UFSC-CTC-INE

INE5384 - Estruturas de Dados

Ordenação de Dados

Prof. Ronaldo S. Mello

2002/2

Ordenação de Dados

- Processo bastante utilizado na computação de uma estrutura de dados
- Dados ordenados garantem uma melhor performance de pesquisa a uma ED
  - busca seqüencial
    - evita a varredura completa de uma lista de dados
  - busca binária
    - só é possível se os dados estão ordenados
    - apresenta baixa complexidade

Compromisso

- *"A complexidade da ordenação da ED não deve exceder a complexidade da computação a ser feita na ED sem o processo de ordenação"*

- Exemplo: deseja-se realizar uma única pesquisa a um vetor
  - busca seqüencial ⇒ *O(n)*
  - ordenação ⇒ *O(n log n)*
  - *Não vale a pena ordenar!*

Considerações

- Dados estão mantidos em um vetor
- Elemento do vetor
  - objeto que possui um atributo chave que deve ser mantido ordenado
- Um método *troca(x,y)* realiza a troca dos elementos presentes nas posições x e y do vetor
- Para fins de exemplo, números inteiros serão utilizados como elementos

Implementação

```
Classe Ordenador
início
    vetor inteiro[ ];
    n inteiro; /* tamanho do vetor */

    construtor Ordenador (REF v[ ] inteiro);
    início
        n ← v.lenght;
        se n < 1 então Exceção VetorVazio();
        vetor ← v;
        ordena();
        v ← vetor;
    fim;
    método ordena();
    início
    fim;
fim;
```

Implementação

```
Classe Ordenador
início
    . . .

    método troca(x inteiro, y inteiro);
    início
        aux inteiro;

        aux ← vetor[x];
        vetor[x] ← vetor[y];
        vetor[y] ← aux;
    fim;
fim;
```

Métodos de Ordenação

- Ordenação por troca
  - *BubbleSort* (método da bolha)
  - *QuickSort* (método da troca e partição)
- Ordenação por inserção
  - *InsertionSort* (método da inserção direta)
  - *BinaryInsertionSort* (método da inserção direta binária)
- Ordenação por seleção
  - *SelectionSort* (método da seleção direta)
  - *HeapSort* (método da seleção em árvore)
- Outros métodos
  - *MergeSort* (método da intercalação)
  - *BucketSort* (método da distribuição de chave)

Métodos de Ordenação Simples

- São três
  - *BubbleSort*
  - *InsertionSort*
  - *SelectionSort*
- Características
  - fácil implementação
  - alta complexidade
  - comparações ocorrem sempre entre posições adjacentes do vetor

“Revisão” de Somatória

- Propriedade 1 (P1)

$$\sum_{i=1}^{n} i = \frac{n\,(n+1)}{2}$$

- Propriedade 2 (P2)

$$\sum_{i=1}^{n} k\,i = k \sum_{i=1}^{n} i$$

BubbleSort

- *BubbleSort* é um método simples de troca
  - ordena através de sucessivas trocas entre pares de elementos do vetor
- Características
  - realiza varreduras no vetor, trocando pares adjacentes de elementos sempre que o próximo elemento for menor que o anterior
  - após uma varredura, o maior elemento está corretamente posicionado no vetor e não precisa mais ser comparado
    - após a *i-ésima* varredura, os *i* maiores elementos estão ordenados

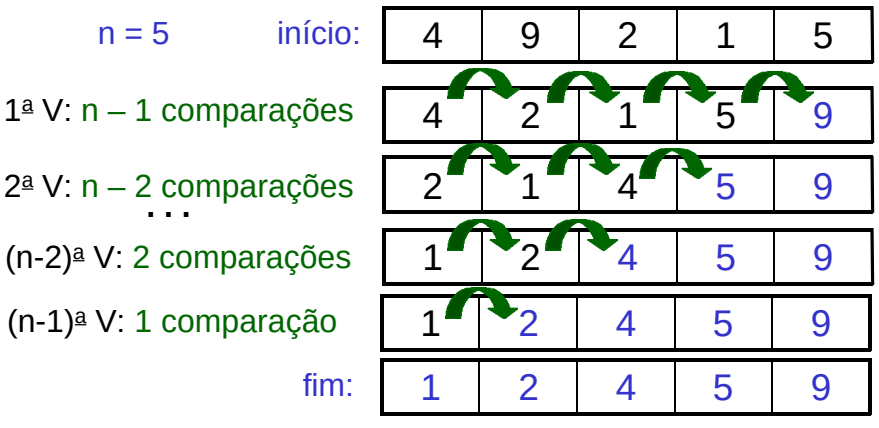
BubbleSort

- Simulação de funcionamento

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

BubbleSort - Complexidade

- Para um vetor de *n* elementos, *n – 1* varreduras são feitas para acertar todos os elementos



BubbleSort - Complexidade

- Definida pelo número de comparações envolvendo a quantidade de dados do vetor
- Número de comparações:

  $(n - 1) + (n – 2) + ... + 2 + 1$
- Complexidade (para qualquer caso):

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

BubbleSort - Implementação

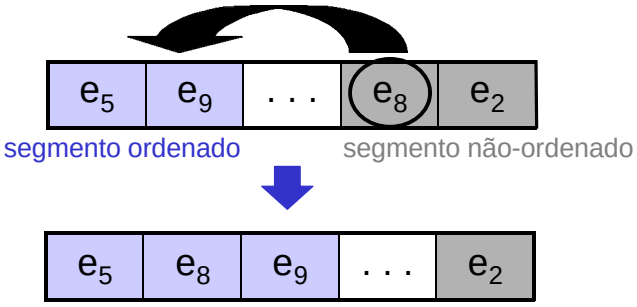
```
Classe OrdenadorBubbleSort
SubClasse de Ordenador
início

   método ordena();
   início
         i, j  inteiro;
         para i de 0 até n-2 faça  /* n-1 varreduras */
           para j de 1 até n-1-i faça            /* desconsidera elementos */
             se vetor[ j -1 ] > vetor[ j ] então   /* a direita já ordenados */
                   troca( j - 1, j);              /* a cada iteração */
   fim;
fim;
```

InsertionSort

- *InsertionSort* é um método simples de inserção
- Características do método de inserção
  - considera dois segmentos (sub-vetores) no vetor: ordenado (aumenta) e não-ordenado (diminui)
  - ordena através da inserção de um elemento por vez (primeiro elemento) do segmento não-ordenado no segmento ordenado, na sua posição correta

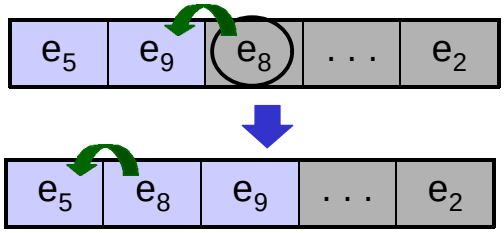
Método de Inserção



- Inicialmente, o segmento ordenado contém apenas o primeiro elemento do vetor

InsertionSort

- realiza uma busca seqüencial no segmento ordenado para inserir corretamente um elemento do segmento não-ordenado
- nesta busca, realiza trocas entre elementos adjacentes para ir acertando a posição do elemento a ser inserido



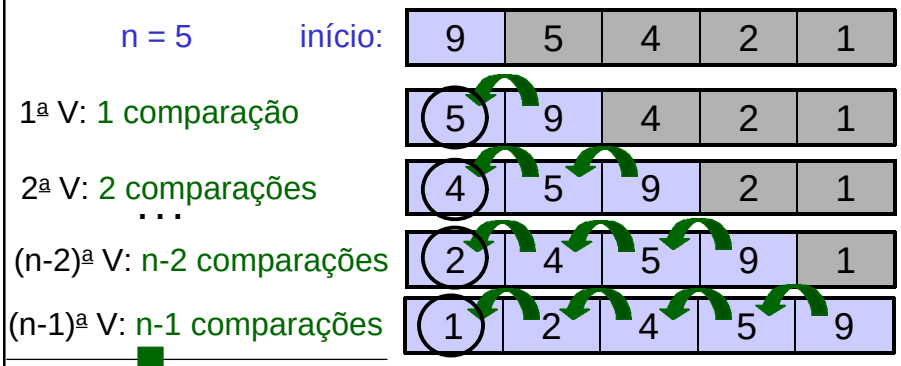
InsertionSort

- Simulação de funcionamento

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

InsertionSort - Complexidade

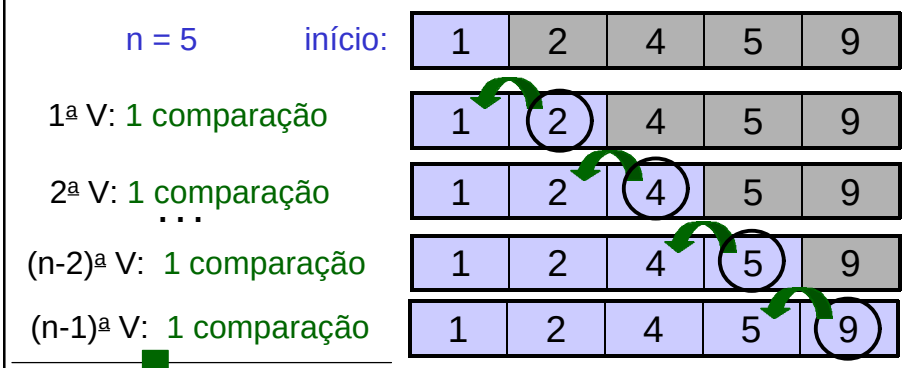
- Pior caso: vetor totalmente desordenado



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

InsertionSort - Complexidade

- Melhor caso: vetor já ordenado



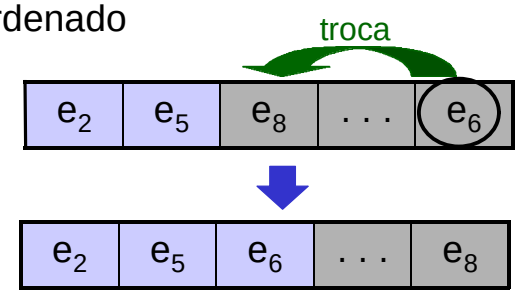
n - 1 $\Rightarrow$ ***O(n)***

InsertionSort X *BubbleSort*

	Melhor caso	Pior caso
InsertionSort	*O(n)*	$O(n^2)$
BubbleSort	$O(n^2)$	$O(n^2)$

SelectionSort

- *SelectionSort* é um método simples de seleção
  - ordena através de sucessivas seleções do elemento de menor valor em um segmento não-ordenado e seu posicionamento no final de um segmento ordenado



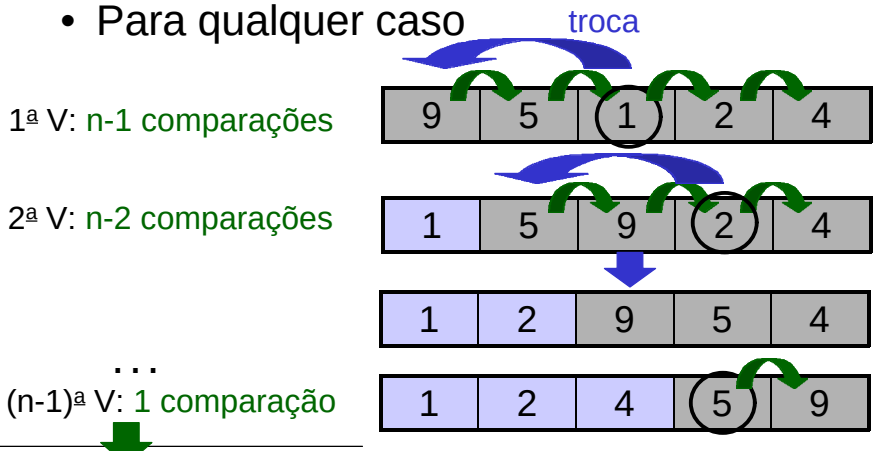
SelectionSort

- Característica particular
  - realiza uma busca seqüencial pelo menor valor no segmento não-ordenado a cada iteração
- Simulação de funcionamento

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

SelectionSort - Complexidade

- Para qualquer caso



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

Comparação

	Melhor caso	Pior caso
InsertionSort	*O(n)*	*O(n^2)*
BubbleSort	*O(n^2)*	*O(n^2)*
SelectionSort	*O(n^2)*	*O(n^2)*

Exercícios

- Implementar o método *sort* para uma subclasse *OrdenadorInsertionSort* da classe *Ordenador*
- Implementar o método *sort* para uma subclasse *OrdenadorSelectionSort* da classe *Ordenador*
- Melhore a complexidade do *BubbleSort* de modo que ele encerre a sua execução quando descobrir que o vetor já está ordenado