Universidade Federal do ABC
Centro de Matemática, Computação e Cognição

Algoritmos para Ordenação

Monael Pinheiro Ribeiro, D.Sc.

Problema da Ordenação

- Formalmente:
  - Suponha uma coleção **V** de elementos de tamanho **n**:

    $$\mathbf{V} = \{v_0, v_1, v_2, ..., v_{n-1}\}$$

  - Deseja-se que **V** tenha a seguinte propriedade:
    - $v_j \leq v_{j+1}$, $0 \leq j < n-1$, $\forall\ v_j \in \mathbf{V}$
- Informalmente:
  - Dado um vetor **V** de tamanho **n**. Garantir que após um procedimento os elementos de **V**, ou seja, $v_0, v_1, ..., v_{n-1}$ estejam ordenados de forma crescente.

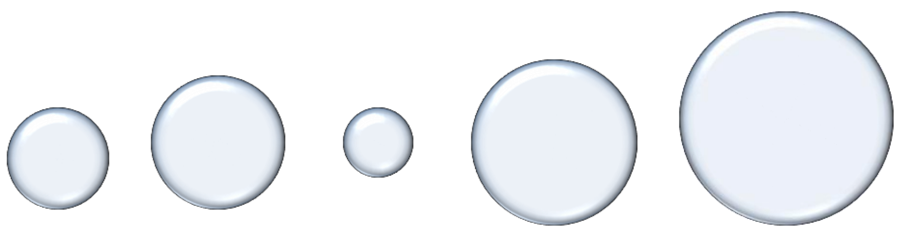
Problema da Ordenação

- Formalmente:
  - Suponha uma coleção **V** de elementos de tamanho **n**:

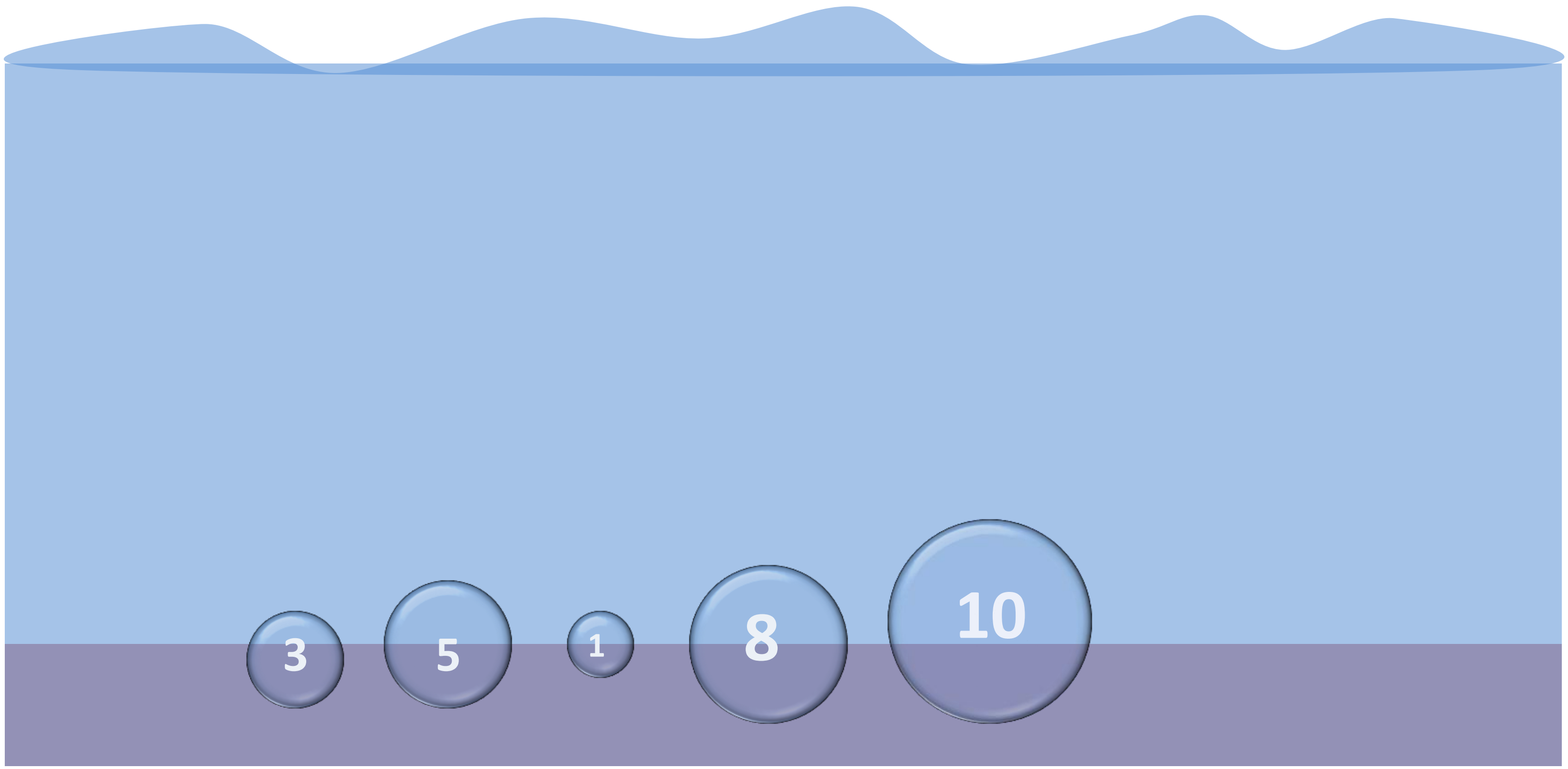
    $$\mathbf{V} = \{v_0, v_1, v_2, ..., v_{n-1}\}$$

  - Deseja-se que **V** tenha a seguinte propriedade:
    - $v_j \geq v_{j+1}$, $0 \leq j < n-1$, $\forall\, v_j \in \mathbf{V}$
- Informalmente:
  - Dado um vetor **V** de tamanho **n**. Garantir que após um procedimento os elementos de **V**, ou seja, $v_0, v_1, ..., v_{n-1}$ estejam ordenados de forma decrescente.

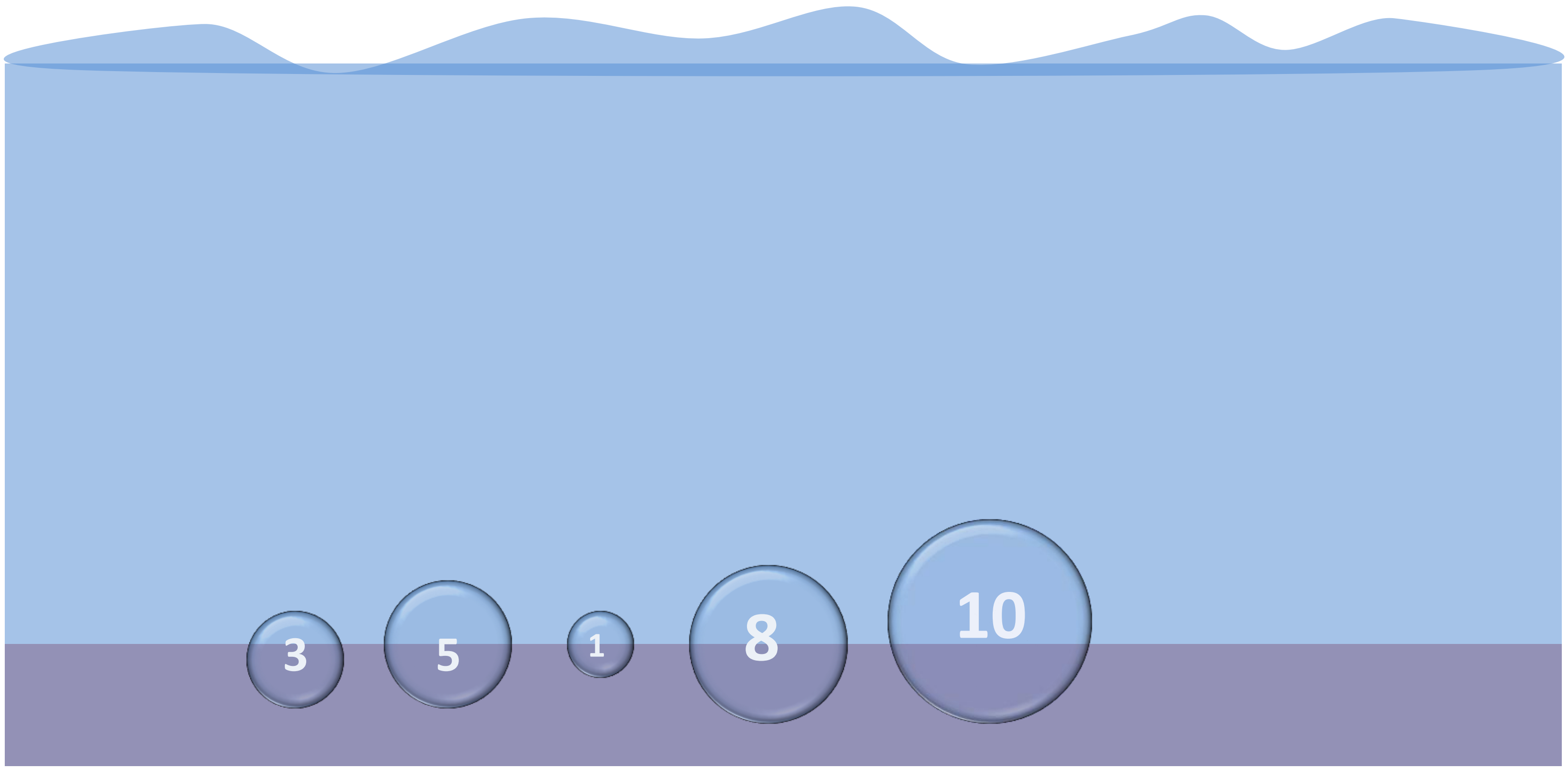
Método da Bolha



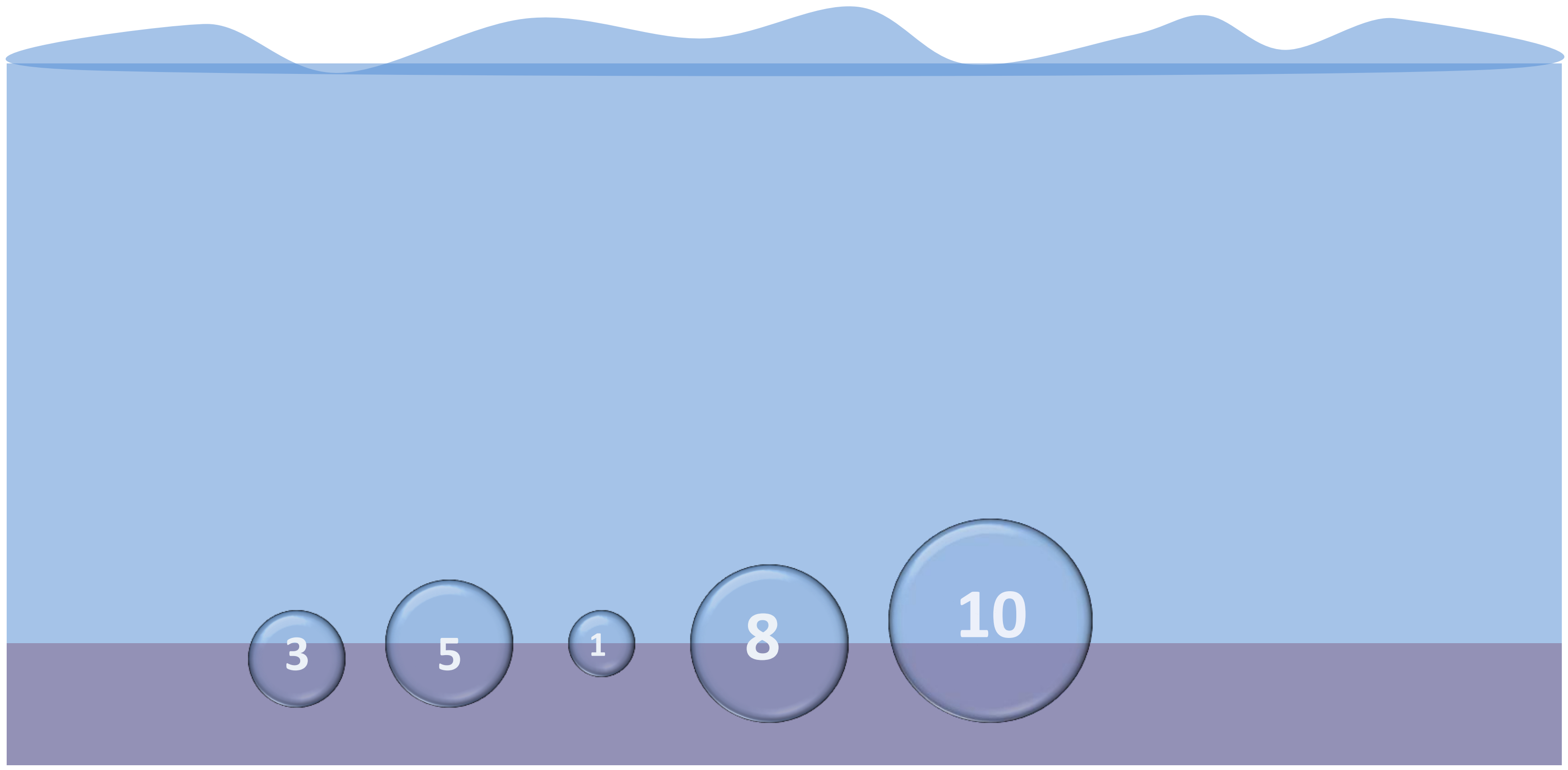
Método da Bolha



Método da Bolha



Método da Bolha



Bubble Sort

- Método da Bolha
  - O método bolha de ordenação usa justamente o conceito das bolha no meio fluídico.
  - Mecanismo:
    - Compara-se os n elementos do vetor, dois a dois.
    - Trocando de posição, quando o i-ésimo elemento for maior que o (i+1)-ésimo elemento.
    - Repete-se para os n-1 elementos restantes.
    - Repete-se para os n-2 elementos restantes.
    - Até que só reste 1 elemento. Que por sua vez, já é ordenado.

Bubble Sort

0	1	2	3
4	6	9	1

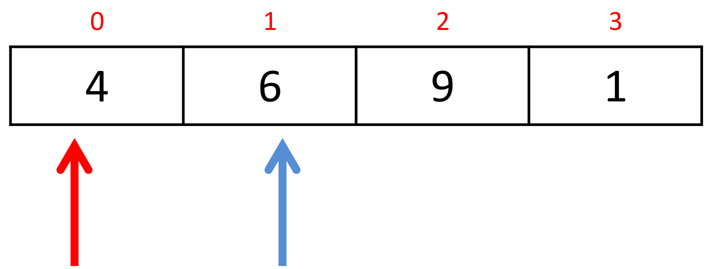
Bubble Sort

0	1	2	3
4	6	9	1

Bubble Sort

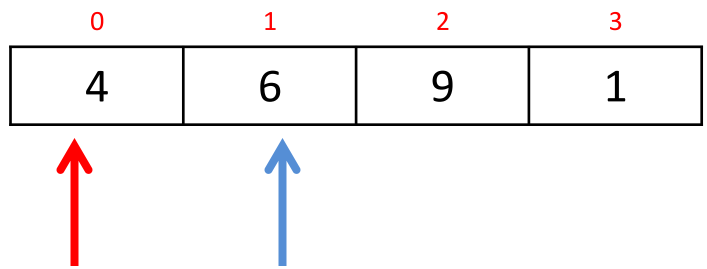
0	1	2	3
4	6	9	1

Bubble Sort



V[0] > V[1] ?

Bubble Sort



V[0] > V[1] ? Não

Bubble Sort

0	1	2	3
4	6	9	1

V[0] > V[1] ? Não
Vá para o próximo

Bubble Sort

0	1	2	3
4	6	9	1

V[1] > V[2] ?

Bubble Sort

0	1	2	3
4	6	9	1

V[1] > V[2] ? Não

Bubble Sort

0	1	2	3
4	6	9	1

V[1] > V[2] ? Não
Vá para o próximo

Bubble Sort

0	1	2	3
4	6	9	1

V[2] > V[3] ?

Bubble Sort

0	1	2	3
4	6	9	1

V[2] > V[3] ? Sim

Bubble Sort

0	1	2	3
4	6	9	1

V[2] > V[3] ? Sim
Então: Troque V[2] e V[3]

Bubble Sort

0	1	2	3
4	6	1	9

V[2] > V[3] ? Sim
Então: Troque V[2] e V[3]
Vá para o próximo

O maior elemento do vetor V de n posições está na posição n-1 (3) do vetor.
Agora reaplique o procedimento no vetor excluindo a posição n-1 (3).

Bubble Sort

0	1	2	3
4	6	1	9

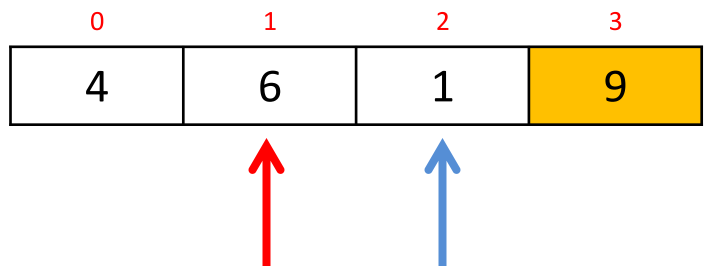
V[0] > V[1] ?

Bubble Sort

0	1	2	3
4	6	1	9

V[0] > V[1] ? Não
Vá para o próximo

Bubble Sort



V[1] > V[2] ?

Bubble Sort

0	1	2	3
4	6	1	9

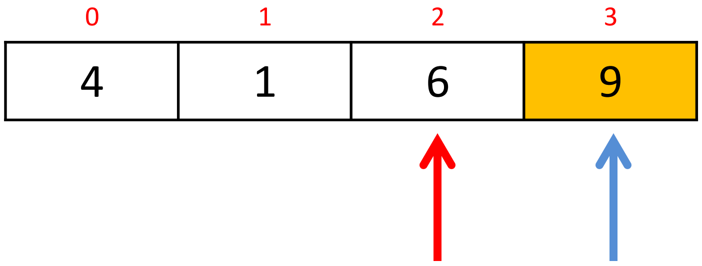
V[1] > V[2] ? Sim

Bubble Sort

0	1	2	3
4	1	6	9

V[1] > V[2] ? Sim
Então: Troque V[1] e V[2]
Vá para o próximo

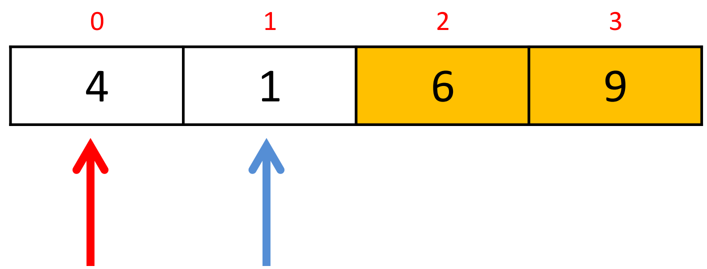
Bubble Sort



V[1] > V[2] ? Sim
Então: Troque V[1] e V[2]
Vá para o próximo

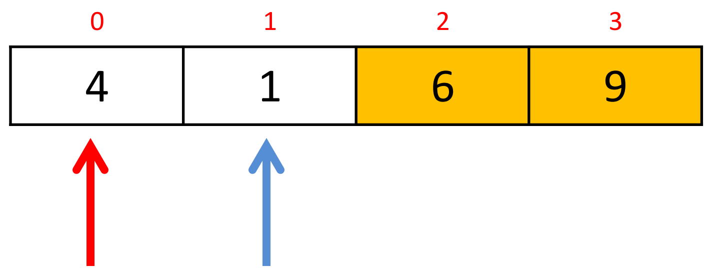
O maior elemento do subvetor V de n-1 posições está na posição n-2 (2) do vetor.
Agora reaplique o procedimento no vetor excluindo a posição n-2 (2) e n-1 (3).

Bubble Sort



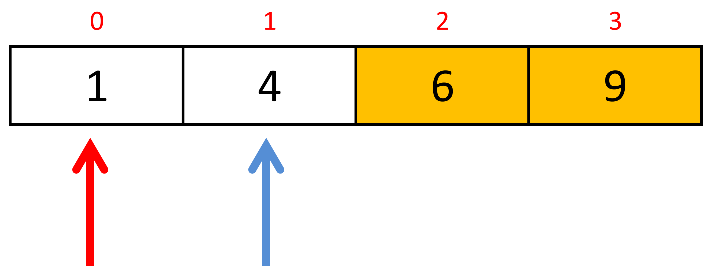
V[0] > V[1] ?

Bubble Sort



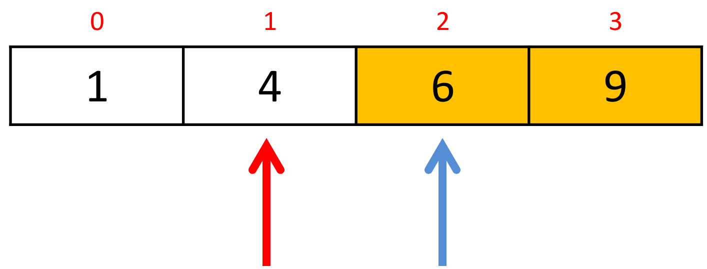
V[0] > V[1] ? Sim

Bubble Sort



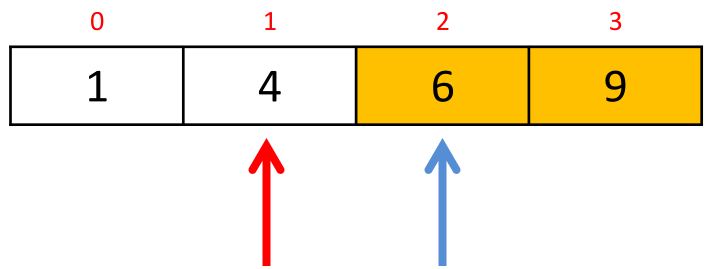
V[0] > V[1] ? Sim
Então: Troque V[1] e V[2]
Vá para o próximo

Bubble Sort



V[0] > V[1] ? Sim
Então: Troque V[1] e V[2]
Vá para o próximo

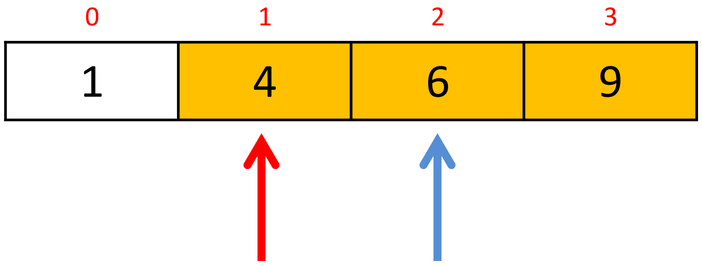
Bubble Sort



V[0] > V[1] ? Sim
Então: Troque V[1] e V[2]
Vá para o próximo

O maior elemento do subvetor V de n-2 posições
está na posição n-3 (1) do vetor.

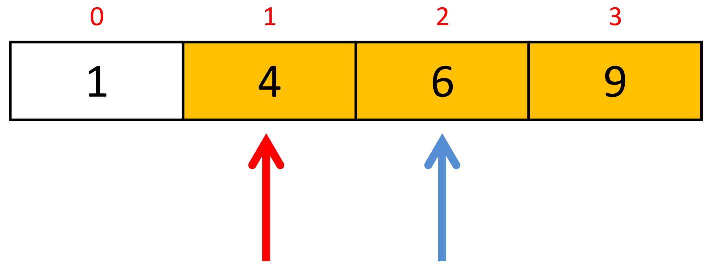
Bubble Sort



V[0] > V[1] ? Sim
Então: Troque V[1] e V[2]
Vá para o próximo

O maior elemento do subvetor V de n-2 posições
está na posição n-3 (1) do vetor.

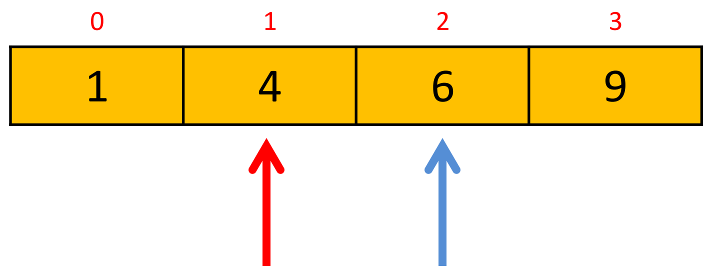
Bubble Sort



V[0] > V[1] ? Sim
Então: Troque V[1] e V[2]
Vá para o próximo

O maior elemento do subvetor V de n-2 posições está na posição n-3 (1) do vetor.
O subvetor restante tem tamanho 1. Um elemento só já está ordenado.
Portanto, o procedimento terminou.

Bubble Sort



V[0] > V[1] ? Sim
Então: Troque V[1] e V[2]
Vá para o próximo

O maior elemento do subvetor V de n-2 posições está na posição n-3 (1) do vetor.
O subvetor restante tem tamanho 1. Um elemento só já está ordenado.
Portanto, o procedimento terminou.

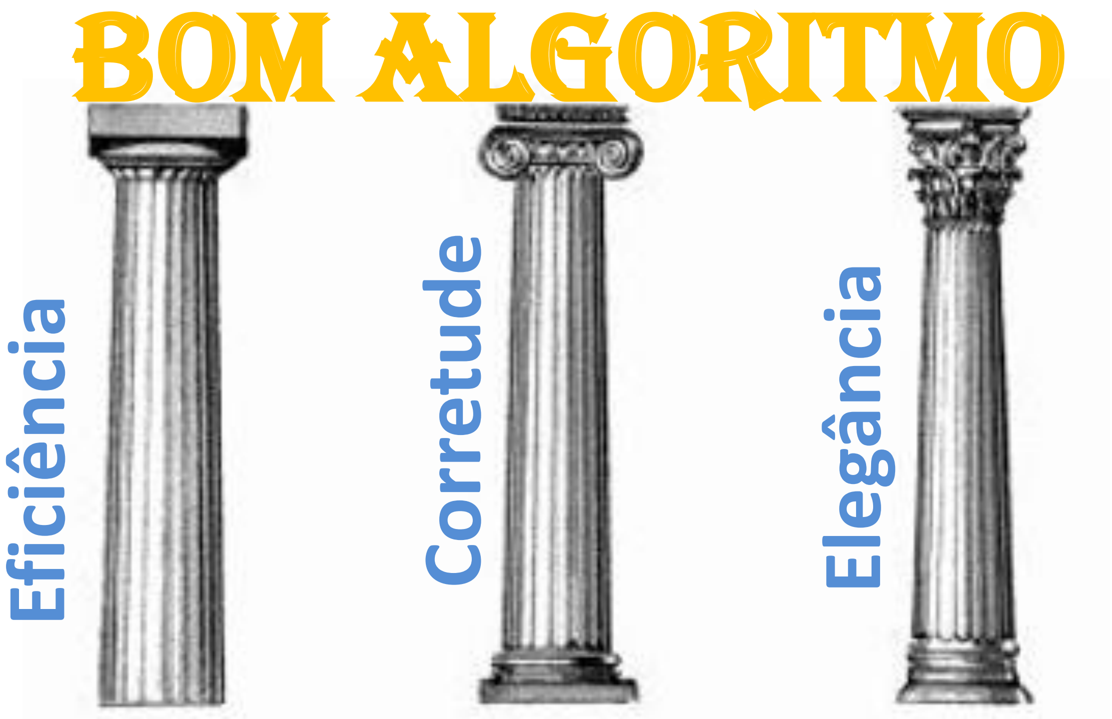
Bubble Sort

```
void bubbleSort(int *v, int n)
{
   int i, j, aux;
   for(i=0; i<n-1; i++)
   {
       for(j=0; j<n-1-i; j++)
       {
           if(v[j] > v[j+1])
           {
                aux = v[j];
                v[j] = v[j+1];
                v[j+1] = aux;
           }
       }
   }
}
```

Análise de Algoritmo

Um bom algoritmo deve ser sustentado por 3 características:

- A Eficiência: O algoritmo deve realizar sua tarefa no menor tempo possível
- A Corretude: O algoritmo deve ser correto. Ou seja, suas saídas devem ser exatas.
- A Elegância: O código deve ser limpo, facilmente entendido e bem organizado.



Corretude

- A corretude de um algoritmo é feita através de uma análise matemática de suas invariantes.
  - Invariante é uma relação entre os valores das variáveis que vale no início de cada iteração e não se altera de uma iteração para outra durante a execução do algoritmo.
  - As invariantes expressam o funcionamento do processo iterativo do algoritmo e PROVA por um processo indutivo que o algoritmo tem o efeito desejado para qualquer entrada.

Corretude

- A cada iteração de i o que ocorreu...

	0	1	2	3
Início:	4	6	9	1

Corretude

- A cada iteração de i o que ocorreu...

	0	1	2	3
Início:	4	6	9	1

Corretude

- A cada iteração de i o que ocorreu...

Início:

0	1	2	3
4	6	9	1

1ª iteração:

0	1	2	3
4	6	1	9

Corretude

- A cada iteração de i o que ocorreu...

	0	1	2	3
Início:	4	6	9	1

	0	1	2	3
1ª iteração:	4	6	1	9

	0	1	2	3
2ª iteração:	4	1	6	9

Corretude

- A cada iteração de i o que ocorreu...

	0	1	2	3
Início:	4	6	9	1
1ª iteração:	4	6	1	9
2ª iteração:	4	1	6	9
3ª iteração:	1	4	6	9

Corretude

- A cada iteração de i o que ocorreu...

	0	1	2	3
Início:	4	6	9	1
1ª iteração:	4	6	1	9
2ª iteração:	4	1	6	9
3ª iteração:	1	4	6	9
Final:	1	4	6	9

Corretude

- Invariante:
  - A cada iteração o vetor V é dividido em dois subvetores:
    - Um ordenado
    - Um não ordenado
  - A princípio o vetor ordenado tem tamanho 0 e o vetor não ordenado tem tamanho n (4). Mas isso se altera a cada iteração.
    - Início: Ordenado = 0, Não Ordenado = n (4)
    - 1ª iteração: Ordenado = 1, Não Ordenado = n-1 (3)
    - 2ª iteração : Ordenado = 2, Não Ordenado = n-2 (2)
    - 3ª iteração : Ordenado = 3, Não Ordenado = n-3 (1)
    - Final: Ordenado = 4, Não Ordenado = n-4 (0)
  - Para cada iteração de i se mantém a seguinte invariante:
    - $\forall k$, $n-i \leq k < n$ e $\forall i$, $0 \leq i < n$, vale: $v_k \leq v_{k+1}$
    - Ou seja, o subvetor $V_{k, \ldots, n-1}$, onde $k < i$ está ordenado.

Corretude

- Invariante:
  - Para cada iteração de i se mantém a seguinte invariante:
    - $\forall k$, $n-i \leq k < n$ e $\forall i$, $0 \leq i < n$, vale: $v_k \leq v_{k+1}$
    - Ou seja, o subvetor $V_{k, \ldots, n-1}$, onde $n-1 \leq k < n$ está ordenado.

i	n-i	n	k
0	4	4	
1	3	4	3
2	2	4	2,3
3	1	4	1,2,3
4	0	4	0,1,2,3

0	1	2	3
4	6	9	1

0	1	2	3
4	6	1	9

0	1	2	3
4	1	6	9

0	1	2	3
1	4	6	9

0	1	2	3
1	4	6	9

Eficiência

- Qual o esforço computacional necessário para o Bubble Sort ordenar n elementos no pior caso?
  - Qual a função primitiva do bubble sort?

```
void bubbleSort(int *v, int n)
{
   int i, j, aux;
   for(i=0; i<n-1; i++)
   {
       for(j=0; j<n-1-i; j++)
       {
           if(v[j] > v[j+1])
           {
                aux = v[j];
                v[j] = v[j+1];
                v[j+1] = aux;
           }
       }
   }
}
```

Eficiência

- Quantas vezes ocorre a função primitiva no pior caso?
  - Pense para o exemplo: n=4

i	j	Linha 8
0	0	1x
0	1	1x
0	2	1x
0	3	-
1	0	1x
1	1	1x
1	2	-
2	0	1x
2	1	-
3	-	-

Total: 6 x

Eficiência

- Quantas vezes ocorre a função primitiva no pior caso?
  - Agora, generalize para um vetor de tamanho n.

i	j	Linha 8
0	0,1,2, ..., n-1	(n-1) x
1	0,1,2, ..., n-2	(n-2) x
2	0,1,2, ..., n-3	(n-3) x
3	0,1,2, ..., n-4	(n-4) x
...	...	...
n-4	0, 1, 2, 3	3x
n-3	0, 1, 2	2x
n-2	0, 1	1x
n-1	-	-

Total: ?

Eficiência

- Quantas vezes ocorre a função primitiva no pior caso?
  - Agora, generalize para um vetor de tamanho n.

i	j	Linha 8
0	0,1,2, ..., n-1	(n-1) x
1	0,1,2, ..., n-2	(n-2) x
2	0,1,2, ..., n-3	(n-3) x
3	0,1,2, ..., n-4	(n-4) x
...	...	...
n-4	0, 1, 2, 3	3x
n-3	0, 1, 2	2x
n-2	0, 1	1x
n-1	-	-

Total: 1 + 2 + 3 + ... + (n-3) + (n-2) + (n-1)

Eficiência

- Quantas vezes ocorre a função primitiva no pior caso?
  - Agora, generalize para um vetor de tamanho n.

$$T(n) = 1 + 2 + 3 + \ldots + (n-3) + (n-2) + (n-1)$$

Eficiência

- Quantas vezes ocorre a função primitiva no pior caso?
  - Agora, generalize para um vetor de tamanho n.

$$T(n) = 1 + 2 + 3 + \ldots + (n-3) + (n-2) + (n-1)$$

$$T(n) = \frac{(n-1) \cdot (1 + (n-1))}{2}$$

Eficiência

- Quantas vezes ocorre a função primitiva no pior caso?
  - Agora, generalize para um vetor de tamanho n.

$$T(n) = 1 + 2 + 3 + \ldots + (n-3) + (n-2) + (n-1)$$

$$T(n) = \frac{(n-1) \cdot (1 + (n-1))}{2}$$

$$T(n) = \frac{n - 1 + (n-1)^2}{2}$$

Eficiência

- Quantas vezes ocorre a função primitiva no pior caso?
  – Agora, generalize para um vetor de tamanho n.

$$T(n) = 1 + 2 + 3 + \ldots + (n-3) + (n-2) + (n-1)$$

$$T(n) = \frac{(n-1) \cdot (1 + (n-1))}{2}$$

$$T(n) = \frac{n - 1 + (n-1)^2}{2}$$

$$T(n) = \frac{n - 1 + n^2 - 2n + 1}{2}$$

Eficiência

- Quantas vezes ocorre a função primitiva no pior caso?
  - Agora, generalize para um vetor de tamanho n.

$$T(n) = 1 + 2 + 3 + \cdots + (n-3) + (n-2) + (n-1)$$

$$T(n) = \frac{(n-1)\cdot(1+(n-1))}{2}$$

$$T(n) = \frac{n-1+(n-1)^2}{2}$$

$$T(n) = \frac{n-1+n^2-2n+1}{2}$$

$$T(n) = \frac{n^2-n}{2}$$

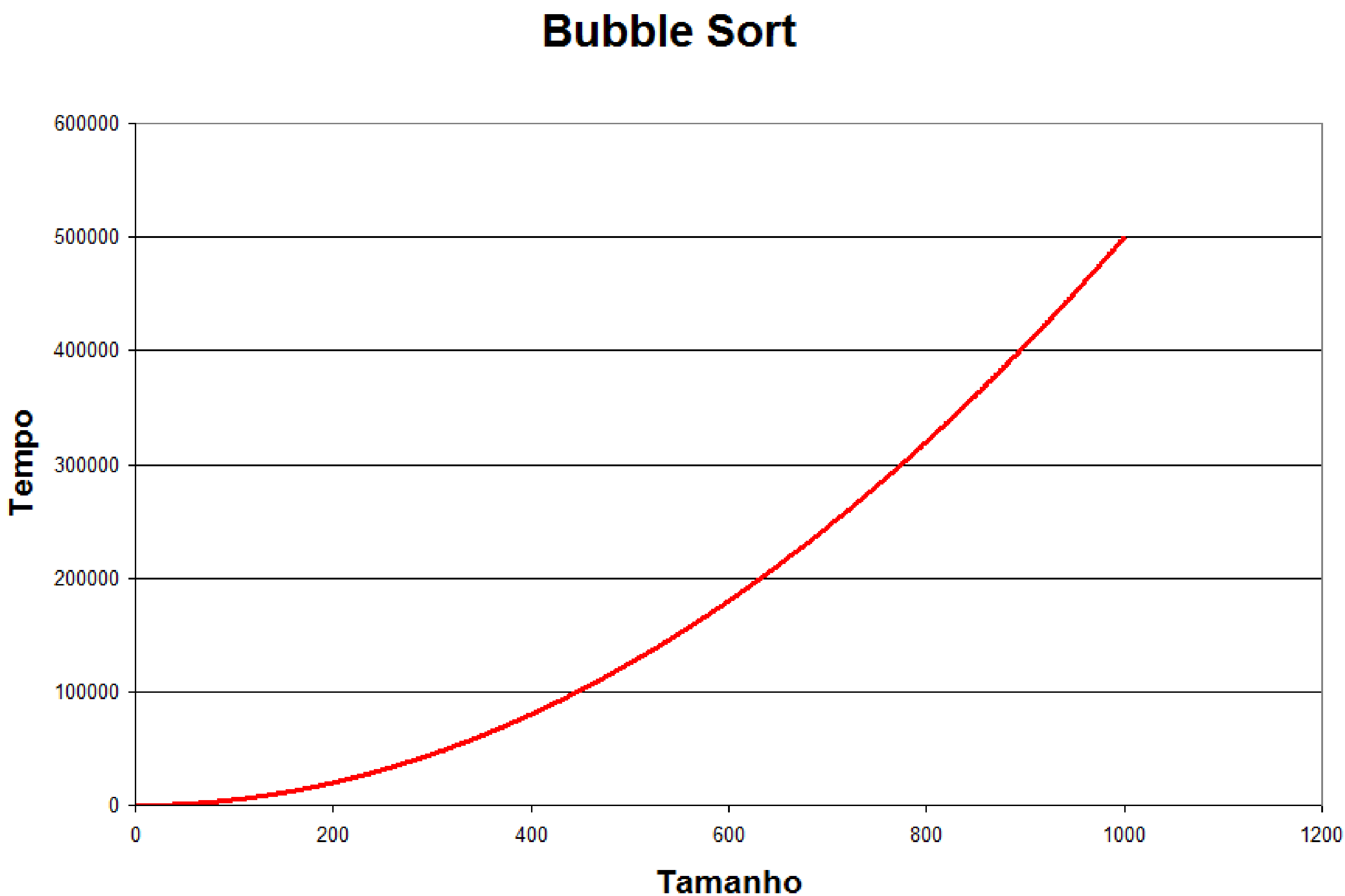
Eficiência

- Comportamento:

Tamanho	Comparações
4	6
10	45
20	190
30	435
100	4950
1000	499500
10000	49995000

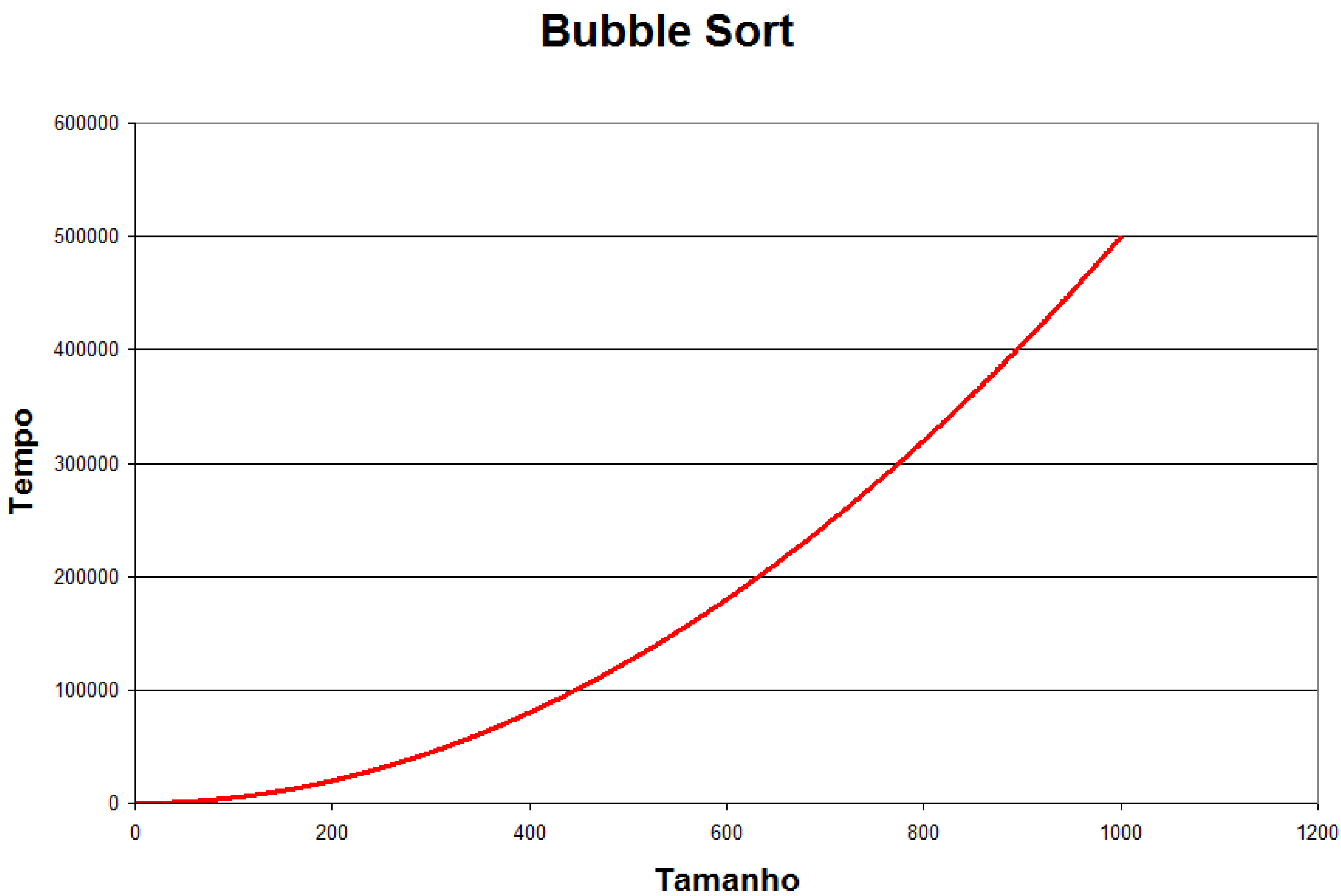
Eficiência

- Comportamento



Eficiência

- Comportamento



$O(n^2)$

Selection Sort

- O Selection Sort ordena um arranjo baseado em localizar o menor elemento e posicioná-lo na 1ª posição. Então encontra o segundo menor elemento e posiciona-o na 2ª posição e assim por diante. Para os n-1 elementos do vetor.

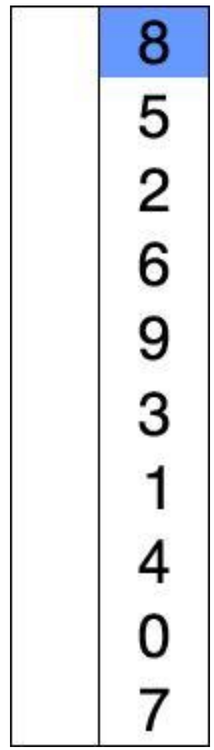
Selection Sort

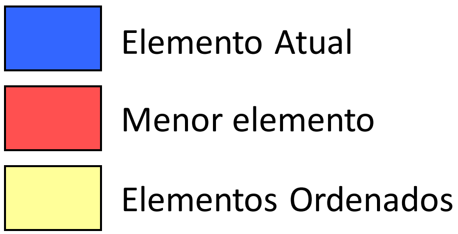
<u>Idéia</u>

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

Selection Sort

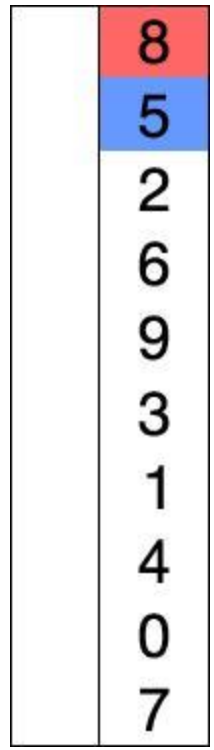
Idéia





Selection Sort

Idéia



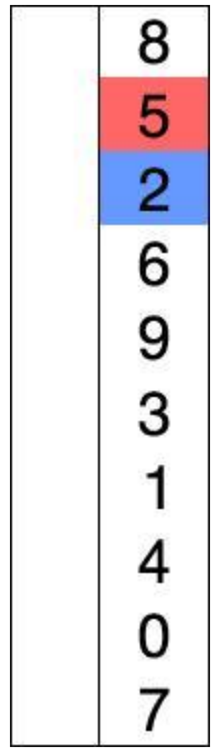
Elemento Atual

Menor elemento

Elementos Ordenados

Selection Sort

Idéia



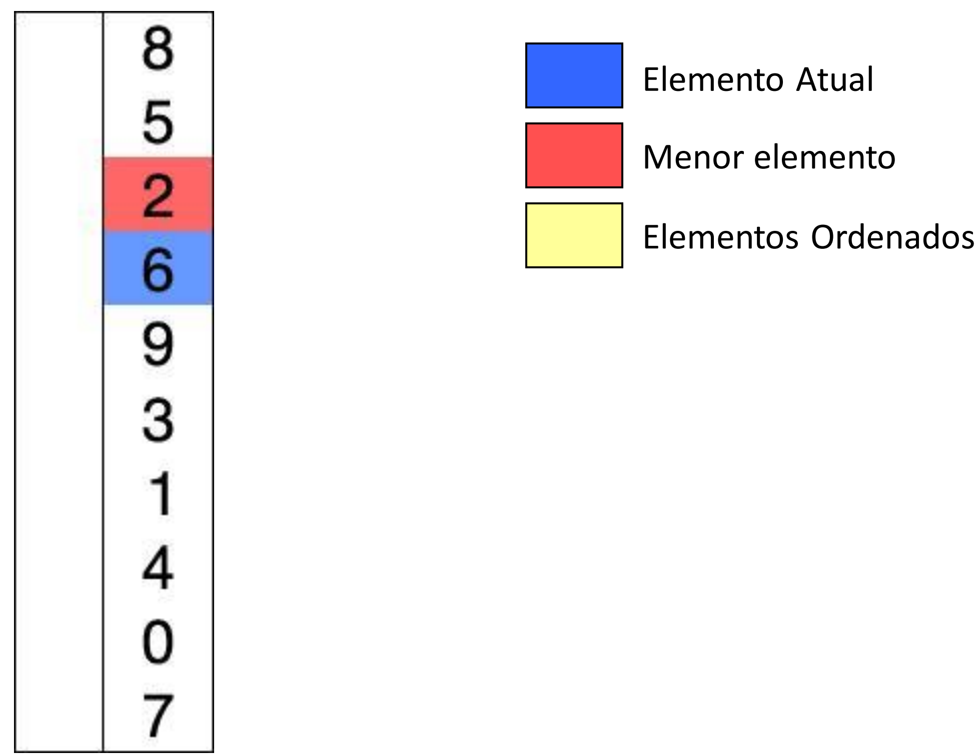
Elemento Atual

Menor elemento

Elementos Ordenados

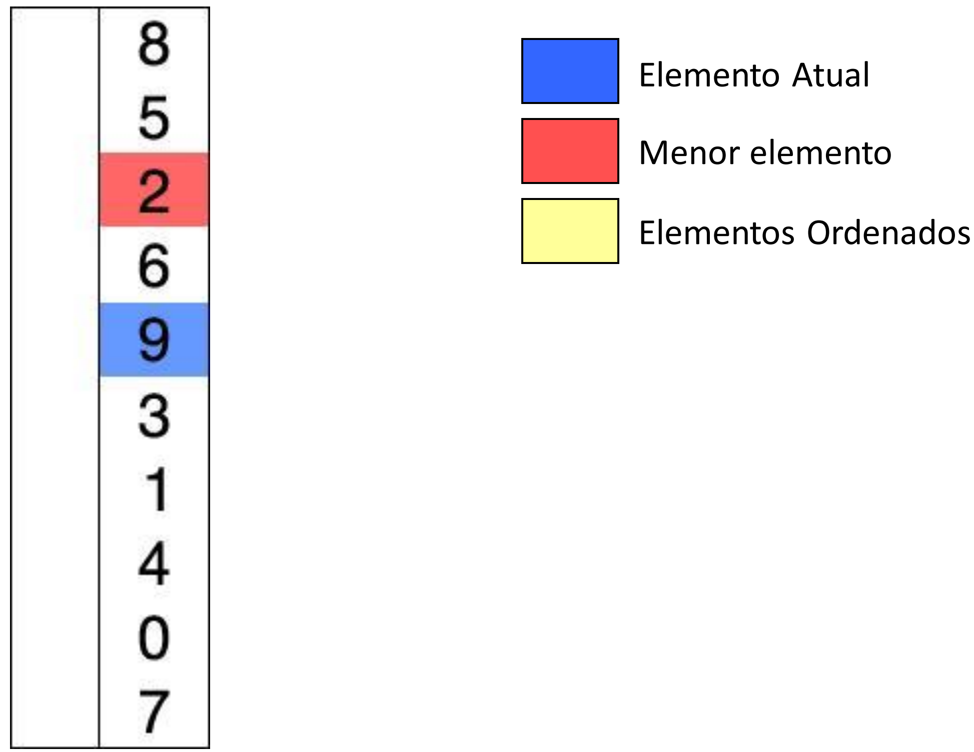
Selection Sort

Idéia



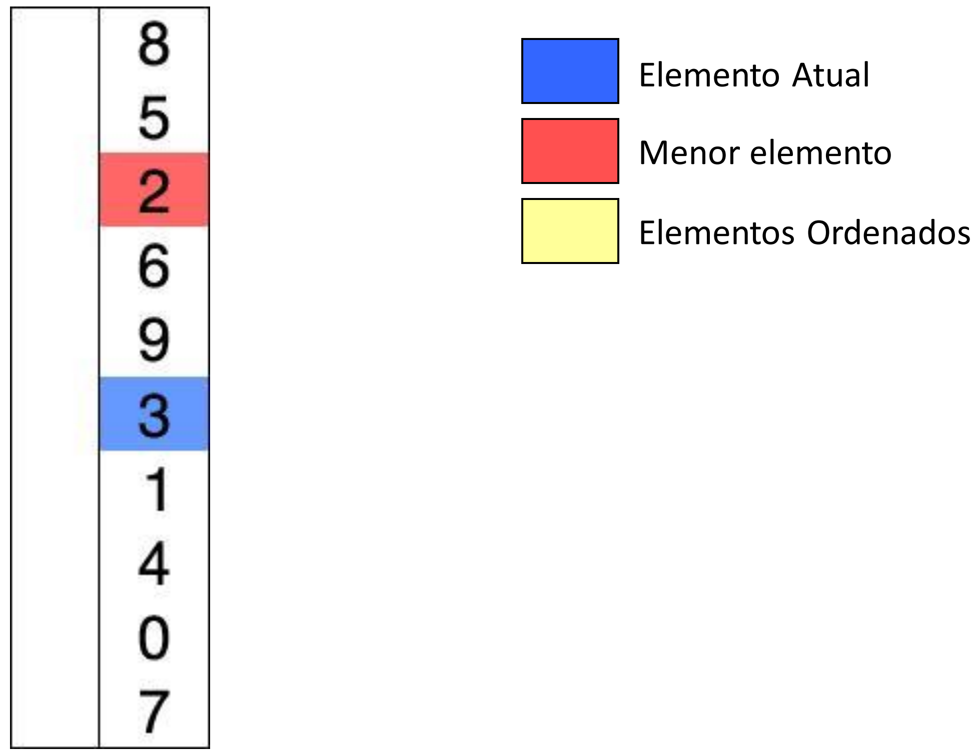
Selection Sort

Idéia



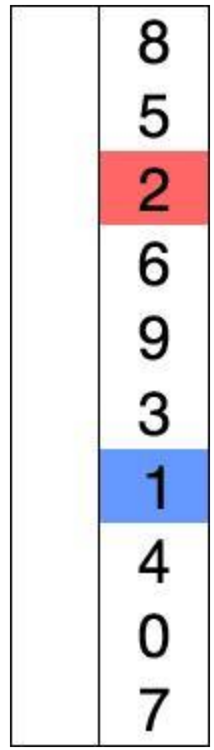
Selection Sort

Idéia



Selection Sort

Idéia



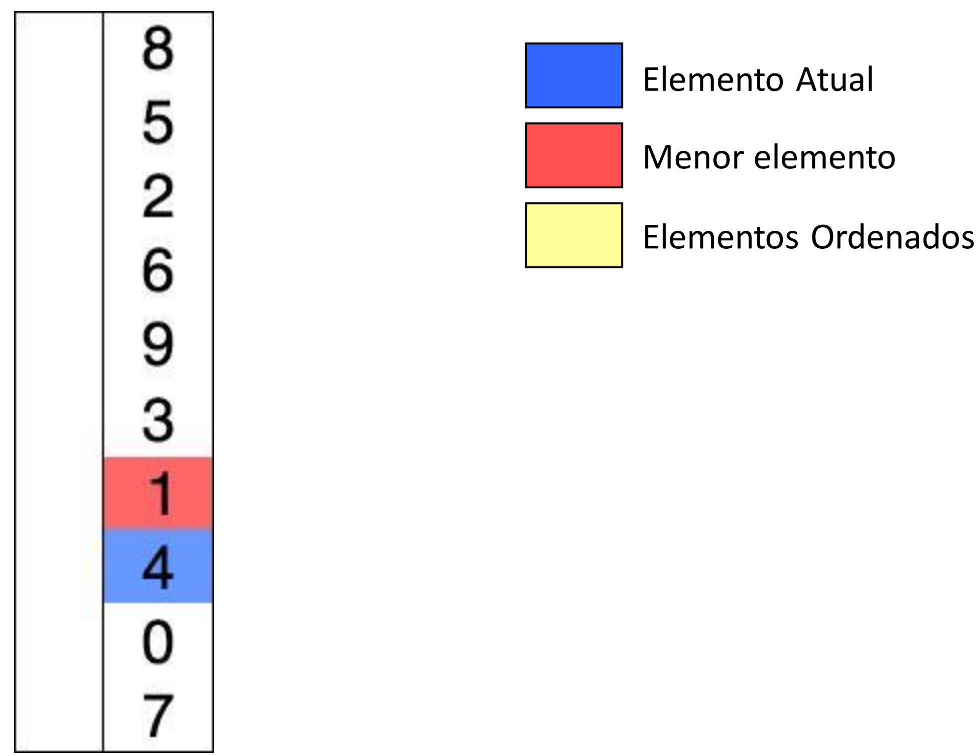
Elemento Atual

Menor elemento

Elementos Ordenados

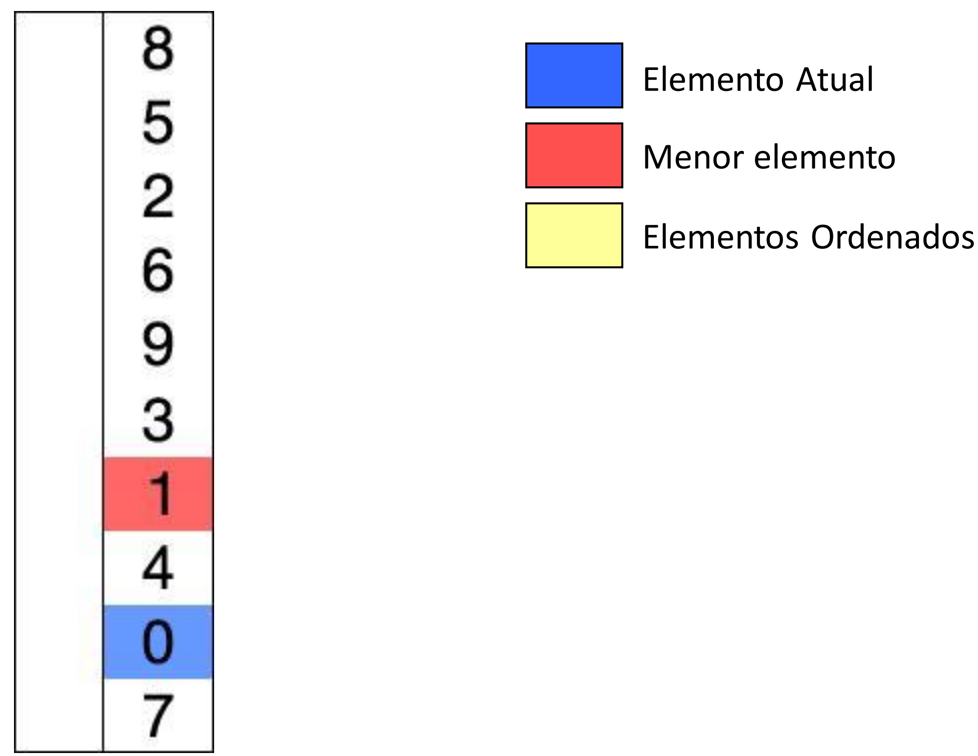
Selection Sort

Idéia



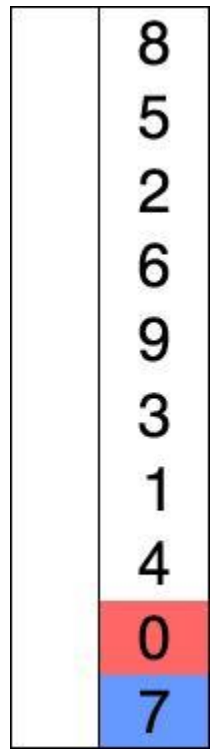
Selection Sort

<u>Idéia</u>



Selection Sort

Idéia



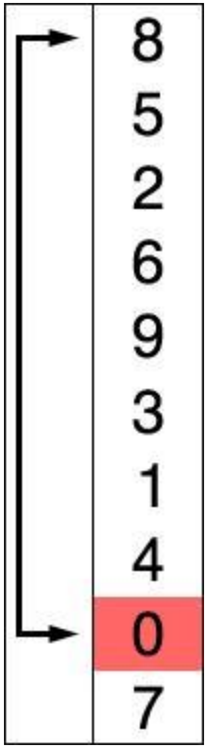
Elemento Atual

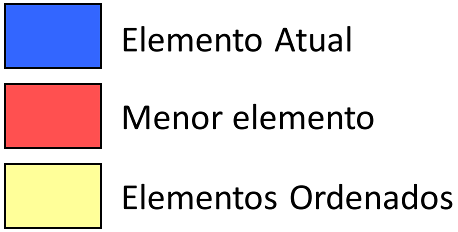
Menor elemento

Elementos Ordenados

Selection Sort

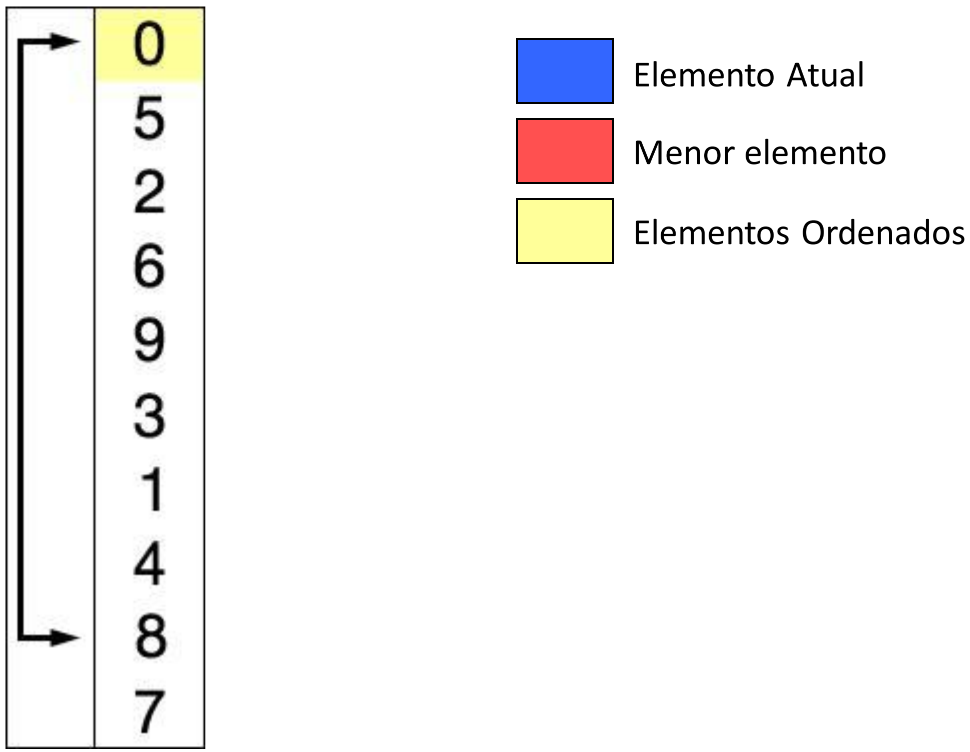
Idéia





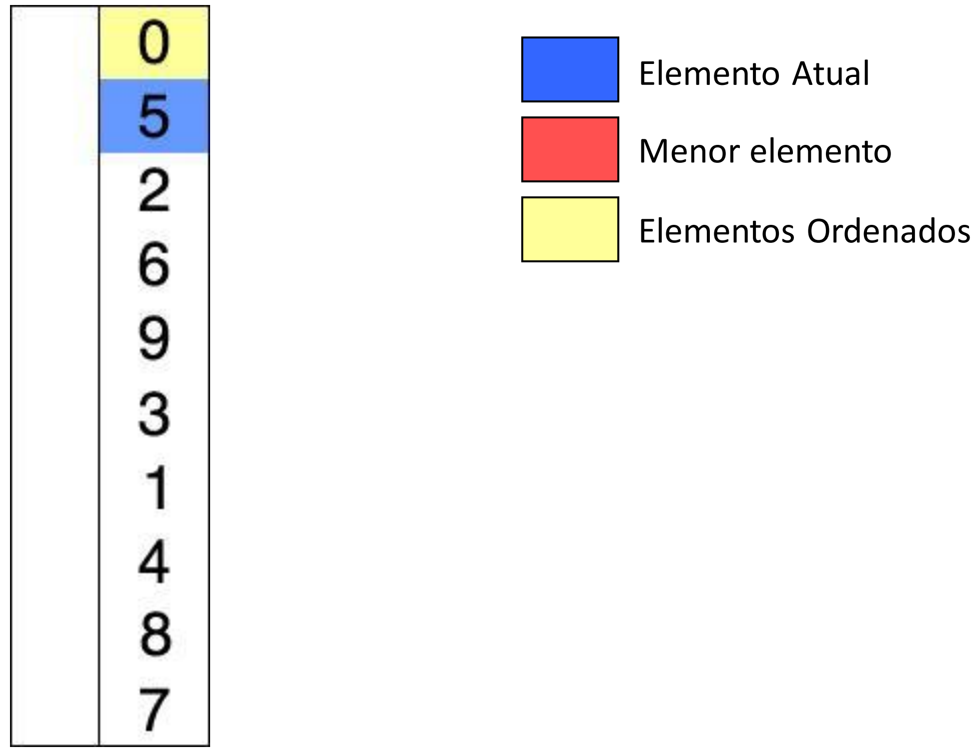
Selection Sort

Idéia



Selection Sort

Idéia

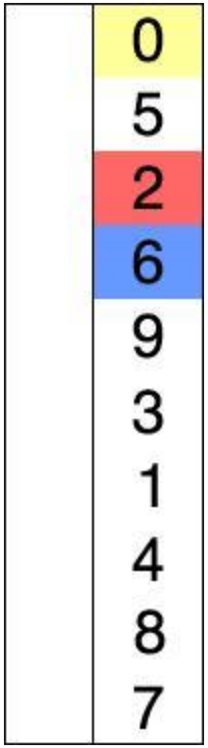


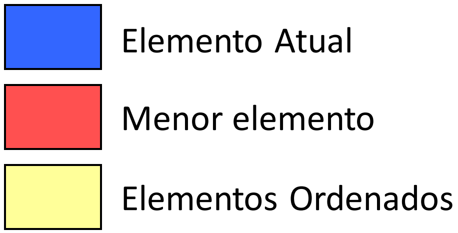
Selection Sort

Idéia

Selection Sort

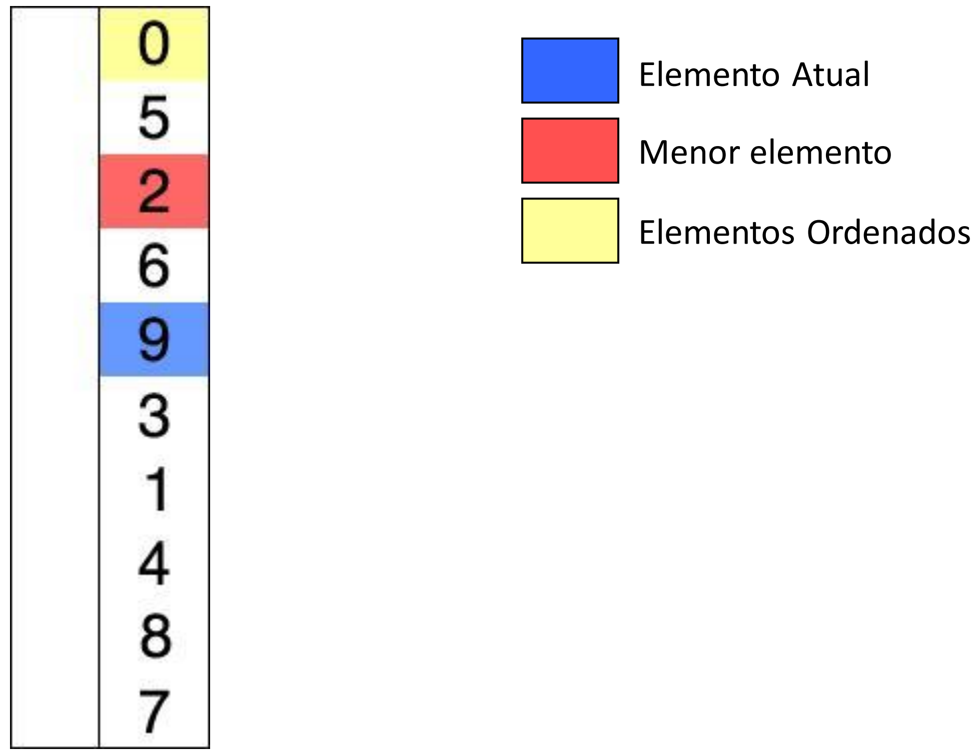
Idéia





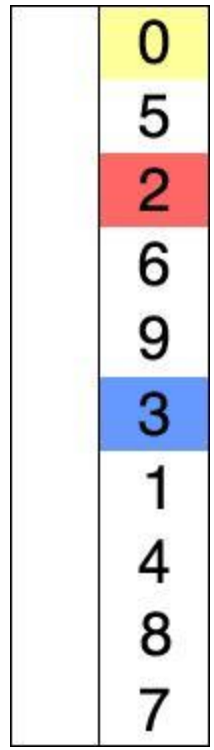
Selection Sort

Idéia



Selection Sort

Idéia



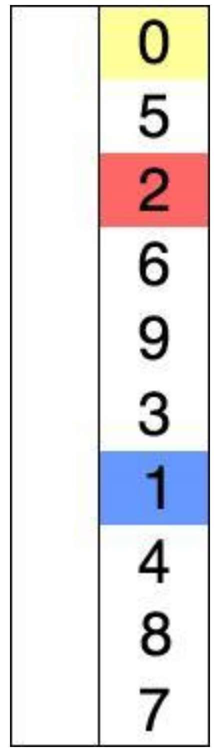
Elemento Atual

Menor elemento

Elementos Ordenados

Selection Sort

Idéia



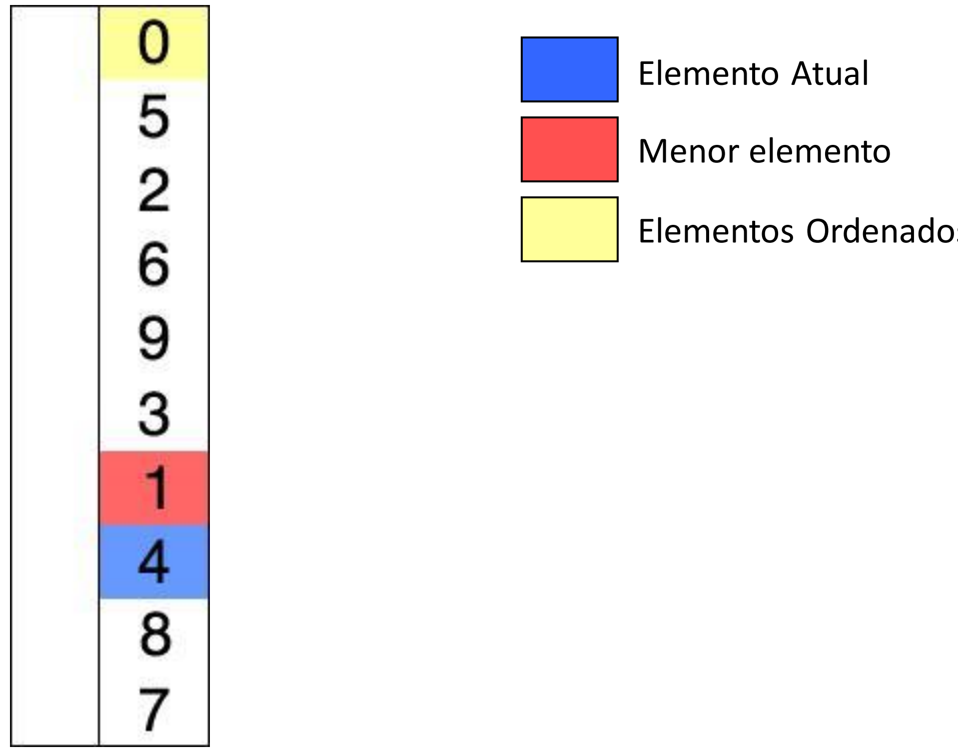
Elemento Atual

Menor elemento

Elementos Ordenados

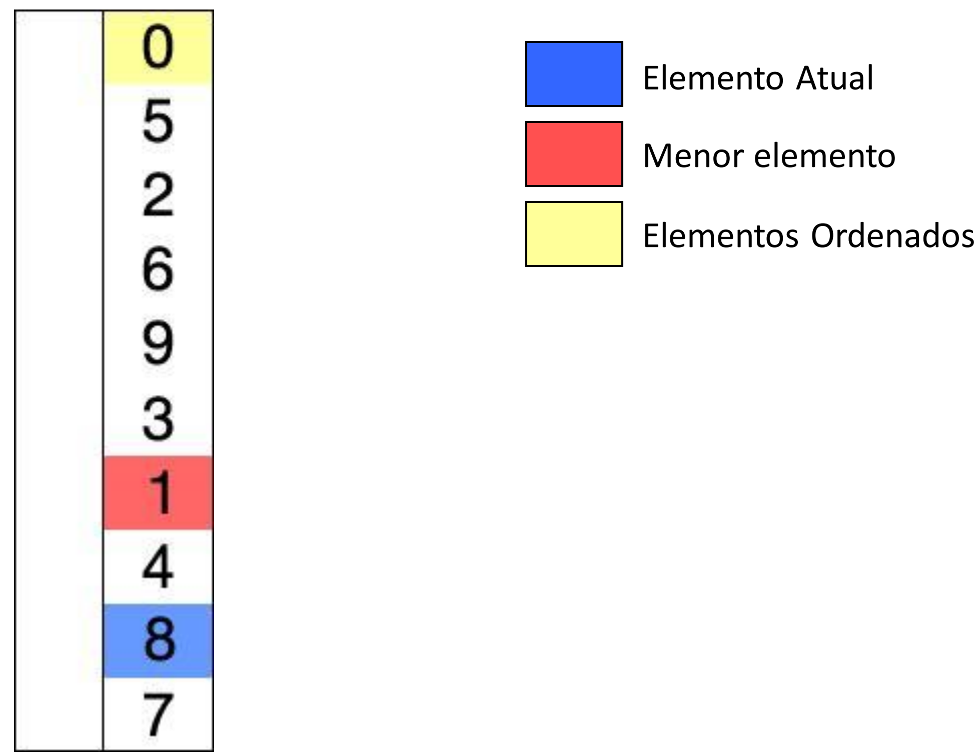
Selection Sort

Idéia



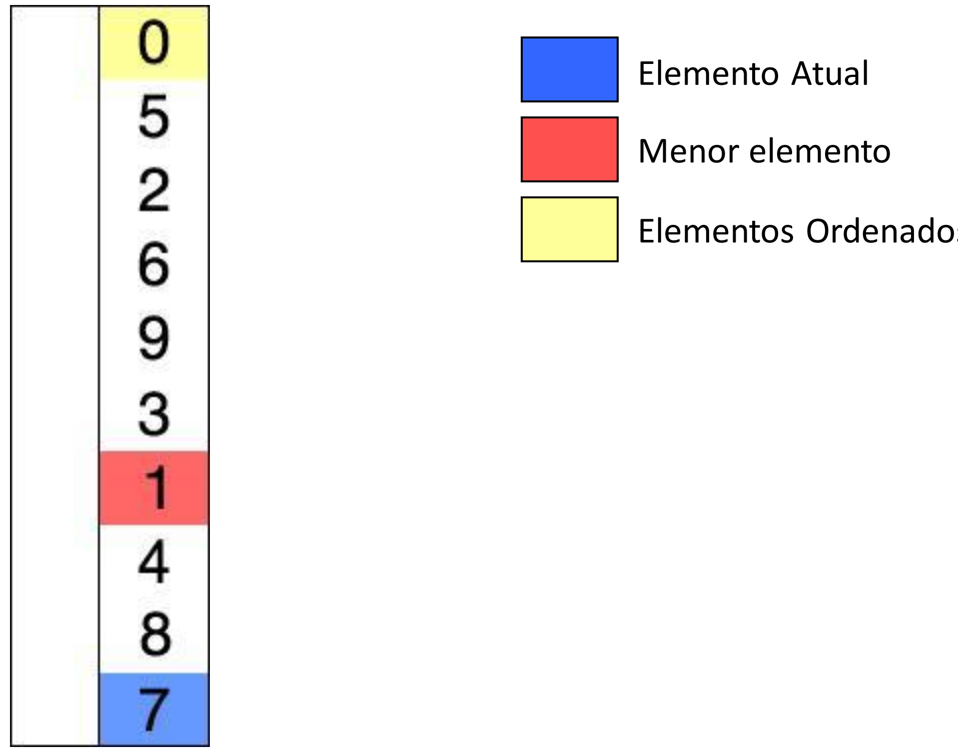
Selection Sort

<u>Idéia</u>



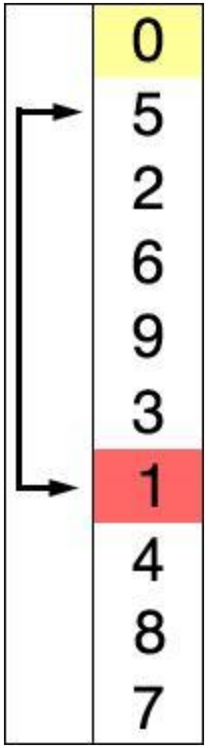
Selection Sort

Idéia



Selection Sort

Idéia



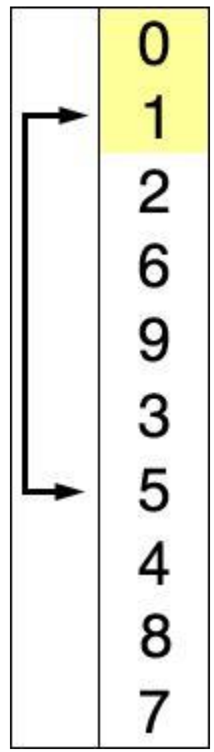
Elemento Atual

Menor elemento

Elementos Ordenados

Selection Sort

Idéia



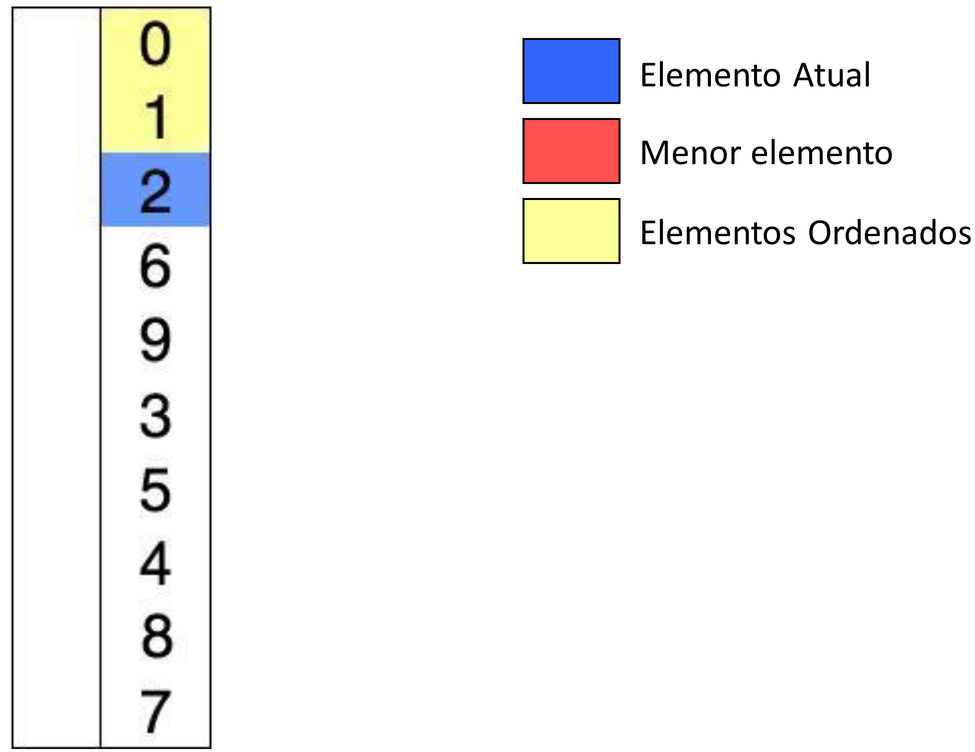
Elemento Atual

Menor elemento

Elementos Ordenados

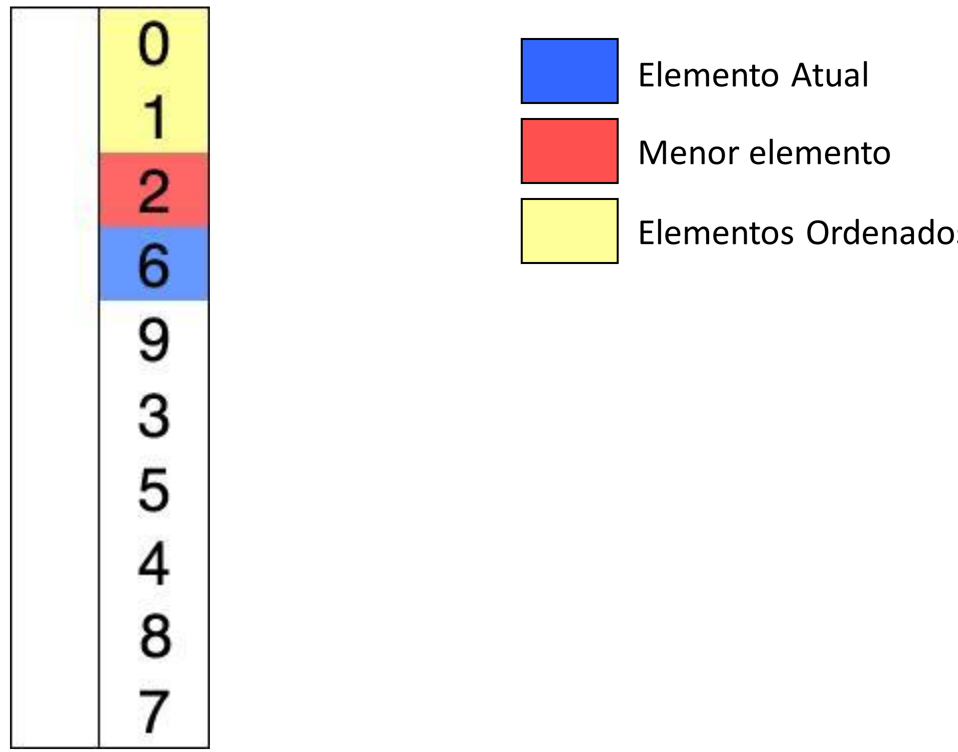
Selection Sort

Idéia



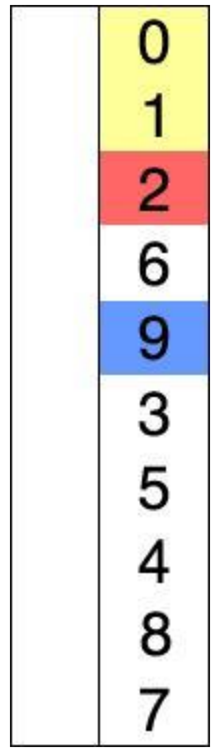
Selection Sort

Idéia



Selection Sort

Idéia



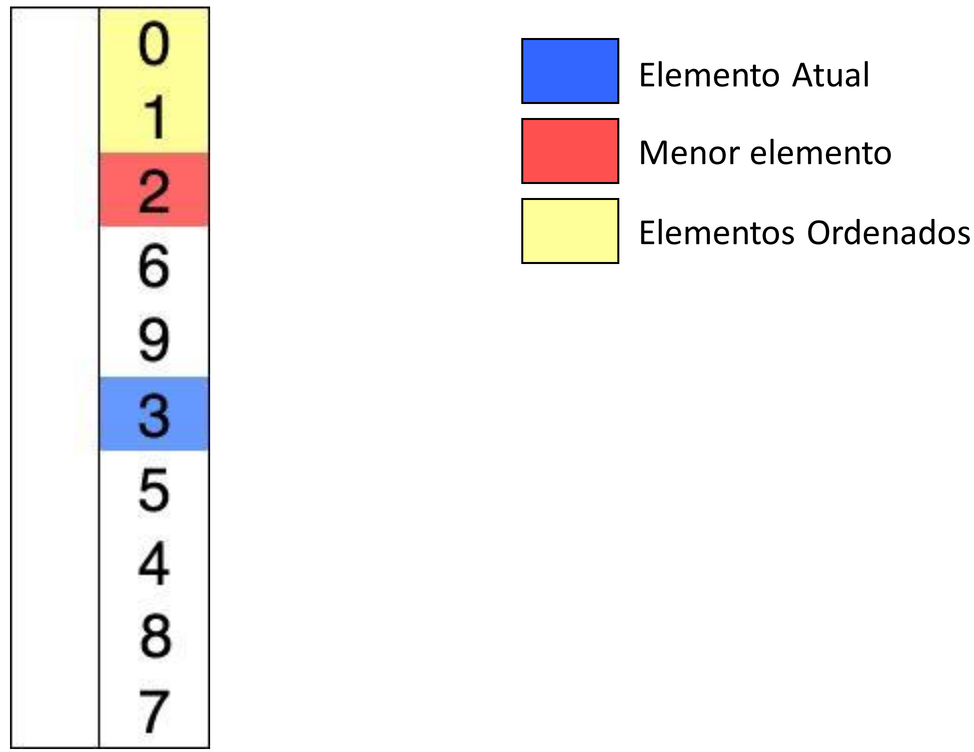
Elemento Atual

Menor elemento

Elementos Ordenados

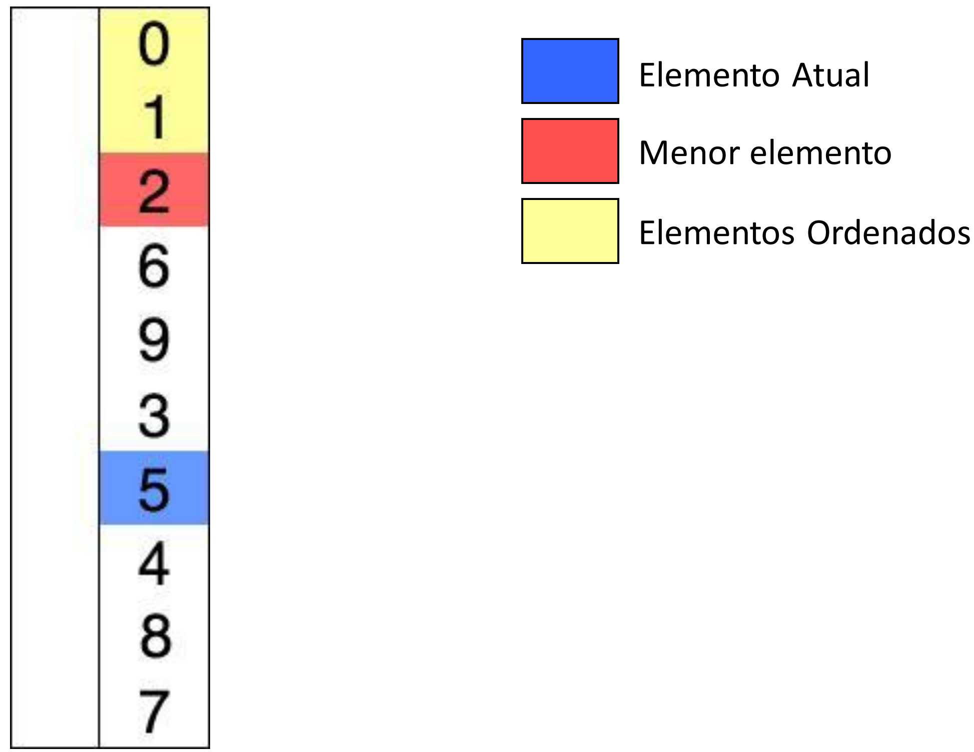
Selection Sort

Idéia



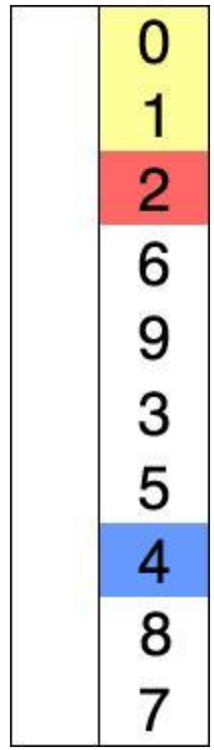
Selection Sort

Idéia



Selection Sort

Idéia



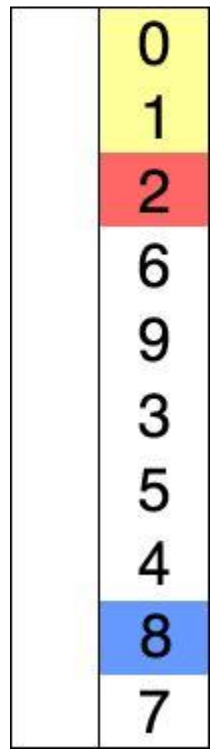
Elemento Atual

Menor elemento

Elementos Ordenados

Selection Sort

Idéia



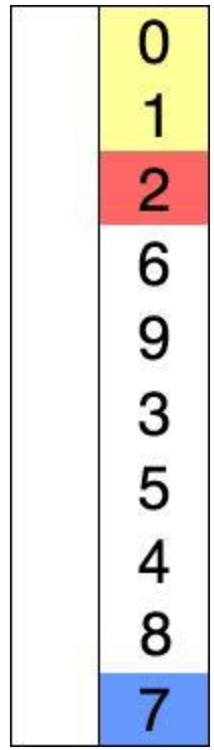
Elemento Atual

Menor elemento

Elementos Ordenados

Selection Sort

Idéia



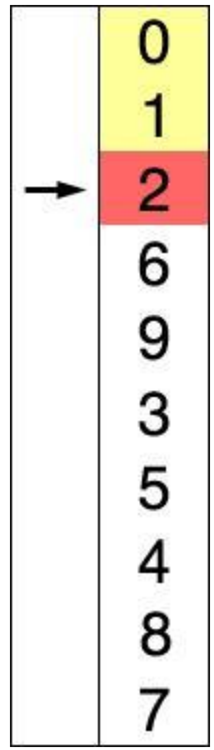
Elemento Atual

Menor elemento

Elementos Ordenados

Selection Sort

Idéia



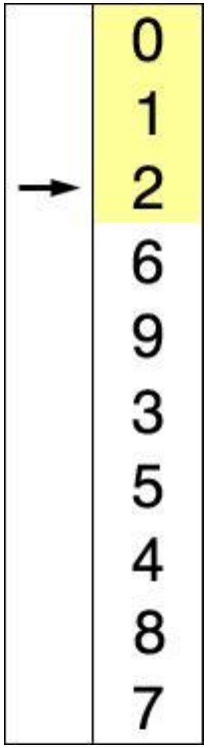
Elemento Atual

Menor elemento

Elementos Ordenados

Selection Sort

Idéia



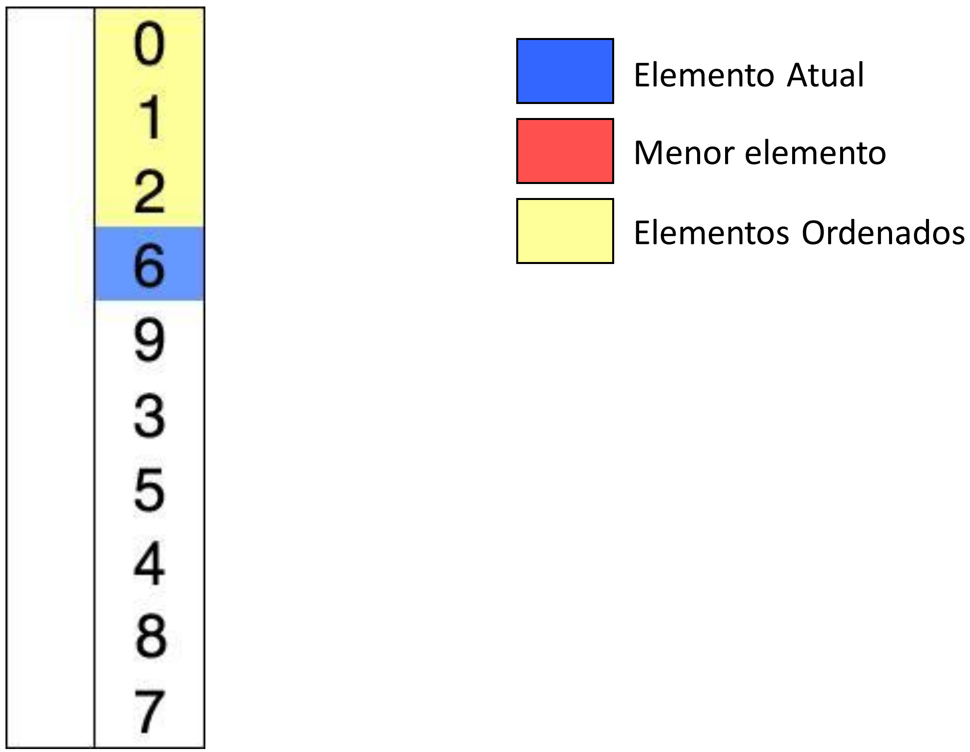
Elemento Atual

Menor elemento

Elementos Ordenados

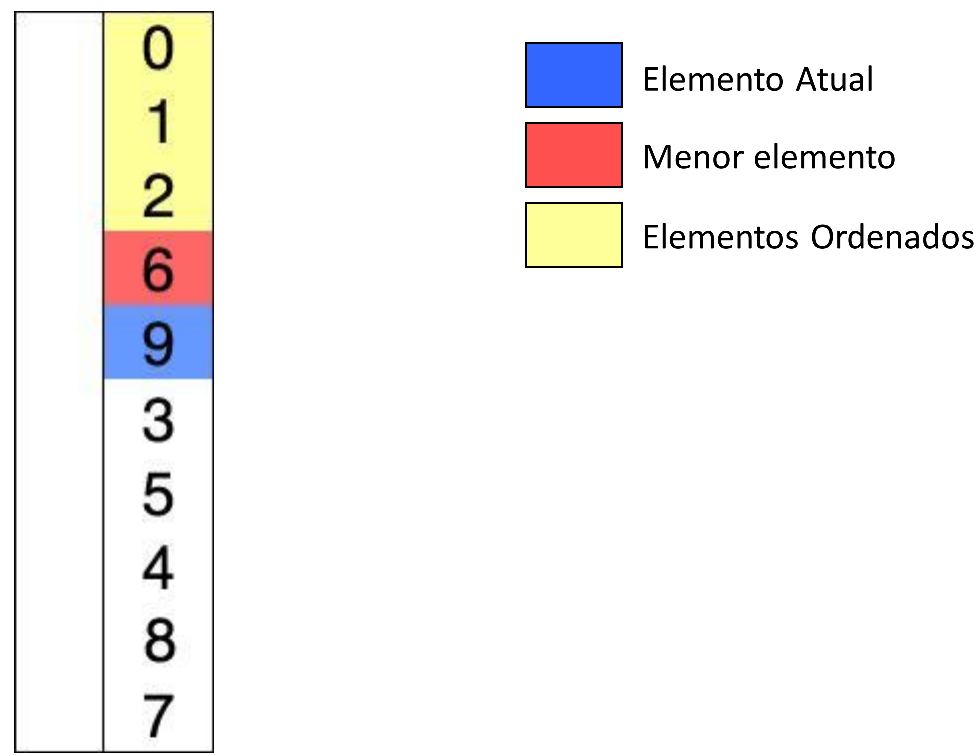
Selection Sort

Idéia



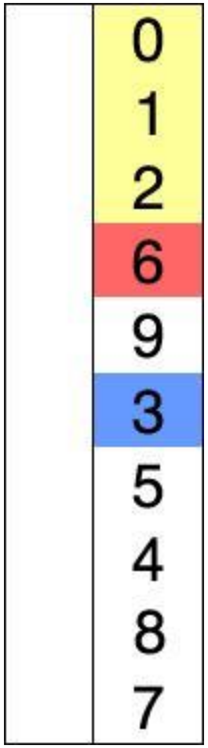
Selection Sort

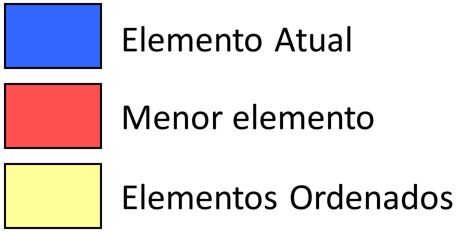
Idéia



Selection Sort

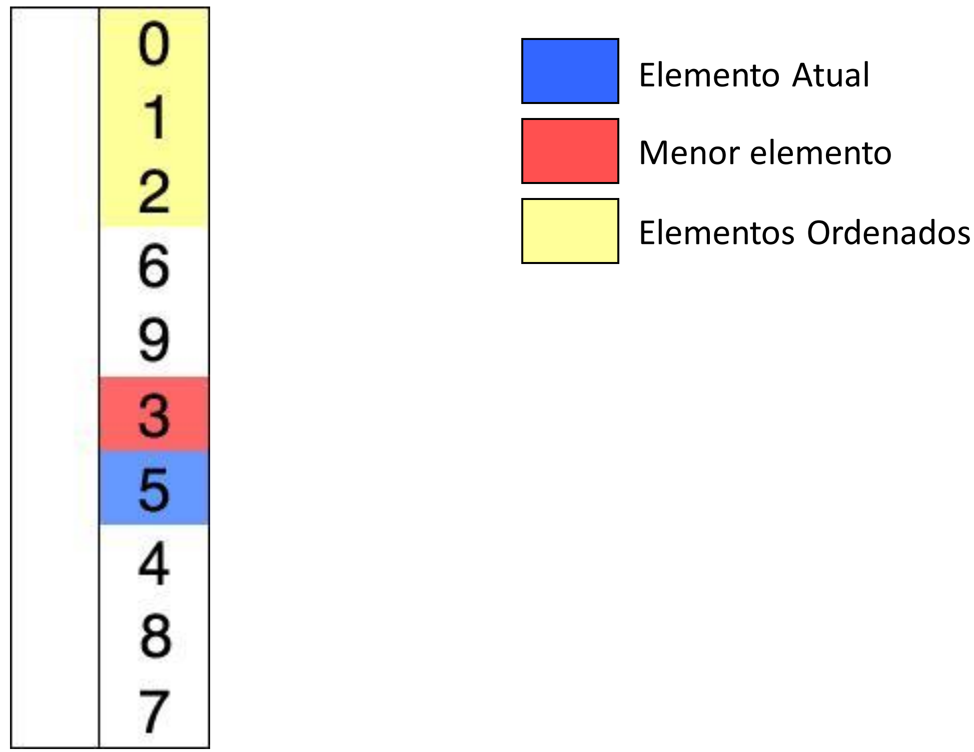
Idéia





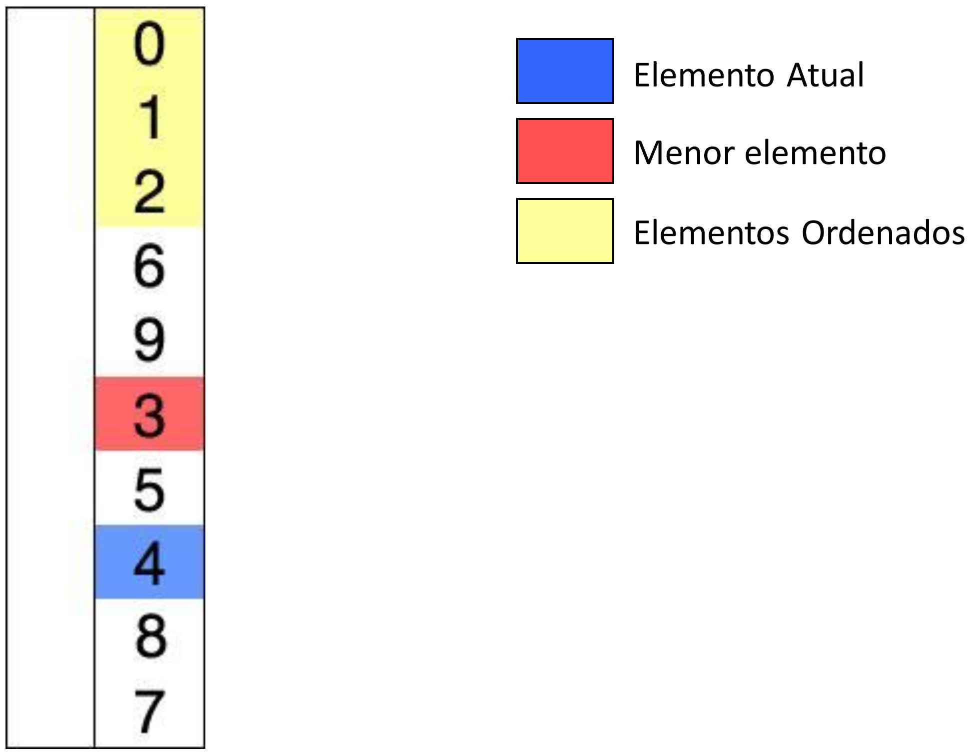
Selection Sort

Idéia



Selection Sort

Idéia

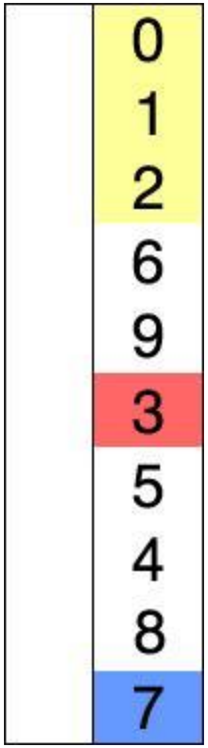


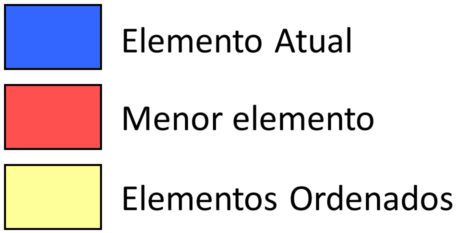
Selection Sort

Idéia

Selection Sort

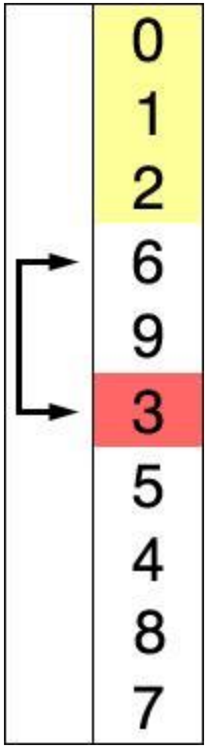
Idéia

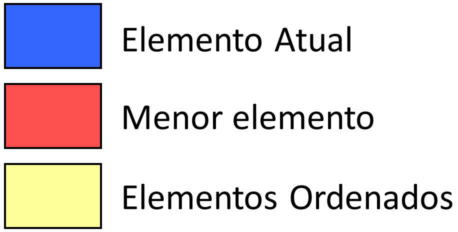




Selection Sort

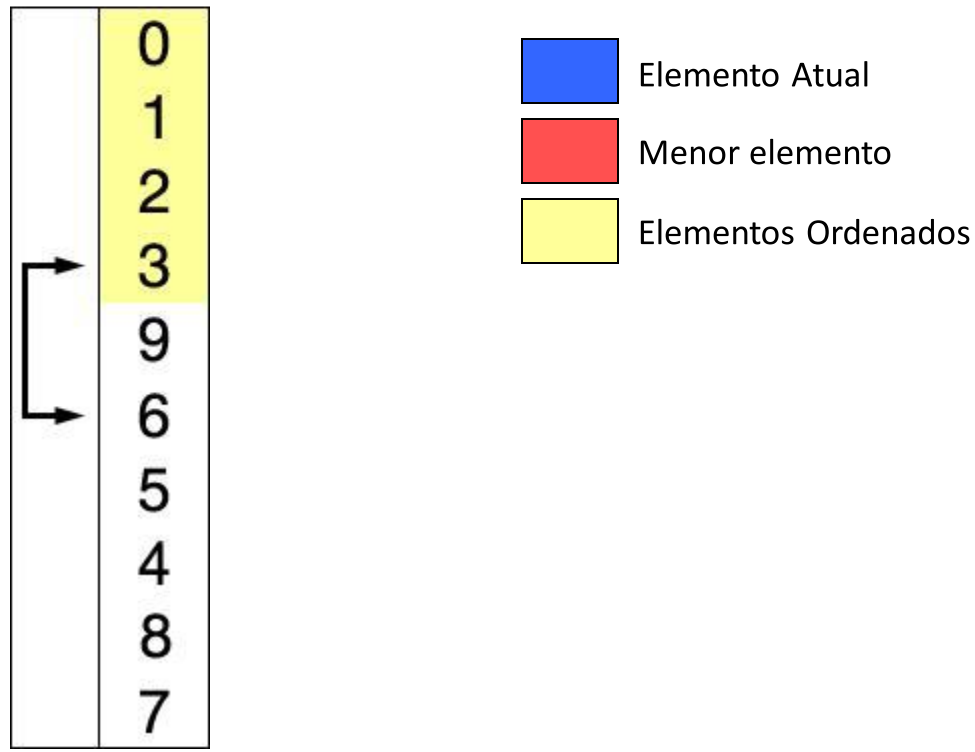
Idéia





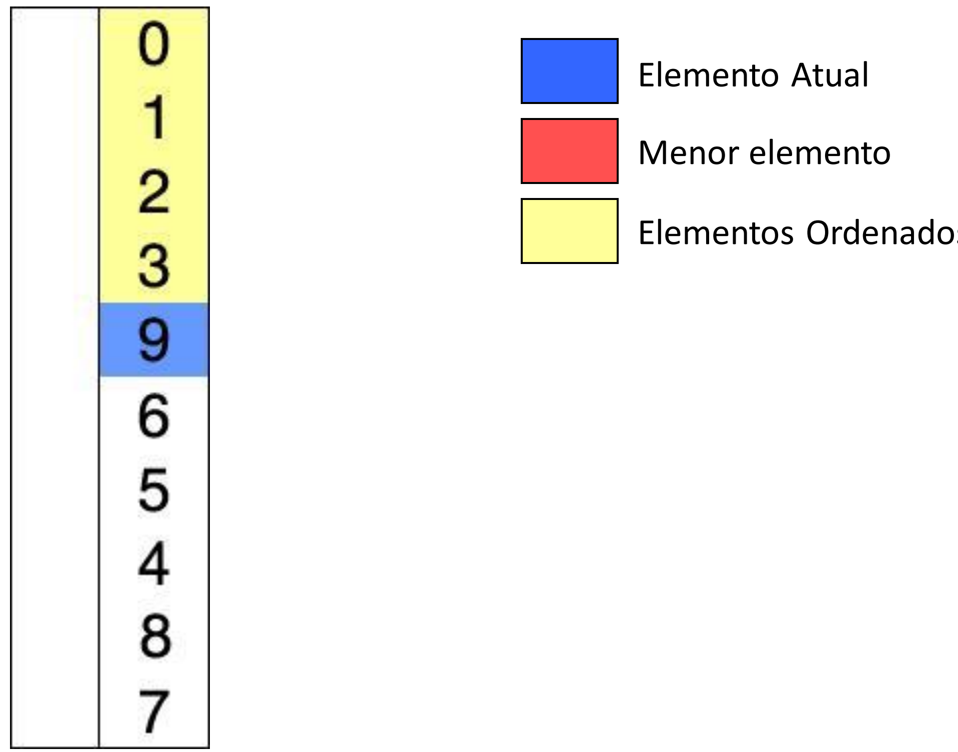
Selection Sort

Idéia



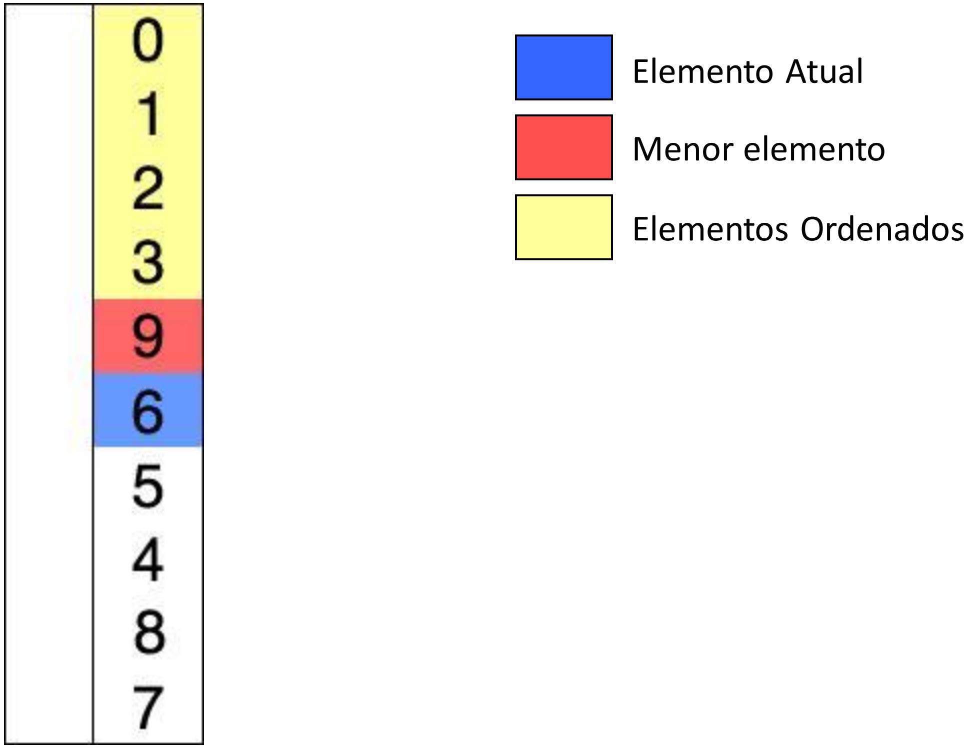
Selection Sort

Idéia



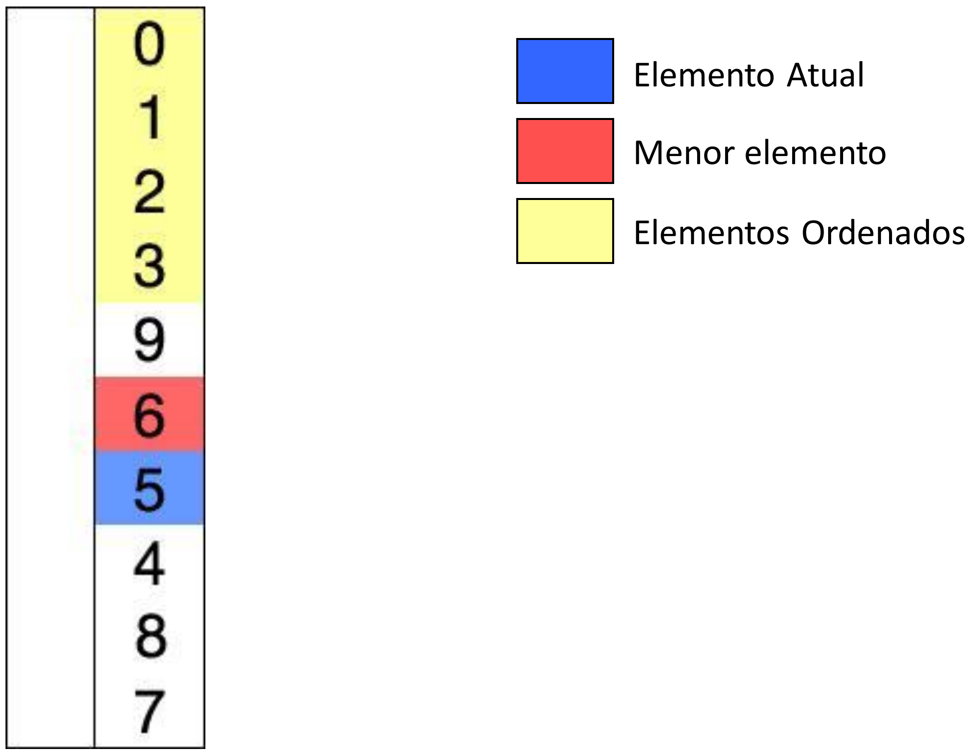
Selection Sort

<u>Idéia</u>



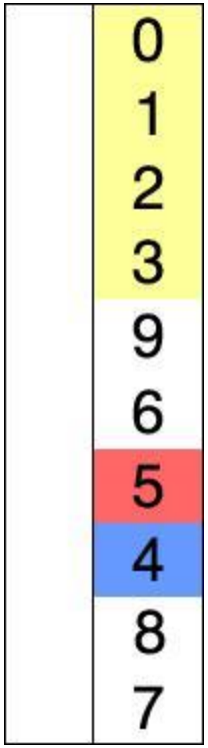
Selection Sort

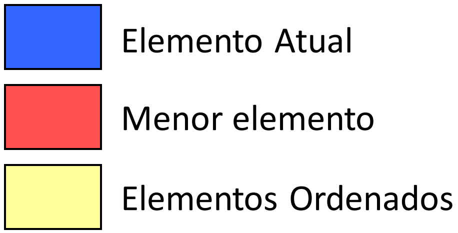
Idéia



Selection Sort

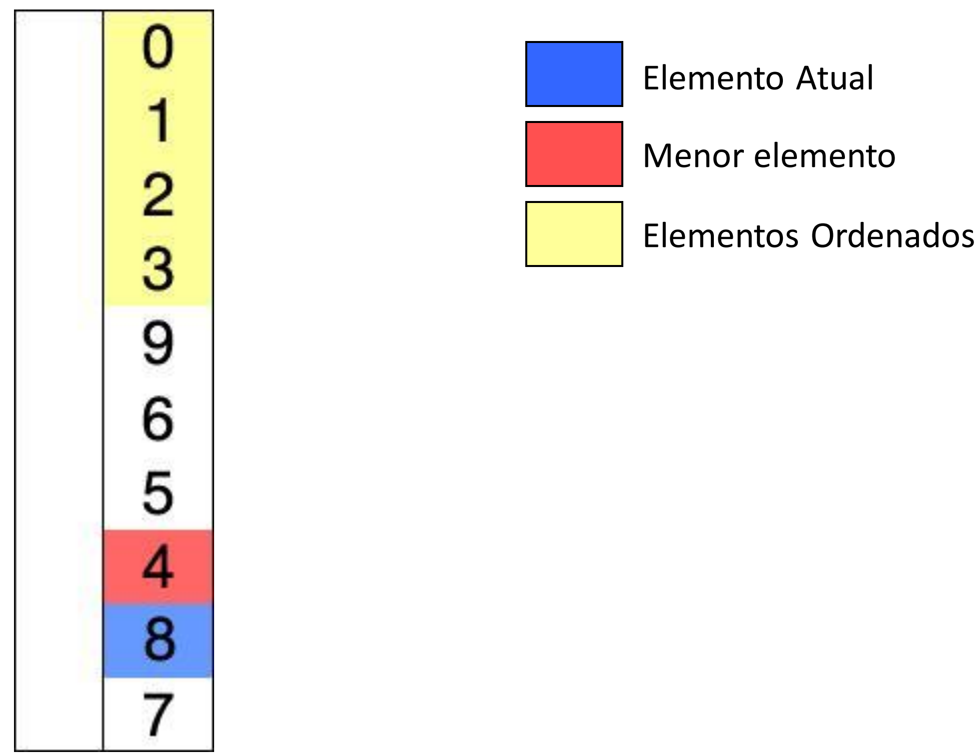
Idéia





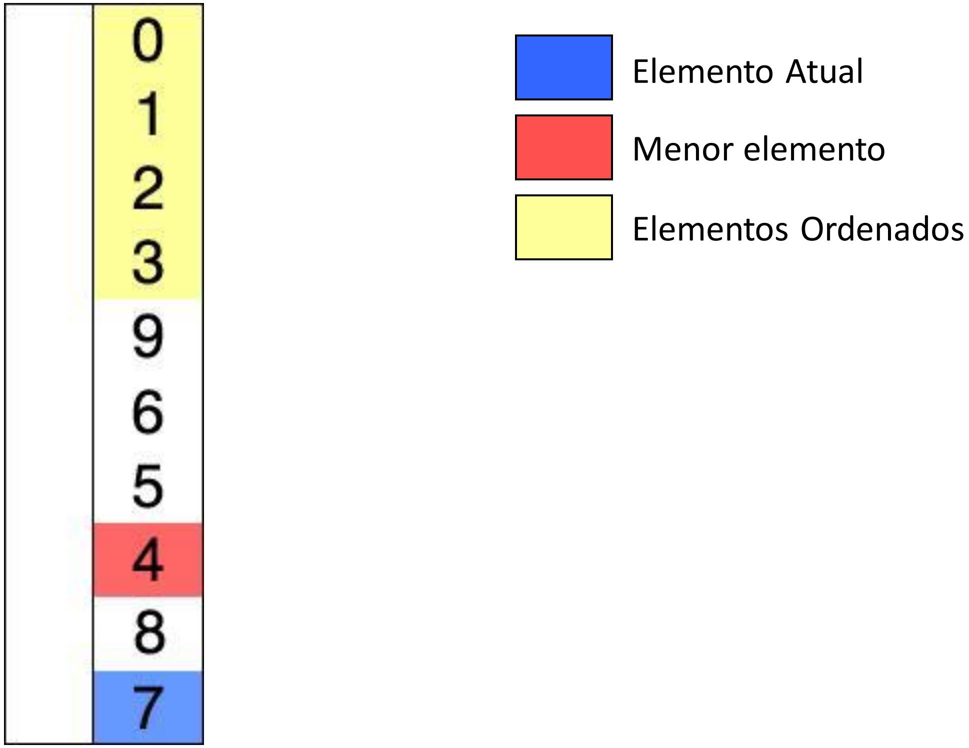
Selection Sort

Idéia



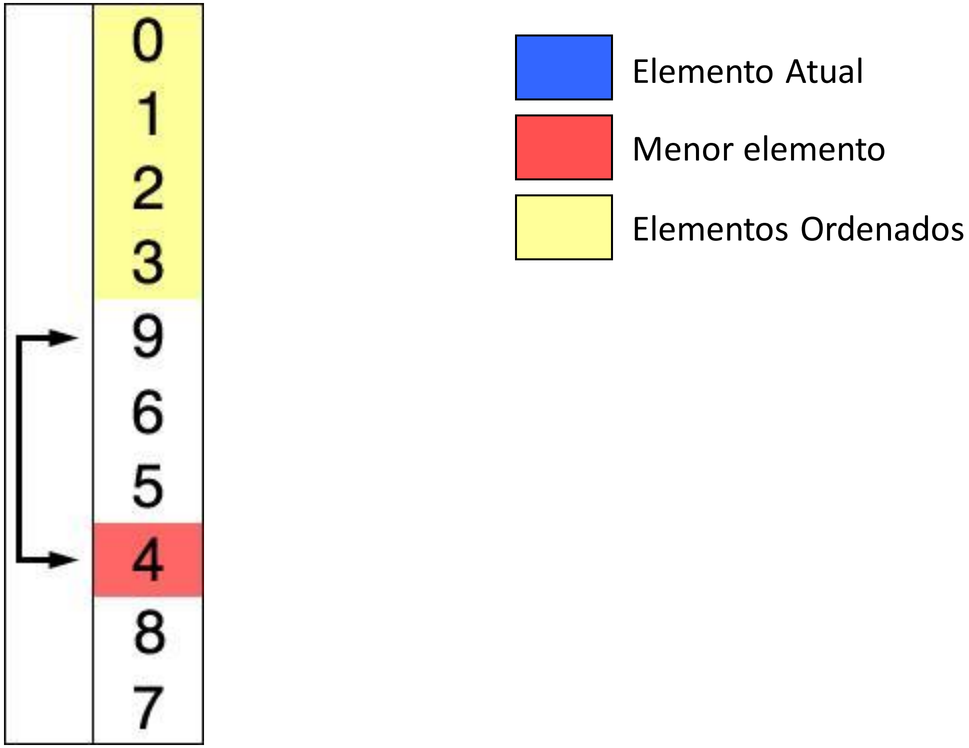
Selection Sort

Idéia



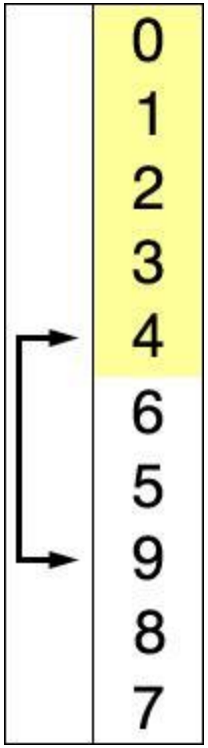
Selection Sort

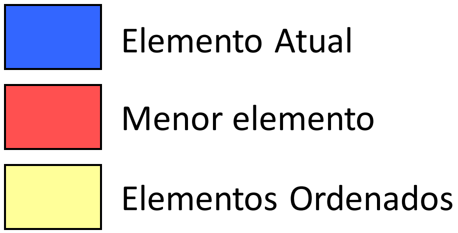
Idéia



Selection Sort

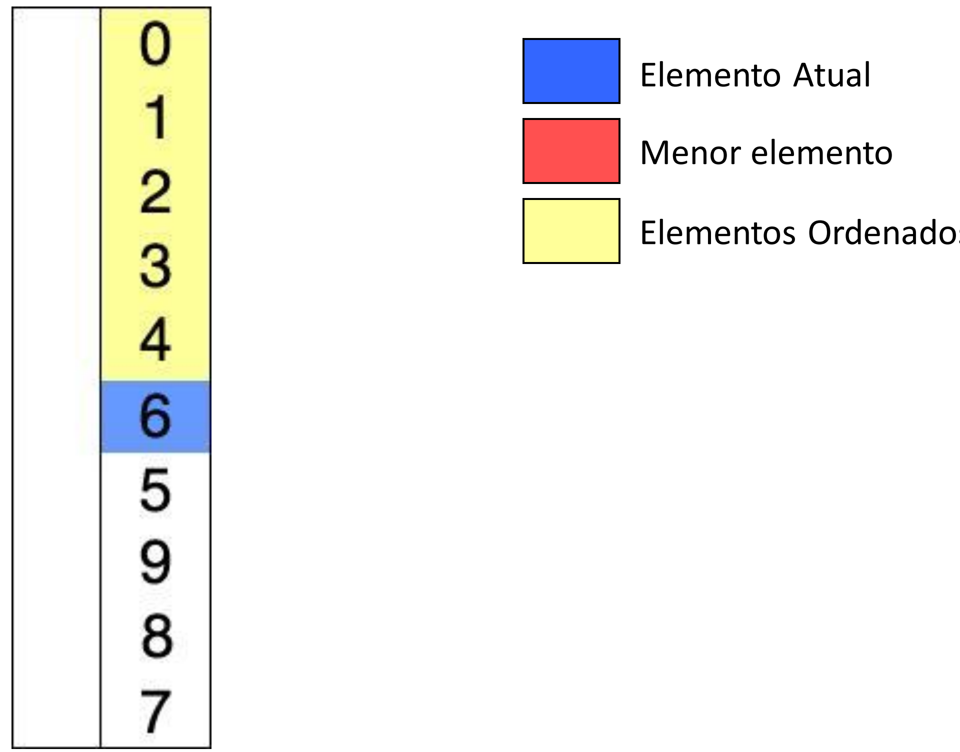
Idéia





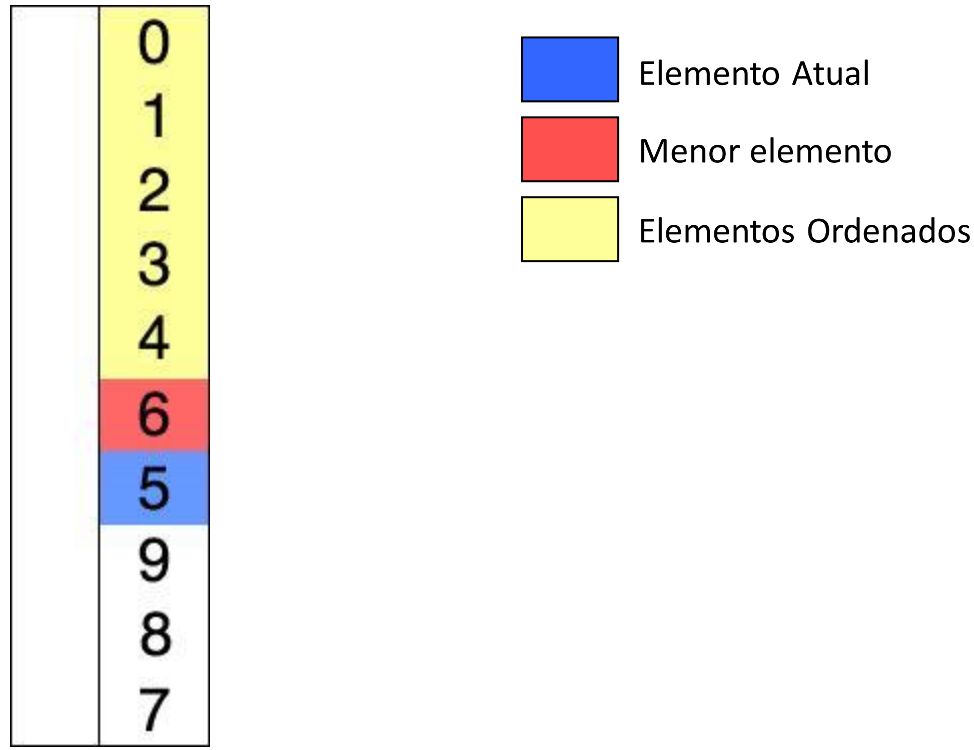
Selection Sort

Idéia



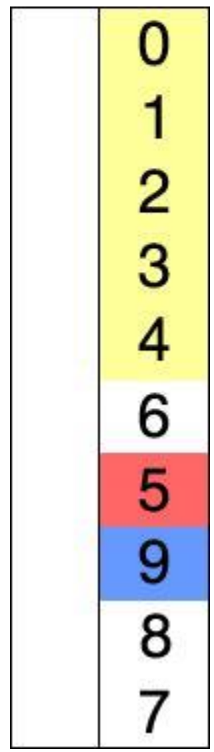
Selection Sort

<u>Idéia</u>



Selection Sort

Idéia



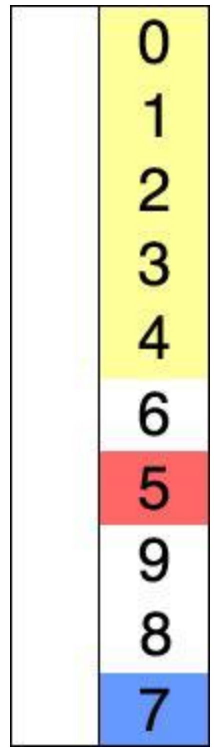
Selection Sort

<u>Idéia</u>

	0
	1
	2
	3
	4
	6
	5
	9
	8
	7

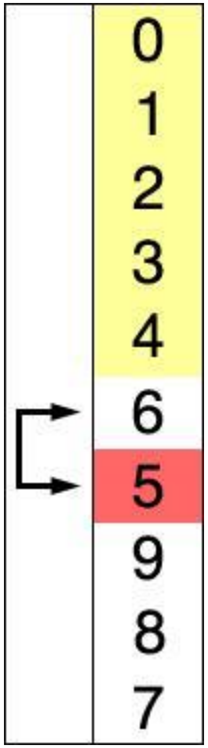
Selection Sort

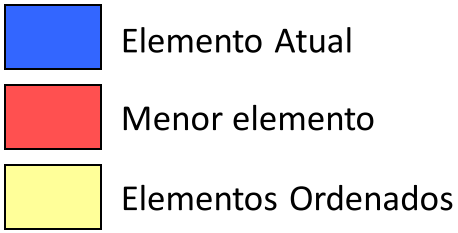
Idéia



Selection Sort

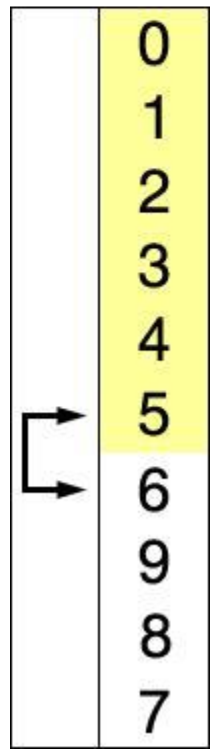
<u>Idéia</u>





Selection Sort

<u>Idéia</u>



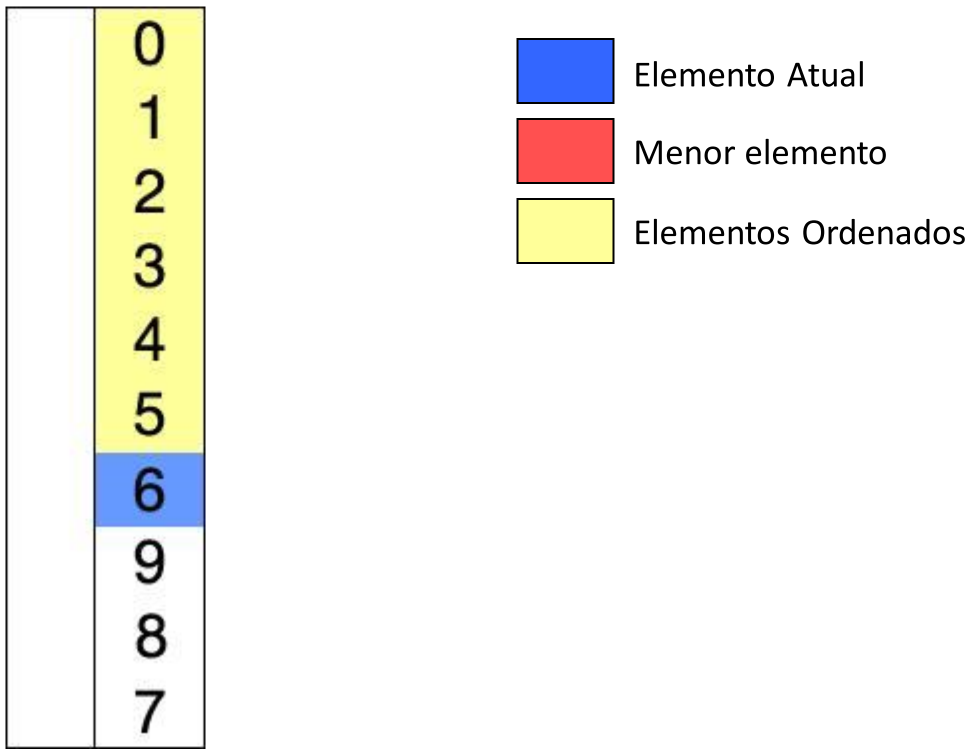
Elemento Atual

Menor elemento

Elementos Ordenados

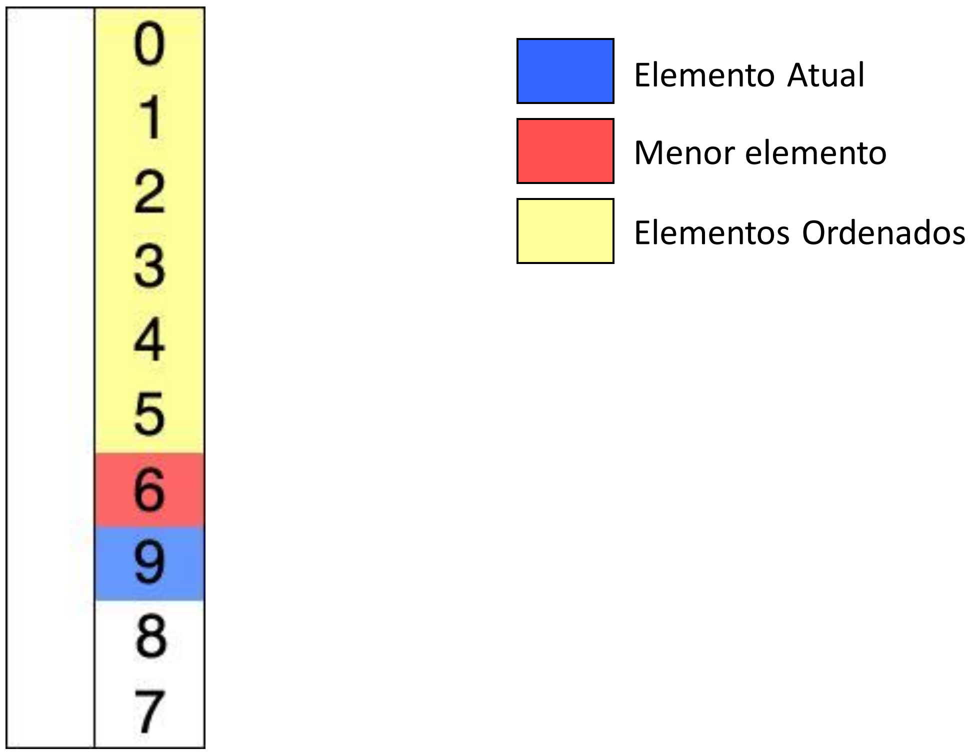
Selection Sort

Idéia



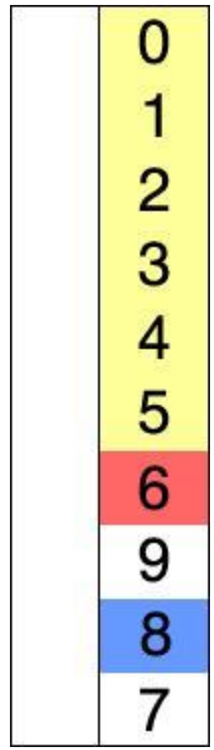
Selection Sort

Idéia



Selection Sort

Idéia



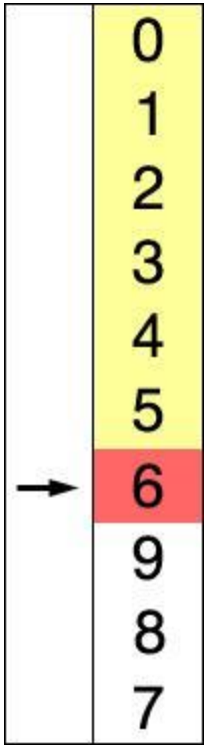
- Elemento Atual
- Menor elemento
- Elementos Ordenados

Selection Sort

Idéia

Selection Sort

Idéia



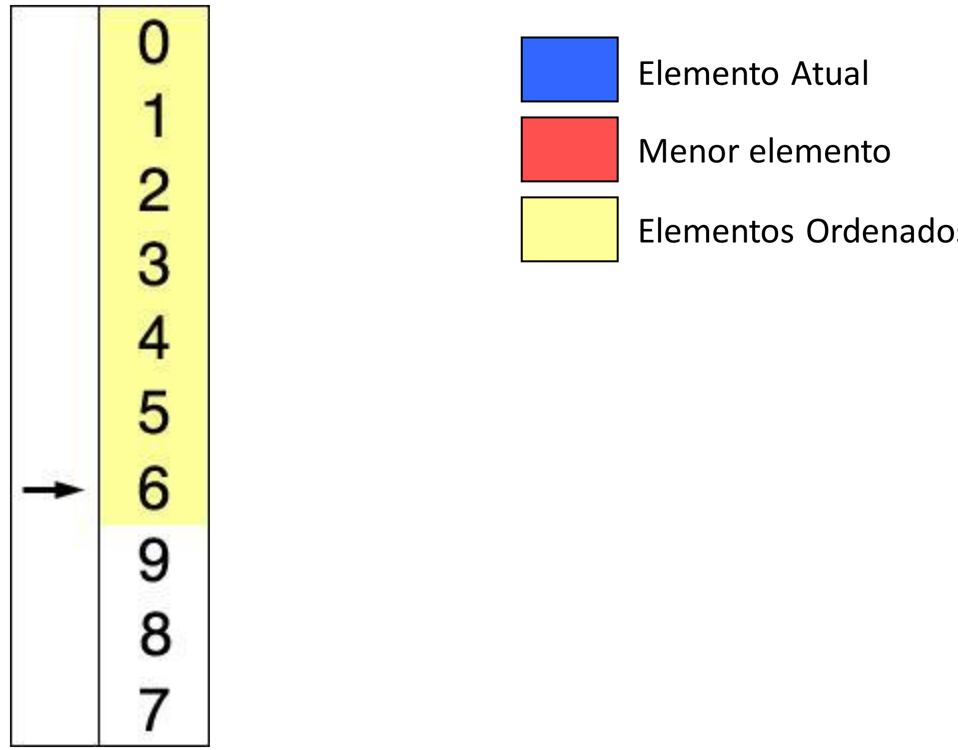
Elemento Atual

Menor elemento

Elementos Ordenados

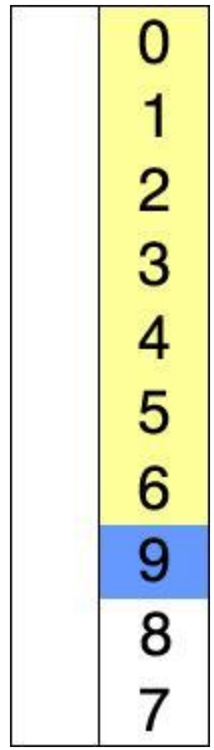
Selection Sort

Idéia



Selection Sort

Idéia



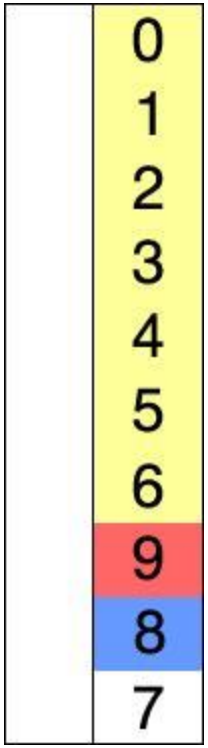
Elemento Atual

Menor elemento

Elementos Ordenados

Selection Sort

Idéia



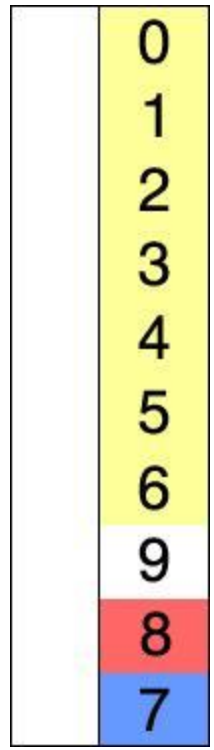
Elemento Atual

Menor elemento

Elementos Ordenados

Selection Sort

Idéia



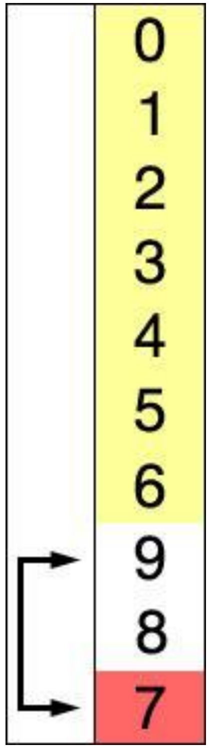
Elemento Atual

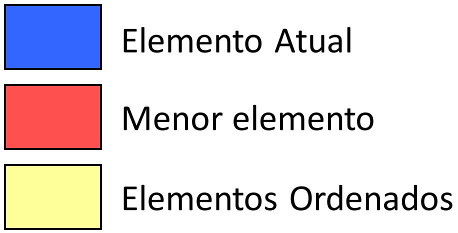
Menor elemento

Elementos Ordenados

Selection Sort

Idéia



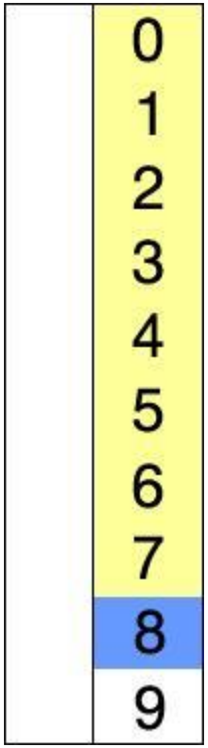


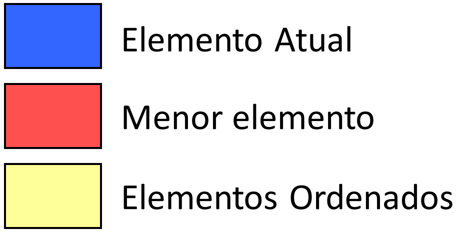
Selection Sort

Idéia

Selection Sort

Idéia



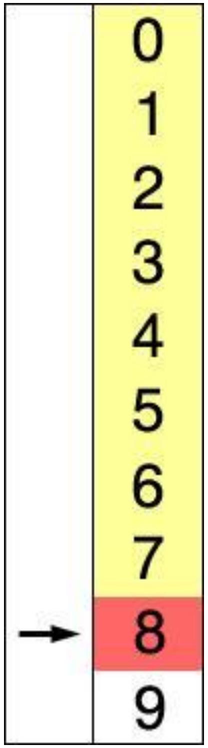


Selection Sort

Idéia

Selection Sort

Idéia



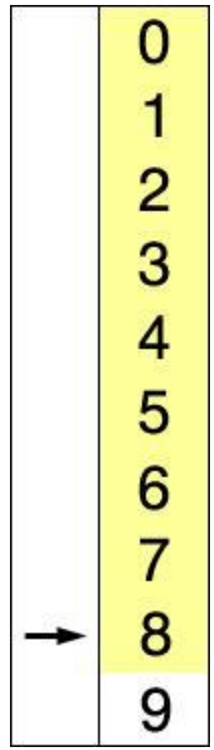
Elemento Atual

Menor elemento

Elementos Ordenados

Selection Sort

Idéia



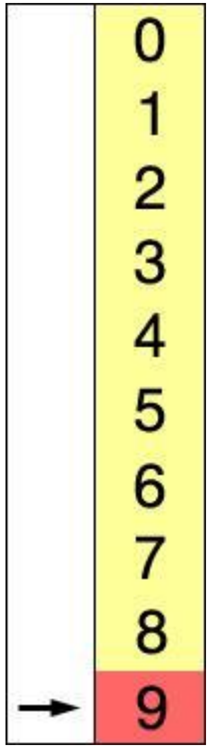
Elemento Atual

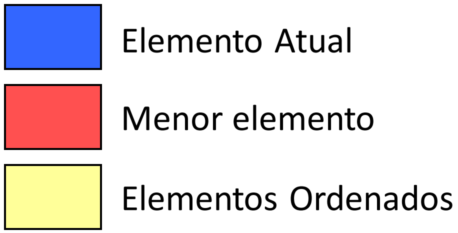
Menor elemento

Elementos Ordenados

Selection Sort

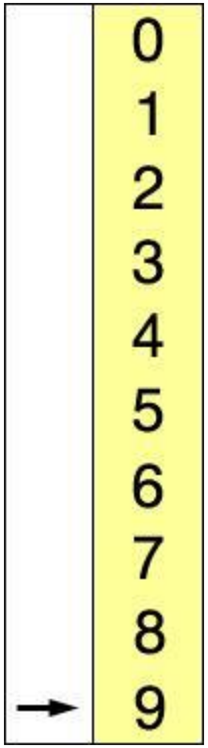
Idéia

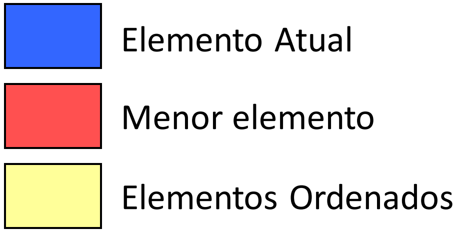




Selection Sort

Idéia





Selection Sort

Idéia

Selection Sort

```
void selectionSort(int *v, int n)
{
   int i, j, aux, iMin;
   for(i=0; i<n-1; i++)
   {
       for(j=i+1, iMin=i; j<n; j++)
       {
             if(v[j] < v[iMin])
             {
                   iMin = j;
             }
       }
       aux = v[i];
       v[i] = v[iMin];
       v[iMin] = aux;
   }
}
```

Eficiência

- Qual o esforço computacional necessário para o Selection Sort ordenar n elementos no pior caso?
  - Qual a função primitiva do selection sort?

```
void selectionSort(int *v, int n)
{
   int i, j, aux, iMin;
   for(i=0; i<n-1; i++)
   {
       for(j=i+1, iMin=i; j<n; j++)
       {
             if(v[j] < v[iMin])
             {
                   iMin = j;
             }
       }
       aux = v[i];
       v[i] = v[iMin];
       v[iMin] = aux;
   }
}
```

Eficiência

- Quantas vezes ocorre a função primitiva no pior caso?
  - Pense para o exemplo: n=4
    - Total: ?

Eficiência

```
void selectionSort(int *v, int n)
{
   int i, j, aux, iMin;
   for(i=0; i<n-1; i++)
   {
       for(j=i+1, iMin=i; j<n; j++)
       {
             if(v[j] < v[iMin])
             {
                   iMin = j;
             }
       }
       aux = v[i];
       v[i] = v[iMin];
       v[iMin] = aux;
   }
}
```

Eficiência

- Quantas vezes ocorre a função primitiva no pior caso?
  - Pense para o exemplo: n=4
    - Total: 6 x
  - Agora generalize para um vetor de tamanho n.

Eficiência

```c
void selectionSort(int *v, int n)
{
   int i, j, aux, iMin;
   for(i=0; i<n-1; i++)
   {
       for(j=i+1, iMin=i; j<n; j++)
       {
             if(v[j] < v[iMin])
             {
                   iMin = j;
             }
       }
       aux = v[i];
       v[i] = v[iMin];
       v[iMin] = aux;
   }
}
```

Eficiência

- Quantas vezes ocorre a função primitiva no pior caso?
  - Pense para o exemplo: n=4
    - Total: 6 x
  - Agora generalize para um vetor de tamanho n.

$$T(n) = 1 + 2 + 3 + \cdots + (n-3) + (n-2) + (n-1)$$

Eficiência

- Quantas vezes ocorre a função primitiva no pior caso?
  - Pense para o exemplo: n=4
    - Total: 6 x
  - Agora generalize para um vetor de tamanho n.

$$T(n) = 1 + 2 + 3 + \cdots + (n-3) + (n-2) + (n-1)$$

$$T(n) = \frac{n^2 - n}{2}$$

Eficiência

- Quantas vezes ocorre a função primitiva no pior caso?
  - Pense para o exemplo: n=4
    - Total: 6 x
  - Agora generalize para um vetor de tamanho n.

$$T(n) = 1 + 2 + 3 + \cdots + (n-3) + (n-2) + (n-1)$$
$$T(n) = \frac{n^2 - n}{2}$$

  - No entanto, o Selection Sort tende a consumir sensivelmente menos tempo que o Bubble Sort. Por que?

Eficiência

- Quantas vezes ocorre a função primitiva no pior caso?
  - Pense para o exemplo: n=4
    - Total: 6 x
  - Agora generalize para um vetor de tamanho n.

$$T(n) = 1 + 2 + 3 + \cdots + (n-3) + (n-2) + (n-1)$$

$$T(n) = \frac{n^2 - n}{2}$$

  - No entanto, o Selection Sort tende a consumir sensivelmente menos tempo que o Bubble Sort. Por que?
  - Perceba que no pior caso o Bubble Sort faz exatamente $(n^2-n)/2$ comparações seguidas da mesma quantidade de trocas.
  - Uma troca significa 3 atribuições envolvendo uma variável indexada.

Eficiência

- Deste modo, pode-se afirmar que no pior o Bubble Sort faz
  - $(n^2 - n)/2$ comparações

Eficiência

- Deste modo, pode-se afirmar que no pior o Bubble Sort faz
  - $(n^2 - n)/2$ comparações
  - $(n^2 - n)/2$ trocas.

Eficiência

- Deste modo, pode-se afirmar que no pior o Bubble Sort faz
  - $(n^2 - n)/2$ comparações
  - $(n^2 - n)/2$ trocas.

```
aux = v[i];
v[i] = v[j];
v[j] = aux;
```

Eficiência

- Deste modo, pode-se afirmar que no pior o Bubble Sort faz
  - $(n^2 - n)/2$ comparações
  - $(n^2 - n)/2$ trocas.

```
aux = v[i];
v[i] = v[j];
v[j] = aux;
```

  - Resolver v[i], consome 3 instruções: base+i * sizeof(tipo).

Eficiência

- Deste modo, pode-se afirmar que no pior o Bubble Sort faz
  - $(n^2 - n)/2$ comparações
  - $(n^2 - n)/2$ trocas.

```
aux = v[i];
v[i] = v[j];
v[j] = aux;
```

  - Resolver v[i], consome 3 instruções: base+i * sizeof(tipo).
  - Logo uma troca consome: 12 instruções para resolver o índice do arranjo mais 3 atribuições. Totalizando: 15 instruções.

Eficiência

- Deste modo, pode-se afirmar que no pior o Bubble Sort faz
  - $(n^2 - n)/2$ comparações
  - $(n^2 - n)/2$ trocas.

```
aux = v[i];
v[i] = v[j];
v[j] = aux;
```

  - Resolver v[i], consome 3 instruções: base+i * sizeof(tipo).
  - Logo uma troca consome: 12 instruções para resolver o índice do arranjo mais 3 atribuições. Totalizando: 15 instruções.
  - Assim, o Bubble consome: $(15n^2-15n)/2$ instruções para efetuar as trocas.

Tamanho	Instruções para troca
4	90

Eficiência

- Deste modo, pode-se afirmar que no pior o Bubble Sort faz
  - $(n^2 - n)/2$ comparações
  - $(n^2 - n)/2$ trocas.

```
aux = v[i];
v[i] = v[j];
v[j] = aux;
```

  - Resolver v[i], consome 3 instruções: base+i * sizeof(tipo).
  - Logo uma troca consome: 12 instruções para resolver o índice do arranjo mais 3 atribuições. Totalizando: 15 instruções.
  - Assim, o Bubble consome: $(15n^2-15n)/2$ instruções para efetuar as trocas.

Tamanho	Instruções para troca
4	90
10	675

Eficiência

- Deste modo, pode-se afirmar que no pior o Bubble Sort faz
  - $(n^2 - n)/2$ comparações
  - $(n^2 - n)/2$ trocas.

```
aux = v[i];
v[i] = v[j];
v[j] = aux;
```

  - Resolver v[i], consome 3 instruções: base+i * sizeof(tipo).
  - Logo uma troca consome: 12 instruções para resolver o índice do arranjo mais 3 atribuições. Totalizando: 15 instruções.
  - Assim, o Bubble consome: $(15n^2-15n)/2$ instruções para efetuar as trocas.

Tamanho	Instruções para troca
4	90
10	675
100	4250

Eficiência

- E quantas instruções consome o Selection Sort para trocas?

Eficiência

- E quantas instruções consome o Selection Sort para trocas?
  - Percebemos que o Selection Sort faz $(n^2-n)/2$ comparações.

Eficiência

- E quantas instruções consome o Selection Sort para trocas?
  - Percebemos que o Selection Sort faz $(n^2-n)/2$ comparações.
  - Mas só faz a troca depois que ter encontrado o menor elemento. Ou seja, fora do laço mais interno (linhas 6-12).

Eficiência

- E quantas instruções consome o Selection Sort para trocas?
  - Percebemos que o Selection Sort faz $(n^2-n)/2$ comparações.
  - Mas só faz a troca depois que ter encontrado o menor elemento. Ou seja, fora do laço mais interno (linhas 6-12).
  - Desta forma, só são realizadas n-1 trocas em todo o procedimento.

Eficiência

- E quantas instruções consome o Selection Sort para trocas?
  - Percebemos que o Selection Sort faz $(n^2-n)/2$ comparações.
  - Mas só faz a troca depois que ter encontrado o menor elemento. Ou seja, fora do laço mais interno (linhas 6-12).
  - Desta forma, só são realizadas n-1 trocas em todo o procedimento.
  - Cada troca são 15 instruções, logo consome 15n-1 instruções.

Eficiência

- E quantas instruções consome o Selection Sort para trocas?
  - Percebemos que o Selection Sort faz $(n^2-n)/2$ comparações.
  - Mas só faz a troca depois que ter encontrado o menor elemento. Ou seja, fora do laço mais interno (linhas 6-12).
  - Desta forma, só são realizadas n-1 trocas em todo o procedimento.
  - Cada troca são 15 instruções, logo consome 15n-1 instruções.

Tamanho	**Instruções para troca**	
	Bubble	Selection
4	90	59

Eficiência

- E quantas instruções consome o Selection Sort para trocas?
  - Percebemos que o Selection Sort faz $(n^2-n)/2$ comparações.
  - Mas só faz a troca depois que ter encontrado o menor elemento. Ou seja, fora do laço mais interno (linhas 6-12).
  - Desta forma, só são realizadas n-1 trocas em todo o procedimento.
  - Cada troca são 15 instruções, logo consome 15n-1 instruções.

Tamanho	Instruções para troca	
	Bubble	Selection
4	90	59
10	675	149

Eficiência

- E quantas instruções consome o Selection Sort para trocas?
  - Percebemos que o Selection Sort faz $(n^2-n)/2$ comparações.
  - Mas só faz a troca depois que ter encontrado o menor elemento. Ou seja, fora do laço mais interno (linhas 6-12).
  - Desta forma, só são realizadas n-1 trocas em todo o procedimento.
  - Cada troca são 15 instruções, logo consome 15n-1 instruções.

Tamanho	Instruções para troca	
	Bubble	Selection
4	90	59
10	675	149
100	4250	1499

Eficiência

- E quantas instruções consome o Selection Sort para trocas?
  - Percebemos que o Selection Sort faz $(n^2-n)/2$ comparações.
  - Mas só faz a troca depois que ter encontrado o menor elemento. Ou seja, fora do laço mais interno (linhas 6-12).
  - Desta forma, só são realizadas n-1 trocas em todo o procedimento.
  - Cada troca são 15 instruções, logo consome 15n-2 instruções.

Tamanho	Instruções para troca	
	Bubble	Selection
4	90	59
10	675	149
100	4250	1499
1000	7492500	14999

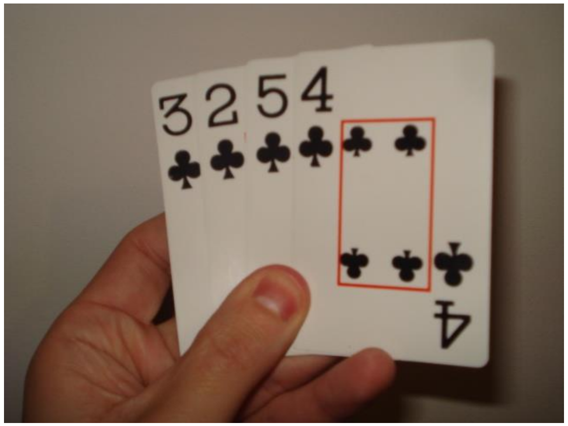
Eficiência

- Por este motivo, o Selection Sort tende a ser sensivelmente mais rápido que o Bubble Sort para entradas suficientemente grandes.

Insertion Sort

- Eficiente para ordenar pequena quantidade de elementos;
- Como o próprio nome sugere, ele insere cada elemento no seu lugar apropriado dentro da seqüência final;
- Percorre um vetor de elementos da esquerda para a direita deixando os elementos mais à esquerda sempre ordenados;
- Funciona da mesma maneira com que muitas pessoas ordenam cartas em um jogo de baralho;
- Prós: Fácil implementação,
- Contra: Ineficiente para entradas suficientemente grandes.

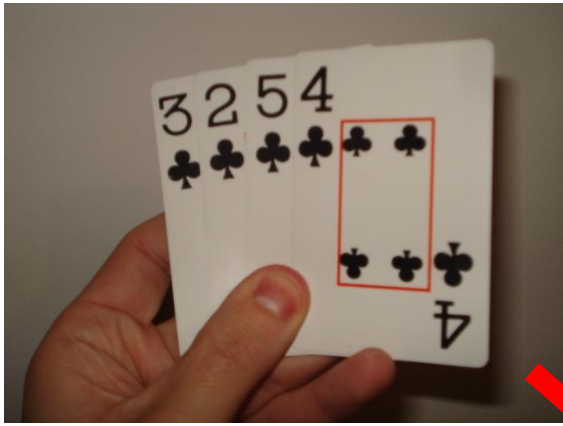
Insertion Sort

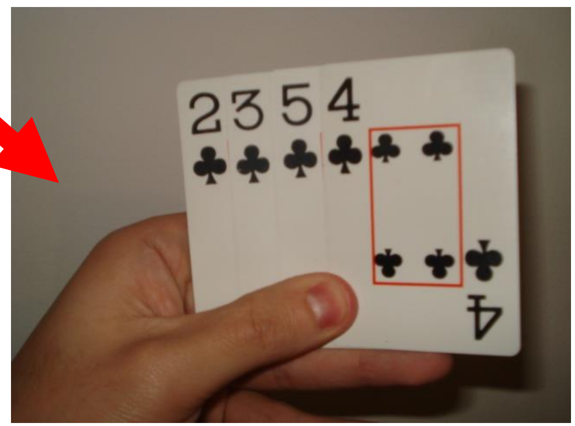


Idéia

Insertion Sort

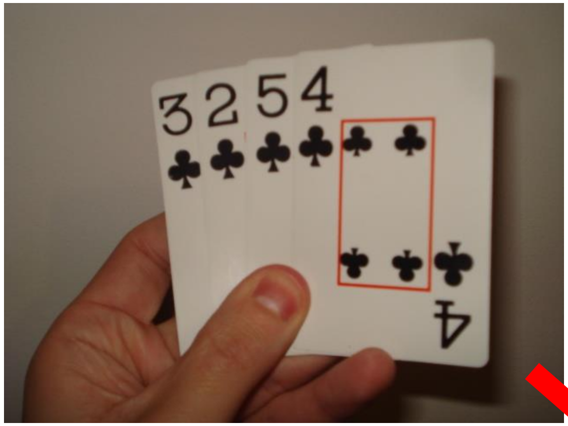
Idéia

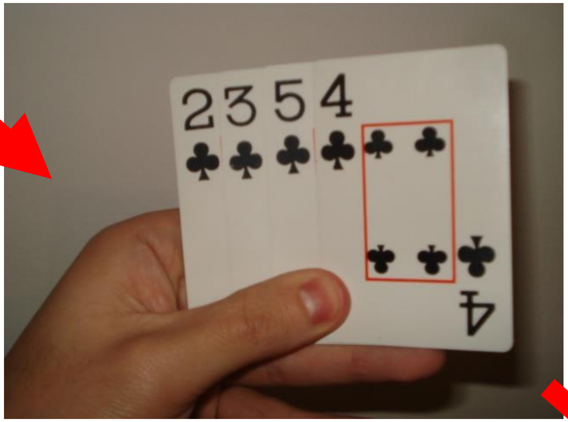


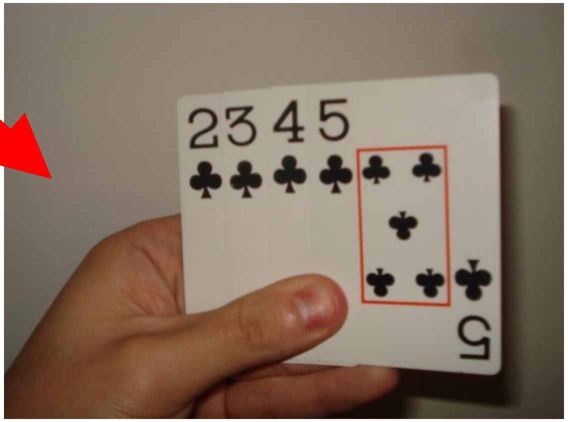


Insertion Sort

Idéia







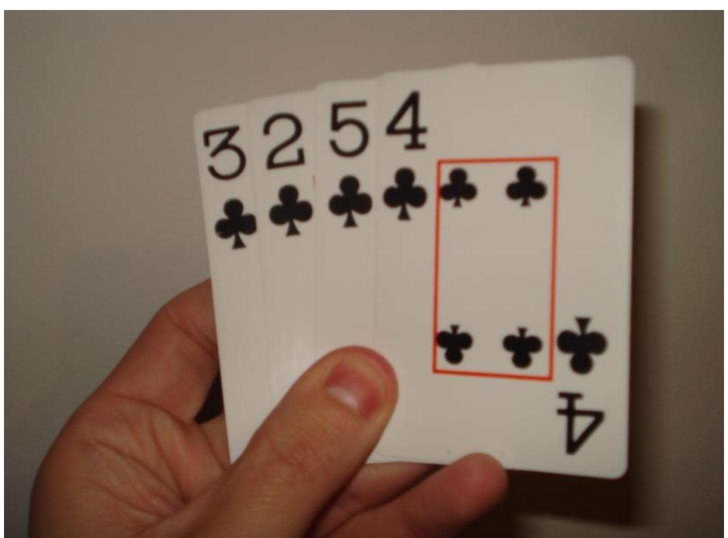
Insertion Sort

Algoritmo

```
void insertion(int *vetor, int n)
{
   int chave, i, j;
   for(i=1; i<n; i++)
   {
     chave = vetor [ i ];
     j = i-1;
     while(j >= 0 && vetor [ j ] >= chave)
     {
        vetor [ j+1 ] = vetor [ j ]
        vetor [ j ] = chave
        j + j - 1
     }
   }
}
```

Insertion Sort

```
void insertion(int *vetor, int n)
{
   int chave, i, j;
   for(i=1; i<n; i++)
   {
     chave = vetor [ i ];
     j = i-1;
     while(j >= 0 && vetor [ j ] >= chave)
     {
        vetor [ j+1 ] = vetor [ j ]
        vetor [ j ] = chave
        j + j - 1
     }
   }
}
```



vetor =

0	1	2	3
3	2	5	4

n = 4

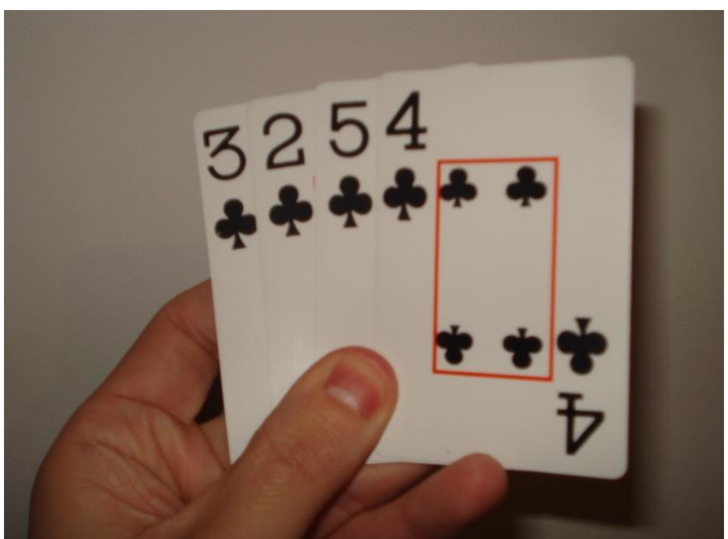
i =

chave =

j =

Insertion Sort

```
void insertion(int *vetor, int n)
{
   int chave, i, j;
   for(i=1; i<n; i++)
   {
     chave = vetor [ i ];
     j = i-1;
     while(j >= 0 && vetor [ j ] >= chave)
     {
        vetor [ j+1 ] = vetor [ j ]
        vetor [ j ] = chave
        j + j - 1
     }
   }
}
```



	0	1	2	3
vetor =	3	2	5	4

n = 4

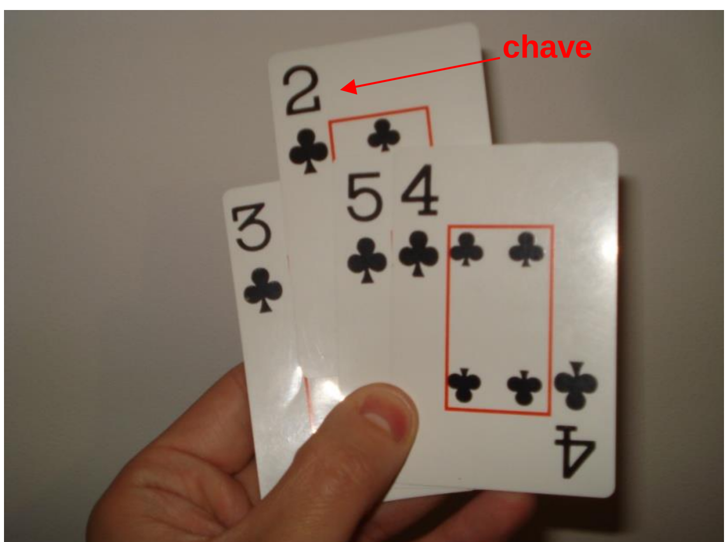
i = 1

chave =

j =

Insertion Sort

```
void insertion(int *vetor, int n)
{
   int chave, i, j;
   for(i=1; i<n; i++)
   {
     chave = vetor [ i ];
     j = i-1;
     while(j >= 0 && vetor [ j ] >= chave)
     {
        vetor [ j+1 ] = vetor [ j ]
        vetor [ j ] = chave
        j + j - 1
     }
   }
}
```



	0	1	2	3
vetor =	3	2	5	4

n = 4

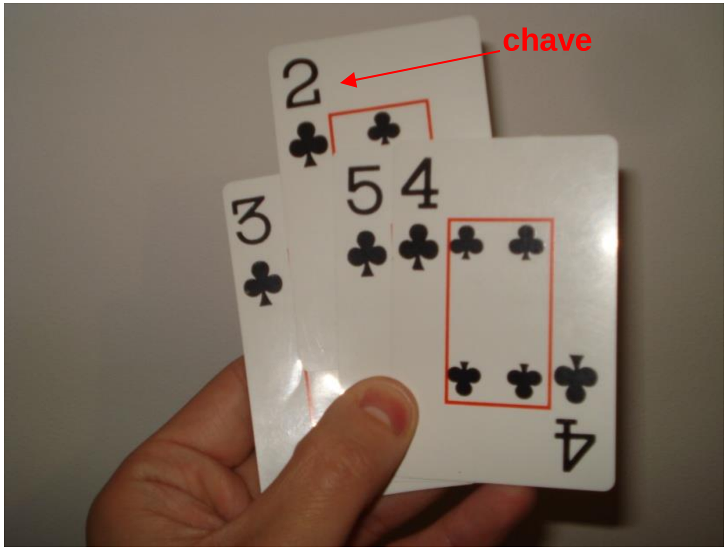
i = 1

chave = 2

j =

Insertion Sort

```
void insertion(int *vetor, int n)
{
   int chave, i, j;
   for(i=1; i<n; i++)
   {
     chave = vetor [ i ];
     j = i-1;
     while(j >= 0 && vetor [ j ] >= chave)
     {
        vetor [ j+1 ] = vetor [ j ]
        vetor [ j ] = chave
        j + j - 1
     }
   }
}
```



0	1	2	3
3	2	5	4

vetor =

n = 4

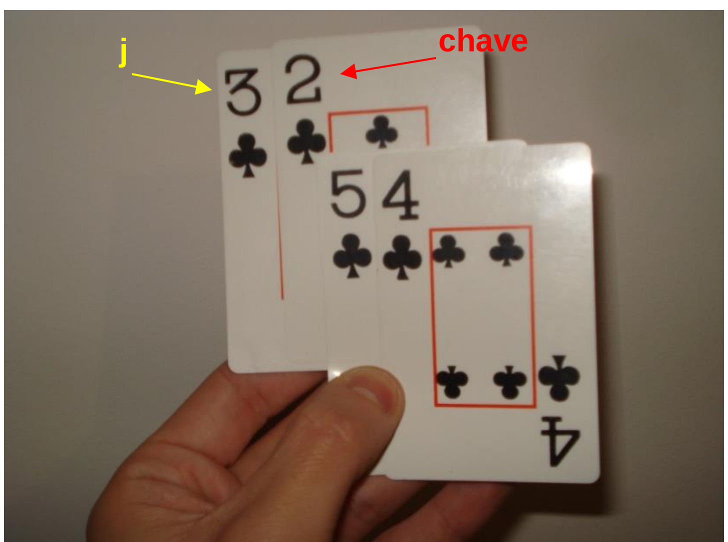
i = 1

chave = 2

j = 0

Insertion Sort

```
void insertion(int *vetor, int n)
{
   int chave, i, j;
   for(i=1; i<n; i++)
   {
     chave = vetor [ i ];
     j = i-1;
     while(j >= 0 && vetor [ j ] >= chave)
     {
        vetor [ j+1 ] = vetor [ j ]
        vetor [ j ] = chave
        j + j - 1
     }
   }
}
```



	0	1	2	3
vetor =	3	2	5	4

n = 4

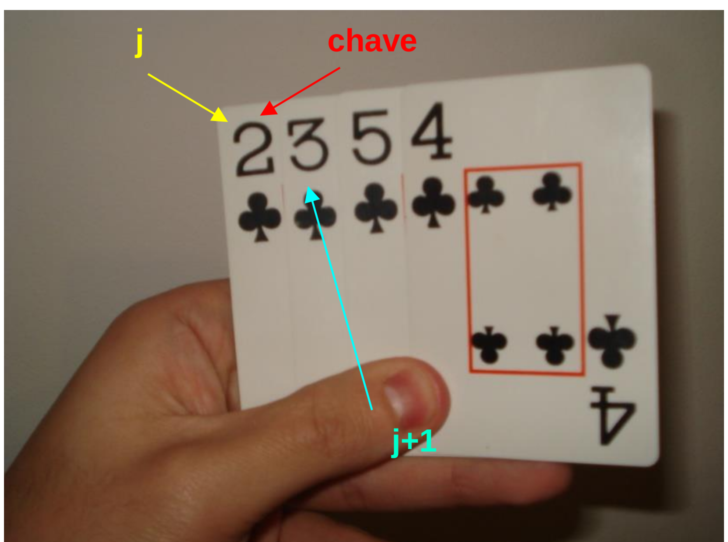
i = 1

chave = 2

j = 0

Insertion Sort

```
void insertion(int *vetor, int n)
{
   int chave, i, j;
   for(i=1; i<n; i++)
   {
     chave = vetor [ i ];
     j = i-1;
     while(j >= 0 && vetor [ j ] >= chave)
     {
        vetor [ j+1 ] = vetor [ j ]
        vetor [ j ] = chave
        j + j - 1
     }
   }
}
```



j+1

	0	1	2	3
vetor =	3	3	5	4

n = 4

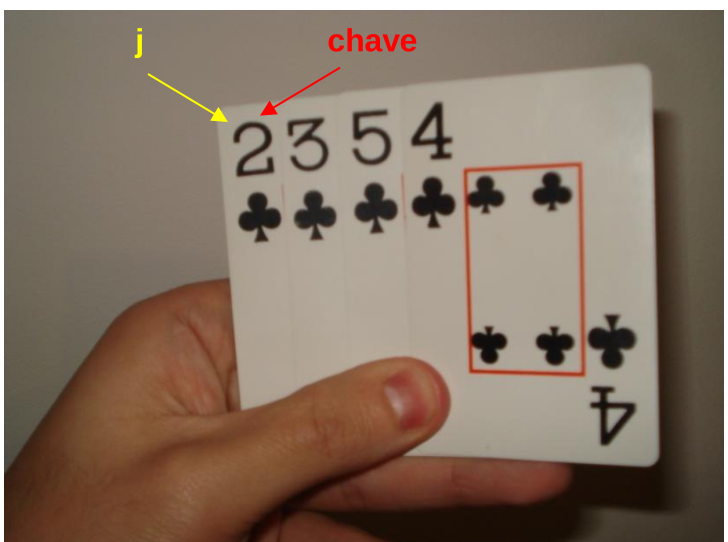
i = 1

chave = 2

j = 0

Insertion Sort

```
void insertion(int *vetor, int n)
{
   int chave, i, j;
   for(i=1; i<n; i++)
   {
     chave = vetor [ i ];
     j = i-1;
     while(j >= 0 && vetor [ j ] >= chave)
     {
        vetor [ j+1 ] = vetor [ j ]
        vetor [ j ] = chave
        j + j - 1
     }
   }
}
```



	0	1	2	3
vetor =	2	3	5	4

n = 4

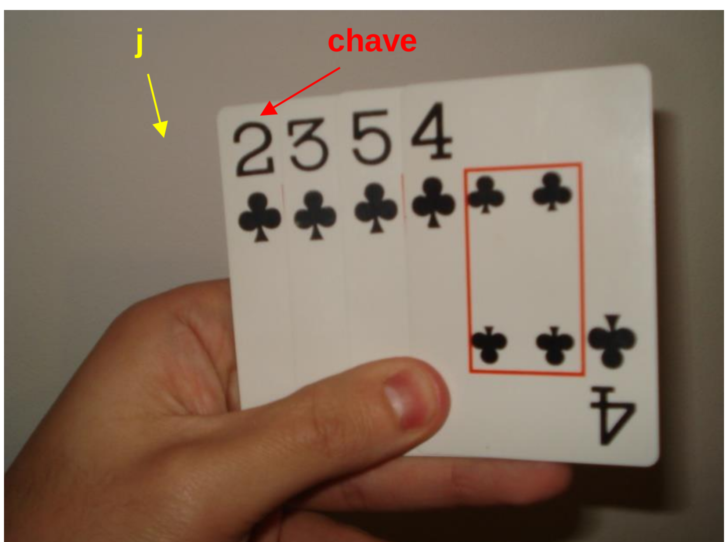
i = 1

chave = 2

j = 0

Insertion Sort

```
void insertion(int *vetor, int n)
{
   int chave, i, j;
   for(i=1; i<n; i++)
   {
     chave = vetor [ i ];
     j = i-1;
     while(j >= 0 && vetor [ j ] >= chave)
     {
        vetor [ j+1 ] = vetor [ j ]
        vetor [ j ] = chave
        j + j - 1
     }
   }
}
```



	0	1	2	3
vetor =	2	3	5	4

n = 4

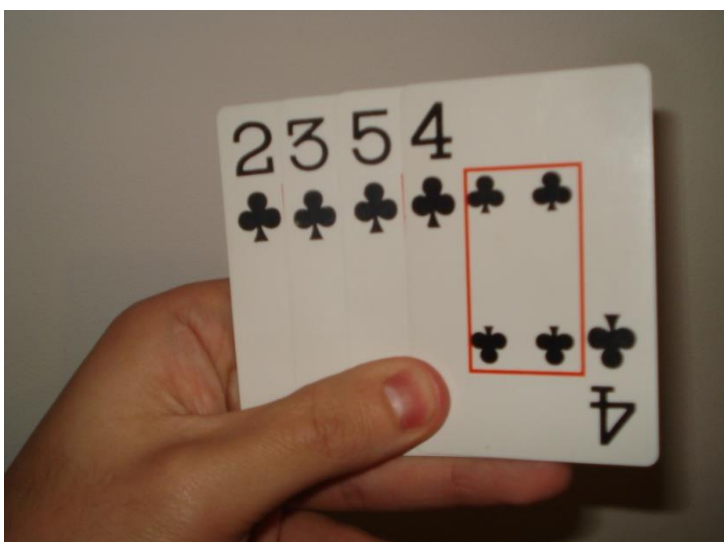
i = 1

chave = 2

$j = \cancel{0}\ -1$

Insertion Sort

```
void insertion(int *vetor, int n)
{
   int chave, i, j;
   for(i=1; i<n; i++)
   {
     chave = vetor [ i ];
     j = i-1;
     while(j >= 0 && vetor [ j ] >= chave)
     {
        vetor [ j+1 ] = vetor [ j ]
        vetor [ j ] = chave
        j + j - 1
     }
   }
}
```



vetor =

0	1	2	3
2	3	5	4

n = 4

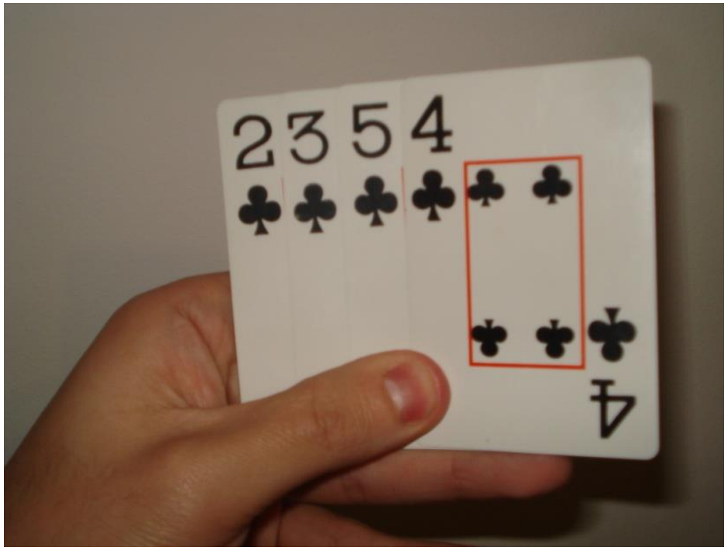
i = 1

chave = 2

j = -1

Insertion Sort

```
void insertion(int *vetor, int n)
{
   int chave, i, j;
   for(i=1; i<n; i++)
   {
     chave = vetor [ i ];
     j = i-1;
     while(j >= 0 && vetor [ j ] >= chave)
     {
        vetor [ j+1 ] = vetor [ j ]
        vetor [ j ] = chave
        j + j - 1
     }
   }
}
```



	0	1	2	3
vetor =	2	3	5	4

n = 4

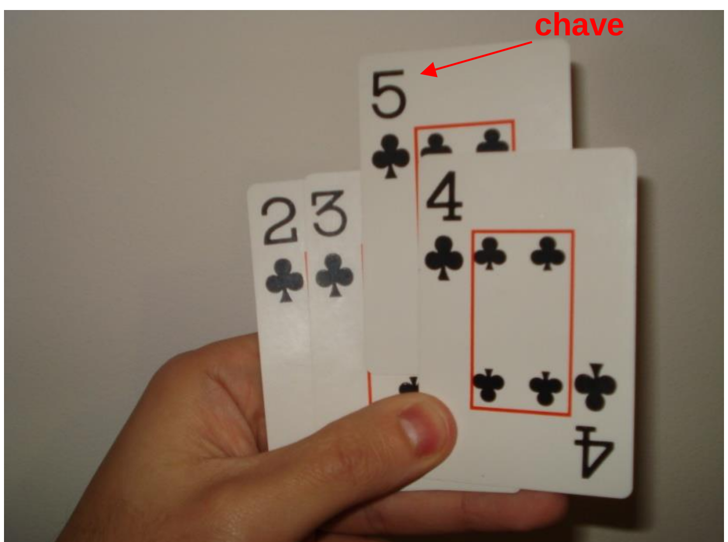
i = ~~1~~ 2

chave = 2

j = -1

Insertion Sort

```
void insertion(int *vetor, int n)
{
   int chave, i, j;
   for(i=1; i<n; i++)
   {
     chave = vetor [ i ];
     j = i-1;
     while(j >= 0 && vetor [ j ] >= chave)
     {
        vetor [ j+1 ] = vetor [ j ]
        vetor [ j ] = chave
        j + j - 1
     }
   }
}
```



	0	1	2	3
vetor =	2	3	5	4

n = 4

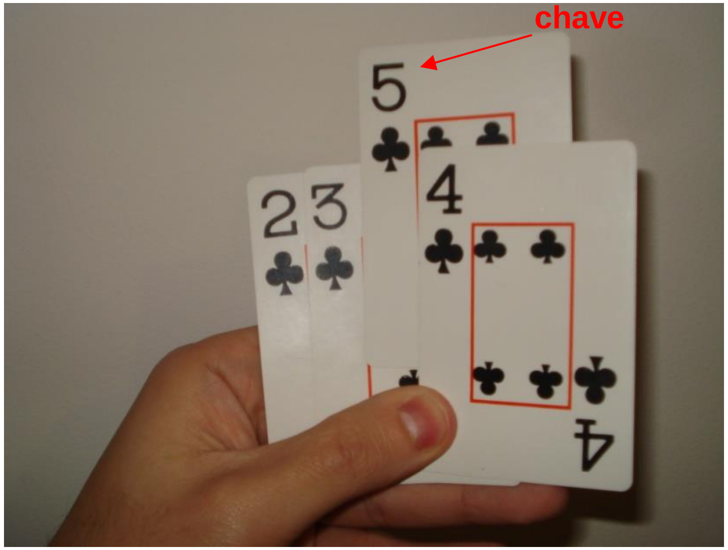
i = 2

chave = 5

j = -1

Insertion Sort

```
void insertion(int *vetor, int n)
{
   int chave, i, j;
   for(i=1; i<n; i++)
   {
     chave = vetor [ i ];
     j = i-1;
     while(j >= 0 && vetor [ j ] >= chave)
     {
        vetor [ j+1 ] = vetor [ j ]
        vetor [ j ] = chave
        j + j - 1
     }
   }
}
```



vetor =

0	1	2	3
2	3	5	4

n = 4

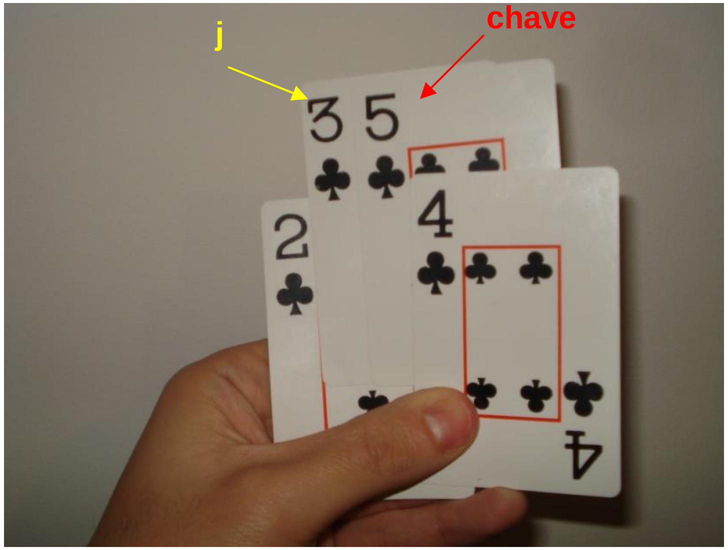
i = 2

chave = 5

j = 1

Insertion Sort

```
void insertion(int *vetor, int n)
{
   int chave, i, j;
   for(i=1; i<n; i++)
   {
     chave = vetor [ i ];
     j = i-1;
     while(j >= 0 && vetor [ j ] >= chave)
     {
        vetor [ j+1 ] = vetor [ j ]
        vetor [ j ] = chave
        j + j - 1
     }
   }
}
```



	0	1	2	3
vetor =	2	3	5	4

n = 4

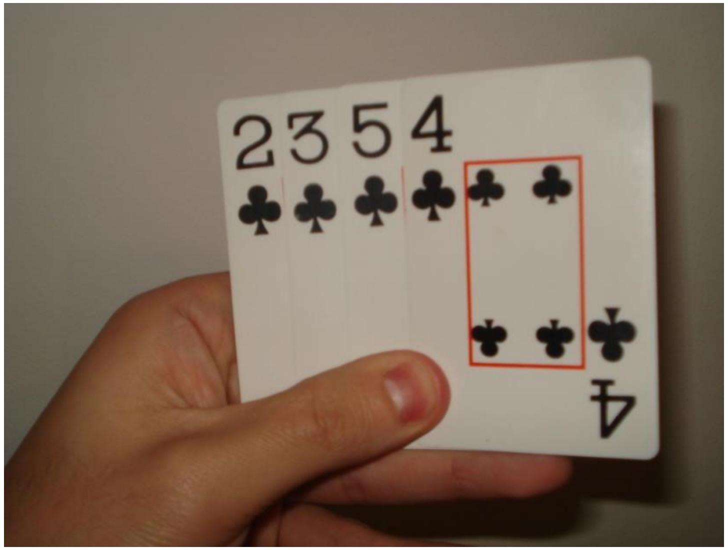
i = 2

chave = 5

j = 1

Insertion Sort

```
void insertion(int *vetor, int n)
{
   int chave, i, j;
   for(i=1; i<n; i++)
   {
     chave = vetor [ i ];
     j = i-1;
     while(j >= 0 && vetor [ j ] >= chave)
     {
        vetor [ j+1 ] = vetor [ j ]
        vetor [ j ] = chave
        j + j - 1
     }
   }
}
```



	0	1	2	3
vetor =	2	3	5	4

n = 4

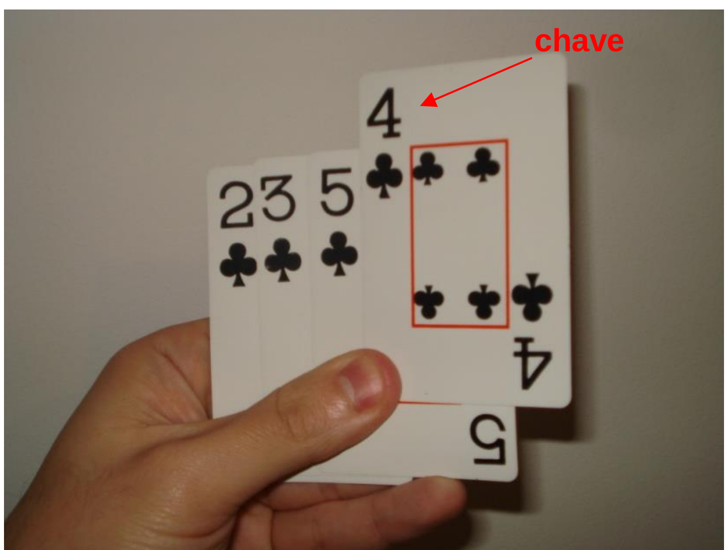
i = ~~2~~ 3

chave = 5

j = 1

Insertion Sort

```
void insertion(int *vetor, int n)
{
   int chave, i, j;
   for(i=1; i<n; i++)
   {
     chave = vetor [ i ];
     j = i-1;
     while(j >= 0 && vetor [ j ] >= chave)
     {
        vetor [ j+1 ] = vetor [ j ]
        vetor [ j ] = chave
        j + j - 1
     }
   }
}
```



	0	1	2	3
vetor =	2	3	5	4

n = 4

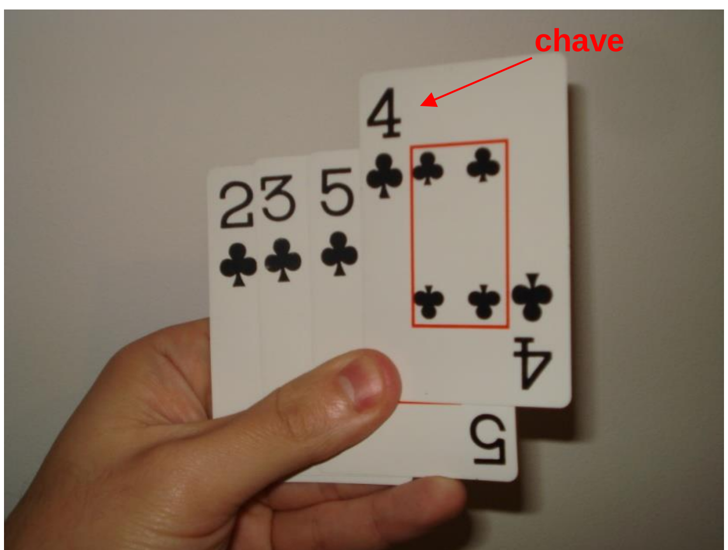
i = 3

chave = 4

j = 1

Insertion Sort

```
void insertion(int *vetor, int n)
{
   int chave, i, j;
   for(i=1; i<n; i++)
   {
     chave = vetor [ i ];
     j = i-1;
     while(j >= 0 && vetor [ j ] >= chave)
     {
        vetor [ j+1 ] = vetor [ j ]
        vetor [ j ] = chave
        j + j - 1
     }
   }
}
```



	0	1	2	3
vetor =	2	3	5	4

n = 4

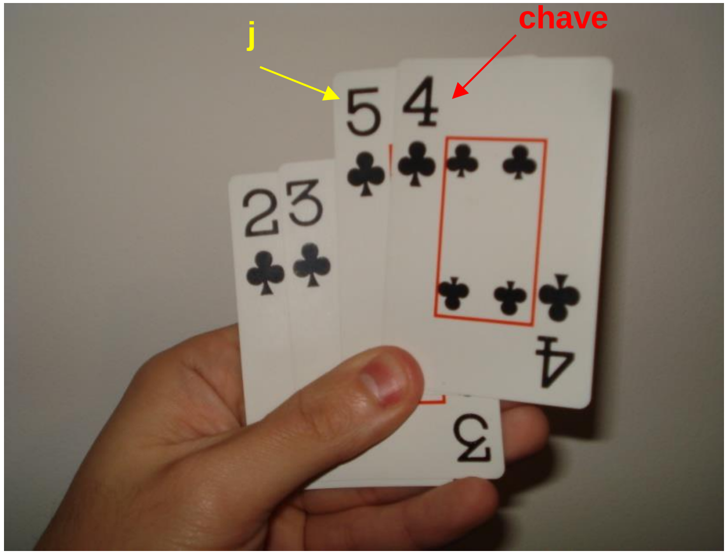
i = 3

chave = 4

j = 2

Insertion Sort

```
void insertion(int *vetor, int n)
{
   int chave, i, j;
   for(i=1; i<n; i++)
   {
     chave = vetor [ i ];
     j = i-1;
     while(j >= 0 && vetor [ j ] >= chave)
     {
        vetor [ j+1 ] = vetor [ j ]
        vetor [ j ] = chave
        j + j - 1
     }
   }
}
```



	0	1	2	3
vetor =	2	3	5	4

n = 4

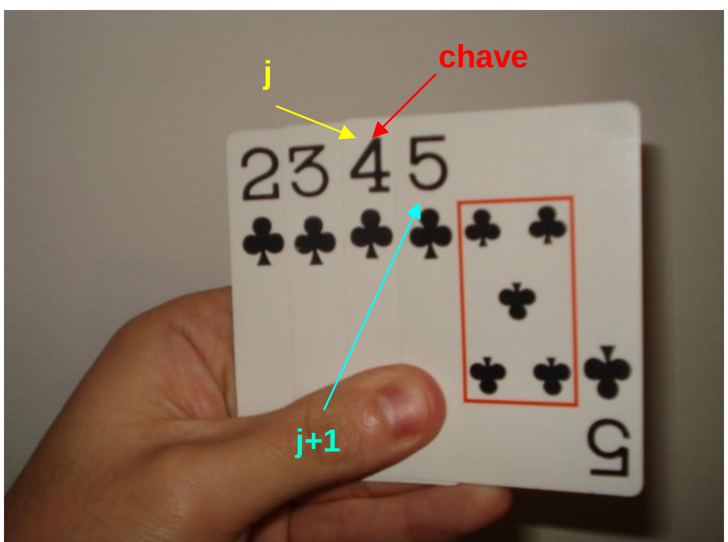
i = 3

chave = 4

j = 2

Insertion Sort

```
void insertion(int *vetor, int n)
{
   int chave, i, j;
   for(i=1; i<n; i++)
   {
     chave = vetor [ i ];
     j = i-1;
     while(j >= 0 && vetor [ j ] >= chave)
     {
        vetor [ j+1 ] = vetor [ j ]
        vetor [ j ] = chave
        j + j - 1
     }
   }
}
```



j+1

0	1	2	3
2	3	5	5

vetor = [2, 3, 5, 5]

n = 4

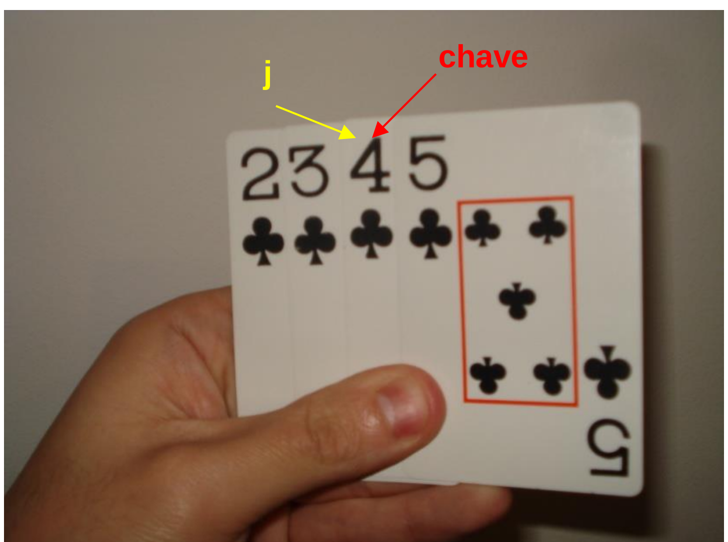
i = 3

chave = 4

j = 2

Insertion Sort

```
void insertion(int *vetor, int n)
{
   int chave, i, j;
   for(i=1; i<n; i++)
   {
     chave = vetor [ i ];
     j = i-1;
     while(j >= 0 && vetor [ j ] >= chave)
     {
        vetor [ j+1 ] = vetor [ j ]
        vetor [ j ] = chave
        j + j - 1
     }
   }
}
```



	0	1	2	3
vetor =	2	3	4	5

n = 4

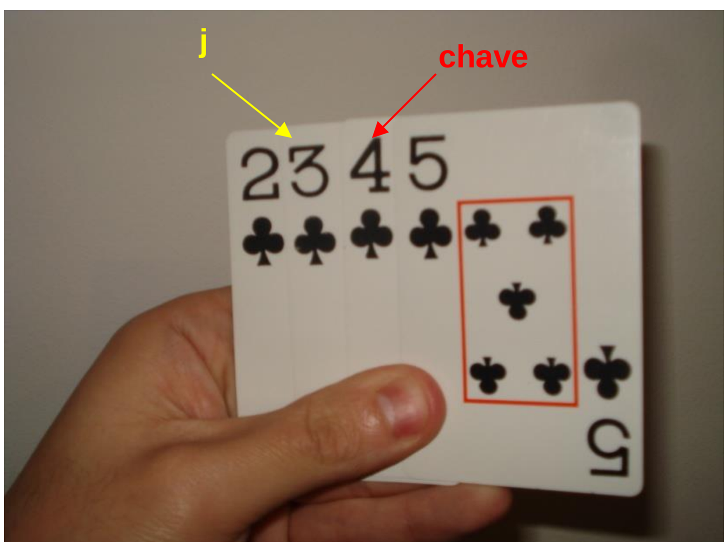
i = 3

chave = 4

j = 2

Insertion Sort

```
void insertion(int *vetor, int n)
{
   int chave, i, j;
   for(i=1; i<n; i++)
   {
     chave = vetor [ i ];
     j = i-1;
     while(j >= 0 && vetor [ j ] >= chave)
     {
        vetor [ j+1 ] = vetor [ j ]
        vetor [ j ] = chave
        j + j - 1
     }
   }
}
```



	0	1	2	3
vetor =	2	3	4	5

n = 4

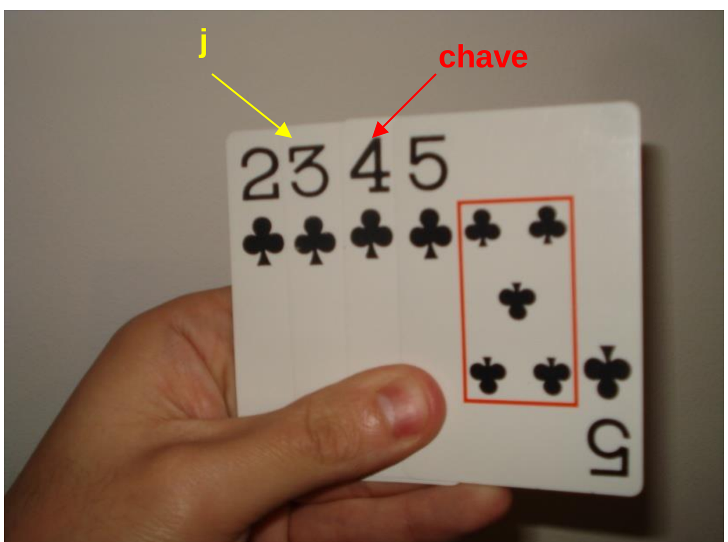
i = 3

chave = 4

j = ~~2~~ 1

Insertion Sort

```
void insertion(int *vetor, int n)
{
   int chave, i, j;
   for(i=1; i<n; i++)
   {
     chave = vetor [ i ];
     j = i-1;
     while(j >= 0 && vetor [ j ] >= chave)
     {
        vetor [ j+1 ] = vetor [ j ]
        vetor [ j ] = chave
        j + j - 1
     }
   }
}
```



	0	1	2	3
vetor =	2	3	4	5

n = 4

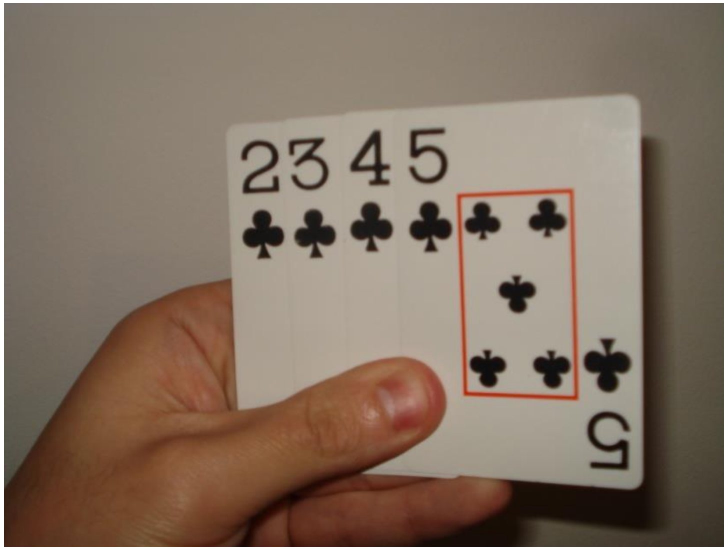
i = 3

chave = 4

j = 1

Insertion Sort

```
void insertion(int *vetor, int n)
{
   int chave, i, j;
   for(i=1; i<n; i++)
   {
     chave = vetor [ i ];
     j = i-1;
     while(j >= 0 && vetor [ j ] >= chave)
     {
        vetor [ j+1 ] = vetor [ j ]
        vetor [ j ] = chave
        j + j - 1
     }
   }
}
```



vetor =

0	1	2	3
2	3	4	5

n = 4

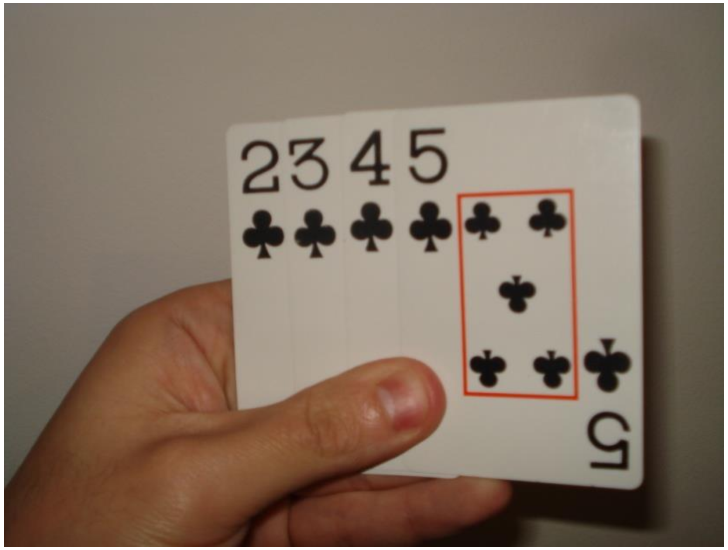
i = ~~3~~ 4

chave = 4

j = 1

Insertion Sort

```
void insertion(int *vetor, int n)
{
   int chave, i, j;
   for(i=1; i<n; i++)
   {
     chave = vetor [ i ];
     j = i-1;
     while(j >= 0 && vetor [ j ] >= chave)
     {
        vetor [ j+1 ] = vetor [ j ]
        vetor [ j ] = chave
        j + j - 1
     }
   }
}
```



	0	1	2	3
vetor =	2	3	4	5

n = 4

i = 4

chave = 4

j = 1

Eficiência

- Qual o esforço computacional necessário para o Insertion Sort ordenar n elementos no pior caso?
  - Qual a função primitiva do Insertion sort?

```
void insertion(int *vetor, int n)
{
   int chave, i, j;
   for(i=1; i<n; i++)
   {
      chave = vetor [ i ];
      j = i-1;
      while(j >= 0 && vetor [ j ] >= chave)
      {
         vetor [ j+1 ] = vetor [ j ]
         vetor [ j ] = chave
         j + j - 1
      }
   }
}
```

Eficiência

- Quantas vezes ocorre a função primitiva no pior caso?
  - Pense para o exemplo: n=4
    - Total: ?

Eficiência

```
void insertion(int *vetor, int n)
{
   int chave, i, j;
   for(i=1; i<n; i++)
   {
      chave = vetor [ i ];
      j = i-1;
      while(j >= 0 && vetor [ j ] >= chave)
      {
         vetor [ j+1 ] = vetor [ j ]
         vetor [ j ] = chave
         j + j - 1
      }
   }
}
```

Eficiência

- Quantas vezes ocorre a função primitiva no pior caso?
  - Pense para o exemplo: n=4
    - Total: 6 x
  - Agora generalize para um vetor de tamanho n.

Eficiência

```
void insertion(int *vetor, int n)
{
   int chave, i, j;
   for(i=1; i<n; i++)
   {
      chave = vetor [ i ];
      j = i-1;
      while(j >= 0 && vetor [ j ] >= chave)
      {
         vetor [ j+1 ] = vetor [ j ]
         vetor [ j ] = chave
         j + j - 1
      }
   }
}
```

Eficiência

- Quantas vezes ocorre a função primitiva no pior caso?
  - Pense para o exemplo: n=4
    - Total: 6 x
  - Agora generalize para um vetor de tamanho n.

Eficiência

```
void insertion(int *vetor, int n)
{
   int chave, i, j;
   for(i=1; i<n; i++)
   {
      chave = vetor [ i ];
      j = i-1;
      while(j >= 0 && vetor [ j ] >= chave)
      {
         vetor [ j+1 ] = vetor [ j ]
         vetor [ j ] = chave
         j + j - 1
      }
   }
}
```

Eficiência

- Quantas vezes ocorre a função primitiva no pior caso?
  - Pense para o exemplo: n=4
    - Total: 6 x
  - Agora generalize para um vetor de tamanho n.

$$T(n) = 1 + 2 + 3 + \cdots + (n-3) + (n-2) + (n-1)$$

Eficiência

- Quantas vezes ocorre a função primitiva no pior caso?
  - Pense para o exemplo: n=4
    - Total: 6 x
  - Agora generalize para um vetor de tamanho n.

$$T(n) = 1 + 2 + 3 + \cdots + (n-3) + (n-2) + (n-1)$$
$$T(n) = \frac{n^2 - n}{2}$$

Eficiência

Eficiência

- Quantas vezes ocorre a função primitiva no **MELHOR** caso?
  - Pense para o exemplo: n=4
    - Total: ?

Eficiência

```
void insertion(int *vetor, int n)
{
   int chave, i, j;
   for(i=1; i<n; i++)
   {
      chave = vetor [ i ];
      j = i-1;
      while(j >= 0 && vetor [ j ] >= chave)
      {
         vetor [ j+1 ] = vetor [ j ]
         vetor [ j ] = chave
         j + j - 1
      }
   }
}
```

Eficiência

- Quantas vezes ocorre a função primitiva no pior caso?
  - Pense para o exemplo: n=4
    - Total: 3 x
  - Agora generalize para um vetor de tamanho n.

Eficiência

```
void insertion(int *vetor, int n)
{
   int chave, i, j;
   for(i=1; i<n; i++)
   {
      chave = vetor [ i ];
      j = i-1;
      while(j >= 0 && vetor [ j ] >= chave)
      {
         vetor [ j+1 ] = vetor [ j ]
         vetor [ j ] = chave
         j + j - 1
      }
   }
}
```

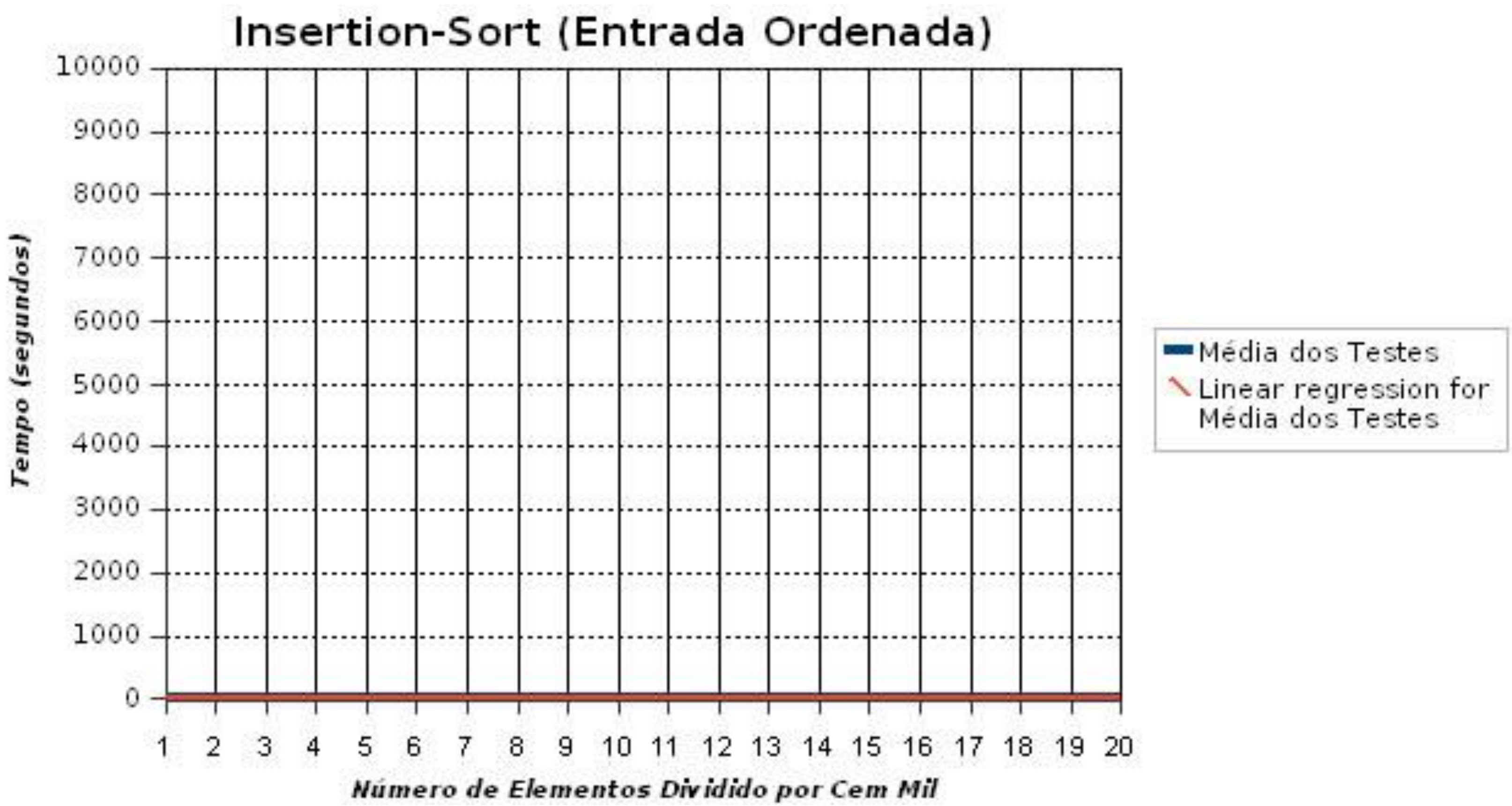
Eficiência

- Quantas vezes ocorre a função primitiva no pior caso?
  - Pense para o exemplo: n=4
    - Total: 3 x
  - Agora generalize para um vetor de tamanho n.

$$T(n) = n - 1$$

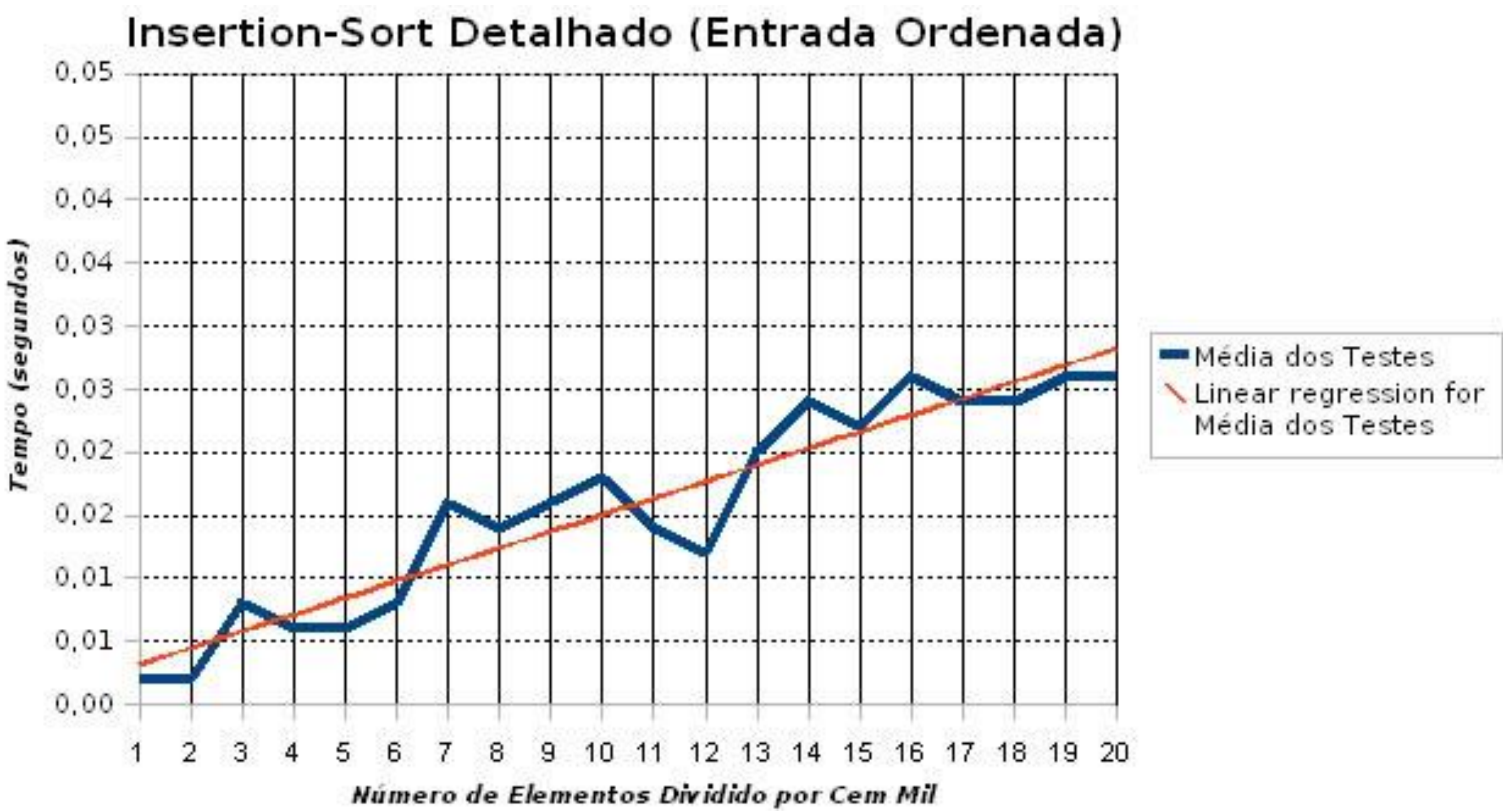
Insertion Sort

- Análise Empírica:



Insertion Sort

- Análise Empírica:

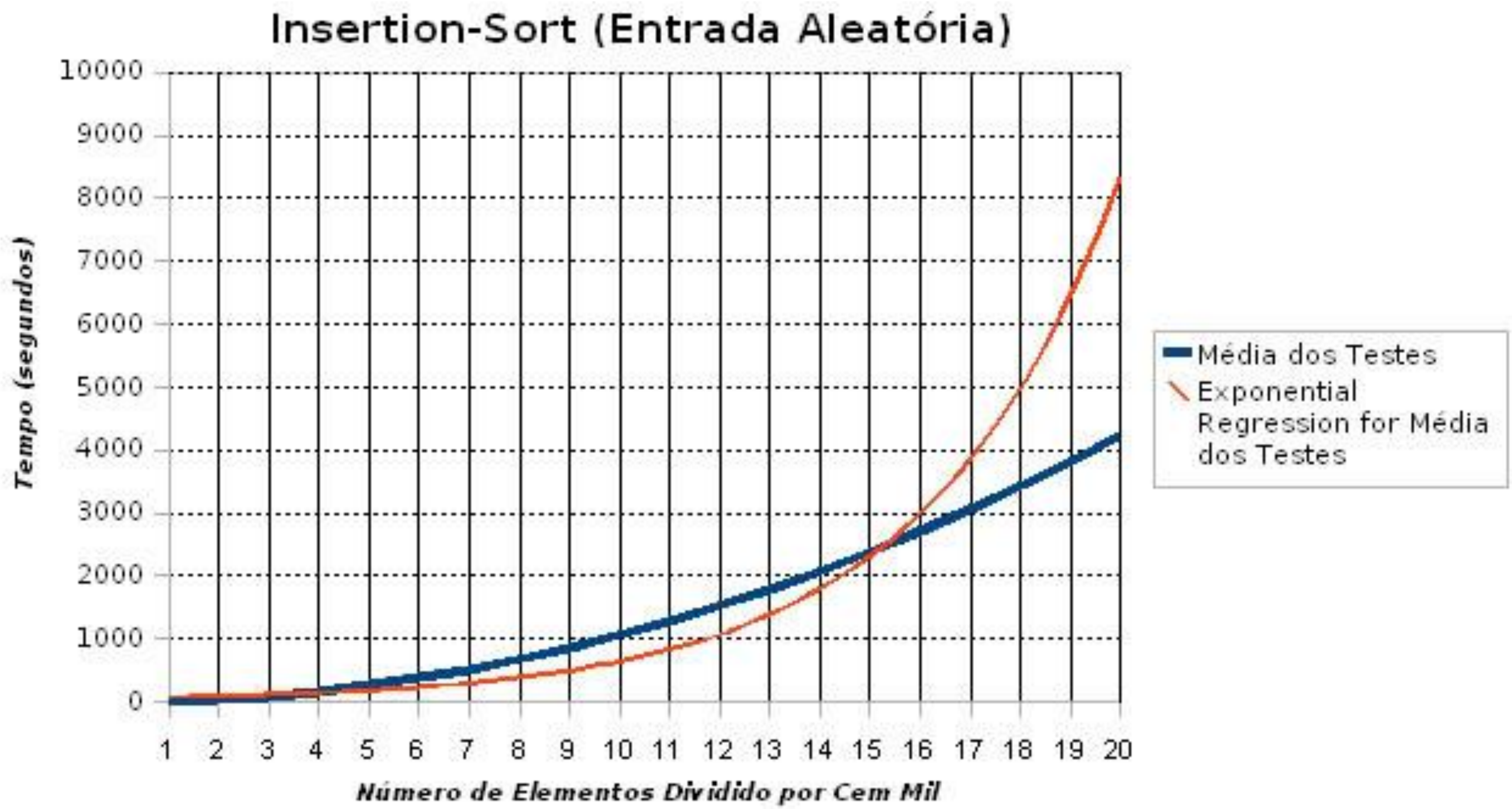


Insertion Sort

- Análise Empírica:

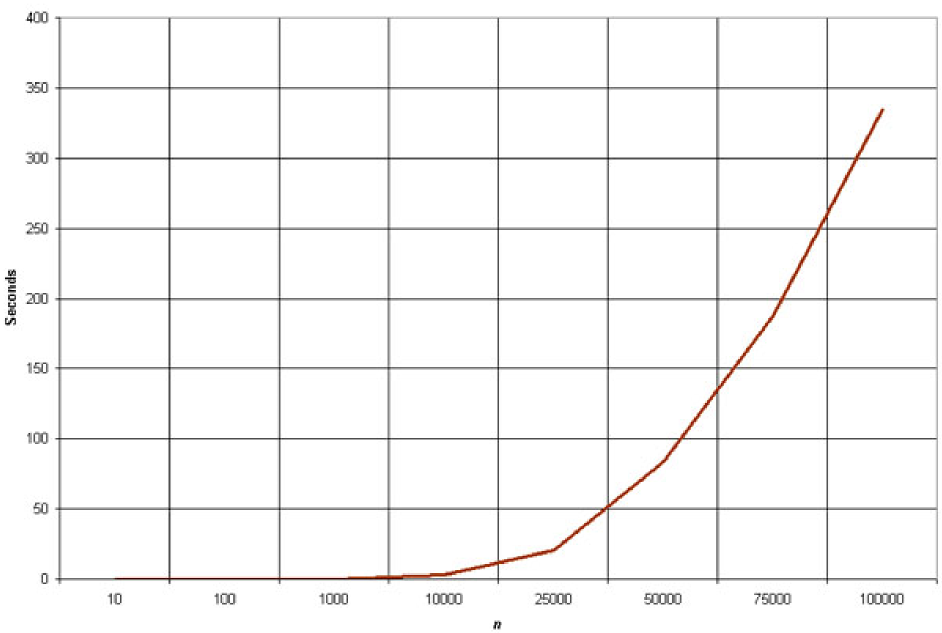
Insertion Sort

- Análise Empírica:



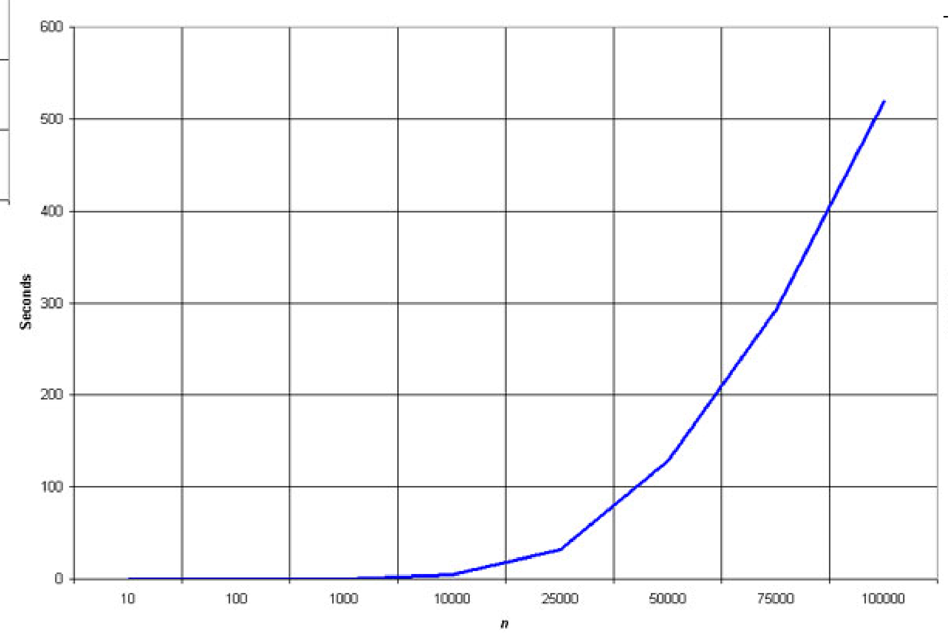
Insertion Sort e Selection Sort

- Análise Empírica:



Insertion Sort Efficiency

Selection Sort Efficiency



Estabilidade

- Um algoritmo de ordenação é estável caso ele preserve a ordem original das chaves iguais.
  - Exemplo: Seja a coleção **V** a seguir:

0	1	2	3
4	9	1	4

  - Após ser submetida à um procedimento de ordenação estável resultará:

0	1	2	3
1	4	4	9

  - De modo que a ordem original das chaves de valor 4 foi preservada

Estabilidade

- O Bubble Sort, Selection Sort e Insertion Sort como apresentados aqui são estáveis.

Vale a Pena Estudar

- Bubble Sort Recursivo
- Selection Sort Recursivo
- Insertion Sort Recursivo