Conteúdo

1. Introdução
2. Listas
3. Pilhas
4. Filas
5. Árvores
   - Árvore Binária e Árvore AVL
   - Árvore N-ária e Árvore B
6. Tabelas de Dispersão (Hashing)
7. Métodos de Acesso a Arquivos
8. Métodos de Ordenação de Dados

Métodos de Ordenação de Dados

(continuação)

Métodos de Ordenação de Dados

Métodos de Ordenação mais conhecidos

- Selection Sort
- Bubble Sort
- Insertion Sort

} métodos simples

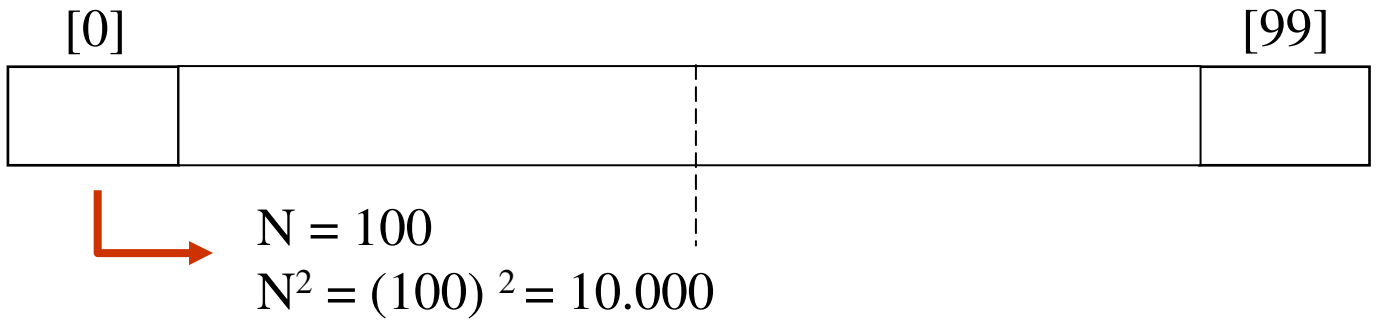
- Merge Sort
- Quick Sort
- Heap Sort

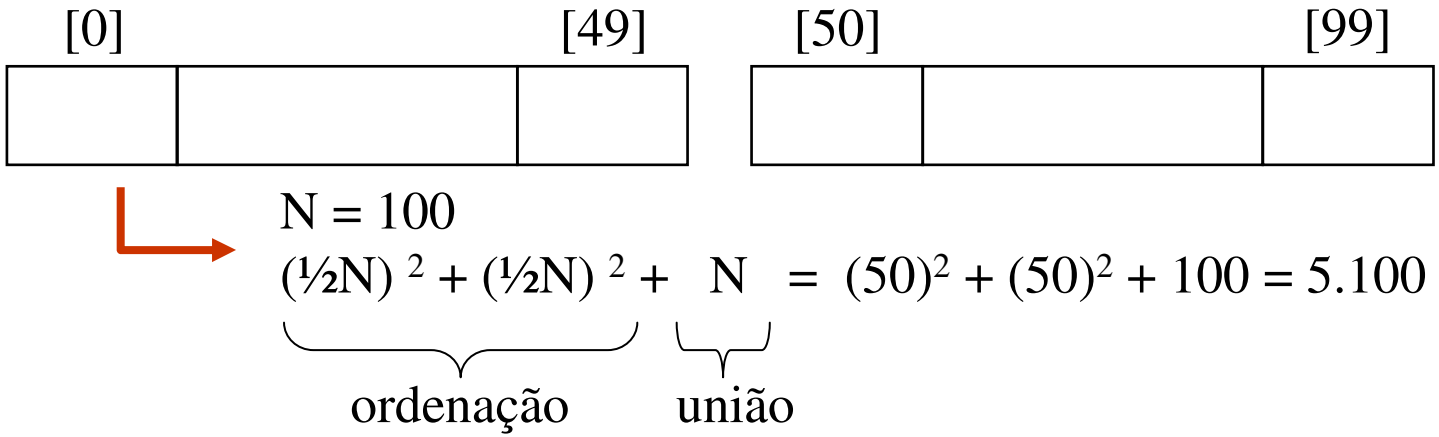
Complexidade dos Métodos Simples

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

Métodos de Divisão e Conquista

Divisão e Conquista: um array é dividido em dois segmentos; cada segmento é ordenado e, então, eles são unidos de novo.





Métodos de Divisão e Conquista

Complexidade dos Métodos de Divisão e Conquista: $O(N \log_2 N)$

Alguns Métodos de Divisão e Conquista:

- Merge Sort
- Quick Sort

Método de Ordenação

Merge Sort

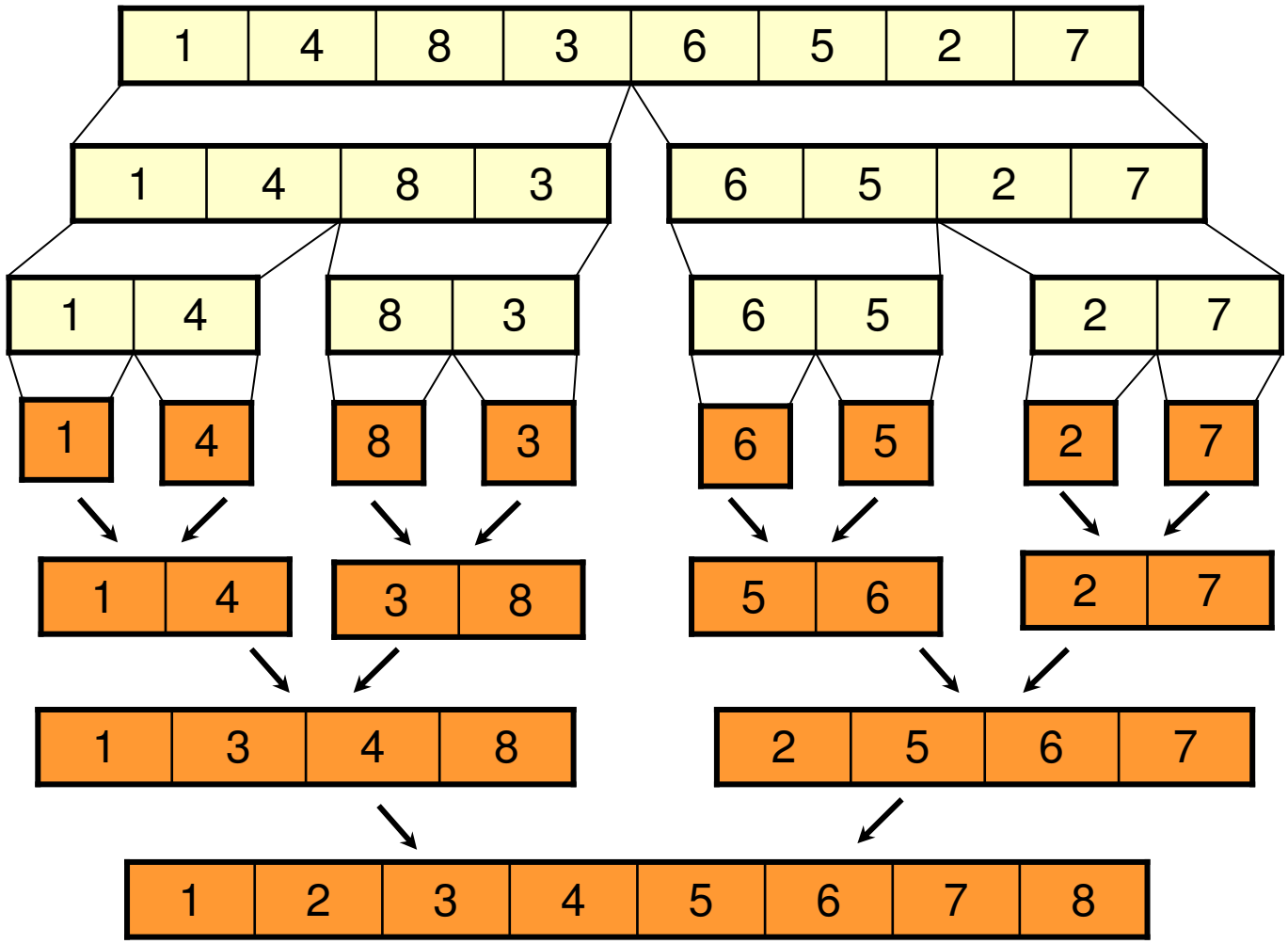
Merge Sort

Baseia-se em junções sucessivas (merge) de 2 seqüências ordenadas em uma única seqüência ordenada

Aplica um método "divisão e conquista"

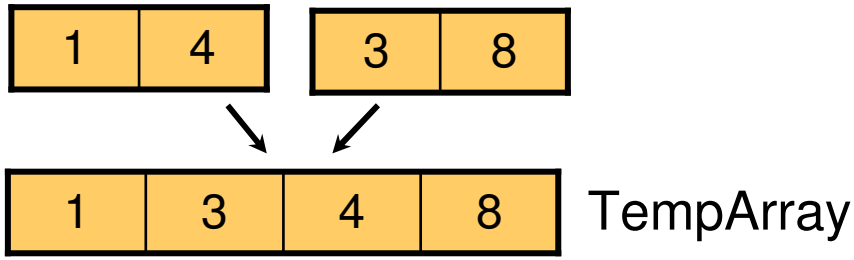
- divide o array em 2 seqüências de comprimento $\lfloor n/2 \rfloor$ e $\lceil n/2 \rceil$
- ordena recursivamente cada seqüência (dividindo novamente, quando possível)
- faz o *merge* das 2 seqüências ordenadas para obter o array ordenado completo

Merge Sort

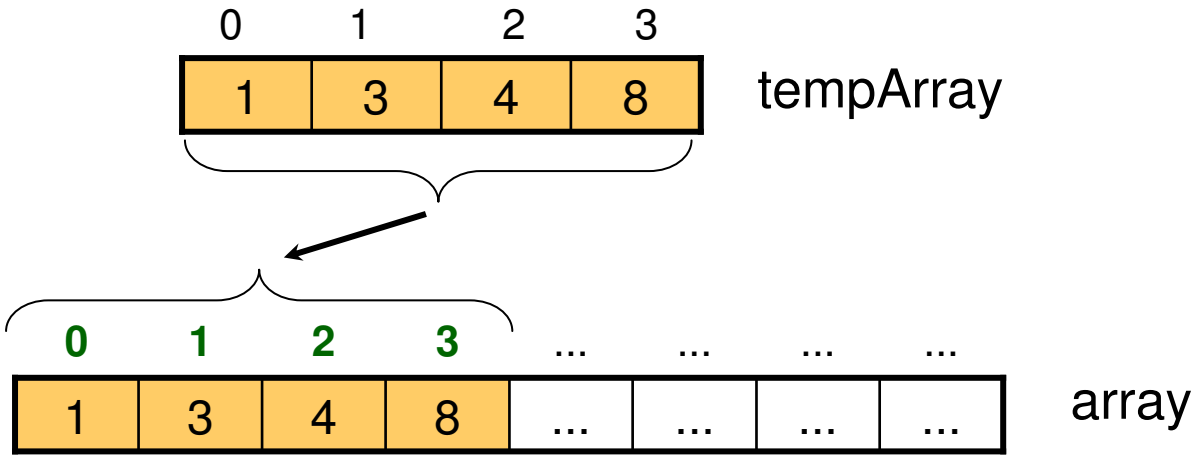

1 4 8 3 6 5 2 7
1 4 8 3
6 5 2 7
1 4
8 3
6 5
2 7
1
4
8
3
6
5
2
7
1 4
3 8
5 6
2 7
1 3 4 8
2 5 6 7
1 2 3 4 5 6 7 8

Merge - Algoritmo

- Utiliza um array temporário (tempArray) para manter o resultado da ordenação das 2 partes



- Após a ordenação, o conteúdo de tempArray é transferido para o array



Merge Sort - Algoritmo

MergeSort () → método recursivo

se o array tem pelo menos 2 elementos

divide o array na metade

MergeSort a metade da esquerda (recursivo)

MergeSort a metade da direita (recursivo)

Merge (junta) as duas partes ordenadas em um array ordenado

Merge - Algoritmo

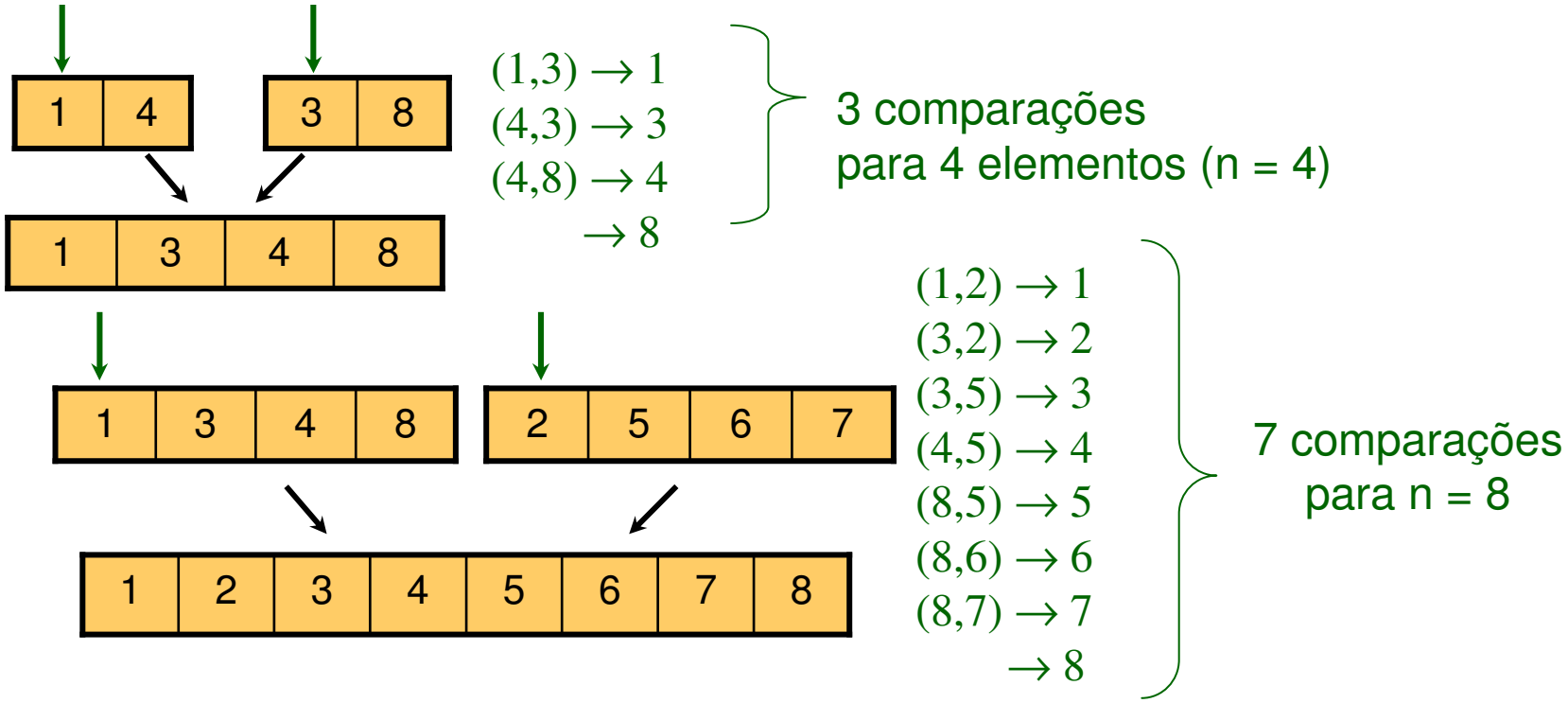
```
Merge  (primEsq, ultEsq, primDir, ultDir)
 indice ← primEsq
 enquanto existem mais elementos na metade esquerda E
                  existem mais elementos na parte direita
    se array[primEsq] < array[primDir] então
         tempArray[indice] ← array[primEsq]
         primEsq ++
    else
         tempArray[indice] ← array[primDir]
         primDir ++
    incrementa indice
 copiar os elementos restantes da metade esquerda para tempArray
 copiar os elementos restantes da metade direita para tempArray
 copiar os elementos ordenados de tempArray de volta para array
```

Merge Sort - Exemplo

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
54	8	47	10	12	70	33	61	48	53

Merge Sort - Complexidade

- Número de comparações?



  - máximo de comparações: n - 1

- Complexidade: ***O(N)***

Merge Sort - Complexidade

- Custo da divisão do array original (número de níveis): O(log N)

- O número de níveis de chamadas do *merge* é igual ao número de vezes que o array original pode ser dividido pela metade: logN

- Em cada nível, são feitas N-1 comparações: O(N)

  ↳ custo total da operação Merge é O(N logN)

⇨ Custo do algoritmo Merge Sort (para qualquer caso):

O(log N + N log N) = O(N logN)

Merge Sort - Desvantagem

- Requer um array auxiliar que é tão grande quanto o array original a ser ordenado.

  - Se espaço é um fator crítico e o array é grande, o algoritmo Merge Sort não é uma boa escolha.

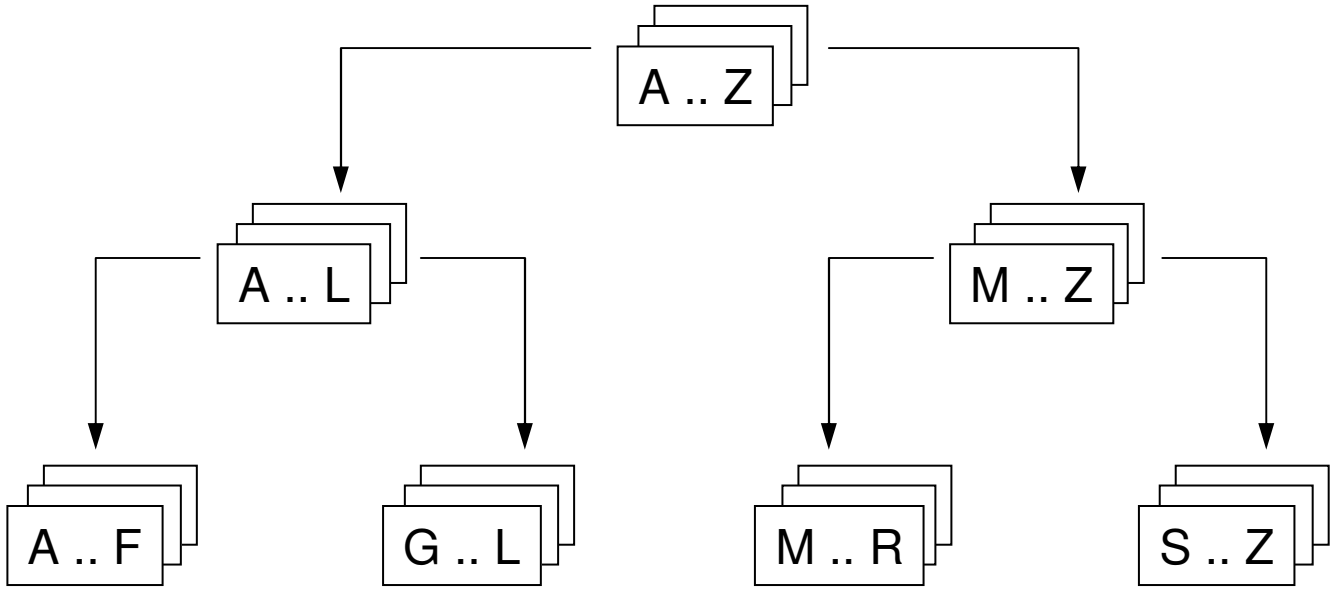
Método de Ordenação

Quick Sort

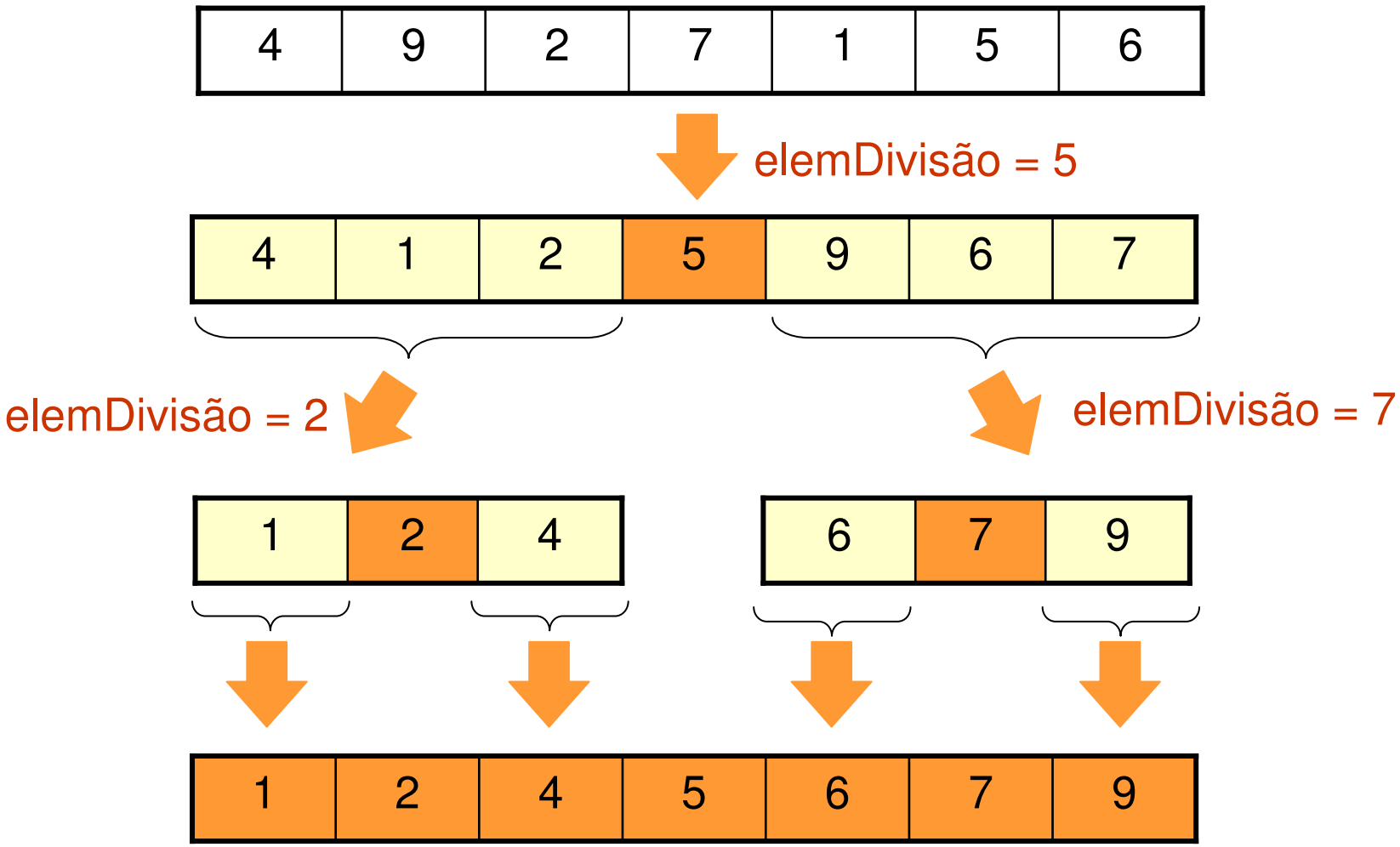
Quick Sort

Ordena através de sucessivas trocas entre pares de elementos do array.

Exemplo: ordenar uma grande pilha de exames pelo nome



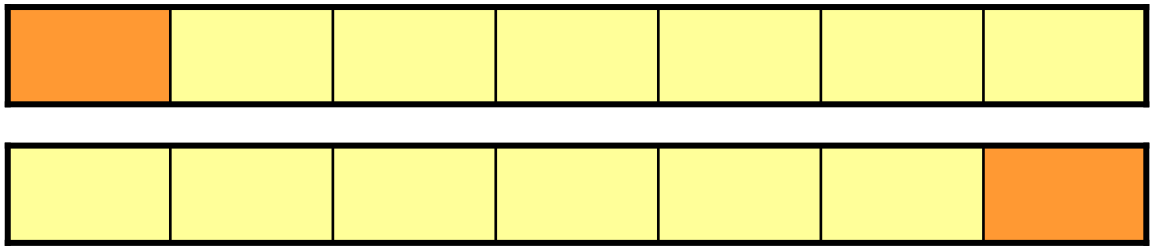
Quick Sort


4 9 2 7 1 5 6
elemDivisão = 5
4 1 2 5 9 6 7
elemDivisão = 2
elemDivisão = 7
1 2 4
6 7 9
1 2 4 5 6 7 9

Quick Sort

A escolha do elemDivisão é crítica para o desempenho

- pior escolha: gera um segmento com tamanho 0 e outro com tamanho n-1

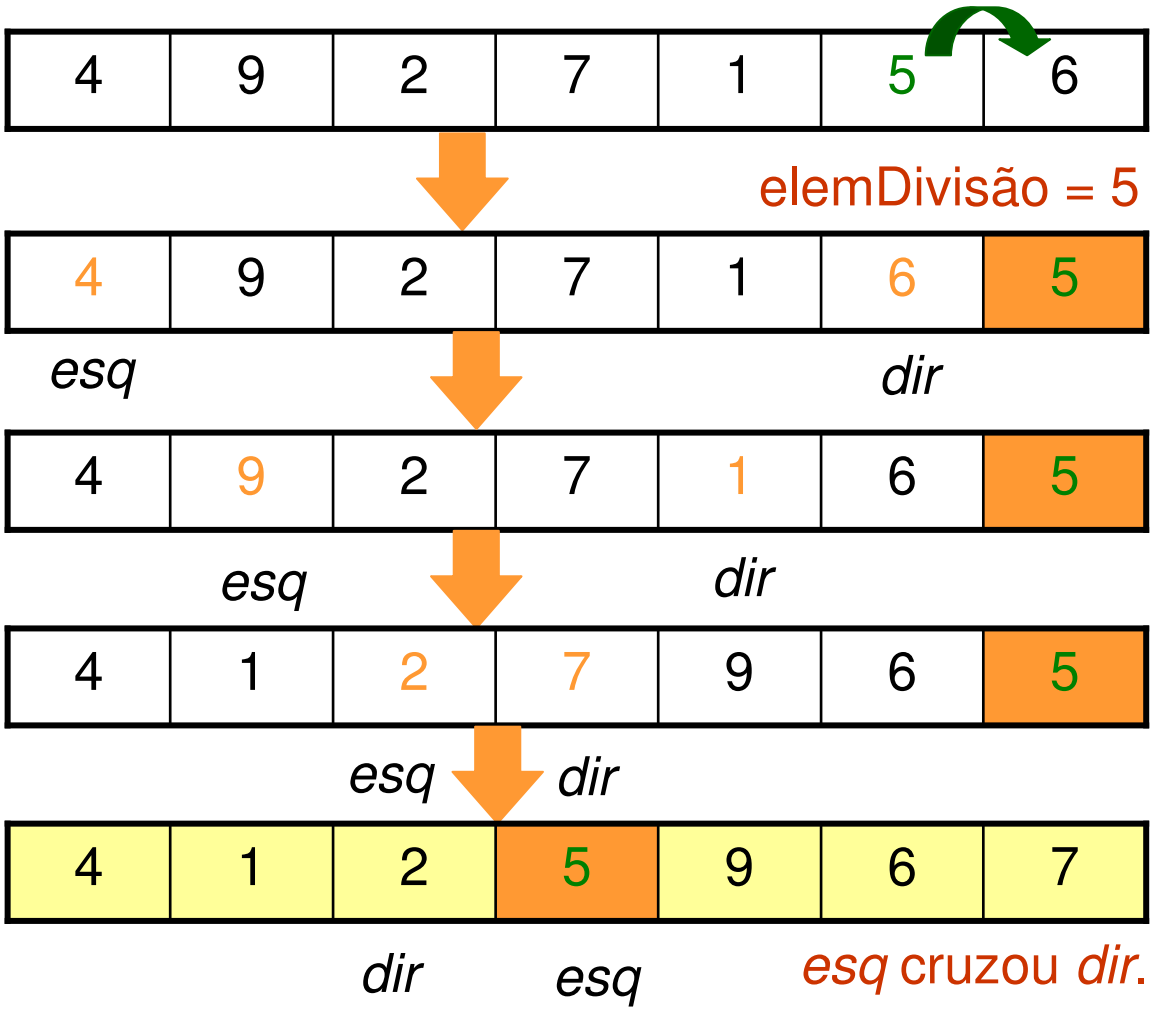


- melhor escolha: gera segmentos balanceados

melhor elemDivisão: elemento mais próximo da média de valores dos elementos

complexidade para determiná-lo: ***O(N)***

Quick Sort


4 9 2 7 1 5 6
elemDivisão = 5
4 9 2 7 1 6 5
esq
dir
4 9 2 7 1 6 5
esq
dir
4 1 2 7 9 6 5
esq
dir
4 1 2 5 9 6 7
dir
esq
esq cruzou dir.

Quick Sort - Algoritmo

QuickSort ()

Se existe mais de um elemento em array[prim] .. array[ult]

- Split
  - seleciona elemDivisao
  - dividir o array de maneira que
    - array[prim]..array[posDivisao-1] ≤ elemDivisao
    - array[posDivisao] = elemDivisao
    - array[posDivisao+1]..array[ult] ≥ elemDivisao
- QuickSort a metade da esquerda
- QuickSort a metade da direita

Quick Sort - Algoritmo

Split

- Transfere o elemDivisão para uma das extremidades do array (direita, p.ex.)
- Utiliza apontadores (*esq* e *dir*) que partem das extremidades do array e se deslocam para o centro do vetor, trocando elementos quando necessário
- O deslocamento termina quando um apontador cruza o outro
- Transfere-se o elemDivisão para a posição apontada por *esq*

Quick Sort - Exemplo

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
54	8	47	10	12	70	33	61	48	53

Quick Sort - Complexidade

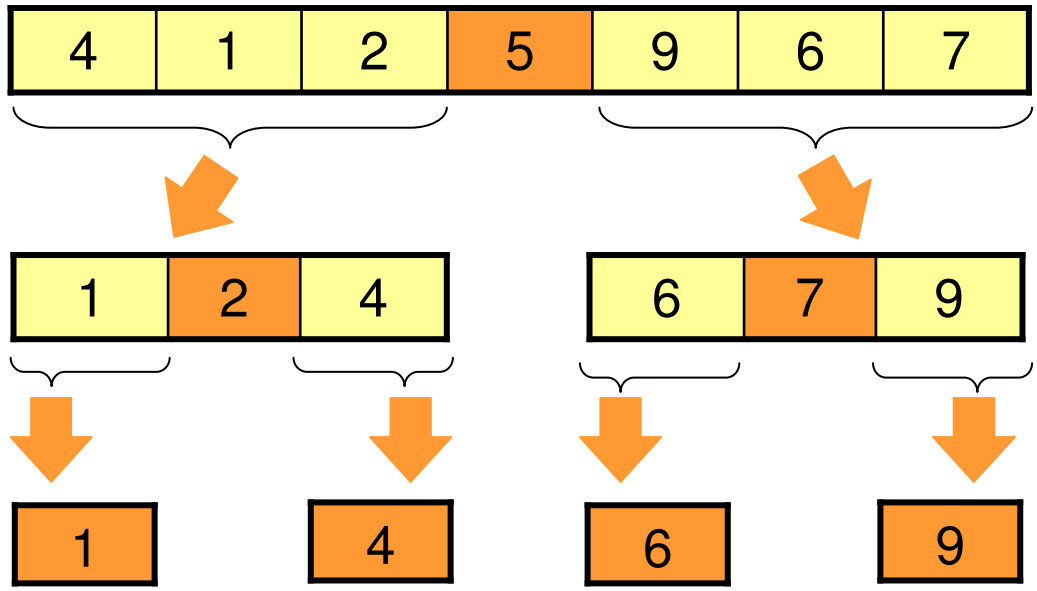
- Melhor caso: segmentos balanceados

  → elemDivisão é o elemento com valor mais próximo da média

1ª V: n – 1 comp.

2ª V: ≈ n – 1 comp.

3ª V: ≈ n – 1 comp.

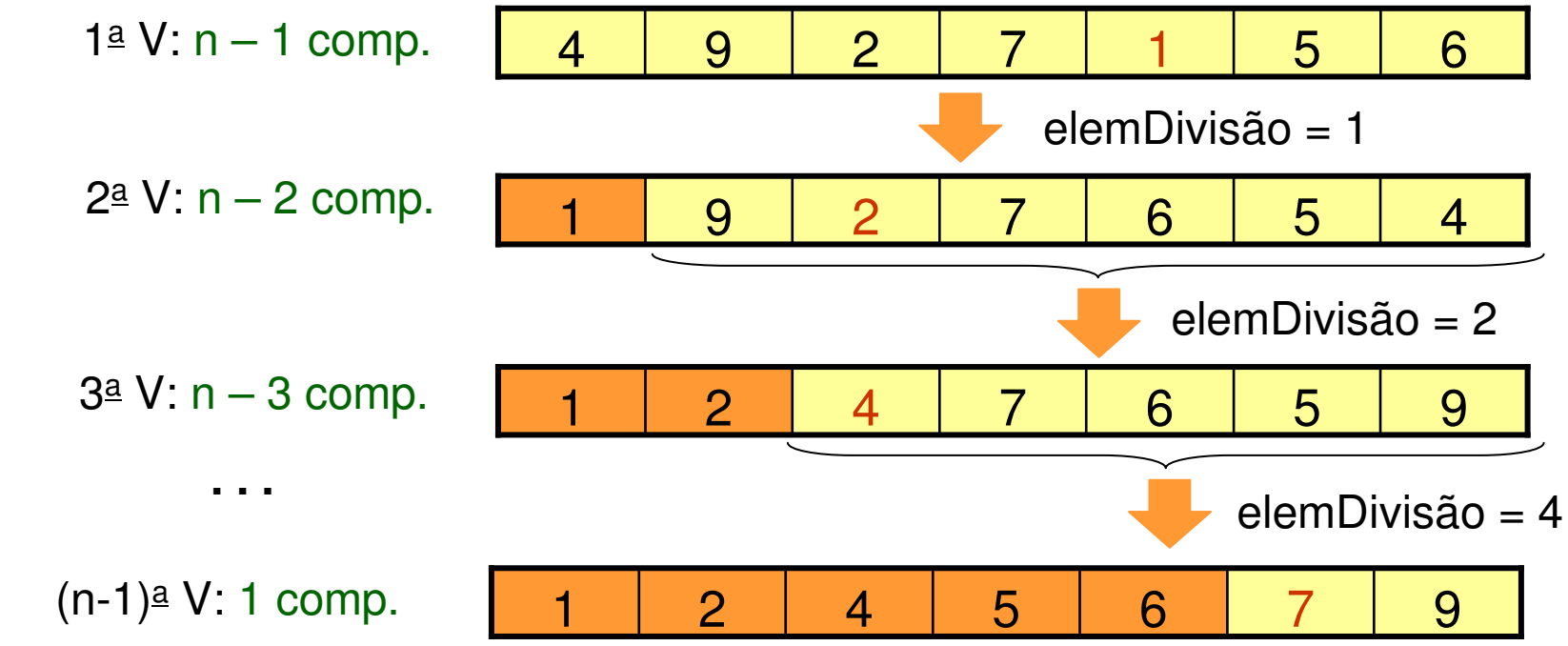


(log n + 1) (n - 1) ⇒ *O(N log N)*

Quick Sort - Complexidade

- Pior caso: segmentos totalmente desbalanceados

  → elemDivisão é o menor elemento do array



...

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

Quick Sort - Complexidade

⇨ Custo do algoritmo para o melhor caso:

- Custo da divisão do array original (número de níveis): O(log N)
- Em cada nível, são feitas N-1 comparações: O(N)

  ↳ custo = O(N logN)

⇨ Custo do algoritmo para o pior caso:

- Custo da divisão do array original (número de níveis): O(N)
- Em cada nível *i*, são feitas N-*i* comparações: O(N)

  ↳ custo = $O(N^2)$

Comparação

	melhor caso	pior caso
Selection Sort	$O(N^2)$	$O(N^2)$
Bubble Sort	$O(N^2)$	$O(N^2)$
Insertion Sort	$O(N)$	$O(N^2)$
Merge Sort	$O(N \log N)$	$O(N \log N)$
Quick Sort	$O(N \log N)$	$O(N^2)$

Comparação de N^2 e N log N

N	N^2	N log N
32	1.024	160
64	4.096	384
128	16.384	896
256	65.536	2.048
512	262.144	4.608
1.024	1.048.576	10.240
2.048	4.194.304	22.528
4.096	16.777.216	49.152

Métodos de Ordenação

Simulação de funcionamento

http://math.hws.edu/TMCM/java/xSortLab

Implementação

- Implementar os seguintes algoritmos de acordo com a hierarquia apresentada a seguir:
  - Merge Sort
  - Quick Sort

Implementação

Hierarquia das Classes de Ordenação