Ordenação

Estruturas de Dados - BCC/IFC
Prof. Dr. Paulo César Rodacki Gomes

Tópicos

- Introdução
- Ordenação bolha (BubbleSort)
- Ordenação Rápida (QuickSort)
- MergeSort

- Referências: Waldemar Celes, Renato Cerqueira, José Lucas Rangel, Introdução a Estruturas de Dados, Editora Campus (2004)

Introdução

- Ordenação de vetores
- Entrada: vetor com elementos a serem ordenados
- Saída: vetor com os elementos na ordem especificada

Introdução

- Ordenação: pode ser aplicada em qualquer conjunto de dados com ordem definida
- vetores de objetos:
  - chave de ordenação escolhida entre os atributos
  - elemento do vetor contém referência para objeto
  - troca de ordem entre dois elementos == troca de referência para objeto

BubbleSort

- Quando dois elementos estão fora de ordem, troque-os de posição até que o i-ésimo elemento de maior valor do vetor seja levado para as posições finais do vetor
- continue o processo até que todo o vetor esteja ordenado

- EXEMPLO:

Maior elemento

v0	**4**	**2**	**5**	**1**
v1	**2**	**4**	**5**	**1**
v2	**2**	**4**	**5**	**1**
v3	**2**	**4**	**1**	**5**
	0	1	2	3

2º Maior elemento

v4	**2**	**4**	**1**	**5**
v5	**2**	**4**	**1**	**5**
	2	**1**	**4**	**5**
	0	1	2	3

3º Maior elemento

v6	**2**	**1**	**4**	**5**
v5	**1**	**2**	**4**	**5**
	0	1	2	3

Exemplo

```
25 48 37 12 57 86 33 92     25x48
25 48 37 12 57 86 33 92     48x37 troca
25 37 48 12 57 86 33 92     48x12 troca
25 37 12 48 57 86 33 92     48x57
25 37 12 48 57 86 33 92     57x86
25 37 12 48 57 86 33 92     86x33 troca
25 37 12 48 57 33 86 92     86x92
25 37 12 48 57 33 86 92     final da primeira passada
```

o maior elemento, 92, já está na sua posição final

Exemplo

```
25 37 12 48 57 33 86 92      25x37
25 37 12 48 57 33 86 92      37x12 troca
25 12 37 48 57 33 86 92      37x48
25 12 37 48 57 33 86 92      48x57
25 12 37 48 57 33 86 92      57x33 troca
25 12 37 48 33 57 86 92      57x86
25 12 37 48 33 57 86 92      final da segunda passada
```

o segundo maior elemento, 86, já está na sua posição final

Exemplo

```
25 12 37 48 33 57 86 92     25x12 troca
12 25 37 48 33 57 86 92     25x37
12 25 37 48 33 57 86 92     37x48
12 25 37 48 33 57 86 92     48x33 troca
12 25 37 33 48 57 86 92     48x57
12 25 37 33 48 57 86 92     final da terceira passada
```

Idem para 57.

```
12 25 37 33 48 57 86 92     12x25
12 25 37 33 48 57 86 92     25x37
12 25 37 33 48 57 86 92     37x33 troca
12 25 33 37 48 57 86 92     37x48
12 25 33 37 48 57 86 92     final da quarta passada
```

Idem para 48.

```
12 25 33 37 48 57 86 92     12x25
12 25 33 37 48 57 86 92     25x33
12 25 33 37 48 57 86 92     33x37
12 25 33 37 48 57 86 92     final da quinta passada
```

Idem para `37`.

```
12 25 33 37 48 57 86 92     12x25
12 25 33 37 48 57 86 92     25x33
12 25 33 37 48 57 86 92     final da sexta passada
```

Idem para `33`.

```
12 25 33 37 48 57 86 92     12x25
12 25 33 37 48 57 86 92     final da sétima passada
```

Idem para `25` e, conseqüentemente, `12`.

```
12 25 33 37 48 57 86 92     final da ordenação
```

BubbleSort

- Algoritmo interativo (versão 1)

Algoritmo: bubbleSort(int v[])

```
int i, j;
int n ← v.length;
para i ← n − 1 até 1 faça
    para j ← 0 até i faça
        se v[j] > v[j + 1] então
            int temp ← v[j];
            v[j] ← v[j + 1];
            v[j + 1] ← temp;
```

Algoritmo 5.1: BubbleSort (versão 1)

BubbleSort

- Algoritmo interativo (versão 2)

Algoritmo: bubbleSort(int v[])

```
int i, j;
int n ← v.length;
para i ← n − 1 até 1 faça
    boolean troca ← false;
    para j ← 0 até i faça
        se v[j] > v[j + 1] então
            int temp ← v[j];
            v[j] ← v[j + 1];
            v[j + 1] ← temp;
            troca ← true;
    se troca ≠ true então retorna
```

Algoritmo 5.2: BubbleSort (versão 2)

BubbleSort

- Esforço computacional $\cong$ número de comparações
- Esforço computacional $\cong$ número máximo de trocas
  - primeira passada: n-1 comparações
  - segunda passada: n-2 comparações
  - terceira passada: n-3 comparações

BubbleSort

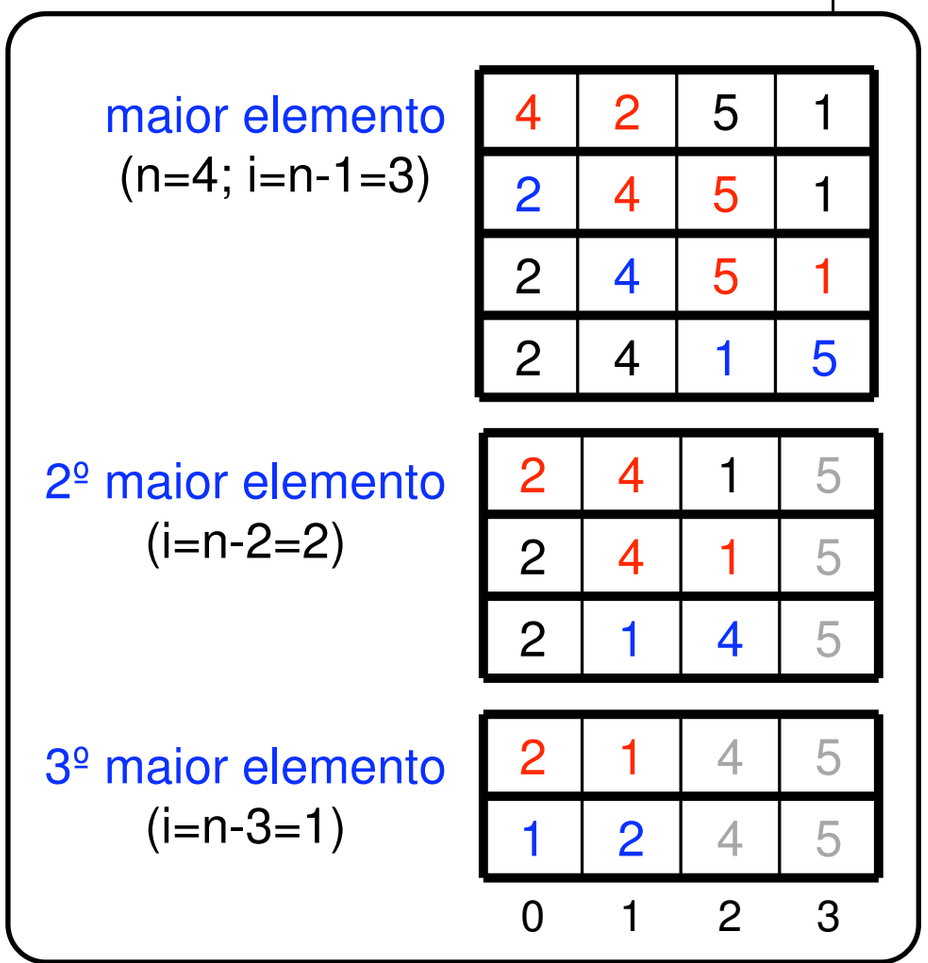
- Tempo total gasto pelo algoritmo:
- T proporcional a
  (n-1) + (n-2) + ... + 2 + 1 = n*(n-1)/2
- Algoritmo de ordem quadrática: $O(n^2)$

BubbleSort

- Implementação recursiva:

Algoritmo: bubbleSortRecursivo(int n, int v[])

```
int j;
boolean troca ← false;
para j ← 0 até n-1 faça
    se v[j] > v[j + 1] então
        int temp ← v[j];
        v[j] ← v[j + 1];
        v[j + 1] ← temp;
        troca ← true;
se troca então
    bubbleSortRecursivo(n-1, v);
```



Algoritmo 5.3: BubbleSort recursivo

BubbleSort

- Algoritmo genérico:
- independente do tipo/classe dos dados armazenados no vetor
- usa função auxiliar para comparar elementos

QuickSort

"Ordenação rápida":

- escolha um elemento arbitrário x, o pivô
- rearrume o vetor de tal forma que x fique na posição correta v[i]
  - x está na posição correta quando todos os elementos v[0], ..., v[i-1] sao menores que x e todos os elementos v[i+1], ..., v[n-1] são maioresque x
- chame recursivamente o algoritmo para ordenar os (sub-)vetores v[0], ..., v[i-1] e v[i+1], ..., v[n-1]
- continue até que os vetores a serem ordenados tenham 0 ou 1 elemento

Ref.: CORMEN, Thomas H. Algoritmos: teoria e prática. Rio de Janeiro: Campus, 2002

QuickSort

Esforço computacional:

- melhor caso:
    - pivô representa o valor mediano do conjunto dos elementos do vetor
    - após mover o pivô em sua posição, restarão dois sub-vetores para serem ordenados, ambos com número de elementos reduzidos à metade em relação ao vetor original
    - o algoritmo é O(n log(n))

QuickSort

Esforço computacional:

- pior caso:
    - pivô é o maior elemento, o algoritmo recai em BubbleSort
- caso médio:
    - o algoritmo é O(n log(n))

QuickSort

Rearrumação do vetor para o pivô de x=v[0]:

- do início para o final, compare x com v[1], v[2], … até encontrar v[a]>x
- do final para o início, compare x com v[n-1], v[n-2], … até encontrar v[b]<=x
- troque v[a] e v[b]
- continue para o final a partir de v[a+1] e para o início a partir de v[b-1]
- termine quando os pontos de busca se encontram (b<a)
- a posição correta de x=v[0] é a posição b e v[0] e v[b] são trocados

QuickSort: Exemplo

- vetor inteiro de v[0] a v[7]: (0-7) 25 48 37 12 57 86 33 92
- determine a posição correta de x=v[0]=25
    - de a=1 para o fim: 48>25 (a=1)
    - de b=7 para o início: 25<92, 25<33, 25<86, 25<57 e 12<=25 (b=3)
- (0-7) 25 **48** 37 **12** 57 86 33 92
  a↑ b↑
- troque v[a]=48 e v[b]=12, incrementando **a** e decrementando **b**
- nova configuração do vetor: (0-7) 25 12 **37** 48 57 86 33 92
  a,b↑

QuickSort: Exemplo

- configuração atual do vetor: (0-7) 25 12 **37** 48 57 86 33 92
  a,b↑
- determine a posição correta de x=v[0]=25
  - de a=2 para o final: 37>25 (a=2)
  - de b=2 para o início: 37>25 e 12<=25(b=1)
- os índices a e b se cruzaram, com b<a : (0-7) 25 **12 37** 48 57 86 33 92
  b↑ a↑
  - todos os elementos de 37 (inclusive) para o final são maiores que 25 e
  - todos os elementos de 12 (inclusive) para o início são menores que 25 –com exceção de 25
- troque o pivô v[0]=25com v[b]=12, o último dos valores menores que 25 encontrado
- nova configuração do vetor, com o pivô 25 na posição correta:

(0-7) 12 2537 48 57 86 33 92

QuickSort: Exemplo

- dois vetores menores para ordenar:

1. valores menores que 25:
    - (0-0) 12 : vetor já está ordenado pois possui apenas um elemento
2. valores maiores que 25:
    - (2-7) 37 48 57 86 33 92 : vetor pode ser ordenado de forma semelhante, com 37 como pivô

```
public void quickSort(int v[], int a, int b)

se a < b então
    int indicePivo ← particiona(v, a, b)
    quickSort(v, a, indicePivo − 1)
    quickSort(v, indicePivo + 1, b)
```

Algorithm 1: Algoritmo de quickSort (índices 0 a n-1)

```
private int particiona(int v[], int a, int b)

int x ← v[a]
enquanto a < b faça
    enquanto v[a] < x faça  a++;
    enquanto v[b] > x faça  b−−;
    troca(v, a, b)
retorna a
```

Algorithm 2: Procedimento Particiona

```
private void troca(int v[], int a, int b)

int temp ← v[a]
v[a] ← v[b]
v[b] ← temp
```

Algorithm 3: Procedimento “troca”

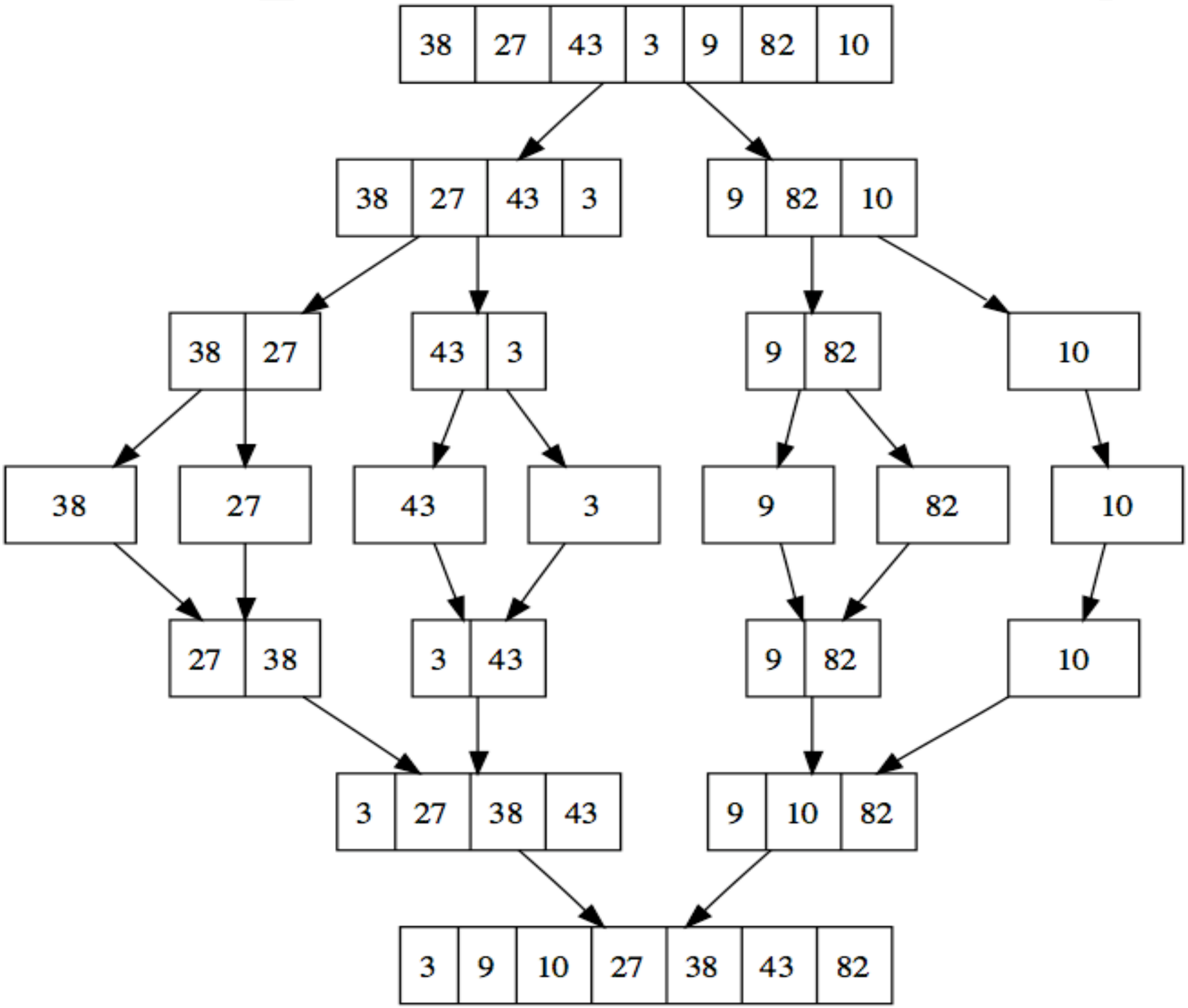
Mergesort

- Abordagem de dividir para conquistar
- Dividir o problema em um determinado número de sub-problemas
- Conquistar os sub-problemas, resolvendo-os recursivamente
- Combinar as soluções dadas aos sub-problemas

MergeSort

- Visão geral: o mergesort divide um vetor pela metade, então usa-se a função **merge** para unir cada metade produzindo um terceiro vetor já classifcado. Então:
  - dividir o vetor a ser classificado ao meio: a cada chamada recursiva do mergesort o vetor é dividido em duas partes até que resultem apenas dois vetores com apenas um único elemento
  - classificar cada metade: classificação simples entre 2 elementos
  - unir as duas metades já classificadas: ao retorno de cada chamda recursiva chama a função Merge para unir os dois vetores em um já classificado
  - O MergeSort possui ordem de complexidade O(n log(n))

MergeSort - Exemplo



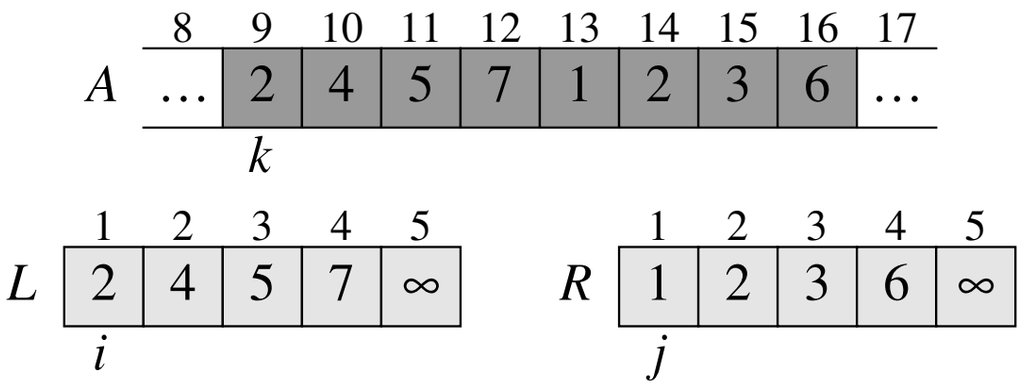
Mergesort

```
MERGE-SORT(A, p, r)
1   if p < r
2       then q ← ⌊(p + r)/2⌋
3            MERGE-SORT(A, p, q)
4            MERGE-SORT(A, q + 1, r)
5            MERGE(A, p, q, r)
```

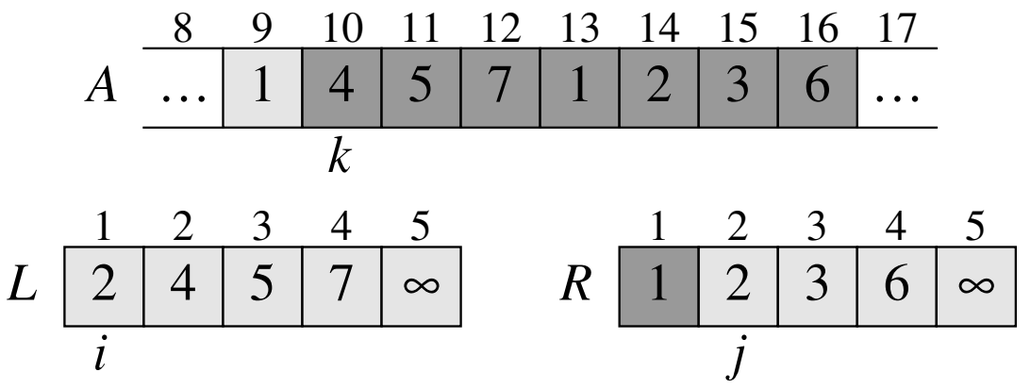
Ref.: CORMEN, Thomas H. Algoritmos: teoria e prática. Rio de Janeiro: Campus, 2002

```
MERGE(A, p, q, r)
 1  n1 ← q − p + 1
 2  n2 ← r − q
 3  create arrays L[1 .. n1 + 1] and R[1 .. n2 + 1]
 4  for i ← 1 to n1
 5        do L[i] ← A[p + i − 1]
 6  for j ← 1 to n2
 7        do R[j] ← A[q + j]
 8  L[n1 + 1] ← ∞
 9  R[n2 + 1] ← ∞
10  i ← 1
11  j ← 1
12  for k ← p to r
13        do if L[i] ≤ R[j]
14              then A[k] ← L[i]
15                   i ← i + 1
16              else A[k] ← R[j]
17                   j ← j + 1
```

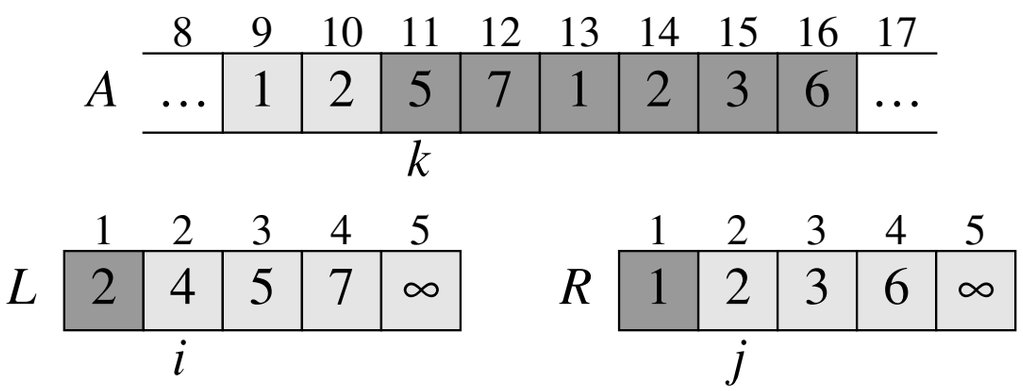
Exemplo: Merge(A, 9, 12, 16)



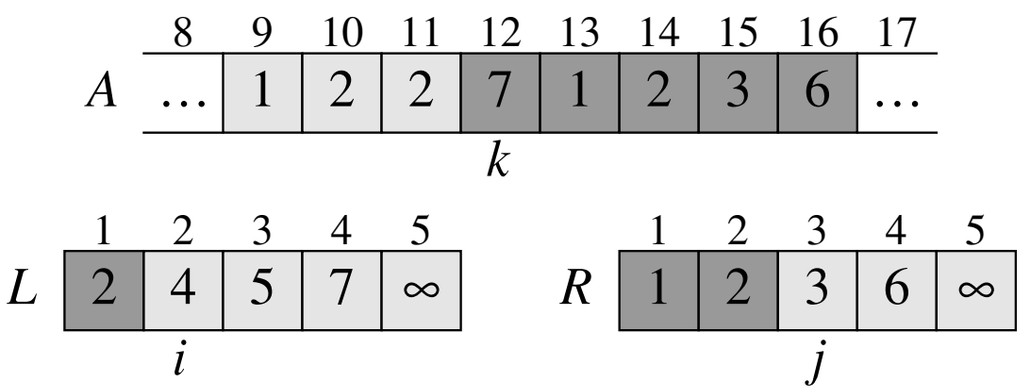
(a)



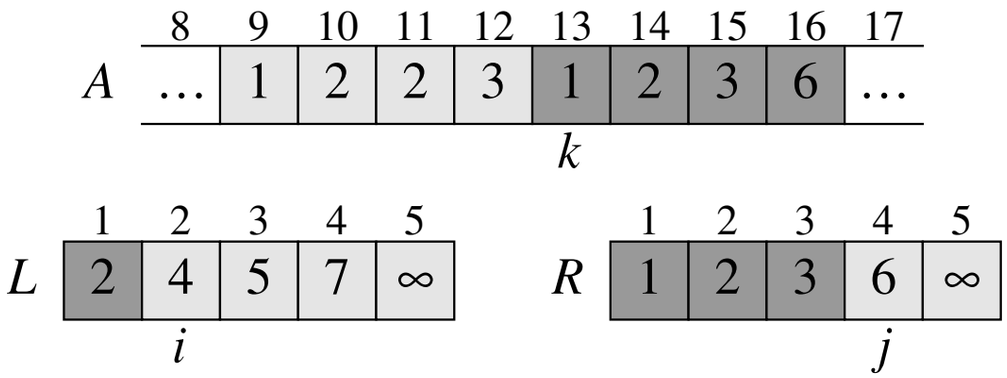
(b)



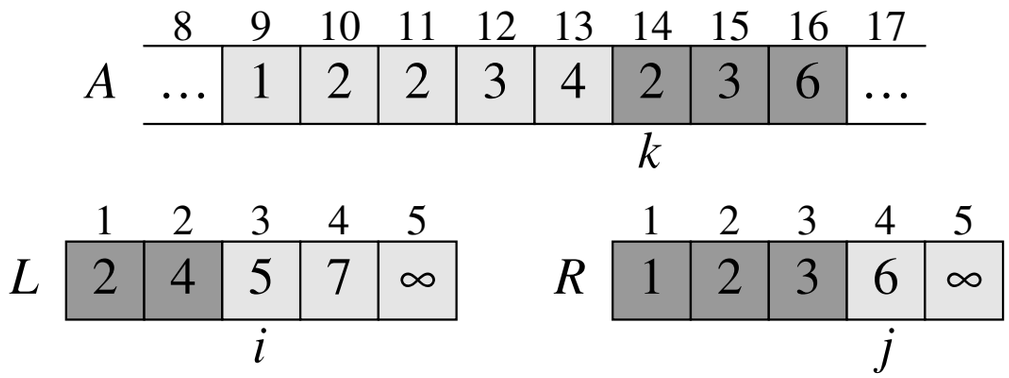
(c)



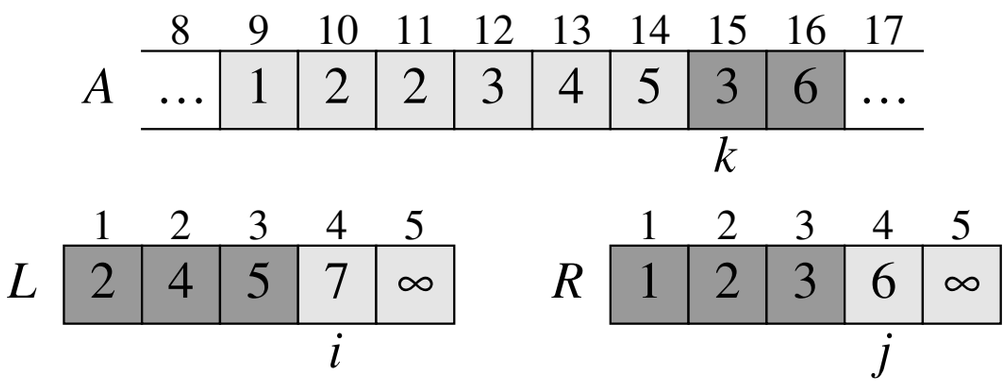
(d)


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 1 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

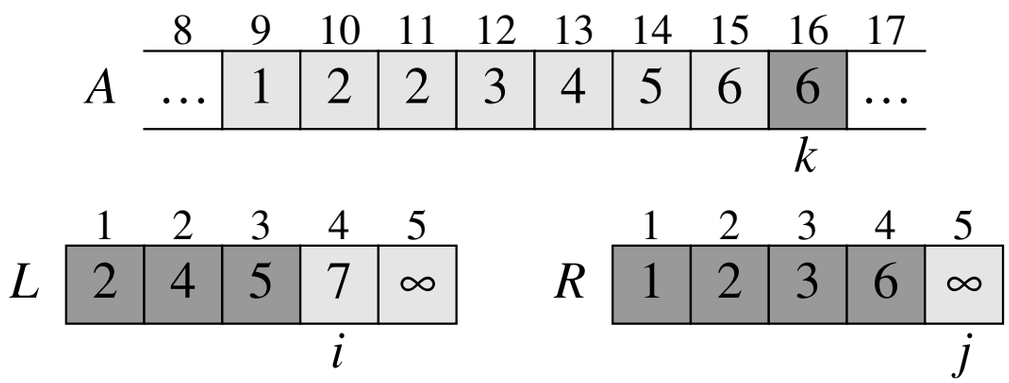
(e)


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

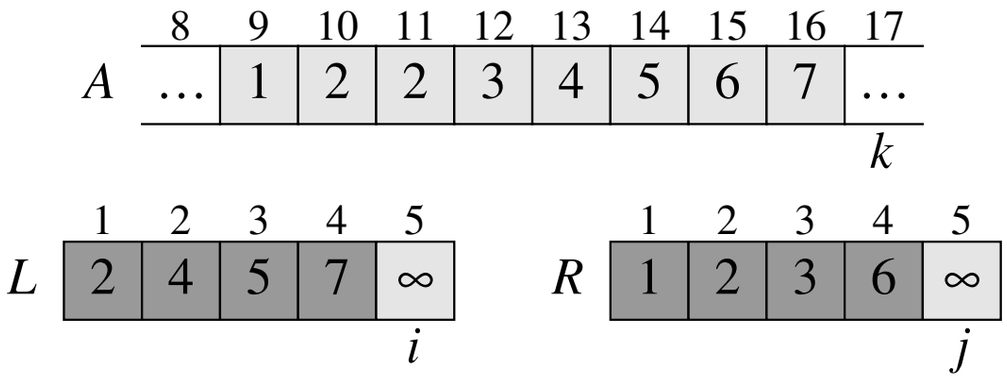
(f)


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

(g)


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 6 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

(h)


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 6 7 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

(i)

Mergesort

- Pseudo código adaptado para indexação de arrays de 0 a n-1
- A: array a ser ordenado
- p: índice do primeiro elemento a ser ordenado (inicia com 0)
- r: índica do último elemento a ser ordenado (inicia com n-1)

Algoritmo: MergeSort(A, p, r)

se $p < r$ **então**
    $q \leftarrow \lfloor (p + r)/2 \rfloor$
    MergeSort(A, p, q)
    MergeSort$(A, q + 1, r)$
    Merge(A, p, q, r)

Algoritmo 1: Algoritmo de MergeSort (índices 0 a n-1)

Algoritmo: Merge(A, p, q, r)

$n_1 \leftarrow q - p + 1$
$n_2 \leftarrow r - q$
Crie arrays $L[0...n_1]$ e $R[0...n_2]$
para $i \leftarrow 0$ **até** $n_1 - 1$ **hacer** $L[i] \leftarrow A[p + i]$;
para $j \leftarrow 0$ **até** $n_2 - 1$ **hacer** $R[j] \leftarrow A[q + j + 1]$;
$L[n_1] \leftarrow \infty$
$R[n_2] \leftarrow \infty$
$i \leftarrow 0$
$j \leftarrow 0$
para $k \leftarrow p$ **até** r **hacer**
    se $L[i] \leq R[j]$ **então**
        $A[k] \leftarrow L[i]$
        $i++$
    senão
        $A[k] \leftarrow R[j]$
        $j++$

Algoritmo 2: Procedimento Merge (índices 0 a n-1)