

# Métodos de Ordenação: *Selection Sort e Insertion Sort*

Pesquisa, Ordenação e Técnicas de  
Armazenamento

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

# Agenda

- Revisão
  - Busca sequencial
  - Busca Binária
- *Selection Sort*
- *Insertion Sort*
  - *Shell Sort*

# Busca Sequencial

- Método mais simples de busca;
- É aplicável a vetores desordenados e ordenados;
- Percorre-se todos os registros até encontrar a chave da busca.

# Busca Sequencial

- A implementação do método sequencial consiste em:
  - Um loop que navega por todo o vetor;
  - Uma regra de comparação, que analisa se o valor do vetor naquela posição é igual ao valor procurado

Algoritmo *buscaSequencial* (int X)

```
1.      Para  $i \leftarrow 0$  até  $n$ 
2.          Se  $\text{vetor}[i] == X$  então
3.              retorne  $i$ 
4.          Fim-se
5.      Fim-Para
6.      retorne  $-1$ 
7.  Fim
```

# Complexidade - Busca Sequencial

- Melhor caso
  - O elemento procurado se encontra na primeira posição da lista
  - A complexidade portanto é:  $O(n) = 1$
- Pior caso
  - O elemento procurado se encontra na última posição da lista ou não se encontra na lista
  - A complexidade portanto é:  $O(n) = n$
- Caso médio
  - O elemento procurado se encontra em uma posição qualquer da lista
  - A complexidade portanto é:  $O(n) = \frac{n+1}{2}$

# Sentinela - Otimizações da Busca Sequencial

Algoritmo *buscaSequencial\_Sentinela* (int X)

```
1.   vetor[ultimaPosicao] ← X
2.   i ← 0
3.   Enquanto vetor[i] != X Faça
4.       i ← i + 1
5.   Fim-Enquanto
6.   Se ( i == n ) então
7.       retorne -1
8.   senão
9.       retorne i
10.  Fim-se
11.  Fim
```

# Lista Ordenada - Otimizações da Busca Sequencial

Algoritmo *buscaSequencial\_ListaOrdenada* (int X)

```
1.   Para  $i \leftarrow 0$  até  $n$ 
2.       Se  $\text{vetor}[i] == X$  então
3.           retorne  $i$ 
4.       senão
5.           Se  $\text{vetor}[i] > X$  então
6.               Retorne  $-1$ 
7.           Fim-se
8.       Fim-se
9.   Fim-Para
10.      retorne  $-1$ 
11.  Fim
```

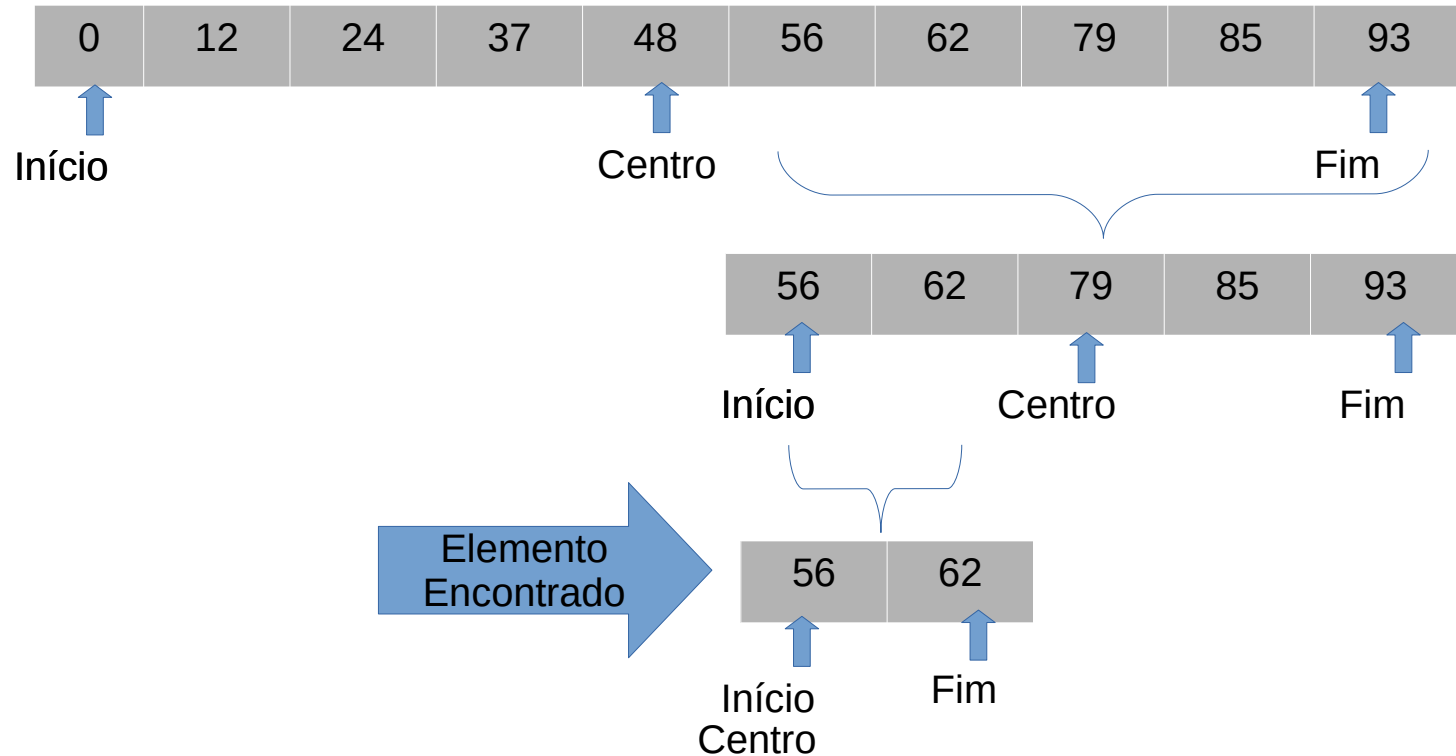
# Busca Binária

- A busca binária se aproveita do fato do vetor estar ordenado, para realizar a busca em menos iterações.
- Ela usa a estratégia **dividir para conquistar**, para dividir o problema da busca em problemas menores.
- Alto custo para manter a tabela ordenada.
- Não deve ser usada em aplicações muito dinâmicas.



# Busca Binária

- Buscando o elemento 56



# Busca Binária Iterativa

Algoritmo *buscaBinariaIterativa* (int X)

```
1.  Esquerda ← 0
2.  Direita ← n-1
3.  enquanto esquerda <= direita faça
4.    i ← (esq+dta)/2
5.    se x == v[i] então
6.      retorna i
7.    Fim-se
8.    se x < v[i] então
9.      Direita ← i-1
10.   Fim-se
11.   se x > v[i] então
12.     Esquerda ← i+1
13.   Fim-se
14. Fim-enquanto
15. Retorna -1
16. fim
```

# Complexidade – Busca Binária

- Melhor caso
  - O elemento procurado se encontra na primeira posição da lista
  - A complexidade portanto é  $O(n) = 1$
- Pior caso
  - O elemento procurado não se encontra na lista
  - A complexidade portanto é  $O(n) = \log(n)$
- Caso médio
  - O elemento procurado se encontra em uma posição qualquer da lista
  - A complexidade portanto é  $O(n) = \log(n)$

# Métodos de Ordenação

A decorative graphic on the left side of the slide. It features a large teal circle partially overlapping a smaller, lighter teal circle. Overlaid on these circles are several sets of parallel diagonal lines in shades of green and yellow, creating a textured, layered effect. The background of the slide is white.

# 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)

```
1.  inicio ← 0
2.  fim ← n-1
3.  enquanto inicio <= fim faça
4.      centro ← (inicio+fim)/2
5.      se x == v[centro] então
6.          retorna centro
7.      Fim-se
8.      se x < v[centro] então
9.          fim ← centro-1
10.     Fim-se
11.     se x > v[centro] então
12.         inicio ← centro+1
13.     Fim-se
14. Fim-enquanto
15. Retorna -1
16. fim
```

$S(n) = 1$

$S(n) = 1$

$S(n) = 1$

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)

```
1.  i ← 0
2.  int vetor[n]
3.  enquanto i <= n faça
4.      aux ← i
5.      Aux2 ← aux + 1
6.      i ← i + 1
7.      Imprima aux2
8.  Fim-enquanto
9.  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.

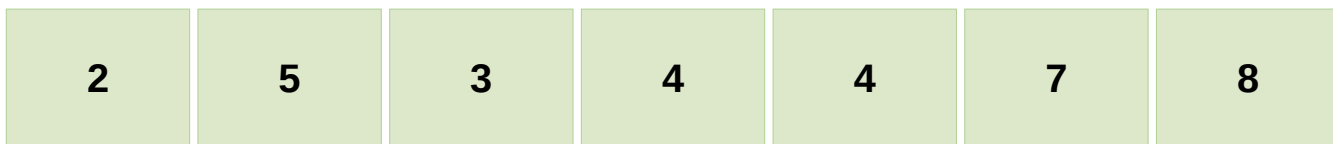
32	45	78	99	42	31	74
----	----	----	----	----	----	----

$\neq$

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;





# *Shotgun Sort*

when you call shotgun but end up  
in the back



# 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[])

```
1. Para i ← 1 até n
2.     Se vetor[i] > vetor[i+1] então
3.         retorna falso
4.     Fim-Se
5. Fim-Para
6. fim
```

Algoritmo *shotgunSort* (int vetor[])

```
1.     enquanto isOrdenado(vetor) == false faça
2.         Embaralha random;
3.     Fim-enquanto
4. fim
```

# *Selection Sort*

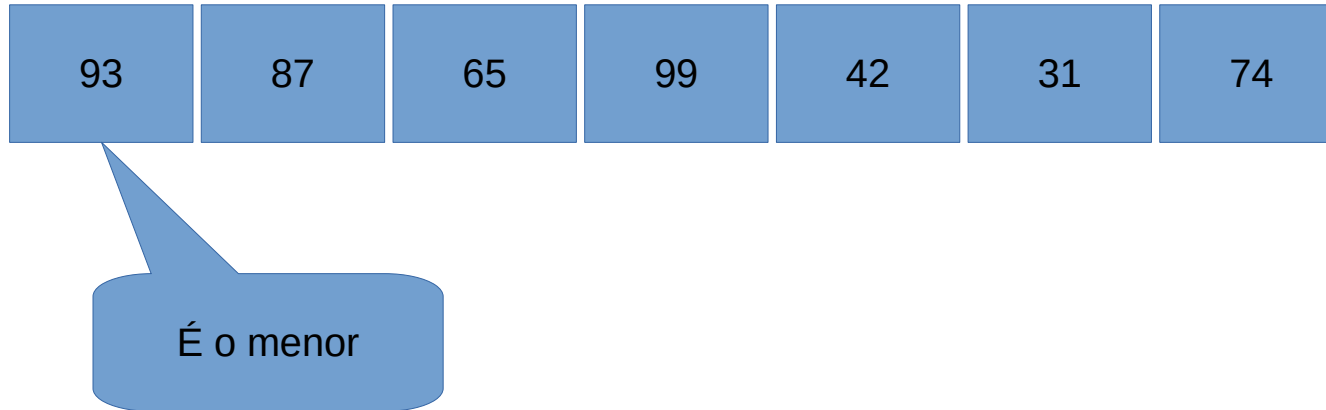


# Selection Sort

- 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.

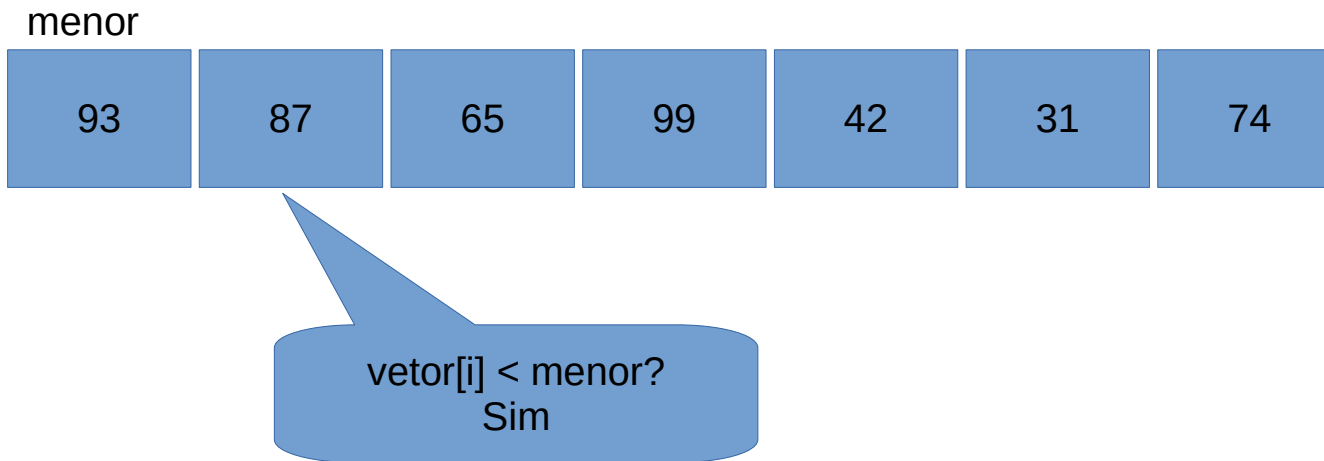
# Selection Sort

- Busca o menor
  - Varredura 01



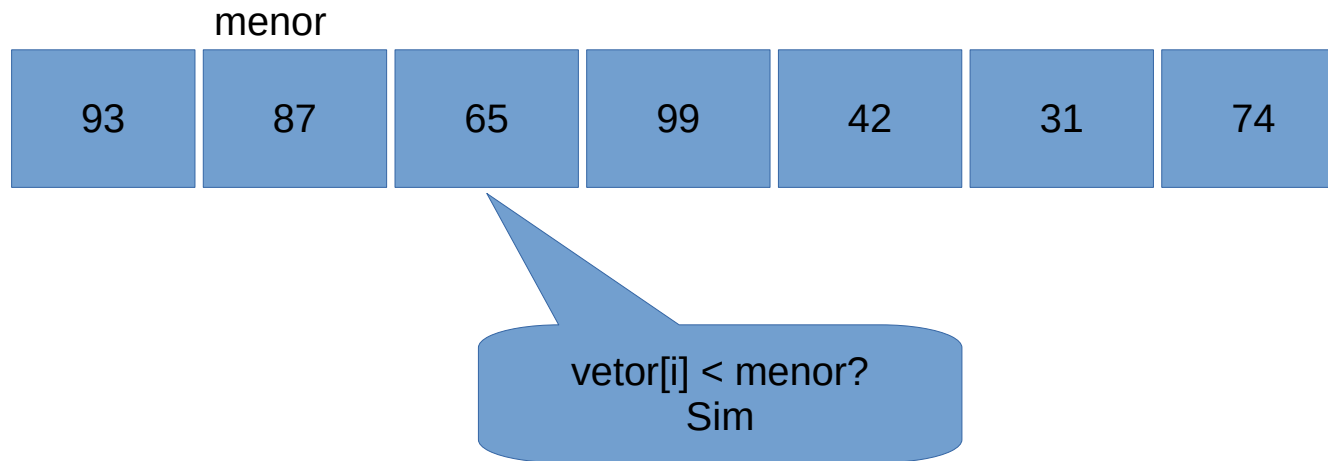
# Selection Sort

- Busca o menor
  - Varredura 01



# Selection Sort

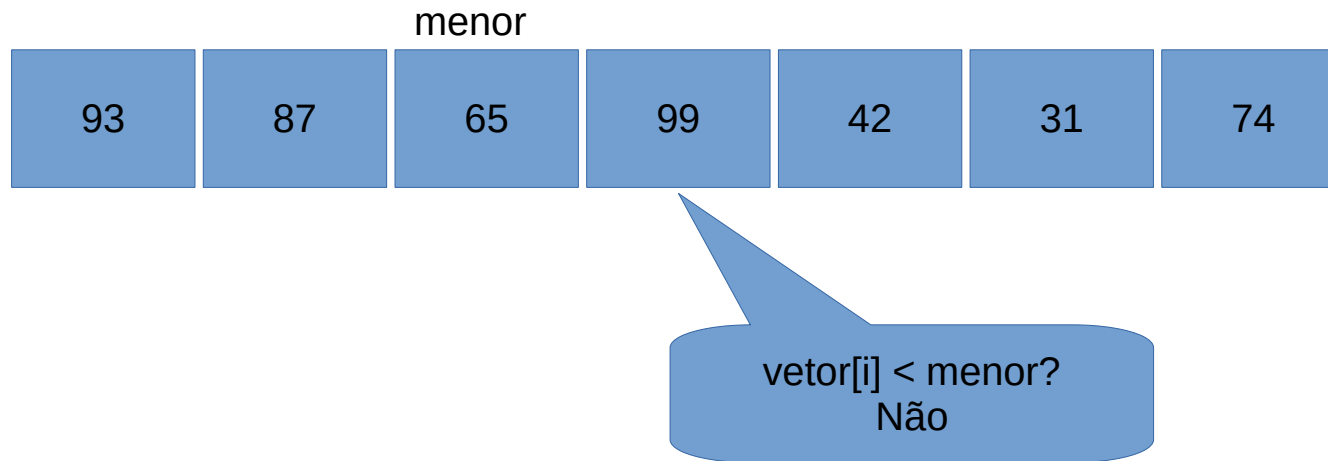
- Busca o menor
  - Varredura 01





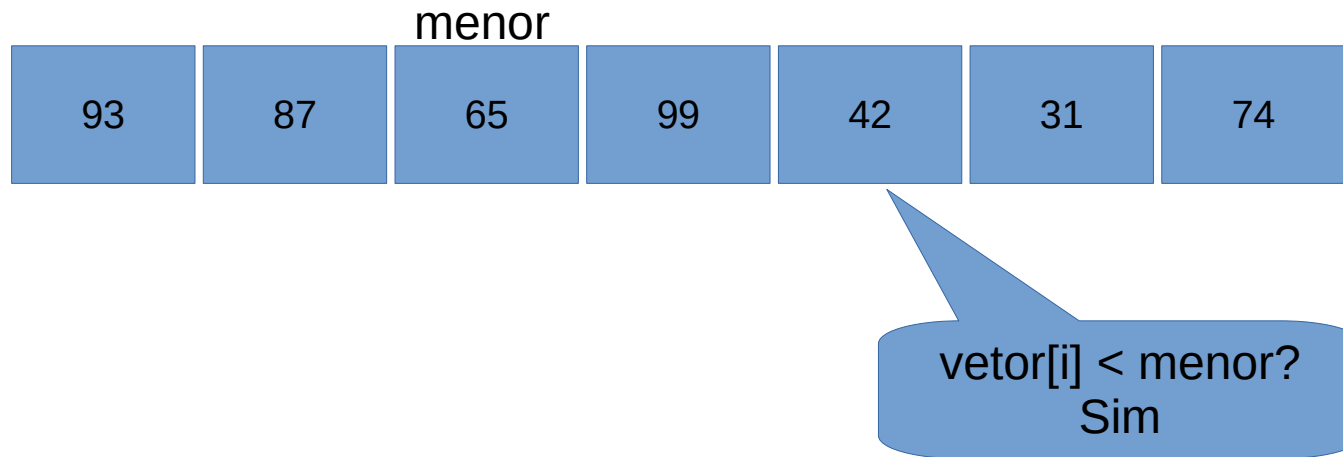
# Selection Sort

- Busca o menor
  - Varredura 01



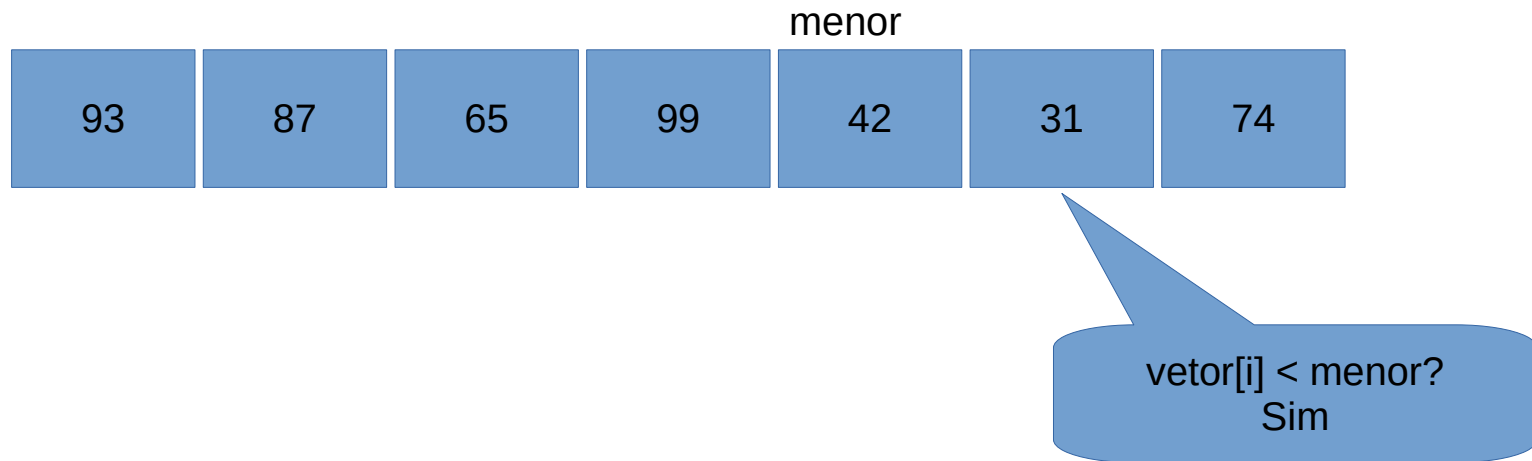
# Selection Sort

- Busca o menor
  - Varredura 01



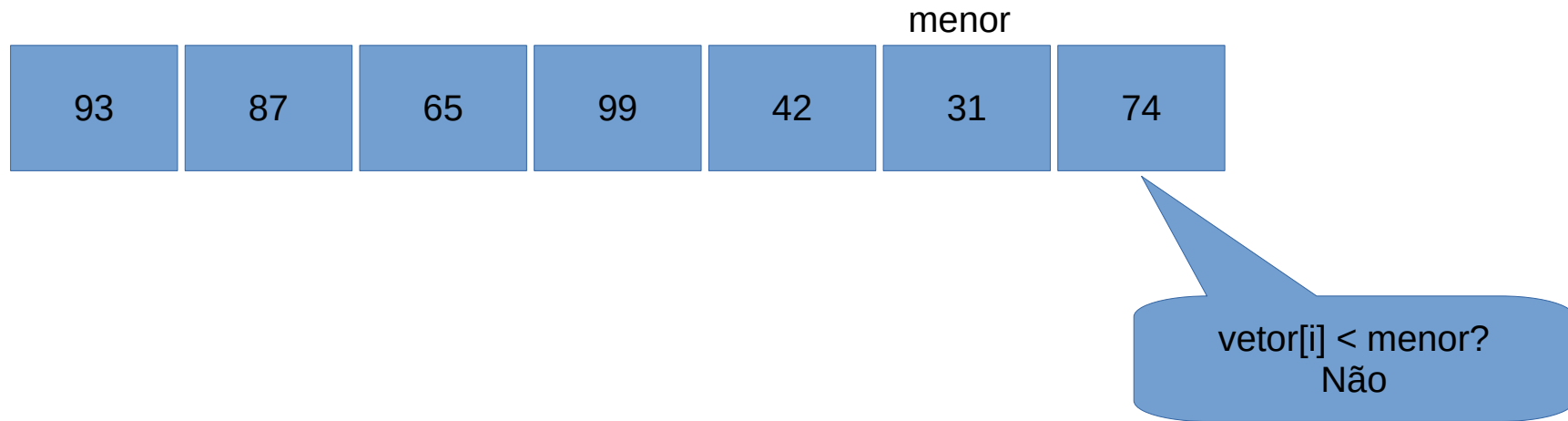
# Selection Sort

- Busca o menor
  - Varredura 01



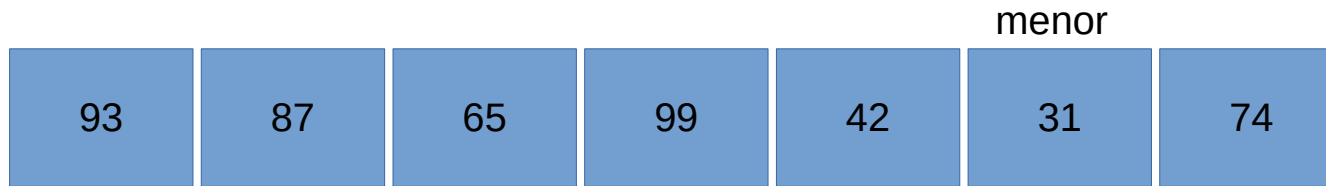
# Selection Sort

- Busca o menor
  - Varredura 01

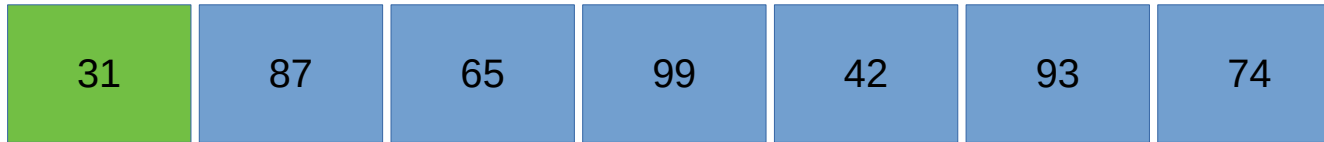


# Selection Sort

- Troca o menor com a primeira posição

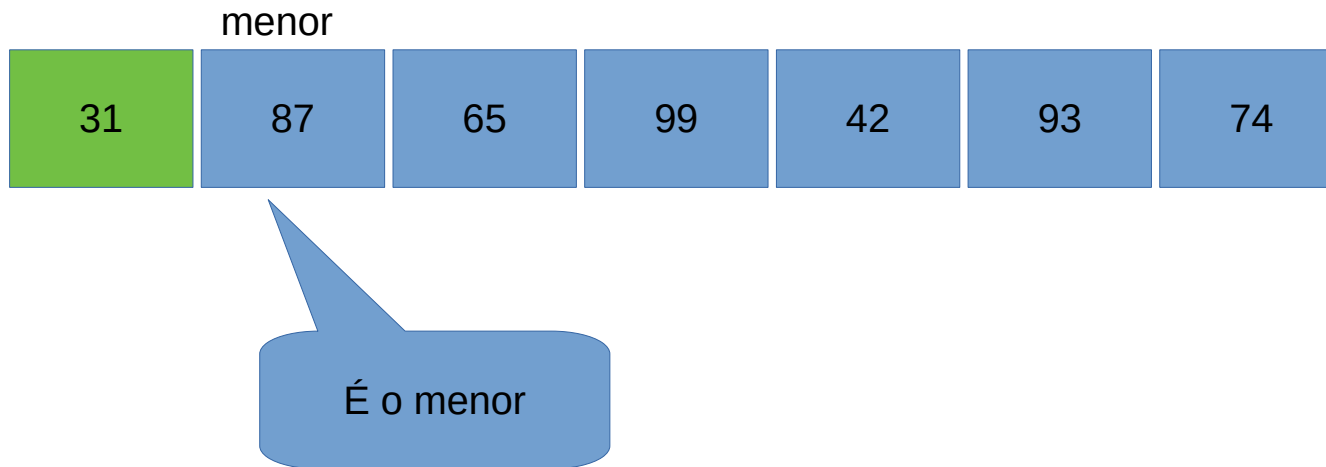


# Selection Sort



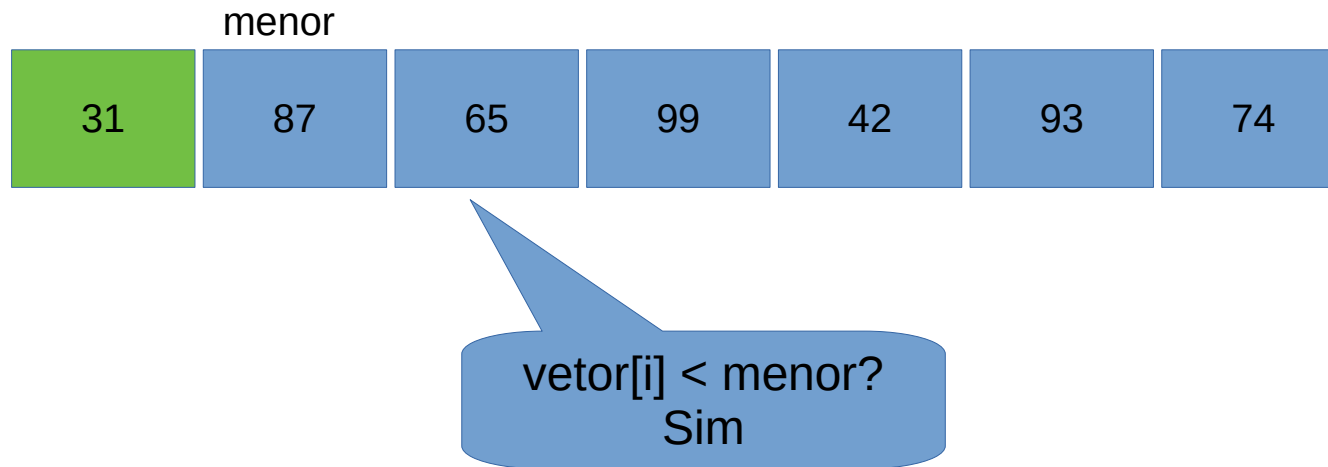
# Selection Sort

- Busca o menor
  - Varredura 02



# Selection Sort

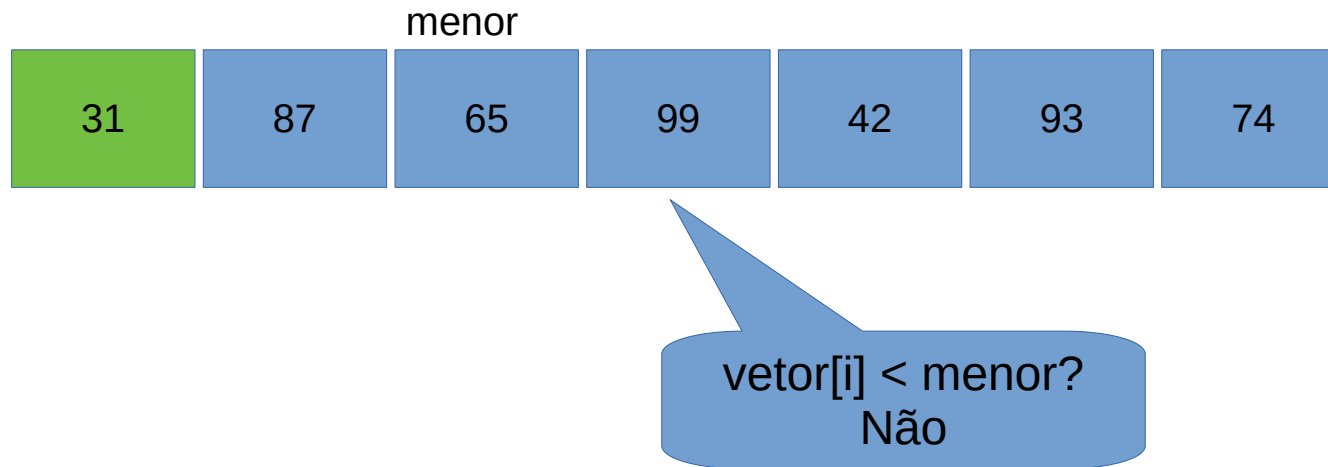
- Busca o menor
  - Varredura 02





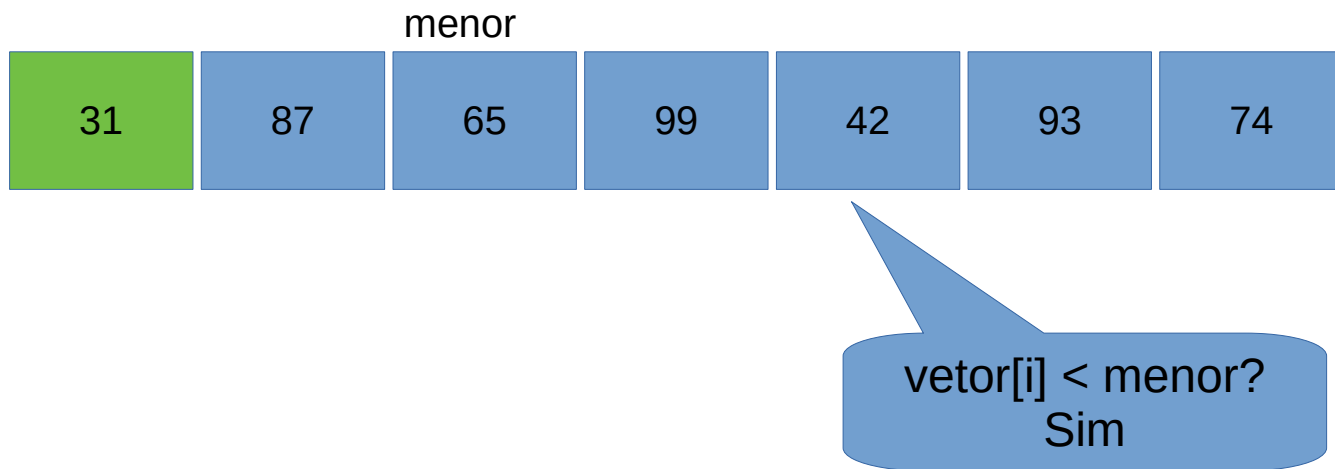
# Selection Sort

- Busca o menor
  - Varredura 02



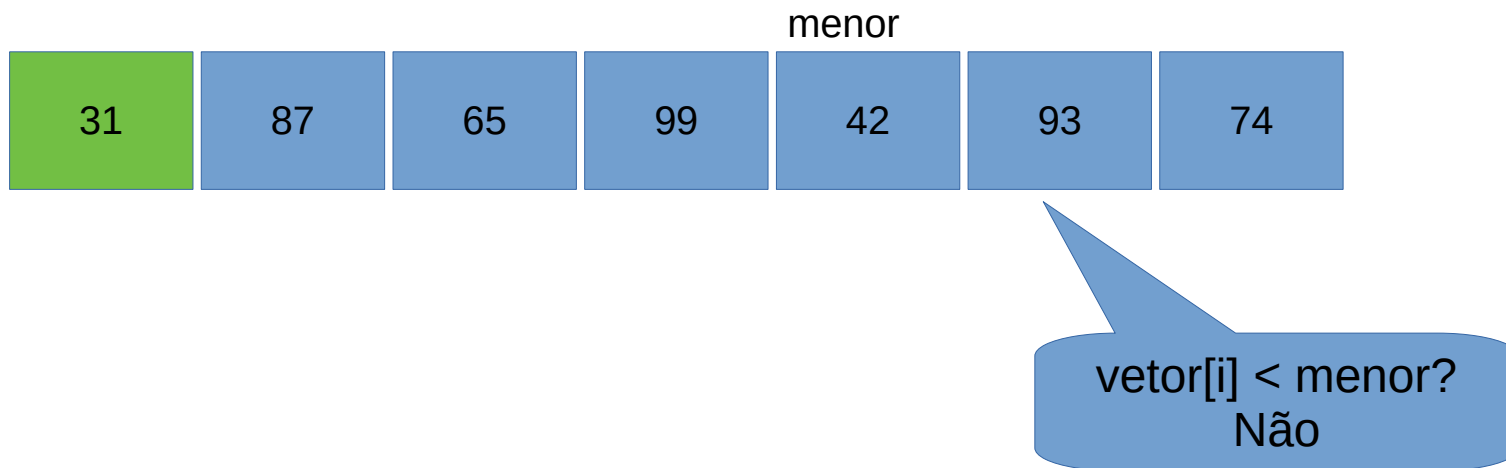
# Selection Sort

- Busca o menor
  - Varredura 02



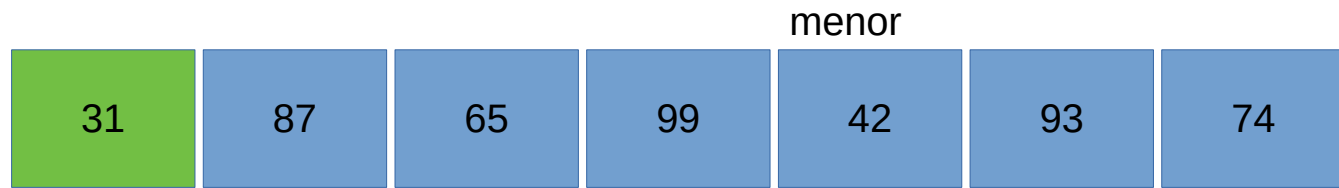
# Selection Sort

- Busca o menor
  - Varredura 02



# Selection Sort

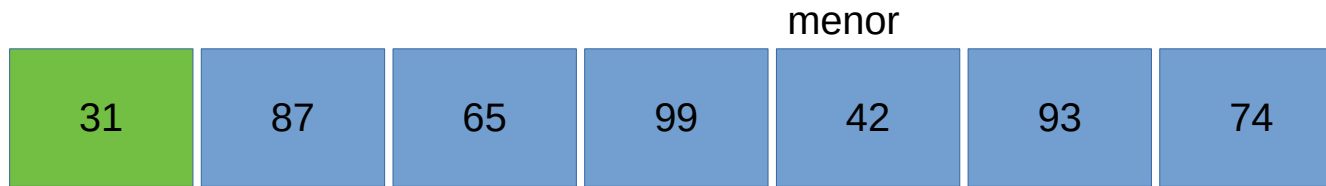
- Busca o menor
  - Varredura 02



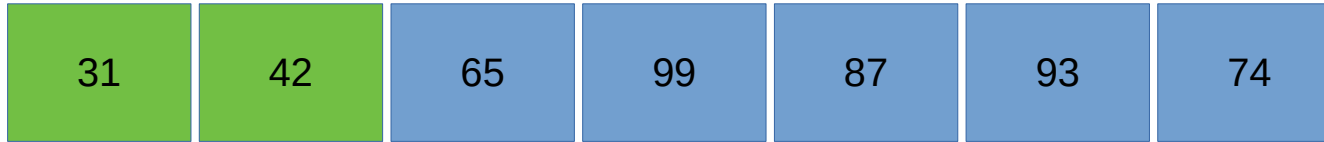
vetor[i] < menor?  
Não

# Selection Sort

- Troca o menor com a primeira posição

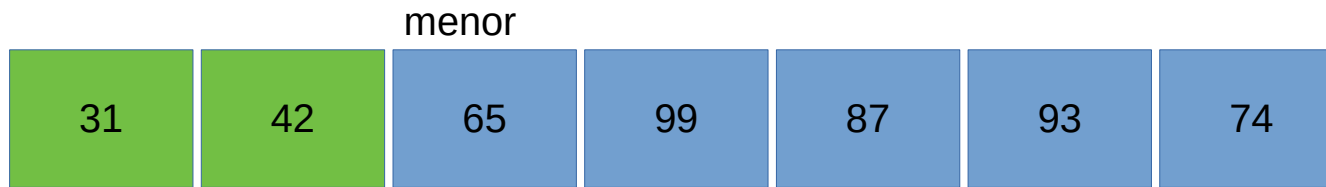


# Selection Sort



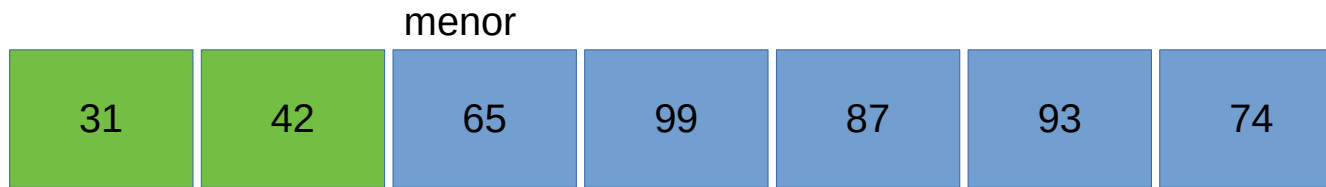
# Selection Sort

- Busca o menor
  - Varredura 03



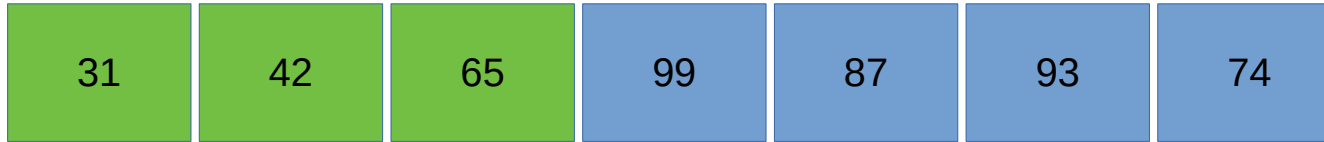
# Selection Sort

- Troca o menor com a primeira posição



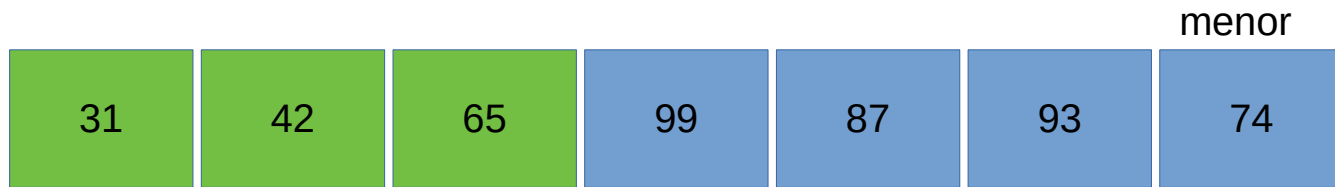


# Selection Sort



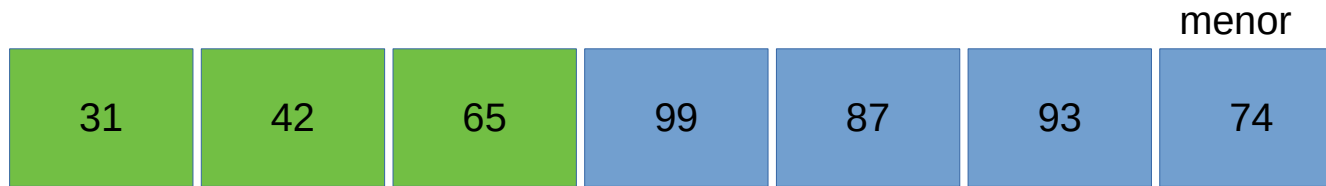
# Selection Sort

- Busca o menor
  - Varredura 04

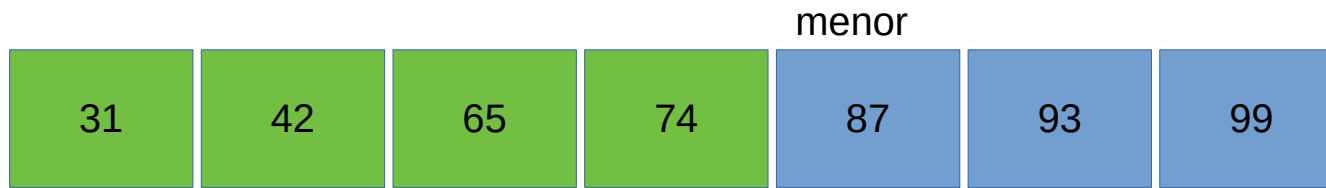


# Selection Sort

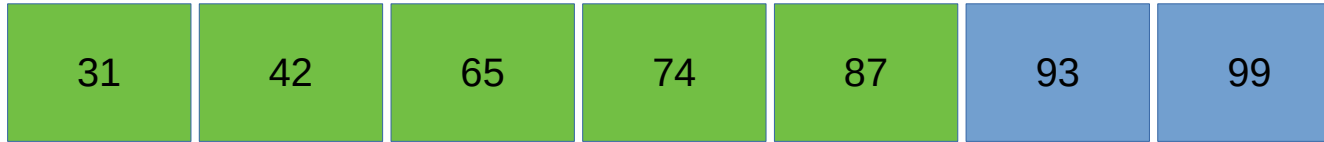
- Troca o menor com a primeira posição



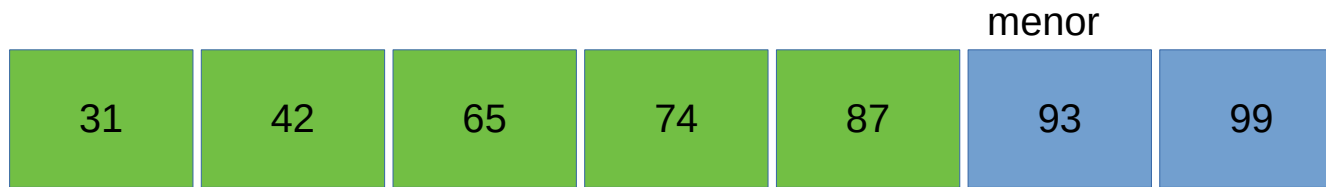
# Selection Sort



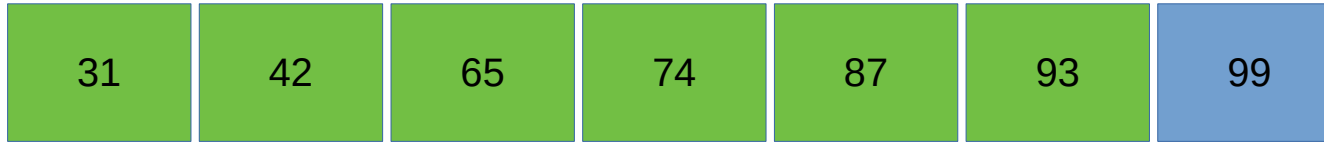
# Selection Sort



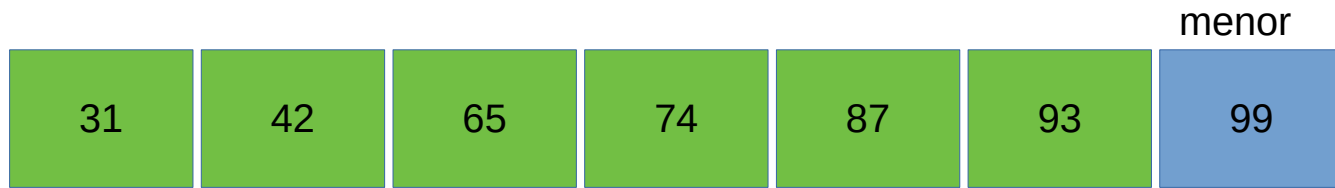
# Selection Sort



# Selection Sort

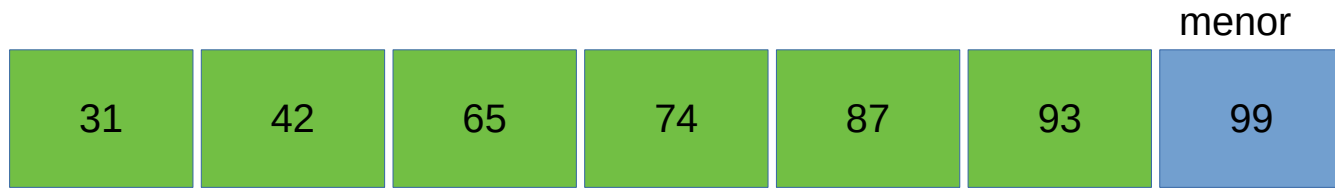


# Selection Sort

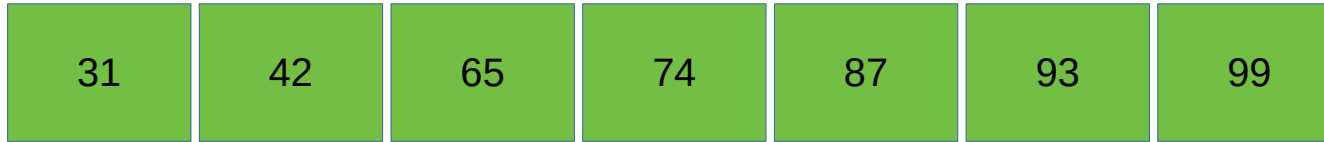




# Selection Sort



# Selection Sort



# 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*



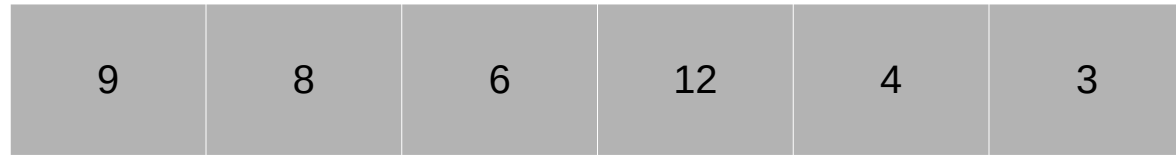
# *Insertion Sort*

- 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



Comparação



(vetor [0] > vetor [1]) ?



SIM



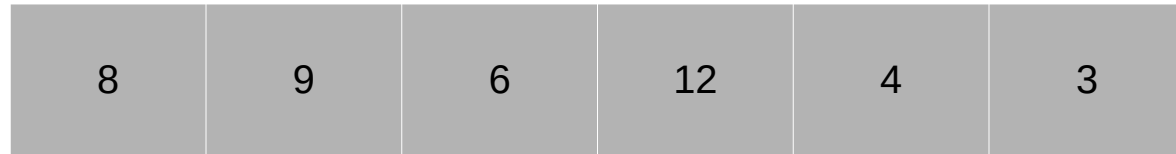
Troca



Fim Varredura 01

# Insertion Sort

- Varredura 02



Comparação



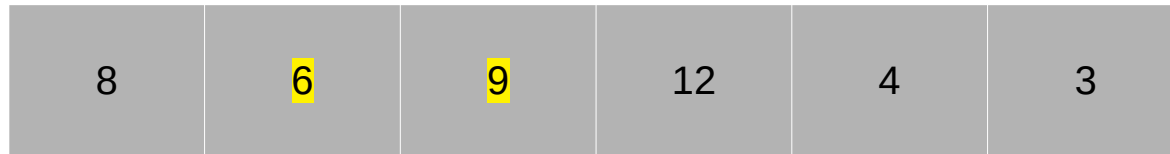
(vetor [1] > vetor [2]) ?



SIM

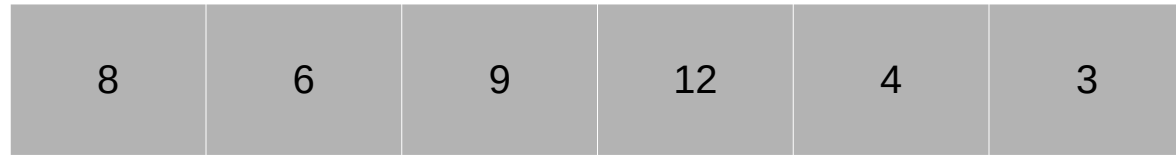


Troca



# Insertion Sort

- Varredura 02



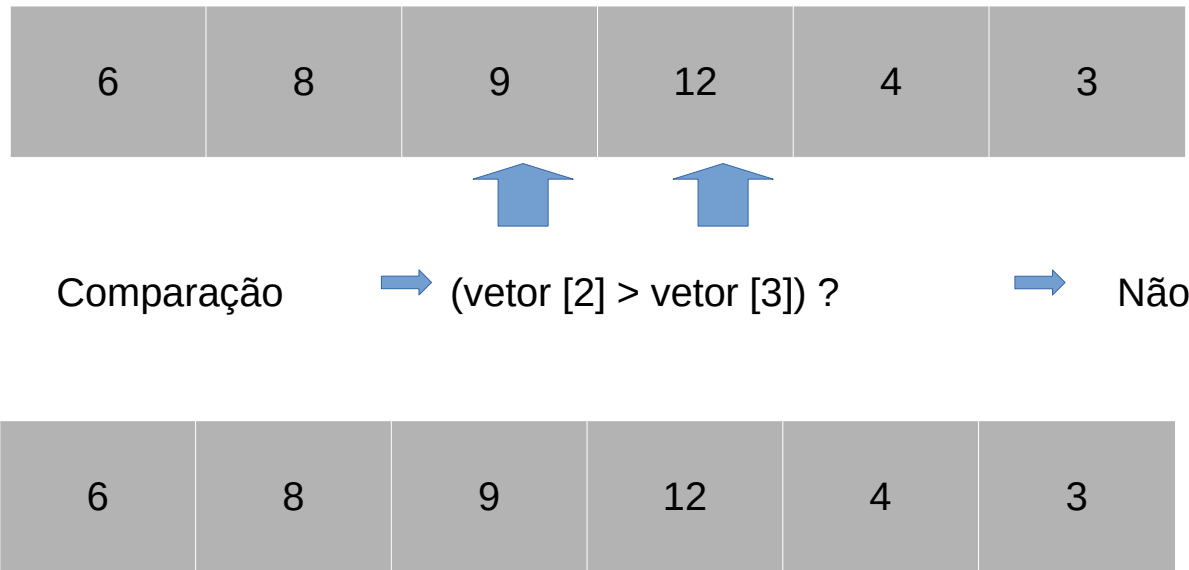
Comparaç o  $\rightarrow$  (vetor [0] > vetor [1]) ?  $\rightarrow$  SIM  $\rightarrow$  Troca



Fim Varredura 02

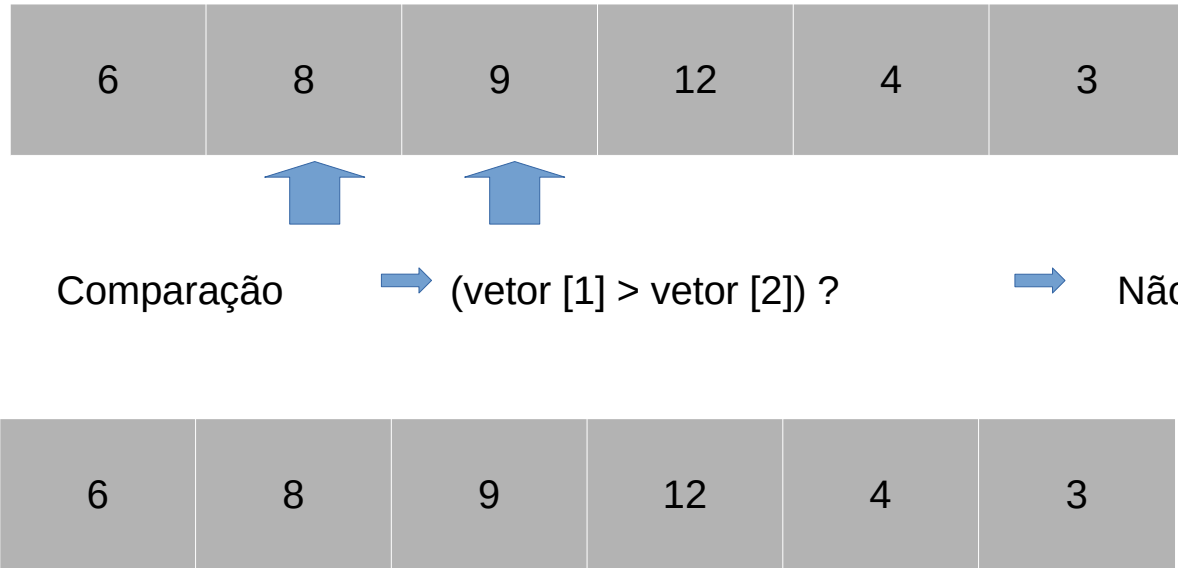
# Insertion Sort

- Varredura 03



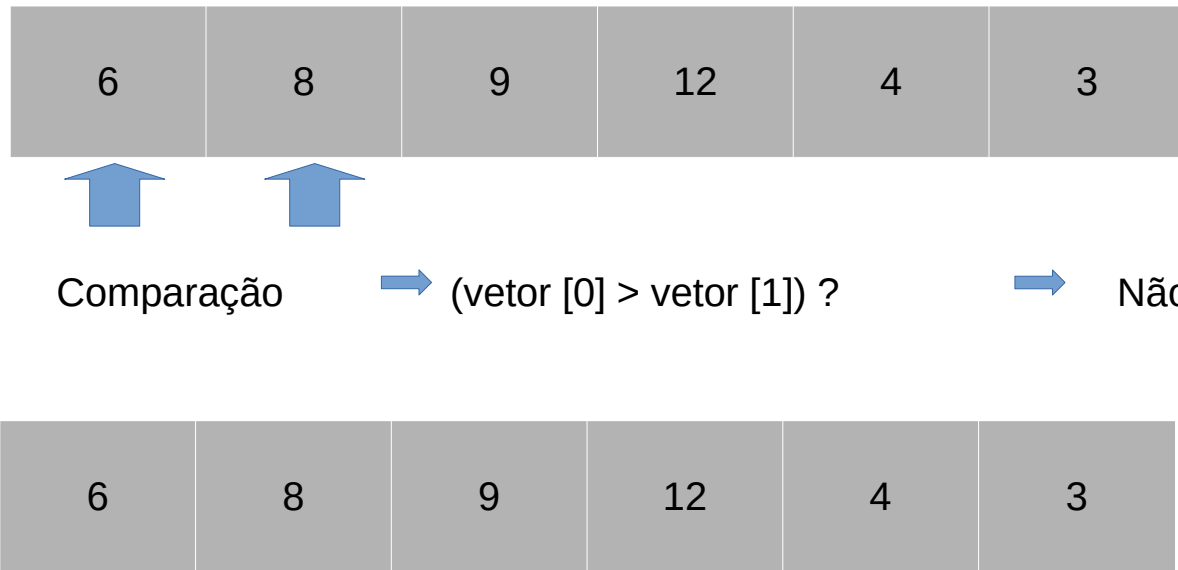
# Insertion Sort

- Varredura 03



# Insertion Sort

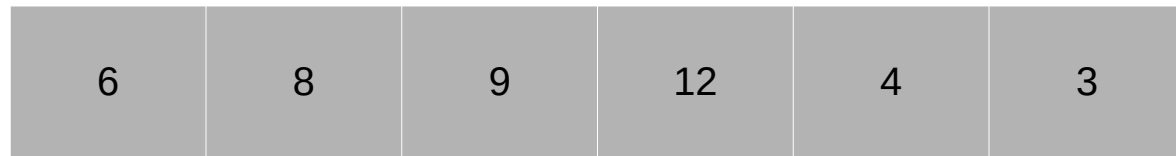
- Varredura 03



Fim Varredura 03

# Insertion Sort

- Varredura 04



Comparação



(vetor [3] > vetor [4]) ?



SIM

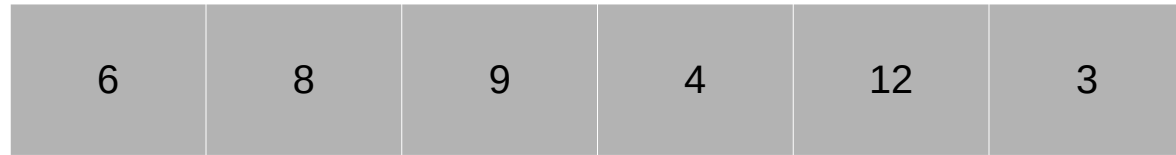


Troca



# Insertion Sort

- Varredura 04



Comparação



(vetor [2] > vetor [3]) ?



SIM



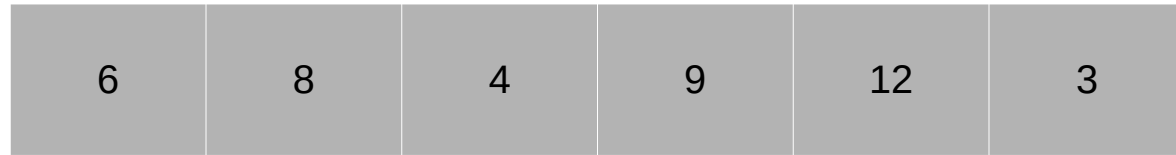
Troca





# Insertion Sort

- Varredura 04



Comparação



(vetor [1] > vetor [2]) ?



SIM

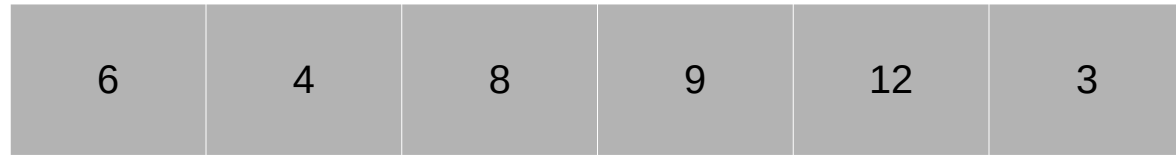


Troca



# Insertion Sort

- Varredura 04



Comparação



(vetor [0] > vetor [1]) ?



SIM



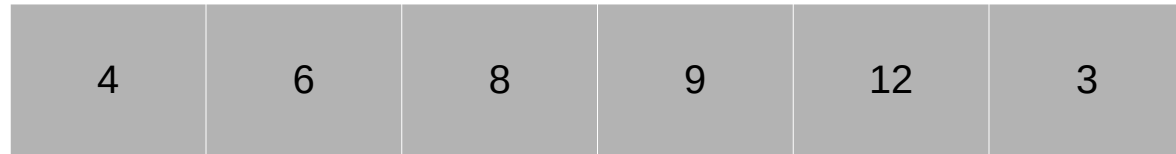
Troca



Fim da Varredura 04

# Insertion Sort

- Varredura 05



Comparação



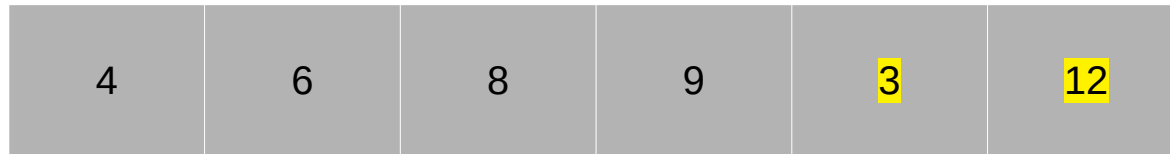
(vetor [4] > vetor [5]) ?



SIM

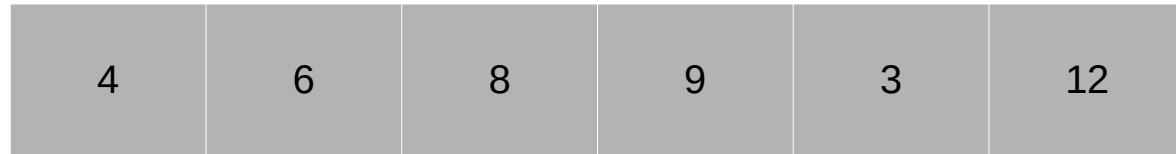


Troca



# Insertion Sort

- Varredura 05



Comparação



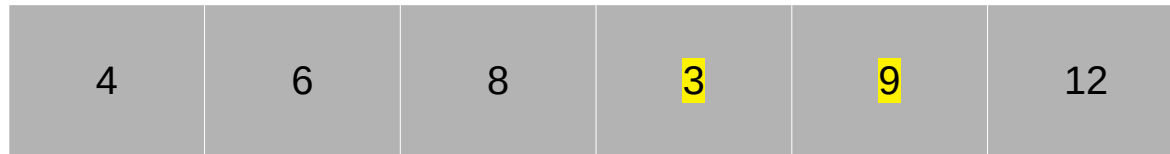
(vetor [3] > vetor [4]) ?



SIM

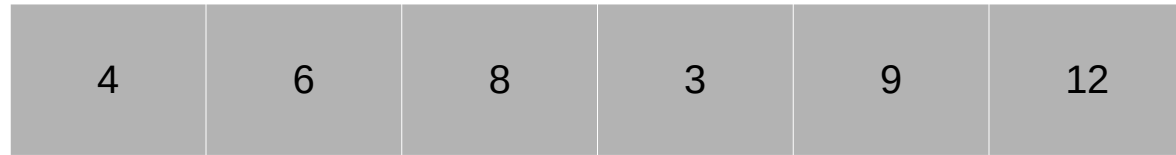


Troca



# Insertion Sort

- Varredura 05



Comparação



(vetor [2] > vetor [3]) ?



SIM

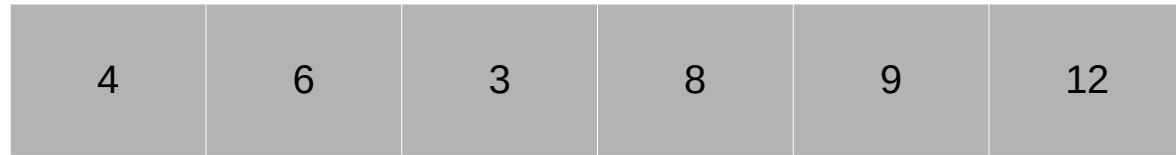


Troca



# Insertion Sort

- Varredura 05



Comparação



(vetor [1] > vetor [2]) ?



SIM

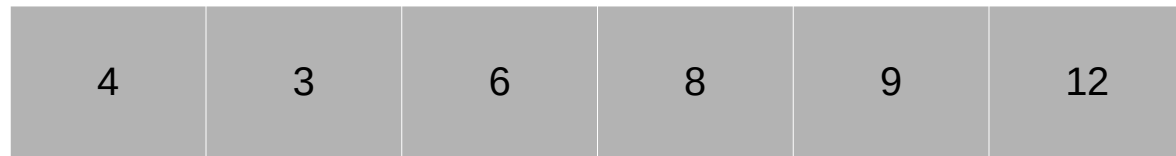


Troca



# Insertion Sort

- Varredura 05



Comparação



(vetor [0] > vetor [1]) ?



SIM



Troca



Fim da Varredura 05

# Algoritmo – *Insertion Sort*

```
insertionSort (A : lista de itens)
1. n = tamanho(A)
2. Para i = 1 até n-1
3.     j=i
4.     Enquanto j > 0 and A[j-1] > A[j]
5.         inverter(A[j], A[j-1])
6.         j=j-1
7.     fim-enquanto
8. fim-para
9. 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.

# Shell Sort

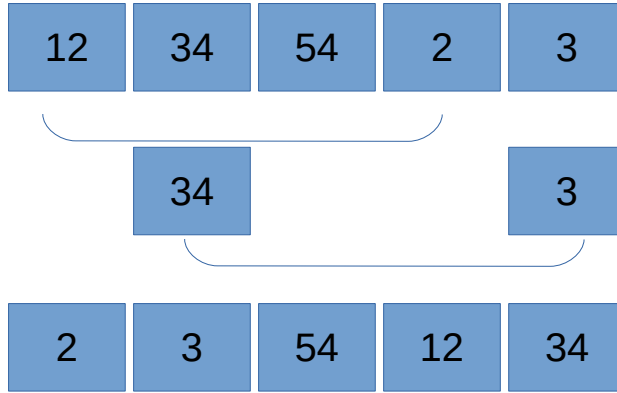
- Baseado no *Insertion Sort*;
- Realiza as mesmas operações do *Insertion Sort*.
- Porém para cada iteração, utiliza vetores que fazem parte da lista original, separados por um gap.
- Quando o gap é igual a um, ela funciona como o *Insertion Sort*.

# Shell Sort

- Mas por que utilizar esses gaps?
  - Trazer elementos que estão muito distantes de sua posição correta mais rapidamente;
  - Lembre do pior caso de todos os métodos, em que os elementos precisam fazer muitas comparações para chegar em sua posição correta
- Sucesso depende do gap escolhido
  - $\text{gap}(i) = 3 * \text{gap}(i-1) + 1, \text{gap}(i) < n$

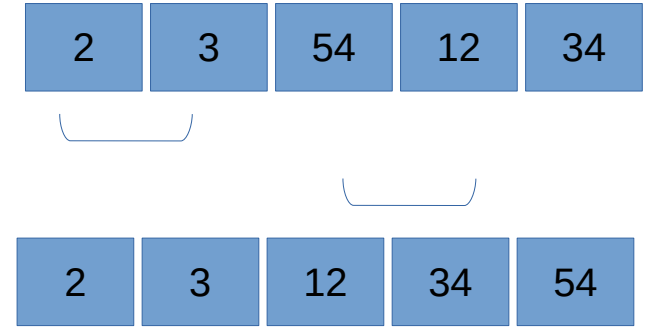
# Shell Sort

Gap = 2



Gap = 1

Gap = 0



# Algoritmo – *Shell Sort*

```
ShellSort (in arr[])
    n = arr.length;
    Para gap = n / 2 até 0
        Para i = gap até n
            temp = arr[i];
            Para j = i até gap && arr[j - gap] > temp
                arr[j] = arr[j - gap];
            Fim-para
            arr[j] = temp;
        Fim-para
    Fim-para
    Return 0;

Fim
```

# *Shell Sort*

- Complexidade ainda desconhecida
  - Sem expressão fechada;
  - Depende de fatores como a sequência dos gaps;
- Boa opção para conjuntos de tamanho médio;
- Instável – pode alterar a ordem de duas chaves iguais



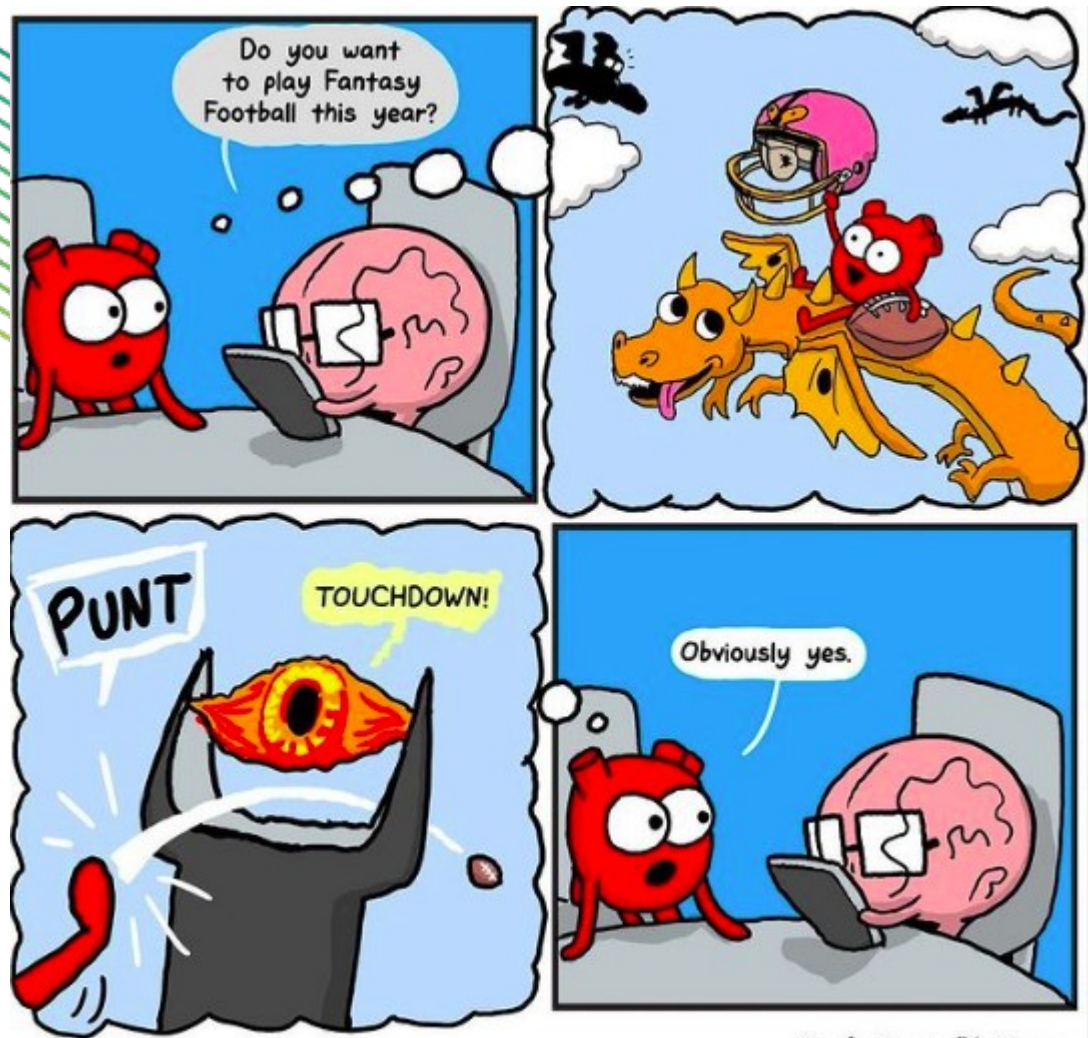
# Dúvidas



# Referência

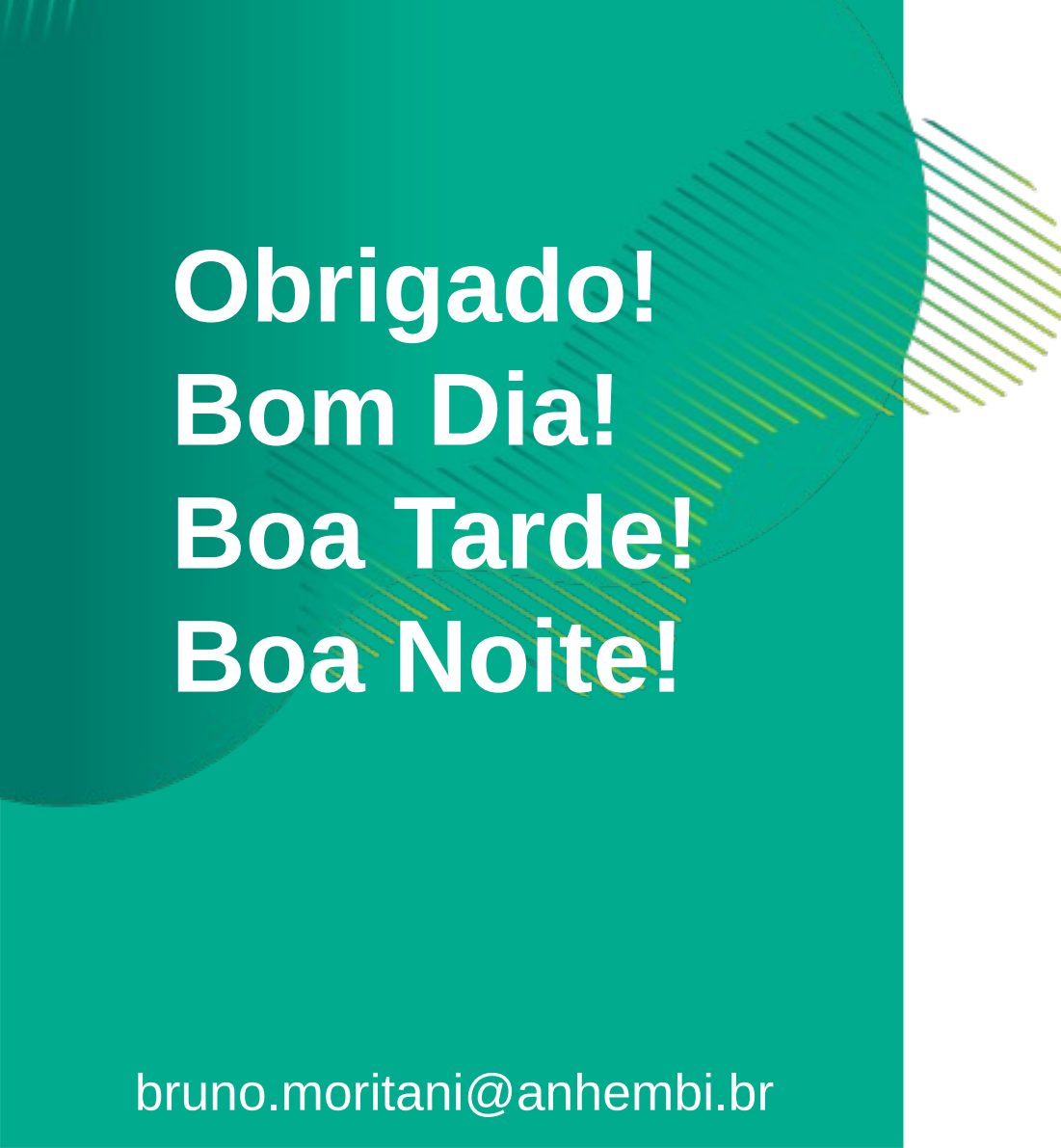
- DOBRUSHKIN, Vladimir A. Métodos para Análise de Algoritmos. LTC, 03/2012

# Exercícios



# Exercícios

- 1) Implemente o *selection sort* e teste com vetores de gerados randomicos.
- 2) Implemente o *insert sort* e teste com vetores de gerados randomicos.



**Obrigado!**  
**Bom Dia!**  
**Boa Tarde!**  
**Boa Noite!**

[bruno.moritani@anhembi.br](mailto:bruno.moritani@anhembi.br)