Agenda

Unidade 5: Algoritmo de Ordenação

1. Bubble sort
2. Insertion sort
3. Selection sort
4. Merge sort
5. Quick sort
6. Heap sort
7. Ordenação em tempo linear: radixsort, bucket sort e **counting sort**
8. Implementações

CountingSort

- Foi proposto por Harold Seward em 1954 e faz a ordenação usando a idéia de contagem
- Restrição: sór ordena listas de números naturais (números inteiros não negativos)
- Não é uma restrição muito severa, pois pode-se converter diversos tipos de listas em listas de número naturais
  - Ex: Uma lista de números inteiros onde o menor valor é –x pode ser convertidade em uma lista de números naturais somando x a cada elemento da lista
  - Uma lista de números racionais pode ser convertidade em uma lista de números inteiros multiplicando-se os elementos da lista por uma constante apropriada

CountingSort

- Seja n o tamanho da lista e k o maior valor contido na lista.
- Faz-se o uso de um vetor C com k+1 posições
- No 1º laço do CountingSort o Vetor C é inicializado com zeros
- No laço seguinte, contamos quantas vezes cada um dos elementos de 0 a k aperece na lista
  - Para fazer isso, a lista deve ser percorrida e para cada valor i contido na lista incrementa-se 1 unidade a posição i do vetor C
  - Dessa forma, ao final desse laço cada posição i de C indica quantas vezes o valor de i ocorre na lista
  - Em seguida, fazemos a soma acumulada do vetor C
  - Após a soma acumulada, cada posição i do vetor C indicará quantos são menores ou iguais a i

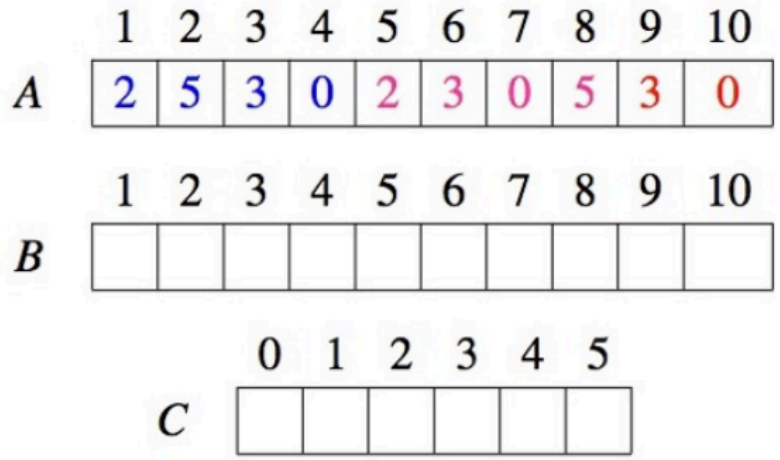
CountingSort

```
ALGORITMO COUNTING SORT
  ENTRADA: UM VETOR L CONTENDO N NÚMEROS NATURAIS NO
           INTERVALO DE 0 A K
  SAÍDA: O VETOR L EM ORDEM CRESCENTE

  PARA i = 0 até k                      // INICIALIZACAO DE C
    c[i]=0
  PARA i = 0 até n - 1                  // CONTAGEM
    c[L[i]]= c[L[i]] +1
  PARA i = 1 até k                      // SOMA ACUMULADA
    c[i]= c[i] + c[i-1]
  PARA i= n - 1 até 0 (passo -1)        // ORDENACAO
    c[L[i]] = c[L[i]] - 1
    aux[c[L[i]]]= L[i]
  PARA i = 0 até n - 1                  // COPIANDO AUX PARA L
    L[i]= aux[i]
FIM {COUNTING SORT}
```

CountingSort

Cada $A[i]$ está em $\{0, \ldots, 5\}$.



CountingSort

Cada $A[i]$ está em $\{0, \ldots, 5\}$.

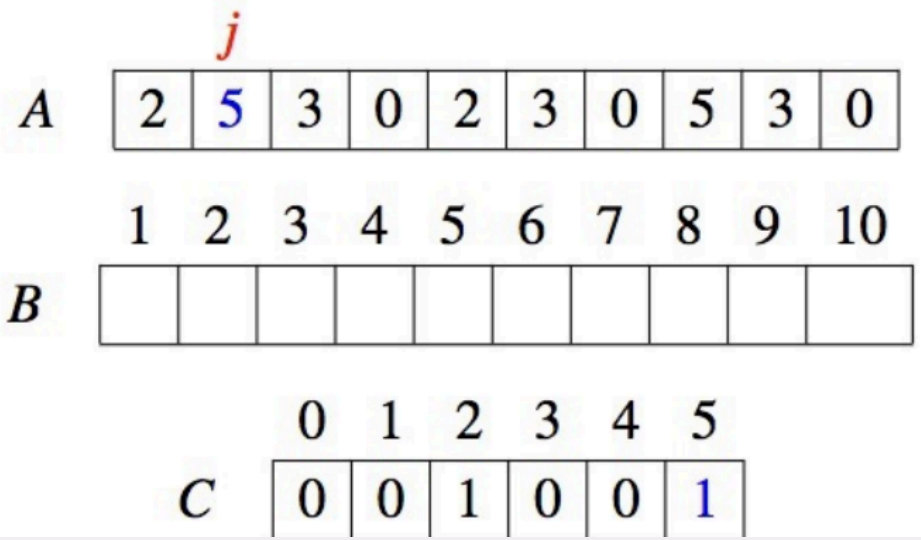
	1	2	3	4	5	6	7	8	9	10
A	2	5	3	0	2	3	0	5	3	0

	1	2	3	4	5	6	7	8	9	10
B										

	0	1	2	3	4	5
C	0	0	0	0	0	0

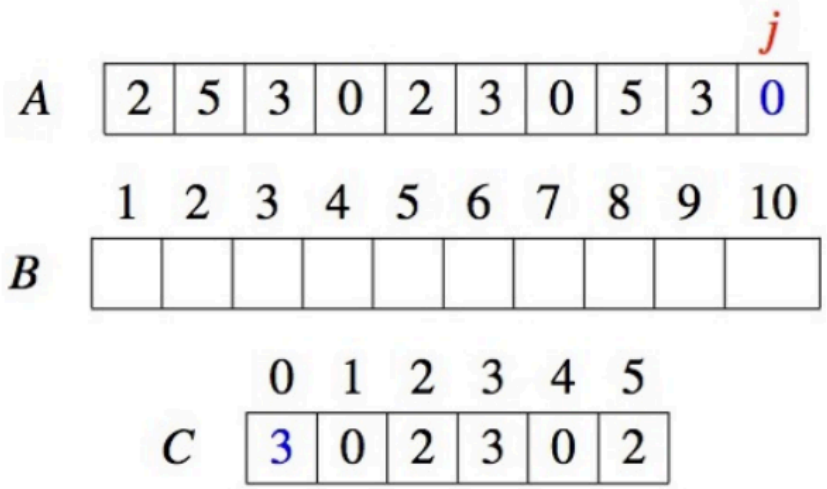
CountingSort

Cada $A[i]$ está em $\{0, \ldots, 5\}$.

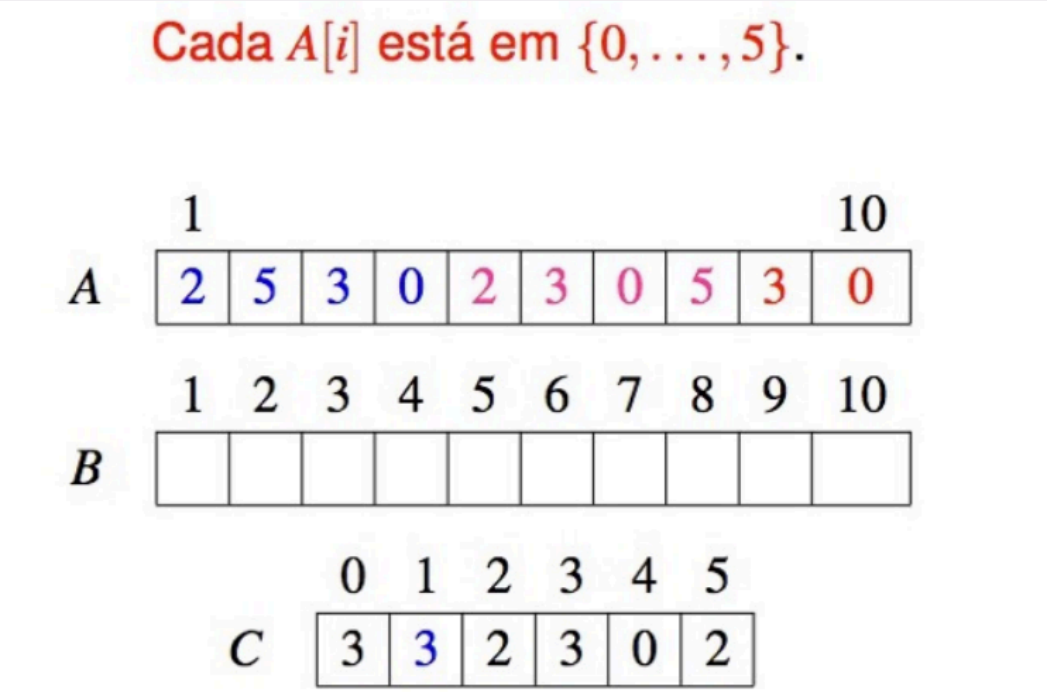


CountingSort

Cada $A[i]$ está em $\{0, \ldots, 5\}$.

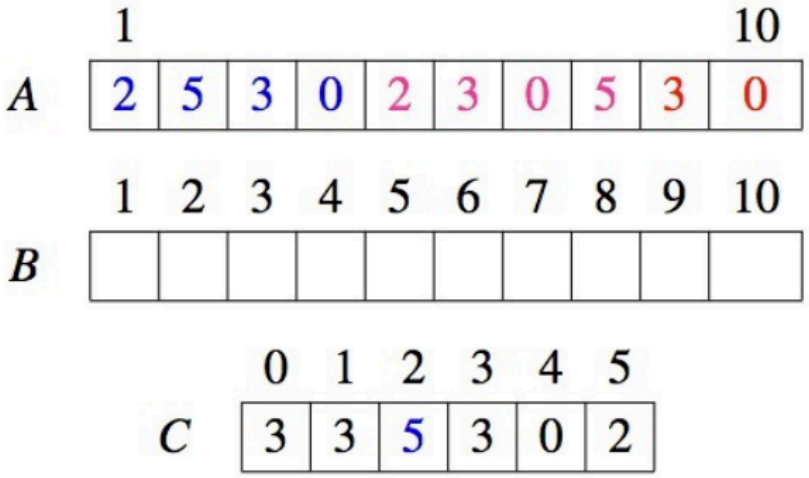


CountingSort

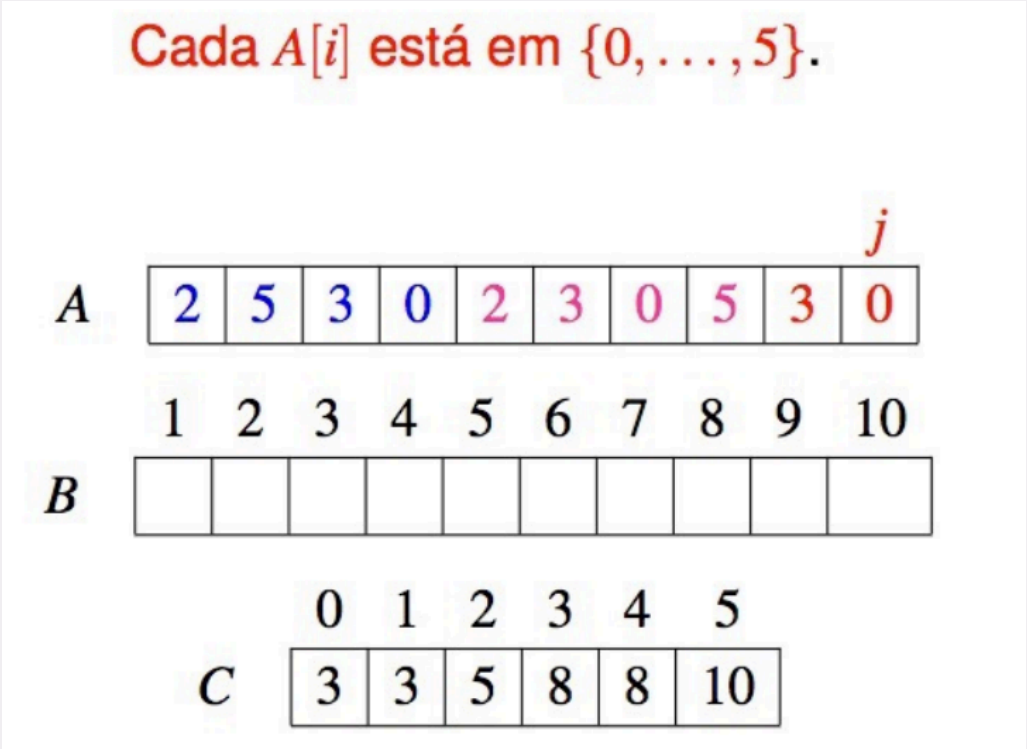


CountingSort

Cada $A[i]$ está em $\{0, \ldots, 5\}$.



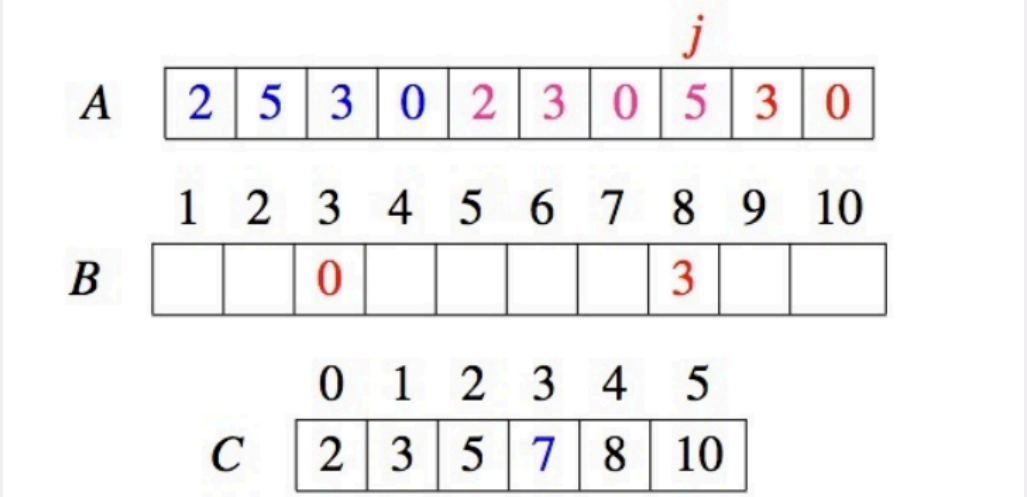
CountingSort



CountingSort

CountingSort

Cada $A[i]$ está em $\{0, \ldots, 5\}$.



CountingSort

Cada $A[i]$ está em $\{0, \ldots, 5\}$.

CountingSort

- Análise
  - Constituído de 5 laços independentes
  - 3 laços gastam tempo Θ(n) e os outros 2 gastam tempo Θ(k)
  - Além disso o vetor C tem k+1 posições e o vetor aux tem n posições
  - Conclui-se: complexidade temporal e espacial Θ(n+k)
  - Para a classe de listas onde k não é muito maior do que n, mais precisamente, se k ε Θ(n), então tais complexidades pertencem a Θ(n)
  - Algoritmo estável

Agenda

Unidade 5: Algoritmo de Ordenação

1. Bubble sort
2. Insertion sort
3. Selection sort
4. Merge sort
5. Quick sort
6. Heap sort
7. Ordenação em tempo linear: radixsort, **bucket sort** e counting sort
8. Implementações

BucketSort

- Se a lista a ser ordenada tem n elementos, será usado um vetor de ponteiros (bucket) com n posições
- Se o maior elemento da lista é k, cada posição do bucket apontará para uma lista encadeada na qual serão inseridos os elementos da lista que pertencem ao intervalo [i*(k+1)/n, (i+1)*(k+1)/n)
- Obs: O intervalo é fechado à esquerda e aberto à direita

BucketSort

- Idéia: dividir o intervalo que vai de 0 até k em n subintervalos de mesmo tamanho
  - Cada subintervalo estará associado a uma lista ligada que irá conter os elementos da lista que pertencem àquele subintervalo
  - Ex: Se a lista tem 8 elementos e o maior deles é 71, temos 8 intervaloes: [0,9), [9,18),…,[63,72)
  - A posição zero do bucket apontará para uma lista encadeada que irá conter os elementos da lista que são maiores ou iguais a 0 e menores que 9, e assim por diante

BucketSort

- Chamaremos o bucket de vetor B.
- Para construir as listas encadeadas devemos inserir cada valor j contido na lista a ser ordenada na lista encadeada apontada por B[j*n/(k+1)]
- Em seguida, ordenamos as listas encadeadas com um método de ordenação qualquer (ex.,inserção)
- Após isso, a concatenação das listas encadeadas produz a lista original ordenada

BucketSort

```
BUCKETSORT(A, n)
 1  para i ← 0 até n − 1 faça
 2      B[i] ← NIL
 3  para i ← 1 até n faça
 4      INSIRA(B[⌊n A[i]⌋], A[i])
 5  para i ← 0 até n − 1 faça
 6      ORDENELISTA(B[i])
 7  C ← CONCATENE(B, n)
 8  devolva C
```

- INSIRA(p, x): insere x na lista apontada por p
  ORDENELISTA(p): ordena a lista apontada por p
- CONCATENE(B, n): devolve a lista obtida da concatenação das listas apontadas por B[0], . . . , B[n – 1].

BucketSort

- Recebe um inteiro n e um vetor A[1 . . n] onde cada elemento é um número no intervalo [0, 1).

A

.47	.93	.82	.12	.42	.03	.62	.38	.77	.91

- Devolve um vetor C[1 . . n] com os elementos de A[1 . . n] em ordem crescente.

C

.03	.12	.38	.42	.47	.62	.77	.82	.91	.93

BucketSort

.47	.93	.82	.12	.42	.03	.62	.38	.77	.91

$B[0]$:	.03	
$B[1]$:	.12	
$B[2]$:		
$B[3]$:	.38	
$B[4]$:	.47	.42
$B[5]$:		
$B[6]$:	.62	
$B[7]$:	.77	
$B[8]$:	.82	
$B[9]$:	.93	.91

BucketSort

.47	.93	.82	.12	.42	.03	.62	.38	.77	.91

$B[0]$:	.03	
$B[1]$:	.12	
$B[2]$:		
$B[3]$:	.38	
$B[4]$:	.42	.47
$B[5]$:		
$B[6]$:	.62	
$B[7]$:	.77	
$B[8]$:	.82	
$B[9]$:	.91	.93

BucketSort

.47	.93	.82	.12	.42	.03	.62	.38	.77	.91

$B[0]$:	.03	
$B[1]$:	.12	
$B[2]$:		
$B[3]$:	.38	
$B[4]$:	.42	.47
$B[5]$:		
$B[6]$:	.62	
$B[7]$:	.77	
$B[8]$:	.82	
$B[9]$:	.91	.93

.03	.12	.38	.42	.47	.62	.77	.82	.91	.93

BucketSort

- Análise
  - Claramente os primeiros laços podem ser executados em tempo Θ(n)
  - O tamanho médio de cada lista encadeada será 1
  - Se os elementos de L estiverem uniformemente distribuídos no intervalo [0,k), é possível que o tamanho esperado de cada lista encadeada seja constante
  - Nesse caso, independente do método utilizado, a ordenação de cada lista encadeada será feita em tempo esperado constante
  - Complexidade temporal: Θ(n)
  - Complexidade espacial: Θ(n)

Agenda

Unidade 5: Algoritmo de Ordenação

1. Bubble sort
2. Insertion sort
3. Selection sort
4. Merge sort
5. Quick sort
6. Heap sort
7. Ordenação em tempo linear: **radixsort**, bucket sort e counting sort
8. Implementações

RadixSort

- Permite ordenar listas cujos elementos sejam dois a dois comparáveis
- Idéia:
  - Quebrar o elemento em vários pedaços
    - Ex: 312 tem os dígitos 3, 1 e 2 na base 10
    - 312 tem os dígitos 100111000 na base 2
    - Exemplo tem 6 caracteres
  - Ordenar de acordo com o primeiro pedaço
    - Número cujo dígito mais à esquerda começa com 0 vêm antes de números cujo dígito mais à esquerda é 1
  - Podemos ordenar repetindo esse processo para todos os pedaços

RadixSort

- Para isso iremos contar quantos elementos existem de cada valor

12**3**	14**2**	08**7**	26**3**	23**3**	01**4**	13**2**

Digito	Contador
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0

RadixSort

- Para isso iremos contar quantos elementos existem de cada valor

12**3**	14**2**	08**7**	26**3**	23**3**	01**4**	13**2**

Digito	Contador
0	0
1	0
2	**2**
3	**3**
4	**1**
5	0
6	0
7	**1**
8	0
9	0

RadixSort

- Depois calcular a posição deles no vetor

123	142	087	263	233	014	132

Dig	C	Posicao
0	0	0
1	0	0
2	2	0
3	3	2
4	1	5
5	0	0
6	0	0
7	1	6
8	0	0
9	0	0

RadixSort

- E finalmente colocar os elementos em suas posições

123	142	087	263	233	014	132
		123				

Dig	C	Posicao
0	0	0
1	0	0
2	2	0
3	3	3
4	1	5
5	0	0
6	0	0
7	1	6
8	0	0
9	0	0

RadixSort

- Para isso iremos contar quantos elementos existem de cada valor

123	142	087	263	233	014	132
142		123				

Dig	C	Posicao
0	0	0
1	0	0
2	2	1
3	3	3
4	1	5
5	0	0
6	0	0
7	1	6
8	0	0
9	0	0

RadixSort

- Para isso iremos contar quantos elementos existem de cada valor

123	142	087	263	233	014	132
142		123				087

Dig	C	Posicao
0	0	0
1	0	0
2	2	1
3	3	3
4	1	5
5	0	0
6	0	0
7	1	7
8	0	0
9	0	0

RadixSort

- Para isso iremos contar quantos elementos existem de cada valor

123	142	087	263	233	014	132
142		123	263			087

Dig	C	Posicao
0	0	0
1	0	0
2	2	1
3	3	4
4	1	5
5	0	0
6	0	0
7	1	7
8	0	0
9	0	0

RadixSort

- Para isso iremos contar quantos elementos existem de cada valor

123	142	087	263	233	014	132
142		123	263	233		087

Dig	C	Posicao
0	0	0
1	0	0
2	2	1
3	3	5
4	1	5
5	0	0
6	0	0
7	1	7
8	0	0
9	0	0

RadixSort

- Para isso iremos contar quantos elementos existem de cada valor

123	142	087	263	233	014	132
142		123	263	233	014	087

Dig	C	Posicao
0	0	0
1	0	0
2	2	1
3	3	5
4	1	6
5	0	0
6	0	0
7	1	7
8	0	0
9	0	0

RadixSort

- Para isso iremos contar quantos elementos existem de cada valor

123	142	087	263	233	014	132
142	132	123	263	233	014	087

Dig	C	Posicao
0	0	0
1	0	0
2	2	1
3	3	5
4	1	6
5	0	0
6	0	0
7	1	7
8	0	0
9	0	0

RadixSort

- Repetimos o mesmo processo para o próximo dígito
  - Funciona por que o método contador usado anteriormente é estável

12**3**	14**2**	08**7**	26**3**	23**3**	01**4**	13**2**
1**4**2	1**3**2	1**2**3	2**6**3	2**3**3	0**1**4	0**8**7
0**1**4	1**2**3	1**3**2	2**3**3	1**4**2	2**6**3	0**8**7

RadixSort

- Repetimos o mesmo processo para o próximo dígito
  - Funciona por que o método contador usado anteriormente é estável

12**3**	14**2**	08**7**	26**3**	23**3**	01**4**	13**2**
1**4**2	1**3**2	1**2**3	2**6**3	2**3**3	0**1**4	0**8**7
014	**1**23	**1**32	**2**33	**1**42	**2**63	**0**87
014	087	123	132	142	233	263

Exercício

- Faça um teste de mesa com cada método de ordenação estudado até o momento, utilizando as seguintes sequências de dados de entrada:
  - S1 = {2,4,6,8,10,12}
  - S2 = {11,9,7,5,3,1}
  - S3 = {5,7,2,8,1,6}
  - S4 = {2,4,6,8,10,12,11,9,7,5,3,1}
  - S5 = {2,4,6,8,10,12,1,3,5,7,9,11}
  - S6 = {8,9,7,9,3,2,3,8,4,6}
  - S7 = {89, 79, 32, 38, 46, 26, 43, 38, 32, 79}
- Em cada caso, mostre o número de comparações e trocas que realizam na ordenação de sequências.

Perguntas?