

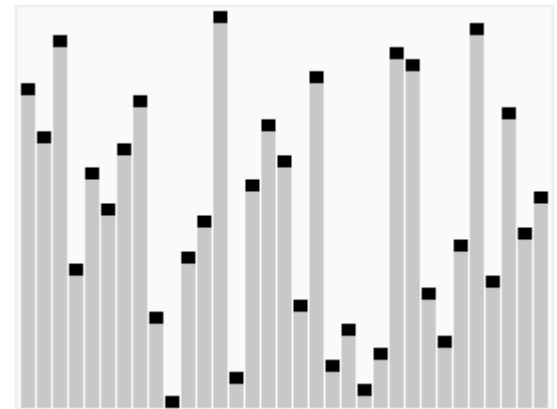
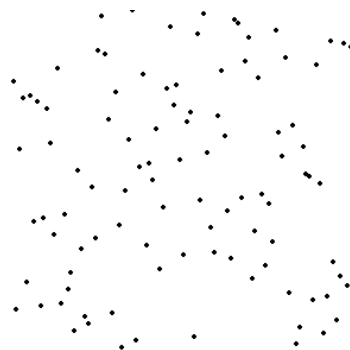
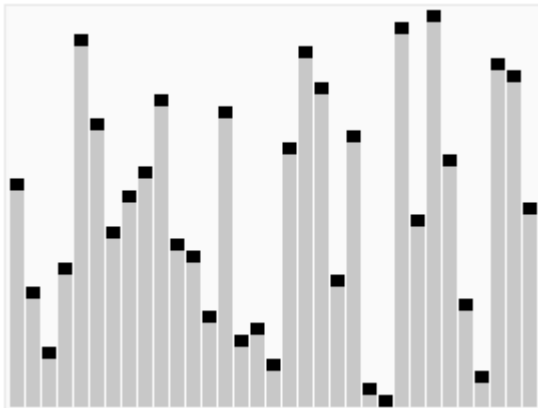
ALGORITMOS E ESTRUTURAS DE DADOS 2018/2019 ALGORITMOS DE ORDENAÇÃO

Armanda Rodrigues

27 de Novembro de 2018

Ordenação (Sorting)

- Pretende-se possibilitar a ordenação dos livros disponíveis na Biblioteca por diversas formas.
- A necessidade destas ordenações é temporária, pretende-se apenas verificar certas características das publicações
- Pretende-se assim criar vetores de Livros ordenados por:
 - Título
 - Tema
- Deverá ser possível alterar a forma como a ordenação é organizada, assim como o algoritmo utilizado para a realizar, quando necessário.



Ordenação

- Dada uma sequência de registos $R_1, R_2, \dots, R_n$, com as chaves $k_1, k_2, \dots, k_n$, pretende-se uma permutação dos registos $R_{i1}, R_{i2}, \dots, R_{in}$ tal que $k_{i1} \leq k_{i2} \leq \dots \leq k_{in}$.
- Um algoritmo de ordenação diz-se estável se preserva a ordem original dos registos com a mesma chave.
- **Exemplo**
 - (3, “Joana”), (7, “Antonio”), (3, “Francisco”), (5, “Teresa”), (5, “Ana”)
- **Permutação estável**
 - (3, “Joana”), (3, “Francisco”), (5, “Teresa”), (5, “Ana”), (7, “Antonio”)

Interface Comparador de Elementos do Tipo E

```
package dataStructures;
import java.io.Serializable;
public interface Comparator<E> extends Serializable {

    // Compares its two arguments for order.
    // Returns a negative integer, zero or a positive integer
    // as the first argument is less than, equal to, or greater
    // than the second.
    int compare( E element1, E element2 );
}
```

Classe de Algoritmos de Ordenação

```
package dataStructures;

public class Sorting {

    public static <E> void xSort( E[] vec, int vecSize, Comparator<E> c ){

        ..... c.compare(vec[i], vec[j]) .....
    }

    public static <E> void ySort( E[] vec, int vecSize, Comparator<E> c ){

        ..... c.compare(vec[i], vec[j]) .....
    }
    .....
}
```

Comparador de Livro por Título

```
import dataStructures.Comparator;

class BookComparatorTitle implements Comparator<Book> {

    static final long serialVersionUID = 0L;

    public int compare( Book book1, Book book2 ) {

        return book1.getTitle().compareTo( book2.getTitle() );

    }

}
```

Ordenações por Título e por Assunto (1)

```
import dataStructures.Sorting;
import dataStructures.Comparator;

class SortingExample {

    private static Comparator<Book> cTitle =
        new BookComparatorTitle();
    private static Comparator<Book> cSubject =
        new Comparator<Book>() {

        static final long serialVersionUID = 0L;

        public int compare( Book book1, Book book2 ){

            return book1.getSubject().compareTo( book2.getSubject() );
        }

    }
}
```

Ordenações por Título e por Assunto (2)

```
private Book[] books;  
private int bookCount;  
  
public void sortByTitle( ) {  
    Sorting.heapSort(books, bookCount, cTitle);  
}  
  
public void sortBySubject( ){  
    Sorting.quickSort(books, bookCount, cSubject);  
}  
  
.....  
}
```

Insertion Sort

Zona azul: zona ordenada



99	32	71	45	50
99	32	71	45	50
32	99	71	45	50
32	99	71	45	50
32	71	99	45	50
32	71	99	45	50
32	45	71	99	50

32	45	71	99	50
32	45	50	71	99

Vetor Ordenado!
Algoritmo Estável

Complexidade
Pior caso: $O(n^2)$
Melhor Caso: $O(n)$
Caso Esperado: $O(n^2)$

Implementação Insertion Sort

```
public static <E> void insertionSort( E[] vec, int vecSize,  
                                     Comparator<E> c ){
```

```
    for ( int pos = 1; pos < vecSize; pos++ ) {
```

Primeira posição não ordenada

```
        E element = vec[pos];
```

```
        int hole = pos;
```

```
        while ( hole > 0 && c.compare(element, vec[hole - 1]) < 0 ) {
```

```
            vec[hole] = vec[hole - 1];
```

```
            hole--;
```

```
        }
```

```
        vec[hole] = element;
```

```
    }
```

As trocas são feitas na zona já ordenada, se necessário

```
}
```

Complexidade – Quantas comparações?

- No pior caso (quando a sequência está ordenada por ordem inversa da pretendida)

$$\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} = O(n^2)$$

- No melhor caso (quando a sequência está ordenada pela ordem pretendida)

$$\sum_{i=1}^{n-1} 1 = n - 1 = O(n)$$

- No caso esperado, a chave na posição j é menor do que metade dos elementos no vetor entre $[0, \dots, j-1]$ e maior que a outra metade. Será necessário, em média, verificar metade das posições para decidir onde posicionar o elemento.
 - Isto determina que o tempo de execução continua a ser quadrático.

Implementação Bubble Sort

```
public static <E> void bubbleSort( E[] vec, int vecSize,  
                                   Comparator<E> c ) {  
    for ( int i = vecSize - 1; i > 0; i-- )  
        for ( int j = 0; j < i; j++ )  
            if ( c.compare(vec[j], vec[j + 1]) > 0 )  
                swapElements(vec, j, j + 1);  
}
```

Todas as posições até à i serão avaliadas relativamente à sua chave e as trocas serão feitas se necessário.

A posição i será a posição do elemento com a maior chave (depois da 1ª iteração)

Bubble Sort

Através de trocas dois a dois o maior valor é levado para a ultima posição a considerar

99	32	71	45	50
32	71	45	50	99
32	71	45	50	99
32	45	50	71	99
32	45	50	71	99
32	45	50	71	99
32	45	50	71	99
32	45	50	71	99

32	45	50	71	99
----	----	----	----	----

Vetor Ordenado!
Algoritmo Estável

Complexidade

Pior caso: $O(n^2)$

Melhor Caso: $O(n^2)$

Caso Esperado: $O(n^2)$

Implementação Bubble Sort

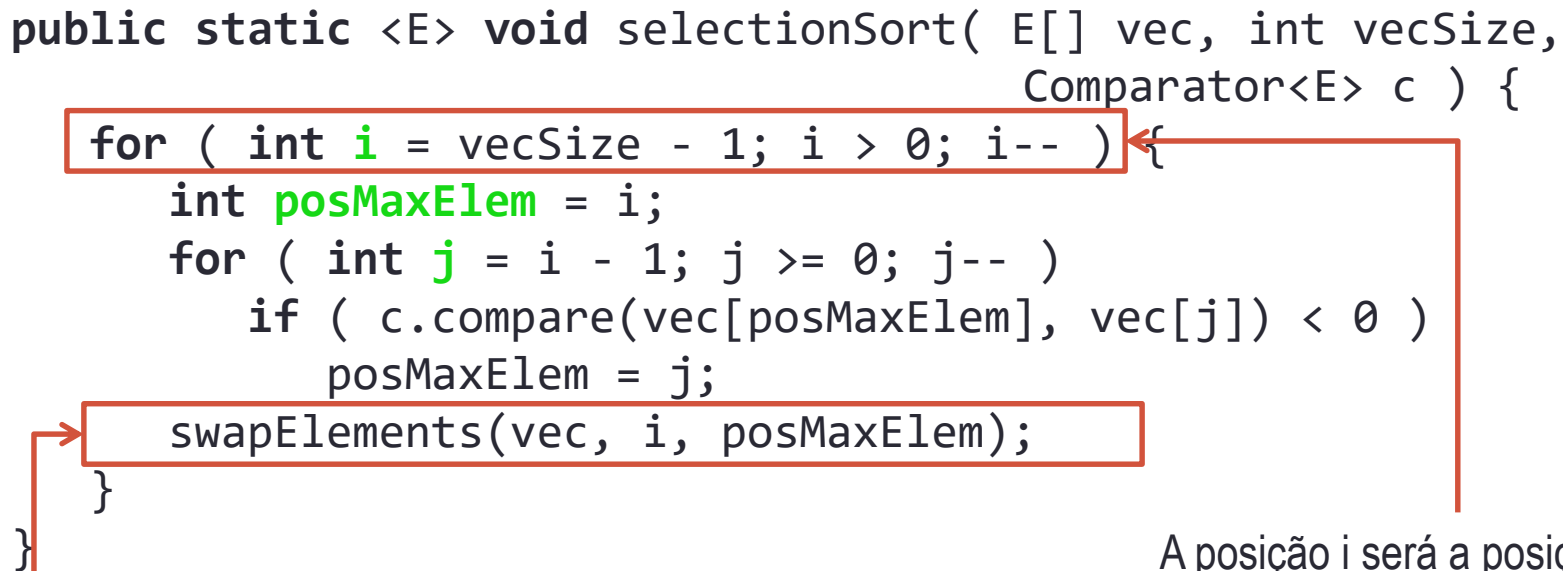
```
public static <E> void bubbleSort( E[] vec, int vecSize,
                                   Comparator<E> c ) {
    for ( int i = vecSize - 1; i > 0; i-- )
        for ( int j = 0; j < i; j++ )
            if ( c.compare(vec[j], vec[j + 1]) > 0 )
                swapElements(vec, j, j + 1);
}
```

Quantas comparações ?

Em todos os casos... $\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} = O(n^2)$

Implementação Selection Sort

```
public static <E> void selectionSort( E[] vec, int vecSize,
                                     Comparator<E> c ) {
    for ( int i = vecSize - 1; i > 0; i-- ) {
        int posMaxElem = i;
        for ( int j = i - 1; j >= 0; j-- )
            if ( c.compare(vec[posMaxElem], vec[j]) < 0 )
                posMaxElem = j;
        swapElements(vec, i, posMaxElem);
    }
}
```



A posição i será a posição do elemento com a maior chave (depois da 1ª iteração)

Todas as posições até à i serão avaliadas relativamente à sua chave, mas é feita uma única troca.

Selection Sort

A maior chave corrente é trocada com o elemento na última posição a ser considerada

99	32	71	45	50
50	32	71	45	99
50	32	71	45	99
50	32	45	71	99
50	32	45	71	99
45	32	50	71	99
45	32	50	71	99
32	45	50	71	99

32	45	50	71	99
----	----	----	----	----

Vetor Ordenado!
Algoritmo Não Estável

Complexidade
Pior caso: $O(n^2)$
Melhor Caso: $O(n^2)$
Caso Esperado: $O(n^2)$

Implementação Selection Sort

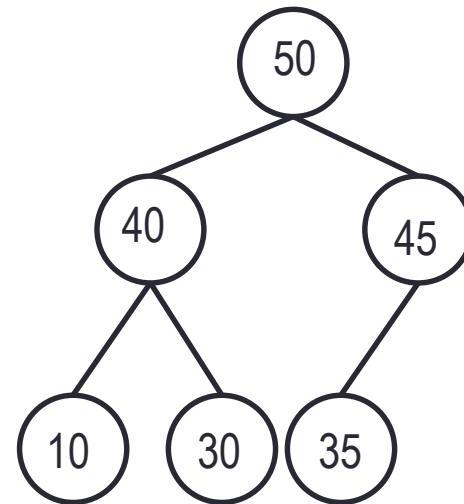
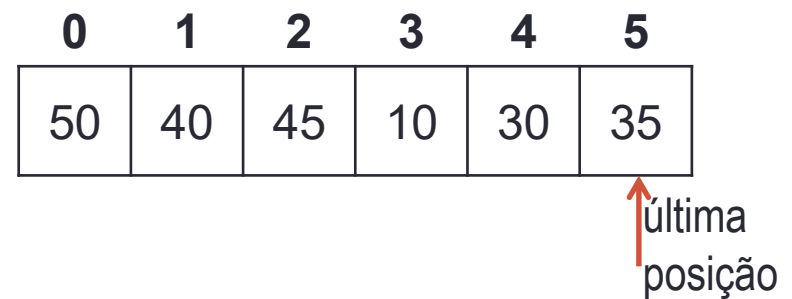
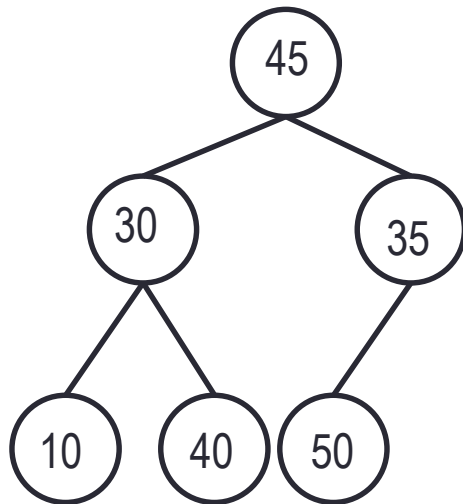
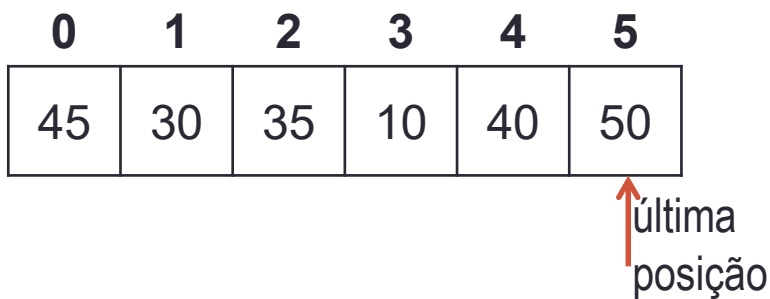
```
public static <E> void selectionSort( E[] vec, int vecSize,
                                     Comparator<E> c ) {
    for ( int i = vecSize - 1; i > 0; i-- ) {
        int posMaxElem = i;
        for ( int j = i - 1; j >= 0; j-- )
            if ( c.compare(vec[posMaxElem], vec[j]) < 0 )
                posMaxElem = j;
        swapElements(vec, i, posMaxElem);
    }
}
```

Quantas comparações ?

Em todos os casos... $\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} = O(n^2)$

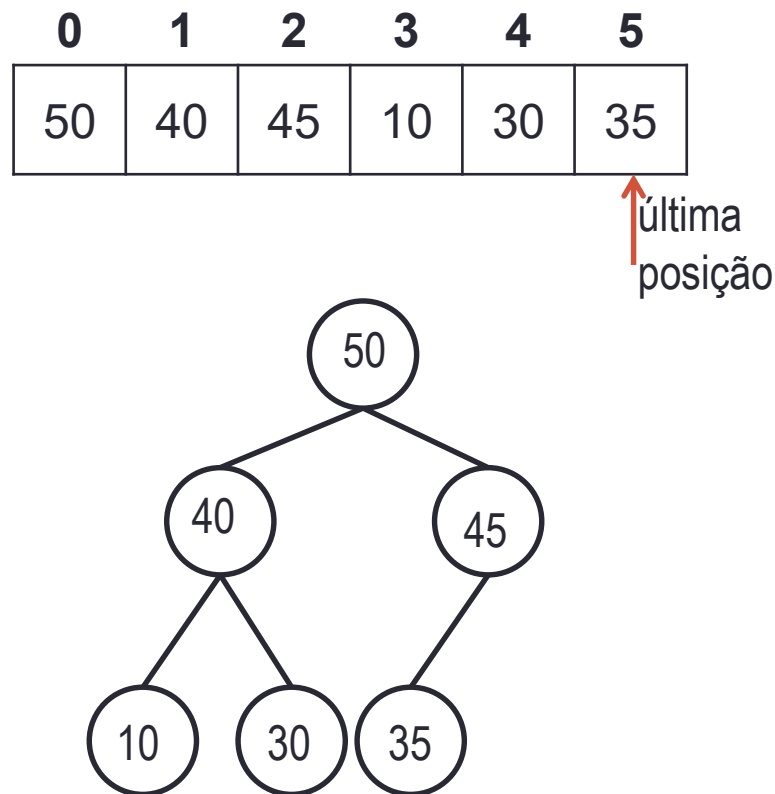
Heap Sort (“Selection Sort inteligente”) (1)

Cria-se um Heap (organizado por máximos) a partir do vetor



Heap Sort (2)

- O algoritmo é constituído por:
 1. Remover o máximo do Heap: Trocar o elemento na 1ª posição pelo último; decrementar o último.
 2. Executar borbulhar [0.. última posição]

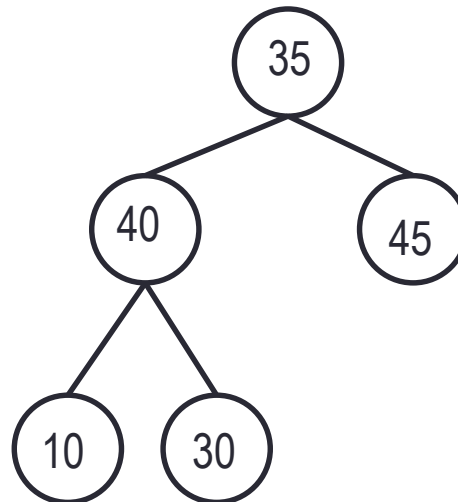


Heap Sort (3)

- O algoritmo é constituído por:
 1. Remover o máximo do Heap: Trocar o elemento na 1ª posição pelo último; decrementar o último.
 2. Executar borbulhar [0.. última posição]

0	1	2	3	4	5
35	40	45	10	30	50

↑ última
posição

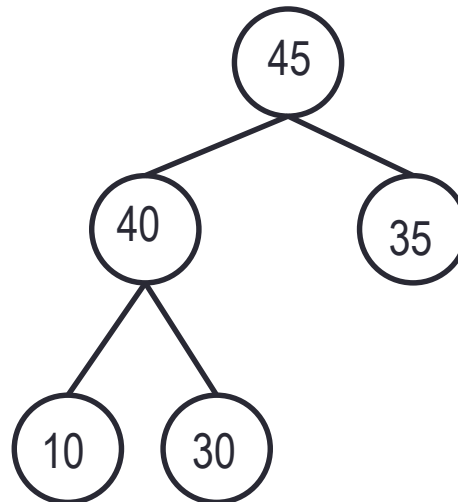


Heap Sort (4)

- O algoritmo é constituído por:
 1. Remover o máximo do Heap: Trocar o elemento na 1ª posição pelo último; decrementar o último.
 2. Executar borbulhar [0.. última posição]

0	1	2	3	4	5
45	40	35	10	30	50

↑ última
posição

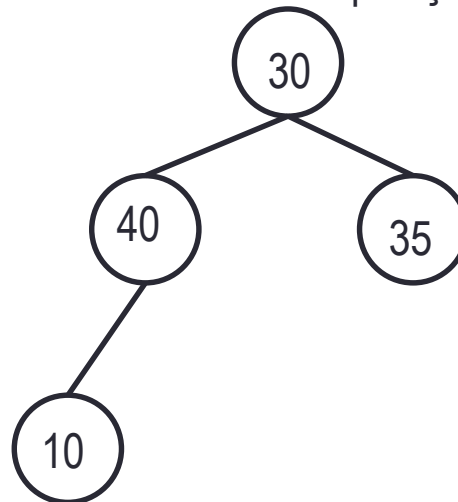


Heap Sort (5)

- O algoritmo é constituído por:
 1. Remover o máximo do Heap: Trocar o elemento na 1ª posição pelo último; decrementar o último.
 2. Executar borbulhar [0.. última posição]

0	1	2	3	4	5
30	40	35	10	45	50

↑ última
posição

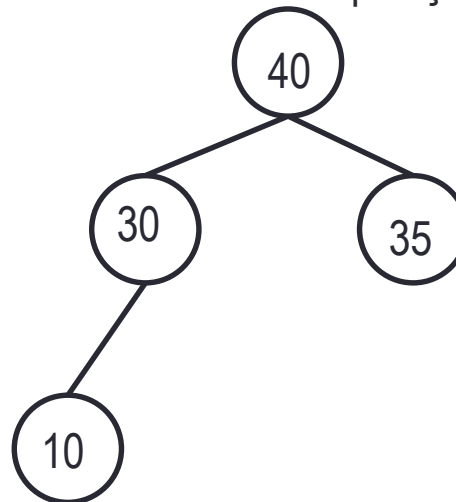


Heap Sort (6)

- O algoritmo é constituído por:
 1. Remover o máximo do Heap: Trocar o elemento na 1ª posição pelo último; decrementar o último.
 2. Executar borbulhar [0.. última posição]

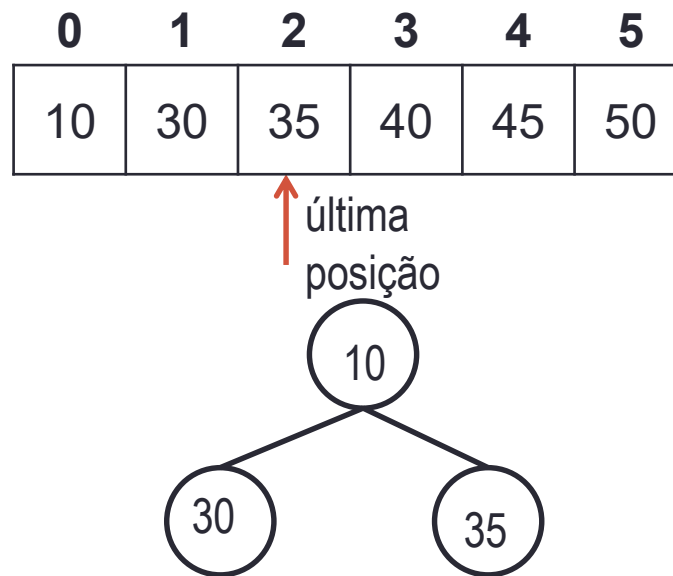
0	1	2	3	4	5
40	30	35	10	45	50

↑ última
posição



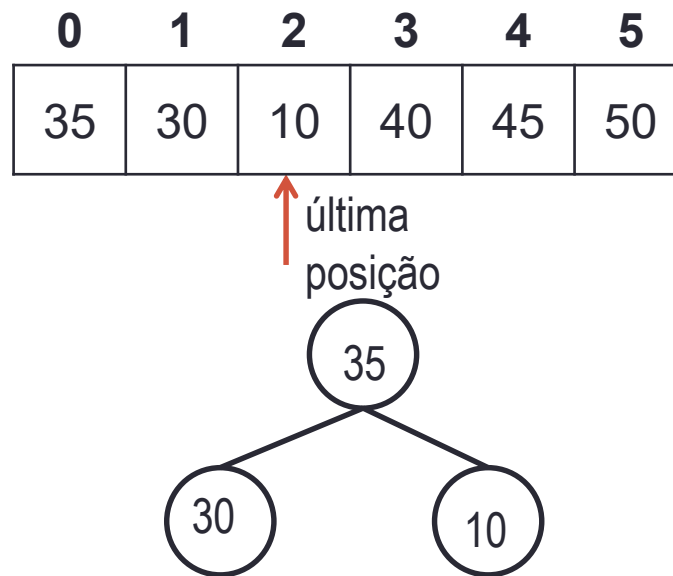
Heap Sort (7)

- O algoritmo é constituído por:
 1. Remover o máximo do Heap: Trocar o elemento na 1ª posição pelo último; decrementar o último.
 2. Executar borbulhar [0.. última posição]



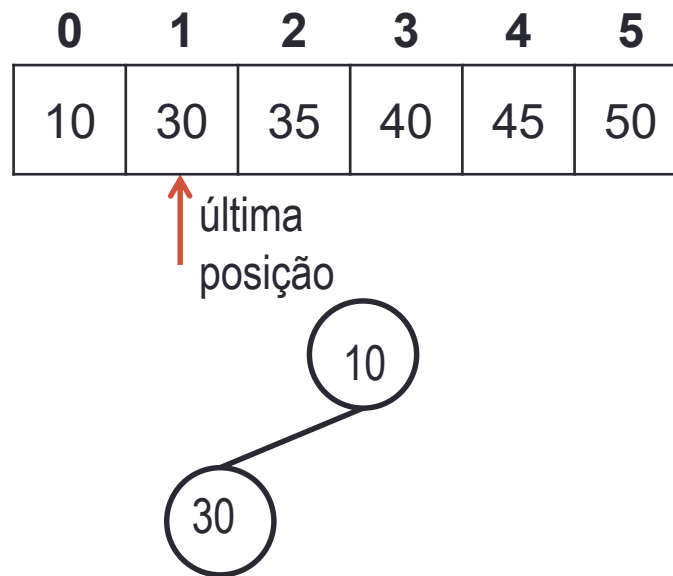
Heap Sort (8)

- O algoritmo é constituído por:
 1. Remover o máximo do Heap: Trocar o elemento na 1ª posição pelo último; decrementar o último.
 2. Executar borbulhar [0.. última posição]



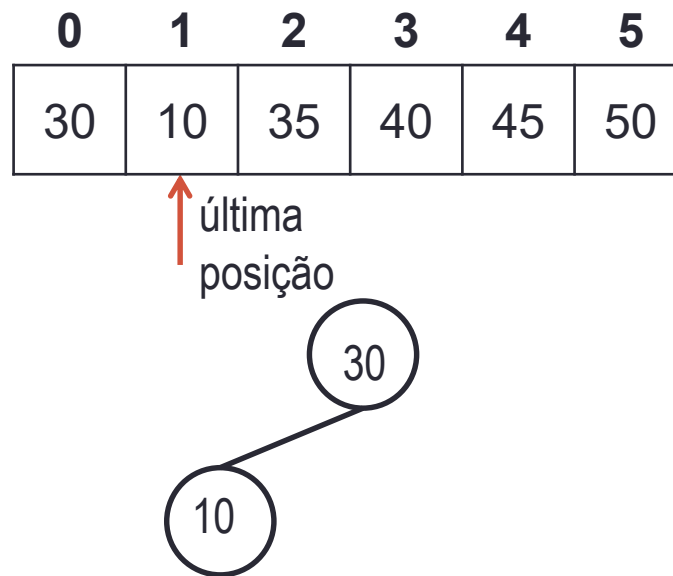
Heap Sort (9)

- O algoritmo é constituído por:
 1. Remover o máximo do Heap: Trocar o elemento na 1ª posição pelo último; decrementar o último.
 2. Executar borbulhar [0.. última posição]



Heap Sort (10)

- O algoritmo é constituído por:
 1. Remover o máximo do Heap: Trocar o elemento na 1ª posição pelo último; decrementar o último.
 2. Executar borbulhar [0.. última posição]



Heap Sort (11)

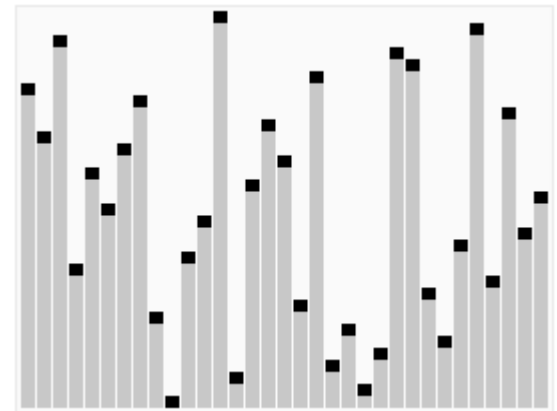
- O algoritmo é constituído por:
 1. Remover o máximo do Heap: Trocar o elemento na 1ª posição pelo último; decrementar o último.
 2. Executar borbulhar [0.. última posição]

0	1	2	3	4	5
10	30	35	40	45	50

↑ última posição

10

Vetor Ordenado!
Algoritmo Não Estável



Implementação Heap Sort (1)

```
protected static <E> void percolateDown( E[] vec, int firstPos,
                                         int lastPos, Comparator<E> c ){
    E rootElement = vec[firstPos];
    int hole = firstPos;
    int child = 2 * hole + 1; // Left child.
    while ( child <= lastPos ){
        // Find the largest child.
        if ( child < lastPos && c.compare(vec[child + 1], vec[child] ) > 0 )
            child++;

        // Compare the largest child with rootElement.
        if ( c.compare(vec[child], rootElement ) > 0 ) {
            vec[hole] = vec[child];
            hole = child;
            child = 2 * hole + 1; // Left child.
        }
        else break;
    }
    vec[hole] = rootElement;
}
```

Implementação Heap Sort (2)

```
public static <E> void heapSort( E[] vec, int vecSize,
                                Comparator<E> c ) {

    // Build a max priority queue (the heap).
    // Node at position i has left child if 2i+1 <= vecSize - 1.
    for ( int i = (vecSize - 2) / 2; i >= 0; i-- )
        percolateDown(vec, i, vecSize - 1, c);

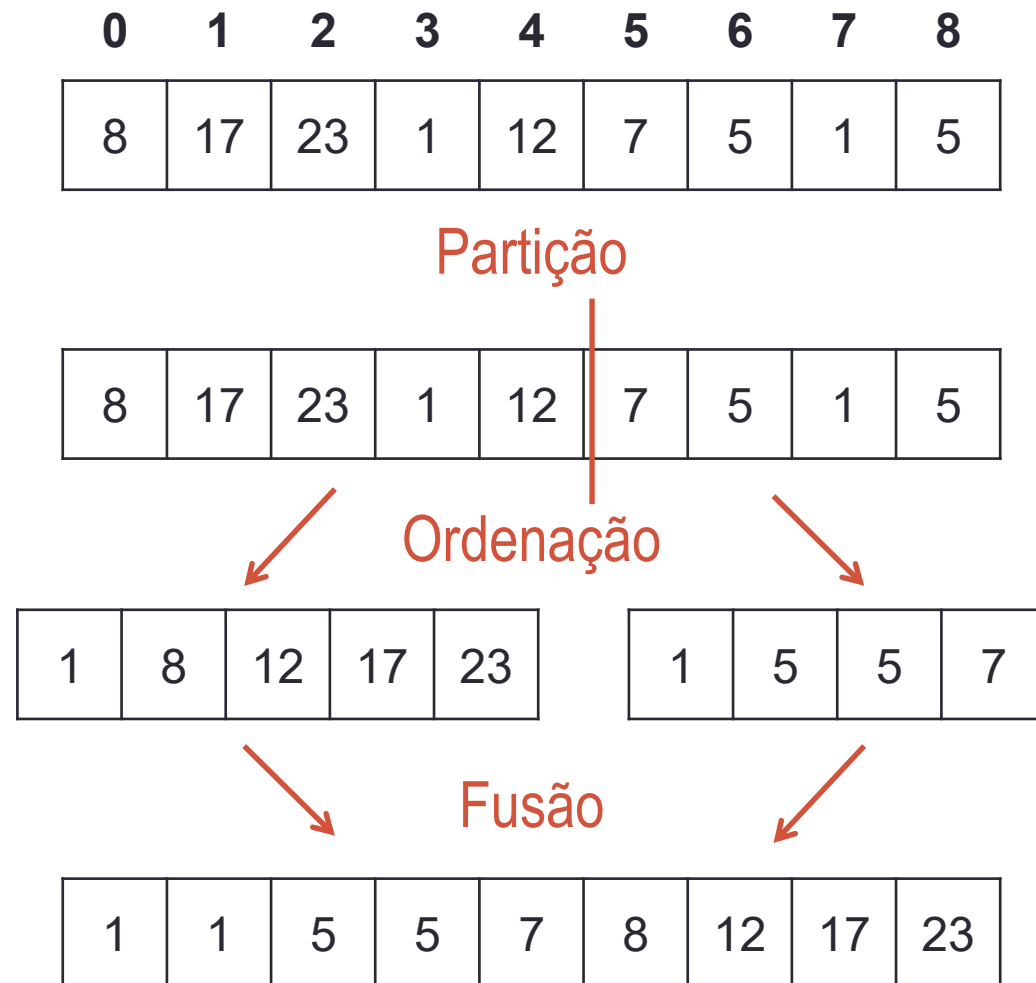
    // Sort by removing a maximum element.
    for ( int i = vecSize - 1; i > 0; i-- ) {
        swapElements(vec, 0, i);
        percolateDown(vec, 0, i - 1, c);
    }
}
```

Complexidade: $O(n \log n)$ (Construção do Heap: $O(n)$; Ordenação: $O(n \log n)$)

Merge Sort

- O algoritmo é constituído por:
 1. Dividir o vetor em dois
 2. Chamar o Merge Sort recursivamente para cada um dos vetores resultantes
 3. Fundir os dois vetores ordenados

Vetor Ordenado!
Algoritmo Estável



Implementação Merge Sort (Recursivo) (1)

```
public static <E> void mergeSortR( E[] vec, int vecSize,  
                                   Comparator<E> c ) {
```

```
    // Compiler gives a warning.
```

```
    E[] auxVec = (E[]) new Object[vecSize];
```

```
    mergeSortR(vec, auxVec, 0, vecSize - 1, c);
```

```
}
```

```
protected static <E> void mergeSortR( E[] vec, E[] auxVec,  
    int firstPos, int lastPos, Comparator<E> c ) {
```

```
    if ( firstPos < lastPos ){
```

```
        int centre = ( firstPos + lastPos ) / 2;
```

```
        mergeSortR(vec, auxVec, firstPos, centre, c);
```

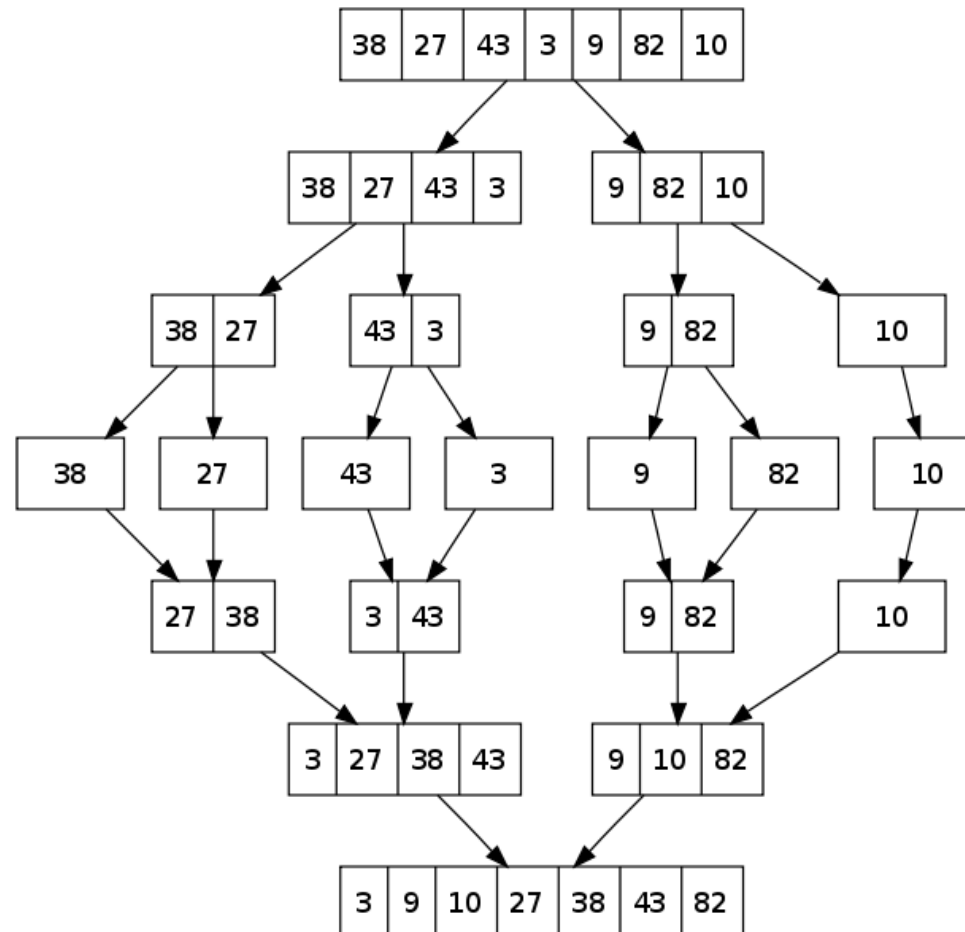
```
        mergeSortR(vec, auxVec, centre + 1, lastPos, c);
```

```
        mergeR(vec, auxVec, firstPos, centre, lastPos, c);
```

```
    }
```

```
}
```

Exemplo



Fusão (1)

```
protected static <E> void mergeR( E[] vec, E[] auxVec,
    int firstLeft, int lastLeft, int lastRight, Comparator<E> c ){

    int left = firstLeft;
    int right = lastLeft + 1;
    int result = firstLeft;
    while ( left <= lastLeft && right <= lastRight )
        if ( c.compare(vec[left], vec[right]) <= 0 )
            auxVec[ result++ ] = vec[ left++ ];
        else
            auxVec[ result++ ] = vec[ right++ ];
    // Copy rest of left sequence.
    while ( left <= lastLeft )
        auxVec[ result++ ] = vec[ left++ ];
    // Rest of right sequence in right place.
    // Copy from auxVec to vec.
    // Number of elements to be copied: (result - 1) - firstLeft + 1.
    System.arraycopy(auxVec, firstLeft, vec, firstLeft, result - firstLeft);
}
```

← Posições que vão ser comparadas

Fusão (1)

```
protected static <E> void mergeR( E[] vec, E[] auxVec,
    int firstLeft, int lastLeft, int lastRight, Comparator<E> c ){

    int left = firstLeft;
    int right = lastLeft + 1;
    int result = firstLeft;
    while ( left <= lastLeft && right <= lastRight )
        if ( c.compare(vec[left], vec[right]) <= 0 )
            auxVec[ result++ ] = vec[ left++ ];
        else
            auxVec[ result++ ] = vec[ right++ ];
    // Copy rest of left sequence.
    while ( left <= lastLeft )
        auxVec[ result++ ] = vec[ left++ ];
    // Rest of right sequence in right place.
    // Copy from auxVec to vec.
    // Number of elements to be copied: (result - 1) - firstLeft + 1.
    System.arraycopy(auxVec, firstLeft, vec, firstLeft, result - firstLeft);
}
```

Posição a usar em auxVec
Para inserir o menor

Fusão (1)

```
protected static <E> void mergeR( E[] vec, E[] auxVec,
    int firstLeft, int lastLeft, int lastRight, Comparator<E> c ){

    int left = firstLeft;
    int right = lastLeft + 1;
    int result = firstLeft;
    while ( left <= lastLeft && right <= lastRight )
        if ( c.compare(vec[left], vec[right]) <= 0 )
            auxVec[ result++ ] = vec[ left++ ];
        else
            auxVec[ result++ ] = vec[ right++ ];
    // Copy rest of left sequence.
    while ( left <= lastLeft )
        auxVec[ result++ ] = vec[ left++ ];
    // Rest of right sequence in right place.
    // Copy from auxVec to vec.
    // Number of elements to be copied: (result - 1) - firstLeft + 1.
    System.arraycopy(auxVec, firstLeft, vec, firstLeft, result - firstLeft);
}
```



Se os maiores estiverem na sequência do lado esquerdo

Fusão (1)

```
protected static <E> void mergeR( E[] vec, E[] auxVec,
    int firstLeft, int lastLeft, int lastRight, Comparator<E> c ){

    int left = firstLeft;
    int right = lastLeft + 1;
    int result = firstLeft;
    while ( left <= lastLeft && right <= lastRight )
        if ( c.compare(vec[left], vec[right]) <= 0 )
            auxVec[ result++ ] = vec[ left++ ];
        else
            auxVec[ result++ ] = vec[ right++ ];
    // Copy rest of left sequence.
    while ( left <= lastLeft )
        auxVec[ result++ ] = vec[ left++ ];
    // Rest of right sequence in right place.
    // Copy from auxVec to vec.
    // Number of elements to be copied: (result - 1) - firstLeft + 1.
    System.arraycopy(auxVec, firstLeft, vec, firstLeft, result - firstLeft);
}
```

Se os maiores estiverem na sequência do lado direito, essas posição não precisam de ser copiadas



Complexidade – Quantas comparações

$$C(n) = \begin{cases} 0 & n = 0, 1 \\ 2 C\left(\frac{n}{2}\right) + F(n) & n \geq 2 \end{cases}$$

$$F(n) = \begin{cases} n - 1 & \text{no pior caso} \\ \frac{n}{2} & \text{no melhor caso} \end{cases}$$

Recorrência 2(b)

$$T(n) = \begin{cases} a & n = 0 & n = 1 \\ bT\left(\frac{n}{c}\right) + O(n) & n \geq 1 & n \geq 2 \end{cases} \quad \text{ou}$$

com $a \geq 0$, $b \geq 1$, $c > 1$ constantes

$$T(n) = \begin{cases} O(n) & b < c \\ O(n \log_c n) & b = c \\ O(n^{\log_c b}) & b > c \end{cases}$$

Aplicando a recorrência 2(b)
ficamos com

$$C(n) = O(n \log n)$$

Logo

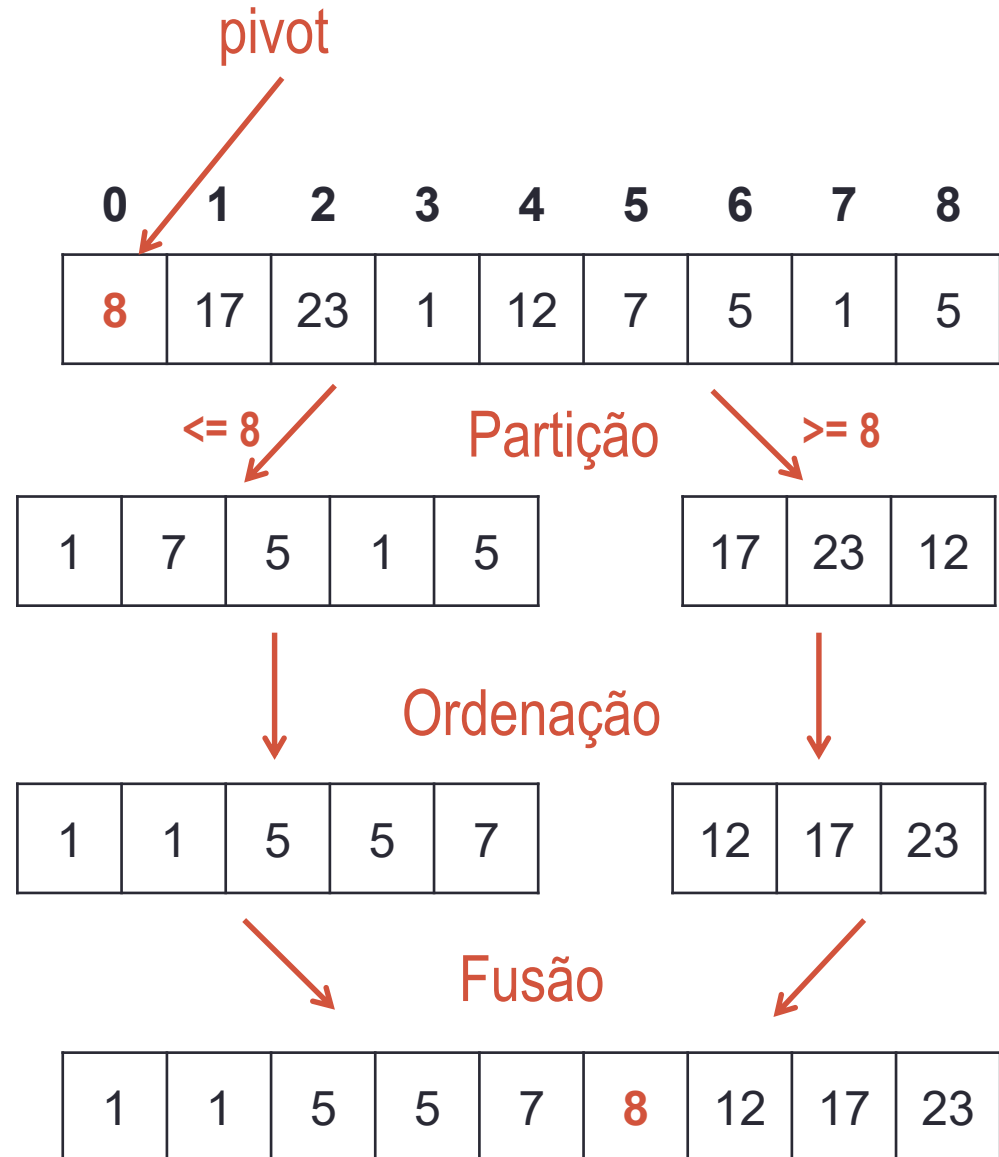
$$\text{mergeSort}(n) = O(n \log n)$$

Quick Sort

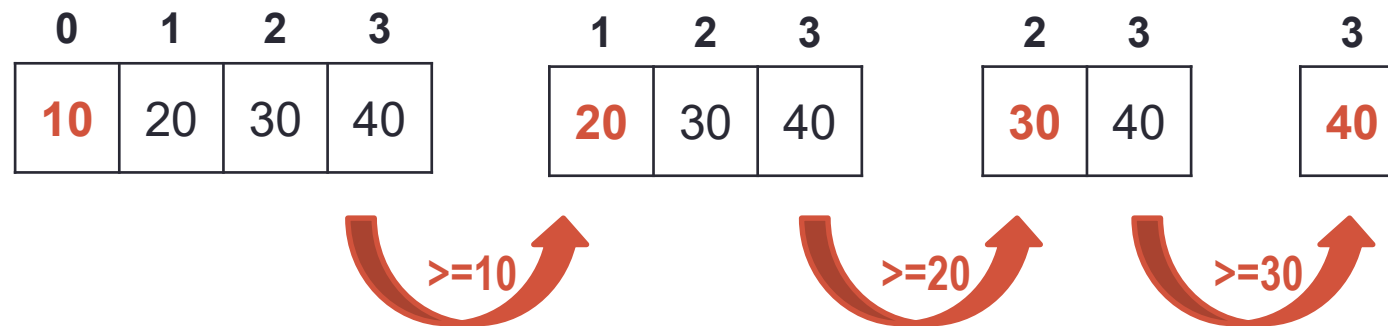
- O algoritmo é constituído por:

- Escolha de pivot
- O vetor é particionado, separando os valores menores que o pivot dos superiores ao pivot
- O algoritmo é chamado recursivamente para as partições
- Quando os vetores não ordenados são suficientemente pequenos chama-se o insertion sort

O algoritmo é pouco eficiente para vetores pequenos, por isso quando as zonas por ordenar são pequenas faz uma chamada ao insertion sort, para o vetor completo



Escolha do Pivot



- Se o pivot for sempre o mínimo ou máximo (se for escolhido sempre na mesma posição e o vetor estiver ordenado)
 - O número de comparações para ordenar n elementos será:

$$\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} = O(n^2)$$

- Para tentar evitar este caso, o pivot será a mediana entre três elementos: o primeiro, o último e o do meio

Como fazer a partição do vetor ? (versão1)

1. O pivot é a Mediana entre o primeiro, último e centro
2. Troca-se a primeira posição com o pivot
3. i avança até encontrar chave $\geq$ pivot (ou chegar ao fim)
4. j recua até encontrar chave $\leq$ pivot (ou chegar ao início)
5. Se $i < j$ trocar i com j e continuar
6. Senão troca-se pivot com j
7. Chama-se quicksort para as partições

12	4	6	1	7	3	11	22	5
----	---	---	---	---	---	----	----	---

7	4	6	1	12	3	11	22	5
---	---	---	---	----	---	----	----	---

i →

← j

7	4	6	1	12	3	11	22	5
---	---	---	---	----	---	----	----	---

i

j

7	4	6	1	5	3	11	22	12
---	---	---	---	---	---	----	----	----

i →

← j

7	4	6	1	5	3	11	22	12
---	---	---	---	---	---	----	----	----

j

i

3	4	6	1	5	7	11	22	12
---	---	---	---	---	---	----	----	----

Problemas com esta versão

- A chegada ao fim (ou ao princípio) do vetor tem de ser controlada pelo ciclo
- Isto pode ser resolvido comparando as chaves das primeira, última e posição central e posicionando-as ordenadas
- Depois posicionamos o pivot na segunda posição
- Assim, i inicia-se na segunda posição e avança até encontrar uma chave $\geq$ à do pivot
- j inicia-se na última posição e recua até encontrar um elemento cuja chave é $\leq$ à do pivot
- **Consequência:** O Algoritmo não é estável

Como fazer a partição do vetor ? (versão2)

1. O pivot é a Mediana entre o primeiro, último e centro
2. Ordenam-se estas 3 posições
3. i avança até encontrar chave $\geq$ pivot
4. j recua até encontrar chave $\leq$ pivot
5. Se $i < j$ trocar i com j e continuar
6. Senão troca-se pivot com j
7. Chama-se quicksort para as partições

12	4	6	1	7	3	11	22	5
----	---	---	---	---	---	----	----	---

5	4	6	1	7	3	11	22	12
---	---	---	---	---	---	----	----	----

5	7	6	1	4	3	11	22	12
---	---	---	---	---	---	----	----	----

i →

← j

5	7	6	1	4	3	11	22	12
---	---	---	---	---	---	----	----	----

j i

5	3	6	1	4	7	11	22	12
---	---	---	---	---	---	----	----	----

5	3	6	1	4	7	11	22	12
---	---	---	---	---	---	----	----	----

Implementação Quick Sort (1)

```
public static final int CUTOFF = 16;

public static <E> void quickSort( E[] vec, int vecSize, Comparator<E> c ) {

    quickSort(vec, 0, vecSize - 1, c);
    insertionSort(vec, vecSize, c);
}
```

Implementação Quick Sort (2)

```
protected static <E> E median3( E[] vec, int firstPos, int lastPos,
                                Comparator<E> c ) {

    int centre = ( firstPos + lastPos ) / 2;
    if ( c.compare(vec[centre], vec[firstPos]) < 0 )
        swapElements(vec, firstPos, centre);
    // vec[firstPos] <= vec[centre].

    if ( c.compare(vec[lastPos], vec[firstPos]) < 0 )
        swapElements(vec, firstPos, lastPos);
    // vec[firstPos] <= vec[centre]; vec[firstPos] <= vec[lastPos].

    if ( c.compare(vec[lastPos], vec[centre]) < 0 )
        swapElements(vec, centre, lastPos);
    // vec[firstPos] <= vec[centre] <= vec[lastPos].

    // The pivot is vec[centre].
    // Place pivot at position firstPos + 1.
    swapElements(vec, firstPos + 1, centre);
    return vec[firstPos + 1];
}
```

Implementação Quick Sort (3)

```
protected static <E> void quickSort( E[] vec, int firstPos, int lastPos,
                                     Comparator<E> c ) {

    if ( lastPos - firstPos >= CUTOFF ) {
        E pivot = median3(vec, firstPos, lastPos, c);
        int i = firstPos + 1;
        int j = lastPos;
        while ( true ) {
            do { i++; } while ( c.compare(vec[i], pivot) < 0 );
            do { j--; } while ( c.compare(vec[j], pivot) > 0 );
            if ( i < j )
                swapElements(vec, i, j);
            else
                break;
        }
        // Restore pivot.
        swapElements(vec, firstPos + 1, j);
        quickSort(vec, firstPos, j - 1, c);
        quickSort(vec, j + 1, lastPos, c);
    }
}
```

Complexidade do Quick Sort (pior caso)

- Número de comparações

$$C(n) = \begin{cases} 0 & n = 0, 1 \\ C(k) + C(n - k - 1) + (n - 1) & n \geq 2 \end{cases}$$

- No pior caso (quando $k = 0$ ou $k = n - 1$)

$$C(n) = \begin{cases} 0 & n = 0, 1 \\ C(n - 1) + (n - 1) & n \geq 2 \end{cases}$$

- No pior caso, $C(n) = O(n^2)$

Complexidade do Quick Sort (melhor caso)

- Número de comparações

$$C(n) = \begin{cases} 0 & n = 0,1 \\ C(k) + C(n - k - 1) + (n - 1) & n \geq 2 \end{cases}$$

- No melhor caso (quando $k = n / 2$)

$$C(n) = \begin{cases} 0 & n = 0,1 \\ 2 C\left(\frac{n}{2}\right) + (n - 1) & n \geq 2 \end{cases}$$

- No melhor caso, $C(n) = O(n \log n)$

Caso Esperado: Se a cada invocação do método, a sequência for dividida ao meio, então o caso médio será parecido com o melhor caso , $C(n) = O(n \log n)$

Resumo – Algoritmos de ordenação

	Melhor Caso	Pior Caso	Caso Esperado	Estabilidade
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	Sim
Bubble	$O(n^2)$	$O(n^2)$	$O(n^2)$	Sim
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	Não
Heap	$O(n)$	$O(n \log n)$	$O(n \log n)$	Não
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Sim
Quick	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	Não