

# **Annexe**

## **– Algorithmes Élémentaires de Tri –**

# Table des Matières

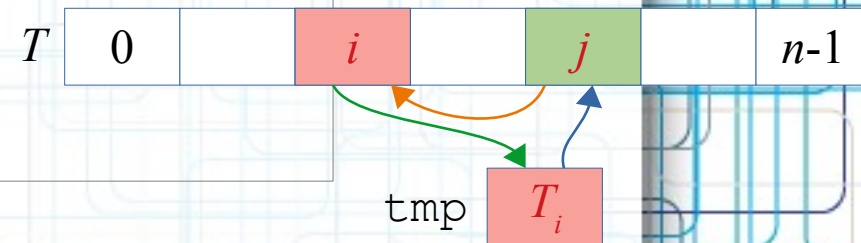
- **Introduction**
  - Echange, comparaison
  - Mélange aléatoire
- Tris élémentaires de tableau
  - Tri à bulle (*bubble sort*)
  - Tri par sélection (*selection sort*)
  - Tri par insertion (*insertion sort*)
  - Tri par fusion (*merge sort*)
- Tri de liste chaînée

# Fonction de support

- **Rappels**

- Echange de 2 éléments

```
public static void exch(Object[] tab,  
                        int i,  
                        int j) {  
    Object tmp = tab[i];  
    tab[i] = tab[j];  
    tab[j] = tmp;  
}
```



- Comparaison de 2 éléments

- **ordre naturel** : `Comparable<E>.compareTo(E e)`
    - **autre ordre** : `Comparator<E>.compare(E e1, E e2)`



# Tester les tris

- Mélanger aléatoirement un tableau

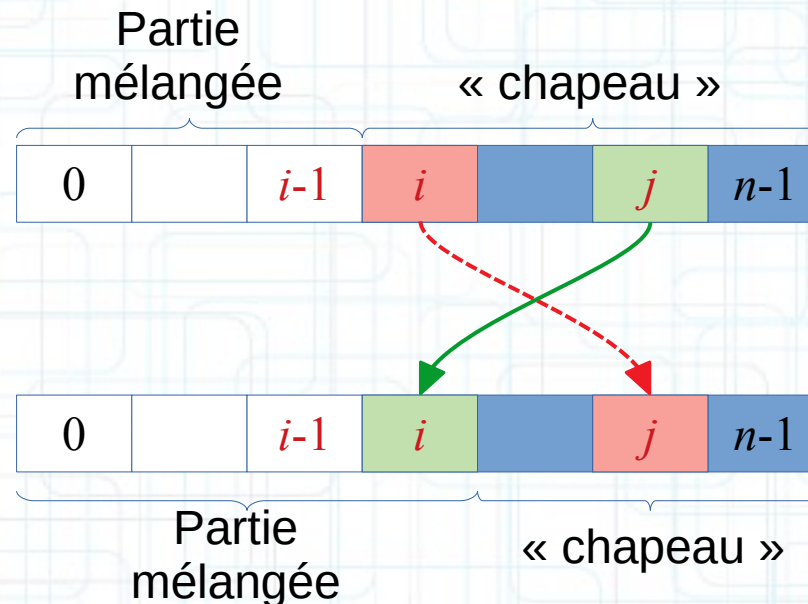
```
public static void shuffle(Object[] tab) {  
    Random r = new Random();  
    for (int i = 0; i < tab.length - 1; i++) {  
        int j = i + r.nextInt(tab.length - i);  
        exch(tab, i, j);  
    }  
}
```

algorithme de Fisher-Yates

aléatoirement dans  $[0, n-i[$   
 $\Rightarrow i \leq j < n$

A l'itération  $i$ , on place en position  $i$  un élément  $j$  choisi aléatoirement dans l'intervalle  $[i, n[$

L'élément qui était en position  $i$  auparavant est « remis dans le chapeau ».



# Table des Matières

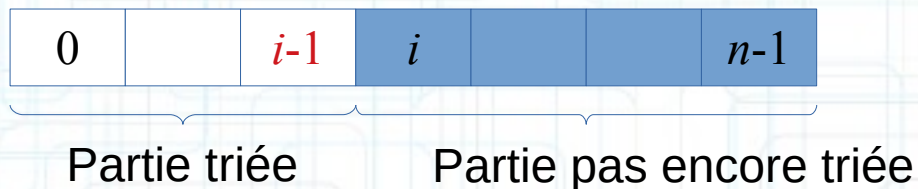
- Introduction
  - Echange, comparaison
  - Mélange aléatoire
- Tris élémentaires de tableau
  - **Tri à bulle (*bubble sort*)**
  - Tri par sélection (*selection sort*)
  - Tri par insertion (*insertion sort*)
  - Tri par fusion (*merge sort*)
- Tri de liste chaînée



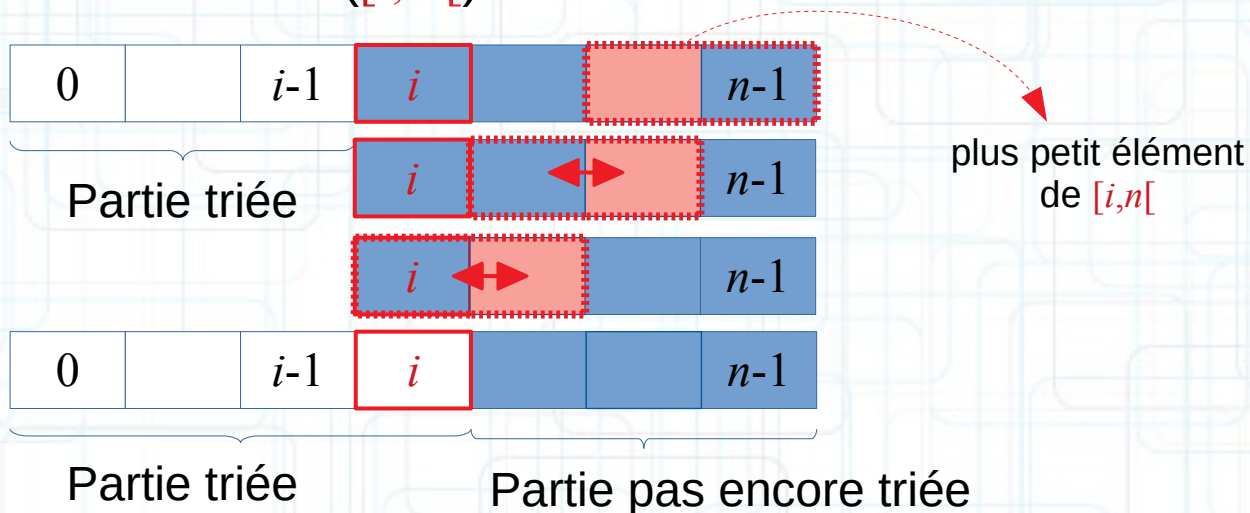
# Bubble sort

- **Principe de fonctionnement**

- Au début de l'itération  $i$ , les éléments  $[0, i-1]$  sont triés définitivement



- l'itération  $i$  fait descendre en position  $i$  le plus petit élément de la partie non-triée ( $[i, n[$ )



# Bubble sort

- Implémentation en Java

```
public static void sort(Comparable[] tab) {  
    int n= tab.length;  
    for (int i= 0; i < n; i++)  
    {  
        for (int j= n-1; j > i; j--)  
            if (tab[j].compareTo(tab[j-1]) < 0)  
                exch(tab, j, j-1);  
    }  
}
```

inspiré de *Algorithms in Java*, 4<sup>th</sup> edition  
de R. Sedgewick et K. Wayne

Boucle interne fait  
"descendre" le plus petit  
élément.

Propriété : au début de la  
 $i^{\text{ème}}$  itération, les éléments  
 $[0, i-1]$  sont triés.

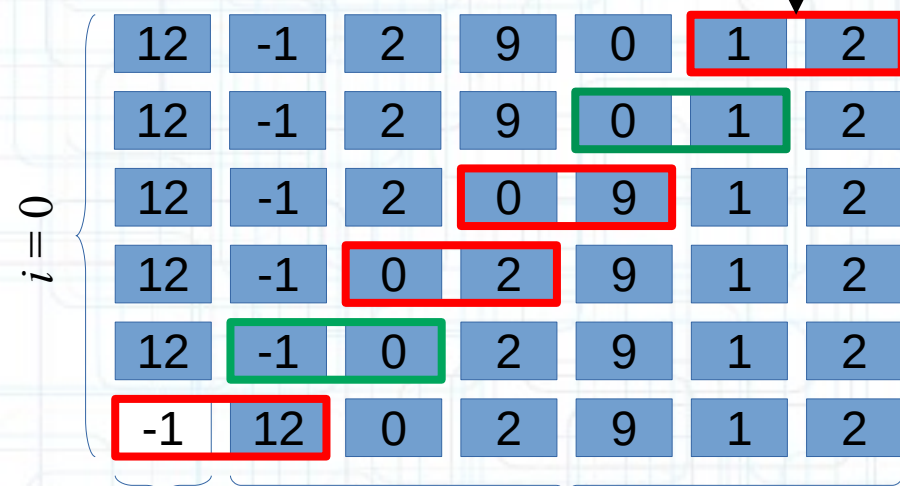
Cette itération ne traite  
donc que les éléments  
 $[i, n-1]$ .

Comme il faut au moins 2  
éléments pour faire une  
comparaison, on a que  
 $i < j \leq n-1$ .

# Bubble Sort

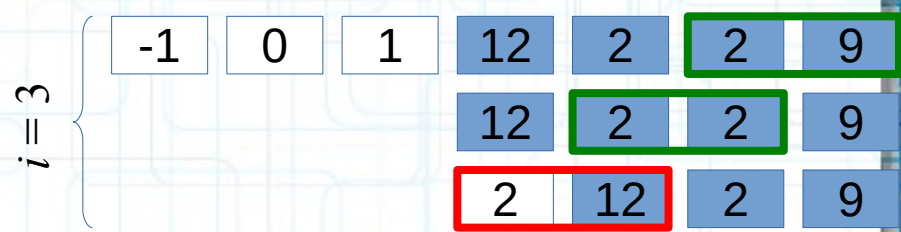
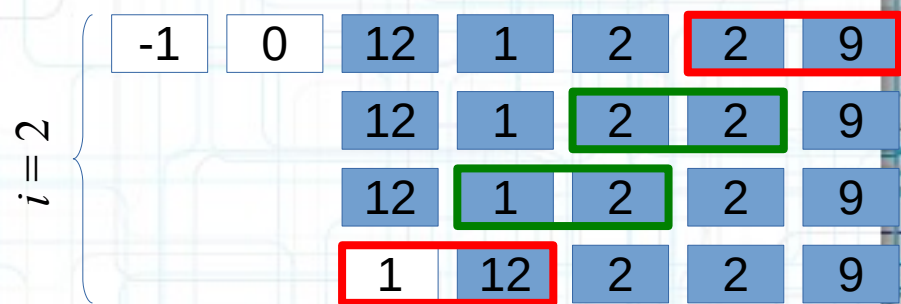
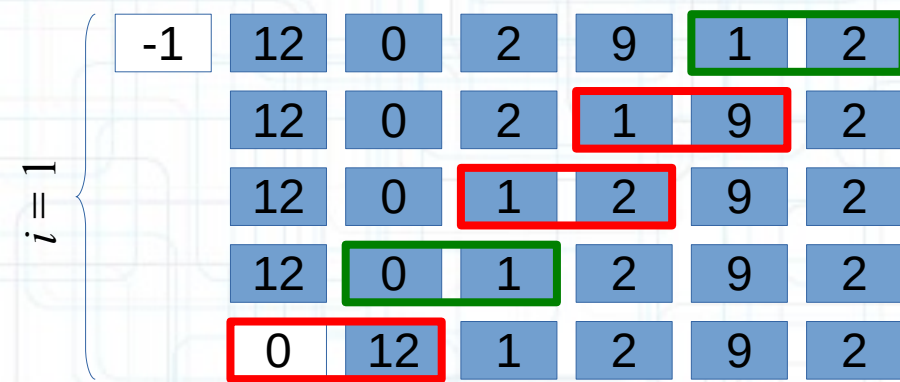


12 -1 2 9 0 2 1



Partie triée

Partie pas encore triée





# Bubble Sort

- Implémentation en Java

- La variante ci-dessous arrête le tri dès qu'une itération a lieu dans laquelle aucun échange n'est effectué, signifiant que le tableau est déjà trié.

```
public static void sort(Comparable[] tab) {  
    int n= tab.length;  
    for (int i= 0; i < n; i++) {  
        int exchanges= 0;  
        for (int j= n-1; j > i; j--) {  
            if (tab[j].compareTo(tab[j-1]) < 0) {  
                exch(tab, j, j-1);  
                exchanges++;  
            }  
        }  
        if (exchanges == 0)  
            break;  
    }  
}
```

Compte le nombre d'échanges durant la dernière itération.

Si pas d'échanges, le tableau est trié : arrêt plus tôt.

inspiré de *Algorithms in Java*, 4<sup>th</sup> edition  
de R. Sedgewick et K. Wayne

# Table des Matières

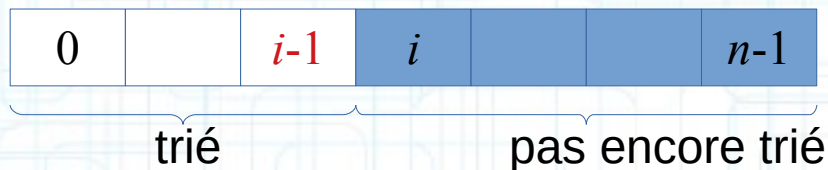
- Introduction
  - Echange, comparaison
  - Mélange aléatoire
- Tris élémentaires de tableau
  - Tri à bulle (*bubble sort*)
  - **Tri par sélection (*selection sort*)**
  - Tri par insertion (*insertion sort*)
  - Tri par fusion (*merge sort*)
- Tri de liste chaînée



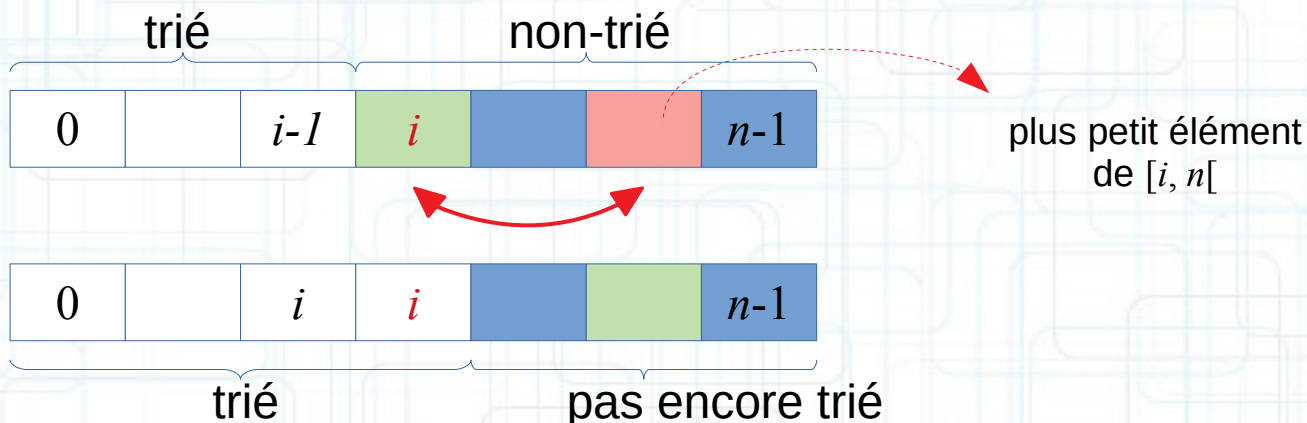
# Selection Sort

- **Principe de fonctionnement**

- Au début de l'itération  $i$ , les éléments  $[0, i-1]$  sont triés définitivement



- L'itération  $i$  **sélectionne** le plus petit élément de la partie non-triée ( $[i, n[$ ) et le place en position  $i$





# Selection Sort

- Implémentation en Java

```
public static void sort(Comparable[] tab) {  
    int n= tab.length;  
    for (int i= 0; i < n-1; i++) {  
        int min= i;  
        for (int j= i+1; j < n; j++) {  
            if (tab[j].compareTo(tab[min]) < 0)  
                min= j;  
        }  
        exch(tab, i, min);  
    }  
}
```

Trouve l'élément  
minimum de  $[i, n[$

# Selection Sort



12 -1 2 9 0 2 1

$$i=0 \left\{ \begin{array}{cccccc} 12 & -1 & 2 & 9 & 0 & 1 & 2 \\ -1 & 12 & 2 & 9 & 0 & 1 & 2 \end{array} \right.$$

Partie triée

Partie pas encore triée

$$i=1 \left\{ \begin{array}{cccccc} -1 & 12 & 2 & 9 & 0 & 1 & 2 \\ -1 & 0 & 2 & 9 & 12 & 1 & 2 \end{array} \right.$$

$$i=2 \left\{ \begin{array}{cccccc} -1 & 0 & 2 & 9 & 12 & 1 & 2 \\ -1 & 0 & 1 & 9 & 12 & 2 & 2 \end{array} \right.$$

$$i=3 \left\{ \begin{array}{cccccc} -1 & 0 & 1 & 9 & 12 & 2 & 2 \\ -1 & 0 & 1 & 2 & 12 & 9 & 2 \end{array} \right.$$

$$i=4 \left\{ \begin{array}{cccccc} -1 & 0 & 1 & 2 & 12 & 9 & 2 \\ -1 & 0 & 1 & 2 & 2 & 9 & 12 \end{array} \right.$$

$$i=5 \left\{ \begin{array}{cccccc} -1 & 0 & 1 & 2 & 2 & 9 & 12 \\ -1 & 0 & 1 & 2 & 2 & 9 & 12 \end{array} \right.$$

pas besoin d'itération  $i=n-1=6$   
car la partie pas encore triée  
n'est plus composée que  
d'un seul élément => triée

# Table des Matières

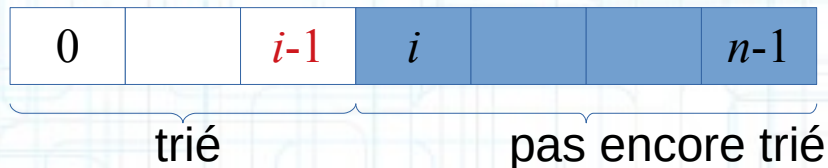
- Introduction
  - Echange, comparaison
  - Mélange aléatoire
- Tris élémentaires de tableau
  - Tri à bulle (*bubble sort*)
  - Tri par sélection (*selection sort*)
  - **Tri par insertion (*insertion sort*)**
  - Tri par fusion (*merge sort*)
- Tri de liste chaînée



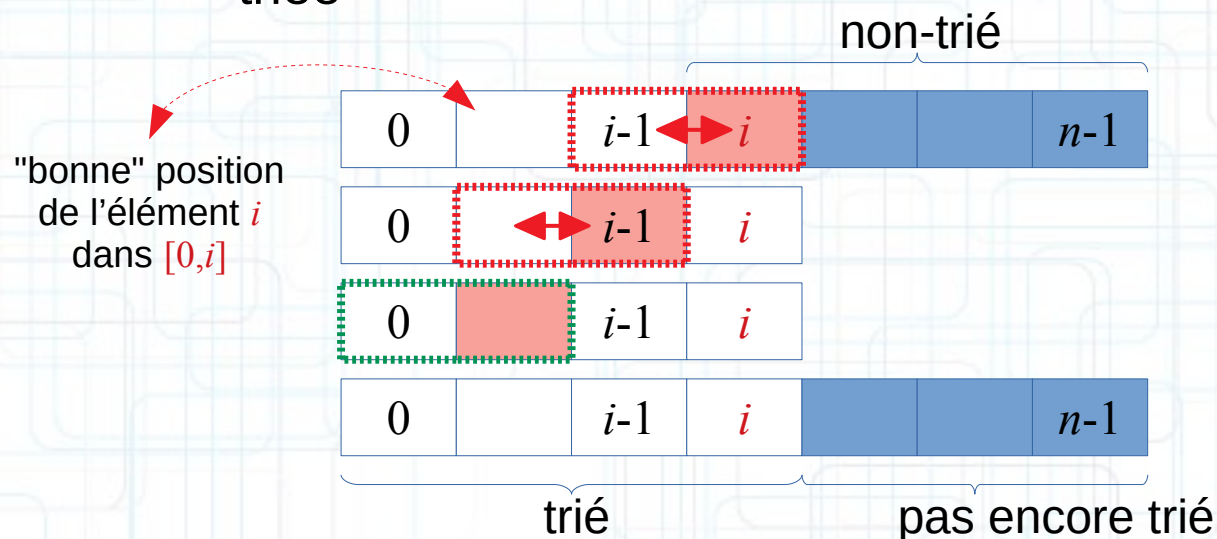
# Insertion Sort

- **Principe de fonctionnement**

- Au début de l'itération  $i$ , les éléments  $[0, i-1]$  sont triés, mais pas forcément à leur position définitive



- L'itération  $i$  **insère** l'élément  $i$  à la "bonne" position dans la partie triée



# Insertion Sort

- Implémentation en Java

Note : à l'itération 0, l'élément 0 est déjà à la bonne place.

```
public static void sort(Comparable[] tab) {  
    int n= tab.length;  
    for (int i= 1; i < n; i++) {  
        for (int j= i; j > 0; j--) {  
            if (tab[j].compareTo(tab[j-1]) < 0)  
                exch(tab, j, j-1);  
            else  
                break;  
        }  
    }  
}
```

inspiré de *Algorithms in Java*, 4<sup>th</sup> edition  
de R. Sedgewick et K. Wayne



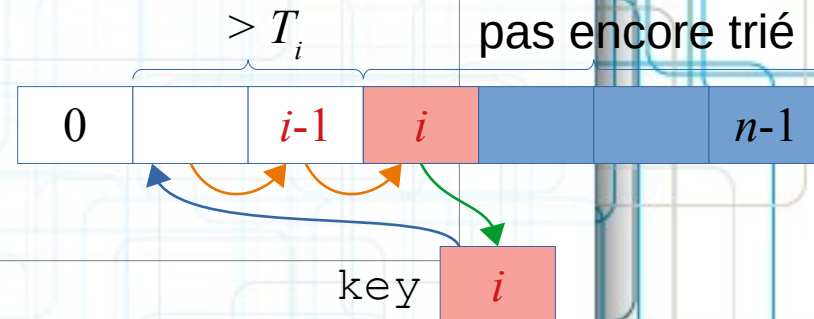
# Insertion Sort

- Implémentation en Java

- La variante suivante effectue l'insertion de manière plus efficace. Plutôt que procéder par échanges successifs, elle fait remonter les éléments de  $[0, i]$  qui sont supérieurs à  $T_i$ .

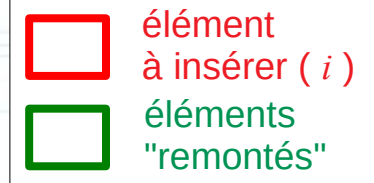
```
public static void sort(Comparable[] tab) {  
    int n= tab.length;  
    for (int i= 1; i < n; i++) {  
        Comparable key = tab[i];  
        for (int j = i-1; j >= 0 &&  
            key.compareTo(tab[j] < 0; j--)  
            tab[j+1] = tab[j];  
        tab[j+1] = key;  
    }  
}
```

inspiré de *Introduction to Algorithms in Java*, 3<sup>rd</sup> edition  
de Th. Cormen et al



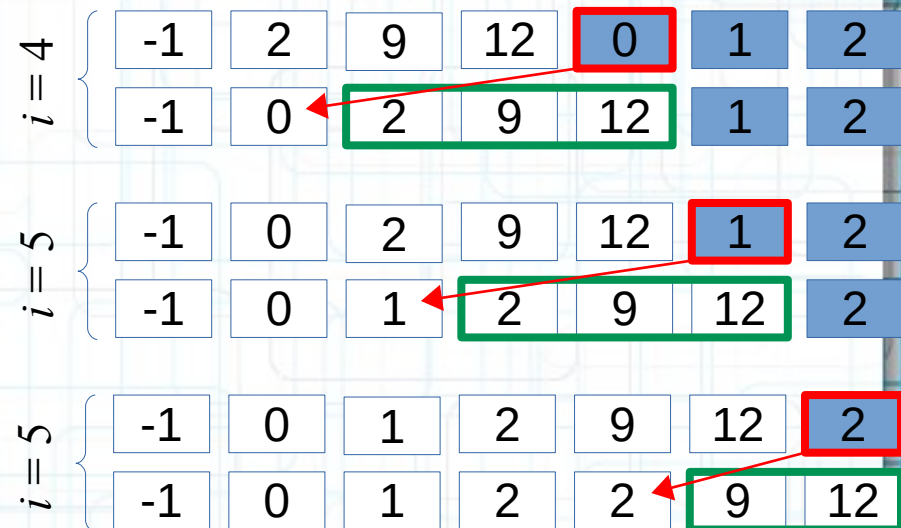
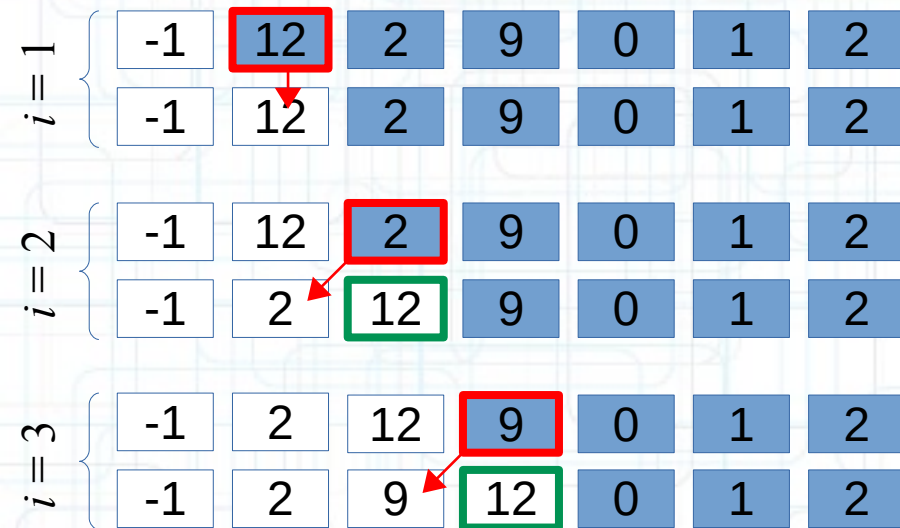


# Insertion Sort



12 -1 2 9 0 2 1

pas besoin d'itération  $i=0$   
 car le tableau  $[0,0]$  n'est  
 composé que d'un seul élément  
 $\Rightarrow$  déjà trié



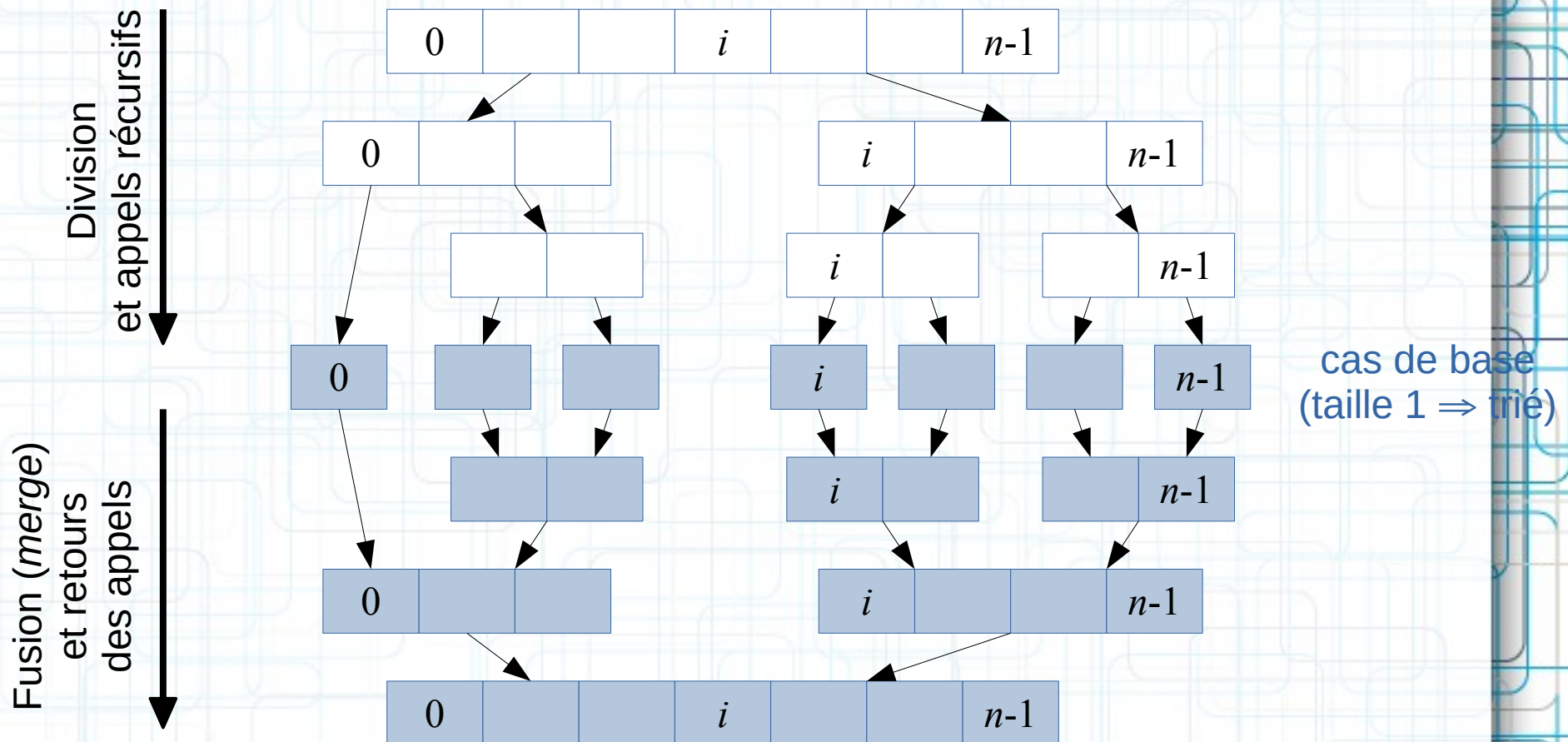
# Table des Matières

- Introduction
  - Echange, comparaison
  - Mélange aléatoire
- Tris élémentaires de tableau
  - Tri à bulle (*bubble sort*)
  - Tri par sélection (*selection sort*)
  - Tri par insertion (*insertion sort*)
  - **Tri par fusion (*merge sort*)**
- Tri de liste chaînée

# Merge Sort

- **Principe de fonctionnement**

- Approche « *diviser pour régner* », réursive





# Merge Sort

- **Fusion**

- L'opération de fusion vise à produire un tableau trié à partir de deux tableaux triés.

```
public static Comparable [] merge(Comparable[] a, Comparable[] b)
{
    Comparable[] tab = new Comparable[a.length + b.length];
    int i = 0, j = 0, k = 0;
    while (k < tab.length) {
        if (i >= a.length)                tab[k] = b[j++];
        else if (j >= b.length)            tab[k] = a[i++];
        else if (a[i].compareTo(b[j]) < 0) tab[k] = a[i++];
        else                               tab[k] = b[j++];
        k++;
    }
}
```

# Merge Sort

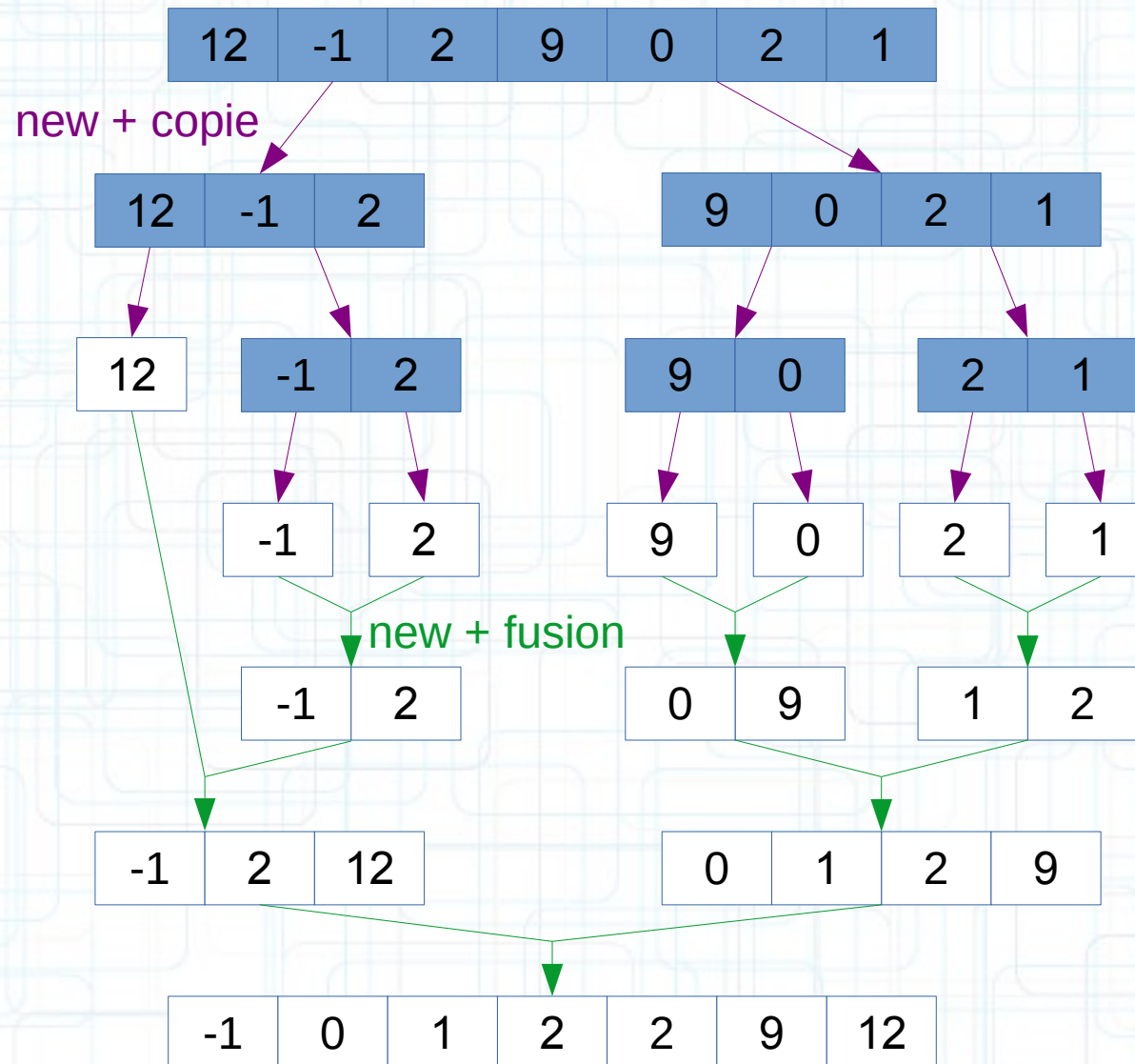
- **Tri**

- Le tri d'un tableau peut alors être réalisé en divisant celui-ci en deux parties, en les triant récursivement et en fusionnant ensuite les parties triées.

```
public static Comparable [] mergeSort(Comparable[] a)
{
    int n = tab.length;
    if (n <= 1)
        return a;

    Comparable t1 = mergeSort(Arrays.copyOfRange(a, 0, n/2));
    Comparable t2 = mergeSort(Arrays.copyOfRange(a, n/2, n));
    return merge(t1, t2);
}
```

# Merge Sort



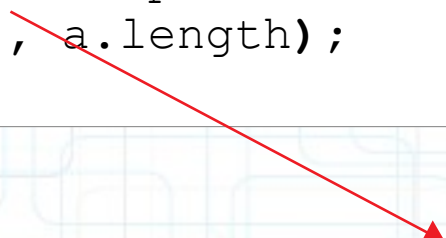


# Merge Sort

- **Variante**

- Il est possible d'implémenter le tri par fusion en effectuant une seule allocation mémoire d'une taille égale à celle du tableau à trier (la complexité spatiale est donc en  $O(N)$ ).
- Cette variante est proposée ci-dessous et au slide suivant.

```
public static void mergeSortDriver(Comparable[] a)
{
    Comparable[] aux= new Comparable[a.length];
    mergeSort2(a, aux, 0, a.length);
}
```



Une seule allocation  
d'un nouveau tableau

# Merge Sort

```
public static void mergeSort2 (Comparable[] a, Comparable[] aux,
                               int b, int e)
{
    int n= e - b;
    if (n < 2)
        return;

    int mid= b + n / 2;
    mergeSort2(a, aux, b, mid);
    mergeSort2(a, aux, mid, e);

    int i= b, j= mid, k= b;
    while (k < e) {
        if (i >= mid)                aux[k]= a[j++];
        else if (j >= e)            aux[k]= a[i++];
        else if (a[i].compareTo(a[j]) < 0) aux[k]= a[i++];
        else                        aux[k]= a[j++];
        k++;
    }
    System.arraycopy(aux, b, a, b, n);
}
```

# Merge Sort

(b=0,e=7)

12	-1	2	9	0	2	1
----	----	---	---	---	---	---

(0,3)

12	-1	2	9	0	2	1
----	----	---	---	---	---	---

(3,7)

(0,1)

(1,3)

(3,5)

(5,7)

12	-1	2	9	0	2	1
----	----	---	---	---	---	---

(0,1)

(1,2)

(2,3)

(3,4)

(4,5)

(5,6)

(6,7)

12	-1	2	9	0	2	1
----	----	---	---	---	---	---

fusion

tableau auxiliaire (aux)

copie

	-1	2	0	9	1	2
--	----	---	---	---	---	---

(0,1)

(1,3)

(3,5)

(5,7)

12	-1	2	0	9	1	2
----	----	---	---	---	---	---

(0,3)

(3,7)

-1	2	12	0	1	2	9
----	---	----	---	---	---	---

(0,7)

-1	0	1	2	2	9	12
----	---	---	---	---	---	----

-1	2	12	0	1	2	9
----	---	----	---	---	---	---

-1	0	1	2	2	9	12
----	---	---	---	---	---	----



# Merge Sort

- **Variante**

- Pourrait-on encore aller plus loin en diminuant le nombre de copies (cf. `System.arraycopy` après la fusion) ?
- ...

# Table des Matières

- Introduction
  - Echange, comparaison
  - Mélange aléatoire
- Tris élémentaire de tableau
  - Tri à bulle (*bubble sort*)
  - Tri par sélection (*selection sort*)
  - Tri par insertion (*insertion sort*)
  - Tri par fusion (*merge sort*)
- **Tri de liste chaînée**

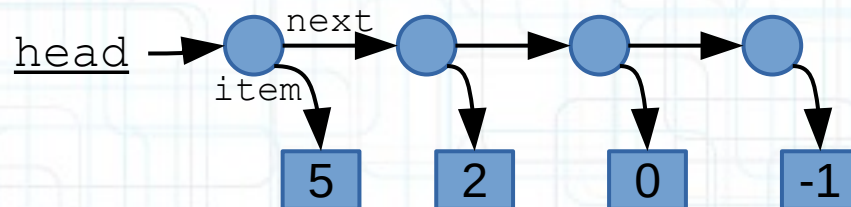
# Tri de Liste Chaînée

- Simple liste chaînée

- Pour cette discussion, nous considérons des *listes simplement chaînées* basées sur la classe Node montrée ci-dessous.

```
public class Node<E>
{
    public Node<E> next;
    public E      item;
    public Node(E item)
    { this.item= item; }
    public Node(E item, Node<E> next)
    { this.item= item;
      this.next= next; }
}
```

```
Node<Integer> head=
    new Node(5,
        new Node(2,
            new Node(0,
                new Node(-1))));
```



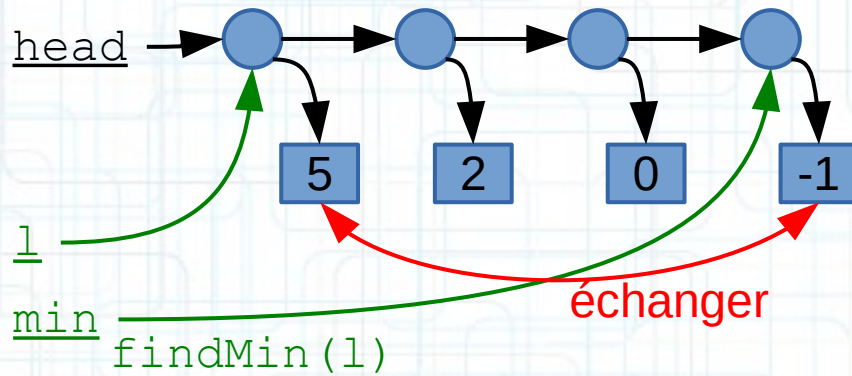


# Selection Sort

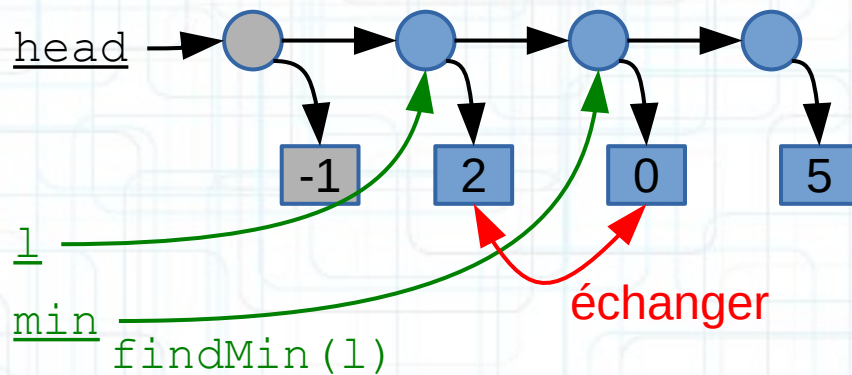
- **Principe de fonctionnement**

- Deux variantes du tri par sélection sont présentées.
- Pour rappel, à chaque itération, le tri par sélection détermine le plus petit élément de la partie non triée et l'ajoute à la fin de la partie déjà triée.
- Variante 1
  - le chaînage de la liste est inchangé
  - seule la partie « donnée » (`item`) des nœuds est déplacée lors des échanges
- Variante 2
  - la partie « donnée » (`item`) n'est pas déplacée<sup>(1)</sup> → il faut recâbler la liste dans le bon ordre
  - à chaque itération, sélection et retrait du plus grand élément; le second plus grand est inséré devant le premier, etc. Au final, la liste est produite dans le bon ordre

# Selection Sort de Liste Chaînée (v1)



1ère itération



2ème itération



# Tri de Liste Chaînée (v1)

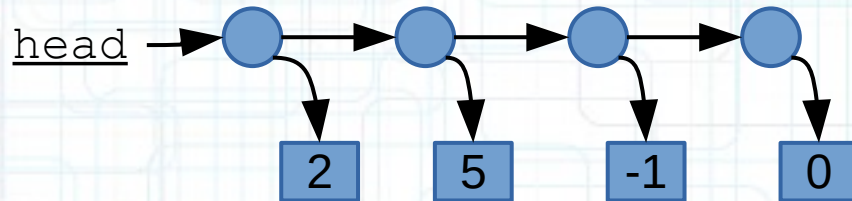
```
private static <E> Node<E> findMin(Node<E> l)
{
    Node<E> min= null;
    while (l != null) {
        if ((min == null) ||
            less(l.item, min.item))
            min= l;
        l= l.next;
    }
    return min;
}
```

`l.item.compareTo(min.item) < 0`

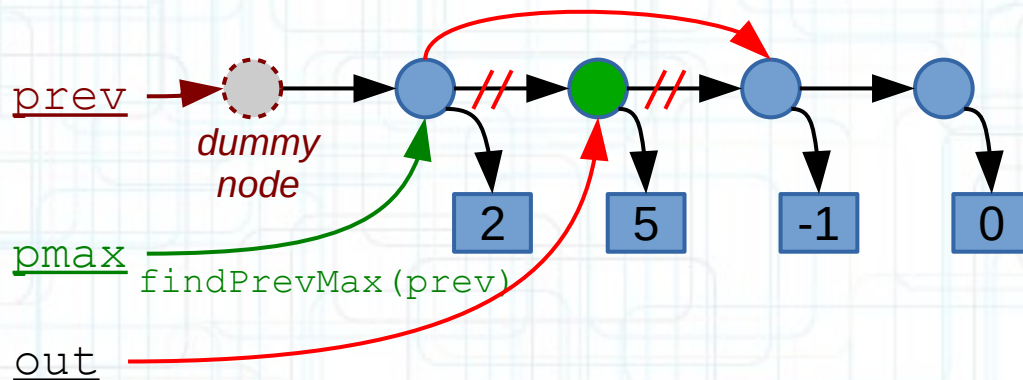
```
private static <E> void sortList(Node<E> l)
{
    while (l != null) {
        Node<E> min= findMin(l);
        exch(l, min);
        l= l.next;
    }
}
```



# Selection Sort de Liste Chaînée (v2)

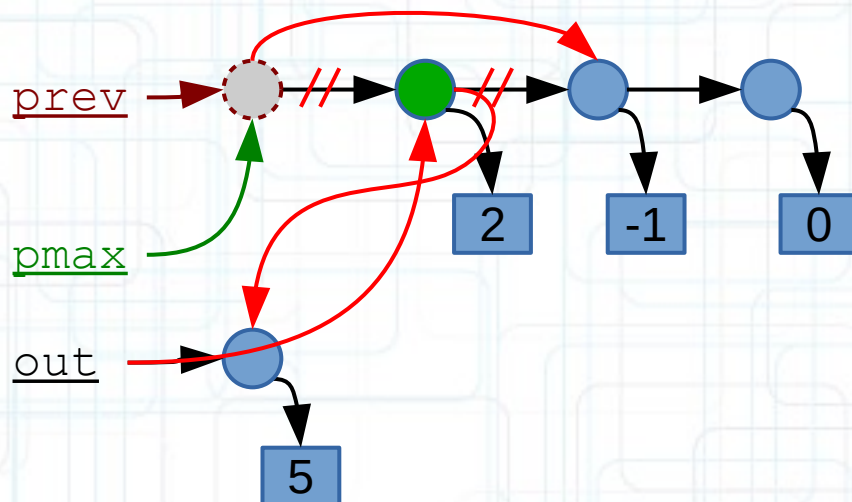


**Liste initiale**



**1<sup>ère</sup> itération**

sélection plus grand élément (5)  
retrait de la liste  
insertion à l'avant de la liste out  
→ 5



**2<sup>ème</sup> itération**

sélection second plus grand (2)  
retrait de la liste  
insertion à l'avant de la liste out  
→ 2;5

# Tri de Liste Chaînée (v2)

```
private static <E> Node<E> findMaxPrev(Node<E> pl)
{
    Node<E> pmax= null;
    while (pl.next != null) {
        if ((pmax == null) ||
            less(pmax.next.item, pl.next.item))
            pmax= pl;
        pl= pl.next;
    }
    return pmax;
}
```

```
private static <E> Node<E> sortList2(Node<E> l)
{
    Node<E> prev= new Node<>(null, l);
    Node<E> out= null;
    while (prev.next != null) {
        Node<E> pmax= findMaxPrev(prev);
        Node<E> max= pmax.next;
        pmax.next= max.next;
        max.next= out;
        out= max;
    }
    return out;
}
```