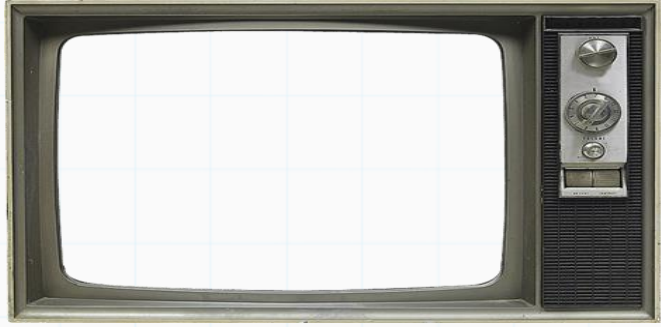
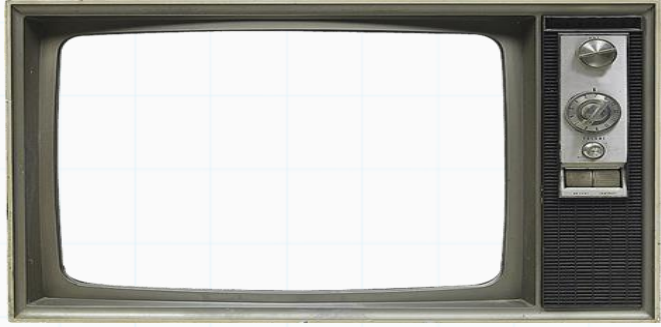
ProgramaçãoEstruturada

Professor : Yuri Frota

yuri@ic.uff.br



Ordenação

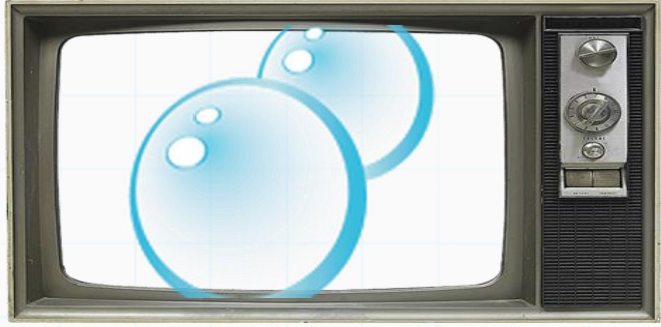


Ordenação: Em Programação 1, vimos 2 métodos simples de ordenação:

- Método da Bolha
- Método da Seleção

- Esses métodos realizavam processos de varreduras nos vetores, que alteravam sua composição, deixando um passo mais perto de estar ordenados.
- Essas varreduras tinham que ser repetidas n vezes (número de elementos do vetor) para garantir a ordenação.

Ordenação

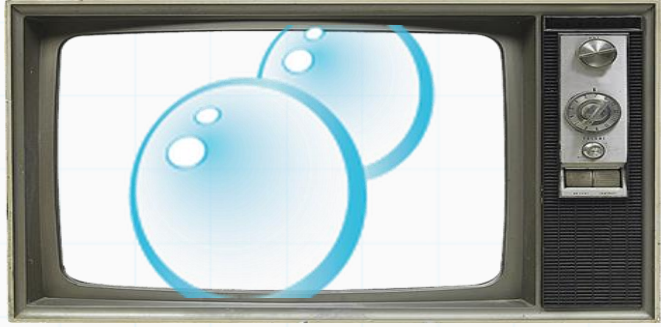


Ordenação da Bolha: seja o seguinte vetor a ser ordenado de forma crescente

4	3	5	1

```
for (j=0; j<n-1; j++)
    if (v[j] > v[j+1])
    {
        temp        = v[j];
        v[j]        = v[j+1];
        v[j+1]      = temp;
    }
```

Ordenação



Ordenação da Bolha: seja o seguinte vetor a ser ordenado de forma crescente

Vamos iterativamente olhar para as posições adjacentes e toda vez que o par de elementos não estiverem ordenados vamos troca-los

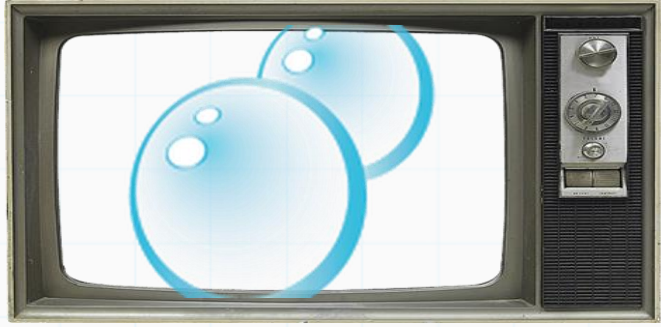
4	3	5	1

VARREDURA

4	3	5	1
i	i+1		

```
for (j=0; j<n-1; j++)
    if (v[j] > v[j+1])
    {
        temp        = v[j];
        v[j]        = v[j+1];
        v[j+1]      = temp;
    }
```

Ordenação



Ordenação da Bolha: seja o seguinte vetor a ser ordenado de forma crescente

Vamos iterativamente olhar para as posições adjacentes e toda vez que o par de elementos não estiverem ordenados vamos troca-los

4	3	5	1

VARREDURA

4	3	5	1

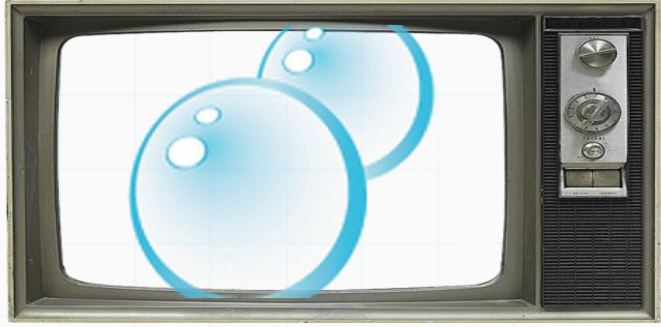
i i+1

3	4	5	1

i i+1

```
for (j=0; j<n-1; j++)
    if (v[j] > v[j+1])
    {
        temp        = v[j];
        v[j]        = v[j+1];
        v[j+1]      = temp;
    }
```

Ordenação



Ordenação da Bolha: seja o seguinte vetor a ser ordenado de forma crescente

Vamos iterativamente olhar para as posições adjacentes e toda vez que o par de elementos não estiverem ordenados vamos troca-los

4	3	5	1

VARREDURA

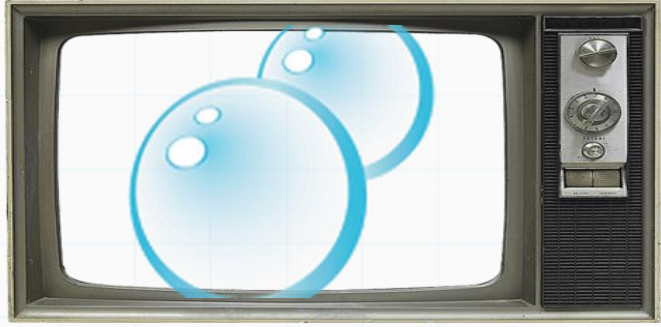
4	3	5	1
i	i+1		

3	4	5	1
	i	i+1	

3	4	5	1
		i	i+1

```
for (j=0; j<n-1; j++)
    if (v[j] > v[j+1])
    {
        temp        = v[j];
        v[j]        = v[j+1];
        v[j+1]      = temp;
    }
```

Ordenação



Ordenação da Bolha: seja o seguinte vetor a ser ordenado de forma crescente

Vamos iterativamente varrer as posições adjacentes e toda vez que o par de elementos não estiverem ordenados vamos troca-los

4	3	5	1

VARREDURA

4	3	5	1
i	i+1		

3	4	5	1
	i	i+1	

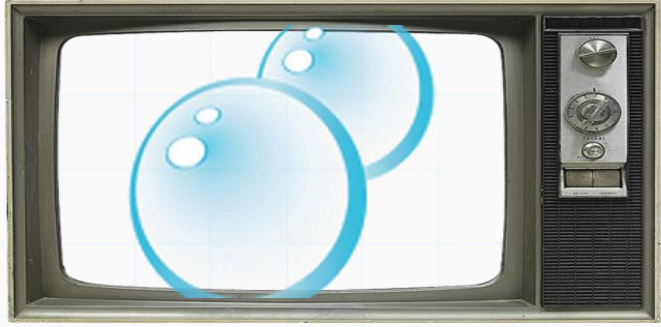
3	4	5	1
		i	i+1

3	4	1	5

o vetor está ordenado ?
o vetor está mais ordenado ?

```
for (j=0; j<n-1; j++)
    if (v[j] > v[j+1])
    {
        temp        = v[j];
        v[j]        = v[j+1];
        v[j+1]      = temp;
    }
```

Ordenação



Ordenação da Bolha: seja o seguinte vetor a ser ordenado de forma crescente

Vamos iterativamente varrer as posições adjacentes e toda vez que o par de elementos não estiverem ordenados vamos troca-los

4	3	5	1

3	4	1	5

VARREDURA

4	**3**	5	1
i	i+1		

3	**4**	5	1
	i	i+1	

3	**4**	**5**	1
		i	i+1

3	4	**1**	**5**

3	**4**	1	5
i	i+1		

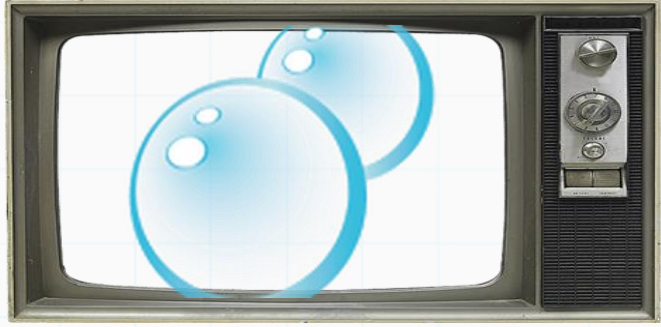
3	**4**	**1**	5
	i	i+1	

3	**1**	**4**	5
		i	i+1

3	1	4	5

```
for (i=0; i<n; i++)
    for (j=0; j<n-1; j++)
        if (v[j] > v[j+1])
        {
            temp        = v[j];
            v[j]        = v[j+1];
            v[j+1]      = temp;
        }
```

Ordenação



Ordenação da Bolha: seja o seguinte vetor a ser ordenado de forma crescente

Vamos iterativamente varrer as posições adjacentes e toda vez que o par de elementos não estiverem ordenados vamos troca-los

4	3	5	1

3	4	1	5

3	1	4	5

VARREDURA

4	3	5	1
i	i+1		

3	4	1	5
i	i+1		

3	1	4	5
i	i+1		

3	4	5	1
	i	i+1	

3	4	1	5
	i	i+1	

1	3	4	5
	i	i+1	

3	4	5	1
		i	i+1

3	1	4	5
		i	i+1

1	3	4	5
		i	i+1

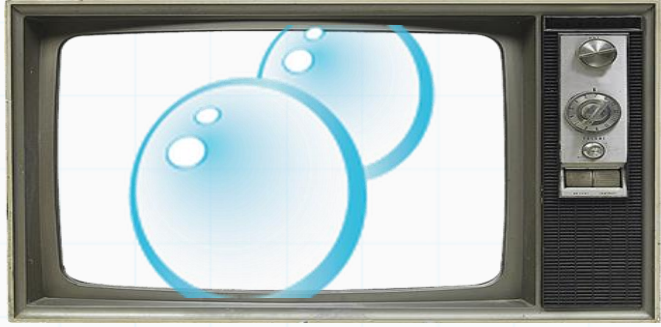
3	4	1	5

3	1	4	5

1	3	4	5

```
for (i=0; i<n; i++)
    for (j=0; j<n-1; j++)
        if (v[j] > v[j+1])
        {
            temp        = v[j];
            v[j]        = v[j+1];
            v[j+1]      = temp;
        }
```

Ordenação



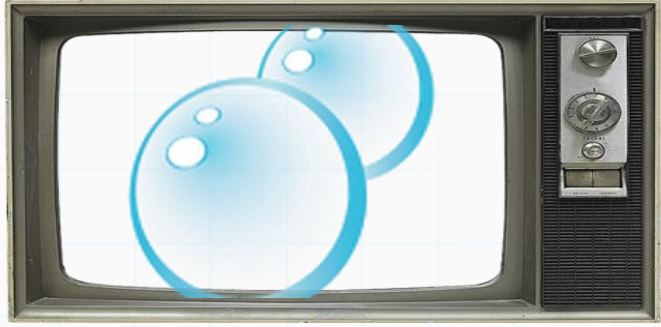
Ordenação da Bolha: Eficiência:

```
for (i=0; i<n; i++)
    for (j=0; j<n-1; j++)
        if (v[j] > v[j+1])
        {
            temp        = v[j];
            v[j]        = v[j+1];
            v[j+1]      = temp;
        }
```

Número de Comparações realizadas:

n . (n-1) $= n^2 - n$ => dizemos que é n^2 pois é o fator que domina

Ordenação



Ordenação da Bolha: Eficiência:

```
for (i=0; i<n; i++)
    for (j=0; j<n-1; j++)
        if (v[j] > v[j+1])
        {
            temp        = v[j];
            v[j]        = v[j+1];
            v[j+1]      = temp;
        }
```

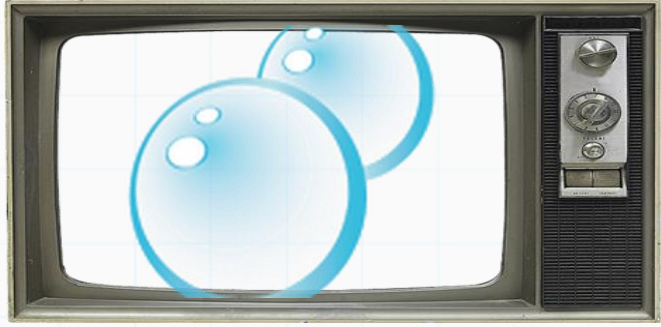
Número de Comparações realizadas:

n . (n-1) = $n^2 - n$ => dizemos que é n^2 pois é o fator que domina

Da para melhorar ? Sabendo que a cada varredura:

Varredura 1 -> Maior elemento deslocado para a ultima posição
Varredura 2 -> Maior elemento deslocado para a penúltima posição
...

Ordenação



Ordenação da Bolha: Eficiência:

```
for (i=0; i<n; i++)
    for (j=0; j<n-1; j++)
        if (v[j] > v[j+1])
        {
            temp            = v[j];
            v[j]            = v[j+1];
            v[j+1]          = temp;
        }
```

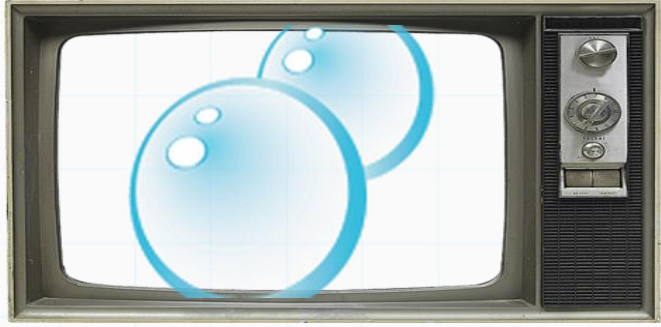
```
for (i=0; i<n; i++)
    for (j=0; j<n-1-i; j++)
        if (v[j] > v[j+1])
        {
            temp            = v[j];
            v[j]            = v[j+1];
            v[j+1]          = temp;
        }
```

Número de Comparações realizadas:

n . (n-1) = $n^2 - n$ => dizemos que é n^2 pois é o fator que domina

(n-1) + (n-2) + (n-3) + ... 2 = $(n^2 - n)/2$ => n^2

Ordenação



Ordenação da Bolha: Eficiência:

```
for (i=0; i<n; i++)
    for (j=0; j<n-1; j++)
        if (v[j] > v[j+1])
        {
            temp        = v[j];
            v[j]        = v[j+1];
            v[j+1]      = temp;
        }
```

Número de Comparações realizadas:

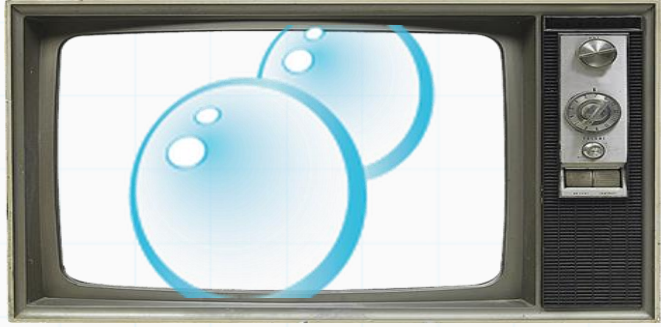
n . (n-1) = $n^2 - n$ => dizemos que é n^2 pois é o fator que domina

```
for (i=0; i<n; i++)
    for (j=0; j<n-1-i; j++)
        if (v[j] > v[j+1])
        {
            temp        = v[j];
            v[j]        = v[j+1];
            v[j+1]      = temp;
        }
```

(n-1) + (n-2) + (n-3) + ... 2 = $(n^2 - n)/2$ => n^2

Da para melhorar ? Podemos evitar varreduras desnecessárias, pois se numa varredura não ocorre troca, então as próximas também não trocarão.

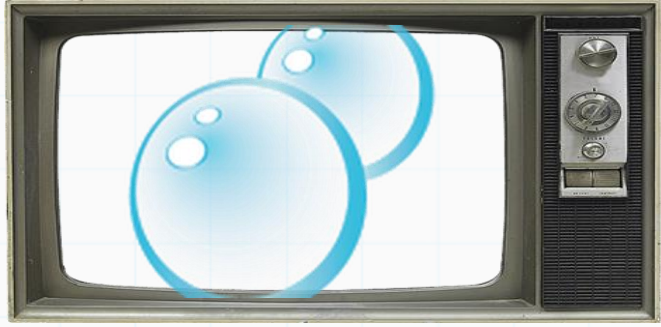
Ordenação



Ordenação da Bolha: Eficiência:

```
for (i=0; i<n; i++)
{
    trocou = 0;
    for (j=0; j<n-1-i; j++)
        if (v[j] > v[j+1])
        {
            temp        = v[j];
            v[j]        = v[j+1];
            v[j+1]      = temp;

            trocou = 1;
        }
    if (troca == 0)
        break;
}
```

Ordenação



Ordenação da Bolha: Eficiência:

```
for (i=0; i<n; i++)
{
    trocou = 0;
    for (j=0; j<n-1-i; j++)
        if (v[j] > v[j+1])
        {
            temp        = v[j];
            v[j]        = v[j+1];
            v[j+1]      = temp;

            trocou = 1;
        }
    if (troca == 0)
        break;
}
```

Melhor versão

Número de Comparações realizadas:

No pior caso:
$(n-1) + (n-2) + (n-3) + \ldots 2 = (n^2 - n)/2 \Rightarrow n^2$

No melhor caso:
$(n-1) \Rightarrow n$

Ordenação

Ordenação por seleção: a lista está dividida em parte ordenada e não ordenada, a cada iteração, encontramos o menor elemento da parte não ordenada para colocar na ordenada.

ord. | ñ ord.

4	3	5	1

```
for (i=0; i<n-1; i++)
    for (j=i+1; j<n; j++)
        if (v[i] > v[j])
        {
            temp        = v[i];
            v[i]        = v[j];
            v[j]        = temp;
        }
```

Ordenação

Ordenação por seleção: a lista está dividida em parte ordenada e não ordenada, a cada iteração, encontramos o menor elemento da parte não ordenada para colocar na ordenada.

ord. | ñ ord.

4	3	5	1

VARREDURA i=0

menor (i)

4	**3**	5	1

cand (j)

```
for (i=0; i<n-1; i++)
    for (j=i+1; j<n; j++)
        if (v[i] > v[j])
        {
            temp    = v[i];
            v[i]    = v[j];
            v[j]    = temp;
        }
```

Ordenação

Ordenação por seleção: a lista está dividida em parte ordenada e não ordenada, a cada iteração, encontramos o menor elemento da parte não ordenada para colocar na ordenada.

ord. | ñ ord.

4	3	5	1

VARREDURA i=0

menor (i)

4	**3**	5	1

cand (j)

menor (i)

3	4	**5**	1

cand (j)

```
for (i=0; i<n-1; i++)
    for (j=i+1; j<n; j++)
        if (v[i] > v[j])
        {
            temp    = v[i];
            v[i]    = v[j];
            v[j]    = temp;
        }
```

Ordenação

Ordenação por seleção: a lista está dividida em parte ordenada e não ordenada, a cada iteração, encontramos o menor elemento da parte não ordenada para colocar na ordenada.

ord. | ñ ord.

4	3	5	1

VARREDURA i=0

menor (i)

4	**3**	5	1

cand (j)

menor (i)

3	4	**5**	1

cand (j)

menor (i)

3	4	5	**1**

cand (j)

```
for (i=0; i<n-1; i++)
    for (j=i+1; j<n; j++)
        if (v[i] > v[j])
        {
            temp    = v[i];
            v[i]    = v[j];
            v[j]    = temp;
        }
```

Ordenação

Ordenação por seleção: a lista está dividida em parte ordenada e não ordenada, a cada iteração, encontramos o menor elemento da parte não ordenada para colocar na ordenada.

ord. | ñ ord.

4	3	5	1

VARREDURA i=0

menor (i)

4	**3**	5	1

cand (j)

menor (i)

3	4	**5**	1

cand (j)

menor (i)

3	4	5	**1**

cand (j)

1	4	5	3

```
for (i=0; i<n-1; i++)
    for (j=i+1; j<n; j++)
        if (v[i] > v[j])
        {
            temp    = v[i];
            v[i]    = v[j];
            v[j]    = temp;
        }
```

Ordenação

Ordenação por seleção: a lista está dividida em parte ordenada e não ordenada, a cada iteração, encontramos o menor elemento da parte não ordenada para colocar na ordenada.

ord. | ñ ord.

4 3 5 1

menor (i)

4 **3** 5 1

cand (j)

menor (i)

3 4 **5** 1

cand (j)

menor (i)

3 4 5 **1**

cand (j)

1 4 5 3

ord. | ñ ord.

menor (i)

1 **4** **5** 3

cand (j)

menor (i)

1 **4** 5 **3**

cand (j)

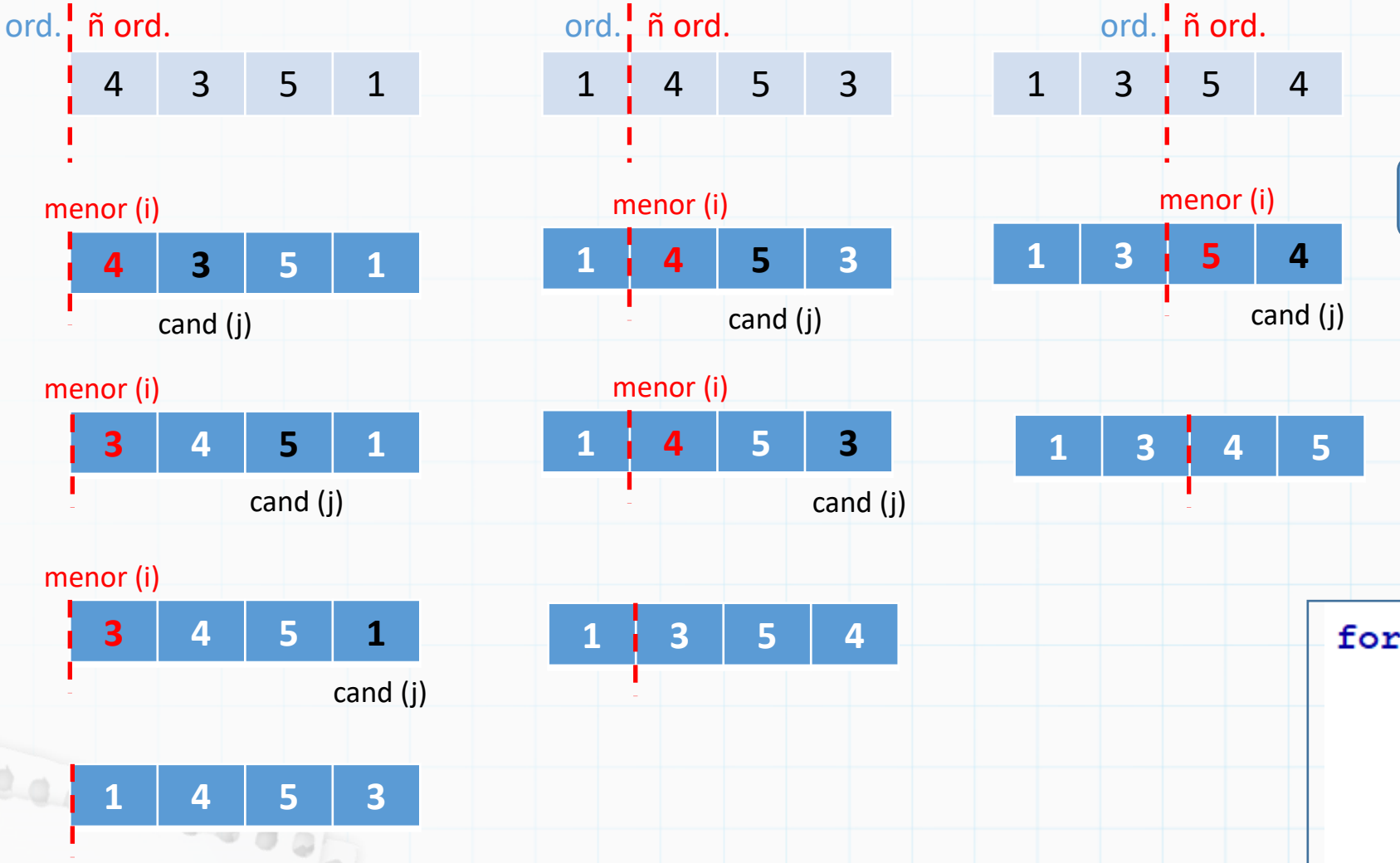
1 3 5 4

VARREDURA i=1

```
for (i=0; i<n-1; i++)
    for (j=i+1; j<n; j++)
        if (v[i] > v[j])
        {
            temp    = v[i];
            v[i]    = v[j];
            v[j]    = temp;
        }
```

Ordenação

Ordenação por seleção: a lista está dividida em parte ordenada e não ordenada, a cada iteração, encontramos o menor elemento da parte não ordenada para colocar na ordenada.



VARREDURA i=2

- Percorremos o vetor (n-1) vezes, procurando sempre o i-ésimo menor elemento

```
for (i=0; i<n-1; i++)
    for (j=i+1; j<n; j++)
        if (v[i] > v[j])
        {
            temp = v[i];
            v[i] = v[j];
            v[j] = temp;
        }
```

Ordenação

Ordenação por Seleção: Eficiência:

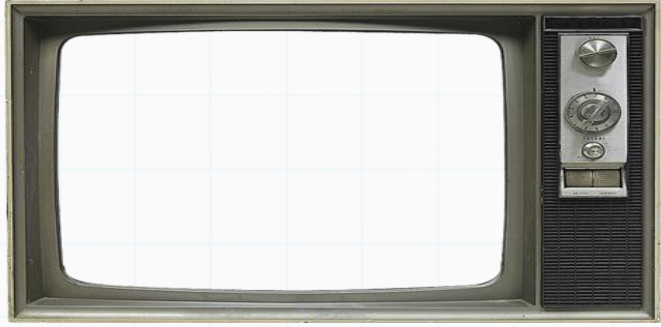
```
for (i=0; i<n-1; i++)
    for (j=i+1; j<n; j++)
        if (v[i] > v[j])
        {
            temp        = v[i];
            v[i]        = v[j];
            v[j]        = temp;
        }
```

Número de Comparações realizadas:

(n-1) + (n-2) + (n-3) ... = $(n^2 - n)/2$ => n^2

Mesma que o Bolha mas não tem como interromper as varreduras, então pior caso = melhor caso = n^2

Buscas

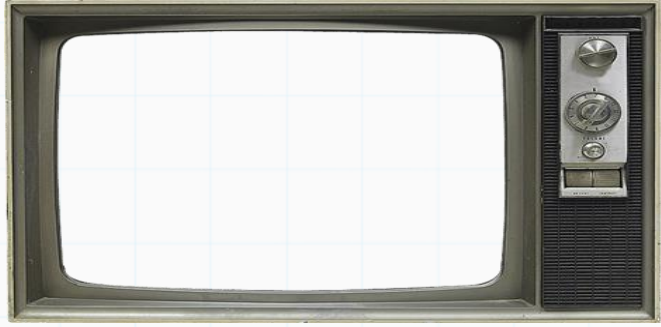


Busca Linear com vetor ordenado: Eficiência

```
void busca_vetor_ordenado(int* v, int tam, int el)
{
    for (int i=0; i<tam; i++)
    {
        if (v[i] == el)
        {
            printf("elemento %d na posicao %d\n\n",el, i);
            return;
        }

        if (el < v[i])
            break;
    }

    return;
}
```

Vamos falar de buscas agora, veja o algoritmo de busca linear ao lado

Buscas



Busca Linear com vetor ordenado: Eficiência

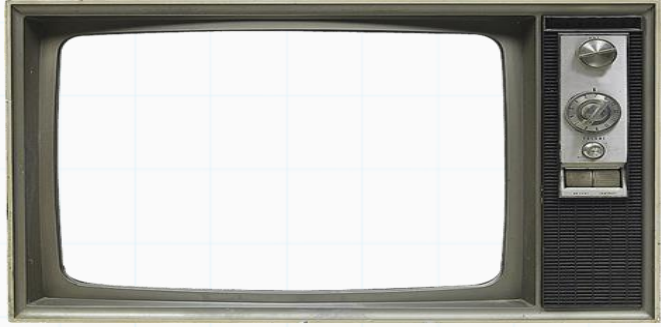
```
void busca_vetor_ordenado(int* v, int tam, int el)
{
    for (int i=0; i<tam; i++)
    {
        if (v[i] == el)
        {
            printf("elemento %d na posicao %d\n\n",el, i);
            return;
        }

        if (el < v[i])
            break;
    }

    return;
}
```

Número de Comparações realizadas:

pior caso =>

melhor caso =>

Buscas



Busca Linear com vetor ordenado: Eficiência

```
void busca_vetor_ordenado(int* v, int tam, int el)
{
    for (int i=0; i<tam; i++)
    {
        if (v[i] == el)
        {
            printf("elemento %d na posicao %d\n\n",el, i);
            return;
        }

        if (el < v[i])
            break;
    }

    return;
}
```

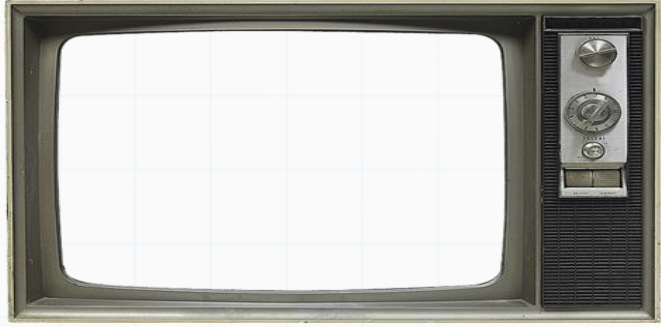
Número de Comparações realizadas:

pior caso => n

melhor caso => 1

Podemos fazer melhor que isso ?

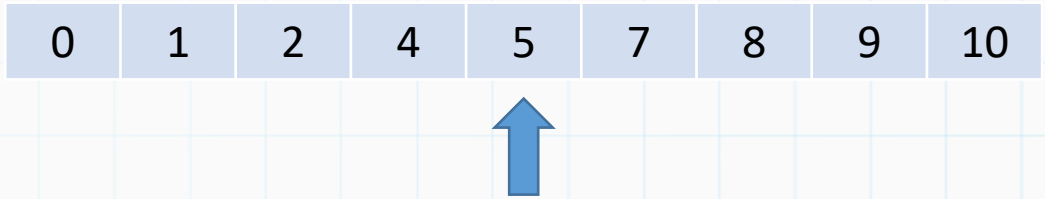
Buscas



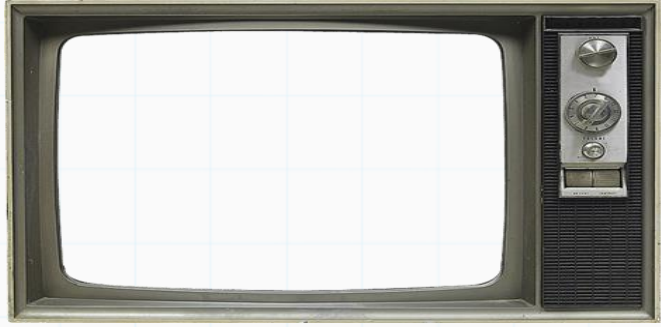
- Busca Binária: da para fazer melhor ainda ?

A ideia do algoritmo é a seguinte (assuma que o vetor está ordenado):

- Verifique se a chave de busca é igual ao valor da posição do meio do vetor.



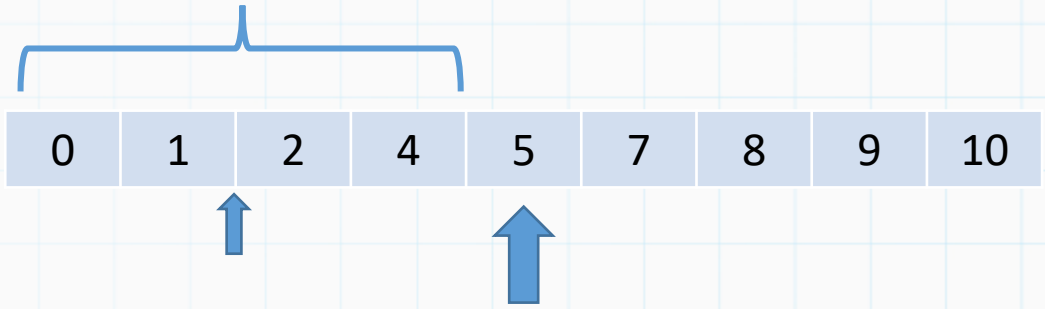
Buscas



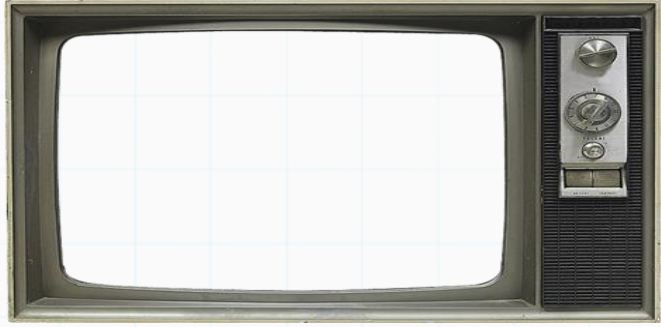
- Busca Binária: da para fazer melhor ainda ?

A ideia do algoritmo é a seguinte (assuma que o vetor está ordenado):

- Verifique se a chave de busca é igual ao valor da posição do meio do vetor.
- Caso o valor desta posição seja maior, então repita o processo mas considere que o vetor tem metade do tamanho, indo até posição anterior a do meio.



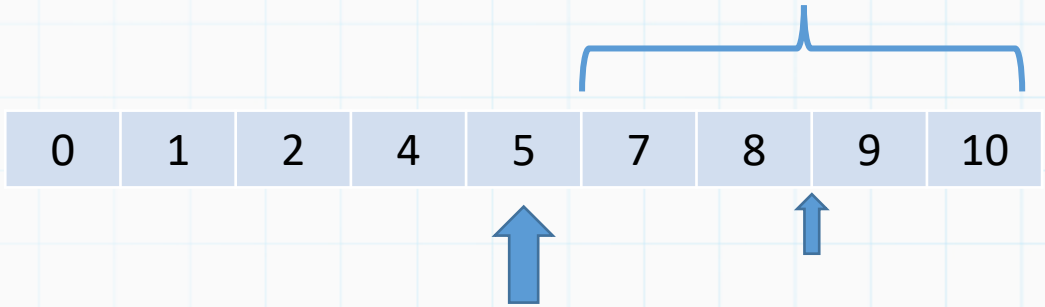
Buscas



- Busca Binária: da para fazer melhor ainda ?

A ideia do algoritmo é a seguinte (assuma que o vetor está ordenado):

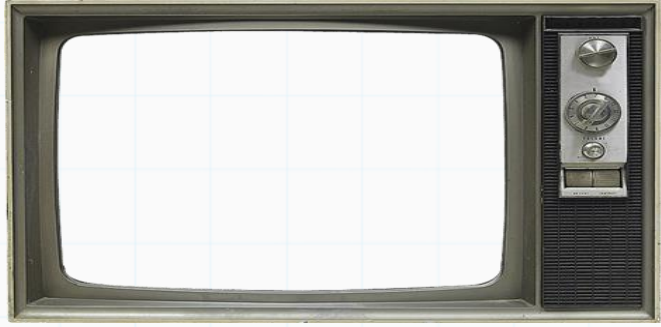
- Verifique se a chave de busca é igual ao valor da posição do meio do vetor.
- Caso o valor desta posição seja maior, então repita o processo mas considere que o vetor tem metade do tamanho, indo até posição anterior a do meio.
- Caso o valor desta posição seja menor, então repita o processo mas considere que o vetor tem metade do tamanho e inicia na posição seguinte a do meio



Buscas

- Busca Binária: Exemplo com vetor de tamanho N=7

meio = (ini + fim) / 2



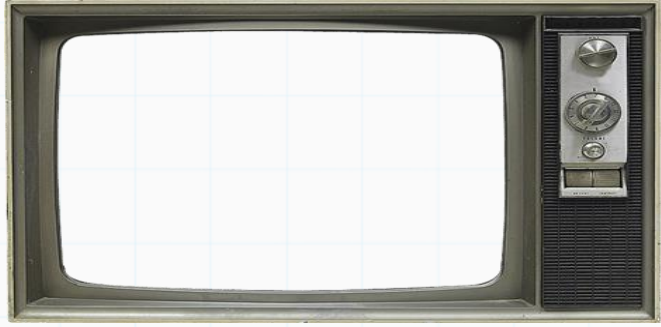
4	10	80	90	91	99	100	101

ini =
fim =
meio =

Buscas

- Busca Binária: Exemplo com vetor de tamanho N=7

meio = (ini + fim) / 2



100?

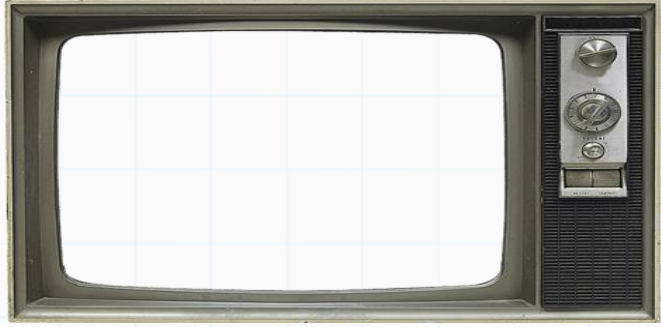
4	10	80	90	91	99	100	101

ini = 0
fim = 7
meio = (0+7)/2 = 3

Buscas

- Busca Binária: Exemplo com vetor de tamanho N=7

meio = (ini + fim) / 2



100?

4	10	80	90	91	99	100	101

menor

4	10	80	90	91	99	100	101

ini = 0
fim = 7
meio = (0+7)/2 = 3

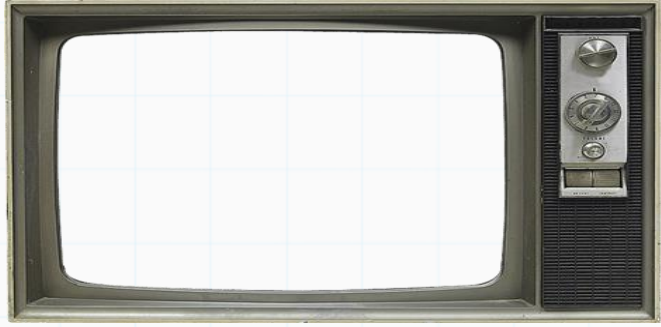
ini = 4
fim = 7
meio = (4+7)/2 = 5

ini = meio+1

Buscas

- Busca Binária: Exemplo com vetor de tamanho N=7

meio = (ini + fim) / 2



100?

4	10	80	90	91	99	100	101

menor

4	10	80	90	91	99	100	101

menor

4	10	80	90	91	99	100	101

ini = 0
fim = 7
meio = (0+7)/2 = 3

ini = 4
fim = 7
meio = (4+7)/2 = 5

ini = meio+1

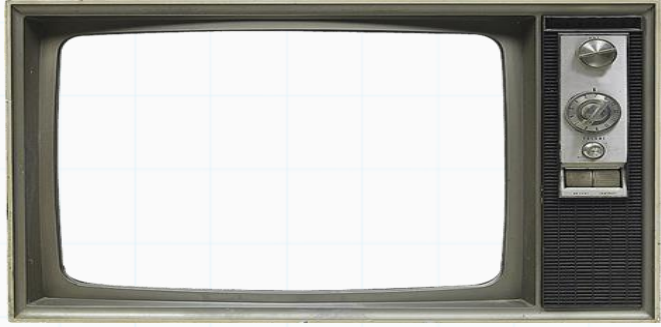
ini = 6
fim = 7
meio = (6+7)/2 = 6

3 movimentos

Buscas

- Busca Binária: Exemplo com vetor de tamanho N=7

meio = (ini + fim) / 2



11?

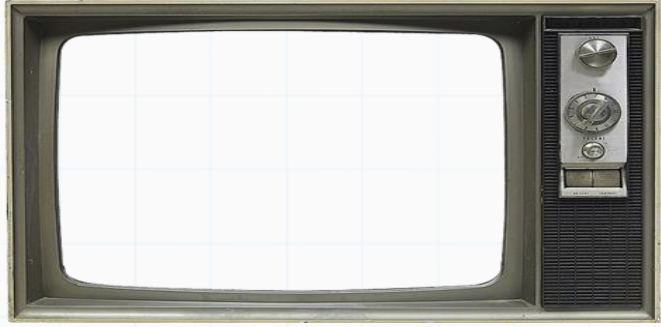
4	10	80	90	91	99	100	101

ini = 0
fim = 7
meio = (0+7)/2 = 3

Buscas

- Busca Binária: Exemplo com vetor de tamanho N=7

meio = (ini + fim) / 2



11?

4	10	80	90	91	99	100	101

maior

ini = 0
fim = 7
meio = (0+7)/2 = 3

4	10	80	90	91	99	100	101

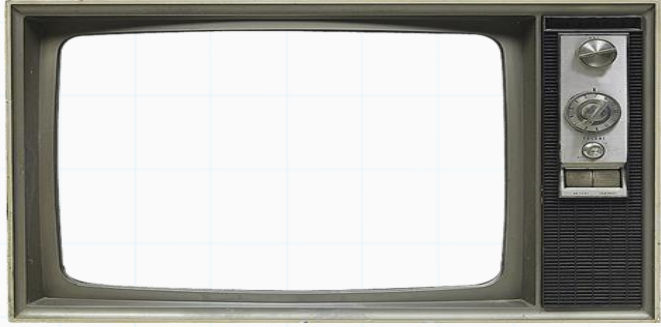
ini = 0
fim = 2
meio = (0+2)/2 = 1

fim = meio-1

Buscas

- Busca Binária: Exemplo com vetor de tamanho N=7

meio = (ini + fim) / 2



11?

4	10	80	90	91	99	100	101

maior

ini = 0
fim = 7
meio = (0+7)/2 = 3

4	10	80	90	91	99	100	101

menor

ini = 0
fim = 2
meio = (0+2)/2 = 1

fim = meio-1

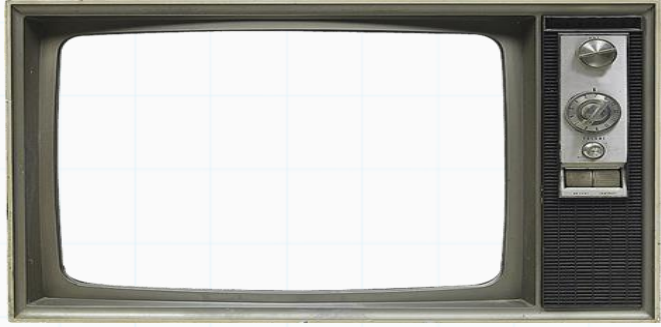
4	10	80	90	91	99	100	101

ini = 2
fim = 2
meio = (2+2)/2 = 2

Buscas

- Busca Binária: Exemplo com vetor de tamanho N=7

meio = (ini + fim) / 2



11?

4	10	80	90	91	99	100	101

maior

ini = 0
fim = 7
meio = (0+7)/2 = 3

4	10	80	90	91	99	100	101

menor

ini = 0
fim = 2
meio = (0+2)/2 = 1

fim = meio-1

4	10	80	90	91	99	100	101

maior

ini = 2
fim = 2
meio = (2+2)/2 = 2

4	10	80	90	91	99	100	101

ini = 2
fim = 1
meio = -1

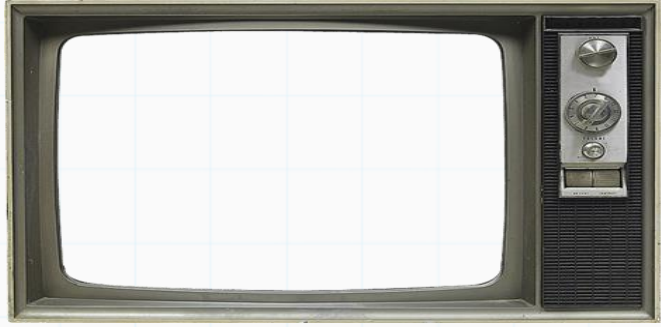
condição de parada
início passou do fim
Não achou, retorne -1

4 movimentos

Buscas

- Busca Binária: Exemplo com vetor de tamanho N=7

meio = (ini + fim) / 2



105?

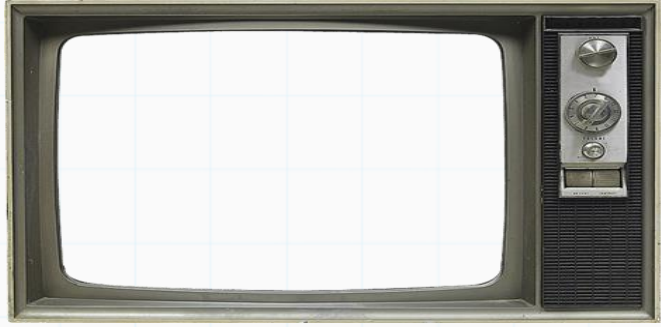
4	10	80	90	91	99	100	101

ini = 0
fim = 7
meio = (0+7)/2 = 3

Buscas

- Busca Binária: Exemplo com vetor de tamanho N=7

meio = (ini + fim) / 2



105?

4	10	80	90	91	99	100	101

menor

4	10	80	90	91	99	100	101

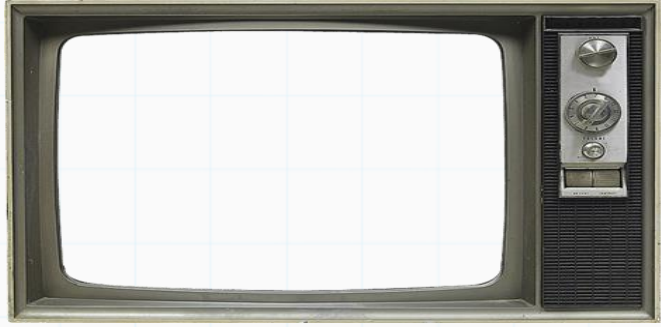
ini = 0
fim = 7
meio = (0+7)/2 = 3

ini = 4
fim = 7
meio = (4+7)/2 = 5

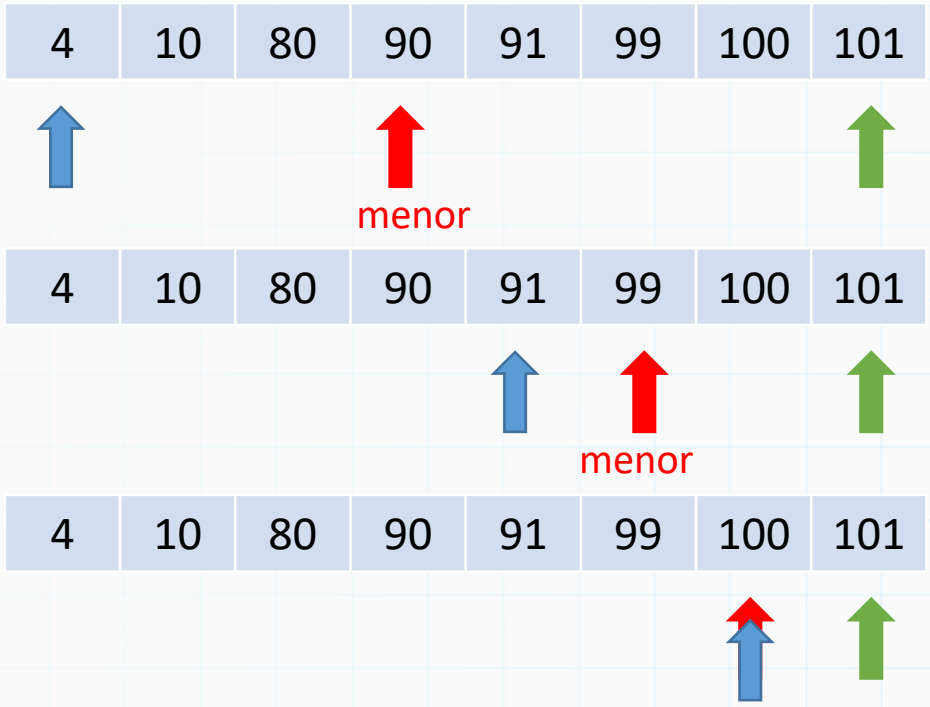
Buscas

- Busca Binária: Exemplo com vetor de tamanho N=7

meio = (ini + fim) / 2



105?



ini = 0
fim = 7
meio = (0+7)/2 = 3

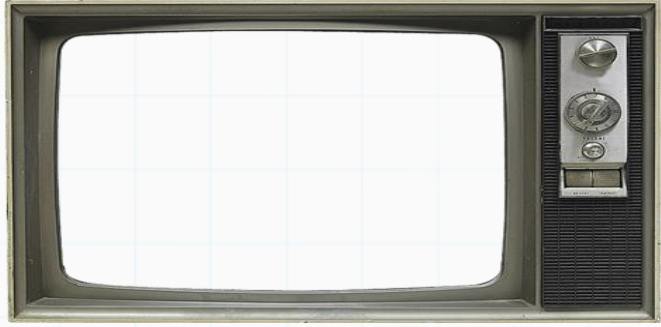
ini = 4
fim = 7
meio = (4+7)/2 = 5

ini = 6
fim = 7
meio = (6+7)/2 = 6

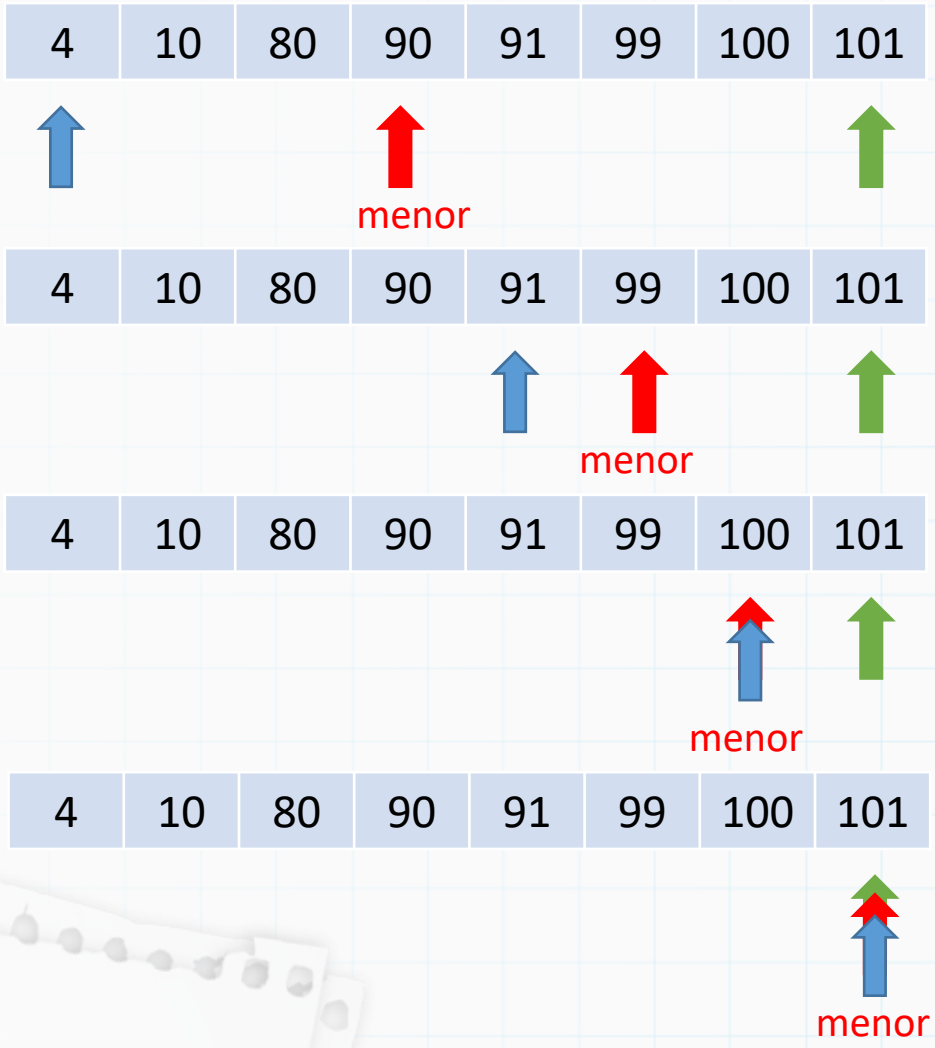
Buscas

- Busca Binária: Exemplo com vetor de tamanho N=7

meio = (ini + fim) / 2



105?



ini = 0
fim = 7
meio = (0+7)/2 = 3

ini = 4
fim = 7
meio = (4+7)/2 = 5

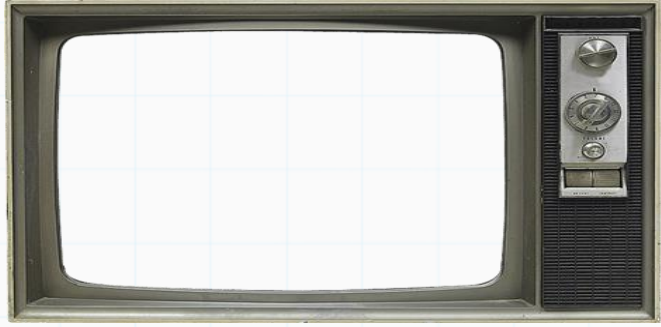
ini = 6
fim = 7
meio = (6+7)/2 = 6

ini = 7
fim = 7
meio = (7+7)/2 = 7

Buscas

- Busca Binária: Exemplo com vetor de tamanho N=7

meio = (ini + fim) / 2



105?

4	10	80	90	91	99	100	101

menor

4	10	80	90	91	99	100	101

menor

4	10	80	90	91	99	100	101

menor

4	10	80	90	91	99	100	101

menor

ini = 0
fim = 7
meio = (0+7)/2 = 3

ini = 4
fim = 7
meio = (4+7)/2 = 5

ini = 6
fim = 7
meio = (6+7)/2 = 6

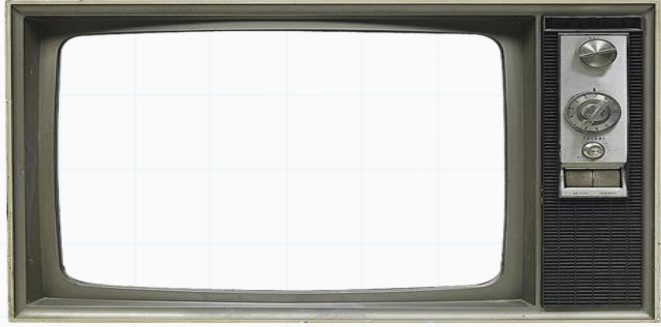
ini = 7
fim = 7
meio = (7+7)/2 = 7

ini = 8
fim = 7
meio =-1

condição de parada
início passou do fim
Não achou, retorne -1

5 movimentos

Buscas



Exercício 1): Faça uma função que receba uma lista de inteiros de tamanho N (ordenada) e um valor a ser buscado e retorne o índice da posição deste valor ou -1 (se o elemento não estiver na lista). A busca deve ser feita por busca binária.

void busca_binaria(int* v, int tam, int el)

4	10	80	90	91	99	100	101

maior

Procurando 11

meio = (ini + fim) / 2

4	10	80	90	91	99	100	101

menor

ini = 0
fim = 2
meio = (0+2)/2 = 1

fim = meio-1

4	10	80	90	91	99	100	101

maior

ini = 2
fim = 2
meio = (2+2)/2 = 2

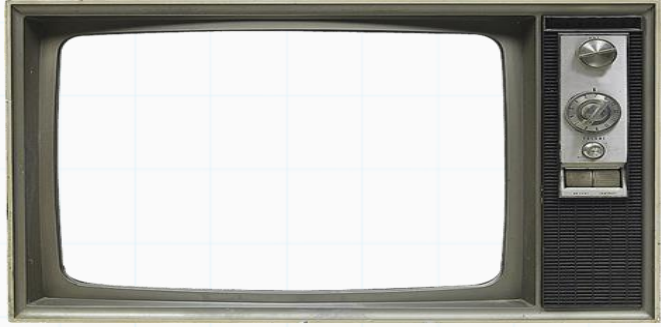
fim = meio-1

4	10	80	90	91	99	100	101

ini = 2
fim = 1
meio = -1

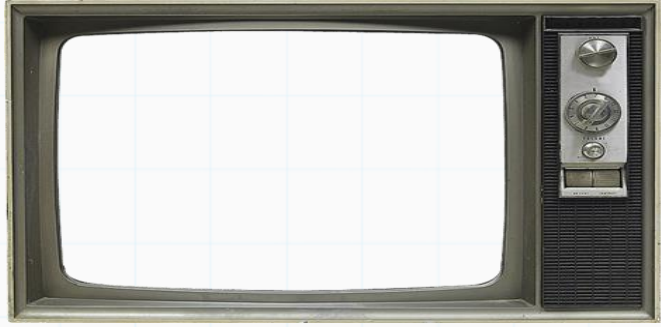
condição de parada
início passou do fim
Não achou, retorne -1

Buscas



```c
void busca_binaria(int* v, int tam, int el)
{
    int ini, fim;

    ini = 0; fim = tam-1;
    while (ini <= fim)
    {
        int meio = ini + (fim - ini) / 2;

        if (v[meio] == el)
        {
            printf("achou %d na posicao %d\n", el, meio);
            return;
        }

        if (v[meio] < el)
          ini = meio + 1;

        else
          fim = meio - 1;
    }

    printf("Nao achou\n");
    return;
}
```

Buscas

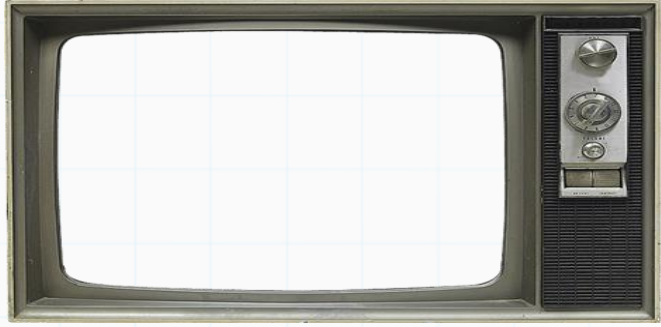


```c
void busca_binaria(int* v, int tam, int el)
{
    int ini, fim;

    ini = 0; fim = tam-1;
    while (ini <= fim)
    {
        int meio = ini + (fim - ini) / 2;

        if (v[meio] == el)
        {
            printf("achou %d na posicao %d\n", el, meio);
            return;
        }

        if (v[meio] < el)
          ini = meio + 1;

        else
          fim = meio - 1;
    }

    printf("Nao achou\n");
    return;
}
```

Número de Comparações realizadas:

melhor caso =>
pior caso =>

Buscas



```c
void busca_binaria(int* v, int tam, int el)
{
    int ini, fim;

    ini = 0; fim = tam-1;
    while (ini <= fim)
    {
        int meio = ini + (fim - ini) / 2;

        if (v[meio] == el)
        {
            printf("achou %d na posicao %d\n", el, meio);
            return;
        }

        if (v[meio] < el)
          ini = meio + 1;

        else
          fim = meio - 1;
    }

    printf("Nao achou\n");
    return;
}
```

Número de Comparações realizadas:

melhor caso => 1
pior caso =>
- A cada teste o tamanho do vetor testado cai pela metade

n -> n/2 -> n/4 -> 1

Em log n partições chegamos em 1, logo pior caso log n

Ordenação

Quicksort: Método muito eficiente e usado por diversas linguagens.

Ordenação

<u>Quicksort</u>: Método muito eficiente e usado por diversas linguagens.

10	5	13	21	45	32	41	78
0	1	2	3	4	5	6	7

menores

<u>Ideia</u>: O elemento 21 está na posição 3, mas veja que todos os 3 elementos menores que 21 (5, 10 e 13) estão antes dele. Então 21 está na sua posição correta na ordenação. Quem mais você pode ver que esta na posição correta ?

Ordenação

Quicksort: Método muito eficiente e usado por diversas linguagens.

10	5	13	21	45	32	41	78
0	1	2	3	4	5	6	7

menores

Ideia: O elemento 21 está na posição 3, mas veja que todos os 3 elementos menores que 21 (5, 10 e 13) estão antes dele. Então 21 está na sua posição correta na ordenação. Quem mais você pode ver que esta na posição correta ?

8	7	6	5	4	3	2	1
0	1	2	3	4	5	6	7

5	4	3	2	1	6	8	7
0	1	2	3	4	5	6	7

menores

PARTIÇÃO: Para descobrir a posição correta do elemento 6, basta percorremos o vetor e colocarmos quem é menor a esquerda de 6 e quem é maior a direita de 6.

Ordenação

<u>Quicksort</u>: Método muito eficiente e usado por diversas linguagens.

10	5	13	21	45	32	41	78
0	1	2	3	4	5	6	7

menores

<u>Ideia</u>: O elemento 21 está na posição 3, mas veja que todos os 3 elementos menores que 21 (5, 10 e 13) estão antes dele. Então 21 está na sua posição correta na ordenação. Quem mais você pode ver que esta na posição correta ?

8	7	6	5	4	3	2	1
0	1	2	3	4	5	6	7

5	4	3	2	1	6	8	7
0	1	2	3	4	5	6	7

menores

<u>PARTIÇÃO</u>: Para descobrir a posição correta do elemento 6, basta percorremos o vetor e colocarmos quem é menor a esquerda de 6 e quem é maior a direita de 6.

Vamos ver agora um exemplo desse processo

Ordenação

<u>Quicksort</u>: Método da partição.

3	6	4	5	1	7	2

1) Escolher elemento central (pivo) para descobrir sua posição na ordenação. Vamos pegar sempre o primeiro.

Ordenação

Quicksort: Método da partição.

3	6	4	5	1	7	2

1) Escolher elemento central (pivo) para descobrir sua posição na ordenação. Vamos pegar sempre o primeiro.
2) Andar o vetor com 2 índices, ini e fim.

Ordenação

Quicksort: Método da partição.

3	6	4	5	1	7	2

3	6	4	5	1	7	2

1) Escolher elemento central (pivo) para descobrir sua posição na ordenação. Vamos pegar sempre o primeiro.
2) Andar o vetor com 2 índices, ini e fim.
3) Percorra o vetor com ini para frente até que vet[ini] ≥ pivo.

Ordenação

Quicksort: Método da partição.

3	6	4	5	1	7	2

3	6	4	5	1	7	2

1) Escolher elemento central (pivo) para descobrir sua posição na ordenação. Vamos pegar sempre o primeiro.
2) Andar o vetor com 2 índices, ini e fim.
3) Percorra o vetor com ini para frente até que vet[ini] ≥ pivo.
4) Percorra o vetor com fim para tras até que vet[j] ≤ pivo.

Ordenação

Quicksort: Método da partição.

3	6	4	5	1	7	2

3	6	4	5	1	7	2

3	**2**	4	5	1	7	**6**

1) Escolher elemento central (pivo) para descobrir sua posição na ordenação. Vamos pegar sempre o primeiro.
2) Andar o vetor com 2 índices, ini e fim.
3) Percorra o vetor com ini para frente até que vet[ini] ≥ pivo.
4) Percorra o vetor com fim para tras até que vet[j] ≤ pivo.
5) Troca vet[i] e vet[j]

Ordenação

Quicksort: Método da partição.

3	6	4	5	1	7	2

3	6	4	5	1	7	2

3	**2**	4	5	1	7	**6**

3	**2**	4	5	1	7	**6**

1) Escolher elemento central (pivo) para descobrir sua posição na ordenação. Vamos pegar sempre o primeiro.
2) Andar o vetor com 2 índices, ini e fim.
3) Percorra o vetor com ini para frente até que vet[ini] ≥ pivo.
4) Percorra o vetor com fim para trás até que vet[fim] ≤ pivo.
5) Troca vet[ini] e vet[fim]
6) Repete até ini e fim se cruzarem

Ordenação

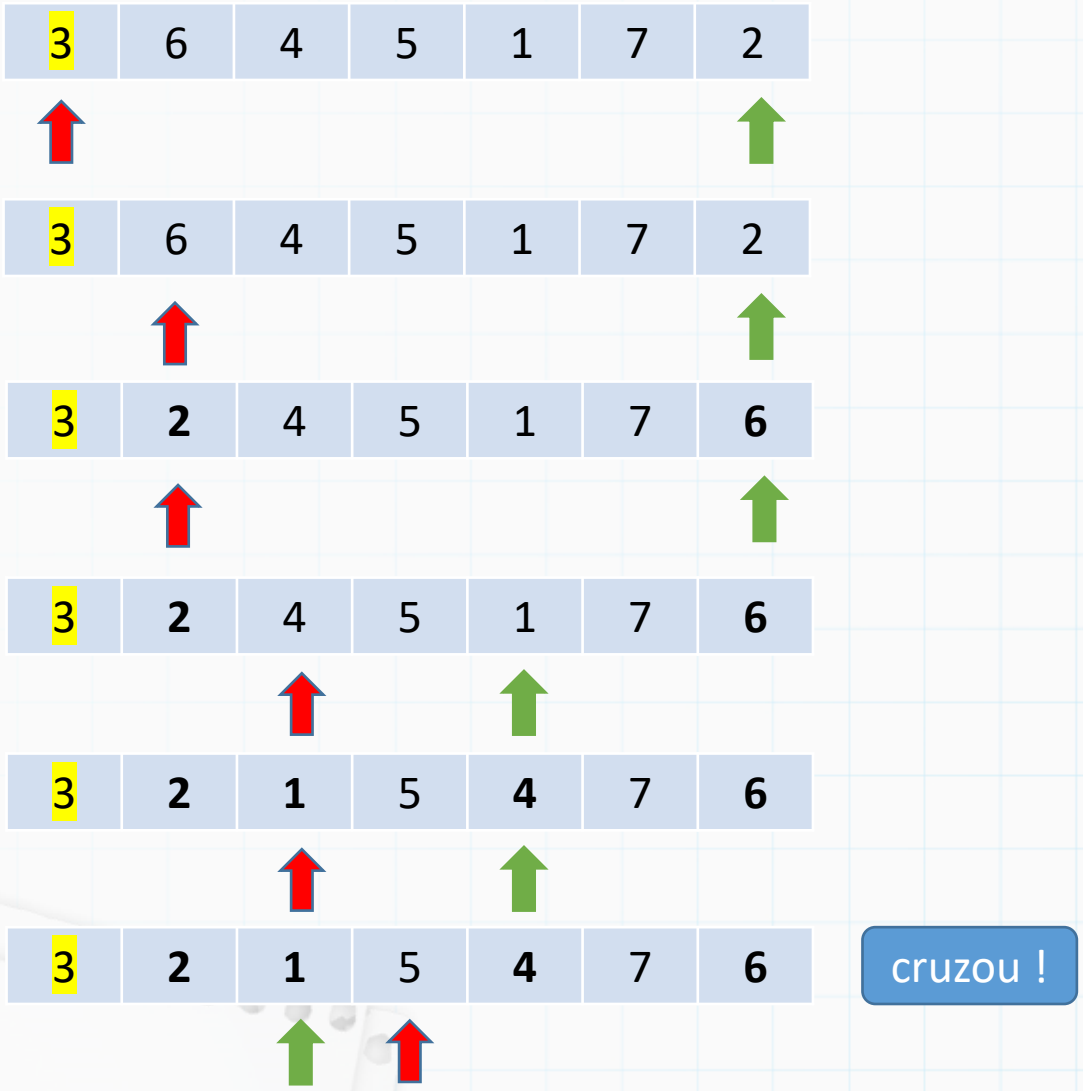
Quicksort: Método da partição.

3	6	4	5	1	7	2
3	6	4	5	1	7	2
3	**2**	4	5	1	7	**6**
3	**2**	4	5	1	7	**6**
3	**2**	**1**	5	**4**	7	**6**

1) Escolher elemento central (pivo) para descobrir sua posição na ordenação. Vamos pegar sempre o primeiro.
2) Andar o vetor com 2 índices, ini e fim.
3) Percorra o vetor com ini para frente até que vet[ini] ≥ pivo.
4) Percorra o vetor com fim para trás até que vet[fim] ≤ pivo.
5) Troca vet[ini] e vet[fim]
6) Repete até ini e fim se cruzarem

Ordenação

Quicksort: Método da partição.



cruzou !

1) Escolher elemento central (pivo) para descobrir sua posição na ordenação. Vamos pegar sempre o primeiro.
2) Andar o vetor com 2 índices, ini e fim.
3) Percorra o vetor com ini para frente até que vet[ini] ≥ pivo.
4) Percorra o vetor com fim para trás até que vet[fim] ≤ pivo.
5) Troca vet[ini] e vet[fim]
6) Repete até ini e fim se cruzarem
7) Cruzou, troca o pivô com o elemento fim, e retorna a nova posição do pivô.

Ordenação

<u>Quicksort</u>: Método da partição.

3	6	4	5	1	7	2

3	6	4	5	1	7	2

3	**2**	4	5	1	7	**6**

3	**2**	4	5	1	7	**6**

3	**2**	**1**	5	**4**	7	**6**

3	**2**	**1**	5	**4**	7	**6**

cruzou !

1	**2**	3	5	**4**	7	**6**

menores maiores

1) Escolher elemento central (pivo) para descobrir sua posição na ordenação. Vamos pegar sempre o primeiro.
2) Andar o vetor com 2 índices, ini e fim.
3) Percorra o vetor com ini para frente até que vet[ini] ≥ pivo.
4) Percorra o vetor com fim para trás até que vet[fim] ≤ pivo.
5) Troca vet[ini] e vet[fim]
6) Repete até ini e fim se cruzarem
7) Cruzou, troca o pivô com o elemento fim, e retorna a nova posição do pivô.

Ordenação

<u>Quicksort</u>: Método da partição.

3	6	4	5	1	7	2

3	6	4	5	1	7	2

3	**2**	4	5	1	7	**6**

3	**2**	4	5	1	7	**6**

3	**2**	**1**	5	**4**	7	**6**

3	**2**	**1**	5	**4**	7	**6**

1	**2**	3	5	**4**	7	**6**

menores maiores

2) Exercício: Faça uma função que implemente a varredura do Quicksort.

int partição(int* v, int ini, int fim)

cruzou !

1) Escolher elemento central (pivo) para descobrir sua posição na ordenação. Vamos pegar sempre o primeiro.
2) Andar o vetor com 2 índices, ini e fim.
3) Percorra o vetor com ini para frente até que vet[ini] ≥ pivo.
4) Percorra o vetor com fim para trás até que vet[fim] ≤ pivo.
5) Troca vet[ini] e vet[fim]
6) Repete até ini e fim se cruzarem
7) Cruzou, troca o pivô com o elemento fim, e retorna a nova posição do pivô.

Ordenação

```
int particao(int* v, int ini, int fim)
{
    int pivo = v[ini];
    int i    = ini;
    int j    = fim;

    while (i < j)
    {
        while (v[i] <= pivo && i <= fim - 1)
            i++;

        while (v[j] > pivo && j >= ini + 1)
            j--;

        if (i < j)
        {
            int temp = v[i];
            v[i]     = v[j];
            v[j]     = temp;
        }
    }
    int temp = v[ini];
    v[ini]   = v[j];
    v[j]     = temp;
    return j;
}
```

Quicksort: Método da partição.

Número de Comparações realizadas:

melhor caso =>
pior caso =>

Ordenação

```
int particao(int* v, int ini, int fim)
{
    int pivo = v[ini];
    int i    = ini;
    int j    = fim;

    while (i < j)
    {
        while (v[i] <= pivo && i <= fim - 1)
            i++;

        while (v[j] > pivo && j >= ini + 1)
            j--;

        if (i < j)
        {
            int temp = v[i];
            v[i]     = v[j];
            v[j]     = temp;
        }
    }
    int temp = v[ini];
    v[ini]   = v[j];
    v[j]     = temp;
    return j;
}
```

Quicksort: Método da partição.

Número de Comparações realizadas:

melhor caso => n
pior caso => n

Ordenação

Quicksort: Depois da partição apenas um elemento está ordenado, precisamos repetir o processo de partição agora para os segmentos de vetores menores e maiores

3	6	4	5	1	7	2

partição

1	**2**	3	5	**4**	7	**6**

menores maiores

Ordenação

Quicksort: Depois da partição apenas um elemento está ordenado, precisamos repetir o processo de partição agora para os segmentos de vetores menores e maiores

3	6	4	5	1	7	2

partição

1	**2**	3	5	**4**	7	**6**

menores

partição

maiores

partição

Vamos chamar recursivamente

Exercício 3) Escreva uma função de ordenação do quicksort que receba vetor de tamanho n.
Lembre-se de usar a função de partição int partição(int* v, int ini, int fim).

A função de partição retorna o índice do pivo

Ordenação

```
void quickSort(int* v, int ini, int fim)
{
    if (ini < fim)
    {
        int pivo_ind = particao(v, ini, fim);

        quickSort(v, ini, pivo_ind - 1);
        quickSort(v, pivo_ind + 1, fim);
    }
}
```

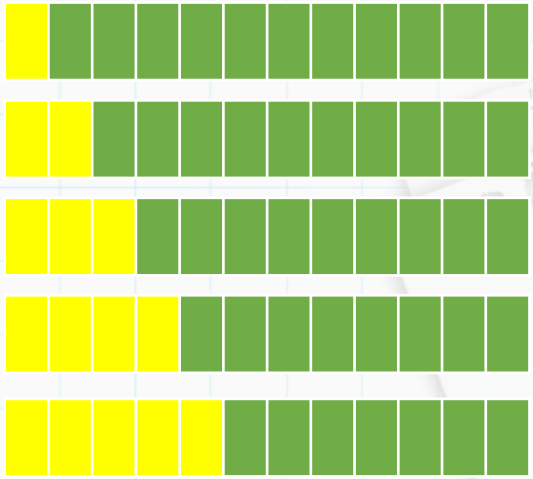
Quicksort:

Ordenação

```
void quickSort(int* v, int ini, int fim)
{
    if (ini < fim)
    {
        int pivo_ind = particao(v, ini, fim);

        quickSort(v, ini, pivo_ind - 1);
        quickSort(v, pivo_ind + 1, fim);
    }
}
```

Quicksort:

partição desbalanceada



Número de Comparações realizadas:

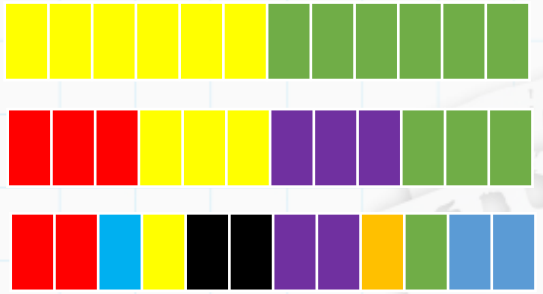
pior caso => partição desbalanceada => (n-1) + (n-2) +... => n^2

Ordenação

```
void quickSort(int* v, int ini, int fim)
{
    if (ini < fim)
    {
        int pivo_ind = particao(v, ini, fim);

        quickSort(v, ini, pivo_ind - 1);
        quickSort(v, pivo_ind + 1, fim);
    }
}
```

Quicksort:

partição desbalanceada



Número de Comparações realizadas:

pior caso => partição desbalanceada => (n-1) + (n-2) +... => n^2

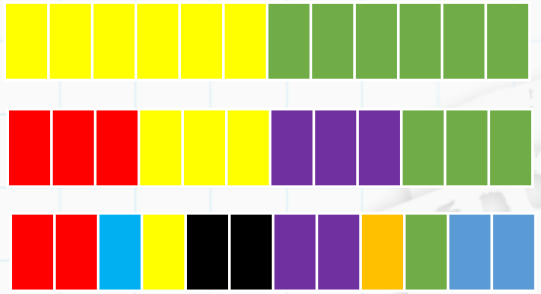
melhor caso => partição balanceada => nlog n

Ordenação

```
void quickSort(int* v, int ini, int fim)
{
    if (ini < fim)
    {
        int pivo_ind = particao(v, ini, fim);

        quickSort(v, ini, pivo_ind - 1);
        quickSort(v, pivo_ind + 1, fim);
    }
}
```

Quicksort:

partição desbalanceada



Número de Comparações realizadas:

pior caso => partição desbalanceada => (n-1) + (n-2) +... => n^2

melhor caso => partição balanceada => nlog n

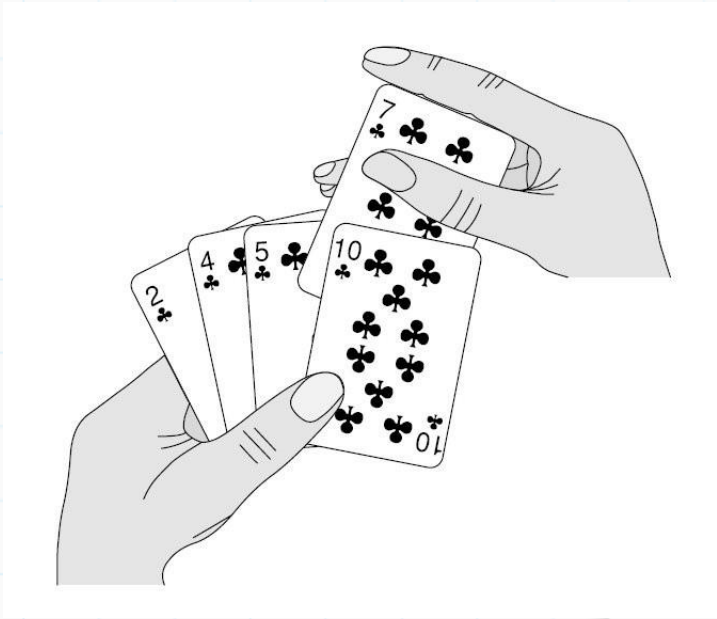
caso médio => n(logn)

calculo do caso médio foge do escopo da disciplina

Ordenação

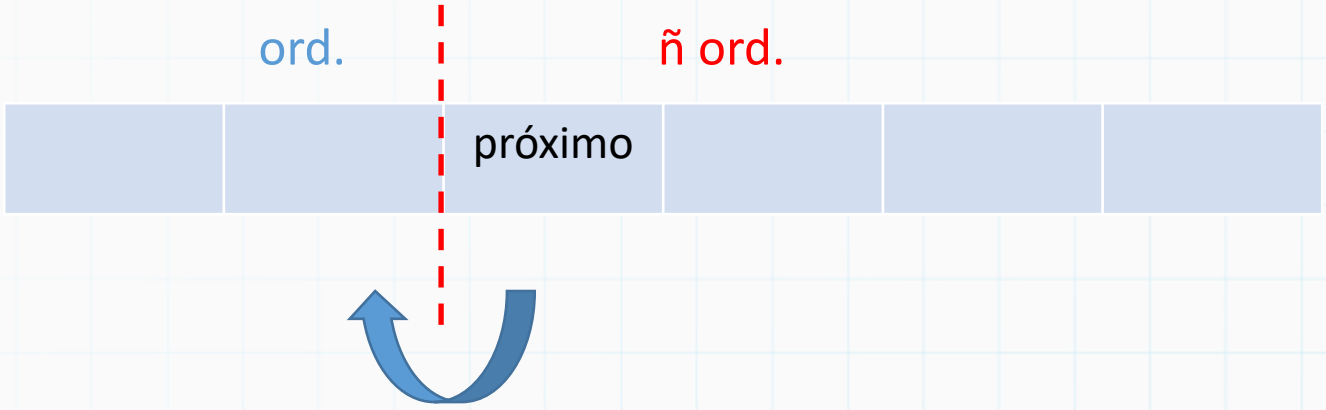
Ordenação por inserção:.

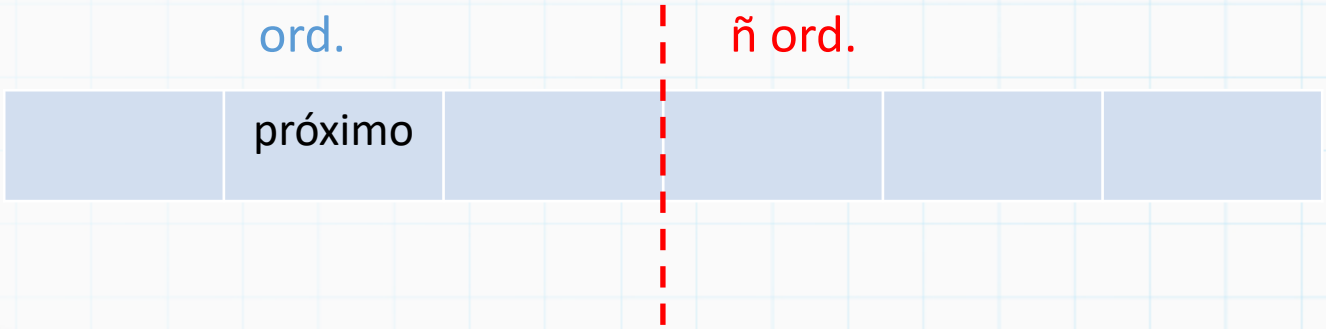
Semelhante ao método da seleção

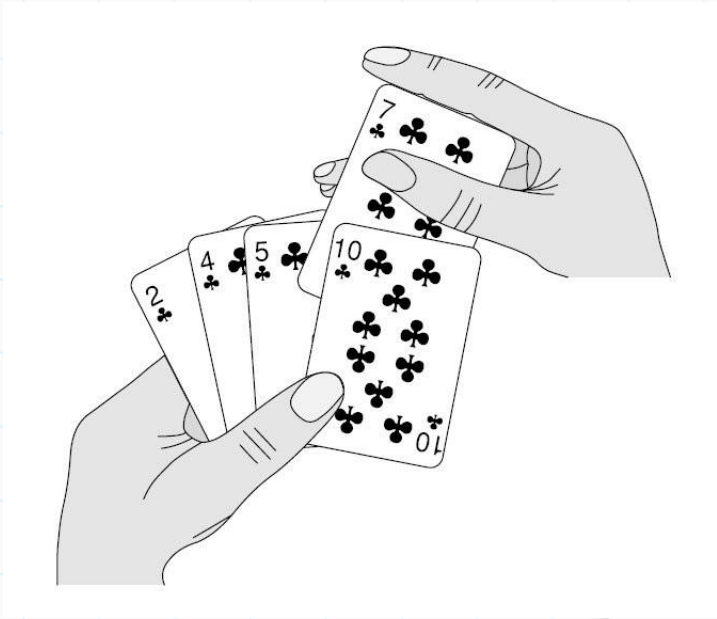


Ordenação

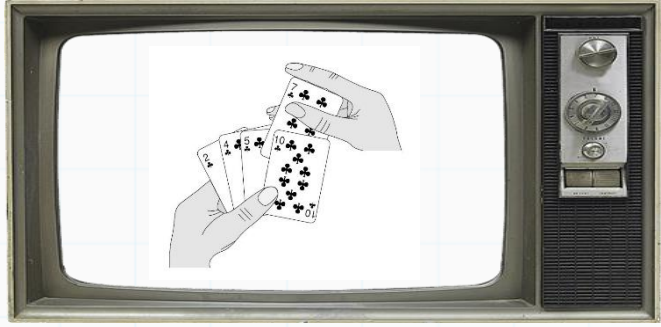
Ordenação por inserção: a lista está dividida em parte ordenada e não ordenada, a cada iteração, encontramos a posição do próximo elemento não ordenado na parte ordenada.







Ordenação

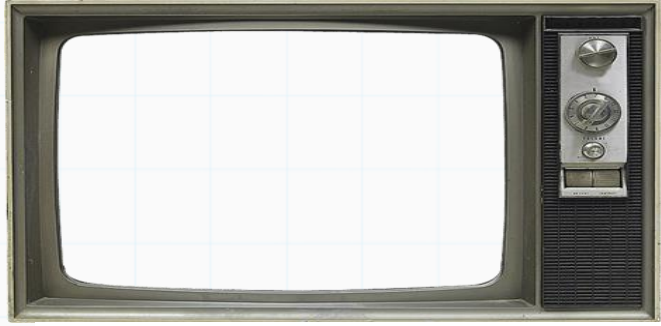


Ordenação por inserção: a lista está dividida em parte ordenada e não ordenada, a cada iteração, encontramos a posição do próximo elemento não ordenado na parte ordenada.

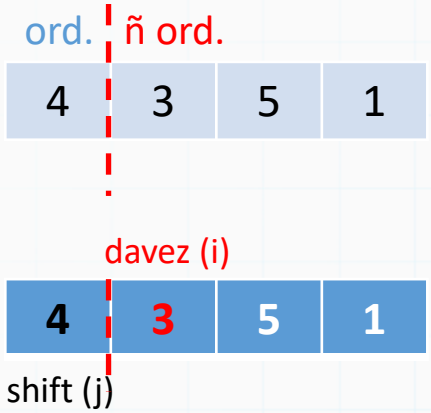
ord. | ñ ord.

4	3	5	1

Ordenação

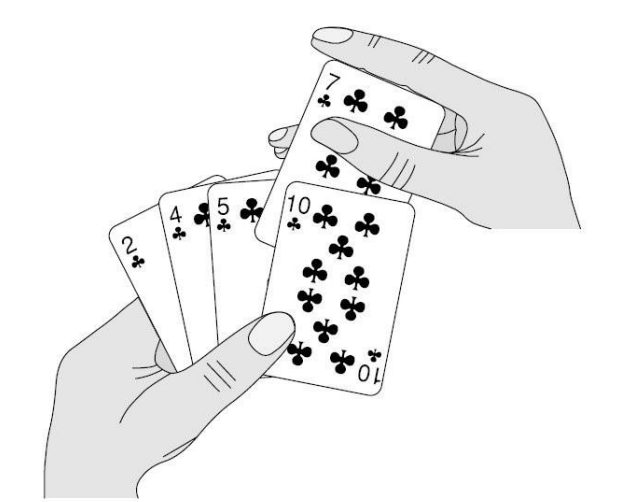


Ordenação por inserção: a lista está dividida em parte ordenada e não ordenada, a cada iteração, encontramos a posição do próximo elemento não ordenado na parte ordenada.

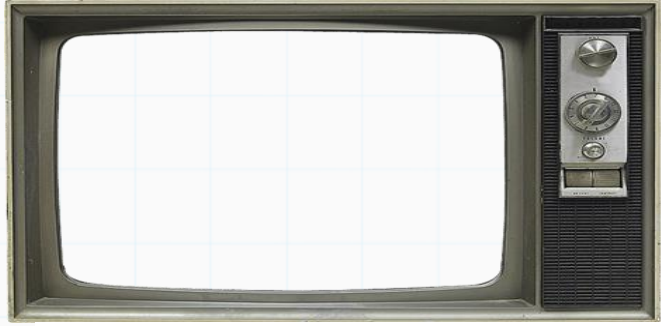


percorre a parte ordenada de traz para frente dando shift nos elementos até achar a posição do 3

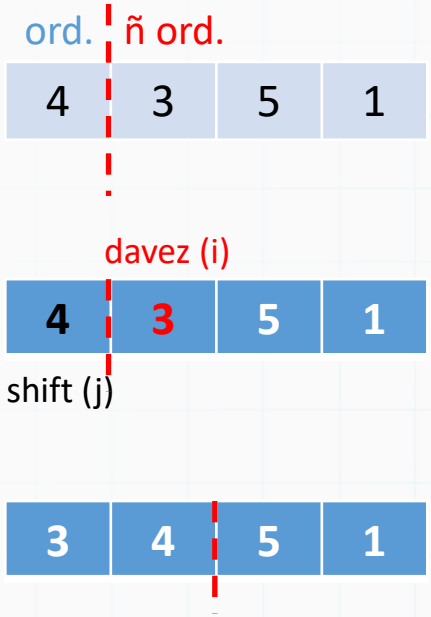
empurra 4 para dar espaço para o 3



Ordenação

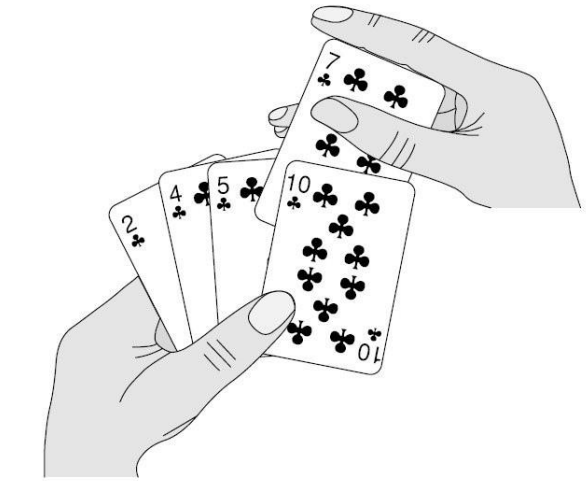


Ordenação por inserção: a lista está dividida em parte ordenada e não ordenada, a cada iteração, encontramos a posição do próximo elemento não ordenado na parte ordenada.

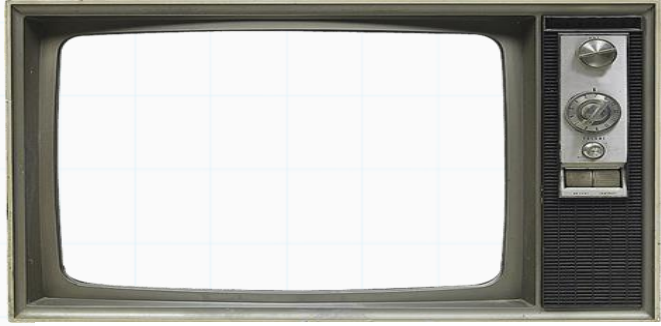


percorre a parte ordenada de traz para frente dando shift nos elementos até achar a posição do 3

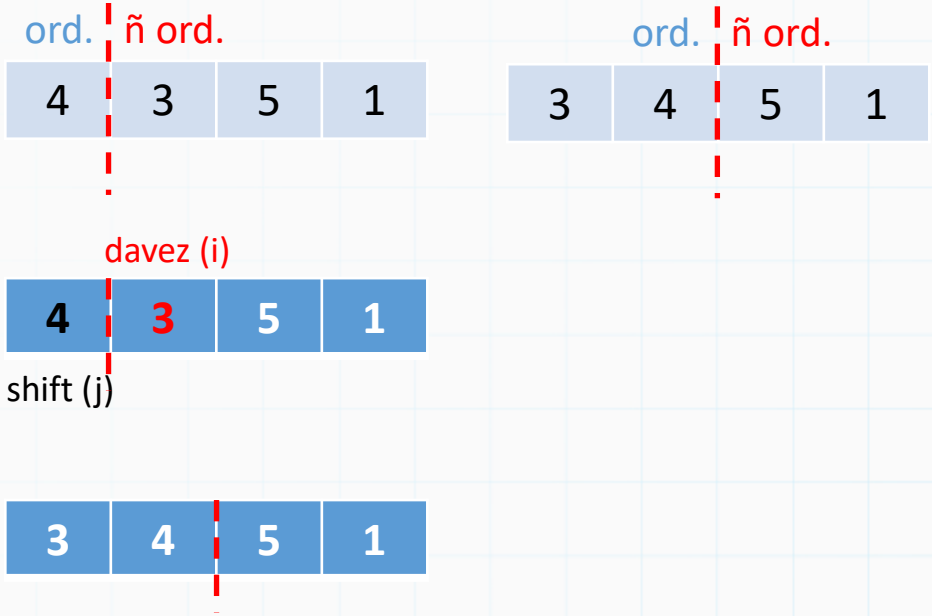
empurra 4 para dar espaço para o 3

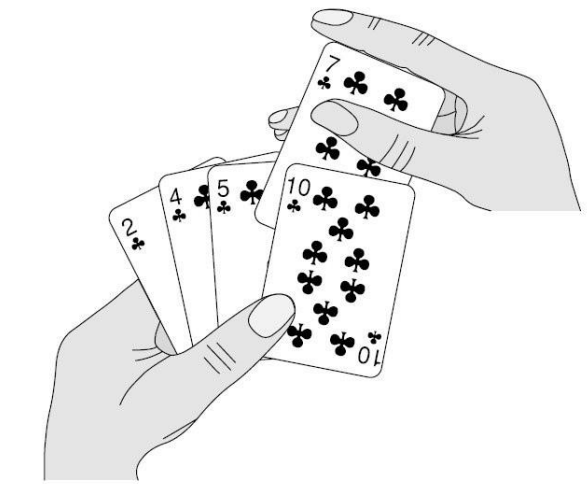


Ordenação

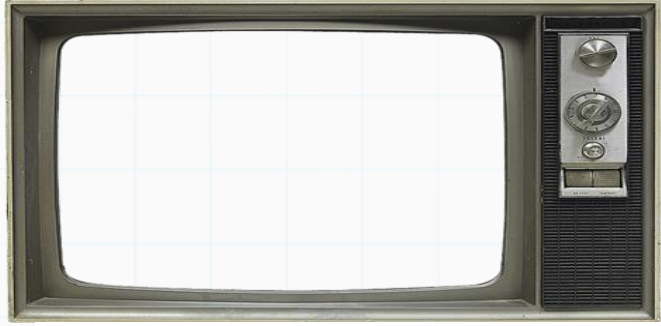


Ordenação por inserção: a lista está dividida em parte ordenada e não ordenada, a cada iteração, encontramos a posição do próximo elemento não ordenado na parte ordenada.

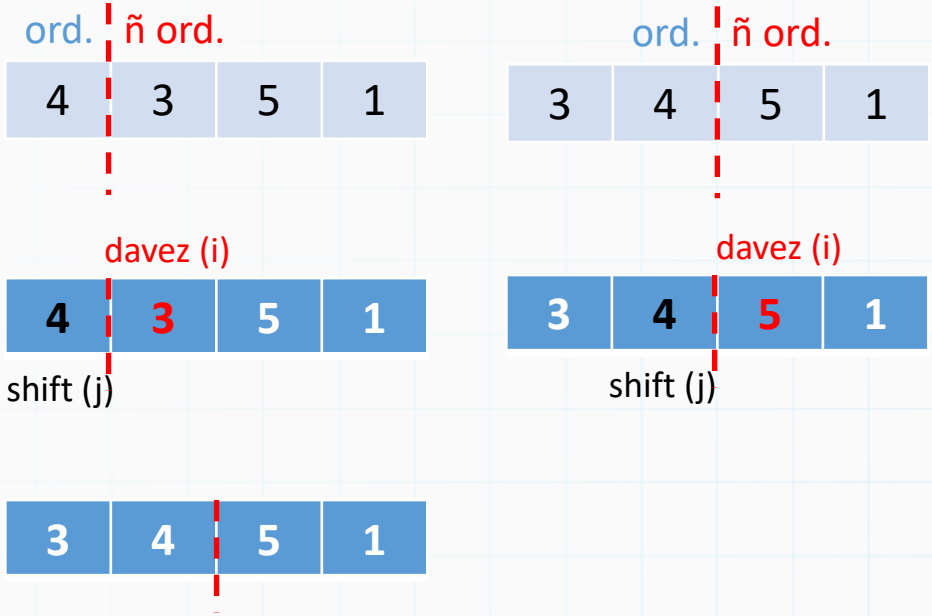




Ordenação

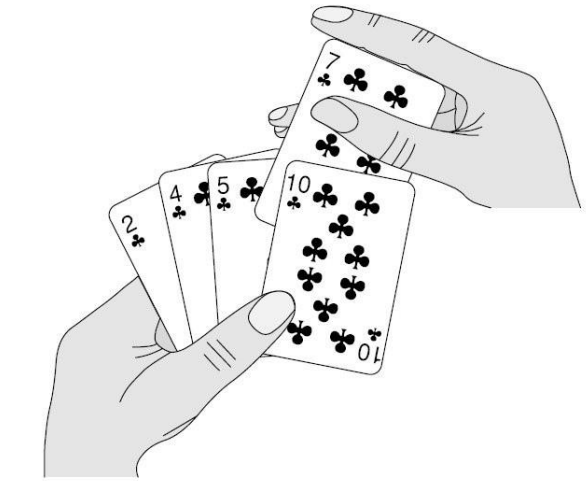


Ordenação por inserção: a lista está dividida em parte ordenada e não ordenada, a cada iteração, encontramos a posição do próximo elemento não ordenado na parte ordenada.

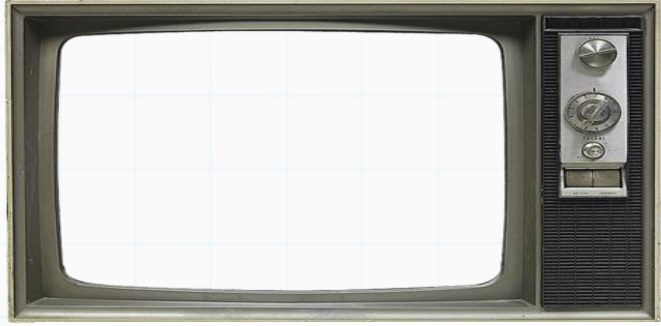


percorre a parte ordenada de traz para frente dando shift nos elementos até achar a posição do 5

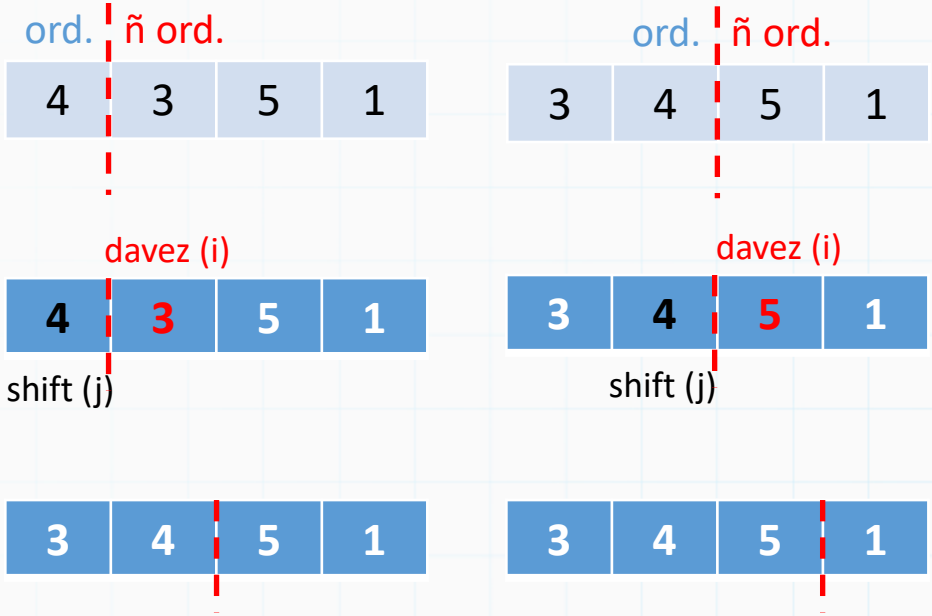
5 já é maior que 4, pode parar



Ordenação

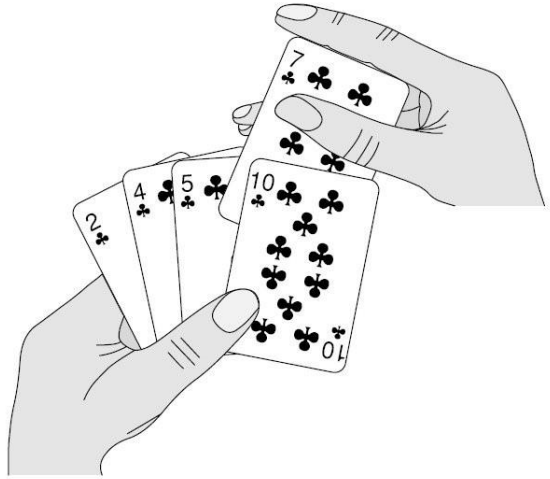


Ordenação por inserção: a lista está dividida em parte ordenada e não ordenada, a cada iteração, encontramos a posição do próximo elemento não ordenado na parte ordenada.

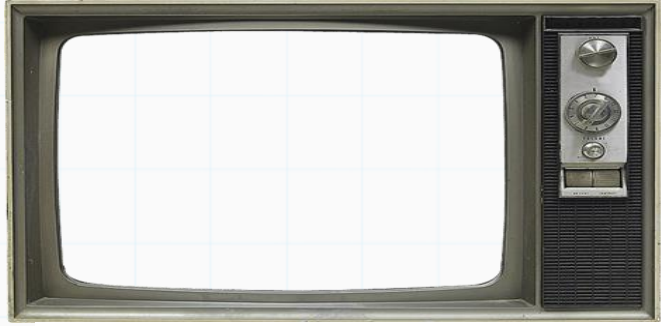


percorre a parte ordenada de traz para frente dando shift nos elementos até achar a posição do 5

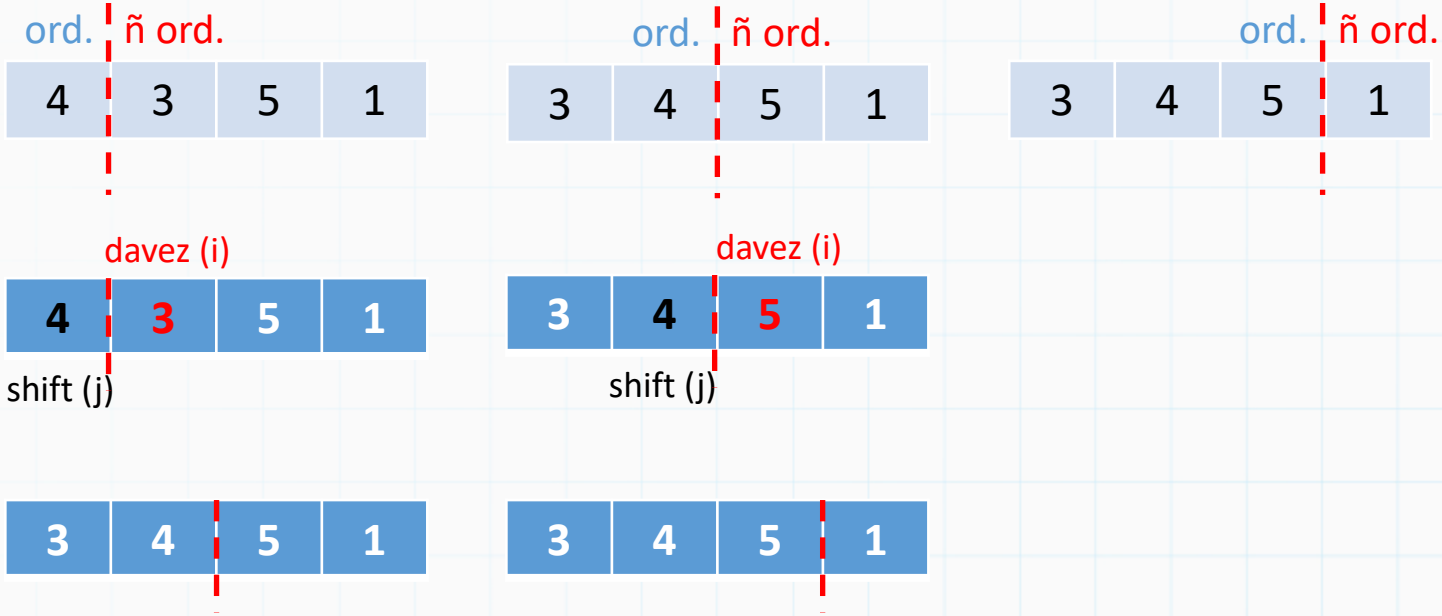
5 já é maior que 4, pode parar

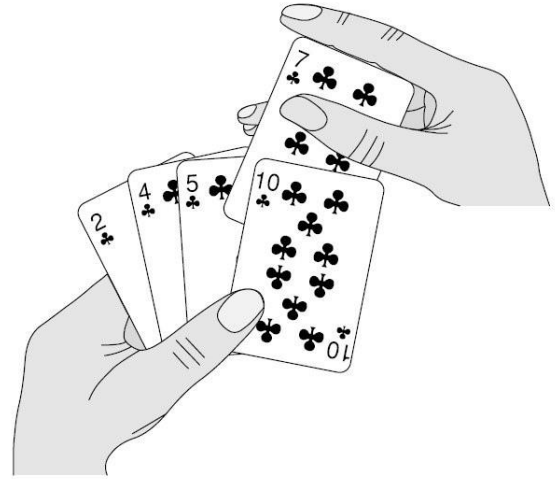


Ordenação

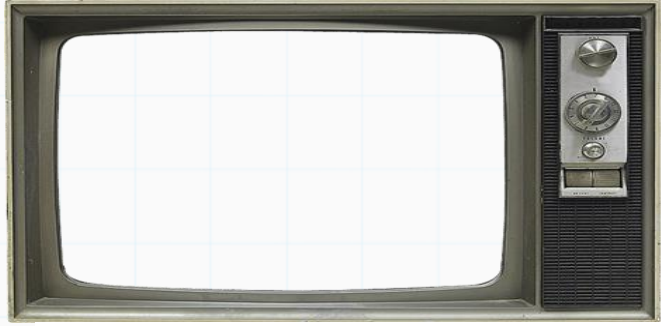


Ordenação por inserção: a lista está dividida em parte ordenada e não ordenada, a cada iteração, encontramos a posição do próximo elemento não ordenado na parte ordenada.

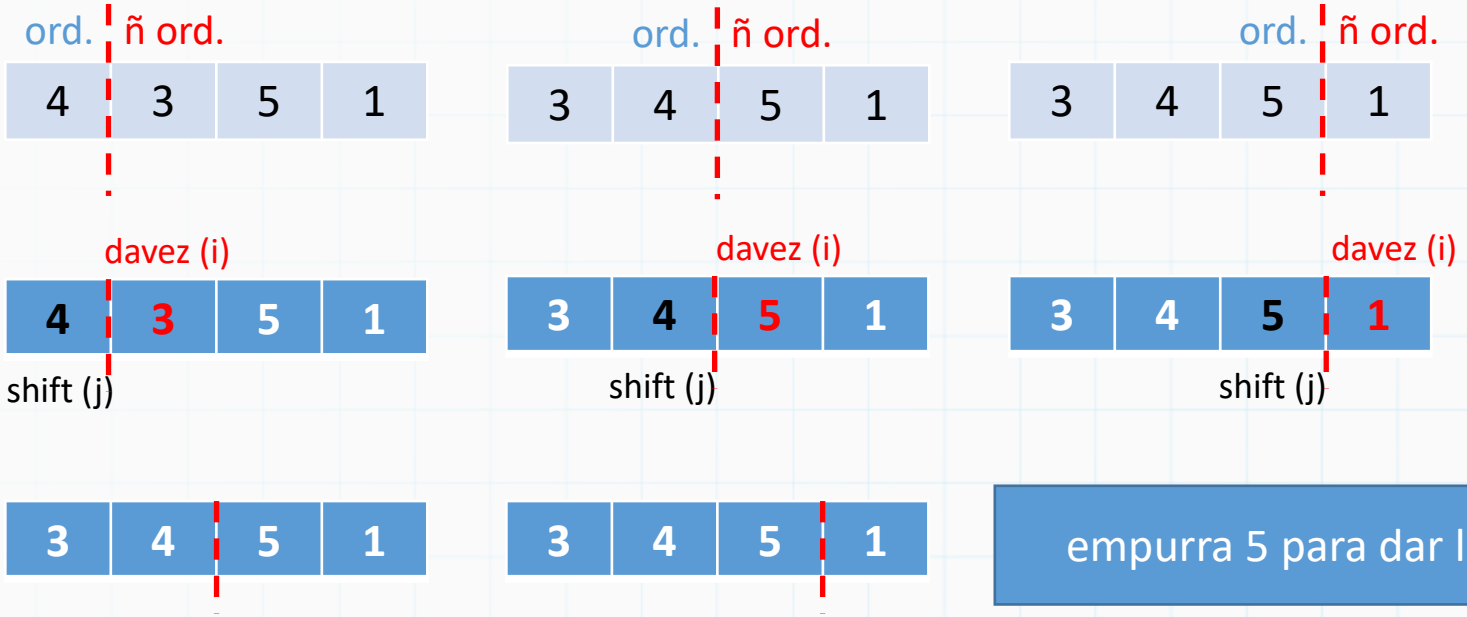


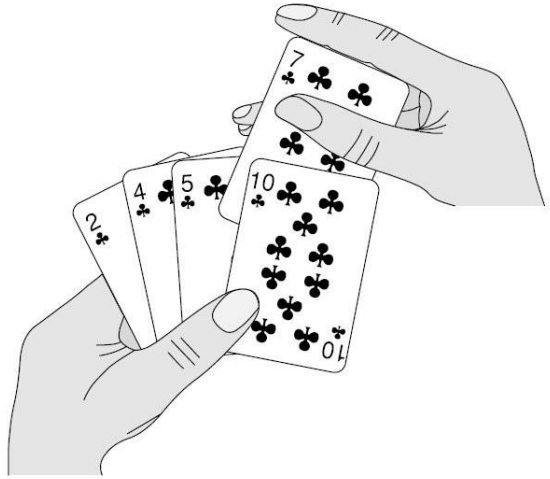


Ordenação

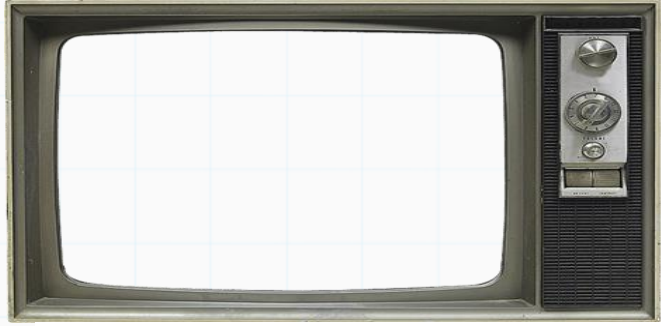


Ordenação por inserção: a lista está dividida em parte ordenada e não ordenada, a cada iteração, encontramos a posição do próximo elemento não ordenado na parte ordenada.

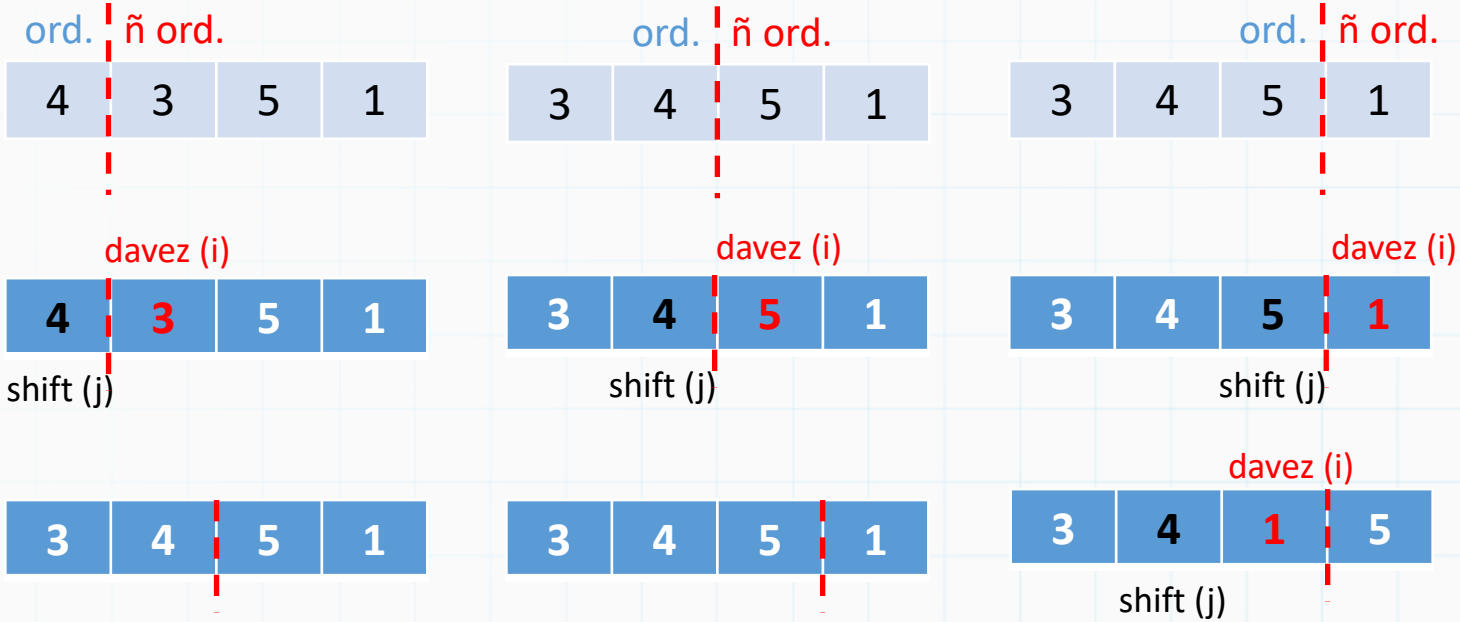


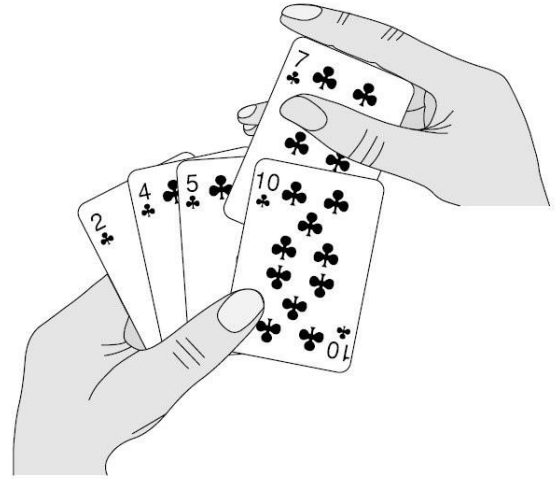


Ordenação



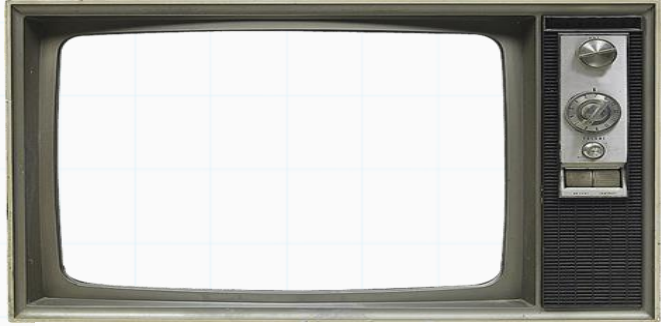
Ordenação por inserção: a lista está dividida em parte ordenada e não ordenada, a cada iteração, encontramos a posição do próximo elemento não ordenado na parte ordenada.



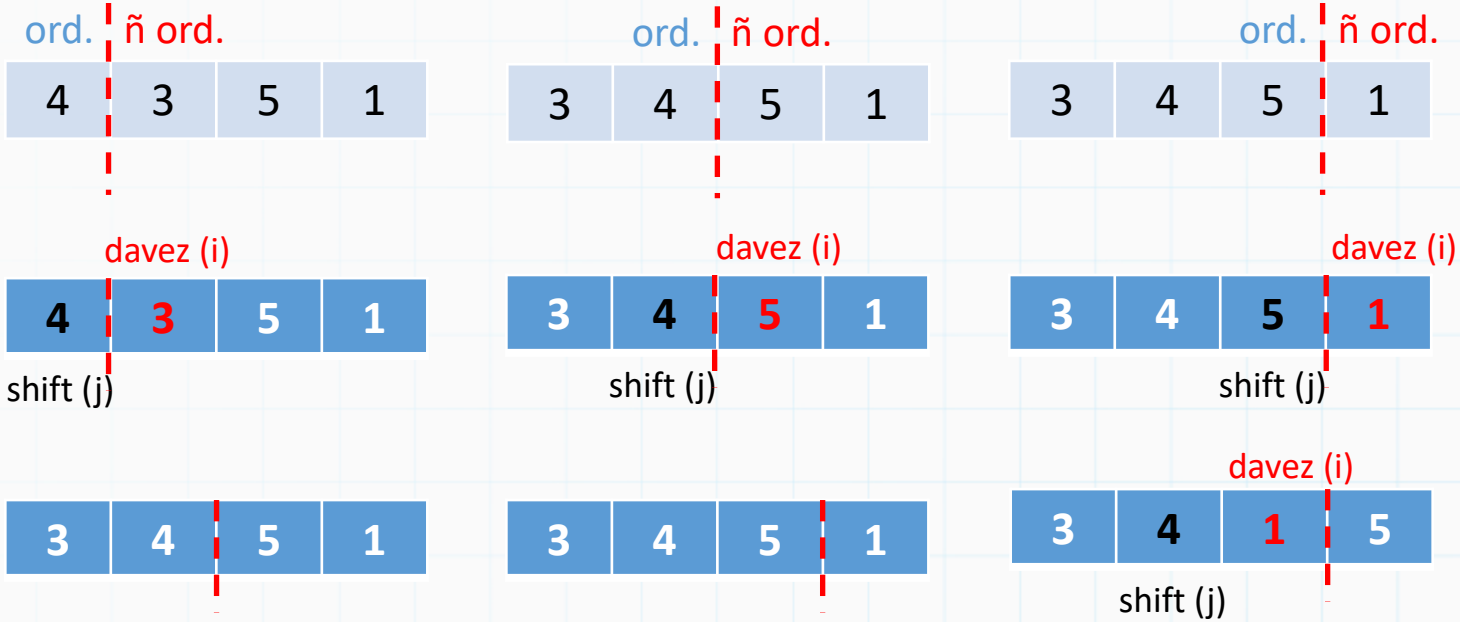


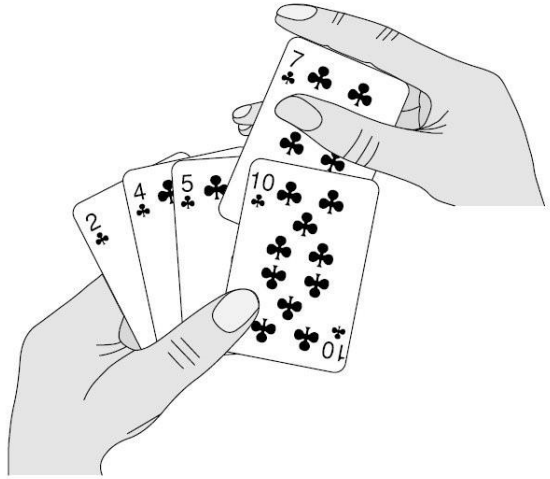
empurra 4 para dar lugar ao 1

Ordenação



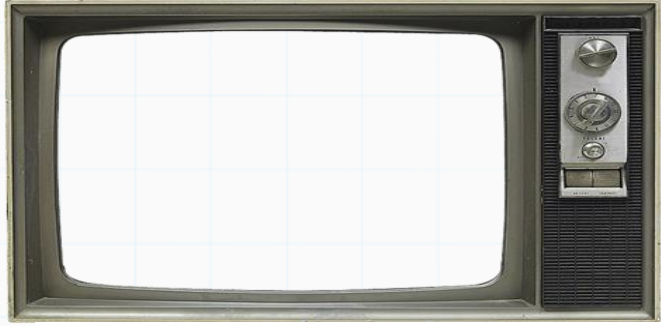
Ordenação por inserção: a lista está dividida em parte ordenada e não ordenada, a cada iteração, encontramos a posição do próximo elemento não ordenado na parte ordenada.



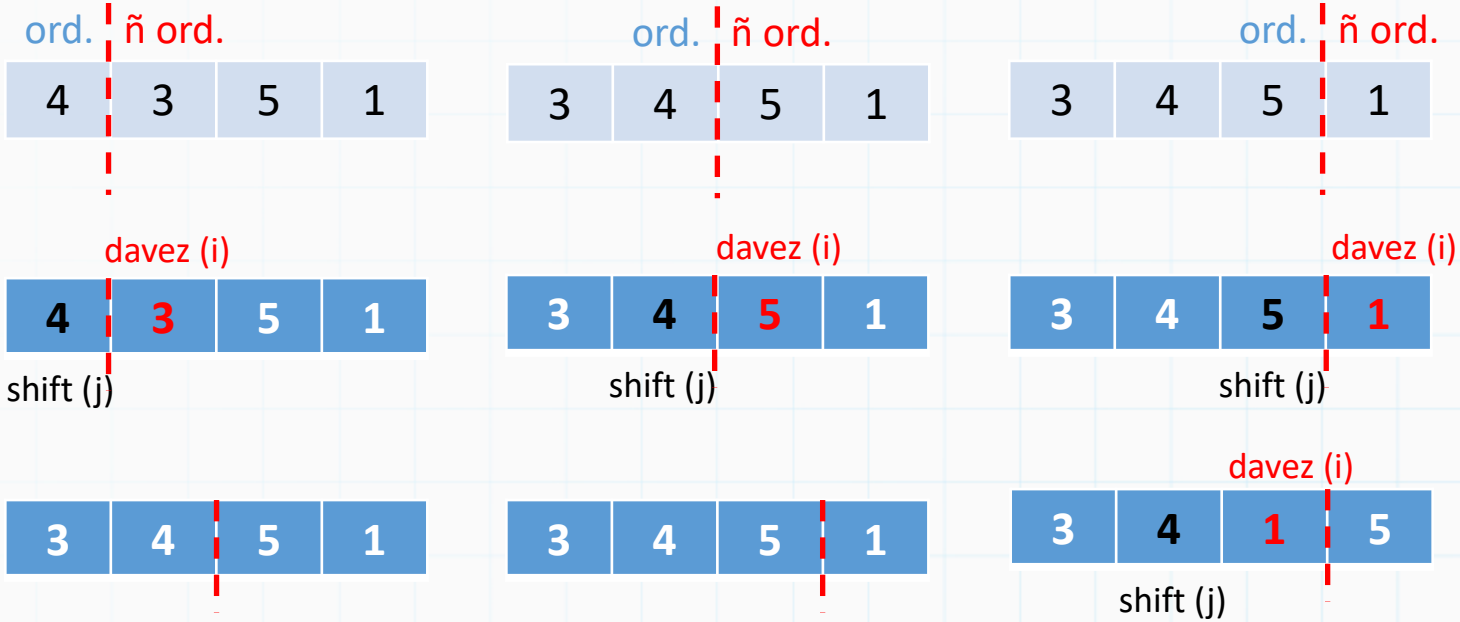


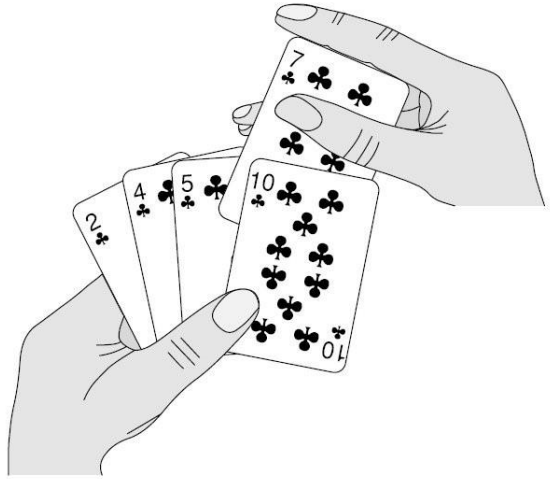
empurra 3 para dar lugar ao 1

Ordenação

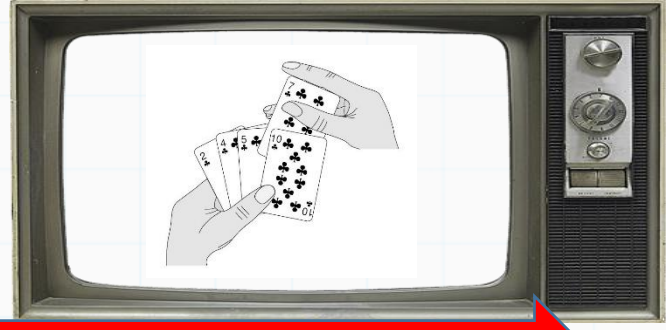


Ordenação por inserção: a lista está dividida em parte ordenada e não ordenada, a cada iteração, encontramos a posição do próximo elemento não ordenado na parte ordenada.





Ordenação

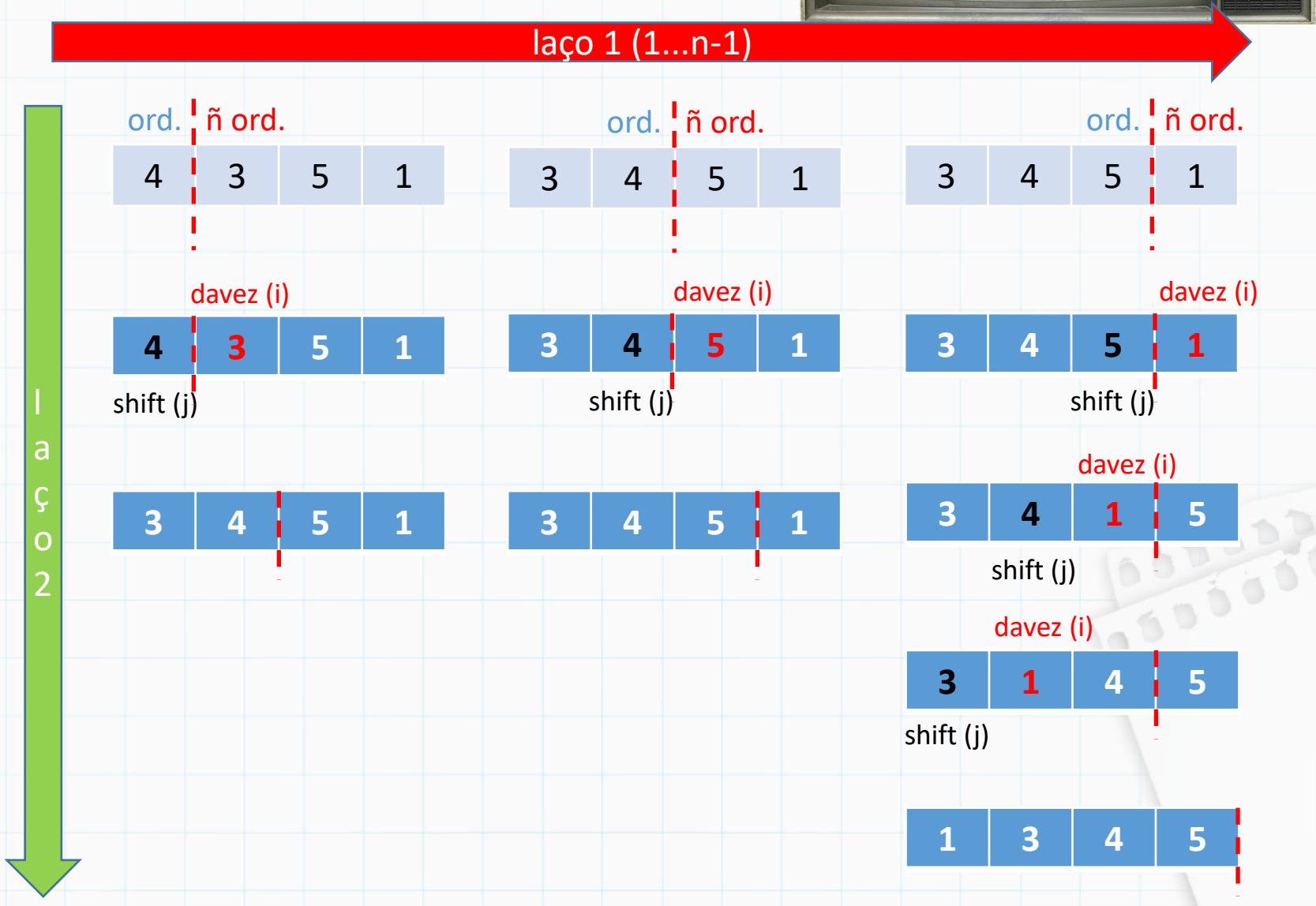


Ordenação por inserção: a lista está dividida em parte ordenada e não ordenada, a cada iteração, encontramos a posição do próximo elemento não ordenado na parte ordenada.

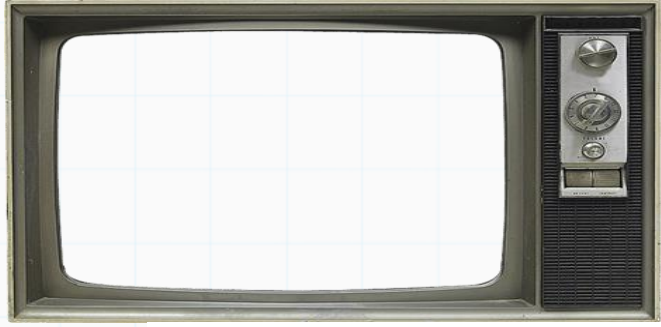
laço 2 (condicional)
com 2 condições de parada:

1) chegou no início
2) davez é maior

Exercício 4) Faça uma função que receba um vetor v de tamanho n e ordene por inserção:

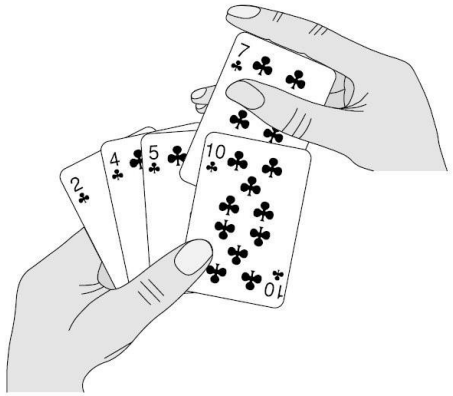


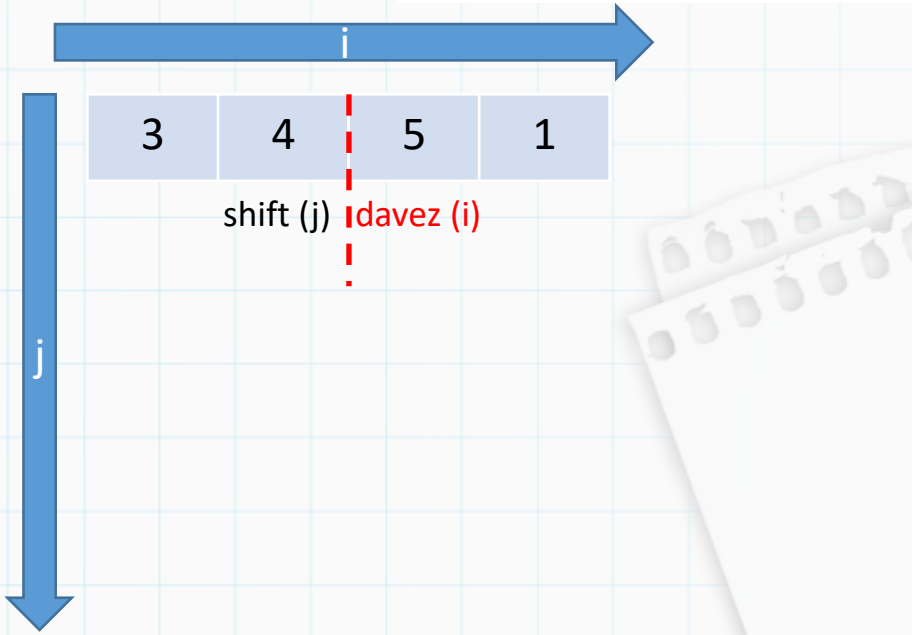
Ordenação



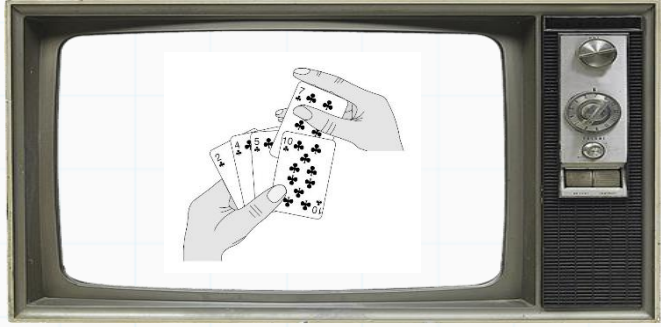
Ordenação por inserção: a lista está dividida em parte ordenada e não ordenada, a cada iteração, encontramos a posição do próximo elemento não ordenado na parte ordenada.
Vamos fazer ?

```
void ordena_insercao(int* v, int tam)
{
    int i,j, davez;

    for (i=1; i<tam; i++)
    {
        davez = v[i];
        j     = i-1;

        while(j>=0 && davez < v[j])
        {
            v[j+1] = v[j];
            j--;
        }
        v[j+1] = davez;
    }
    return;
}
```





Ordenação

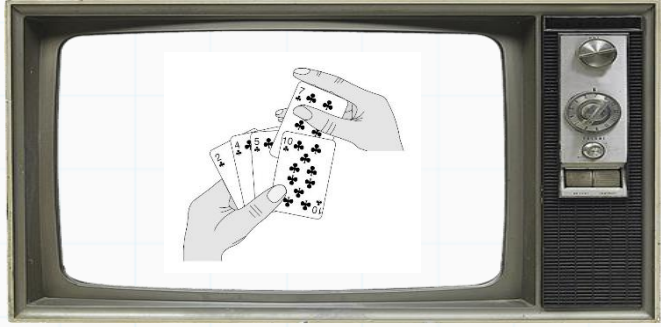


Ordenação por inserção: Eficiência

```
void ordena_insercao(int* v, int tam)
{
    int i,j, davez;

    for (i=1; i<tam; i++)
    {
        davez = v[i];
        j     = i-1;

        while(j>=0 && davez < v[j])
        {
            v[j+1] = v[j];
            j--;
        }
        v[j+1] = davez;
    }
    return;
}
```

Número de Comparações realizadas:

Ordenação



Ordenação por inserção: Eficiência

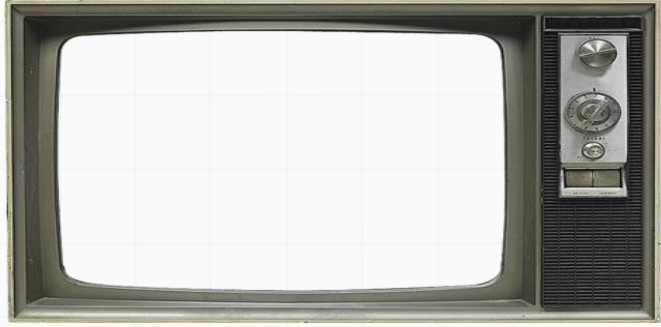
```
void ordena_insercao(int* v, int tam)
{
    int i,j, davez;

    for (i=1; i<tam; i++)
    {
        davez = v[i];
        j     = i-1;

        while(j>=0 && davez < v[j])
        {
            v[j+1] = v[j];
            j--;
        }
        v[j+1] = davez;
    }
    return;
}
```

Número de Comparações realizadas:

$1 + 2 + 3 \ldots = (n^2 - n)/2 \Rightarrow n^2$

Mesma que Bolha e Inserção

Buscas



Busca Linear: Eficiência

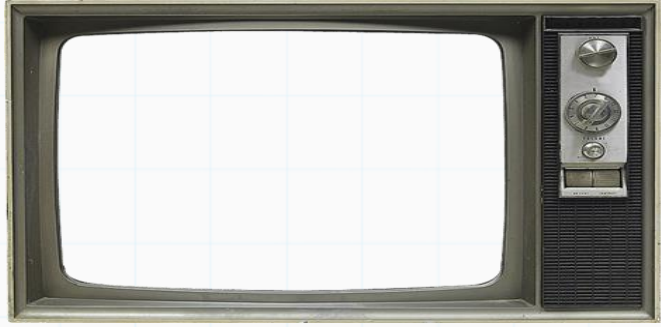
```
void busca_vetor(int* v, int tam, int el)
{
    for (int i=0; i<tam; i++)
    {
        if (v[i] == el)
        {
            printf("elemento %d na posicao %d\n\n",el, i);
            return;
        }
    }

    return;
}
```

Número de Comparações realizadas:

pior caso =>

melhor caso =>

Buscas



Busca Linear: Eficiência

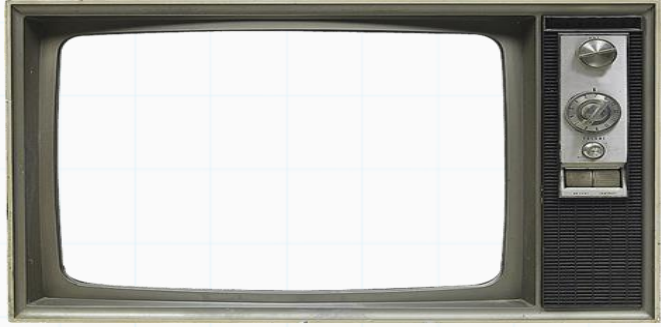
```
void busca_vetor(int* v, int tam, int el)
{
    for (int i=0; i<tam; i++)
    {
        if (v[i] == el)
        {
            printf("elemento %d na posicao %d\n\n",el, i);
            return;
        }
    }

    return;
}
```

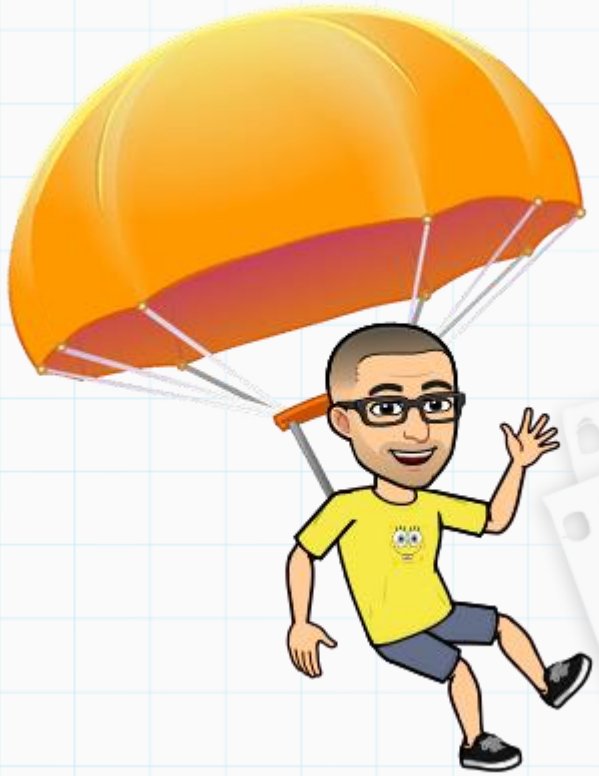
Número de Comparações realizadas:

pior caso => n

melhor caso => 1

Até a próxima





Slides baseados no curso de Aline Nascimento