

Ordenação

Prof. Paulo Henrique Pisani

novembro/2021

Ordenação

- Ordenação é o processo de **rearranjar uma sequência de elementos em ordem ascendente ou descendente**, de acordo com a chave de cada elemento;
- Um dos principais objetivos de realizar a ordenação é facilitar a recuperação dos elementos por sua chave.

Algoritmos de ordenação

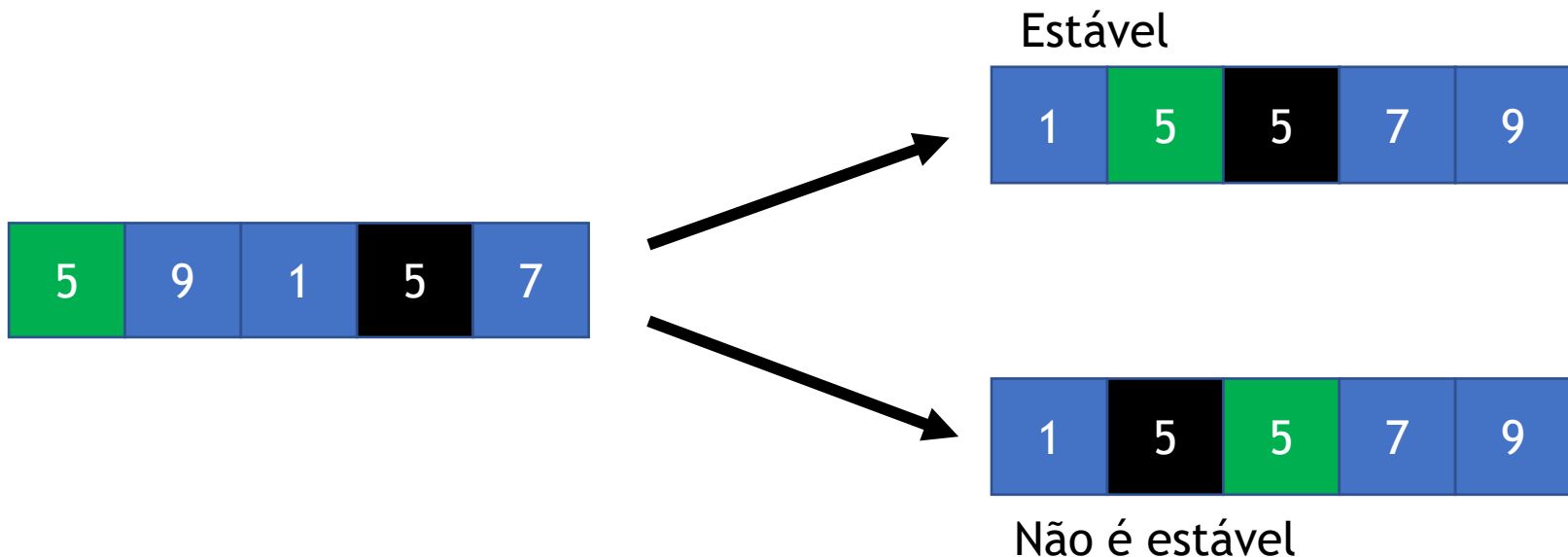
- Há dois tipos de ordenação:
 - **Interna:** todas as chaves estão na memória principal;
 - **Externa:** grande parte das chaves estão em memória externa.

Algoritmos de ordenação

- Os algoritmos de ordenação podem ser divididos entre os **baseados em comparação**:
 - Bubble sort;
 - Selection sort;
 - Insertion sort;
 - Merge sort;
 - Quick sort;
 - Heap sort.
- E os **baseados em distribuição**:
 - Count sort;
 - Radix sort;
 - Bucket sort.

Algoritmos de ordenação

- Os algoritmos de ordenação podem ser classificados como estável ou não estável;
- Um algoritmo de ordenação é **estável** se a ordem relativa de elementos com chaves iguais é mantida após a ordenação.



Ordenação

- Faremos a **ordenação de elementos em vetores/arrays** nesta aula, mas podemos aplicar os algoritmos de ordenação para outras estruturas (e.g. listas ligadas).

6	9	40	3	5	16
---	---	----	---	---	----



Ordenação crescente

3	5	6	9	16	40
---	---	---	---	----	----

Selection sort

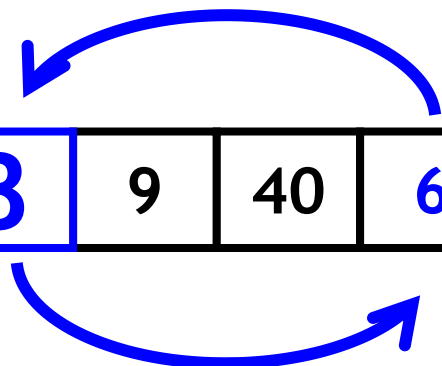
Selection sort

- Ideia geral:
 - Encontra menor elemento:

6	9	40	3	5	16
---	---	----	---	---	----

- Troca menor elemento com o primeiro elemento do vetor:

3	9	40	6	5	16
---	---	----	---	---	----



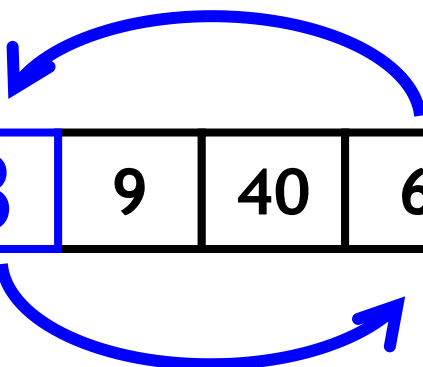
Selection sort

- Ideia geral:
 - Encontra menor elemento:

6	9	40	3	5	16
---	---	----	---	---	----

- Troca menor elemento com o primeiro elemento do vetor:

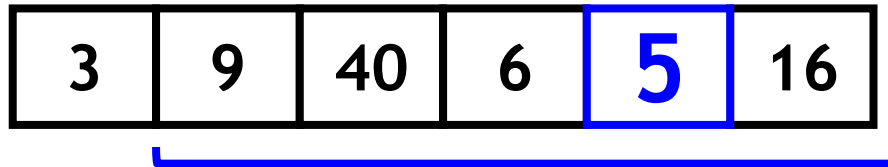
3	9	40	6	5	16
---	---	----	---	---	----



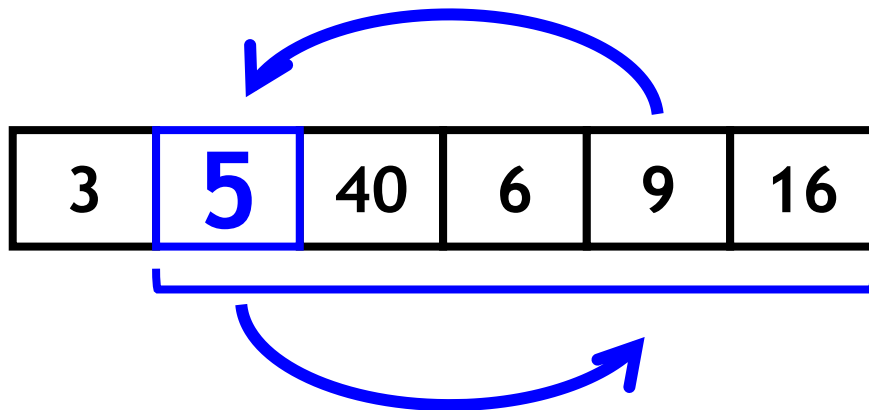
Agora o primeiro elemento já está em ordem. Então repetimos o processo para o restante do vetor.

Selection sort

- Ideia geral:
 - Encontra menor elemento no restante do vetor:

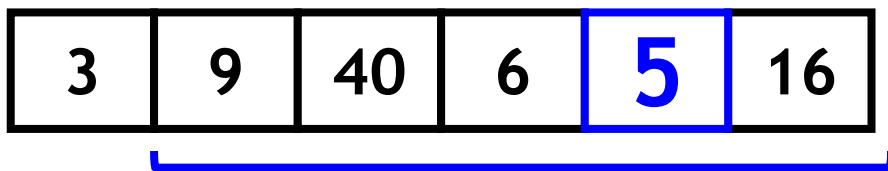


- Troca segundo elemento com o menor elemento do subvetor:

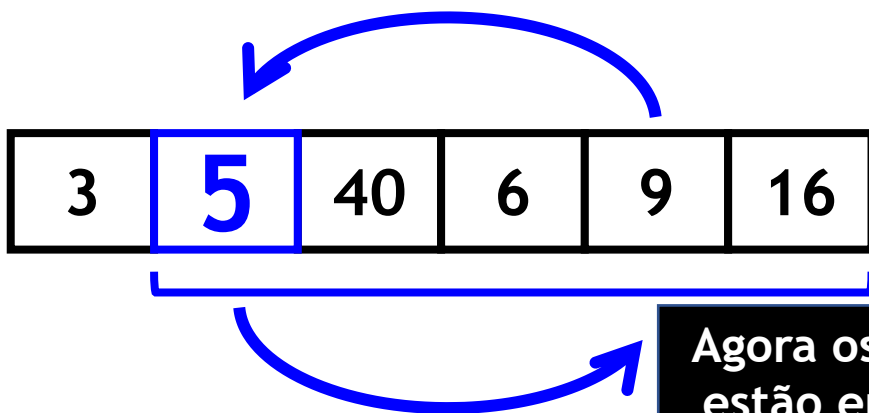


Selection sort

- Ideia geral:
 - Encontra menor elemento no restante do vetor:



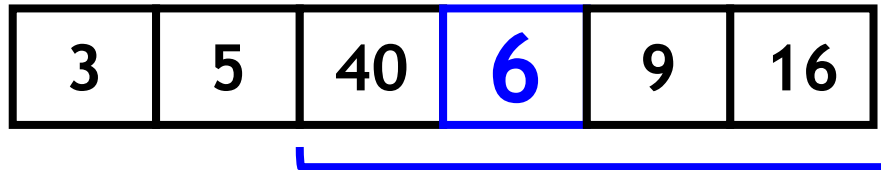
- Troca segundo elemento com o menor elemento do subvetor:



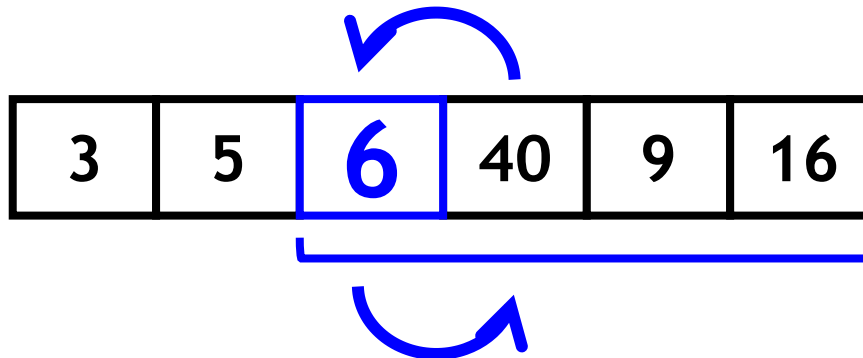
Agora os dois primeiros elementos já estão em ordem. Então repetimos o processo para o restante do vetor.

Selection sort

- Ideia geral:
 - Encontra menor elemento no restante do vetor:

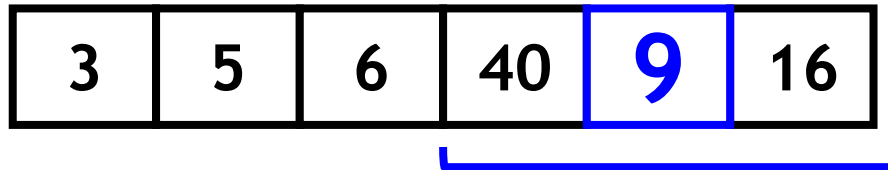


- Troca terceiro elemento com o menor elemento do subvetor:

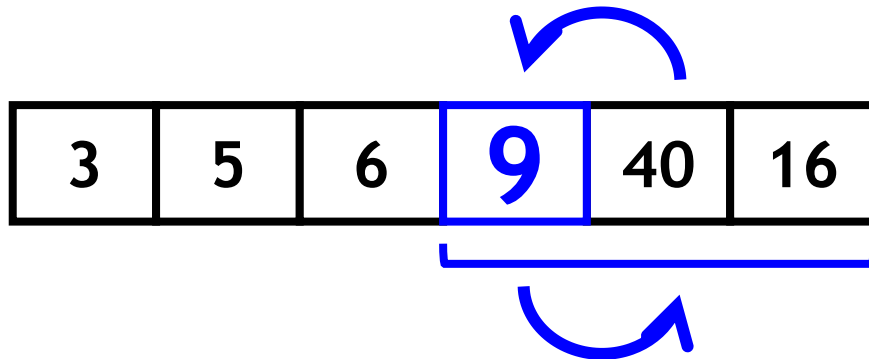


Selection sort

- Ideia geral:
 - Encontra menor elemento no restante do vetor:



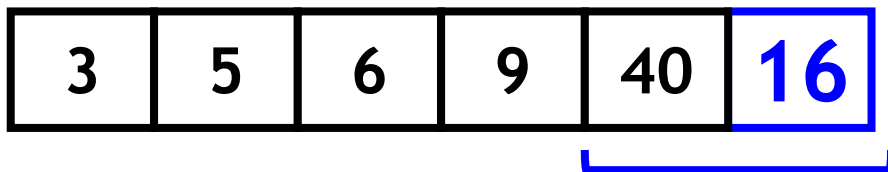
- Troca quarto elemento com o menor elemento do subvetor:



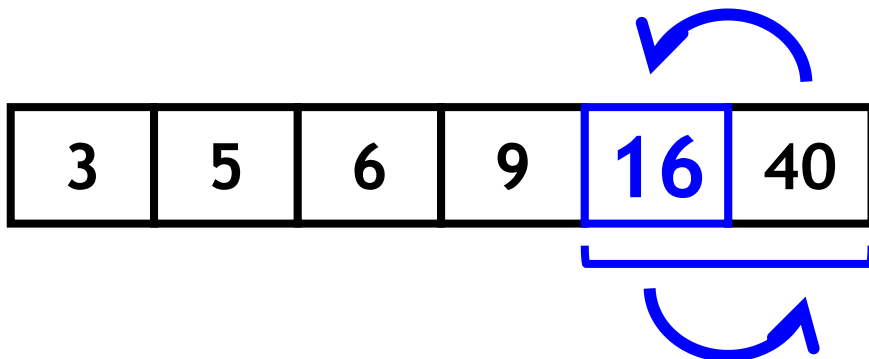
Selection sort

- Ideia geral:

- Encontra menor elemento no restante do vetor:



- Troca quinto elemento com o menor elemento do subvetor:



Após isso, o algoritmo é finalizado!

Selection sort

- Implementação em C

Chamada:

```
selection_sort(vetor, n);
```

Implementação

```
void selection_sort(int *v, int n) {  
    int i;  
    for (i = 0; i < n-1; i++) {  
        int indice_menor = i;  
        int k;  
        for (k = i+1; k < n; k++)  
            if (v[k] < v[indice_menor])  
                indice_menor = k;  
  
        int tmp = v[i];  
        v[i] = v[indice_menor];  
        v[indice_menor] = tmp;  
    }  
}
```

Encontra o menor
elemento no
restante do vetor.

Troca o menor com o
elemento atual.

Implementação

```
void selection_sort(int *v, int n) {  
    int i;  
    for (i = 0; i < n-1; i++) {  
        int indice_menor = i;  
        int k;  
        for (k = i+1; k < n; k++)  
            if (v[k] < v[indice_menor])  
                indice_menor = k;  
  
        if (indice_menor != i) {  
            int tmp = v[i];  
            v[i] = v[indice_menor];  
            v[indice_menor] = tmp;  
        }  
    }  
}
```



Apenas faz a troca
se o índice for
diferente.

Ok, mas e para ordenar
em ordem decrescente?

Ok, mas e para ordenar em ordem decrescente?

Ao invés de trocar pelo menor elemento, trocamos pelo maior elemento!

No algoritmo isso implica apenas em mudar nossa comparação de:

```
if (v[k] < v[indice_menor])
```

Para:

```
if (v[k] > v[indice_menor])
```

Também é recomendável renomear a variável `indice_menor` para `indice_maior`

Ok, mas e para ordenar em ordem decrescente?

Ao invés de trocar pelo menor elemento, trocamos pelo maior elemento!

No algoritmo isso implica apenas em mudar nossa comparação de:

```
if (v[k] < v[indice_menor])
```

Para:

```
if (v[k] > v[indice_maior])
```

Quantas comparações de elementos do vetor realizamos nesse algoritmo?

```
void selection_sort(int *v, int n) {  
    int i;  
    for (i = 0; i < n-1; i++) {  
        int indice_menor = i;  
        int k;  
        for (k = i+1; k < n; k++)  
            if (v[k] < v[indice_menor])  
                indice_menor = k;  
  
        if (indice_menor != i) {  
            int tmp = v[i];  
            v[i] = v[indice_menor];  
            v[indice_menor] = tmp;  
        }  
    }  
}
```

Encontra o menor elemento no restante do vetor.

Troca o menor com o elemento atual.

Número de comparações

```
void selection_sort(int *v, int n) {  
    int i;  
    for (i = 0; i < n-1; i++) {  
        int indice_menor = i;  
        int k;  
        for (k = i+1; k < n; k++)  
            if (v[k] < v[indice_menor])  
                indice_menor = k;  
  
        if (indice_menor != i) {  
            int tmp = v[i];  
            v[i] = v[indice_menor];  
            v[indice_menor] = tmp;  
        }  
    }  
}
```

← Executa $\sum_0^{n-2} q_i$ vezes.

q_i : representa a quantidade de comparações realizadas na iteração i .

Número de comparações

```
void selection_sort(int *v, int n) {  
    int i;  
    for (i = 0; i < n-1; i++) {  
        int indice_menor = i;  
        int k;  
        for (k = i+1; k < n; k++)  
            if (v[k] < v[indice_menor])  
                indice_menor = k;  
  
        if (indice_menor != i) {  
            int tmp = v[i];  
            v[i] = v[indice_menor];  
            v[indice_menor] = tmp;  
        }  
    }  
}
```

← Executa $\sum_0^{n-2} q_i$ vezes.

q_i : representa a quantidade de comparações realizadas na iteração i .

Os valores de q_i são: $q_0 = n - 1; q_1 = n - 2; \dots; q_{n-2} = 1$

Número de comparações

- O número de comparações é: $\sum_0^{n-2} q_i$
- E os valores q_i são: $q_0 = n - 1; q_1 = n - 2; \dots; q_{n-2} = 1$
- Os valores de q_i formam uma progressão aritmética (PA). O número de comparações é a soma dos elementos dessa PA:

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

Selection sort

- Número de comparações no pior caso: $\frac{n(n-1)}{2}$

$$O(n^2)$$

- Número de comparações no melhor caso: $\frac{n(n-1)}{2}$

$$O(n^2)$$

Bubble sort

Bubble sort

- Ideia geral:
 - Inicia no primeiro elemento e compara os elementos dois a dois;
 - Se $\text{elemento}[k] > \text{elemento}[k+1]$, troca os dois elementos;
 - Repete o processo $n - 1$ vezes. Contudo, não é necessário ir até o fim do vetor nas demais iterações:
 - O processo aplicado garante que o maior elemento estará na última posição;
 - Na segunda iteração, o segundo maior elemento estará na penúltima posição, e assim por diante.

Bubble sort

$v =$

6	9	40	3	5	16
k	$k+1$				

$v[k] > v[k+1]$? Não.

Bubble sort

$v =$

6	9	40	3	5	16
	k	$k+1$			

$v[k] > v[k+1]$? Não.

Bubble sort

$v =$

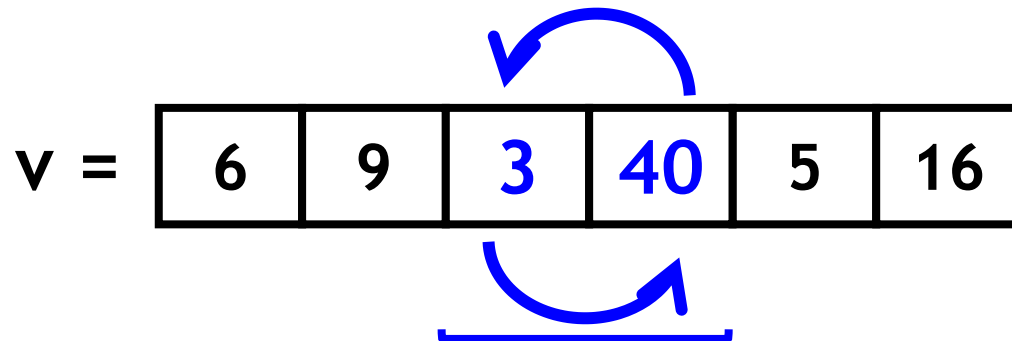
6	9	40	3	5	16
---	---	----	---	---	----

$k \quad k+1$

$v[k] > v[k+1]$? **Sim.**

Troca os dois elementos.

Bubble sort



$v[k] > v[k+1]$? **Sim.**

Troca os dois elementos.

Bubble sort

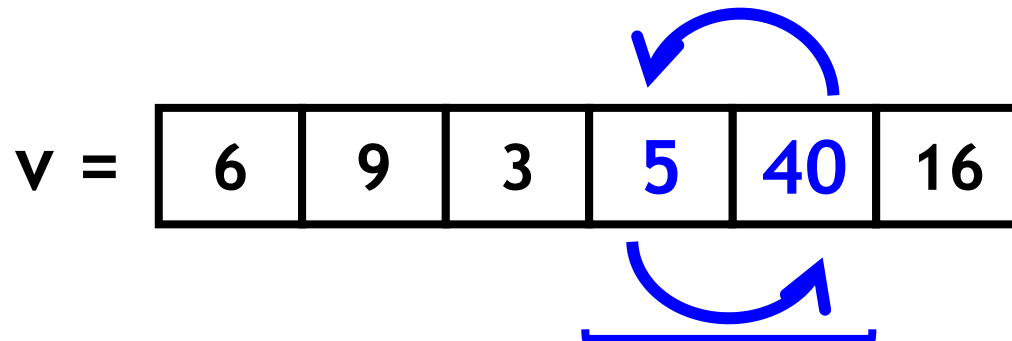
$v =$

6	9	3	40	5	16
			k	$k+1$	

$v[k] > v[k+1]$? **Sim.**

Troca os dois elementos.

Bubble sort



$v[k] > v[k+1]$? **Sim.**

Troca os dois elementos.

Bubble sort

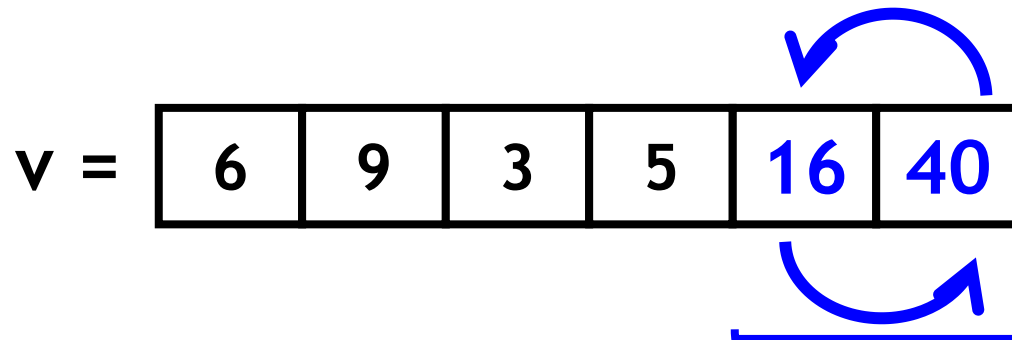
$v =$

6	9	3	5	40	16
---	---	---	---	----	----

k $k+1$

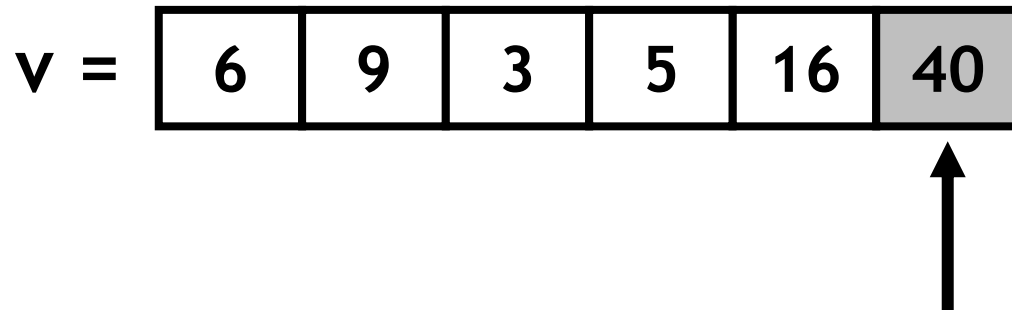
$v[k] > v[k+1]$? **Sim.**
Troca os dois elementos.

Bubble sort



$v[k] > v[k+1]$? **Sim.**
Troca os dois elementos.

Bubble sort



- Primeira iteração finalizada! Veja que o maior elemento está no final do vetor;
- Agora vamos repetir o processo, mas não precisamos ir até o último elemento.

Bubble sort

$v =$

6	9	3	5	16	40
k	$k+1$				

$v[k] > v[k+1]$? Não.

Bubble sort

$v =$

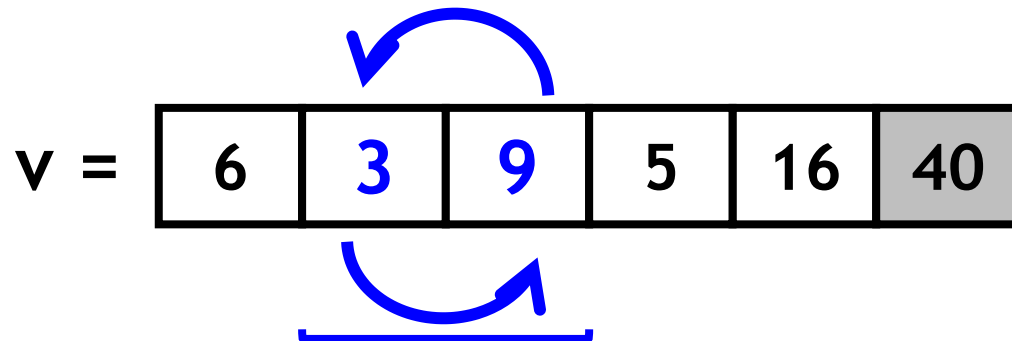
6	9	3	5	16	40
---	---	---	---	----	----

$k \quad k+1$

$v[k] > v[k+1]$? **Sim.**

Troca os dois elementos.

Bubble sort



$v[k] > v[k+1]$? **Sim.**

Troca os dois elementos.

Bubble sort

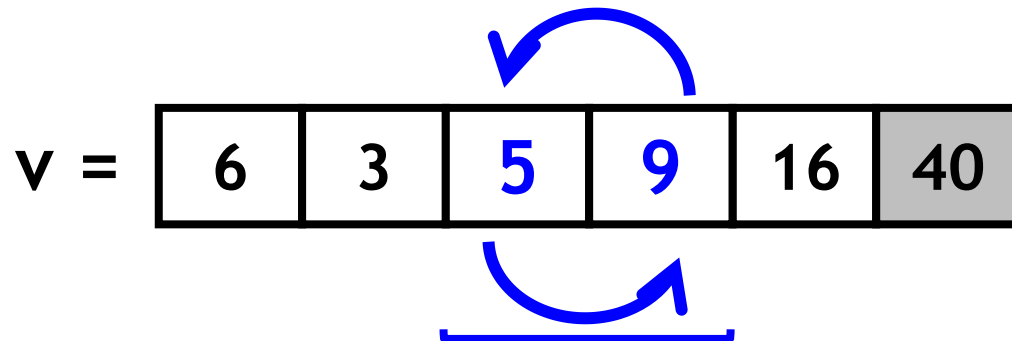
$v =$

6	3	9	5	16	40
		k	$k+1$		

$v[k] > v[k+1]$? **Sim.**

Troca os dois elementos.

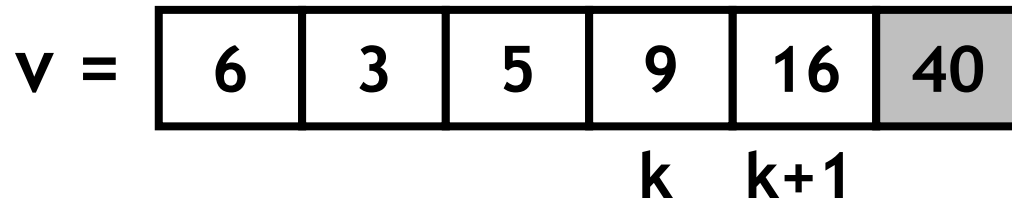
Bubble sort



$v[k] > v[k+1]$? **Sim.**

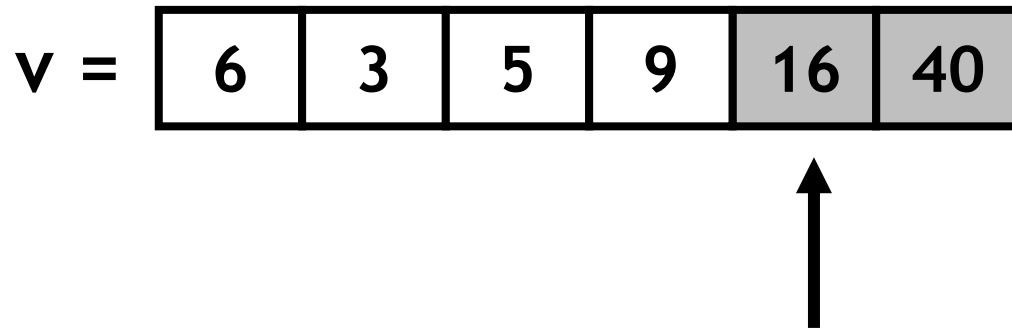
Troca os dois elementos.

Bubble sort



$v[k] > v[k+1]$? Não.

Bubble sort



- Segunda iteração finalizada! Veja que o segundo maior elemento está no final do vetor;
- Agora vamos repetir o processo, mas não precisamos ir até o penúltimo elemento.

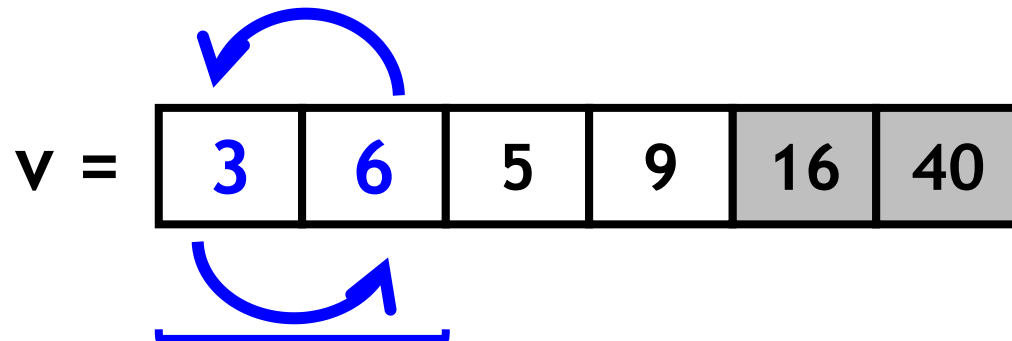
Bubble sort

$v =$

6	3	5	9	16	40
k	$k+1$				

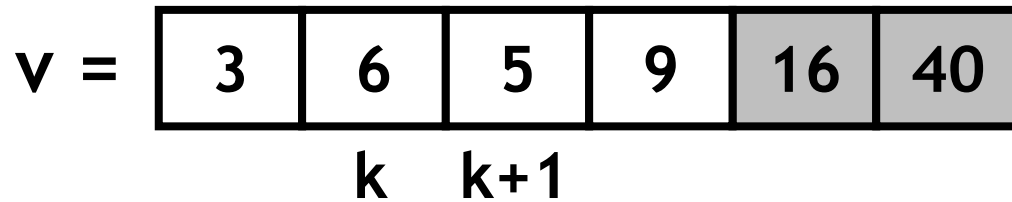
$v[k] > v[k+1]$? **Sim.**
Troca os dois elementos.

Bubble sort



$v[k] > v[k+1]$? **Sim.**
Troca os dois elementos.

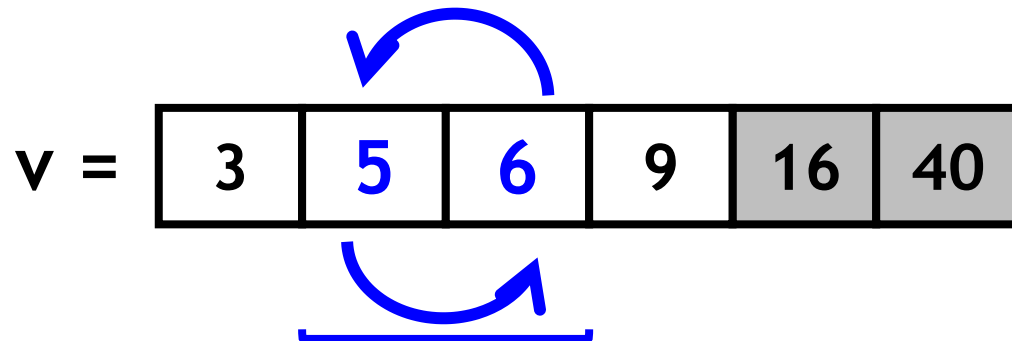
Bubble sort



$v[k] > v[k+1]$? **Sim.**

Troca os dois elementos.

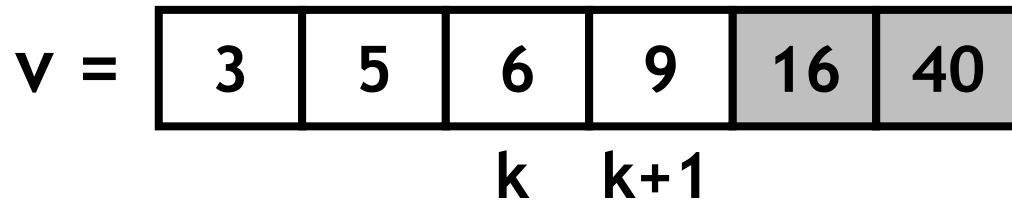
Bubble sort



$v[k] > v[k+1]$? **Sim.**

Troca os dois elementos.

Bubble sort



$v[k] > v[k+1]$? Não.

Bubble sort

$v =$

3	5	6	9	16	40
---	---	---	---	----	----

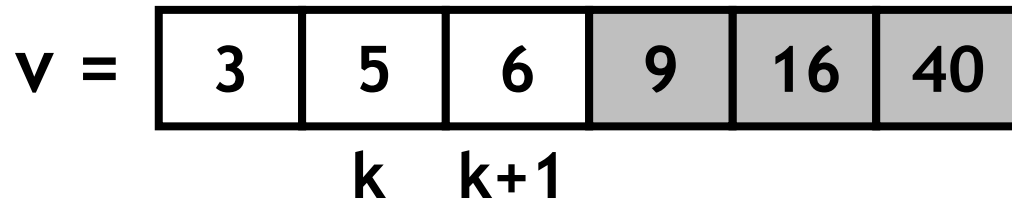
Bubble sort

$v =$

3	5	6	9	16	40
k	$k+1$				

$v[k] > v[k+1]$? Não.

Bubble sort



$v[k] > v[k+1]$? Não.

Bubble sort

$v =$

3	5	6	9	16	40
---	---	---	---	----	----

Bubble sort

$v =$

3	5	6	9	16	40
k	$k+1$				

$v[k] > v[k+1]$? Não.

Bubble sort

$v =$

3	5	6	9	16	40
---	---	---	---	----	----

Bubble sort

v =

3	5	6	9	16	40
---	---	---	---	----	----

Ordenação finalizada.

Bubble sort

- Implementação em C

Chamada:

```
bubblesort(vetor, n);
```


Implementação

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

A cada iteração,
percorre um elemento
a menos (-i).

Troca elementos
consecutivos se $v[k] > v[k+1]$

Ok, mas e para ordenar
em ordem decrescente?

Ok, mas e para ordenar em ordem decrescente?

Apenas precisamos mudar a condição de comparação de elementos de:

$$v[k] > v[k+1]$$

Para:

$$v[k] < v[k+1]$$

Quantas comparações de elementos do vetor realizamos nesse algoritmo?

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

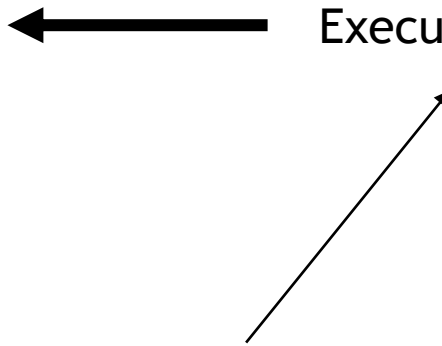
A cada iteração,
percorre um elemento
a menos (-i).

Troca elementos
consecutivos se $v[k] > v[k+1]$

Número de comparações

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

Executa $\sum_0^{n-2} q_i$ vezes.



q_i : representa a quantidade de comparações realizadas na iteração i .

Os valores de q_i são: $q_0 = n - 1; q_1 = n - 2; \dots; q_{n-2} = 1$

Número de comparações

- O número de comparações é: $\sum_0^{n-2} q_i$
- E os valores q_i são: $q_0 = n - 1; q_1 = n - 2; \dots; q_{n-2} = 1$
- Os valores de q_i formam uma progressão aritmética (PA). O número de comparações é a soma dos elementos dessa PA:

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

Bubble sort

- Número de comparações no pior caso: $\frac{n(n-1)}{2}$

$$O(n^2)$$

- Número de comparações no melhor caso: $\frac{n(n-1)}{2}$

$$O(n^2)$$

Será que podemos otimizar o Bubble sort?

Será que podemos otimizar o Bubble sort?

**Pense no caso que o vetor já está ordenado.
Há necessidade de continuar com as demais
iterações do algoritmo?**

Será que podemos otimizar o Bubble sort?

**Pense no caso que o vetor já está ordenado.
Há necessidade de continuar com as demais
iterações do algoritmo?**

NÃO!

**Ok, mas como sabemos que já está
ordenado?**

Será que podemos otimizar o Bubble sort?

**Pense no caso que o vetor já está ordenado.
Há necessidade de continuar com as demais
iterações do algoritmo?**

NÃO!

**Ok, mas como sabemos que já está
ordenado?**

**Se não foi realizada nenhuma troca na última
iteração, podemos parar a ordenação.**

Bubble sort (com parada antecipada)

- Implementação em C

Chamada:

```
bubblesort_es(vetor, n);
```

Implementação

```
void bubblesort_es(int *v, int n) {  
    int i, k;  
    for (i = 0; i < n - 1; i++) {  
        int trocou = 0;  
        for (k = 0; k < n - 1 - i; k++) {  
            if (v[k] > v[k+1]) {  
                int tmp = v[k];  
                v[k] = v[k + 1];  
                v[k + 1] = tmp;  
                trocou = 1;  
            }  
        }  
        if (!trocou) break;  
    }  
}
```

A cada iteração,
percorre um elemento
a menos (-i).

Troca elementos
consecutivos se $v[k] > v[k+1]$

Outra implementação (sem break)

```
void bubblesort_es(int *v, int n) {  
    int i, k, trocou = 1;  
    for (i = 0; i < n - 1 && trocou; i++) {  
        trocou = 0;  
        for (k = 0; k < n - 1 - i; k++)  
            if (v[k] > v[k+1]) {  
                int tmp = v[k];  
                v[k] = v[k + 1];  
                v[k + 1] = tmp;  
                trocou = 1;  
            }  
    }  
}
```

Quantas comparações de elementos do vetor realizamos nesse algoritmo?

```
void bubblesort_es(int *v, int n) {  
    int i, k, trocou = 1;  
    for (i = 0; i < n - 1 && trocou; i++) {  
        trocou = 0;  
        for (k = 0; k < n - 1 - i; k++) {  
            if (v[k] > v[k+1]) {  
                int tmp = v[k];  
                v[k] = v[k + 1];  
                v[k + 1] = tmp;  
                trocou = 1;  
            }  
        }  
    }  
}
```

A cada iteração,
percorre um elemento
a menos (-i).

Troca elementos
consecutivos se $v[k] > v[k+1]$

Número de comparações

```
void bubblesort_es(int *v, int n) {  
    int i, k, trocou = 1;  
    for (i = 0; i < n - 1 && trocou; i++) {  
        trocou = 0;  
        for (k = 0; k < n - 1 - i; k++)  
            if (v[k] > v[k+1]) {  
                int tmp = v[k];  
                v[k] = v[k + 1];  
                v[k + 1] = tmp;  
                trocou = 1;  
            }  
        }  
    }  
}
```

Executa até $\sum_0^{n-2} q_i$ vezes.

q_i : representa a quantidade de comparações realizadas na iteração i .

Número de comparações

Melhor caso

```
void bubblesort_es(int *v, int n) {  
    int i, k, trocou = 1;  
    for (i = 0; i < n - 1 && trocou; i++) {  
        trocou = 0;  
        for (k = 0; k < n - 1 - i; k++)  
            if (v[k] > v[k+1]) { ← No melhor caso (vetor já ordenado  
                int tmp = v[k];      na ordem desejada), executa  $n - 1$   
                v[k] = v[k + 1];     vezes.  
                v[k + 1] = tmp;  
                trocou = 1;  
            }  
        }  
    }  
}
```

No melhor caso, o número de comparações é $n-1$

Número de comparações

Pior caso

```
void bubblesort_es(int *v, int n) {  
    int i, k, trocou = 1;  
    for (i = 0; i < n - 1 && trocou; i++) {  
        trocou = 0;  
        for (k = 0; k < n - 1 - i; k++)  
            if (v[k] > v[k+1]) {  
                int tmp = v[k];  
                v[k] = v[k + 1];  
                v[k + 1] = tmp;  
                trocou = 1;  
            }  
        }  
    }
```

No pior caso, executa $\sum_{i=0}^{n-2} q_i$ vezes.

q_i : representa a quantidade de comparações realizadas na iteração i .

Os valores de q_i são: $q_0 = n - 1$; $q_1 = n - 2$; ...; $q_{n-2} = 1$

No pior caso, o número de comparações é igual ao selection sort

Bubble sort (com parada antecipada)

- Número de comparações no pior caso: $\frac{n(n-1)}{2}$

$$O(n^2)$$

- Número de comparações no melhor caso: $n - 1$

$$O(n)$$

Insertion sort

Insertion sort

- Ideia geral:
 - Inicia com subvetor de um elemento (primeiro elemento do vetor) - este será o subvetor ordenado;
 - Depois avalia o próximo elemento e o insere na posição correta no subvetor ordenado; Agora o subvetor ordenado tem dois elementos;
 - Esse processo é repetido, inserindo o próximo elemento no subvetor ordenado até que o subvetor seja o vetor completo.

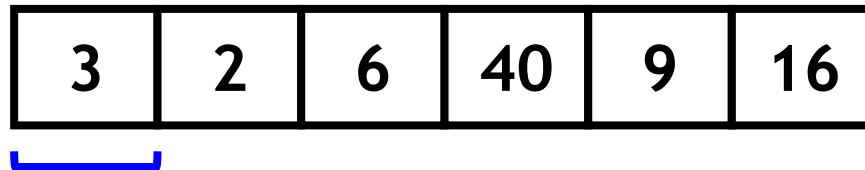
Insertion sort

3	2	6	40	9	16
---	---	---	----	---	----

Subvetor ordenado

Insertion sort

Onde este elemento deveria ser inserido para manter o subvetor ordenado?



Subvetor ordenado

Insertion sort

2	3	6	40	9	16
---	---	---	----	---	----

Subvetor ordenado

Insertion sort

Onde este elemento deveria ser inserido para manter o subvetor ordenado?



2	3	6	40	9	16
---	---	---	----	---	----

Subvetor ordenado

Insertion sort

2	3	6	40	9	16
---	---	---	----	---	----



Subvetor ordenado

Insertion sort

Onde este elemento deveria ser inserido para manter o subvetor ordenado?



2	3	6	40	9	16
---	---	---	----	---	----



Subvetor ordenado

Insertion sort

2	3	6	40	9	16
---	---	---	----	---	----

Subvetor ordenado

Insertion sort

Onde este elemento deveria ser inserido para manter o subvetor ordenado?

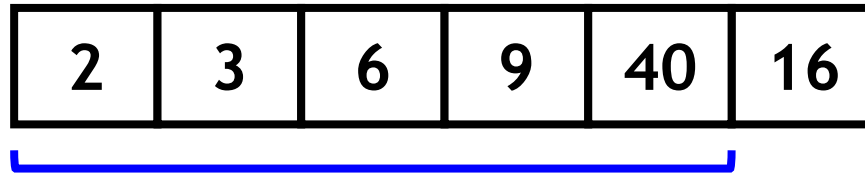


2	3	6	40	9	16
---	---	---	----	---	----



Subvetor ordenado

Insertion sort



Subvetor ordenado

Insertion sort

Onde este elemento deveria ser inserido para manter o subvetor ordenado?




2	3	6	9	40	16
---	---	---	---	----	----



Subvetor ordenado

Insertion sort

2	3	6	9	16	40
---	---	---	---	----	----



Insertion sort

- Implementação em C

Chamada:
`insertionsort(vetor, n);`

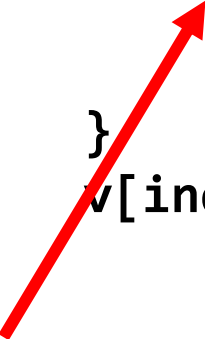
Implementação

```
void insertion_sort(int *v, int n) {  
    int i, k;  
    for (i = 1; i < n; i++) {  
        int item_atual = v[i];  
        int indice_para_inserir = i;  
        for (k = i - 1; k >= 0  
            && item_atual < v[k]; k--) {  
            v[k+1] = v[k];  
            indice_para_inserir--;  
        }  
        v[indice_para_inserir] = item_atual;  
    }  
}
```

Guarda elemento atual

Encontra índice para inserção e desloca elementos para a direita.

Inserir elemento atual.



Veja que apenas fazemos deslocamentos até encontrar a posição correta para inserção.

Ok, mas e para ordenar
em ordem decrescente?

Ok, mas e para ordenar em ordem decrescente?

Assim como no selection sort, apenas temos que mudar a condição de comparação de:

`item_atual < v[k]`

Para:

`item_atual > v[k]`

Quantas comparações de elementos do vetor realizamos nesse algoritmo?

```
void insertion_sort(int *v, int n) {  
    int i, k;  
    for (i = 1; i < n; i++) {  
        int item_atual = v[i];  
        int indice_para_inserir = i;  
        for (k = i - 1; k >= 0  
            && item_atual < v[k]; k--) {  
            v[k+1] = v[k];  
            indice_para_inserir--;  
        }  
        v[indice_para_inserir] = item_atual;  
    }  
}
```

Guarda elemento atual

Encontra índice para inserção e desloca elementos para a direita.

Inserir elemento atual.

Número de comparações

```
void insertion_sort(int *v, int n) {  
    int i, k;  
    for (i = 1; i < n; i++) {  
        int item_atual = v[i];  
  
        int indice_para_inserir = i;  
        for (k = i - 1; k >= 0  
            && item_atual < v[k]; k--) {  
            v[k+1] = v[k];  
            indice_para_inserir--;  
        }  
        v[indice_para_inserir] = item_atual;  
    }  
}
```

← Executa até $\sum_1^{n-1} q_i$ vezes.

q_i : representa a quantidade
de comparações realizadas
na iteração i .

Número de comparações

Melhor caso

(vetor já em
ordem crescente)

```
void insertion_sort(int *v, int n) {  
    int i, k;  
    for (i = 1; i < n; i++) {  
        int item_atual = v[i];  
  
        int indice_para_inserir = i;  
        for (k = i - 1; k >= 0  
            && item_atual < v[k]; k--) {  
            v[k+1] = v[k];  
            indice_para_inserir--;  
        }  
        v[indice_para_inserir] = item_atual;  
    }  
}
```

← Executa até $\sum_1^{n-1} q_i$ vezes.

Todos os q_i teriam valor 1.

No melhor caso, o número de comparações é $n-1$

Número de comparações

Pior caso

(vetor em ordem decrescente)

```
void insertion_sort(int *v, int n) {  
    int i, k;  
    for (i = 1; i < n; i++) {  
        int item_atual = v[i];  
  
        int indice_para_inserir = i;  
        for (k = i - 1; k >= 0  
            && item_atual < v[k]; k--) {  
            v[k+1] = v[k];  
            indice_para_inserir--;  
        }  
        v[indice_para_inserir] = item_atual;  
    }  
}
```

← Executa até $\sum_{i=1}^{n-1} q_i$ vezes.

Os valores de q_i são: $q_1 = 1; q_2 = 2; \dots; q_{n-1} = n - 1$

No pior caso, o número de comparações é igual ao selection sort

Número de comparações

- O número de comparações é: $\sum_1^{n-1} q_i$
- E os valores q_i são: $q_1 = 1; q_2 = 2; \dots; q_{n-1} = n - 1$
- Os valores de q_i formam uma progressão aritmética (PA). O número de comparações é a soma dos elementos dessa PA:

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

Insertion sort

- Número de comparações no pior caso: $\frac{n(n-1)}{2}$

$$O(n^2)$$

- Número de comparações no melhor caso: $n - 1$

$$O(n)$$

Exemplos dos algoritmos

- Selection sort:
 - <https://visualgo.net/en/sorting?slide=7>
- Bubble sort:
 - <https://visualgo.net/en/sorting?slide=6>
- Bubble sort (early stopping):
 - <https://visualgo.net/en/sorting?slide=6-2>
- Insertion sort:
 - <https://visualgo.net/en/sorting?slide=8>

Referências

- Slides do Prof. Monael Pinheiro Riberio:
 - <https://sites.google.com/site/aed2019q1/>
- Slides da Profa. Mirtha Lina Fernández Venero
 - Algoritmos e Estruturas de Dados I - 2019

Referências

- Nivio Ziviani. Projeto de Algoritmos: com implementações em Pascal e C. Cengage Learning, 2015.
- Jayme L. Szwarcfiter, Lilian Markenzon. Estruturas de Dados e Seus Algoritmos. 3ª edição. LTC, 2012.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Algoritmos: Teoria e Prática. Elsevier, 2012.