

# Algorithmique Appliquée

BTS SIO SISR

## Algorithmes de recherche et de tri



CHAMBRE DE COMMERCE  
ET D'INDUSTRIE

1<sup>er</sup> ACCÉLÉRATEUR DES ENTREPRISES



Loïc Yvonnet



# Plan

- Algorithmiques classiques
- Recherche en Python
- Recherche linéaire
- Recherche binaire
- Tri en Python
- Algorithmes de tri en  $O(N^2)$
- Partition
- Tri Rapide
- Tri Fusion

# Correction du travail à la maison



# **DM : Retour sur la complexité et les tests**

**Lien vers le sujet de DM.**

# Retour sur les classes de problèmes usuelles en algorithmique

# Familles d'algorithmes classiques

Famille d'algorithmes	Exemple de problème	Exemple d'algorithme
Recherche	Trouver un nombre dans une liste	Recherche binaire
Tri	Trier une liste	Tri Fusion
Graphes	Trouver le plus court chemin	Bellman-Ford
Chaînes de caractères	Trouver une sous-chaîne	Boyer-Moore

# Intérêt

- De nombreux problèmes peuvent se décomposer en **sous-problèmes**.
- Ces sous-problèmes se ramènent souvent à ceux **résolus par les algorithmes classiques**.

# Exemples d'autres problèmes

- Optimisation :
  - Graphes, Tri, Recherche.
- Décision :
  - Graphes, Tri, Recherche.
- Classification :
  - Graphes, Tri, Recherche.
- Résolution d'équations (solver )



# Recherche en Python

# Opérateur **in**

```
L = [1, 4, 8, 62]  
if 4 in L:  
    print("On a trouvé 4")
```



On a trouvé 4

# Egalement pour les **set** et **tuple**

```
S = {1, 4, 8, 62}
if 4 in S:
    print("On a trouvé 4")

T = (1, 4, 8, 62)
if 4 in T:
    print("On a trouvé 4")
```



```
On a trouvé 4
On a trouvé 4
```

# Chaînes de caractères

```
Ch = "1, 4, 8, 62"  
if "4" in Ch:  
    print("On a trouvé 4")
```



```
On a trouvé 4
```

# Dictionnaires

```
D = {"un": 1, "quatre": 4, "huit": 8, "soixante deux": 62}
if "quatre" in D:
    print("On a trouvé quatre")

if 4 in D.values():
    print("On a trouvé 4")
```



On a trouvé quatre  
On a trouvé 4

# Valeur par défaut

```
resultat = D.get("trois", -1)  
print(resultat)
```



-1

# Recherche linéaire

# Implémentation itérative

```
def recherche_lineaire(collection, cle):  
    for i in range(len(collection)):  
        if collection[i] == cle:  
            return i  
    return -1
```



# Preuve d'algorithme

## Preuve triviale

- On parcourt chaque élément de la collection une unique fois, donc l'algorithme s'arrête quand chaque élément est traité.
- Chaque élément est comparé à la clé.
- Donc si un élément est égal à la clé, il sera trouvé.

# Complexité

- $O(N)$  : on parcourt chaque élément une fois.
- $\Omega(1)$  : si le 1er élément est égal à la clé, l'algorithme s'arrête immédiatement.

# Implémentation récursive

```
def recherche_lineaire(collection, cle):  
    def recherche_lineaire_impl(collection, cle, index):  
        if index == len(collection):  
            return -1  
        if collection[index] == cle:  
            return index  
        return recherche_lineaire_impl(collection, cle, index + 1)  
    return recherche_lineaire_impl(collection, cle, 0)
```

# Preuve de la version récursive

- L'index est incrémenté à chaque récursion.
- La récursion s'arrête lorsque l'index est égal à la taille de la collection.
- A chaque récursion, on teste l'élément à l'index actuel.
- La récursion s'arrête si l'élément à l'index actuel est égal à la clé.
- On parcourt donc chaque élément une fois et le reste est identique à la version itérative.

# Recherche binaire

Binary search 

# Implémentation itérative

```
def recherche_binaire(collection, cle):  
    debut = 0  
    fin = len(collection) - 1  
  
    while debut <= fin:  
        milieu = debut + (fin - debut) // 2  
        actuel = collection[milieu]  
  
        if cle < actuel:  
            fin = milieu - 1  
        elif cle > actuel:  
            debut = milieu + 1  
        else:  
            return milieu  
  
    return -1
```

# Illustration de l'exécution

## Recherche du chiffre 9

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----

# Exécution sous forme d'arbre (1/4)

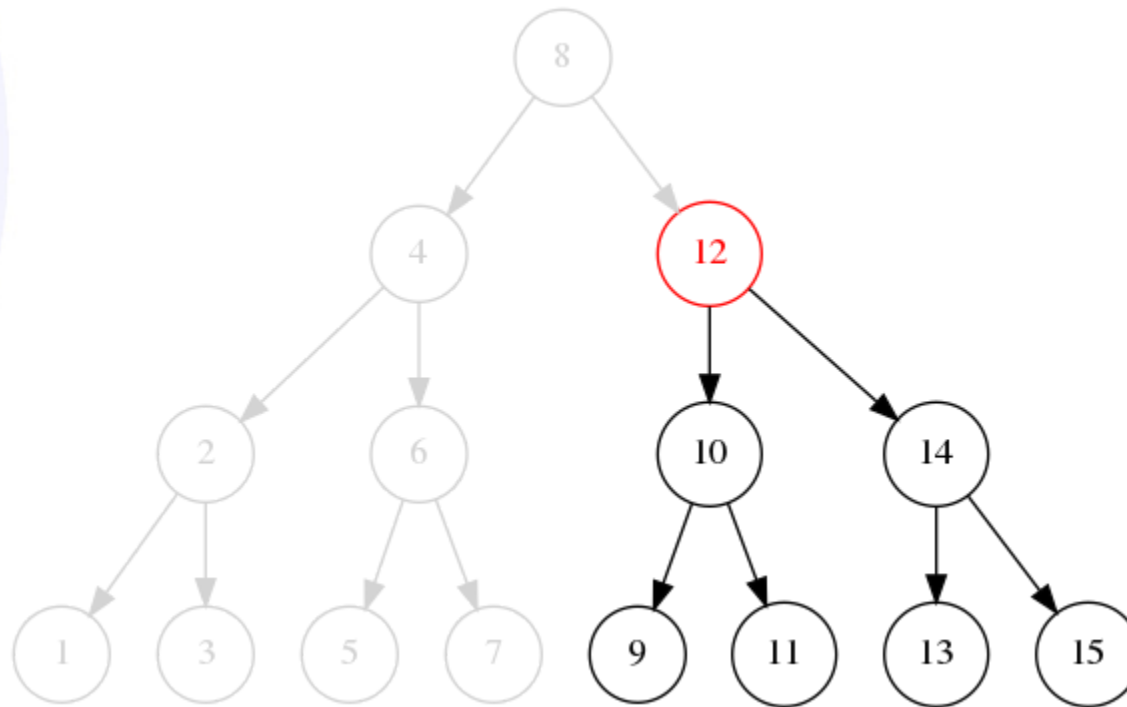
Recherche du chiffre 9





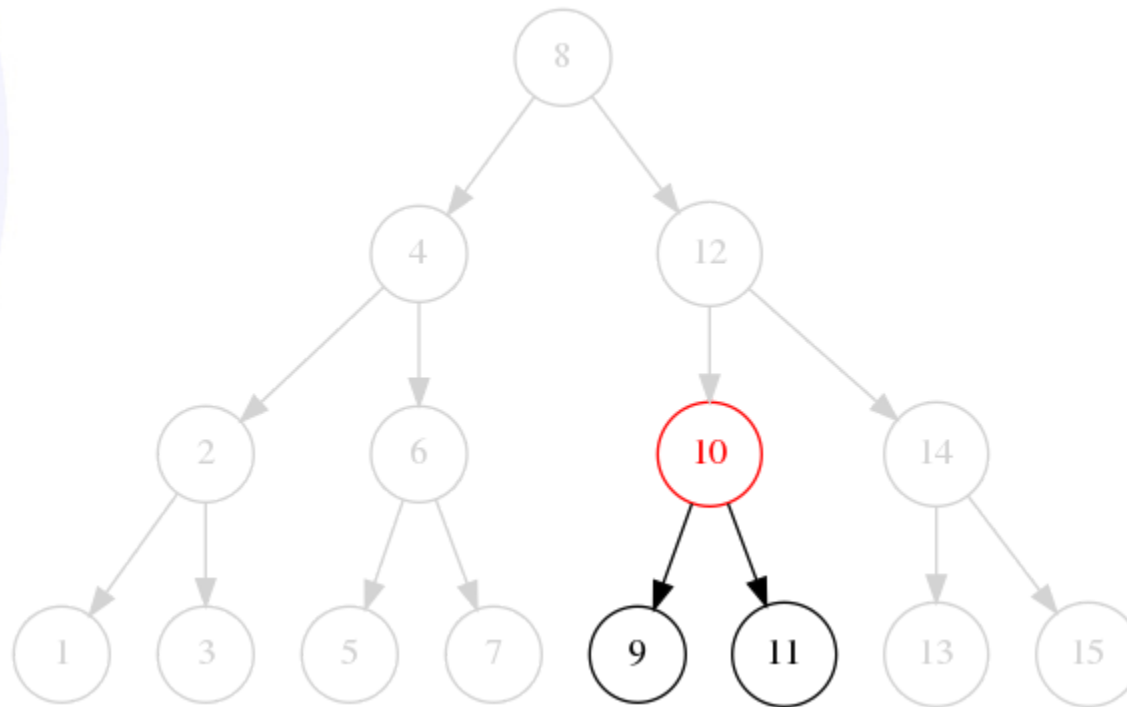
# Exécution sous forme d'arbre (2/4)

Recherche du chiffre 9



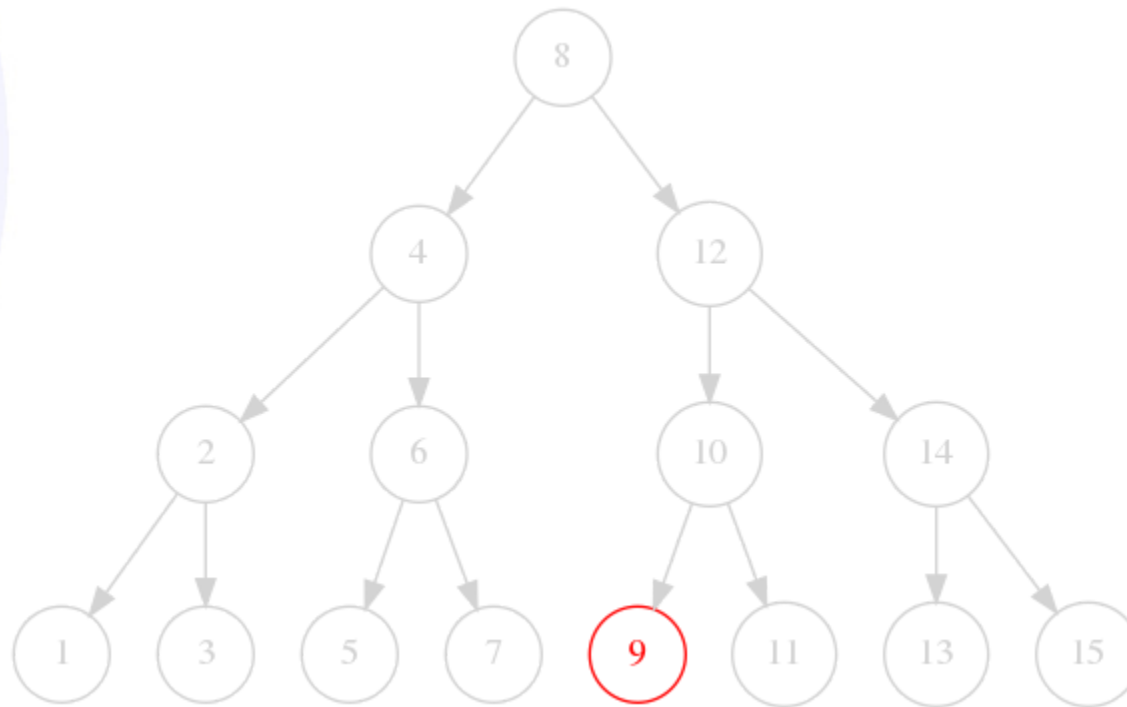
# Exécution sous forme d'arbre (3/4)

Recherche du chiffre 9



# Exécution sous forme d'arbre (4/4)

Recherche du chiffre 9



# Preuve d'algorithme

- A chaque itération :
  - Soit on trouve la clé et l'algorithme s'arrête.
  - Soit l'intervalle de recherche est réduit de moitié et converge vers la clé car la collection est triée :
    - Soit la borne de fin va au milieu,
    - Soit la borne de début va au milieu.





# Complexité

- $O(\log N)$  : on parcourt chaque étage de l'arbre binaire une fois au maximum.
- $\Omega(1)$  : si l'élément du milieu est égal à la clé, l'algorithme s'arrête immédiatement.

# Profondeur de l'arbre (1/6)

- A la racine de l'arbre (profondeur  $p = 1$ ), on a un seul noeud.
- A chaque niveau, on multiplie le nombre de noeuds par 2.
- A un niveau donné, on a donc  $2^{p-1}$  noeuds.

# Profondeur de l'arbre (2/6)

Profondeur $p$	Arbre	Nombre de noeuds
1		$2^0 = 1$ $2^{p-1}$
2		$2^1 = 2$ $2^{p-1}$
3		$2^2 = 4$ $2^{p-1}$
4		$2^3 = 8$ $2^{p-1}$

$$\text{Total} = \sum_{i=0}^{p-1} 2^i = 15$$

# Profondeur de l'arbre (3/6)

Donc le nombre total maximal de noeuds  $N$  est relié à la profondeur  $p$  :

$$N = \sum_{i=0}^{p-1} 2^i$$



# Profondeur de l'arbre (4/6)

Un peu d'arithmétique

$$\begin{aligned} 2^p &= 2^p \times 1 \\ &= 2^p \times (2 - 1) \\ &= 2^{p+1} - 2^p \end{aligned}$$

# Profondeur de l'arbre (5/6)

## Application à la somme des profondeurs

$$\begin{aligned} N &= \sum_{i=0}^{p-1} 2^i = 2^{p-1} + 2^{p-2} + \dots + 2^0 \\ &= (2^p - 2^{p-1}) + (2^{p-1} - 2^{p-2}) + \dots + (2^2 - 2^1) + (2^1 - 2^0) \\ &= 2^p + (-2^{p-1} + 2^{p-1}) + \dots + (-2^1 + 2^1) - 2^0 \\ &= 2^p - 2^0 \\ &= 2^p - 1 \end{aligned}$$

# Profondeur de l'arbre (6/6)

Retour sur le logarithme

$$2^p - 1 = N$$

$$2^p = N + 1$$

$$\log_2(2^p) = \log_2(N + 1)$$

$$p \log_2(2) = \log_2(N + 1)$$

$$p = \log_2(N + 1)$$

# Preuve de la complexité

$$\begin{aligned} O(p) &= O(\log_2(N + 1)) \\ &= O(\log(N)) \end{aligned}$$

On avait vu que la complexité de la recherche binaire était proportionnelle à la profondeur  $p$  de l'arbre, c'est-à-dire  $O(p)$  soit  $O(\log N)$ .

# Implémentation récursive

```
def recherche_binaire(collection, cle):  
    def recherche_binaire_impl(collection, cle, debut, fin):  
        if fin < debut:  
            return -1  
  
        milieu = debut + (fin - debut) // 2  
        actuel = collection[milieu]  
  
        if cle < actuel:  
            return recherche_binaire_impl(collection, cle, debut, milieu - 1)  
        elif cle > actuel:  
            return recherche_binaire_impl(collection, cle, milieu + 1, fin)  
        else:  
            return milieu  
  
    debut = 0  
    fin = len(collection) - 1  
  
    return recherche_binaire_impl(collection, cle, debut, fin)
```

# Relation de récurrence

- Une autre manière de prouver la complexité serait d'utiliser une **relation de récurrence**.
- On pose que le nombre maximal de comparaisons  $C$  pour  $N = 1$  est  $C(1) = 1$ .
- On établit alors que  $C(N) = 1 + C(\frac{N}{2})$ .
- La résolution de la relation de récurrence donne également  $O(\log_2(N))$ .

# TP : Recherche dans une collection



# **TP : Recherche dans une collection**

[Lien vers le sujet de TP.](#)





# Tri en Python

# Tri interne

In-place sort 

```
L = [6, 2, 5, 1, 9, 3, 8, 7, 4]  
L.sort()  
print(L)
```



```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

# Renvoie une liste triée

```
L1 = [6, 2, 5, 1, 9, 3, 8, 7, 4]  
L2 = sorted(L1)  
print(f"L1 = {L1}")  
print(f"L2 = {L2}")
```



```
L1 = [6, 2, 5, 1, 9, 3, 8, 7, 4]  
L2 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

# Cas des tuples

```
T = (6, 2, 5, 1, 9, 3, 8, 7, 4)  
T2 = sorted(T)  
print(T2)
```



```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

# Cas des sets

```
S = {6, 2, 5, 1, 9, 3, 8, 7, 4}  
S2 = sorted(S)  
print(S2)
```



```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

# Cas des chaînes de caractères (1/2)

```
Ch = "6, 2, 5, 1, 9, 3, 8, 7, 4"  
Ch2 = sorted(Ch)  
print(Ch2)
```



```
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',  
 ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',  
 '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

# Cas des chaînes de caractères (2/2)

```
Ch = "6, 2, 5, 1, 9, 3, 8, 7, 4"  
Ch3 = sorted(Ch.replace(" ", "").replace(",", "", ""))  
print(Ch3)
```



```
['1', '2', '3', '4', '5', '6', '7', '8', '9']
```

# Cas des dictionnaires (1/2)

```
D = {"un": 1, "deux": 2, "trois": 3}  
D2 = sorted(D)  
print(D2)
```



```
['deux', 'trois', 'un']
```



## Cas des dictionnaires (2/2)

```
D = {"un": 1, "deux": 2, "trois": 3}  
D3 = sorted(D.values())  
print(D3)
```



```
[1, 2, 3]
```

# Tri décroissant

```
L = [6, 2, 5, 1, 6, 9, 3, 8, 7, 4]  
L.sort(reverse=True)  
print(L)
```



```
[9, 8, 7, 6, 6, 5, 4, 3, 2, 1]
```

# Fonction de tri

```
L = [(4, 3, 2, 1), [3, 2, 1], "ba"]  
L.sort(key=len)  
print(L)
```



```
['ba', [3, 2, 1], (4, 3, 2, 1)]
```

# Tri d'une structure de données

```
from dataclasses import dataclass

@dataclass
class paiement:
    euros: int = 0
    centimes: int = 0

L = [paiement(10, 0), paiement(3, 55), paiement(3, 99)]
L.sort(key=lambda x: (x.euros, x.centimes))
print(L)
```



```
[paiement(euros=3, centimes=55), paiement(euros=3, centimes=99),
paiement(euros=10, centimes=0)]
```

# Combinaison

```
L = [paiement(10, 0), paiement(3, 55), paiement(3, 99)]  
L.sort(key=lambda x: (x.euros, x.centimes), reverse=True)  
print(L)
```



```
[paiement(euros=10, centimes=0), paiement(euros=3, centimes=99),  
paiement(euros=3, centimes=55)]
```

# Tri dans des ordres inversés sur différentes clés

```
from dataclasses import dataclass

@dataclass
class outil:
    nom: str = ""
    masse: float = 0.

L = [outil("marteau", 1.), outil("niveau", 0.5),
      outil("cutter", 0.3), outil("compas", 0.3)]
L.sort(key=lambda x: (-x.masse, x.nom))
print(L)
```



```
[outil(nom='marteau', masse=1.0), outil(nom='niveau', masse=0.5),
 outil(nom='compas', masse=0.3), outil(nom='cutter', masse=0.3)]
```

# Algorithmes de tri en $O(N^2)$

# Intérêt de l'étude du tri

- Dans le cas général, utilisez `sort` et `sorted` pour trier en Python.
- Vous pourriez avoir à implémenter **votre propre structure de données** et **avoir à la trier**.
- Les algorithmes de tri sont des cas d'école à connaître **en entretien d'embauche**.
- Ils présentent un véritable intérêt pédagogique pour aborder les **algorithmes linéarithmiques**.



# Différents algorithmes

- Il existe de nombreux algorithmes de tri.
- Nous allons en étudier 6.
- Ils sont séparés en 2 familles :
  - Algorithmes en  $O(N^2)$  donc **quadratiques**.
  - Algorithmes en  $O(N \log N)$  donc **linéarithmique**.

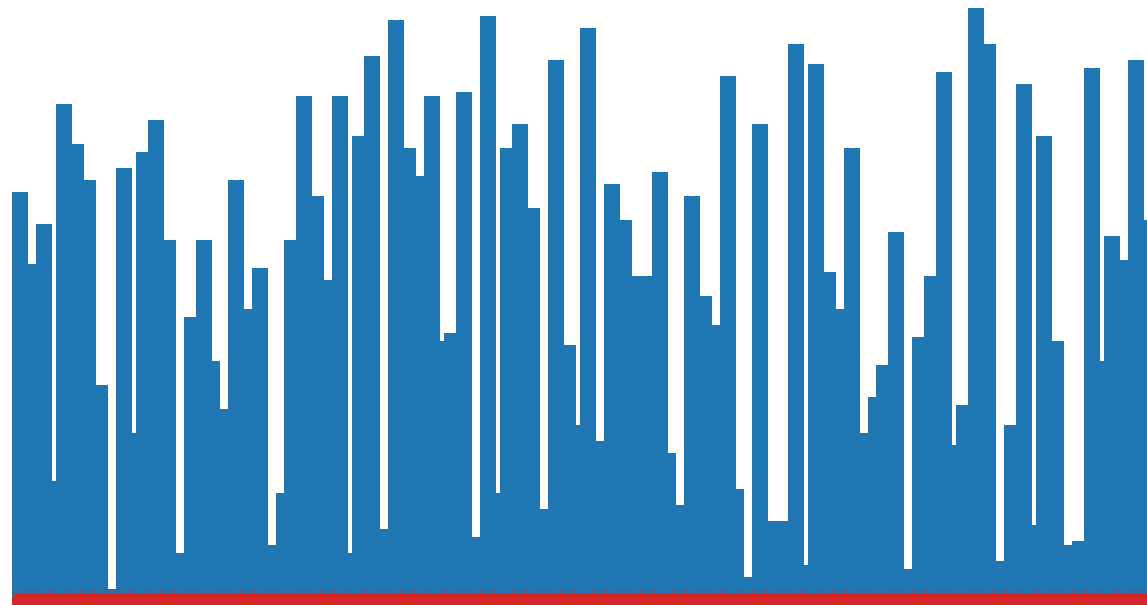
# Tri sélection

## Algorithme (selection sort )

```
def tri_selection(a):  
    N = len(a)  
  
    for i in range(N):  
        min = i  
        for j in range(i, N):  
            if a[j] < a[min]:  
                min = j  
        a[i], a[min] = a[min], a[i]  
  
    return a
```

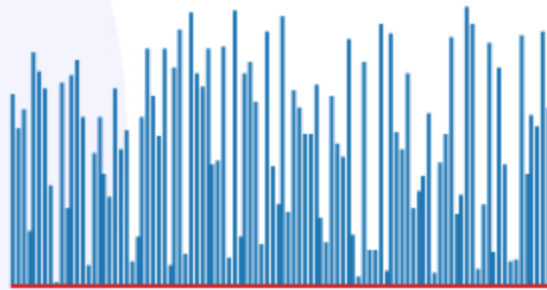
# Tri sélection

## Exécution animée

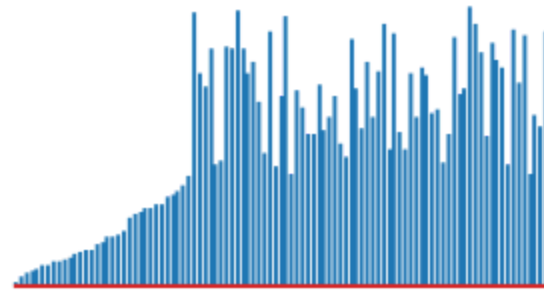


# Tri sélection

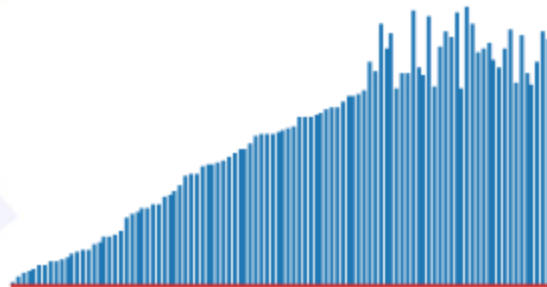
## Quelques étapes d'exécution



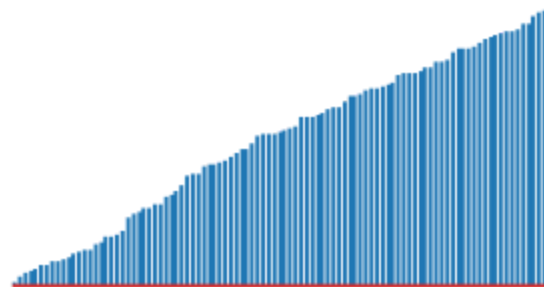
initial



$i = 33$



$i = 66$



trié

# Tri sélection

## Complexité

- On a  $N \frac{N-1}{2}$  comparaisons et  $N$  échanges.
- Par conséquent, on a  $\sim \frac{N^2}{2}$  comparaisons.
- Donc on est en  $O(N^2)$ .

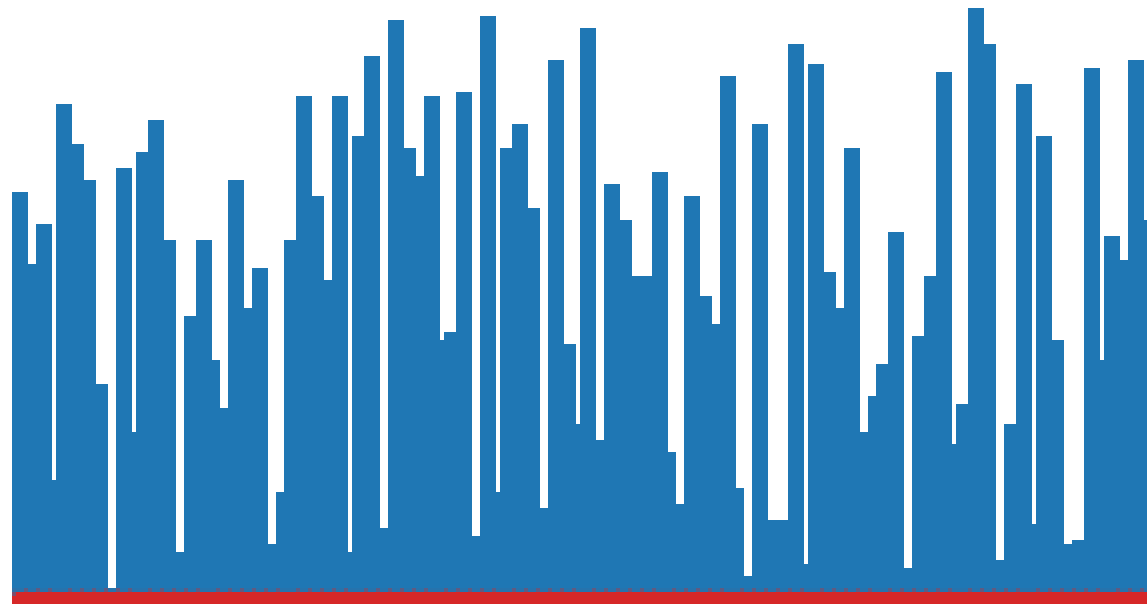
# Tri à bulles

## Algorithme (bubble sort )

```
def tri_bulles(a):  
    N = len(a)  
  
    for i in range(N - 1):  
        for j in range(N - (i + 1)):  
            if a[j] > a[j+1]:  
                a[j], a[j+1] = a[j+1], a[j]  
  
    return a
```

# Tri à bulles

## Exécution animée



# Tri à bulles

## Quelques étapes d'exécution



initial



$i = 33$



$i = 66$



trié



# Tri à bulles

## Complexité

- On a  $N \frac{N-1}{2}$  comparaisons et au pire  $N \frac{N-1}{2}$  échanges.
- Par conséquent, on a  $\sim \frac{N^2}{2}$  comparaisons.
- Donc on est en  $O(N^2)$ .

# Tri insertion

## Algorithme (insertion sort )

```
def tri_insertion(a):  
    N = len(a)  
  
    for i in range(1, N):  
        j = i  
        while j > 0 and a[j] < a[j-1]:  
            a[j], a[j-1] = a[j-1], a[j]  
            j -= 1  
  
    return a
```

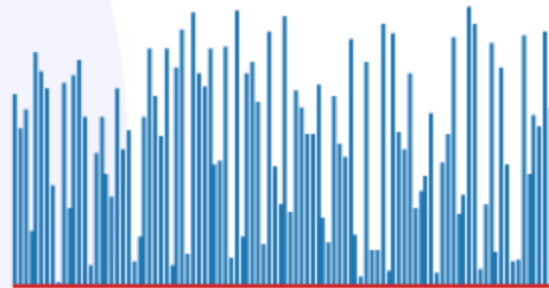
# Tri insertion

## Exécution animée

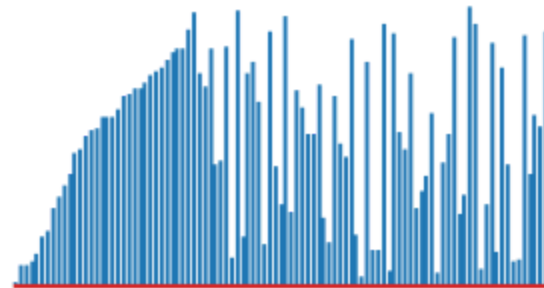


# Tri insertion

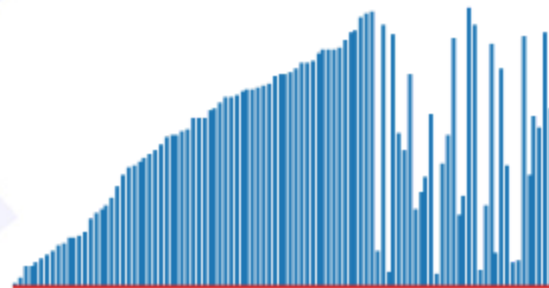
## Quelques étapes d'exécution



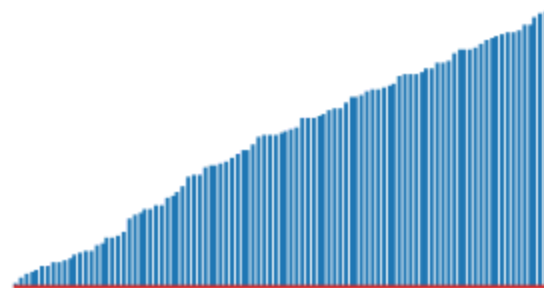
initial



$i = 33$



$i = 66$



trié

# Tri insertion

## Complexité

- On a  $\sim \frac{N^2}{4}$  comparaisons et  $\sim \frac{N^2}{4}$  échanges.
- Donc on est en  $O(N^2)$ .

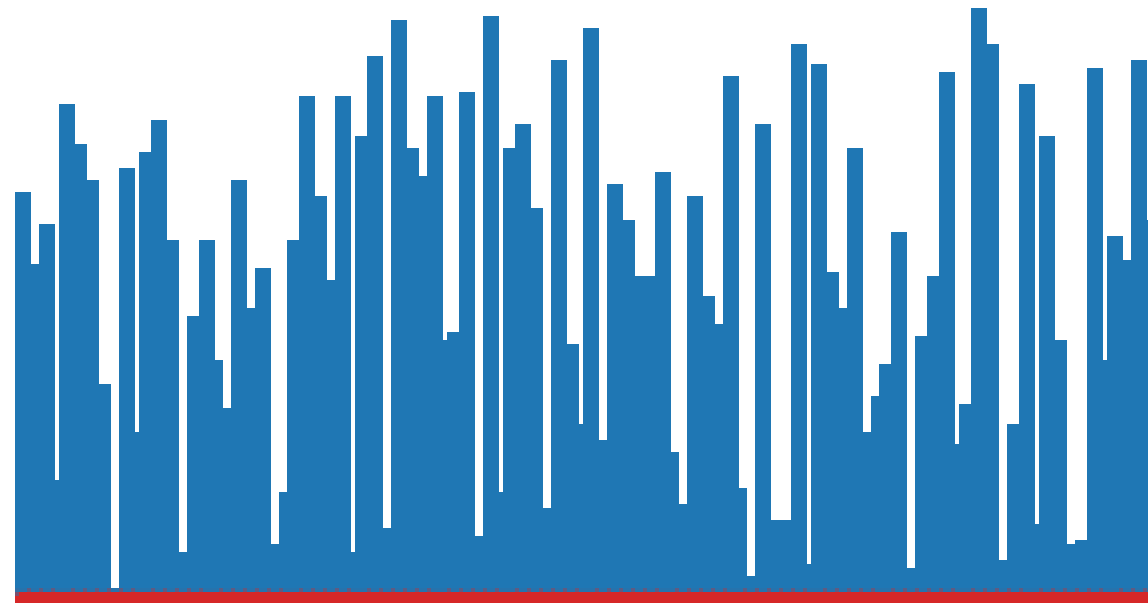
# Tri coquille

## Algorithme (shell sort )

```
def tri_coquille(a):  
    N = len(a)  
    h = 1  
    while h < N // 3:  
        h = 3 * h + 1  
  
    while h >= 1:  
        for i in range(h, N):  
            j = i  
            while j >= h and a[j] < a[j-h]:  
                a[j], a[j-h] = a[j-h], a[j]  
                j -= h  
  
        h //= 3  
  
    return a
```

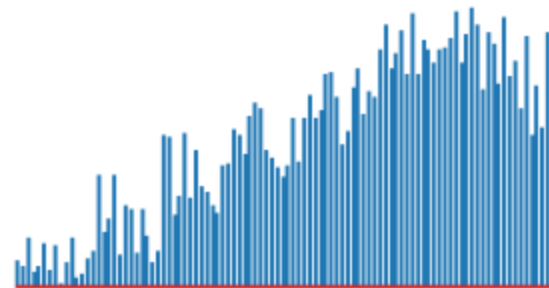
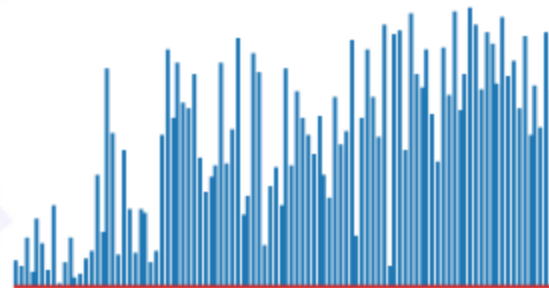
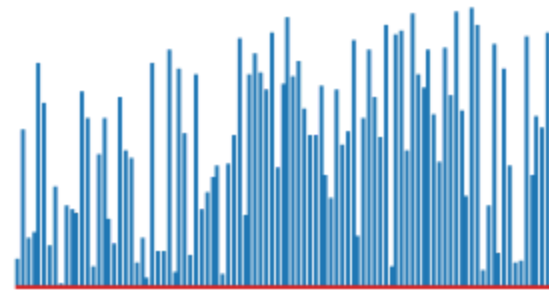
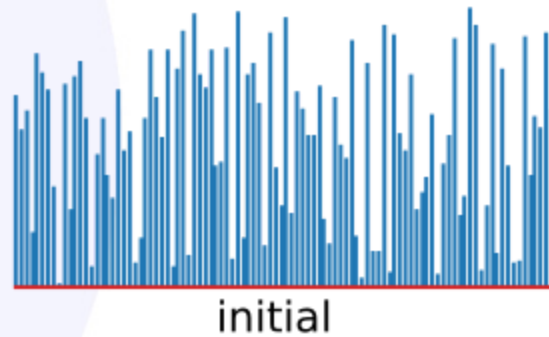
# Tri coquille

## Exécution animée



# Tri coquille

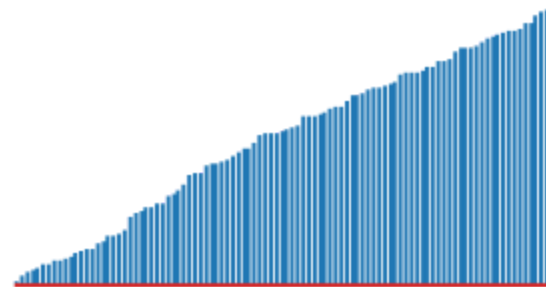
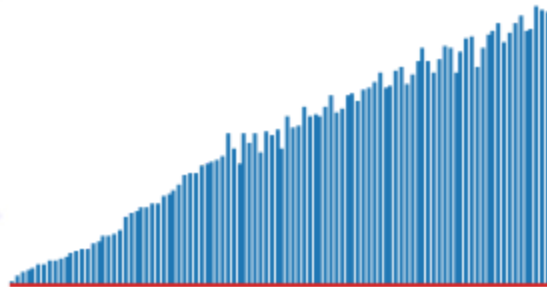
## Quelques étapes d'exécution (1/2)





# Tri coquille

## Quelques étapes d'exécution (2/2)



trié

# Tri coquille

## Complexité

- On a  $\sim \sqrt{N^3}$  comparaisons.
- Donc on est en  $O(N^{3/2})$ .

# Comparaison



---

Tri sélection



Tri à bulles

# Comparaison



Tri sélection



Tri insertion

# Comparaison



Tri insertion



Tri coquille

# Partition : diviser et conquérir

Divide-and-Conquer 

# Diviser et conquérir

- Le principe de **diviser et conquérir** est fondamental en algorithmique.
- On a vu avec la recherche binaire que le fait de diviser en 2 un problème permet de le résoudre beaucoup plus rapidement.

# Rappel sur la partition

- En **mathématiques**, une partition d'un ensemble est un regroupement de ses éléments dans des sous-ensembles non-vides tel que chaque élément est inclu dans exactement un sous-ensemble.
- Exemple :
  - pour l'ensemble  $E = \{6, 2, 5, 1, 9, 3, 8, 7, 4\}$ ,
  - le sous-ensemble  $C_1 = \{2, 5, 1, 3, 4\}$ ,
  - le sous-ensemble  $C_2 = \{6\}$ ,
  - le sous-ensemble  $C_3 = \{9, 8, 7\}$ ,
  - on a  $C_1, C_2, C_3$  qui forment une partition de  $E$ .



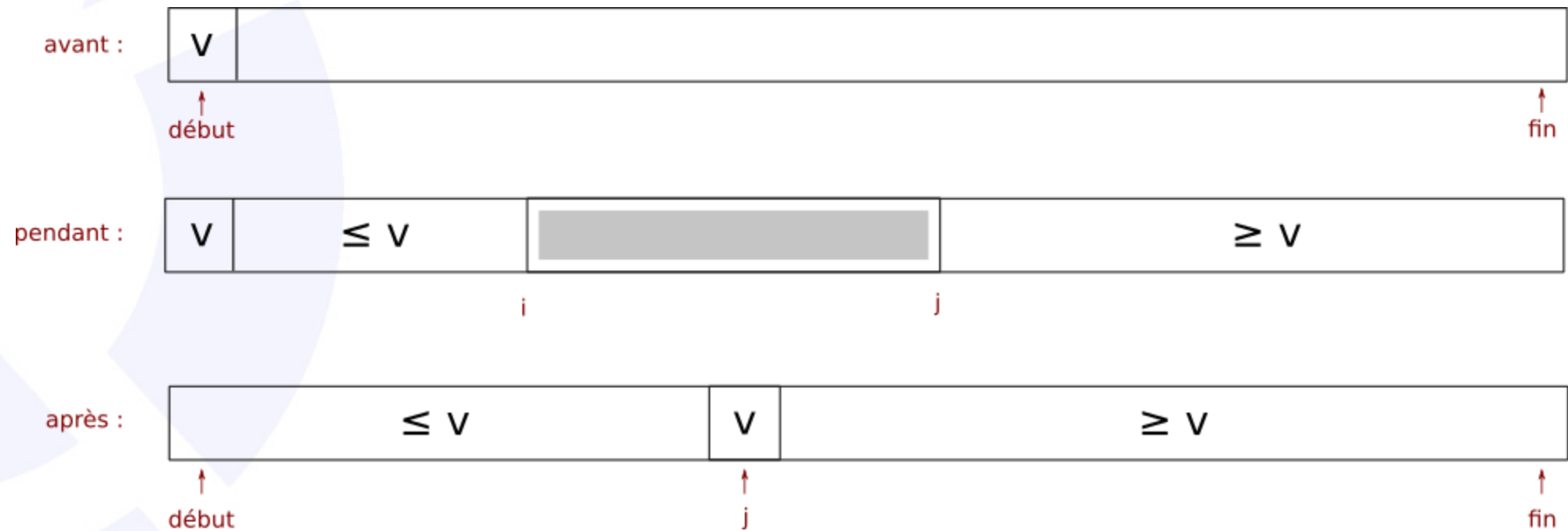
# Partitionner en 2

- **L'algorithme de partition** vise à diviser un ensemble en 2 sous-ensembles :
  - L'ensemble des éléments strictement plus petit qu'une valeur.
  - L'ensemble des autres éléments.

# Partition

```
def partition(e):  
    N = len(e)  
    valeur = e[0]  
    i = 0  
    j = N  
  
    while True:  
        i += 1 # Garanti la progression à droite  
        while e[i] < valeur and i != N: i += 1 # Scan vers la droite  
  
        j -= 1 # Garanti la progression à gauche  
        while valeur < e[j] and j != 0: j -= 1 # Scan vers la gauche  
  
        if i >= j: break # Si les indices se croisent on s'arrête  
  
        # Echange des éléments entre les 2 partitions  
        e[j], e[i] = e[i], e[j]  
  
    # Met la valeur de partitionnement entre les 2 partitions  
    e[j], e[0] = e[0], e[j]
```

# Illustration de l'exécution



# Example

```
L = [6, 2, 5, 1, 9, 3, 8, 7, 4]  
partition(L)  
print(L)
```



```
[3, 2, 5, 1, 4, 6, 8, 7, 9]
```

```
L = [6, 2, 5, 1, 6, 9, 3, 8, 7, 4]  
partition(L)  
print(L)
```



```
[3, 2, 5, 1, 4, 6, 9, 8, 7, 6]
```

# Complexité

- On a  $N + 1$  comparaisons.
- On est donc en  $\sim N$ , et  $O(N)$ .

# Tri Rapide

Quick Sort 

# Tri rapide

## Introduction

- Le tri rapide a une **meilleure complexité** que les autres algorithmes de tri vu jusqu'ici.
- Il est également plus complexe à comprendre.
- Il repose sur la **partition** et une définition **naturellement récursive**.



# Tri rapide - Partition

```
def partition(a, debut, fin):  
    i = debut  
    j = fin + 1  
    valeur = a[debut]  
  
    while True:  
        i += 1  
        while a[i] < valeur and i != fin: i += 1  
  
        j -= 1  
        while valeur < a[j] and j != debut: j -= 1  
  
        if i >= j: break  
  
        a[j], a[i] = a[i], a[j]  
  
    a[j], a[debut] = a[debut], a[j]  
  
    return j
```

# Tri rapide

## Algorithme (quick sort )

```
def tri_rapide_recuratif(a, debut, fin):  
    if fin > debut:  
        j = partition(a, debut, fin)  
        tri_rapide_recuratif(a, debut, j - 1)  
        tri_rapide_recuratif(a, j + 1, fin)
```

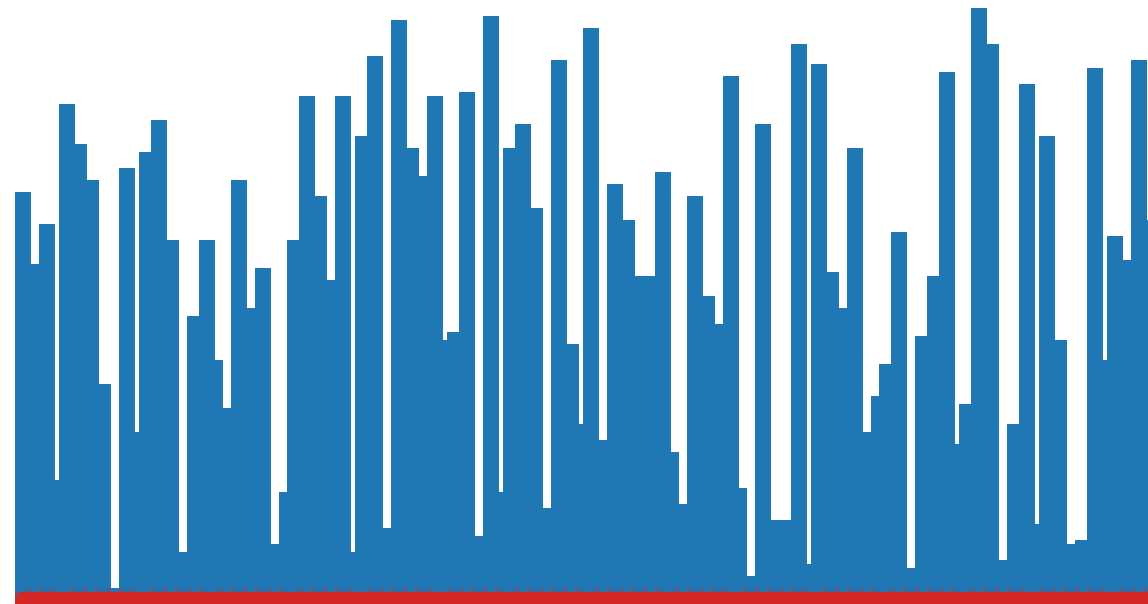
# Tri rapide

## Interface

```
def tri_rapide(a):  
    N = len(a)  
    tri_rapide_recuratif(a, 0, N - 1)
```

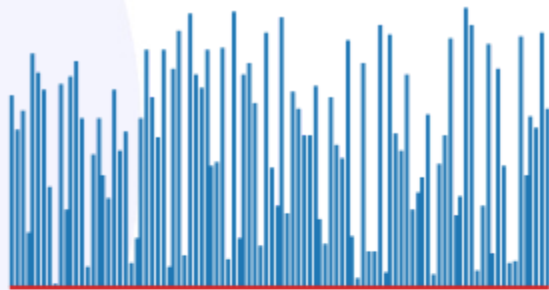
# Tri rapide

## Exécution animée

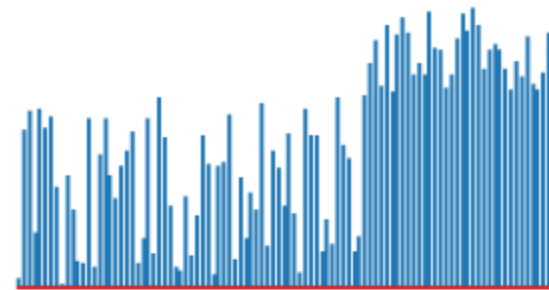


# Tri rapide

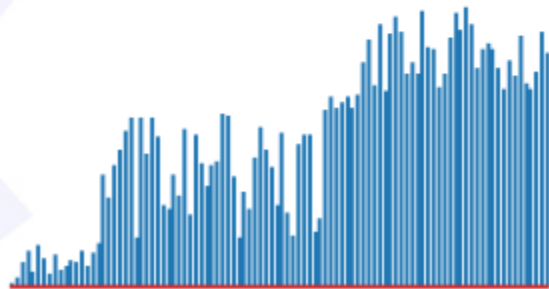
## Quelques étapes d'exécution (1/2)



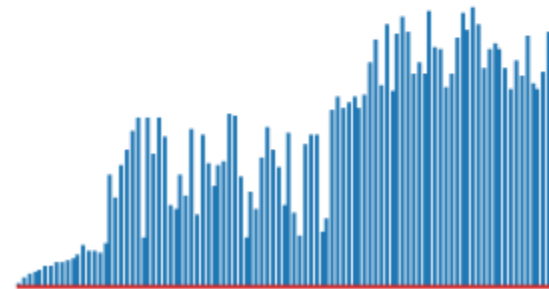
initial



partition 1



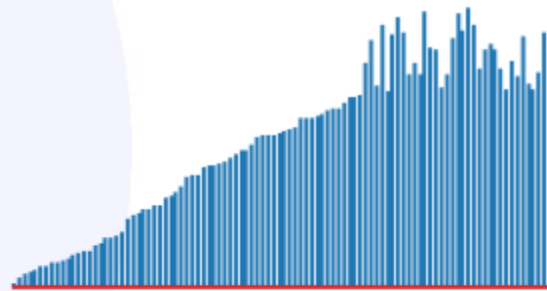
partition 4



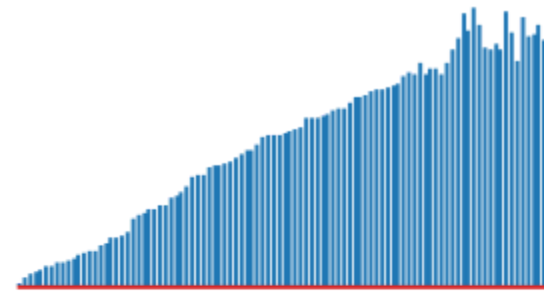
partition 10

# Tri rapide

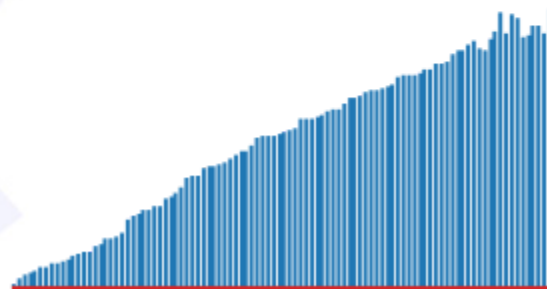
## Quelques étapes d'exécution (2/2)



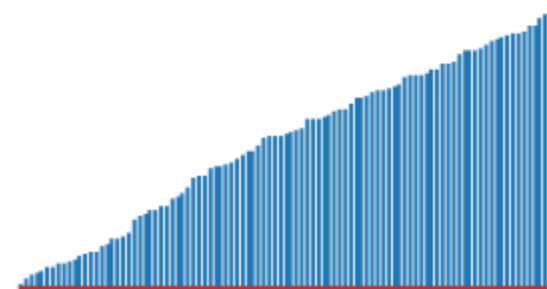
partition 45



partition 50



partition 60



trié

# Tri rapide

## Complexité

- On a  $\sim 2N \log N$  comparaisons en moyenne.
- On a  $\sim \frac{N^2}{2}$  comparaisons dans le pire cas.
- Comme on peut facilement se prévenir du pire cas, on admet  $O(N \log N)$  en pratique.

# Le *pire* cas (1/2)

- Le pire cas survient lorsque la collection est **déjà triée**.
- En effet, le partitionnement n'a **aucun effet** dans ce cas.
- On a vu dans la partie sur l'algorithme de partition que la valeur `v` ne se retrouve pas forcément au milieu.
- Si la valeur `v` se retrouve **toujours en premier**, cela signifie que la collection est déjà triée et le tri rapide sera lent et inutile.



# ***Le pire cas (2/2)***

- On peut se prévenir du pire cas en **testant initialement** si le tableau est trié.
- On peut s'éloigner du pire cas en **mélangeant les éléments**.

# Le *meilleur* cas

- Le tri rapide est à son maximum lorsque **v** se retrouve toujours *exactement* au milieu à chaque partitionnement.
- Dans ce cas, la relation de récurrence  $C$  définissant le nombre de comparaisons  $C_N = 2C_{N/2} + N$ .
- $C_N \sim N \log N$ , ce qui est un début de preuve pour la complexité de cet algorithme.

# Tri Fusion

Merge Sort 

# Tri Fusion

## Introduction

- Dans le tri rapide, on partitionne en 2 sous-ensembles puis on applique l'algorithme récursivement à chaque sous-ensemble.
- Dans le tri fusion, on fait les opérations dans le sens inverse : on applique **d'abord** récursivement l'algorithme puis on **fusionne** les résultats.
- C'est la fusion qui entraine le tri.

# Tri Fusion - Fusion

```
def fusion(a, debut, milieu, fin):  
    i = debut  
    j = milieu + 1  
    auxiliaire = a[:]  
    for k in range(debut, fin + 1):  
        if i > milieu:  
            a[k] = auxiliaire[j]  
            j += 1  
        elif j > fin:  
            a[k] = auxiliaire[i]  
            i += 1  
        elif auxiliaire[j] < auxiliaire[i]:  
            a[k] = auxiliaire[j]  
            j += 1  
        else:  
            a[k] = auxiliaire[i]  
            i += 1
```

# Tri Fusion

## Algorithme (merge sort )

```
def tri_fusion_rekursif(a, debut, fin):  
    if fin > debut:  
        milieu = debut + (fin - debut) // 2  
        tri_fusion_rekursif(a, debut, milieu)  
        tri_fusion_rekursif(a, milieu + 1, fin)  
        fusion(a, debut, milieu, fin)
```

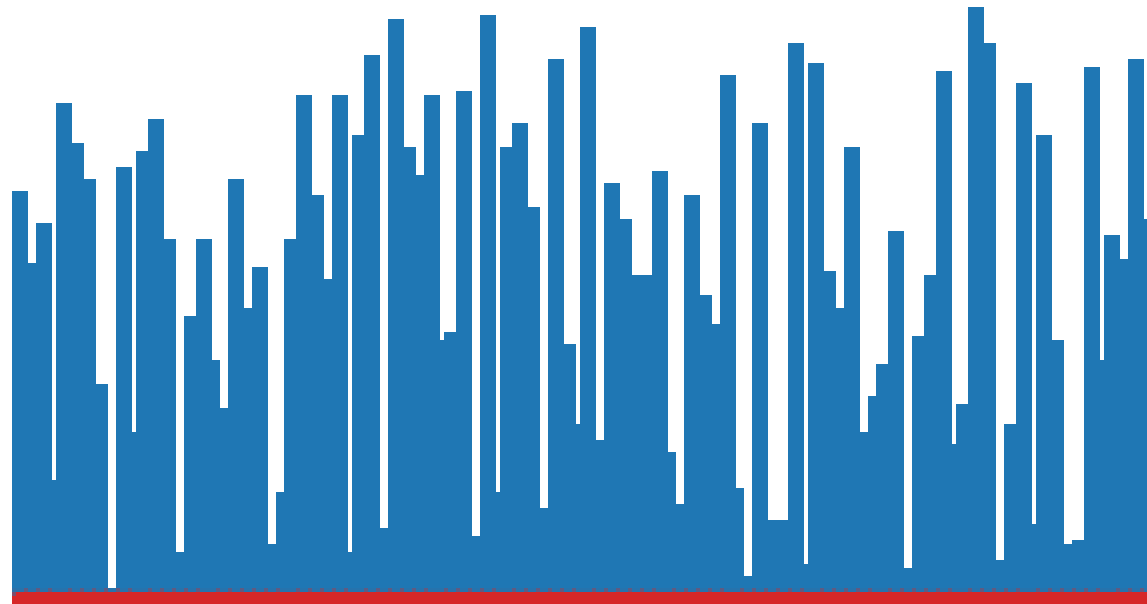
# Tri Fusion

## Interface

```
def tri_fusion(a):  
    N = len(a)  
    tri_fusion_recurisif(a, 0, N - 1)
```

# Tri Fusion

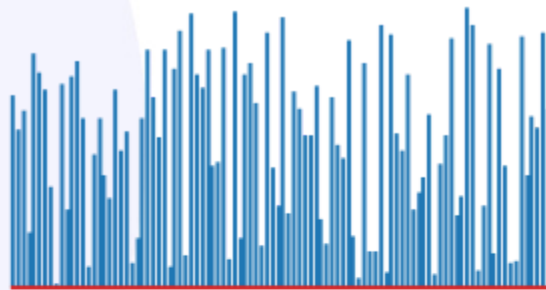
## Exécution animée



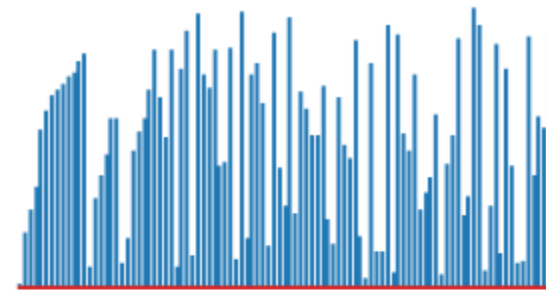


# Tri Fusion

## Quelques étapes d'exécution (1/2)



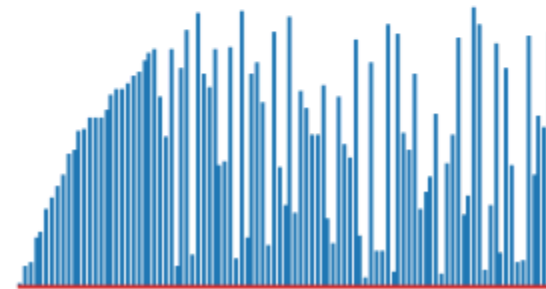
initial



fusion 22



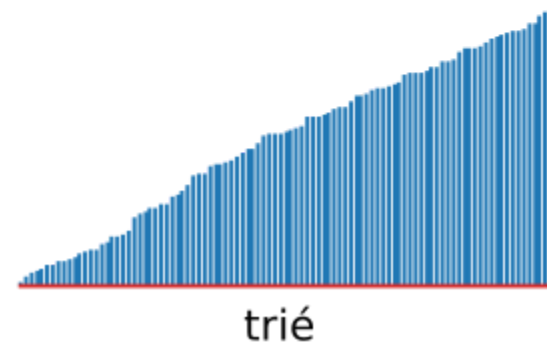
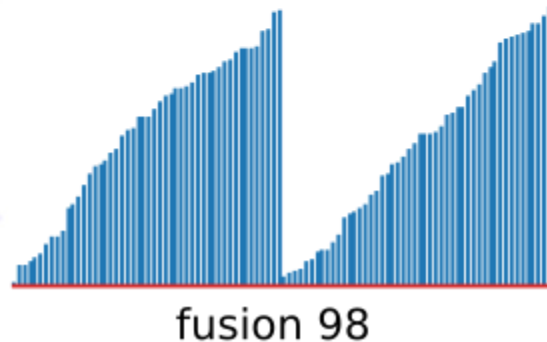
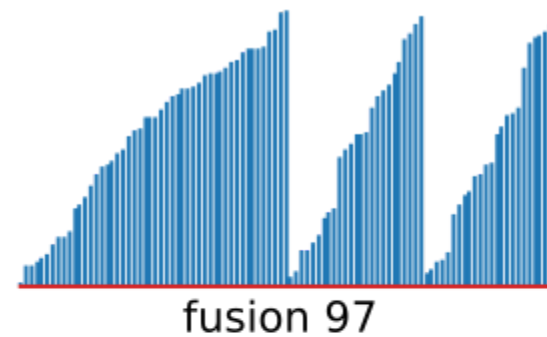
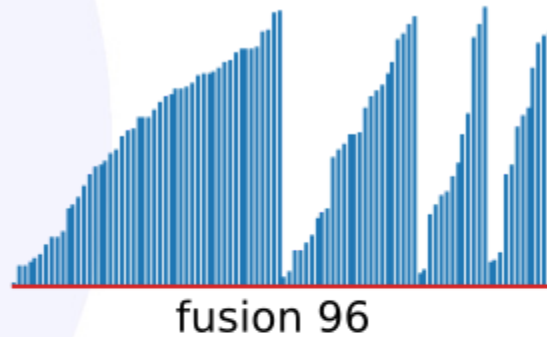
fusion 23



fusion 24

# Tri Fusion

## Quelques étapes d'exécution (2/2)



# Tri Fusion

## Complexité

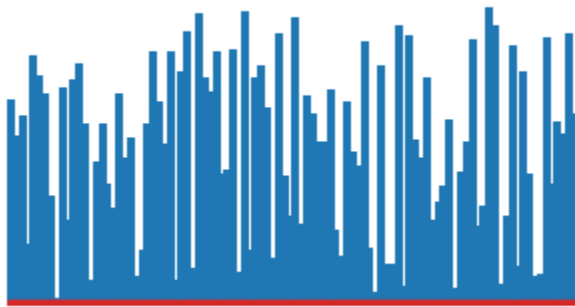
- La complexité de la fusion elle-même est  $\Theta(N)$
- Pour le tri fusion :
  - On a entre  $\frac{1}{2} N \log N$  et  $N \log N$  comparaisons.
  - On est en  $\Theta(N \log N)$ .

# Tri Fusion

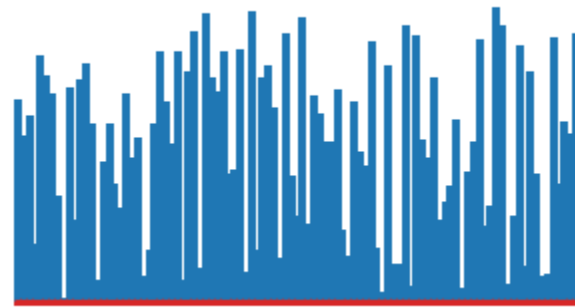
## Intuition de preuve

- A chaque récursion, on divise l'espace en 2.
- On peut dessiner un arbre binaire pour représenter les appels.
- La profondeur  $p$  de cet arbre binaire est proportionnel à  $O(\log N)$ .
- On a donc  $\log N$  fusions, soit  $O(N \log N)$  opérations au total.

# Comparaison



Tri rapide



Tri fusion

# Existe-t-il un algorithme plus efficace ?

- Autrement dit, existe-t-il un algorithme ayant une meilleure complexité que  $O(N \log N)$  pour trier une collection ?
- **Non**, il est possible de prouver que **la meilleure complexité** pour le tri est  $O(N \log N)$ .
- En revanche, les implémentations peuvent recevoir de **petites améliorations**.
- Par exemple, il est possible de **paralléliser** tri rapide ou tri fusion.

# Éléments de preuve

- On considère que tous les éléments à trier sont distincts.
- On construit un **arbre binaire** de toutes les **permutations possibles**.
- Il y a  $N!$  permutations possibles (par définition).
- On s'intéresse à la **profondeur**  $p$  et comme l'arbre est binaire, on a  $O(p) = O(\log(N!))$ .
- L'**approximation de Stirling** nous donne  $\log(N!) \sim N \log(N)$ .

# TP : Tri de collections





# **TP : Tri de collections**

[Lien vers le sujet de TP.](#)