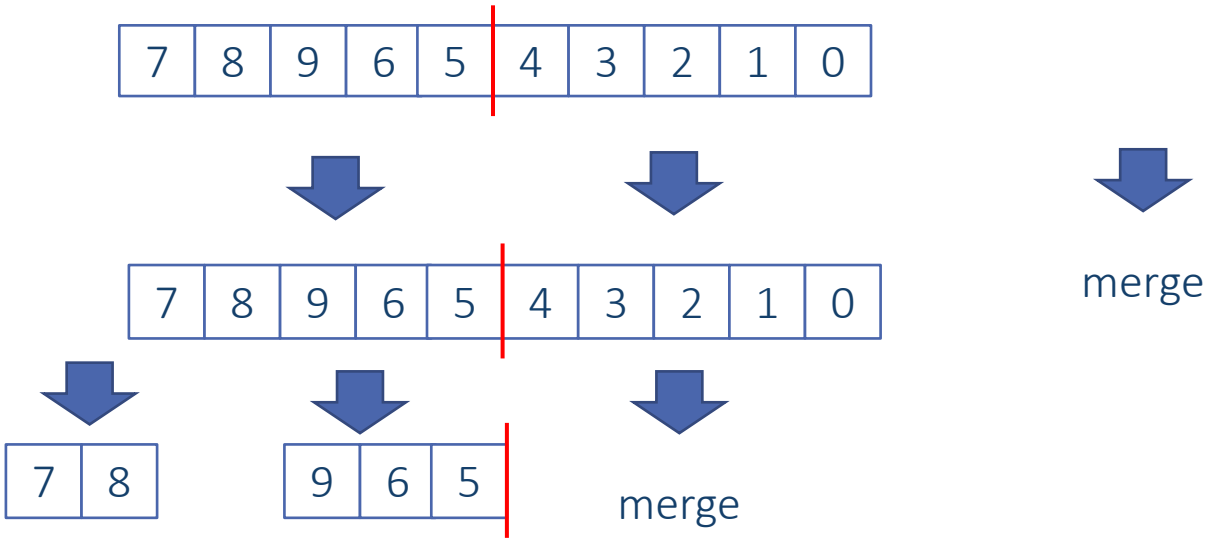
Aula 11 e 12
Ordenação
Mergesort

Algoritmos e Estruturas de Dados

Mergesort

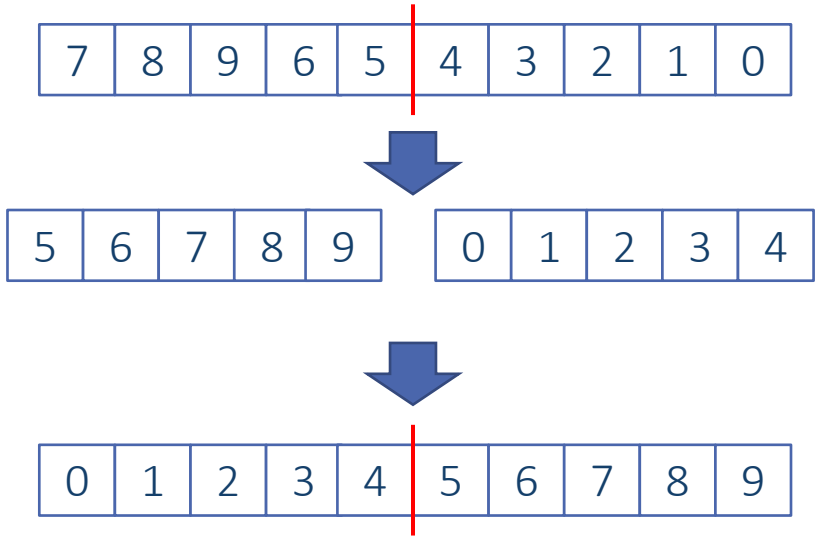
Mergesort

- Ideia:
  - Dividir para conquistar
  - ordenar um array de 1000 elementos é muito difícil!
  - ordenar metade, e depois outra metade, e depois juntar tudo



Mergesort

- Ideia:
  - Dividir para conquistar
  - ordenar um array de 1000 elementos é muito difícil!
  - ordenar metade, e depois outra metade, e depois juntar tudo



Merge

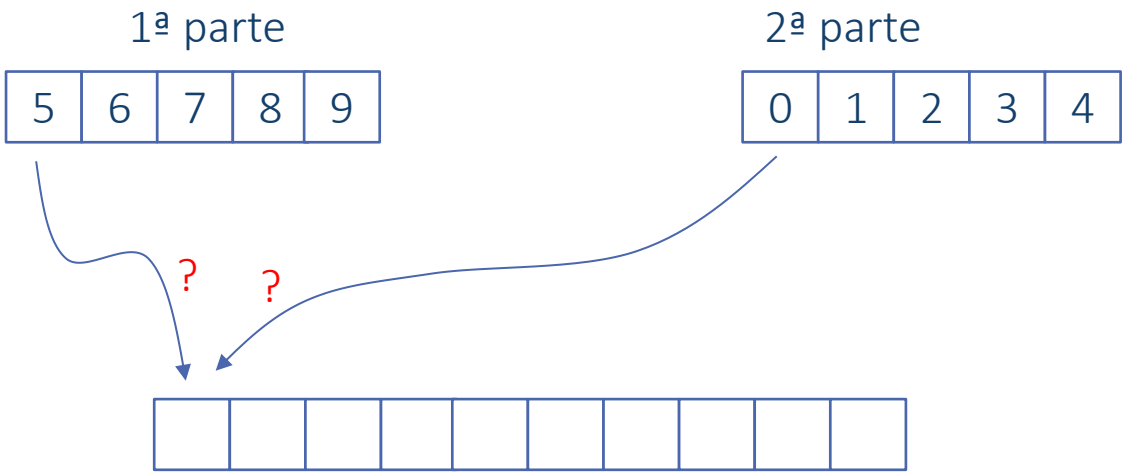
- Ideia
  - Juntar de forma ordenada os elementos de 2 partes do array
  - Sabendo que:

    Os elementos da 1.ª parte estão já ordenados entre si

    Os elementos da 2.ª parte estão já ordenados entre si

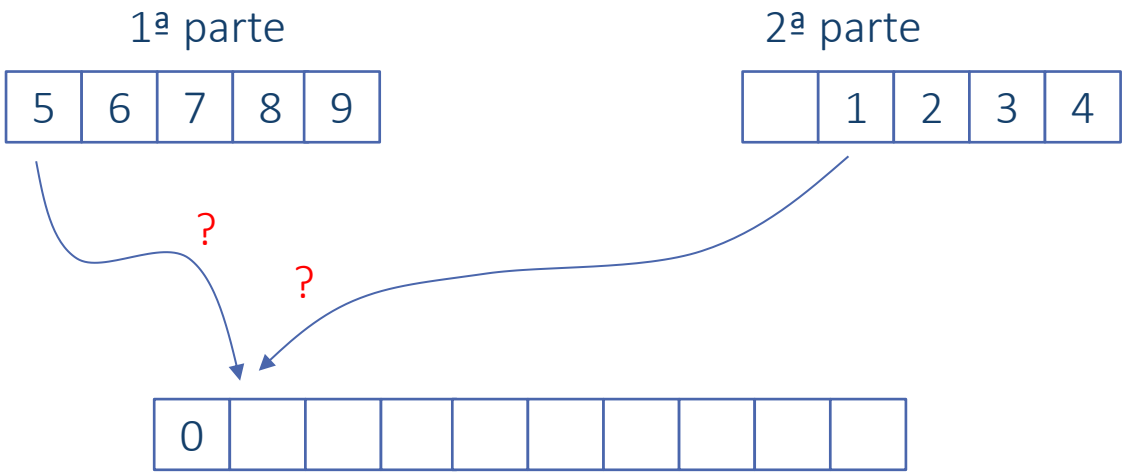
Merge

- Como os subarrays estão ordenados sabemos que o elemento mais à esquerda é sempre o menor de cada subarray
  - Ou seja, basta comparar os elementos mais à esquerda para saber qual dos dois colocar



Merge

- Como os subarrays estão ordenados sabemos que o elemento mais à esquerda é sempre o menor de cada subarray
  - Ou seja, basta comparar os elementos mais à esquerda para saber qual dos dois colocar



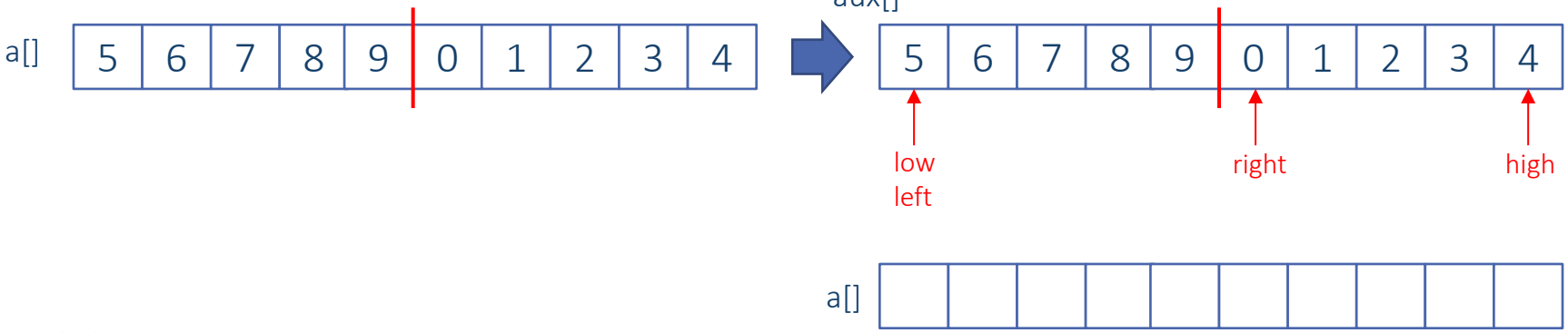
Merge

```
private static void merge(Comparable[] a, Comparable[] aux, int low, int mid, int high)
{
    int left = low;
    int right = mid+1;

    for(int i = low; i <= high; i++)
    {
        aux[i] = a[i];
    }

    for(int i = low; i <= high; i++)
    {
        //left array is exausted
        if (left > mid)                          a[i] = aux[right++];
        //right array is exausted
        else if (right > high)                   a[i] = aux[left++];
        else if (less(aux[right],aux[left]))     a[i] = aux[right++];
        else                                     a[i] = aux[left++];
    }
}
```

Copia os elementos para um array auxiliar

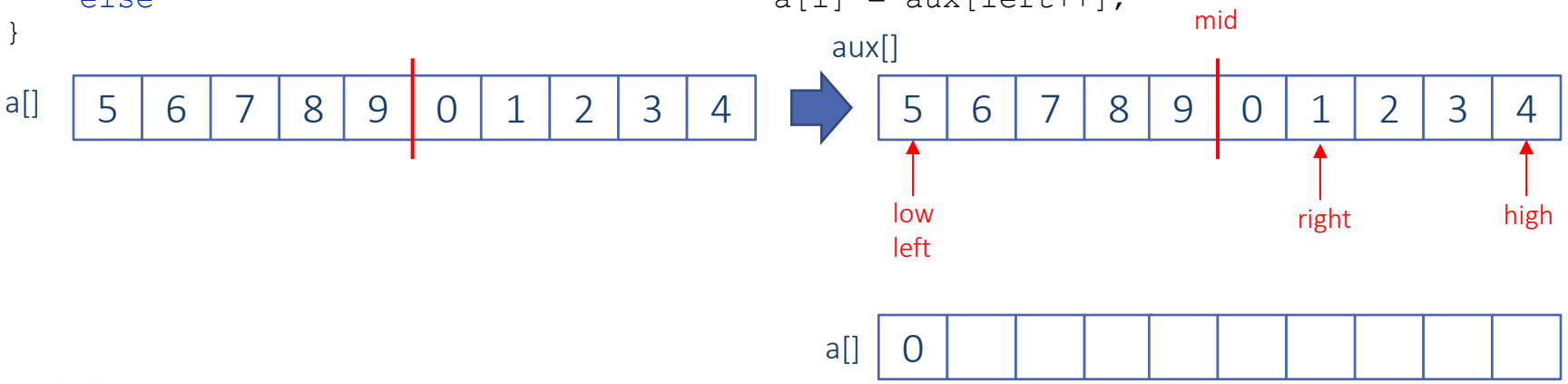


Merge

```
private static void merge(Comparable[] a, Comparable[] aux, int low, int mid, int high)
{
    int left = low;
    int right = mid+1;

    for(int i = low; i <= high; i++)
    {
        aux[i] = a[i];
    }

    for(int i = low; i <= high; i++)
    {
        //left array is exausted
        if (left > mid)                          a[i] = aux[right++];
        //right array is exausted
        else if (right > high)                   a[i] = aux[left++];
        else if (less(aux[right],aux[left]))     a[i] = aux[right++];
        else                                     a[i] = aux[left++];
    }
}
```

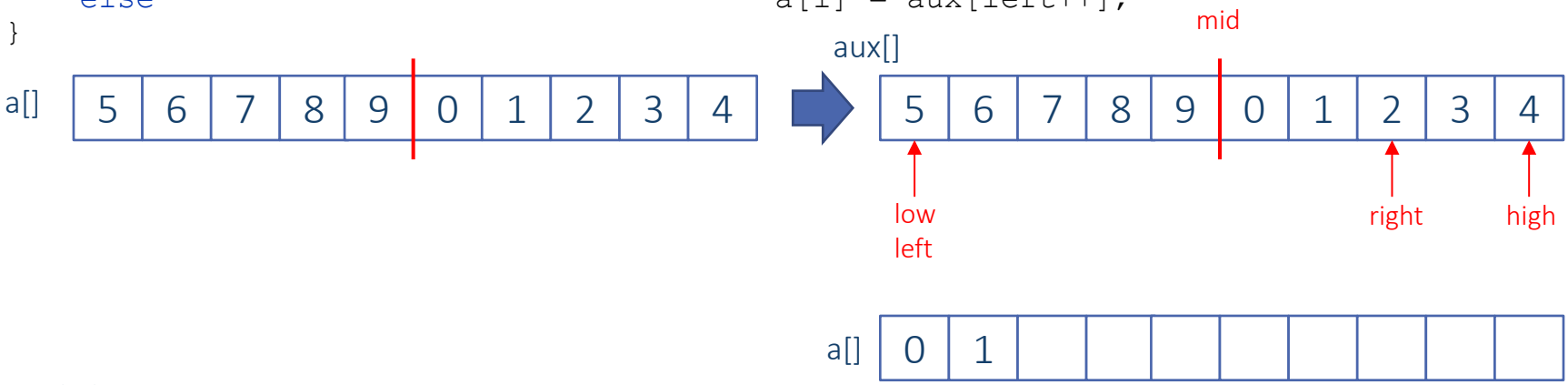


Merge

```
private static void merge(Comparable[] a, Comparable[] aux, int low, int mid, int high)
{
    int left = low;
    int right = mid+1;

    for(int i = low; i <= high; i++)
    {
        aux[i] = a[i];
    }

    for(int i = low; i <= high; i++)
    {
        //left array is exausted
        if (left > mid)                          a[i] = aux[right++];
        //right array is exausted
        else if (right > high)                   a[i] = aux[left++];
        else if (less(aux[right],aux[left]))     a[i] = aux[right++];
        else                                     a[i] = aux[left++];
    }
}
```

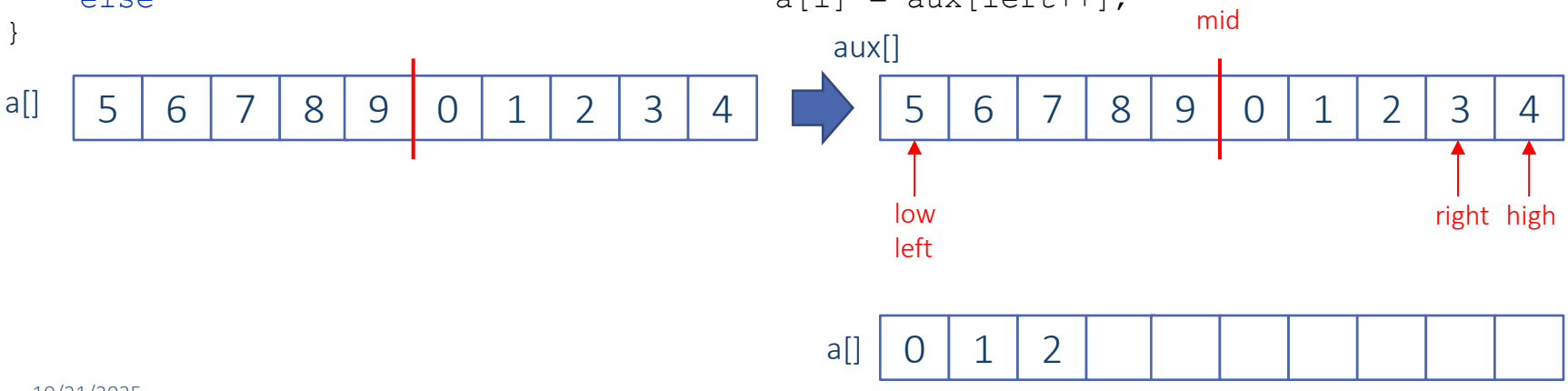


Merge

```
private static void merge(Comparable[] a, Comparable[] aux, int low, int mid, int high)
{
    int left = low;
    int right = mid+1;

    for(int i = low; i <= high; i++)
    {
        aux[i] = a[i];
    }

    for(int i = low; i <= high; i++)
    {
        //left array is exausted
        if (left > mid)                          a[i] = aux[right++];
        //right array is exausted
        else if (right > high)                   a[i] = aux[left++];
        else if (less(aux[right],aux[left]))     a[i] = aux[right++];
        else                                     a[i] = aux[left++];
    }
}
```

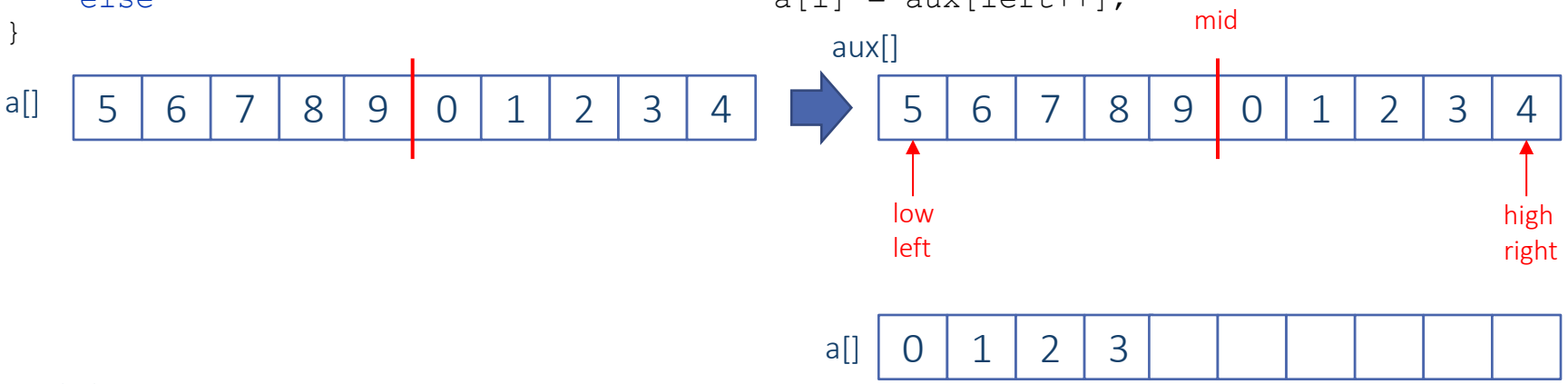


Merge

```
private static void merge(Comparable[] a, Comparable[] aux, int low, int mid, int high)
{
    int left = low;
    int right = mid+1;

    for(int i = low; i <= high; i++)
    {
        aux[i] = a[i];
    }

    for(int i = low; i <= high; i++)
    {
        //left array is exausted
        if (left > mid)                          a[i] = aux[right++];
        //right array is exausted
        else if (right > high)                   a[i] = aux[left++];
        else if (less(aux[right],aux[left]))     a[i] = aux[right++];
        else                                     a[i] = aux[left++];
    }
}
```

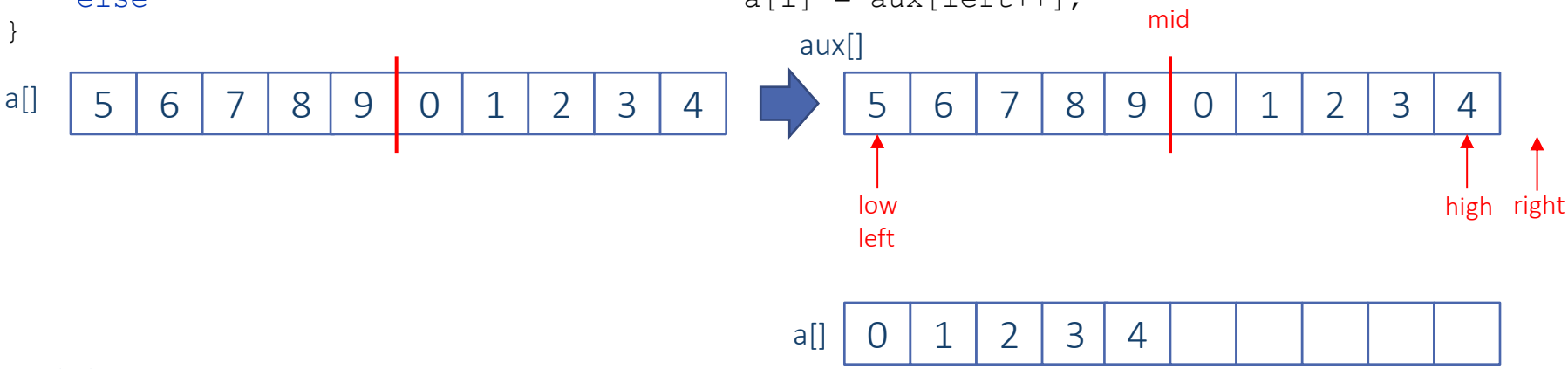


Merge

```
private static void merge(Comparable[] a, Comparable[] aux, int low, int mid, int high)
{
    int left = low;
    int right = mid+1;

    for(int i = low; i <= high; i++)
    {
        aux[i] = a[i];
    }

    for(int i = low; i <= high; i++)
    {
        //left array is exausted
        if (left > mid)                          a[i] = aux[right++];
        //right array is exausted
        else if (right > high)                   a[i] = aux[left++];
        else if (less(aux[right],aux[left]))     a[i] = aux[right++];
        else                                     a[i] = aux[left++];
    }
}
```

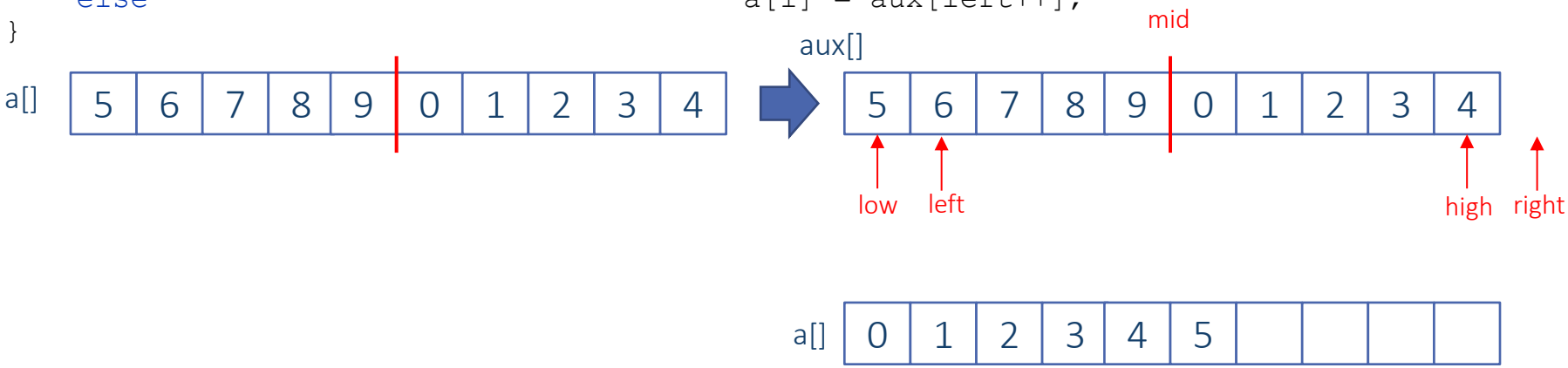


Merge

```
private static void merge(Comparable[] a, Comparable[] aux, int low, int mid, int high)
{
    int left = low;
    int right = mid+1;

    for(int i = low; i <= high; i++)
    {
        aux[i] = a[i];
    }

    for(int i = low; i <= high; i++)
    {
        //left array is exausted
        if (left > mid)                          a[i] = aux[right++];
        //right array is exausted
        else if (right > high)                   a[i] = aux[left++];
        else if (less(aux[right],aux[left]))     a[i] = aux[right++];
        else                                     a[i] = aux[left++];
    }
}
```



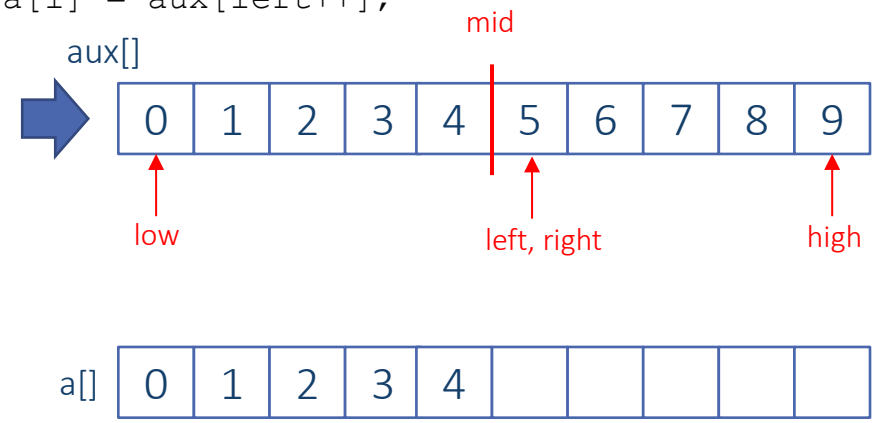
Merge

```
private static void merge(Comparable[] a, Comparable[] aux, int low, int mid, int high)
{
    int left = low;
    int right = mid+1;

    for(int i = low; i <= high; i++)
    {
        aux[i] = a[i];
    }

    for(int i = low; i <= high; i++)
    {
        //left array is exausted
        if (left > mid)                          a[i] = aux[right++];
        //right array is exausted
        else if (right > high)                   a[i] = aux[left++];
        else if (less(aux[right],aux[left]))     a[i] = aux[right++];
        else                                     a[i] = aux[left++];
    }
}
```

Tudo o que estava do lado esquerdo já foi colocado, só nos falta colocar elementos do lado direito



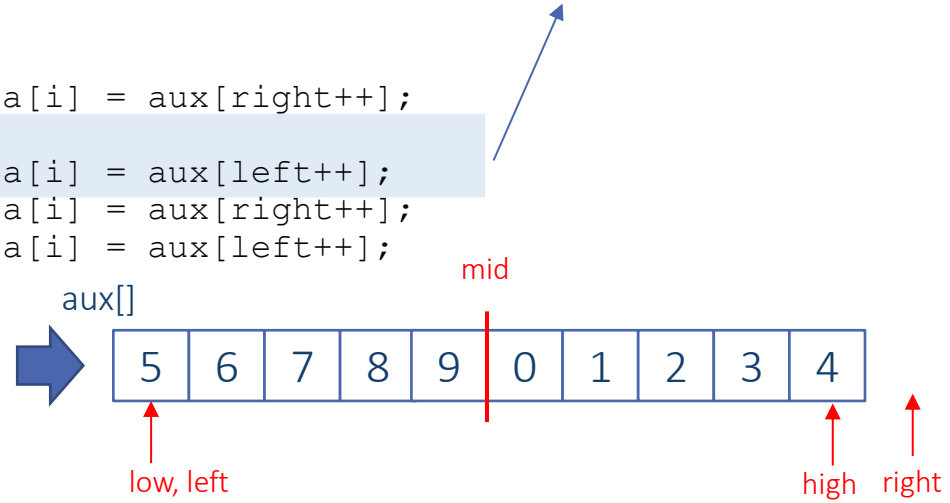
Merge

```
private static void merge(Comparable[] a, Comparable[] aux, int low, int mid, int high)
{
    int left = low;
    int right = mid+1;

    for(int i = low; i <= high; i++)
    {
        aux[i] = a[i];
    }

    for(int i = low; i <= high; i++)
    {
        //left array is exausted
        if (left > mid)                          a[i] = aux[right++];
        //right array is exausted
        else if (right > high)                   a[i] = aux[left++];
        else if (less(aux[right],aux[left]))     a[i] = aux[right++];
        else                                     a[i] = aux[left++];
    }
}
```

Tudo o que estava do lado direito já foi colocado, só nos falta colocar elementos do lado esquerdo



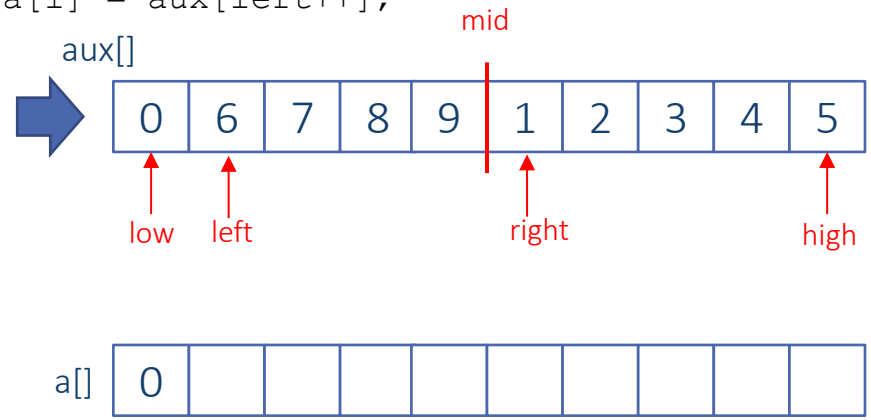
Merge

```
private static void merge(Comparable[] a, Comparable[] aux, int low, int mid, int high)
{
    int left = low;
    int right = mid+1;

    for(int i = low; i <= high; i++)
    {
        aux[i] = a[i];
    }

    for(int i = low; i <= high; i++)
    {
        //left array is exausted
        if (left > mid)                          a[i] = aux[right++];
        //right array is exausted
        else if (right > high)                   a[i] = aux[left++];
        else if (less(aux[right],aux[left]))     a[i] = aux[right++];
        else                                     a[i] = aux[left++];
    }
}
```

Há elementos dos dois lados, temos de comparar o elemento da esquerda com o da direita
Se o da direita for menor, colocamos da direita



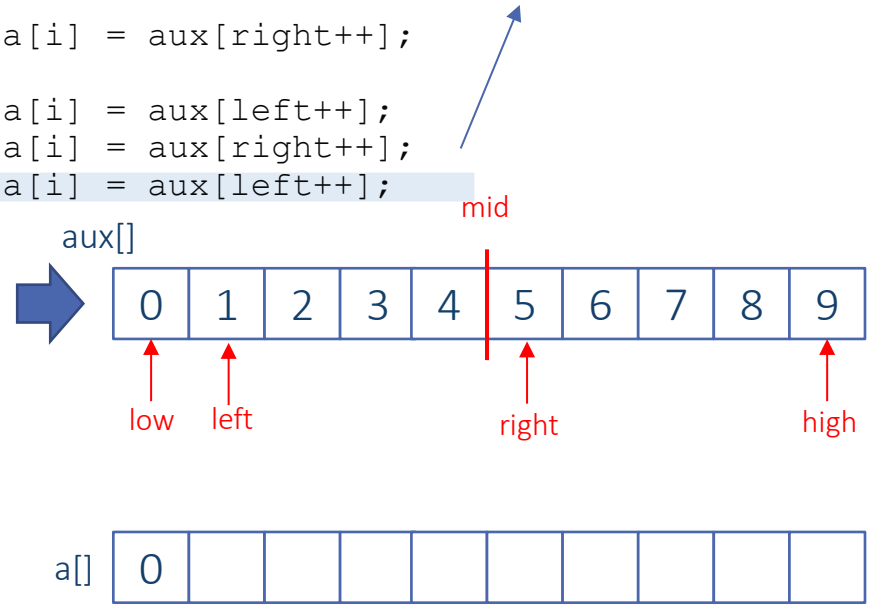
Merge

```
private static void merge(Comparable[] a, Comparable[] aux, int low, int mid, int high)
{
    int left = low;
    int right = mid+1;

    for(int i = low; i <= high; i++)
    {
        aux[i] = a[i];
    }

    for(int i = low; i <= high; i++)
    {
        //left array is exausted
        if (left > mid)                          a[i] = aux[right++];
        //right array is exausted
        else if (right > high)                   a[i] = aux[left++];
        else if (less(aux[right],aux[left]))     a[i] = aux[right++];
        else                                     a[i] = aux[left++];
    }
}
```

Há elementos dos dois lados, temos de comparar o elemento da esquerda com o da direita
Caso contrário, colocamos da esquerda



Mergesort

Recursivo

Mergesort recursivo

Observação:
Abordagens recursivas a um problema tentam obter uma solução para o problema original a partir da solução para uma versão mais simples do problema original

Mergesort recursivo

Observação:
Abordagens recursivas a um problema tentam obter uma solução para o problema original a partir da solução para uma versão mais simples do problema original

- 1) "Parto" o array original em 2 subarrays com metade do tamanho
- 2) Ordeno o subarray da esquerda
- 3) Ordeno o subarray da direita
- 4) Faço merge dos 2 subarrays

Mergesort recursivo

```
public static void sort(Comparable[] a)
{
    Comparable[] aux = new Comparable[a.length];
    sort(a, aux, 0, a.length-1);
}

public static void sort(Comparable[] a, Comparable[] aux, int low,
int high)
{
    if(high <= low) return;
    int mid = low + (high - low)/2;
    sort(a, aux, low, mid);
    sort(a, aux, mid+1,high);
    merge(a, aux, low, mid, high);
}
```

Complexidade

do Mergesort

Complexidade merge (exchanges)

```
private static void merge(Comparable[] a, Comparable[] aux, int low, int mid, int high)
{
    int left = low;
    int right = mid+1;

    for(int i = low; i <= high; i++)
    {
        aux[i] = a[i];
    }

    for(int i = low; i <= high; i++)
    {
        //left array is exausted
        if (left > mid)                          a[i] = aux[right++];
        //right array is exausted
        else if (right > high)                   a[i] = aux[left++];
        else if (less(aux[right],aux[left]))     a[i] = aux[right++];
        else                                     a[i] = aux[left++];
    }
}
```

Observação:
dado n = high – low +1

n exchanges

+

n exchanges

Só é feito 1 exchance por iteração

=

~2n exchanges

Complexidade merge (compares)

```
private static void merge(Comparable[] a, Comparable[] aux, int low, int mid, int high)
{
    int left = low;
    int right = mid+1;

    for(int i = low; i <= high; i++)
    {
        aux[i] = a[i];
    }

    for(int i = low; i <= high; i++)
    {
        //left array is exausted
        if (left > mid)                          a[i] = aux[right++];
        //right array is exausted
        else if (right > high)                   a[i] = aux[left++];
        else if (less(aux[right],aux[left]))     a[i] = aux[right++];
        else                                     a[i] = aux[left++];
    }
}
```

Observação:
dado n = high – low +1

n-1 compares no pior caso

~n

Complexidade merge (compares)

```
private static void merge(Comparable[] a, Comparable[] aux, int low, int mid, int high)
{
    int left = low;
    int right = mid+1;

    for(int i = low; i <= high; i++)
    {
        aux[i] = a[i];
    }

    for(int i = low; i <= high; i++)
    {
        //left array is exausted
        if (left > mid)                          a[i] = aux[right++];
        //right array is exausted
        else if (right > high)                   a[i] = aux[left++];
        else if (less(aux[right],aux[left]))     a[i] = aux[right++];
        else                                     a[i] = aux[left++];
    }
}
```

Observação:
dado n = high – low +1

n/2 compares no mehor caso

~n/2

Observação:
No melhor caso, o merge apenas vai comparar metade dos elementos

Primeiro gastamos todos os elementos de um dos lados, e já não precisamos de fazer mais compares pq (left > mid) ou (right > high)

Complexidade merge

função merge	Melhor caso	Pior caso	Aleatório	O
less/compare	½ *n*	*n*	*n*	O(*n*)
exchange	*2 n*	*2 n*	*2 n*	

Complexidade Sort (exchanges)

- $T_{sort}(n) = T_{sort}(n/2) + T_{sort}(n/2) + T_{merge}(n)$

sort metade esquerda; *sort* metade direita; *merge*

```
public static void sort(Comparable[] a,
 Comparable[] aux, int low, int high)
{
    if(high <= low) return;
    int mid = low + (high - low)/2;
    sort(a, aux, low, mid);
    sort(a, aux, mid+1,high);
    merge(a, aux, low, mid, high);
}
```

Complexidade Sort (exchanges)

- $T_{sort}(n)$ — *sort* metade esquerda: $T_{sort}(n/2)$; *sort* metade direita: $T_{sort}(n/2)$; *merge*: $T_{merge}(n)$

  $= T_{sort}(n/2) + T_{sort}(n/2) + T_{merge}(n)$

  $= 2T_{sort}(n/2) + 2n$

```
public static void sort(Comparable[] a,
 Comparable[] aux, int low, int high)
{
    if(high <= low) return;
    int mid = low + (high - low)/2;
    sort(a, aux, low, mid);
    sort(a, aux, mid+1,high);
    merge(a, aux, low, mid, high);
}
```

Complexidade Sort (exchanges)

- $T_{sort}(n)$ — *sort* metade esquerda; *sort* metade direita; *merge*

$= T_{sort}(n/2) + T_{sort}(n/2) + T_{merge}(n)$

$= 2T_{sort}(n/2) + 2n$

$= 2(2T_{sort}(n/4) + T_{merge}(n/2)) + 2n$

$= 4T_{sort}(n/4) + 2n + 2n$

$= 8T_{sort}(n/8) + 2n + 2n + 2n$

...

$= \underbrace{2n + \ldots + 2n}_{\log_2 n}$

$= 2n \log_2 n$

```
public static void sort(Comparable[] a,
 Comparable[] aux, int low, int high)
{
    if(high <= low) return;
    int mid = low + (high - low)/2;
    sort(a, aux, low, mid);
    sort(a, aux, mid+1,high);
    merge(a, aux, low, mid, high);
}
```

Complexidade Sort (compares)

- $T_{sort}(n) = T_{sort}(n/2) + T_{sort}(n/2) + T_{merge}(n)$

sort metade esquerda; *sort* metade direita; *merge*

```
public static void sort(Comparable[] a,
 Comparable[] aux, int low, int high)
{
    if(high <= low) return;
    int mid = low + (high - low)/2;
    sort(a, aux, low, mid);
    sort(a, aux, mid+1,high);
    merge(a, aux, low, mid, high);
}
```

Complexidade Sort (compares)

- $T_{sort}(n)$ — *sort* metade esquerda; *sort* metade direita; *merge*

$$= T_{sort}(n/2) + T_{sort}(n/2) + T_{merge}(n)$$

$$= 2T_{sort}(n/2) + n$$

```
public static void sort(Comparable[] a,
 Comparable[] aux, int low, int high)
{
    if(high <= low) return;
    int mid = low + (high - low)/2;
    sort(a, aux, low, mid);
    sort(a, aux, mid+1,high);
    merge(a, aux, low, mid, high);
}
```

No caso médio, e no pior caso o método merge faz ~n comparações

Complexidade Sort (compares)

- $T_{sort}(n)$ (*sort* metade esquerda; *sort* metade direita; *merge*)
  $= T_{sort}(n/2) + T_{sort}(n/2) + T_{merge}(n)$
  $= 2T_{sort}(n/2) + n$
  $= 2(2T_{sort}(n/4) + T_{merge}(n/2)) + n$
  $= 4T_{sort}(n/4) + n + 2$
  $= 8T_{sort}(n/8) + n + n + n$
  ...
  $= \underbrace{n + \ldots + n}_{\log_2 n}$
  $= n \log_2 n$

```
public static void sort(Comparable[] a,
 Comparable[] aux, int low, int high)
{
    if(high <= low) return;
    int mid = low + (high - low)/2;
    sort(a, aux, low, mid);
    sort(a, aux, mid+1,high);
    merge(a, aux, low, mid, high);
}
```

Complexidade Mergesort

```
public static void sort(Comparable[] a,
 Comparable[] aux, int low, int high)
{
    if(high <= low) return;
    int mid = low + (high - low)/2;
    sort(a, aux, low, mid);
    sort(a, aux, mid+1,high);
    merge(a, aux, low, mid, high);
}
```

No melhor caso, o merge apenas vai comparar metade dos elementos

Mergesort	Melhor caso	Pior caso	Aleatório	O
less/compare	$\frac{1}{2} n \log_2 n$	$n \log_2 n$	$n \log_2 n$	$O(n \log_2 n)$
exchange	$2 n \log_2 n$	$2 n \log_2 n$	$2 n \log_2 n$	

Mergesort

Truques para melhorar a eficiência

Truques para melhorar *mergesort*

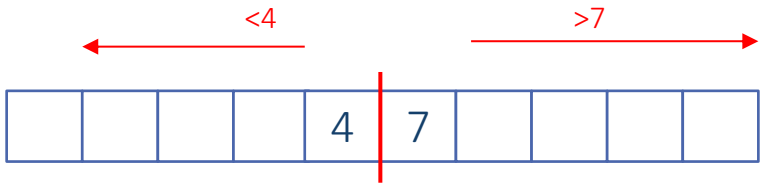
- 1) usar insertion sort para arrays pequenos
  - A partir de um valor de cutoff (ex: 10) usar *insertionsort* em vez de *mergesort*

```
public static void sort(Comparable[] a, Comparable[] aux, int low, int high)
{
    if(high <= low + CUTOFF - 1)
    {
        InsertionSort.sort(a,low, high);
        return;
    }
    int mid = low + (high - low)/2;
    sort(a, aux, low, mid);
    sort(a, aux, mid+1,high);
    merge(a, aux, low, mid, high);
}
```

Truques para melhorar *mergesort*

- *2) Não fazer merge para arrays que já estão ordenados*
  - *Se o maior do lado esquerdo for menor ou igual que o menor direito*

```
public static void sort(Comparable[] a, Comparable[] aux, int low, int high)
{
    if(high <= low) return;
    int mid = low + (high - low)/2;
    sort(a, aux, low, mid);
    sort(a, aux, mid+1,high);
    if(greater(a[mid], a[mid+1]))
        merge(a, aux, low, mid, high);
}
```



Truques para melhorar *mergesort*

- *3) Evitar cópia para array auxiliar*
  - *É possível evitar cópia se*

    alternarmos entre array original e array auxiliar

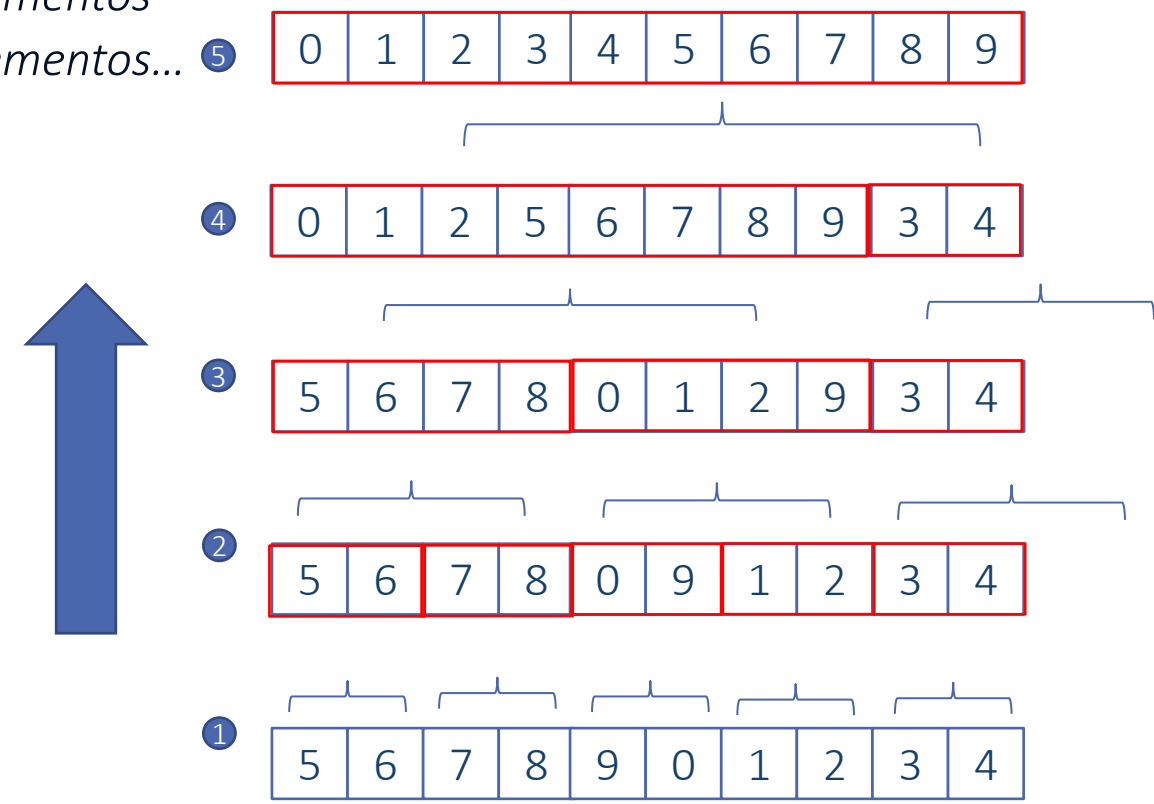
    nas chamadas recursivas

Mergesort

Bottom Up

Mergesort Bottom Up

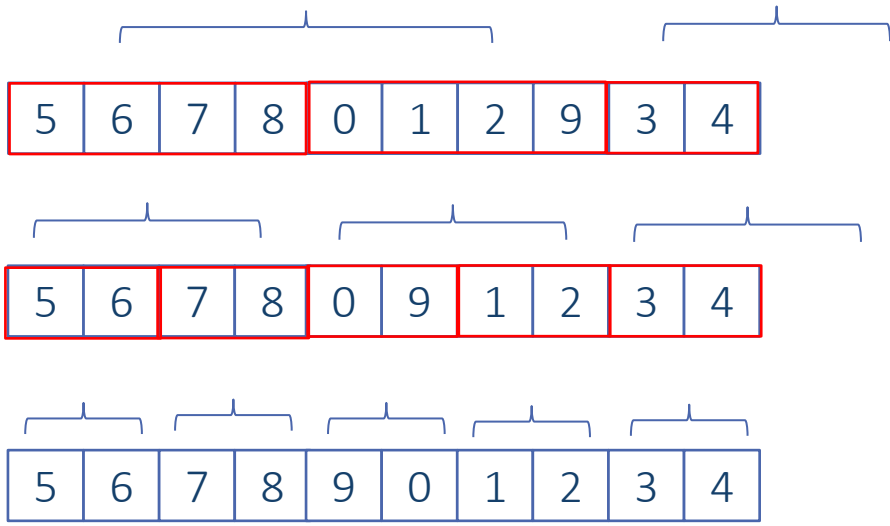
- Implementação não recursiva do *mergesort*
  - *Merge de subarrays de 1 elemento*
  - *Merge de subarrays de 2 elementos*
  - *Merge de subarrays de 4 elementos…*



Mergesort Bottom Up

```
public static void sort(Comparable[] a)
{
    int n = a.length;
    Comparable[] aux = new Comparable[n];

    for(int groupSize = 1; groupSize < n; groupSize *= 2)
    {
        for(int low = 0; low < n - groupSize; low += 2*groupSize)
        {
            merge(a, aux, low, low+groupSize-1, Math.min(low+2*groupSize-1,n-1));
        }
    }
}
```



Complexidade Temporal

```
public static void sort(Comparable[] a)
{
    int n = a.length;
    Comparable[] aux = new Comparable[n];

    for(int groupSize = 1; groupSize < n; groupSize *= 2)
    {
        for(int low = 0; low < n - groupSize; low += 2*groupSize)
        {
            merge(a, aux, low, low+groupSize-1, Math.min(low+2*groupSize-1,n-1));
        }
    }
}
```

$O(\log_2 n)$

$n/2;n/4;n/8;\cdots$

$2;4;8;\cdots$

$O(n)$

Mergesort BU	Melhor caso	Pior caso	Aleatório	O
less/compare	$\frac{1}{2}\, n \log_2 n$	$n \log_2 n$	$n \log_2 n$	$O(n \log_2 n)$
exchange	$2\, n \log_2 n$	$2\, n \log_2 n$	$2\, n \log_2 n$	

Complexidade Temporal

- Quando *n* é uma potência de 2
  - Eficiência *Mergesort Bottom Up* ~ *Mergesort recursivo*

- Quando *n* não é potência de 2
  - *Mergesort Bottom Up* é ligeiramente mais lento *~10%*