".
- Algorithme récursif : algorithme qui fait référence à lui-même.
  Un tel algorithme, pour se terminer, doit donc inclure une **condition de fin**.
- Avant de voir le tri fusion, il est utile de s'habituer au concept d'algorithme récursif. Nous allons faire cela au travers de l'algorithme permettant de calculer une factorielle, à titre d'exemple.

Calcul de factorielle (exemple d'algorithme récursif)

Nous utilisons Python en lieu et place d'un pseudocode :

Calcul de factorielle (exemple d'algorithme récursif)

Nous utilisons Python en lieu et place d'un pseudocode :

```
def fact(x):
    if x == 0:
        return 1
    else:
        return x * fact(x-1)
```

Calcul de factorielle (exemple d'algorithme récursif)

Exemple de factorielle : fact(4) = 4! = 4 · 3 · 2 · 1

```
def fact(x):
    if x == 0:
        return 1
    else:
        return x * fact(x-1)
```

Condition de fin

Condition de récursion

Appel récursif

Calcul de factorielle (exemple d'algorithme récursif)

Examinons les étapes de calcul pour un cas concret.

```
r = fact(4)
  = ...
```

```
def fact(x):
    if x == 0:
        return 1
    else:
        return x * fact(x-1)
```

Calcul de factorielle (exemple d'algorithme récursif)

Examinons les étapes de calcul pour un cas concret.

```
def fact(x):
    if x == 0:
        return 1
    else:
        return x * fact(x-1)
```

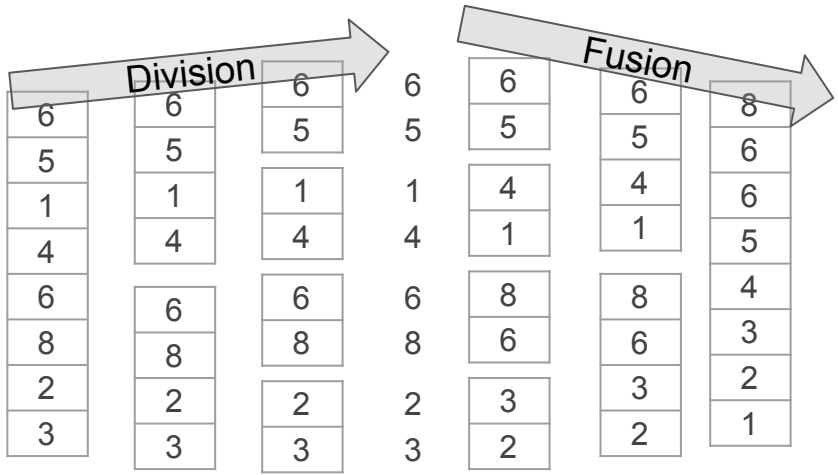
```
r = fact(4)
  = 4 * fact(3)
  = 4 * 3 * fact(2)
  = 4 * 3 * 2 * fact(1)
  = 4 * 3 * 2 * 1 * fact(0)
  = 4 * 3 * 2 * 1 * 1
  = 24
```

Algorithme de tri fusion | **Idée de l'algorithme**

- **Subdiviser** récursivement le problème, jusqu'à n'obtenir que des séquences extrêmement simples à trier.
- Ensuite, **fusionner** ces séquences simples entre elles, chacune étant déjà triée.
- Subdivision : l'algorithme sépare la séquence reçue en deux parties, à moins que cette séquence ne contienne qu'un seul élément.
- La fusion nécessite davantage de réflexion.
- Visualisation :

Algorithme de tri fusion | **Idée de l'algorithme**

- Il faut de l'ordre de log(n) étapes pour parvenir au cas central composé de n listes élémentaires.
- Si l'étape de recombinaison (fusion) est mieux que $O(n^2)$, nous aurons trouvé notre nouveau meilleur algorithme de tri.

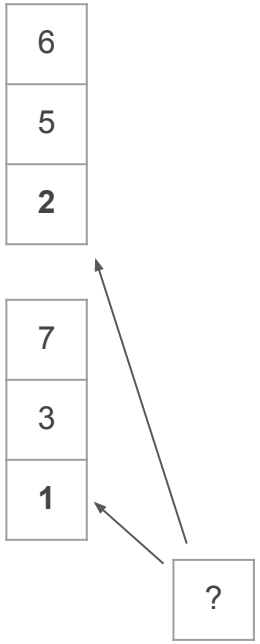


Tri Fusion | **Fusion de deux séquences déjà triées**

- Fusion(A, B) retourne une séquence avec les éléments de A et B triés.
- Cas trivial : l'une des deux séquences est vide. Dans ce cas, on retourne l'unique séquence non-vide (cas de fin).
- Cas non trivial :
    - On compare le premier élément de chaque séquence. Le plus petit d'entre eux est forcément le plus petit du tout.
    - On retourne une séquence dont le premier élément est m, et dont les éléments suivants sont le résultat de la fusion des nouvelles sous-séquences (cas récursif).

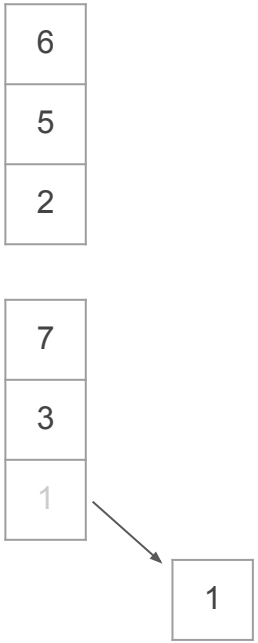
Fusion de deux séquences déjà triées

Fusion de deux séquences déjà triées



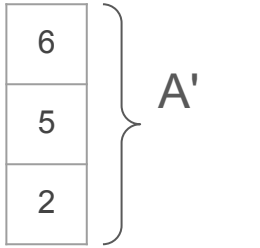
On garde le plus petit des premiers éléments. Ce sera le premier de la séquence fusionnée.

Fusion de deux séquences déjà triées



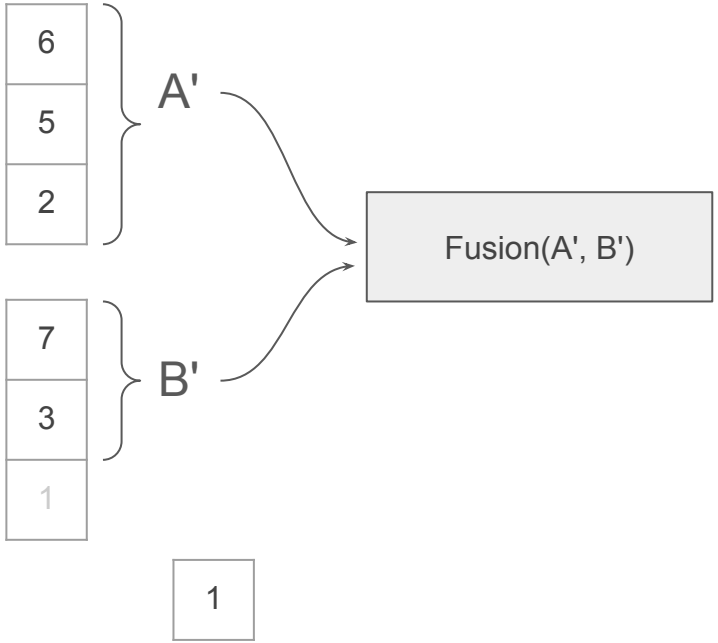
On garde le plus petit des premiers éléments. Ce sera le premier de la séquence fusionnée.

Fusion de deux séquences déjà triées



Ce choix définit A' et B'.

Fusion de deux séquences déjà triées



On fusionne les sous-séquences A' et B'.

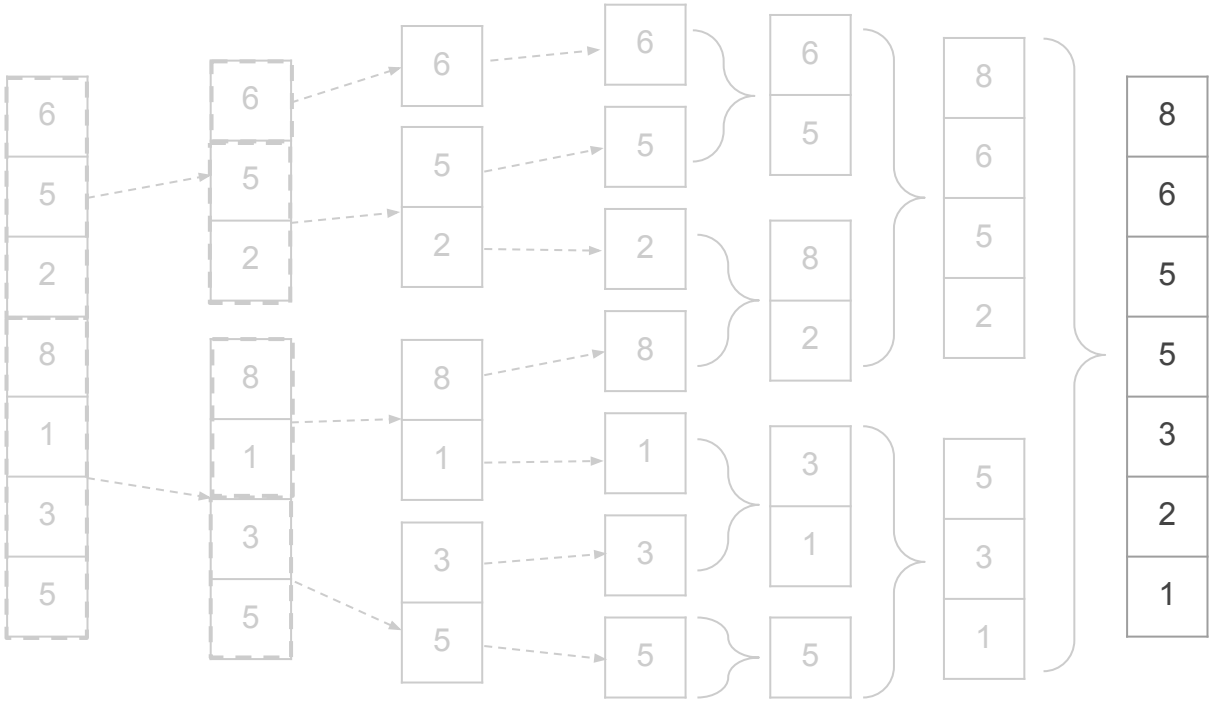
Cf. slides spéciales pour l'**illustration pas à pas** de l'algorithme.

Utilisation de l'algorithme de fusion

- L'algorithme de fusion nous offre la garantie de pouvoir assembler autant de sous-séquences triées que l'on veut en une seule grande séquence triée, et ceci en $O(n)$.
- Question : comment aboutir à la situation où l'on possède ces sous-séquences triées ?
- Réponse : on peut diviser (simplifier) la tâche autant que nécessaire, jusqu'à aboutir uniquement à des cas triviaux, et donc triés.

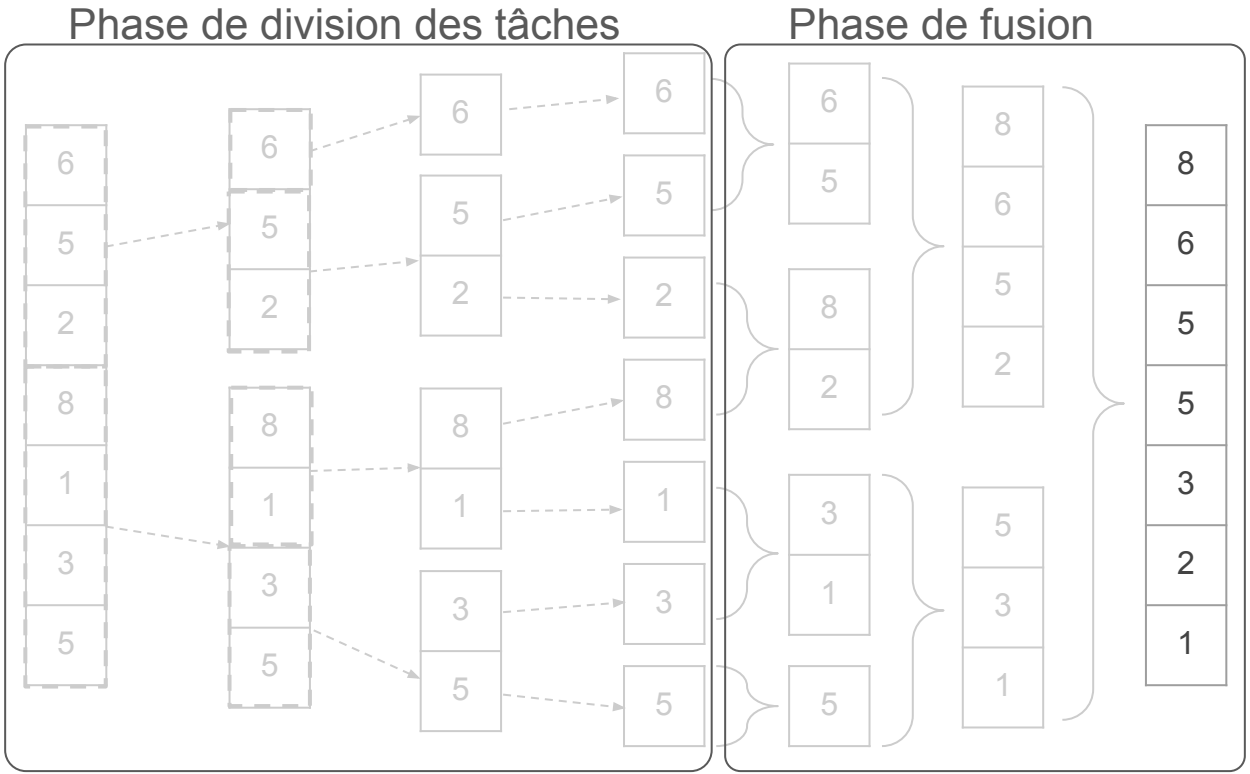
Tri fusion | **Exemple**

Cf. slides à part pour l'**illustration pas à pas**



Tri fusion | **Complexité**

Tri fusion | **Exemple**



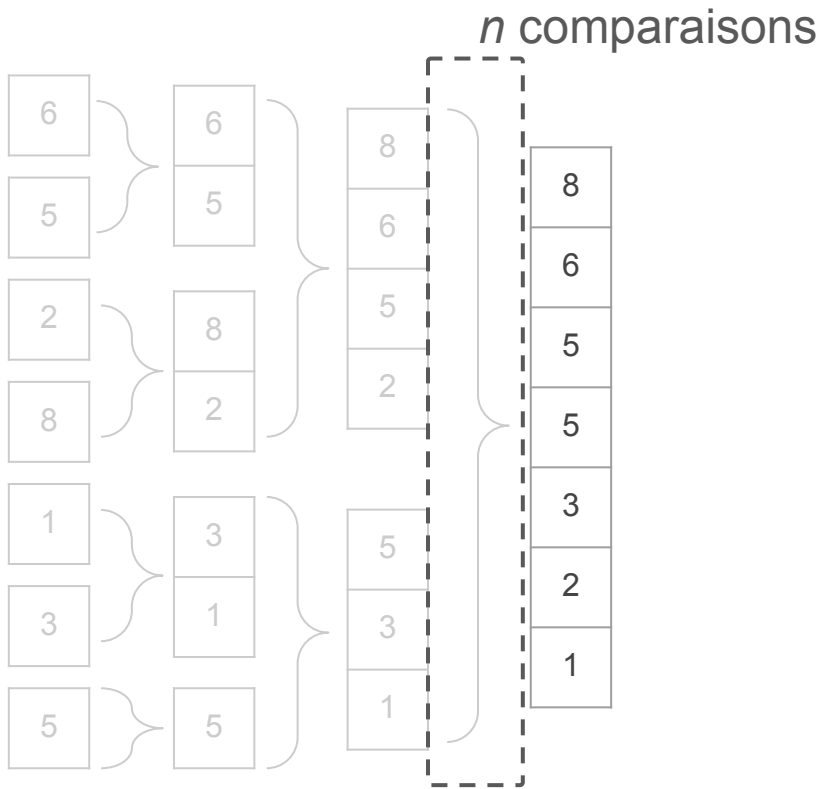
Complexité du tri fusion | **Phase de division**

À chaque étape, on divise la séquence en **deux**. Cela nécessite donc $\log_2(n)$ étapes de division.

Complexité du tri fusion | **Pourquoi $\log_2(n)$ récursions ?**

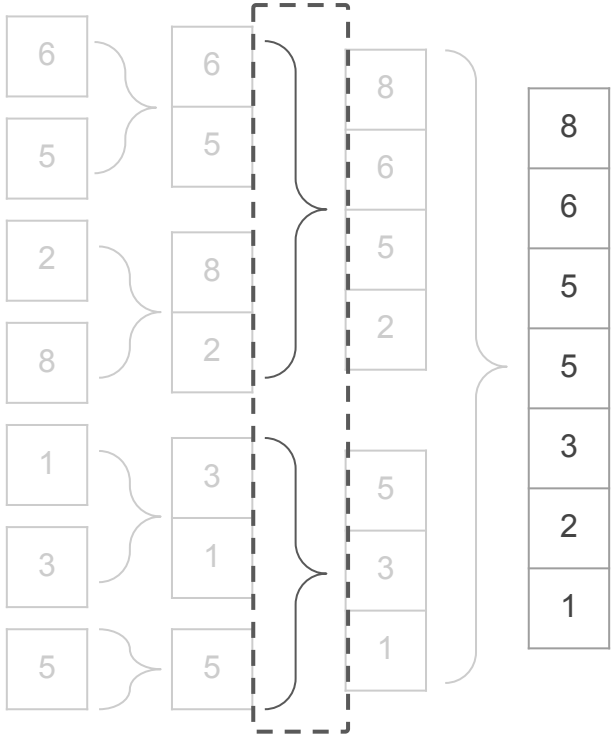
- À chaque niveau de récursion, on divise les tableaux en **deux**. Cela nécessite donc $\log_2(n)$ étapes de division.
- En effet, dans le cas le plus évident, on double le nombre de tableaux à chaque étape. S'il y a k étapes, on produit n tableaux : $n = 2^k$, donc $k = \log_2(n)$.
- Si n n'est pas une puissance de deux cela ne change rien à la complexité : $n = 2^k + c \Leftrightarrow k = \log_2(n-c)$ est $O(\log(n))$ également.

Complexité du tri fusion | **Phase de fusion**



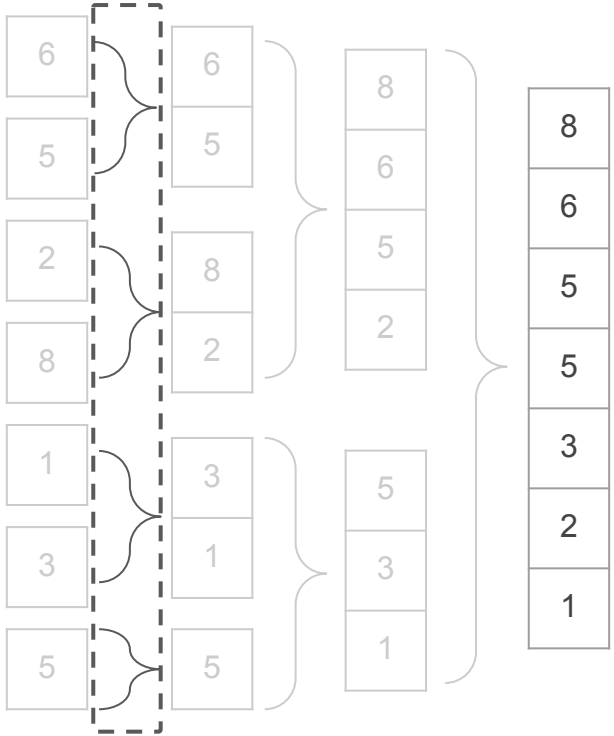
Complexité du tri fusion | **Phase de fusion**

$2 \cdot n / 2$ comparaisons



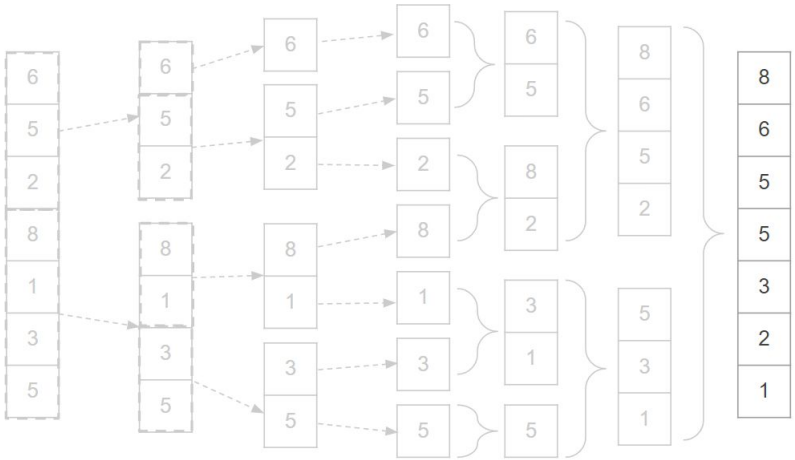
Complexité du tri fusion | **Phase de fusion**

$4 \cdot n / 4$ comparaisons



Complexité du tri fusion

- Chaque **division** est *O*(n) car il y a n éléments à copier, et il y a $\log_2(n)$ divisions.
- Chaque **fusion** est *O*(n), et il y a $\log_2(n)$ fusions.
- L'algorithme est donc **O(n · log(n))**, ce qui est bien **mieux** que le tri à bulles ou le tri sélection !



Complexité du tri fusion | **Note sur l'écriture de *O*(log(n))**

Rappelons que $\log_a(x) = \log_b(x) / \log_b(a)$: une quantité logarithmique exprimée dans une base ne diffère que d'un **facteur constant** par rapport à la même quantité dans une autre base. Comme nous l'avons vu, dans l'analyse de complexité asymptotique, ces facteurs constants ne nous importent pas.

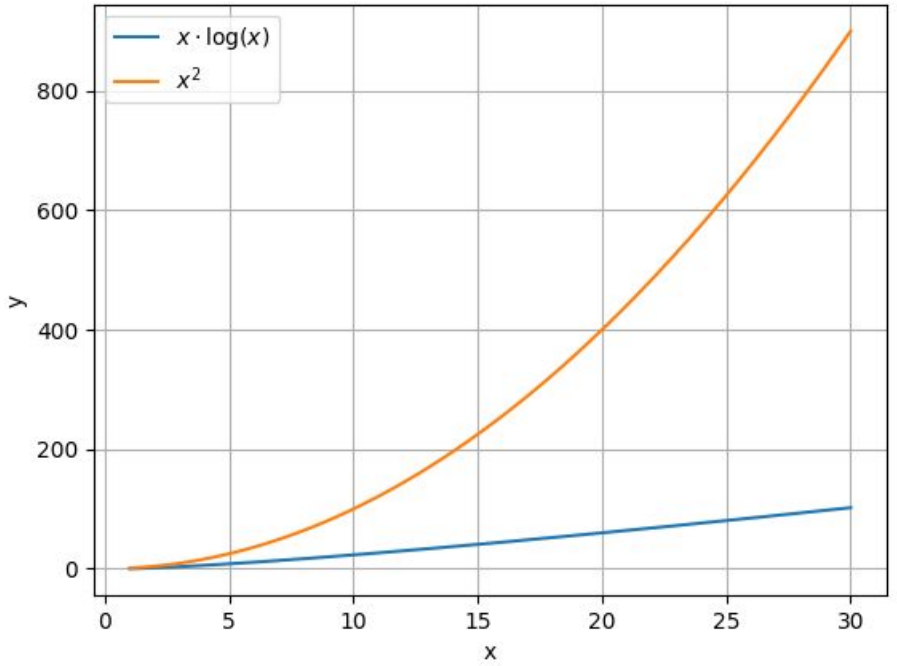
⇒ On écrit O(n log(n)) **sans spécifier la base** du logarithme.

Complexité en temps du tri fusion

Tri fusion : $O(n \cdot \log(n))$

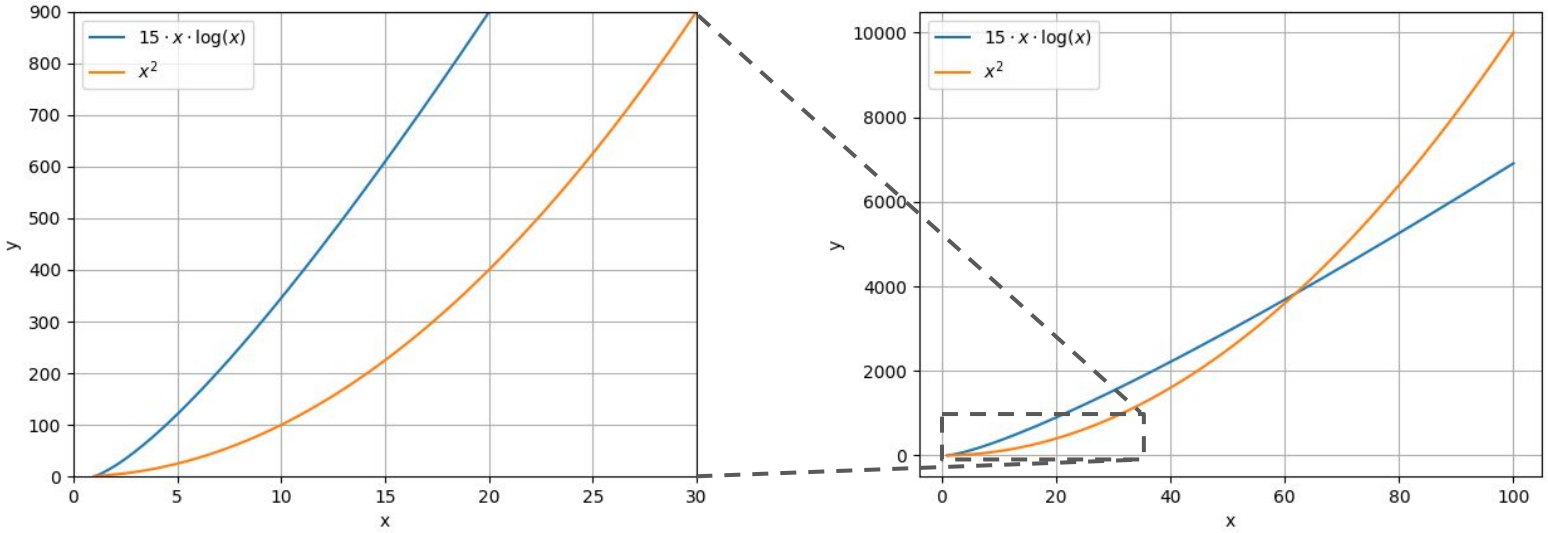
Bubble sort : $O(n^2)$

Tri sélection : $O(n^2)$



Complexité du tri fusion

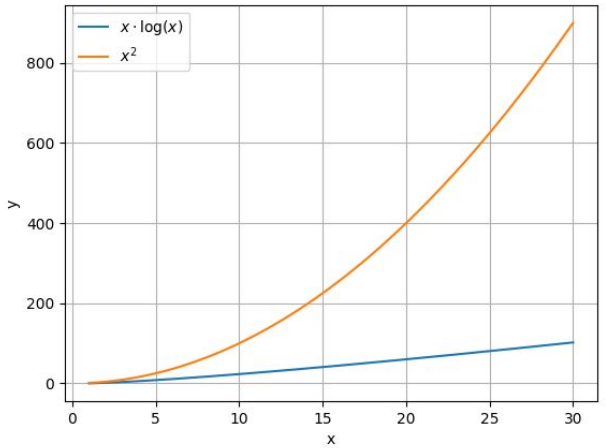
Avantage asymptotique de $O(n \log(n))$ sur $O(n^2)$:
Il existera toujours une taille de séquence telle que l'approche en $O(n \log(n))$ sera plus efficace.



Note sur la complexité $O(n \log(n))$

Pourquoi être satisfait d'une complexité $O(n \log(n))$?

- À l'évidence, un algorithme de tri ne peut en tout cas pas être mieux que $O(n)$.
- $n \log(n)$ "s'éloigne" d'un comportement linéaire bien plus lentement que n'importe quel polynôme ne s'en éloignerait.



Comparaison avec le tri à Bulles

Commençons par examiner le processus d'implémentation d'un algorithme.

Nous proposons la méthode suivante :

1. Écrire son intention en commentaire ;
2. Invoquer des noms de variables et de fonctions là où a besoin d'elles - peu importe qu'elles soient définies ;
3. Définir ce qui manque.

Exemple pas à pas d'implémentation du tri à bulles

```python
def tri_bulle(a):
    n = len(a)
    # Parcours des éléments et swap si besoin
```

Étape 1 : décrire structure générale telle qu'on l'imagine.

```c
// compilation:
// gcc -o tribulle tribulle.c

#include <stdio.h>

void triBulle(int* a, int n){
    //Parcours des éléments et swap si besoin
}
```

Exemple pas à pas d'implémentation du tri à bulles

```python
def tri_bulle(a):
    n = len(a)
    # Pour i de 0 à n-2 compris ...
        # ... si a[i] > a[i+1], swap
```

Étape 2 : détailler et préparer la structure.

```c
#include <stdio.h>

void triBulle(int* a, int n){
    //Pour i de 0 à n-2 …
            // si a[i] > a[i+1], swap
}
```

Exemple pas à pas d'implémentation du tri à bulles

```python
def tri_bulle(a):
    n = len(a)
    for i in range(n-1):
        if a[i] > a[i+1] :
            swap(a, i, i+1)
```

Étape 3 : écrire le code, même si on sait qu'il n'est pas définitif !
Ici, il manque une boucle, et la fonction swap n'est pas définie.

```c
#include <stdio.h>

void triBulle(int* a, int n){
    for(int i=0; i<n-1; i++){
        if(a[i] > a[i+1]){
            int tmp = a[i];
            a[i] = a[i+1];
            a[i+1] = tmp;
        }
    }
}
```

Exemple pas à pas d'implémentation du tri à bulles

```python
def tri_bulle(a):
    n = len(a)
    for i in range(n-1):
        if a[i] > a[i+1] :
            swap(a, i, i+1)
```

```c
#include <stdio.h>

void triBulle(int* a, int N){
    for(int i=0; i<n-1; i++){
        if(a[i] > a[i+1]){
            int tmp = a[i];
            a[i] = a[i+1];
            a[i+1] = tmp;
        }
    }
}
```

On veut répéter tout ça, mais pas à chaque fois jusqu'à **n-1**.

D'abord n-1, ensuite n-2, etc.

Exemple pas à pas d'implémentation du tri à bulles

```python
def tri_bulle(a):
    n = len(a)
    # Pour i_fin de n-1 à 0 ...
    for i in range(i_fin):
        if a[i] > a[i+1] :
            swap(a, i, i+1)
```

```c
#include <stdio.h>

void triBulle(int* a, int n){
    // Pour iFin de n-1 à 0 ...
    for(int i=0; i<iFin; i++){
        if(a[i] > a[i+1]){
            int tmp = a[i];
            a[i] = a[i+1];
            a[i+1] = tmp;
        }
    }
}
```

On veut répéter tout ça, mais pas à chaque fois jusqu'à **n-1**.

D'abord n-1, ensuite n-2, etc.

Exemple pas à pas d'implémentation du tri à bulles

```python
def tri_bulle(a):
    n = len(a)
    for i_fin in range(n-1,-1,-1):
        for i in range(i_fin):
            if a[i] > a[i+1] :
                swap(a, i, i+1)
```

```c
#include <stdio.h>

void triBulle(int* a, int n){
    // Pour iFin de n-1 à 0 ...
    for(int i=0; i<iFin; i++){
        if(a[i] > a[i+1]){
            int tmp = a[i];
            a[i] = a[i+1];
            a[i+1] = tmp;
        }
    }
}
```

On veut répéter tout ça, mais pas à chaque fois jusqu'à **n-1**.

D'abord n-1, ensuite n-2, etc.

Exemple pas à pas d'implémentation du tri à bulles

```python
def tri_bulle(a):
    n = len(a)
    for i_fin in range(n-1,-1,-1):
        for i in range(i_fin):
            if a[i] > a[i+1] :
                swap(a, i, i+1)
```

```c
#include <stdio.h>

void triBulle(int* a, int n){
    for(int iFin=n-1; iFin>=0; iFin--){
        for(int i=0; i<iFin; i++){
            if(a[i] > a[i+1]){
                int tmp = a[i];
                a[i] = a[i+1];
                a[i+1] = tmp;
            }
        }
    }
}
```

Exemple pas à pas d'implémentation du tri à bulles

```python
def tri_bulle(a):
    n = len(a)
    for i_fin in range(n-1,-1,-1):
        for i in range(i_fin):
            if a[i] > a[i+1] :
                a[i], a[i+1] = a[i+1], a[i]
```

```c
#include <stdio.h>

void triBulle(int* a, int n){
    for(int iFin=n-1; iFin>=0; iFin--){
        for(int i=0; i<iFin; i++){
            if(a[i] > a[i+1]){
                int tmp = a[i];
                a[i] = a[i+1];
                a[i+1] = tmp;
            }
        }
    }
}
```

Exemple pas à pas d'implémentation du tri à bulles

```python
def tri_bulle(a):
    n = len(a)
    for i_fin in range(n-1,-1,-1):
        for i in range(i_fin):
            if a[i] > a[i+1] :
                a[i], a[i+1] = a[i+1], a[i]
```

```c
#include <stdio.h>

void triBulle(int* a, int n){
    for(int iFin=n-1; iFin>=0; iFin--){
        for(int i=0; i<iFin; i++){
            if(a[i] > a[i+1]){
                int tmp = a[i];
                a[i] = a[i+1];
                a[i+1] = tmp;
            }
        }
    }
}
```

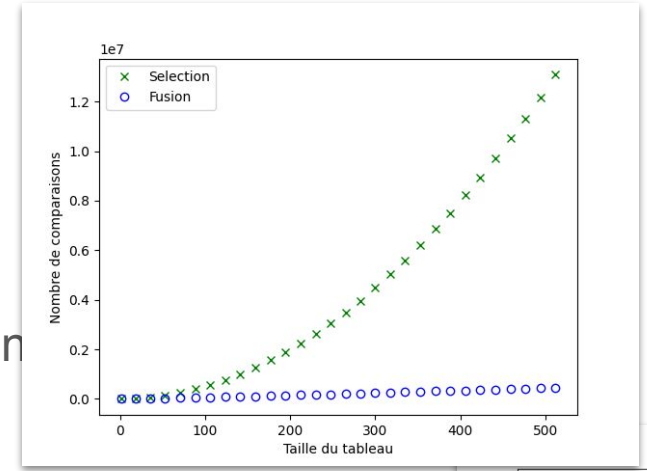
Amélioration possible : s'arrêter lorsque la boucle n'a donné lieu à aucun swap.

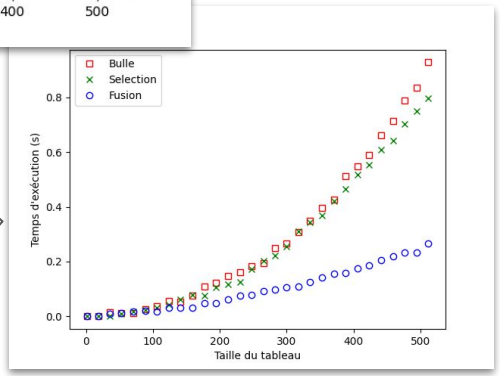
Implémentation du tri fusion

TP à venir

Comparaison entre le tri à bulles et le tri fusion

- On peut compter le nombre de comparaisons effectuées par chaque algorithme.
- Ici, les résultats sont des moyen sur de nombreux tableaux aléatoires.
- Cf. discussion sur le temps d'exécution.





Retour sur le problème du calcul de la médiane

- Approche 1: pour chaque élément de la séquence, **compter** le nombre *k* d'éléments plus petits que lui. Retourner l'élément tel que *k* = (*n-1*)/2.

VS

- Approche 2 : **trier** la séquence, puis retourner l'élément d'indice int(*n*/2).
- Finalement, il semble que l'approche 2 puisse être largement préférable à l'approche 1, du moins pour de grandes valeurs de *n*.
- Pseudocode pour l'approche 2:

```
1. Trier S avec l'algorithme de tri fusion
2. Retourner S[int(n/2)]
```

Retour sur les formulations récursives

- Permettent souvent de simplifier la formulation d'un problème (cf. tri fusion).
- En revanche, peuvent poser des problèmes de mémoire.
- Exemple avec `fact(4)` en version récursive : doit stocker sur une pile d'appels les valeurs en attente d'être multipliées.

Retour sur les formulations récursives

Exemple avec fact(4) en version récursive : doit stocker sur une **pile d'appels** les valeurs en attente d'être multipliées.

```
def fact(x):
    if x == 0:
        return 1
    else:
        return x * fact(x-1)
```

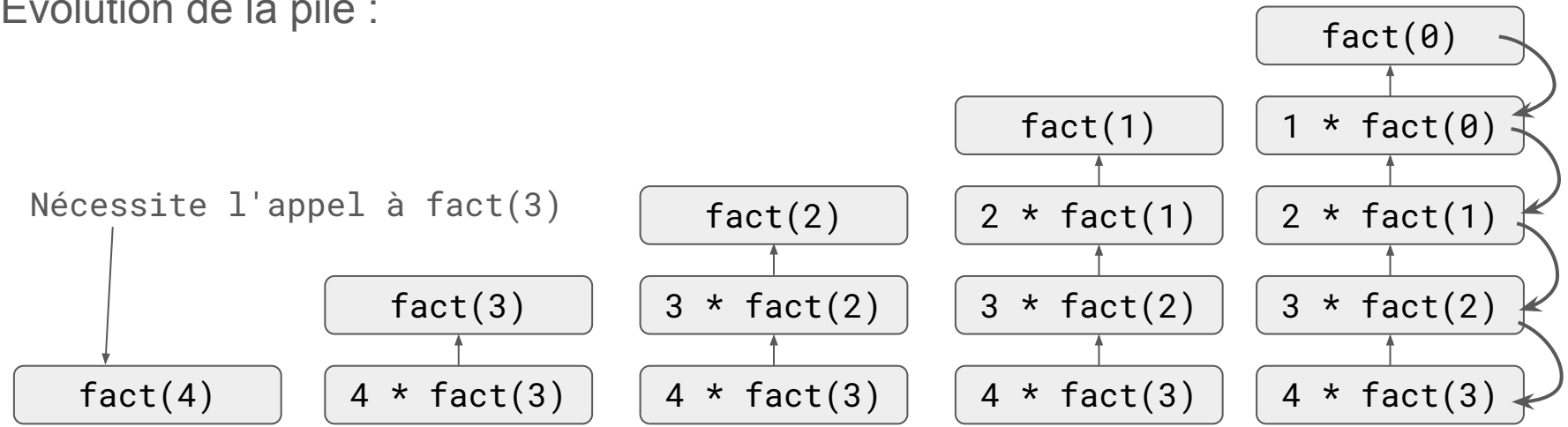
Nécessite 5 valeurs en mémoire

```
r = fact(4)
  = 4 * fact(3)
  = 4 * 3 * fact(2)
  = 4 * 3 * 2 * fact(1)
  = 4 * 3 * 2 * 1 * fact(0)
  = 4 * 3 * 2 * 1 * 1
  = 24
```

Retour sur les formulations récursives

Exemple avec fact(4) en version récursive : doit stocker sur une **pile d'appels** les valeurs en attente d'être multipliées.

Évolution de la pile :



Retour sur les formulations récursives

À l'inverse, une formulation itérative a besoin de moins de mémoire :

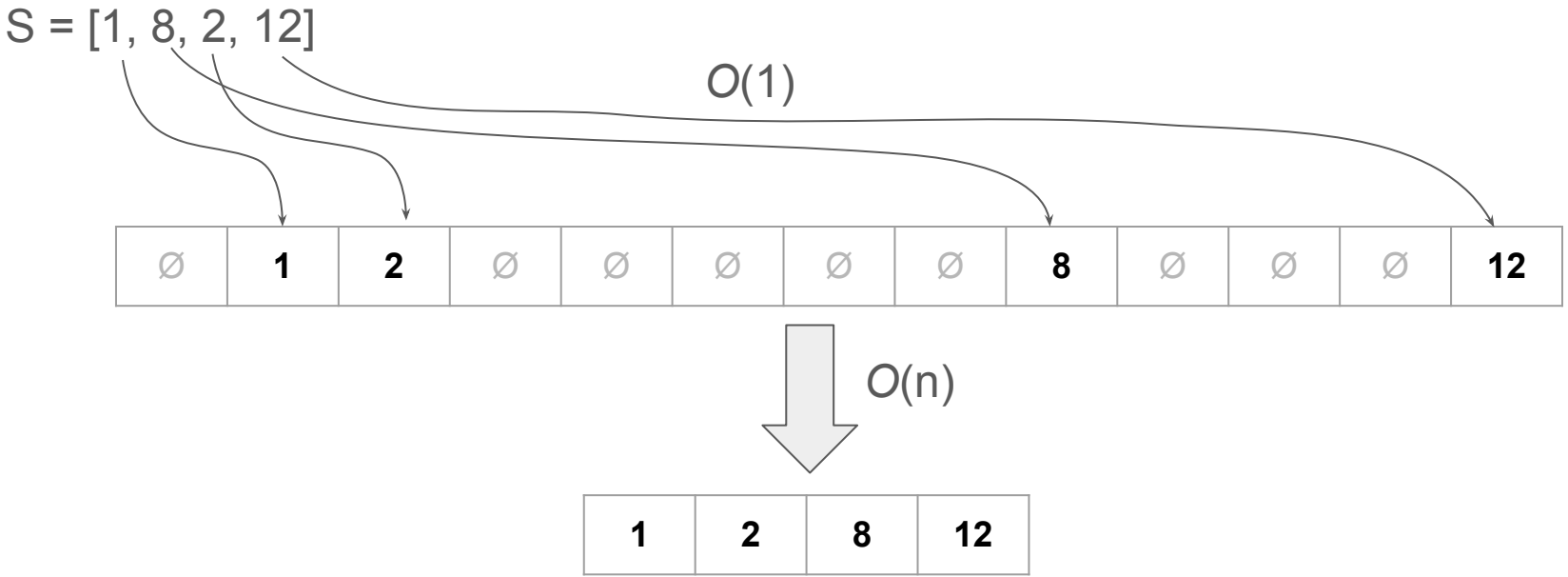
```
def fact(n):
    n = 1
    for i in range(1, n):
        n = n * i
    return n
```

En pratique, si la mémoire/performance est importante, on préférera les versions itératives, peut-être moins "élégantes" mais plus pragmatiques.

Pigeonhole sort

- Idée de base pour trier une séquence d'entiers S : on crée une séquence S'. Chaque élément de S peut être rangé en $O(1)$ à un l'indice de S' qui correspond à sa valeur ⇒ Tri sans comparaisons (!)
- Au cours d'un second passage, on crée une séquence S'' faite de tous les éléments non-vides de S'.
- S'il n'y pas de doublons, le tri est $O(n)$ dans le pire des cas.
- Où est le piège ?
- NB : Pigeonhole fonctionne vraiment, et il inspire une famille d'algorithmes de tris (bucket sort et radix sort) très efficaces dans certaines situations.
- Visualisation :

Pigeonhole sort



Analyse de complexité | **Commentaire final**

Voici un algorithme de complexité en temps $O(1)$ pour trier une liste :

```
def tri_escroquerie(a): # Connu sous le nom de sleep sort
    liste_triee = []
    for element in a:
        create_parallel_thread_and_sleep(element)
        liste_triee.append(element)
    return liste_triee
```

Analyse de complexité | **Commentaire final**

- On a vu qu'on pouvait s'intéresser à la complexité en mémoire (non traitée dans ce cours) ou en temps ("nombre d'étapes", "nombre d'opérations élémentaires").
- Ce que nous enseigne le sleep sort : il faut être vigilant quant à la nature de la variable considérée.
- En théorie, le sleep sort est $O(1)$ en temps si l'on considère n comme la variable d'intérêt, mais on peut aussi voir que, par ailleurs, il est $O(\max(S))$.
- Pigeonhole sort est $O(n)$ en temps, mais $O(\max(S)-\min(S))$ en mémoire.