Algoritmos e Estruturas de Dados

Vetores: Ordenação

Prof. Maiquel de Brito
maiquel.b@ufsc.br

Ordenação de Vetores

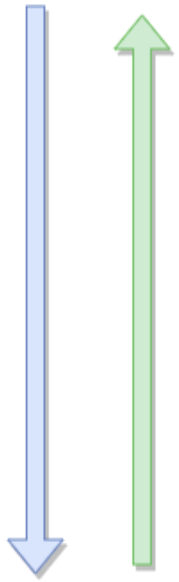
Problema: dado um vetor (*v*) com **N** elementos, rearranjar esses elementos por ordem crescente.

- Entrada: vetor com elementos a serem ordenados
- Saída: mesmo vetor com elementos na ordem especificada

- Ordenação:
  - Pode ser aplicado a qualquer dado com ordem bem definida (inclusive *structs*)
  - Ordenação de *structs:* a ordenação é baseada em um ou mais campos

Algoritmos de Ordenação

Ordenação por inserção (*insertion sort*)
Ordenação por seleção (*selection sort*)
BubbleSort
ShellSort
MergeSort
QuickSort
HeapSort



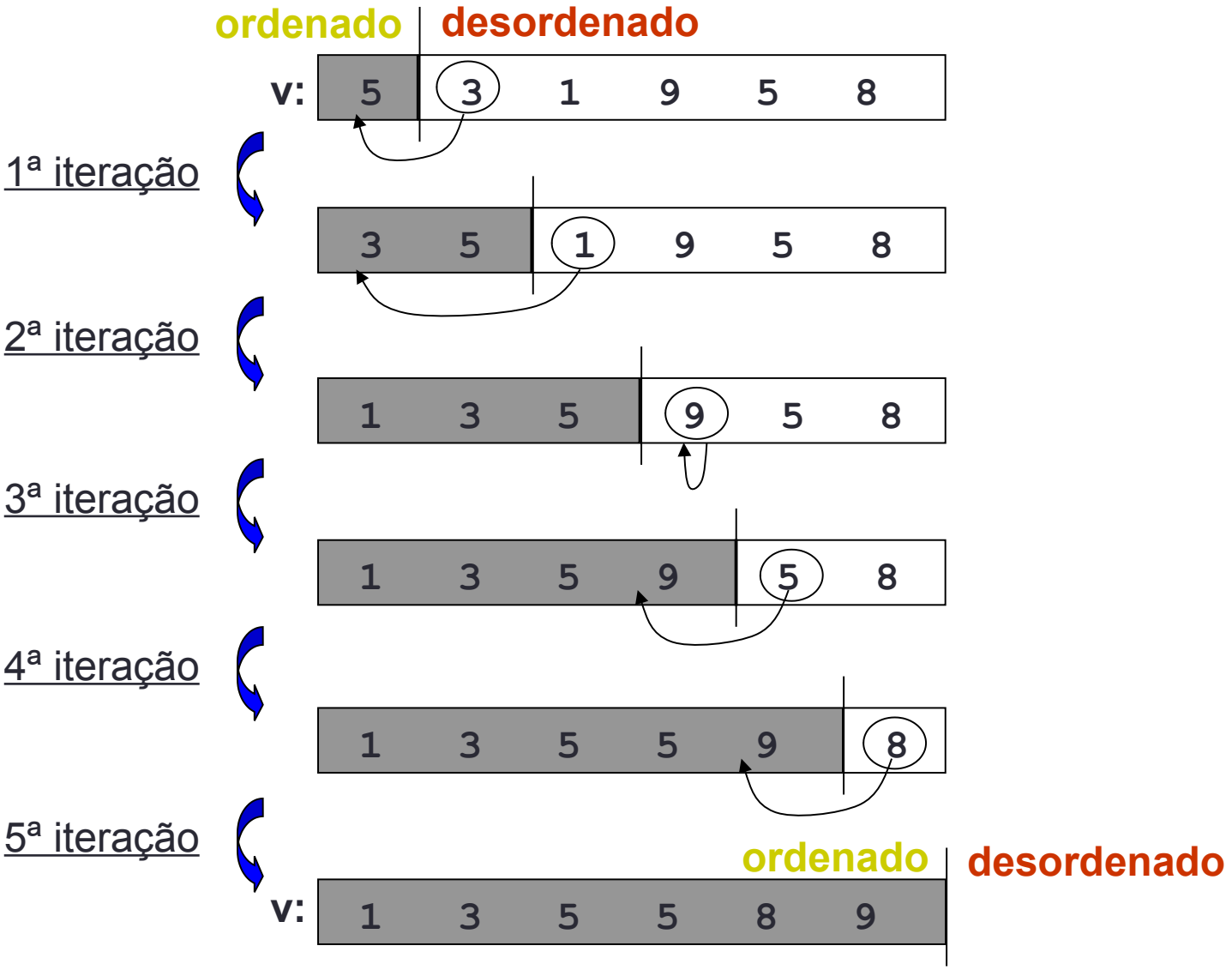
Ordenação por Inserção

Percorre-se um vetor de elementos da esquerda para a direita e à medida que avança vai deixando os elementos mais à esquerda ordenados

Algoritmo

- Considera-se o vetor dividido em dois sub-vetores (esquerdo e direito), com o da esquerda ordenado e o da direita desordenado
- Começa-se com um elemento apenas no sub-vetor da esquerda
- Move-se um elemento de cada vez do sub-vetor da direita para o sub-vetor da esquerda, inserindo-o na posição correta de forma a manter o sub-vetor da esquerda ordenado
- Termina-se quando o sub-vetor da direita fica vazio

Ordenação por Inserção



Ordenação por Inserção (pseudocódigo)

Entrada: vetor *v* com *n* elementos (i.e. *v[0..n-1]*)

```
para i de 1 até n-1{
   Atual = v[i]
   j = i
   enquanto (j > 0) & (v[j-1] > atual){
      v[j] = v[j-1]
      j = j - 1
   }
   v[j] = atual
}
```

Ordenação por Inserção - Análise

Pior caso: vetor em ordem inversa:

cada elemento da parte não ordenada tem que ser trocado com todos os elementos da parte ordenada

Para um vetor de *n* elementos:

- 2º elemento faz uma troca;
- 3º elemento faz duas trocas;
- . . .
- *n*º elemento faz n-1 troca

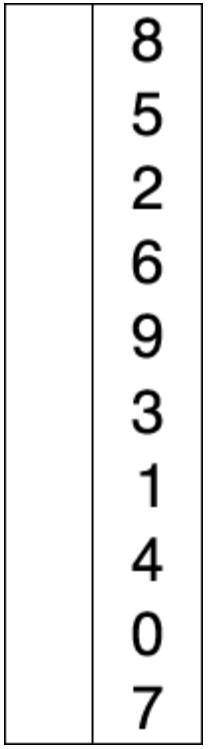
$$\sum_{i=1}^{n-1} i = (n-1) \times (n/2) = \frac{(n^2 - n)}{2} = O(n^2)$$

Ordenação por Seleção

Estratégia: seleciona o menor elemento do vetor, depois o segundo menor, depois o terceiro menor, e assim por diante

Em cada etapa F:

- Procura-se (sequencialmente) a posição M com o menor elemento guardado nas posições de F a N;
- Troca-se o valor guardado na posição F com o valor guardado na posição M (excepto se M for igual a F)



Ordenação por Seleção (vetor na vertical)

Índice	início	**passo 1**	**passo 2**	**passo 3**	**passo 4**	**passo 5**	**passo 6**	**passo 7**
0	7	7	2	2	2	2	2	2
1	21	21	21	7	7	7	7	7
2	10	10	10	10	10	10	10	10
3	15	15	15	15	15	11	11	11
4	2	2	7	21	21	21	13	13
5	13	13	13	13	13	13	21	15
6	11	11	11	11	11	15	15	21

Ordenação por Seleção (pseudocódigo)

Entrada: vetor *v* com *n* elementos (i.e. *v[0..n-1]*)

```
para i de 0 to N-2{
   menor = i
   para j de i+1 to N-1{
      if v[j] < v[menor]
         menor = j
   }
   troca(v[i],v[menor])
}
```

Ordenação por Seleção - Análise

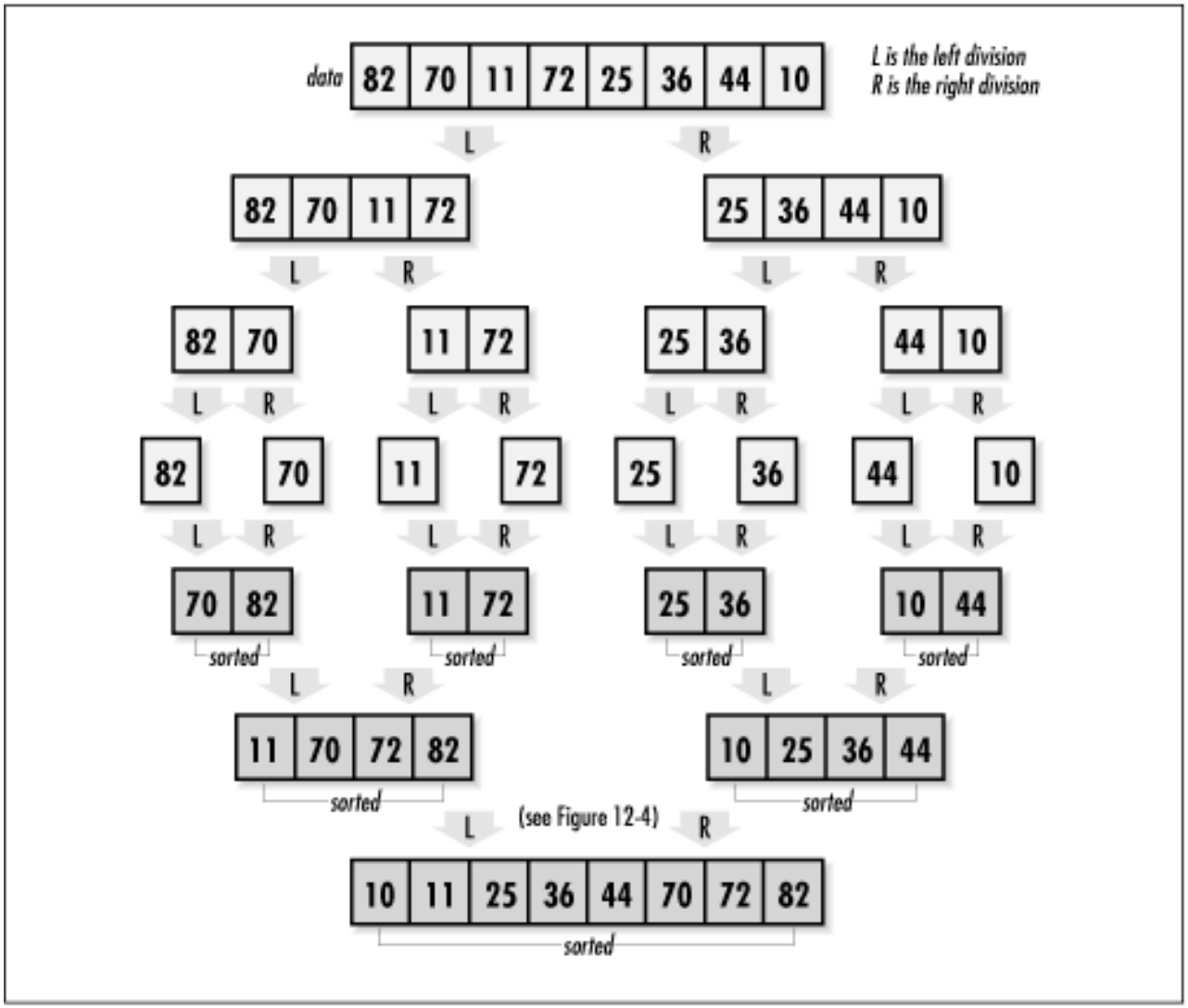
Cada posição é comparada com todas as posições seguintes

Para um vetor de *n* elementos:

- 1º elemento faz n-1 comparações;
- 2º elemento faz n-2 comparações;
- . . .
- *(n*-1)º elemento faz 1 comparação

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

Merge Sort


data
82 70 11 72 25 36 44 10
L is the left division
R is the right division
L
R
82 70 11 72
25 36 44 10
82 70
11 72
25 36
44 10
82
70
11
72
25
36
44
10
70 82
11 72
25 36
10 44
sorted
11 70 72 82
10 25 36 44
(see Figure 12-4)
10 11 25 36 44 70 72 82

Merge Sort - Complexidade

São executados sucessivos processos de fusão. Cada etapa deste processo tem complexidade $O(n)$

A quantidade de processos de fusão corresponde à quantidade de divisões executadas sobre o vetor original, que é $\log_2^n$

Logo, a complexidade do *MergeSort* é $O(n.\log_2^n)$

HeapSort

Em um *max heap* de *n* elementos, troca-se o *1º* e o *nº* elementos.
O 1º elemento é o maior de todos e deve ocupar a última posição em um vetor ordenado.

O *nº* elemento pode ser descartado. O max-heap fica com n-1 elementos.

Todos os elementos do novo vetor formam um *max-heap*, exceto o primeiro

Aplica-se *max-heapify(H,1)* para posicionar o primero elemento adequadamente.

Exemplo: v = [16,14,10,8,7,9,3,2,4,1]

Complexidade: $O(n.log_2 n)$:
n trocas; para cada troca, um *max-heapify* com complexidade $O(n.log_2 n)$

Para mais detalhes em exemplos sobre HeapSort, consultar *Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. Algoritmos: Teoria E Prática. 3 ed.: Campus, 2012. ISBN: 9788535236996.*

Fonte: http://c2.com/cgi/wiki?SortingAlgorithms

Comparativo

	Worst case	Average case	Best case	Extra space	Stable
BubbleSort	O(n^2)	O(n^2)?	O(n)	O(1)	yes
SelectionSort	O(n^2)	O(n^2)	O(n^2)	O(1)	No (i
InsertionSort	O(n^2)	O(n^2)	O(n)	O(1)	yes
BitonicSort	O(n log^2 n)	O(n log^2 n)?	?	O(1)?	?
ShellSort	O(n^2)	O(n log n)?	O(n)	O(1)	no
QuickSort	O(n^2)	O(n log n)	O(n log n)	O(log n)	no
HeapSort	O(n log n)	O(n log n)	O(n log n)	O(1)	no
SmoothSort	O(n log n)	O(n log n)?	O(n)	O(1)	no
MergeSort	O(n log n)	O(n log n)	O(n log n)	O(n)	yes
TimSort	O(n log n)	O(n log n)?	O(n)	O(n)	yes
CountingSort	O(n+k)	O(n+k)	O(n+k)	O(n+k)	yes
RadixSort	O(n+k)	O(n+k)	O(n+k)	O(n+k)	yes
BucketSort	O(n^2)	O(n+k)	?????	O(n*k) or O(n+k)	?
BogoSort	unbounded	O(n!)	O(n)	O(1)	no
SlowSort	O(n^(log n))	O(n^(log n))	O(n^(log n))	O(1)	yes
QuantumBogoSort	O(1)	O(1)	O(1)	O(0)	no

Simuladores de Algoritmos de Ordenação

https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

https://www.toptal.com/developers/sorting-algorithms

https://visualgo.net/bn/sorting