INF2010 – ASD

Algorithmes de tri

Plan

I. Notions de base
II. Les tris simples
III. Mergesort
IV. Quicksort

Plan

I. Notions de base
II. Les tris simples
III. Mergesort
IV. Quicksort

I – Notions de base : l'utilité des tris

Problématique:

- Un grand nombre de problèmes réels ont recours à la recherche d'éléments dans un vecteur (tableau). Cette recherche devient quasiment impossible lorsqu'elle s'effectue dans un vecteur désordonné, d'où le besoin de trier les vecteurs avant d'y effectuer un traitement.

Exemples:

- Les mots dans un dictionnaire.
- Fichiers dans un répertoire de PC.
- Les CD répertoriés dans un magasin de musique.
- Les sigles des cours de Poly offert pour une session.

Définitions

- Tri interne : se dit d'un algorithme qui n'utilise pas de tableau temporaire pour effectuer le tri.

- Tri externe : se dit d'un algorithme de tri qui nécessite l'utilisation d'un tableau temporaire.

Plan

I. Notions de base
II. Les tris simples
III. Mergesort
IV. Quicksort

II – Les tris simples

- Tri par sélection
- Tri en bulle
- Tri par insertion

Tri par sélection

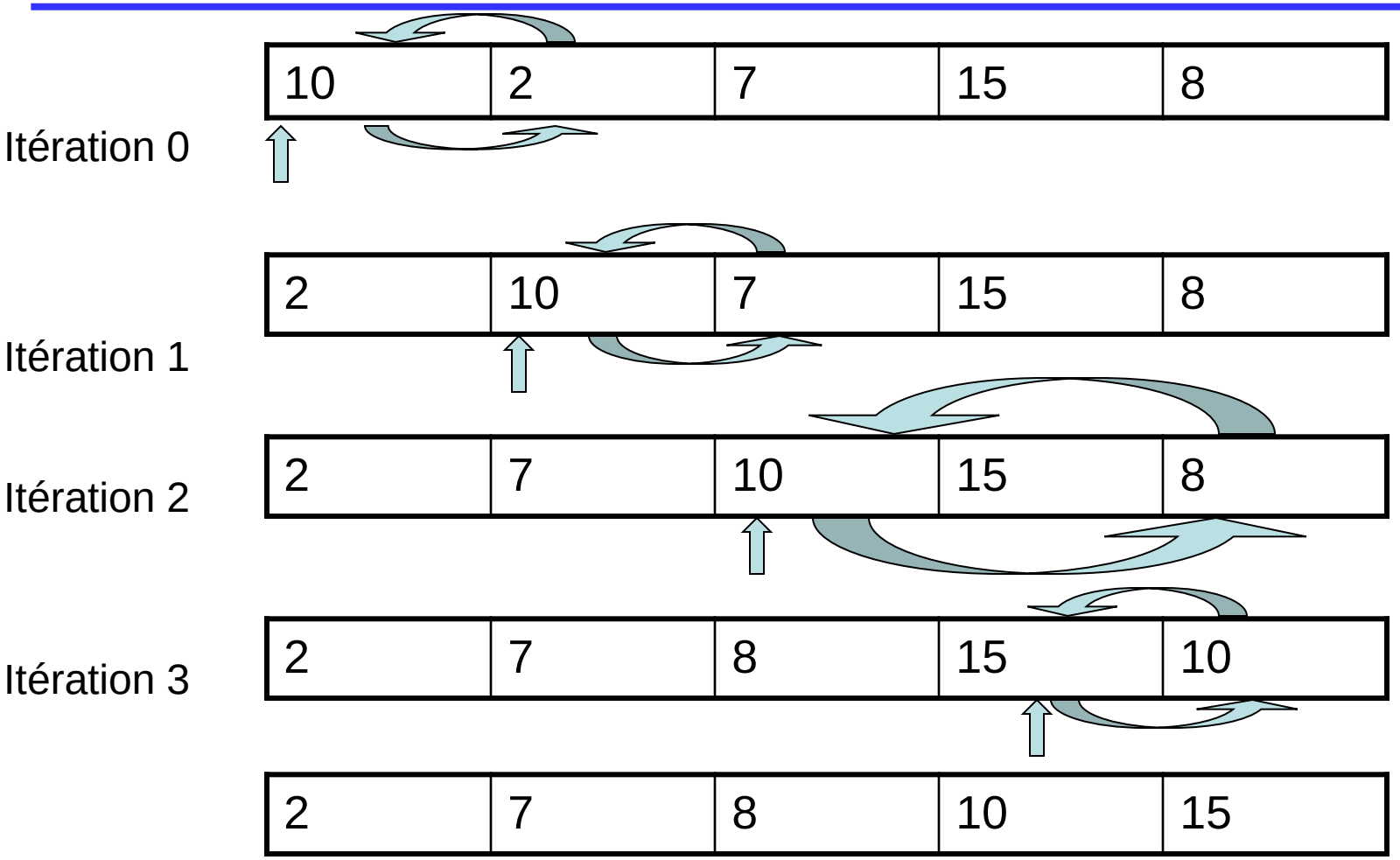
- Idée : Cette technique consiste à parcourir séquentiellement le vecteur à trier. À l'itération i, la plus petite valeur du tableau est interchangée avec la valeur située dans la case d'indice i.

Programmation du tri par sélection

```
public static <AnyType extends Comparable<? super AnyType>>
void selectionSort( AnyType [] a )
{
    int i, j, min;  AnyType tmp;

    for( i=0 ; i< a.length -1 ; i++ )
    {
        //Identification de l'index du plus petit élément
        min = i;
        for ( j=i+1 ; j< a.length ; j++ )  {
           if ( a[j]. compareTo(a[min] )<0 )
              min = j;
        }
        //Permutation des éléments
        tmp = a[i];
        a[i] = a[min];
        a[min] = tmp;
    }
}
```

Exemple d'exécution du tri par sélection



Tri en bulle

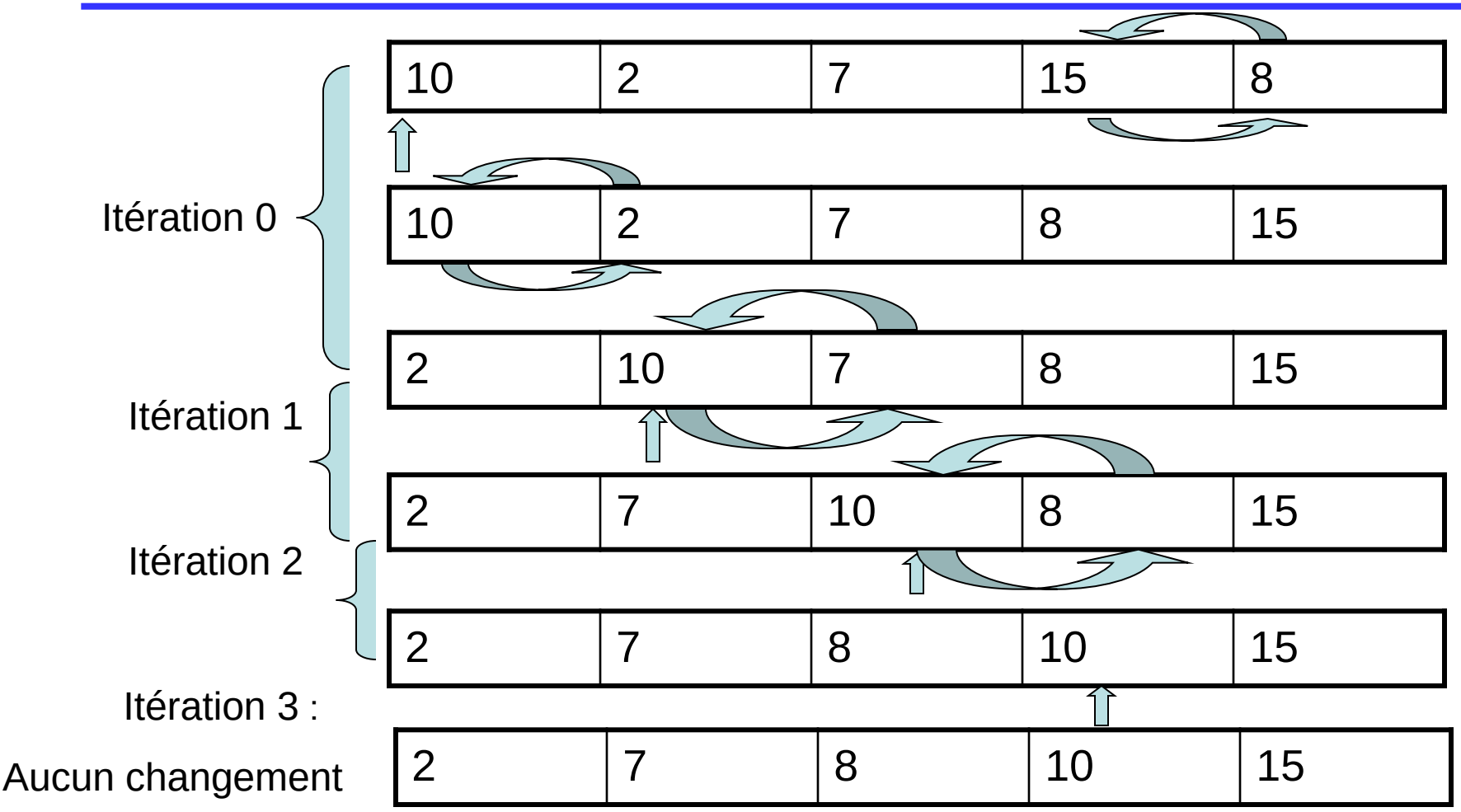
- Idée : Les petits éléments du tableau « remontent » (comme des bulles) vers le début du tableau pour atteindre leur position finale. Les plus « légers » remontent le plus rapidement. Une fois tous les éléments remonter, le tableau est trié.

Programmation du tri en bulle

```
public static <AnyType extends Comparable<? super AnyType>>
void bubbleSort( AnyType [ ] a )
{
   int i,j; AnyType tmp;

   for ( i = 0; i< a.length - 1; i++)
      for ( j = a.length - 1; j>i; j--)
        if ( a[j-1].compareTo(a[ j ]) > 0 )
        {
            //Permutation des 2 éléments
           tmp = a[ j-1 ];
           a[ j-1 ] = a[ j ];
           a[ j ] = tmp;
        }
}
```

Exemple d'exécution du tri en bulles



Tri par insertion

- Idée : Cette stratégie est semblable à la méthode qu'utilise naturellement un joueur de cartes pour organiser sa main. À chaque étape, une partie du tableau est déjà ordonnée et une nouvelle valeur est insérée à l'endroit approprié.

Programmation du tri par insertion

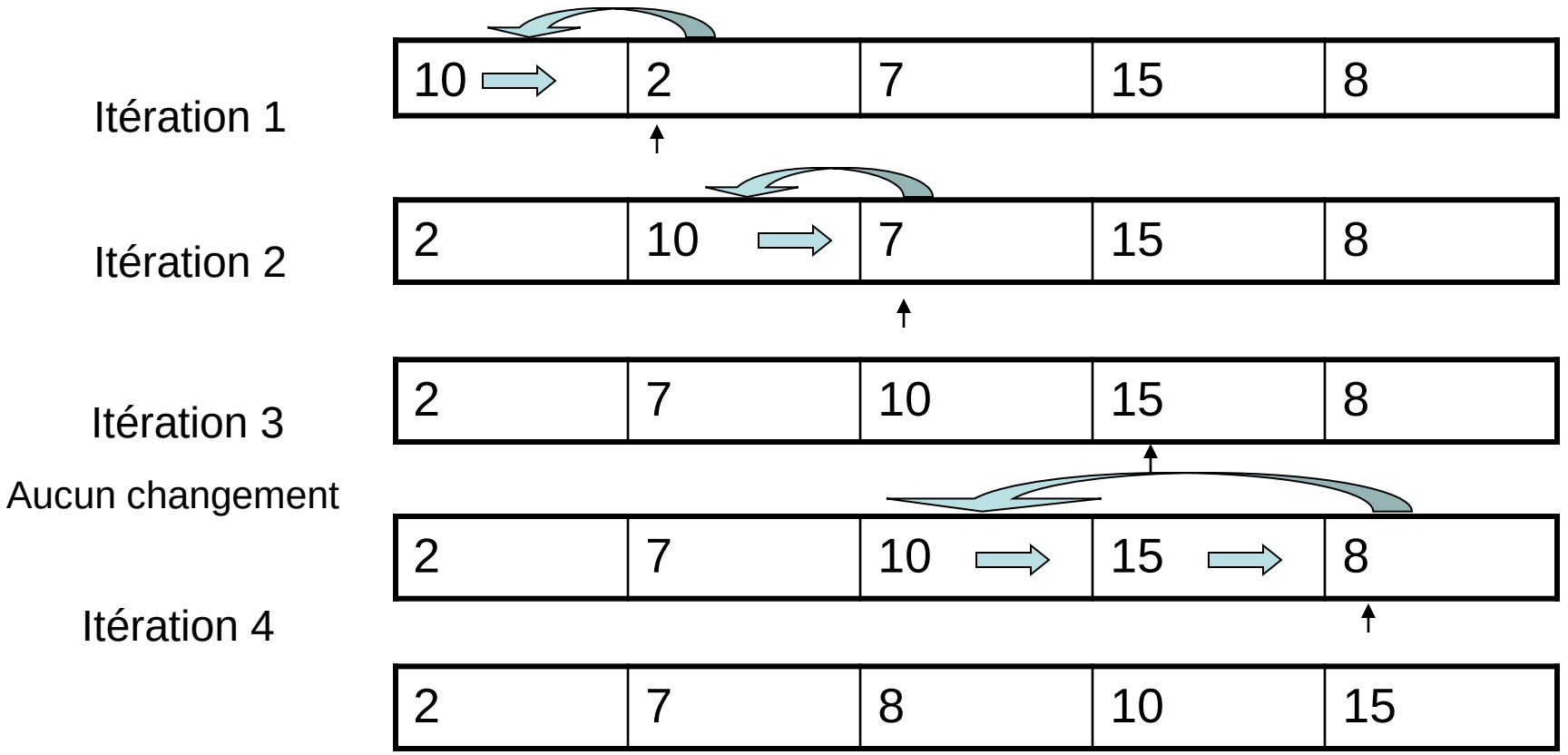
```
public static <AnyType extends Comparable<? super AnyType>>
void insertionSort( AnyType [ ] a )
{
   int j;

   for( int p = 1; p < a.length; p++ )
   {
      AnyType tmp = a[ p ];

      for( j = p; j > 0 && tmp.compareTo( a[ j - 1 ] ) < 0; j-- )
         a[ j ] = a[ j - 1 ];

      a[ j ] = tmp;
   }
}
```

Exemple d'exécution du tri par insertion



Analyse des tris simples

- Meilleur cas: tableau déjà trié

1	3	5	7	9	10	13	18	21	25

- Cas moyen: tableau aléatoire

7	10	21	9	1	25	18	3	5	13

- Pire cas: tableau trié à l'envers

25	21	18	13	10	9	7	5	3	1

Comparaison des tris simples

	Pire cas	Cas moyen	Meilleur cas
Tri par sélection	$O(n^2)$	$O(n^2)$	$O(n^2)$
Tri en bulle	$O(n^2)$	$O(n^2)$	$O(n^2)$
Tri par insertion	$O(n^2)$	$O(n^2)$	$O(n)$

Plan

I. Notions de base
II. Les tris simples
III. Mergesort
IV. Quicksort

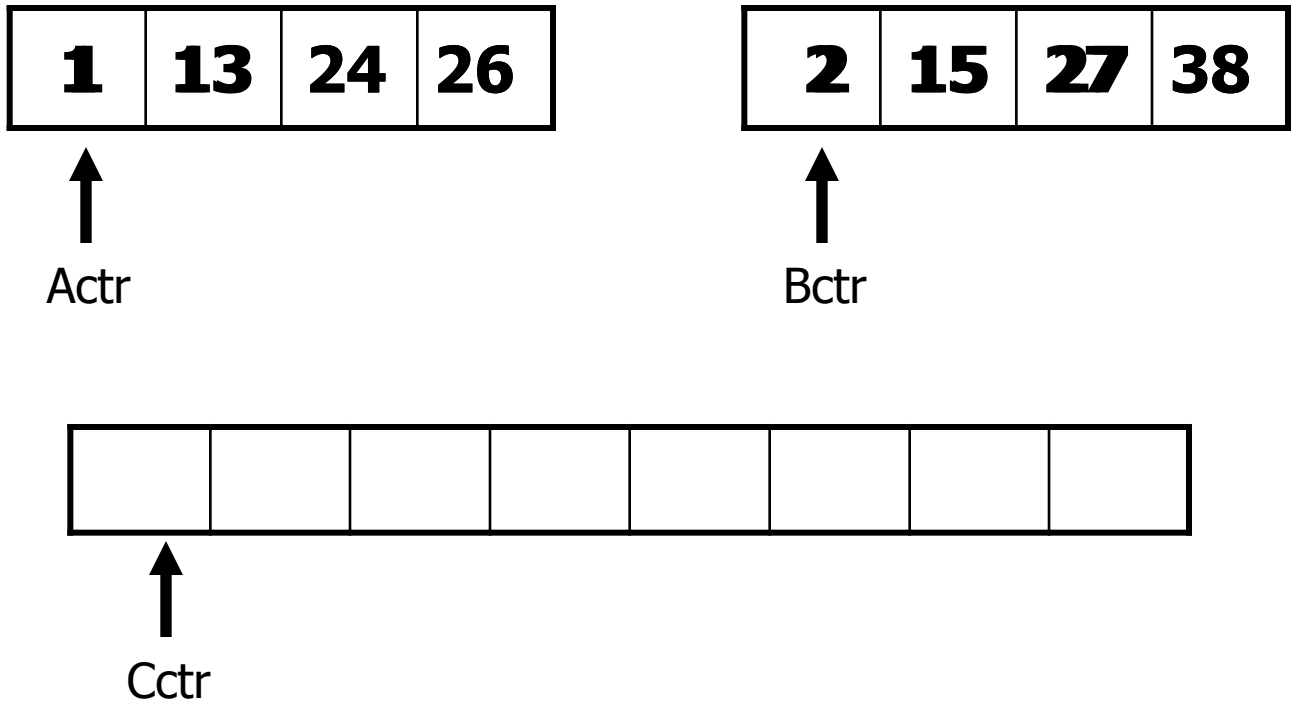
III – Le mergesort

Problématique :

- Le tri est effectué en utilisant la technique de *diviser pour régner*, en divisant le problème en deux sous-problèmes de complexité $\theta(n)$ résolus récursivement, offrant une complexité globale de $\theta(n \log_2 n)$

- L'algorithme de tri Mergesort :
  1. Si le nombre d'éléments à trier == (0 ou 1), retourner.
  2. Trier récursivement la première et deuxième moitié séparément.
  3. Regrouper les deux parties triées dans un nouveau groupe trié.

MergeSort



mergeSort (public)

```
public static <AnyType extends Comparable<? super AnyType>>
void mergeSort( AnyType [ ] a )
{
    //Création d'un tableau temporaire
    AnyType [ ] tmpArray = (AnyType[]) new Comparable[ a.length ];
    //Appel de mergeSort pour trier tout le tableau a
    mergeSort( a, tmpArray, 0, a.length - 1 );
}
```

mergeSort

```
/**
 * Méthode interne effectuant des appels récursifs.
 * @param a : Un tableau d'items de type Comparable.
 * @param tmpArray: Un tableau pour placer le résultat de la fusion.
 * @param left : L'index de l'élément le plus à gauche du sous-tableau à trier.
 * @param right:  L'index de l'élément le plus à droite du sous-tableau à trier.
 */
private static <AnyType extends Comparable<? super AnyType>>
void mergeSort( AnyType [ ] a, AnyType [ ] tmpArray, int left, int right )
{
    if( left < right ) // Le tableau contient au moins 1 élément
    {
        int center = ( left + right ) / 2;
        mergeSort( a, tmpArray, left, center );// Trier la partie gauche
        mergeSort( a, tmpArray, center + 1, right ); // Trier la partie droite
        merge( a, tmpArray, left, center + 1, right ); // Fusionner les deux parties
    }
}
```

merge

```
private static <AnyType extends Comparable<? super AnyType>>
void merge( AnyType [ ] a, AnyType [ ] tmpArray, int leftPos, int rightPos, int rightEnd )
{

   int leftEnd = rightPos - 1; // Extrémité droite de la partie gauche
   int tmpPos = leftPos; // Index utilisé pour mettre le résultat de la fusion dans tmpArray
   int numElements = rightEnd - leftPos + 1; // NB d'éléments à fusionner

   while( leftPos <= leftEnd && rightPos <= rightEnd )
      if( a[ leftPos ].compareTo( a[ rightPos ] ) <= 0 )
         tmpArray[ tmpPos++ ] = a[ leftPos++ ];
      else
         tmpArray[ tmpPos++ ] = a[ rightPos++ ];

   while( leftPos <= leftEnd )    // Copier le reste de la partie gauche
      tmpArray[ tmpPos++ ] = a[ leftPos++ ];

   while( rightPos <= rightEnd )  // Copier le reste de la partie de droite
      tmpArray[ tmpPos++ ] = a[ rightPos++ ];

   // Copier le résultat de la fusion dans la partie correspondante du tableau a
   for( int i = 0; i < numElements; i++, rightEnd-- )
      a[ rightEnd ] = tmpArray[ rightEnd ];
}
```

MergeSort

Preuve de complexité:

$T(N) = 2*T(N/2) + N$ et $T(1)=1$

$T(N)/N = 2*T(N/2)/N + 1$
$T(N)/N = T(N/2)/(N/2) + 1$

Sans perte de généralité, posons $N=2^k$:

$(N/1)=2^k$: $T(N)/N = T(N/2)/(N/2) + 1$
$(N/2)=2^{k-1}$: $T(N/2)/(N/2) = T(N/4)/(N/4) + 1$
$(N/4)=2^{k-2}$: $T(N/4)/(N/4) = T(N/8)/(N/8) + 1$
…
$(N/2^{k-1})=2$: $T(N/2^{k-1})/(N/2^{k-1}) = T(N/2^k)/(N/2^k) + 1 = T(1) + 1$

$T(N)/N = T(1) + k = 1 + k$
$T(N) = N + N \cdot k = N + N \cdot log(N)$

Plan

I. Notions de base
II. Les tris simples
III. Mergesort
IV. Quicksort

IV - Le tri rapide (Quicksort)

Problématique :

- Le tri est une tâche importante en programmation. Il est donc essentiel que celui-ci soit effectué avec efficacité et rapidité.

- L'algorithme de tri Quicksort tente d'atteindre cet objectif par l'utilisation du concept de récursivité.

Le tri rapide (suite)

Idée générale :

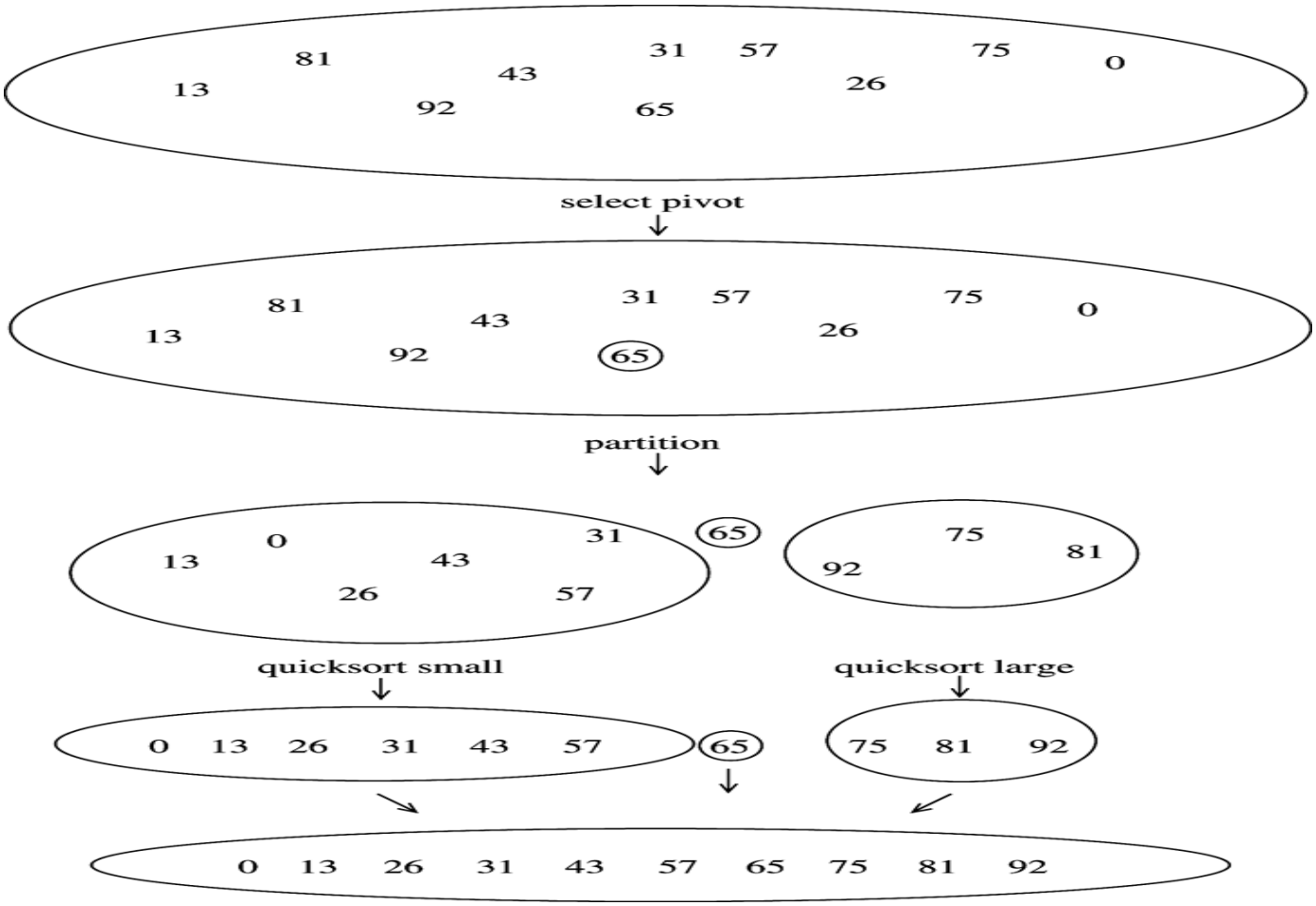
Dans cet algorithme, l'exécution du tri est simplement déléguée à une série d'appels récursifs. Une fonction de partitionnement est d'abord appelée. Cette fonction choisit arbitrairement un pivot qui est utilisé pour subdiviser le tableau en deux parties:

1. une partie de gauche où tous les éléments ont une valeur plus petite ou égale au pivot;
2. une partie de droite où tous les éléments ont une valeur plus grande ou égale au pivot.

Par la suite, les deux parties ainsi obtenues sont triées par deux appels récursifs à la fonction de tri rapide.

En plus de la fonction principale de tri rapide, une autre fonction est donc requise: celle qui procède à la partition du tableau.

Example


81 43 31 57 75 0
13 26
92 65
select pivot
81 43 31 57 75 0
13 26
92 65
partition
0 31 65
13 43
26 57
75 81
92
quicksort small
quicksort large
0 13 26 31 43 57
65
75 81 92
0 13 26 31 43 57 65 75 81 92

QuickSort (public)

```
public static <AnyType extends Comparable<? super AnyType>>
void quicksort( AnyType [ ] a )
{
   //Appel de Quicksort pour trier tout le tableau a
   quicksort( a, 0, a.length - 1 );
}
```

QuickSort

```
private static <AnyType extends Comparable<? super AnyType>>
void quicksort( AnyType [ ] a, int left, int right )
{
   if( left + CUTOFF <= right ) // la taille du sous-tableau mérite l'application du tri quicksort
   {
      AnyType pivot = median3( a, left, right );

      // Début du partitionnement du tableau a en fonction du pivot
      int i = left, j = right - 1; // i à l'extrémité gauche et j à l'extrémité droite (-1)
      for( ; ; )
      {
         //avancer l'indice i vers la droite tant que les éléments à gauche sont < au pivot
         while( a[ ++i ].compareTo( pivot ) < 0 ) { }
         //avancer l'indice j vers la gauche tant que les éléments à droite sont > au pivot
         while( a[ --j ].compareTo( pivot ) > 0 ) { }
         if( i < j ) // il y a deux éléments à inter-changer
            swapReferences( a, i, j ); // Interchanger ces deux éléments
         else
            break; // sortir de la boucle car la partition du tableau est terminée
      }
      swapReferences( a, i, right - 1 );   // Placer le pivot dans la position approprié
      quicksort( a, left, i - 1 );         // Trier les éléments inférieurs au pivot
      quicksort( a, i + 1, right );        // Trier les éléments supérieurs au pivot
   }
   else  // Effectuer un tri par insertion sur cette partie du tableau
      insertionSort( a, left, right );
}
```

QuickSort

```
/**
   * Retourne une médiane de left, center et right.
   * Tri ses trois éléments comme suit:  1: le plus petit dans la position left
   *                                     2: le plus grand dans la position right
   *                                     3: le pivot dans la position right -1
   */
  private static <AnyType extends Comparable<? super AnyType>>
  AnyType median3( AnyType [ ] a, int left, int right )
  {
      int center = ( left + right ) / 2;
      if( a[ center ].compareTo( a[ left ] ) < 0 ) swapReferences( a, left, center );
      if( a[ right ].compareTo( a[ left ] ) < 0 )  swapReferences( a, left, right );
      if( a[ right ].compareTo( a[ center ] ) < 0 ) swapReferences( a, center, right );

      // Place le pivot à la position right - 1
      swapReferences( a, center, right - 1 );
      return a[ right - 1 ];
  }
```

QuickSort

```
int i = left + 1, j = right ;
for( ; ; )
{
    while( a[ i ].compareTo( pivot ) < 0 ) i++;
    while( a[ j ].compareTo( pivot ) > 0 ) j--

    if( i < j )
        swapReferences( a, i, j );
    else
        break;
}
```

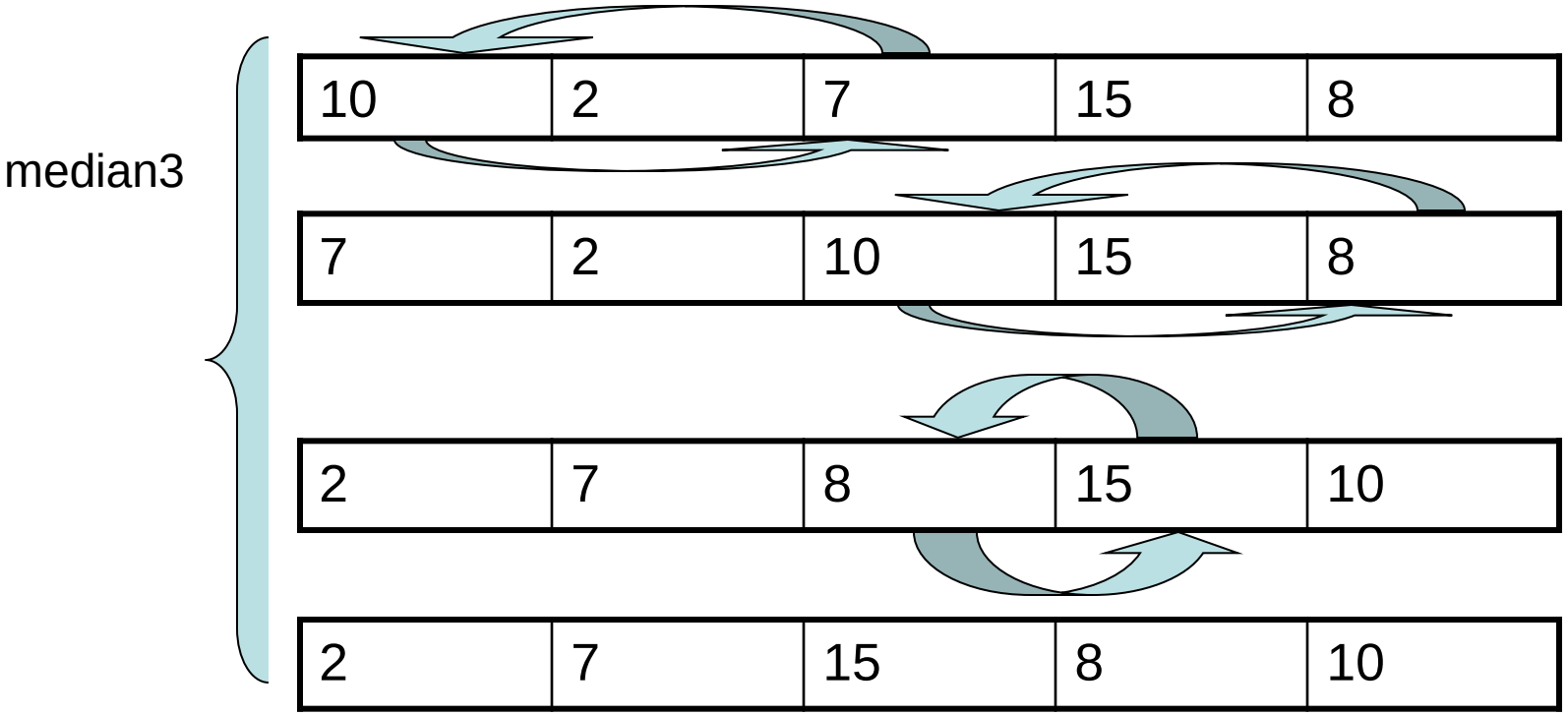
Cette variante de l'algorithme n'est pas bonne:

La boucle sera infinie si a[i] == a[j] == pivot

(essayer de trier a = {3, 3, 3, 3, 3, 3, 3, …, 3}

Exemple d'exécution de median3

Premier appel: Quicksort(a,0,4) center= Tab[2] = 7



Return Tab[4-1] = Tab[3] = 8

Complexité du QuickSort

Cas moyen et meilleur cas

- Complexité du Quicksort dans le cas moyen et dans le meilleur cas .

$$O(n \log_2 n)$$

Pire cas

- Complexité du Quicksort dans le pire cas:

$$O(n^2)$$