Métodos de Ordenação: *Bubble Sort*

Pesquisa, Ordenação e Técnicas de Armazenamento

Prof. Msc. Bruno de A. Iizuka Moritani
bruno.moritani@anhembi.br

Agenda

- Revisão
  - *Selection Sort*
  - *Insertion Sort*
- *Bubble Sort*
  - *Odd-Even Sort*
  - *Cocktail Sort*

Métodos de Ordenação

O Problema da Ordenação

“Dado um conjunto de elementos **desordenados**, identificados por chaves, o objetivo da ordenação é **retornar uma lista de elementos ordenados** segundo uma regra aplicada nas chaves desses elementos.”

71, 194, 38, 1701, 89, 76, 11, 83, 1629, 48, 94, 63, 132, 16, 111, 95, 84, 341, 975,
14, 40, 64, 27, 81, 139, 213, 63, 90, 1120, 8, 15, 3, 126, 2018, 40, 74, 758, 485,
604, 230, 436, 664, 582, 150, 251, 284, 308, 231, 124, 211, 486, 225, 401, 370,
11, 101, 305, 139, 189, 17, 33, 88, 208, 193, 145, 1, 94, 73, 416, 918, 263, 28, 500,
538, 356, 117, 136, 219, 27, 176, 130, 10, 460, 25, 485, 18, 436, 65, 84, 200, 283,
118, 320, 138, 36, 416, 280, 15, 71, 224, 961, 44, 16, 401, 39, 88, 61, 304, 12, 21,
24, 283, 134, 92, 63, 246, 486, 682, 7, 219, 184, 360, 780, 18, 64, 463, 474, 131,
160, 79, 73, 440, 95, 18, 64, 581, 34, 69, 128, 367, 460, 17, 81, 12, 103, 820, 62,
116, 97, 103, 862, 70, 60, 1317, 471, 540, 208, 121, 890, 346, 36, 150, 59, 568,
614, 13, 120, 63, 219, 812, 2160, 1780, 99, 35, 18, 21, 136, 872, 15, 28, 170, 88, 4,
30, 44, 112, 18, 147, 436, 195, 320, 37, 122, 113, 6, 140, 8, 120, 305, 42, 58, 461,
44, 106, 301, 13, 408, 680, 93, 86, 116, 530, 82, 568, 9, 102, 38, 416, 89, 71, 216,
728, 965, 818, 2, 38, 121, 195, 14, 326, 148, 234, 18, 55, 131, 234, 361, 824, 5,
81, 623, 48, 961, 19, 26, 33, 10, 1101, 365, 92, 88, 181, 275, 346, 201, 206, 86,
36, 219, 324, 829, 840, 64, 326, 19, 48, 122, 85, 216, 284, 919, 861, 326, 985,
233, 64, 68, 232, 431, 960, 50, 29, 81, 216, 321, 603, 14, 612, 81, 360, 36, 51, 62,
194, 78, 60, 200, 314, 676, 112, 4, 28, 18, 61, 136, 247, 819, 921, 1060, 464, 895,
10, 6, 66, 119, 38, 41, 49, 602, 423, 962, 302, 294, 875, 78, 14, 23, 111, 109, 62,
31, 501, 823, 216, 280, 34, 24, 150, 1000, 162, 286, 19, 21, 17, 340, 19, 242, 31,
86, 234, 140, 607, 115, 33, 191, 67, 104, 86, 52, 88, 16, 80, 121, 67, 95, 122, 216,
548, 96, 11, 201, 77, 364, 218, 65, 667, 890, 236, 154, 211, 10, 98, 34, 119, 56,
216, 119, 71, 218, 1164, 1496, 1817, 51, 39, 210, 36, 3, 19, 540, 232, 22, 141, 617,
84, 290, 80, 46, 207, 411, 150, 29, 38, 46, 172, 85, 194, 39, 261, 543, 897, 624, 18,
212, 416, 127, 931, 19, 4, 63, 96, 12, 101, 418, 16, 140, 230, 460, 538, 19, 27, 88,
612, 1431, 90, 716, 275, 74, 83, 11, 426, 89, 72, 84, 1300, 1706, 814, 221, 132,
40, 102, 34, 868, 975, 1101, 84, 16, 79, 23, 16, 81, 122, 324, 403, 912, 227, 936,
447, 55, 86, 34, 43, 212, 107, 96, 314, 264, 1065, 323, 428, 601, 203, 124, 95, 216,
814, 2906, 654, 820, 2, 301, 112, 176, 213, 71, 87, 96, 202, 35, 10, 2, 41, 17, 84,
221, 736, 820, 214, 11, 60, 760.

O Problema da Ordenação

- A importância da ordenação na computação está ligada às vantagens de realizarmos buscas em listas ordenadas.

- Uma outra função muito utilizada - o *merge* - também é facilitada por listas ordenadas.

Métodos de Ordenação

- *Insert Sort;*
- *Selection Sort;*
- *Bubble Sort;*
- *Merge Sort ...*
- Alguns algoritmos possuem pequenas variações entre si.
- Curiosidade:
  - Animações dos métodos de ordenação:
    - http://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

Condições para Ordenação

1) A entrada é uma lista desordenada;
2) Ter uma regra para a ordenação;
3) A saída é uma lista ordenada;
4) A saída é uma permutação da entrada do algoritmo.

Classificação

- Complexidade em tempo;
- Complexidade em memória;
  - Elementos auxiliares que precisam ser criados;
- Adaptabilidade;
- Estabilidade;

Complexidades - Classificação

- Complexidade envolve a necessidade de
  - recursos computacionais melhores;
  - demanda de tempo para os algoritmos.

Complexidade em Memória

- Contagem de quanto armazenamento é feito na memória em tempo de execução;
- Usa este número *s* como uma estimativa do espaço de memória do algoritmo;
- Assume-se implicitamente que os espaços de memória de operações diferentes são similares.

Operações Primitivas de Memória em Algoritmos

- Declaração de variáveis;
  - Exemplo:
    - `Int variavel;`
  - `S(n) = 1`
- Declaração de um arranjo;
  - Exemplo:
    - `Int Vetor[10];`
  - `S(n) = n`

Exemplo - Complexidade em Memória

```
Algoritmo buscaBinariaIterativa (int X)
    inicio ← 0                                  S(n) = 1
    fim ← n-1                                   S(n) = 1
    enquanto inicio <= fim faça
        centro ← (inicio+fim)/2                 S(n) = 1
        se x == v[centro] então
            retorna centro
        Fim-se
        se x < v[centro] então
            fim ← centro-1
        Fim-se
        se x > v[centro]então
            inicio ← centro+1
        Fim-se
    Fim-enquanto
    Retorna -1
fim
```

Por que S(n) = 1?
Ao fechar uma iteração a variável é liberada da memória

Pior Caso

$S(n) = 3$

$O(n) = 1$

Exemplo - Complexidade em Memória

```
Algoritmo algoritmo1 (int n)
    i ← 0
    int vetor[n]
    enquanto i <= n faça
        aux ← i
        Aux2 ← aux + 1
        i ← i + 1
         Imprima aux2
     Fim-enquanto
 fim
```

$S(n) = 1$

$S(n) = n$

$S(n) = 1$

$S(n) = 1$

Por que S(n) = 1?
Ao fechar uma iteração a variável é liberada da memória

$S(n) = n + 3$

$O(n) = n$

Adaptabilidade - Classificação

- O quanto o fato da lista estar parcialmente ordenada afeta o desempenho do algoritmo.

32	**45**	**78**	**99**	**42**	**31**	**74**

$$\neq$$

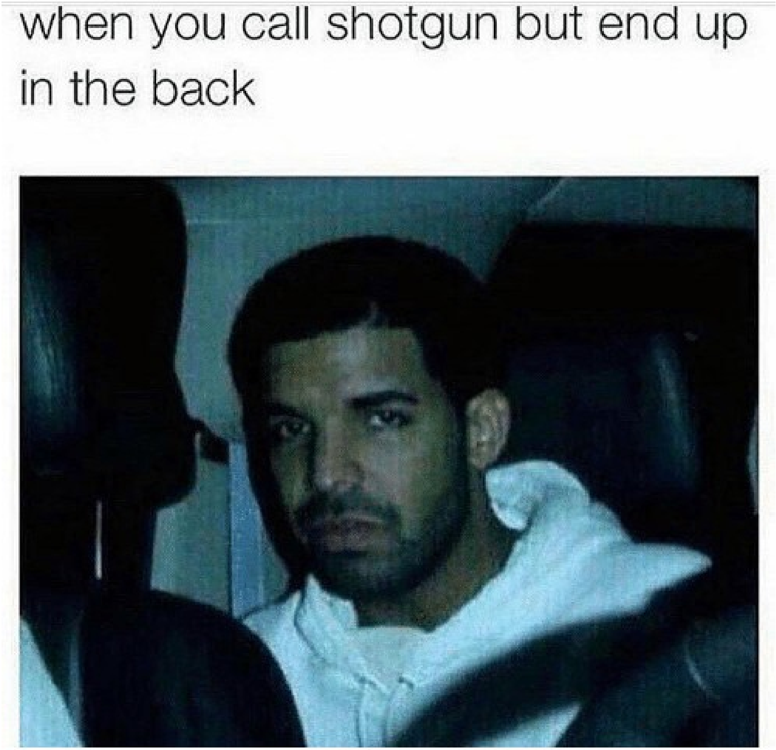
14	**97**	**53**	**99**	**42**	**31**	**74**

Estabilidade - Classificação

- Um algoritmo estável mantém a ordem em que encontrou os itens com a **mesma chave**.
  - Nome e idade
  - No baralho, a ordenação por valor não alteraria a ordem dos naipes, por exemplo;

2	**5**	**3**	**4**	**4**	**7**	**8**

Shotgun Sort



Shotgun Sort

- Pior algoritmo para ordenação;
- Por que?
  - Ele embaralha a lista até que ela esteja ordenada.
- Também conhecido como:
  - *Bogosort, permutation sort, stupid sort, slow sort, monkey sort*

Algoritmo – *Shotgun Sort*

```
Algoritmo boolean isOrdenado (int vetor[])
Para i ← 1 até n
    Se vetor[i] > vetor[i+1] então
        retorna falso
    Fim-Se
Fim-Para
fim
```

```
Algoritmo shotgunSort (int vetor[])
  enquanto isOrdenado(vetor) == false faça
      Embaralha random;
  Fim-enquanto
fim
```

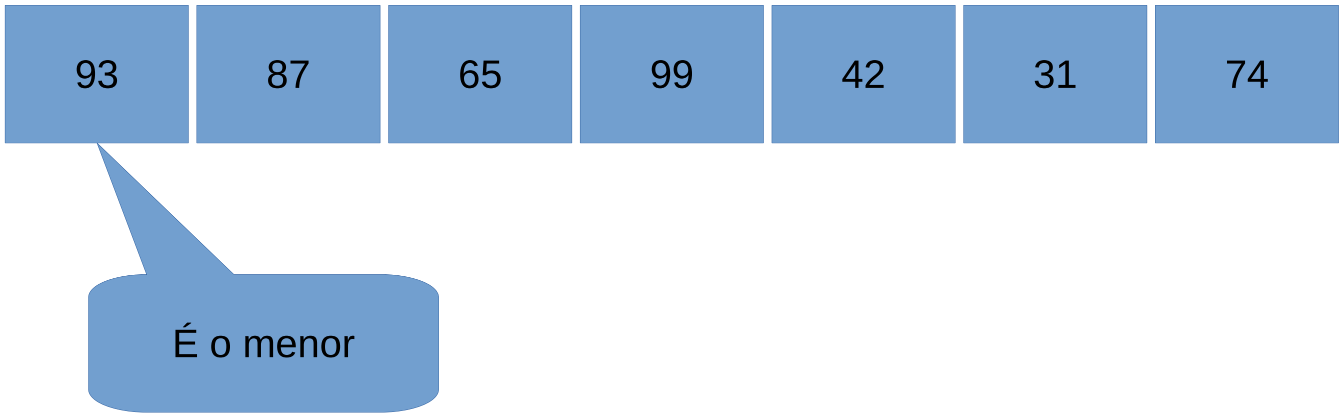
Selection Sort

Selection Sort

- Algoritmo bastante simples, por seleção, que é recomendado para conjuntos pequenos de dados.
- O algoritmo do Selection Sort consiste em três passos:
  - Navegue pelo vetor até encontrar o menor valor desse vetor;
  - Remova esse valor do vetor e insira na primeira posição do vetor de resposta;
  - Repita esse passo para cada item presente no vetor.

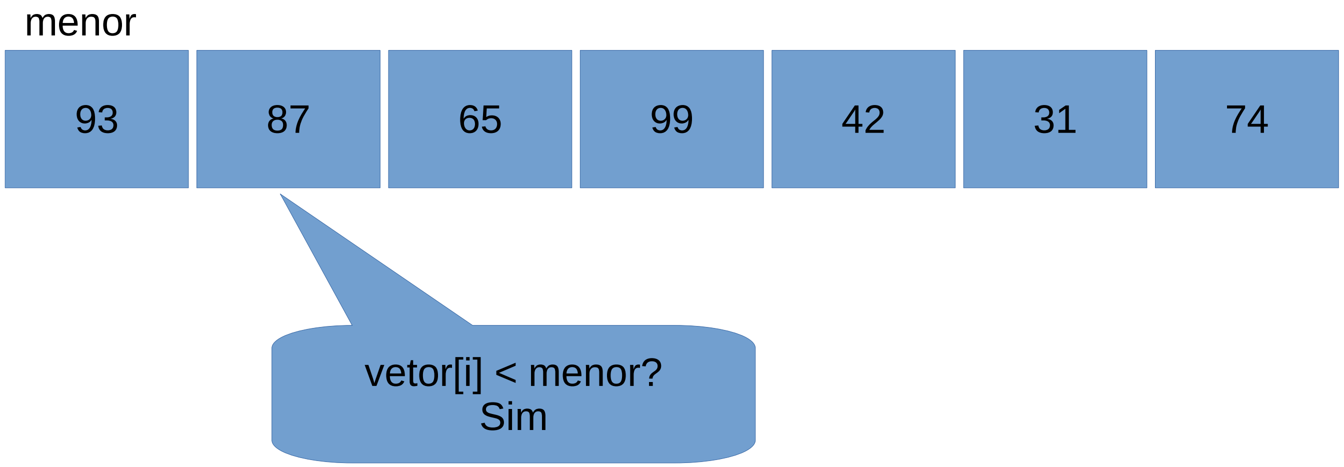
Selection Sort

- Busca o menor
  - Varredura 01



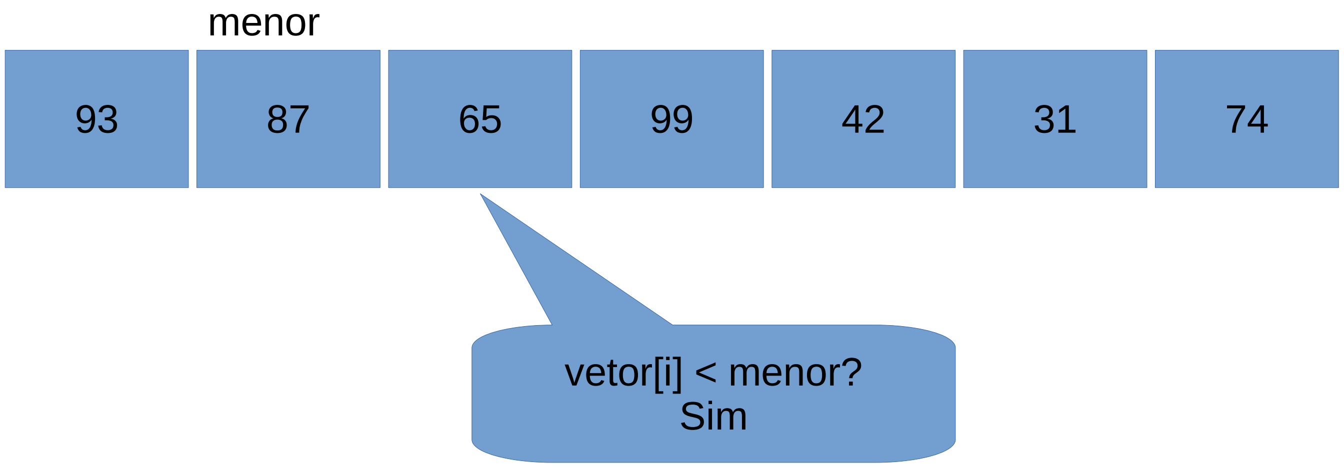
Selection Sort

- Busca o menor
  - Varredura 01



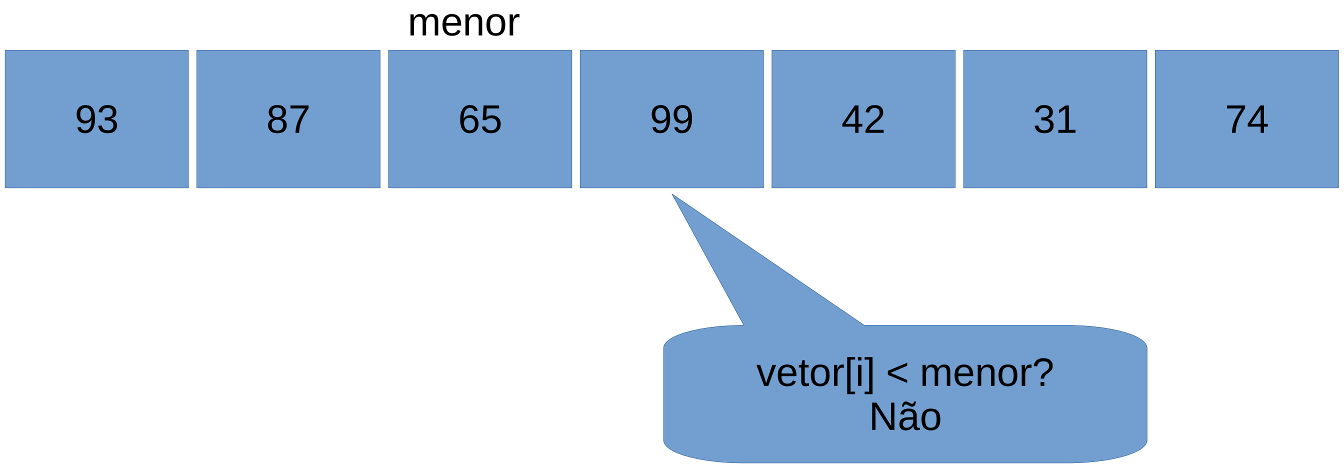
Selection Sort

- Busca o menor
  - Varredura 01



Selection Sort

- Busca o menor
  - Varredura 01



Selection Sort

- Busca o menor
  - Varredura 01

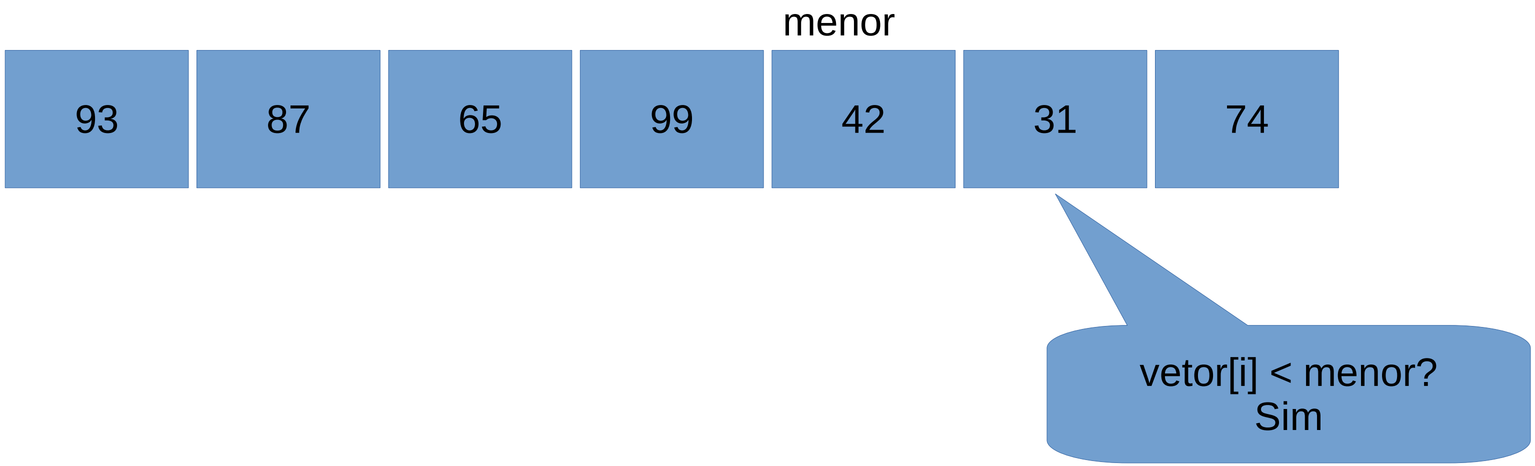
menor

93	87	65	99	42	31	74

vetor[i] < menor?
Sim

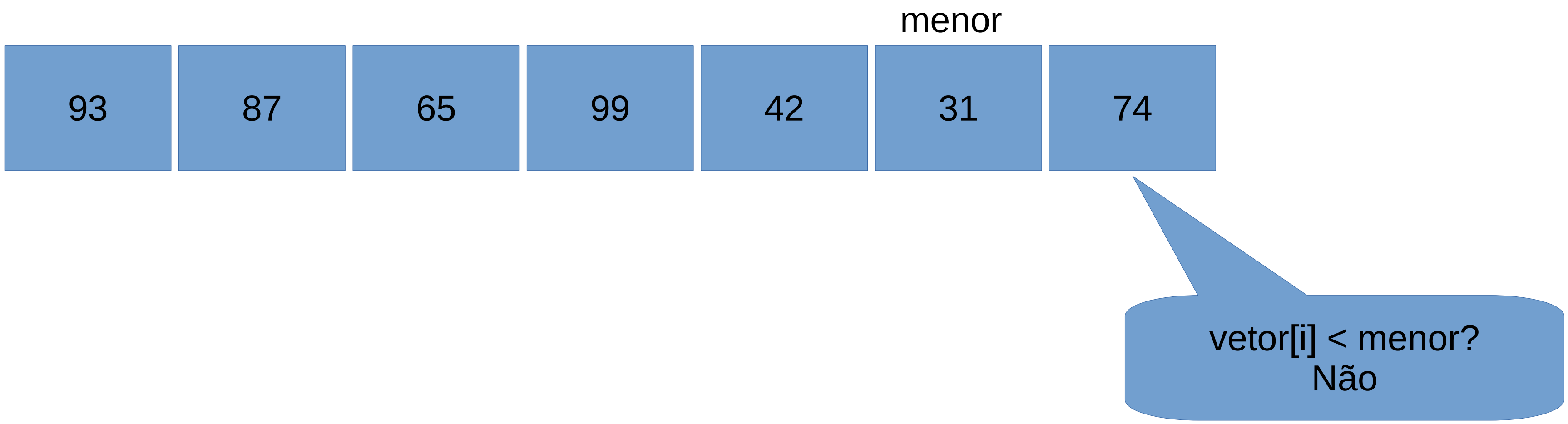
Selection Sort

- Busca o menor
  - Varredura 01



Selection Sort

- Busca o menor
  - Varredura 01



Selection Sort

- Troca o menor com a primeira posição

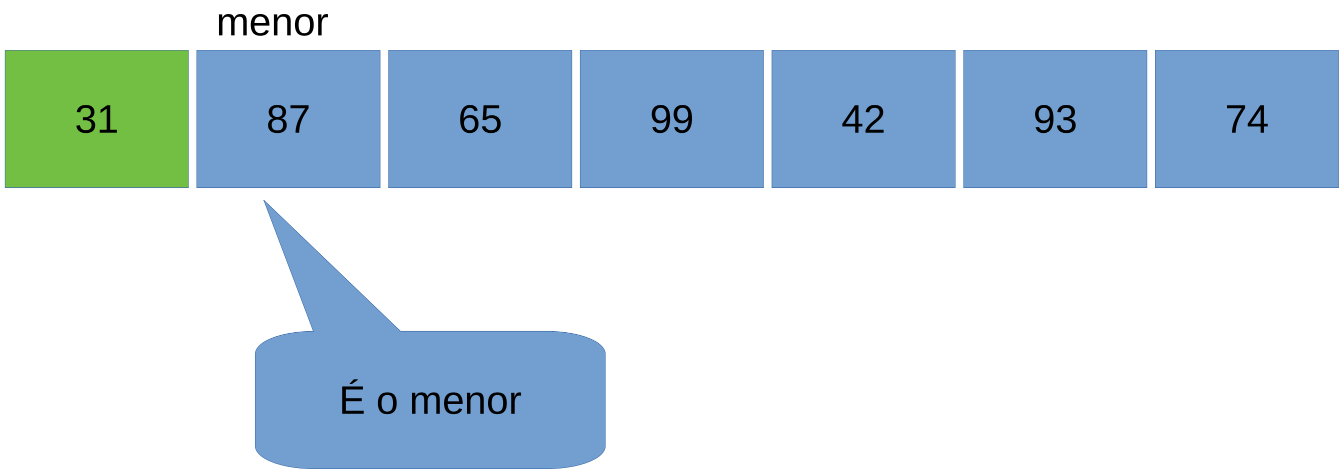
					menor	
93	87	65	99	42	31	74

Selection Sort

31	87	65	99	42	93	74

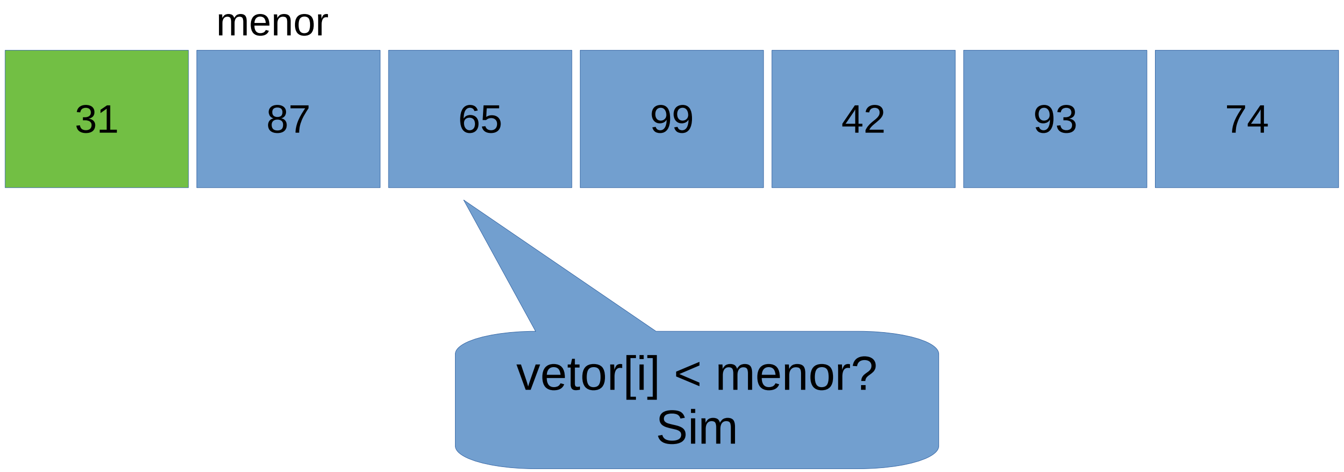
Selection Sort

- Busca o menor
  - Varredura 02



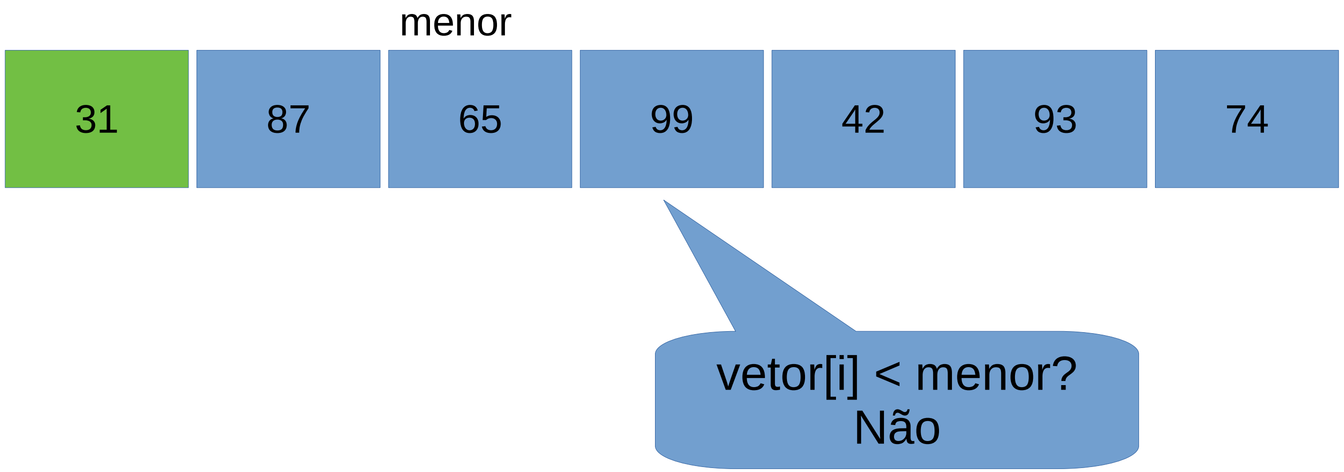
Selection Sort

- Busca o menor
  - Varredura 02



Selection Sort

- Busca o menor
  - Varredura 02



Selection Sort

- Busca o menor
  - Varredura 02

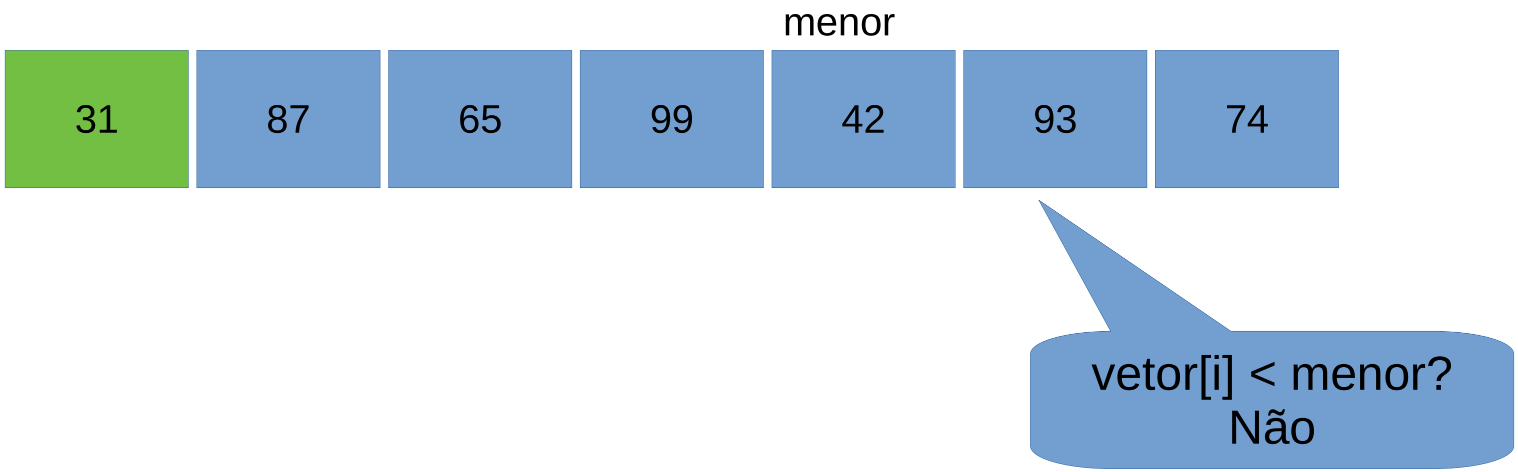
menor

31	87	65	99	42	93	74

vetor[i] < menor?
Sim

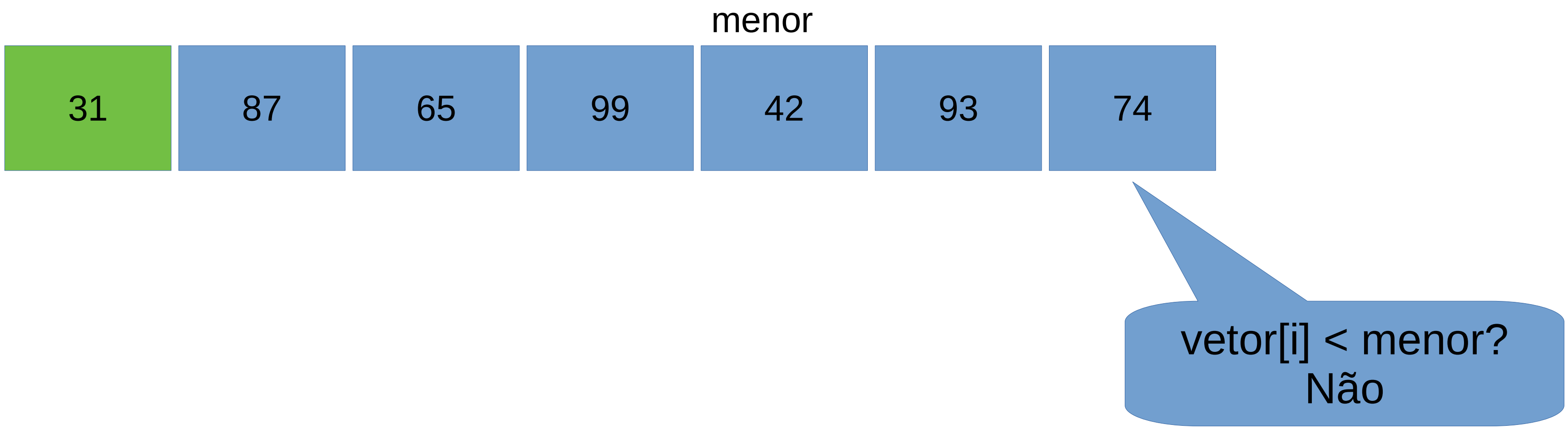
Selection Sort

- Busca o menor
  - Varredura 02



Selection Sort

- Busca o menor
  - Varredura 02



Selection Sort

- Troca o menor com a primeira posição

				menor		
31	87	65	99	42	93	74

Selection Sort

31	42	65	99	87	93	74

Selection Sort

- Busca o menor
  - Varredura 03

menor

31	42	65	99	87	93	74

Selection Sort

- Troca o menor com a primeira posição

		menor				
31	42	65	99	87	93	74

Selection Sort

31	42	65	99	87	93	74

Selection Sort

- Busca o menor
  - Varredura 04

menor

31	42	65	99	87	93	74

Selection Sort

- Troca o menor com a primeira posição

						menor
31	42	65	99	87	93	74

Selection Sort

				menor		
31	42	65	74	87	93	99

Selection Sort

31	42	65	74	87	93	99

Selection Sort

					menor	
31	42	65	74	87	93	99

Selection Sort

31	42	65	74	87	93	99

Selection Sort

						menor
31	42	65	74	87	93	99

Selection Sort

						menor
31	42	65	74	87	93	99

Selection Sort

31	42	65	74	87	93	99

Algoritmo – Selection Sort

```
para i ← 0 até tamanho - 1, faça
    Minimo ← i
    para j ← i+1 até tamanho, faça
        se vetor[j] < vetor[minimo], então
            minimo ← j
        fim-se
    fim-para
    temp ← vetor[i]
    vetor[i] ← vetor[minimo]
    vetor[minimo] ← temp
fim-para
```

Selection Sort

- O algoritmo do Selection Sort consiste em três passos:
  - Navegue pelo vetor até encontrar o menor valor desse vetor;
    - Depende do tamanho do vetor – O(n)
  - Remova esse valor do vetor e insira na primeira posição do vetor de resposta;
    - Operação de troca de posições, independe do tamanho do vetor – O(1)
  - Repita esses passos para cada item existente no vetor.
    - Depende do tamanho do vetor – O(n)

Selection Sort

- Como os dois passos que dependem do tamanho do vetor, estão aninhados a complexidade deles acaba se multiplicando.

$$O\ (n) = n^2$$

Selection Sort

- O algoritmo do Selection Sort é usado muitas vezes em sistemas de tempo real, porque ele tem o mesmo desempenho não importa a ordenação prévia do vetor.
- O Selection Sort não é um algoritmo estável, ou seja, os elementos que possuem o mesmo valor nem sempre irá manter a posição relativa de antes do início da ordenação.

Insertion Sort

Insertion Sort

- Algoritmo simples, por inserção, que percorre um vetor de elementos da esquerda para a direita.
- À medida que avança, vai deixando os elementos mais à esquerda ordenados, comparando o elemento com os anteriores.

Insertion Sort

- Varredura 01

9	8	6	12	4	3

Comparação → (vetor [0] > vetor [1]) ? → SIM → Troca

8	9	6	12	4	3

Fim Varredura 01

Insertion Sort

- Varredura 02

8	9	6	12	4	3

Comparação → (vetor [1] > vetor [2]) ? → SIM → Troca

8	6	9	12	4	3

Insertion Sort

- Varredura 02

8	6	9	12	4	3

Comparação → (vetor [0] > vetor [1]) ? → SIM → Troca

6	8	9	12	4	3

Fim Varredura 02

Insertion Sort

- Varredura 03

6	8	9	12	4	3

Comparação → (vetor [2] > vetor [3]) ? → Não

6	8	9	12	4	3

Insertion Sort

- Varredura 03

6	8	9	12	4	3

Comparação → (vetor [1] > vetor [2]) ? → Não

6	8	9	12	4	3

Insertion Sort

- Varredura 03

6	8	9	12	4	3

Comparação → (vetor [0] > vetor [1]) ? → Não

6	8	9	12	4	3

Fim Varredura 03

Insertion Sort

- Varredura 04

6	8	9	12	4	3

Comparação → (vetor [3] > vetor [4]) ? → SIM → Troca

6	8	9	4	12	3

Insertion Sort

- Varredura 04

6	8	9	4	12	3

Comparação → (vetor [2] > vetor [3]) ? → SIM → Troca

6	8	4	9	12	3

Insertion Sort

- Varredura 04

6	8	4	9	12	3

Comparação → (vetor [1] > vetor [2]) ? → SIM → Troca

6	4	8	9	12	3

Insertion Sort

- Varredura 04

6	4	8	9	12	3

Comparação → (vetor [0] > vetor [1]) ? → SIM → Troca

4	6	8	9	12	3

Fim da Varredura 04

Insertion Sort

- Varredura 05

4	6	8	9	12	3

Comparação → (vetor [4] > vetor [5]) ? → SIM → Troca

4	6	8	9	3	12

Insertion Sort

- Varredura 05

4	6	8	9	3	12

Comparação → (vetor [3] > vetor [4]) ? → SIM → Troca

4	6	8	3	9	12

Insertion Sort

- Varredura 05

4	6	8	3	9	12

Comparação → (vetor [2] > vetor [3]) ? → SIM → Troca

4	6	3	8	9	12

Insertion Sort

- Varredura 05

4	6	3	8	9	12

Comparação → (vetor [1] > vetor [2]) ? → SIM → Troca

4	3	6	8	9	12

Insertion Sort

- Varredura 05

4	3	6	8	9	12

Comparação (vetor [0] > vetor [1]) ? SIM Troca

3	4	6	8	9	12

Fim da Varredura 05

Algoritmo – *Insertion Sort*

```
insertionSort (A : lista de itens)
n = tamanho(A)
Para i = 1 até n-1
    j=i
    Enquanto j > 0 and A[j-1] > A[j]
        inverter(A[j], A[j-1])
        j=j-1
    fim-enquanto
fim-para
fim
```

Análise de Complexidade *Insertion Sort*

- Se o vetor a ordenar possui n elementos, o algoritmo irá realizar n - 1 etapas.
- No melhor caso
  - Vetor ordenado
  - Complexidade: O(n) = n
- No caso médio:
  - Complexidade: O(n) = n^2

Análise de Complexidade *Insertion Sort*

- No pior caso (vetor invertido):
  - Como para um vetor de n elementos, n – 1 varreduras são feitas para acertar todos os elementos, o número de comparações é:

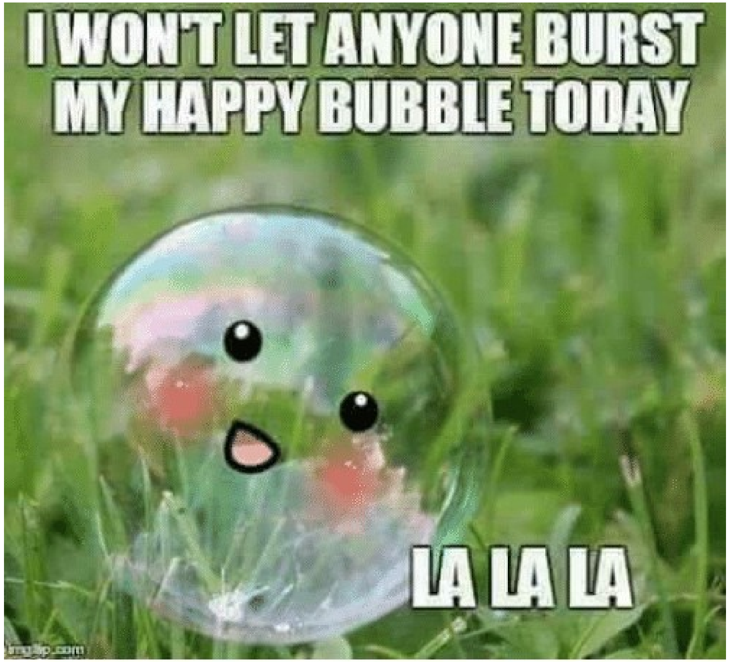
$$(n - 1) + (n - 2) + \dots + 2 + 1$$

  - Assim, serão feitas $(n^2 - n) / 2$ trocas.
  - Complexidade: $O(n) = n^2$

Insertion Sort

- Eficiente para ordenar uma pequena quantidade de elementos.
- Para um vetor que está quase ordenado, esse algoritmo também é a melhor escolha (entre os apresentados nesses slides).
  - Economia de comparações quando uma troca não é feita;
- É um algoritmo estável.

Bubble Sort


I WON'T LET ANYONE BURST
MY HAPPY BUBBLE TODAY
LA LA LA
imgflip.com

Bubble Sort

- É um método de ordenação por troca que atua comparando sucessivamente pares de elementos e mudando-os de posição quando se apresentam fora da ordem desejada.
- Nessa estratégia, o maior elemento é “empurrado” para o final da estrutura que está sendo ordenada.

Bubble Sort

- Vamos considerar o seguinte arranjo:

9	8	6	12	4	3

- Consiste em efetuar uma varredura no arranjo, da esquerda para a direita, comparando pares de elementos consecutivos e trocando-os de lugar os que estão fora de ordem.

Bubble Sort

- Varredura 01:

9	8	6	12	4	3

Comparação → (vetor [0] > vetor [1]) ? → SIM → Troca

8	9	6	12	4	3

Bubble Sort

- Varredura 01:

8	9	6	12	4	3

Comparação → (vetor [1] > vetor [2]) ? → SIM → Troca

8	6	9	12	4	3

Bubble Sort

- Varredura 01:

8	6	9	12	4	3

Comparação → (vetor [2] > vetor [3]) ? → Não → Não Troca

8	6	9	12	4	3

Bubble Sort

- Varredura 01:

8	6	9	12	4	3

Comparação ➡ (vetor [3] > vetor [4]) ? ➡ SIM ➡ Troca

8	6	9	4	12	3

Bubble Sort

- Varredura 01:

8	6	9	4	12	3

Comparação ➡ (vetor [4] > vetor [5]) ? ➡ SIM ➡ Troca

8	6	9	4	3	12

Fim da Varredura 01

Bubble Sort

- Varredura 02:

8	6	9	4	3	12

Comparação → (vetor [0] > vetor [1]) ? → SIM → Troca

6	8	9	4	3	12

Bubble Sort

- Varredura 02:

6	8	9	4	3	12

Comparação → (vetor [1] > vetor [2]) ? → NÃO

6	8	9	4	3	12

Bubble Sort

- Varredura 02:

6	8	9	4	3	12

Comparação → (vetor [2] > vetor [3]) ? → SIM → Troca

6	8	4	9	3	12

Bubble Sort

- Varredura 02:

6	8	4	9	3	12

Comparação ➡ (vetor [3] > vetor [4]) ? ➡ SIM ➡ Troca

6	8	4	3	9	12

Fim da Varredura 02

Bubble Sort

- Varredura 03:

6	8	4	3	9	12

Comparação → (vetor [0] > vetor [1]) ? → NÃO

6	8	4	3	9	12

Bubble Sort

- Varredura 03:

6	8	4	3	9	12

Comparação → (vetor [1] > vetor [2]) ? → SIM → Troca

6	4	8	3	9	12

Bubble Sort

- Varredura 03:

6	4	8	3	9	12

Comparação → (vetor [2] > vetor [3]) ? → SIM → Troca

6	4	3	8	9	12

Fim da Varredura 03

Bubble Sort

- Varredura 04:

6	4	3	8	9	12

Comparação → (vetor [0] > vetor [1]) ? → SIM → Troca

4	6	3	8	9	12

Bubble Sort

- Varredura 04:

4	6	3	8	9	12

Comparação → (vetor [1] > vetor [2]) ? → SIM → Troca

4	3	6	8	9	12

Fim da Varredura 04

Bubble Sort

- Varredura 05:

4	3	6	8	9	12

Comparação (vetor [0] > vetor [1]) ? SIM Troca

3	4	6	8	9	12

Fim da Varredura 05

Bubble Sort

- Varredura 05:

4	3	6	8	9	12

3	4	6	8	9	12

Fim da Varredura 05

- Para um vetor de ***n*** elementos, ***n – 1*** varreduras são feitas para acertar todos os elementos.
- Neste caso:
  - n = 6
  - varreduras = 5

Coelhos e Tartarugas - *Bubble Sort*

- Coelhos
  - Valores altos são movidos para o fim da lista rapidamente;

- Tartarugas
  - Valores baixos são movidos para o início da lista vagarosamente;

Algoritmo - *Bubble Sort*

```
Algoritmo bubbleSort( A : lista de itens)
n = tamanho(A)
Para i ←  0 até n
    Para j ←  1 até n
        Se A[j - 1] > A[j] então
            inverter( A[i-1], A[i] )
        Fim-se
    Fim-para
Fim-para
fim procedure
```

Análise de Complexidade de Tempo - *Bubble Sort*

- Pior Caso?
  - Vetor ordenado ao contrário (Invertido)
- Como para um vetor de *n* elementos, *n* varreduras são feitas para acertar todos os elementos, o número de comparações é:

$$n*(n – 1) = n^2 – n$$

  - O número de trocas, por outro lado, é igual a:

$$(n\text{-}1) + (n\text{-}2) + ... + 1 = n^2 - n$$

  - Complexidade: $O(n) = n^2$

Análise de Complexidade de Tempo - *Bubble Sort*

- Melhor Caso?
  - Vetor Ordenado
- Como para um vetor de n elementos, n varreduras são feitas para acertar todos os elementos, o número de comparações é:

$$n$$

- Não irá ocorrer trocas.
- Complexidade: $O(n) = n$

Análise de Complexidade de Tempo - *Bubble Sort*

- Caso Médio?
  - Vetor não ordenado
- Como para um vetor de *n* elementos, *n* varreduras são feitas para acertar todos os elementos, o número de comparações é:
  - $n*(n - 1) = n^2 - n$
- O número de trocas, por outro lado, é igual a:
  - $(n-1) + (n-2) + ... + 1 = n^2 - n$
- Complexidade: O(*n*) = n^2, pois não sabemos ao certo quantas comparações serão necessárias.

Análise de Complexidade Espacial - *Bubble Sort*

```
Algoritmo bubbleSort( A : lista de itens)
n = tamanho(A)
Para i ←  0 até n
      Para j ←  1 até n
         Se A[j - 1] > A[j] então
            inverter( A[i-1], A[i] )
         Fim-se
      Fim-para
Fim-para
fim procedure
```

$S(n) = 1$

$S(n) = 1$

$S(n) = 1$

$S(n) = 1$

$O(n) = 1$

Otimização - *Bubble Sort*

- Não existe a necessidade de percorrermos o vetor completo em todas as iterações.

- O valor máximo do vetor é carregado até a última posição.

Otimização - *Bubble Sort*

```
Algoritmo bubbleSort_Ot1( A : lista de itens)
n = tamanho(A)
Para i ←  0 até n
     Para j ←  1 até n - i
        Se A[j - 1] > A[j] então
           inverter( A[i-1], A[i] )
        Fim-se
     Fim-para
Fim-para
fim procedure
```

Otimização - *Bubble Sort*

- Análise de Complexidade
  - No pior caso (vetor invertido):
    - Na i-ésima repetição, n – 1 – i comparações são feitas para acertar todos os elementos. Assim, o número de comparações é:

  $(n - 1) + (n - 2) + ... + 2 + 1 = (n^2 - n)/2$
  - Número de trocas se mantém em $(n^2 - n)/2$;
  - Complexidade: $O(n) = n^2$

Bubble Sort

- Entre os algoritmos quadráticos, é o mais lento.
- Não é muito prático quando temos uma grande quantidade de elementos a serem ordenados.
- É estável, ou seja, os registros com chaves iguais sempre irão manter a mesma posição relativa de antes do início da ordenação.

Odd-Even Sort

- Método criado para permitir que o *Bubble Sort* fosse utilizado em processadores paralelos.
- Separa por índices pares e impares.

Odd-Even Sort

Não Ordenado

3 2 3 8 5 6 4 1 Fase 1 (Ímpar)

Complexidade - *Odd-Even Sort*

- Complexidade de tempo:
  - $O(n) = n^2$

- Complexidade de espaço:
  - $O(n) = 1$

Odd-Even Sort

```
oddEvenSort(A : lista de itens)
    ordenado = false;
    enquanto ordenado == false
        ordenado = true
        Para i = 1 até n-1, i = i + 2
            Se A[i-1] > A[i] então
                inverter(A[i-1], A[i]);
                ordenado = false;
            Fim-se
        Fim-para
        Para i = 2 até n-1, i = i + 2
            Se A[i-1] > A[i] então
                inverter(A[i-1], A[i]);
                ordenado = false;
            Fim-se
        Fim-para
    Fim-enquanto
Fim
```

Cocktail Sort

- Método que resolve a questão dos coelhos e tartarugas.

- Realiza uma ordenação do menor para o maior seguida de uma outra do maior para o menor.

Cocktail Sort

- Exemplo :
  - {5 1 4 2 8 0 2}
- Primeiro Passo Para Frente:
  - (**5 1** 4 2 8 0 2)
    - 5 > 1
    - (**1 5** 4 2 8 0 2)
  - (1 **5 4** 2 8 0 2)
    - 5 > 4
    - (1 **4 5** 2 8 0 2)
  - (1 4 **5 2** 8 0 2)
    - 5 > 2
    - (1 4 **2 5** 8 0 2)
  - (1 4 2 **5 8** 0 2)
    - (1 4 2 **5 8** 0 2)
  - (1 4 2 5 **8 0** 2)
    - 8 > 0
    - (1 4 2 5 **0 8** 2)
  - (1 4 2 5 0 **8 2**)
    - 8 > 2
    - (1 4 2 5 0 **2 8**)

Cocktail Sort

- Primeiro Passo Para Trás:
- (1 4 2 5 **0 2** 8)
  - (1 4 2 5 0 2 8)
- (1 4 2 **5 0** 2 8)
  - 5 > 0
  - (1 4 2 **0 5** 2 8)
- (1 4 **2 0** 5 2 8)
  - 2 > 0
  - (1 4 **0 2** 5 2 8)
- (1 **4 0** 2 5 2 8)
  - 4 > 0
  - (1 **0 4** 2 5 2 8)
- (**1 0** 4 2 5 2 8)
  - 1 > 0
  - (**0 1** 4 2 5 2 8)

Cocktail Sort

- Segundo Passo Para Frente:
- (**0 1** 4 2 5 2 8)
  - (**0 1** 4 2 5 2 8)
- (0 **1 4** 2 5 2 8)
  - (0 **1 4** 2 5 2 8)
- (0 1 **4 2** 5 2 8)
  - 4 > 2
  - (0 1 **2 4** 5 2 8)
- (0 1 2 **4 5** 2 8)
  - (0 1 2 **4 5** 2 8)
- (0 1 2 4 **5 2** 8)
  - 5 > 2
  - (0 1 2 4 **2 5** 8)

Cocktail Sort

- Segundo Passo Para Trás:
- (0 1 2 **4 2** 5 8)
  - 4 > 2
  - (0 1 2 2 4 5 8)

Cocktail Sort

```
cocktailSort( A : lista de itens ):
        invertido = true;
        start = 0;
        end = a.length;
        enquanto (invertido == true)
             invertido = false;
             Para i = start até end
                 Se (a[i] > a[i + 1]) então
                       Inverte (a[i], a[i+1])
                       invertido = true;
                  Fim-se
             Fim-Para
             Se (invertido == false) então
                  break;
             invertido = false;
             end = end - 1;
             Para i = end até start
                  Se (a[i] > a[i + 1]) então
                       Inverte (a[i], a[i+1])
                       invertido = true;
                  Fim-se
             Fim-para
             start = start + 1;
        Fim-enquanto
Fim
```

Complexidade – *Cocktail Sort*

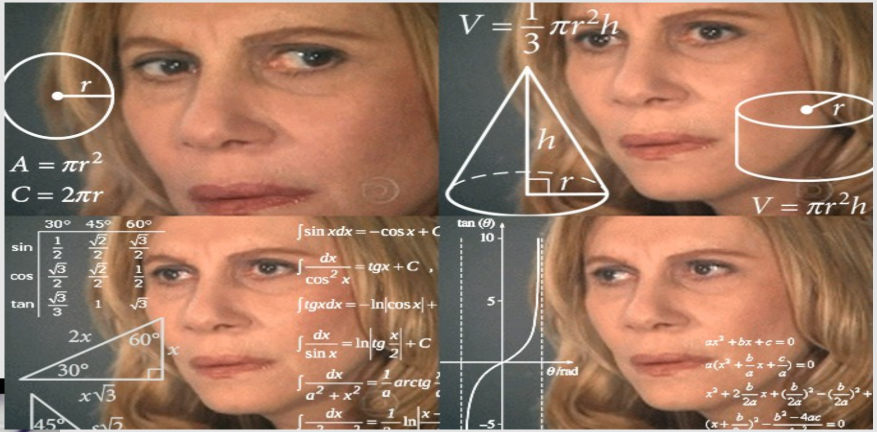
- Complexidade de tempo:
  - Pior caso
    - $O(n) = n^2$
  - Melhor caso
    - $O(n) = n$
- Complexidade de espaço:
  - $O(n) = 1$

Dúvidas

Referência

- DOBRUSHKIN, Vladimir A. Métodos para Análise de Algoritmos. LTC, 03/2012

Exercícios


$A = \pi r^2$
$C = 2\pi r$
$V = \frac{1}{3}\pi r^2 h$
$V = \pi r^2 h$


O MONSTRO
TÁ SAINDO DA JAULA

Exercícios

1) Implemente o *bubble sort* e teste com vetores de gerados randomicos.

Obrigado!
Boa Noite!

bruno.moritani@anhembi.br