MPII
ITC
Séquence 7

Résumé
Partie 1 - Tris quadratiques

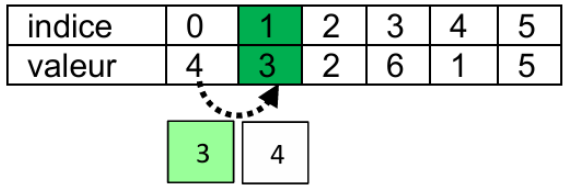
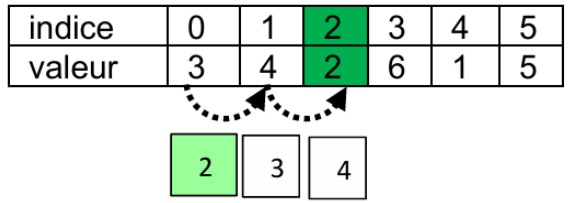
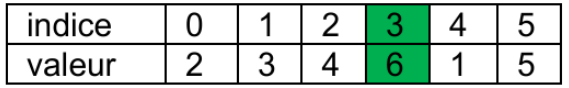
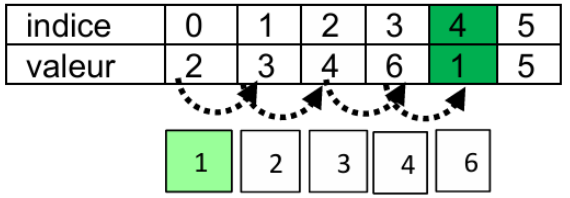
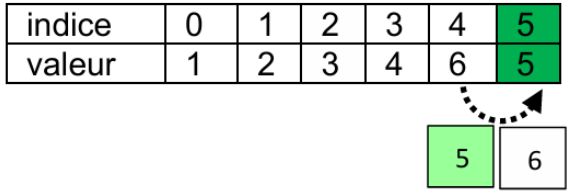
2.1 Tri par sélection

i = 1	j de 2 à 5 **imin ← 1**	indice 0 1 2 3 4 5 valeur 1 3 7 6 9 8
	Test 1 = 1 Pas de permut	indice 0 1 2 3 4 5 valeur 1 3 7 6 9 8
i = 2	j de 3 à 5 **imin ← 3**	indice 0 1 2 3 4 5 valeur 1 3 7 6 9 8
	Test 3 ≠ 2	indice 0 1 2 3 4 5 valeur 1 3 **6** **7** 9 8
i = 3	j de 4 à 5 **imin ← 3**	indice 0 1 2 3 4 5 valeur 1 3 6 7 9 8
	Test 3 = 3 Pas de permut	indice 0 1 2 3 4 5 valeur 1 3 6 7 9 8
i = 4	j de 5 à 5 **imin ← 5**	indice 0 1 2 3 4 5 valeur 1 3 6 7 9 8
	Test 5 ≠ 4	indice 0 1 2 3 4 5 valeur 1 3 6 7 8 9

```
pour i ← 0 à n − 2 faire
        imin ← i
        pour j ← i + 1 à n-1 faire
                si A [j] < A [imin]
                   imin ← j
                fin si
        fin pour
        si imin ≠ i
                temp ← A [imin]
                A [imin] ← A [i]
                A [i] ← temp
        fin si
fin pour
```

- En place
- Non stable
- Complexité $O(n^2)$

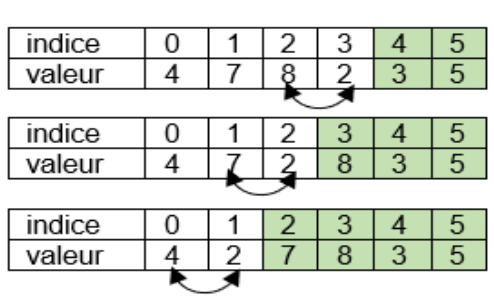
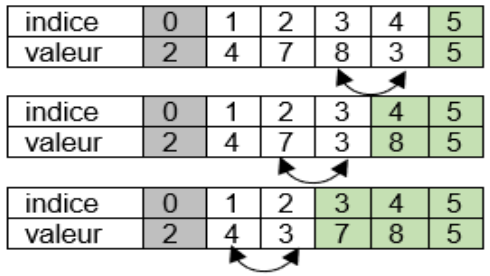
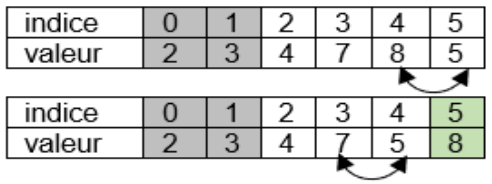
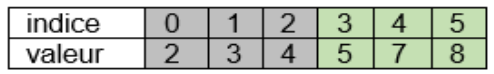
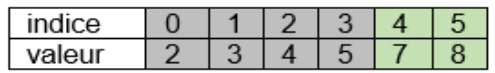
2.2 Tri par insertion

Boucle i	Fin du tant que	Décalage(s) ⟵ INSERTION carte	Carte
i = 1 carte⟵ A[1] = 3	j = -1 INSERTION de la carte : A[0] = 3		
i = 2 carte⟵ A[2] = 2	j = -1 Placement de la carte : A[0] = 2		
i = 3 carte⟵ A[3] = 6	A[2] > 6 faux j = 3 Placement de la carte : A[3] = 6		
i = 4 carte⟵ A[4] = 1	j = -1 Placement de la carte : A[0] = 1		
i = 5 carte⟵ A[5] = 5	A[3] > 4 faux j = 4 Placement de la carte : A[4] = 5		

```
pour i ← 1 à n-1 faire
        carte ← A[i]
        j ← i - 1
        tant que  j≥0 et que A[j] > carte  faire
                A [j+1] ← A [j]
                j ← j-1
        fin tant que
        A [j+1] ← carte
fin pour
```

- En place
- Stable
- Complexité $O(n^2)$

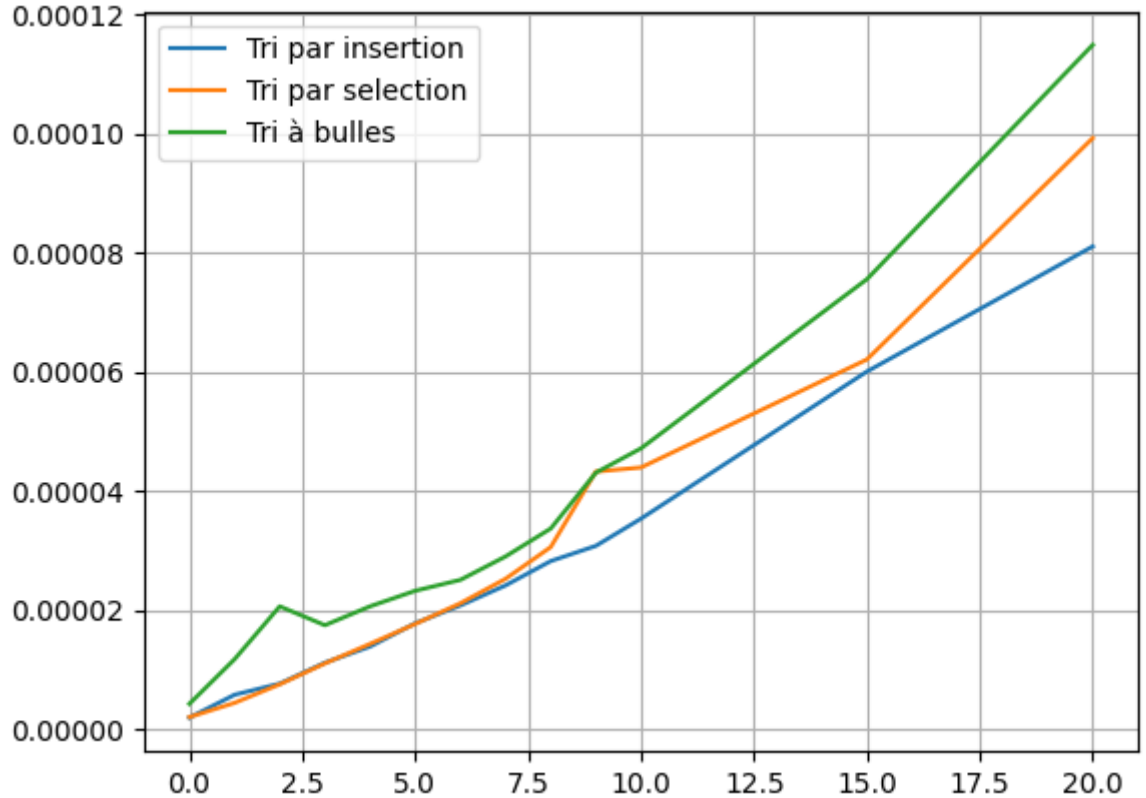
2.3 Tri à bulles

Boucle i	Boucle j	Echanges après test positif
i = 0	j de 5 à 1	
i = 1	j de 5 à 2	
i = 2	j de 5 à 3	
i = 3	j de 5 à 4	
i = 4	j de 5 à 5	


```
pour i ← 0 à n-2 faire
        pour j ← n – 1 à i + 1 par pas de (-1) faire
                si  A[j] < A [j-1]    faire
                        temp ← A [j]
                        A [j] ← A [j-1]
                        A [j-1] ← temp
                fin si
        fin pour
fin pour
```

- En place
- Stable
- Complexité $O(n^2)$


Tri par insertion
Tri par selection
Tri à bulles
0.00175
0.00150
0.00125
0.00100
0.00075
0.00050
0.00025
0.00000
0
20
40
60
80
100


Tri par insertion
Tri par selection
Tri à bulles
0.00012
0.00010
0.00008
0.00006
0.00004
0.00002
0.00000
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

MPII
ITC
Séquence 8

Tris
Partie 2 - Tris rapides - Optimisation
I - Tri fusion et tri rapide

« Diviser pour (mieux)régner »

Caractéristiques des algorithmes de tri récursifs :

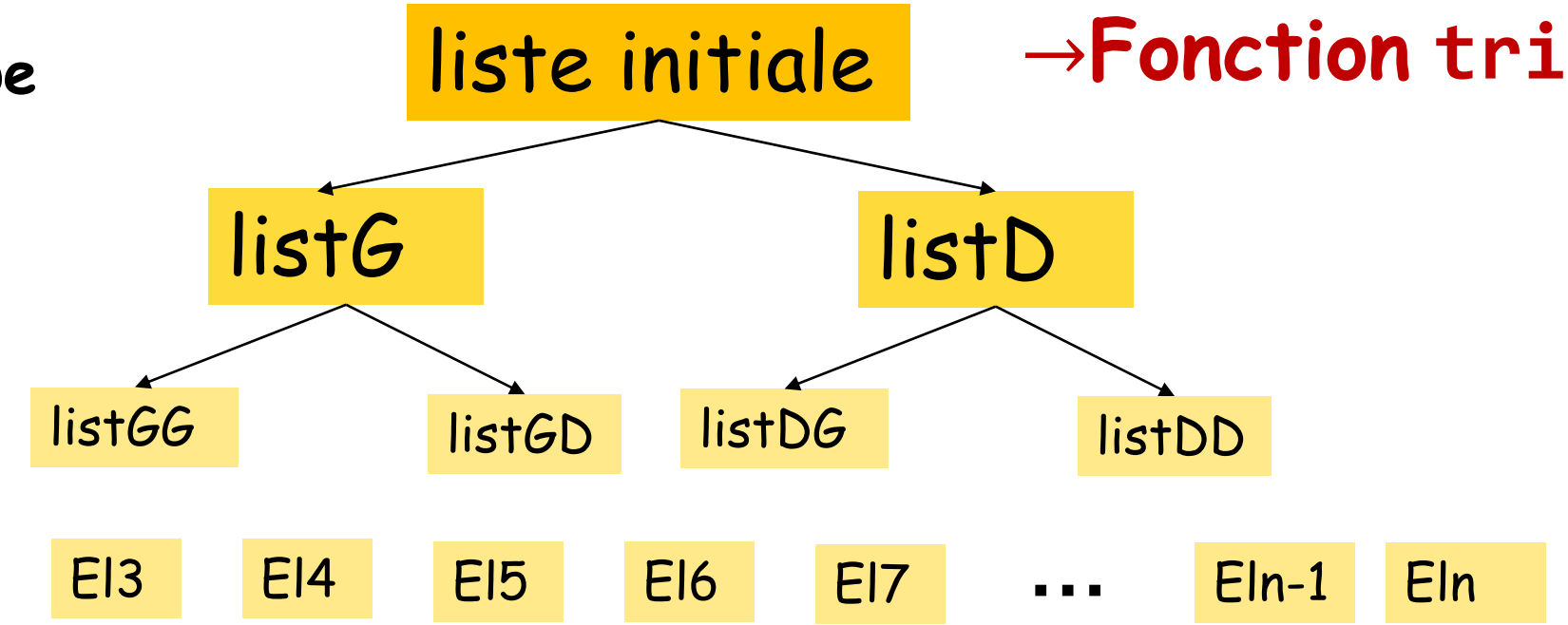
- « **Diviser** »: séparation de la liste initiale en listes intermédiaires
- « **Régner** »: ranger par ordre les listes, sous-listes, sous-sous-listes… (croissant/décroissant)
- « **Combiner** »: former la liste finale à l'aide des sous listes triées

1. ALGORITHMES RÉCURSIFS

1.1. Tri fusion

1.1.1. Principe

- « **Diviser** » :



→**Fonction triFusion**

- « **Régner** » :

- « **Combiner** » : Fusion des deux sous-listes triées

→**Fonction fusion**

  - Parcours en parallèle des 2 listes
  - Placement de la plus petite valeur dans une liste résultat.

1. ALGORITHMES

1.1. Tri fusion

1.1.2. Algorithme :

vérification fin de liste droite ou gauche

Insertion en parallèle

cas où une liste droite ou gauche est finie

```
Fonction fusion(listG,listD)
listFusion ← [ ]
iG ← 0 ; iD ← 0
nG← taille(listG) ; nD← taille(listD)
Tant que taille(listFusion) < nG + nD faire
    si iG < nG et iD < nD
        si listG [iG] < listD [iD]
            listFusion ⊕ listG [iG]
            iG ← iG + 1
        sinon
            listFusion ⊕ listD [iD]
            iD ← iD + 1
        fin si
    sinon
        si iG == nG
            listFusion ⊕ listD [iD . . . nD-1]
        sinon
            listFusion ⊕ listG [iG . . . nG-1]
        fin si
    fin si
fin tant que
retourner listFusion
```

1. ALGORITHMES

1.1. Tri fusion

1.1.2. Algorithme :

```
Fonction triFusion(listaTrier)
n← taille(listaTrier)
si n ≤ 1
    retourner listaTrier
Gauche ← triFusion(listaTrier [0 . . . n//2 [)
Droite ← triFusion(listaTrier [n//2 . . . n [)
listeTriee ← fusion(Gauche, Droite)
retourner listeTriee
```

Condition d'arrêt

```
Fonction fusion(listG,listD)
listFusion ← [ ]
iG ← 0 ; iD ← 0
nG← taille(listG) ; nD← taille(listD)
Tant que taille(listFusion) < nG + nD faire
    si iG < nG et iD < nD
        si listG [iG] < listD [iD]
            listFusion ⊕ listG [iG]
            iG ← iG + 1
        sinon
            listFusion ⊕ listD [iD]
            iD ← iD + 1
        fin si
    sinon
        si iG == nG
            listFusion ⊕ listD [iD . . . nD-1]
        sinon
            listFusion ⊕ listG [iG . . . nG-1]
        fin si
    fin si
fin tant que
retourner listFusion
```

1.1. Tri fusion

liste initiale

triFusion

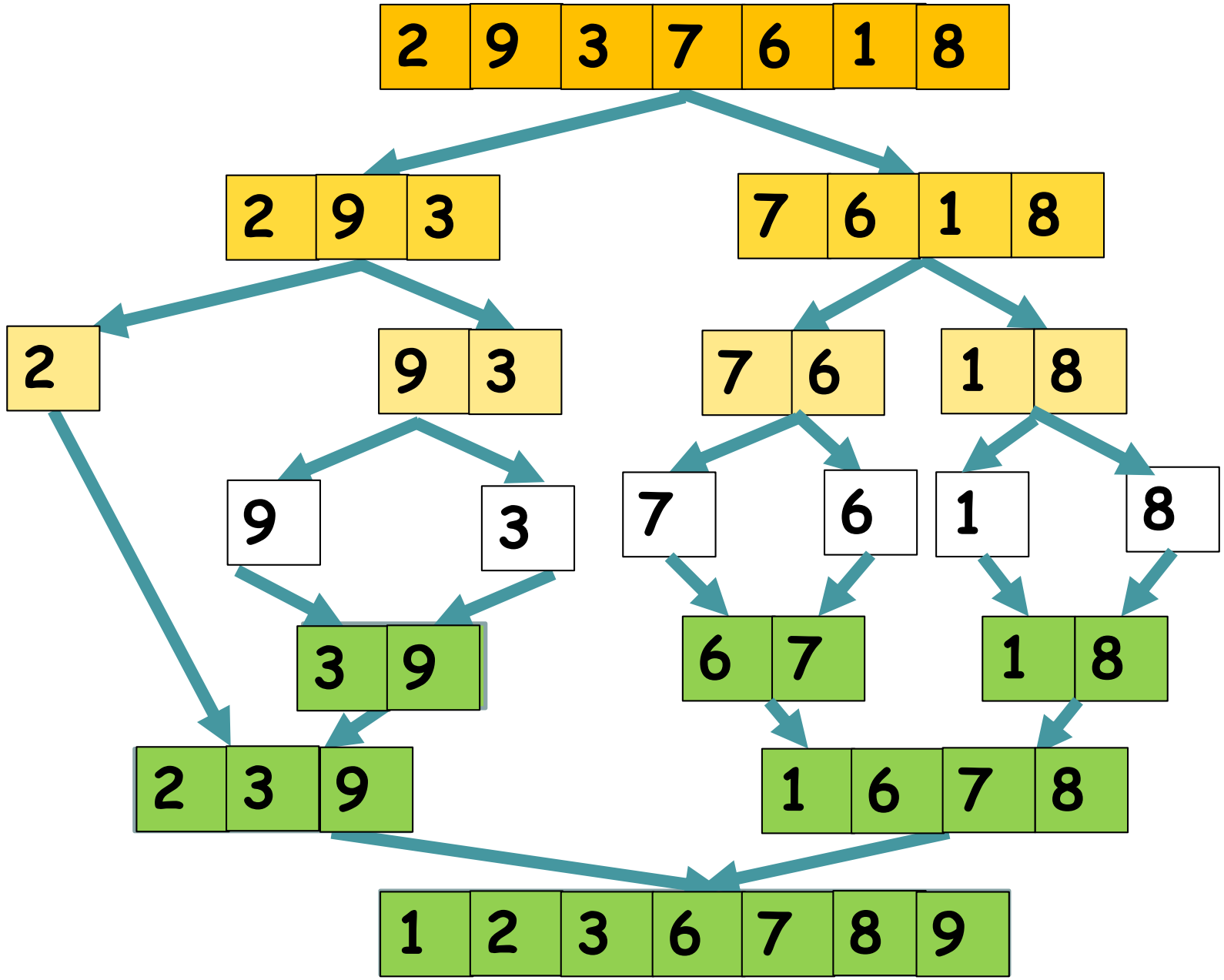
triFusion

triFusion

fusion

fusion

fusion



1. ALGORITHMES RÉCURSIFS

1.1. Tri fusion

1.1.3. Caractéristiques :

- **Stabilité :**
  - Un tel algorithme est stable si on privilégie la liste de gauche à celle de droite
- **Tri en place :**

  Sous-tableaux intermédiaires.

  - Un tel algorithme ne trie pas en place

- **Complexité:**
  - Complexité de la fonction `fusion`
  - Complexité de la fonction `triFusion`

1. ALGORITHMES

1.1. Tri fusion

1.1.3. Caractéristiques :

- **Complexité:**
  - Complexité de la fonction `fusion`
  - Complexité de la fonction `triFusion`

```
Fonction fusion(listG,listD)
listFusion ← [ ]
iG ← 0 ; iD ← 0
nG← taille(listG) ; nD← taille(listD)
Tant que taille(listFusion) < nG + nD faire
    si iG < nG et iD < nD
        si listG [iG] < listD [iD]
            listFusion ⊕ listG [iG]
            iG ← iG + 1
        sinon
            listFusion ⊕ listD [iD]
            iD ← iD + 1
        fin si
    sinon
        si iG == nG
            listFusion ⊕ listD [iD . . . nD-1]
        sinon
            listFusion ⊕ listG [iG . . . nG-1]
        fin si
    fin si
fin tant que
retourner listFusion
```

1. ALGORITHMES

1.1. Tri fusion

1.1.3. Caractéristiques :

- **Complexité:**
  - Complexité de la fonction `fusion`

$$\mathcal{C}f(n) = \mathcal{C}0f + \mathcal{C}1.n$$

$$\mathcal{C}f(n) < A.n$$

  - Complexité de la fonction `triFusion`

```
Fonction fusion(listG,listD)
listFusion ← [ ]
iG ← 0 ; iD ← 0
nG← taille(listG) ; nD← taille(listD)
Tant que taille(listFusion) < nG + nD faire
    si iG < nG et iD < nD
        si listG [iG] < listD [iD]
            listFusion ⊕ listG [iG]
            iG ← iG + 1
        sinon
            listFusion ⊕ listD [iD]
            iD ← iD + 1
        fin si
    sinon
        si iG == nG
            listFusion ⊕ listD [iD . . . nD-1]
        sinon
            listFusion ⊕ listG [iG . . . nG-1]
        fin si
    fin si
fin tant que
retourner listFusion
```

1. ALGORITHMES RÉCURSIFS

1.1. Tri fusion

1.1.3. Caractéristiques :

- **Complexité:**
  - Complexité de la fonction `fusion`
    $\mathcal{C}f(n) < A.n$
  - Complexité de la fonction `triFusion`

$$\mathcal{C}t(n) = \mathcal{C}0t + 2\mathcal{C}t(n/2) + \mathcal{C}f(n)$$
$$= \mathcal{C}0t + 2\mathcal{C}t(n/2) + A.n$$

```
Fonction triFusion(listaTrier)
    n← taille(listaTrier)
    si n ≤ 1
        retourner listaTrier
    Gauche ← triFusion(listaTrier [0 . . . n//2 [)
    Droite ← triFusion(listaTrier [n//2 . . . n [)
    listeTriee ← fusion(Gauche, Droite)
    retourner listeTriee
```

- Appel 0 $\quad \mathcal{C}t(n) = \mathcal{C}0t + 2\mathcal{C}t(n/2) + A.n \quad$ X 2^0 +
- Appel 1 $\quad \mathcal{C}t(n/2^1) = \mathcal{C}0t + 2\mathcal{C}t(n/2^2) + A.(n/2^1) \quad$ X 2^1 +
- Appel k-1 $\quad \mathcal{C}t(n/2^{k-1}) = \mathcal{C}0t + 2\mathcal{C}t(n/2^k) + A.(n/2^{k-1}) \quad$ X 2^{k-1} +
- Appel k Avec $n = 2^k$ $\quad \mathcal{C}t(n/2^k) = \mathcal{C}0t \quad$ X 2^k +

Opération nécessaire: $2^0.(L0) + 2^1.(L1) + 2^2.(L2) + \ldots + 2^{k-1}.(Lk-1) + 2^k.(Lk)$

1. ALGORITHMES RÉCURSIFS

1.1. Tri fusion

1.1.3. Caractéristiques :

- **Complexité:**
  - Complexité de la fonction `fusion`
    $\mathcal{C}f(n) < A.n$
  - Complexité de la fonction `triFusion`

$$\mathcal{C}t(n) = \mathcal{C}0t + 2\mathcal{C}t(n/2) + \mathcal{C}f(n)$$
$$= \mathcal{C}0t + 2\mathcal{C}t(n/2) + A.n$$

```
Fonction triFusion(listaTrier)
    n← taille(listaTrier)
    si n ≤ 1
        retourner listaTrier
    Gauche ← triFusion(listaTrier [0 . . . n//2 [)
    Droite ← triFusion(listaTrier [n//2 . . . n [)
    listeTriee ← fusion(Gauche, Droite)
    retourner listeTriee
```

- Appel 0 $\quad \mathcal{C}t(n) = \mathcal{C}0t + 2\mathcal{C}t(n/2) + A.n \quad$ X 2^0 +
- Appel 1 $\quad \mathcal{C}t(n/2^1) = \mathcal{C}0t + 2\mathcal{C}t(n/2^2) + A.(n/2^1) \quad$ X 2^1 +
- Appel k-1 $\quad \mathcal{C}t(n/2^{k-1}) = \mathcal{C}0t + 2\mathcal{C}t(n/2^k) + A.(n/2^{k-1}) \quad$ X 2^{k-1} +
- Appel k Avec $n = 2^k$ $\quad \mathcal{C}t(n/2^k) = \mathcal{C}0t \quad$ X 2^k +

$$Ct(n) = C0t(2^0 + 2^1 + \cdots + 2^k) + A.n.k$$

1. ALGORITHMES RÉCURSIFS

1.1. Tri fusion

1.1.3. Caractéristiques :

- **Complexité:**
  - Complexité de la fonction `fusion`
    $\mathcal{C}f(n) < A.n$
  - Complexité de la fonction `triFusion`

$$\mathcal{C}t(n) = \mathcal{C}0t + 2\mathcal{C}t(n/2) + \mathcal{C}f(n)$$
$$= \mathcal{C}0t + 2\mathcal{C}t(n/2) + A.n$$

```
Fonction triFusion(listaTrier)
    n← taille(listaTrier)
    si n ≤ 1
        retourner listaTrier
    Gauche ← triFusion(listaTrier [0 . . . n//2 [)
    Droite ← triFusion(listaTrier [n//2 . . . n [)
    listeTriee ← fusion(Gauche, Droite)
    retourner listeTriee
```

Avec $n = 2^k$

$$Ct(n) = C0t\left(2^0 + 2^1 + \cdots + 2^k\right) + A.n.k$$

$$= C0t\left(\frac{1-2^{k+1}}{1-2}\right) + A.n\log_2(n)$$

$$= (2^{k+1}-1)C0t + A.n\log_2(n)$$

$$= (2.2^k - 1).C0t + A.nlog_2(n)$$

$$= (2n-1).C0t + A.nlog_2(n)$$

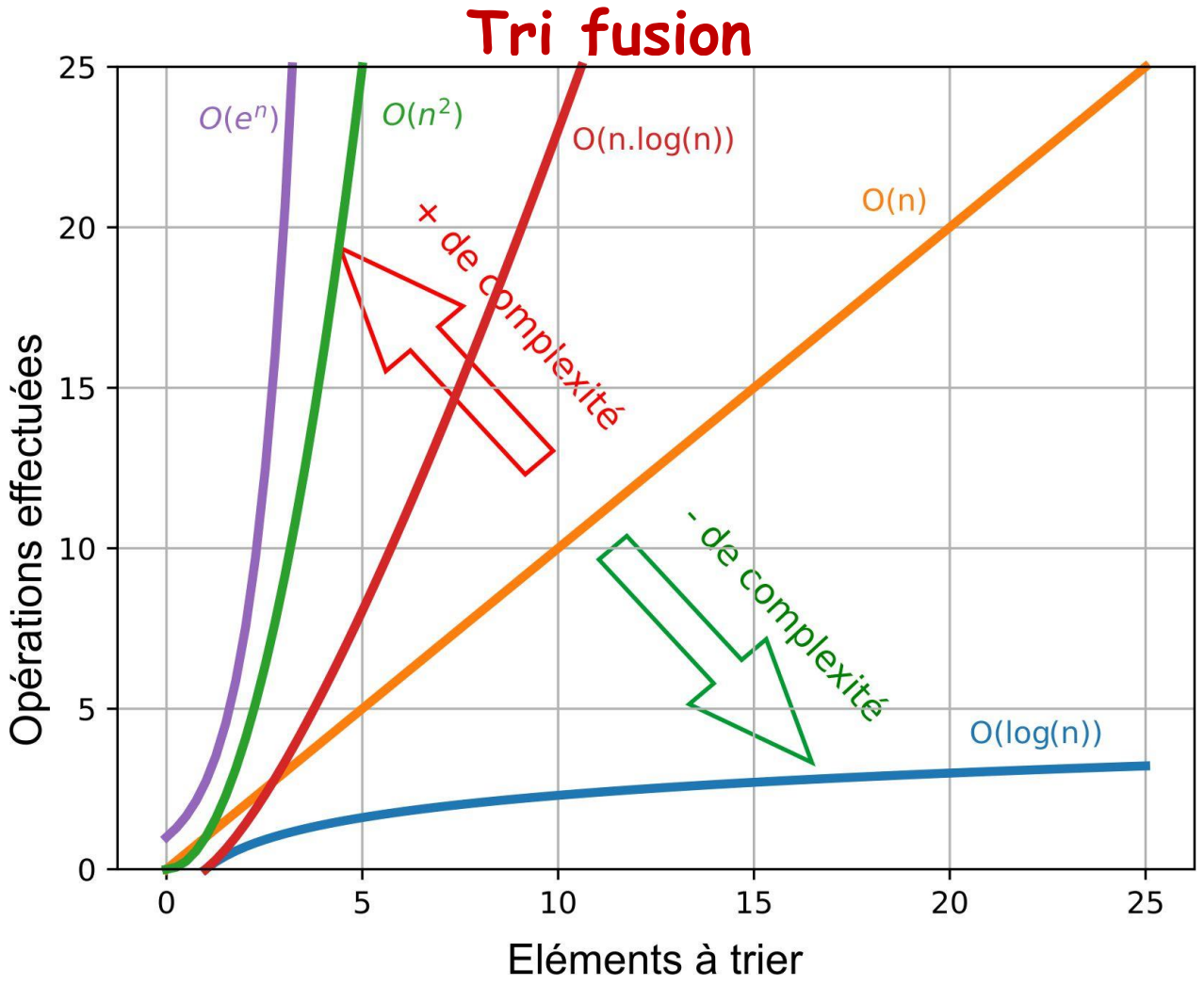
$$Sk = 2^0 + 2^1 + \cdots + 2^k$$

Suite géométrique de raison 2

D'où

$$Sk = \frac{1-2^{k+1}}{1-2} = -(1-2.2^k) = -(1-2n)$$

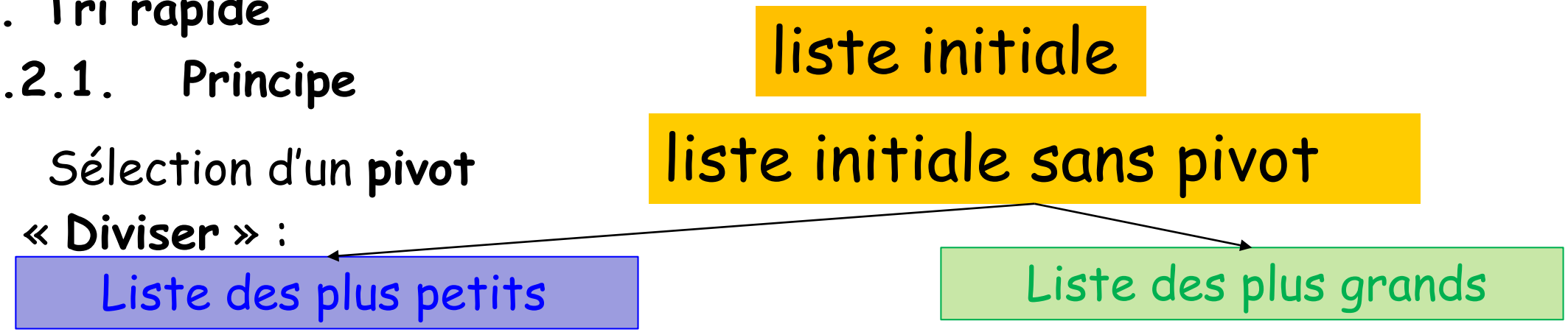
➢ **Complexité quasi-linéaire en $\mathcal{O}(n.log_2(n))$**


Tri fusion
$O(e^n)$
$O(n^2)$
O(n.log(n))
O(n)
O(log(n))
+ de complexité
- de complexité
Opérations effectuées
Eléments à trier
0
5
10
15
20
25

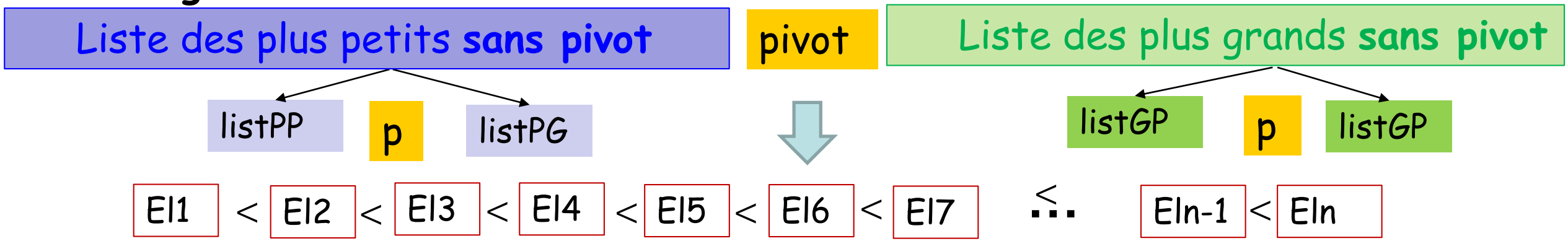
1.2. Tri rapide

1.2.1. Principe

- Sélection d'un **pivot**
- « **Diviser** » :



- « **Régner** » :



- « **Combiner** » : concaténation des sous-listes

ListeTriee =

triRapide(L. des plus petits ss pivot) + [pivot] + triRapide(L. des plus grands ss pivot)

1.2. Tri rapide

1.2.2. Algorithme :

```
Fonction triRapide(listaTrier)
si listaTrier = [ ]
    retourner [ ]
pivot← listaTrier[0]; listG ← [ ] ; listD ← [ ]
pour i de 1 à [taille(listaTrier)-1] faire
    si listaTrier[i] < pivot
          listG  ⊕ listaTrier[i]
    sinon
          listD  ⊕ listaTrier[i]
    fin si
retourner triRapide(listG) ⊕[pivot] ⊕ triRapide(listD)
```

Condition d'arrêt

1.2. Tri rapide-pivot extrémum

liste initiale

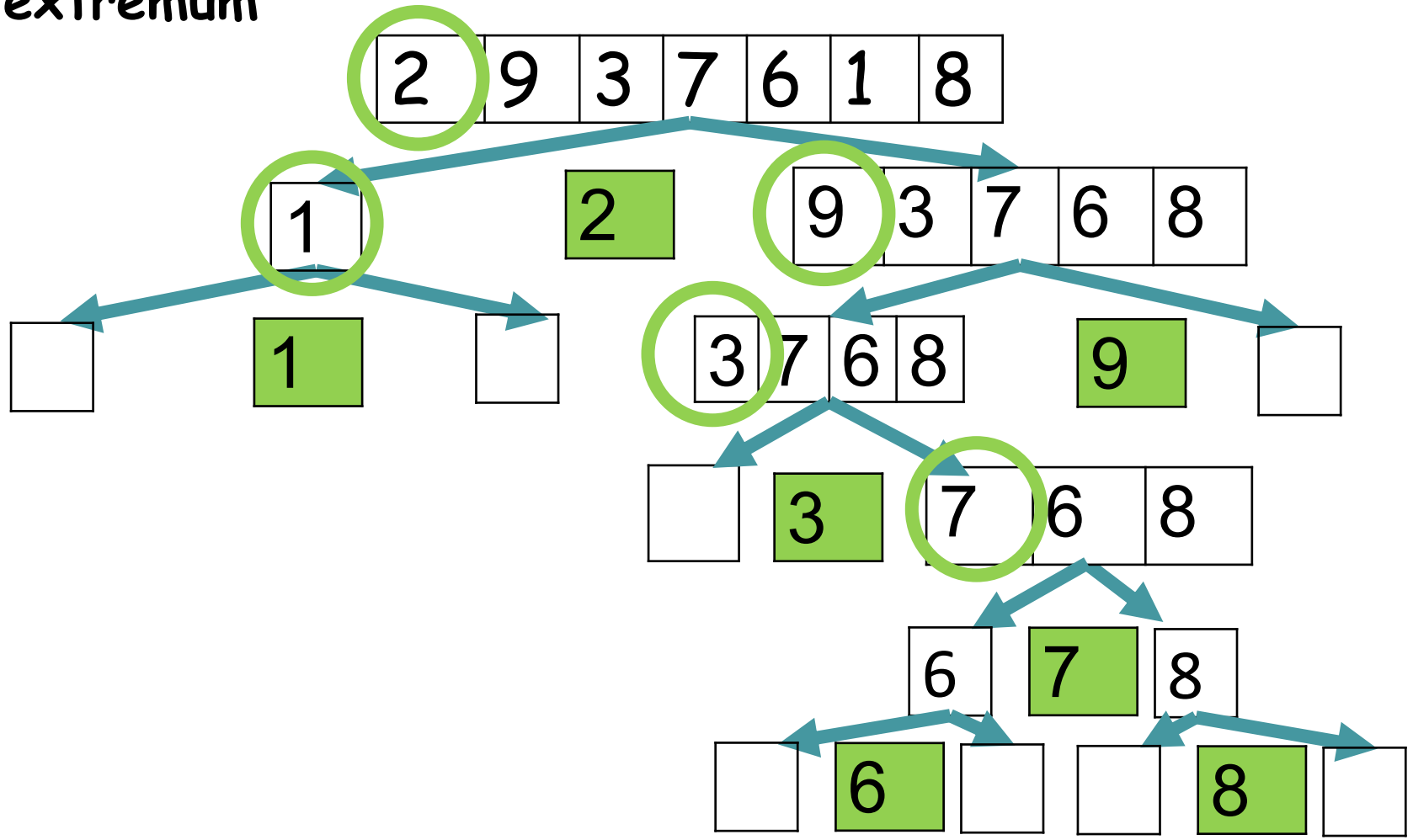
triRapide

triRapide

triRapide

triRapide

triRapide



combinaison

1 2 3 6 7 8 9

1.2. Tri rapide

1.2.3. Caractéristiques :

- **Stabilité :**

  ➢ **Un tel algorithme est non stable (pivot autre qu'extremum)**

- **Tri en place :**

  Sous-tableaux intermédiaires.

  ➢ **Un tel algorithme ne trie pas en place**

1.2. Tri rapide

1.2.3. Caractéristiques :

- **Complexité:**
  - p étant le rang du pivot (indice dans la future liste triée)

$$\mathcal{C}(n) = \mathcal{C}(p) + \mathcal{C}(n-p-1) + (n-1)\mathcal{C}i + \mathcal{C}_0$$

  - Dans le pire des cas: p=0

$$\boldsymbol{\mathcal{C}(n) = \mathcal{C}(0) + \mathcal{C}(n-1) + (n-1)\mathcal{C}i + \mathcal{C}_0}$$

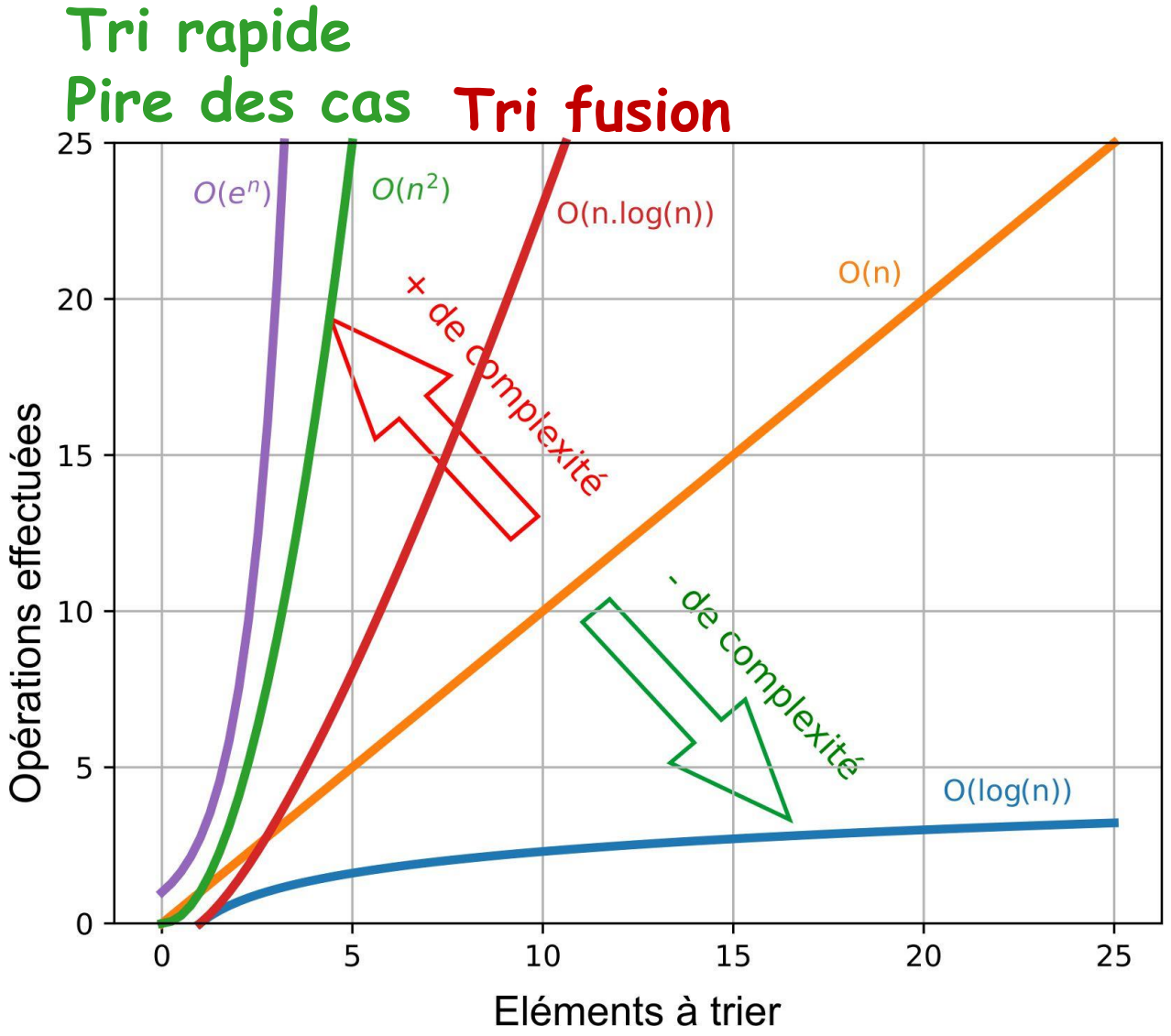
```
Fonction triRapide(listaTrier)
si listaTrier = [ ]
   retourner [ ]
pivot← listaTrier[0]; listG ← [ ] ; listD ← [ ]
pour i de 1 à taille(listaTrier) faire
   si listaTrier[i] < pivot
         listG ⊕ listaTrier[i]
   sinon
         listD ⊕ listaTrier[i]
   fin si
retourner triRapide(listG) ⊕[pivot] ⊕ triRapide(listD)
```

- Appel 0 $\quad \mathcal{C}(n) = \mathcal{C}(0) + \mathcal{C}(n-1) + (n-1)\mathcal{C}i + \mathcal{C}_0$
- Appel 1 $\quad \mathcal{C}(n-1) = \mathcal{C}(0) + \mathcal{C}(n-2) + (n-2)\mathcal{C}i + \mathcal{C}_0$
- Appel n-1 $\quad \mathcal{C}(1) = \mathcal{C}(0) + \mathcal{C}(0) + 0.\mathcal{C}i + \mathcal{C}_0$

Opération nécessaire:
(L0) + (L1) + (L2)+…+(Ln-1)

$$\mathcal{C}(n) = (n+1).\mathcal{C}(0) + \frac{n(n-1)}{2}\mathcal{C}i + n.\mathcal{C}_0$$

➢ **Complexité quadratique en $\mathcal{O}(n^2)$**


Tri rapide
Pire des cas
Tri fusion
$O(e^n)$
$O(n^2)$
O(n.log(n))
O(n)
O(log(n))
+ de complexité
- de complexité
Opérations effectuées
Eléments à trier
0
5
10
15
20
25

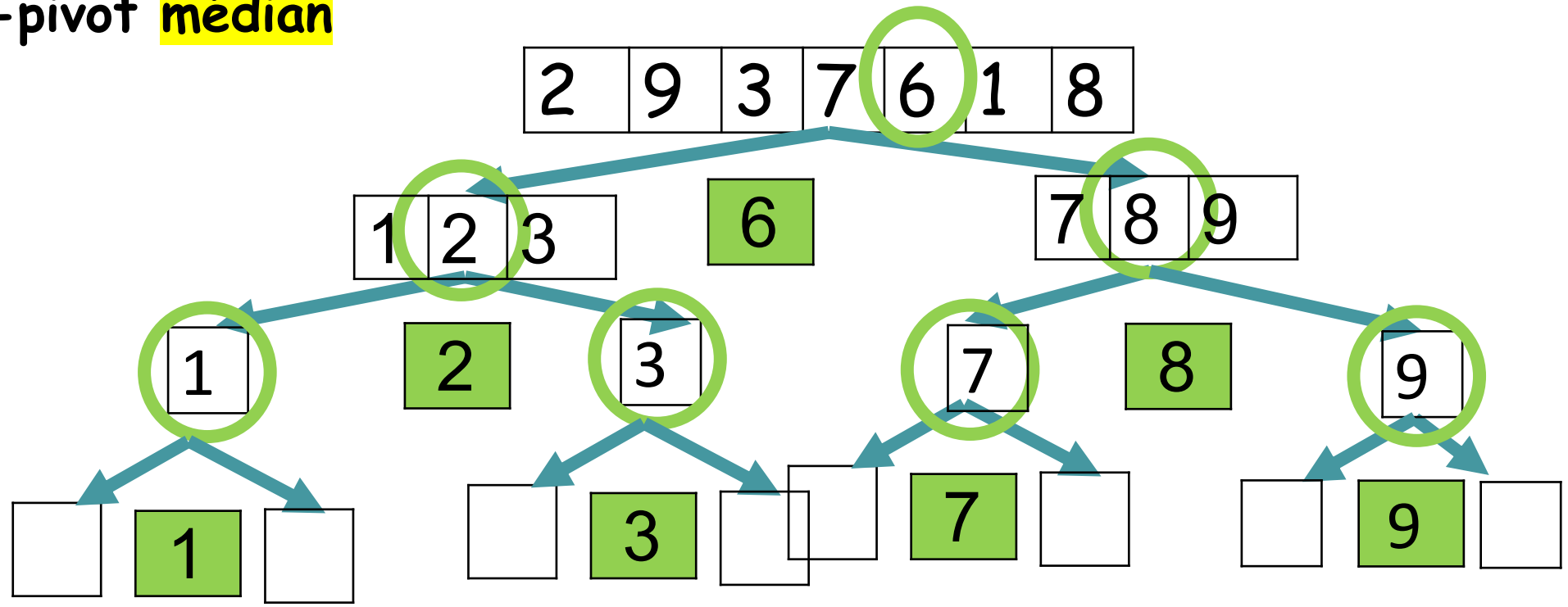
1.2. Tri rapide-pivot médian

liste initiale

triRapide

triRapide

triRapide



combinaison

1 2 3 6 7 8 9

1. ALGORITHMES RÉCURSIFS

1.2. Tri rapide

1.2.3. Caractéristiques :

- **Complexité:**
  - p étant le rang du pivot

$$\mathcal{C}(n) = \mathcal{C}(p) + \mathcal{C}(n-p-1) + (n-1)\mathcal{C}i + \mathcal{C}_0$$

  - Pour p=n/2

$$\boldsymbol{\mathcal{C}(n) = 2.\mathcal{C}(n/2) + (n-1)\mathcal{C}i + \mathcal{C}_0}$$

```
Fonction triRapide(listaTrier)
si listaTrier = [ ]
   retourner [ ]
pivot← listaTrier[0]; listG ← [ ] ; listD ← [ ]
pour i de 1 à taille(listaTrier) faire
   si listaTrier[i] < pivot
         listG ⊕ listaTrier[i]
   sinon
         listD ⊕ listaTrier[i]
   fin si
retourner triRapide(listG) ⊕[pivot] ⊕ triRapide(listD)
```

- Appel 0 $\mathcal{C}(n) = 2.\mathcal{C}(n/2^1) + (n-1)\mathcal{C}i + \mathcal{C}_0$
- Appel 1 $\mathcal{C}(n/2^1) = 2.\mathcal{C}(n/2^2) + (n/2^1 - 1)\mathcal{C}i + \mathcal{C}_0$
- Appel k-1 $\mathcal{C}(2) = 2.\mathcal{C}(1) + (2-1).\mathcal{C}i + \mathcal{C}_0$

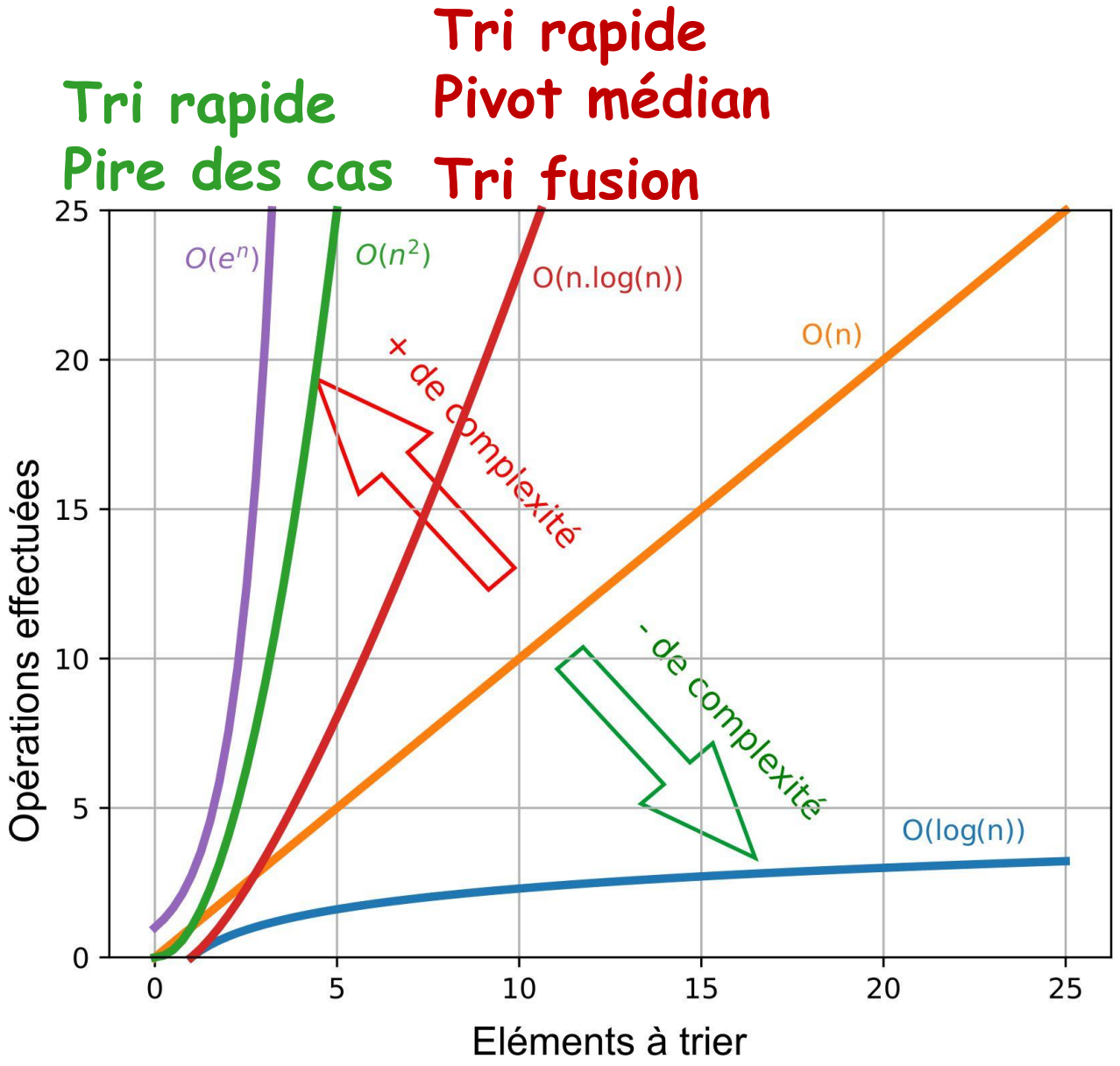
$Sk = 2^0 + 2^1 + \cdots + 2^{k-1}$
Suite géométrique de raison 2
D'où
$Sk = \frac{1-2^k}{1-2} = -(1-n)$

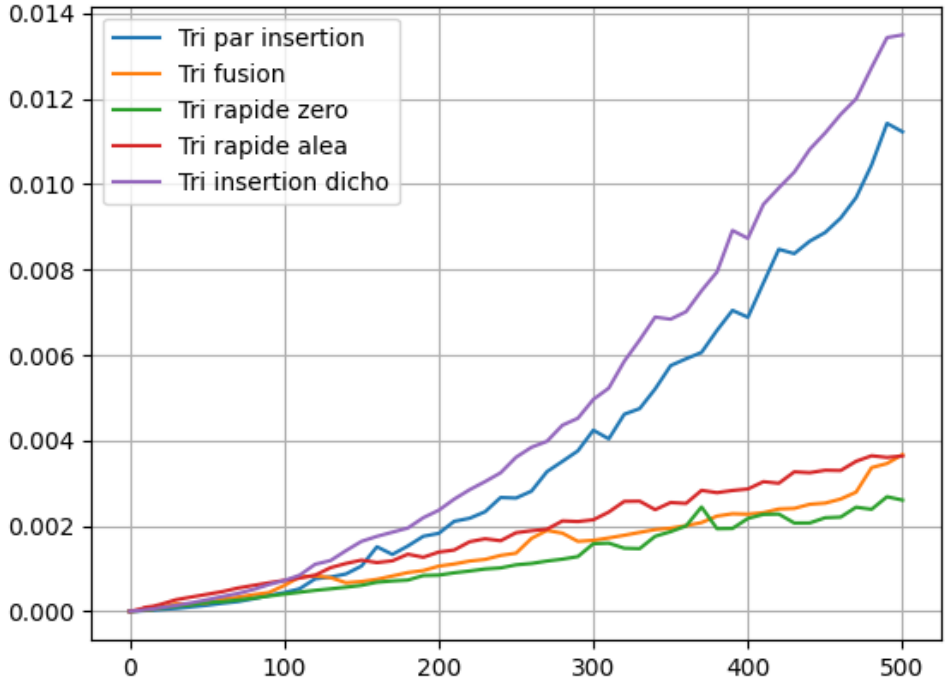
(L0) + 2.(L1) + 2^2.(L2)+…+2^{k-1}.(Lk-1)

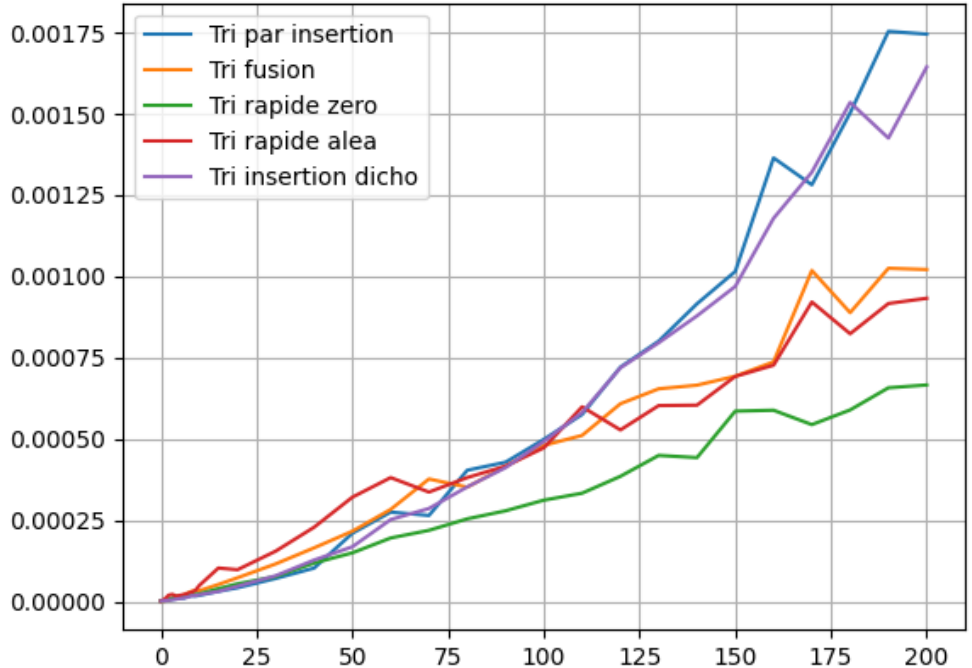
$$\mathcal{C}(n) = n.\mathcal{C}(1) + k.n.\mathcal{C}i + (1-n)\mathcal{C}i + (n-1)\mathcal{C}_0$$

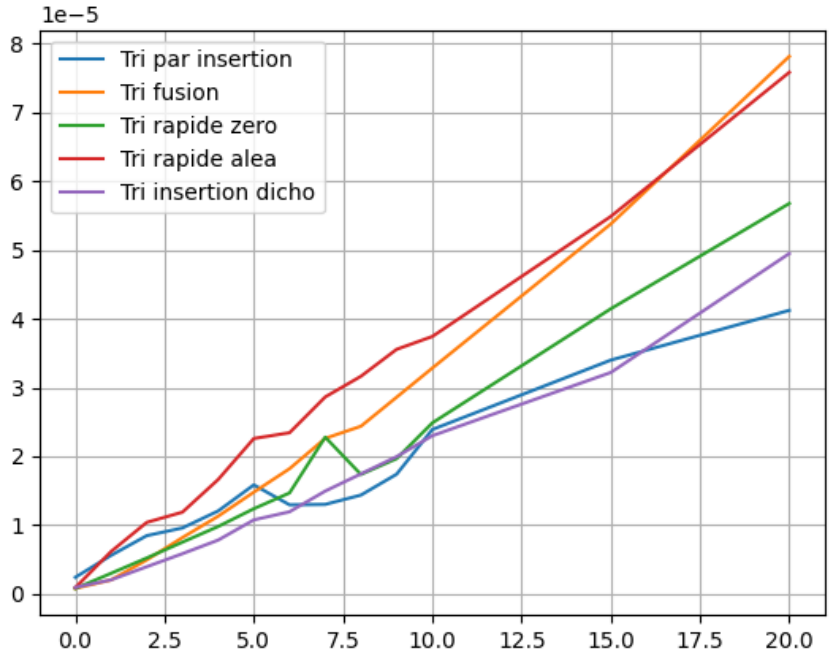
$$\mathcal{C}(n) = n.\mathcal{C}(1) + \mathcal{C}i.n.\log_2(n) + (1-n)\mathcal{C}i + (n-1)\mathcal{C}_0$$

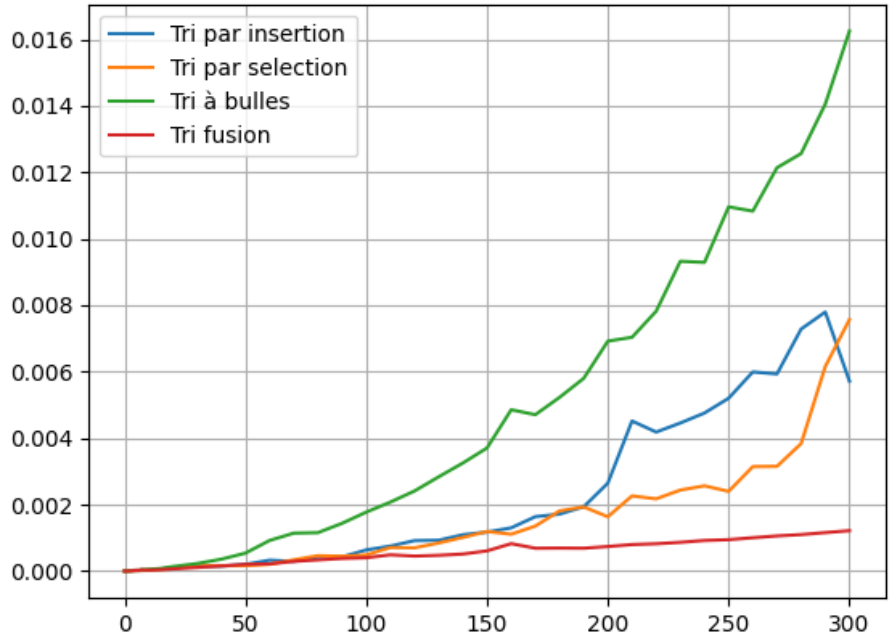
➢ **Complexité pour pivot médian « quasi-linéaire » en $\mathcal{O}(n\log_2(n))$**


Tri rapide
Pivot médian
Tri rapide
Pire des cas
Tri fusion
$O(e^n)$
$O(n^2)$
O(n.log(n))
O(n)
+ de complexité
- de complexité
O(log(n))
Opérations effectuées
Eléments à trier
0
5
10
15
20
25


Tri par insertion
Tri fusion
Tri rapide zero
Tri rapide alea
Tri insertion dicho
0.014
0.012
0.010
0.008
0.006
0.004
0.002
0.000
0
100
200
300
400
500


Tri par insertion
Tri fusion
Tri rapide zero
Tri rapide alea
Tri insertion dicho
0.00175
0.00150
0.00125
0.00100
0.00075
0.00050
0.00025
0.00000
0
25
50
75
100
125
150
175
200


1e−5
Tri par insertion
Tri fusion
Tri rapide zero
Tri rapide alea
Tri insertion dicho
8
7
6
5
4
3
2
1
0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0


Tri par insertion
Tri par selection
Tri à bulles
Tri fusion
0.016
0.014
0.012
0.010
0.008
0.006
0.004
0.002
0.000
0
50
100
150
200
250
300

MPII
ITC
Séquence 8

Tris
Partie 2 – Tris rapides - Optimisation
II – Tri par comptage
(liste d'entiers positifs à trier)

2. TRI PAR COMPTAGE

2.1. Principe

① Création d'un histogramme des valeurs de la liste

```
list = generer_liste(8, 0, 5)
```
[4, 5, 1, 5, 0, 4, 4, 1]

```
histogramme(list)
```
[1, 2, 0, 0, 3, 2]

valeur : 0 , 1 , 2 , 3 , 4 , 5

L'astuce consiste à remplacer le rôle de l'élément `i` de `list` qui devient un indice pour `histogramme(list)`

② Création de la liste triée à l'aide de l'histogramme

[0, 1, 1, 4, 4, 4, 5, 5]

place : 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7

2. TRI PAR COMPTAGE

2.1. Principe

① Création d'un histogramme des valeurs de la liste

`list = generer_liste(8, 0, 5)` [4, 5, 1, 5, 0, 4, 4, 1]

`histogramme(list)` [1, 2, 0, 0, 3, 2]

valeur : 0 , 1 , 2 , 3 , 4 , 5

2.2. Algorithme

Algorithme pour la construction d'un histogramme (**sous forme de liste**)

```
Fonction histogramme(listaTrier)
histo ← [0] * (max(listaTrier)+1)  # initialisation
pour element dans listaTrier faire
    histo[element] ← +1
retourner histo
```

2. TRI PAR COMPTAGE

2.1. Principe

`list` [4, 5, 1, 5, 0, 4, 4, 1]

`histogramme(list)` [1, 2, 0, 0, 3, 2]

valeur : 0 , 1 , 2 , 3 , 4 , 5

② Création de la liste triée à l'aide de l'histogramme

[0, 1, 1, 4, 4, 4, 5, 5]

place : 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7

2.2. Algorithme

Algorithme pour la construction d'une liste triée d'après un histogramme donné sous forme de liste.

```
Fonction triComptage(listaTrier)
hist ← histogramme(listaTrier)
listeTriee ← [0] * taille(listaTrier)
place ← 0  # indice de l'élément dans la liste triée
```

2. TRI PAR COMPTAGE

2.1. Principe

`list` [4, 5, 1, 5, 0, 4, 4, 1]

`histogramme(list)` [1, 2, 0, 0, 3, 2]

valeur : 0 , 1 , 2 , 3 , 4 , 5

② Création de la liste triée à l'aide de l'histogramme

[0, 1, 1, 4, 4, 4, 5, 5]

place : 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7

2.2. Algorithme

Algorithme pour la construction d'une liste triée d'après un histogramme donné sous forme de liste.

```
Fonction triComptage(listaTrier)
hist ← histogramme(listaTrier)
listeTriee ← [0] * taille(listaTrier)
place ← 0  # indice de l'élément dans la liste triée
pour val de 0 à max(listaTrier) faire  # valeur
```

2. TRI PAR COMPTAGE

2.1. Principe

`list` [4, 5, 1, 5, 0, 4, 4, 1]

`histogramme(list)` [1, 2, 0, 0, 3, 2]

valeur : 0 , 1 , 2 , 3 , 4 , 5

② Création de la liste triée à l'aide de l'histogramme

[0, 1, 1, 4, 4, 4, 5, 5]

place : 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7

2.2. Algorithme

Algorithme pour la construction d'une liste triée d'après un histogramme donné sous forme de liste.

```
Fonction triComptage(listaTrier)
hist ← histogramme(listaTrier)
listeTriee ← [0] * taille(listaTrier)
place ← 0  # indice de l'élément dans la liste triée
pour val de 0 à max(listaTrier) faire  # valeur
    n_val ← 0  # nb d'élément de cette valeur
```

2. TRI PAR COMPTAGE

2.1. Principe

`list` [4, 5, 1, 5, 0, 4, 4, 1]

`histogramme(list)` [1, 2, 0, 0, 3, 2]

valeur : 0 , 1 , 2 , 3 , 4 , 5

② Création de la liste triée à l'aide de l'histogramme

[0, 1, 1, 4, 4, 4, 5, 5]

place : 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7

2.2. Algorithme

Algorithme pour la construction d'une liste triée d'après un histogramme donné sous forme de liste.

```
Fonction triComptage(listaTrier)
hist ← histogramme(listaTrier)
listeTriee ← [0] * taille(listaTrier)
place ← 0  # indice de l'élément dans la liste triée
pour val de 0 à max(listaTrier) faire  # valeur
    n_val ← 0  # nb d'élément de cette valeur
    tant que  n_val < hist[val] faire
```

2. TRI PAR COMPTAGE

2.1. Principe

`list` [4, 5, 1, 5, 0, 4, 4, 1]

`histogramme(list)` [1, 2, 0, 0, 3, 2]

valeur : 0 , 1 , 2 , 3 , 4 , 5

② Création de la liste triée à l'aide de l'histogramme

[0, 1, 1, 4, 4, 4, 5, 5]

place : 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7

2.2. Algorithme

Algorithme pour la construction d'une liste triée d'après un histogramme donné sous forme de liste.

```
Fonction triComptage(listaTrier)
hist ← histogramme(listaTrier)
listeTriee ← [0] * taille(listaTrier)
place ← 0  # indice de l'élément dans la liste triée
pour val de 0 à max(listaTrier) faire  # valeur
    n_val ← 0  # nb d'élément de cette valeur
    tant que  n_val < hist[val] faire
        listeTriee[place] = val
```

2. TRI PAR COMPTAGE

2.1. Principe

`list` [4, 5, 1, 5, 0, 4, 4, 1]

`histogramme(list)` [1, 2, 0, 0, 3, 2]

valeur : 0 , 1 , 2 , 3 , 4 , 5

② Création de la liste triée à l'aide de l'histogramme

[0, 1, 1, 4, 4, 4, 5, 5]

place : 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7

2.2. Algorithme

Algorithme pour la construction d'une liste triée d'après un histogramme donné sous forme de liste.

```
Fonction triComptage(listaTrier)
hist ← histogramme(listaTrier)
listeTriee ← [0] * taille(listaTrier)
place ← 0  # indice de l'élément dans la liste triée
pour val de 0 à max(listaTrier) faire  # valeur
    n_val ← 0  # nb d'élément de cette valeur
    tant que  n_val < hist[val] faire
        listeTriee[place] = val
        n_val ← +1  # comptage nb élément
```

2. TRI PAR COMPTAGE

2.1. Principe

`list` [4, 5, 1, 5, 0, 4, 4, 1]

`histogramme(list)` [1, 2, 0, 0, 3, 2]

valeur : 0 , 1 , 2 , 3 , 4 , 5

② Création de la liste triée à l'aide de l'histogramme

[0, 1, 1, 4, 4, 4, 5, 5]

place : 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7

2.2. Algorithme

Algorithme pour la construction d'une liste triée d'après un histogramme donné sous forme de liste.

```
Fonction triComptage(listaTrier)
hist ← histogramme(listaTrier)
listeTriee ← [0] * taille(listaTrier)
place ← 0  # indice de l'élément dans la liste triée
pour val de 0 à max(listaTrier) faire  # valeur
    n_val ← 0  # nb d'élément de cette valeur
    tant que  n_val < hist[val] faire
        listeTriee[place] = val
        n_val ← +1  # comptage nb élément
        place ← +1  # décalage pour élément suivant
```

2. TRI PAR COMPTAGE

2.1. Principe

`list` [4, 5, 1, 5, 0, 4, 4, 1]

`histogramme(list)` [1, 2, 0, 0, 3, 2]

valeur : 0 , 1 , 2 , 3 , 4 , 5

② Création de la liste triée à l'aide de l'histogramme

[0, 1, 1, 4, 4, 4, 5, 5]

place : 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7

2.2. Algorithme

Algorithme pour la construction d'une liste triée d'après un histogramme donné sous forme de liste.

```
Fonction triComptage(listaTrier)
hist ← histogramme(listaTrier)
listeTriee ← [0] * taille(listaTrier)
place ← 0  # indice de l'élément dans la liste triée
pour val de 0 à max(listaTrier) faire  # valeur
    n_val ← 0  # nb d'élément de cette valeur
    tant que  n_val < hist[val] faire
        listeTriee[place] = val
        n_val ← +1  # comptage nb élément
        place ← +1  # décalage pour élément suivant
```

2. TRI PAR COMPTAGE

2.1. Principe

`list` [4, 5, 1, 5, 0, 4, 4, 1]

`histogramme(list)` [1, 2, 0, 0, 3, 2]

valeur : 0 , 1 , 2 , 3 , 4 , 5

② Création de la liste triée à l'aide de l'histogramme

[0, 1, 1, 4, 4, 4, 5, 5]

place : 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7

2.2. Algorithme

Algorithme pour la construction d'une liste triée d'après un histogramme donné sous forme de liste.

```
Fonction triComptage(listaTrier)
hist ← histogramme(listaTrier)
listeTriee ← [0] * taille(listaTrier)
place ← 0  # indice de l'élément dans la liste triée
pour val de 0 à max(listaTrier) faire  # valeur
    n_val ← 0  # nb d'élément de cette valeur
    tant que  n_val < hist[val] faire
        listeTriee[place] = val
        n_val ← +1  # comptage nb élément
        place ← +1  # décalage pour élément  suivant
retourner listeTriee
```

2. TRI PAR COMPTAGE

2.2. Algorithme

Algorithme pour la construction d'un histogramme (**sous forme de liste**)

```
Fonction histogramme(listaTrier, max(listaTrier))
histo ← [0] * (max(listaTrier)+1)
pour element dans listaTrier faire
    histo[element] ← +1
retourner histo
```

Algorithme pour la construction d'une liste triée d'après un histogramme donné sous forme de liste.

```
Fonction triComptage(listaTrier)
hist ← histogramme(listaTrier)
listeTriee ← [0] * taille(listaTrier)
place ← 0  # indice de l'élément dans la liste triée
pour val de 0 à max(listaTrier) faire  # valeur
    n_val ← 0  # nb d'élément de cette valeur
    tant que  n_val < hist[val] faire
        listeTriee[place] = val
        n_val ← +1  # comptage nb élément
        place ← +1  # décalage pour élément  suivant
retourner listeTriee
```

2. TRI PAR COMPTAGE

2.2. Algorithme: illustration

liste initiale

4	5	1	5	0	4	4	1

`histogramme(liste initiale )` [1, 2, 0, 0, 3, 2]

valeur : 0, 1 , 2 , 3 , 4 , 5

`triComptage(histogramme(liste initiale ))`

0	0	0	0	0	0	0	0

place : 0, 1 , 2 , 3 , 4 , 5 , 6 , 7

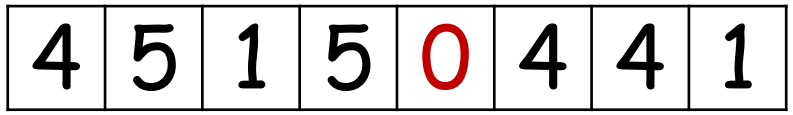
valeur = 0, n_val ← 0 , n_val < hist[0] ? OUI

2. TRI PAR COMPTAGE

2.2. Algorithme: illustration

liste initiale

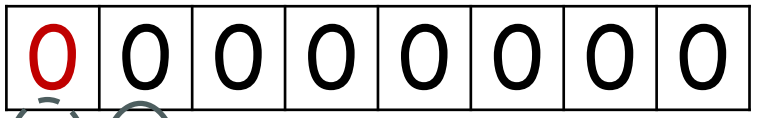
[4, 5, 1, 5, 0, 4, 4, 1]



`histogramme(liste initiale )` [1, 2, 0, 0, 3, 2]

valeur : 0 , 1 , 2 , 3 , 4 , 5

`triComptage(histogramme(liste initiale ))`



place : 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7

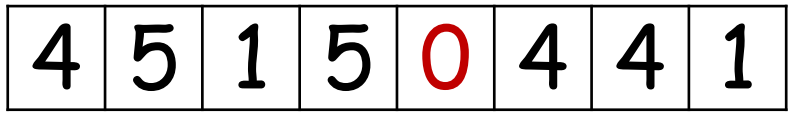
valeur = 0, n_val ← 0 , n_val < hist[0] ? OUI n_val ← 1 place ← 1 n_val < hist[0] ? NON

2. TRI PAR COMPTAGE

2.2. Algorithme: illustration

liste initiale

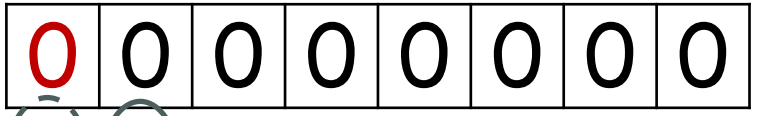
[4, 5, 1, 5, 0, 4, 4, 1]



`histogramme(liste initiale )` [1, 2, 0, 0, 3, 2]

valeur : 0, 1, 2 , 3 , 4 , 5

`triComptage(histogramme(liste initiale ))`



place : 0, 1, 2 , 3 , 4 , 5 , 6 , 7

valeur = 0, n_val ← 0 , n_val < hist[0] ? OUI n_val ← 1 place ← 1 n_val < hist[0] ? NON

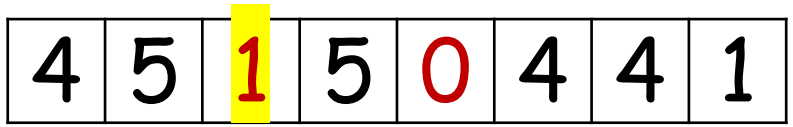
valeur = 1, n_val ← 0 , n_val < hist[1] ? OUI

2. TRI PAR COMPTAGE

2.2. Algorithme: illustration

liste initiale

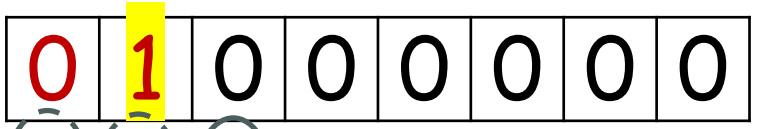
[4, 5, 1, 5, 0, 4, 4, 1]



`histogramme(liste initiale )` [1, 2, 0, 0, 3, 2]

valeur : 0, 1, 2 , 3 , 4 , 5

`triComptage(histogramme(liste initiale ))`



place : 0, 1, 2, 3 , 4 , 5 , 6 , 7

valeur = 0, n_val ← 0 , n_val < hist[0] ? OUI n_val ← 1 place ← 1 n_val < hist[0] ? NON

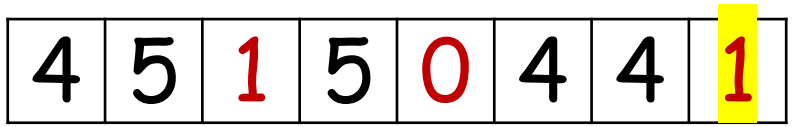
valeur = 1, n_val ← 0 , n_val < hist[1] ? OUI n_val ← 1 place ← 2 n_val < hist[1] ? OUI

2. TRI PAR COMPTAGE

2.2. Algorithme: illustration

liste initiale

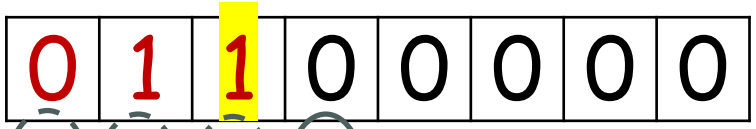
[4, 5, 1, 5, 0, 4, 4, 1]



`histogramme(liste initiale )` [1, 2, 0, 0, 3, 2]

valeur : 0, 1, 2 , 3 , 4 , 5

`triComptage(histogramme(liste initiale ))`



place : 0, 1, 2, 3, 4 , 5 , 6 , 7

valeur = 0, n_val ← 0 , n_val < hist[0] ? OUI n_val ← 1 place ← 1 n_val < hist[0] ? NON

valeur = 1, n_val ← 0 , n_val < hist[1] ? OUI n_val ← 1 place ← 2 n_val < hist[1] ? OUI

n_val ← 2 place ← 3 n_val < hist[1] ? NON

2. TRI PAR COMPTAGE

2.2. Algorithme: illustration

liste initiale

[4, 5, 1, 5, 0, 4, 4, 1]

4	5	1	5	0	4	4	1

`histogramme(liste initiale )` [1, 2, 0, 0, 3, 2]

valeur : 0 , 1 , 2 , 3 , 4 , 5

`triComptage(histogramme(liste initiale ))`

0	1	1	0	0	0	0	0

place : 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7

valeur = 0, n_val ← 0 , n_val < hist[0] ? OUI n_val ← 1 place ← 1 n_val < hist[0] ? NON

valeur = 1, n_val ← 0 , n_val < hist[1] ? OUI n_val ← 1 place ← 2 n_val < hist[1] ? OUI

n_val ← 2 place ← 3 n_val < hist[1] ? NON

valeur = 2, n_val ← 0 , n_val < hist[2] ? NON

2. TRI PAR COMPTAGE

2.2. Algorithme: illustration

liste initiale

[4, 5, 1, 5, 0, 4, 4, 1]

4	5	1	5	0	4	4	1

`histogramme(liste initiale )`

[1, 2, 0, 0, 3, 2]

valeur : 0 , 1 , 2 , 3 , 4 , 5

`triComptage(histogramme(liste initiale ))`

0	1	1	0	0	0	0	0

place : 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7

valeur = 0, n_val ← 0 , n_val < hist[0] ? OUI n_val ← 1 place ← 1 n_val < hist[0] ? NON

valeur = 1, n_val ← 0 , n_val < hist[1] ? OUI n_val ← 1 place ← 2 n_val < hist[1] ? OUI

n_val ← 2 place ← 3 n_val < hist[1] ? NON

valeur = 2, n_val ← 0 , n_val < hist[2] ? NON

valeur = 3, n_val ← 0 , n_val < hist[3] ? NON

2. TRI PAR COMPTAGE

2.2. Algorithme: illustration

liste initiale

[4, 5, 1, 5, 0, 4, 4, 1]

4	5	1	5	0	4	4	1

`histogramme(liste initiale )`

[1, 2, 0, 0, 3, 2]

valeur : 0, 1, 2, 3, 4, 5

`triComptage(histogramme(liste initiale ))`

0	1	1	0	0	0	0	0

place : 0, 1, 2, 3, 4, 5, 6, 7

valeur = 0, n_val ← 0 , n_val < hist[0] ? OUI n_val ← 1 place ← 1 n_val < hist[0] ? NON

valeur = 1, n_val ← 0 , n_val < hist[1] ? OUI n_val ← 1 place ← 2 n_val < hist[1] ? OUI

n_val ← 2 place ← 3 n_val < hist[1] ? NON

valeur = 2, n_val ← 0 , n_val < hist[2] ? NON

valeur = 3, n_val ← 0 , n_val < hist[3] ? NON

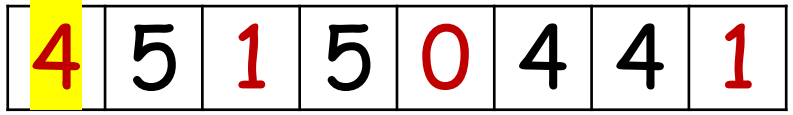
valeur = 4, n_val ← 0 , n_val < hist[4] ? OUI

2. TRI PAR COMPTAGE

2.2. Algorithme: illustration

liste initiale

[4, 5, 1, 5, 0, 4, 4, 1]



`histogramme(liste initiale )`

[1, 2, 0, 0, 3, 2]

valeur : 0, 1, 2, 3, 4, 5

`triComptage(histogramme(liste initiale ))`

place : 0, 1, 2, 3, 4, 5, 6, 7

valeur = 0, n_val ← 0 , n_val < hist[0] ? OUI n_val ← 1 place ← 1 n_val < hist[0] ? NON

valeur = 1, n_val ← 0 , n_val < hist[1] ? OUI n_val ← 1 place ← 2 n_val < hist[1] ? OUI

n_val ← 2 place ← 3 n_val < hist[1] ? NON

valeur = 2, n_val ← 0 , n_val < hist[2] ? NON

valeur = 3, n_val ← 0 , n_val < hist[3] ? NON

valeur = 4, n_val ← 0 , n_val < hist[4] ? OUI

2. TRI PAR COMPTAGE

2.2. Algorithme: illustration

liste initiale

[4, 5, 1, 5, 0, 4, 4, 1]

4	5	1	5	0	4	4	1

`histogramme(liste initiale )`

[1, 2, 0, 0, 3, 2]

valeur : 0, 1, 2, 3, 4, 5

`triComptage(histogramme(liste initiale ))`

0	1	1	4	0	0	0	0

place : 0, 1, 2, 3, 4, 5, 6, 7

valeur = 0, n_val ← 0 , n_val < hist[0] ? OUI n_val ← 1 place ← 1 n_val < hist[0] ? NON

valeur = 1, n_val ← 0 , n_val < hist[1] ? OUI n_val ← 1 place ← 2 n_val < hist[1] ? OUI

n_val ← 2 place ← 3 n_val < hist[1] ? NON

valeur = 2, n_val ← 0 , n_val < hist[2] ? NON

valeur = 3, n_val ← 0 , n_val < hist[3] ? NON

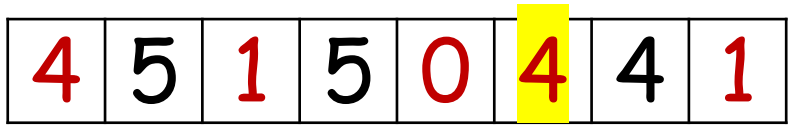
valeur = 4, n_val ← 0 , n_val < hist[4] ? OUI n_val ← 1 place ← 4 n_val < hist[4] ? OUI

2. TRI PAR COMPTAGE

2.2. Algorithme: illustration

liste initiale

[4, 5, 1, 5, 0, 4, 4, 1]



`histogramme(liste initiale )` [1, 2, 0, 0, 3, 2]

valeur : 0, 1, 2, 3, 4, 5

`triComptage(histogramme(liste initiale ))`

place : 0, 1, 2, 3, 4, 5, 6, 7

valeur = 0, n_val ← 0 , n_val < hist[0] ? OUI n_val ← 1 place ← 1 n_val < hist[0] ? NON

valeur = 1, n_val ← 0 , n_val < hist[1] ? OUI n_val ← 1 place ← 2 n_val < hist[1] ? OUI

n_val ← 2 place ← 3 n_val < hist[1] ? NON

valeur = 2, n_val ← 0 , n_val < hist[2] ? NON

valeur = 3, n_val ← 0 , n_val < hist[3] ? NON

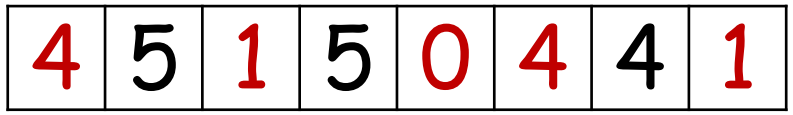
valeur = 4, n_val ← 0 , n_val < hist[4] ? OUI n_val ← 1 place ← 4 n_val < hist[4] ? OUI

2. TRI PAR COMPTAGE

2.2. Algorithme: illustration

liste initiale

[4, 5, 1, 5, 0, 4, 4, 1]



`histogramme(liste initiale )` [1, 2, 0, 0, 3, 2]

valeur : 0, 1, 2, 3, 4, 5

`triComptage(histogramme(liste initiale ))`

place : 0, 1, 2, 3, 4, 5, 6 , 7

valeur = 0, n_val ← 0 , n_val < hist[0] ? OUI n_val ← 1 place ← 1 n_val < hist[0] ? NON

[…]

valeur = 4, n_val ← 0 , n_val < hist[4] ? OUI n_val ← 1 place ← 4 n_val < hist[4] ? OUI

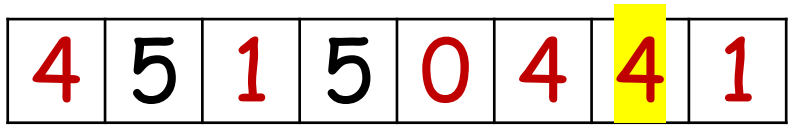
n_val ← 2 place ← 5 n_val < hist[4] ? OUI

2. TRI PAR COMPTAGE

2.2. Algorithme: illustration

liste initiale

[4, 5, 1, 5, 0, 4, 4, 1]



`histogramme(liste initiale )` [1, 2, 0, 0, 3, 2]

valeur : 0, 1, 2, 3, 4, 5

`triComptage(histogramme(liste initiale ))`

place : 0, 1, 2, 3, 4, 5, 6, 7

valeur = 0, n_val ← 0 , n_val < hist[0] ? OUI n_val ← 1 place ← 1 n_val < hist[0] ? NON

[…]

valeur = 4, n_val ← 0 , n_val < hist[4] ? OUI n_val ← 1 place ← 4 n_val < hist[4] ? OUI

n_val ← 2 place ← 5 n_val < hist[4] ? OUI

2. TRI PAR COMPTAGE

2.2. Algorithme: illustration

liste initiale

[4, 5, 1, 5, 0, 4, 4, 1]

4	5	1	5	0	4	4	1

`histogramme(liste initiale )` [1, 2, 0, 0, 3, 2]

valeur : 0, 1, 2, 3, 4, 5

`triComptage(histogramme(liste initiale ))`

0	1	1	4	4	4	0	0

place : 0, 1, 2, 3, 4, 5, 6, 7

valeur = 0, n_val ← 0 , n_val < hist[0] ? OUI n_val ← 1 place ← 1 n_val < hist[0] ? NON

[…]

valeur = 4, n_val ← 0 , n_val < hist[4] ? OUI n_val ← 1 place ← 4 n_val < hist[4] ? OUI

n_val ← 2 place ← 5 n_val < hist[4] ? OUI

n_val ← 3 place ← 6 n_val < hist[4] ? NON

2. TRI PAR COMPTAGE

2.2. Algorithme: illustration

liste initiale

[4, 5, 1, 5, 0, 4, 4, 1]

4	5	1	5	0	4	4	1

`histogramme(liste initiale )` [1, 2, 0, 0, 3, 2]

valeur : 0, 1, 2, 3, 4, 5

`triComptage(histogramme(liste initiale ))`

0	1	1	4	4	4	0	0

place : 0, 1, 2, 3, 4, 5, 6, 7

valeur = 0, n_val ← 0 , n_val < hist[0] ? OUI n_val ← 1 place ← 1 n_val < hist[0] ? NON

[…]

valeur = 4, n_val ← 0 , n_val < hist[4] ? OUI n_val ← 1 place ← 4 n_val < hist[4] ? OUI
n_val ← 2 place ← 5 n_val < hist[4] ? OUI
n_val ← 3 place ← 6 n_val < hist[4] ? NON

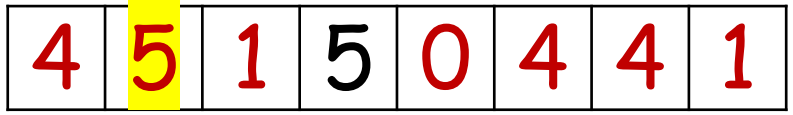
valeur = 5, n_val ← 0 , n_val < hist[5] ? OUI

2. TRI PAR COMPTAGE

2.2. Algorithme: illustration

liste initiale

[4, 5, 1, 5, 0, 4, 4, 1]



`histogramme(liste initiale )` [1, 2, 0, 0, 3, 2]

valeur : 0, 1, 2, 3, 4, 5

`triComptage(histogramme(liste initiale ))`

place : 0, 1, 2, 3, 4, 5, 6, 7

valeur = 0, n_val ← 0 , n_val < hist[0] ? OUI n_val ← 1 place ← 1 n_val < hist[0] ? NON

[…]

valeur = 4, n_val ← 0 , n_val < hist[4] ? OUI n_val ← 1 place ← 4 n_val < hist[4] ? OUI
n_val ← 2 place ← 5 n_val < hist[4] ? OUI
n_val ← 3 place ← 6 n_val < hist[4] ? NON

valeur = 5, n_val ← 0 , n_val < hist[5] ? OUI

2. TRI PAR COMPTAGE

2.2. Algorithme: illustration

liste initiale

[4, 5, 1, 5, 0, 4, 4, 1]

4	5	1	5	0	4	4	1

`histogramme(liste initiale )` [1, 2, 0, 0, 3, 2]

valeur : 0, 1, 2, 3, 4, 5

`triComptage(histogramme(liste initiale ))`

0	1	1	4	4	4	5	0

place : 0, 1, 2, 3, 4, 5, 6, 7

valeur = 0, n_val ← 0 , n_val < hist[0] ? OUI n_val ← 1 place ← 1 n_val < hist[0] ? NON

[…]

valeur = 4, n_val ← 0 , n_val < hist[4] ? OUI n_val ← 1 place ← 4 n_val < hist[4] ? OUI
n_val ← 2 place ← 5 n_val < hist[4] ? OUI
n_val ← 3 place ← 6 n_val < hist[4] ? NON

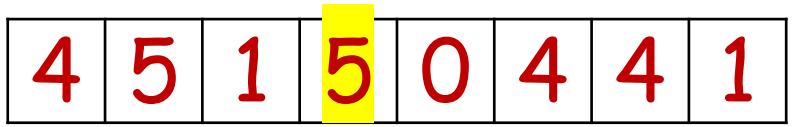
valeur = 5, n_val ← 0 , n_val < hist[5] ? OUI n_val ← 1 place ← 7 n_val < hist[5] ? OUI

2. TRI PAR COMPTAGE

2.2. Algorithme: illustration

liste initiale

[4, 5, 1, 5, 0, 4, 4, 1]



`histogramme(liste initiale )` [1, 2, 0, 0, 3, 2]

valeur : 0, 1, 2, 3, 4, 5

`triComptage(histogramme(liste initiale ))`

place : 0, 1, 2, 3, 4, 5, 6, 7

valeur = 0, n_val ← 0 , n_val < hist[0] ? OUI n_val ← 1 place ← 1 n_val < hist[0] ? NON

[…]

valeur = 4, n_val ← 0 , n_val < hist[4] ? OUI n_val ← 1 place ← 4 n_val < hist[4] ? OUI
n_val ← 2 place ← 5 n_val < hist[4] ? OUI
n_val ← 3 place ← 6 n_val < hist[4] ? NON

valeur = 5, n_val ← 0 , n_val < hist[5] ? OUI n_val ← 1 place ← 7 n_val < hist[5] ? OUI

2.3. Caractéristiques :

- **Stabilité :**
  - ➢ **Un tel algorithme est non stable**
- **Tri en place :**

  Histogramme = variable intermédiaire

  - ➢ **Un tel algorithme ne trie pas en place**

2. TRI PAR COMPTAGE

2.3. Caractéristiques :

- **Complexité:**

$$\mathcal{C}\mathrm{h}(n) = n.C_1 + C_0$$

$$\mathcal{C}(n) = \mathcal{C}\mathrm{h}(\mathrm{n}) + C_1 + \max(\mathrm{listaTrier}).\mathcal{C}i$$

Avec $\max(\mathrm{listaTrier}) = \mathrm{k}$

$$\mathcal{C}(n) = n.C_1 + C_0 + C_1 + \mathrm{k}.\mathcal{C}i$$

➢ **Complexité linéaire en $\mathcal{O}(n+k)$**

```
Fonction histogramme(listaTrier)
histo ← [0] * max(listaTrier)
pour element dans listaTrier faire
    histo[element] ← +1
retourner histo
```

```
Fonction triComptage(listaTrier)
hist ← histogramme(listaTrier)
listeTriee ← [0] * taille(listaTrier)
place ← 0
pour val de 0 à max(listaTrier) faire
    n_val ← 0
    tant que  n_val < hist[val] faire
        listeTriee[place] = val
        n_val ← +1
        place ← +1
retourner  listeTriee
```

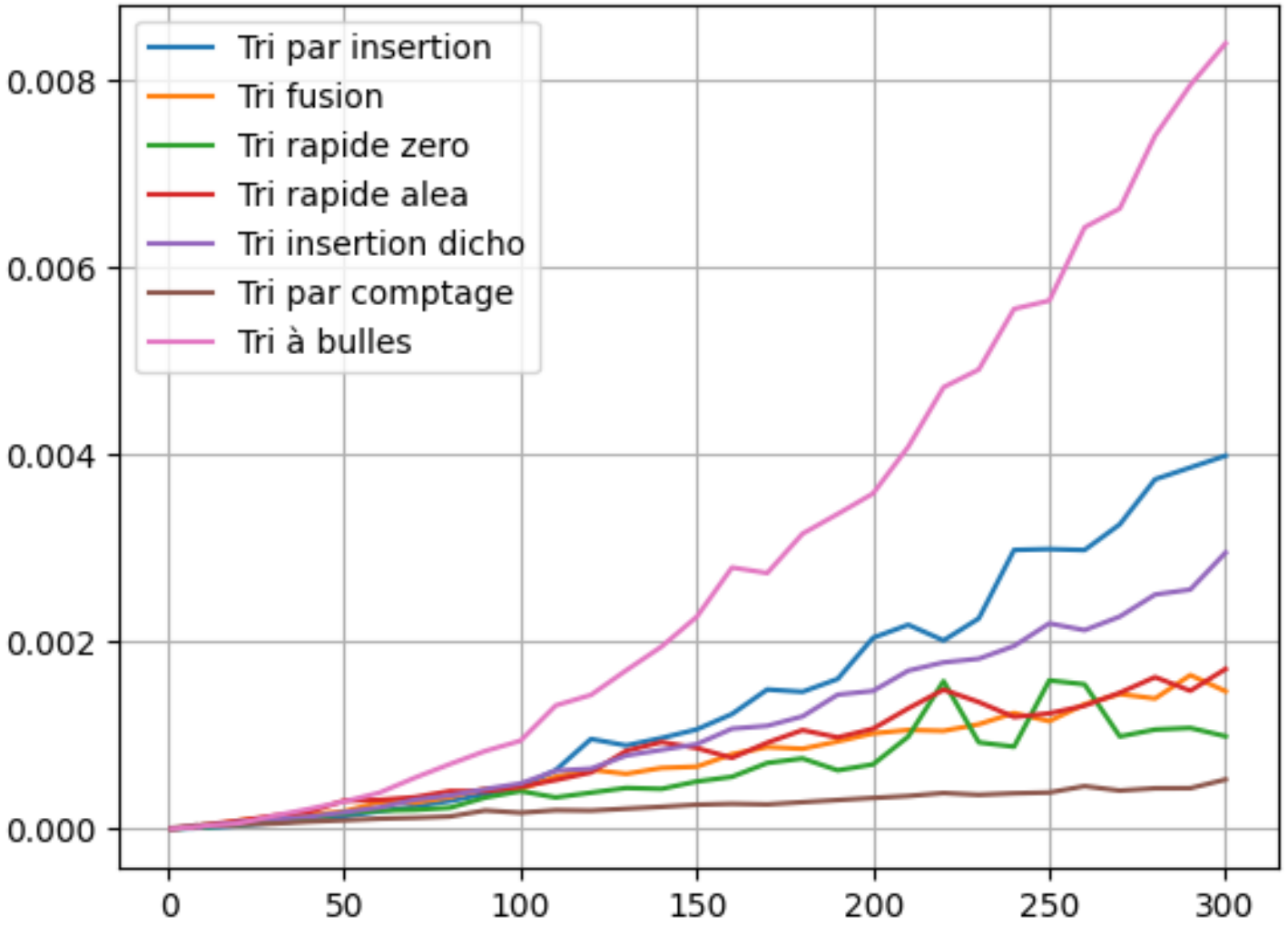
MPII
ITC
Séquence 7&8

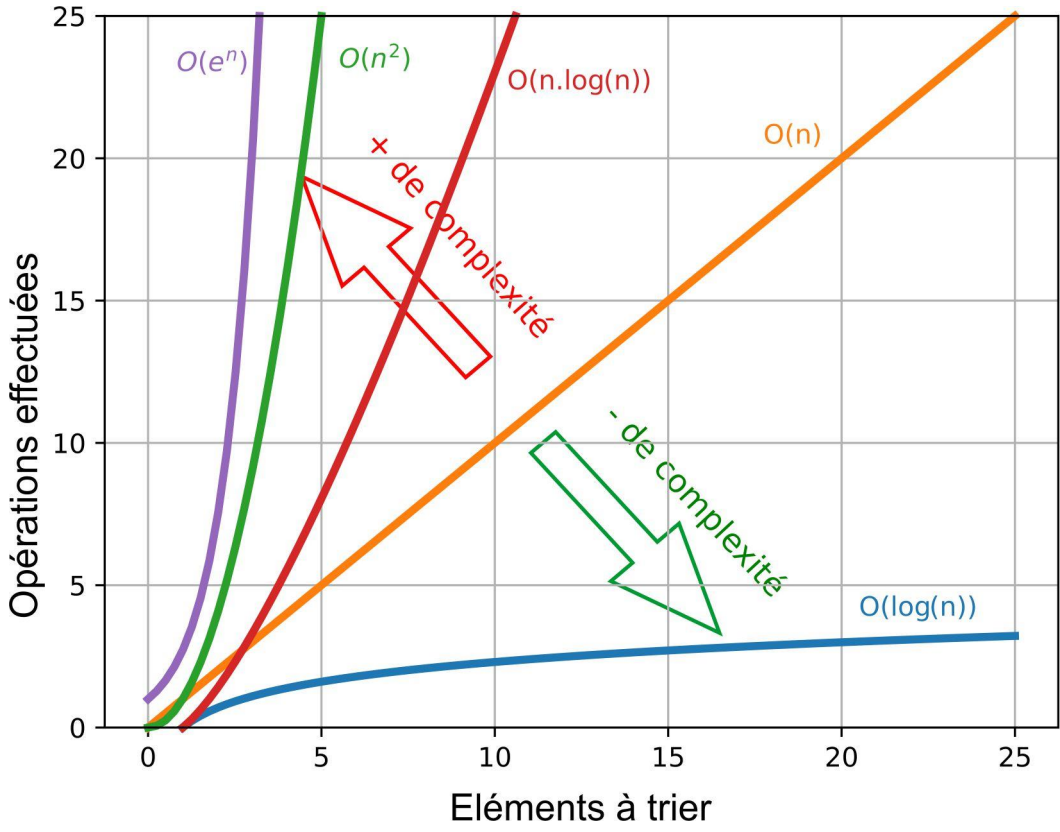
Résumé Tris

RÉSUMÉ TRIS

Tris	Principe Soit L la liste à trier	Complexité Au mieux	 Au pire	Stable	En place
Sélection	1. On recherche le min de L et on l'échange avec le 1er terme de L, 2. On sélection le min de L privé de son premier terme et on échange, avec le 2nd terme de L, 3. On procède ainsi sur les sous-listes suivantes.		$\mathcal{O}(n^2)$	non	oui
Insertion	1. On trie les deux premiers éléments, 2. On insère le troisième élément, « clé » à placer, de façon à avoir les trois premiers éléments ordonnés : on parcourt la liste L, on décale les éléments supérieurs à la « clé » et on répète ce décalage autant de fois que nécessaire afin d'insérer la « clé » à sa place. 3. On réitère sur les éléments suivants à placer.	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	oui	oui
À bulle	On parcourt la liste L, si 2 éléments consécutifs ne sont pas dans l'ordre, ils sont échangés et on recommence jusqu'à exécuter un parcours de L sans échange.	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	oui	oui
Fusion **(type diviser pour régner)**	• Si L n'a qu'un seul élément, elle est déjà triée, • Sinon, on sépare L en 2 parties à peu près égales, • On trie récursivement les 2 parties, • On fusionne les 2 sous-listes triées en une seule liste triée.		$\mathcal{O}(n.log(n))$	oui (si on privilégie la liste de gauche)	non
Rapide **(quick sort type diviser pour régner)**	• On choisit un pivot à placer (naïvement la 1ère valeur de L), • On partitionne L en plaçant les éléments inférieurs au pivot à gauche, les supérieurs à droite, on combine les 2 listes obtenues et le pivot placé, • Pour chacune des sous-listes, on définit un nouveau pivot et on réitère récursivement le partitionnement jusqu'à obtenir des listes à 1 élément.	$\mathcal{O}(nlog(n))$	$\mathcal{O}(n^2)$	non	non
Comptage	Soit L une liste de n entiers positifs à trier : 1. Initialisation : m la valeur maximale de la liste L et on crée une liste H remplie de $m+1$ valeurs nulles (chaque élément H[i] est le nombre d'apparition de la valeur i dans L), 2. Dénombrement : on compte les apparitions de chaque terme de L par un parcours de la liste L en incrémentant de 1 l'élément $\mathrm{H}[\mathrm{L}[\mathrm{i}]]$. A la fin de cette étape, H contient le nb d'apparition de chaque élément de L, 3. Reconstruction : on crée une nouvelle liste contenant dans l'ordre, les éléments de L autant de fois qu'ils apparaissent, à l'aide de la liste H.		$\mathcal{O}(n+m) \approx \mathcal{O}(n)$ Avec m la valeur maximale de la liste L, en général petit	non	Non

RÉSUMÉ TRIS


Tri par insertion
Tri fusion
Tri rapide zero
Tri rapide alea
Tri insertion dicho
Tri par comptage
Tri à bulles
0.008
0.006
0.004
0.002
0.000
0
50
100
150
200
250
300


$O(e^n)$
$O(n^2)$
O(n.log(n))
O(n)
O(log(n))
+ de complexité
- de complexité
Opérations effectuées
Eléments à trier
0
5
10
15
20
25