

# MC-202 — Unidade 10

## Ordenação

Rafael C. S. Schouery  
rafael@ic.unicamp.br

Universidade Estadual de Campinas

1º semestre/2018

# Ordenação

Queremos ordenar um vetor

# Ordenação

Queremos ordenar um vetor

3	7	1	6	5	2	4	0	8	9
---	---	---	---	---	---	---	---	---	---



0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

# Ordenação

Queremos ordenar um vetor

3	7	1	6	5	2	4	0	8	9
---	---	---	---	---	---	---	---	---	---



0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

Nos códigos vamos ordenar vetores de `int`

# Ordenação

Queremos ordenar um vetor

3	7	1	6	5	2	4	0	8	9
---	---	---	---	---	---	---	---	---	---



0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

Nos códigos vamos ordenar vetores de `int`

- Mas é fácil alterar para comparar `double` ou `string`

# Ordenação

Queremos ordenar um vetor

3	7	1	6	5	2	4	0	8	9
---	---	---	---	---	---	---	---	---	---



0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

Nos códigos vamos ordenar vetores de `int`

- Mas é fácil alterar para comparar `double` ou `string`
- ou comparar `struct` por algum de seus campos

# Ordenação

Queremos ordenar um vetor

3	7	1	6	5	2	4	0	8	9
---	---	---	---	---	---	---	---	---	---



0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

Nos códigos vamos ordenar vetores de `int`

- Mas é fácil alterar para comparar `double` ou `string`
- ou comparar `struct` por algum de seus campos
  - O valor usado para a ordenação é a `chave` de ordenação

# Ordenação

Queremos ordenar um vetor

3	7	1	6	5	2	4	0	8	9
---	---	---	---	---	---	---	---	---	---



0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

Nos códigos vamos ordenar vetores de `int`

- Mas é fácil alterar para comparar `double` ou `string`
- ou comparar `struct` por algum de seus campos
  - O valor usado para a ordenação é a *chave* de ordenação
  - Podemos até desempatar por outros campos



# Ordenação

Queremos ordenar um vetor

3	7	1	6	5	2	4	0	8	9
---	---	---	---	---	---	---	---	---	---



0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

Nos códigos vamos ordenar vetores de `int`

- Mas é fácil alterar para comparar `double` ou `string`
- ou comparar `struct` por algum de seus campos
  - O valor usado para a ordenação é a *chave* de ordenação
  - Podemos até desempatar por outros campos

Ao invés de ordenar o vetor inteiro

# Ordenação

Queremos ordenar um vetor

3	7	1	6	5	2	4	0	8	9
---	---	---	---	---	---	---	---	---	---



0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

Nos códigos vamos ordenar vetores de `int`

- Mas é fácil alterar para comparar `double` ou `string`
- ou comparar `struct` por algum de seus campos
  - O valor usado para a ordenação é a `chave` de ordenação
  - Podemos até desempatar por outros campos

Ao invés de ordenar o vetor inteiro

- iremos ordenar da posição `l` até a posição `r`

# Ordenação

Queremos ordenar um vetor

3	7	1	6	5	2	4	0	8	9
---	---	---	---	---	---	---	---	---	---



0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

Nos códigos vamos ordenar vetores de `int`

- Mas é fácil alterar para comparar `double` ou `string`
- ou comparar `struct` por algum de seus campos
  - O valor usado para a ordenação é a `chave` de ordenação
  - Podemos até desempatar por outros campos

Ao invés de ordenar o vetor inteiro

- iremos ordenar da posição `l` até a posição `r`
- isso será útil nas próximas unidades...

# Ordenação

Queremos ordenar um vetor

3	7	1	6	5	2	4	0	8	9
---	---	---	---	---	---	---	---	---	---



0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

Nos códigos vamos ordenar vetores de `int`

- Mas é fácil alterar para comparar `double` ou `string`
- ou comparar `struct` por algum de seus campos
  - O valor usado para a ordenação é a *chave* de ordenação
  - Podemos até desempatar por outros campos

Ao invés de ordenar o vetor inteiro

- iremos ordenar da posição  $l$  até a posição  $r$
- isso será útil nas próximas unidades...
- usaremos  $n := r - l + 1$  (número de elementos entre  $l$  e  $r$ )

## A função troca

Várias vezes iremos trocar dois elementos de posição

## A função troca

Várias vezes iremos trocar dois elementos de posição

Para tanto, vamos usar a seguinte função:

# A função troca

Várias vezes iremos trocar dois elementos de posição

Para tanto, vamos usar a seguinte função:

```
1 void troca(int *a, int *b) {  
2     int t = *a;  
3     *a = *b;  
4     *b = t;  
5 }
```

# A função troca

Várias vezes iremos trocar dois elementos de posição

Para tanto, vamos usar a seguinte função:

```
1 void troca(int *a, int *b) {  
2     int t = *a;  
3     *a = *b;  
4     *b = t;  
5 }
```

Ou seja, `troca(&v[i], &v[j])` troca os valores de `v[i]` e `v[j]`



# A função troca

Várias vezes iremos trocar dois elementos de posição

Para tanto, vamos usar a seguinte função:

```
1 void troca(int *a, int *b) {  
2     int t = *a;  
3     *a = *b;  
4     *b = t;  
5 }
```

Ou seja, `troca(&v[i], &v[j])` troca os valores de `v[i]` e `v[j]`

Outra opção é colocar diretamente no código da função

# A função troca

Várias vezes iremos trocar dois elementos de posição

Para tanto, vamos usar a seguinte função:

```
1 void troca(int *a, int *b) {  
2     int t = *a;  
3     *a = *b;  
4     *b = t;  
5 }
```

Ou seja, `troca(&v[i], &v[j])` troca os valores de `v[i]` e `v[j]`

Outra opção é colocar diretamente no código da função

- não precisa chamar outra função

# A função troca

Várias vezes iremos trocar dois elementos de posição

Para tanto, vamos usar a seguinte função:

```
1 void troca(int *a, int *b) {  
2     int t = *a;  
3     *a = *b;  
4     *b = t;  
5 }
```

Ou seja, `troca(&v[i], &v[j])` troca os valores de `v[i]` e `v[j]`

Outra opção é colocar diretamente no código da função

- não precisa chamar outra função
- um pouco mais rápido

# A função troca

Várias vezes iremos trocar dois elementos de posição

Para tanto, vamos usar a seguinte função:

```
1 void troca(int *a, int *b) {  
2     int t = *a;  
3     *a = *b;  
4     *b = t;  
5 }
```

Ou seja, `troca(&v[i], &v[j])` troca os valores de `v[i]` e `v[j]`

Outra opção é colocar diretamente no código da função

- não precisa chamar outra função
- um pouco mais rápido
- código um pouco mais longo e difícil de entender

# Ordenação por Seleção

Ideia:

# Ordenação por Seleção

Ideia:

- Trocar  $v[1]$  com o mínimo de  $v[1], v[1+1], \dots, v[r]$

# Ordenação por Seleção

Ideia:

- Trocar  $v[1]$  com o mínimo de  $v[1], v[1+1], \dots, v[r]$
- Trocar  $v[1+1]$  com o mínimo de  $v[1+1], v[1+2], \dots, v[r]$

# Ordenação por Seleção

Ideia:

- Trocar  $v[1]$  com o mínimo de  $v[1], v[1+1], \dots, v[r]$
- Trocar  $v[1+1]$  com o mínimo de  $v[1+1], v[1+2], \dots, v[r]$
- ...



# Ordenação por Seleção

Ideia:

- Trocar  $v[1]$  com o mínimo de  $v[1], v[1+1], \dots, v[r]$
- Trocar  $v[1+1]$  com o mínimo de  $v[1+1], v[1+2], \dots, v[r]$
- ...
- Trocar  $v[1+i]$  com o mínimo de  $v[1+i], \dots, v[r]$

# Ordenação por Seleção

Ideia:

- Trocar  $v[1]$  com o mínimo de  $v[1], v[1+1], \dots, v[r]$
- Trocar  $v[1+1]$  com o mínimo de  $v[1+1], v[1+2], \dots, v[r]$
- ...
- Trocar  $v[1+i]$  com o mínimo de  $v[1+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {
```

# Ordenação por Seleção

Ideia:

- Trocar  $v[1]$  com o mínimo de  $v[1], v[1+1], \dots, v[r]$
- Trocar  $v[1+1]$  com o mínimo de  $v[1+1], v[1+2], \dots, v[r]$
- ...
- Trocar  $v[1+i]$  com o mínimo de  $v[1+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {
```

# Ordenação por Seleção

Ideia:

- Trocar  $v[1]$  com o mínimo de  $v[1], v[1+1], \dots, v[r]$
- Trocar  $v[1+1]$  com o mínimo de  $v[1+1], v[1+2], \dots, v[r]$
- ...
- Trocar  $v[1+i]$  com o mínimo de  $v[1+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {  
2     int i, j, min;
```

# Ordenação por Seleção

Ideia:

- Trocar  $v[1]$  com o mínimo de  $v[1], v[1+1], \dots, v[r]$
- Trocar  $v[1+1]$  com o mínimo de  $v[1+1], v[1+2], \dots, v[r]$
- ...
- Trocar  $v[1+i]$  com o mínimo de  $v[1+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {  
2     int i, j, min;  
3     for (i = l; i < r; i++) {
```

# Ordenação por Seleção

Ideia:

- Trocar  $v[1]$  com o mínimo de  $v[1], v[1+1], \dots, v[r]$
- Trocar  $v[1+1]$  com o mínimo de  $v[1+1], v[1+2], \dots, v[r]$
- ...
- Trocar  $v[1+i]$  com o mínimo de  $v[1+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {  
2     int i, j, min;  
3     for (i = l; i < r; i++) {  
4         min = i;
```

# Ordenação por Seleção

Ideia:

- Trocar  $v[l]$  com o mínimo de  $v[l], v[l+1], \dots, v[r]$
- Trocar  $v[l+1]$  com o mínimo de  $v[l+1], v[l+2], \dots, v[r]$
- ...
- Trocar  $v[l+i]$  com o mínimo de  $v[l+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {
2     int i, j, min;
3     for (i = l; i < r; i++) {
4         min = i;
5         for (j = i+1; j <= r; j++)
6             if (v[j] < v[min])
7                 min = j;
```

# Ordenação por Seleção

Ideia:

- Trocar  $v[l]$  com o mínimo de  $v[l], v[l+1], \dots, v[r]$
- Trocar  $v[l+1]$  com o mínimo de  $v[l+1], v[l+2], \dots, v[r]$
- ...
- Trocar  $v[l+i]$  com o mínimo de  $v[l+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {
2     int i, j, min;
3     for (i = l; i < r; i++) {
4         min = i;
5         for (j = i+1; j <= r; j++)
6             if (v[j] < v[min])
7                 min = j;
8         troca(&v[i], &v[min]);
9     }
10 }
```

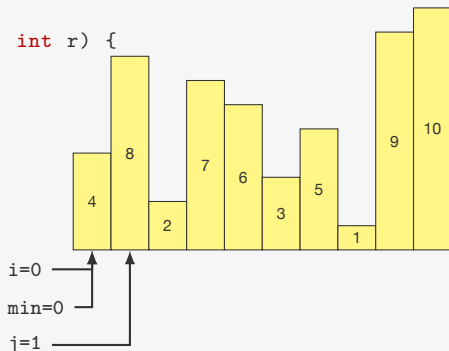


# Ordenação por Seleção

Ideia:

- Trocar  $v[1]$  com o mínimo de  $v[1], v[1+1], \dots, v[r]$
- Trocar  $v[1+1]$  com o mínimo de  $v[1+1], v[1+2], \dots, v[r]$
- ...
- Trocar  $v[1+i]$  com o mínimo de  $v[1+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {  
2     int i, j, min;  
3     for (i = l; i < r; i++) {  
4         min = i;  
5         for (j = i+1; j <= r; j++)  
6             if (v[j] < v[min])  
7                 min = j;  
8         troca(&v[i], &v[min]);  
9     }  
10 }
```

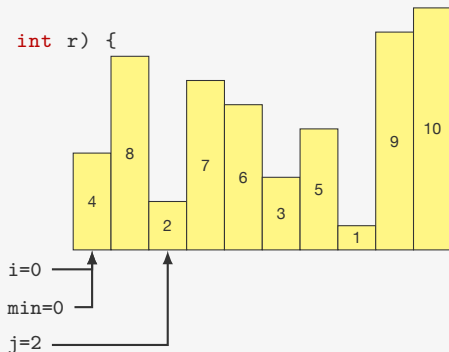


# Ordenação por Seleção

Ideia:

- Trocar  $v[1]$  com o mínimo de  $v[1], v[1+1], \dots, v[r]$
- Trocar  $v[1+1]$  com o mínimo de  $v[1+1], v[1+2], \dots, v[r]$
- ...
- Trocar  $v[1+i]$  com o mínimo de  $v[1+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {  
2     int i, j, min;  
3     for (i = l; i < r; i++) {  
4         min = i;  
5         for (j = i+1; j <= r; j++)  
6             if (v[j] < v[min])  
7                 min = j;  
8         troca(&v[i], &v[min]);  
9     }  
10 }
```

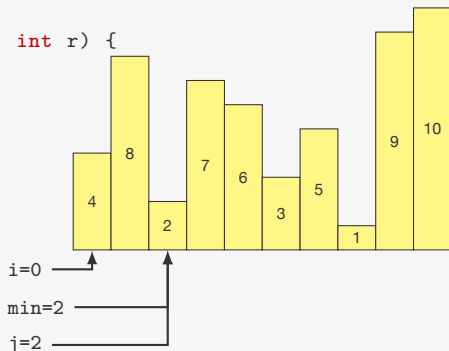


# Ordenação por Seleção

Ideia:

- Trocar  $v[1]$  com o mínimo de  $v[1], v[1+1], \dots, v[r]$
- Trocar  $v[1+1]$  com o mínimo de  $v[1+1], v[1+2], \dots, v[r]$
- ...
- Trocar  $v[1+i]$  com o mínimo de  $v[1+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {  
2     int i, j, min;  
3     for (i = l; i < r; i++) {  
4         min = i;  
5         for (j = i+1; j <= r; j++)  
6             if (v[j] < v[min])  
7                 min = j;  
8         troca(&v[i], &v[min]);  
9     }  
10 }
```

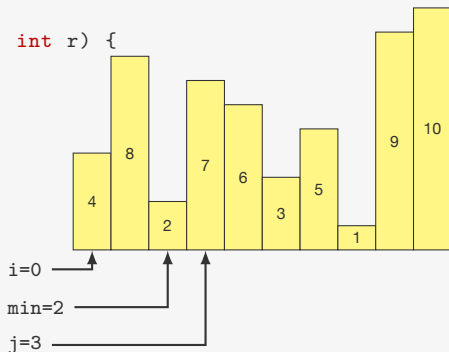


# Ordenação por Seleção

Ideia:

- Trocar  $v[1]$  com o mínimo de  $v[1], v[1+1], \dots, v[r]$
- Trocar  $v[1+1]$  com o mínimo de  $v[1+1], v[1+2], \dots, v[r]$
- ...
- Trocar  $v[1+i]$  com o mínimo de  $v[1+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {
2   int i, j, min;
3   for (i = l; i < r; i++) {
4     min = i;
5     for (j = i+1; j <= r; j++)
6       if (v[j] < v[min])
7         min = j;
8     troca(&v[i], &v[min]);
9   }
10 }
```

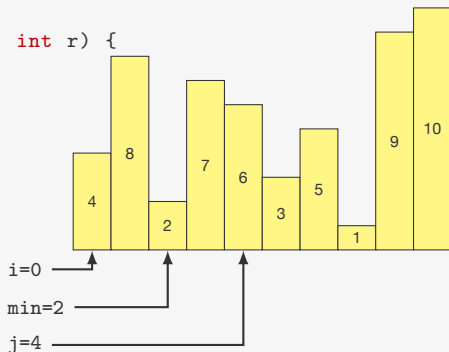


# Ordenação por Seleção

Ideia:

- Trocar  $v[1]$  com o mínimo de  $v[1], v[1+1], \dots, v[r]$
- Trocar  $v[1+1]$  com o mínimo de  $v[1+1], v[1+2], \dots, v[r]$
- ...
- Trocar  $v[1+i]$  com o mínimo de  $v[1+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {  
2     int i, j, min;  
3     for (i = l; i < r; i++) {  
4         min = i;  
5         for (j = i+1; j <= r; j++)  
6             if (v[j] < v[min])  
7                 min = j;  
8         troca(&v[i], &v[min]);  
9     }  
10 }
```

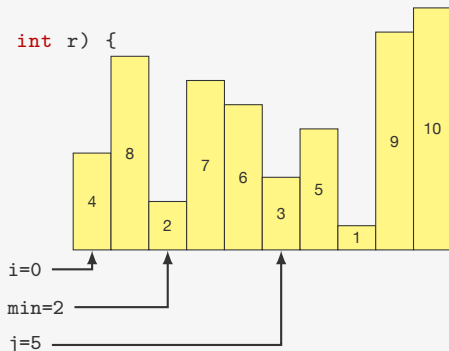


# Ordenação por Seleção

Ideia:

- Trocar  $v[1]$  com o mínimo de  $v[1], v[1+1], \dots, v[r]$
- Trocar  $v[1+1]$  com o mínimo de  $v[1+1], v[1+2], \dots, v[r]$
- ...
- Trocar  $v[1+i]$  com o mínimo de  $v[1+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {  
2     int i, j, min;  
3     for (i = l; i < r; i++) {  
4         min = i;  
5         for (j = i+1; j <= r; j++)  
6             if (v[j] < v[min])  
7                 min = j;  
8         troca(&v[i], &v[min]);  
9     }  
10 }
```

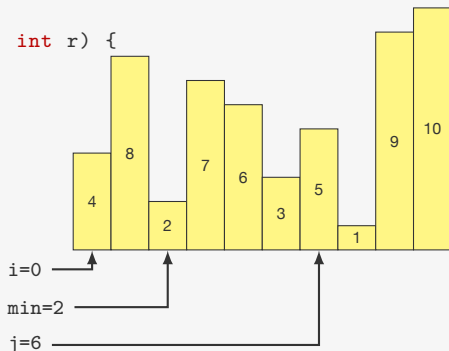


# Ordenação por Seleção

Ideia:

- Trocar  $v[1]$  com o mínimo de  $v[1], v[1+1], \dots, v[r]$
- Trocar  $v[1+1]$  com o mínimo de  $v[1+1], v[1+2], \dots, v[r]$
- ...
- Trocar  $v[1+i]$  com o mínimo de  $v[1+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {  
2     int i, j, min;  
3     for (i = l; i < r; i++) {  
4         min = i;  
5         for (j = i+1; j <= r; j++)  
6             if (v[j] < v[min])  
7                 min = j;  
8         troca(&v[i], &v[min]);  
9     }  
10 }
```

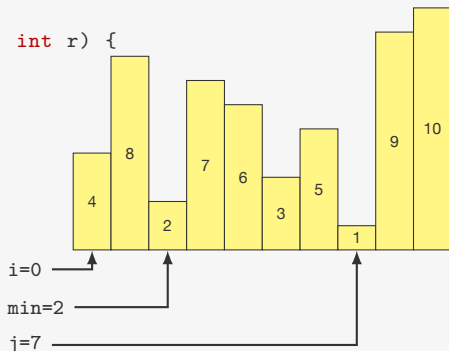


# Ordenação por Seleção

Ideia:

- Trocar  $v[1]$  com o mínimo de  $v[1], v[1+1], \dots, v[r]$
- Trocar  $v[1+1]$  com o mínimo de  $v[1+1], v[1+2], \dots, v[r]$
- ...
- Trocar  $v[1+i]$  com o mínimo de  $v[1+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {
2   int i, j, min;
3   for (i = l; i < r; i++) {
4     min = i;
5     for (j = i+1; j <= r; j++)
6       if (v[j] < v[min])
7         min = j;
8     troca(&v[i], &v[min]);
9   }
10 }
```



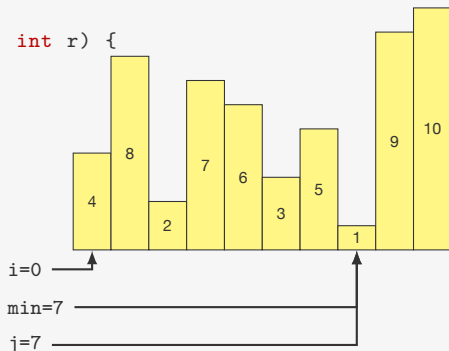


# Ordenação por Seleção

Ideia:

- Trocar  $v[1]$  com o mínimo de  $v[1], v[1+1], \dots, v[r]$
- Trocar  $v[1+1]$  com o mínimo de  $v[1+1], v[1+2], \dots, v[r]$
- ...
- Trocar  $v[1+i]$  com o mínimo de  $v[1+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {  
2     int i, j, min;  
3     for (i = l; i < r; i++) {  
4         min = i;  
5         for (j = i+1; j <= r; j++)  
6             if (v[j] < v[min])  
7                 min = j;  
8         troca(&v[i], &v[min]);  
9     }  
10 }
```

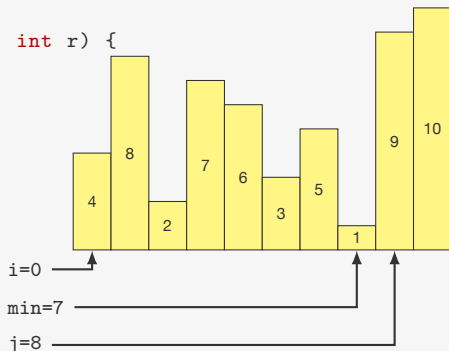


# Ordenação por Seleção

Ideia:

- Trocar  $v[l]$  com o mínimo de  $v[l], v[l+1], \dots, v[r]$
- Trocar  $v[l+1]$  com o mínimo de  $v[l+1], v[l+2], \dots, v[r]$
- ...
- Trocar  $v[l+i]$  com o mínimo de  $v[l+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {  
2     int i, j, min;  
3     for (i = l; i < r; i++) {  
4         min = i;  
5         for (j = i+1; j <= r; j++)  
6             if (v[j] < v[min])  
7                 min = j;  
8         troca(&v[i], &v[min]);  
9     }  
10 }
```

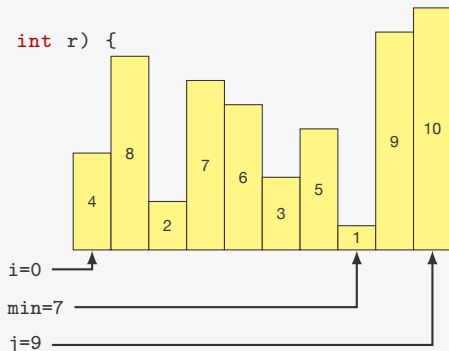


# Ordenação por Seleção

Ideia:

- Trocar  $v[l]$  com o mínimo de  $v[l], v[l+1], \dots, v[r]$
- Trocar  $v[l+1]$  com o mínimo de  $v[l+1], v[l+2], \dots, v[r]$
- ...
- Trocar  $v[l+i]$  com o mínimo de  $v[l+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {  
2     int i, j, min;  
3     for (i = l; i < r; i++) {  
4         min = i;  
5         for (j = i+1; j <= r; j++)  
6             if (v[j] < v[min])  
7                 min = j;  
8         troca(&v[i], &v[min]);  
9     }  
10 }
```

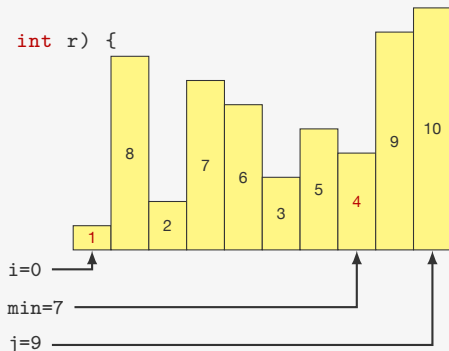


# Ordenação por Seleção

Ideia:

- Trocar  $v[1]$  com o mínimo de  $v[1], v[1+1], \dots, v[r]$
- Trocar  $v[1+1]$  com o mínimo de  $v[1+1], v[1+2], \dots, v[r]$
- ...
- Trocar  $v[1+i]$  com o mínimo de  $v[1+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {  
2     int i, j, min;  
3     for (i = l; i < r; i++) {  
4         min = i;  
5         for (j = i+1; j <= r; j++)  
6             if (v[j] < v[min])  
7                 min = j;  
8         troca(&v[i], &v[min]);  
9     }  
10 }
```

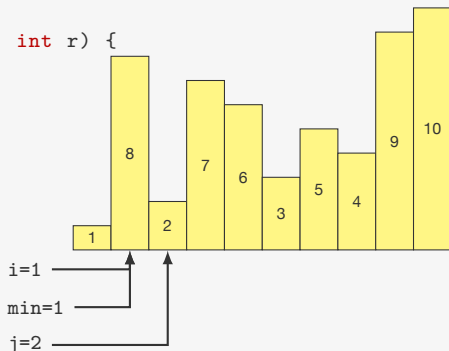


# Ordenação por Seleção

Ideia:

- Trocar  $v[1]$  com o mínimo de  $v[1], v[1+1], \dots, v[r]$
- Trocar  $v[1+1]$  com o mínimo de  $v[1+1], v[1+2], \dots, v[r]$
- ...
- Trocar  $v[1+i]$  com o mínimo de  $v[1+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {  
2     int i, j, min;  
3     for (i = l; i < r; i++) {  
4         min = i;  
5         for (j = i+1; j <= r; j++)  
6             if (v[j] < v[min])  
7                 min = j;  
8         troca(&v[i], &v[min]);  
9     }  
10 }
```

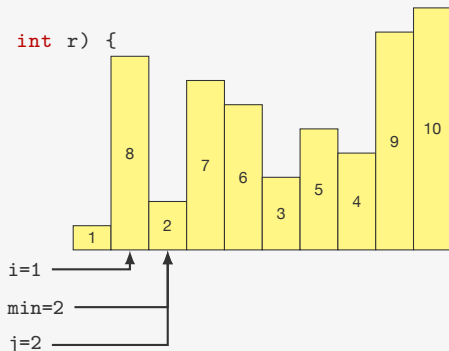


# Ordenação por Seleção

Ideia:

- Trocar  $v[1]$  com o mínimo de  $v[1], v[1+1], \dots, v[r]$
- Trocar  $v[1+1]$  com o mínimo de  $v[1+1], v[1+2], \dots, v[r]$
- ...
- Trocar  $v[1+i]$  com o mínimo de  $v[1+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {
2   int i, j, min;
3   for (i = l; i < r; i++) {
4     min = i;
5     for (j = i+1; j <= r; j++)
6       if (v[j] < v[min])
7         min = j;
8     troca(&v[i], &v[min]);
9   }
10 }
```

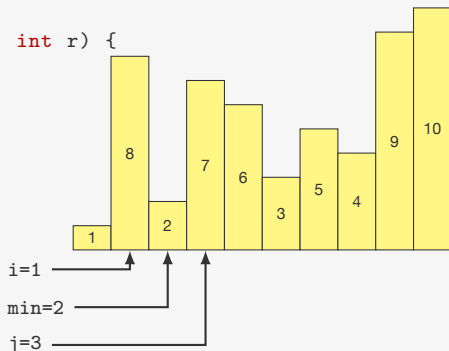


# Ordenação por Seleção

Ideia:

- Trocar  $v[1]$  com o mínimo de  $v[1], v[1+1], \dots, v[r]$
- Trocar  $v[1+1]$  com o mínimo de  $v[1+1], v[1+2], \dots, v[r]$
- ...
- Trocar  $v[1+i]$  com o mínimo de  $v[1+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {  
2     int i, j, min;  
3     for (i = l; i < r; i++) {  
4         min = i;  
5         for (j = i+1; j <= r; j++)  
6             if (v[j] < v[min])  
7                 min = j;  
8         troca(&v[i], &v[min]);  
9     }  
10 }
```

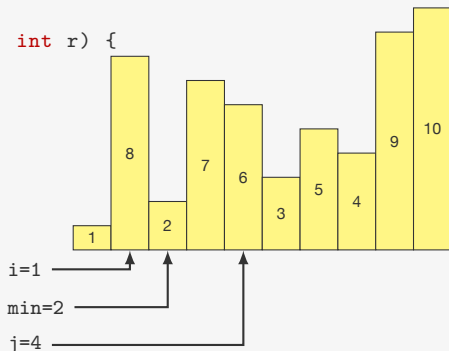


# Ordenação por Seleção

Ideia:

- Trocar  $v[1]$  com o mínimo de  $v[1], v[1+1], \dots, v[r]$
- Trocar  $v[1+1]$  com o mínimo de  $v[1+1], v[1+2], \dots, v[r]$
- ...
- Trocar  $v[1+i]$  com o mínimo de  $v[1+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {  
2     int i, j, min;  
3     for (i = l; i < r; i++) {  
4         min = i;  
5         for (j = i+1; j <= r; j++)  
6             if (v[j] < v[min])  
7                 min = j;  
8         troca(&v[i], &v[min]);  
9     }  
10 }
```



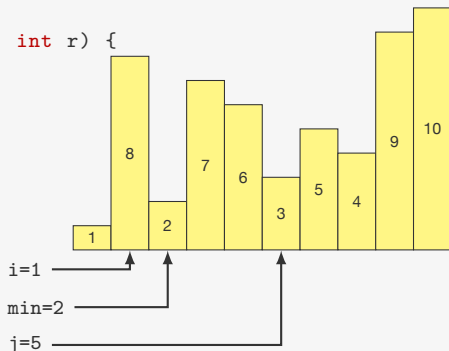


# Ordenação por Seleção

Ideia:

- Trocar  $v[1]$  com o mínimo de  $v[1], v[1+1], \dots, v[r]$
- Trocar  $v[1+1]$  com o mínimo de  $v[1+1], v[1+2], \dots, v[r]$
- ...
- Trocar  $v[1+i]$  com o mínimo de  $v[1+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {
2   int i, j, min;
3   for (i = l; i < r; i++) {
4     min = i;
5     for (j = i+1; j <= r; j++)
6       if (v[j] < v[min])
7         min = j;
8     troca(&v[i], &v[min]);
9   }
10 }
```

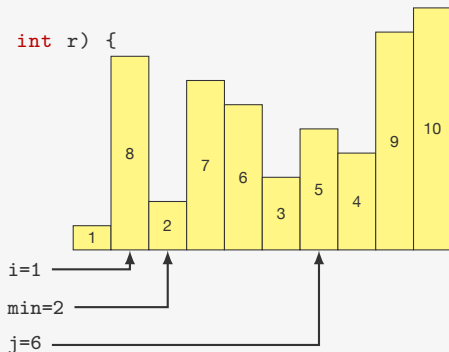


# Ordenação por Seleção

Ideia:

- Trocar  $v[1]$  com o mínimo de  $v[1], v[1+1], \dots, v[r]$
- Trocar  $v[1+1]$  com o mínimo de  $v[1+1], v[1+2], \dots, v[r]$
- ...
- Trocar  $v[1+i]$  com o mínimo de  $v[1+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {  
2     int i, j, min;  
3     for (i = l; i < r; i++) {  
4         min = i;  
5         for (j = i+1; j <= r; j++)  
6             if (v[j] < v[min])  
7                 min = j;  
8         troca(&v[i], &v[min]);  
9     }  
10 }
```

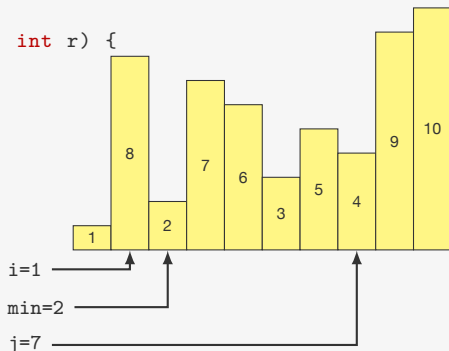


# Ordenação por Seleção

Ideia:

- Trocar  $v[1]$  com o mínimo de  $v[1], v[1+1], \dots, v[r]$
- Trocar  $v[1+1]$  com o mínimo de  $v[1+1], v[1+2], \dots, v[r]$
- ...
- Trocar  $v[1+i]$  com o mínimo de  $v[1+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {  
2     int i, j, min;  
3     for (i = l; i < r; i++) {  
4         min = i;  
5         for (j = i+1; j <= r; j++)  
6             if (v[j] < v[min])  
7                 min = j;  
8         troca(&v[i], &v[min]);  
9     }  
10 }
```

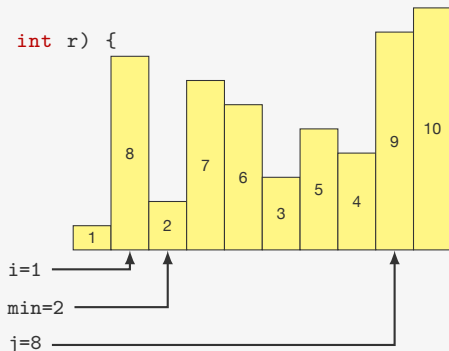


# Ordenação por Seleção

Ideia:

- Trocar  $v[1]$  com o mínimo de  $v[1], v[1+1], \dots, v[r]$
- Trocar  $v[1+1]$  com o mínimo de  $v[1+1], v[1+2], \dots, v[r]$
- ...
- Trocar  $v[1+i]$  com o mínimo de  $v[1+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {  
2     int i, j, min;  
3     for (i = l; i < r; i++) {  
4         min = i;  
5         for (j = i+1; j <= r; j++)  
6             if (v[j] < v[min])  
7                 min = j;  
8         troca(&v[i], &v[min]);  
9     }  
10 }
```

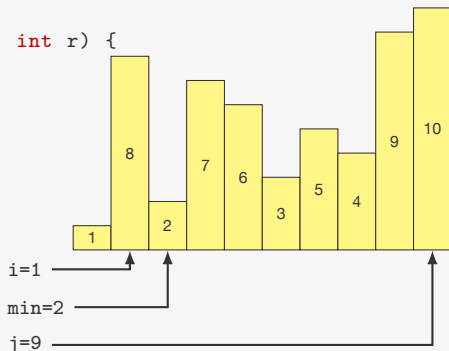


# Ordenação por Seleção

Ideia:

- Trocar  $v[1]$  com o mínimo de  $v[1], v[1+1], \dots, v[r]$
- Trocar  $v[1+1]$  com o mínimo de  $v[1+1], v[1+2], \dots, v[r]$
- ...
- Trocar  $v[1+i]$  com o mínimo de  $v[1+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {  
2     int i, j, min;  
3     for (i = l; i < r; i++) {  
4         min = i;  
5         for (j = i+1; j <= r; j++)  
6             if (v[j] < v[min])  
7                 min = j;  
8         troca(&v[i], &v[min]);  
9     }  
10 }
```

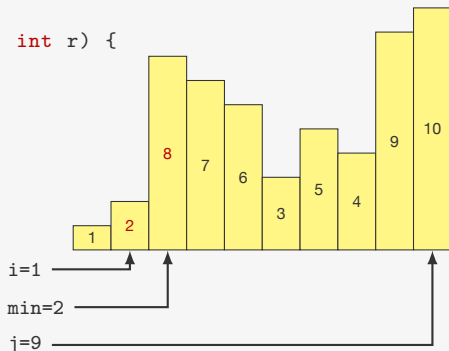


# Ordenação por Seleção

Ideia:

- Trocar  $v[1]$  com o mínimo de  $v[1], v[1+1], \dots, v[r]$
- Trocar  $v[1+1]$  com o mínimo de  $v[1+1], v[1+2], \dots, v[r]$
- ...
- Trocar  $v[1+i]$  com o mínimo de  $v[1+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {  
2     int i, j, min;  
3     for (i = l; i < r; i++) {  
4         min = i;  
5         for (j = i+1; j <= r; j++)  
6             if (v[j] < v[min])  
7                 min = j;  
8         troca(&v[i], &v[min]);  
9     }  
10 }
```

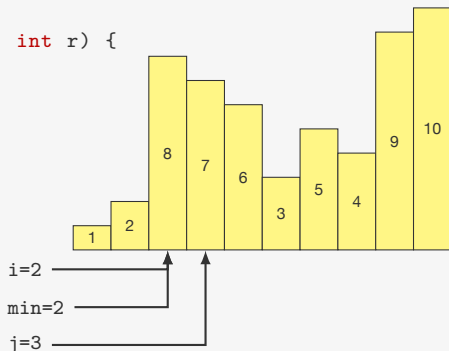


# Ordenação por Seleção

Ideia:

- Trocar  $v[l]$  com o mínimo de  $v[l], v[l+1], \dots, v[r]$
- Trocar  $v[l+1]$  com o mínimo de  $v[l+1], v[l+2], \dots, v[r]$
- ...
- Trocar  $v[l+i]$  com o mínimo de  $v[l+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {  
2     int i, j, min;  
3     for (i = l; i < r; i++) {  
4         min = i;  
5         for (j = i+1; j <= r; j++)  
6             if (v[j] < v[min])  
7                 min = j;  
8         troca(&v[i], &v[min]);  
9     }  
10 }
```

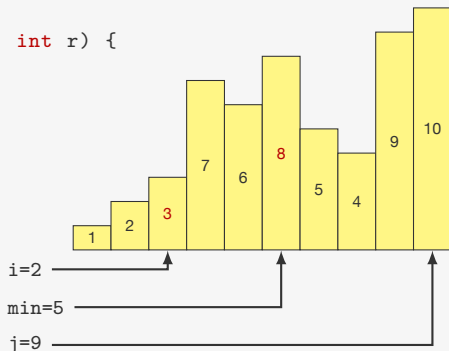


# Ordenação por Seleção

Ideia:

- Trocar  $v[l]$  com o mínimo de  $v[l], v[l+1], \dots, v[r]$
- Trocar  $v[l+1]$  com o mínimo de  $v[l+1], v[l+2], \dots, v[r]$
- ...
- Trocar  $v[l+i]$  com o mínimo de  $v[l+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {  
2     int i, j, min;  
3     for (i = l; i < r; i++) {  
4         min = i;  
5         for (j = i+1; j <= r; j++)  
6             if (v[j] < v[min])  
7                 min = j;  
8         troca(&v[i], &v[min]);  
9     }  
10 }
```



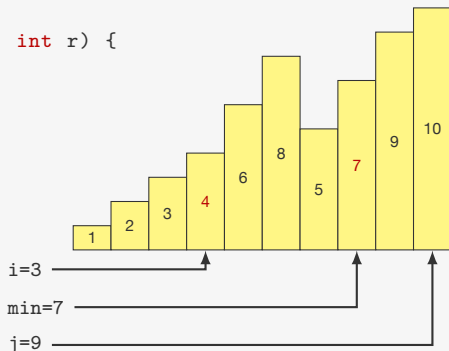


# Ordenação por Seleção

Ideia:

- Trocar  $v[l]$  com o mínimo de  $v[l], v[l+1], \dots, v[r]$
- Trocar  $v[l+1]$  com o mínimo de  $v[l+1], v[l+2], \dots, v[r]$
- ...
- Trocar  $v[l+i]$  com o mínimo de  $v[l+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {  
2     int i, j, min;  
3     for (i = l; i < r; i++) {  
4         min = i;  
5         for (j = i+1; j <= r; j++)  
6             if (v[j] < v[min])  
7                 min = j;  
8         troca(&v[i], &v[min]);  
9     }  
10 }
```

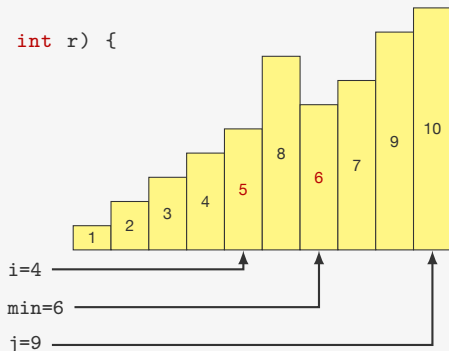


# Ordenação por Seleção

Ideia:

- Trocar  $v[1]$  com o mínimo de  $v[1], v[1+1], \dots, v[r]$
- Trocar  $v[1+1]$  com o mínimo de  $v[1+1], v[1+2], \dots, v[r]$
- ...
- Trocar  $v[1+i]$  com o mínimo de  $v[1+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {
2     int i, j, min;
3     for (i = l; i < r; i++) {
4         min = i;
5         for (j = i+1; j <= r; j++)
6             if (v[j] < v[min])
7                 min = j;
8         troca(&v[i], &v[min]);
9     }
10 }
```

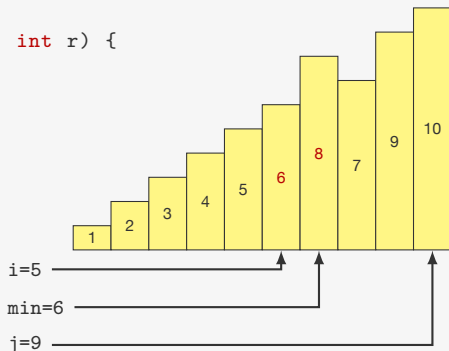


# Ordenação por Seleção

Ideia:

- Trocar  $v[1]$  com o mínimo de  $v[1], v[1+1], \dots, v[r]$
- Trocar  $v[1+1]$  com o mínimo de  $v[1+1], v[1+2], \dots, v[r]$
- ...
- Trocar  $v[1+i]$  com o mínimo de  $v[1+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {  
2     int i, j, min;  
3     for (i = l; i < r; i++) {  
4         min = i;  
5         for (j = i+1; j <= r; j++)  
6             if (v[j] < v[min])  
7                 min = j;  
8         troca(&v[i], &v[min]);  
9     }  
10 }
```

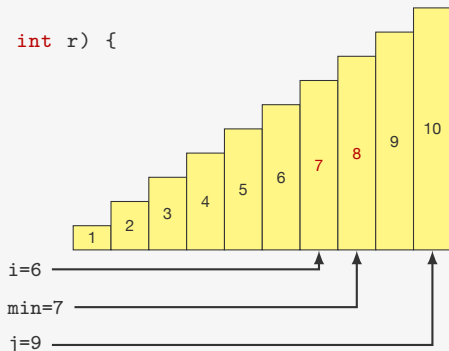


# Ordenação por Seleção

Ideia:

- Trocar  $v[1]$  com o mínimo de  $v[1], v[1+1], \dots, v[r]$
- Trocar  $v[1+1]$  com o mínimo de  $v[1+1], v[1+2], \dots, v[r]$
- ...
- Trocar  $v[1+i]$  com o mínimo de  $v[1+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {  
2     int i, j, min;  
3     for (i = l; i < r; i++) {  
4         min = i;  
5         for (j = i+1; j <= r; j++)  
6             if (v[j] < v[min])  
7                 min = j;  
8         troca(&v[i], &v[min]);  
9     }  
10 }
```

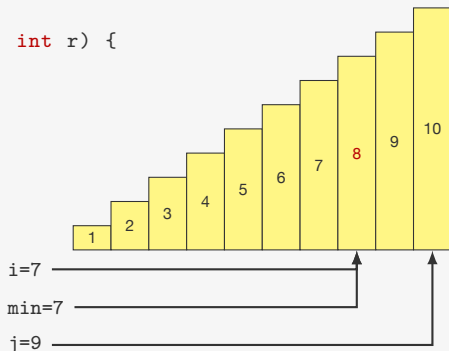


# Ordenação por Seleção

Ideia:

- Trocar  $v[l]$  com o mínimo de  $v[l], v[l+1], \dots, v[r]$
- Trocar  $v[l+1]$  com o mínimo de  $v[l+1], v[l+2], \dots, v[r]$
- ...
- Trocar  $v[l+i]$  com o mínimo de  $v[l+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {  
2     int i, j, min;  
3     for (i = l; i < r; i++) {  
4         min = i;  
5         for (j = i+1; j <= r; j++)  
6             if (v[j] < v[min])  
7                 min = j;  
8         troca(&v[i], &v[min]);  
9     }  
10 }
```

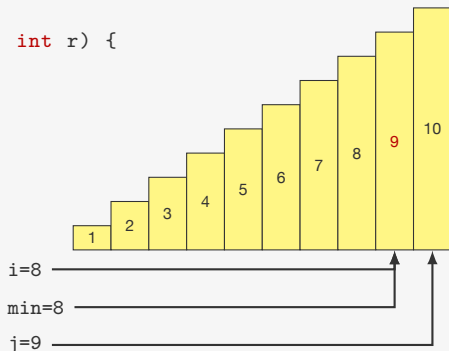


# Ordenação por Seleção

Ideia:

- Trocar  $v[l]$  com o mínimo de  $v[l], v[l+1], \dots, v[r]$
- Trocar  $v[l+1]$  com o mínimo de  $v[l+1], v[l+2], \dots, v[r]$
- ...
- Trocar  $v[l+i]$  com o mínimo de  $v[l+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {  
2     int i, j, min;  
3     for (i = l; i < r; i++) {  
4         min = i;  
5         for (j = i+1; j <= r; j++)  
6             if (v[j] < v[min])  
7                 min = j;  
8         troca(&v[i], &v[min]);  
9     }  
10 }
```

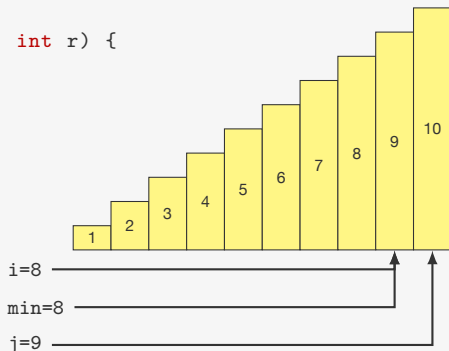


# Ordenação por Seleção

Ideia:

- Trocar  $v[l]$  com o mínimo de  $v[l], v[l+1], \dots, v[r]$
- Trocar  $v[l+1]$  com o mínimo de  $v[l+1], v[l+2], \dots, v[r]$
- ...
- Trocar  $v[l+i]$  com o mínimo de  $v[l+i], \dots, v[r]$

```
1 void selectionsort(int *v, int l, int r) {  
2     int i, j, min;  
3     for (i = l; i < r; i++) {  
4         min = i;  
5         for (j = i+1; j <= r; j++)  
6             if (v[j] < v[min])  
7                 min = j;  
8         troca(&v[i], &v[min]);  
9     }  
10 }
```



# Ordenação por Seleção

```
1 void selectionsort(int *v, int l, int r) {
2     int i, j, min;
3     for (i = l; i < r; i++) {
4         min = i;
5         for (j = i+1; j <= r; j++)
6             if (v[j] < v[min])
7                 min = j;
8         troca(&v[i], &v[min]);
9     }
10 }
```



# Ordenação por Seleção

```
1 void selectionsort(int *v, int l, int r) {
2     int i, j, min;
3     for (i = l; i < r; i++) {
4         min = i;
5         for (j = i+1; j <= r; j++)
6             if (v[j] < v[min])
7                 min = j;
8         troca(&v[i], &v[min]);
9     }
10 }
```

- número de comparações:

$$(n - 1) + (n - 2) + \dots + 1 = n(n - 1)/2 = O(n^2)$$

# Ordenação por Seleção

```
1 void selectionsort(int *v, int l, int r) {
2     int i, j, min;
3     for (i = l; i < r; i++) {
4         min = i;
5         for (j = i+1; j <= r; j++)
6             if (v[j] < v[min])
7                 min = j;
8         troca(&v[i], &v[min]);
9     }
10 }
```

- número de comparações:

$$(n - 1) + (n - 2) + \dots + 1 = n(n - 1)/2 = O(n^2)$$

- número de trocas:  $n - 1 = O(n)$

# Ordenação por Seleção

```
1 void selectionsort(int *v, int l, int r) {
2     int i, j, min;
3     for (i = l; i < r; i++) {
4         min = i;
5         for (j = i+1; j <= r; j++)
6             if (v[j] < v[min])
7                 min = j;
8         troca(&v[i], &v[min]);
9     }
10 }
```

- número de comparações:

$$(n - 1) + (n - 2) + \dots + 1 = n(n - 1)/2 = O(n^2)$$

- número de trocas:  $n - 1 = O(n)$ 
  - Muito bom quando trocas são muito caras

# Ordenação por Seleção

```
1 void selectionsort(int *v, int l, int r) {
2     int i, j, min;
3     for (i = l; i < r; i++) {
4         min = i;
5         for (j = i+1; j <= r; j++)
6             if (v[j] < v[min])
7                 min = j;
8         troca(&v[i], &v[min]);
9     }
10 }
```

- número de comparações:

$$(n - 1) + (n - 2) + \dots + 1 = n(n - 1)/2 = O(n^2)$$

- número de trocas:  $n - 1 = O(n)$ 
  - Muito bom quando trocas são muito caras
  - Porém, talvez seja melhor usar ponteiros nesse caso

# Ordenação por Seleção

```
1 void selectionsort(int *v, int l, int r) {
2     int i, j, min;
3     for (i = l; i < r; i++) {
4         min = i;
5         for (j = i+1; j <= r; j++)
6             if (v[j] < v[min])
7                 min = j;
8         troca(&v[i], &v[min]);
9     }
10 }
```

- número de comparações:

$$(n - 1) + (n - 2) + \dots + 1 = n(n - 1)/2 = O(n^2)$$

- número de trocas:  $n - 1 = O(n)$ 
  - Muito bom quando trocas são muito caras
  - Porém, talvez seja melhor usar ponteiros nesse caso
- É um algoritmo **não-adaptativo**

# Ordenação por Seleção

```
1 void selectionsort(int *v, int l, int r) {
2     int i, j, min;
3     for (i = l; i < r; i++) {
4         min = i;
5         for (j = i+1; j <= r; j++)
6             if (v[j] < v[min])
7                 min = j;
8         troca(&v[i], &v[min]);
9     }
10 }
```

- número de comparações:

$$(n - 1) + (n - 2) + \dots + 1 = n(n - 1)/2 = O(n^2)$$

- número de trocas:  $n - 1 = O(n)$ 
  - Muito bom quando trocas são muito caras
  - Porém, talvez seja melhor usar ponteiros nesse caso
- É um algoritmo **não-adaptativo**
  - o conteúdo do vetor não importa para o tempo de execução

# Ordenação por Seleção

```
1 void selectionsort(int *v, int l, int r) {
2     int i, j, min;
3     for (i = l; i < r; i++) {
4         min = i;
5         for (j = i+1; j <= r; j++)
6             if (v[j] < v[min])
7                 min = j;
8         troca(&v[i], &v[min]);
9     }
10 }
```

- número de comparações:

$$(n - 1) + (n - 2) + \dots + 1 = n(n - 1)/2 = O(n^2)$$

- número de trocas:  $n - 1 = O(n)$ 
  - Muito bom quando trocas são muito caras
  - Porém, talvez seja melhor usar ponteiros nesse caso
- É um algoritmo **não-adaptativo**
  - o conteúdo do vetor não importa para o tempo de execução
  - sempre faz  $n(n - 1)/2$  comparações e  $n - 1$  trocas

# BubbleSort

Ideia:



# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos

# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- eventualmente, encontramos o elemento mais leve

# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- eventualmente, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- eventualmente, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int l, int r) {
```

# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- eventualmente, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int l, int r) {
```

# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- eventualmente, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l; i < r; i++)
```

# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- eventualmente, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l; i < r; i++)  
4         for (j = r; j > i; j--)
```

# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- eventualmente, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int l, int r) {
2     int i, j;
3     for (i = l; i < r; i++)
4         for (j = r; j > i; j--)
5             if (v[j] < v[j-1])
6                 troca(&v[j-1], &v[j]);
7 }
```

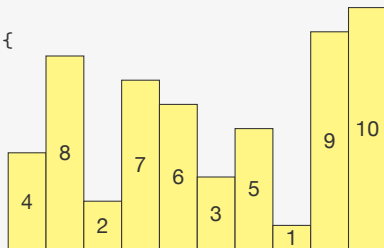


# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- eventualmente, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l; i < r; i++)  
4         for (j = r; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```



i

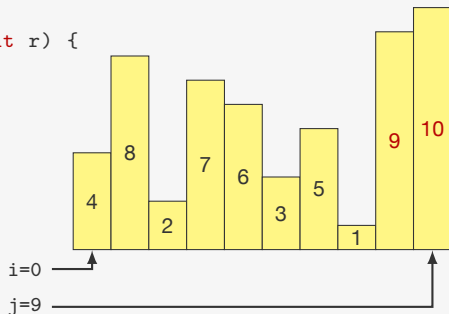
j

# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- eventualmente, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l; i < r; i++)  
4         for (j = r; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

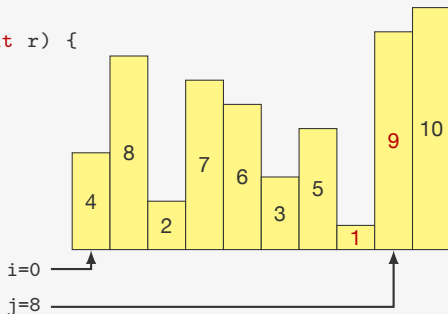


# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- eventualmente, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l; i < r; i++)  
4         for (j = r; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

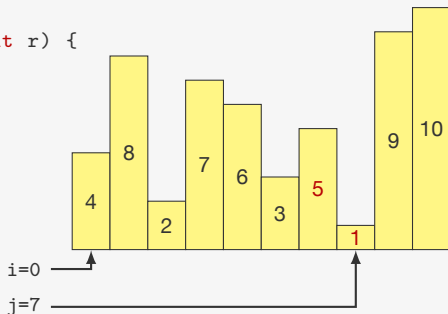


# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- eventualmente, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l; i < r; i++)  
4         for (j = r; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

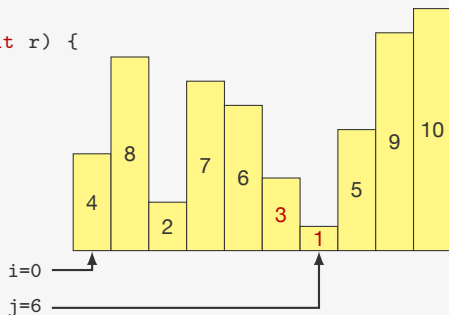


# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- eventualmente, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l; i < r; i++)  
4         for (j = r; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

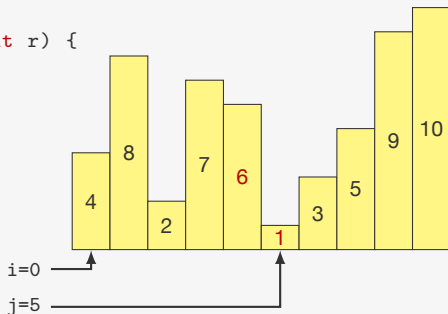


# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- eventualmente, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l; i < r; i++)  
4         for (j = r; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

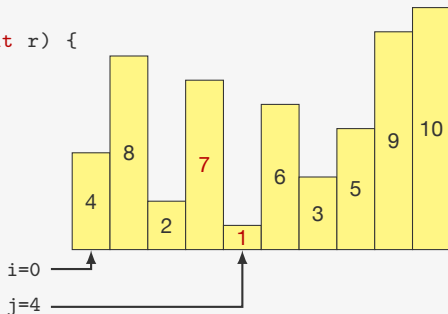


# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- eventualmente, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l; i < r; i++)  
4         for (j = r; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

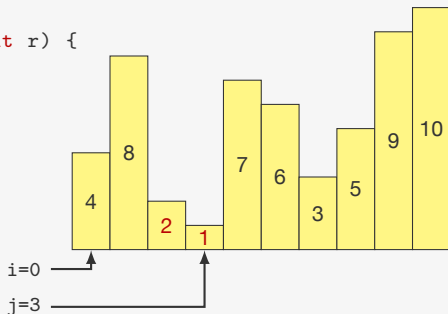


# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- eventualmente, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l; i < r; i++)  
4         for (j = r; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```



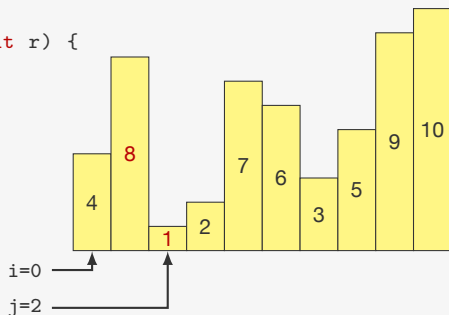


# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- eventualmente, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l; i < r; i++)  
4         for (j = r; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

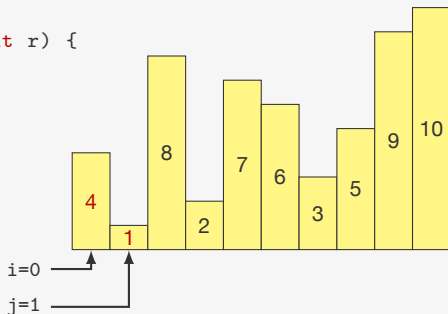


# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- eventualmente, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l; i < r; i++)  
4         for (j = r; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

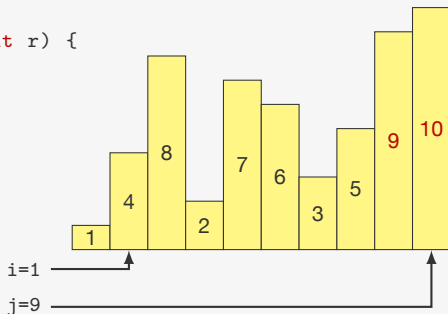


# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- eventualmente, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l; i < r; i++)  
4         for (j = r; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

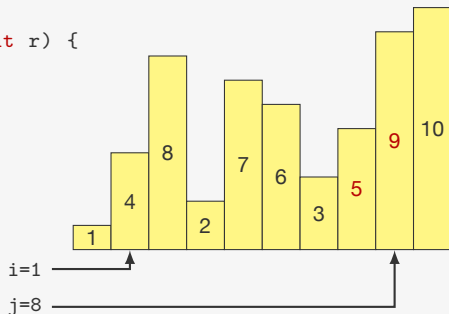


# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- eventualmente, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l; i < r; i++)  
4         for (j = r; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

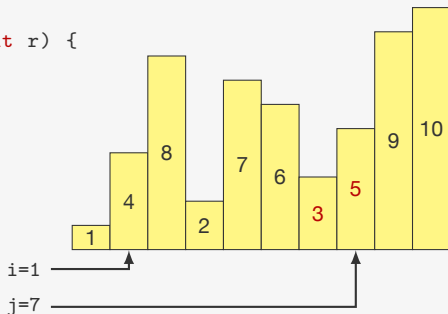


# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- eventualmente, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l; i < r; i++)  
4         for (j = r; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

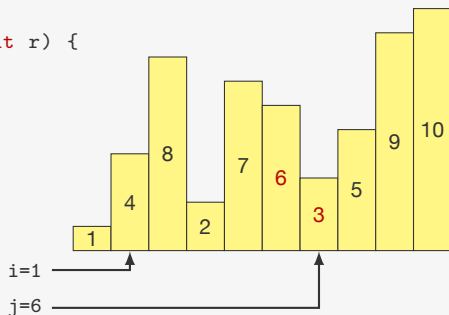


# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- eventualmente, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l; i < r; i++)  
4         for (j = r; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

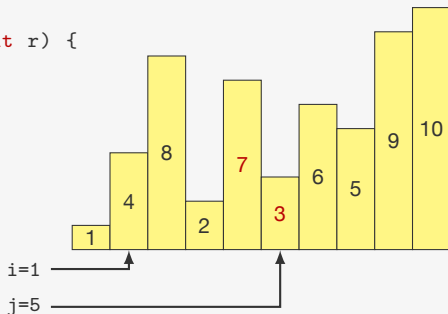


# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- eventualmente, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l; i < r; i++)  
4         for (j = r; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

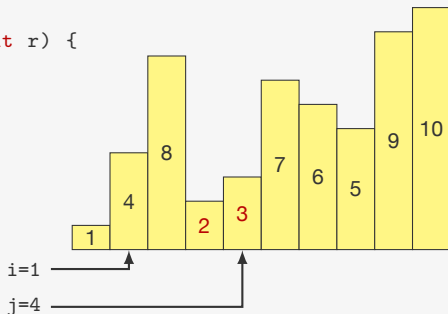


# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- eventualmente, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l; i < r; i++)  
4         for (j = r; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```



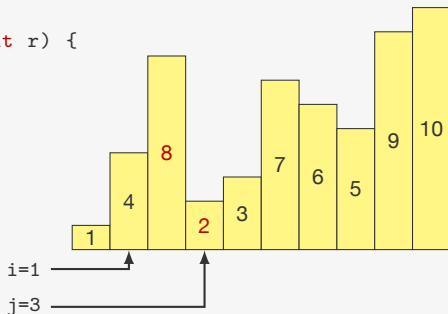


# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- eventualmente, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l; i < r; i++)  
4         for (j = r; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

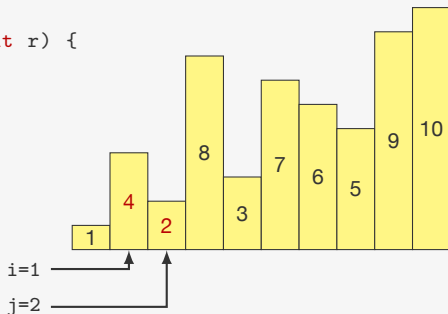


# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- eventualmente, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l; i < r; i++)  
4         for (j = r; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

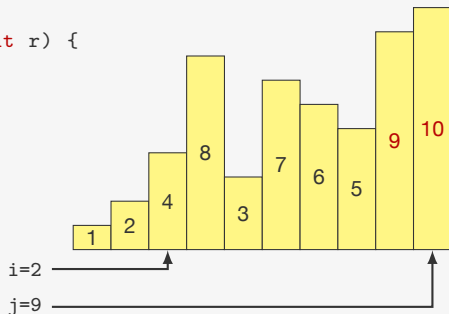


# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- eventualmente, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l; i < r; i++)  
4         for (j = r; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

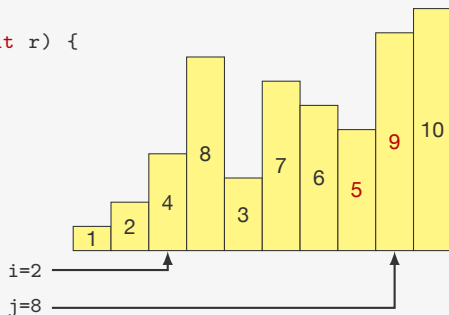


# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- eventualmente, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l; i < r; i++)  
4         for (j = r; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

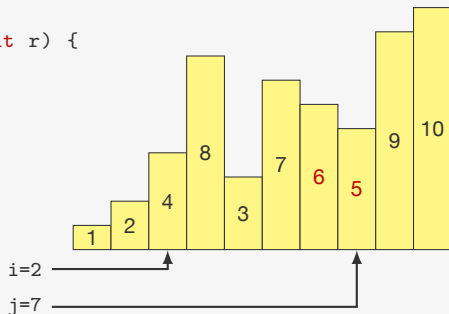


# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- eventualmente, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l; i < r; i++)  
4         for (j = r; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

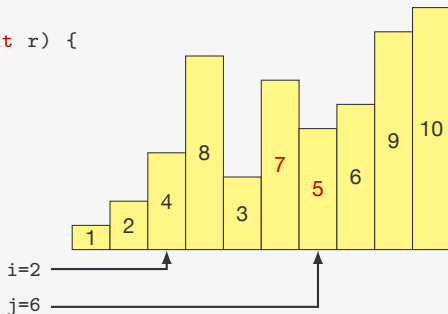


# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- eventualmente, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l; i < r; i++)  
4         for (j = r; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

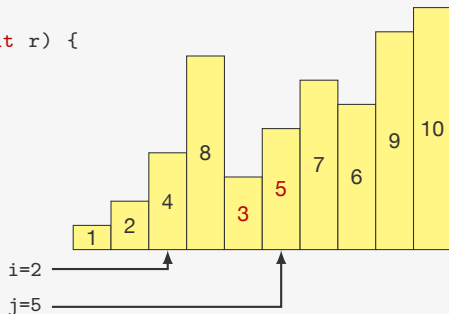


# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- eventualmente, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l; i < r; i++)  
4         for (j = r; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

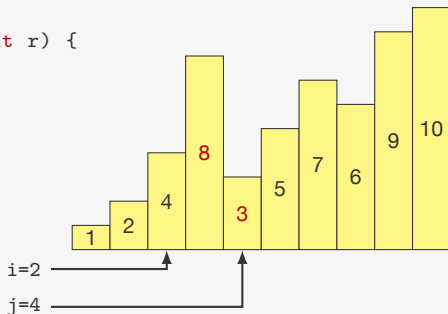


# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- eventualmente, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l; i < r; i++)  
4         for (j = r; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```



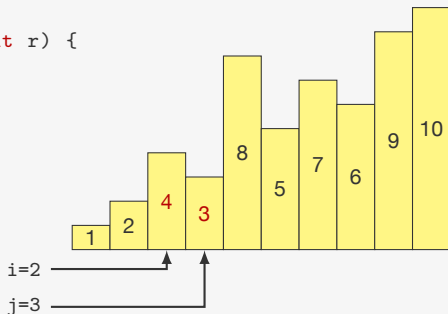


# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- eventualmente, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l; i < r; i++)  
4         for (j = r; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

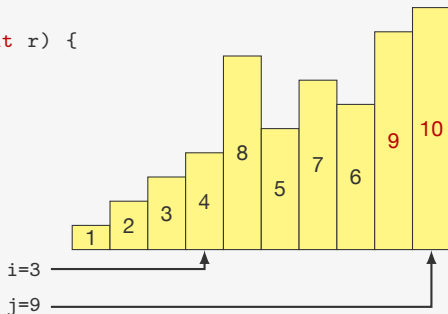


# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- eventualmente, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l; i < r; i++)  
4         for (j = r; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

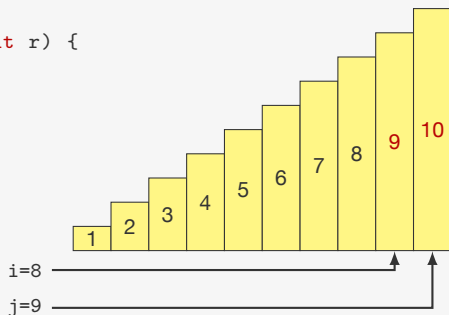


# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- eventualmente, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l; i < r; i++)  
4         for (j = r; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

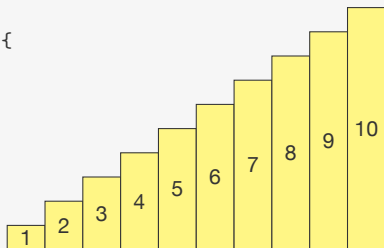


# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- eventualmente, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l; i < r; i++)  
4         for (j = r; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```



i

j

## BubbleSort Adaptativo

Se não aconteceu nenhuma troca, podemos parar o algoritmo

# BubbleSort Adaptativo

Se não aconteceu nenhuma troca, podemos parar o algoritmo

```
1 void bubblesort_v2(int *v, int l, int r) {
2     int i, j, trocou = 1;
3     for (i = l; i < r && trocou; i++){
4         trocou = 0;
5         for (j = r; j > i; j--)
6             if (v[j] < v[j-1]) {
7                 troca(&v[j-1], &v[j]);
8                 trocou = 1;
9             }
10    }
11 }
```

# BubbleSort Adaptativo

Se não aconteceu nenhuma troca, podemos parar o algoritmo

```
1 void bubblesort_v2(int *v, int l, int r) {
2     int i, j, trocou = 1;
3     for (i = l; i < r && trocou; i++){
4         trocou = 0;
5         for (j = r; j > i; j--)
6             if (v[j] < v[j-1]) {
7                 troca(&v[j-1], &v[j]);
8                 trocou = 1;
9             }
10    }
11 }
```

No pior caso toda comparação gera uma troca:

# BubbleSort Adaptativo

Se não aconteceu nenhuma troca, podemos parar o algoritmo

```
1 void bubblesort_v2(int *v, int l, int r) {
2     int i, j, trocou = 1;
3     for (i = l; i < r && trocou; i++){
4         trocou = 0;
5         for (j = r; j > i; j--)
6             if (v[j] < v[j-1]) {
7                 troca(&v[j-1], &v[j]);
8                 trocou = 1;
9             }
10    }
11 }
```

No pior caso toda comparação gera uma troca:

- comparações:  $n(n-1)/2 = O(n^2)$



# BubbleSort Adaptativo

Se não aconteceu nenhuma troca, podemos parar o algoritmo

```
1 void bubblesort_v2(int *v, int l, int r) {
2     int i, j, trocou = 1;
3     for (i = l; i < r && trocou; i++){
4         trocou = 0;
5         for (j = r; j > i; j--)
6             if (v[j] < v[j-1]) {
7                 troca(&v[j-1], &v[j]);
8                 trocou = 1;
9             }
10    }
11 }
```

No pior caso toda comparação gera uma troca:

- comparações:  $n(n-1)/2 = O(n^2)$
- trocas:  $n(n-1)/2 = O(n^2)$

# BubbleSort Adaptativo

Se não aconteceu nenhuma troca, podemos parar o algoritmo

```
1 void bubblesort_v2(int *v, int l, int r) {
2     int i, j, trocou = 1;
3     for (i = l; i < r && trocou; i++){
4         trocou = 0;
5         for (j = r; j > i; j--){
6             if (v[j] < v[j-1]) {
7                 troca(&v[j-1], &v[j]);
8                 trocou = 1;
9             }
10    }
11 }
```

No pior caso toda comparação gera uma troca:

- comparações:  $n(n-1)/2 = O(n^2)$
- trocas:  $n(n-1)/2 = O(n^2)$

No caso médio:

# BubbleSort Adaptativo

Se não aconteceu nenhuma troca, podemos parar o algoritmo

```
1 void bubblesort_v2(int *v, int l, int r) {
2     int i, j, trocou = 1;
3     for (i = l; i < r && trocou; i++){
4         trocou = 0;
5         for (j = r; j > i; j--){
6             if (v[j] < v[j-1]) {
7                 troca(&v[j-1], &v[j]);
8                 trocou = 1;
9             }
10    }
11 }
```

No pior caso toda comparação gera uma troca:

- comparações:  $n(n-1)/2 = O(n^2)$
- trocas:  $n(n-1)/2 = O(n^2)$

No caso médio:

- comparações:  $\approx n^2/2 = O(n^2)$

# BubbleSort Adaptativo

Se não aconteceu nenhuma troca, podemos parar o algoritmo

```
1 void bubblesort_v2(int *v, int l, int r) {
2     int i, j, trocou = 1;
3     for (i = l; i < r && trocou; i++){
4         trocou = 0;
5         for (j = r; j > i; j--)
6             if (v[j] < v[j-1]) {
7                 troca(&v[j-1], &v[j]);
8                 trocou = 1;
9             }
10    }
11 }
```

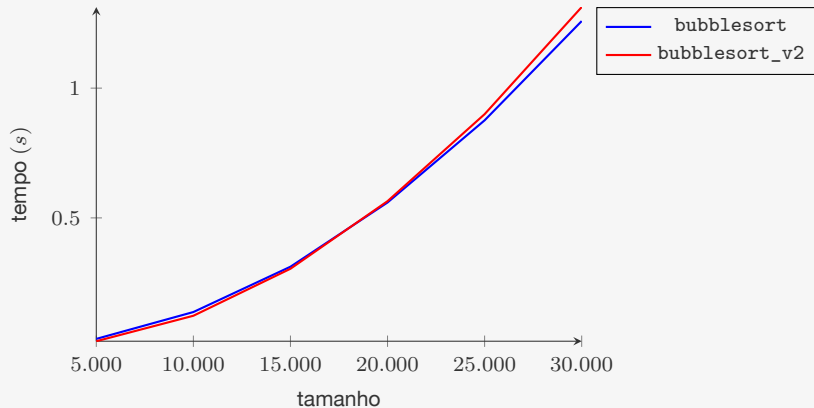
No pior caso toda comparação gera uma troca:

- comparações:  $n(n-1)/2 = O(n^2)$
- trocas:  $n(n-1)/2 = O(n^2)$

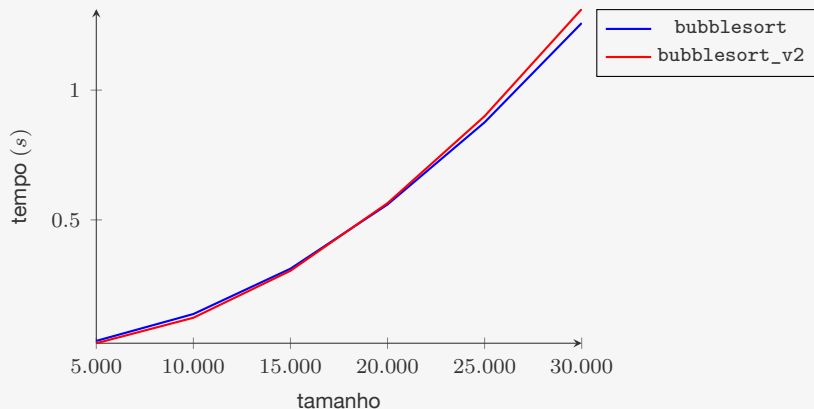
No caso médio:

- comparações:  $\approx n^2/2 = O(n^2)$
- trocas:  $\approx n^2/2 = O(n^2)$

# Gráfico de comparação do BubbleSort

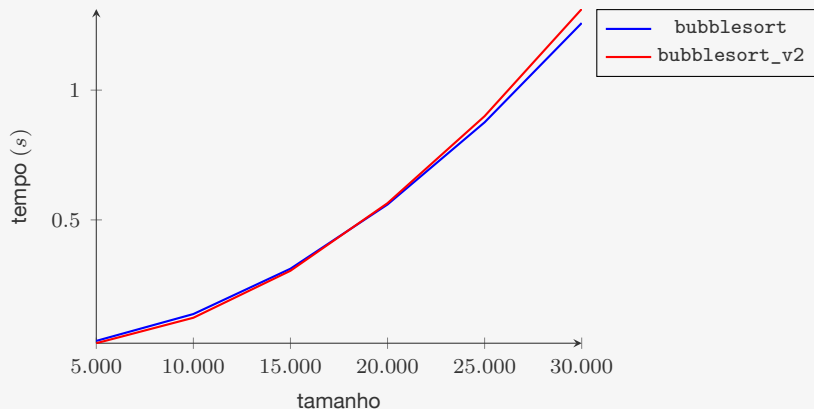


## Gráfico de comparação do BubbleSort



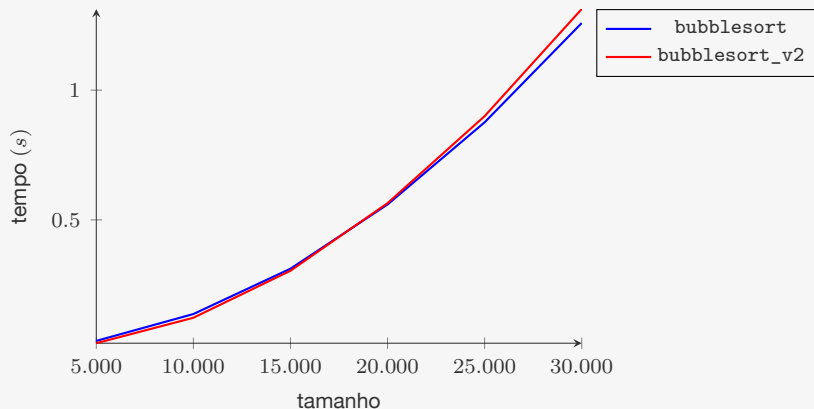
- O adaptativo foi pior para vetores aleatórios

# Gráfico de comparação do BubbleSort



- O adaptativo foi pior para vetores aleatórios
  - por causa overhead de registrar se ocorreu alguma troca

# Gráfico de comparação do BubbleSort



- O adaptativo foi pior para vetores aleatórios
  - por causa overhead de registrar se ocorreu alguma troca
- Mas pode ser melhor em vetores parcialmente ordenados



# Ordenação por Inserção

Ideia:

# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado

# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta

# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort

# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {
```

# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {
```

# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l+1; i <= r; i++)
```



# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l+1; i <= r; i++)  
4         for (j = i; j > l; j--)
```

# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

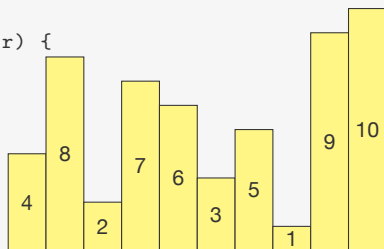
```
1 void insertionsort(int *v, int l, int r) {
2     int i, j;
3     for (i = l+1; i <= r; i++)
4         for (j = i; j > l; j--)
5             if (v[j] < v[j-1])
6                 troca(&v[j], &v[j-1]);
7 }
```

# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {
2     int i, j;
3     for (i = l+1; i <= r; i++)
4         for (j = i; j > l; j--)
5             if (v[j] < v[j-1])
6                 troca(&v[j], &v[j-1]);
7 }
```



i

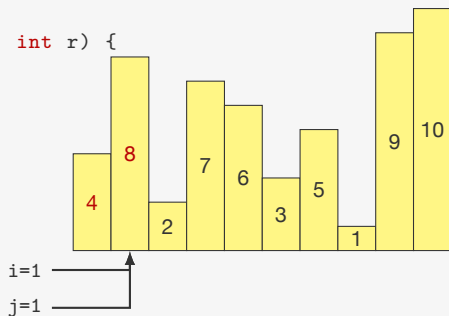
j

# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l+1; i <= r; i++)  
4         for (j = i; j > l; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

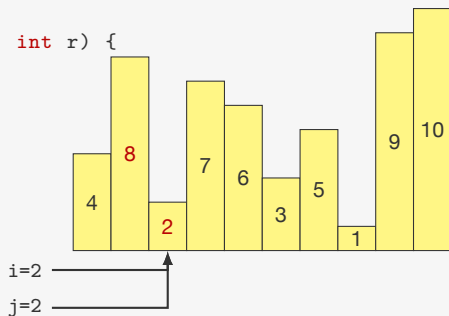


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l+1; i <= r; i++)  
4         for (j = i; j > l; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

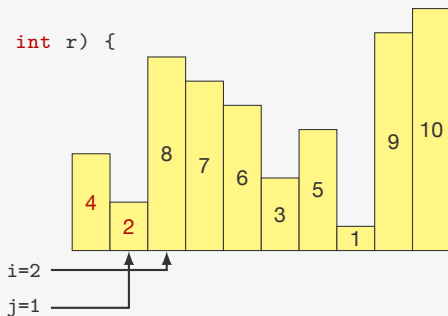


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l+1; i <= r; i++)  
4         for (j = i; j > l; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

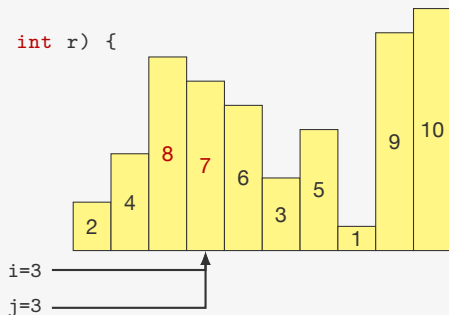


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l+1; i <= r; i++)  
4         for (j = i; j > l; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

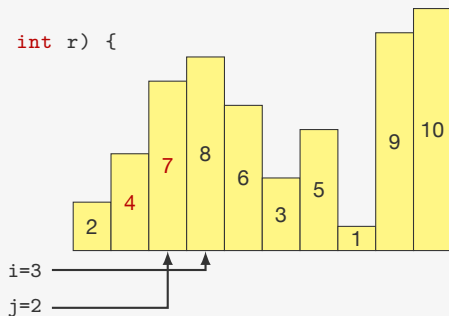


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l+1; i <= r; i++)  
4         for (j = i; j > l; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```



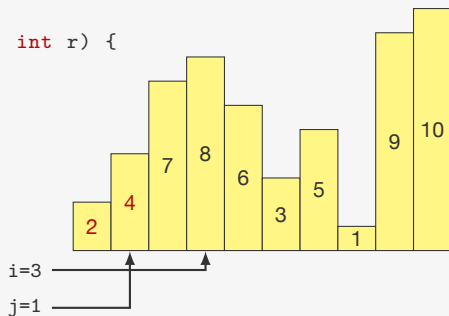


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l+1; i <= r; i++)  
4         for (j = i; j > l; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

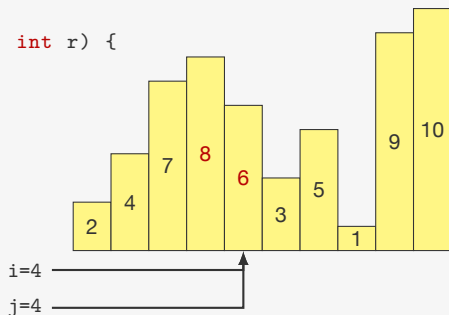


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l+1; i <= r; i++)  
4         for (j = i; j > l; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

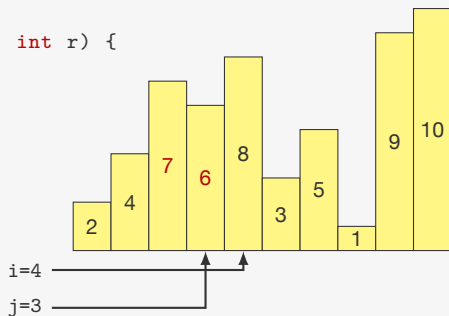


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l+1; i <= r; i++)  
4         for (j = i; j > l; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

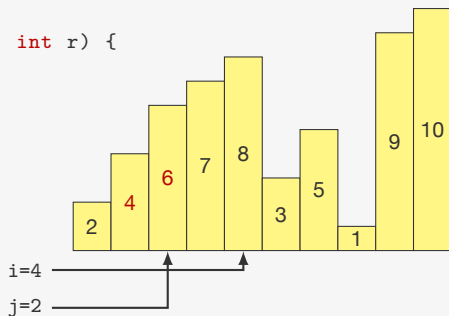


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l+1; i <= r; i++)  
4         for (j = i; j > l; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

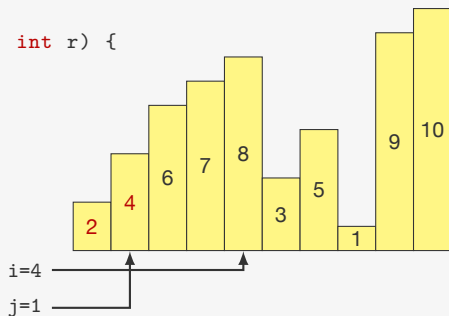


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l+1; i <= r; i++)  
4         for (j = i; j > l; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

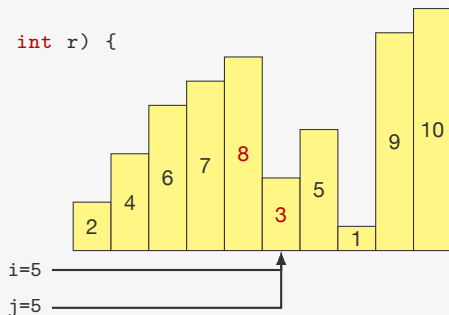


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l+1; i <= r; i++)  
4         for (j = i; j > l; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

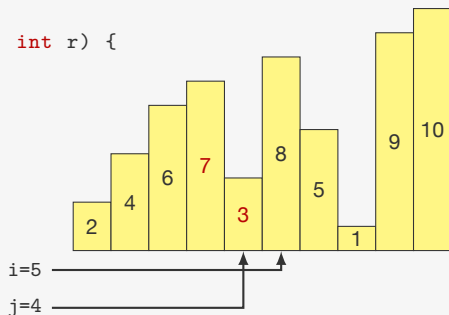


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l+1; i <= r; i++)  
4         for (j = i; j > l; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

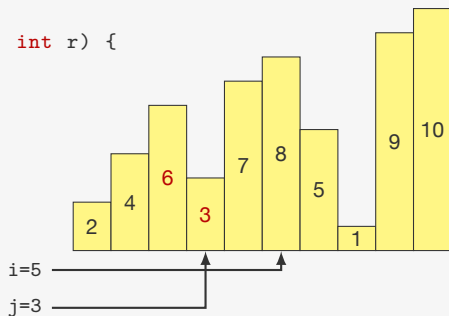


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l+1; i <= r; i++)  
4         for (j = i; j > l; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```



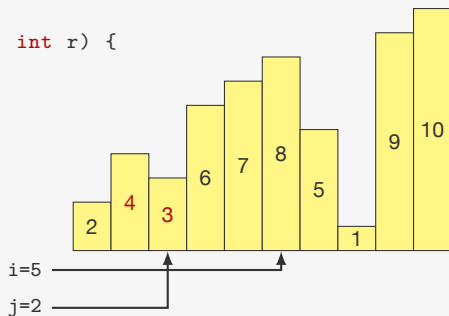


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l+1; i <= r; i++)  
4         for (j = i; j > l; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

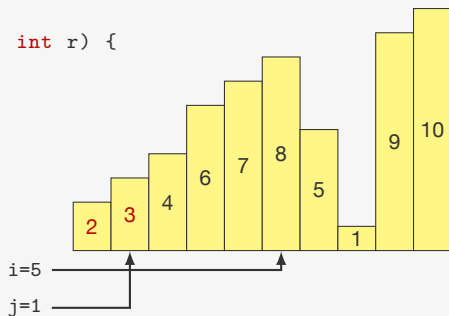


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l+1; i <= r; i++)  
4         for (j = i; j > l; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

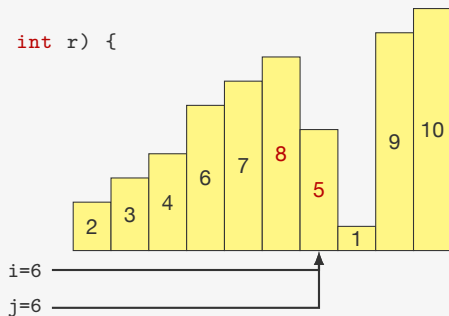


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l+1; i <= r; i++)  
4         for (j = i; j > l; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

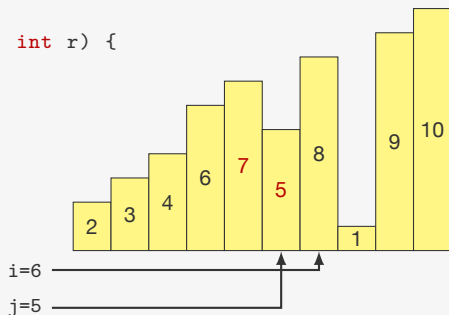


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l+1; i <= r; i++)  
4         for (j = i; j > l; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

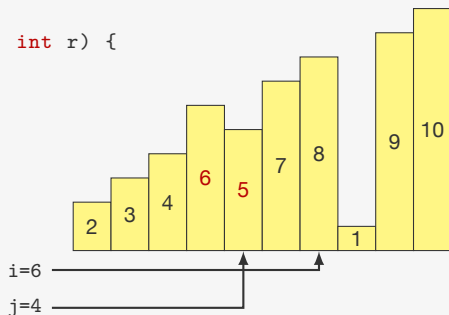


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l+1; i <= r; i++)  
4         for (j = i; j > l; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

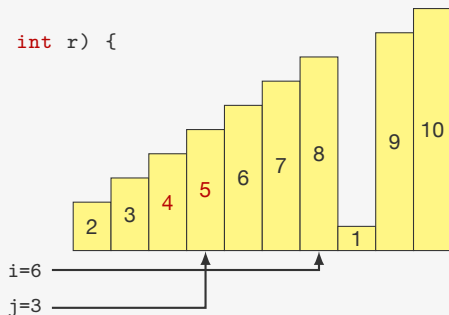


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {
2   int i, j;
3   for (i = l+1; i <= r; i++)
4     for (j = i; j > l; j--)
5       if (v[j] < v[j-1])
6         troca(&v[j], &v[j-1]);
7 }
```

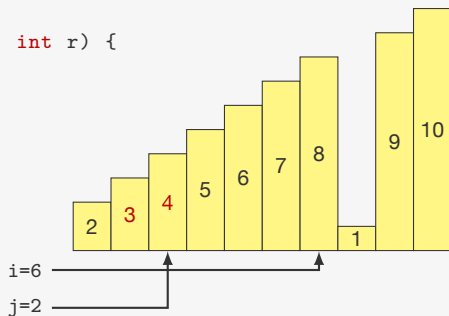


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {
2   int i, j;
3   for (i = l+1; i <= r; i++)
4     for (j = i; j > l; j--)
5       if (v[j] < v[j-1])
6         troca(&v[j], &v[j-1]);
7 }
```

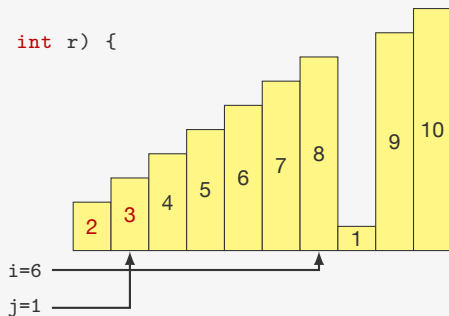


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {
2   int i, j;
3   for (i = l+1; i <= r; i++)
4     for (j = i; j > l; j--)
5       if (v[j] < v[j-1])
6         troca(&v[j], &v[j-1]);
7 }
```



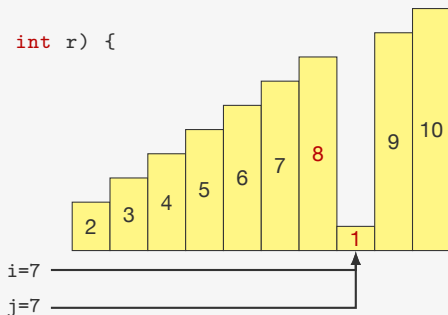


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l+1; i <= r; i++)  
4         for (j = i; j > l; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

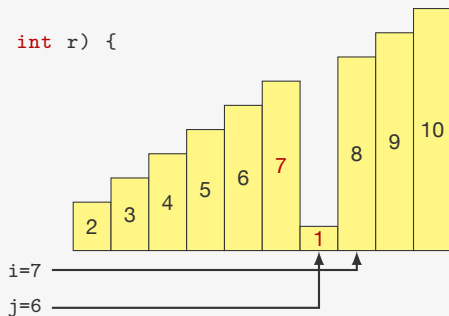


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {
2   int i, j;
3   for (i = l+1; i <= r; i++)
4     for (j = i; j > l; j--)
5       if (v[j] < v[j-1])
6         troca(&v[j], &v[j-1]);
7 }
```

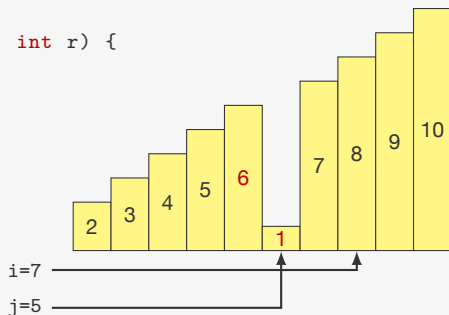


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l+1; i <= r; i++)  
4         for (j = i; j > l; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

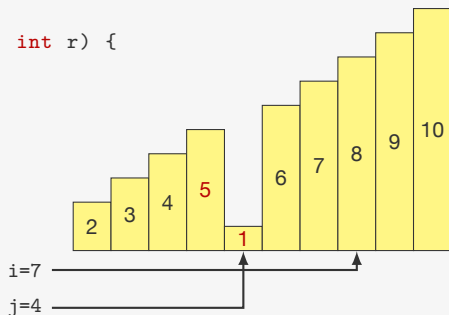


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {
2   int i, j;
3   for (i = l+1; i <= r; i++)
4     for (j = i; j > l; j--)
5       if (v[j] < v[j-1])
6         troca(&v[j], &v[j-1]);
7 }
```

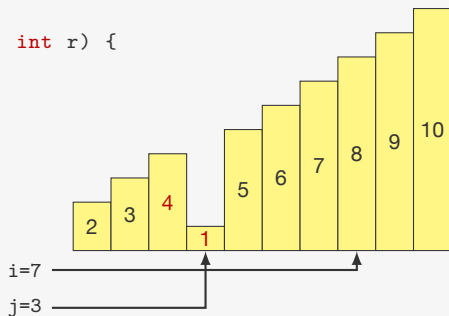


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {
2   int i, j;
3   for (i = l+1; i <= r; i++)
4     for (j = i; j > l; j--)
5       if (v[j] < v[j-1])
6         troca(&v[j], &v[j-1]);
7 }
```

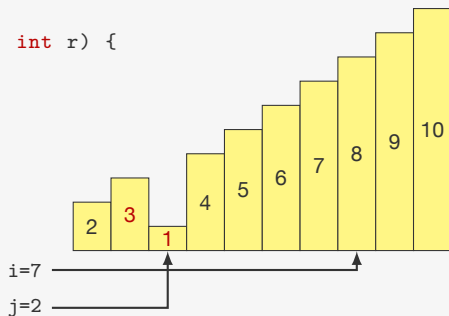


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l+1; i <= r; i++)  
4         for (j = i; j > l; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

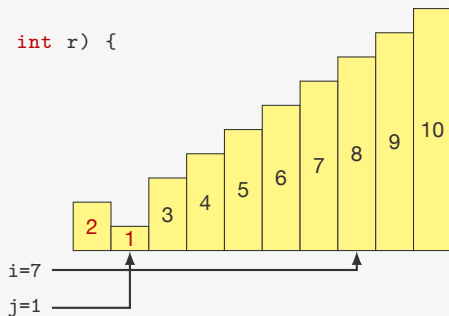


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l+1; i <= r; i++)  
4         for (j = i; j > l; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

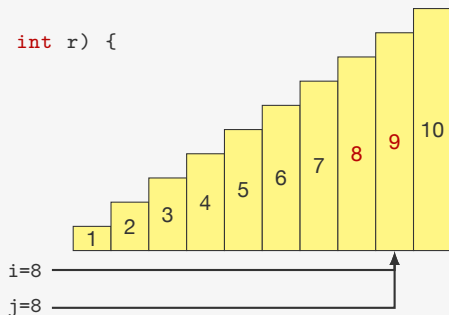


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {
2   int i, j;
3   for (i = l+1; i <= r; i++)
4     for (j = i; j > l; j--)
5       if (v[j] < v[j-1])
6         troca(&v[j], &v[j-1]);
7 }
```



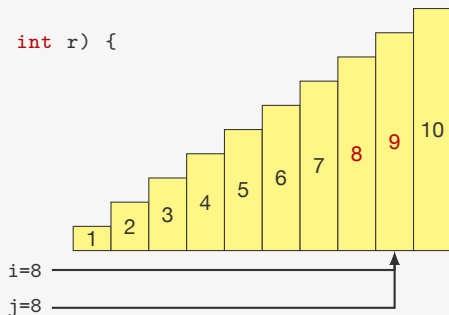


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l+1; i <= r; i++)  
4         for (j = i; j > l; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

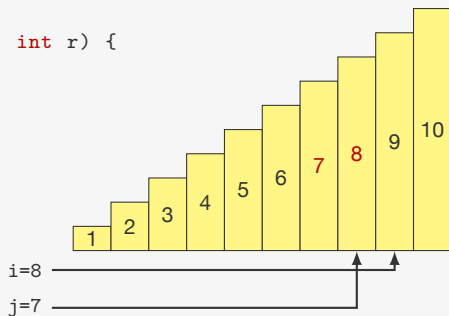


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l+1; i <= r; i++)  
4         for (j = i; j > l; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

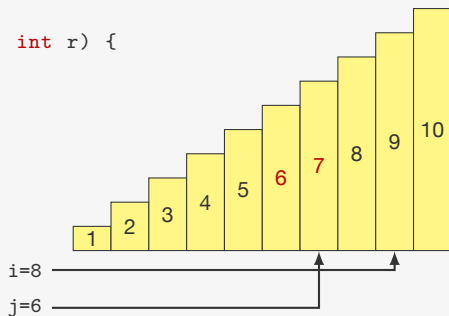


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l+1; i <= r; i++)  
4         for (j = i; j > l; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

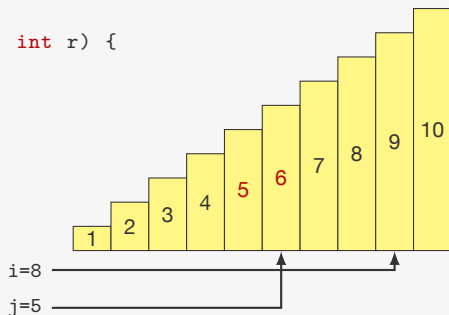


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l+1; i <= r; i++)  
4         for (j = i; j > l; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

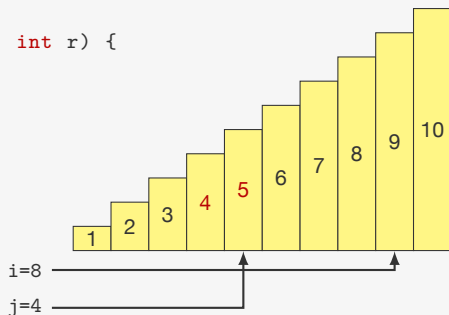


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l+1; i <= r; i++)  
4         for (j = i; j > l; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

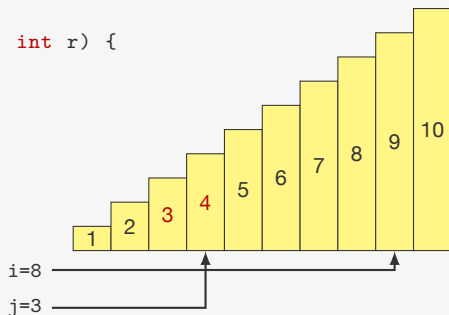


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l+1; i <= r; i++)  
4         for (j = i; j > l; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

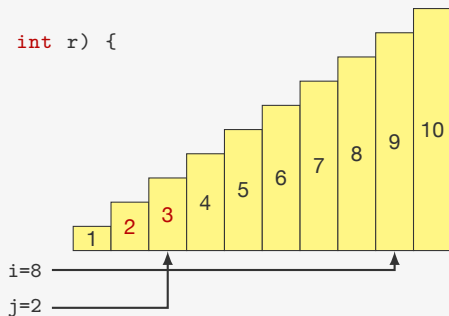


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l+1; i <= r; i++)  
4         for (j = i; j > l; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

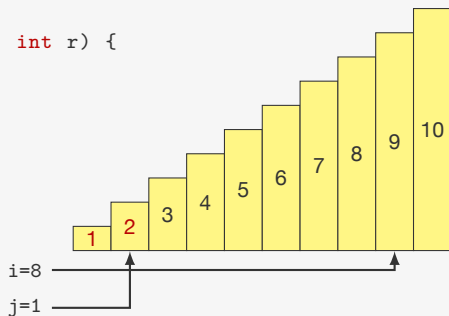


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {
2   int i, j;
3   for (i = l+1; i <= r; i++)
4     for (j = i; j > l; j--)
5       if (v[j] < v[j-1])
6         troca(&v[j], &v[j-1]);
7 }
```



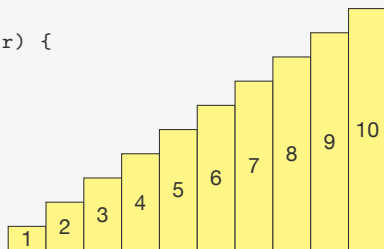


# Ordenação por Inserção

Ideia:

- Se já temos  $v[1], v[1+1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[1], v[1+1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int l, int r) {  
2     int i, j;  
3     for (i = l+1; i <= r; i++)  
4         for (j = i; j > l; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```



i

j

# Ordenação por Inserção - Versão Adaptativa

```
1 void insertionsort(int *v, int l, int r) {
2     int i, j;
3     for (i = l+1; i <= r; i++)
4         for (j = i; j > l; j--)
5             if (v[j] < v[j-1])
6                 troca(&v[j], &v[j-1]);
7 }
```

## Ordenação por Inserção - Versão Adaptativa

```
1 void insertionsort(int *v, int l, int r) {
2     int i, j;
3     for (i = l+1; i <= r; i++)
4         for (j = i; j > l; j--)
5             if (v[j] < v[j-1])
6                 troca(&v[j], &v[j-1]);
7 }
```

Quando o elemento já está na sua posição correta não é necessário mais percorrer o vetor testando se  $v[j] < v[j-1]$

## Ordenação por Inserção - Versão Adaptativa

```
1 void insertionsort(int *v, int l, int r) {
2     int i, j;
3     for (i = l+1; i <= r; i++)
4         for (j = i; j > l; j--)
5             if (v[j] < v[j-1])
6                 troca(&v[j], &v[j-1]);
7 }
```

Quando o elemento já está na sua posição correta não é necessário mais percorrer o vetor testando se  $v[j] < v[j-1]$

```
1 void insertionsort_v2(int *v, int l, int r) {
2     int i, j;
3     for (i = l+1; i <= r; i++)
4         for (j = i; j > l && v[j] < v[j-1]; j--)
5             troca(&v[j], &v[j-1]);
6 }
```

# Ordenação por Inserção - Versão Adaptativa

```
1 void insertionsort(int *v, int l, int r) {
2     int i, j;
3     for (i = l+1; i <= r; i++)
4         for (j = i; j > l; j--)
5             if (v[j] < v[j-1])
6                 troca(&v[j], &v[j-1]);
7 }
```

Quando o elemento já está na sua posição correta não é necessário mais percorrer o vetor testando se  $v[j] < v[j-1]$

```
1 void insertionsort_v2(int *v, int l, int r) {
2     int i, j;
3     for (i = l+1; i <= r; i++)
4         for (j = i; j > l && v[j] < v[j-1]; j--)
5             troca(&v[j], &v[j-1]);
6 }
```

O algoritmo se torna adaptativo!

## Ordenação por Inserção - Segunda Otimização

```
1 void insertionsort_v2(int *v, int l, int r) {
2     int i, j;
3     for (i = l+1; i <= r; i++)
4         for (j = i; j > l && v[j] < v[j-1]; j--)
5             troca(&v[j], &v[j-1]);
6 }
```

## Ordenação por Inserção - Segunda Otimização

```
1 void insertionsort_v2(int *v, int l, int r) {
2     int i, j;
3     for (i = l+1; i <= r; i++)
4         for (j = i; j > l && v[j] < v[j-1]; j--)
5             troca(&v[j], &v[j-1]);
6 }
```

Se trocamos  $v[j]$  com  $v[j-1]$  e  $v[j-1]$  com  $v[j-2]$

## Ordenação por Inserção - Segunda Otimização

```
1 void insertionsort_v2(int *v, int l, int r) {
2     int i, j;
3     for (i = l+1; i <= r; i++)
4         for (j = i; j > l && v[j] < v[j-1]; j--)
5             troca(&v[j], &v[j-1]);
6 }
```

Se trocamos  $v[j]$  com  $v[j-1]$  e  $v[j-1]$  com  $v[j-2]$

- fazemos 3 atribuições para cada troca = 6 atribuições



## Ordenação por Inserção - Segunda Otimização

```
1 void insertionsort_v2(int *v, int l, int r) {
2     int i, j;
3     for (i = l+1; i <= r; i++)
4         for (j = i; j > l && v[j] < v[j-1]; j--)
5             troca(&v[j], &v[j-1]);
6 }
```

Se trocamos  $v[j]$  com  $v[j-1]$  e  $v[j-1]$  com  $v[j-2]$

- fazemos 3 atribuições para cada troca = 6 atribuições
- é melhor fazer:

$t = v[j]; v[j] = v[j-1]; v[j-1] = v[j-2]; v[j-2] = t;$

## Ordenação por Inserção - Segunda Otimização

```
1 void insertionsort_v2(int *v, int l, int r) {
2     int i, j;
3     for (i = l+1; i <= r; i++)
4         for (j = i; j > l && v[j] < v[j-1]; j--)
5             troca(&v[j], &v[j-1]);
6 }
```

Se trocamos  $v[j]$  com  $v[j-1]$  e  $v[j-1]$  com  $v[j-2]$

- fazemos 3 atribuições para cada troca = 6 atribuições
- é melhor fazer:

$t = v[j]; v[j] = v[j-1]; v[j-1] = v[j-2]; v[j-2] = t;$

```
1 void insertionsort_v3(int *v, int l, int r) {
2     int i, j, t;
3     for (i = l+1; i <= r; i++) {
4         t = v[i];
5         for (j = i; j > l && t < v[j-1]; j--)
6             v[j] = v[j-1];
7         v[j] = t;
8     }
9 }
```

## Ordenação por Inserção - Análise

```
1 void insertionsort_v3(int *v, int l, int r) {
2     int i, j, t;
3     for (i = l+1; i <= r; i++) {
4         t = v[i];
5         for (j = i; j > l && t < v[j-1]; j--)
6             v[j] = v[j-1];
7         v[j] = t;
8     }
9 }
```

## Ordenação por Inserção - Análise

```
1 void insertionsort_v3(int *v, int l, int r) {
2     int i, j, t;
3     for (i = l+1; i <= r; i++) {
4         t = v[i];
5         for (j = i; j > l && t < v[j-1]; j--)
6             v[j] = v[j-1];
7         v[j] = t;
8     }
9 }
```

No caso pior caso:

## Ordenação por Inserção - Análise

```
1 void insertionsort_v3(int *v, int l, int r) {
2     int i, j, t;
3     for (i = l+1; i <= r; i++) {
4         t = v[i];
5         for (j = i; j > l && t < v[j-1]; j--)
6             v[j] = v[j-1];
7         v[j] = t;
8     }
9 }
```

No caso pior caso:

- comparações:  $\approx n^2/2 = O(n^2)$

## Ordenação por Inserção - Análise

```
1 void insertionsort_v3(int *v, int l, int r) {
2     int i, j, t;
3     for (i = l+1; i <= r; i++) {
4         t = v[i];
5         for (j = i; j > l && t < v[j-1]; j--)
6             v[j] = v[j-1];
7         v[j] = t;
8     }
9 }
```

No caso pior caso:

- comparações:  $\approx n^2/2 = O(n^2)$
- atribuições (ao invés de trocas):  $\approx n^2/2 = O(n^2)$

# Ordenação por Inserção - Análise

```
1 void insertionsort_v3(int *v, int l, int r) {
2     int i, j, t;
3     for (i = l+1; i <= r; i++) {
4         t = v[i];
5         for (j = i; j > l && t < v[j-1]; j--)
6             v[j] = v[j-1];
7         v[j] = t;
8     }
9 }
```

No caso pior caso:

- comparações:  $\approx n^2/2 = O(n^2)$
- atribuições (ao invés de trocas):  $\approx n^2/2 = O(n^2)$

No caso médio é metade disso:

# Ordenação por Inserção - Análise

```
1 void insertionsort_v3(int *v, int l, int r) {
2     int i, j, t;
3     for (i = l+1; i <= r; i++) {
4         t = v[i];
5         for (j = i; j > l && t < v[j-1]; j--)
6             v[j] = v[j-1];
7         v[j] = t;
8     }
9 }
```

No caso pior caso:

- comparações:  $\approx n^2/2 = O(n^2)$
- atribuições (ao invés de trocas):  $\approx n^2/2 = O(n^2)$

No caso médio é metade disso:

- cada elemento anda metade do prefixo do vetor em média



## Ordenação por Inserção - Terceira Otimização

```
1 void insertionsort_v3(int *v, int l, int r) {
2     int i, j, t;
3     for (i = l+1; i <= r; i++) {
4         t = v[i];
5         for (j = i; j > l && t < v[j-1]; j--)
6             v[j] = v[j-1];
7         v[j] = t;
8     }
9 }
```

## Ordenação por Inserção - Terceira Otimização

```
1 void insertionsort_v3(int *v, int l, int r) {
2     int i, j, t;
3     for (i = l+1; i <= r; i++) {
4         t = v[i];
5         for (j = i; j > l && t < v[j-1]; j--)
6             v[j] = v[j-1];
7         v[j] = t;
8     }
9 }
```

Podemos evitar a comparação  $j > l$  colocando o menor valor na primeira posição (**sentinela**)

## Ordenação por Inserção - Terceira Otimização

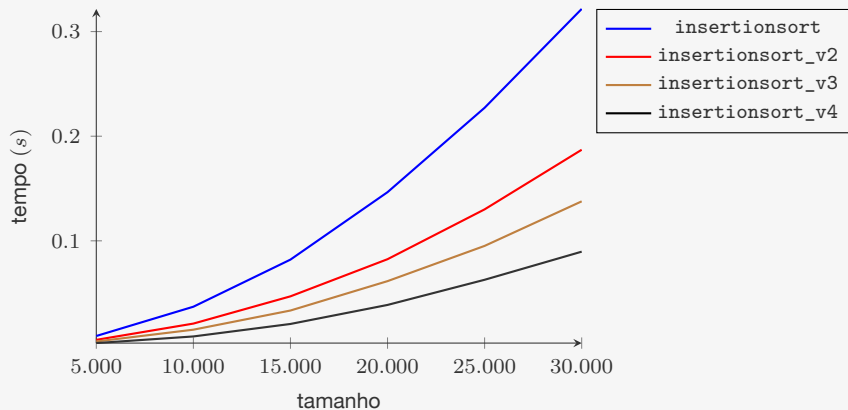
```
1 void insertionsort_v3(int *v, int l, int r) {
2     int i, j, t;
3     for (i = l+1; i <= r; i++) {
4         t = v[i];
5         for (j = i; j > l && t < v[j-1]; j--)
6             v[j] = v[j-1];
7         v[j] = t;
8     }
9 }
```

Podemos evitar a comparação  $j > l$  colocando o menor valor na primeira posição (**sentinela**)

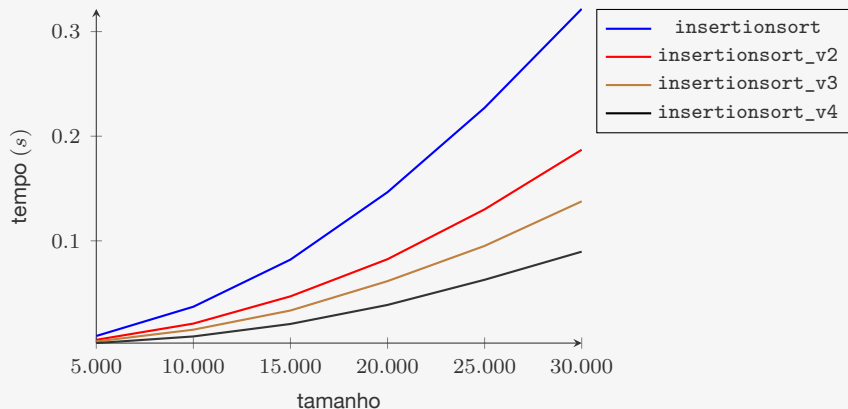
```
1 void insertionsort_v4(int *v, int l, int r) {
2     int i, j, t;
3     for (i = r; i > l; i--)
4         if (v[i] < v[i-1])
5             troca(&v[i], &v[i-1]);
6     for (i = l+2; i <= r; i++) {
7         t = v[i];
8         for (j = i; t < v[j-1]; j--)
9             v[j] = v[j-1];
10        v[j] = t;
11    }
12 }
```

} primeira passada do **bubblesort**

# Gráfico de comparação do InsertionSort

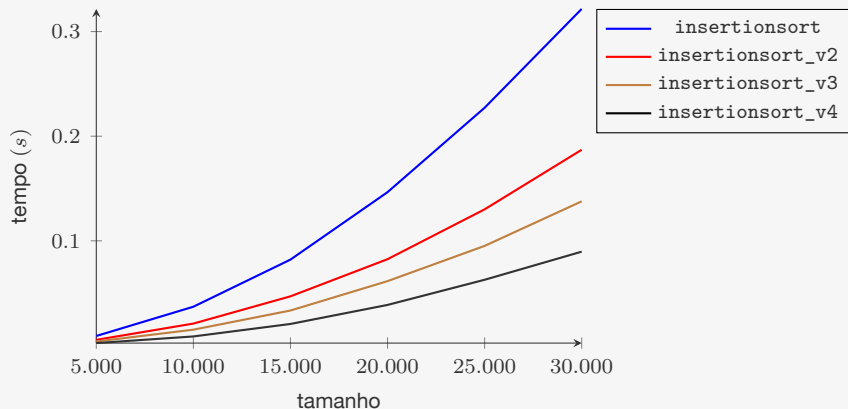


## Gráfico de comparação do InsertionSort



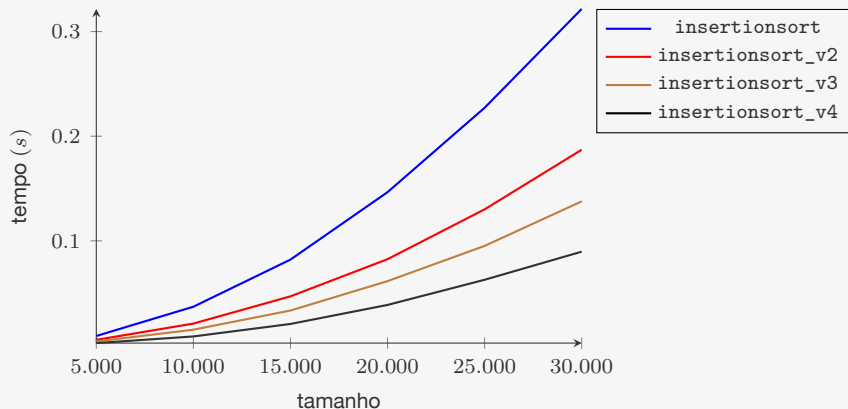
- A complexidade teórica do algoritmo não melhorou

## Gráfico de comparação do InsertionSort



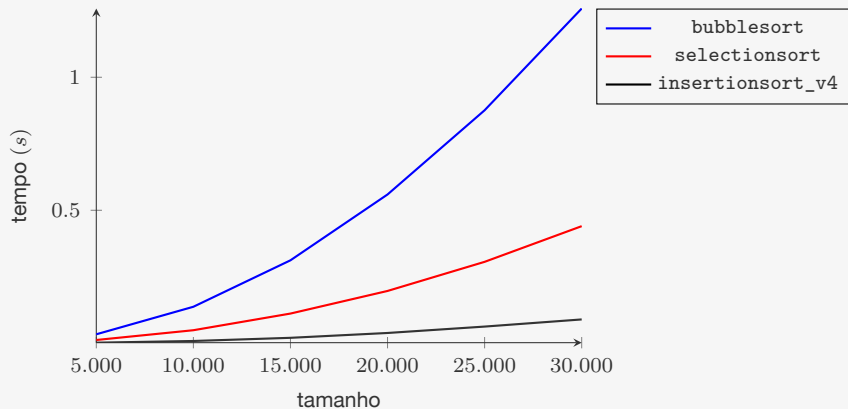
- A complexidade teórica do algoritmo não melhorou
  - continua  $O(n^2)$

# Gráfico de comparação do InsertionSort



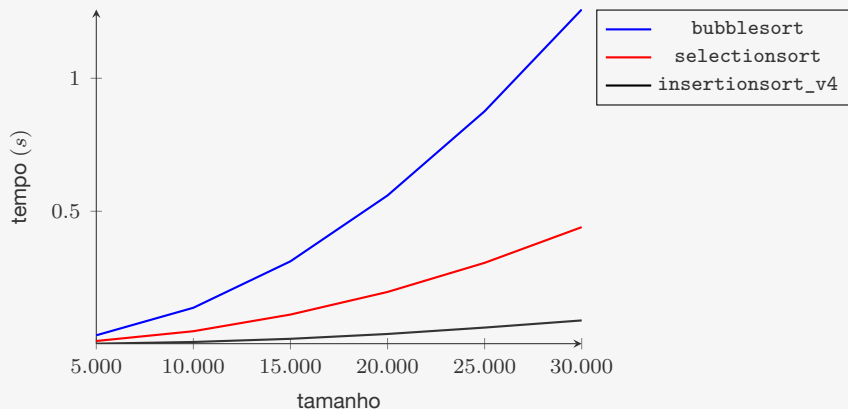
- A complexidade teórica do algoritmo não melhorou
  - continua  $O(n^2)$
- Mas as otimizações levaram a um ganho na performance

# Gráfico de comparação do três algoritmos



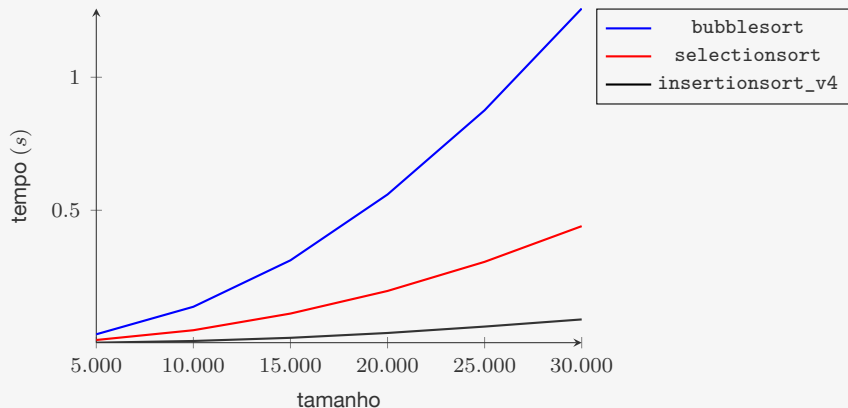


# Gráfico de comparação do três algoritmos



- **bubblesort** demora  $1,258s$  para  $n = 30.000$

# Gráfico de comparação do três algoritmos



- **bubblesort** demora  $1,258s$  para  $n = 30.000$
- **insertionsort\_v4** demora  $1,248s$  para  $n = 110.000$

# Conclusão

Vimos três algoritmos:

# Conclusão

Vimos três algoritmos:

- `bubblesort`: na prática é o pior dos três, raramente usado

# Conclusão

Vimos três algoritmos:

- **bubblesort**: na prática é o pior dos três, raramente usado
- **selectionsort**: bom quando comparações são muito mais baratas que trocas

# Conclusão

Vimos três algoritmos:

- `bubblesort`: na prática é o pior dos três, raramente usado
- `selectionsort`: bom quando comparações são muito mais baratas que trocas
- `insertionsort`: o melhor dos três na prática

# Conclusão

Vimos três algoritmos:

- **bubblesort**: na prática é o pior dos três, raramente usado
- **selectionsort**: bom quando comparações são muito mais baratas que trocas
- **insertionsort**: o melhor dos três na prática
  - Vimos várias otimizações do código que melhoraram os resultados empíricos

# Conclusão

Vimos três algoritmos:

- **bubblesort**: na prática é o pior dos três, raramente usado
- **selectionsort**: bom quando comparações são muito mais baratas que trocas
- **insertionsort**: o melhor dos três na prática
  - Vimos várias otimizações do código que melhoraram os resultados empíricos
  - Melhorar o código interno dos laços pode fazer muita diferença



# Conclusão

Vimos três algoritmos:

- **bubblesort**: na prática é o pior dos três, raramente usado
- **selectionsort**: bom quando comparações são muito mais baratas que trocas
- **insertionsort**: o melhor dos três na prática
  - Vimos várias otimizações do código que melhoraram os resultados empíricos
  - Melhorar o código interno dos laços pode fazer muita diferença
  - Durante o curso, não focaremos em otimizações como essas...

# Conclusão

Vimos três algoritmos:

- **bubblesort**: na prática é o pior dos três, raramente usado
- **selectionsort**: bom quando comparações são muito mais baratas que trocas
- **insertionsort**: o melhor dos três na prática
  - Vimos várias otimizações do código que melhoraram os resultados empíricos
  - Melhorar o código interno dos laços pode fazer muita diferença
  - Durante o curso, não focaremos em otimizações como essas...

Mas veremos algoritmos que são melhores **assintoticamente**

# Conclusão

Vimos três algoritmos:

- **bubblesort**: na prática é o pior dos três, raramente usado
- **selectionsort**: bom quando comparações são muito mais baratas que trocas
- **insertionsort**: o melhor dos três na prática
  - Vimos várias otimizações do código que melhoraram os resultados empíricos
  - Melhorar o código interno dos laços pode fazer muita diferença
  - Durante o curso, não focaremos em otimizações como essas...

Mas veremos algoritmos que são melhores **assintoticamente**

- Na próxima unidade, veremos um algoritmo  $O(n \lg n)$

# Conclusão

Vimos três algoritmos:

- **bubblesort**: na prática é o pior dos três, raramente usado
- **selectionsort**: bom quando comparações são muito mais baratas que trocas
- **insertionsort**: o melhor dos três na prática
  - Vimos várias otimizações do código que melhoraram os resultados empíricos
  - Melhorar o código interno dos laços pode fazer muita diferença
  - Durante o curso, não focaremos em otimizações como essas...

Mas veremos algoritmos que são melhores **assintoticamente**

- Na próxima unidade, veremos um algoritmo  $O(n \lg n)$
- Melhor que qualquer algoritmo  $O(n^2)$

# Conclusão

Vimos três algoritmos:

- **bubblesort**: na prática é o pior dos três, raramente usado
- **selectionsort**: bom quando comparações são muito mais baratas que trocas
- **insertionsort**: o melhor dos três na prática
  - Vimos várias otimizações do código que melhoraram os resultados empíricos
  - Melhorar o código interno dos laços pode fazer muita diferença
  - Durante o curso, não focaremos em otimizações como essas...

Mas veremos algoritmos que são melhores **assintoticamente**

- Na próxima unidade, veremos um algoritmo  $O(n \lg n)$
- Melhor que qualquer algoritmo  $O(n^2)$ 
  - Mesmo na versão mais otimizada

# Exercício

```
1 void bubblesort_v2(int *v, int l, int r) {
2     int i, j, trocou = 1;
3     for (i = l; i < r && trocou; i++){
4         trocou = 0;
5         for (j = r; j > i; j--){
6             if (v[j] < v[j-1]) {
7                 troca(&v[j-1], &v[j]);
8                 trocou = 1;
9             }
10    }
11 }
```

Quando ocorre o pior caso do `bubblesort_v2`?

Quando ocorre o melhor caso do `bubblesort_v2`?

- Quantas comparações são feitas no melhor caso?
- Quantas trocas são feitas no melhor caso?

# Exercício

```
1 void insertionsort_v3(int *v, int l, int r) {
2     int i, j, t;
3     for (i = l+1; i <= r; i++) {
4         t = v[i];
5         for (j = i; j > l && t < v[j-1]; j--)
6             v[j] = v[j-1];
7         v[j] = t;
8     }
9 }
```

Quando ocorre o pior caso do `insertionsort_v3`?

Quando ocorre o melhor caso do `insertionsort_v3`?

- Quantas comparações são feitas no melhor caso?
- Quantas atribuições são feitas no melhor caso?