Introduction à l'informatique

pour les mathématiques, la physique et les sciences computationnelles

Yann Thorimbert

Partie 2 | Chapitre 2
Algorithmes de tri et complexité

Yann Thorimbert

Chapitres du cours (seconde partie du cours)

0. Introduction

1. Structures de données : tableaux, listes et dictionnaires

2. Algorithmes de tri ←

3. Algorithmes de recherche au sein d'une séquence

4. Algorithmes sur graphes

Rappel

- Nous avons vu plusieurs structures de données permettant d'ordonner ou d'associer des nombres (ou d'autres quantités).
  - Tableaux
  - Listes chaînées
  - Tables de hachage

- Le but de ce chapitre est de découvrir différentes stratégies pour obtenir, à partir d'une séquence de nombres quelconque, une version triée de cette même séquence.

- Dans ce qui suit, nous utiliserons le terme "séquence" de façon générique pour désigner n'importe quel **type abstrait** permettant de lier un indice à une valeur. Nous ne désignons pas une structure de donnée spécifique.

Avant d'entrer dans le vif du sujet | **Retour sur la complexité**

- Nous allons souvent comparer différents algorithmes entre eux en nous référant à leur complexité en temps (cf. chapitre précédent).

- Afin de discuter plus en profondeur de la notion de complexité, abordons ici un algorithme qui nous permet de trouver la valeur minimale au sein d'une séquence de nombres.

Complexité | **Algo de recherche du minimum**

- On supposera toujours que les éléments de la séquence S sont numérotés à partir de zéro. Le minimum de la séquence `S = [2, -4, 5, 7, 1]` vaut `-4`.

- On notera toujours n le nombre d'éléments de la séquence.

- L'algorithme est le suivant (on abstrait ici le concept de boucle) :

```
1. Noter m = S[0]
2. Pour chaque valeur i de 0 à n - 1 :
    a. Noter v = S[i]
    b. Si v < m, alors m = v
3. Retourner m
```

Complexité | **Algo de recherche du minimum**

Toutes les instructions ne sont pas exécutées le même nombre de fois :

1. `Noter m = L[0]` ← Effectué une seule fois
2. `Pour chaque valeur i de 0 à n - 1 :`
    a. `Noter v = L[i]`
    b. `Si v < m, alors m = v`
3. `Retourner m`

Complexité | **Algo de recherche du minimum**

Toutes les instructions ne sont pas exécutées le même nombre de fois :

1. `Noter m = L[0]` ← Effectué une seule fois
2. `Pour chaque valeur i de 0 à n - 1 :`
    a. `Noter v = L[i]`
    b. `Si v < m, alors m = v`
3. `Retourner m`

Effectué n fois

Complexité | **Algo de recherche du minimum**

Toutes les instructions ne sont pas exécutées le même nombre de fois :

1. `Noter m = L[0]` ← Effectué une seule fois
2. `Pour chaque valeur i de 0 à n - 1 :`
    a. `Noter v = L[i]`
    b. `Si v < m, alors m = v`
3. `Retourner m`

Effectué n fois

Globalement, on arrive au bout après n "étapes".

Complexité | **Algo de recherche du minimum**

- N'y a-t-il pas plus de n étapes ?
  - Certes, il faut effectuer n comparaisons (étape 2.b) …
  - … mais pour pouvoir effectuer chacune de ces comparaisons, on doit payer le prix de l'accès à l'élément `L[i]` (étape 2.a)
  - Et dans le pire des cas, l'assignation m = v doit aussi être effectuée n fois !
  - Et puis, la valeur de i est mise à jour n fois également.

Complexité | **Algo de recherche du minimum**

1. Noter `m = L[0]` ← Effectué une seule fois
2. Pour chaque valeur `i` de `0` à `n - 1` : ← Mis à jour n fois
   a. Noter `v = L[i]`
   b. Si `v < m`, alors `m = v` ← Effectué au pire n fois
3. Retourner `m`

Effectué n fois

Complexité | **Algo de recherche du minimum**

- N'y a-t-il pas plus de n étapes ?
  - Certes, il faut effectuer n comparaisons (étape 2.b) …
  - … mais pour pouvoir effectuer chacune de ces comparaisons, on doit payer le prix de l'accès à l'élément `L[i]` (étape 2.a)
  - Et dans le pire des cas, l'assignation m = v doit aussi être effectuée n fois !
  - Et puis, la valeur de i est mise à jour n fois également.

- Le nombre d'**instructions** est bien de la forme $k_1 \cdot n + k_2 \cdot n + ... + b = a \cdot n + b$.

- Qu'entend-on donc par "n étapes" ?

Complexité | **Algo de recherche du minimum**

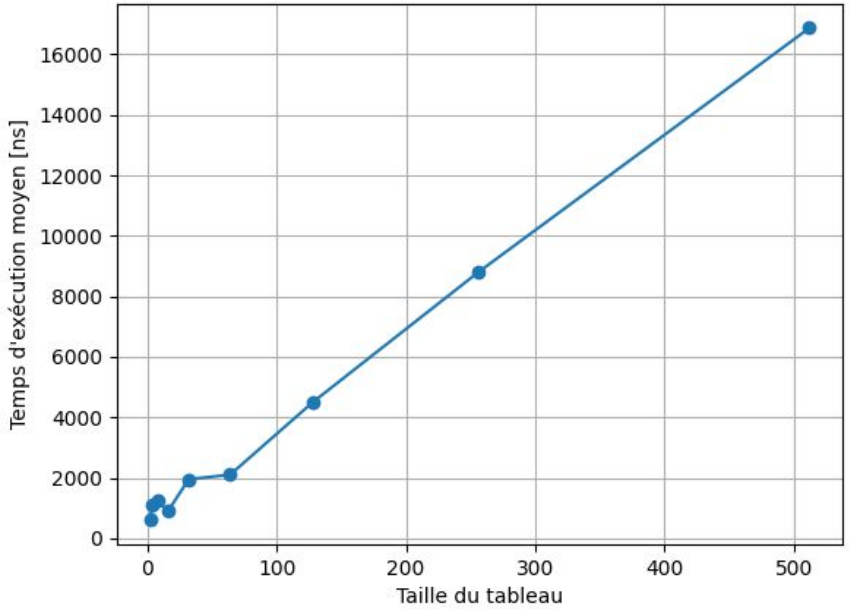
- Nous ne sommes **pas intéressés par l'expression exacte** du nombre d'étapes. Ce qui nous intéresse est la **forme** de cette expression, que nous nommerons complexité en temps.

- L'expression exacte f(n) du nombre d'étapes, pour une **implémentation** donnée de l'algorithme, ressemble certainement à une fonction du type $f(n) = a \cdot n + b$, mais la valeur des constantes dépend de détails d'implémentation.

- À un niveau algorithmique, il est pour nous suffisant d'observer que **l'algorithme est *O*(n)**, car nous sommes intéressés par le **terme dominant**.

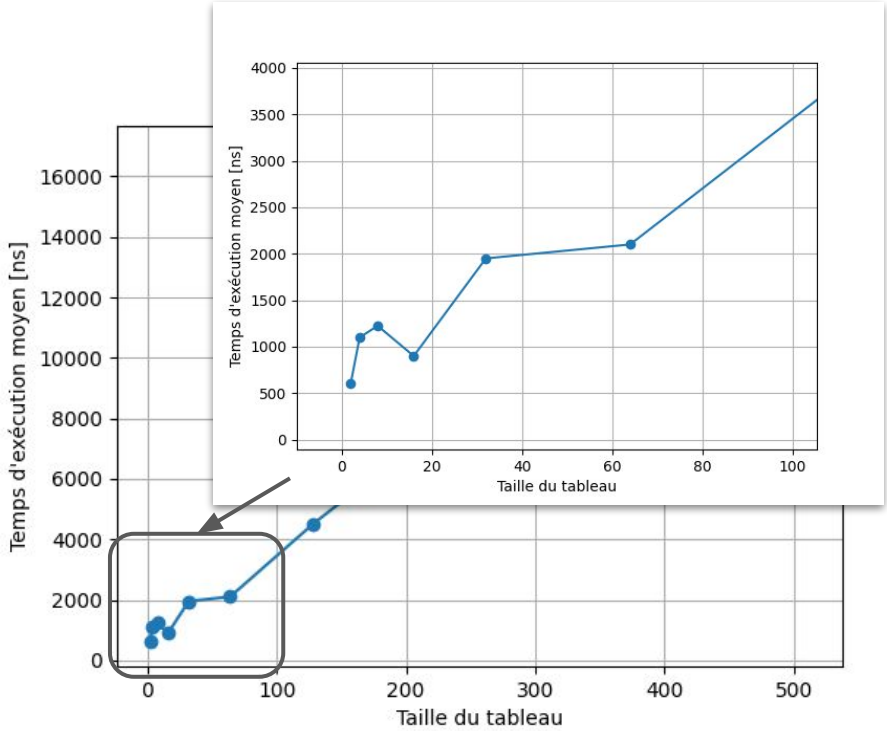
Complexité | **Algo de recherche du minimum**

- Exemple : chronométrage de la fonction min( ) de Python.
- À "l'infini", la fonction est linéaire.
- Pour de petites séquences, le temps d'exécution est dominé par d'autres effets non triviaux qui ne nous intéressent pas ici.



Complexité | **Algo de recherche du minimum**

- Exemple : chronométrage de la fonction min( ) de Python.
- À "l'infini", la fonction est linéaire.
- Pour de petites séquences, le temps d'exécution est dominé par d'autres effets non triviaux qui ne nous intéressent pas ici.



Complexité | **Algo de recherche du minimum**

- Il faut noter que l'algorithme de recherche du minimum est $O(n)$ seulement si l'accès aux éléments `L[i]` est $O(1)$.

- Dans le cas contraire, il faut en tenir compte ! Par exemple, si la complexité moyenne d'accès aux éléments `L[i]` est $O(n^3)$, alors l'algorithme dans son ensemble est $O(n^5)$.

- $O(n)$ dénote le **comportement asymptotique** de la fonction.

- On peut s'intéresser à d'autres complexités que la complexité en temps, comme la complexité en mémoire. Comme ce n'est pas le cas dans ce cours, nous omettons souvent de préciser qu'il s'agit de la complexité en temps.

Complexité | **Algo de recherche du minimum**

L'analyse de complexité peut être plus précise et poussée, mais ce n'est pas le sujet du cours. Voici néanmoins des définitions que l'on trouve fréquemment dans la littérature :

$T(n) = O(g(n))$ dénote en général le pire des cas : $\exists\ k, n_0 \in \mathbb{R}_+ \mid T(n) \leq k \cdot g(n),\ \forall\ n > n_0$
(autrement dit, "T est **tout au plus** d'une complexité g(n)")

$T(n) = \Theta(g(n))$ signifie que $\exists\ a, b, n_0 \in \mathbb{R}_+ \mid a \cdot g(n) \leq T(n) \leq b \cdot g(n),\ \forall\ n > n_0$
(autrement dit, "T est d'une complexité g(n)")

(Non traité en cours)

Complexité

Ce qu'il faut retenir de l'analyse de complexité :

- Utile pour comprendre comment un algorithme se comporte **asymptotiquement** en **fonction de la taille du problème**.

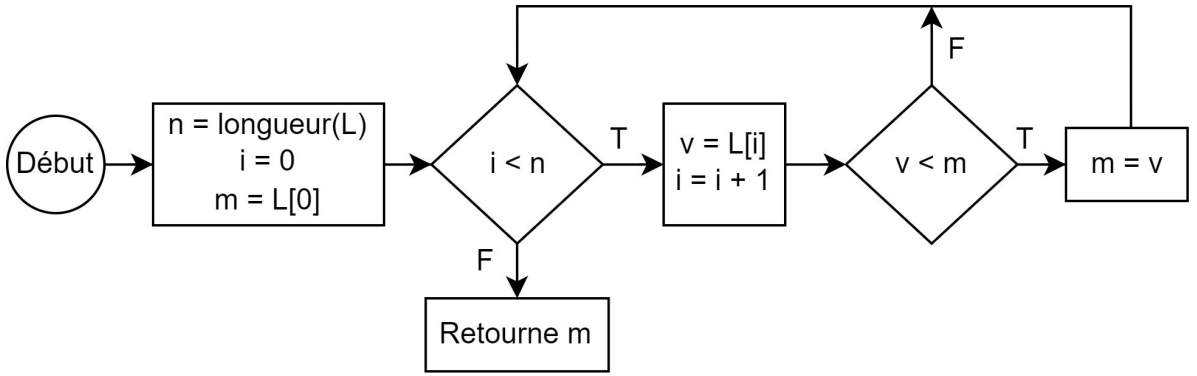
- Utile pour comparer entre eux des algorithmes pour des problèmes de tailles arbitrairement grandes.

- **Inutile** pour connaître le **temps d'exécution** d'une implémentation, car suivant la taille du problème en pratique ainsi que d'autres détails informatiques, de nombreux effets peuvent se produire.

Algorigrammes | **Algo de recherche du minimum**

Les algorigrammes sont une manière classique de **représenter graphiquement** un algorithme en utilisant peu de pseudocode.

Algorigramme (*flow-chart*) de notre algorithme, où la boucle n'est plus abstraite :

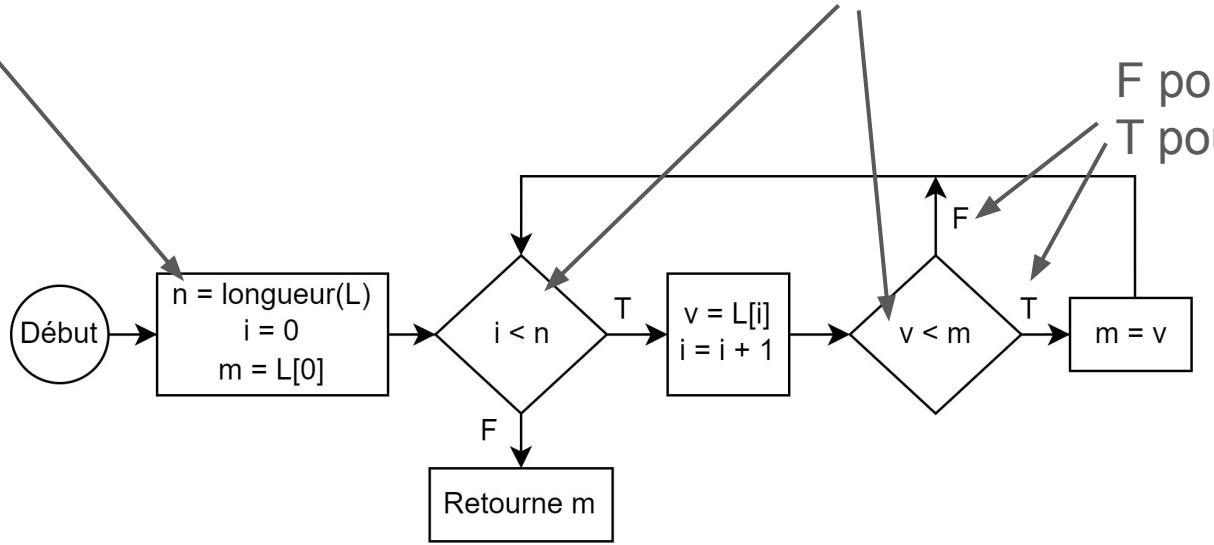


Algorigrammes | **Algo de recherche du minimum**

Les **instructions** sont encadrées

Les **conditions** sont dans un losange

F pour **False**,
T pour **True**



Les algorithmes de tri

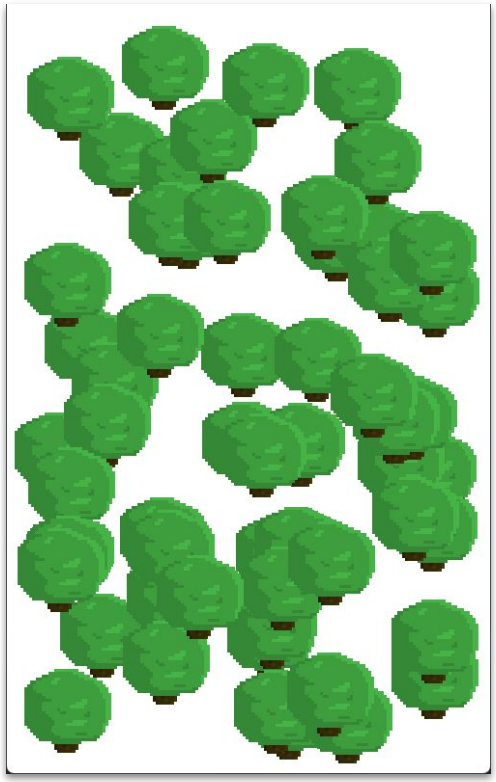
Pourquoi s'intéresser au tri ?

- Comme on l'a vu, les séquences de valeurs sont omniprésentes au sein des programmes informatiques.

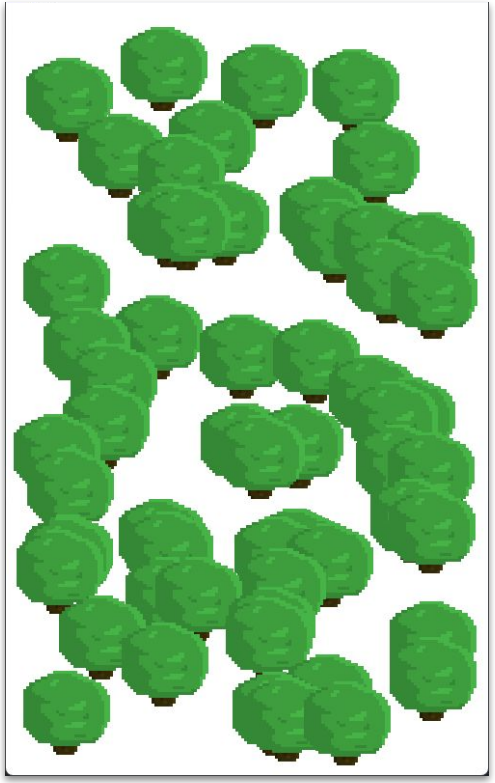
- Dans beaucoup de cas, on désirera chercher des valeurs spécifiques au sein d'une séquence. L'efficacité de telles recherches est impactée par l'ordre des valeurs.

- Dans beaucoup de cas, posséder des données triées offre des garanties désirables sur la structure du problème.

Exemples d'utilisation du tri | **Affichage graphique**



Affichage d'éléments triés en fonction de leur coordonnée verticale.



Exemples d'utilisation du tri | **Calcul de médiane**

- Définition informelle de la médiane d'une séquence de nombres : valeur telle que la moitié des éléments de la séquence sont plus petits, et l'autre moitié plus grands.

- Question : quel est le salaire médian parmi la population suisse ?
  `salaires = [4500, 3850, 7600, 2500, ..., 5650]` # n ≈ 5 millions.

- Pour simplifier l'exemple, supposons un nombre impair de salaires, noté *n*. On suppose également que tous les salaires sont uniques.

Exemples d'utilisation du tri | **Calcul de médiane**

- Quel est le salaire médian parmi la population suisse ?

```
salaires = [4500, 3850, 7600, 2500, ..., 5650] # n ≈ 5 millions.
```

- Une approche : 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.

- Une autre approche : ?

Exemples d'utilisation du tri | **Calcul de médiane**

- Quel est le salaire médian parmi la population suisse ?

```
salaires = [4500, 3850, 7600, 2500, ..., 5650] # n ≈ 5 millions.
```

- Une approche : 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.

- Une autre approche : **trier la séquence**, puis retourner l'élément d'indice *n*/2.

Exemples d'utilisation du tri | **Calcul de médiane**

- Quel est le salaire médian parmi la population suisse ?

```
salaires = [4500, 3850, 7600, 2500, ..., 5650] # n ≈ 5 millions.
```

- Une approche : 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.

- Une autre approche : **trier la séquence**, puis retourner l'élément d'indice *n*/2.

Très souvent, un algorithme **commence par effectuer un tri** sur les données en vue d'un traitement subséquent !

Exemples d'utilisation du tri | **Calcul de 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 *n*/2.

Exemples d'utilisation du tri | **Calcul de 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.

Pour comparer les deux approches, nous devrons étudier les algorithmes de tri. Mais commençons par l'approche 1.

Approche 2 : **trier la séquence**, puis retourner l'élément d'indice *n*/2.

Exemples d'utilisation du tri | **Calcul de médiane**

1ère approche : 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.
Quelle est la complexité de cette approche ?

```
salaires = [4500, 3850, 7600, 2500, … , 5650]
```

```
k = 0
4500 > 3850 ? Oui, k = k + 1.
4500 > 7600 ? Non.
4500 > 3850 ? Oui, k = k + 1.
...
4500 > 5650 ? Non.
```

Exemples d'utilisation du tri | **Calcul de médiane**

1ère approche : 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.
Quelle est la complexité de cette approche ?

```
salaires = [4500, 3850, 7600, 2500, … , 5650]
```

```
k = 0
5650 > 3850 ? Oui, k = k + 1.
5650 > 3850 ? Oui, k = k + 1.
5650 > 7600 ? Non.
5650 > 3850 ? Oui, k = k + 1.
...
```

Exemples d'utilisation du tri | **Calcul de médiane**

1ère approche : 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.
Quelle est la complexité de cette approche ?

```
def mediane_approche1(salaires):
    n = len(salaires)
    k = 0
    for s1 in salaires:
        for s2 in salaires:
            if s2 < s1:
                k = k + 1
        if k == (n-1)/2:
            return s1
```

Exemples d'utilisation du tri | **Calcul de médiane**

1ère approche : 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.
Quelle est la complexité de cette approche ?

```
def mediane_approche1(salaires):
    n = len(salaires)
    k = 0
    for s1 in salaires:
        for s2 in salaires:
            if s2 < s1:
                k = k + 1
        if k == (n-1)/2:
            return s1
```

Combien de fois cette comparaison aura-t-elle lieu ?

Exemples d'utilisation du tri | **Calcul de médiane**

1ère approche : 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.
Quelle est la complexité de cette approche ?

Nombre de salaires à examiner ~ n.
Pour chacun, nombre de salaires avec lesquels le comparer ~ n.
⇒ Cet algorithme nécessite un nombre de comparaisons proportionnel à n^2.

Cet algorithme possède une complexité $O(n^2)$.

Exemples d'utilisation du tri | **Calcul de médiane**

- Cet algorithme possède une complexité temporelle de $O(n^2)$.
  Cela signifie que le nombre d'étapes de l'algorithme pour trouver la médiane a un **comportement quadratique** par rapport au nombre de salaires.

- Par rapport à un algorithme en $O(n^3)$ par exemple, il existera toujours une valeur de n pour laquelle cet algorithme est plus performant.

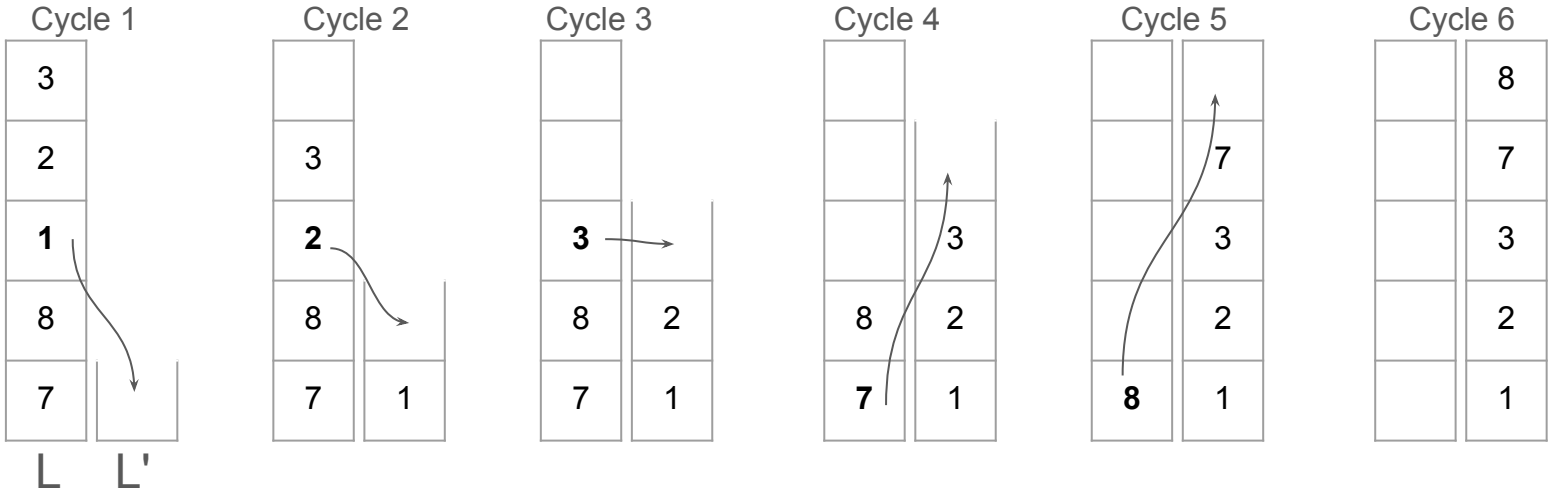
- Quid de l'approche n°2 ?
  ⇒ Cela va dépendre de l'algorithme de tri utilisé. Est-il possible de faire mieux que $O(n^2)$?

Exemple de tri naïf

1. Préparer une séquence vide *S*' = [ ]
2. Tant que length(*S*') ≠ *n* :
   a. *m* = min(*S*)
   b. Enlever m de la séquence *S*
   c. Ajouter m dans la séquence *S*'

Exemple de tri naïf

1. Préparer une séquence vide *S'* = [ ]
2. Tant que length(*S'*) ≠ *n* :
   a. *m* = min(*S*)
   b. Enlever m de la séquence *S*
   c. Ajouter m dans la séquence *S'*



Exemple de tri naïf

1. Préparer une séquence vide *S*' = [ ]
2. Tant que length(*S*') ≠ *n* :
   a. *m* = min(*S*)
   b. Enlever m de la séquence *S*
   c. Ajouter m dans la séquence *S*'

Cet algorithme a l'avantage d'être facile à imaginer, mais il serait intéressant de ne pas construire un tableau L ' au fur et à mesure.

Formulons une version dite "**en place**" (*in place*), où le tableau S est **modifié** de telle sorte qu'il soit trié à la fin de l'algorithme.

Tri sélection

7	8	1	3	2

Reprend l'idée de la recherche des minimums successifs.

Cf. slides dévolus à l'**illustration pas à pas**.

Tri sélection | **Pseudocode**

Reprend l'idée de la recherche des minimums successifs :

1. Pour chaque entier i_debut de 0 jusqu'à n-2 :
    a. i_min = indice du minimum entre L[i_debut+1] et L[n-1]
    b. Intervertir les éléments d'indice i_min et i_debut.

Tri sélection | **Pseudocode**

Reprend l'idée de la recherche des minimums successifs :

```
1. Pour chaque entier i_debut de 0 jusqu'à n-2 :
    a. i_min = indice du minimum entre L[i_debut+1] et L[n-1]
    b. Intervertir les éléments d'indice i_min et i_debut.
```

Notez que cette formulation est d'un haut degré d'abstraction.

Cependant, nous avons décrit précédemment l'algorithme de recherche du min.

Tri sélection | **Pseudocode**

Reprend l'idée de la recherche des minimums successifs :

```
1. Pour chaque entier i_debut de 0 jusqu'à n-2 :
    a. i_min = indice du minimum entre L[i_debut+1] et L[n-1]
    b. Intervertir les éléments d'indice i_min et i_debut.
```

Vocabulaire : **swap**(i_min, i_debut)

Tri sélection | **Swap**

- Le tri sélection est un exemple de tri en place, car aucune copie de la séquence n'est construite : la séquence elle-même est modifiée.

- Lorsqu'on intervertit deux valeurs, on passe néanmoins par une copie temporaire. Exemple de fonction `swap(i, j, a)` en Python :

```
def swap(i, j, a):
    tmp = a[i] # valeur temporaire pour ne pas "perdre" a[i]
    a[i] = a[j]
    a[j] = tmp
```

Tri sélection | **Complexité en temps**

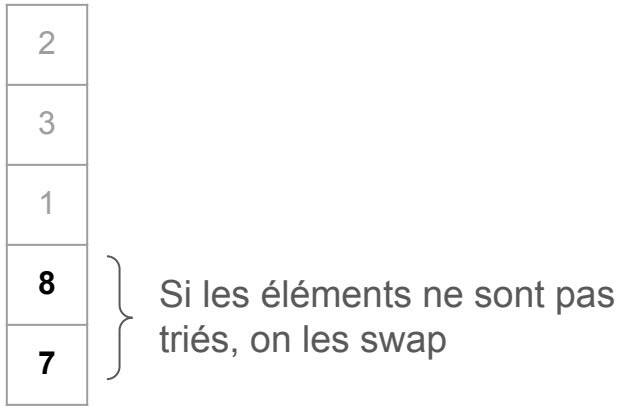
- Le tri sélection est en $O(n^2)$.
- Avec ce tri, nos deux approches pour le calcul de la médiane sont équivalentes du point de vue de la complexité.
- En pratique, le tri sélection est peu utilisé, son intérêt étant principalement pédagogique.

Tri à bulles | **Idée**

- Dans le tri sélection, on fait n - 1 passages sur la séquence (nommés "cycles" dans les exemples détaillés) …
- … et à chacun de ces passages, on effectue un "sous-passage" pour trouver le minimum du reste de la séquence.
- Le tri à bulle exploite le fait que, durant les sous-passages, on peut faire des aménagements pratiques de la séquence.
- À chaque cycle, on compare les éléments voisins tout en "remontant" la séquence, et on les intervertit s'ils sont désordonnés.

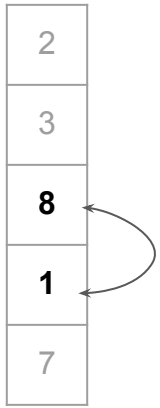
Tri à bulles

Tri à bulles



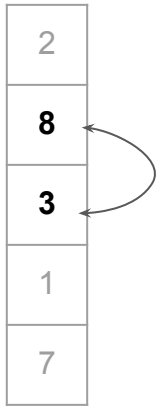
Tri à bulles

Tri à bulles

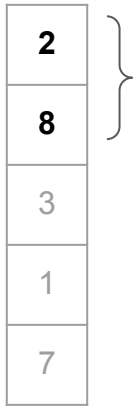


Tri à bulles

Tri à bulles

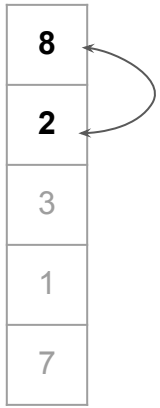


Tri à bulles



Si les éléments ne sont pas triés, on les swap

Tri à bulles



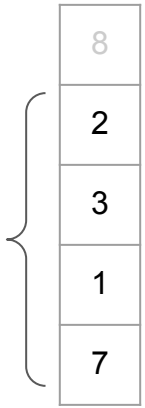
Tri à bulles

Fin du premier cycle.

La plus grande valeur est "remontée à la surface" comme une bulle.

Tri à bulles

Nouvelle sous-séquence à considérer pour les prochains passages



On réitère le processus jusqu'à ce qu'un passage ne produise aucun swap.

Tri à bulles | **Complexité en temps**

- Le tri à bulle est en moyenne $O(n^2)$, car on doit faire de l'ordre de n passages, chacun impliquant n comparaisons/swaps.
- Contrairement au tri sélection, le meilleur des cas est $O(n)$.
- En pratique, le tri à bulles est également peu utilisé.
- Nous donnons un exemple de code du tri à bulles plus tard.
- 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 fusion que nous nous apprêtons à voir se formule bien **récursivement**. Il est basé sur une **division** récursive **des tâches**, comme bien d'autres algorithmes d'une famille de stratégies nommée "Divide & Conquer".

- Algorithme récursif : algorithme qui s'appelle 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 :

```
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.

- Ici, la subdivision est simple : l'algorithme subdivise la séquence reçue en deux parties, à moins que cette séquence ne contienne qu'un seul élément.

- La fusion nécessite davantage de réflexion.

Fusion de deux séquences déjà triées

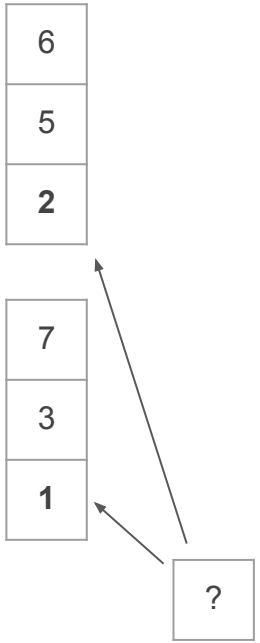
- Fusion(A, B) retourne une séquence avec les éléments de A et B triés.

- 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).

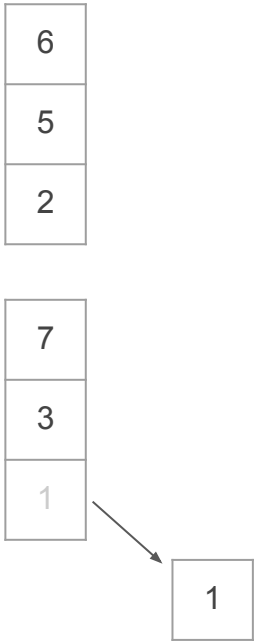
Fusion de deux séquences déjà triées

Fusion de deux séquences déjà triées



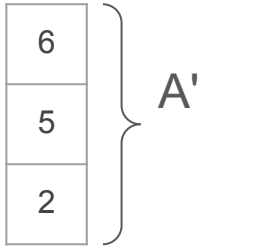
On garde le plus petit des premiers éléments. Ce sera le premier de la séquence fusionnée.

Fusion de deux séquences déjà triées



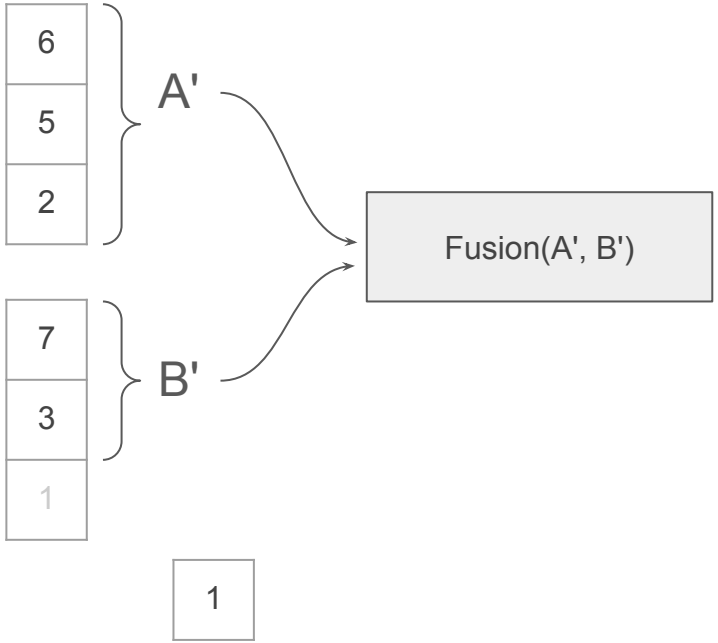
On garde le plus petit des premiers éléments. Ce sera le premier de la séquence fusionnée.

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

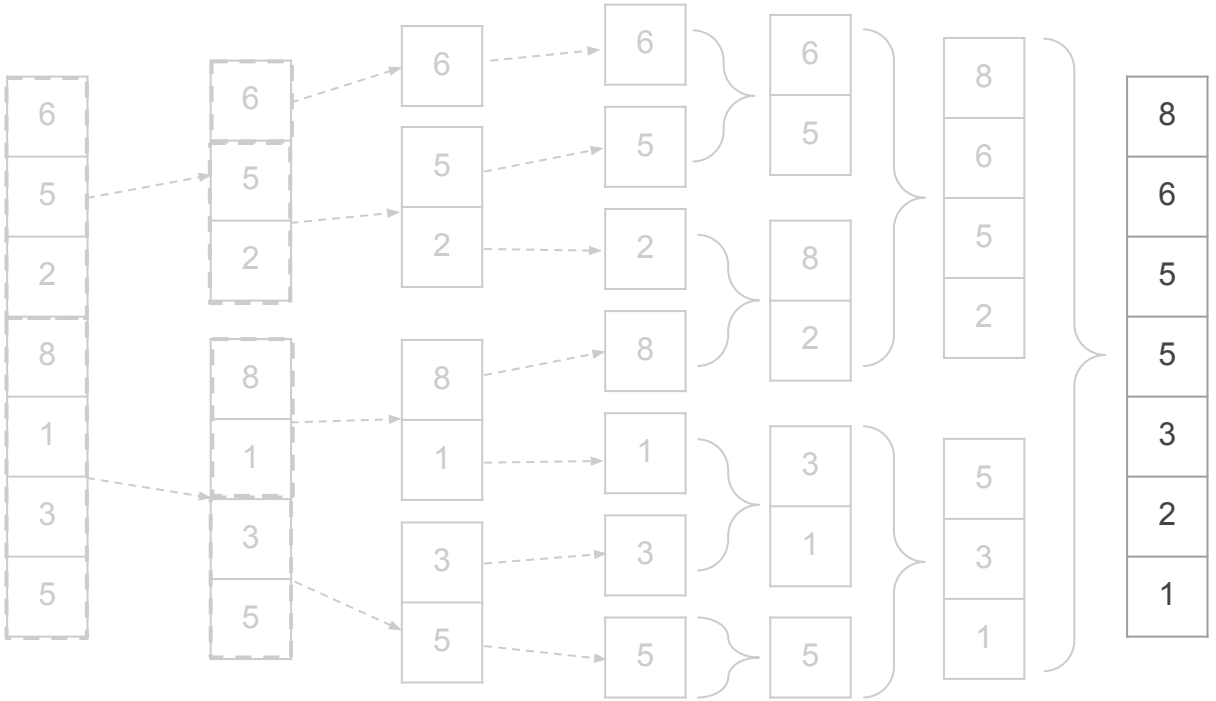
- L'algorithme de fusion nous offre la garantie de pouvoir assembler autant de sous-séquences triées que l'on veut en une seule grande séquence triée.

- Question : comment aboutir à la situation où l'on possède ces sous-séquences triées ?

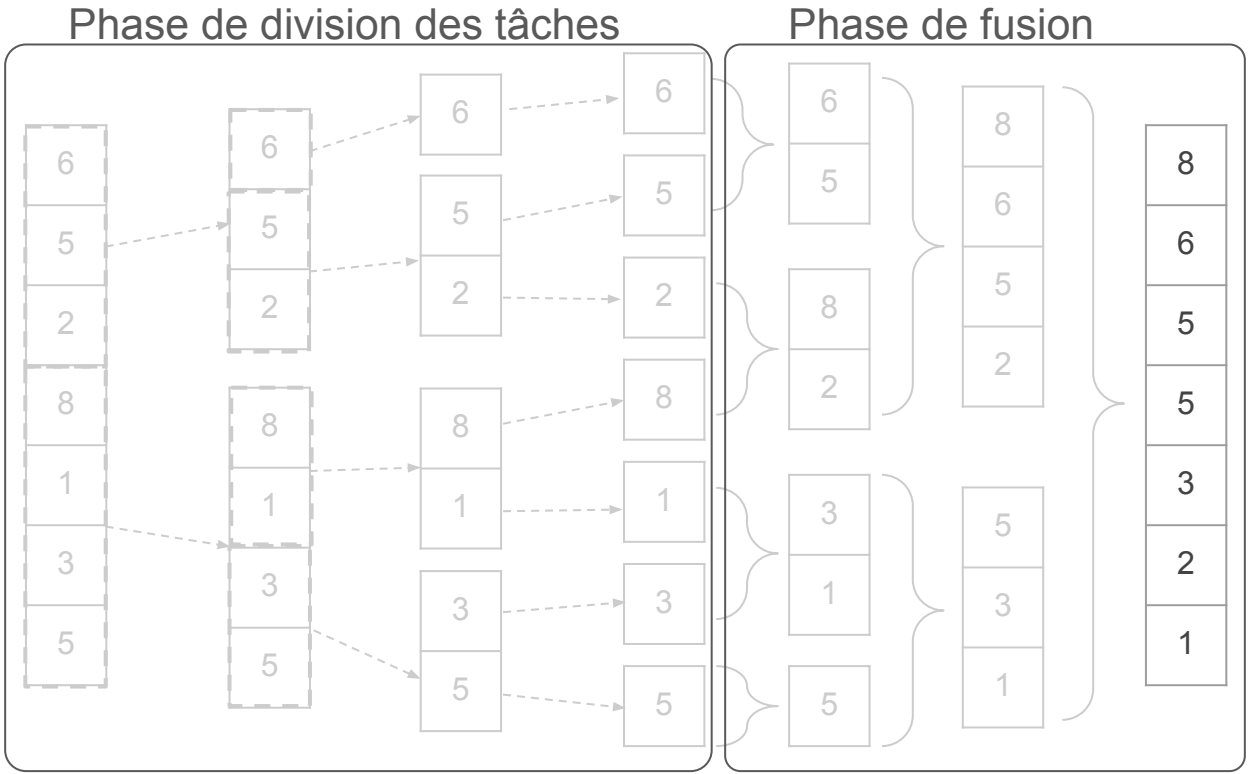
- Réponse : on peut diviser (simplifier) la tâche autant que nécessaire, jusqu'à aboutir uniquement à des cas triviaux, et donc triés!

Tri fusion | **Exemple**

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



Tri fusion | **Exemple**



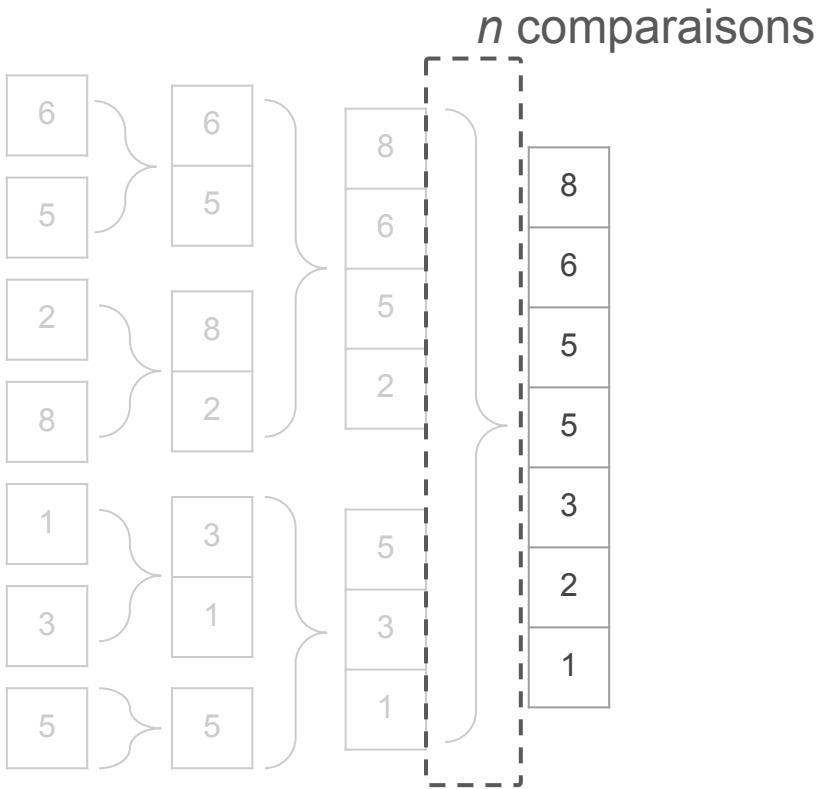
Complexité du tri fusion | **Phase de division**

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 ?**

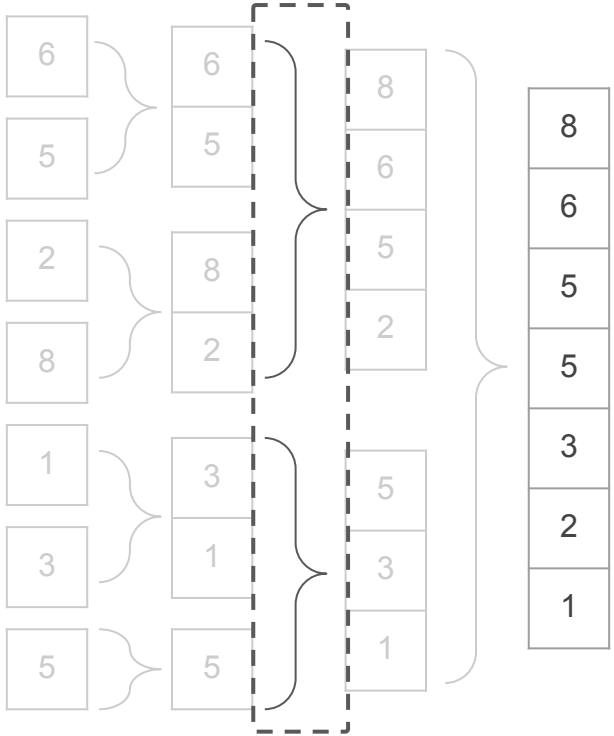
- À chaque niveau de récursion, on divise les tableaux en **deux**. Cela nécessite donc $\log_2(n)$ étapes de division.
- En effet, dans le cas le plus évident, on double le nombre de tableaux à chaque étape. S'il y a k étapes pour diviser n tableaux, la relation $n = 2^k$ est vraie, donc $k = \log_2(n)$.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

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

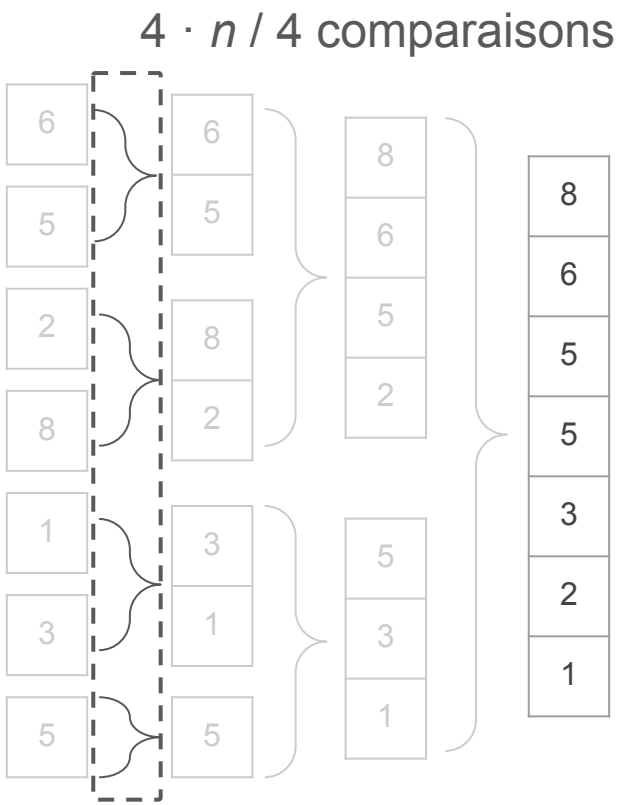


Complexité du tri fusion | **Phase de fusion**

$2 \cdot n / 2$ comparaisons



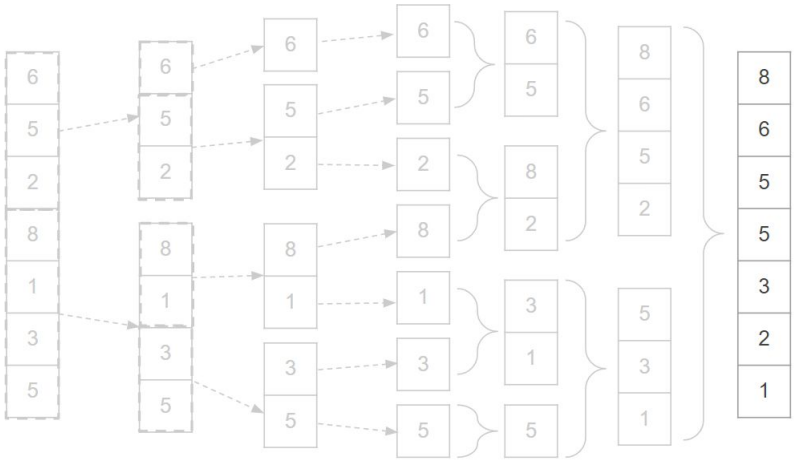
Complexité du tri fusion | **Phase de fusion**



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)**

- L'algorithme est donc $O(n \cdot \log(n))$, ce qui est bien mieux que le tri à bulles ou le tri sélection !

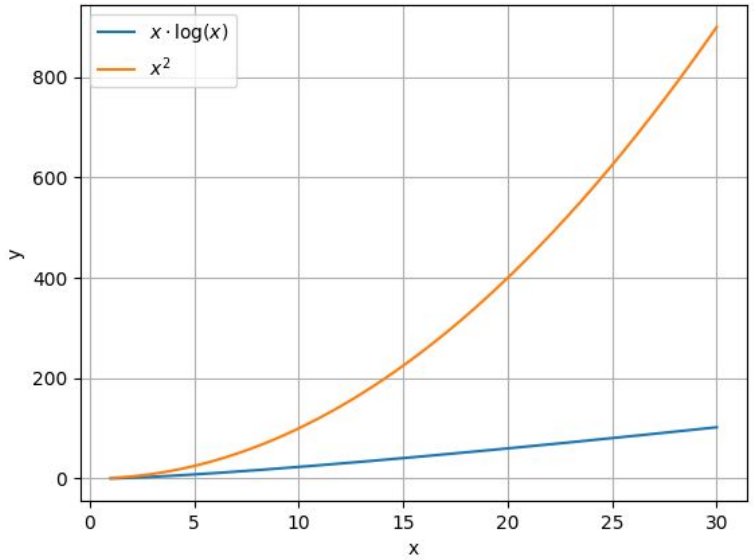
- Notons que $\log_a(x) = \log_b(x) / \log_b(a)$: une quantité logarithmique exprimée dans une base ne diffère que d'un **facteur constant** par rapport à la même quantité dans une autre base. Comme nous l'avons vu, dans l'analyse de complexité asymptotique, ces facteurs constants ne nous importent pas.
  ⇒ On écrit $O(n \log(n))$ **sans spécifier la base** du logarithme.

Complexité en temps du tri fusion

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

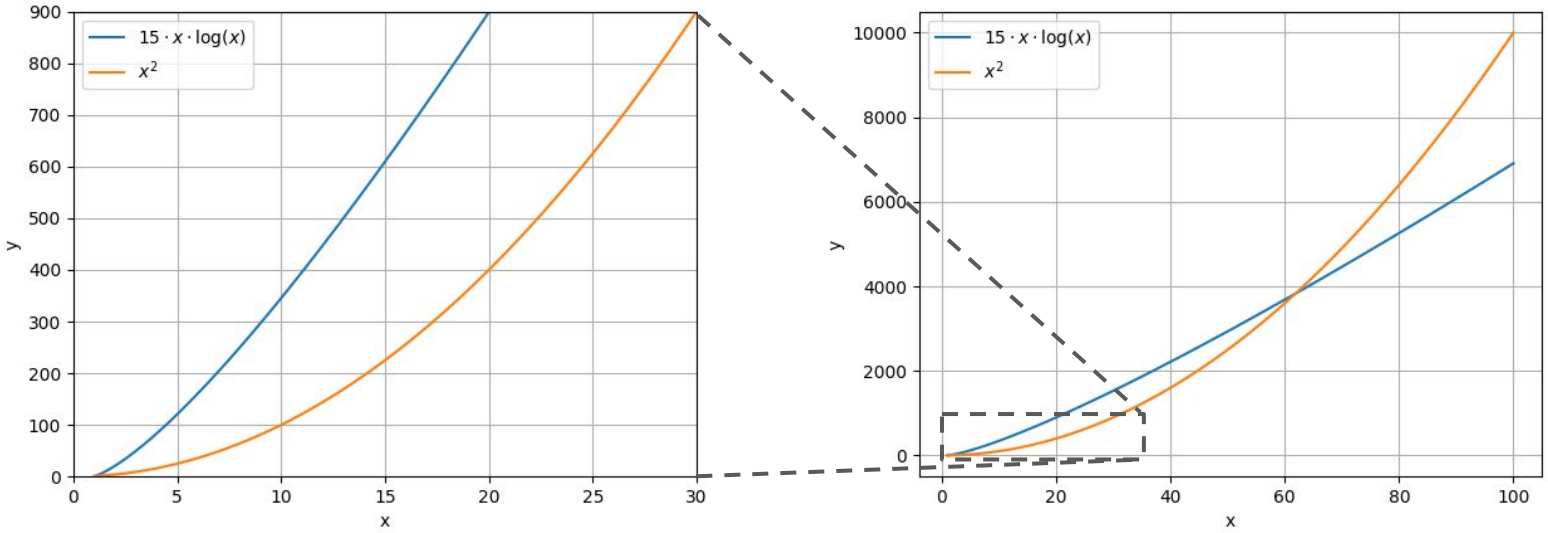
Bubble sort : $O(n^2)$

Tri sélection : $O(n^2)$



Complexité du tri fusion

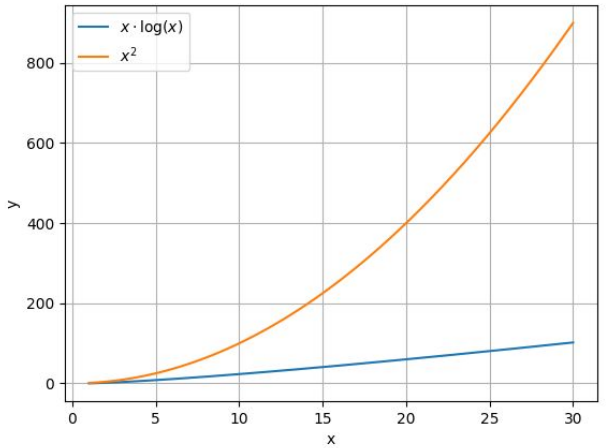
Avantage asymptotique de $O(n \log(n))$ sur $O(n^2)$:
Il existera toujours une taille de séquence telle que l'approche en $O(n \log(n))$ sera plus efficace.



Note sur la complexité $n \cdot \log(n)$

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

- À l'évidence, un algorithme de tri ne peut en tout cas pas être mieux que $O(n)$.
- $n \cdot \log(n)$ "s'éloigne" d'un comportement linéaire bien plus lentement que n^2 ne s'éloigne de $n \cdot \log(n)$.
- Autrement dit, $n \cdot \log(n)$ est en quelque sorte "aussi linéaire que possible", vu sous cet angle.



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 *n*/2.

Retour au problème de la médiane

- Finalement, il semble que l'approche 2 puisse être largement préférable à l'approche 1, du moins pour de grandes valeurs de n.

- En effet, pour l'approche 1 le temps de calcul sera $T(n) = O(n^2)$, tandis que pour l'approche 2 on a $T(n) = O(n \log(n))$.

- Pseudocode pour l'approche 2:

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

Commentaires finaux sur la complexité

- Nous avons différents algorithmes possédant des complexités en temps différentes. Par exemple : `min` et `factorielle` sont $O(n)$, `tri_fusion` est $O(n \log(n))$, `tri_bulles` est $O(n^2)$.

- Pour déterminer la complexité en temps :
  a. Exprimer le temps de calcul T(n), par exemple : $T(n) = 4 \cdot n^2 + 3 \cdot n + 25 \cdot \log(n) + 12$
  b. Ôter les facteurs constants de chaque terme : $T(n) = O(n^2 + n + \log(n) + 1)$
  c. Garder le terme dominant pour n très grand : $T(n) = O(n^2)$

Cf. démonstration pour le comptage des opérations dans une double boucle.

Types courants de complexités

Nom croissance	Complexité en temps	Exemples d'algorithmes
Constante	$\mathcal{O}(1)$	Trouver si un nombre binaire est pair
Logarithmique	$\mathcal{O}(\log(n))$	Recherche dichotomique
Linéaire	$\mathcal{O}(n)$	Trouver le minimum d'une liste
Linéarithmique	$\mathcal{O}(n \log(n))$	Tri fusion
Quadratique	$\mathcal{O}(n^2)$	Calculer la couleur moyenne d'une image
Cubique	$\mathcal{O}(n^3)$	Multiplication naïve de matrices $n \times n$
Exponentielle	$\mathcal{O}(2^n)$	Casser un mot de passe par force brute
Factorielle	$\mathcal{O}(n!)$ ou $\mathcal{O}(n^n)$	Générer toutes les permutations d'un ensemble

Commentaires finaux sur la complexité

- Dans certains cas, une complexité élevée peut être désirable.

- Exemple en cryptographie : nous avons parlé du fait qu'il est difficile de trouver une clé à partir d'un hash. En fait, si la fonction de hash a une "randomness" parfaite, c'est exponentiellement difficile en fonction de la longueur de la clé.

  Exemple avec une clé de longueur *n* écrite avec des lettres minuscules :

  Nombre de possibilités à tester : 26^n

  ⇒ l'algorithme `trouver_cle_de_longueur_n(hash)` est $O(26^n) = O(2^n)$.

Commentaires finaux sur la complexité

Autre exemple lié à l'authentification :

- On choisit deux nombres premiers a et b et on calcule $c = a \cdot b$. Le nombre c admet donc deux diviseurs.
- Un site stocke la valeur c associée à notre compte. Pour prouver notre identité, nous devons donner un diviseur de c. Le site peut facilement vérifier que c est divisible par le nombre qu'on lui donne (a ou b), en effectuant la division.
- Si c possède k chiffres décimaux, un attaquant doit quant à lui vérifier $O(10^k)$ nombres (avec une approche naïve).
  Avec une approche non-naïve, la complexité reste quasiment exponentielle.
- Beaucoup d'algorithmes de cryptographie fonctionnent sur ce principe, où tout repose sur la difficulté pratique à résoudre un problème de factorisation.

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
    return a
```

```
function a=tri_bulle(a)
    n = length(a);
    % Parcours des éléments et swap si besoin
end
```

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

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 ...
        # ... si a[i] > a[i+1], swap
    return a
```

```matlab
function a=tri_bulle(a)
    n = length(a);
    % Pour i de 1 à n-1 ...
          % ... si a(i) > a(i+1), swap
end
```

Etape 2 : détailler et préparer la structure.

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)
    return a
```

```julia
function a=tri_bulle(a)
    n = length(a);
    for i=1:n-1
        if a(i) > a(i+1)
            swap(a, i, i+1);
        end
    end
end
```

Etape 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.

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)
    return a
```

```
function a=tri_bulle(a)
    n = length(a);
    for i=1:n-1
        if a(i) > a(i+1)
            swap(a, i, i+1);
        end
    end
end
```

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)
    return a
```

```matlab
function a=tri_bulle(a)
    n = length(a);
    % Pour i_fin de n-1 à 1 ...
    for i=1:i_fin-1
        if a(i) > a(i+1)
            swap(a, i, i+1);
        end
    end
end
```

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)
    return a
```

```
function a=tri_bulle(a)
    n = length(a);
    for i_fin=n:-1:1
        for i=1:i_fin-1
            if a(i) > a(i+1)
                swap(a, i, i+1);
            end
        end
    end
end
```

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)
    return a
```

```
function a=tri_bulle(a)
    n = length(a);
    for i_fin=n:-1:1
        for i=1:i_fin-1
            if a(i) > a(i+1)
                swap(a, i, i+1);
            end
        end
    end
end
```

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]
    return a
```

```julia
function a=tri_bulle(a)
    n = length(a);
    for i_fin=n:-1:1
        for i=1:i_fin-1
            if a(i) > a(i+1)
                tmp = a(i);
                a(i) = a(i+1);
                a(i+1) = tmp;
            end
        end
    end
end
```

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]
    return a
```

```julia
function a=tri_bulle(a)
    n = length(a);
    for i_fin=n:-1:1
        for i=1:i_fin-1
            if a(i) > a(i+1)
                tmp = a(i);
                a(i) = a(i+1);
                a(i+1) = tmp;
            end
        end
    end
```

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

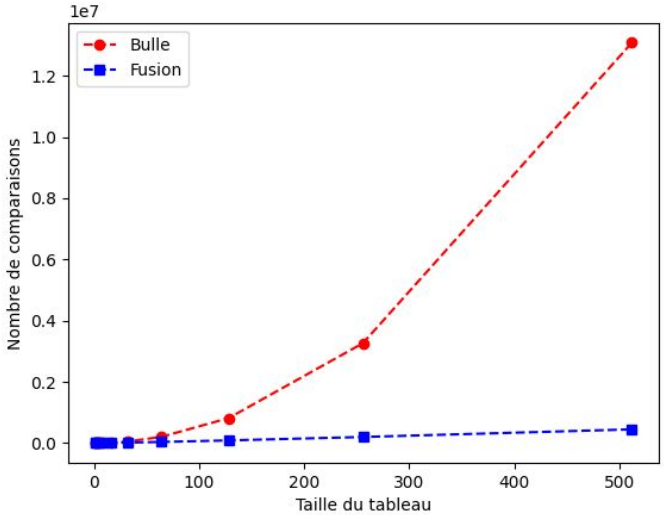
Exemple d'implémentation du tri fusion

```
def fusionner(a1, a2):
    """Prend 2 tableaux triés et les fusionne en un seul tableau trié."""
    if not a1:
        return a2
    elif not a2:
        return a1
    elif a1[0] < a2[0]:
        return [a1[0], ] + fusionner(a1[1:], a2)
    else:
        return [a2[0], ] + fusionner(a1, a2[1:])

def tri_fusion(a):
    n = len(a)
    if n <= 1:
        return a
    a1 = a[0:n//2] #Du début à la moitié
    a2 = a[n//2:n] #De la moitié à la fin
    return fusionner(tri_fusion(a1), tri_fusion(a2))
```

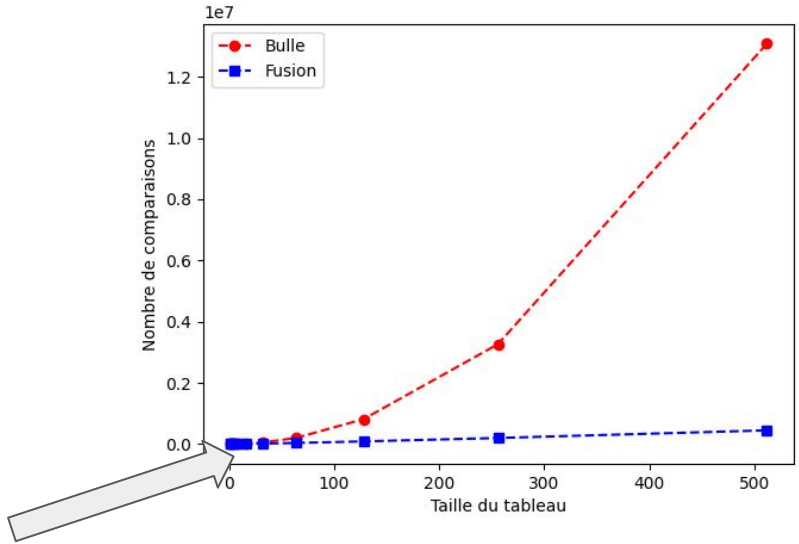
Comparaison entre le tri à bulles et le tri fusion

- On peut compter le nombre de comparaisons effectuées par chaque algorithme.
- Ici, les résultats sont des moyennes sur de nombreux tableaux aléatoires.



Comparaison entre le tri à bulle et le tri fusion

- On peut compter le nombre de comparaisons effectuées par chaque algorithme.
- Ici, les résultats sont des moyennes sur de nombreux tableaux aléatoires.
- Pour de petites valeurs de n, il faut voir les détails d'implémentation.



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)
```

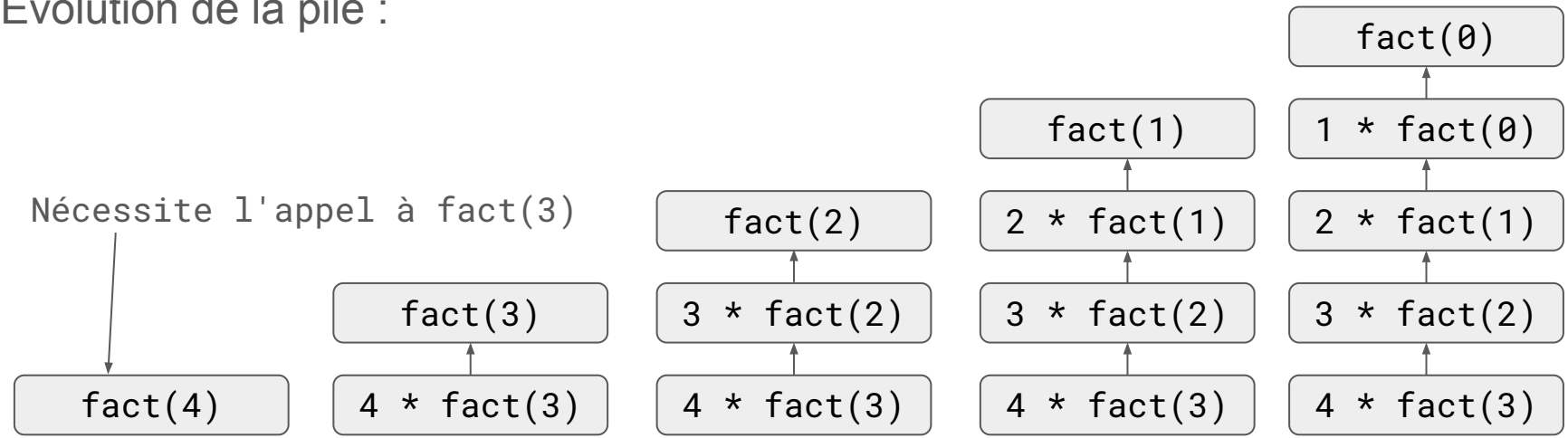
Nécessite 5 valeurs en mémoire

```
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.

Analyse de complexité | **Commentaire final**

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

```
def tri_escroquerie(a): # ou plus honnetement, 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 $O(\max(S))$.